Using JPA to Persist Application Data in SAP HANA

Appplies to:
HANA Database, JAVA, JPA, JPaaS, Netweaver neo, SAP Research Security and Trust

Summary
In this document we propose a detailed solution to use the Java Persistence API framework JPA to persist Java classes in HANA DB. This action is not automatically supported, and we propose a user guide to perform it with few efforts.

Authors:
Saheed Abiola Lasisi
Jakub Sendor
Dr Slim Trabelsi

Company: SAP Research, SAP Labs, France
Created on: 22 May 2012

Author Bio

Saheed Abiola Lasisi: is an intern in the Security the Security & Trust practice at SAP Labs France Sophia-Antipolis. His thesis work is under the theme preservation of privacy and trust in cloud computing. He is currently working on accelerated privacy preservation mechanisms for enterprise cloud scenarios.

Jakub Sendor: is a developer working in the Security & Trust research practice at SAP Labs France in Sophia-Antipolis. In addition to the research prototypes development he is also a member of Security Task Force where he is involved in the internal secure programming trainings initiative.

Slim Trabelsi PhD: is a Senior researcher in the Security & Trust practice at SAP Labs France Sophia-Antipolis. He is currently managing the security WP of the EU PPP Fi-Ware project. His main research fields are: privacy, trust, could, social networks, anonymization.
### Table of Contents

Introduction ......................................................................................................................................................... 3  
Background on JPA and persistent entities ........................................................................................................ 3  
JPA Integration with SAP HANA ......................................................................................................................... 4  
  Required libraries ............................................................................................................................................ 4  
  Creation of a JDBC connection ....................................................................................................................... 4  
  Configuration of the required libraries ......................................................................................................... 6  
  Creation of a reference implementation ....................................................................................................... 9  
Data types supported by SAP HANA DB ......................................................................................................... 10  
Copyright........................................................................................................................................................... 11
Introduction

Developing web, Java, or mobile applications in the cloud strongly require a persistency layer in order to store the different dynamic data objects created or used during the execution process. This persistency layer can for example take care of the mapping from Java classes to database tables, and from Java data types to relational data types. In addition, it can provide seamless data query and retrieval facilities through Java objects.

During the migration task of our applications from JPaaS/SAP NetWeaver Neo based on MaxDB to HANA DB, we faced some problems when using the Java persistency layer based on JPA in order to automatically persist our application data in HANA DB. Such action was not yet developed or documented. After some research and hackings, we found a useful solution to use JPA persistency with HANA DB in the face of current limitations. In this document we share our experience and we provide a sort of user guide to help developers that want to migrate their JPA based applications to HANA DB without efforts.

This solution is of course not "official", it may not work for all the applications. Your comments, feedbacks, and recommendations are welcome.

Background on JPA and persistent entities

The Java Persistence API (JPA) is a framework specification established to provide a possibility to persist Java objects in the relational databases. This concept is often referred to as object-relational mapping (ORM) and is also existing in other languages and modern web frameworks, e.g. Active Records provide the implementation in Ruby on Rails, GORM is implementation for Groovy and its web framework Grails.

There are several implementations of JPA, most popular among them are Hibernate, TopLink, EclipseLink and OpenJPA. JPA implementation is called also persistence provider as it is the library task to establish and maintain the connection to the underlying database technology.

In this article we will focus mostly on EclipseLink implementation. Steps for other JPA providers should be similar - however there exist some differences between them.

JPA provides a developer with possibility to perform CRUD (create, read, update and delete) operations from relational databases directly on the Java object layer rather than using SQL which used to be a standard method of accessing databases. The translation from object calls to SQL statements is the job of JPA provider. Of course it might be sometimes necessary to perform a customized call to database, e.g. with specific grouping or filtering requirements. JPA has mechanism to create custom queries using similar syntax to SQL.

The mapping between Java objects and corresponding database tables as well as other properties are defined via the persistence metadata. The persistence provider will use the persistence metadata information to perform the correct database operations.

JPA typically defines the metadata via annotations in the Java class. Alternatively the metadata can be defined via XML or a combination of both. The XML configuration takes precedence over the annotations.

JPA entities are annotated Java classes which will be represented as tables in the database. The fields of these classes are field definitions of the created tables while instance of the class are traditionally mapped as rows of the tables.
JPA Integration with SAP HANA

Required libraries

1. EclipseLink library (can be downloaded from here: [EclipseLink Download Site](http://www.eclipse.org eclipselink/download/))
   a. Eclipselink.jar
   b. Javax.persistence_2.0.3.v*.jar

2. SAP HANA DB JDBC driver: ngdbc.jar (can be found in SAP HANA Client tools installation kit or directory)

3. SAP HANA Platform Dialect: com.sap.core.persistence.osgi.hdb.platform.jar

In the description below it is assumed that you already have all access credentials to running installation of SAP HANA DB server and know its JDBC URL.

Creation of a JDBC connection

First we start with creating a JDBC connection in the Eclipse.

1. Go to Windows -> Preferences -> Data Management -> Connectivity -> Driver Definitions -> Add…
   From the list you can choose, e.g. MAX DB 7.7, because it has similar property template.

2. Change “Driver name” to “SAP HANA JDBC Driver”.

![Figure 1: SAP HANA Driver Definition](image-url)
3. Switch to the “JAR List” tab, click “Add JAR/Zip” and add the ngdbc.jar file to the “Driver files” list (if there are any entries already there, remove them using “Clean All” button).

4. Provide the appropriate properties (connection URL, driver class, password and user id) on the next tab:
5. Click on OK when you are through.

Configuration of the required libraries

1. In your project you need to add external JARs: *.hdb.platform*.jar, ngdbc.jar and eclipselink.jar. To do that, right click on the project name, select “Properties”, then “Java Build Path” from the opened menu and there you should find “Add External JARs” on the right. Other files in the list on the screenshot can be found in the EclipseLink download and may be needed depending on what you are implementing.
2. Go to JPA on the menu list tree on the left:

3. Create and prepare the META-INF/persistence.xml file in the project with the following properties. You can either directly edit the XML code or use graphical interface.

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <persistence-unit name="JPA-HANA">
    <description>JPA-HANA</description>
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>jpa.simple.model.Tasktodo</class>
    <class>jpa.simple.model.Connectme</class>

    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.sap.db.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:sap://MY-SAP_HANA_SERVER:30115/"/>
      <property name="javax.persistence.jdbc.user" value="jafkjksd"/>
      <property name="javax.persistence.jdbc.password" value="Sirkflal1"/>
      <property name="eclipselink.target-database" value="com.sap.persistence.platform.database.HDBPlatform"/>
    </properties>
  </persistence-unit>
</persistence>
```
<!-- EclipseLink should create the database schema automatically -->
<property name="eclipselink.ddl-generation" value="create-tables"/>
<property name="eclipselink.ddl-generation.output-mode" value="both"/>
</properties>
</persistence-unit>
</persistence>

</div>
</div>
Creation of a reference implementation

To test our project we may either try some unit test framework or a main class implementation like the one below. It will try to persist and load few objects from SAP HANA DB.

The following example is taken from the JPA 2.0 with EclipseLink Tutorial (http://www.vogella.com/articles/JavaPersistenceAPI/article.html)

```java
package jpa.simple.model;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import jpa.simple.model.Tasktodo;

public class Main {
    private static final String PERSISTENCE_UNIT_NAME = "JPA-HANA";
    private static EntityManagerFactory factory;

    public static void main(String[] args) {
        factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
        EntityManager em = factory.createEntityManager();
        // Read the existing entries and write to console
        Query q = em.createQuery("select t from Tasktodo t");
        List<Tasktodo> todoList = q.getResultList();
        for (Tasktodo todo : todoList) {
            System.out.println(todo);
        }
        System.out.println("Size: "+todoList.size());

        // Create new Tasktodo
        em.getTransaction().begin();
        Tasktodo todo = new Tasktodo();
        todo.setSummary("Test Summary Great");
        todo.setDescription("Test Description Good");
        em.persist(todo);
        em.getTransaction().commit();

        //Generate members of the Connect Me Class
        em.getTransaction().begin();
        for(int i=0; i<100; i++) {
            Connectme connectMe = new Connectme();
            connectMe.setFirstname("Sagatarius"+i+150);
            connectMe.setLastname("Robasco"+i+150);
            em.persist(connectMe);
        }
        em.getTransaction().commit();
        em.close();
    }
}
```
Data types supported by SAP HANA DB

It is important that we examine the data types supported by SAP HANA DB because some of them that are supported by most DBMS are not available in HANA and we may need to improvise to circumvent such an issue.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime types</td>
<td>DATE, TIME, SECONDDATE, TIMESTAMP</td>
</tr>
<tr>
<td>Numeric types</td>
<td>TINYINT, SMALLINT, INTEGER, BIGINT, SMALLDECIMAL, DECIMAL, REAL, DOUBLE, FLOAT</td>
</tr>
<tr>
<td>Character string types</td>
<td>VARCHAR, NVARCHAR, ALPHANUM</td>
</tr>
<tr>
<td>Binary types</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>Large Object types</td>
<td>BLOB, CLOB, NCLOB</td>
</tr>
</tbody>
</table>

(SAP HANA Database = SQL Reference Manual)

Any other data type definition in an entity file apart from the ones above would cause exceptions at runtime because they are not defined in HANA (e.g. `@Column(columnDefinition="LONG VARCHAR")`).

NB: Column tables cannot be generated by JPA Entities as for now (no provider’s support Column table annotations yet).