Politecnico di Milano

Software Engineering 2 Project
a.y. 2015-16

MyTaxiService
Inspection Document
Version 1.0

Casati Fabrizio, 853195
Castelli Valerio, 853992

Referent professor: Di Nitto Elisabetta

December 28, 2015

Contents

Chapter 1

Introduction

Chapter 2

Functional Role

This chapter provides an overview of the functional role of the **ActiveJmsResourceAdapter** class, and specifically of its **createManagedConnectionFactory** and **createManagedConnectionFactories** methods, in the general context of the whole Glassfish Project.

In order to make the analysis more clear and understandable, we provide a brief list of acronyms and definitions which are used throughout the code:

- **EIS: Enterprise Information System**. It's a generic term that refers to any kind of information system (or component thereof) used in an enterprise context, including ERP (Enterprise Resource Planning) systems, message brokers and others.
- **RA: Resource Adapter**. It's a component that implements the adapter design pattern to let two systems communicate with each other when there is a discrepancy between the offered and required interfaces. In the specific case of Glassfish, resource adapters are used to interface with external EISs.
- **MQ: Message Queue**. Despite the name, this component usually doesn't simply implement a location where messages are stored until delivery, but takes care of the whole stack of messaging functionalities across a network of interplaying components.
- **imq properties**: set of configuration properties of the IBM WebSphere MQ suite, implemented by Glassfish as part of the resource adapter interface with IBM MQ compliant Message Brokers. Supported mainly for compatibility reasons since they've been superseded by later revisions of the MQ API.
- **AS7 properties**: set of properties of Application Servers, implemented both by JBoss and Glassfish. They include the already mentioned imq properties.

- **XA standard**: specification written by The Open Group for distributed transaction processing (DTP). It describes the interface between the global transaction manager and the local resource manager. The goal of XA is to allow multiple resources (such as databases, application servers, message queues, transactional caches, etc.) to be accessed within the same transaction, thereby preserving the ACID properties across applications. XA uses a two-phase commit to ensure that all resources either commit or rollback any particular transaction consistently (all do the same).[1]

- **RM: Resource Manager**. It is the component responsible for managing a set of resources; in the particular case of Message Queues, it coincides with the message broker.

- **MCF: Managed Connection Factory**. It is the common interface implemented by all the adapter classes which provide connections toward an external service. As specified by the official Oracle Java EE Documentation[2], a connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator and provides to clients the necessary methods to create a connection to a provider.

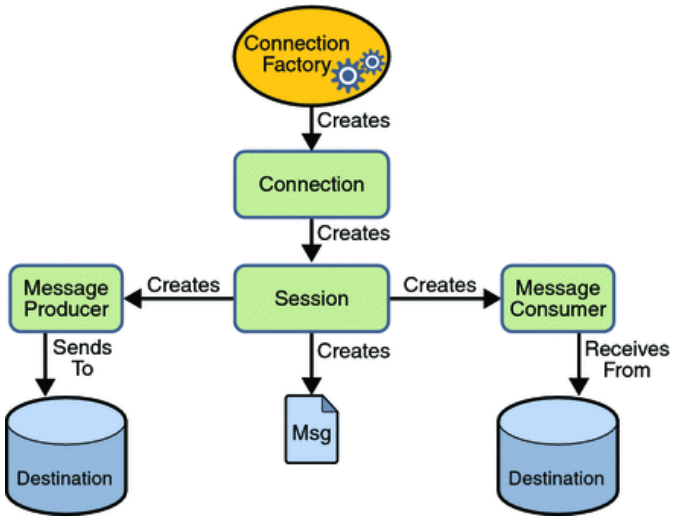
The primary purpose of ActiveJmsResourceAdapter is to provide a communication interface between Glassfish and any kind of external message broker supporting the JMS API.

As defined by Wikipedia[3],

> the **Java Message Service (JMS) API** is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. It is a messaging standard that allows application components based on the Java Enterprise Edition (Java EE) to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous.

The full JMS specification is available at Oracle's Developer Website[4]; we include the key architectural diagram here:

[1] `https://en.wikipedia.org/wiki/X/Open_XA`
[2] `https://docs.oracle.com/javaee/5/tutorial/doc/bnceh.html`
[3] `https://en.wikipedia.org/wiki/Java_Message_Service`
[4] `http://www.oracle.com/technetwork/java/jms/index.html`



The ActiveJmsResourceAdapter component is essential for Glassfish to provide its hosted applications the messaging functionalities of the JMS API without having to implement a full message broker directly; instead, the system administrator is able to freely choose one (or more) of the existing commercial message brokers and configure Glassfish to route all JMS calls to it. In practice, it gives hosted applications a virtual, monolithic JMS message broker that they can interact with, fully abstracting all the implementation details of the specific adopted software solution.

In particular, it has the ability to connect with multiple message brokers at the same time to evenly distribute the amount of messages to be routed for performance and scalability reasons. Since each message broker has its own specific set of APIs, the actual implementation of the communication adapter is not directly provided by ActiveJmsResourceAdapter, which only acts as a general interface, but is offered by a set of different classes (one for each message broker) implementing the ManagedConnectionFactory interface which are dynamically loaded at runtime when needed.

Each distinct message broker is identified in Glassfish with a connector, which in turn can be associated with multiple connections at the same time. All the implementation details required to correctly interact with the message broker, including the name of the adapter class to be loaded, the values of the configuration properties and parameters and the maximum number of supported connections are specified by a ConnectorConnectionPool object. Despite the name, this object doesn't implement itself a connection pool, as it doesn't keep track of the connections that have been opened toward a certain message broker, instead being specifically concerned with the configuration details of the interface. The information provided by the ConnectorConnectionPool object is then used to dynamically load and instantiate the required ManagedConnectionFactory class.

The createManagedConnectionFactory method is precisely involved with the setup of the ManagedConnectionFactory object responsible for the communication with JMS compliant message brokers. The first instruction is a call to the inherited version of this same method, which performs three tasks:

- It dynamically loads the class implementing the desired ManagedConnectionFactory implementation and instantiates it;
- For every property enumerated in the given ConnectorConnectionPool, it uses Java reflection to find out which is the corresponding setter method and invokes it to assign the proper value to all the specified class attributes;
- If everything has been successful, it returns a pointer to the created ManagedConnectionFactory instance; otherwise, it returns null.

However, this code is only able to configure properties for which a setter with the same name exists. Apparently, older implementations of the MQ API only provided properties in the form of ⟨property, value⟩ tuples which had to be passed to a setProperty(String property, String value) method; this properties starting with the "imq" prefix are thus not set by the initial call of the super createManagedConnectionFactory method.

For this reason, if a valid ManagedConnectionFactory instance is returned, the method proceeds to check all the properties stated in the ConnectorConnectionPool object looking for those who have "imq" as a prefix and, if any of them is found, it uses Java reflection to obtain the setProperty(String property, String value) method of the ManagedConnectionFactory and invokes it with the corresponding ⟨property, value⟩ tuple. If no such method is found, then an exception is raised.

For compatibility reasons, most properties still have an old, deprecated imq version and a newer official version, which should be used in lieu of the old one. If the two are present with different values in the same file, it is assumed that the newer one should be considered as the legal one.

However, since the imq loop is executed after all "newer" properties have already been set, if a lazy developer has actually included both the imq version and the newer official version of the same property in the same file, the value of the imq version will overwrite the newer one as its corresponding setter is called later.

As this is in contrast with the specification, Glassfish includes a fix for this behavior which consists of simply re-invoking all the setter methods for the official properties, with the single exception of the addressList property which is removed in case the address is equal to localhost or is empty (as it is considered to be "less important" than a non-empty and non-localhost imq addressList). This check is only performed if the interfacing module is a JMS

module, which in fact is a superfluous check as ActiveJmsResourceAdapter only deals with JMS modules.

As we were mentioning earlier, ActiveJmsResourceAdapter is able to associate multiple message brokers to the same virtual connector in order to achieve better scalability and performances. The interactions between all the resource managers (in this case, message brokers) are governed by the XA standard, which uses two-phase commit to ensure that transactions related to the processing of messages are correctly handled by all nodes.

The createManagedConnectionFactories method is invoked when it's necessary to gather all the ManagedConnectionFactories corresponding to a single pool of message broker addresses.

As the author of the code states, this is particularly useful when the XA resources of all resource managers in a broker cluster are required for a certain transaction and thus we need to simultaneously get a reference to all the message brokers in the cluster to eventually instantiate a connection with each of them. In this case, the process is quite straightforward.

At first, the method determines how many message brokers are part of the cluster by counting the number of remote addresses in the pool. If only one message broker is present, a single ManagedConnectionFactory is created; otherwise, a list of all addresses is obtained by analyzing both the imqAddressList and the AddressList properties (if present) and for each of them a ManagedConnectionFactory is created. The process is similar to what we've already seen for the single ManagedConnectionFactory: the super method is invoked for the initial allocation and configuration phase, then all imq properties are set. The conflict resolution policy for imq and non-imq properties adopted in this case is different from what we've seen in the single factory case: this time, only the addressList property is resolved giving more importance to its non-imq version, while for all other properties the imq version overrides the non-imq one if both are present.

This different behavior is not documented at all, though, so our understanding is that the conflict resolution policy may not be stated by the properties specifications.

Chapter 3

Code issues

This chapter highlights all the bad practices that we have recognized during the inspection of ActiveJmsResourceAdapter, and specifically in methods createManagedConnectionFactory and createManagedConnectionFactories. The issues we have identified can be either traced back to specific violations of the checklist, or to other bad programming practices.

3.1 Notation

- Single lines of code are denoted as L.123.
- Intervals of lines of code are denoted as L.123-456.
- Specific points in the code inspection checklist are denoted as **C1**, **C2**, ...

3.2 Checklist issues

3.2.1 ActiveJmsResourceAdapter class

1. **C1** Constant ADDRESSLIST at L.239 and variable addressList at L.255 rely only on Java's case sensitivity to be distinguished one from the other.
2. **C1** Variable addressList at L.255 and variable urlList at L.253 refer to similar concepts and should probably be more clearly described.
3. **C1** Variable nm at L.306 and variable sm at L.250 don't really suggest that they represent instances of GlassfishNamingManager and StringManager, respectively.
4. **C1** The name of variable brkrPort at L.257 is unnecessarily short (variable brokerInstanceName at L.297 uses the full word "broker").

5. **C1** The name of method handles(ConnectorDescriptor cd, String moduleName) at L.1161 doesn't clearly described what this method is supposed to, nor why it is returning a boolean.

6. **C5** The name of method listToProperties at L.1101 is not a verb. The same is valid for method postConstruct at L.342 and for method postRAConfiguration at L.860.

7. **C7** There is a huge number of constants which do not follow the naming convention at L.166-202, L.207-211, L.213, L.233-237, L.242, L.246, L.262-264.

8. **C8** and **C9** are not followed consistently throughout the code. Most of the class uses four spaces indentation, but tabs are also occasionally used as at L.357, L.493, L.537-556, L.992-997, L.1008-1010-, L.1026-1030, L.1102-1112, L.1351-1357, L.1976-1978.

9. **C11** There are a number of single-line if statements that do not follow the convention. It would be very time consuming to enumerate them all, but at L.1245, L.1254, L.1303, L.1307 there are a few examples of this behavior.

10. **C14** There are a number of occurrences in which the line length exceeds the 120 character limit: at L.153(130), L.423(134), L.604(169), L.756(123), L.794(130), L.1355(137), L.1358(136), L.1841(131), L.1922(125), L.1958(141), L.2008(132), L.2021(141), L.2140(130), L.2262(125), L.2285(149), L.2290(132), L.2296(131), L.2320(126), L.2333(145), L.2334(129), L.2340(135), L.2342(125), L.2349(129), L.2361(129), L.2364(125), L.2369(149), L.2396(135), L.2410(133), L.2412(135), L.2443(123), L.2474(128), L.2590(135), L.2605(131).

11. **C18** Some methods don't have an explanation of what they're supposed to do. Also, large pieces of code are left uncommented or only have extremely brief explanations of their purpose.

12. **C19** None of the occurrences of commented out code contains an explanation as to why it has been commented out.

13. **C23** Javadoc is not complete. In particular, the following public methods are not documented:

    - public int getAddressListCount() at L.2512
    - public static String getBrokerInstanceName(JmsService js) at L.1137
    - public boolean getDoBind() at L.1534
    - public Set⟨String⟩ getGrizzlyListeners() at L.441 OSS FAR NOTARE QUESTA BAD PRACTICE DI RITORNARE VARIABILI PRIVATECI SONO TONNELLATE DI GETJMSDESTINATION oss far notare questa bad practice di ritornare variabili private

TUTTE UGUALI A CUI CAMBIANO SOLO I PARAMETRI, DA CONTROLLARE PIÙ A FONDO

ci sono tonnellate di getjms-destination tutte uguali a cui cambiano solo i parametri, da controllare più a fondo

- public void handleRequest(SelectableChannel selectableChannel) at L.2556
- public boolean handles(ConnectorDescriptor cd, String moduleName) at L.1161 (questo poi è proprio controintuitivo, da segnalare)
- public boolean initializeService() at L.2541
- public void postConstruct() at L.342
- public void setMasterBroker(String newMasterBroker) at L.2581
- public void setup() at L.562
- public void validateActivationSpec(ActivationSpec spec) at L.1165

14. **C25** There are a number of issues in the order of declarations:
    - Static variables are not listed in order of visibility. Package level and protected static variables are not used, however private and public static variables are mixed together. See L.156-246, L.262-271, L.276-291 for reference.
    - Instance variables are interleaved with static variables instead of being declared all at once after them. See L.250-259, L.273, L.295-332 for reference.
15. **C26** WE SHOULD CHECK THIS ONE MORE THOROUGHLY

we should check this one more thoroughly

16. **C27** Issues:
    - There are a number of methods which seem to be too long and should probably be split. They are:
      - private void setAvailabilityProperties() at L.582-762
      - private void loadDBProperties(JmsAvailability jmsAvailability, ClusterMode clusterMode) at L.764-828
      - private void setLifecycleProperties() at L.919-1099
      - private void setJmsServiceProperties(JmsService service) at L.1601-1673
      - public ManagedConnectionFactory [] createManagedConnectionFactories (com.sun.enterprise.connectors.ConnectorConnectionPool cpr, ClassLoader loader) at L.1860-1941
      - public ManagedConnectionFactory createManagedConnectionFactory(com.sun.enterprise.connectors.ConnectorConnectionPool cpr, ClassLoader loader) at L.1970-2046
      - public void updateMDBRuntimeInfo(EjbMessageBeanDescriptor descriptor_, BeanPoolDescriptor poolDescriptor) at L.2064-2223

3.2.2 createManagedConnectionFactory method

Chapter 4

Other Problems

Appendix A

Checklist

Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
2. If one-character variables are used, they are used only for temporary "throwaway" variables, such as those used in for loops.
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`;
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.
6. Class variables, also called attributes, are mixed case, but might begin with an underscore ('_') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: `_windowHeight`, `timeSeriesData`.
7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

Indention

8. Three or four spaces are used for indentation and done so consistently.
9. No tabs are used to indent.

Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).

11. All `if`, `while`, `do-while`, `try-catch`, and `for` statements that have only one statement to execute are surrounded by curly braces. Example: avoid this:

```
if ( condition )
    doThis();
```

instead do this:

```
if ( condition )
{
    doThis();
}
```

File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).

13. Where practical, line length does not exceed 80 characters.

14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

15. Line break occurs after a comma or an operator.

16. Higher-level breaks are used.

17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.

19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

Java Source Files

20. Each Java source file contains a single public class or interface.

21. The public class is the first class or interface in the file.

22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.

23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

Class and Interface Declarations

25. The class or interface declarations shall be in the following order:

    (a) class/interface documentation comment;
    (b) class or interface statement;
    (c) class/interface implementation comment, if necessary;
    (d) class (static) variables;
        i. first public class variables;
        ii. next protected class variables;
        iii. next package level (no access modifier);
        iv. last private class variables.
    (e) instance variables;
        i. first public instance variables;
        ii. next protected instance variables;
        iii. next package level (no access modifier);

iv. last private instance variables.

(f) constructors;

(g) methods.

26. Methods are grouped by functionality rather than by scope or accessibility.

27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).

29. Check that variables are declared in the proper scope.

30. Check that constructors are called when a new object is desired.

31. Check that all object references are initialized before use.

32. Variables are initialized where they are declared, unless dependent upon a computation.

33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '{' and '}'). The exception is a variable can be declared in a `for` loop.

Method Calls

34. Check that parameters are presented in the correct order.

35. Check that the correct method is being called, or should it be a different method with a similar name.

36. Check that method returned values are used properly.

Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).

38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.

39. Check that constructors are called when a new array item is desired.

Object Comparison

40. Check that all objects (including Strings) are compared with `equals` and not with `==`.

Output Format

41. Check that displayed output is free of spelling and grammatical errors.
42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.
43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming": (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).
45. Check order of computation/evaluation, operator precedence and parenthesizing.
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
47. Check that all denominators of a division are prevented from being zero.
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.
49. Check that the comparison and Boolean operators are correct.
50. Check throw-catch expressions, and check that the error condition is actually legitimate.
51. Check that the code is free of any implicit type conversions.

Exceptions

52. Check that the relevant exceptions are caught.
53. Check that the appropriate action are taken for each catch block.

Flow of Control

54. In a `switch` statement, check that all cases are addressed by break or return.
55. Check that all switch statements have a default branch.
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. Check that all files are properly declared and opened.
58. Check that all files are closed properly, even in the case of an error.
59. Check that EOF conditions are detected and handled correctly.
60. Check that all file exceptions are caught and dealt with accordingly.

Appendix B

Hours of work

To redact this document, we spent 10 hours per person.