RFC 122 - Database Access

Draft

12 Pages

Abstract

This RFC describes approaches for accessing JDBC resources in an OSGi environment.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 7.1.

```
Source code is shown in this typeface.
```

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Aug 14 2007	Initial draft with information based on RFP 77 Stuart McCulloch, Jayway Malaysia stuart.mcculloch@jayway.net
0.1	Aug 14 2007	Add thoughts / notes so far, as well as basic traceability matrix Stuart McCulloch, Jayway Malaysia stuart.mcculloch@jayway.net
0.2	Jan 7 2008	Cover legacy use of java.sql.DriverManager utility class Stuart McCulloch, Jayway Malaysia stuart.mcculloch@jayway.net
0.3	Sep 28 2008	Add DataSourceFactory interface Stuart McCulloch, Jayway Malaysia stuart.mcculloch@jayway.net Mike Keith, Oracle Corp. michael.keith@oracle.com
0.4	Oct 27 2008	Reworked and reorganized sections Added sections on transactions and pooling Mike Keith, Oracle Corp. michael.keith@oracle.com
0.8	Jan 10 2009	Reorganization to put more of the descriptive background explanation in earlier sections. Took out sections that prescribed dynamic and declarative registration for the DriverManager case. Mike Keith, Oracle Corp. michael.keith@oracle.com
0.9	Feb 6 2009	Added clarifications about the goal of this service not being a data source service. Mike Keith, Oracle Corp. michael.keith@oracle.com
1.0	Feb 9 2009	Draft for vote. Mike Keith, Oracle Corp. michael.keith@oracle.com

Revision	Date	Comments
1.01	Feb 24 2009	Changes requested to 5.5.2 by Ian Robinson and Roman Roelofsen to indicate that responsibility for transactions will be in a different layer. Mike Keith, Oracle Corp. michael.keith@oracle.com
1.02	Feb 26 2009	Took out obsolete best practices section 5.4. Mike Keith, Oracle Corp. michael.keith@oracle.com
1.03	Mar 10, 2009	Reworded requirements (bug 1194). Added support for ConnectionPoolDataSource and XADataSource (bug 1128 and bug 1215) JJ Snyder, Oracle Corp. j.j.snyder@oracle.com
1.04	Jul 17, 2009	Reworded the section about the driver properties. (bug 1194). Modified service properties to include a driver name and driver version (bug 1129) JJ Snyder, Oracle Corp. j.j.snyder@oracle.com
1.05	Sep 2, 2009	Added a getDriver method (bug 1426) to the DataSourceFactory interface. JJ Snyder, Oracle Corp. j.j.snyder@oracle.com

1 Introduction

This document will only cover resource access between OSGi applications and Relational Database Management Systems (RDBMS) through JDBC. Other kinds of data sources, such as hierarchical, network-oriented or object-oriented databases, are not discussed.

The goal is to both allow applications to use existing APIs, as well as be able to benefit from the paradigms and practices that are prevalent within the OSGi framework.

This specification does not attempt to deal with the definition and registration of named and configured data sources. That would be an additional layer above this one, and is not precluded from existence by anything defined in this layer. In fact, this layer provides an appropriate level of support for such a layer to be generically implemented.

2 Application Domain

Enterprise applications are typically coupled and connected to a relational database. This RFC deals with the circumstances around database connectivity for OSGi applications in the enterprise sector.

Enterprise application bundles are often independent of each other, and many applications can reside on many OSGi frameworks, potentially 100's or even 1000's, in some form of cluster.

This document describes support for existing JDBC Specifications [4] that are both implemented by a large number of vendors and used either directly or indirectly by virtually every enterprise Java application. Ensuring compatibility with that legacy is paramount if existing applications are to be able to migrate to OSGi environments. Furthermore, as more OSGi application development tools and frameworks emerge, new enterprise applications developed specifically for OSGi will want to be able to leverage and take advantage of the features that attracted them to use OSGi to begin with. This document thus describes a standard way for JDBC to be used in future applications as well as existing ones.

3 Problem Description

OSGi is now an enterprise technology, and one of the cornerstones of enterprise applications is database connectivity. There are currently no specifications relating to the installation and use of database resource services within OSGi, and without a unified way to get hold of connections to databases, each bundle or application is left to define its own way of doing so. This RFC endeavors to define mechanisms to standardize such access within the OSGI context.

Many existing technologies leverage relational database systems to make them more object friendly. JPA and proprietary ORM products are examples of this. However, if we create a persistence implementation bundle, we also need to find a way to provide the JDBC driver to that bundle from another bundle, as it is not reasonable to assume that the persistence implementation is delivered with all possible JDBC drivers.

RBDMS's may be limited in the number of concurrent connections that they can handle. Since establishing the connection represents a cost, applications may decide to hold on to connections, keeping them open and re-using them to the highest degree possible. In OSGi, this poses a problem that bundles don't know of each others existence, and by deploying a large number of bundles that each uses its own connection, the maximum connection number may be reached fairly quickly.

If we look at OSGi deployments in clustered environments the problem becomes amplified, and connections become very limited resources.

3.1 DriverManager Lookup of JDBC Drivers

In JDBC 1.0, clients used the DriverManager utility class to register and find JDBC drivers. The DriverManager class filters out drivers that are not accessible to the caller's class loader. In OSGi this means that bundles are only able to access drivers whose main implementation class can be loaded by their bundle classloader.

In JDBC 4.0 the DriverManager class allows drivers to be discovered and loaded using either the **jdbc.drivers** system property to specify one or more driver classes, or the Java SE Service Provider mechanism, where JDBC-compliant drivers ship a **META-INF/services/java.sql.Driver** file that contains the name of the driver class.

3.2 JNDI Lookup of JDBC Data Sources

JDBC 2.0 introduced the DataSource class which was designed to work with the Java Naming and Directory Interface (JNDI) naming service, and also allowed properties to be associated with the driver instance. For mobile devices this is the only way to register and lookup JDBC drivers, as the DriverManager class is not part of J2ME. This is also the most common way to obtain data sources in Java EE environments.

3.3 Using a Native DataSourceFactory

Most JDBC drivers provide a vendor-specific DataSourceFactory class that can be used to create data sources in the absence of JNDI, or when accessing the driver directly. This is less common, however, since it results in a compile-time dependency in the application to a driver-specific class.

4 Requirements

RFP0077-1	The solution MAY leverage the Service registry in OSGi.
RFP0077-2	The solution MUST allow multiple connections to many databases, incl usage of JDBC drivers of different versions, within and amongst the bundles.
RFP0077-3	The solution SHOULD provide a way where the JDBC driver is installed as a bundle, and the connections are instantiated declaratively in runtime.
RFP0077-4	The solution MAY provide a mechanism to associate a connection description in the client with an actual connection of the provider. Typically this is done by specifying additional properties when getting a connection from a DataSource.
RFP0077-5	The solution MUST support JDBC 2.0 and later.
RFP0077-6	The solution SHOULD support all types of JDBC drivers, Type 1, 2, 3 & 4.
RFP0077-7	The solution MAY include a connection pooling mechanism.
RFP0077-8	If the solution provides a connection pool then the solution MUST support the ability to set the properties of the connection pool including, but not limited to, the properties defined in section 11.8 ConnectionPoolDataSource Properties of the JDBC 4.0 specification.
RFP0077-9	The solution MUST not disallow monitoring of the activity on the database connections.
RFP0077-10	The solution MUST allow the creation of bundles for existing JDBC drivers.
RFP0077-11	The solution SHOULD allow for inter-framework connection pools.
RFP0077-12	The solution MUST not rely on the existence of the JDBC DriverManager class.
RFP0077-13	The solution SHOULD provide full XA support for ***any*** resource via the javax.sql.XADataSource

5 Technical Solution

5.1 Existing Driver Access Approaches

Successful use of existing and traditional methods of JDBC access involves correct configuration of the client and driver. Existing methods of access are described below.

5.1.1 DriverManager Lookup of JDBC

Applications that call the DriverManager class to obtain connections will need to have the JDBC driver of their choice on their bundle classpath, or add the driver as a fragment bundle. Any approach that ensures that the class loader of the driver class is part of the class loader hierarchy of the application client will achieve the desired result.

5.1.2 JNDI Lookup of JDBC Data Sources

JNDI lookup of JDBC drivers is really just a specialized case of general JNDI lookup in OSGi, covered by RFP 84, thus when running in an environment where Java EE or JNDI is supported then the implementation will support the data source lookup in JNDI as per RFP 84. This specification does not attempt to either replace or redefine such a mechanism in OSGi, although specifications may be created in the future to do that in more OSGi-friendly ways.

5.1.3 Using a Native DataSourceFactory

Applications that rely upon the native driver-supplied factory class, or other driver-specific classes, can continue to be dependent upon the driver code by importing the driver classes that are required. They may even choose to bundle the driver with the application.

5.2 Going Forward with JDBC Drivers in OSGi

As JDBC driver implementations evolve to include built-in support for OSGi they should implement the org.osgi.service.jdbc.DataSourceFactory interface shown below and register their implementation as an OSGi service, with one or more of the following service properties:

- “osgi.jdbc.driver.class” (JDBC_DRIVER_CLASS constant in DataSourceFactory interface). The value of this service property is the driver class name.
- “osgi.jdbc.driver.name” (JDBC_DRIVER_NAME constant in DataSourceFactory interface). The value of this service property is the driver name.
- “osgi.jdbc.driver.version” (JDBC_DRIVER_VERSION constant in DataSourceFactory interface). The value of this service property is the driver version. Note that this is not an OSGi bundle version but the vendor-specific version of the driver. Therefore, the value of this property is not required to follow any specific naming or versioning conventions.

Data sources may be managed, or perform other implementation-specific functions according to vendor-specific properties that may be offered. The provider detects the standard properties, plus any potential vendor-specific ones that clients may have supplied, and returns a DataSource instance based upon those properties.

```
package org.osgi.service.jdbc;

import java.sql.Driver;
import java.sql.SQLException;
import java.util.Properties;

import javax.sql.ConnectionPoolDataSource;
import javax.sql.DataSource;
import javax.sql.XADataSource;

/**
 * A factory for JDBC connection factories.  There are 3 preferred connection
 * factories for getting JDBC connections: {@link javax.sql.DataSource},
 * {@link javax.sql.ConnectionPoolDataSource}, and {@link javax.sql.XADataSource}.
 *
 * DataSource providers should implement this interface and register it as an
 * OSGi service with the JDBC driver class name in the
 * "osgi.jdbc.driver.class" property.
 */
public interface DataSourceFactory {
    /**
     * Property used by a JDBC driver to declare the driver class when registering
     * a JDBC DataSourceFactory service. Clients may filter or test this
     * property to determine if the driver is suitable, or the desired one.
     */
    public static final String JDBC_DRIVER_CLASS = "osgi.jdbc.driver.class";

    /**
     * Property used by a JDBC driver to declare the driver name when registering
     * a JDBC DataSourceFactory service. Clients may filter or test this
     * property to determine if the driver is suitable, or the desired one.
     */
    public static final String JDBC_DRIVER_NAME = "osgi.jdbc.driver.name";

    /**
     * Property used by a JDBC driver to declare the driver version when registering
     * a JDBC DataSourceFactory service. Clients may filter or test this
     * property to determine if the driver is suitable, or the desired one.
     */
    public static final String JDBC_DRIVER_VERSION = "osgi.jdbc.driver.version";

    /**
     * Common property keys that DataSource clients should supply values for
     * when calling {@link #createDataSource(Properties)}.
     */
    public static final String JDBC_DATABASE_NAME = "databaseName";
    public static final String JDBC_DATASOURCE_NAME = "dataSourceName";
    public static final String JDBC_DESCRIPTION = "description";
    public static final String JDBC_NETWORK_PROTOCOL = "networkProtocol";
    public static final String JDBC_PASSWORD = "password";
    public static final String JDBC_PORT_NUMBER = "portNumber";
    public static final String JDBC_ROLE_NAME = "roleName";
    public static final String JDBC_SERVER_NAME = "serverName";
```

```
    public static final String JDBC_USER = "user";
    public static final String JDBC_URL = "url";

    /**
     * Additional property keys that ConnectionPoolDataSource and XADataSource
     * clients can supply values for when calling
     * {@link #createConnectionPoolDataSource(Properties)} or
     * {@link #createXAPoolDataSource(Properties)}.
     */
    public static final String JDBC_INITIAL_POOL_SIZE = "initialPoolSize";
    public static final String JDBC_MAX_IDLE_TIME = "maxIdleTime";
    public static final String JDBC_MAX_POOL_SIZE = "maxPoolSize";
    public static final String JDBC_MAX_STATEMENTS = "maxStatements";
    public static final String JDBC_MIN_POOL_SIZE = "minPoolSize";
    public static final String JDBC_PROPERTY_CYCLE = "propertyCycle";

    /**
     * Create a new {@link DataSource} using the given properties.
     *
     * @param props The properties used to configure the DataSource.  Null
     *              indicates no properties.
     *              If the property cannot be set on the DataSource being
     *              created then a SQLException must be thrown.
     * @return A configured DataSource.
     * @throws SQLException If the DataSource cannot be created.
     */
    public DataSource createDataSource( Properties props ) throws SQLException;

    /**
     * Create a new {@link ConnectionPoolDataSource} using the given properties.
     *
     * @param props The properties used to configure the ConnectionPoolDataSource.
     *              Null indicates no properties. If the property cannot be set on
     *              the ConnectionPoolDataSource being created then a SQLException
     *              must be thrown.
     * @return A configured ConnectionPoolDataSource.
     * @throws SQLException If the ConnectionPoolDataSource cannot be created.
     */
    public ConnectionPoolDataSource createConnectionPoolDataSource( Properties
props )
       throws SQLException;

    /**
     * Create a new {@link XADataSource} using the given properties.
     *
     * @param props The properties used to configure the XADataSource.  Null
     *              indicates no properties. If the property cannot be set on
     *              the XADataSource being created then a SQLException must be
     *              thrown.
     * @return A configured XADataSource.
     * @throws SQLException If the XADataSource cannot be created.
     */
    public XADataSource createXADataSource( Properties props ) throws SQLException;

    /**
```

```
     * Create a new {@link Driver} using the given properties.
     *
     * @param props The properties used to configure the Driver.  Null
     *              indicates no properties.
     *              If the property cannot be set on the Driver being
     *              created then a SQLException must be thrown.
     * @return A configured Driver.
     * @throws SQLException If the Driver cannot be created.
     */
    public Driver createDriver( Properties props ) throws SQLException;

}
```

Prospective JDBC clients look up the DataSourceFactory service, with the option to filter discovery by the driver class that they want to use. They can then invoke the factory's create method, passing in the appropriate properties, to obtain a data source that can be used to obtain JDBC connections. The sample client code below shows how a third party layer, or client application, that is written to leverage the OSGi framework can use a service tracker to get a data source.

```
ServiceTracker tracker =
    new ServiceTracker( bundleContext,
                        "org.osgi.service.jdbc.DataSourceFactory",
                        new DataSourceTracker( this, driverName ) );
tracker.open();
```

Then, in the DataSourceTracker, the addingService method might look like:

```
public Object addingService( ServiceReference ref ) {

     if (ref.getProperty(DataSourceFactory.JDBC_DRIVER_NAME).equals(driverName) {
         Properties props = new Properties();
         props.put( DataSourceFactory.JDBC_URL, "jdbc:derby:MyDB" );
         props.put( DataSourceFactory.JDBC_USER, "foo" );
         props.put( DataSourceFactory.JDBC_PASSWORD, "bar" );
         DataSourceFactory df = ( DataSourceFactory )
         ref.getBundle().getBundleContext().getService( ref );
         DataSource ds =  df.getDataSource( props );
         client.setDataSource( ds );
     }
}
```

5.3 Access to JDBC Packages in OSGi

As of JDK 1.4 the JDBC 3.0 API (both java.sql package and the extension javax.sql package) is available in the core runtime, thus standard OSGi classloading rules suffice.

No changes are required to the OSGi framework to support this.

5.4 JDBC Resource Management

JDBC drivers often manage connection resources and other associated resources to reduce allocation and cleanup costs as well as provide a level of oversight, monitoring and reuse. Driver implementations that register as OSGi services may continue to perform the kind of management activities that they might normally perform.

5.4.1 Sharing and Pooling of JDBC connections

When a driver registers as a service in the Service Registry it is provided an opportunity to offer whatever kind of data source proxy it deems necessary to support sharing of underlying connections obtained from that data source in an OSGi context. Sharing may be based upon a transaction context, a thread, a bundle, or any criteria the driver sees as providing value. Sharing parameters may be configurable, or they may be preset based on driver support.

When a driver maintains a pool of connections it vends out connections from its pool when a getConnection() method call is made on its data source, or returns them when connections are closed. When additional environment support is present, such as a Java EE runtime, then additional layers or proxies may be present, providing further pooling and resource management.

5.4.2 Transaction Support

While simple JDBC transactions are offered through the driver, higher level transaction support, including XA and 2-phase commit, are supported through the transaction manager in the framework. The driver must support XA in order to be enlisted, of course, but the responsibility for transaction initiation and completion and XAResource enlistment (using the mechanisms described in RFC 98) rests in the hands of either the application using the Connection or the container which hosts the application.

Again, in Java EE OSGi environments the transaction support will be part of the implementing container, and will offer transactional XA data source availability through JNDI lookup or dependency injection.

6 Security Considerations

Service permissions may be placed on the data source service if desired, but the real security is in the established user-level mechanisms that are already in place at the database level.

7 Document Support

7.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. JSR-54 “JDBC 3.0 Specification”, JSR-221 “JDBC 4.0 Specification”, JSR-991 “JDBC 2.1 Errata Sheet”

[4]. http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#Service%20Provider

7.2 Author’s Address

Name	Stuart McCulloch
Company	Jayway Malaysia
Address	B-7-6, Megan Avenue 1, 50450 Kuala Lumpur, Malaysia
Voice	mobile: +60 16 320 5785
e-mail	stuart.mcculloch@jayway.net

Name	Mike Keith
Company	Oracle Corporation
Address	45 O'Connor Street, Suite 400, Ottawa, Canada K1P 1A4
Voice	+1 613 288 4625
e-mail	michael.keith@oracle.com

7.3 Acronyms and Abbreviations

RDBMS – Relational Database Management System

JDBC – Java Database Connectivity

JNDI – Java Naming and Directory Interface

7.4 End of Document