Enterprise Applications Tutorial

Fast notes on using JPA, Hibernate, EJB, and Enterprise Archives

Adapted from http://eai-course.blogspot.pt/2012/04/java-persistence-api-with-eclipse.html

In this tutorial we learn how to use the Java Persistence API (JPA), supported by Hibernate and PostgreSQL, in two environments: **standalone** and **JBoss AS 7** (respectively in Part I and Part II of this tutorial). Eclipse IDE for Java EE Developers is the development environment used throughout this document.

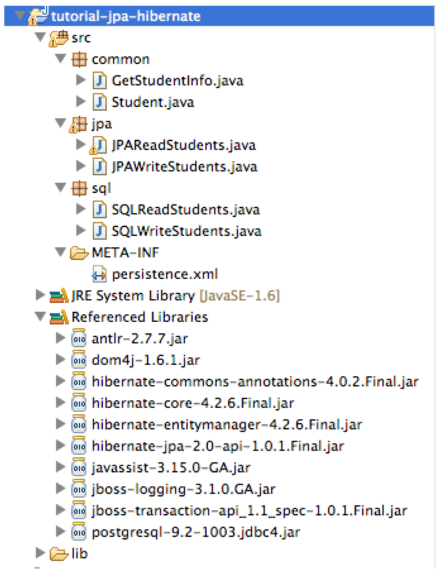
Software

Hibernate 4.2.6.Final – https://sourceforge.net/projects/hibernate/files/hibernate4
JBoss AS 7.1.1.Final – http://www.jboss.org/jbossas/downloads/
Eclipse IDE for Java EE Developers – http://www.eclipse.org/downloads/
PostgreSQL 9.x – http://www.postgresql.org/download/
PostgreSQL JDBC Driver – http://jdbc.postgresql.org/download.html

Part I – Using JPA in a Standalone application

Create a Simple Java Project in Eclipse. Now create a 'lib' directory and drag and drop the Hibernate jar files (see the next Figure to know which specific jars you need – you will find these inside the Hibernate zip distribution) and the PostgreSQL JDBC Driver jar in the 'lib' directory.

Let's now configure the build path to make use of the jar files. Go to the project properties (you can right-click your project for this) and choose Java Build Path -> Add Jars. Select and add all the Jars files that you just dragged and dropped into the project. Now create a persistence.xml file under a src/META-INF directory. The next figure shows the final project layout for your reference (later we will create the Java classes that are already visible in the figure).



```
tutorial-jpa-hibernate
  src
    common
      GetStudentInfo.java
      Student.java
    jpa
      JPAReadStudents.java
      JPAWriteStudents.java
    sql
      SQLReadStudents.java
      SQLWriteStudents.java
    META-INF
      persistence.xml
  JRE System Library [JavaSE-1.6]
  Referenced Libraries
    antlr-2.7.7.jar
    dom4j-1.6.1.jar
    hibernate-commons-annotations-4.0.2.Final.jar
    hibernate-core-4.2.6.Final.jar
    hibernate-entitymanager-4.2.6.Final.jar
    hibernate-jpa-2.0-api-1.0.1.Final.jar
    javassist-3.15.0-GA.jar
    jboss-logging-3.1.0.GA.jar
    jboss-transaction-api_1.1_spec-1.0.1.Final.jar
    postgresql-9.2-1003.jdbc4.jar
  lib
```

The following figure shows the contents of your persistence.xml file, where we configure several details regarding the persistence provider. Note that you may have to change the url, the driver, the user name and the password (or other elements) according to your system's needs. You may also want to change the table generation policies.

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
        xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
        <persistence-unit name="TestPersistence"
                transaction-type="RESOURCE_LOCAL"> <!-- use transaction-type="JTA" -->
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                 <!-- With a standard configuration and hibernate, we don't need to identify the Entity classes here -->
                <class>common.Student</class>
                <properties>
                        <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
                        <!-- possible values: validate, update, create, create-drop -->
                        <property name="hibernate.hbm2ddl.auto" value="update" />
                        <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
                        <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost/postgres" />
                        <property name="javax.persistence.jdbc.user" value="postgres" />
                        <property name="javax.persistence.jdbc.password" value="postgres" />
                </properties>
        </persistence-unit>
</persistence>
```

Before we write anything to the database, we need to define the classes that will hold our data. These will belong to the **common** package. Note that the Student class is ready for "JPA time", as it already contains all the annotations.

```
package common;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "STUDENTS_TABLE")
public class Student implements Serializable
{

        private static final long serialVersionUID = 1L;
        // we use this generation type to match that of SQLWriteStudents
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "studentid")
        private Long id;
        private String name;
        private String phone;

        public Student()
        {
        }

        public Student(String name, String phone)
        {
                this.name = name;
                this.phone = phone;
        }

        public Long getId()
        {
                return id;
        }

        public void setId(Long id)
        {
                this.id = id;
        }

        public String getName()
        {
                return name;
        }

        public void setName(String name)
        {
                this.name = name;
        }

        public String getPhone()
        {
                return phone;
        }

        public void setPhone(String phone)
        {
                this.phone = phone;
        }
```

```java
	@Override
	public int hashCode()
	{
		int hash = 0;
		hash += (id != null ? id.hashCode() : 0);
		return hash;
	}

	@Override
	public boolean equals(Object object)
	{
		// TODO: Warning - this method won't work in the case the id fields are not set
		if (!(object instanceof Student))
		{
			return false;
		}
		Student other = (Student) object;
		if ((this.id == null && other.id != null)
				|| (this.id != null && !this.id.equals(other.id)))
		{
			return false;
		}
		return true;
	}

	@Override
	public String toString()
	{
		return "Student " + id + ": " + name + " " + phone;
	}
}
```

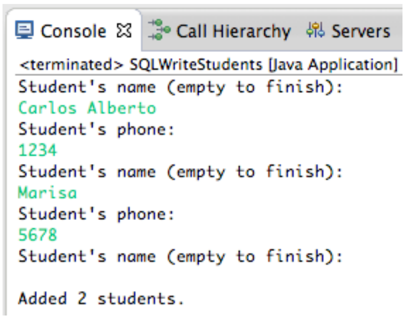
```java
package common;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class GetStudentInfo
{
	static List<Student> list;

	public static List<Student> get()
	{
		list = new ArrayList<Student>();
		boolean done = false;
		Scanner sc = new Scanner(System.in);
		while (!done)
		{
			System.out.println("Student's name (empty to finish): ");
			String name = sc.nextLine();
			done = name.equals("");
			if (!done)
			{
				System.out.println("Student's phone: ");
				String phone = sc.nextLine();
				list.add(new Student(name, phone));
			}
		}
		return list;
	}
}
```

We will access the database using two different ways, for comparison purposes: 1) using standard SQL; 2) using JPA. Furthermore, they will interoperate. Let's start with option 1), a class named *SQLWriteStudents* that writes to the database using SQL:

```java
package sql;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import common.GetStudentInfo;
import common.Student;

public class SQLWriteStudents
{
	private final static String name = "postgres";
	private final static String password = "postgres";
	private final static String tablename = "STUDENTS_TABLE";
	private final static String url = "jdbc:postgresql://localhost/postgres";

	public static void main(String[] args) throws SQLException
	{
		Connection conn = (Connection) DriverManager.getConnection(url, name, password);
		Statement stmt = (Statement) conn.createStatement();

		String sql = "CREATE TABLE IF NOT EXISTS "
			+ tablename
			+ "(studentid SERIAL NOT NULL , Name VARCHAR(254), Phone VARCHAR(20), PRIMARY KEY (studentid))";
		stmt.executeUpdate(sql);

		List<Student> mylist = GetStudentInfo.get();

		int rows = 0;
		for (Student st : mylist)
		{
			sql = "INSERT INTO " + tablename + "(Name, Phone)" + "VALUES"
					+ "('" + st.getName() + "','" + st.getPhone() + "')";

			rows += stmt.executeUpdate(sql);
		}

		System.out.println("Added " + rows + " students.");
	}
}
```

Note that this tutorial uses the ‘postgres’ database that is created by default with every typical PostgreSQL installation, the default user ‘postgres’ and password ‘postgres’, but you can use your own. If you have the right configuration, try to run it now. It should already work.

In the following figure, we’re adding two students to the database. Try this also and check in PGAdmin (a graphical DBMS management tool that comes with the PostgreSQL distribution) if the data has been added properly.



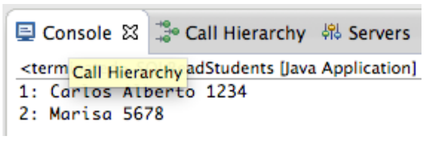
```
<terminated> SQLWriteStudents [Java Application]
Student's name (empty to finish):
Carlos Alberto
Student's phone:
1234
Student's name (empty to finish):
Marisa
Student's phone:
5678
Student's name (empty to finish):

Added 2 students.
```

Let's now try to programmatically check if the data was added to the database (i.e., without using PGAdmin):

```
package sql;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class SQLReadStudents
{

	private final static String name = "postgres";
	private final static String password = "postgres";
	private final static String tablename = "STUDENTS_TABLE";
	private final static String url = "jdbc:postgresql://localhost/postgres";

	public static void main(String[] args)
	{
```

```
                try
                {
                        Connection conn = (Connection) DriverManager.getConnection(url,
                                        name, password);
                        Statement stmt = (Statement) conn.createStatement();
                        ResultSet rs;

                        rs = stmt.executeQuery("SELECT * FROM " + tablename);
                        while (rs.next())
                        {
                                Long id = rs.getLong("studentid");
                                String nme = rs.getString("Name");
                                String phone = rs.getString("Phone");
                                System.out.println(id + ": " + nme + " " + phone);
                        }
                        conn.close();
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
        }
}
```

The result is the following:

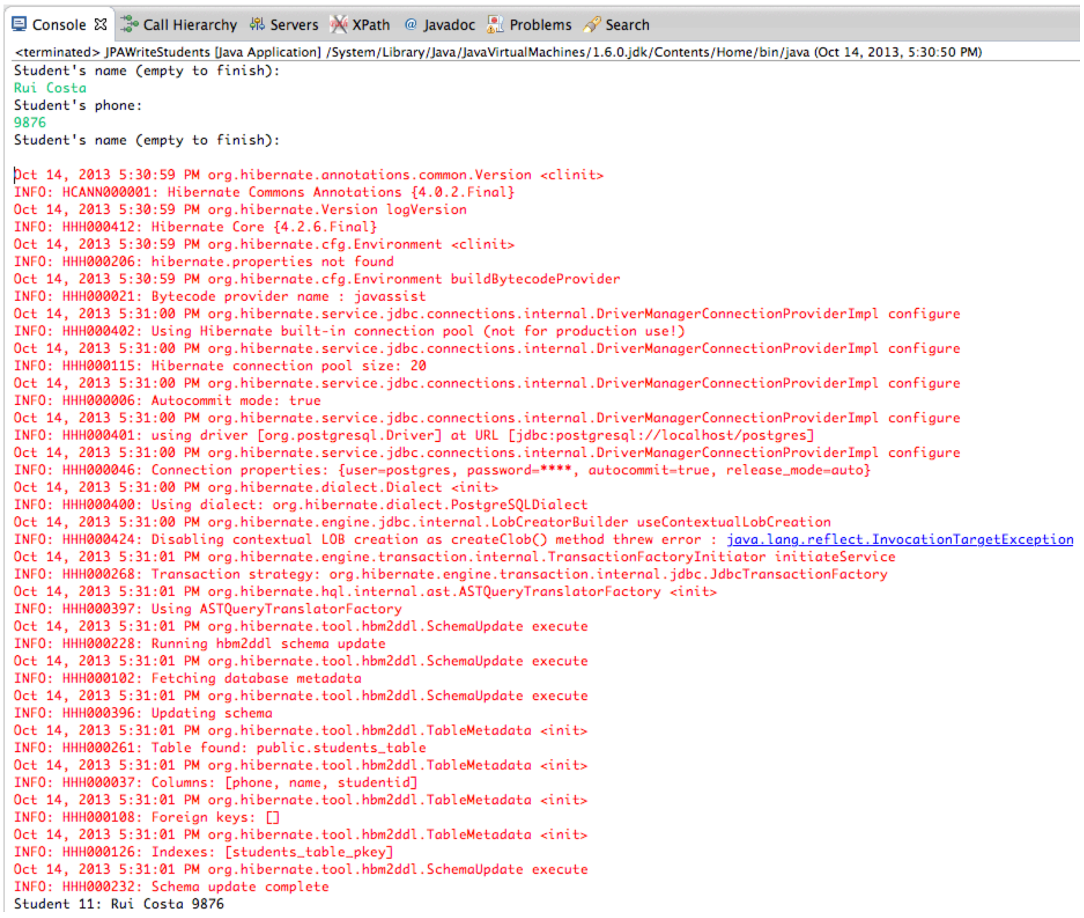


```
<term                 adStudents [Java Application]
1: Carlos Alberto 1234
2: Marisa 5678
```

But the real motivation for this tutorial was JPA. Therefore, let's write two correspondent classes, one to write, the other to read from the Student Database (table STUDENTS_TABLE, as you can see from the Student entity class above). First, the JPAWriteStudents class:

```
package jpa;

import common.Student;
import common.GetStudentInfo;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class JPAWriteStudents
{

        public static void main(String[] args)
        {
                List<Student> mylist = GetStudentInfo.get();

                EntityManagerFactory emf = Persistence
                                .createEntityManagerFactory("TestPersistence");
                EntityManager em = emf.createEntityManager();
                EntityTransaction tx = em.getTransaction();

                tx.begin();
                for (Student st : mylist)
                        em.persist(st);
                tx.commit();

                // after commit we have ids:
                for (Student st : mylist)
                        System.out.println(st);
        }
}
```

We can add new students, running the JPAWriteStudents application that we just created:

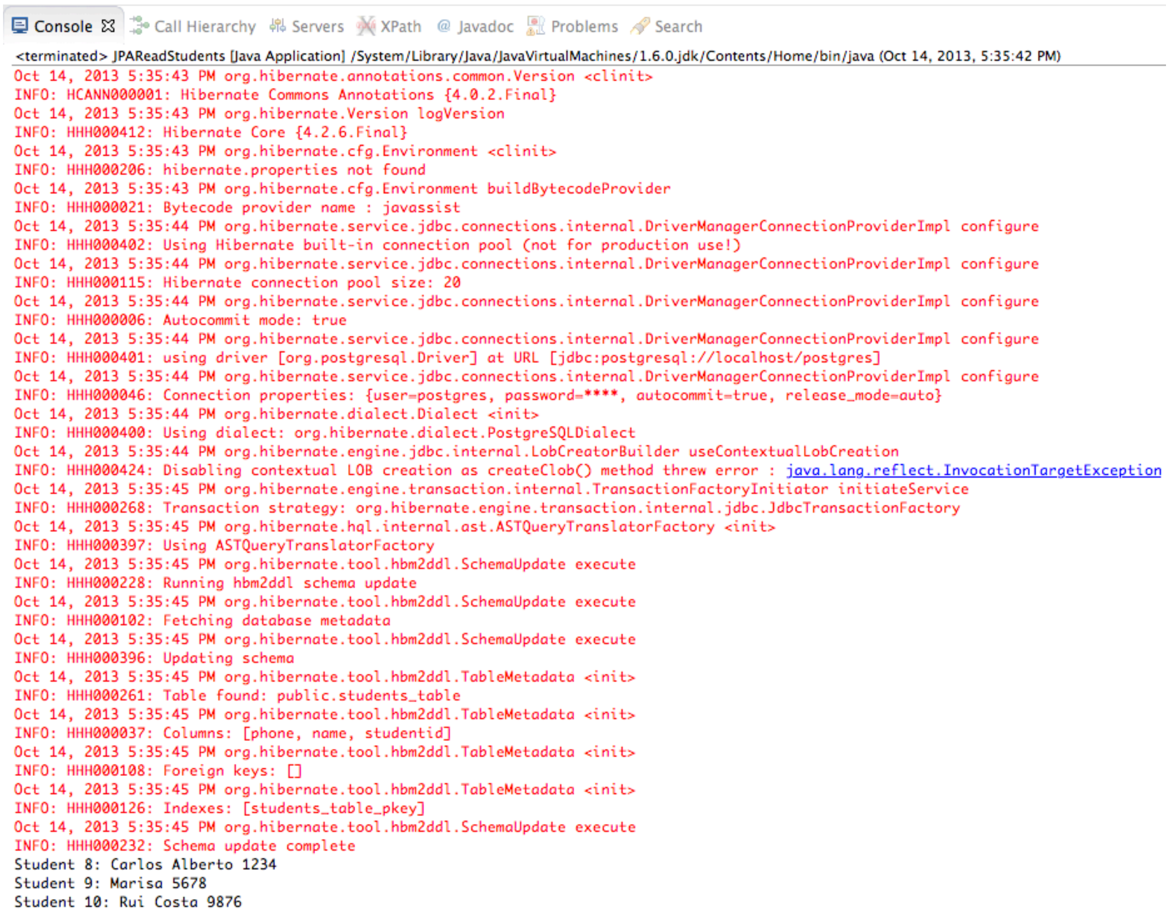


```
<terminated> JPAWriteStudents [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 14, 2013, 5:30:50 PM)
Student's name (empty to finish):
Rui Costa
Student's phone:
9876
Student's name (empty to finish):

Oct 14, 2013 5:30:59 PM org.hibernate.annotations.common.Version <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {4.0.2.Final}
Oct 14, 2013 5:30:59 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {4.2.6.Final}
Oct 14, 2013 5:30:59 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Oct 14, 2013 5:30:59 PM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
Oct 14, 2013 5:31:00 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000402: Using Hibernate built-in connection pool (not for production use!)
Oct 14, 2013 5:31:00 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000115: Hibernate connection pool size: 20
Oct 14, 2013 5:31:00 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000006: Autocommit mode: true
Oct 14, 2013 5:31:00 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000401: using driver [org.postgresql.Driver] at URL [jdbc:postgresql://localhost/postgres]
Oct 14, 2013 5:31:00 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000046: Connection properties: {user=postgres, password=****, autocommit=true, release_mode=auto}
Oct 14, 2013 5:31:00 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.PostgreSQLDialect
Oct 14, 2013 5:31:00 PM org.hibernate.engine.jdbc.internal.LobCreatorBuilder useContextualLobCreation
INFO: HHH000424: Disabling contextual LOB creation as createClob() method threw error : java.lang.reflect.InvocationTargetException
Oct 14, 2013 5:31:01 PM org.hibernate.engine.transaction.internal.TransactionFactoryInitiator initiateService
INFO: HHH000268: Transaction strategy: org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory
Oct 14, 2013 5:31:01 PM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory <init>
INFO: HHH000397: Using ASTQueryTranslatorFactory
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000228: Running hbm2ddl schema update
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000102: Fetching database metadata
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000396: Updating schema
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000261: Table found: public.students_table
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000037: Columns: [phone, name, studentid]
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000108: Foreign keys: []
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000126: Indexes: [students_table_pkey]
Oct 14, 2013 5:31:01 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000232: Schema update complete
Student 11: Rui Costa 9876
```

And let's now read from the database using the following JPAReadStudents class:

```java
package jpa;

import common.Student;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JPAReadStudents
{
        public static void main(String args[])
        {
                EntityManagerFactory emf = Persistence
                                .createEntityManagerFactory("TestPersistence");
                EntityManager em = emf.createEntityManager();

                String query = "SELECT s FROM Student s";

                @SuppressWarnings("unchecked")
                List<Student> mylist = (List<Student>) em.createQuery(query)
                                .getResultList();
                for (Student st : mylist)
                        System.out.println(st);
        }
}
```

We get the following after we ran the JPAReadStudents class:



```
<terminated> JPAReadStudents [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Oct 14, 2013, 5:35:42 PM)
Oct 14, 2013 5:35:43 PM org.hibernate.annotations.common.Version <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {4.0.2.Final}
Oct 14, 2013 5:35:43 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {4.2.6.Final}
Oct 14, 2013 5:35:43 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Oct 14, 2013 5:35:43 PM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
Oct 14, 2013 5:35:44 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000402: Using Hibernate built-in connection pool (not for production use!)
Oct 14, 2013 5:35:44 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000115: Hibernate connection pool size: 20
Oct 14, 2013 5:35:44 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000006: Autocommit mode: true
Oct 14, 2013 5:35:44 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000401: using driver [org.postgresql.Driver] at URL [jdbc:postgresql://localhost/postgres]
Oct 14, 2013 5:35:44 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000046: Connection properties: {user=postgres, password=****, autocommit=true, release_mode=auto}
Oct 14, 2013 5:35:44 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.PostgreSQLDialect
Oct 14, 2013 5:35:44 PM org.hibernate.engine.jdbc.internal.LobCreatorBuilder useContextualLobCreation
INFO: HHH000424: Disabling contextual LOB creation as createClob() method threw error : java.lang.reflect.InvocationTargetException
Oct 14, 2013 5:35:45 PM org.hibernate.engine.transaction.internal.TransactionFactoryInitiator initiateService
INFO: HHH000268: Transaction strategy: org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory
Oct 14, 2013 5:35:45 PM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory <init>
INFO: HHH000397: Using ASTQueryTranslatorFactory
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000228: Running hbm2ddl schema update
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000102: Fetching database metadata
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000396: Updating schema
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000261: Table found: public.students_table
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000037: Columns: [phone, name, studentid]
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000108: Foreign keys: []
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.TableMetadata <init>
INFO: HHH000126: Indexes: [students_table_pkey]
Oct 14, 2013 5:35:45 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000232: Schema update complete
Student 8: Carlos Alberto 1234
Student 9: Marisa 5678
Student 10: Rui Costa 9876
```

You should use this quick setup to learn JPA, but then move on to the JBoss managed environment described in the next section.

Part II – Using JPA in a JBoss managed environment

Go to the JBoss Home directory and find modules\org folder. Create two new folders, so you get modules\org\postgresql\main and insert the PostgreSQL JDBC driver .jar file there. In this same folder create a file named module.xml with the following content (set in <resource-root path="" > the exact name of your .jar-file):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.0" name="org.postgresql">
        <resources>
                <resource-root path="postgresql-9.2-1003.jdbc4.jar"/>
        </resources>
        <dependencies>
                <module name="javax.api"/>
                <module name="javax.transaction.api"/>
        </dependencies>
</module>
```

Edit JBoss's standalone-full.xml and add the following code between the <drivers></drivers> tags.

```xml
<driver name="postgresql" module="org.postgresql">
        <xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-datasource-class>
</driver>
```

Now start the server and check if it is loading the driver. The console should show something like:

```
INFO  [org.jboss.as.connector.subsystems.datasources] (ServerService Thread Pool -- 33) JBAS010404:
Deploying non-JDBC-compliant driver class org.postgresql.Driver (version 9.2)
```

Let's now add a datasource in standalone-full.xml. Add the following code between the <datasources></datasources> tags (set your connection-url, username and password, jndi-name can be *java:/jboss/datasources/postgresDS*.

```
<datasource jndi-name="java:jboss/datasources/postgresDS" pool-name="postgresDS"
        enabled="true" jta="true" use-java-context="true" use-ccm="false">
        <connection-url>jdbc:postgresql://localhost:5432/postgres</connection-url>
        <driver-class>org.postgresql.Driver</driver-class>
        <driver>postgresql</driver>
        <pool>
                <min-pool-size>2</min-pool-size>
                <max-pool-size>20</max-pool-size>
        </pool>
        <security>
                <user-name>postgres</user-name>
                <password>postgres</password>
        </security>
        <validation>
                <validate-on-match>false</validate-on-match>
                <background-validation>false</background-validation>
                <background-validation-millis>60000</background-validation-millis>
        </validation>
        <statement>
                <prepared-statement-cache-size>0</prepared-statement-cache-size>
                <share-prepared-statements>false</share-prepared-statements>
        </statement>
</datasource>
```

Now create an EJB in a new **EJB Project** – you should check the **course blog and additional references** to learn how to create this type of project and also a client for it. Once you got both the EJB and EJB-client working, try to add JPA capabilities to your EJB (to simplify this explanation, for now we will have EJB and JPA in the same project) and make it write a `Student` instance to the database. For this, follow the next **4 steps**:

1) Copy the `Student` Entity to your EJB project.
2) Copy the following simpler (why?) META–INF/persistence.xml file to your EJB project:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
        xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
        <persistence-unit name="TestPersistence" transaction-type="JTA">
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <jta-data-source>java:jboss/datasources/postgresDS</jta-data-source>
                <properties>
                        <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
                        <property name="hibernate.hbm2ddl.auto" value="update" />
                        <property name="hibernate.transaction.jta.platform"
                                value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform" />
                </properties>
        </persistence-unit>
</persistence>
```

3) Use dependency injection to inject the entity manager in your EJB (check the following example for details). Use the injected entity manager to persist a `Student` instance to the database.

In a JBoss managed environment, you can use dependency injection to inject references to different objects, including entity managers (using the `@PersistenceContext` annotation) and EJBs (using, for instance, the `@EJB` annotation). The following code shows an example of how to inject an `EntityManager instance` using the `@PersistenceContext` annotation. The container will inject the reference at runtime and you will be able to use the `EntityManager` without further initialization code.

```
@Stateless
public class MyEjb implements MyEjbRemote
{

        @PersistenceContext(name = "TestPersistence")
        private EntityManager em;

        public … doSomething(…)
        {
                Student s = new Student();
                s.setName("x");
                s.setPhone("x");

                // em doesn't need to be 'manually' created, the container will inject a reference for us to use!
                em.persist(s);
        }
```

4) Deploy your EJB in JBoss and use the client to test it (invoke the EJB operation that persists the `Student` to the database). Check the result in PgAdmin.

Enterprise applications are more complex and are many times composed of different tiers. When developing such applications it's helpful to keep separate projects to decrease the complexity of managing the whole system. For instance, an enterprise application project can be composed of: **1)** a *JPA Project* that can hold Entities and persistence configuration; **2)** an *EJB Project* that implements the business logic; and **3)** a *Dynamic Web Project* (e.g., with Servlets/JSP) that takes care of the presentation layer. A good option to manage and deploy such complex projects is to use Eclipse and **create an Enterprise Application Project**, which you can deploy as a **.EAR** (Enterprise ARchive) file in JBoss. This Enterprise Application Project can include references to other projects (i.e., your .EAR project can reference an EJB Project which in turn can reference a JPA project). Check the **course blog and additional references** for further details on creating this type of solution.

References

References used to create this tutorial:
EAI Course Blog – JPA tutorial: http://eai-course.blogspot.pt/2012/04/java-persistence-api-with-eclipse.html
https://community.jboss.org/blogs/amartin-blog/2012/02/08/how-to-set-up-a-postgresql-jdbc-driver-on-jboss-7

EAI Course Blog references:
EJB Project: http://eai-course.blogspot.pt/2012/10/a-simple-enterprise-javabeans-31.html
Enterprise Application Project: http://eai-course.blogspot.pt/2012/04/creating-enterprise-application.html

Additional references:
https://zorq.net/b/2011/07/12/adding-a-mysql-datasource-to-jboss-as-7/
https://community.jboss.org/wiki/DataSourceConfigurationinAS7
https://docs.jboss.org/author/display/AS7/Admin+Guide#AdminGuide-Datasources
http://www.vogella.com/articles/JavaPersistenceAPI/article.html

Books:
In addition to the links above, the following books are highly recommended:

- Enterprise JavaBeans 3.1 (6th Edition), Andrew Lee Rubinger, Bill Burke (O'Reilly), ISBN 0596158025, 2010.
- the Java EE 6 Tutorial, Oracle, http://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf