Red Hat Fuse 7.9

Apache Karaf Transaction Guide

Write transactional applications for the Apache Karaf container

Last Updated: 2023-07-24

Red Hat Fuse 7.9 Apache Karaf Transaction Guide

Write transactional applications for the Apache Karaf container

Legal Notice

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Developing transaction-aware applications for Fuse

Table of Contents

PREFACE 5

MAKING OPEN SOURCE MORE INCLUSIVE 6

CHAPTER 1. INTRODUCTION TO TRANSACTIONS 7
1.1. WHAT IS A TRANSACTION? 7
1.2. ACID PROPERTIES OF A TRANSACTION 7
1.3. ABOUT TRANSACTION CLIENTS 7
1.4. DESCRIPTIONS OF TRANSACTION TERMS 8
1.5. MANAGING TRANSACTIONS THAT MODIFY MULTIPLE RESOURCES 9
1.6. RELATIONSHIP BETWEEN TRANSACTIONS AND THREADS 9
1.7. ABOUT TRANSACTION SERVICE QUALITIES 10
1.7.1. Qualities of service provided by resources 10
1.7.1.1. Transaction isolation levels 10
1.7.1.2. Support for the XA standard 10
1.7.2. Qualities of service provided by transaction managers 11
1.7.2.1. Support for suspend/resume and attach/detach 11
1.7.2.2. Support for multiple resources 11
1.7.2.3. Distributed transactions 11
1.7.2.4. Transaction monitoring 11
1.7.2.5. Recovery from failure 12

CHAPTER 2. GETTING STARTED WITH TRANSACTIONS ON KARAF (OSGI) 13
2.1. PREREQUISITES 13
2.2. BUILDING THE CAMEL-JMS PROJECT 14
2.3. EXPLANATION OF THE CAMEL-JMS PROJECT 16

CHAPTER 3. INTERFACES FOR CONFIGURING AND REFERENCING TRANSACTION MANAGERS 20
3.1. WHAT TRANSACTION MANAGERS DO 20
3.2. ABOUT LOCAL, GLOBAL, AND DISTRIBUTED TRANSACTION MANAGERS 20
3.2.1. About local transaction managers 20
3.2.2. About global transaction managers 21
3.2.3. About distributed transaction managers 21
3.3. USING A JAVAEE TRANSACTION CLIENT 22
3.4. USING A SPRING BOOT TRANSACTION CLIENT 23
3.4.1. Using the Spring PlatformTransactionManager interface 24
3.4.1.1. Definition of the PlatformTransactionManager interface 24
3.4.1.2. About the TransactionDefinition interface 25
3.4.1.3. Definition of the TransactionStatus interface 25
3.4.1.4. Methods defined by the PlatformTransactionManager interface 25
3.4.2. Steps for using the transaction manager 25
3.4.3. About Spring PlatformTransactionManager implementations 26
3.4.3.1. Local PlatformTransactionManager implementations 26
3.4.3.2. Global PlatformTransactionManager implementation 26
3.5. OSGI INTERFACES BETWEEN TRANSACTION CLIENTS AND THE TRANSACTION MANAGER 27

CHAPTER 4. CONFIGURING THE NARAYANA TRANSACTION MANAGER 29
4.1. ABOUT NARAYANA INSTALLATION 29
4.2. TRANSACTION PROTOCOLS SUPPORTED 31
4.3. ABOUT NARAYANA CONFIGURATION 31
4.4. CONFIGURING LOG STORAGE 32

CHAPTER 5. USING THE NARAYANA TRANSACTION MANAGER 34

5.1. USING USERTRANSACTION OBJECTS 34
5.1.1. Definition of the UserTransaction interface 34
5.1.2. Description of UserTransaction methods 34
5.2. USING TRANSACTIONMANAGER OBJECTS 35
5.2.1. Definition of the TransactionManager interface 36
5.2.2. Description of TransactionManager methods 36
5.3. USING TRANSACTION OBJECTS 37
5.3.1. Definition of the Transaction interface 37
5.3.2. Description of Transaction methods 37
5.4. RESOLVING THE XA ENLISTMENT PROBLEM 38
5.4.1. How to enlist an XA resource 38
5.4.2. About auto-enlistment 38

CHAPTER 6. USING JDBC DATA SOURCES 40
6.1. ABOUT THE CONNECTION INTERFACE 40
6.2. OVERVIEW OF JDBC DATA SOURCES 41
6.2.1. Database specific and generic data sources 42
6.2.2. Some generic data sources 43
6.2.3. Pattern to use 44
6.3. CONFIGURING JDBC DATA SOURCES 45
6.4. USING THE OSGI JDBC SERVICE 46
6.4.1. PAX-JDBC configuration service 49
6.4.2. Summary of handled properties 52
6.4.3. How the pax-jdb-config bundle handles properties 54
6.5. USING JDBC CONSOLE COMMANDS 56
6.6. USING ENCRYPTED CONFIGURATION VALUES 57
6.7. USING JDBC CONNECTION POOLS 58
6.7.1. Introduction to using JDBC connection pools 58
6.7.2. Using the dbcp2 connection pool module 60
6.7.2.1. Configuration properties for BasicDataSource 60
6.7.2.2. Example of how to configure DBCP2 pool 62
6.7.3. Using the narayana connection pool module 66
6.7.4. Using the transx connection pool module 66
6.8. DEPLOYING DATA SOURCES AS ARTIFACTS 66
6.8.1. Manual deployment of data sources 67
6.8.2. Factory deployment of data sources 69
6.8.3. Mixed deployment of data sources 71
6.9. USING DATA SOURCES WITH THE JAVA™ PERSISTENCE API 75
6.9.1. About data source references 75
6.9.2. Referring to JNDI names 75

CHAPTER 7. USING JMS CONNECTION FACTORIES 77
7.1. ABOUT THE OSGI JMS SERVICE 77
7.2. ABOUT THE PAX-JMS CONFIGURATION SERVICE 78
7.2.1. Creating a connection factory for AMQ 7.1 79
7.2.2. Creating a connection factory for IBM MQ 8 or IBM MQ 9 82
7.2.3. Using JBoss A-MQ 6.3 Client in Fuse on Apache Karaf 84
7.2.3.1. Prerequisites 84
7.2.3.2. Procedure 84
7.2.4. Summary of handled properties 86
7.3. USING JMS CONSOLE COMMANDS 86
7.4. USING ENCRYPTED CONFIGURATION VALUES 88
7.5. USING JMS CONNECTION POOLS 88

7.5.1. Introduction to using JMS connection pools 88
7.5.2. Using the pax-jms-pool-pooledjms connection pool module 89
7.5.3. Using the pax-jms-pool-narayana connection pool module 93
7.5.4. Using the pax-jms-pool-transx connection pool module 93
7.6. DEPLOYING CONNECTION FACTORIES AS ARTIFACTS 93
7.6.1. Manual deployment of connection factories 94
7.6.2. Factory deployment of connection factories 95
7.6.3. Mixed deployment of connection factories 97

CHAPTER 8. ABOUT JAVA CONNECTOR ARCHITECTURE 101
8.1. SIMPLE JDBC ANALOGY 101
8.2. OVERVIEW OF USING JCA 101
8.3. ABOUT THE PAX-TRANSX PROJECT 103

CHAPTER 9. WRITING A CAMEL APPLICATION THAT USES TRANSACTIONS 107
9.1. TRANSACTION DEMARCATION BY MARKING THE ROUTE 107
9.1.1. Sample route with JDBC resource 108
9.1.2. Route definition in Java DSL 108
9.1.3. Route definition in Blueprint XML 109
9.1.4. Default transaction manager and transacted policy 109
9.1.5. Transaction scope 109
9.1.6. No thread pools in a transactional route 110
9.1.7. Breaking a route into fragments 110
9.1.8. Resource endpoints 111
9.1.9. Sample route with resource endpoints 112
9.2. DEMARCATION BY TRANSACTIONAL ENDPOINTS 112
9.2.1. Sample route with a JMS endpoint 113
9.2.2. Route definition in Java DSL 113
9.2.3. Route definition in Blueprint XML 113
9.2.4. DSL transacted() command not required 114
9.2.5. Transactional endpoints at start of route 114
9.3. DEMARCATION BY DECLARATIVE TRANSACTIONS 114
9.3.1. Bean-level declaration 115
9.3.2. Top-level declaration 115
9.3.3. Description of tx:transaction attributes 116
9.4. TRANSACTION PROPAGATION POLICIES 117
9.4.1. About Spring transaction policies 117
9.4.2. Descriptions of propagation behaviors 117
9.4.3. Defining policy beans in Blueprint XML 118
9.4.4. Sample route with PROPAGATION_NEVER policy in Java DSL 119
9.4.5. Sample route with PROPAGATION_NEVER policy in Blueprint XML 120
9.5. ERROR HANDLING AND ROLLBACKS 120
9.5.1. How to roll back a transaction 120
9.5.1.1. Using runtime exceptions to trigger rollbacks 120
9.5.1.2. Using the rollback() DSL command 121
9.5.1.3. Using the markRollbackOnly() DSL command 122
9.5.2. How to define a dead letter queue 122
9.5.3. Catching exceptions around a transaction 123

PREFACE

This guide provides information and instructions for implementing Fuse transactional applications. The information is organized as follows:

- Chapter 1, *Introduction to transactions*
- Chapter 2, *Getting started with transactions on Karaf (OSGi)*
- Chapter 3, *Interfaces for configuring and referencing transaction managers*
- Chapter 4, *Configuring the Narayana transaction manager*
- Chapter 5, *Using the Narayana transaction manager*
- Chapter 6, *Using JDBC data sources*
- Chapter 7, *Using JMS connection factories*
- Chapter 8, *About Java connector architecture*
- Chapter 9, *Writing a Camel application that uses transactions*

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. INTRODUCTION TO TRANSACTIONS

This chapter introduces transactions by discussing some basic transaction concepts as well as the service qualities that are important in a transaction manager. The information is organized as follows:

- Section 1.1, “What is a transaction?”
- Section 1.2, “ACID properties of a transaction”
- Section 1.3, “About transaction clients”
- Section 1.4, “Descriptions of transaction terms”
- Section 1.5, “Managing transactions that modify multiple resources”
- Section 1.6, “Relationship between transactions and threads”
- Section 1.7, “About transaction service qualities”

1.1. WHAT IS A TRANSACTION?

The prototype of a transaction is an operation that conceptually consists of a single step (for example, transfer money from account A to account B), but must be implemented as a series of steps. Such operations are vulnerable to system failures because a failure is likely to leave some of the steps unfinished, which leaves the system in an inconsistent state. For example, consider the operation of transferring money from account A to account B. Suppose that the system fails after debiting account A, but before crediting account B. The result is that some money disappears.

To ensure that an operation like this is reliable, implement it as a *transaction*. A transaction guarantees reliable execution because it is atomic, consistent, isolated, and durable. These properties are referred to as a transaction’s ACID properties.

1.2. ACID PROPERTIES OF A TRANSACTION

The *ACID* properties of a transaction are defined as follows:

- **Atomic**—a transaction is an all or nothing procedure. Individual updates are assembled and either committed or aborted (rolled back) simultaneously when the transaction completes.
- **Consistent**—a transaction is a unit of work that takes a system from one consistent state to another consistent state.
- **Isolated**—while a transaction is executing, its partial results are hidden from other entities.
- **Durable**—the results of a transaction are persistent even if the system fails immediately after a transaction has been committed.

1.3. ABOUT TRANSACTION CLIENTS

A *transaction client* is an API or object that enables you to initiate and end transactions. Typically, a transaction client exposes operations that **begin**, **commit**, or **roll back** a transaction.

In a standard JavaEE application, the **javax.transaction.UserTransaction** interface exposes the transaction client API. In the context of the Spring Framework, Spring Boot, the **org.springframework.transaction.PlatformTransactionManager** interface exposes a transaction

client API.

1.4. DESCRIPTIONS OF TRANSACTION TERMS

The following table defines some important transaction terms:

Term	Description
Demarcation	Transaction demarcation refers to starting and ending transactions. Ending transactions means that the work done in the transaction is either committed or rolled back. Demarcation can be explicit, for example, by calling a transaction client API, or implicit, for example, whenever a message is polled from a transactional endpoint. For details, see Chapter 9, *Writing a Camel application that uses transactions*
Resources	A *resource* is any component of a computer system that can undergo a persistent or permanent change. In practice, a resource is almost always a database or a service layered over a database, for example, a message service with persistence. Other kinds of resource are conceivable, however. For example, an Automated Teller Machine (ATM) is a kind of resource. After a customer has physically accepted cash from the machine, the transaction cannot be reversed.
Transaction manager	A *transaction manager* is responsible for coordinating transactions across one or more resources. In many cases, a transaction manager is built into a resource. For example, enterprise-level databases typically include a transaction manager that is capable of managing transactions that change content in that database. Transactions that involve **more** than one resource usually require an **external** transaction manager.
Transaction context	A *transaction context* is an object that encapsulates the information needed to keep track of a transaction. The format of a transaction context depends entirely on the relevant transaction manager implementation. At a minimum, the transaction context contains a unique transaction identifier.
Distributed transactions	A distributed transaction refers to a transaction in a distributed system, where the transaction scope spans multiple network nodes. A basic prerequisite for supporting distributed transactions is a network protocol that supports transmission of transaction contexts in a canonical format. **Distributed transactions are outside the scope of Apache Camel transactions.** See also: Section 3.2.3, “About distributed transaction managers”.
X/Open XA standard	The X/Open XA standard describes an interface for integrating resources with a transaction manager. To manage a transaction that includes more than one resource, participating resources must support the XA standard. Resources that support the XA standard expose a special object, the *XA switch*, which enables transaction managers (or transaction processing monitors) to take control of the resource’s transactions. The XA standard supports both the 1-phase commit protocol and the 2-phase commit protocol.

1.5. MANAGING TRANSACTIONS THAT MODIFY MULTIPLE RESOURCES

For transactions that involve a **single** resource, the transaction manager built into the resource can usually be used. For transactions that involve **multiple** resources, it is necessary to use an external transaction manager or a transaction processing (TP) monitor. In this case, the resources must be integrated with the transaction manager by registering their XA switches.

There is an important difference between the protocol that is used to commit a transaction that operates on a single-resource system and the protocol that is used to commit a transaction that operates on a multiple-resource systems:

- **1-phase commit**—is for single-resource systems. This protocol commits a transaction in a single step.

- **2-phase commit**—is for multiple-resource systems. This protocol commits a transaction in two steps.

Including multiple resources in a transaction adds the risk that a system failure might occur after committing the transaction on some, but not all, of the resources. This would leave the system in an inconsistent state. The 2-phase commit protocol is designed to eliminate this risk. It ensures that the system can **always** be restored to a consistent state after it is restarted.

1.6. RELATIONSHIP BETWEEN TRANSACTIONS AND THREADS

To understand transaction processing, it is crucial to appreciate the basic relationship between transactions and threads: **transactions are thread-specific.** That is, when a transaction is started, it is attached to a specific thread. (Technically, a *transaction context* object is created and associated with the current thread). From this point until the transaction ends, all of the activity in the thread occurs within this transaction scope. Activity in any other thread does **not** fall within this transaction's scope. However, activity in any other thread can fall within the scope of some other transaction.

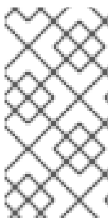
This relationship between transactions and thread means:

- **An application can process multiple transactions simultaneously**as long as each transaction is created in a separate thread.

- **Beware of creating subthreads within a transaction** If you are in the middle of a transaction and you create a new pool of threads, for example, by calling the **threads()** Camel DSL command, the new threads are **not** in the scope of the original transaction.

- **Beware of processing steps that implicitly create new threads**for the same reason given in the preceding point.

- **Transaction scopes do not usually extend across route segments**That is, if one route segment ends with **to(*JoinEndpoint*)** and another route segment starts with **from(*JoinEndpoint*)**, these route segments typically do **not** belong to the same transaction. There are exceptions, however.



NOTE

Some advanced transaction manager implementations give you the freedom to detach and attach transaction contexts to and from threads at will. For example, this makes it possible to move a transaction context from one thread to another thread. In some cases, it is also possible to attach a single transaction context to multiple threads.

1.7. ABOUT TRANSACTION SERVICE QUALITIES

When it comes to choosing the products that implement your transaction system, there is a great variety of database products and transaction managers available, some free of charge and some commercial. All of them have nominal support for transaction processing, but there are considerable variations in the qualities of service supported by these products. This section provides a brief guide to the kind of features that you need to consider when comparing the reliability and sophistication of different transaction products.

1.7.1. Qualities of service provided by resources

The following features determine the quality of service of a resource:

- Section 1.7.1.1, “Transaction isolation levels”
- Section 1.7.1.2, “Support for the XA standard”

1.7.1.1. Transaction isolation levels

ANSI SQL defines four *transaction isolation levels*, as follows:

SERIALIZABLE

Transactions are perfectly isolated from each other. That is, nothing that one transaction does can affect any other transaction until the transaction is committed. This isolation level is described as **serializable**, because the effect is as if all transactions were executed one after the other (although in practice, the resource can often optimize the algorithm, so that some transactions are allowed to proceed simultaneously).

REPEATABLE_READ

Every time a transaction reads or updates the database, a read or write lock is obtained and held until the end of the transaction. This provides almost perfect isolation. But there is one case where isolation is not perfect. Consider a SQL **SELECT** statement that reads a range of rows by using a **WHERE** clause. If another transaction adds a row to this range while the first transaction is running, the first transaction can see this new row, if it repeats the **SELECT** call (a *phantom read*).

READ_COMMITTED

Write locks are held until the end of a transaction. Read locks are **not** held until the end of a transaction. Consequently, repeated reads can give different results because updates committed by other transactions become visible to an ongoing transaction.

READ_UNCOMMITTED

Neither read locks nor write locks are held until the end of a transaction. Hence, dirty reads are possible. A dirty ready is when uncommitted changes made by other transactions are visible to an ongoing transaction.

Databases generally do not support all of the different transaction isolation levels. For example, some free databases support only **READ_UNCOMMITTED**. Also, some databases implement transaction isolation levels in ways that are subtly different from the ANSI standard. Isolation is a complicated issue that involves trade offs with database performance (for example, see Isolation in Wikipedia).

1.7.1.2. Support for the XA standard

For a resource to participate in a transaction that involves multiple resources, it needs to support the X/Open XA standard. Be sure to check whether the resource’s implementation of the XA standard is subject to any special restrictions. For example, some implementations of the XA standard are restricted

to a single database connection, which implies that only one thread at a time can process a transaction that involves that resource.

1.7.2. Qualities of service provided by transaction managers

The following features determine the quality of service of a transaction manager:

- Section 1.7.2.1, “Support for suspend/resume and attach/detach” .
- Section 1.7.2.2, “Support for multiple resources”.
- Section 1.7.2.3, “Distributed transactions”.
- Section 1.7.2.4, “Transaction monitoring”.
- Section 1.7.2.5, “Recovery from failure” .

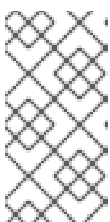
1.7.2.1. Support for suspend/resume and attach/detach

Some transaction managers support advanced capabilities for manipulating the associations between a transaction context and application threads, as follows:

- **Suspend/resume current transaction**—enables you to suspend temporarily the current transaction context, while the application does some non-transactional work in the current thread.
- **Attach/detach transaction context**—enables you to move a transaction context from one thread to another or to extend a transaction scope to include multiple threads.

1.7.2.2. Support for multiple resources

A key differentiator for transaction managers is the ability to support multiple resources. This normally entails support for the XA standard, where the transaction manager provides a way for resources to register their XA switches.



NOTE

Strictly speaking, the XA standard is not the only approach you can use to support multiple resources, but it is the most practical one. The alternative typically involves writing tedious (and critical) custom code to implement the algorithms normally provided by an XA switch.

1.7.2.3. Distributed transactions

Some transaction managers have the capability to manage transactions whose scope includes multiple nodes in a distributed system. The transaction context is propagated from node to node by using special protocols such as WS-AtomicTransactions or CORBA OTS.

1.7.2.4. Transaction monitoring

Advanced transaction managers typically provide visual tools to monitor the status of pending transactions. This kind of tool is particularly useful after a system failure, where it can help to identify and resolve transactions that were left in an uncertain state (heuristic exceptions).

1.7.2.5. Recovery from failure

There are significant differences among transaction managers with respect to their robustness in the event of a system failure (crash). The key strategy that transaction managers use is to write data to a persistent log before performing each step of a transaction. In the event of a failure, the data in the log can be used to recover the transaction. Some transaction managers implement this strategy more carefully than others. For example, a high-end transaction manager would typically duplicate the persistent transaction log and allow each of the logs to be stored on separate host machines.

CHAPTER 2. GETTING STARTED WITH TRANSACTIONS ON KARAF (OSGI)

This section describes a Camel application that uses transactions to access an Artemis JMS broker. The information is organized as follows:

- Section 2.1, "Prerequisites"
- Section 2.2, "Building the camel-jms project"
- Section 2.3, "Explanation of the camel-jms project"

2.1. PREREQUISITES

Implementation of this Camel application has the following prerequisites:

- An external AMQ 7 JMS message broker must be running.
  The following sample code runs a standalone (non-Docker) version of **amq-broker-7.1.0-bin.zip**. Execution creates and runs an **amq7** instance:

```
$ pwd
/data/servers/amq-broker-7.1.0

$ bin/artemis create --user admin --password admin --require-login amq7
Creating ActiveMQ Artemis instance at: /data/servers/amq-broker-7.1.0/amq7

Auto tuning journal ...
done! Your system can make 27.78 writes per millisecond, your journal-buffer-timeout will be
36000

You can now start the broker by executing:

   "/data/servers/amq-broker-7.1.0/amq7/bin/artemis" run

Or you can run the broker in the background using:

   "/data/servers/amq-broker-7.1.0/amq7/bin/artemis-service" start

$ amq7/bin/artemis run
           __  __  ____    ____            _
     /\   |  \/  |/ __ \  |  _ \          | |
    /  \  | \  / | |  | | | |_) |_ __ ___ | | _____ _ __
   / /\ \ | |\/| | |  | | |  _ <| '__/ _ \| |/ / _ \ '__|
  / ____ \| |  | | |__| | | |_) | | | (_) |   <  __/ |
 /_/    \_\_|  |_|\___\_\ |____/|_|  \___/|_|\_\___|_|

 Red Hat JBoss AMQ 7.1.0.GA


018-05-02 16:37:19,294 INFO  [org.apache.activemq.artemis.integration.bootstrap]
AMQ101000: Starting ActiveMQ Artemis Server
...
```

- Client libraries are required. Artemis libraries are available in Maven Central or a Red Hat repository. For example, you can use:
  - **mvn:org.apache.activemq/artemis-core-client/2.4.0.amq-710008-redhat-1**
  - **mvn:org.apache.activemq/artemis-jms-client/2.4.0.amq-710008-redhat-1**

  Alternatively, Artemis/AMQ 7 client libraries can be installed as Karaf features, for example:
  - **karaf@root()> feature:install artemis-jms-client artemis-core-client**
- Some supporting features that provide Karaf shell commands or dedicated Artemis support are required:

  ```
  karaf@root()> feature:install jms pax-jms-artemis pax-jms-config
  ```
- Required Camel features are:

  ```
  karaf@root()> feature:install camel-jms camel-blueprint
  ```

2.2. BUILDING THE CAMEL-JMS PROJECT

You can download the **quickstarts** from the Fuse Software Downloads page.

Extract the contents of the zip file to a local folder, for example a new folder named **quickstarts**.

You can then build and install the **/camel/camel-jms** example as an OSGi bundle. This bundle contains a Blueprint XML definition of a Camel route that sends messages to an AMQ 7 JMS queue.

In the following example, **$FUSE_HOME** is the location of the unzipped Fuse distribution. To build this project:

1. Invoke Maven to build the project:

   ```
   $ cd quickstarts

   $ mvn clean install -f camel/camel-jms/
   ```
2. Create a JMS connection factory configuration so that the **javax.jms.ConnectionFactory** service is published in the OSGi runtime. To do this, copy **quickstarts/camel/camel-jms/src/main/resources/etc/org.ops4j.connectionfactory-amq7.cfg** into the **$FUSE_HOME/etc** directory. This configuration will be processed to create a working connection factory. For example:

   ```
   $ cp camel/camel-jms/src/main/resources/etc/org.ops4j.connectionfactory-amq7.cfg ../etc/
   ```
3. Verify the published connection factory:

   ```
   karaf@root()> service:list javax.jms.ConnectionFactory
   [javax.jms.ConnectionFactory]
   ----------------------------
    felix.fileinstall.filename = file:$FUSE_HOME/etc/org.ops4j.connectionfactory-amq7.cfg
    name = artemis
    osgi.jndi.service.name = artemis
    password = admin
   ```

```
 pax.jms.managed = true
 service.bundleid = 251
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 436
 service.pid = org.ops4j.connectionfactory.d6207fcc-3fe6-4dc1-a0d8-0e76ba3b89bf
 service.scope = singleton
 type = artemis
 url = tcp://localhost:61616
 user = admin
Provided by :
 OPS4J Pax JMS Config (251)

karaf@root()> jms:info -u admin -p admin artemis
Property │ Value
─────────┼──────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:queues -u admin -p admin artemis
JMS Queues
────────────────────────────────────
df2501d1-aa52-4439-b9e4-c0840c568df1
DLQ
ExpiryQueue
```

4. Install the bundle:

```
karaf@root()> install -s mvn:org.jboss.fuse.quickstarts/camel-jms/7.0.0.redhat-SNAPSHOT
Bundle ID: 256
```

5. Confirm that it is working:

```
karaf@root()> camel:context-list
 Context              Status       Total #    Failed #    Inflight #   Uptime
 -------              ------       -------    --------    ----------   ------
 jms-example-context  Started            0           0             0   2 minutes
karaf@root()> camel:route-list
 Context              Route              Status       Total #    Failed #    Inflight #   Uptime
 -------              -----              ------       -------    --------    ----------   ------
 jms-example-context  file-to-jms-route  Started            0           0             0   2 minutes
 jms-example-context  jms-cbr-route      Started            0           0             0   2 minutes
```

6. As soon as the Camel routes have started, you can see a directory, **work/jms/input**, in your Fuse installation. Copy the files you find in this quickstart’s **src/main/data** directory to the newly created **work/jms/input** directory.

7. Wait a few moments and you will find the same files organized by country under the **work/jms/output** directory:

   - **order1.xml**, **order2.xml** and **order4.xml** in **work/jms/output/others**
   - **order3.xml** and **order5.xml** in **work/jms/output/us**
   - **order6.xml** in **work/jms/output/fr**

8. See the logs to check out the business logging:

```
2018-05-02 17:20:47,952 | INFO  | ile://work/jms/input | file-to-jms-route                | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Receiving order order1.xml
2018-05-02 17:20:48,052 | INFO  | umer[incomingOrders] | jms-cbr-route                    | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Sending order order1.xml to another
country
2018-05-02 17:20:48,053 | INFO  | umer[incomingOrders] | jms-cbr-route                    | 58 -
org.apache.camel.camel-core - 2.21.0.fuse-000077 | Done processing order1.xml
```

9. See that the queue was dynamically created:

```
karaf@root()> jms:queues -u admin -p admin artemis
JMS Queues
────────────────────────────────────
DLQ
17767323-937f-4bad-a403-07cd63311f4e
ExpiryQueue
incomingOrders
```

10. Check Camel route statistics:

```
karaf@root()> camel:route-info jms-example-context file-to-jms-route
Camel Route file-to-jms-route
 Camel Context: jms-example-context
 State: Started
 State: Started


Statistics
 Exchanges Total: 1
 Exchanges Completed: 1
 Exchanges Failed: 0
 Exchanges Inflight: 0
 Min Processing Time: 67 ms
 Max Processing Time: 67 ms
 Mean Processing Time: 67 ms
 Total Processing Time: 67 ms
 Last Processing Time: 67 ms
 Delta Processing Time: 67 ms
 Start Statistics Date: 2018-05-02 17:14:17
 Reset Statistics Date: 2018-05-02 17:14:17
 First Exchange Date: 2018-05-02 17:20:48
 Last Exchange Date: 2018-05-02 17:20:48
```

2.3. EXPLANATION OF THE CAMEL-JMS PROJECT

Camel routes are using the following endpoint URIs:

```
<route id="file-to-jms-route">
...
    <to uri="jms:queue:incomingOrders?transacted=true" />
</route>

<route id="jms-cbr-route">
```

```
    <from uri="jms:queue:incomingOrders?transacted=true" />
...
</route>
```

The **jms** component is configured by using this snippet:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <reference interface="javax.jms.ConnectionFactory" />
    </property>
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

While the **transactionManager** reference is:

```
<reference id="transactionManager"
interface="org.springframework.transaction.PlatformTransactionManager" />
```

As you can see, both the JMS connection factory and the Spring interface of **PlatformTransactionManager** are only references. There is no need to *define* them in Blueprint XML. These *services* are exposed by Fuse itself.

You have already seen that **javax.jms.ConnectionFactory** was created by using **etc/org.ops4j.connectionfactory-amq7.cfg**.

The transaction manager is:

```
karaf@root()> service:list org.springframework.transaction.PlatformTransactionManager
[org.springframework.transaction.PlatformTransactionManager]
----------------------------------------------------------
 service.bundleid = 21
 service.id = 527
 service.scope = singleton
Provided by :
 Red Hat Fuse :: Fuse Modules :: Transaction (21)
Used by:
 Red Hat Fuse :: Quickstarts :: camel-jms (256)
```

Check for other interfaces under which the actual transaction manager is registered:

```
karaf@root()> headers 21

Red Hat Fuse :: Fuse Modules :: Transaction (21)
----------------------------------------------
...
Bundle-Name = Red Hat Fuse :: Fuse Modules :: Transaction
Bundle-SymbolicName = fuse-pax-transx-tm-narayana
Bundle-Vendor = Red Hat
...

karaf@root()> bundle:services -p 21

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
--------------------------------------------------------
objectClass = [org.osgi.service.cm.ManagedService]
```

```
service.bundleid = 21
service.id = 519
service.pid = org.ops4j.pax.transx.tm.narayana
service.scope = singleton
 ----
objectClass = [javax.transaction.TransactionManager]
provider = narayana
service.bundleid = 21
service.id = 520
service.scope = singleton
 ----
objectClass = [javax.transaction.TransactionSynchronizationRegistry]
provider = narayana
service.bundleid = 21
service.id = 523
service.scope = singleton
 ----
objectClass = [javax.transaction.UserTransaction]
provider = narayana
service.bundleid = 21
service.id = 524
service.scope = singleton
 ----
objectClass = [org.jboss.narayana.osgi.jta.ObjStoreBrowserService]
provider = narayana
service.bundleid = 21
service.id = 525
service.scope = singleton
 ----
objectClass = [org.ops4j.pax.transx.tm.TransactionManager]
provider = narayana
service.bundleid = 21
service.id = 526
service.scope = singleton
 ----
objectClass = [org.springframework.transaction.PlatformTransactionManager]
service.bundleid = 21
service.id = 527
service.scope = singleton
```

The transaction manager is available from these interfaces:

- **javax.transaction.TransactionManager**
- **javax.transaction.TransactionSynchronizationRegistry**
- **javax.transaction.UserTransaction**
- **org.jboss.narayana.osgi.jta.ObjStoreBrowserService**
- **org.ops4j.pax.transx.tm.TransactionManager**
- **org.springframework.transaction.PlatformTransactionManager**

You can use any of them in any context that you need. For example **camel-jms** requires that the **org.apache.camel.component.jms.JmsConfiguration.transactionManager** field be initialized. This is why the example uses:

```
<reference id="transactionManager"
interface="org.springframework.transaction.PlatformTransactionManager" />
```

instead of, for example:

```
<reference id="transactionManager" interface="javax.transaction.TransactionManager" />
```

CHAPTER 3. INTERFACES FOR CONFIGURING AND REFERENCING TRANSACTION MANAGERS

JavaEE and Spring Boot each provide a transaction client interface for configuring the transaction manager in Fuse and for using the transaction manager in deployed applications. There is a clear distinction between configuration, which is an administrative task, and referencing, which is a development task. The application developer is responsible for pointing the application to a previously configured transaction manager.

- Section 3.1, "What transaction managers do"
- Section 3.2, "About local, global, and distributed transaction managers"
- Section 3.3, "Using a JavaEE transaction client"
- Section 3.4, "Using a Spring Boot transaction client"
- Section 3.5, "OSGi interfaces between transaction clients and the transaction manager"

3.1. WHAT TRANSACTION MANAGERS DO

A transaction manager is the part of an application that is responsible for coordinating transactions across one or more resources. The responsibilities of the transaction manager are as follows:

- Demarcation – starting and ending transactions by using begin, commit, and rollback methods.
- Managing the transaction context – a transaction context contains the information that a transaction manager needs to keep track of a transaction. The transaction manager is responsible for creating transaction contexts and attaching them to the current thread.
- Coordinating the transaction across multiple resources – enterprise-level transaction managers typically have the capability to coordinate a transaction across multiple resources. This feature requires the 2-phase commit protocol and resources must be registered and managed using the XA protocol. See Section 1.7.1.2, "Support for the XA standard".
  This is an advanced feature that is not supported by all transaction managers.
- Recovery from failure – transaction managers are responsible for ensuring that resources are not left in an inconsistent state if there is a system failure and the application fails. In some cases, manual intervention might be required to restore the system to a consistent state.

3.2. ABOUT LOCAL, GLOBAL, AND DISTRIBUTED TRANSACTION MANAGERS

A transaction manager can be local, global, or distributed.

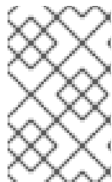
3.2.1. About local transaction managers

A *local transaction manager* is a transaction manager that can coordinate transactions for only a single resource. The implementation of a local transaction manager is typically embedded in the resource itself and the transaction manager used by application is a thin wrapper around this built-in transaction manager.

For example, the Oracle database has a built-in transaction manager that supports demarcation operations (by using SQL **BEGIN**, **COMMIT**, or **ROLLBACK** statements or by using a native Oracle API)

and various levels of transaction isolation. Control over the Oracle transaction manager can be exported through JDBC, and this JDBC API is used by applications to demarcate transactions.

It is important to understand what constitutes a resource, in this context. For example, if you are using a JMS product, the JMS resource is the single running instance of the JMS product, not the individual queues and topics. Moreover, sometimes, what appears to be multiple resources might actually be a single resource, if the same underlying resource is accessed in different ways. For example, your application might access a relational database both directly (through JDBC) and indirectly (through an object-relational mapping tool like Hibernate). In this case, the same underlying transaction manager is involved, so it should be possible to enroll both of these code fragments in the same transaction.



NOTE

It cannot be guaranteed that this will work in every case. Although it is possible in principle, some detail in the design of the Spring Framework or other wrapper layers might prevent it from working in practice.

It is possible for an application to have many different local transaction managers working independently of each other. For example, you could have one Camel route that manipulates JMS queues and topics, where the JMS endpoints reference a JMS transaction manager. Another route could access a relational database through JDBC. But you could not combine JDBC and JMS access in the same route and have them both participate in the same transaction.

3.2.2. About global transaction managers

A global transaction manager is a transaction manager that can coordinate transactions over multiple resources. This is required when you cannot rely on the transaction manager built into the resource itself. An external system, sometimes called a transaction processing monitor (TP monitor), is capable of coordinating transactions across different resources.

The following are the prerequisites for transactions that operate on multiple resources:

- Global transaction manager or TP monitor – an external transaction system that implements the 2-phase commit protocol for coordinating multiple XA resources.
- Resources that support the *XA standard* – to participate in a 2-phase commit, resources must support the XA standard. See Section 1.7.1.2, “Support for the XA standard”. In practice, this means that the resource is capable of exporting an XA *switch* object, which gives complete control of transactions to the external TP monitor.

TIP

The Spring Framework does not by itself provide a TP monitor to manage global transactions. It does, however, provide support for integrating with an OSGi-provided TP monitor or with a JavaEE-provided TP monitor (where the integration is implemented by the JtaTransactionManager class). Hence, if you deploy your application into an OSGi container with full transaction support, you can use multiple transactional resources in Spring.

3.2.3. About distributed transaction managers

Usually, a server connects directly to the resources involved in a transaction. In a distributed system, however, it is occasionally necessary to connect to resources that are exposed only indirectly, through a Web service. In this case, you require a TP monitor that is capable of supporting distributed transactions. Several standards are available that describe how to support transactions for various distributed protocols, for example, the WS-AtomicTransactions specification for Web services.

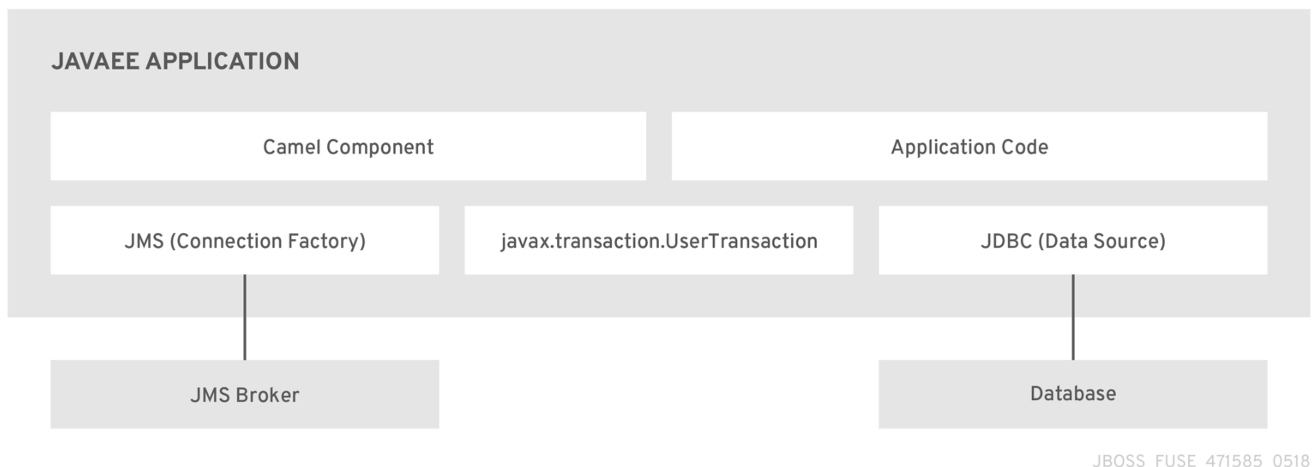
3.3. USING A JAVAEE TRANSACTION CLIENT

When using JavaEE, the most fundamantal and standard method to interact with a transaction manager is the Java Transaction API (JTA) interface, **javax.transaction.UserTransaction**. The canonical usage is:

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction) context.lookup("java:comp/UserTransaction");
ut.begin();

// Access transactional, JTA-aware resources such as database and/or message broker

ut.commit(); // or ut.rollback()
```

Obtaining a **UserTransaction** instance from JNDI (Java Naming and Directory Interface) is one way of getting a transaction client. In a JavaEE environment, you can access a transaction client, for example, with CDI (context and dependency injection).

The following figure shows a typica JavaEE Camel application.



The figure shows that both Camel code and application code may access:

- A **javax.transaction.UserTransaction** instance to demarcate transactions either directly from an application or internally through transaction-aware Camel components by using the Spring **TransactionTemplate** class.
- Databases through JDBC APIs either directly or, for example, by using Spring's **JdbcTemplate**, or by using the **camel-jdbc** component.
- Message brokers through a JMS API either directly, by using Spring's **JmsTemplate** class or by using the **camel-jms** component.

When using a **javax.transaction.UserTransaction** object, you do not need to be aware of the actual transaction manager that is being used because you are working directly with only the transaction client. (See Section 1.3, "About transaction clients".) A different approach is taken by Spring and Camel, as it uses Spring's transaction facilities internally.

JavaEE Application

In typical JavaEE scenario, the application is deployed to a JavaEE application server, usually as a **WAR** or **EAR** archive. By means of JNDI or CDI, the application may access an instance of the

javax.transaction.UserTransaction service. The aplication then uses this transaction client instance to demarcate transactions. Within a transaction, the application performs JDBC and/or JMS access.

Camel component and application code

These represent the code that performs JMS/JDBC operations. Camel has its own advanced methods to access JMS/JDBC resources. The application code may use a given API directly.

JMS Connection Factory

This is the **javax.jms.ConnectionFactory** interface that is used to obtain instances of **javax.jms.Connection** and then **javax.jms.Session** (or **javax.jms.JmsContext** in JMS 2.0). This may be used directly by the application or indirectly in Camel components, which may use **org.springframework.jms.core.JmsTemplate** internally. Neither application code nor a Camel component require the details of this connection factory. The connection factory is configured at the application server. You can see this configuration in a JavaEE server. An OSGi server such as Fuse is similar. A system administrator configures the connection factory independently of the application. Typically, the connection factory implements pooling capabilities.

JDBC Data Source

This is the **javax.sql.DataSource** interface that is used to obtain instances of **java.sql.Connection**. As with JMS, this data source may be used directly or indirectly. For example, the **camel-sql** component uses the **org.springframework.jdbc.core.JdbcTemplate** class internally. As with JMS, neither application code nor Camel require the details of this data source. The configuration is done inside the application server or inside the OSGi server by using methods that are described in Chapter 4, *Configuring the Narayana transaction manager*.

3.4. USING A SPRING BOOT TRANSACTION CLIENT

One of the main goals of the Spring Framework (and Spring Boot) is to make JavaEE APIs easier to use. All major JavaEE *vanilla* APIs have their part in the Spring Framework (Spring Boot). These are not **alternatives** or **replacements** of given APIs, but rather wrappers that add more configuration options or more consistent usage, for example, with respect to exception handling.

The following table matches a given JavaEE API with its Spring-related interface:

JavaEE API	Spring Utility	Configured With
JDBC	**org.springframework.jdbc.core.JdbcTemplate**	**javax.sql.DataSource**
JMS	**org.springframework.jms.core.JmsTemplate**	**javax.jms.ConnectionFactory**
JTA	**org.springframework.transaction.support.TransactionTemplate**	**org.springframework.transaction.PlatformTransactionManager**

JdbcTemplate and **JmsTemplate** directly use **javax.sql.DataSource** and **javax.jms.ConnectionFactory** respectively. But **TransactionTemplate** uses the Spring interface of **PlatformTransactionManager**. This is where Spring does not simply **improve** JavaEE, but replaces the JavaEE client API with its own.

Spring treats **javax.transaction.UserTransaction** as an interface that is too simple for real-world

scenarios. Also, because **javax.transaction.UserTransaction** does not distinguish between local, single resource transactions and global, multi-resource transactions, implementations of **org.springframework.transaction.PlatformTransactionManager** give developers more freedom.

Following is the canonical API usage of Spring Boot:

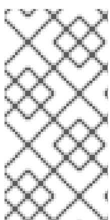
```
// Create or get from ApplicationContext or injected with @Inject/@Autowired.
JmsTemplate jms = new JmsTemplate(...);
JdbcTemplate jdbc = new JdbcTemplate(...);
TransactionTemplate tx = new TransactionTemplate(...);

tx.execute((status) -> {
    // Perform JMS operations within transaction.
    jms.execute((SessionCallback<Object>)(session) -> {
        // Perform operations on JMS session
        return ...;
    });
    // Perform JDBC operations within transaction.
    jdbc.execute((ConnectionCallback<Object>)(connection) -> {
        // Perform operations on JDBC connection.
        return ...;
    });
    return ...;
});
```

In the above example, all three kinds of *templates* are simply instantiated, but they may also be obtained from Spring's **ApplicationContext**, or injected by using **@Autowired** annotations.

3.4.1. Using the Spring PlatformTransactionManager interface

As mentioned earlier, **javax.transaction.UserTransaction** is usually obtained from JNDI in a JavaEE application. But Spring provides explicit implementations of this interface for many scenarios. You do not always need full JTA scenarios and sometimes an application requires access to just a single resource, for example, JDBC.

Usually, **org.springframework.transaction.PlatformTransactionManager** is **the** Spring transaction client API that provides the classic transaction client operations: **begin**, **commit** and **rollback**. In other words, this interface provides the essential methods for controlling transactions at runtime.



NOTE

The other key aspect of any transaction system is the API for implementing transactional resources. But transactional resources are usually implemented by the underlying database, so this aspect of transactional programming is rarely a concern for the application programmer.

3.4.1.1. Definition of the PlatformTransactionManager interface

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

```

```
    void rollback(TransactionStatus status) throws TransactionException;
}
```

3.4.1.2. About the TransactionDefinition interface

You use the **TransactionDefinition** interface to specify the characteristics of a newly created transaction. You can specify the isolation level and the propagation policy of the new transaction. For details, see Section 9.4, "Transaction propagation policies".

3.4.1.3. Definition of the TransactionStatus interface

You can use the **TransactionStatus** interface to check the status of the current transaction, that is, the transaction that is associated with the current thread, and to mark the current transaction for rollback. This is the interface definition:

```
public interface TransactionStatus extends SavepointManager, Flushable {

    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();
}
```

3.4.1.4. Methods defined by the PlatformTransactionManager interface

The **PlatformTransactionManager** interface defines the following methods:

getTransaction()

Creates a new transaction and associates it with the current thread by passing in a **TransactionDefinition** object that defines the characteristics of the new transaction. This is analogous to the **begin()** method of many other transaction client APIs.

commit()

Commits the current transaction, which makes all of the pending changes to the registered resources permanent.

rollback()

Rolls back the current transaction, which undoes all pending changes to the registered resources.

3.4.2. Steps for using the transaction manager

Usually, you do not use the **PlatformTransactionManager** interface directly. In Apache Camel, you typically use a transaction manager as follows:

1. Create an instance of a transaction manager. There are several different implementations available in Spring, see Section 3.4, "Using a Spring Boot transaction client").

2. Pass the transaction manager instance to either an Apache Camel component or to the **transacted()** DSL command in a route. The transactional component or the **transacted()** command is then responsible for demarcating transactions. For details, see Chapter 9, *Writing a Camel application that uses transactions*).

3.4.3. About Spring PlatformTransactionManager implementations

This section provides a brief overview of the transaction manager implementations that are provided by the Spring Framework. The implementations fall into two categories: local transaction managers and global transaction managers.

Starting from Camel:

- The **org.apache.camel.component.jms.JmsConfiguration** object that is used by the **camel-jms** component requires an instance of the **org.springframework.transaction.PlatformTransactionManager** interface.
- The **org.apache.camel.component.sql.SqlComponent** uses the **org.springframework.jdbc.core.JdbcTemplate** class internally and this JDBC template also integrates with **org.springframework.transaction.PlatformTransactionManager**.

As you can see, you must have *some* implementation of this interface. Depending on the scenario, you can configure the required platform transaction manager.

3.4.3.1. Local PlatformTransactionManager implementations

The list below summarizes the local transaction manager implementations that are provided by the Spring Framework. These transaction managers support only a single resource.

org.springframework.jms.connection.JmsTransactionManager

This transaction manager implementation is capable of managing a *single* JMS resource. You can connect to any number of queues or topics, but only if they belong to the same underlying JMS messaging product instance. Moreover, you cannot enlist any other type of resource in a transaction.

org.springframework.jdbc.datasource.DataSourceTransactionManager

This transaction manager implementation is capable of managing a *single* JDBC database resource. You can update any number of different database tables, but *only* if they belong to the same underlying database instance.

org.springframework.orm.jpa.JpaTransactionManager

This transaction manager implementation is capable of managing a Java Persistence API (JPA) resource. It is not possible, however, to simultaneously enlist any other kind of resource in a transaction.

org.springframework.orm.hibernate5.HibernateTransactionManager

This transaction manager implementation is capable of managing a Hibernate resource. It is not possible, however, to simultaneously enlist any other kind of resource in a transaction. Moreover, the JPA API is preferred over the native Hibernate API.

There are also other, less frequently used, implementations of **PlatformTransactionManager**.

3.4.3.2. Global PlatformTransactionManager implementation

The Spring Framework provides one global transaction manager implementation for use in the OSGi runtime. The **org.springframework.transaction.jta.JtaTransactionManager** supports operations on multiple resources in a transaction. This transaction manager supports the XA transaction API and can

enlist more than one resource in a transaction. To use this transaction manager, you must deploy your application inside either an OSGi container or a JavaEE server.

While single-resource implementations of **PlatformTransactionManager** are actual *implementations*, **JtaTransactionManager** is more of a wrapper for an actual implementation of the standard **javax.transaction.TransactionManager**.

This is why it is better to use the **JtaTransactionManager** implementation of **PlatformTransactionManager** in an environment where you can access (by means of JNDI or CDI) an already configured instance of **javax.transaction.TransactionManager** and usually also **javax.transaction.UserTransaction**. Usually, both these JTA interfaces are implemented by a single object/service.

Here is an example of configuring/using **JtaTransactionManager**:

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction) context.lookup("java:comp/UserTransaction");
TransactionManager tm = (TransactionManager) context.lookup("java:/TransactionManager");

JtaTransactionManager jta = new JtaTransactionManager();
jta.setUserTransaction(ut);
jta.setTransactionManager(tm);

TransactionTemplate jtaTx = new TransactionTemplate(jta);

jtaTx.execute((status) -> {
    // Perform resource access in the context of global transaction.
    return ...;
});
```

In the above example, the actual instances of JTA objects (**UserTransaction** and **TransactionManager**) are taken from JNDI. In OSGi, they may as well be obtained from the OSGi service registry.

3.5. OSGI INTERFACES BETWEEN TRANSACTION CLIENTS AND THE TRANSACTION MANAGER

After a description of the JavaEE transaction client API and the Spring Boot transaction client API, it is helpful to see the relationships within an OSGi server, such as Fuse. One of the features of OSGi is the global service registry, which may be used to:

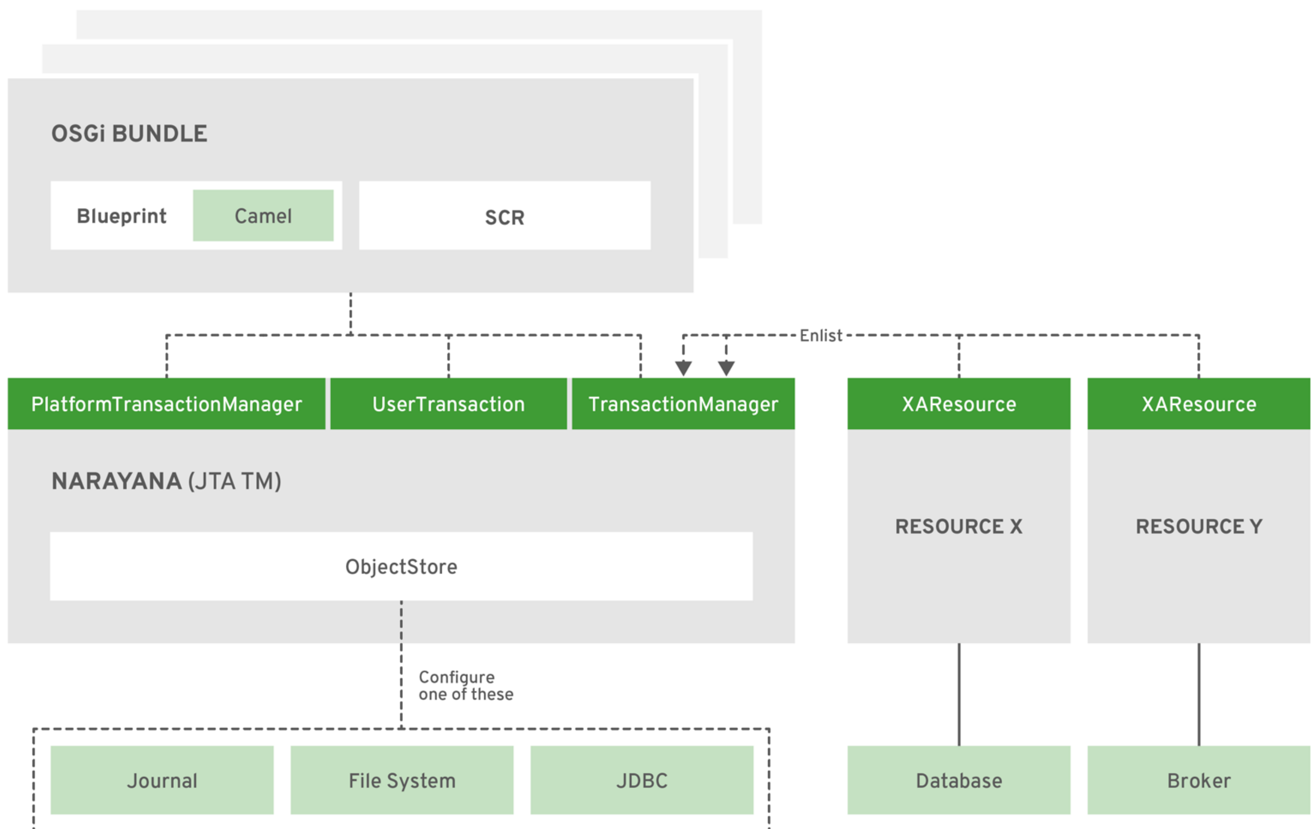
- Look up services by filter or interface(s).
- Register services with given interface(s) and properties.

In the same way that applications that are deployed in a JavaEE application server obtain references to **javax.transaction.UserTransaction** by using JNDI (*service locator* method) or get them injected by CDI (*dependency injection* method), in OSGi you can obtain the same references (directly or indirectly) in any of the following ways:

- Invoking the **org.osgi.framework.BundleContext.getServiceReference()** method (*service locator*).
- Get them injected in a Blueprint container.

- Use Service Component Runtime (SCR) annotations (*dependency injection*).

The following figure shows a Fuse application that is deployed in the OSGi runtime. Application code and/or Camel components use their APIs to obtain references to the transaction manager, data sources, and connection factories.



Applications (bundles) interact with services that are registered in the OSGi registry. The access is performed through *interfaces* and this is all that should be relevant to applications.

In Fuse, the fundamental object that implements (directly or through a tiny wrapper) transactional client interfaces is **org.jboss.narayana.osgi.jta.internal.OsgiTransactionManager**. You can use the following interfaces to access the transaction manager:

- **javax.transaction.TransactionManager**
- **javax.transaction.UserTransaction**
- **org.springframework.transaction.PlatformTransactionManager**
- **org.ops4j.pax.transx.tm.TransactionManager**

You can use any of these interfaces directly or you can use them implicitly by choosing a framework or library, such as Camel.

For information about the ways to configure **org.jboss.narayana.osgi.jta.internal.OsgiTransactionManager** in Fuse, see *Chapter 4, Configuring the Narayana transaction manager*. Later chapters in this guide build on the information in that chapter and describe how to configure and use other services, such as JDBC data sources and JMS connection factories.

CHAPTER 4. CONFIGURING THE NARAYANA TRANSACTION MANAGER

In Fuse, the built-in, global transaction manager is JBoss Narayana Transaction Manager, which is the same transaction manager that is used by Enterprise Application Platform (EAP) 7.

In the OSGi runtime, as in Fuse for Karaf, the additional integration layer is provided by the PAX TRANSX project.

The following topics discuss Narayana configuration:

- Section 4.1, "About Narayana installation"
- Section 4.2, "Transaction protocols supported"
- Section 4.3, "About Narayana configuration"
- Section 4.4, "Configuring log storage"

4.1. ABOUT NARAYANA INSTALLATION

The Narayana transaction manager is exposed for use in OSGi bundles under the following interfaces, as well as a few additional support interfaces:

- **javax.transaction.TransactionManager**
- **javax.transaction.UserTransaction**
- **org.springframework.transaction.PlatformTransactionManager**
- **org.ops4j.pax.transx.tm.TransactionManager**

The **7.9.0.fuse-790071-redhat-00001** distribution makes these interfaces available from the start.

The **pax-transx-tm-narayana** feature contains an overridden bundle that embeds Narayana:

```
karaf@root()> feature:info pax-transx-tm-narayana
Feature pax-transx-tm-narayana 0.3.0
Feature has no configuration
Feature has no configuration files
Feature depends on:
  pax-transx-tm-api 0.0.0
Feature contains followed bundles:
  mvn:org.jboss.fuse.modules/fuse-pax-transx-tm-narayana/7.0.0.fuse-000191-redhat-1 (overriden
from mvn:org.ops4j.pax.transx/pax-transx-tm-narayana/0.3.0)
Feature has no conditionals.
```

The services provided by the **fuse-pax-transx-tm-narayana** bundle are:

```
karaf@root()> bundle:services fuse-pax-transx-tm-narayana

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
----------------------------------------------------------
[org.osgi.service.cm.ManagedService]
[javax.transaction.TransactionManager]
```

```
[javax.transaction.TransactionSynchronizationRegistry]
[javax.transaction.UserTransaction]
[org.jboss.narayana.osgi.jta.ObjStoreBrowserService]
[org.ops4j.pax.transx.tm.TransactionManager]
[org.springframework.transaction.PlatformTransactionManager]
```

Because this bundle registers **org.osgi.service.cm.ManagedService**, it tracks and reacts to the changes in CM configurations:

```
karaf@root()> bundle:services -p fuse-pax-transx-tm-narayana

Red Hat Fuse :: Fuse Modules :: Transaction (21) provides:
---------------------------------------------------------
objectClass = [org.osgi.service.cm.ManagedService]
service.bundleid = 21
service.id = 232
service.pid = org.ops4j.pax.transx.tm.narayana
service.scope = singleton
...
```

The default **org.ops4j.pax.transx.tm.narayana** PID is:

```
karaf@root()> config:list '(service.pid=org.ops4j.pax.transx.tm.narayana)'
----------------------------------------------------------------
Pid:            org.ops4j.pax.transx.tm.narayana
BundleLocation: ?
Properties:
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.localOSRoot =
communicationStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreDir =
/data/servers/7.9.0.fuse-790071-redhat-00001/data/narayana

com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType
= com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.localOSRoot = defaultStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreDir =
/data/servers/7.9.0.fuse-790071-redhat-00001/data/narayana
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.localOSRoot = stateStore
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreDir =
/data/servers/7.9.0.fuse-790071-redhat-00001/data/narayana
   com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
   com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean.recoveryBackoffPeriod = 10
   felix.fileinstall.filename = file:/data/servers/7.9.0.fuse-790071-redhat-
00001/etc/org.ops4j.pax.transx.tm.narayana.cfg
   service.pid = org.ops4j.pax.transx.tm.narayana
```

In summary:

- Fuse for Karaf includes the fully-featured, global, Narayana transaction manager.
- The transaction manager is correctly exposed under various client interfaces (JTA, Spring-tx, PAX JMS).

- You can configure Narayana by using the standard OSGi method, Configuration Admin, which is available in **org.ops4j.pax.transx.tm.narayana**.
- The default configuration is provided in **$FUSE_HOME/etc/org.ops4j.pax.transx.tm.narayana.cfg**.

4.2. TRANSACTION PROTOCOLS SUPPORTED

The Narayana transaction manager is the JBoss/Red Hat product that is used in EAP. Narayana is a transactions toolkit that provides support for applications that are developed using a broad range of standards-based transaction protocols:

- JTA
- JTS
- Web-Service Transactions
- REST Transactions
- STM
- XATMI/TX

4.3. ABOUT NARAYANA CONFIGURATION

The **pax-transx-tm-narayana** bundle includes the **jbossts-properties.xml** file, which provides the default configuration of different aspects of the transaction manager. All of these properties may be overriden in the **$FUSE_HOME/etc/org.ops4j.pax.transx.tm.narayana.cfg** file directly or by using the Configuration Admin API.

The basic configuration of Narayana is done through various **EnvironmentBean** objects. Every such bean may be configured by using properties with different prefixes. The following table provides a summary of configuration objects and prefixes used:

Configuration Bean	Property Prefix
com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean	**com.arjuna.ats.arjuna.coordinator**
com.arjuna.ats.arjuna.common.CoreEnvironmentBean	**com.arjuna.ats.arjuna**
com.arjuna.ats.internal.arjuna.objectstore.hornetq.HornetqJournalEnvironmentBean	**com.arjuna.ats.arjuna.hornetqjournal**
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean	**com.arjuna.ats.arjuna.objectstore**
com.arjuna.ats.arjuna.common.RecoveryEnvironmentBean	**com.arjuna.ats.arjuna.recovery**
com.arjuna.ats.jdbc.common.JDBCEnvironmentBean	**com.arjuna.ats.jdbc**

Configuration Bean	Property Prefix
com.arjuna.ats.jta.common.JTAEnvironmentBean	**com.arjuna.ats.jta**
com.arjuna.ats.txoj.common.TxojEnvironmentBean	**com.arjuna.ats.txoj.lockstore**

The *prefix* can simplify the configuration. However, you should typically use either of the following formats:

NameEnvironmentBean.propertyName (the preferred format), or

fully-qualified-class-name.field-name

For example, consider the **com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.commitOnePhase** field. It may be configured by using the **com.arjuna.ats.arjuna.common.CoordinatorEnvironmentBean.commitOnePhase** property or it can be configured by using the simpler (preferred) form **CoordinatorEnvironmentBean.commitOnePhase**. Full details of how to set properties and which beans can be configured can be found in the Narayana Product Documentation .

Some beans, such as the **ObjectStoreEnvironmentBean**, may be configured multiple times with each *named* instance providing configuration for a different purposes. In this case, the name of the instance is used between the prefix (any of the above) and **field-name**. For example, a type of object store for an **ObjectStoreEnvironmentBean** instance that is named **communicationStore** may be configured by using properties that are named:

- **com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType**
- **ObjectStoreEnvironmentBean.communicationStore.objectStoreType**

4.4. CONFIGURING LOG STORAGE

The most important configuration is the type and location of object log storage. There are typically three implementations of the **com.arjuna.ats.arjuna.objectstore.ObjectStoreAPI** interface:

com.arjuna.ats.internal.arjuna.objectstore.hornetq.HornetqObjectStoreAdaptor
: Uses **org.apache.activemq.artemis.core.journal.Journal** storage from AMQ 7 internally.

com.arjuna.ats.internal.arjuna.objectstore.jdbc.JDBCStore
: Uses JDBC to keep TX log files.

com.arjuna.ats.internal.arjuna.objectstore.FileSystemStore (and specialized implementations)
: Uses custom file-based log storage.

By default, Fuse uses **com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore**, which is a specialized implementation of **FileSystemStore**.

There are three *stores* that are used by Narayana for which transaction/object logs are kept:

- **defaultStore**

- **communicationStore**
- **stateStore**

See State management in Narayana documentation for more details.

The default configuration of these three *stores* is:

```
# default store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.localOSRoot = defaultStore
# communication store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreType
= com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.communicationStore.localOSRoot =
communicationStore
# state store
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreType =
com.arjuna.ats.internal.arjuna.objectstore.ShadowNoFileLockStore
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.objectStoreDir =
${karaf.data}/narayana
com.arjuna.ats.arjuna.common.ObjectStoreEnvironmentBean.stateStore.localOSRoot = stateStore
```

ShadowNoFileLockStore is configured with the base directory (**objectStoreDir**) and the particular store's directory (**localOSRoot**).

The many configuration options are contained in the Narayana documentation guide. However, the Narayana documentation states that the canonical reference for configuration options is the Javadoc for the various **EnvironmentBean** classes.

CHAPTER 5. USING THE NARAYANA TRANSACTION MANAGER

This section provides details for using the Narayana transaction manager by implementing the **javax.transaction.UserTransaction** interface, the **org.springframework.transaction.PlatformTransactionManager** interface, or the **javax.transaction.Transaction** interface. Which interface you choose to use depends on the needs of your application. At the end of this chapter, there is a discussion of the resolution of the problem of enlisting XA resources. The information is organized as follows:

- Section 5.1, "Using UserTransaction objects"
- Section 5.2, "Using TransactionManager objects"
- Section 5.3, "Using Transaction objects"
- Section 5.4, "Resolving the XA enlistment problem"

For Java transaction API details, see the Java Transaction API (JTA) 1.2 specification and the Javadoc.

5.1. USING USERTRANSACTION OBJECTS

Implement the **javax.transaction.UserTransaction** interface for transaction demarcation. That is, for beginning, committing, or rolling back transactions. This is the JTA interface that you are most likely to use directly in your application code. However, the **UserTransaction** interface is just one of the ways to demarcate transactions. For a discussion of different ways that you can demarcate transactions, see Chapter 9, *Writing a Camel application that uses transactions* .

5.1.1. Definition of the UserTransaction interface

The JTA **UserTransaction** interface is defined as follows:

```
public interface javax.transaction.UserTransaction {

    public void begin();

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);
}
```

5.1.2. Description of UserTransaction methods

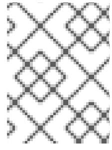
The **UserTransaction** interface defines the following methods:

begin()

Starts a new transaction and associates it with the current thread. If any XA resources get associated with this transaction, the transaction implicitly becomes an XA transaction.

commit()

Completes the current transaction normally, so that all pending changes become permanent. After the commit, there is no longer a transaction associated with the current thread.



NOTE

If the current transaction is marked as rollback only, however, the transaction would actually be rolled back when **commit()** is called.

rollback()

Aborts the transaction immediately, so that all pending changes are discarded. After the rollback, there is no longer a transaction associated with the current thread.

setRollbackOnly()

Modifies the state of the current transaction, so that a rollback is the only possible outcome, but does not perform the rollback yet.

getStatus()

Returns the status of the current transaction, which can be one of the following integer values, as defined in the **javax.transaction.Status** interface:

- **STATUS_ACTIVE**
- **STATUS_COMMITTED**
- **STATUS_COMMITTING**
- **STATUS_MARKED_ROLLBACK**
- **STATUS_NO_TRANSACTION**
- **STATUS_PREPARED**
- **STATUS_PREPARING**
- **STATUS_ROLLEDBACK**
- **STATUS_ROLLING_BACK**
- **STATUS_UNKNOWN**

setTransactionTimeout()

Customizes the timeout of the current transaction, specified in units of seconds. If the transaction is not resolved within the specified timeout, the transaction manager automatically rolls it back.

5.2. USING TRANSACTIONMANAGER OBJECTS

The most common way to use a **javax.transaction.TransactionManager** object is to pass it to a framework API, for example, to the Camel JMS component. This enables the framework to look after transaction demarcation for you. Occasionally, you might want to use a **TransactionManager** object directly. This is useful when you need to access advanced transaction APIs such as the **suspend()** and **resume()** methods.

5.2.1. Definition of the TransactionManager interface

The JTA TransactionManager interface has the following definition:

```
interface javax.transaction.TransactionManager {

    // Same as UserTransaction methods

    public void begin();

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public void setTransactionTimeout(int seconds);

    // Extra TransactionManager methods

    public Transaction getTransaction();

    public Transaction suspend() ;

    public void resume(Transaction tobj);
}
```

5.2.2. Description of TransactionManager methods

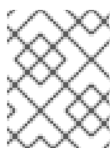
The **TransactionManager** interface supports all of the methods found in the **UserTransaction** interface. You can use a **TransactionManager** object for transaction demarcation. In addition, a **TransactionManager** object supports these methods:

getTransaction()

Gets a reference to the current transaction, which is the transaction that is associated with the current thread. If there is no current transaction, this method returns **null**.

suspend()

Detaches the current transaction from the current thread and returns a reference to the transaction. After calling this method, the current thread no longer has a transaction context. Any work that you do after this point is no longer done in the context of a transaction.



NOTE

Not all transaction managers support suspending transactions. This feature is supported by Narayana, however.

resume()

Re-attaches a suspended transaction to the current thread context. After calling this method, the transaction context is restored and any work that you do after this point is done in the context of a transaction.

5.3. USING TRANSACTION OBJECTS

You might need to use a **javax.transaction.Transaction** object directly if you are suspending/resuming transactions or if you need to enlist a resource explicitly. As discussed in Section 5.4, "Resolving the XA enlistment problem", a framework or container usually takes care of enlisting resources automatically.

5.3.1. Definition of the Transaction interface

The JTA **Transaction** interface has the following definition:

```
interface javax.transaction.Transaction {

    public void commit();

    public void rollback();

    public void setRollbackOnly();

    public int getStatus();

    public boolean enlistResource(XAResource xaRes);

    public boolean delistResource(XAResource xaRes, int flag);

    public void registerSynchronization(Synchronization sync);
}
```

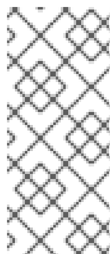
5.3.2. Description of Transaction methods

The **commit()**, **rollback()**, **setRollbackOnly()**, and **getStatus()** methods have the same behavior as the corresponding methods from the **UserTransaction** interface. In fact, a **UserTransaction** object is a convenient wrapper that retrieves the current transaction and then invokes the corresponding methods on the **Transaction** object.

Additionally, the **Transaction** interface defines the following methods, which have no counterparts in the **UserTransaction** interface:

enlistResource()

Associates an XA resource with the current transaction.



NOTE

This method is of key importance in the context of XA transactions. It is precisely the capability to enlist multiple XA resources with the current transaction that characterizes XA transactions. On the other hand, enlisting resources explicitly is a nuisance and you would normally expect your framework or container to do this for you. For example, see Section 5.4, "Resolving the XA enlistment problem" .

delistResource()

Disassociates the specified resource from the transaction. The flag argument can take one of the following integer values as defined in the **javax.transaction.Transaction** interface:

- **TMSUCCESS**

- **TMFAIL**
- **TMSUSPEND**

registerSynchronization()

Registers a **javax.transaction.Synchronization** object with the current transaction. The **Synchronization** object receives a callback just before the prepare phase of a commit and receives a callback just after the transaction completes.

5.4. RESOLVING THE XA ENLISTMENT PROBLEM

The standard JTA approach to enlisting XA resources is to add the XA resource explicitly to the current **javax.transaction.Transaction** object, which represents the current transaction. In other words, you must explicitly enlist an XA resource each time a new transaction starts.

5.4.1. How to enlist an XA resource

Enlisting an XA resource with a transaction involves invoking the **enlistResource()** method on the **Transaction** interface. For example, given a **TransactionManager** object and an **XAResource** object, you could enlist the **XAResource** object as follows:

```
// Java
import javax.transaction.Transaction;
import javax.transaction.TransactionManager;
import javax.transaction.xa.XAResource;
...
// Given:
// 'tm' of type TransactionManager
// 'xaResource' of type XAResource

// Start the transaction
tm.begin();

Transaction transaction = tm.getTransaction();
transaction.enlistResource(xaResource);

// Do some work...
...

// End the transaction
tm.commit();
```

The tricky aspect of enlisting resources is that the resource must be enlisted on *each* new transaction and the resource must be enlisted before you start to use the resource. If you enlist resources explicitly, you could end up with error-prone code that is littered with **enlistResource()** calls. Moreover, sometimes it can be difficult to call **enlistResource()** in the right place, for example, this is the case if you are using a framework that hides some of the transaction details.

5.4.2. About auto-enlistment

Instead of explicitly enlisting XA resources, it is easier and safer to use features that support auto-enlistment of XA resources. For example, in the context of using JMS and JDBC resources, the standard technique is to use wrapper classes that support auto-enlistment.

The common pattern, both for JDBC and JMS access is:

1. The application code expects **javax.sql.DataSource** for JDBC access and **javax.jms.ConnectionFactory** for JMS to get JDBC or JMS connections.

2. Within an application/OSGi server, database or broker specific implementations of these interfaces are registered.

3. An application/OSGi server *wraps* the database/broker-specific factories into generic, pooling, enlisting factories.

In this way, application code still uses **javax.sql.DataSource** and **javax.jms.ConnectionFactory**, but internally when these are accessed, there is additional functionality, which usually concerns:

- *Connection pooling* - instead of creating new connections to a database/message broker every time, a *pool* of pre-initialized connections is used. Another aspect of *pooling* may be, for example, periodical validation of connections.

- *JTA enlistment* - before returning an instance of **java.sql.Connection** (JDBC) or **javax.jms.Connection** (JMS), the real connection objects are registered if they are true XA resources. Registration happens within the JTA transaction if it is available.

With auto-enlistment, application code does not have to change.

For more information about pooling and enlisting wrappers for JDBC data sources and JMS connection factories, see Chapter 6, *Using JDBC data sources* and Chapter 7, *Using JMS connection factories*.

CHAPTER 6. USING JDBC DATA SOURCES

The following topics discuss the use of JDBC data sources in the Fuse OSGi runtime:

- Section 6.1, “About the Connection interface”
- Section 6.2, “Overview of JDBC data sources”
- Section 6.3, “Configuring JDBC data sources”
- Section 6.4, “Using the OSGi JDBC service”
- Section 6.5, “Using JDBC console commands”
- Section 6.6, “Using encrypted configuration values”
- Section 6.7, “Using JDBC connection pools”
- Section 6.8, “Deploying data sources as artifacts”
- Section 6.9, “Using data sources with the Java™ persistence API”

6.1. ABOUT THE CONNECTION INTERFACE

The most important *object* used to perform data manipulation is an implementation of the **java.sql.Connection** interface. From the perspective of Fuse configuration, it is important to learn how to *obtain* a **Connection** object.

The libraries that contain the relevant objects are:

- PostgreSQL: **mvn:org.postgresql/postgresql/42.2.5**
- MySQL: **mvn:mysql/mysql-connector-java/5.1.34**

The existing implementations (contained in *driver JARs*) provide:

- PostgreSQL: **org.postgresql.jdbc.PgConnection**
- MySQL: **com.mysql.jdbc.JDBC4Connection** (see also the various **connect*()** methods of **com.mysql.jdbc.Driver**)

These implementations contain database-specific logic to perform DML, DDL, and simple transaction management.

In theory, it is possible to manually create these connection objects, but there are two JDBC methods that hide the details to provide a cleaner API:

- **java.sql.Driver.connect()** - This method was used in standalone applications a long time ago.
- **javax.sql.DataSource.getConnection()** - This is the preferred method for using the *factory* pattern. A similar method is used to obtain JMS connections from a JMS connection factory.

The *driver manager* approach is not discussed here. It is enough to state that this method is just a tiny *layer* above a plain constructor for a given connection object.

In addition to **java.sql.Connection**, which effectively implements database-specific communication protocols, there are two other specialized *connection* interfaces:

- **javax.sql.PooledConnection** represents a physical connection. Your code does not interact with this pooled connection directly. Instead, the connection obtained from the **getConnection()** method is used. This indirection enables management of connection pools at the level of an application server. The connection obtained by using **getConnection()** is usually a proxy. When such a proxy connection is closed, the physical connection is not closed and instead it becomes available again in the managed connection pool.

- **javax.sql.XAConnection** allows obtaining a **javax.transaction.xa.XAResource** object that is associated with XA-aware connection for use with **javax.transaction.TransactionManager**. Because **javax.sql.XAConnection** extends **javax.sql.PooledConnection, it also provides the `getConnection()** method, which provides access to a JDBC connection object with typical DML/DQL methods.

6.2. OVERVIEW OF JDBC DATA SOURCES

The JDBC 1.4 standard introduced the **javax.sql.DataSource** interface, which acted as a *factory* for **java.sql.Connection** objects. Usually such data sources were bound to a JNDI registry and were located inside or injected into Java EE components such as servlets or EJBs. The key aspect is that these data sources were **configured** inside the *application server* and **referenced** in deployed applications by name.

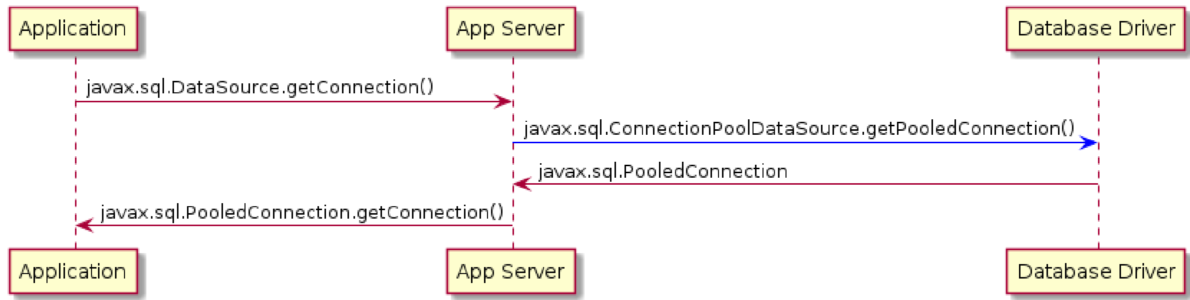
The following *connection* objects have their own *data sources*:

Data Source	Connection
javax.sql.DataSource	**java.sql.Connection**
javax.sql.ConnectionPoolDataSource	**javax.sql.PooledConnection**
javax.sql.XADataSource	**javax.sql.XAConnection**

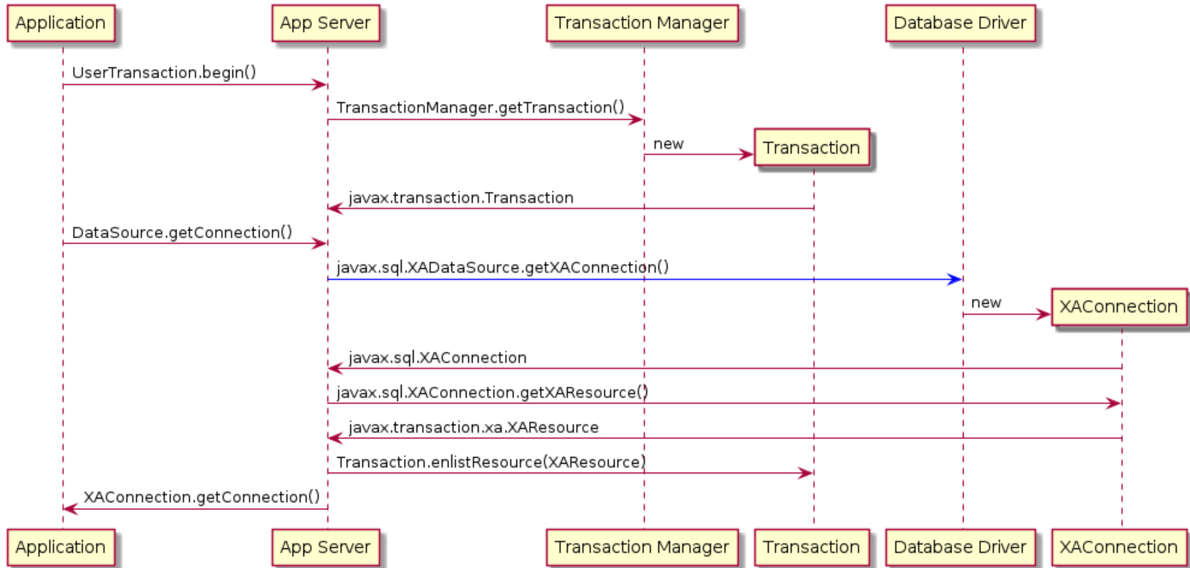
The most important differences between each of the above *data sources* is as follows:

- **javax.sql.DataSource** is most importantly a *factory*-like object for obtaining **java.sql.Connection** instances. The fact that most **javax.sql.DataSource** implementations usually perform *connection pooling* should not change the picture. This is the only interface that should be used by application code. It does not matter which of the following you are implementing:
  - Direct JDBC access
  - JPA persistence unit configuration (**either <jta-data-source>** or **<non-jta-data-source>**)
  - Popular library such as Apache Camel or Spring Framework

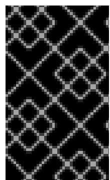
- **javax.sql.ConnectionPoolDataSource** is most importantly a *bridge* between a generic (non database-specific) connection pool/data source and a database-specific data source. It may be treated as an SPI interface. Application code usually deals with a generic **javax.sql.DataSource** object that was obtained from JNDI and implemented by an application server (probably using a library such as **commons-dbcp2**). On the other end, application code does not interface with **javax.sql.ConnectionPoolDataSource** directly. It is used between an application server and a database-specific driver. The following sequence diagram shows this:



- **javax.sql.XADataSource** is a way to obtain **javax.sql.XAConnection** and **javax.transaction.xa.XAResource**. Same as **javax.sql.ConnectionPoolDataSource**, it's used between application server and database-specific driver. Here's slightly modified diagram with different actors, this time including JTA Transaction Manager:



As shown in two previous diagrams, you interact with the *App Server*, which is a generalized entity in which you can configure **javax.sql.DataSource** and **javax.transaction.UserTransaction** instances. Such instances may be accessed either by means of JNDI or by injection using CDI or another dependency mechanism.



IMPORTANT

The important point is that even if the application uses XA transactions and/or connection pooling, the application interacts with **javax.sql.DataSource** and not the two other JDBC data source interfaces.

6.2.1. Database specific and generic data sources

The JDBC data source implementations fall into two categories:

- Generic **javax.sql.DataSource** implementations such as :
  - Apache Commons DBCP(2)
  - Apache Tomcat JDBC (based on DBCP)
- Database specific implementations of **javax.sql.DataSource**, **javax.sql.XADataSource**, and **javax.sql.ConnectionPoolDataSource**

It might be confusing that a *generic* **javax.sql.DataSource** implementation cannot create database-

specific connections on its own. Even if a *generic* data source could use **java.sql.Driver.connect()** or **java.sql.DriverManager.getConnection()**, it is usually better/cleaner to configure this *generic* data source with a database-specific **javax.sql.DataSource** implementation.

When a *generic* data source is going to interact with JTA, it **must** be configured with a database-specific implementation of **javax.sql.XADataSource**.

To close the picture, a *generic* data source usually **does not** need a database-specific implementation of **javax.sql.ConnectionPoolDataSource** to perform connection pooling. Existing pools usually handle pooling without standard JDBC interfaces (**javax.sql.ConnectionPoolDataSource** and **javax.sql.PooledConnection**) and instead use their own custom implementation.

6.2.2. Some generic data sources

Consider a sample, well-known, generic data source, Apache Commons DBCP(2).

javax.sql.XADataSource implementations

DBCP2 does not include any implementation of **javax.sql.XADataSource**, which is expected.

javax.sql.ConnectionPoolDataSource implementations

DBCP2 **does** include an implementation of **javax.sql.ConnectionPoolDataSource**: **org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS**. It creates **javax.sql.PooledConnection** objects by calling **java.sql.DriverManager.getConnection()**. This pool should not be used directly and it should be treated as an *adapter* for drivers that:

- Do not provide their own **javax.sql.ConnectionPoolDataSource** implementation
- You want to use according to JDBC *recommendations* for connection pools

As shown in the sequence diagram above, the driver provides **javax.sql.ConnectionPoolDataSource** directly or with the help of an **org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS** *adapter*, while DBCP2 implements the *application server* contract with one of:

- **org.apache.commons.dbcp2.datasources.PerUserPoolDataSource**
- **org.apache.commons.dbcp2.datasources.SharedPoolDataSource**

Both these pools take an instance of **javax.sql.ConnectionPoolDataSource** at the configuration stage.

This is the most important and interesting part of DBCP2:

javax.sql.DataSource implementations

To implement the connection pooling feature, you do not have to follow JDBC *recommendations* to use **javax.sql.ConnectionPoolDataSource → javax.sql.PooledConnection** SPI.

Here is a list of *normal* data sources of DBCP2:

- **org.apache.commons.dbcp2.BasicDataSource**
- **org.apache.commons.dbcp2.managed.BasicManagedDataSource**
- **org.apache.commons.dbcp2.PoolingDataSource**
- **org.apache.commons.dbcp2.managed.ManagedDataSource**

There are two *axes* here:

basic vs *pooling*

This *axis* determines the *pooling configuration* aspect.

Both kinds of data sources perform *pooling* of **java.sql.Connection** objects. The **only** difference is that:

- A *basic* data source is configured by using bean properties such as **maxTotal** or **minIdle** used to configure an internal instance of **org.apache.commons.pool2.impl.GenericObjectPool**.

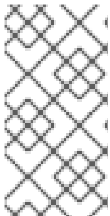
- A *pooling* data source is configured with an externally created/configured **org.apache.commons.pool2.ObjectPool**.

managed vs *non-managed*

This *axis* determines the *connection creation* aspect and the JTA behavior:

- A *non-managed basic* data source creates **java.sql.Connection** instances by using **java.sql.Driver.connect()** internally.
  A *non-managed pooling* data source creates **java.sql.Connection** instances using the passed **org.apache.commons.pool2.ObjectPool** object.

- A *managed pooling* data source wraps **java.sql.Connection** instances inside **org.apache.commons.dbcp2.managed.ManagedConnection** objects that ensure that **javax.transaction.Transaction.enlistResource()** is called if needed in the JTA context. But still the actual connection that is wrapped is obtained from any **org.apache.commons.pool2.ObjectPool** object that the pool is configured with.
  A *managed basic* data source frees you from configuring a dedicated **org.apache.commons.pool2.ObjectPool**. Instead, it is enough to configure existing, real, database-specific **javax.sql.XADataSource** objects. Bean properties are used to create an internal instance of **org.apache.commons.pool2.impl.GenericObjectPool**, which is then passed to an internal instance of a *managed pooling* data source (**org.apache.commons.dbcp2.managed.ManagedDataSource**).



NOTE

The only thing that DBCP2 cannot do is *XA transaction recovery*. DBCP2 correctly enlists XAResources in active JTA transactions, but it is not performing the recovery. This should be done separately and the configuration is usually specific to the chosen transaction manager implementation (such as Narayana).

6.2.3. Pattern to use

The recommended pattern is:

- Create or obtain a **database-specific javax.sql.DataSource** or **javax.sql.XADataSource** instance with database-specific configuration (URL, credentials, and so on) that can create connections/XA connections.

- Create or obtain a **non database-specific javax.sql.DataSource** instance (internally configured with the above, database-specific data source) with non database-specific configuration (connection pooling, transaction manager, and so on).

- Use **javax.sql.DataSource** to get an instance of **java.sql.Connection** and perform JDBC operations.

Here is a *canonical* example:

```
// Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
PGXADataSource postgresql = new org.postgresql.xa.PGXADataSource();
// Database-specific configuration
postgresql.setUrl("jdbc:postgresql://localhost:5432/reportdb");
postgresql.setUser("fuse");
postgresql.setPassword("fuse");
postgresql.setCurrentSchema("report");
postgresql.setConnectTimeout(5);
// ...

// Non database-specific, pooling, enlisting javax.sql.DataSource
BasicManagedDataSource pool = new
org.apache.commons.dbcp2.managed.BasicManagedDataSource();
// Delegate to database-specific XADatasource
pool.setXaDataSourceInstance(postgresql);
// Delegate to JTA transaction manager
pool.setTransactionManager(transactionManager);
// Non database-specific configuration
pool.setMinIdle(3);
pool.setMaxTotal(10);
pool.setValidationQuery("select schema_name, schema_owner from
information_schema.schemata");
// ...

// JDBC code:
javax.sql.DataSource applicationDataSource = pool;

try (Connection c = applicationDataSource.getConnection()) {
   try (Statement st = c.createStatement()) {
      try (ResultSet rs = st.executeQuery("select ...")) {
         // ....
```

In a Fuse environment, there are many configuration options and there is no requirement to use DBCP2.

6.3. CONFIGURING JDBC DATA SOURCES

As discussed in OSGi transaction architecture, some services must be registered in the OSGi service registry. Just as you can *find* (*lookup*) a transaction manager instance by using, for example, the **javax.transaction.UserTransaction** interface, you can do the same with JDBC data sources by using the **javax.sql.DataSource** interface. The requirements are:

- Database-specific data source that can communicate with the target database
- Generic data source where you can configure pooling and possibly transaction management (XA)

In an OSGi environment, such as Fuse, data sources become accessible from applications if they are registered as OSGi services. Fundamentally, it is done as follows:

```
org.osgi.framework.BundleContext.registerService(javax.sql.DataSource.class,
                                                 dataSourceObject,
                                                 properties);
```

```
org.osgi.framework.BundleContext.registerService(javax.sql.XADataSource.class,
                                                  xaDataSourceObject,
                                                  properties);
```

There are two methods for registering such services:

- Publishing data sources by using the **jdbc:ds-create** Karaf console command. This is the *configuration method*.
- Publishing data sources by using methods such as Blueprint, OSGi Declarative Services (SCR) or just a **BundleContext.registerService()** API call. This method requires a dedicated OSGi bundle that contains the code and/or metadata. This is the_deployment method_.

6.4. USING THE OSGI JDBC SERVICE

Chapter 125 of the OSGi Enterprise R6 specification defines a single interface in the **org.osgi.service.jdbc** package. This is how OSGi handles data sources:

```
public interface DataSourceFactory {

    java.sql.Driver createDriver(Properties props);

    javax.sql.DataSource createDataSource(Properties props);

    javax.sql.ConnectionPoolDataSource createConnectionPoolDataSource(Properties props);

    javax.sql.XADataSource createXADataSource(Properties props);
}
```

As mentioned before, plain **java.sql.Connection** connections may be obtained directly from **java.sql.Driver**.

Generic org.osgi.service.jdbc.DataSourceFactory

The simplest implementation of **org.osgi.service.jdbc.DataSourceFactory** is **org.ops4j.pax.jdbc.impl.DriverDataSourceFactory** provided by **mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0** bundle. All it does is track bundles that may include the **/META-INF/services/java.sql.Driver** descriptor for the standard Java™ *ServiceLoader* utility. If you install any standard JDBC driver, the **pax-jdbc** bundle registers a **DataSourceFactory** that can be used (not directly) to obtain connections by means of a **java.sql.Driver.connect()** call.

```
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 223
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 224
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 225
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 226

karaf@root()> bundle:services -p org.postgresql.jdbc42

PostgreSQL JDBC Driver JDBC42 (225) provides:
---------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
```

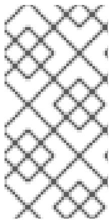
```
osgi.jdbc.driver.class = org.postgresql.Driver
osgi.jdbc.driver.name = PostgreSQL JDBC Driver
osgi.jdbc.driver.version = 42.2.5
service.bundleid = 225
service.id = 242
service.scope = singleton

karaf@root()> bundle:services -p com.mysql.jdbc

Oracle Corporation's JDBC Driver for MySQL (226) provides:
----------------------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.jdbc.Driver
osgi.jdbc.driver.name = com.mysql.jdbc
osgi.jdbc.driver.version = 5.1.34
service.bundleid = 226
service.id = 243
service.scope = singleton
-----
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.fabric.jdbc.FabricMySQLDriver
osgi.jdbc.driver.name = com.mysql.jdbc
osgi.jdbc.driver.version = 5.1.34
service.bundleid = 226
service.id = 244
service.scope = singleton

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = org.postgresql.Driver
 osgi.jdbc.driver.name = PostgreSQL JDBC Driver
 osgi.jdbc.driver.version = 42.2.5
 service.bundleid = 225
 service.id = 242
 service.scope = singleton
Provided by :
 PostgreSQL JDBC Driver JDBC42 (225)

[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = com.mysql.jdbc
 osgi.jdbc.driver.version = 5.1.34
 service.bundleid = 226
 service.id = 243
 service.scope = singleton
Provided by :
 Oracle Corporation's JDBC Driver for MySQL (226)

[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = com.mysql.fabric.jdbc.FabricMySQLDriver
 osgi.jdbc.driver.name = com.mysql.jdbc
 osgi.jdbc.driver.version = 5.1.34
 service.bundleid = 226
```

```
 service.id = 244
 service.scope = singleton
Provided by :
 Oracle Corporation's JDBC Driver for MySQL (226)
```

With the above commands, the **javax.sql.DataSource** service is still not registered, but you are one step closer. The above intermediary **org.osgi.service.jdbc.DataSourceFactory** services can be used to obtain:

- **java.sql.Driver**
- **javax.sql.DataSource** by passing properties: **url**, **user** and **password** to the **createDataSource()** method.

You cannot obtain **javax.sql.ConnectionPoolDataSource** or **javax.sql.XADataSource** from the generic **org.osgi.service.jdbc.DataSourceFactory** created by a non database-specific **pax-jdbc** bundle.



NOTE

The **mvn:org.postgresql/postgresql/42.2.5** bundle correctly implements the OSGi JDBC specification and registers an **org.osgi.service.jdbc.DataSourceFactory** instance with all methods that are implemented, including the ones that create XA and ConnectionPool data sources.

Dedicated, database-specific org.osgi.service.jdbc.DataSourceFactory implementations

There are additional bundles such as the following:

- **mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0**
- **mvn:org.ops4j.pax.jdbc/pax-jdbc-db2/1.3.0**
- ...

These bundles register database-specific **org.osgi.service.jdbc.DataSourceFactory** services that can return all kinds of *factories*, including **javax.sql.ConnectionPoolDataSource** and **javax.sql.XADataSource**. For example:

```
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 227

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.mysql

OPS4J Pax JDBC MySQL Driver Adapter (227) provides:
----------------------------------------------------
objectClass = [org.osgi.service.jdbc.DataSourceFactory]
osgi.jdbc.driver.class = com.mysql.jdbc.Driver
osgi.jdbc.driver.name = mysql
service.bundleid = 227
service.id = 245
service.scope = singleton

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
...
[org.osgi.service.jdbc.DataSourceFactory]
```

```
----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 227
 service.id = 245
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (227)
```

6.4.1. PAX-JDBC configuration service

With **pax-jdbc** (or **pax-jdbc-mysql**, **pax-jdbc-oracle**, ...) bundles, you can have **org.osgi.service.jdbc.DataSourceFactory** services registered that can be used to obtain data sources for a given database (see Section 6.2.1, "Database specific and generic data sources"). But you do not have actual data sources yet.

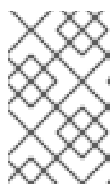
The **mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0** bundle provides a managed service factory that does two things:

- Tracks **org.osgi.service.jdbc.DataSourceFactory** OSGi services in order to invoke its methods:

  ```
  public DataSource createDataSource(Properties props);
  public XADataSource createXADataSource(Properties props);
  public ConnectionPoolDataSource createConnectionPoolDataSource(Properties props);
  ```

- Tracks **org.ops4j.datasource** *factory PIDs* to collect properties that are required by the above methods. If you create a *factory configuration* by using any method available to the Configuration Admin service, for example, by creating a **${karaf.etc}/org.ops4j.datasource-mysql.cfg** file, you can perform the final step to expose an actual database-specific data source.

Here is a detailed, *canonical* step-by-step guide for starting from a fresh installation of Fuse.



NOTE

You explicitly install bundles instead of features, to show exactly which bundles are needed. For convenience, the PAX JDBC project provides features for several database products and configuration approaches.

1. Install a JDBC driver with **/META-INF/services/java.sql.Driver**:

   ```
   karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
   Bundle ID: 223
   ```

2. Install the OSGi JDBC service bundle and **pax-jdbc-mysql** bundle that registers *intermediary***org.osgi.service.jdbc.DataSourceFactory**:

   ```
   karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
   Bundle ID: 224
   karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
   Bundle ID: 225

   karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
   ```

```
[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 225
 service.id = 242
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (225)
```

3. Install the **pax-jdbc** bundle and the **pax-jdbc-config** bundle that tracks **org.osgi.service.jdbc.DataSourceFactory** services **andorg.ops4j.datasource** *factory PIDs*:

```
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.config

OPS4J Pax JDBC Config (228) provides:
-------------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 228
service.id = 245
service.pid = org.ops4j.datasource
service.scope = singleton
```

4. Create the *factory configuration* (assume a MySQL server is running):

```
karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set user fuse
karaf@root()> config:property-set password fuse
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
FactoryPid:     org.ops4j.datasource
BundleLocation: ?
Properties:
   dataSourceName = mysqlds
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
   osgi.jdbc.driver.name = mysql
   password = fuse
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
   url = jdbc:mysql://localhost:3306/reportdb
   user = fuse
```

5. Check if **pax-jdbc-config** processed the configuration into the **javax.sql.DataSource** service:

```
karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 password = fuse
 pax.jdbc.managed = true
 service.bundleid = 228
 service.factoryPid = org.ops4j.datasource
 service.id = 246
 service.pid = org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313
 service.scope = singleton
 url = jdbc:mysql://localhost:3306/reportdb
 user = fuse
Provided by :
 OPS4J Pax JDBC Config (228)
```

You now have an actual database-specific (no pooling yet) data source. You can already inject it where it is needed. For example, you can use Karaf commands to query the database:

```
karaf@root()> feature:install -v jdbc
Adding features: jdbc/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jdbc:ds-list
Mon May 14 08:46:22 CEST 2018 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit option isn't set. For compliance with existing applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification.
Name    │ Product │ Version │ URL                                  │ Status
────────┼─────────┼─────────┼──────────────────────────────────────┼───────
mysqlds │ MySQL   │ 5.7.21  │ jdbc:mysql://localhost:3306/reportdb │ OK

karaf@root()> jdbc:query mysqlds 'select * from incident'
Mon May 14 08:46:46 CEST 2018 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit option isn't set. For compliance with existing applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification.
date                  │ summary    │ name   │ details                     │ id │ email
──────────────────────┼────────────┼────────┼─────────────────────────────┼────┼──────────────────
2018-02-20 08:00:00.0 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │ user1@redhat.com
2018-02-20 08:10:00.0 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │ user2@redhat.com
2018-02-20 08:20:00.0 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │
```

```
user3@redhat.com
2018-02-20 08:30:00.0 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │
user4@redhat.com
```

In the above example, you can see a MySQL warning. This is not a problem. Any property (not only OSGi JDBC specific ones) may be provided:

```
karaf@root()> config:property-set --pid org.ops4j.datasource.a7941498-9b62-4ed7-94f3-8c7ac9365313 useSSL false

karaf@root()> jdbc:ds-list
Name    │ Product │ Version │ URL                          │ Status
────────┼─────────┼─────────┼──────────────────────────────────────
────┼──────
mysqlds │ MySQL   │ 5.7.21  │ jdbc:mysql://localhost:3306/reportdb │ OK
```

6.4.2. Summary of handled properties

Properties from the configuration of the admin *factory PID* are passed to the relevant **org.osgi.service.jdbc.DataSourceFactory** implementation.

Generic

org.ops4j.pax.jdbc.impl.DriverDataSourceFactory properties:

- **url**
- **user**
- **password**

DB2

org.ops4j.pax.jdbc.db2.impl.DB2DataSourceFactory properties include all bean properties of these implementation classes:

- **com.ibm.db2.jcc.DB2SimpleDataSource**
- **com.ibm.db2.jcc.DB2ConnectionPoolDataSource**
- **com.ibm.db2.jcc.DB2XADataSource**

PostgreSQL

Nnative **org.postgresql.osgi.PGDataSourceFactory** properties include all properties that are specified in **org.postgresql.PGProperty**.

HSQLDB

org.ops4j.pax.jdbc.hsqldb.impl.HsqldbDataSourceFactory properties:

- **url**
- **user**
- **password**

- **databaseName**
- All bean properties of
  - **org.hsqldb.jdbc.JDBCDataSource**
  - **org.hsqldb.jdbc.pool.JDBCPooledDataSource**
  - **org.hsqldb.jdbc.pool.JDBCXADataSource**

SQL Server and Sybase

org.ops4j.pax.jdbc.jtds.impl.JTDSDataSourceFactory properties include all bean properties of **net.sourceforge.jtds.jdbcx.JtdsDataSource**.

SQL Server

org.ops4j.pax.jdbc.mssql.impl.MSSQLDataSourceFactory properties:

- **url**
- **user**
- **password**
- **databaseName**
- **serverName**
- **portNumber**
- All bean properties of
  - **com.microsoft.sqlserver.jdbc.SQLServerDataSource**
  - **com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource**
  - **com.microsoft.sqlserver.jdbc.SQLServerXADataSource**

MySQL

org.ops4j.pax.jdbc.mysql.impl.MysqlDataSourceFactory properties:

- **url**
- **user**
- **password**
- **databaseName**
- **serverName**
- **portNumber**
- All bean properties of
  - **com.mysql.jdbc.jdbc2.optional.MysqlDataSource**

- **com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource**
- **com.mysql.jdbc.jdbc2.optional.MysqlXADataSource**

Oracle

org.ops4j.pax.jdbc.oracle.impl.OracleDataSourceFactory properties:

- **url**
- **databaseName**
- **serverName**
- **user**
- **password**
- All bean properties of
  - **oracle.jdbc.pool.OracleDataSource**
  - **oracle.jdbc.pool.OracleConnectionPoolDataSource**
  - **oracle.jdbc.xa.client.OracleXADataSource**

SQLite

org.ops4j.pax.jdbc.sqlite.impl.SqliteDataSourceFactory properties:

- **url**
- **databaseName**
- All bean properties of **org.sqlite.SQLiteDataSource**

6.4.3. How the pax-jdb-config bundle handles properties

The **pax-jdbc-config** bundle handles properties that prefixed with **jdbc.**. All of these properties will have this prefix removed and the remaining names will be passed over.

Here is the example, again, starting with a fresh installation of Fuse:

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
```

```
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3
FactoryPid:     org.ops4j.datasource
BundleLocation: ?
Properties:
   dataSourceName = mysqlds
   dataSourceType = DataSource
   felix.fileinstall.filename = file:/data/servers/7.9.0.fuse-790071-redhat-
00001/etc/org.ops4j.datasource-mysql.cfg
   jdbc.password = fuse
   jdbc.url = jdbc:mysql://localhost:3306/reportdb
   jdbc.useSSL = false
   jdbc.user = fuse
   osgi.jdbc.driver.name = mysql
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3

karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 dataSourceType = DataSource
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 jdbc.password = fuse
 jdbc.url = jdbc:mysql://localhost:3306/reportdb
 jdbc.user = fuse
 jdbc.useSSL = false
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 pax.jdbc.managed = true
 service.bundleid = 228
 service.factoryPid = org.ops4j.datasource
 service.id = 246
 service.pid = org.ops4j.datasource.7c3ee718-7309-46a0-ae3a-64b38b17a0a3
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (228)
```

The **pax-jdbc-config** bundle requires these properties:

- **osgi.jdbc.driver.name**
- **dataSourceName**
- **dataSourceType**

to locate and invoke relevant **org.osgi.service.jdbc.DataSourceFactory** methods. Properties that are prefixed with **jdbc.** are passed (after removing the prefix) to, for example, **org.osgi.service.jdbc.DataSourceFactory.createDataSource(properties)**. However, these properties are added, without the prefix removed, as properties of, for example, the **javax.sql.DataSource** OSGi service.

6.5. USING JDBC CONSOLE COMMANDS

Fuse provides the **jdbc** feature, which includes shell commands in the **jdbc:*** scope. A previous example showed the use of **jdbc:query**. There are also commands that hide the need to create Configuration Admin configurations.

Starting with a fresh instance of Fuse, you can register a database-specific data source with a generic **DataSourceFactory** service as follows:

```
karaf@root()> feature:install jdbc

karaf@root()> jdbc:ds-factories
Name │ Class │ Version
───────┼──────────┼──────────

karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 228

karaf@root()> jdbc:ds-factories
Name          │ Class                                 │ Version
──────────────┼───────────────────────────────────────┼────────
────
com.mysql.jdbc │ com.mysql.jdbc.Driver                │ 5.1.34
com.mysql.jdbc │ com.mysql.fabric.jdbc.FabricMySQLDriver │ 5.1.34
```

Here is an example of registering a MySQL-specific **DataSourceFactory** service:

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl

karaf@root()> feature:install pax-jdbc-mysql

karaf@root()> la -l|grep mysql

232 │ Active  │  80 │ 5.1.34          │ mvn:mysql/mysql-connector-java/5.1.34

233 │ Active  │  80 │ 1.3.0           │ mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0

karaf@root()> jdbc:ds-factories
Name          │ Class                                 │ Version
──────────────┼───────────────────────────────────────┼────────
────
com.mysql.jdbc │ com.mysql.jdbc.Driver                │ 5.1.34
mysql         │ com.mysql.jdbc.Driver                 │
com.mysql.jdbc │ com.mysql.fabric.jdbc.FabricMySQLDriver │ 5.1.34
```

The above table may be confusing, but as mentioned above, only one of the **pax-jdbc-*database*** bundles may register **org.osgi.service.jdbc.DataSourceFactory** instances that can create standard/XA/connection pool data sources that do **not** simply delegate to **java.sql.Driver.connect()**.

The following example creates and checks a MySQL data source:

```
karaf@root()> jdbc:ds-create -dt DataSource -dn mysql -url 'jdbc:mysql://localhost:3306/reportdb?
useSSL=false' -u fuse -p fuse mysqlds

karaf@root()> jdbc:ds-list
Name    │ Product │ Version │ URL                                        │ Status
────────┼─────────┼─────────┼────────────────────────────────────────────┼───────
mysqlds │ MySQL   │ 5.7.21  │ jdbc:mysql://localhost:3306/reportdb?useSSL=false │ OK

karaf@root()> jdbc:query mysqlds 'select * from incident'
date                  │ summary    │ name   │ details                     │ id │ email
──────────────────────┼────────────┼────────┼─────────────────────────────┼────┼──────────────────
2018-02-20 08:00:00.0 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │
user1@redhat.com
2018-02-20 08:10:00.0 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │
user2@redhat.com
2018-02-20 08:20:00.0 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │
user3@redhat.com
2018-02-20 08:30:00.0 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │
user4@redhat.com

karaf@root()> config:list '(service.factoryPid=org.ops4j.datasource)'
----------------------------------------------------------------
Pid:            org.ops4j.datasource.55b18993-de4e-4e0b-abb2-a4c13da7f78b
FactoryPid:     org.ops4j.datasource
BundleLocation: mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Properties:
   dataSourceName = mysqlds
   dataSourceType = DataSource
   osgi.jdbc.driver.name = mysql
   password = fuse
   service.factoryPid = org.ops4j.datasource
   service.pid = org.ops4j.datasource.55b18993-de4e-4e0b-abb2-a4c13da7f78b
   url = jdbc:mysql://localhost:3306/reportdb?useSSL=false
   user = fuse
```

As can see, the **org.ops4j.datasource** factory PID is created for you. However it is not automatically stored in **${karaf.etc}**, which is possible with **config:update**.

6.6. USING ENCRYPTED CONFIGURATION VALUES

The **pax-jdbc-config** feature is able to process Configuration Admin configurations in which values are encrypted. A popular solution is to use Jasypt encryption services, which are also used by Blueprint.

If there are any **org.jasypt.encryption.StringEncryptor** services registered in OSGi with any **alias** service property, you can refrence it in a data source *factory PID* and use encrypted passwords. Here is an example:

```
felix.fileinstall.filename = */etc/org.ops4j.datasource-mysql.cfg
dataSourceName = mysqlds
dataSourceType = DataSource
decryptor = my-jasypt-decryptor
```

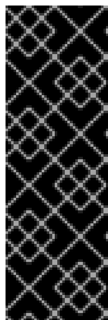
```
osgi.jdbc.driver.name = mysql
url = jdbc:mysql://localhost:3306/reportdb?useSSL=false
user = fuse
password = ENC(<encrypted-password>)
```

The service filter used to find the decryptor service is **(&(objectClass=org.jasypt.encryption.StringEncryptor)(alias=<alias>))**, where **<alias>** is the value of the **decryptor** property from the data source configuration *factory PID*.

6.7. USING JDBC CONNECTION POOLS

This section provides an introduction to using JDBC connection pools and then shows how to use these connection pool modules:

- pax-jdbc-pool-dbcp2
- pax-jdbc-pool-narayana
- pax-jdbc-pool-transx



IMPORTANT

This chapter presents exhaustive information about the internals of data source management. While information about the DBCP2 connection pool feature is provided, keep in mind that this connection pool provides proper JTA enlisting capabilities, but not XA Recovery.

To ensure that XA recovery is in place, use the **pax-jdbc-pool-transx** or **pax-jdbc-pool-narayana** connection pool module.

6.7.1. Introduction to using JDBC connection pools

Previous examples showed how to register a database-specific data source factory. Because *data source* itself is a factory for connections, **org.osgi.service.jdbc.DataSourceFactory** may be treated as a *meta factory* that should be able to produce three kinds of data sources, plus, as a bonus, a **java.sql.Driver**):

- **javax.sql.DataSource**
- **javax.sql.ConnectionPoolDataSource**
- **javax.sql.XADataSource**

For example, **pax-jdbc-mysql** registers an **org.ops4j.pax.jdbc.mysql.impl.MysqlDataSourceFactory** that produces:

- **javax.sql.DataSource → com.mysql.jdbc.jdbc2.optional.MysqlDataSource**
- **javax.sql.ConnectionPoolDataSource → com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource**
- **javax.sql.XADataSource → com.mysql.jdbc.jdbc2.optional.MysqlXADataSource**
- **java.sql.Driver → com.mysql.jdbc.Driver**

The PostgreSQL driver itself implements the OSGi JDBC service and produces:

- **javax.sql.DataSource → org.postgresql.jdbc2.optional.PoolingDataSource** (if there are pool-related properties specified) or **org.postgresql.jdbc2.optional.SimpleDataSource**
- **javax.sql.ConnectionPoolDataSource → org.postgresql.jdbc2.optional.ConnectionPool**
- **javax.sql.XADataSource → org.postgresql.xa.PGXADataSource**
- **java.sql.Driver → org.postgresql.Driver**

As shown in the canonical DataSource example, any *pooling*, *generic* data source, if it is going to work in a JTA environment, needs a *database specific* data source to actually obtain (XA) connections.

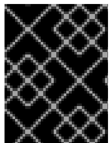
We already have the latter, and we need actual, generic, reliable connection pool.

The canonical DataSource example shows how to configure a generic pool with a database-specific data source. The **pax-jdbc-pool-*** bundles work smoothly with the above described **org.osgi.service.jdbc.DataSourceFactory** services.

Just as the OSGI Enterprise R6 JDBC specification provides the **org.osgi.service.jdbc.DataSourceFactory** standard interface, **pax-jdbc-pool-common** provides *proprietary* **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** interface:

```
public interface PooledDataSourceFactory {

    javax.sql.DataSource create(org.osgi.service.jdbc.DataSourceFactory dsf, Properties config)

}
```

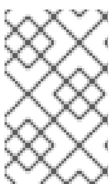
This interface is perfectly conformant with what this important note that was presented previously and is worth repeating:



IMPORTANT

Even if the application uses XA transactions and/or connection pooling, the application interacts with **javax.sql.DataSource** and not the two other JDBC data source interfaces.

This interface simply creates a pooling data source out of a database-specific, non-pooling datas ource. Or more precisely, it is a *data source factory* (*meta factory*) that turns a factory of database-specific data sources into a factory of pooling data sources.



NOTE

There is nothing that prevents an application from configuring pooling for a **javax.sql.DataSource** object by using an **org.osgi.service.jdbc.DataSourceFactory** service that already returns pooling for **javax.sql.DataSource** objects.

The following table shows which bundles register pooled data source factories. In the table, instances of **o.o.p.j.p** represent **org.ops4j.pax.jdbc.pool**.

Bundle	PooledDataSourceFactory	Pool Key
pax-jdbc-pool-narayana	**o.o.p.j.p.narayana.impl.Dbcp(XA)PooledDataSourceFactory**	**narayana**

Bundle	PooledDataSourceFactory	Pool Key
pax-jdbc-pool-dbcp2	**o.o.p.j.p.dbcp2.impl.Dbcp(XA)PooledDataSourceFactory**	**dbcp2**
pax-jdbc-pool-transx	**o.o.p.j.p.transx.impl.Transx(Xa)PooledDataSourceFactory**	**transx**

The above bundles install only data source factories and not the data sources themselves. The application needs something that calls the **javax.sql.DataSource create(org.osgi.service.jdbc.DataSourceFactory dsf, Properties config)** method.

6.7.2. Using the dbcp2 connection pool module

The section about generic data sources provides an example of how to use and configure the Apache Commons DBCP module. This section shows how to do this in the Fuse OSGi environment.

Consider the Section 6.4.1, "PAX-JDBC configuration service" bundle. In addition to tracking the following:

- **org.osgi.service.jdbc.DataSourceFactory** services
- **org.ops4j.datasource** *factory PIDs*

The bundle also tracks instances of **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** that are registered by one of the **pax-jdbc-pool-*** bundles.

If the *factory configuration* contains the **pool** property, then the ultimate data source registered by the **pax-jdbc-config** bundle is the database-specific datas ource, but wrapped inside one of the following if **pool=dbcp2**):

- **org.apache.commons.dbcp2.PoolingDataSource**
- **org.apache.commons.dbcp2.managed.ManagedDataSource**

This is consistent with the generic data source example . In addition to the **pool** property, and the boolean **xa** property, which selects a non-xa or an xa data source, the **org.ops4j.datasource** *factory PID* may contain *prefixed* properties:

- **pool.***
- **factory.***

Where each property is used depends on which **pax-jdbc-pool-*** bundle isused. For DBCP2, it is:

- **pool.***: bean properties of **org.apache.commons.pool2.impl.GenericObjectPoolConfig** (both xa and non-xa scenario)
- **factory.***: bean properties of **org.apache.commons.dbcp2.managed.PoolableManagedConnectionFactory** (xa) or **org.apache.commons.dbcp2.PoolableConnectionFactory** (non-xa)

6.7.2.1. Configuration properties for BasicDataSource

The following table lists the generic configuration properties for BasicDataSource.

Parameter	Default	Description
username		The connection user name to be passed to our JDBC driver to establish a connection.
password		The connection password to be passed to our JDBC driver to establish a connection.
url		The connection URL to be passed to our JDBC driver to establish a connection.
driverClassName		The fully qualified Java class name of the JDBC driver to be used.
initialSize	0	The initial number of connections that are created when the pool is started.
maxTotal	8	The maximum number of active connections that can be allocated from this pool at the same time, or negative for no limit.
maxIdle	8	The maximum number of connections that can remain idle in the pool, without extra ones being released, or negative for no limit.
minIdle	0	The minimum number of connections that can remain idle in the pool, without extra ones being created, or zero to create none.
maxWaitMillis	indefinitely	The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or -1 to wait indefinitely.
validationQuery		The SQL query that will be used to validate connections from this pool before returning them to the caller. If specified, this query MUST be an SQL SELECT statement that returns at least one row. If not specified, connections will be validation by calling the isValid() method.

Parameter	Default	Description
validationQueryTimeout	no timeout	The timeout in seconds before connection validation queries fail. If set to a positive value, this value is passed to the driver via the setQueryTimeout method of the Statement used to execute the validation query.
testOnCreate	false	The indication of whether objects will be validated after creation. If the object fails to validate, the borrow attempt that triggered the object creation will fail.
testOnBorrow	true	The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another.
testOnReturn	false	The indication of whether objects will be validated before being returned to the pool.
testWhileIdle	false	The indication of whether objects will be validated by the idle object evictor (if any). If an object fails to validate, it will be dropped from the pool.
timeBetweenEvictionRunsMillis	-1	The number of milliseconds to sleep between runs of the idle object evictor thread. When non-positive, no idle object evictor thread will be run.
numTestsPerEvictionRun	3	The number of objects to examine during each run of the idle object evictor thread (if any).
minEvictableIdleTimeMillis	1000 * 60 * 30	The minimum amount of time an object may sit idle in the pool before it is eligible for eviction by the idle object evictor (if any).

6.7.2.2. Example of how to configure DBCP2 pool

The following is a realistic example (except **useSSL=false**) of a configuration of a DBCP2 pool (**org.ops4j.datasource-mysql** *factory PID*) that uses convenient syntax with **jdbc.**-prefixed properties:

```
# Configuration for pax-jdbc-config to choose and configure specific
org.osgi.service.jdbc.DataSourceFactory
dataSourceName = mysqlds
```

```
dataSourceType = DataSource
osgi.jdbc.driver.name = mysql
jdbc.url = jdbc:mysql://localhost:3306/reportdb
jdbc.user = fuse
jdbc.password = fuse
jdbc.useSSL = false

# Hints for pax-jdbc-config to use org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory
pool = dbcp2
xa = false

# dbcp2 specific configuration of org.apache.commons.pool2.impl.GenericObjectPoolConfig
pool.minIdle = 10
pool.maxTotal = 100
pool.initialSize = 8
pool.blockWhenExhausted = true
pool.maxWaitMillis = 2000
pool.testOnBorrow = true
pool.testWhileIdle = false
pool.timeBetweenEvictionRunsMillis = 120000
pool.evictionPolicyClassName = org.apache.commons.pool2.impl.DefaultEvictionPolicy

# dbcp2 specific configuration of org.apache.commons.dbcp2.PoolableConnectionFactory
factory.maxConnLifetimeMillis = 30000
factory.validationQuery  = select schema_name from information_schema.schemata
factory.validationQueryTimeout = 2
```

In the above configuration, **pool** and **xa** keys are *hints* (service filter properties) to choose one of the registered **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** services. In the case of DBCP2, this is:

```
karaf@root()> feature:install pax-jdbc-pool-dbcp2

karaf@root()> bundle:services -p org.ops4j.pax.jdbc.pool.dbcp2

OPS4J Pax JDBC Pooling DBCP2 (230) provides:
--------------------------------------------
objectClass = [org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
pool = dbcp2
service.bundleid = 230
service.id = 337
service.scope = singleton
xa = false
-----
objectClass = [org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
pool = dbcp2
service.bundleid = 230
service.id = 338
service.scope = singleton
xa = true
```

For completeness, here is a full example with connection pool configuration added to the previous example. Again, this assumes that you are starting with a fresh Fuse installation.

1. Install a JDBC driver:

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
```

2. Install the **jdbc**, **pax-jdbc-mysql** and **pax-jdbc-pool-dbcp2** features:

```
karaf@root()> feature:repo-add mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl
Adding feature url mvn:org.ops4j.pax.jdbc/pax-jdbc-features/1.3.0/xml/features-gpl

karaf@root()> feature:install jdbc pax-jdbc-mysql pax-jdbc-pool-dbcp2

karaf@root()> service:list org.osgi.service.jdbc.DataSourceFactory
...
[org.osgi.service.jdbc.DataSourceFactory]
-----------------------------------------
 osgi.jdbc.driver.class = com.mysql.jdbc.Driver
 osgi.jdbc.driver.name = mysql
 service.bundleid = 232
 service.id = 328
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC MySQL Driver Adapter (232)

karaf@root()> service:list org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory
[org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
--------------------------------------------------------
 pool = dbcp2
 service.bundleid = 233
 service.id = 324
 service.scope = singleton
 xa = false
Provided by :
 OPS4J Pax JDBC Pooling DBCP2 (233)

[org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory]
--------------------------------------------------------
 pool = dbcp2
 service.bundleid = 233
 service.id = 332
 service.scope = singleton
 xa = true
Provided by :
 OPS4J Pax JDBC Pooling DBCP2 (233)
```

3. Create the *factory configuration*:

```
karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:property-set pool dbcp2
```

```
karaf@root()> config:property-set xa false
karaf@root()> config:property-set pool.minIdle 2
karaf@root()> config:property-set pool.maxTotal 10
karaf@root()> config:property-set pool.blockWhenExhausted true
karaf@root()> config:property-set pool.maxWaitMillis 2000
karaf@root()> config:property-set pool.testOnBorrow true
karaf@root()> config:property-set pool.testWhileIdle alse
karaf@root()> config:property-set pool.timeBetweenEvictionRunsMillis 120000
karaf@root()> config:property-set factory.validationQuery 'select schema_name from
information_schema.schemata'
karaf@root()> config:property-set factory.validationQueryTimeout 2
karaf@root()> config:update
```

4. Check if **pax-jdbc-config** processed the configuration into the **javax.sql.DataSource** service:

```
karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 dataSourceName = mysqlds
 dataSourceType = DataSource
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.datasource-mysql.cfg
 jdbc.password = fuse
 jdbc.url = jdbc:mysql://localhost:3306/reportdb
 jdbc.user = fuse
 jdbc.useSSL = false
 osgi.jdbc.driver.name = mysql
 osgi.jndi.service.name = mysqlds
 pax.jdbc.managed = true
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = alse
 pool.timeBetweenEvictionRunsMillis = 120000
 service.bundleid = 225
 service.factoryPid = org.ops4j.datasource
 service.id = 338
 service.pid = org.ops4j.datasource.fd7aa3a1-695b-4342-b0d6-23d018a46fbb
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (225)
```

5. Use the data source:

```
karaf@root()> jdbc:query mysqlds 'select * from incident'
date                  │ summary    │ name   │ details                   │ id │ email
──────────────────────────────────────┼─────────────────────┼──────────────┼──────────────────────
────────────────────────────┼───────┼───────────────────────
2018-02-20 08:00:00.0 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │
user1@redhat.com
2018-02-20 08:10:00.0 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │
user2@redhat.com
```

```
2018-02-20 08:20:00.0 | Incident 3 | User 3 | This is a report incident 003 | 3 |
user3@redhat.com
2018-02-20 08:30:00.0 | Incident 4 | User 4 | This is a report incident 004 | 4 |
user4@redhat.com
```

6.7.3. Using the narayana connection pool module

The **pax-jdbc-pool-narayna** module does almost everything as **pax-jdbc-pool-dbcp2**. It installs the DBCP2-specific **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory**, for both XA and non-XA scenarios. The **only** difference is that in XA scenarios there is an additional integration point. The **org.jboss.tm.XAResourceRecovery** OSGi service is registered to be picked up by **com.arjuna.ats.arjuna.recovery.RecoveryManager**, which is part of the Narayana transaction manager.

6.7.4. Using the transx connection pool module

The **pax-jdbc-pool-transx** bundle bases its implementation of **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory** services on the **pax-transx-jdbc** bundle. The **pax-transx-jdbc** bundle creates **javax.sql.DataSource** pools by using the **org.ops4j.pax.transx.jdbc.ManagedDataSourceBuilder** facility. This is a JCA (Java™ Connector Architecture) solution and it is described in later.

6.8. DEPLOYING DATA SOURCES AS ARTIFACTS

This chapter introduced OSGi JDBC services, showed how **pax-jdbc** bundles help with registration of database-specific and generic data sources, and how it all looks from the perspective of OSGi services and Configuration Admin configurations. While configuration of both categories of data sources may be done by using Configuration Admin factory PIDs (with help from the **pax-jdbc-config** bundle), it is usually preferred to use the *deployment method*.

In the *deployment method*, **javax.sql.DataSource** services are registered directly by application code, usually inside a Blueprint container. Blueprint XML may be part of an ordinary OSGi bundle, installable by using a **mvn:** URI and stored in a Maven repository (local or remote). It is much easier to version-control such bundles by comparing them to Configuration Admin configurations.

The **pax-jdbc-config** bundle version 1.3.0 adds a *deployment method* for data source configuration. An application developer registers the **javax.sql.(XA)DataSource** service (usually by using Bluerpint XML) and specifies service properties. The **pax-jdbc-config** bundle detects such registered database-specific data sources and (using service properties) wraps the service inside a generic, non database-specific, connection pool.

For completeness, following are three *deployment methods* that use Blueprint XML. Fuse provides a **quickstarts** download with various examples of different aspects of Fuse. You can download the **quickstarts** zip file from the Fuse Software Downloads page.

Extract the contents of the quickstarts zip file to a local folder.

In the following examples, the **quickstarts/persistence** directory is referred to as **$PQ_HOME**.

- Section 6.8.1, “Manual deployment of data sources”
- Section 6.8.2, “Factory deployment of data sources”
- Section 6.8.3, “Mixed deployment of data sources”

6.8.1. Manual deployment of data sources

This example of manual deployment of data sources uses a docker-based PostgreSQL installation. In this method, the **pax-jdbc-config** is not needed. Application code is responsible for registration of both database-specific and generic data sources.

These three bundles are needed:

- **mvn:org.postgresql/postgresql/42.2.5**
- **mvn:org.apache.commons/commons-pool2/2.5.0**
- **mvn:org.apache.commons/commons-dbcp2/2.1.1**

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Fuse/Karaf exports this service from fuse-pax-transx-tm-narayana bundle
-->
<reference id="tm" interface="javax.transaction.TransactionManager" />

<!--
   Non database-specific, generic, pooling, enlisting javax.sql.DataSource
-->
<bean id="pool" class="org.apache.commons.dbcp2.managed.BasicManagedDataSource">
   <property name="xaDataSourceInstance" ref="postgresql" />
   <property name="transactionManager" ref="tm" />
   <property name="minIdle" value="3" />
   <property name="maxTotal" value="10" />
   <property name="validationQuery" value="select schema_name, schema_owner from
information_schema.schemata" />
</bean>

<!--
   Expose datasource to use by application code (like Camel, Spring, ...)
-->
<service interface="javax.sql.DataSource" ref="pool">
   <service-properties>
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
   </service-properties>
</service>
```

The above Blueprint XML fragment matches the canonical DataSource example . Here are the shell commands that show how it should be used:

```
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 233
```

```
karaf@root()> install -s mvn:org.apache.commons/commons-pool2/2.5.0
Bundle ID: 224
karaf@root()> install -s mvn:org.apache.commons/commons-dbcp2/2.1.1
Bundle ID: 225
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-manual.xml
Bundle ID: 226

karaf@root()> bundle:services -p 226

Bundle 226 provides:
--------------------
objectClass = [javax.sql.DataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = pool
service.bundleid = 226
service.id = 242
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-manual.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 226
service.id = 243
service.scope = singleton

karaf@root()> feature:install jdbc

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                    │ URL
│ Status
────────────────────────┼───────────────────┼──────────────────────────────────────────────┼──
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────┼──────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK
```

```
karaf@root()> jdbc:query jdbc/postgresql 'select * from incident';
date                │ summary    │ name   │ details                    │ id │ email
───────────────────┼────────────┼────────┼────────────────────────────┼────┼───────────────────
2018-02-20 08:00:00 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │ user1@redhat.com
2018-02-20 08:10:00 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │ user2@redhat.com
2018-02-20 08:20:00 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │ user3@redhat.com
2018-02-20 08:30:00 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │ user4@redhat.com
```

As shown in the above listing, the Blueprint bundle exports the **javax.sql.DataSource** service, which is a generic, non database-specific, connection pool. The database-specific **javax.sql.XADataSource** bundle is **not** registered as an OSGi service, because Blueprint XML does not have an explicit **<service ref="postgresql">** declaration.

6.8.2. Factory deployment of data sources

Factory deployment of data sources uses the **pax-jdbc-config** bundle in a *canonical* way. This is a bit different from the method that was recommended in Fuse 6.x, which required specification of the pooling configuration as service properties.

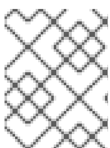
Here is the Blueprint XML example:

```
<!--
   A database-specific org.osgi.service.jdbc.DataSourceFactory that can create
DataSource/XADataSource/
   /ConnectionPoolDataSource/Driver using properties. It is registered by pax-jdbc-* or for example
   mvn:org.postgresql/postgresql/42.2.5 bundle natively.
-->
<reference id="dataSourceFactory"
      interface="org.osgi.service.jdbc.DataSourceFactory"
      filter="(osgi.jdbc.driver.class=org.postgresql.Driver)" />

<!--
   Non database-specific org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory that can create
   pooled data sources using some org.osgi.service.jdbc.DataSourceFactory. dbcp2 pool is registered
   by pax-jdbc-pool-dbcp2 bundle.
-->
<reference id="pooledDataSourceFactory"
      interface="org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory"
      filter="(&amp;(pool=dbcp2)(xa=true))" />

<!--
   Finally, use both factories to expose pooled, xa-aware data source.
-->
<bean id="pool" factory-ref="pooledDataSourceFactory" factory-method="create">
   <argument ref="dataSourceFactory" />
   <argument>
      <props>
         <!--
            Properties needed by postgresql-specific org.osgi.service.jdbc.DataSourceFactory.
            Cannot prepend them with 'jdbc.' prefix as the DataSourceFactory is implemented directly
            by PostgreSQL driver, not by pax-jdbc-* bundle.
         -->
         <prop key="url" value="jdbc:postgresql://localhost:5432/reportdb" />
         <prop key="user" value="fuse" />
```

```
            <prop key="password" value="fuse" />
            <prop key="currentSchema" value="report" />
            <prop key="connectTimeout" value="5" />
            <!-- Properties needed by dbcp2-specific
org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory -->
            <prop key="pool.minIdle" value="2" />
            <prop key="pool.maxTotal" value="10" />
            <prop key="pool.blockWhenExhausted" value="true" />
            <prop key="pool.maxWaitMillis" value="2000" />
            <prop key="pool.testOnBorrow" value="true" />
            <prop key="pool.testWhileIdle" value="false" />
            <prop key="factory.validationQuery" value="select schema_name from
information_schema.schemata" />
            <prop key="factory.validationQueryTimeout" value="2" />
        </props>
    </argument>
</bean>

<!--
    Expose data source for use by application code (such as  Camel, Spring, ...).
-->
<service interface="javax.sql.DataSource" ref="pool">
    <service-properties>
        <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
    </service-properties>
</service>
```

This example uses *factory* beans that create data sources by using data source factories. You do not need to explicitly reference the **javax.transaction.TransactionManager** service, as this is tracked internally by the XA-aware **PooledDataSourceFactory**.

The following is the same example but in a Fuse/Karaf shell.



NOTE

To have the native **org.osgi.service.jdbc.DataSourcFactory** bundle registered, install **mvn:org.osgi/org.osgi.service.jdbc/1.0.0** and then install a PostgreSQL driver.

```
karaf@root()> feature:install jdbc pax-jdbc-config pax-jdbc-pool-dbcp2
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 232
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-pax-jdbc-factory-
dbcp2.xml
Bundle ID: 233
karaf@root()> bundle:services -p 233

Bundle 233 provides:
--------------------
objectClass = [javax.sql.DataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = pool
service.bundleid = 233
service.id = 336
service.scope = bundle
-----
```

```
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-pax-jdbc-factory-dbcp2.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 233
service.id = 337
service.scope = singleton

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                     │ URL
│ Status
────────────────────────┼───────────────────┼──────────────────────────────────────────────┼─
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────┼────────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&bi
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&loginTi
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncodingC
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckSe
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident';
date                │ summary    │ name    │ details                    │ id │ email
──────────────────────────────┼───────────────────┼─────────────┼──────────────────────────
────────────────┼───────┼─────────────────────────
2018-02-20 08:00:00 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │ user1@redhat.com
2018-02-20 08:10:00 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │ user2@redhat.com
2018-02-20 08:20:00 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │ user3@redhat.com
2018-02-20 08:30:00 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │ user4@redhat.com
```

As shown in the above listing, the Blueprint bundle exports the **javax.sql.DataSource** service, which is a generic, non database-specific, connection pool. The database-specific **javax.sql.XADataSource** is **not** registered as an OSGi service, because Blueprint XML does not have an explicit **<service ref="postgresql">** declaration.

6.8.3. Mixed deployment of data sources

In a mixed deployment of data sources, the **pax-jdbc-config** 1.3.0 bundle adds another way of *wrapping* database-specific data sources within pooling data sources by using service properties. This method matches the way it worked in Fuse 6.x.

The following is the Blueprint XML example:

```
<!--
   Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
   <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
   <property name="user" value="fuse" />
   <property name="password" value="fuse" />
   <property name="currentSchema" value="report" />
   <property name="connectTimeout" value="5" />
</bean>

<!--
   Expose database-specific data source with service properties.
   No need to expose pooling, enlisting, non database-specific javax.sql.DataSource. It is registered
   automatically by pax-jdbc-config with the same properties as this <service>, but with higher
service.ranking.
-->
<service id="pool" ref="postgresql" interface="javax.sql.XADataSource">
   <service-properties>
      <!-- "pool" key is needed for pax-jdbc-config to wrap database-specific data source inside
connection pool -->
      <entry key="pool" value="dbcp2" />
      <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
      <!-- Other properties that configure given connection pool, as indicated by pool=dbcp2 -->
      <entry key="pool.minIdle" value="2" />
      <entry key="pool.maxTotal" value="10" />
      <entry key="pool.blockWhenExhausted" value="true" />
      <entry key="pool.maxWaitMillis" value="2000" />
      <entry key="pool.testOnBorrow" value="true" />
      <entry key="pool.testWhileIdle" value="false" />
      <entry key="factory.validationQuery" value="select schema_name from
information_schema.schemata" />
      <entry key="factory.validationQueryTimeout" value="2" />
   </service-properties>
</service>
```

In the above example, only a database-specific data source is manually registered. The **pool=dbcp2** service property is a hint for the data source tracker that is managed by the **pax-jdbc-config** bundle. Data source services with this service property are wrapped within a pooling data source, in this example, **pax-jdbc-pool-dbcp2**.

The following is the same example in a Fuse/Karaf shell:

```
karaf@root()> feature:install jdbc pax-jdbc-config pax-jdbc-pool-dbcp2
karaf@root()> install -s mvn:org.postgresql/postgresql/42.2.5
Bundle ID: 232
karaf@root()> install -s blueprint:file://$PQ_HOME/databases/blueprints/postgresql-pax-jdbc-
discovery.xml
Bundle ID: 233
karaf@root()> bundle:services -p 233

Bundle 233 provides:
--------------------
```

```
factory.validationQuery = select schema_name from information_schema.schemata
factory.validationQueryTimeout = 2
objectClass = [javax.sql.XADataSource]
osgi.jndi.service.name = jdbc/postgresql
osgi.service.blueprint.compname = postgresql
pool = dbcp2
pool.blockWhenExhausted = true
pool.maxTotal = 10
pool.maxWaitMillis = 2000
pool.minIdle = 2
pool.testOnBorrow = true
pool.testWhileIdle = false
service.bundleid = 233
service.id = 336
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = postgresql-pax-jdbc-discovery.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 233
service.id = 338
service.scope = singleton

karaf@root()> service:list javax.sql.XADataSource
[javax.sql.XADataSource]
------------------------
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 osgi.jndi.service.name = jdbc/postgresql
 osgi.service.blueprint.compname = postgresql
 pool = dbcp2
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = false
 service.bundleid = 233
 service.id = 336
 service.scope = bundle
Provided by :
 Bundle 233
Used by:
 OPS4J Pax JDBC Config (224)

karaf@root()> service:list javax.sql.DataSource
[javax.sql.DataSource]
----------------------
 factory.validationQuery = select schema_name from information_schema.schemata
 factory.validationQueryTimeout = 2
 osgi.jndi.service.name = jdbc/postgresql
 osgi.service.blueprint.compname = postgresql
 pax.jdbc.managed = true
 pax.jdbc.service.id.ref = 336
 pool.blockWhenExhausted = true
 pool.maxTotal = 10
```

```
 pool.maxWaitMillis = 2000
 pool.minIdle = 2
 pool.testOnBorrow = true
 pool.testWhileIdle = false
 service.bundleid = 224
 service.id = 337
 service.ranking = 1000
 service.scope = singleton
Provided by :
 OPS4J Pax JDBC Config (224)

karaf@root()> jdbc:ds-list
Name            │ Product    │ Version                      │ URL
│ Status
───────────────┼──────────┼──────────────────────────┼──
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────────────────
──────────────────────────────────┼──────
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK
jdbc/postgresql │ PostgreSQL │ 10.3 (Debian 10.3-1.pgdg90+1) │
jdbc:postgresql://localhost:5432/reportdb?
prepareThreshold=5&preparedStatementCacheQueries=256&preparedStatementCacheSizeMiB=5&da
tabaseMetadataCacheFields=65536&databaseMetadataCacheFieldsMiB=5&defaultRowFetchSize=0&b
naryTransfer=true&readOnly=false&binaryTransferEnable=&binaryTransferDisable=&unknownLength=
2147483647&logUnclosedConnections=false&disableColumnSanitiser=false&tcpKeepAlive=false&login
meout=0&connectTimeout=5&socketTimeout=0&cancelSignalTimeout=10&receiveBufferSize=-
1&sendBufferSize=-1&ApplicationName=PostgreSQL JDBC
Driver&jaasLogin=true&useSpnego=false&gsslib=auto&sspiServiceClass=POSTGRES&allowEncoding
hanges=false&currentSchema=report&targetServerType=any&loadBalanceHosts=false&hostRecheckS
conds=10&preferQueryMode=extended&autosave=never&reWriteBatchedInserts=false │ OK

karaf@root()> jdbc:query jdbc/postgresql 'select * from incident'
date                │ summary    │ name   │ details                     │ id │ email
──────────────────────────┼──────────────────┼────────────┼──────────
────────────┼──────┼────────────────────────
2018-02-20 08:00:00 │ Incident 1 │ User 1 │ This is a report incident 001 │ 1  │ user1@redhat.com
```

```
2018-02-20 08:10:00 │ Incident 2 │ User 2 │ This is a report incident 002 │ 2  │ user2@redhat.com
2018-02-20 08:20:00 │ Incident 3 │ User 3 │ This is a report incident 003 │ 3  │ user3@redhat.com
2018-02-20 08:30:00 │ Incident 4 │ User 4 │ This is a report incident 004 │ 4  │ user4@redhat.com
```

In this listing, as you can see in the **jdbc:ds-list** output, there are **two** data sources, the original data source and the wrapper data source.

javax.sql.XADataSource is registered from the Blueprint bundle and has the **pool = dbcp2** property declared.

javax.sql.DataSource is registered from the **pax-jdbc-config** bundle and:

- Does not have the **pool = dbcp2** property (it was removed when registering the wrapper data source).
- Has the **service.ranking = 1000** property, so it is always the preferred version when, for example, looking for data source by name.
- Has the **pax.jdbc.managed = true** property, so it is not tried to be wrapped again.
- Has the **pax.jdbc.service.id.ref = 336** property, to indicate the original data source service that is wrapped inside the connection pool.

6.9. USING DATA SOURCES WITH THE JAVA™ PERSISTENCE API

From the perspective of transaction management, it is important to understand how data sources are used with the Java™ Persistence API (JPA). This section does not describe the details of the JPA specification itself, nor the details about Hibernate, which is the most known JPA implementation. Instead, this section shows how to point JPA persistent units to data sources.

6.9.1. About data source references

The **META-INF/persistence.xml** descriptor (see the JPA 2.1 specification, *8.2.1.5 jta-data-source, non-jta-data-source*) defines two kinds of data source references:

- **<jta-data-source>** - This is a JNDI reference to JTA-enabled data source to use with **JTA** transactions.
- **<non-jta-data-source>** - This is a JNDI reference to JTA-enabled data source to use outside of **JTA** transactions. This data source is usually also used in the initialization phase, for example, with the **hibernate.hbm2ddl.auto** property that configures Hibernate to auto-create database schema.

These two data sources are **not** related to **javax.sql.DataSource** or **javax.sql.XADataSource**! This is common misconception when developing JPA applications. Both JNDI names must refer to JNDI-bound **javax.sql.DataSource** services.

6.9.2. Referring to JNDI names

When you register an OSGi service with the **osgi.jndi.service.name** property, it is *bound* in the OSGi JNDI service. In an OSGi runtime (such as Fuse/Karaf), JNDI is not a simple dictionary of name → value pairs. Referring to objects by means of JNDI names in OSGi involves service lookups and other, more complex OSGi mechanisms, such as service hooks.

In a fresh Fuse installation, the following listing shows how data sources are registered in JNDI:

–

```
karaf@root()> install -s mvn:mysql/mysql-connector-java/5.1.34
Bundle ID: 223
karaf@root()> install -s mvn:org.osgi/org.osgi.service.jdbc/1.0.0
Bundle ID: 224
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-mysql/1.3.0
Bundle ID: 225
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc/1.3.0
Bundle ID: 226
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-pool-common/1.3.0
Bundle ID: 227
karaf@root()> install -s mvn:org.ops4j.pax.jdbc/pax-jdbc-config/1.3.0
Bundle ID: 228

karaf@root()> config:edit --factory --alias mysql org.ops4j.datasource
karaf@root()> config:property-set osgi.jdbc.driver.name mysql
karaf@root()> config:property-set dataSourceName mysqlds
karaf@root()> config:property-set osgi.jndi.service.name jdbc/mysqlds
karaf@root()> config:property-set dataSourceType DataSource
karaf@root()> config:property-set jdbc.url jdbc:mysql://localhost:3306/reportdb
karaf@root()> config:property-set jdbc.user fuse
karaf@root()> config:property-set jdbc.password fuse
karaf@root()> config:property-set jdbc.useSSL false
karaf@root()> config:update

karaf@root()> feature:install jndi

karaf@root()> jndi:names
JNDI Name                 │ Class Name
──────────────────────────┼──────────────────────────────────────────────
────────────
osgi:service/jndi         │ org.apache.karaf.jndi.internal.JndiServiceImpl
osgi:service/jdbc/mysqlds │ com.mysql.jdbc.jdbc2.optional.MysqlDataSource
```

As you can see, the data source is available under the **osgi:service/jdbc/mysqlds** JNDI name.

But in case of JPA in OSGi, you must use **full JNDI names**. The following is the sample **META-INF/persistence.xml** fragment that specifies data source references:

```
<jta-data-source>
    osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/mysqlds)
</jta-data-source>
<non-jta-data-source>
    osgi:service/javax.sql.DataSource/(osgi.jndi.service.name=jdbc/mysqlds)
</non-jta-data-source>
```

Without the above configuration, you might get this error:

```
Persistence unit "pu-name" refers to a non OSGi service DataSource
```

CHAPTER 7. USING JMS CONNECTION FACTORIES

This chapter describes how to use JMS connection factories in OSGi. Fundamentally, you do it by using:

```
org.osgi.framework.BundleContext.registerService(javax.jms.ConnectionFactory.class,
                                                 connectionFactoryObject,
                                                 properties);
org.osgi.framework.BundleContext.registerService(javax.jms.XAConnectionFactory.class,
                                                 xaConnectionFactoryObject,
                                                 properties);
```

There are two different methods to register such services:

- Publishing connection factories by using the **jms:create** Karaf console command. This is the *configuration method*.
- Publishing connection factories by using methods such as Blueprint, OSGi Declarative Services (SCR) or just a **BundleContext.registerService()** API call. This method requires a dedicated OSGi bundle that contains the code and/or metadata. This is the *deployment method*.

Details are in the following topics:

- Section 7.1, “About the OSGi JMS service”
- Section 7.2, “About the PAX-JMS configuration service”
- Section 7.3, “Using JMS console commands”
- Section 7.4, “Using encrypted configuration values”
- Section 7.5, “Using JMS connection pools”
- Section 7.6, “Deploying connection factories as artifacts”

7.1. ABOUT THE OSGI JMS SERVICE

The OSGi way of handling JDBC data sources is related to two interfaces:

- *standard* **org.osgi.service.jdbc.DataSourceFactory**
- *proprietary* **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory**

For JMS, consider these analogies:

- *proprietary* **org.ops4j.pax.jms.service.ConnectionFactoryFactory** with the same purpose as *standard* OSGi JDBC **org.osgi.service.jdbc.DataSourceFactory**
- *proprietary* **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** with the same purpose as *proprietary* pax-jdbc **org.ops4j.pax.jdbc.pool.common.PooledDataSourceFactory**

For the dedicated, broker-specific, **org.ops4j.pax.jms.service.ConnectionFactoryFactory** implementations, there are bundles such as:

- **mvn:org.ops4j.pax.jms/pax-jms-artemis/1.0.0**

- **mvn:org.ops4j.pax.jms/pax-jms-ibmmq/1.0.0**
- **mvn:org.ops4j.pax.jms/pax-jms-activemq/1.0.0**

These bundles register broker-specific **org.ops4j.pax.jms.service.ConnectionFactoryFactory** services that can return JMS *factories* such as **javax.jms.ConnectionFactory** and **javax.jms.XAConnectionFactory**. For example:

```
karaf@root()> feature:install pax-jms-artemis

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-config

OPS4J Pax JMS Config (248) provides:
-----------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 248
service.id = 328
service.pid = org.ops4j.connectionfactory
service.scope = singleton

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-artemis

OPS4J Pax JMS Artemis Support (247) provides:
-------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 247
service.id = 327
service.scope = singleton
type = artemis
```

7.2. ABOUT THE PAX-JMS CONFIGURATION SERVICE

The **mvn:org.ops4j.pax.jms/pax-jms-config/1.0.0** bundle provides a Managed Service Factory that does three things:

- Tracks **org.ops4j.pax.jms.service.ConnectionFactoryFactory** OSGi services to invoke its methods:

```
public ConnectionFactory createConnectionFactory(Map<String, Object> properties);

public XAConnectionFactory createXAConnectionFactory(Map<String, Object> properties);
```

- Tracks **org.ops4j.connectionfactory** *factory PIDs* to collect properties that are required by the above methods. If you create a *factory configuration* by using any method available for Configuration Admin service, for example, by creating a **${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg** file, you can perform the final step to expose a broker-specific connection factory.
- Tracks **javax.jms.ConnectionFactory** and **javax.jms.XAConnectionFactory** services to wrap them inside pooling JMS connection factories.

Details are in the following topics:

- Section 7.2.1, “Creating a connection factory for AMQ 7.1”

- Section 7.2.2, "Creating a connection factory for IBM MQ 8 or IBM MQ 9"
- Section 7.2.4, "Summary of handled properties"

7.2.1. Creating a connection factory for AMQ 7.1

Following is the detailed, *canonical*, step-by-step guide for creating a connection factor for an Artemis broker.

1. Install the Artemis driver by using the **pax-jms-artemis** feature and the **pax-jms-config** feature:

```
karaf@root()> feature:install pax-jms-artemis

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-config

OPS4J Pax JMS Config (248) provides:
-----------------------------------
objectClass = [org.osgi.service.cm.ManagedServiceFactory]
service.bundleid = 248
service.id = 328
service.pid = org.ops4j.connectionfactory
service.scope = singleton

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-artemis

OPS4J Pax JMS Artemis Support (247) provides:
--------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 247
service.id = 327
service.scope = singleton
type = artemis
```

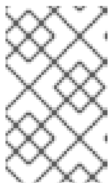
2. Create a *factory configuration*:

```
karaf@root()> config:edit --factory --alias artemis org.ops4j.connectionfactory
karaf@root()> config:property-set type artemis
karaf@root()> config:property-set osgi.jndi.service.name jms/artemis # "name" property may
be used too
karaf@root()> config:property-set connectionFactoryType ConnectionFactory # or
XAConnectionFactory
karaf@root()> config:property-set jms.url tcp://localhost:61616
karaf@root()> config:property-set jms.user admin
karaf@root()> config:property-set jms.password admin
karaf@root()> config:property-set jms.consumerMaxRate 1234
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:          org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
FactoryPid:   org.ops4j.connectionfactory
BundleLocation: ?
Properties:
  connectionFactoryType = ConnectionFactory
```

```
felix.fileinstall.filename = file:${karar.etc}/org.ops4j.connectionfactory-artemis.cfg
jms.consumerMaxRate = 1234
jms.password = admin
jms.url = tcp://localhost:61616
jms.user = admin
osgi.jndi.service.name = jms/artemis
service.factoryPid = org.ops4j.connectionfactory
service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
type = artemis
```

3. Check if **pax-jms-config** processed the configuration into the **javax.jms.ConnectionFactory** service:

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
 jms.consumerMaxRate = 1234
 jms.password = admin
 jms.url = tcp://localhost:61616
 jms.user = admin
 osgi.jndi.service.name = jms/artemis
 pax.jms.managed = true
 service.bundleid = 248
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 342
 service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Config (248)
```



NOTE

If you specify additional Artemis configuration, specifically **protocol=amqp**, the QPID JMS library would be used instead of the Artemis JMS client. The **amqp://** protocol has to be used then for **jms.url** property.

4. Test the connection.

You now have a broker-specific (no pooling yet) connection factory that you can inject where needed. For example, you can use Karaf commands from the **jms** feature:

```
karaf@root()> feature:install -v jms
Adding features: jms/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jms:connectionfactories
JMS Connection Factory
────────────────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
```

```
──────────┼───────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                          │ Content       │ Charset │ Type │ Correlation ID │ Delivery Mode │
Destination                │ Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
──────────────────────────────────────────────┼─────────────────┼───────
───┼───────┼───────────────────┼─────────────────┼─────────────────
──────────┼──────────────┼───────────┼───────────────┼──────────┼──────
───────────────────────────
ID:2b6ea56d-574d-11e8-971a-7ee9ecc029d4 │ Hello Artemis │ UTF-8  │      │              │ Persistent
│ ActiveMQQueue[DEV.QUEUE.1] │ Never      │ 4       │ false      │         │ Mon May 14 10:02:38
CEST 2018
```

The following listing shows what happens when you switch the protocol:

```
karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: ?
Properties:
   connectionFactoryType = ConnectionFactory
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
   jms.consumerMaxRate = 1234
   jms.password = fuse
   jms.url = tcp://localhost:61616
   jms.user = fuse
   osgi.jndi.service.name = jms/artemis
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.965d4eac-f5a7-4f65-ba1a-15caa4c72703
   type = artemis

karaf@root()> config:edit org.ops4j.connectionfactory.312eb09a-d686-4229-b7e1-2ea38a77bb0f
karaf@root()> config:property-set protocol amqp
karaf@root()> config:property-delete user
karaf@root()> config:property-set username admin # mind the difference between artemis-jms-client
and qpid-jms-client
karaf@root()> config:property-set jms.url amqp://localhost:61616
karaf@root()> config:update

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
──────────┼──────────────────
product  │ QpidJMS
version  │ 0.30.0.redhat-1

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID │ Content       │ Charset │ Type │ Correlation ID │ Delivery Mode │ Destination │
Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
────────────┼─────────────────┼──────────┼───────┼─────────────────┼──
───────────────┼──────────────┼─────────────┼──────────┼──────────
```

```
──────┼─────────────┼──────────────────────────────────
         │ Hello Artemis │ UTF-8  │     │           │ Persistent    │ DEV.QUEUE.1 │ Never      │ 4
│ false      │        │ Mon May 14 10:02:38 CEST 2018
```

7.2.2. Creating a connection factory for IBM MQ 8 or IBM MQ 9

This section shows how to connect to IBM MQ 8 and IBM MQ 9. Even though **pax-jms-ibmmq** installs the relevant **pax-jms** bundles, the IBM MQ driver is not installed due to licensing reasons.

1. Go to https://developer.ibm.com/messaging/mq-downloads/

2. Log in.

3. Click the version that you want to install, for example, click **IBM MQ 8.0 Client** or **IBM MQ 9.0 Client**.

4. In the page that appears, at the bottom, in the table of download versions, click the version that you want.

5. In the next page, select the latest version that has the suffix **IBM-MQ-Install-Java-All**. For example, download **8.0.0.10-WS-MQ-Install-Java-All** or **9.0.0.4-IBM-MQ-Install-Java-All**.

6. Extract the content of the downloaded JAR file.

7. Execute the **bundle:install** command. For example, if you extracted the content into your /**home**/**Downloads** directory, you would enter a command such as the following:

```
`bundle:install -s wrap:file:////home/Downloads/9.0.0.4-IBM-MQ-Install-Java-
All/ibmmq9/wmq/JavaSE/com.ibm.mq.allclient.jar`.
```

8. Create the connection factory as follows:

   a. Install **pax-jms-ibmmq**:

```
karaf@root()> feature:install pax-jms-ibmmq

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-ibmmq

OPS4J Pax JMS IBM MQ Support (239) provides:
--------------------------------------------
objectClass = [org.ops4j.pax.jms.service.ConnectionFactoryFactory]
service.bundleid = 239
service.id = 346
service.scope = singleton
type = ibmmq
```

   b. Create *factory configuration*:

```
karaf@root()> config:edit --factory --alias ibmmq org.ops4j.connectionfactory
karaf@root()> config:property-set type ibmmq
karaf@root()> config:property-set osgi.jndi.service.name jms/mq9 # "name" property may
be used too
karaf@root()> config:property-set connectionFactoryType ConnectionFactory # or
XAConnectionFactory
karaf@root()> config:property-set jms.queueManager FUSEQM
```

```
karaf@root()> config:property-set jms.hostName localhost
karaf@root()> config:property-set jms.port 1414
karaf@root()> config:property-set jms.transportType 1 #
com.ibm.msg.client.wmq.WMQConstants.WMQ_CM_CLIENT
karaf@root()> config:property-set jms.channel DEV.APP.SVRCONN
karaf@root()> config:property-set jms.CCSID 1208 #
com.ibm.msg.client.jms.JmsConstants.CCSID_UTF8
karaf@root()> config:update

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:          org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
FactoryPid:    org.ops4j.connectionfactory
BundleLocation: ?
Properties:
   connectionFactoryType = ConnectionFactory
   felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-ibmmq.cfg
   jms.CCSID = 1208
   jms.channel = DEV.APP.SVRCONN
   jms.hostName = localhost
   jms.port = 1414
   jms.queueManager = FUSEQM
   jms.transportType = 1
   osgi.jndi.service.name = jms/mq9
   service.factoryPid = org.ops4j.connectionfactory
   service.pid = org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
   type = ibmmq
```

c. Check if **pax-jms-config** processed the configuration into **javax.jms.ConnectionFactory** service:

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:/data/servers/7.9.0.fuse-790071-redhat-
00001/etc/org.ops4j.connectionfactory-ibmmq.cfg
 jms.CCSID = 1208
 jms.channel = DEV.APP.SVRCONN
 jms.hostName = localhost
 jms.port = 1414
 jms.queueManager = FUSEQM
 jms.transportType = 1
 osgi.jndi.service.name = jms/mq9
 pax.jms.managed = true
 service.bundleid = 237
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 347
 service.pid = org.ops4j.connectionfactory.eee4a757-a95d-46b8-b8b6-19aa3977d863
 service.scope = singleton
 type = ibmmq
Provided by :
 OPS4J Pax JMS Config (237)
```

d. Test the connection:

```
karaf@root()> feature:install -v jms
Adding features: jms/[4.2.0.fuse-000237-redhat-1,4.2.0.fuse-000237-redhat-1]
...
karaf@root()> jms:connectionfactories
JMS Connection Factory
───────────────────────
jms/mq9

karaf@root()> jms:info -u app -p fuse jms/mq9
Property │ Value
─────────┼──────────────────────
product  │ IBM MQ JMS Provider
version  │ 8.0.0.0

karaf@root()> jms:send -u app -p fuse jms/mq9 DEV.QUEUE.1 "Hello IBM MQ 9"

karaf@root()> jms:browse -u app -p fuse jms/mq9 DEV.QUEUE.1
Message ID                               │ Content              │ Charset │ Type │
Correlation ID │ Delivery Mode │ Destination          │ Expiration │ Priority │ Redelivered
│ ReplyTo │ Timestamp
──────────────────────────────────────────────────────────────────────────────────┼──
─────────────────────────────────────────────┼──────────────┼──────────┼───────────
──────────┼─────────────────────┼───────────────────────────────────┼─────────────
────┼───────────┼──────────────────────┼─────────────┼──────────────────────────
──────────────
ID:414d512046555345514d202020202020c940f95a038b3220 │ Hello IBM MQ 9
│ UTF-8 │     │         │ Persistent │ queue:///DEV.QUEUE.1 │ Never      │ 4
│ false      │       │ Mon May 14 10:17:01 CEST 2018
```

You can also check if the message was sent from IBM MQ Explorer or from the web console.

7.2.3. Using JBoss A-MQ 6.3 Client in Fuse on Apache Karaf

You can download Fuse **quickstarts** from the Fuse Software Downloads page.

Extract the contents of the quickstarts zip file to a local folder, for example a folder named **quickstarts**.

You can build and install the **quickstarts/camel/camel-jms** example as an OSGi bundle. This bundle contains a Blueprint XML definition of a Camel route that sends messages to an JBoss A-MQ 6.3 JMS queue. The procedure for creating a connection factory for JBoss A-MQ 6.3 broker is as follows.

7.2.3.1. Prerequisites

- You have installed Maven 3.3.1 or higher.
- You have Red Hat Fuse installed on your machine.
- You have JBoss A-MQ Broker 6.3 installed on your machine.
- You have downloaded and extracted the Fuse on Karaf quickstarts zip file from the customer portal.

7.2.3.2. Procedure

1. Navigate to **quickstarts/camel/camel-jms/src/main/resources/OSGI-INF/blueprint/** directory.

2. Replace the following bean with id="jms" from the **camel-context.xml** file:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
       <reference interface="javax.jms.ConnectionFactory" />
    </property>
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

   With the following section to instantiate the JBoss A-MQ 6.3 connection factory:

```
 <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
        <property name="connectionFactory" ref="activemqConnectionFactory"/>
        <property name="transactionManager" ref="transactionManager"/>
 </bean>
 <bean id="activemqConnectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
  <property name="userName" value="admin"/>
  <property name="password" value="admin"/>
 </bean>
```

   The JBoss A-MQ 6.3 connection factory is configured to connect to a broker listening at **tcp://localhost:61616**. By default JBoss A-MQ uses the IP port value **61616**. The connection factory is also configured to use the userName/password credentials, admin/admin. Make sure that this user is enable in your broker cofiguration (or you can customize these settings here to match your broker configuration).

3. Save the **camel-context.xml** file.

4. Build the **camel-jms** quickstart:

```
$ cd quickstarts/camel/camel-jms
$ mvn install
```

5. After the quickstart is successfully installed, navigate to **$FUSE_HOME/** directory and run the following command to start the Fuse on Apache Karaf server:

```
$ ./bin/fuse
```

6. On the Fuse on Apache Karaf instance install **activemq-client** feature and **camel-jms** feature:

```
karaf@root()> feature:install activemq-client
karaf@root()> feature:install camel-jms
```

7. Install the **camel-jms** quickstart bundle:

```
karaf@root()> install -s mvn:org.jboss.fuse.quickstarts/camel-jms/{$fuseversion}
```

   Where replace **{$fuseversion}** with the actual version of the Maven artifact that you just built (consult the camel-jms quickstart README file).

8. Start the **JBoss A-MQ 6.3** broker (you need an installation of JBoss A-MQ 6.3 for this). Open another terminal window and navigate to JBOSS_AMQ_63_INSTALLDIR directory:

```
$ cd JBOSS_AMQ_63_INSTALLDIR
$ ./bin/amq
```

9. As soon as the Camel routes have started, you can see a directory **work/jms/input** in your Fuse installation. Copy the files you find in this quickstart's **src/main/data directory** to the newly created **work/jms/input** directory.

10. Wait a few moments and you will find the same files organized by country under the **work/jms/output** directory:

```
order1.xml, order2.xml and order4.xml in work/jms/output/others
order3.xml and order5.xml in work/jms/output/us
order6.xml in work/jms/output/fr
```

11. Use **log:display** to check out the business logging:

```
Receiving order order1.xml

Sending order order1.xml to another country

Done processing order1.xml
```

7.2.4. Summary of handled properties

Properties from the Configuration Admin *factory PID* are passed to the relevant **org.ops4j.pax.jms.service.ConnectionFactoryFactory** implementation.

- ActiveMQ
  org.ops4j.pax.jms.activemq.ActiveMQConnectionFactoryFactory (JMS 1.1 only)

  Properties that are passed to the **org.apache.activemq.ActiveMQConnectionFactory.buildFromMap()** method

- Artemis
  org.ops4j.pax.jms.artemis.ArtemisConnectionFactoryFactory

  If **protocol=amqp**, properties are passed to the **org.apache.qpid.jms.util.PropertyUtil.setProperties()** method to configure the **org.apache.qpid.jms.JmsConnectionFactory** instance.

  Otherwise, **org.apache.activemq.artemis.utils.uri.BeanSupport.setData()** is called for the **org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory** instance.

- IBM MQ
  org.ops4j.pax.jms.ibmmq.MQConnectionFactoryFactory

  Bean properties of **com.ibm.mq.jms.MQConnectionFactory** or **com.ibm.mq.jms.MQXAConnectionFactory** are handled.

7.3. USING JMS CONSOLE COMMANDS

Apache Karaf provides the **jms** feature, which includes shell commands in the **jms:*** scope. You already saw some examples of using these commands to check the manually configured connection factories. There are also commands that hide the need to create Configuration Admin configurations.

Starting with a fresh instance of Fuse, you can register a broker-specific connection factory. The following listing shows install of the **jms** feature from Karaf and installation of **pax-jms-artemis** from **pax-jms**:

```
karaf@root()> feature:install jms pax-jms-artemis

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
karaf@root()> service:list javax.jms.ConnectionFactory # should be empty

karaf@root()> service:list org.ops4j.pax.jms.service.ConnectionFactoryFactory
[org.ops4j.pax.jms.service.ConnectionFactoryFactory]
----------------------------------------------------
 service.bundleid = 250
 service.id = 326
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Artemis Support (250)
```

The following listing shows how to create and check an Artemis connection factory:

```
karaf@root()> jms:create -t artemis -u admin -p admin --url tcp://localhost:61616 artemis

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                              │ Content       │ Charset │ Type │ Correlation ID │ Delivery Mode │
Destination                 │ Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
──────────────────────────────────────────────────────────────────────────┼──────────────────────────┼────────
────┼───────┼────────────────────────────┼────────────────────────────┼─────────────────────────────────
──────────────┼────────────────────┼─────────────────┼──────────────────────┼───────────────┼────────
─────────────────────────────────────────
ID:7a944470-574f-11e8-918e-7ee9ecc029d4 │ Hello Artemis │ UTF-8   │      │                │ Persistent
│ ActiveMQQueue[DEV.QUEUE.1] │ Never      │ 4        │ false       │         │ Mon May 14 10:19:10
CEST 2018

karaf@root()> config:list '(service.factoryPid=org.ops4j.connectionfactory)'
----------------------------------------------------------------
Pid:            org.ops4j.connectionfactory.9184db6f-cb5f-4fd7-b5d7-a217090473ad
FactoryPid:     org.ops4j.connectionfactory
BundleLocation: mvn:org.ops4j.pax.jms/pax-jms-config/1.0.0
Properties:
```

```
name = artemis
osgi.jndi.service.name = jms/artemis
password = admin
service.factoryPid = org.ops4j.connectionfactory
service.pid = org.ops4j.connectionfactory.9184db6f-cb5f-4fd7-b5d7-a217090473ad
type = artemis
url = tcp://localhost:61616
user = admin
```

As you can see, the **org.ops4j.connectionfactory** factory PID is created for you. However it is not automatically stored in **${karaf.etc}**, which is possible with **config:update**. It is not possible to specify other properties, but you can add them later.

7.4. USING ENCRYPTED CONFIGURATION VALUES

As with the **pax-jdbc-config** bundle, you can use Jasypt to encrypt properties.

If there is any **org.jasypt.encryption.StringEncryptor** service that is registered in OSGi with any **alias** service property, you can reference it in a connection factory *factory PID* and use encrypted passwords. Following is an example:

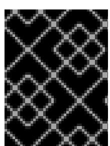
```
felix.fileinstall.filename = */etc/org.ops4j.connectionfactory-artemis.cfg
name = artemis
type = artemis
decryptor = my-jasypt-decryptor
url = tcp://localhost:61616
user = fuse
password = ENC(<encrypted-password>)
```

The service filter used to find the decryptor service is **(&(objectClass=org.jasypt.encryption.StringEncryptor)(alias=<alias>))**, where **<alias>** is the value of the **decryptor** property from the connection factory configuration *factory PID*.

7.5. USING JMS CONNECTION POOLS

This section discusses JMS connection/session pooling options. There are fewer choices than there are for JDBC. The information is organized into the following topics:

- Section 7.5.1, “Introduction to using JMS connection pools”
- Section 7.5.2, “Using the pax-jms-pool-pooledjms connection pool module”
- Section 7.5.3, “Using the pax-jms-pool-narayana connection pool module”
- Section 7.5.4, “Using the pax-jms-pool-transx connection pool module”



IMPORTANT

To use XA recovery, you should use the **pax-jms-pool-transx** or **pax-jms-pool-narayana** connection pool module.

7.5.1. Introduction to using JMS connection pools

So far, you have registered a broker-specific connection **factory**. Because a *connection factory* itself is a factory for connection factories, the **org.ops4j.pax.jms.service.ConnectionFactoryFactory** service may be treated as a *meta factory*. It should be able to produce two kinds of connection factories:

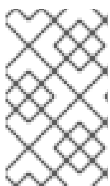
- **javax.jms.ConnectionFactory**
- **javax.jms.XAConnectionFactory**

The **pax-jms-pool-*** bundles work smoothly with the **org.ops4j.pax.jms.service.ConnectionFactoryFactory** service. These bundles provide implementations of **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** that can be used to create pooled connection factories by using a set of properties and the original **org.ops4j.pax.jms.service.ConnectionFactoryFactory** in a kind of *wrapper* way. For example:

```
public interface PooledConnectionFactoryFactory {

    ConnectionFactory create(ConnectionFactoryFactory cff, Map<String, Object> props);

}
```

The following table shows which bundles register pooled connection factory factories. In the table, **o.o.p.j.p** represents **org.ops4j.pax.jms.pool**.

Bundle	PooledConnectionFactoryFactory	Pool Key
pax-jms-pool-pooledjms	**o.o.p.j.p.pooledjms.PooledJms(XA)PooledConnectionFactoryFactory**	**pooledjms**
pax-jms-pool-narayana	**o.o.p.j.p.narayana.PooledJms(XA)PooledConnectionFactoryFactory**	**narayana**
pax-jms-pool-transx	**o.o.p.j.p.transx.Transx(XA)PooledConnectionFactoryFactory**	**transx**



NOTE

The **pax-jms-pool-narayana** factory is called **PooledJms(XA)PooledConnectionFactoryFactory** because it is based on the **pooled-jms** library. It adds integration with the Narayana transaction manager for XA recovery.

The above bundles install only connection factory factories. The bundles to not install the connection factories themselves. Consequently, something is needed that calls the **javax.jms.ConnectionFactory org.ops4j.pax.jms.service.PooledConnectionFactoryFactory.create()** method.

7.5.2. Using the pax-jms-pool-pooledjms connection pool module

An understanding of how to use the **pax-jms-pool-pooledjms** bundle helps you use not only the **pax-jms-pool-pooledjms** bundle, but also the **pax-jms-pool-narayna** bundle, which does almost everything as **pax-jms-pool-pooledjms**.

The pax-jms-config bundle tracks the following:

- **org.ops4j.pax.jms.service.ConnectionFactoryFactory** services

- **org.ops4j.connectionfactory** *factory PIDs*
- Instances of **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** that are registered by one of **pax-jms-pool-*** bundles.

If a *factory configuration* contains a **pool** property, the ultimate connection factory registered by the **pax-jms-config** bundle is the broker-specific connection factory. If **pool=pooledjms** then the connection factory is wrapped inside one of the following:

- **org.messaginghub.pooled.jms.JmsPoolConnectionFactory** (**xa=false**)
- **org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory** (**xa=true**)

Besides the **pool** property (and the Boolean **xa** property, which selects one of non-xa/xa connection factories), the **org.ops4j.connectionfactory** *factory PID* may contain properties that are prefixed with **pool.**.

For the **pooled-jms** library, these prefixed properties are used (after removing the prefix) to configure an instance of:

- **org.messaginghub.pooled.jms.JmsPoolConnectionFactory**, or
- **org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory**

The following listing is a realistic configuration of a **pooled-jms** pool (**org.ops4j.connectionfactory-artemis** *factory PID*) that is using a convenient syntax with **jms.**-prefixed properties:

```
# configuration for pax-jms-config to choose and configure specific
org.ops4j.pax.jms.service.ConnectionFactoryFactory
name = jms/artemis
connectionFactoryType = ConnectionFactory
jms.url = tcp://localhost:61616
jms.user = fuse
jms.password = fuse
# org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory specific coniguration
jms.callTimeout = 12000
# ...

# hints for pax-jms-config to use selected org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
pool = pooledjms
xa = false

# pooled-jms specific configuration of org.messaginghub.pooled.jms.JmsPoolConnectionFactory
pool.idleTimeout = 10
pool.maxConnections = 100
pool.blockIfSessionPoolIsFull = true
# ...
```

In the above configuration, **pool** and **xa** keys are *hints* (service filter properties) to choose one of the registered **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** services. In the case of the **pooled-jms** library it is:

```
karaf@root()> feature:install pax-jms-pool-pooledjms

karaf@root()> bundle:services -p org.ops4j.pax.jms.pax-jms-pool-pooledjms
```

```
OPS4J Pax JMS MessagingHub JMS Pool implementation (252) provides:
----------------------------------------------------------------
objectClass = [org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
pool = pooledjms
service.bundleid = 252
service.id = 331
service.scope = singleton
xa = false
-----
objectClass = [org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
pool = pooledjms
service.bundleid = 252
service.id = 335
service.scope = singleton
xa = true
```

Following is a complete example of the steps for creating and configuring a connection pool:

1. Install the required features:

```
karaf@root()> feature:install -v pax-jms-pool-pooledjms pax-jms-artemis
Adding features: pax-jms-pool-pooledjms/[1.0.0,1.0.0]
...
```

2. Install the **jms** feature:

```
karaf@root()> feature:install jms

karaf@root()> service:list org.ops4j.pax.jms.service.ConnectionFactoryFactory
[org.ops4j.pax.jms.service.ConnectionFactoryFactory]
----------------------------------------------------
 service.bundleid = 249
 service.id = 327
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Artemis Support (249)

karaf@root()> service:list org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
[org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
----------------------------------------------------------
 pool = pooledjms
 service.bundleid = 251
 service.id = 328
 service.scope = singleton
 xa = false
Provided by :
 OPS4J Pax JMS MessagingHub JMS Pool implementation (251)

[org.ops4j.pax.jms.service.PooledConnectionFactoryFactory]
----------------------------------------------------------
 pool = pooledjms
 service.bundleid = 251
 service.id = 333
 service.scope = singleton
```

```
 xa = true
Provided by :
 OPS4J Pax JMS MessagingHub JMS Pool implementation (251)
```

3. Create a *factory configuration*:

```
karaf@root()> config:edit --factory --alias artemis org.ops4j.connectionfactory
karaf@root()> config:property-set connectionFactoryType ConnectionFactory
karaf@root()> config:property-set osgi.jndi.service.name jms/artemis
karaf@root()> config:property-set type artemis
karaf@root()> config:property-set protocol amqp # so we switch to
org.apache.qpid.jms.JmsConnectionFactory
karaf@root()> config:property-set jms.url amqp://localhost:61616
karaf@root()> config:property-set jms.username admin
karaf@root()> config:property-set jms.password admin
karaf@root()> config:property-set pool pooledjms
karaf@root()> config:property-set xa false
karaf@root()> config:property-set pool.idleTimeout 10
karaf@root()> config:property-set pool.maxConnections 123
karaf@root()> config:property-set pool.blockIfSessionPoolIsFull true
karaf@root()> config:update
```

4. Check if **pax-jms-config** processed the configuration into **javax.jms.ConnectionFactory** service:

```
karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 connectionFactoryType = ConnectionFactory
 felix.fileinstall.filename = file:${karaf.etc}/org.ops4j.connectionfactory-artemis.cfg
 jms.password = admin
 jms.url = amqp://localhost:61616
 jms.username = admin
 osgi.jndi.service.name = jms/artemis
 pax.jms.managed = true
 pool.blockIfSessionPoolIsFull = true
 pool.idleTimeout = 10
 pool.maxConnections = 123
 protocol = amqp
 service.bundleid = 250
 service.factoryPid = org.ops4j.connectionfactory
 service.id = 347
 service.pid = org.ops4j.connectionfactory.fc1b9e85-91b4-421b-aa16-1151b0f836f9
 service.scope = singleton
 type = artemis
Provided by :
 OPS4J Pax JMS Config (250)
```

5. Use the connection factory:

```
karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
jms/artemis
```

```
karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────
product  │ QpidJMS
version  │ 0.30.0.redhat-1

karaf@root()> jms:send -u admin -p admin jms/artemis DEV.QUEUE.1 "Hello Artemis"

karaf@root()> jms:browse -u admin -p admin jms/artemis DEV.QUEUE.1
Message ID                            │ Content      │ Charset │ Type │ Correlation ID │
Delivery Mode │ Destination │ Expiration │ Priority │ Redelivered │ ReplyTo │ Timestamp
─────────────────────────────────────────────────────────────────────────────────────┼────────
───────┼──────────┼───────┼──────────────────┼─────────┼────────
───────────────┼──────────────┼───────────┼────────────────┼──────────
──┼───────────────────────────────────────
ID:64842f99-5cb2-4850-9e88-f50506d49d20:1:1:1-1 │ Hello Artemis │ UTF-8   │      │
│ Persistent   │ DEV.QUEUE.1 │ Never      │ 4        │ false       │         │ Mon May 14
12:47:13 CEST 2018
```

7.5.3. Using the pax-jms-pool-narayana connection pool module

The **pax-jms-pool-narayna** module does almost everything as **pax-jms-pool-pooledjms**. It installs the pooled-jms-specific **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory**, both for XA and non-XA scenarios. The **only** difference is that in an XA scenario, there is an additional integration point. The **org.jboss.tm.XAResourceRecovery** OSGi service is registered to be picked up by **com.arjuna.ats.arjuna.recovery.RecoveryManager**.

7.5.4. Using the pax-jms-pool-transx connection pool module

The **pax-jms-pool-transx** module provides an implementation of **org.ops4j.pax.jms.service.PooledConnectionFactoryFactory** services that is based on the **pax-transx-jms** bundle. The **pax-transx-jms** bundle creates **javax.jms.ConnectionFactory** pools by using the **org.ops4j.pax.transx.jms.ManagedConnectionFactoryBuilder** facility. This is a JCA (Java™ Connector Architecture) solution that is discussed in Section 8.3, “About the pax-transx project”.

7.6. DEPLOYING CONNECTION FACTORIES AS ARTIFACTS

This topic discusses real-world recommendations.

In the *deployment method*, **javax.jms.ConnectionFactory** services are registered directly by application code. Usually, this code is inside a Blueprint container. Blueprint XML may be part of an ordinary OSGi bundle, installable by using **mvn:** URI, and stored in a Maven repository (local or remote). It is easier to version-control such bundles as compared to Configuration Admin configurations.

The **pax-jms-config** version 1.0.0 bundle adds a *deployment method* for connection factory configuration. An application developer registers the **javax.jms.(XA)ConnectionFactory** service (usually by using Bluerpint XML) and specifies service properties. Then **pax-jms-config** detects the registered, broker-specific connection factory and (using service properties) wraps the service inside a generic, non broker-specific, connection pool.

Following are three *deployment methods* that use Blueprint XML.

- Section 7.6.1, “Manual deployment of connection factories”
- Section 7.6.2, “Factory deployment of connection factories”

- Section 7.6.3, “Mixed deployment of connection factories”

7.6.1. Manual deployment of connection factories

In this method, the **pax-jms-config** bundle is not needed. Application code is responsible for registration of both broker-specific and generic connection pools.

```
<!--
   Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
   <argument value="tcp://localhost:61616" />
   <property name="callTimeout" value="2000" />
   <property name="initialConnectAttempts" value="3" />
</bean>

<!--
   Fuse exports this service from fuse-pax-transx-tm-narayana bundle.
-->
<reference id="tm" interface="javax.transaction.TransactionManager" />

<!--
   Non broker-specific, generic, pooling, enlisting javax.jms.ConnectionFactory
-->
<bean id="pool" class="org.messaginghub.pooled.jms.JmsPoolXAConnectionFactory">
   <property name="connectionFactory" ref="artemis" />
   <property name="transactionManager" ref="tm" />
   <property name="maxConnections" value="10" />
   <property name="idleTimeout" value="10000" />
</bean>

<!--
   Expose connection factory for use by application code (such as Camel, Spring, ...)
-->
<service interface="javax.jms.ConnectionFactory" ref="pool">
   <service-properties>
      <!-- Giving connection factory a name using one of these properties makes identification easier
in jms:connectionfactories: -->
      <entry key="osgi.jndi.service.name" value="jms/artemis" />
      <!--<entry key="name" value="jms/artemis" />-->
      <!-- Without any of the above, name will fall back to "service.id" -->
   </service-properties>
</service>
```

Here are the shell commands that show how it should be used:

```
karaf@root()> feature:install artemis-core-client artemis-jms-client
karaf@root()> install -s mvn:org.apache.commons/commons-pool2/2.5.0
Bundle ID: 244
karaf@root()> install -s mvn:org.messaginghub/pooled-jms/0.3.0
Bundle ID: 245
karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-manual.xml
Bundle ID: 246

karaf@root()> bundle:services -p 246
```

```
Bundle 246 provides:
--------------------
objectClass = [javax.jms.ConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = pool
service.bundleid = 246
service.id = 340
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-manual.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 246
service.id = 341
service.scope = singleton

karaf@root()> feature:install jms

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

As shown in the above listing, the Blueprint bundle exports the **javax.jms.ConnectionFactory** service, which is a generic, non broker-specific, connection pool. The broker-specific **javax.jms.XAConnectionFactory** is **not** registered as an OSGi service, because Blueprint XML does not have an explicit **<service ref="artemis">** declaration.

7.6.2. Factory deployment of connection factories

This method shows the use of **pax-jms-config** in a *canonical* way. This is a bit different than the method that was recommended for Fuse 6.x, where the requirement was to specify pooling configuration as service properties.

Here is the Blueprint XML example:

```
<!--
   A broker-specific org.ops4j.pax.jms.service.ConnectionFactoryFactory that can create
(XA)ConnectionFactory
   using properties. It is registered by pax-jms-* bundles
-->
<reference id="connectionFactoryFactory"
        interface="org.ops4j.pax.jms.service.ConnectionFactoryFactory"
        filter="(type=artemis)" />

<!--
   Non broker-specific org.ops4j.pax.jms.service.PooledConnectionFactoryFactory that can create
   pooled connection factories with the help of org.ops4j.pax.jms.service.ConnectionFactoryFactory
```

```
    For example, pax-jms-pool-pooledjms bundle registers
org.ops4j.pax.jms.service.PooledConnectionFactoryFactory
    with these properties:
     - pool = pooledjms
     - xa = true|false (both are registered)
-->
<reference id="pooledConnectionFactoryFactory"
        interface="org.ops4j.pax.jms.service.PooledConnectionFactoryFactory"
        filter="(&amp;(pool=pooledjms)(xa=true))" />

<!--
    When using XA connection factories, javax.transaction.TransactionManager service is not needed
here.
    It is used internally by xa-aware pooledConnectionFactoryFactory.
-->
<!--<reference id="tm" interface="javax.transaction.TransactionManager" />-->

<!--
    Finally, use both factories to expose the pooled, xa-aware, connection factory.
-->
<bean id="pool" factory-ref="pooledConnectionFactoryFactory" factory-method="create">
    <argument ref="connectionFactoryFactory" />
    <argument>
        <props>
            <!--
                Properties needed by artemis-specific org.ops4j.pax.jms.service.ConnectionFactoryFactory
            -->
            <prop key="jms.url" value="tcp://localhost:61616" />
            <prop key="jms.callTimeout" value="2000" />
            <prop key="jms.initialConnectAttempts" value="3" />
            <!-- Properties needed by pooled-jms-specific
org.ops4j.pax.jms.service.PooledConnectionFactoryFactory -->
            <prop key="pool.maxConnections" value="10" />
            <prop key="pool.idleTimeout" value="10000" />
        </props>
    </argument>
</bean>

<!--
    Expose connection factory for use by application code (such as Camel, Spring, ...)
-->
<service interface="javax.jms.ConnectionFactory" ref="pool">
    <service-properties>
        <!-- Giving connection factory a name using one of these properties makes identification easier
in jms:connectionfactories: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Without any of the above, name will fall back to "service.id" -->
    </service-properties>
</service>
```

The previous example uses *factory* beans that create connection factories by using connection factory factories (…). There is no need for an explicit reference to the **javax.transaction.TransactionManager** service, as this is tracked internally by the XA-aware **PooledConnectionFactoryFactory**.

Here is how it looks in a Fuse/Karaf shell:

```
karaf@root()> feature:install jms pax-jms-artemis pax-jms-pool-pooledjms

karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-pax-jms-
factory-pooledjms.xml
Bundle ID: 253
karaf@root()> bundle:services -p 253

Bundle 253 provides:
--------------------
objectClass = [javax.jms.ConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = pool
service.bundleid = 253
service.id = 347
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-pax-jms-factory-pooledjms.xml
osgi.blueprint.container.version = 0.0.0
service.bundleid = 253
service.id = 348
service.scope = singleton

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

As shown in the above listing, the Blueprint bundle exports the **javax.jms.ConnectionFactory** service, which is a generic, non broker-specific, connection pool. The broker-specific **javax.jms.XAConnectionFactory** is **not** registered as an OSGi service because Blueprint XML does not have an explicit **<service ref="artemis">** declaration.

7.6.3. Mixed deployment of connection factories

The **pax-jms-config** 1.0.0 bundle adds another way of *wrapping* broker-specific connection factories within pooling connection factories by using service properties. This method matches the way it used to work in Fuse 6.x.

Here is the Blueprint XML example:

```
<!--
   Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
   <argument value="tcp://localhost:61616" />
   <property name="callTimeout" value="2000" />
```

```
    <property name="initialConnectAttempts" value="3" />
</bean>

<!--
    Expose broker-specific connection factory with service properties.
    No need to expose pooling, enlisting, non broker-specific javax.jms.XAConnectionFactory. It will be
registered
    automatically by pax-jms-config with the same properties as this <service>, but with a higher
service.ranking
-->
<service id="pool" ref="artemis" interface="javax.jms.XAConnectionFactory">
    <service-properties>
        <!-- "pool" key is needed for pax-jms-config to wrap broker-specific connection factory inside
connection pool -->
        <entry key="pool" value="pooledjms" />
        <!-- <service>/@id attribute does not propagate, but name of the connection factory is required
using one of: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!-- or: -->
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Other properties, that normally by e.g., pax-jms-pool-pooledjms -->
        <entry key="pool.maxConnections" value="10" />
        <entry key="pool.idleTimeout" value="10000" />
    </service-properties>
</service>
```

In the above example, you can see the manual register of only the broker-specific connection factory. The **pool=pooledjms** service property is a hint for the connection factory tracker that is managed by the **pax-jms-config** bundle. Connection factory services with this service property are wrapped within a pooling connection factory, in this example, **pax-jms-pool-pooledjms**.

Here is how it looks in a Fuse/Karaf shell:

```
karaf@root()> feature:install jms pax-jms-config pax-jms-artemis pax-jms-pool-pooledjms

karaf@root()> install -s blueprint:file://$PQ_HOME/message-brokers/blueprints/artemis-pax-jms-
discovery.xml
Bundle ID: 254

karaf@root()> bundle:services -p 254

Bundle 254 provides:
--------------------
objectClass = [javax.jms.XAConnectionFactory]
osgi.jndi.service.name = jms/artemis
osgi.service.blueprint.compname = artemis
pool = pooledjms
pool.idleTimeout = 10000
pool.maxConnections = 10
service.bundleid = 254
service.id = 349
service.scope = bundle
-----
objectClass = [org.osgi.service.blueprint.container.BlueprintContainer]
osgi.blueprint.container.symbolicname = artemis-pax-jms-discovery.xml
osgi.blueprint.container.version = 0.0.0
```

```
service.bundleid = 254
service.id = 351
service.scope = singleton

karaf@root()> service:list javax.jms.XAConnectionFactory
[javax.jms.XAConnectionFactory]
-------------------------------
 osgi.jndi.service.name = jms/artemis
 osgi.service.blueprint.compname = artemis
 pool = pooledjms
 pool.idleTimeout = 10000
 pool.maxConnections = 10
 service.bundleid = 254
 service.id = 349
 service.scope = bundle
Provided by :
 Bundle 254
Used by:
 OPS4J Pax JMS Config (251)

karaf@root()> service:list javax.jms.ConnectionFactory
[javax.jms.ConnectionFactory]
-----------------------------
 osgi.jndi.service.name = jms/artemis
 osgi.service.blueprint.compname = artemis
 pax.jms.managed = true
 pax.jms.service.id.ref = 349
 pool.idleTimeout = 10000
 pool.maxConnections = 10
 service.bundleid = 251
 service.id = 350
 service.ranking = 1000
 service.scope = singleton
Provided by :
 OPS4J Pax JMS Config (251)

karaf@root()> jms:connectionfactories
JMS Connection Factory
──────────────────────
jms/artemis

karaf@root()> jms:info -u admin -p admin jms/artemis
Property │ Value
─────────┼──────────────────────────
product  │ ActiveMQ
version  │ 2.4.0.amq-711002-redhat-1
```

In the previous example, **jms:connectionfactories** shows only one service, because this command removes duplicate names. Two services were presented by **jdbc:ds-list** in the mixed deployment of data sources.

javax.jms.XAConnectionFactory is registered from the Blueprint bundle and it has the **pool = pooledjms** property declared.

javax.jms.ConnectionFactory is registered from the **pax-jms-config** bundle and:

- It does not have the **pool = pooledjms** property. It was removed when registering the wrapper connection factory.
- It has the **service.ranking = 1000** property, so it is always the preferred version when, for example, looking for a connection factory by name.
- It has the **pax.jms.managed = true** property, so it is not tried to be wrapped again.
- It has the **pax.jms.service.id.ref = 349** property, which indicates the original connection factory service that is wrapped inside the connection pool.

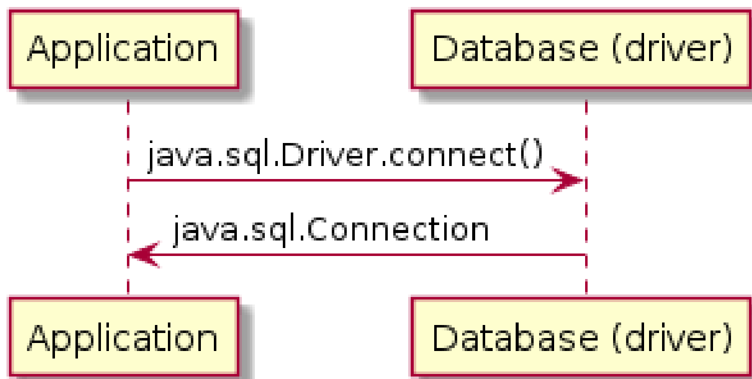
CHAPTER 8. ABOUT JAVA CONNECTOR ARCHITECTURE

The JCA specification was created to (among other things) *generalize* the scenarios that have these three participants:

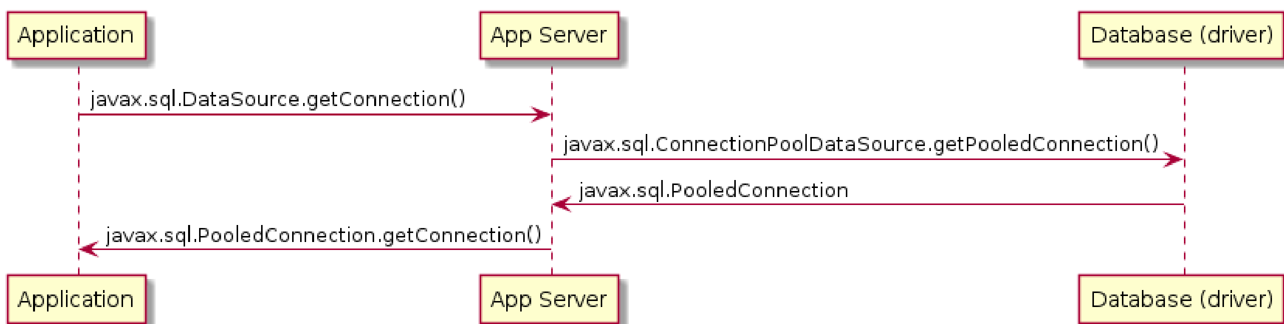
- An external system such as a database or *generally* an EIS system
- A JavaEE application server
- A deployed application

8.1. SIMPLE JDBC ANALOGY

In the simplest scenario, where there is only an application and database, you have:



Adding an application server that exposes **javax.sql.DataSource**, you have the following (without recalling different aspects of data sources like XA):



8.2. OVERVIEW OF USING JCA

JCA generalizes the concept of a *database driver* by adding two-way communication between the *driver* and the application server. The driver becomes a *resource adapter* that is represented by **javax.resource.spi.ResourceAdapter**.

There are two important interfaces:

- **javax.resource.spi.ManagedConnectionFactory** implemented by a resource adapter.
- **javax.resource.spi.ConnectionManager** implemented by an application server.

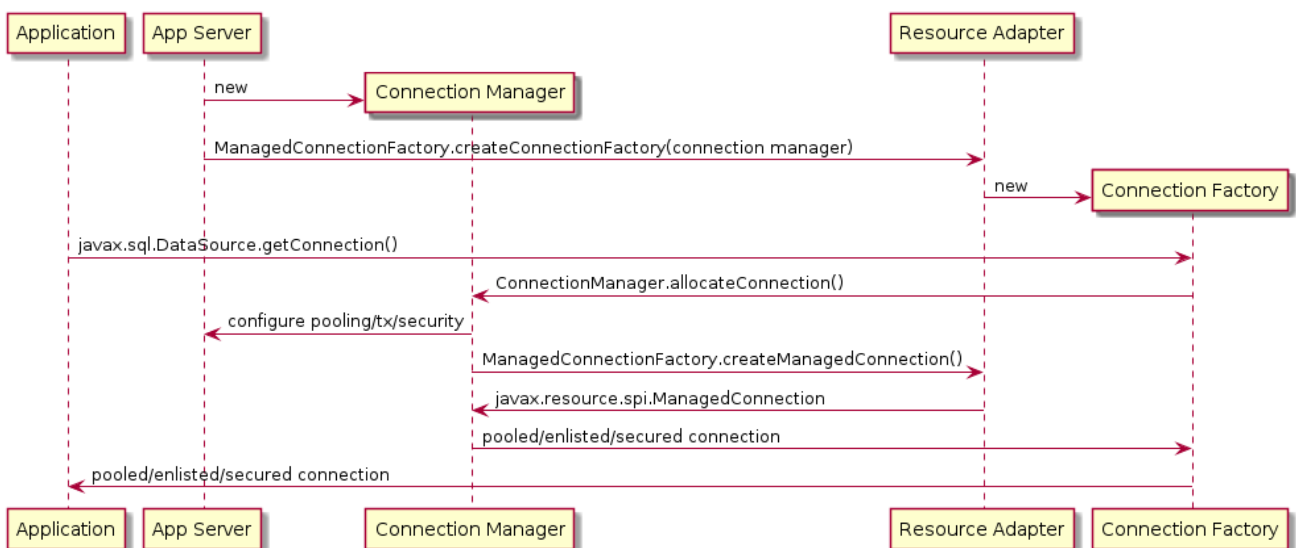
The **ManagedConnectionFactory** interface serves two purposes:

- The **Object createConnectionFactory(ConnectionManager cxManager)** method may be used to produce a *connection factory* for a given EIS (or database or message broker) that can be used by application code. The returned **Object** may be:
  - A generic **javax.resource.cci.ConnectionFactory** (not described here further, see JCA 1.6, chapter 17: *Common Client Interface*)
  - EIS specific connection factory like the well-known **javax.sql.DataSource** or **javax.jms.ConnectionFactory**. That is the type of *connection factory* that is used by the **pax-transx-jdbc** and **pax-transx-jms** bundles.
- The **javax.resource.spi.ManagedConnection ManagedConnectionFactory.createManagedConnection()** method used by an *application server*, creates actual physical connections to the EIS/database/broker.

ConnectionManager is implemented by an *application server* and used by a *resource adapter*. It is the *application server* that first performs QoS operations (pooling, security, transaction management) and finally delegates to the **ManagedConnectionFactory** of the *resource adapter* to create **ManagedConnection** instances. The flow looks like this:

1. Application code uses *connection factory* created and exposed by *application server* using object returned from **ManagedConnectionFactory.createConnectionFactory()**. It may be generic CCI interface or e.g., **javax.sql.DataSource**.
2. this *connection factory* doesn't create *connections* on its own, instead it delegates to **ConnectionManager.allocateConnection()** passing *resource adapter*-specific **ManagedConnectionFactory**
3. **ConnectionManager** implemented by *application server* creates *supporting objects*, manages transactions, pooling, etc. and eventually obtains *physical (managed) connection* from passed **ManagedConnectionFactory**.
4. Application code gets *connection* which is usually a wrapper/proxy created by *application server* which eventually delegates to *resource adapter*'s specific *physical connection*.

Following is the diagram, where *application server* created non-CCI *connection factory* which is EIS-specific. Simply - access to EIS (here: database) is done using **javax.sql.DataSource** interface, the driver's task is to provide *physical connection*, while *application server* will wrapp it inside (typically) a proxy that does pooling/enlisting/security.



8.3. ABOUT THE PAX-TRANSX PROJECT

The **pax-transx** project provides support for JTA/JTS transaction management in OSGi, as well as resource pooling for JDBC and JMS. It closes the gap between **pax-jdbc** and **pax-jms**.

- **pax-jdbc** adds configuration options and discovery for **javax.sql.(XA)ConnectionFactory** services and ships some JDBC pooling implementations
- **pax-jms** does the same for **javax.jms.(XA)ConnectionFactory** services and ships some JMS pooling implementations
- **pax-transx** adds configuration options and discovery for **javax.transaction.TransactionManager** implementations and (finally) provides JCA-based JDBC/JMS connection management with pooling and tranasction support.

The sections about JDBC connection pools and about JMS connection pools are still valid. The only change needed to use JCA-based pools is to use **pool=transx** properties when registering JDBC data sources and JMS connection factories.

- **pax-jdbc-pool-transx** uses **org.ops4j.pax.transx.jdbc.ManagedDataSourceBuilder** from **pax-transx-jdbc**
- **pax-jms-pool-transx** uses **org.ops4j.pax.transx.jms.ManagedConnectionFactoryBuilder** from **pax-transx-jms**

While the pooled data sources/connection factories are created in *builder style* (no Java™ bean properties), these properties are supported for JDBC:

- **name**
- **userName**
- **password**
- **commitBeforeAutocommit**
- **preparedStatementCacheSize**
- **transactionIsolationLevel**
- **minIdle**
- **maxPoolSize**
- **aliveBypassWindow**
- **houseKeepingPeriod**
- **connectionTimeout**
- **idleTimeout**
- **maxLifetime**

These properties are supported for JMS:

- **name**

- **userName**
- **password**
- **clientID**
- **minIdle**
- **maxPoolSize**
- **aliveBypassWindow**
- **houseKeepingPeriod**
- **connectionTimeout**
- **idleTimeout**
- **maxLifetime**

userName and **password** properties are needed for XA recovery to work (just like it was with **aries.xa.username** and **aries.xa.password** properties in Fuse 6.x).

With this JDBC configuration in Blueprint (mind **pool=transx**):

```
<!--
    Database-specific, non-pooling, non-enlisting javax.sql.XADataSource
-->
<bean id="postgresql" class="org.postgresql.xa.PGXADataSource">
    <property name="url" value="jdbc:postgresql://localhost:5432/reportdb" />
    <property name="user" value="fuse" />
    <property name="password" value="fuse" />
    <property name="currentSchema" value="report" />
    <property name="connectTimeout" value="5" />
</bean>

<!--
    Expose database-specific data source with service properties
    No need to expose pooling, enlisting, non database-specific javax.sql.DataSource - it'll be
registered
    automatically by pax-jdbc-config with the same properties as this <service>, but with higher
service.ranking
-->
<service id="pool" ref="postgresql" interface="javax.sql.XADataSource">
    <service-properties>
        <!-- "pool" key is needed for pax-jdbc-config to wrap database-specific data source inside
connection pool -->
        <entry key="pool" value="transx" />
        <!-- <service>/@id attribute doesn't propagate, but name of the datasource is required using one
of: -->
        <entry key="osgi.jndi.service.name" value="jdbc/postgresql" />
        <!-- or: -->
        <!--<entry key="dataSourceName" value="jdbc/postgresql" />-->
        <!-- Other properties, that normally are needed by e.g., pax-jdbc-pool-transx -->
        <entry key="pool.maxPoolSize" value="13" />
        <entry key="pool.userName" value="fuse" />
```

```
        <entry key="pool.password" value="fuse" />
    </service-properties>
</service>
```

And with this JMS configuration in Blueprint (mind **pool=transx**):

```
<!--
    Broker-specific, non-pooling, non-enlisting javax.jms.XAConnectionFactory
-->
<bean id="artemis" class="org.apache.activemq.artemis.jms.client.ActiveMQXAConnectionFactory">
    <argument index="0" value="tcp://localhost:61616" />
    <!-- credentials needed for JCA-based XA-recovery -->
    <argument index="1" value="admin" />
    <argument index="2" value="admin" />
    <property name="callTimeout" value="2000" />
    <property name="initialConnectAttempts" value="3" />
</bean>

<!--
    Expose broker-specific connection factory with service properties
    No need to expose pooling, enlisting, non broker-specific javax.jms.XAConnectionFactory - it'll be
registered
    automatically by pax-jms-config with the same properties as this <service>, but with higher
service.ranking
-->
<service id="pool" ref="artemis" interface="javax.jms.XAConnectionFactory">
    <service-properties>
        <!-- "pool" key is needed for pax-jms-config to wrap broker-specific connection factory inside
connection pool -->
        <entry key="pool" value="transx" />
        <!-- <service>/@id attribute doesn't propagate, but name of the connection factory is required
using one of: -->
        <entry key="osgi.jndi.service.name" value="jms/artemis" />
        <!-- or: -->
        <!--<entry key="name" value="jms/artemis" />-->
        <!-- Other properties, that normally are needed e.g., pax-jms-pool-transx -->
        <entry key="pool.maxPoolSize" value="13" />
        <entry key="pool.userName" value="admin" />
        <entry key="pool.password" value="admin" />
    </service-properties>
</service>
```

You have a JDBC data source and a JMS connection factory registered that leverage JCA-based resource management. transx-based pools will properly integrate with **pax-transx-tm-narayana** with respect to XA recovery.

The features that are needed are:

- **pax-jdbc-pool-tranx**
- **pax-jms-pool-tranx**
- **pax-transx-jdbc**
- **pax-transx-jms**

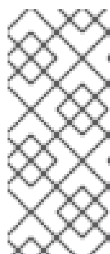
- **pax-jms-artemis** (when using A-MQ 7)

CHAPTER 9. WRITING A CAMEL APPLICATION THAT USES TRANSACTIONS

After you configure three, available-to-be-referenced, types of services, you are ready to write an application. The three types of services are:

- One transaction manager that is an implementation of one of the following interfaces:
  - **javax.transaction.UserTransaction**
  - **javax.transaction.TransactionManager**
  - **org.springframework.transaction.PlatformTransactionManager**
- At least one JDBC data source that implements the **javax.sql.DataSource**. interface. Often, there is more than one data source.
- At least one JMS connection factory that implements the **javax.jms.ConnectionFactory** interface. Often, there is more than one.

This section describes a Camel-specific configuration related to management of transactions, data sources, and connection factories.



NOTE

This section describes several Spring-related concepts such as **SpringTransactionPolicy**. There is a clear distinction between **Spring XML DSL** and **Blueprint XML DSL**, which are both XML languages that define Camel contexts. Spring XML DSL is now *deprecated* in Fuse. However, the Camel transaction mechanisms still uses the Spring library internally.

Most of the information here is not dependent on the kind of **PlatformTransactionManager** that is used. If the **PlatformTransactionManager** is the Narayana transaction manager, then full JTA transactions are used. If **PlatformTransactionManager** is defined as a local Blueprint **<bean>**, for example, **org.springframework.jms.connection.JmsTransactionManager**, then local transactions are used.

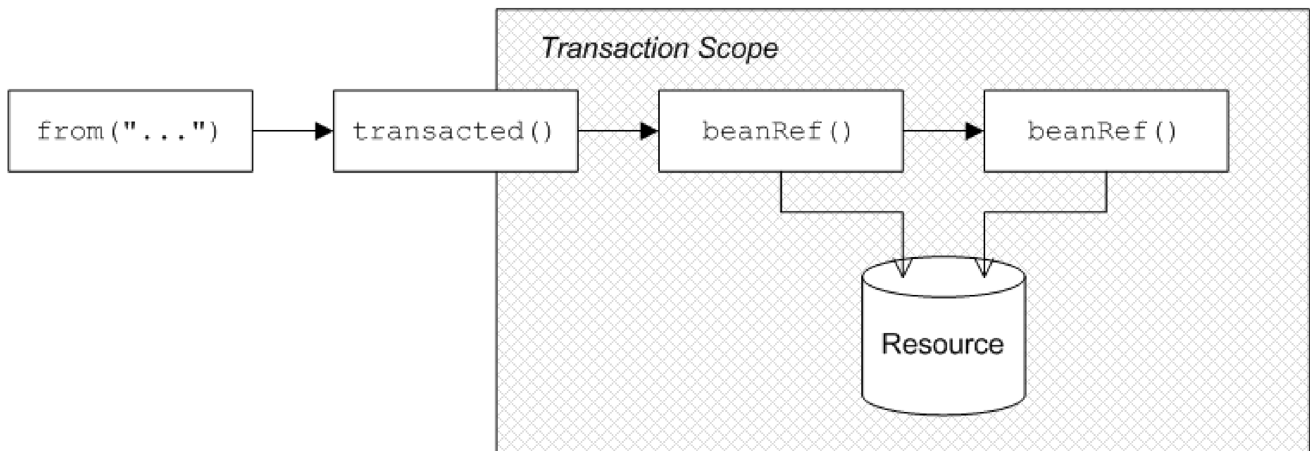
Transaction demarcation refers to the procedures for starting, committing, and rolling back transactions. This section describes the mechanisms that are available for controlling transaction demarcation, both by programming and by configuration.

- Section 9.1, “Transaction demarcation by marking the route”
- Section 9.2, “Demarcation by transactional endpoints”
- Section 9.3, “Demarcation by declarative transactions”
- Section 9.4, “Transaction propagation policies”
- Section 9.5, “Error handling and rollbacks”

9.1. TRANSACTION DEMARCATION BY MARKING THE ROUTE

Apache Camel provides a simple mechanism for initiating a transaction in a route. Insert the **transacted()** command in the Java DSL or insert the **<transacted/>** tag in the XML DSL.

Figure 9.1. Demarcation by Marking the Route



The transacted processor demarcates transactions as follows:

1. When an exchange enters the transacted processor, the transacted processor invokes the default transaction manager to begin a transaction and attaches the transaction to the current thread.

2. When the exchange reaches the end of the remaining route, the transacted processor invokes the transaction manager to commit the current transaction.

9.1.1. Sample route with JDBC resource

Figure 9.1, "Demarcation by Marking the Route" shows an example of a route that is made transactional by adding the **transacted()** DSL command to the route. All of the route nodes that follow the **transacted()** node are included in the transaction scope. In this example, the two following nodes access a JDBC resource.

9.1.2. Route definition in Java DSL

The following Java DSL example shows how to define a transactional route by marking the route with the **transacted()** DSL command:

```
import org.apache.camel.builder.RouteBuilder;

class MyRouteBuilder extends RouteBuilder {
    public void configure() {
        from("file:src/data?noop=true")
                .transacted()
                .bean("accountService","credit")
                .bean("accountService","debit");
    }
}
```

In this example, the file endpoint reads some XML format files that describe a transfer of funds from one account to another. The first **bean()** invocation credits the specified sum of money to the beneficiary's account and then the second **bean()** invocation subtracts the specified sum of money from the sender's account. Both of the **bean()** invocations cause updates to be made to a database resource. It is assumed that the database resource is bound to the transaction through the transaction manager, for example, see Chapter 6, *Using JDBC data sources*.

9.1.3. Route definition in Blueprint XML

The preceding route can also be expressed in Blueprint XML. The **<transacted />** tag marks the route as transactional, as shown in the following XML:

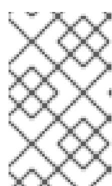
```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <route>
            <from uri="file:src/data?noop=true" />
            <transacted />
            <bean ref="accountService" method="credit" />
            <bean ref="accountService" method="debit" />
        </route>
    </camelContext>

</blueprint>
```

9.1.4. Default transaction manager and transacted policy

To demarcate transactions, the transacted processor must be associated with a particular transaction manager instance. To save you having to specify the transaction manager every time you invoke **transacted()**, the transacted processor automatically picks a sensible default. For example, if there is only one instance of a transaction manager in your configuration, the transacted processor implicitly picks this transaction manager and uses it to demarcate transactions.

A transacted processor can also be configured with a transacted policy, of **TransactedPolicy** type, which encapsulates a propagation policy and a transaction manager (see Section 9.4, “Transaction propagation policies” for details). The following rules are used to pick the default transaction manager or transaction policy:

1. If there is only one bean of **org.apache.camel.spi.TransactedPolicy** type, use this bean.



NOTE

The **TransactedPolicy** type is a base type of the **SpringTransactionPolicy** type that is described in Section 9.4, “Transaction propagation policies” . Hence, the bean referred to here could be a **SpringTransactionPolicy** bean.

2. If there is a bean of type, **org.apache.camel.spi.TransactedPolicy**, which has the **ID**, **PROPAGATION_REQUIRED**, use this bean.

3. If there is only one bean of **org.springframework.transaction.PlatformTransactionManager** type, use this bean.

You also have the option of specifying a bean explicitly by providing the bean ID as an argument to **transacted()**. See Section 9.4.4, “Sample route with **PROPAGATION_NEVER** policy in Java DSL” .

9.1.5. Transaction scope

If you insert a transacted processor into a route, the transaction manager creates a new transaction each time an exchange passes through this node. The transaction’s scope is defined as follows:

- The transaction is associated with only the current thread.
- The transaction scope encompasses all of the route nodes that follow the transacted processor.

Any route nodes that precede the transacted processor are not in the transaction. However, if the route begins with a transactional endpoint then all nodes in the route are in the transaction. See Section 9.2.5, “Transactional endpoints at start of route”.

Consider the following route. It is incorrect because the **transacted()** DSL command mistakenly appears after the first **bean()** call, which accesses the database resource:

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
   ...
   public void configure() {
      from("file:src/data?noop=true")
              .bean("accountService", "credit")
              .transacted()  // <-- WARNING: Transaction started in the wrong place!
              .bean("accountService", "debit");
   }
}
```

9.1.6. No thread pools in a transactional route

It is crucial to understand that a given transaction is associated with only the current thread. You must not create a thread pool in the middle of a transactional route because the processing in the new threads will not participate in the current transaction. For example, the following route is bound to cause problems:

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
   ...
   public void configure() {
      from("file:src/data?noop=true")
              .transacted()
              .threads(3)  // WARNING: Subthreads are not in transaction scope!
              .bean("accountService", "credit")
              .bean("accountService", "debit");
   }
}
```

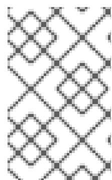
A route such as the preceding one is certain to corrupt your database because the **threads()** DSL command is incompatible with transacted routes. Even if the **threads()** call precedes the **transacted()** call, the route will not behave as expected.

9.1.7. Breaking a route into fragments

If you want to break a route into fragments and have each route fragment participate in the current transaction, you can use **direct:** endpoints. For example, to send exchanges to separate route fragments, depending on whether the transfer amount is big (greater than 100) or small (less than or equal to 100), you can use the **choice()** DSL command and direct endpoints, as follows:

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        from("file:src/data?noop=true")
                .transacted()
                .bean("accountService", "credit")
                .choice().when(xpath("/transaction/transfer[amount > 100]"))
                .to("direct:txbig")
                .otherwise()
                .to("direct:txsmall");

        from("direct:txbig")
                .bean("accountService", "debit")
                .bean("accountService", "dumpTable")
                .to("file:target/messages/big");

        from("direct:txsmall")
                .bean("accountService", "debit")
                .bean("accountService", "dumpTable")
                .to("file:target/messages/small");
    }
}
```

Both the fragment beginning with **direct:txbig** and the fragment beginning with **direct:txsmall** participate in the current transaction because the direct endpoints are synchronous. This means that the fragments execute in the same thread as the first route fragment and, therefore, they are included in the same transaction scope.



NOTE

You must not use **seda** endpoints to join the route fragments. **seda** consumer endpoints create a new thread (or threads) to execute the route fragment (asynchronous processing). Hence, the fragments would not participate in the original transaction.

9.1.8. Resource endpoints

The following Apache Camel components act as resource endpoints when they appear as the destination of a route, for example, if they appear in the **to()** DSL command. That is, these endpoints can access a transactional resource, such as a database or a persistent queue. The resource endpoints can participate in the current transaction, as long as they are associated with the same transaction manager as the transacted processor that initiated the current transaction.

- ActiveMQ
- AMQP
- Hibernate
- iBatis
- JavaSpace

- JBI
- JCR
- JDBC
- JMS
- JPA
- LDAP

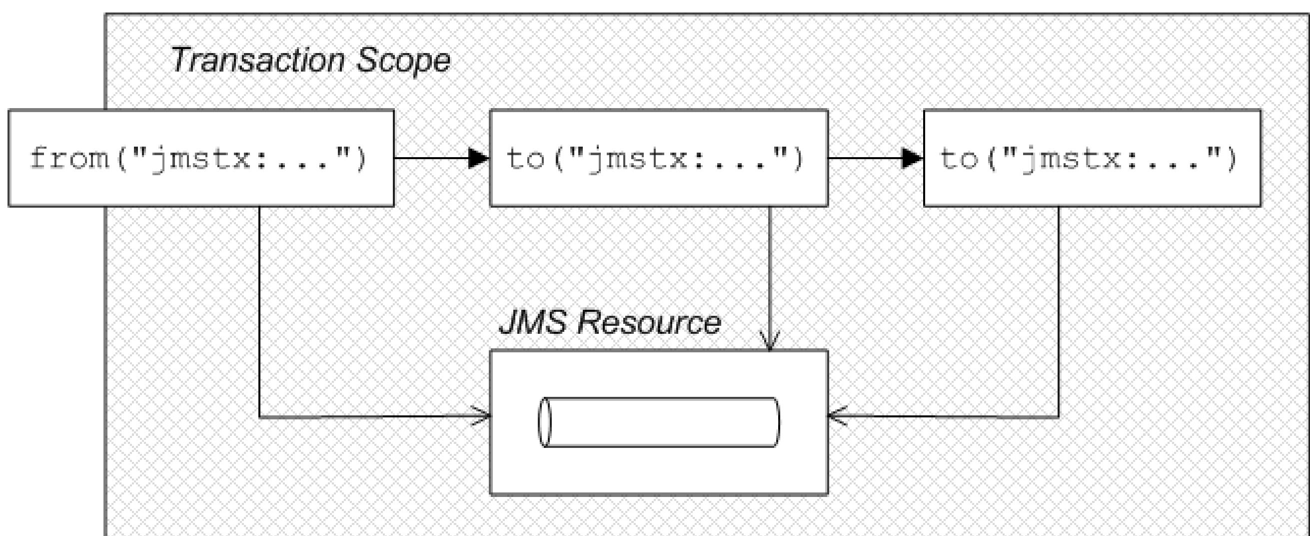
9.1.9. Sample route with resource endpoints

The following example shows a route with resource endpoints. This sends the order for a money transfer to two different JMS queues. The **credits** queue processes the order to credit the receiver's account. The **debits** queue processes the order to debit the sender's account. There should be a credit only if there is a corresponding debit. Consequently, you want to enclose the enqueueing operations in a single transaction. If the transaction succeeds, both the credit order and the debit order will be enqueued. If an error occurs, neither order will be enqueued.

```
from("file:src/data?noop=true")
        .transacted()
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

9.2. DEMARCATION BY TRANSACTIONAL ENDPOINTS

If a consumer endpoint at the start of a route accesses a resource, the **transacted()** command is of no use, because it initiates the transaction after an exchange is polled. In other words, the transaction starts too late to include the consumer endpoint within the transaction scope. In this case, the correct approach is to make the endpoint itself responsible for initiating the transaction. An endpoint that is capable of managing transactions is known as a *transactional endpoint*.



There are two different models of demarcation by transactional endpoint, as follows:

- General case – normally, a transactional endpoint demarcates transactions as follows:

1. When an exchange arrives at the endpoint, or when the endpoint successfully polls for an exchange, the endpoint invokes its associated transaction manager to begin a transaction.
2. The endpoint attaches the new transaction to the current thread.
3. When the exchange reaches the end of the route, the transactional endpoint invokes the transaction manager to commit the current transaction.

- JMS endpoint with an *InOut* exchange — when a JMS consumer endpoint receives an *InOut* exchange and this exchange is routed to another JMS endpoint, this must be treated as a special case. The problem is that the route can deadlock, if you try to enclose the entire request/reply exchange in a single transaction.

9.2.1. Sample route with a JMS endpoint

Section 9.2, "Demarcation by transactional endpoints" shows an example of a route that is made transactional by the presence of a transactional endpoint at the start of the route (in the **from()** command). All of the route nodes are included in the transaction scope. In this example, all of the endpoints in the route access a JMS resource.

9.2.2. Route definition in Java DSL

The following Java DSL example shows how to define a transactional route by starting the route with a transactional endpoint:

```
from("jmstx:queue:giro")
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

In the previous example, the transaction scope encompasses the endpoints, **jmstx:queue:giro**, **jmstx:queue:credits**, and **jmstx:queue:debits**. If the transaction succeeds, the exchange is permanently removed from the **giro** queue and pushed on to the **credits** queue and the **debits** queue. If the transaction fails, the exchange does not get put on to the **credits** and **debits** queues and the exchange is pushed back on to the **giro** queue. By default, JMS automatically attempts to redeliver the message. The JMS component bean, **jmstx**, must be explicitly configured to use transactions, as follows:

```
<blueprint ...>
    <bean id="jmstx" class="org.apache.camel.component.jms.JmsComponent">
        <property name="configuration" ref="jmsConfig" />
    </bean>

    <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="jmsConnectionFactory" />
        <property name="transactionManager" ref="jmsTransactionManager" />
        <property name="transacted" value="true" />
    </bean>
    ...
</blueprint>
```

In the previous example, the transaction manager instance, **jmsTransactionManager**, is associated with the JMS component and the **transacted** property is set to **true** to enable transaction demarcation for *InOnly* exchanges.

9.2.3. Route definition in Blueprint XML

The preceding route can equivalently be expressed in Blueprint XML, as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="jmstx:queue:giro" />
      <to uri="jmstx:queue:credits" />
      <to uri="jmstx:queue:debits" />
    </route>
  </camelContext>

</blueprint>
```

9.2.4. DSL transacted() command not required

The **transacted()** DSL command is not required in a route that starts with a transactional endpoint. Nevertheless, assuming that the default transaction policy is **PROPAGATION_REQUIRED** (see Section 9.4, "Transaction propagation policies"), it is usually harmless to include the **transacted()** command, as in this example:

```
from("jmstx:queue:giro")
        .transacted()
        .to("jmstx:queue:credits")
        .to("jmstx:queue:debits");
```

However, it is possible for this route to behave in unexpected ways, for example, if a single **TransactedPolicy** bean having a non-default propagation policy is created in Blueprint XML. See Section 9.1.4, "Default transaction manager and transacted policy". Consequently, it is usually better not to include the **transacted()** DSL command in routes that start with a transactional endpoint.

9.2.5. Transactional endpoints at start of route

The following Apache Camel components act as transactional endpoints when they appear at the start of a route (for example, if they appear in the **from()** DSL command). That is, these endpoints can be configured to behave as a transactional client and they can also access a transactional resource.

- ActiveMQ
- AMQP
- JavaSpace
- JMS
- JPA

9.3. DEMARCATION BY DECLARATIVE TRANSACTIONS

When using Blueprint XML, you can also demarcate transactions by declaring transaction policies in your Blueprint XML file. By applying the appropriate transaction policy to a bean or bean method, for example, the **Required** policy, you can ensure that a transaction is started whenever that particular bean or bean method is invoked. At the end of the bean method, the transaction is committed. This approach is analogous to the way that transactions are dealt with in Enterprise Java Beans.

OSGi declarative transactions enable you to define transaction policies at the following scopes in your Blueprint file:

- Section 9.3.1, “Bean-level declaration”
- Section 9.3.2, “Top-level declaration”

See also: Section 9.3.3, “Description of **tx:transaction** attributes”.

9.3.1. Bean-level declaration

To declare transaction policies at the bean level, insert a **tx:transaction** element as a child of the **bean** element, as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

  <bean id="accountFoo" class="org.jboss.fuse.example.Account">
    <tx:transaction method="*" value="Required" />
    <property name="accountName" value="Foo" />
  </bean>

  <bean id="accountBar" class="org.jboss.fuse.example.Account">
    <tx:transaction method="*" value="Required" />
    <property name="accountName" value="Bar" />
  </bean>

</blueprint>
```

In the preceding example, the required transaction policy is applied to all methods of the **accountFoo** bean and the **accountBar** bean, where the method attribute specifies the wildcard, *****, to match all bean methods.

9.3.2. Top-level declaration

To declare transaction policies at the top level, insert a **tx:transaction** element as a child of the **blueprint** element, as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:tx="http://aries.apache.org/xmlns/transactions/v1.1.0">

  <tx:transaction bean="account*" value="Required" />

  <bean id="accountFoo" class="org.jboss.fuse.example.Account">
    <property name="accountName" value="Foo" />
  </bean>

  <bean id="accountBar" class="org.jboss.fuse.example.Account">
    <property name="accountName" value="Bar" />
  </bean>

</blueprint>
```

In the preceding example, the **Required** transaction policy is applied to all methods of every bean whose **ID** matches the pattern, **account***.

9.3.3. Description of tx:transaction attributes

The **tx:transaction** element supports the following attributes:

bean

(Top-level only) Specifies a list of bean IDs (comma or space separated) to which the transaction policy applies. For example:

```
<blueprint ...>
  <tx:transaction bean="accountFoo,accountBar" value="..." />
</blueprint>
```

You can also use the wildcard character, *, which may appear at most once in each list entry. For example:

```
<blueprint ...>
  <tx:transaction bean="account*,jms*" value="..." />
</blueprint>
```

If the bean attribute is omitted, it defaults to * (matching all non-synthetic beans in the blueprint file).

method

(Top-level and bean-level) Specifies a list of method names (comma or space separated) to which the transaction policy applies. For example:

```
<bean id="accountFoo" class="org.jboss.fuse.example.Account">
  <tx:transaction method="debit,credit,transfer" value="Required" />
  <property name="accountName" value="Foo" />
</bean>
```

You can also use the wildcard character, *, which may appear at most once in each list entry.

If the method attribute is omitted, it defaults to * (matching all methods in the applicable beans).

value

(Top-level and bean-level) Specifies the transaction policy. The policy values have the same semantics as the policies defined in the EJB 3.0 specification, as follows:

- **Required** – support a current transaction; create a new one if none exists.
- **Mandatory** – support a current transaction; throw an exception if no current transaction exists.
- **RequiresNew** – create a new transaction, suspending the current transaction if one exists.
- **Supports** – support a current transaction; execute non-transactionally if none exists.
- **NotSupported** – do not support a current transaction; rather always execute non-transactionally.
- **Never** – do not support a current transaction; throw an exception if a current transaction exists.

9.4. TRANSACTION PROPAGATION POLICIES

If you want to influence the way a transactional client creates new transactions, you can use **JmsTransactionManager** and specify a transaction policy for it. In particular, Spring transaction policies enable you to specify a propagation behavior for your transaction. For example, if a transactional client is about to create a new transaction and it detects that a transaction is already associated with the current thread, should it go ahead and create a new transaction, suspending the old one? Or should it let the existing transaction take over? These kinds of behavior are regulated by specifying the propagation behavior on a transaction policy.

Transaction policies are instantiated as beans in Blueprint XML. You can then reference a transaction policy by providing its bean **ID** as an argument to the **transacted()** DSL command. For example, if you want to initiate transactions subject to the behavior, **PROPAGATION_REQUIRES_NEW**, you could use the following route:

```
from("file:src/data?noop=true")
        .transacted("PROPAGATION_REQUIRES_NEW")
        .bean("accountService","credit")
        .bean("accountService","debit")
        .to("file:target/messages");
```

Where the **PROPAGATION_REQUIRES_NEW** argument specifies the bean **ID** of a transaction policy bean that is configured with the **PROPAGATION_REQUIRES_NEW** behavior. See Section 9.4.3, “Defining policy beans in Blueprint XML”.

9.4.1. About Spring transaction policies

Apache Camel lets you define Spring transaction policies using the **org.apache.camel.spring.spi.SpringTransactionPolicy** class, which is essentially a wrapper around a native Spring class. The **SpringTransactionPolicy** class encapsulates two pieces of data:

- A reference to a transaction manager of **PlatformTransactionManager** type
- A propagation behavior

For example, you could instantiate a Spring transaction policy bean with **PROPAGATION_MANDATORY** behavior, as follows:

```
<blueprint ...>
  <bean id="PROPAGATION_MANDATORY
"class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager" />
    <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY" />
  </bean>
  ...
</blueprint>
```

9.4.2. Descriptions of propagation behaviors

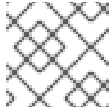
The following propagation behaviors are supported by Spring. These values were originally modeled on the propagation behaviors supported by JavaeEE:

PROPAGATION_MANDATORY

Support a current transaction. Throw an exception if no current transaction exists.

PROPAGATION_NESTED

Execute within a nested transaction if a current transaction exists, else behave like **PROPAGATION_REQUIRED**.



NOTE

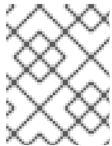
Nested transactions are not supported by all transaction managers.

PROPAGATION_NEVER

Do not support a current transaction. Throw an exception if a current transaction exists.

PROPAGATION_NOT_SUPPORTED

Do not support a current transaction. Always execute non-transactionally.



NOTE

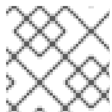
This policy requires the current transaction to be suspended, a feature which is not supported by all transaction managers.

PROPAGATION_REQUIRED

(Default) Support a current transaction. Create a new one if none exists.

PROPAGATION_REQUIRES_NEW

Create a new transaction, suspending the current transaction if one exists.



NOTE

Suspending transactions is not supported by all transaction managers.

PROPAGATION_SUPPORTS

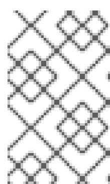
Support a current transaction. Execute non-transactionally if none exists.

9.4.3. Defining policy beans in Blueprint XML

The following example shows how to define transaction policy beans for all of the supported propagation behaviors. For convenience, each of the bean IDs matches the specified value of the propagation behavior value, but in practice you can use whatever value you like for the bean IDs.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <bean id="PROPAGATION_MANDATORY "
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
       <property name="transactionManager" ref="txManager" />
       <property name="propagationBehaviorName" value="PROPAGATION_MANDATORY" />
   </bean>

   <bean id="PROPAGATION_NESTED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
       <property name="transactionManager" ref="txManager" />
       <property name="propagationBehaviorName" value="PROPAGATION_NESTED" />
```

```
    </bean>

    <bean id="PROPAGATION_NEVER"
  class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_NEVER" />
    </bean>

    <bean id="PROPAGATION_NOT_SUPPORTED"
  class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_NOT_SUPPORTED" />
    </bean>

    <!-- This is the default behavior. -->
    <bean id="PROPAGATION_REQUIRED"
  class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
    </bean>

    <bean id="PROPAGATION_REQUIRES_NEW"
  class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_REQUIRES_NEW" />
    </bean>

    <bean id="PROPAGATION_SUPPORTS"
  class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager" />
        <property name="propagationBehaviorName" value="PROPAGATION_SUPPORTS" />
    </bean>

  </blueprint>
```



NOTE

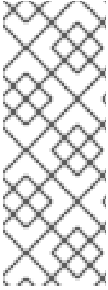
If you want to paste any of these bean definitions into your own Blueprint XML configuration, remember to customize the references to the transaction manager. That is, replace references to **txManager** with the actual **ID** of your transaction manager bean.

9.4.4. Sample route with PROPAGATION_NEVER policy in Java DSL

A simple way of demonstrating that transaction policies have some effect on a transaction is to insert a **PROPAGATION_NEVER** policy into the middle of an existing transaction, as shown in the following route:

```
from("file:src/data?noop=true")
        .transacted()
        .bean("accountService","credit")
        .transacted("PROPAGATION_NEVER")
        .bean("accountService","debit");
```

Used in this way, the **PROPAGATION_NEVER** policy inevitably aborts every transaction, leading to a transaction rollback. You should easily be able to see the effect of this on your application.



NOTE

Remember that the string value passed to **transacted()** is a bean **ID**, not a propagation behavior name. In this example, the bean **ID** is chosen to be the same as a propagation behavior name, but this need not always be the case. For example, if your application uses more than one transaction manager, you might end up with more than one policy bean having a particular propagation behavior. In this case, you could not simply name the beans after the propagation behavior.

9.4.5. Sample route with PROPAGATION_NEVER policy in Blueprint XML

The preceding route can be defined in Blueprint XML, as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="file:src/data?noop=true" />
      <transacted />
      <bean ref="accountService" method="credit" />
      <transacted ref="PROPAGATION_NEVER" />
      <bean ref="accountService" method="debit" />
    </route>
  </camelContext>

</blueprint>
```

9.5. ERROR HANDLING AND ROLLBACKS

While you can use standard Apache Camel error handling techniques in a transactional route, it is important to understand the interaction between exceptions and transaction demarcation. In particular, you need to consider that thrown exceptions usually cause transaction rollback. See the following topics:

- Section 9.5.1, “How to roll back a transaction”
- Section 9.5.2, “How to define a dead letter queue”
- Section 9.5.3, “Catching exceptions around a transaction”

9.5.1. How to roll back a transaction

You can use one of the following approaches to roll back a transaction:

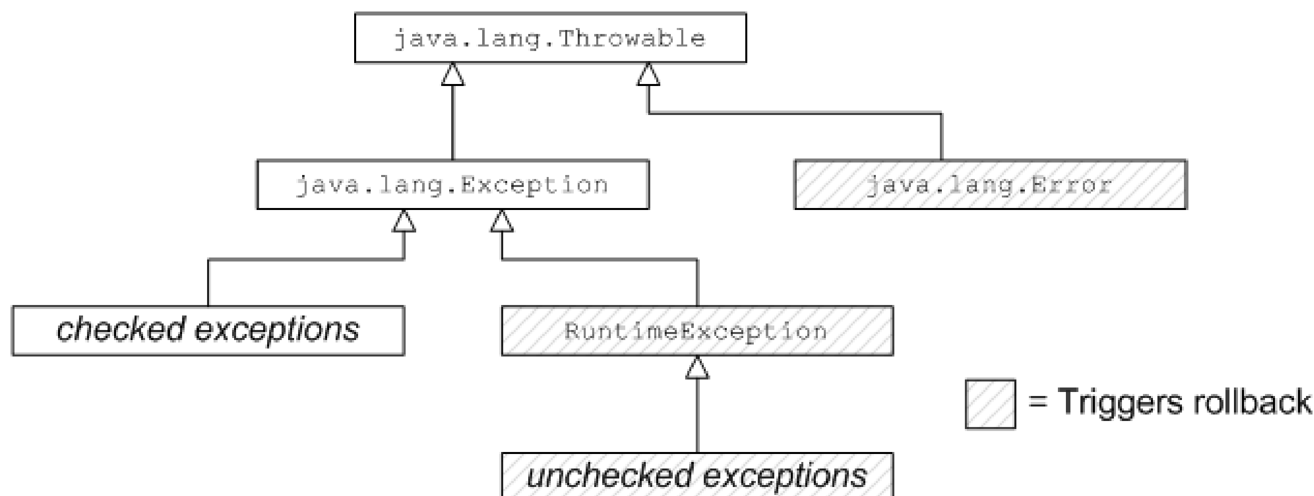
- Section 9.5.1.1, “Using runtime exceptions to trigger rollbacks”
- Section 9.5.1.2, “Using the **rollback()** DSL command”
- Section 9.5.1.3, “Using the **markRollbackOnly()** DSL command”

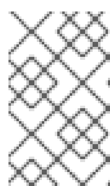
9.5.1.1. Using runtime exceptions to trigger rollbacks

The most common way to roll back a Spring transaction is to throw a runtime (unchecked) exception. In other words, the exception is an instance or subclass of **java.lang.RuntimeException**. Java errors, of

java.lang.Error type, also trigger transaction rollback. Checked exceptions, on the other hand, do not trigger rollback.

The following figure summarizes how Java errors and exceptions affect transactions, where the classes that trigger roll back are shaded gray.





NOTE

The Spring framework also provides a system of XML annotations that enable you to specify which exceptions should or should not trigger roll backs. For details, see "Rolling back" in the Spring Reference Guide.

WARNING

If a runtime exception is handled within the transaction, that is, before the exception has the chance to percolate up to the code that does the transaction demarcation, the transaction will not be rolled back. For details, see Section 9.5.2, "How to define a dead letter queue".

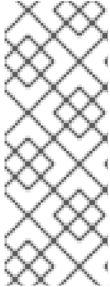
9.5.1.2. Using the rollback() DSL command

If you want to trigger a rollback in the middle of a transacted route, you can do this by calling the **rollback()** DSL command, which throws an **org.apache.camel.RollbackExchangeException** exception. In other words, the **rollback()** command uses the standard approach of throwing a runtime exception to trigger the rollback.

For example, suppose that you decide that there should be an absolute limit on the size of money transfers in the account services application. You could trigger a rollback when the amount exceeds 100 by using the code in the following example:

```
from("file:src/data?noop=true")
    .transacted()
    .bean("accountService","credit")
    .choice().when(xpath("/transaction/transfer[amount > 100]"))
        .rollback()
    .otherwise()
```

```
        .to("direct:txsmall");

from("direct:txsmall")
    .bean("accountService","debit")
    .bean("accountService","dumpTable")
    .to("file:target/messages/small");
```



NOTE

If you trigger a rollback in the preceding route, it will get trapped in an infinite loop. The reason for this is that the **RollbackExchangeException** exception thrown by **rollback()** propagates back to the **file** endpoint at the start of the route. The File component has a built-in reliability feature that causes it to resend any exchange for which an exception has been thrown. Upon resending, of course, the exchange just triggers another rollback, leading to an infinite loop. The next example shows how to avoid this infinite loop.

9.5.1.3. Using the markRollbackOnly() DSL command

The **markRollbackOnly()** DSL command enables you to force the current transaction to roll back, without throwing an exception. This can be useful when throwing an exception has unwanted side effects, such as the example in Section 9.5.1.2, “Using the **rollback()** DSL command”.

The following example shows how to modify the example in Section 9.5.1.2, “Using the **rollback()** DSL command” by replacing the **rollback()** command with the **markRollbackOnly()** command. This version of the route solves the problem of the infinite loop. In this case, when the amount of the money transfer exceeds 100, the current transaction is rolled back, but no exception is thrown. Because the file endpoint does not receive an exception, it does not retry the exchange, and the failed transactions is quietly discarded.

The following code rolls back an exception with the **markRollbackOnly()** command:

```
from("file:src/data?noop=true")
    .transacted()
    .bean("accountService","credit")
    .choice().when(xpath("/transaction/transfer[amount > 100]"))
        .markRollbackOnly()
    .otherwise()
        .to("direct:txsmall");
...
```

The preceding route implementation is not ideal, however. Although the route cleanly rolls back the transaction (leaving the database in a consistent state) and avoids the pitfall of infinite looping, it does not keep any record of the failed transaction. In a real-world application, you would typically want to keep track of any failed transaction. For example, you might want to write a letter to the relevant customer in order to explain why the transaction did not succeed. A convenient way of tracking failed transactions is to add a dead-letter queue to the route.

9.5.2. How to define a dead letter queue

To keep track of failed transactions, you can define an **onException()** clause, which enables you to divert the relevant exchange object to a dead-letter queue. When used in the context of transactions, however, you need to be careful about how you define the **onException()** clause, because of potential

interactions between exception handling and transaction handling. The following example shows the correct way to define an **onException()** clause, assuming that you need to suppress the rethrown exception.

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
        onException(IllegalArgumentException.class)
            .maximumRedeliveries(1)
            .handled(true)
            .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .markRollbackOnly();  // NB: Must come *after* the dead letter endpoint.

        from("file:src/data?noop=true")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit")
            .bean("accountService","dumpTable")
            .to("file:target/messages");
    }
}
```

In the preceding example, **onException()** is configured to catch the **IllegalArgumentException** exception and send the offending exchange to a dead letter file, **deadLetters.xml**. Of course, you can change this definition to catch whatever kind of exception arises in your application. The exception rethrow behavior and the transaction rollback behavior are controlled by the following special settings in the **onException()** clause:

- **handled(true)** — suppress the rethrown exception. In this particular example, the rethrown exception is undesirable because it triggers an infinite loop when it propagates back to the file endpoint. See Section 9.5.1.3, “Using the **markRollbackOnly()** DSL command”. In some cases, however, it might be acceptable to rethrow the exception (for example, if the endpoint at the start of the route does not implement a retry feature).

- **markRollbackOnly()** — marks the current transaction for rollback without throwing an exception. Note that it is essential to insert this DSL command after the **to()** command that routes the exchange to the dead letter queue. Otherwise, the exchange would never reach the dead letter queue, because **markRollbackOnly()** interrupts the chain of processing.

9.5.3. Catching exceptions around a transaction

Instead of using **onException()**, a simple approach to handling exceptions in a transactional route is to use the **doTry()** and **doCatch()** clauses around the route. For example, the following code shows how you can catch and handle the **IllegalArgumentException** in a transactional route, without the risk of getting trapped in an infinite loop.

```
// Java
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {
    ...
    public void configure() {
```

```
        from("file:src/data?noop=true")
            .doTry()
                .to("direct:split")
            .doCatch(IllegalArgumentException.class)
                .to("file:target/messages?fileName=deadLetters.xml&fileExist=Append")
            .end();

        from("direct:split")
            .transacted()
            .bean("accountService","credit")
            .bean("accountService","debit")
            .bean("accountService","dumpTable")
            .to("file:target/messages");
    }
}
```

In this example, the route is split into two segments. The first segment (from the **file:src/data** endpoint) receives the incoming exchanges and performs the exception handling using **doTry()** and **doCatch()**. The second segment (from the **direct:split** endpoint) does all of the transactional work. If an exception occurs within this transactional segment, it propagates first of all to the **transacted()** command, causing the current transaction to be rolled back, and it is then caught by the **doCatch()** clause in the first route segment. The **doCatch()** clause does not rethrow the exception, so the file endpoint does not do any retries and infinite looping is avoided.