Voyager Database Developer’s Guide

Version 1.0 for Voyager 8.0

Table of Contents

Introduction

Overview

Voyager™ provides the ability to execute database transactions in a mobile agent and report the results back to the caller. In addition, Voyager provides a convenient API for accessing databases directly through JDBC. Agent implementations are provided that can execute raw SQL via JDBC or manipulate a database through a Hibernate session.

Preface

The purpose of this manual is to provide an introduction to the basic database support provided by Voyager. This is not meant to be a treatise on relational databases, Hibernate, or JDBC. This guide assumes a basic knowledge of Java and relational database concepts.

This preface covers the following topics:

- Database Requirements
- Contacting technical support

Database Requirements

Before attempting to use the database features of Voyager*:*

- A Java Development Kit (JDK) installed on your computer. VOYAGER requires JDK 1.5 or later. You can download the latest release of JDK from www.javasoft.com at no charge.
- A working version of Voyager installed.
- A database with a JDBC driver.

Contacting Technical Support

Recursion Software welcomes your problem reports, and appreciates all comments and suggestions for improving VOYAGER. Please send all feedback to the Recursion Software Technical Support department.

Technical support for VOYAGER is available via the web, email, and phone. The Recursion Software Support website, at http://support.recursionsw.com, provides access to searchable FAQs (Frequently Asked Questions), technical articles, and other information. It also provides access to the Recursion Software Technical Support incident tracking system, where you can submit support issues and track them. You can also contact Technical Support by sending email to psupport@recursionsw.com or by calling (972) 731-8800.

Voyager JDBC API Overview

Using the JDBC API can be cumbersome and error-prone. The Voyager JDBC API is intended to provide some convenient abstractions on top of JDBC that greatly reduce the amount of code needed to perform basic JDBC operations. The Voyager JDBC API has a limited ability to map a JDBC ResultSet row to a Java object. It is not intended however to be a full-featured ORM (Object Relational Mapping) layer such as Hibernate. The intent is that common JDBC operations should be easy.

Connection Configuration

Before any interaction with a database can occur, a connection must first be established. To establish a connection, several attributes must be specified such as database location, user name, password, etc. The class com.recursionsw.ve.jdbc.JDBCProperties acts as a container of connection attributes. Initialization is straightforward. For example, the following code fragment initializes a connection to an HSQL in-memory database named rsi:

```
IJDBCProperties dbCfg = new JDBCProperties()
                              .setDriver("org.hsqldb.jdbcDriver")
                              .setURL("jdbc:hsqldb:mem:rsi")
                              .setPassword("")
                              .setUser("sa");
```

This dbCfg object can now be used to execute SQL statements. This method of initializing connections relies on JDBC's driver manager mechanism. JDBC also allows connections to be specified via a javax.sql.DataSource. The Voyager JDBC API supports both mechanisms.

Database Actions

Voyager executes SQL statements as a group contained within an instance of IDBAction. For example, the following code fragment creates a database action, adds a single SQL select statement and executes it:

```
List results = DBActionFactory.create()
```

```
                        .addQuery("SELECT * FROM MYTABLE")
                        .execute(dbCfg);
```

The call to DBActionFactory.create() returns an instance of IDBAction. The addQuery and execute invocations then add an SQL statement to the action and executes it. A connection to the database is made based on the dbCfg object passed to execute. The execution result is always a list of IDBResult objects. The list can be manually processed to retrieve the data. This is cumbersome however since the list could contain different types of result objects depending on what SQL statements were executed. A more convenient method of processing the results is provided. A ResultProcessor object can iterate over a result list and invoke a callback interface as appropriate. For example:

```
ResultProcessor.execute(new ResultHandler(), DBActionFactory.create()
                .addQuery("SELECT * FROM MYTABLE")
                .execute(dbCfg));
```

This executes the previous query, and passes the results to an instance of ResultHandler. The implementation of ResultHandler could be the following:

```
class ResultHandler implements IDBResultHandler
{
        public void handleObjectListQuery(List row)
        {
                // process row…
        }

        public void handleUpdate(int rowCnt) {}

        public void handleException(Throwable e)
        {
                e.printStackTrace();
        }

        public void handleBeanQuery(Object bean) {}
}
```

The ResultProcessor will invoke a method of the ResultHandler for each object contained in the result list. The type of the result object determines the callback method invoked. The following table show the mapping between ResultHandler methods and result types:

	handleObjectListQuery	handleBeanQuery	handleUpdate	handleException
ExceptionResult				X
QueryResult	X	X		
UpdateResult			X	

Note that QueryResult corresponds to two callbacks. The QueryResult will cause an invocation of handleBeanQuery() if the addQuery() specified a Java object to be mapped to the row. Otherwise, handleObjectListQuery is invoked with the row represented as a List of Object.

Object Mapping

The Voyager JDBC API will always perform mapping of row data to Java objects when executing a query. No mapping is performed for update operations (INSERT, DELETE, CREATE, UPDATE). If a query does not specify a mapping object, then each row is mapped to a List of Object where each element in the List is of a type determined by the JDBC driver. This prevents the application from having to iterate over the result set executing getString(), getInt(), etc. When a mapping object is specified, the fields of the object are initialized via reflection. The database columns are mapped automatically to the object's fields based on a case-insensitive comparison of the JavaBean property name. This provides a convenient mapping without the need for configuration. This technique is limited however since JavaBean property names must match the database table column names. A discussion of how database types are mapped to JDBC types can be found here. For example, consider the following HSQLDB table definition representing a simple generic log:

```
CREATE MEMORY TABLE GENLOG
(
 LOG_ID BIGINT GENERATED BY DEFAULT AS IDENTITY NOT NULL PRIMARY KEY,
 LOG_DATE TIMESTAMP DEFAULT 'now',
 TEXT VARCHAR(255),
 IP VARCHAR(32)
)
```

A Java object that can be mapped to a row in this table would be:

```
import java.io.Serializable;
import java.sql.Timestamp;

public class LogEntry implements Serializable
{
        private String      text;
        private String      ip;
        private Timestamp   log_date;
        private int         log_id;

        public LogEntry() {}

        public void setLog_Date(Timestamp d) { log_date = d;  }

        public String getText() { return text; }

        public void setText(String text) { this.text = text; }

        public String getIP() { return ip; }

        public void setIP(String ip) { this.ip = ip; }

        public String toString()
        {
                return super.toString() + " " + log_id + " " +
                log_date + " " + ip +  " " + text;
        }

        public void setLog_ID(int id) { log_id = id; }

        public int getLog_ID() { return log_id; }

        public Timestamp getLog_Date() { return log_date; }
}
```

An application specifies that a ResultSet row should be mapped to an object by specifying the class instance when performing a query. For example, to map JDBC ResultSet rows to this object, an application could execute the following:

```
ResultProcessor.execute(new ResultHandler(), DBActionFactory.create()
        .addQuery("SELECT * FROM GENLOG", LogEntry.class)
        .execute(dbCfg));
```

After execution of the SELECT, the ResultHandler's handleBeanQuery() method will be invoked with an instance of the LogEntry class. The LogEntry instance will have all its data fields initialized with the values retrieved from the database. The handleBeanQuery() method will be invoked once for each row in the JDBC ResultSet.

JDBC Transactions

When using the Voyager JDBC API, transactions are very simple. By default, most JDBC drivers enable auto commit on each connection. This means that each SQL statement executed is within a transaction. This is suitable for ad-hoc queries, but not usually acceptable when updating database tables. Within a transaction, all operations must succeed or any modifications are rolled back when there is a failure to guarantee the integrity of the data. An application using the Voyager JDBC API specifies the scope of a transaction via an instance of an object that implement the IDBAction interface. Recall from the previous examples that the DBActionFactory creates a database action. This factory has two create methods: create() and createTransaction(). The create method returns an IDBAction that executes each SQL statement as a transaction. This is the typical default behavior of a JDBC connection (auto commit = true). Each statement added to the database action via addQuery and addUpdate is executed within a transaction. The statements are executed in sequence until they are all executed or an exception occurs. If an exception occurs, there is no rollback since logically all the statements are independent. If an IDBAction is created via the createTransaction() method, then all SQL statements are considered to be one transaction. If an exception occurs, then a rollback is executed automatically by Voyager. This is convenient for the application since it does not need to have explicit code concerned with catching exceptions and executing a rollback. The rollback will happen automatically when necessary.

JDBC Agent

Voyager provides an agent implementation that can execute JDBC operations. The Agent is constructed with a database action specifying the SQL to be executed. The agent is also initialized with the JDBC connection properties needed to establish communication with the database. For example, the following code fragment will create a JDBC Agent:

```
IJDBCProperties dbCfg = new JDBCProperties()
        .setDriver("org.hsqldb.jdbcDriver")
        .setURL("jdbc:hsqldb:mem:rsi")
        .setPassword("")
        .setUser("sa");

IDBAction dbAction = DBActionFactory.create()
        .addQuery("SELECT * FROM GENLOG", LogEntry.class);

IAgentAction agent = new JDBCAgentAction(dbAction, new
SealedJDBCProperties(dbCfg));
```

The JDBCAgentAction can then be deployed to the AgentPeer for execution. By default the agent will return a List of IDBResult objects generated by executing the database action. The JDBCAgentAction class can be extended with an application specific implementation to override what is returned by the default agent implementation. Subclasses can implement the method filterResults() to customize what the agent returns.

JDBC Troubleshooting

Most JDBC problems are due to improper SQL or incorrect connection configuration. Whenever a SQL statement throws an exception, the exception will be propagated to the application. The exception chain of SQLException usually contains the most descriptive information. Enabling the logging for the database-specific JDBC driver can also be helpful. Consult the JDBC driver documentation for instructions on enabling JDBC logging.

Hibernate Agent

Hibernate is an open source feature-rich ORM (Object Relational Mapping) service. It provides a much greater level of abstraction than JDBC. Voyager provides an agent implementation capable of creating a Hibernate session and executing session operations. The number of operations allowed on a hibernate session is rather large. Rather than attempt to provide an agent capable of doing all possible session operations, implementations of the basic operations are provided. Applications can subclass the basic agent to specialize session operations. The Hibernate agent is initialized with a Hibernate configuration object. When the agent is deployed, it uses this configuration to create a Hibernate session and execute a specific command on the session. For example, the following code fragment creates a Hibernate Agent that performs a simple query:

```
IHibernateCfg hCfg = new SealedHibernateCfg
(new org.hibernate.cfg.Configuration().configure
        ("examples/agent/hibernate.cfg.xml"));

IAgentAction action = new HibernateQuery(hCfg, "from LogDAO");
```

The HibernateQuery can then be deployed to an AgentPeer for execution. The Hibernate Configuration object can be initialized in many different ways. This example assumes that there is a file named hibernate.cfg.xml in the class path that specifies the configuration. This example also assumes that the mapping configuration specifies an object named LogDAO.

Security Considerations

When a Voyager agent is deployed, it usually is serialized and transmitted over the network. Because of this, care should be taken to prevent sending the database credentials needed by JDBC and Hibernate agents in the clear. When deploying a database agent, it is recommended that SealedJDBCProperties or SealedHibernateCfg be used when specifying the database connection information. Doing so will prevent the credentials from being sent in the clear over the network even if Voyager is not configured to use SSL for communications. The default implementation of SealedJDBCProperties and SealedHibernateCfg encrypt the database connection properties based on the Advanced Encryption Standard (AES) algorithm and employs a static internal symmetric secret key. These configuration classes can be subclassed to specify a custom secret key or use a different symmetric encryption algorithm.

Deployment Considerations

Consult the Voyager Core Developer for details concerning deploying the Voyager database features.

Sample JDBC Connection Configurations

The following sections contain examples of JDBC connection properties for several popular databases.

MySQL

```
IJDBCProperties dbCfg = new JDBCProperties()
        .setDriver("com.mysql.jdbc.Driver")
        .setURL("jdbc:mysql://localhost/<db>")
        .setPassword("<passwd>")
        .setUser("<user>");
```

SQL Server

```
IJDBCProperties dbCfg = new JDBCProperties()
        .setDriver("com.microsoft.sqlserver.jdbc.SQLServerDriver")
        .setURL("jdbc:sqlserver://localhost:1433;databaseName=<db>")
        .setPassword("<pwd>")
        .setUser("<user>");
```

Pointbase Micro

```
IJDBCProperties dbCfg = new JDBCProperties()
.setDriver("com.pointbase.me.jdbc.jdbcDriver")
.setURL("jdbc:pointbase:micro:<db>;pbmic.lic=/pointbase54/pbmic.lic")
.setUser("PBPUBLIC")
.setPassword("PBPUBLIC");
```

Appendices – Examples

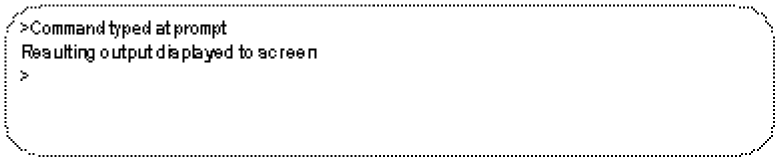
This appendix provides an overview of the examples that cover the Voyager database module features.

Running the Examples

After you install Voyager, the source code for these examples is located in the `examples` directory. Each example description specifies the directory in which the example resides. For Java the CLASSPATH must include `ve-db.jar,commons-dbutils.jar` which can be find in the $VoyagerInstallationRoot/platform/jse/lib/database directory. Also you need the examples class files to successfully run the examples. For .NET, do **not** set CLASSPATH but use the provided project file to build the examples, "Rebuild Solution", and then use a command shell to execute the example. Each example is presented as follows.

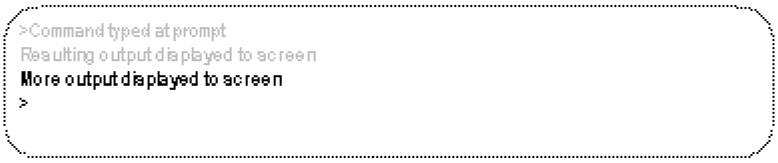
1. The command(s) used to prepare the example program for execution are presented. Commands that generate interfaces, generate holders, and compile source code belong in this category.
2. The command(s) used to run the example program are presented, followed by the program output.
3. The source code for the example programs is listed.

Commands the user types and the resulting output displayed are presented as shown below



```
>Command typed at prompt
Resulting output displayed to screen
>
```

Sometimes, not all output from a command displays to the screen at once. When subsequent output is presented, the original command and output text are shaded gray and new output is presented in bold. For example:



```
>Command typed at prompt
Resulting output displayed to screen
More output displayed to screen
>
```

Basics

In order to execute database operations in Voyager, two Objects are needed

- JDBC Configuration – contains the configuration needed to communicate with the database.
- Database Action – contains one or more SQL statements.

Hsql Example

This example illustrates using the Voyager database APIs with the HSQL database. To run this example, you need to have hsqldb.jar in your classpath. The jar file can be found at $VoyagerInstallationRoot/platform/jse/lib/database directory.

Source code location

The example can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

```
Java
java\examples\hsql\HSQLSample.java
```

Dbagent Examples

A Database Agent can be useful for aggregating data that is spread across multiple databases. Voyager provides two database agent implementations:

- An AgentAction for executing JDBC transactions
- An AgentAction for executing Hibernate transactions

JDBCAgentSample

This example creates an Agent that will execute a query agent against a HSQL database

To run this example, you need to have hsqldb.jar and ve-core.jar in your classpath. Both jar files can be found at $VoyagerInstallationRoot/platform/jse/lib/database directory.

Source code location

The example can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

java\examples\dbagent\JDBCAgentSample.java

HibernateAgentSample

A Hibernate Agent is constructed with a Hibernate-specific configuration object. Once the Agent is deployed this configuration is used to construct a Hibernate session and execute a specific command on the session.

This example demonstrates how to create and use a Hibernate Agent. The Agents shown here merely persist an object representing a log entry and then read back the object from a database.

To run this example, you need to have hsqldb.jar, commons-collections-2.1.1.jar, commons-logging-1.0.4.jar, dom4j-1.6.1.jar, hibernate3.jar, jta.jar, asm.jar, asm-attrs.jar, antlr-2.7.6.jar, cglib-2.1.3.jar and ve-core.jar in your classpath. Those jar files can be found at $VoyagerInstallationRoot/platform/jse/lib/database directory.

Source code location

The example can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

java\examples\dbagent\HibernateAgentSample.java

MySQL

This is the example of using the Voyager database APIs with the MySQL database. To run this example, you need to have mysql-connector-java-3.1.13-bin.jar in your classpath. The jar file can be found at $VoyagerInstallationRoot/platform/jse/lib/database directory.

This example assumes that an instance of the MySQL database is running locally. Furthermore, it also assumes a database named rsi exists that is accessible by a user named test. Refer to the MySQL tutorial for more details on how to create a MySQL database.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

```Java
java\examples\mysql\MySQLSample.java
```

Pointbase

This example illustrates using the Voyager database API with the Pointbase database. To run this example, you need to have the pbmicro55.jar in your classpath. You can download an eval version of pointbase at http://www.pointbase.com/.

PBMicroDataSourceSample uses Pointbase micro as the database implementation. It illustrates using the java.sql.DataSource interface rather than java.sql.DriverManager.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

```Java
java\examples\pointbase\PBMicroDataSourceSample.java
java\examples\pointbase\PBMicroDriverManagerSample.java
```