realMethods Framework

Development Guide

realMethods Framework Development Guide

sales@realmethods.com
http://www.realmethods.com

realMethods model driven; pattern based

This document completely describes the development related aspects of the realMethods Framework and an application it generates. It covers:

- What's New
- Installation
- Using the Code Generator
- Framework Configuration
- Using Jakarta Ant to build your application
- Testing generated CRUD functionality
- JUnit Testing the Framework
- The installed Proof of Concept example app
- FAQ

To get a complete understanding of the design and strategies behind each Framework component and service, please review the Architecture Guide.

Document Naming Conventions

The following words and acronyms are used throughout this document:

Framework – realMethods Framework

RM_HOME – Framework installation root directory location

AIB – Application Infrastructure Builder (code generator)

Patterns – refers to the core J2EE Design Patterns

MDA – refers to the OMG specification for Model Driven Architectures

XMI – refers to XML Meta-Data Interchange

UML – refers to Unified Modeling Language

AJAX – refers to Asynchronous Javascript and XML

What's New

AJAX

Struts 2.0

Mule ESB

EJB 3.0

What's is No Longer Supported

Presentation Tier Cache	No longer uses customized cache; generated Struts Action classes cache to the HttpServletRequest
Hooks and Events	No longer supported
Entity Beans	No longer an option to be generated
Container Managed Persistence (CMP)	Since Entity Beans are no longer generated,
Transfer Objects (TO)	No longer supported during code generation

Installation

This section provides you with information on what was placed where during the un-jarring of the installation related jar files. Each sub-section below represents a directory under the *RM_HOME* directory.

aib

This directory contains all the files necessary to use the Framework Application Infrastructure Builder. Under this directory is a sub-directory named *CodeTemplate*. This directory contains all the template files used to generate the code from the AIB. Take some time to open up these files to get a better understanding of the code generated, but use caution if you decide to modify these files.

buildfiles

- This directory contains all the common files that will be included in the build of a generated application

docs

This directory contains documentation included with the Framework.

- Getting Started web page
- AIB Start web page
- Release notes web page

lib

This directory contains the necessary archive files to generate, modify, and deploy Framework-based applications, as well as deploy the Proof of Concept. Consult your application server vendor's documentation concerning the deployment of Servlets and EJBs.

poc\model

This directory contains the following files:

catalog.xmi	UML 1.3 compliant XMI file representation of a Catalog based business model
company.xmi	UML 1.3 compliant XMI file representation of Company based business model

Code Generation

The entire code generation section is now a part of the AIB Help system, available as both as a separate Eclipse Plugin (3.x and 2.1) as well as integrated in the AIB Standalone version.

Configuring the Framework

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Contents of the framework.xml file

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Contents of the connectionpool.xml file

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Database Configuration

This has been move to the AIB integrated Help facility, under the Application Configuration section.

JMS

This has been move to the AIB integrated Help facility, under the Application Configuration section.

IBM MQ

This has been move to the AIB integrated Help facility, under the Application Configuration section.

LDAP

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Contents of the security.xml file

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Framework ACL Security Manager properties

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Contents of the loghandlers.xml file

This has been move to the AIB integrated Help facility, under the Application Configuration section.

Contents of the task.xml file

Use this file to relate a fully qualified task class name with a unique name. The Framework allows you to execute a Task by simply providing its name. The Framework does this by using JMS. Read more about how to create and execute a task in the JMS Command/Task Architecture section. You can also associate a name with a comma delimited list of Command class names.

```
<tasks>

<!-- the following definition is used internally by the Framework to handle
application logging -->

   <task name="FrameworkLogTask"
        commands=
        "com.framework.integration.command.FrameworkLogCommand"/>

   <!-- an example with no commands, just a referrenced task -->
   <task name="CreateEmployee" task="com.poc.task.CreateEmployeeTask"
/>

</tasks>
```

The mandatory Framework log service task

For the purpose of using the Framework's Logging Service, the following named Task must be contained within the task.xml file:

```
FrameworkLogTask=\
   Commands=com.framework.command.FrameworkLogCommand
```

In order to override the default implementation of the Logging Services, simply provide the name of a different Command class, a different Task class, or optionally append another Command class to the FrameworkLogCommand.

How to specify which Connection Pool to use

Note: Configuring a connection pool is NOT necessary if you are using the default database connection provided by the generated Hibernate configuration files.

A database-based DAO acquires a database connection by using Framework connection obtained from a Connection Pool. There are 3 ways by which a DAO instance is assigned which named database Connection Pool to use.

Method 1: Default Assignment

By default, unless assigned externally (normally by a generated Business Delegate, Session Bean, or Entity Bean), base class FrameworkDatabaseDAO will use the Connection Pool name associated with the framework.xml property DefaultDatabaseConnectionName.

Method 2: Client Programmatic Assignment

The Connection Pool name can be assigned by making the following call before actually using the DAO:

```
dao.setConnectionPoolName( "MyNewConnectionPoolName" );
```

Application Deployment

Follow the instructions your application server provides to deploy your application EAR located in the */dist* directory after the ANT build. You can also refer to the *POC → App. Server Specific Deployment Instructions* outlined below.

Using Key Framework Services

Logging

Logging a message to the default log handler

Use the following steps to enable and utilize the Logging Services.

Step #1 - Configuring loghandlers.xml
This property file allows you to specify any number of Log Handlers.

```
<logHandler name="FrameworkDefaultLog"
          synchronous="true"
          debug="poc"
          info="poc"
         warn="poc"
         error="poc"
         dateTimeStampFormat="yyyy.mm.dd-hh:mm:ss"
       />
```

Step #2 - Verifying the Log Task
Logging Services makes use of the JMS/Command/Task architecture to handle the actual logging. Importantly, the following entry must exist within the task.xml file:

```
<task name="FrameworkLogTask"
commands="com.framework.integration.command.FrameworkLogCommand"/>
```

Step #3 - Configuring the framework.xml
Logging Services makes use of the following properties, found within framework.xml configuration file.

```
DEFAULT_LOG_HANDLER_NAME=FrameworkDefaultLog
```

This parameter correlates to a named Log Handler found in the task.xml configuration file.

Step4 - Configuring the log4j.xml (Optional)
Looking above at the Log Handler titled *FrameworkDefaultLog*, it has defined an output destination labeled *poc*. In this example, suppose this output destination is correlated to a Log4J Logger. If so, you will have to add a logger entry to the log4j.xml file. Also, if you want the output for this Log4j Logger to go to a different location than the internal Framework output (file:rm_framework_log4j.log), you will also have to define an additional Appender in this file and make use of it within your newly defined Log4j Logger.

Step #5 - Programmatic Usage
Class

```
com.framework.integration.log.FrameworkDefaultLogger
```

contains 4 static methods that simplify sending a log message to the Default Log Handler.

```
static public void debug( String msg )
static public void info( String msg )
static public void warn( String msg )
static public void error( String msg )
```

Logging a message to a user defined log handler

If you need to access another FrameworkLogHandler other the default one, take the following steps:

Step 1 – Adding a Log Handler

To define one or more additional Log Handlers, place them in the loghandlers.xml configuration file.

```
<logHandler name="FrameworkDefaultLog"
          synchronous="true"
          debug="poc"
          info="poc"
         warn="poc"
         error="poc"
         dateTimeStampFormat="yyyy.mm.dd-hh:mm:ss"
       />

<logHandler name=”AnotherLogHandler"
          synchronous="true"
          info=" SYSTEM_OUT|FOO"
         warn=" SYSTEM_OUT|FOO"
         error=" SYSTEM_OUT|FOO"
         dateTimeStampFormat="yyyy.mm.dd-hh:mm:ss"
       />
```

Step 2 - Configuring the log4j.xml (Optional)

See **Step 4** from the section titled *Logging a message to the default log handler*

Step #3 - Programmatic Usage

```
IFrameworkLogHandler logHandler = null;
logHandler = FrameworkLogHandlerFactory.getObject("AnotherLogHandler);
logHandler.debug( "The log message." );
```

Connection Pooling

Note: If you only need to use the connection pooling mechanism provided by the generated Hibernate configuration files, the following section is not necessary.

Access a Connection from the Connection Pool Manager

The following example applies to getting any type of Connection from the Connection Pool Manager, although it is specific to getting a database type:

```
ConnectionPoolMananger mgr = ConnectionPoolManagerFactory.getObject();
IDatabaseQuerier dbQuerier = null;

dbQuerier = (IDatabaseQuerier)mgr.getConnection( "pool_name" );
```

Release When Done

Once you are finished with any connection, make sure to release it back to the Connection Pool Manager:

```
dbQuerier.releaseToConnectionPoolManager()
```

Adding a custom connection to be pooled

The Framework's Connection Pooling provides four (4) connection types:

- Database
- JMS
- LDAP
- IBM/MQ

Anytime you need a different resource that has transactional characteristics and would benefit from pooling, consider defining a new connection type. For instance, suppose your application requires a good amount of file input/output. Often, writing to a file successfully is conditional, and may require "undoing". These requirements are just right for defining a file connection implementation.

Framework connection pooling requires the implementation of interface IConnectionImpl. The simplest way to implement this interface is to derive your connection from ConnectionImpl and implement the following abstract methods:

```
abstract public void startTransaction()
abstract public void commit()
abstract public void rollback()
abstract public void disconnect()
```

Executing SQL Directly

You can execute a query directly to a database connection managed by the ConnectionPoolManager without using a DAO. Observe the following steps:

Step 1

Get a db connection straight from a ConnectionPoolManager, by name.

```
IDatabaseQuerier dbQuerier =
(DatabaseQuerier)ConnectionPoolManagerFactory.getObject().getConnection(
"MyOracleDBConnection" );
```

Step 2

Execute a select statement or stored procedure calling one of the following methods on DatabaseQuerier:

```
public Collection executeSelectStatement( String selStmt, IResultSetCallback
cb )
public Collection executeSelectStatement( String selStmt, Vector parms,
        IResultSetCallback cb )
public Collection executeStoredProcedure( String procName, Vector parms,
        IResultSetCallback cb )
public Collection executeStoredProcedure( String procName, Vector parms )
```

Step 3

Implement interface

```
com.framework.integration.objectpool.db.IResultSetCallback
```

The only method to implement for this interface is:

```
public Collection notifyResultSet( ResultSet rs, String sqlStmt )
```

Your implementation of this interface can iterate over the provided ResultSet and package the row/column data into a Collection as you see fit. Importantly, the Collection returned method is the same Collection returned by the first three methods above. The last method returns a Collection that represents the OUT type parameters as discovered in the *parrms* Vector argument.

The Framework comes with a helper class called:

```
com.framework.integration.objectpool.db.ResultsetToStringUnloader
```

This class implements the IResultSetCallback interface and returns a Collection that consists of two ArrayLists. The first ArrayList contains the columns names of the ResultSet. Each entry of the second ArrayList represents a row of the ResultSet, as a HashMap. The key into each Hashmap is a column name and the key's corresponding value is a String representation of the data.

Step 4

Make sure you release the connection back to the ConnectionPoolManager when you are done with it.

```
dbQuerier.releaseToConnectionPoolManager()
```

Security

Accessing a Security Manager

The default Security Manager

If the application needs to access a Security Manager, and doesn't need a specific instance of one, you can use the default Security Manager.

```
FrameworkSecurityManagerFactory factory = null;
IFrameworkSecurityManager mgr = null;

factory = FrameworkSecurityManagerFactory.getInstance();
securityManager = factory.getDefaultSecurityManager();
```

A specific Security Manager

If the application needs to access a specific Security Manager, you must provide its name to the Security Manager factory.

```
FrameworkSecurityManagerFactory factory = null;
IFrameworkSecurityManager mgr = null;

factory = FrameworkSecurityManagerFactory.getInstance();
securityManager = factory.getSecurityManager( "my_ldap_sec_mgr" );
```

Using the Security Helper Classes

com.framework.integration.security.FrameworkServletSecurityHelper

Binds the default Security Manager with a provided HttpSession, allowing for user specific authentication and authorization. This class can be used to easily provide a security implementation to a Jakarta Struts presentation tier, or any application using a Servlet. The realMethods Presentation tier (represented by the JSP/Servlet Framework) already incorporates security management.

com..framework.security.FrameworkLDAPSecurityAdapter

Provides an LDAP based security implementation of interfaces IAuthenticateUser, ISecurityRoleLoader, and ISecurityUserLoader. This class can be specified for properties authenticator, roleLoader, and userLoader when defining a Framework ACL SecurityManager.

Add a new Security Manager Implementation

Although the Framework defines both different flavors for both ACL and JAAS based security, you may need to interface to another security system. Follow these steps to define a new Framework Security Manager implementation.

Step 1
Define a single Java class that extends class

```
com.framework.integrationsecurity.FrameworkSecurityManager
```

Step 2
FrameworkSecurityManager is defined as an abstract class, so you will be required to implement the following methods:

```
public boolean isUserAuthorized( ISecurityUser user, ISecurityRole role
public boolean unAuthenticateUser( ISecurityUser user )
protected IAuthenticateUser createAuthenticator()
protected ISecurityRoleLoader createSecurityRoleLoader()
protected ISecurityUserLoader createSecurityUserLoader()
protected void applyUserAndRoles( ISecurityUser user, Collection roles )
```

Step 3
Methods *createAuthenticator*, *createSeurityRoleLoader*, and *createSecurityUserLoader* all require you to provide an implementation of 3 Framework specific interfaces of the com.ramework.integration.security:

IAuthenticateUser	An implementation will handle the responsibility of user authentication
ISecurityRoleLoader	An implementation will handle loading all of the roles (identities) associated with a user
ISecurityUserLoader	An implementation will handle loading relevant user security data

Your implementation of *applyUserAndRoles(...)* will be called once user authentication has taken place, and all related roles have been loaded. You may choose to cache this data.

Step 4
Define your Security Manager implementation within security.xml, providing any required parameters. These parameters will be made available to you by overloading method

```
public void initialize( String name, Map props )
```

Note
Make sure you delegate to the base class version of this method if you override it.

Authentication/Authorization Failure in the rM Presentation Tier

Authorization Failure

In the event a user is not authorized of a particular action, the Framework will attempt to display the named view referenced by the framework.xml property:

```
UserAuthorizationFailureScreenName=UNAUTHORIZED_USER
```

The value for this property must be located in the viewdefs.xml file.

Authentication Failure

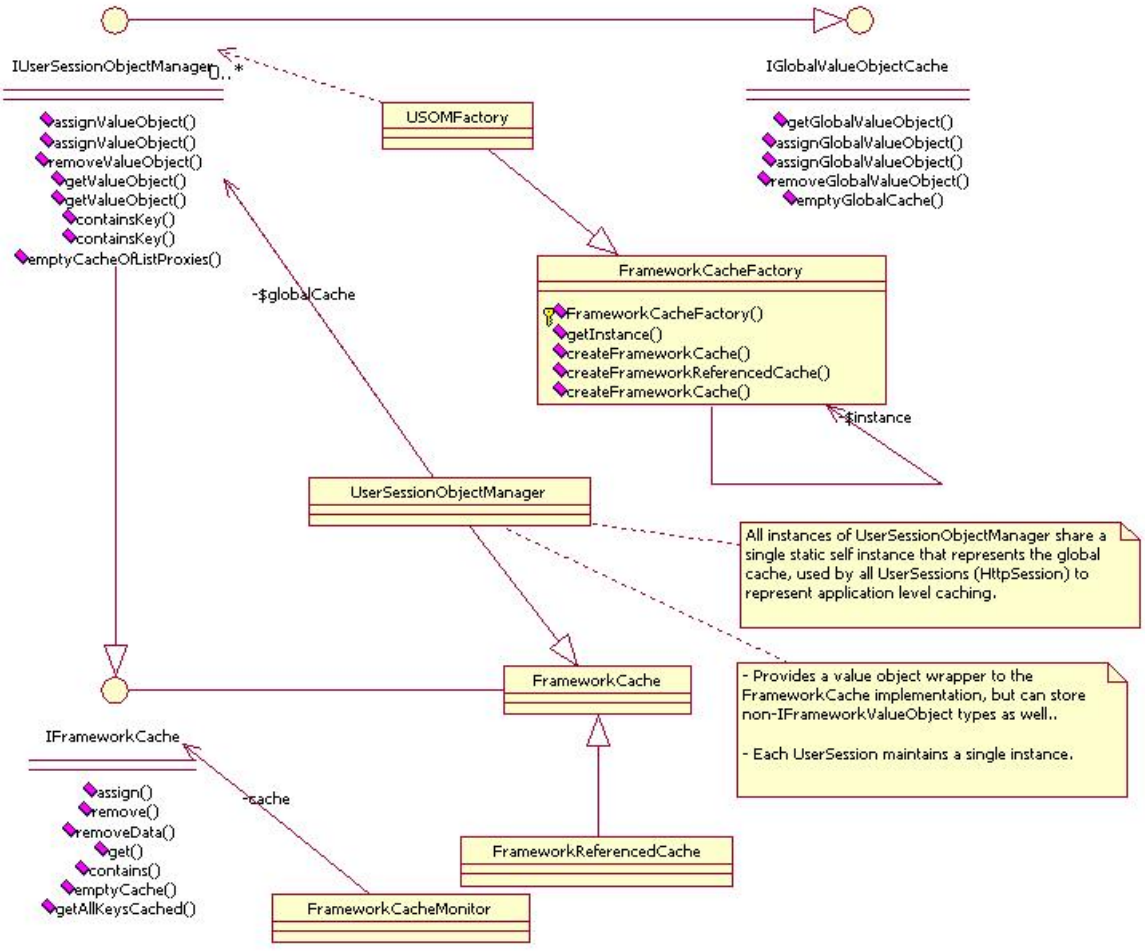
In the event user authentication fails, the Framework will attempt to display the named view reference by the framework.xml property:

```
UserAuthenticationFailureScreenName=UNAUTHENTICATED_USER
```

The value for this property must be located in the viewdefs.xml file.

Framework Caching

The Framework provides its own caching mechanism to apply application logic to a java.util.Map implementation. The AIB generates a sub-class to the Framework cache, providing type-safe specific access method to the underlying cache itself. Although this caching implementation is no longer used on the Presentation tier within the generated Struts Action classes, it is still an option to use anywhere within your application. Optionally via configuration, it can be used within the DAO layer in order to speed data retrieval, but should be used with caution. There is more on this later in this section.



Using the Global Cache

Class UserSessionObjectManager contains a single static instance of an IUserSessionObjectManager. This can be used as the global cache for the application. UserSessionObjectManager implements interface:

```
com.framework.integraton.cache.IGlobalValueObjectCache
```

You can also gain direct access to the global cache by calling the following method on com.framework.integration.cache.UserSessionObjectManager.

```
static public IUserSessionObjectManager getGlobalCache()
```

Auto-purging with the Framework Reference Cache

There may exist a cache scenario where the application requires the periodic purging of its least used cached objects. If this is the case, you should use the class

```
com.framework.integration.cache.FrameworkReferencedCache
```

This class is sub-class to *FrameworkCache*, and objects assigned to the cache are wrapped in a *SoftReference*, to ensure any object not regularly used is purged during garbage collection. Always check for a null return value when retrieving data from this cache

Managing the size of the User Session Cache

If you would like to manage the size of the cache managed by each UserSession, use the following parameters within the framework.xml configuration file:

```
USOM_MAX
        The # of objects to cache before emptying; -1 means no monitoring

USOM_MAX_CHECK_PERIOD_IN_MILLIS
        How often to check cache size
```

Note
In order for this feature to work, you must have the following Framework properties in use:

JMXSelfRegistration=TRUE

Enabling DAO Caching

The Framework provides an option whereby the DAO layer caches objects as they are persisted and read from the data store. To enable this option, assign the following within the framework.xml configuration file:

```
DAO_CACHE_INTERNALLY="true"
```

This option is only useful in situations where the DAO layer is not in a clustered environment, and is the only means of persisting data. If enabled in such conditions, the likely hood of serving up a stale Value Object is extremely high.

Learn more about DAO Caching in the Architecture Guide.

Framework Startup Scenarios

In order for the Framework to support a generated application at runtime, the Framework must be started. The Framework implementation provides for the automatic startup under the following conditions:

- **Initialization of a FrameworkFrontControllerServlet**
- **Initialization of a FrameworkStrutsActionServlet**
- **First access to any Framework generated EJB**
- **First access to any mandatory Framework Startup Component**

Manually starting the Framework for your application

If you have defined your application in a way that one of the automatic startup scenarios doesn't occur, you will have to manual start the framework. This can be accomplished by using the StartupManager:

Default Startup Classes

The framework.xml configuration file defines a startup element that contains a list of name/value pairings that relate to components that are created and initialized during the startup phase. The following startup components are the default startup components, which correlate to key services provided by the framework:

```
<startups
ConnectionPoolManager="com.framework.integration.objectpool.ConnectionPoolManagerFactory"
TaskManager="com.framework.integration.task.TaskJMSExecutionRegistryFactory"
SecurityMannager="com.framework.integration.security.FrameworkSecurityManagerFactory"
LogManager="com.framework.integration.log.FrameworkLogHandlerFactory"
HookManager="com.framework.integration.hook.FrameworkHookManager"
 />
```

Defining a Startup Class

If you need to define a process or service that is made available to the rest of the application after startup, take the following steps:

Step 1
Extend class

```
com.framework.common.startup.FrameworkStartup
```

This class contains a static block that attempts to call StartupManager.start(…)

Step 2
Add your class to the *startups* section of framework.xml, applying a unique name to it.

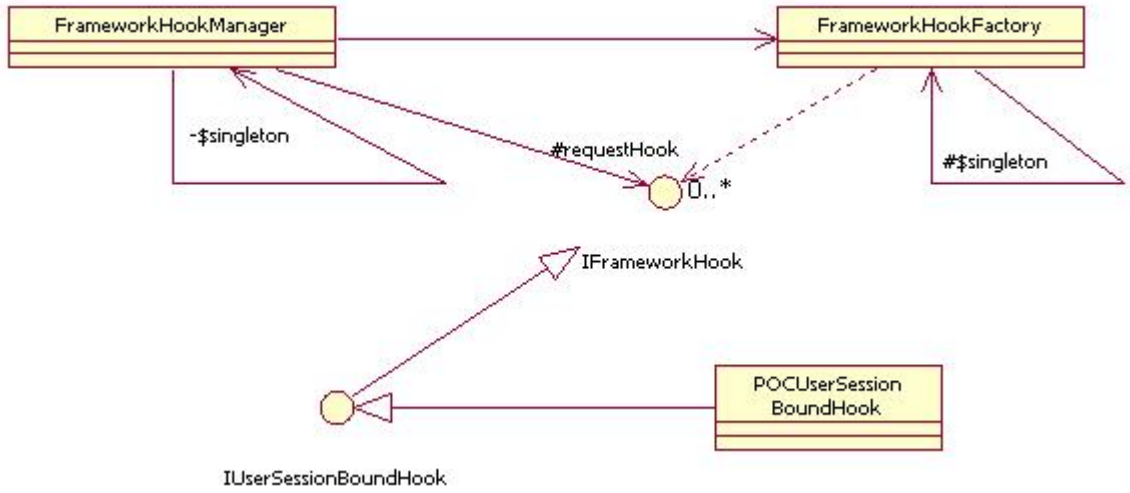
Using the Hook Infrastructure

In an attempt to provide an open structure to build your application upon, the Framework has been designed to give you the ability to interject functionality to some of the major components of the system. This facility allows you to introduce your owns means of handling a particular process.

For example, you could provide an implementation of the

```
com.framework.integration.hook.IUserSessionBoundHook
```

to execute logic each time a new UserSession is created and bound to a newly discovered HttpSession.



You make a Hook available to the Framework hook manager by modifying the framework.xml configuration file. This file contains a section for defining hooks. Add the name of the hook class you've implemented and wish to associate with a particular hook. By doing this, you are telling the Framework hook manager to associate your hook class with the name used by the Framework when requesting a hook of the Framework hook manager.

Note:
It's important to understand that your hook is part of an execution process. This means that your hook implementation should do its work in a timely manner and eventually return control back to the caller to continue on with the normal course of execution. Equally important is the fact that your hook implementation will only be instantiated once for the life of the application. This means it needs to be thread-safe since concurrent calls on it are highly likely.

Hook related properties and interfaces

Although available in the framework code base, this feature is no longer supported as of version 5.0

If interested in providing a hook implementation to the framework, use the following table to determine the interface to implement.

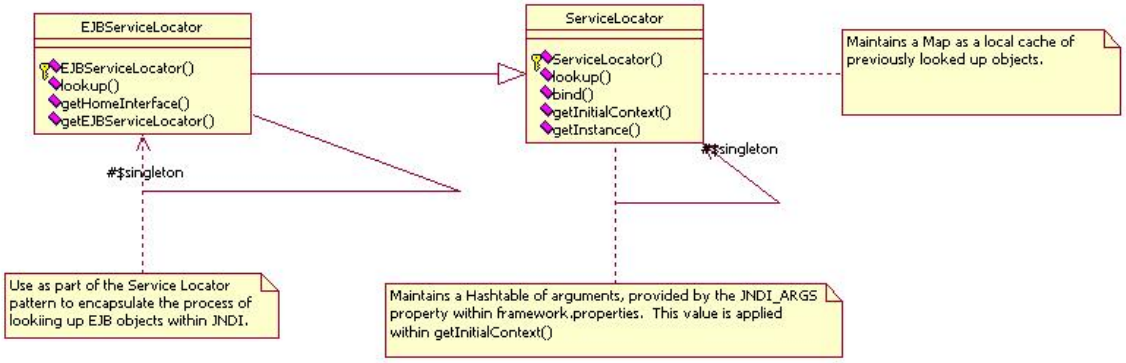
Property	Framework Interface to Implement
PreHttpServletRequestProcessor	IPreHttpServletRequestProcessorHook
PostHttpServletRequestProcessor	IPostHttpServletRequestProcessorHook
HttpServletRequestProcessorError	IHttpServletRequestProcessorErrorHook
PreTaskExecute	IPreTaskExecuteHook
PostTaskExecute	IPostTaskExecuteHook
TaskExecutionFailure	ITaskExecutionFailureHook
PreCommandExecute	IPreCommandExecuteHook
PostCommandExecute	IPostCommandExecuteHook
CommandExecutionFailure	ICommandExecutionFailureHook
PrePageRequestProcessor	IPrePageRequestProcessorHook
PostPageRequestProcessor	IPostPageRequestProcessorHook
PageRequestProcessorError	IPageRequestProcessorErrorHook
UserSessionBound	IUserSessionBoundHook
UserSessionUnBound	IUserSessionUnBoundHook
CheckedExceptionThrown	ICheckedExceptionThrown

Apply the fully qualified path of your Hook class name to the hooks property in framework.xml configuration file.

Service Locator

The Service Locator provides a simple uniform means of using an InitialContext. It uses the value assigned to the JNDI_ARGS property of the framework.xml configuration file to create the InitialContext.

Note
This property must be set correctly based on the parameters required of the *InitialContext* provider. Otherwise, all clients depending on the objects returned by the Service Locator will more than likely fail



Using the EJB service locator

The generated EJB Service Locators make use of:

```
com.framework.integration.locator.EJBServiceLocator
```

This class provides helper method

```
getHomeInterface( String jdniServiceHome, Class homeClass )
```

that uses the sub-class ServiceLocator to look up the session bean home interface using the provided name, and the narrows the home interface before returning it to the caller.

Rather than use this class directly, you may be better served using the appropriate generated service locator.

Data Access within JNDI

Binding an Object

Rather than bind data directly to JNDI, use the following code:

```
ServiceLocator.getInstance().bind( "myObject", theObject );
```

Retrieving an Object

To leverage the caching capabilities of the Service Locator, use the following code to retrieve an object from JNDI:

```
ServiceLocator.getInstance().lookup( "myObject" );
```

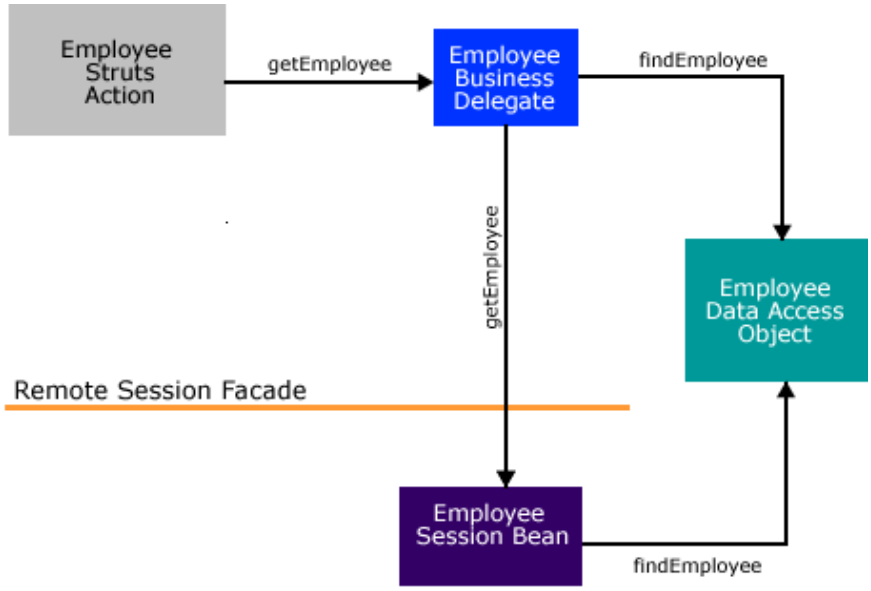
Business Tier Data Access Scenarios

When developing your application, there will be times when you will require different amounts of business data for a given client request. For example, perhaps one presentation tier client needs to display only the immediate data of an Employee within a form, while another client needs to display the immediate and associative data of an Employee. In the former case, less data is required of the business tier, while in the later more data is required. If the Employee business delegate only returned data to satisfy the latter case, it would implicitly satisfy the former as well, but at possible cost in terms of network traffic when a remote session façade is involved.

The business delegates have been designed and generated to accommodate both cases.

Scenario 1 – Accessing a Single Business Object

In this scenario, the client only requires the data of an instance of ***EmployeeBO***. This is accomplished using an ***EmployeePrimaryKey***. The data returned within ***EmployeeBO*** will include all immediate attributes, as well as other business objects associated with it. From the Proof of Concept model, *EmployeeBO* will contain an *AddressBO* as well as a List of *InvestmentBO* instances.



This example shows both Session Façade access or optional direct to DAO (no EJBS) options.

Anatomy of a Business Object

A Business Object is composed of two types of data:

Modeled attributes

These are the same attributes imported for the related class in the domain model:

```
EmployeeValueObject, from the  POC
protected Long employeeID;
protected String firstName;
protected String lastName;
protected String socialSecurity;
protected String password;
protected Double annualSalary;
protected java.sql.Date birthDate;
protected java.sql.Date hireDate;
protected java.sql.Date eligibilityDate;
protected AddressBO address = null;
protected List investments = Collections.EMPTY_LIST;
```

Deduced Associations

A typed primary key class of the associated business object represents this data type.

```
protected Long employeeID;
protected String firstName;
protected String lastName;
protected String socialSecurity;
protected String password;
protected Double annualSalary;
protected java.sql.Date birthDate;
protected java.sql.Date hireDate;
protected java.sql.Date eligibilityDate;
protected AddressBO address = null;
protected List investments = Collections.EMPTY_LIST;
```

Note
AddressBO represents a 1-1 association between Employee and Address. The List named investments represents a 1-m association between Employee and Investment, where an Employee has many Investments.

Testing your application with JUnit

The AIB is capable of generating a set of JUnit Test cases, one for each persistent value object. The generated classes allow you to unit test all CRUD functionality by making all calls initiated from the generated Business Delegates.

In order for the test cases to execute successfully be aware of the following:

- Database (and optionally LDAP) connectivity is in place.
- Based on your application server, make sure the JNDI_ARGS set correctly within the framework.xml configuration file used by the JUnit Test Case. This is necessary in cases where the business delegates need to communicate with the Session Facade.

  Note
  If you generated the business delegates to talk directly to the generated DAO classes, this step is not necessary.

- If EJBs are in use, be sure the application server is running and EJBs are deployed.

Imported Methods of the Generated JUnit Test Case

```
protected void setUp()
```

This method explicitly calls the StartupManager to start the Framework on behalf of the test case.

```
public void CRUD()
```

This helper method calls the following generated test methods:

testCreate	Creates a new instance of the associated Value Object
testRead	Reads the previously created Value Object
testUpdate	Modifies and stores the read Value Object
testGetAll	Loads all Value Objects
testDelete	Deletes the created Value Object

Proof of Concept

The Framework install comes with two XMI files located in the ***\poc\model*** directory. Both of these XMI files can be loaded into the AIB in order to generate an application to deploy on your application server. If you don't create your own XMI file at first, use one of these as reference so you can quickly create an application.

Note: The instructions below assume you are creating an application by the name of **poc**. If you choose a different name, be sure to replace that name in these instructions as well.

License file location

You must provide the Framework with the directory location of the framework.license file. This accomplished using the following Java runtime system properties

-DFRAMEWORK_HOME=*directory-location-of-license-file*

App. Server Specific Deployment Instructions

Weblogic 8.1

{RM_HOME} is the root directory for realMethods installation.
{WL_HOME} is the root directory for the Weblogic installation.
{APP_ROOT} is the root directory where you generated the POC code base.

Step 1

Modify {WL_HOME}/server/bin/startWLS.bat (or startWLS.sh), and add the following Java runtime parameters to the start up line:

```
-DFRAMEWORK_HOME={location of the framework.license file}
```

Step 2

Execute the Weblogic Application Server

Step 3

Run the Weblogic Admin Console from your web browser:

```
http://localhost:7001/console
```

Step 4

Upon admin login, click on *'Applications'* and then click on *'Deploy a new Application'*

Step 5

Click on the *'upload your file(s)*' link and browse to locate the *poc.ear* file in the *dist* subdirectory under the *{APP_ROOT}* directory. Click *upload* when located, and click on the *'Deploy'* button once loading has completed.

Step 6

Select the *poc.ear* radio button then click *'Continue* and then the *'Deploy'* on the next screen.

Step 7

The application will now automatically deploy. Deployment is complete and successful if EJB and Web Application Modules display a *'Status of Last Action'* of *Success*.

Step 8 - Execution

Once the console indicates that the POC application has started, you should be able to access it using the following URL from your:

```
http://localhost:7001/poc/
```

Websphere 5.1

{RM_HOME} is the root directory for realMethods installation
{WAS_ROOT} is the root directory for the WAS installation
{APP_ROOT} is the root directory where you generated the POC code base.

Step 1- Starting the Websphere Application Server

If WAS is not already running, execute *startserver* located in your WAS_INSTALL_ROOT\Appserver\bin directory.

Step 2- Adminstrative Console Settings

Run the administrative console from a browser using the following URL:

```
http:\\localhost:9090\admin
```

From the left pane's tree control, click *Servers --> Application Servers*. Then click the name of the server you will deploy the POC application to (default is server1).

Click Process Definitions

Under the Configuration tab, set the following to the Executable Arguments:

```
-DFRAMEWORK_HOME={location of the framework.license file}
```

Note:
This is the location of your framework.license file.

Click Java Virtual Machine

Add the following to the Generic JVM arguments:

```
-DFRAMEWORK_HOME={location of the framework.license file}
```

Step 3 – Deploy the POC

From the tree control in the left pane, click on *Applications --> Install New Application.* Under *Local Path*, click *Browse* to locate *poc.ear* in the *dist* subdirectory of the {*APP_ROOT}* directory. Click *Next*.

Preparing for the application installation - Click Next
Application Security Warnings - ignore the warnings, and click Continue

Installing New Applications

Step 1: Provide options to perform the installation
- Apply the default by clicking *Next*.

Step 2: Provide options to perform the EJB Deploy
- Apply the default by clicking *Next*.

Step 3: Provide JNDI Names for Beans
- Cut and paste each name from the EJB column into the corresponding JNDI name column.
- Click Next

Step 4: Map virtual hosts for web modules
- Select the poc.war checkbox, targetted to the default_host

Step 5: Map modules to application servers
- Select the Module check box

Step 6: Ensure all unprotected 1.x methods have the correct level of protection
- Apply the default by clicking *Next*.

Step 7: Click Finish

Step 4 – Starting the POC

Once the application is successfully deployed, use the left-hand tree control to click *Applications --> Enterprise Applications*. Then select the checkbox associated with poc, and click the *Start* button.

Step 5 - Execution

Once the console indicates that the POC application has started, you should be able to access it using the following URL from your:

```
http://localhost:9080/poc/
```

Websphere 6.0

{RM_HOME} is the root directory for realMethods installation
{WAS_ROOT} is the root directory for the WAS installation
{APP_ROOT} is the root directory where you generated the POC code base.

Step 1- Starting the Websphere Application Server

If WAS is not already running, execute *startserver* located in your WAS_INSTALL_ROOT\Appserver\profiles\default\bin directory. In 6.0, WAS is installed as a Windows Service, so it might already be running

Step 2- Adminstrative Console Settings

Run the administrative console from a browser using the following URL:

```
http:\\localhost:9060\ibm\console
```

From the left pane's tree control, click *Servers --> Application Servers*. Then click the name of the server you will deploy the POC application to (default is server1).

- **Assign Framework Home**

Click *Application Server → server1 → Java and Process Management →Process Definition → Java Virtual Machine*

Under the Generic JVM Arguments option, enter the following

```
-DFRAMEWORK_HOME={location of the framework.license file}
```

Note:
This is the location of your framework.license file. If this location is not assigned properly, at runtime the framework will fail to license validation and will cause the application to cease running.

- **Assign the JDBC Driver Jar(s)**

In order to execute the POC, you will have to provide the name(s) of the jar file(s) that contain the JDBC Drivers to the database you are using. These classes are really required by Hibernate.

Click *Environment → Shared Libraries → New*

Assign a unique name to the jar file(s).

Now you need to apply this name to the deployed POC application by doing the following:
Click Application → Enterprise Applications → poc → Application Properties → Libraries → Add → Select the previously named shared library.

Step 3 – Deploy the POC
From the tree control in the left pane, click on *Applications --> Install New Application.* Under *Local Path*, click *Browse* to locate *poc.ear* in the *dist* subdirectory of the {*APP_ROOT}* directory. Click *Next*.

a.) Preparing for the application installation
- Click Next

b.) Application Security Warnings
- Ignore the warnings, and click *Continue*

c.) Installing New Applications

Step 1: Select installation options
- Use the defaults by clicking *Next*.

Step 2: Map modules to servers
- Click the checkboxes for each module.
- Click *Next*.

Step 3: Provide options to perform the EJB deploy
- Use the defaults by clicking *Next*.

Step 4: Provide JNDI Names for Beans
- Click *Next*

Step 5: Map EJB references to beans
- Click *Next*

Step 6: Map virtual hosts for web modules
- Select the poc Application checkbox, targeted to the *default_host*
- Click *Next*

Step 7: Ensure all unprotected 1.x methods have the correct level of protection
- Apply the default by clicking *Next.*

Step 8: Click Finish

Step 4 – Starting the POC
Once the application is successfully deployed, use the left-hand tree control to click *Applications --> Enterprise Applications*.

Click on the POC and select *Libraries* from the right-hand pane under *Additional Properties*. Click the *Add* button and select the name you previously provided when creating the JDBC library variable as outlined above.

Next, use the left-hand tree control again to click *Applications --> Enterprise Applications* select the checkbox associated with POC, and click the *Start* button.

Step 5 - Execution
Once the console indicates that the POC application has started, you should be able to access it using the following URL from your:

http://localhost:9080/poc/

JBoss

{RM_HOME} is the root directory for realMethods installation.
{JBOSS_HOME} is the root directory for the JBoss installation.
{APP_ROOT} is the root directory where you generated the POC code base.

Step 1

Modify {JBOSS_HOME}/jboss/bin/run.bat (or run.sh), and add the following to java runtime parameters

```
-DFRAMEWORK_HOME={location of the framework.license file}
```

Step 2

Copy the built *poc.ear* from the *dist* subdirectory of the {APP_ROOT} directory to the {JBOSS_HOME}/server/default/deploy directory.

Step 3

Modify file jboss-service.xml located in
{JBOSS_HOME}\server\default\deploy\jbossweb-tomcat55.sar\META-INF

and make the following change to attribute UseJBossWebLoader

```
<attribute name="UseJBossWebLoader">true</attribute>
```

This step is absolutely necessary so the deployed WAR and JAR file share the same classloader. If this step is skipped, the application will not run.

Step 4

Open your web browser and go to the following URL:

```
http://localhost:8080/poc/
```

Oracle 9iAS

{RM_HOME} is the root directory for realMethods installation.
{OC4J_HOME} is the root directory for the Oracle 9iAS/Orion installation.
{APP_ROOT} is the root directory where you generated the POC code base.

Step 1 - Copy ear file
Copy poc.ear from the *dist* subdirectory of the {APP_ROOT} directory to the {OC4J_HOME}\j2ee\home\applications directory

Step 2 - Copy jar files
Do not forget to include your database driver classes, etc.

Step 3 - Modify web-site XML file
Add the following entry to {OC4J_HOME}\j2ee\home\config\http-web-site.xml {this is default-web-site.xml for Orion and earlier versions of Oracle 9iAS}.

```
<web-app
   application="poc"
   name="poc
   root="/poc
/>
```

Step 4 - Modify server.xml file
Add the following entry to {OC4J_HOME}\j2ee\home\config\server.xml

```
<application
   name="poc"
   path="../applications/poc.ear"
/>
```

Step 5 - Provide Java System property
Start the application server by providing the following -D parameter

```
java -DFRAMEWORK_HOME={location of framework.license file}-jar oc4j.jar
```

Step 6 - Execute
Open your web browser and go to the following URL:

```
http://localhost:8888/poc/
```

Execution of the POC

You should refer to your application server user's guide to determine how to access a deployed servlet within a web application package.

In general, you will simply use the following URL to access the Proof of Concept:

```
http://hostname:port/poc/
```

As an example, for Weblogic users, the URL would look like:

```
http://localhost:7001/poc/
```

Once loaded, you have successfully deployed the Proof of Concept if the following screen appears:

Click on the Log On button to access the generated JSP screens. These screens and links will allow you to exercise all the tiers of your application.

Framework JUnit Test Cases

A class that represents a single component or an aspect of a service within the Framework is a candidate to be a JUnit Test Case. In the case of a service, it often may have a dependency on another service, but such dependencies should not cause any one test to fail, rather the result of that dependency should be logged.

Some classes' are purely intended to be sub-classed, and therefore are best tested within the context of a complete application. Within the Framework these include:

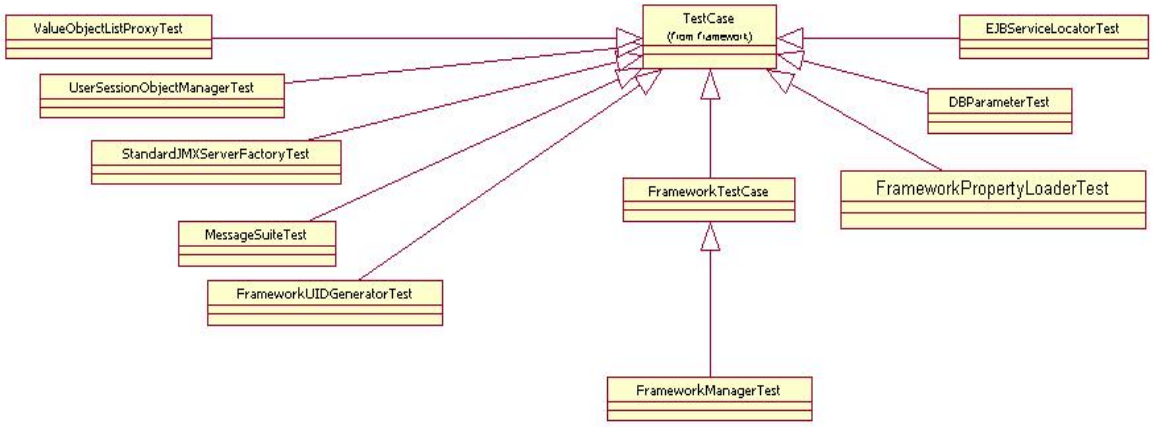
- EJBs
- Business Delegates
- Business Objects
- Primary keys
- Servlets
- JSPs
- Axis serializer classes

Using the AIB will simplify the generation and testing of these files because it is capable of generating an entire multi-tier J2EE application complete with working CRUD functionality.

Core Framework Components and Services

Component Level Testing

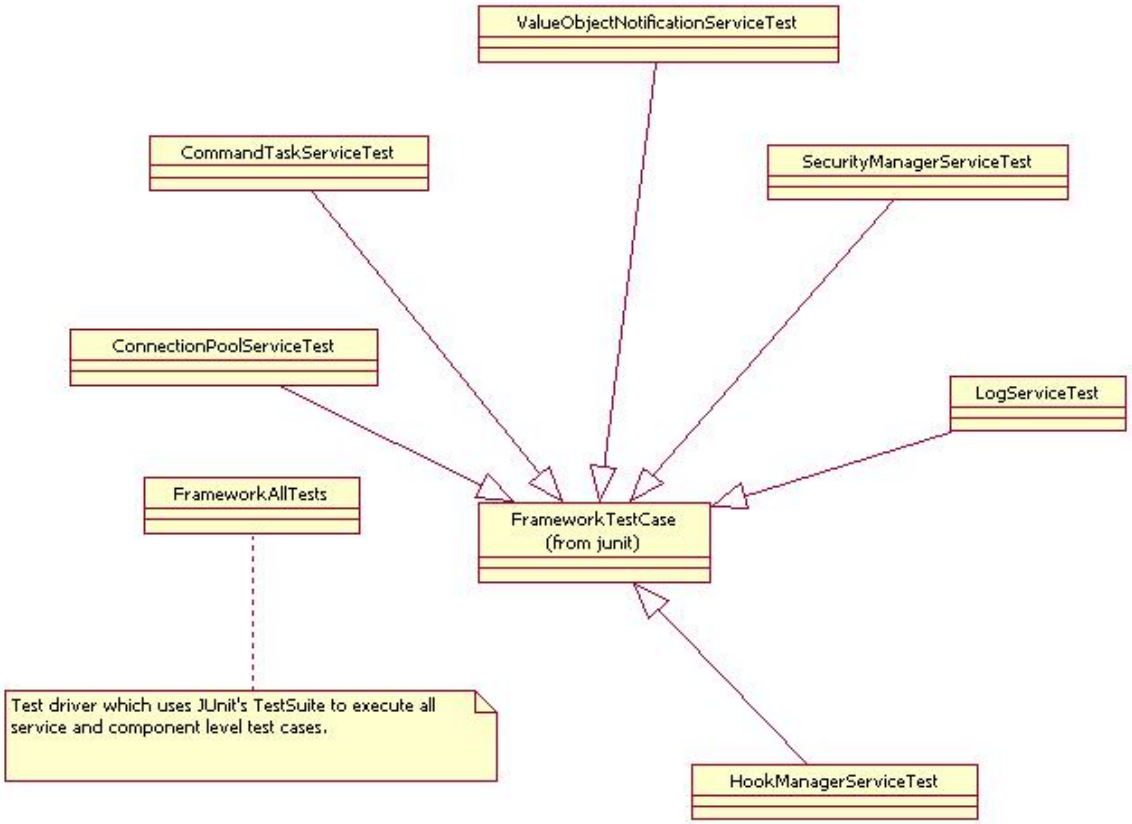
A candidate for this level of testing is normally leveraged within the context of a Framework service, or can provide its functionality with little or no external assistance. Each component level test is a JUnit TestCase, and is located in the com.framework.test.junit package. This package includes:



Service Level Testing

A candidate for this level involves the testing of 2 or more related classes, which as a whole define the Framework service. Because of its dependencies, testing such a class outside the context of the intended service will lead to unpredictable results.

Each service level test is a JUnit TestCase, and is located in the com.framework.test.junit.service package. This package includes:



Test Helpers

A few helper classes exist that simply sub-class a few important classes required for testing. Such classes include the sub-classing of FrameworkPrimaryKey and FrameworkValueObject. Each sub-class is found in the com.framework.junit.test.etc package.

Test definitions for each Test Case

To better understand the testing provided by each of the Framework test cases, refer to the Framework Javadocs.

Generated JUnit Test Cases

The AIB is capable of generating a set of JUnit Test cases, one for each persistent value object. The generated classes allow you to unit test all CRUD functionality by making all calls initiate from the generated business delegates. For the test cases to execute successfully be aware of the following:

- Database (and optionally LDAP) connectivity is in place.
- Based on your application server, make sure the JNDI_ARGS set correctly within the framework.xml config file used by the JUnit test case. This is necessary in cases where the business delegates need to communicate with the Session Facade. If you generated the business delegates to talk directly to the generated DAO classes, this step is not necessary.
- If EJBs are in use, be sure the application server is running and EJBs are deployed.

Servlet Test Helpers

When testing a servlet-based application, it may be more convenient to do so in a simulated environment, rather than within an actual running servlet container. The following open source project is JUnit based, and will assist in this.

HttpUnit/ServletUnit

- Open source project hosted on SourceForge
- HttpUnit
  - Java API for accessing web sites
  - Not a test tool but a programmable web client
  - "black-box" response; functional testing of web sites
- ServletUnit
  - Enables unit test for servlets
  - Provides a suitable servlet environment
  - Based on a simulated servlet container

StrutsTestCase for JUnit - http://strutstestcase.sourceforge.net/

It is an extension of the standard JUnit TestCase class that provides facilities for testing code based on the Struts framework. StrutsTestCase uses mock servlet objects to actually run the Struts ActionServlet, allowing you to test your Struts code without a running servlet engine. Because StrutsTestCase uses the ActionServlet controller to test your code, you can test not only the implementation of your Action objects, but also your mappings, form beans, and forwards declarations. And since StrutsTestCase does not require a running servlet engine, it's quick and easy to write unit test cases.

Executing the Framework Test Suite

In order to execute all included JUnit test cases as one test, you will need to execute the Framework Test Suite, handled by the single class:

```
com.framework.test.junit.service.FrameworkAllTests
```

Cut and paste the following into a batch file or shell script, modifying the necessary variable values for your application server, database in use (if applicable), and directory locations in general:

```
set RM_LIBS=c:\realmethods\lib

set
DATABASE_JARS=C:\sqlserver\lib\msbase.jar;C:\sqlserver\lib\mssqlserver.ja
r; C:\sqlserver\lib\msutil.jar

set APP_SERVER_JARS=C:\jboss-3.2.0\client\jbossall-client.jar

set J2EE_JAR=C:\java\j2sdkee1.3.1\lib\j2ee.jar

set TEST_CP=%APP_SERVER_JARS%;%DATABASE_JARS;
%RM_LIBS%\realmethodsframework.jar;%RM_LIBS%\struts.jar;
%RM_LIBS%\log4j-1.2.4.jar

set TEST_CP=%TEST_CP%;%RM_LIBS%\xercesImpl.jar;
%RM_LIBS%\xml-apis.jar;%RM_LIBS%\soap.jar;%RM_LIBS%\jmxri.jar;

set TEST_CP=%TEST_CP%;%RM_LIBS%\junit.jar;%RM_LIBS%\jmxtools.jar;

set TEST_CP=%TEST_CP%;%J2EE_JAR%

set TEST_CP=%TEST_CP%;%RM_LIBS%\poc.jar; %RM_LIBS%\poc_ejb.jars

set PROPS_DIR=C:\realmethods\ test\properties

set ARGS=-DPROPERTIES_LOCATION=%PROPS_DIR% -
Djava.security.manager -Djava.security.policy==%PROPS_DIR%/java2.policy
Djava.security.auth.login.config==%PROPS_DIR%/jaas.config
```

```
C:\java\j2sdk1.4.1_02\bin\java -classpath
%TEST_CP%;C:\java\j2sdkee1.4\lib\j2ee.jar %ARGS%
com.framework.test.junit.service.FrameworkAllTests
```

Note:

In order to test the Connection Pool Management aspects of the Framework, as well as the Standard JMS Connections, make sure you have the \RM_HOME\test\connectionpool.xml file set up correctly. Also, the Framework related property files used for testing are different than those located in the \realmethods\home directory, used by the POC. The test related configuration files are located in the \RM_HOME\test directory

In order to test a JAAS Security Manager as defined in the security.xml file, be sure the included java2.policy and jaas.config files are provided as arguments, illustrated by the ARGS property above. Also, you will need to use JDK 1.4 or above for both J2SE and J2EE.

Due to some known class loader issues when using JUnit's graphical tester , junit.swingui.TestRunnner, we recommend you do not use it when testing a JAAS Security Manager.

If the test requires the use of EJB's on a remote application server, make sure the Framework has been started on that app. server.

Web Services

Overview

A client will access the business functionality via one or more generated business delegates. When a session façade is generated, the AIB provides the option to use Apache Axis (SOAP Engine) as the means of a business delegate to communicate with the session façade. In this fashion, the business tier is now readily available as a set of Web Services, to be accessed from any application supporting the standard.

As of version 5.0, the framework uses the Mule ESB to expose the framework generated business delegates as Web Services.

Frequently Asked Questions

Installation

Where do I put the framework.license file?

For the purpose of evaluation the Framework and deploying the Proof of Concept application, the framework.license file should be placed in both the RM_HOME\home and the RM_HOME\aib directories.

In fact, this file can reside anywhere, so long as your -DFRAMEWORK_HOME System property refers to the same directory.

What jar files are required?

All required jar files are located in the RM_HOME\lib directory and subdirectories. When the AIB generates all application files, the resulting ANT build.xml file will build an application that contains all the jar files necessary to deploy the application. **However**, it will not contain the jar files required to access your database when DAO/Hibernate is in use. Make sure these jar files are available when running the build.xml (see property *db.jars* in the build.xml file) and when running the application on your app. Server.

Proof of Concept

Problems executing the POC

1. Database preparation

It is important that your database be accessible using either:

a.) *userid, password, url, and driver manager* properties provided to Hibernate for direct database connection scenarios.
b.) Datasource JNDI name for DAO/Hibernate.

For Hibernate, these values are provided during code generation by clicking the "Hibernate Properties" button within the AIB. The values provided here are then placed within the generated *\properties\hibernate.cfg.xml* file

Note:
Make sure the db.jars property of the generated ANT build.xml file points to the jar files required of your database.

2. Failed EJB deployment

Although it may seem as though your initial step of deploying the POC was successful, you may encounter the following error when executing:

```
2002-09-19 15:10:15,659 - Framework Msg -
ProcessingException::ProcessingException() - Inside
HTTPRequestHandler:handleProcessingException() - processing exception
because of: com.framework.exception.ProcessingException:
LogonWorkerBean:loadAllEmployers() - successfully loaded all
EmployerValueObjects - com.framework.exception.ProcessingException:
Inside EmployerProxy:getEmployer(..) - javax.naming.NamingException:
Utility:lookupHomeInterface() - javax.naming.NameNotFoundException:
Unable to resolve 'com_poc_EmployerServiceHome' Resolved: ''
Unresolved:'com_poc_EmployerServiceHome' ; remaining name
'com_poc_EmployerServiceHome'
```

This error, taken from the rm_framework_log4j.log file, indicates that JNDI does not contain an entity by the name of com_poc_EmployerServiceHome. This means the POC EJBs have not been deployed successfully.

3. License File validation

Assuming you have an up to date, valid license file, make sure your application server execution includes the following Java System property:

FRAMEWORK_HOME={location of the framework.license file}

If provided on the command line, use the -D argument:

-D FRAMEWORK_HOME={location of the framework.license file}

If this Java System property is not provided or is provided incorrectly, the Framework will not be able to validate your framework.license file, causing Framework startup to terminate.

Logging

Framework Logging doesn't seem to be working.

When an application is generated using the AIB, the Framework Default Logger is already defined within the \properties\loghandlers.xml configuration fiile:

```
<logHandler name="FrameworkDefaultLog"
          synchronous="true"
          debug="SYSTEM_OUT, poc"
          info="SYSTEM_OUT, ,poc"
        warn="SYSTEM_OUT, poc"
        error="SYSTEM_OUT, poc"
        dateTimeStampFormat="yyyy.mm.dd-hh:mm:ss"
      />
```

The name *poc* is the name of the application provided to the AIB during code generatioin. It is also the name of a Log4J Logger, defined in the generated log4j.xml.

```
        <logger name="poc">
                <level value="debug"/>
                <appender-ref ref="TEMP"/>
        </logger>
```

If you have changed the name of your application (in the web.xml), make sure you have changed the name here as well as in the log4j.xml file.

Also, since the Framework Logging Service makes use of the Command/Task architecture, the following standard entry for the task.xml configuration file must exist:

```
<task name="FrameworkLogTask"
commands="com.framework.integration.command.FrameworkLogCommand"/
>
```

Without the FrameworkLogTask entry, the Framework Logging Service will not work, nor will the FrameworkDefaultLogger.

Note
Since internal Framework Logging, and optionally defined Log Handlers, make use of Log4J, make sure the Log4j jar file included in the \lib directory of the realmethods installation is able to be located by the classloader.

Execution

The Framework fails during license validation

Framework license validation normally fails due to one of the following reasons:

1. The license file is not in the directory location as specified by the -DFRAMEWORK_HOME System property.

2. The license file has expired. For evaluators, the normal license file is good for up to 30 days. Licensed users should never encounter this problem.

3. You have an invalid (outdated in terms of format) or corrupt license file if you see the following message:

```
*** The Framework is shutting down due to the following reason:
*** You are using the realMethods Framework with an unauthorized or invalid license.
*** Contact support@realmethods.com for a valid license.
```

If this is the case, you need to request another license file from support.