PriDE 3 User Manual

Table of Contents

What is PriDE

PriDE ist the Java world's smallest object-relational mapper for SQL databases. O/R mapping is the wide-spread approach to map records of a relation SQL database to objects of an object-oriented application. The application should operate on its persistent entities as object-oriented as possible, not regarding that some of them come from a database or must be saved in one. PriDE provides functionality to

- Describe the mapping of database tables to Java classes
- Read and write data records without accessing the complicated JDBC interface, and - as far as possible - don't write any SQL at all
- Simplify the assembly of complicated expressions and selection conditions

While O/R mapping is usually based on single-object operations, PriDE also supports efficient database mass processing within Java. The goal is to avoid moving application logic into procedures within the database as far as possible and not to break the DRY principle. However, if stored procedures and functions are required sometimes, PriDE provides a convenient way to call them.

PriDE was designed for usage in JSE and JEE environments and is used identically everywhere except some initialization operations and the transaction management. The framework follows a very pragmatic approach to provide basic development support quickly and easily. It does not claim to conform with established persistence management standards but follows common design patterns and proved to be suitable in mission-critical projects over many years. The detailed feature list may help to figure out whether PriDE meets the requirements of individual development projects, and allows to roughly compare this toolkit with existing well-known O/R mapping products and standards like JPA, JOOQ or MyBatis.

PriDE is so small that it can actually be understood in any single line of its code, providing the developer full control over how data is exchanged between an SQL database and a Java application. The runtime library is less than 200 kByte in size without any dependencies beside the JDBC driver library of the database in use.

Did you ever wonder how to conveniently access your SQLite database in a mobile application which has to keep its footprint small? Well, here is your answer :-)

Or did you ever worked in a multi-million lines of code project and got the feeling that JPA magic causes more loss of control than convenience? Guess what the alternative may be.

The chapter PriDE design principles gives an overview about the concepts which the framework is based on. However, before diving into more theory, it is recommended to walk through the quick start tutorial and get into touch with the real world.

Quick Start Tutorial

Up and working in 15 minutes

This short tutorial gives an introduction into the general working principles of PriDE, based on a simple example. It takes less than half an hour to set up a simple PriDE application which allows to perform basic operations on a single database table. The directory examples/quickstart of the PriDE delivery contains the complete source code for the tutorial example.

Setting up an application includes the following steps:

- Preparing the development project
- Database table design
- Writing or generating entity classes
- Writing application classes
- Calling the application

I.e. there’s only a few minutes time for each step now, so let’s hurry up ;-)

Preparing the development project

Working with PriDE requires to add the library pride.jar into the CLASSPATH of the working environment, as well as the JDBC driver of the database to access. E.g. in case of a MySQL 6 database this is the library mysql-connector-java-6.0.6.jar, for Oracle 11 the library ojdbc8.jar, and for HSQL 2.x the library hsqldb-2.x.y.jar. The ultra light in-memory database HSQL is the best choice for first experiments. Driver class, database URL, database user, and password are supposed to be provided as system properties in this tutorial examples. For HSQL and its default database, the properties look like this:

```
pride.driver=org.hsqldb.jdbc.JDBCDriver
pride.db=jdbc:hsqldb:test
pride.user=sa
pride.password=root
pride.dbtype=hsql
pride.logfile=sql.log
```

Providing the DB type is recommended, to keep PriDE from making a wrong guess, and logging all SQL operations is usually a good idea - especially for beginners.

Database table design

PriDE follows a database-first approach, so in the next step, the required database table must be designed. PriDE does not provide its own tool for that but assumes one being included in your database installation. If nothing appropriate is around, their are lots of tools available for that, e.g. the free DB Designer online tool which supports various common databases. The tutorial examples uses a database table according to the following definition:

```
create table CUSTOMER (
    id integer not null primary key,
    name varchar(20),
    first_name varchar(30)
);
```

Add this table now to your HSQL database, using HSQL's command shell.

Writing or generating entity classes

Accessing a table via PriDE requires a corresponding entity class (usually a simple Java Bean) and a mapping descriptor object. 1:1 mappings of a database table to a Java class can be generated with a code generator provided with PriDE. For the table CUSTOMER above, the source code for a corresponding entity class Customer.java and an incorporated descriptor can be generated by the following call:

```
java
-Dpride.driver=org.hsqldb.jdbc.JDBCDriver
-Dpride.db=jdbc:hsqldb:test
-Dpride.user=sa
-Dpride.password=root
-Dpride.dbtype=hsql
-Dpride.logfile=sql.log
pm.pride.util.generator.EntityGenerator CUSTOMER quickstart.Customer >
Customer.java
```

Note that you may also generate the descriptive parts in a *separate* class to keep the entity bean class free from database aspects. For the tutorial example under examples/quickstart we generate a hybrid class which looks like this:

```
public class Customer extends MappedObject<Customer> {
    public static final String TABLE = "CUSTOMER";
    public static final String COL_ID = "id";
    public static final String COL_NAME = "name";
    public static final String COL_FIRST_NAME = "first_name";

    protected static final RecordDescriptor red = new RecordDescriptor
        (Customer.class, TABLE, null, new String[][] {
            { COL_ID,   "getId",   "setId" },
            { COL_NAME,   "getName",   "setName" },
            { COL_FIRST_NAME,   "getFirstName",   "setFirstName" },
        });

    public RecordDescriptor getDescriptor() { return red; }

    private static String[] keyFields = new String[] { COL_ID };
    public String[] getKeyFields() { return keyFields; }

    private long id;
    private String name;
    private String firstName;

    // Read access functions
    public long getId()   { return id; }
    public String getName()   { return name; }
    public String getFirstName()   { return firstName; }

    // Write access functions
    public void setId(long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setFirstName(String firstName) { this.firstName = firstName;
}



    // Reconstructor
    public Customer(long id) throws SQLException {
        setId(id);
        find();
    }

    public Customer() {}

}
```

Without going into details now, you can see an important design principle of PriDE: the mapping descriptor is code. You won't find any descriptive languages included in PriDE -

neither XML nor property nor JSON files. Everything in PriDE is Java code and can be examined with a debugger if necessary.

Writing application classes

Based on the entity classes, you can design the actual application. First of all the PriDE runtime library must be initialized by a so-called "resource accessor". A JSE application requires only a single line of code for an initialization based on system properties:

```
ResourceAccessorJSE.fromSystemProperties();
```

The database operations are performed by invoking corresponding member functions of the entity classes, e.g.

```
public void create(int id, String name, String firstName)
    throws SQLException {
    Customer c = new Customer(id, name, firstName);
    c.create();
}

public void update(int id, String name, String firstName)
    throws SQLException {
    Customer c = new Customer(id, name, firstName);
    c.update();
}

public void queryByName( String name )
    throws SQLException {
    Customer c = new Customer(0, name, null);
    ResultIterator ri = c.query(COL_NAME);
    if (ri != null) {
        do {
            System.out.println(
                c.getId() + ": " +
                c.getName() + "," +
                c.getFirstName());
        } while(ri.next());
    }
}
```

The tutorial example under examples/quickstart includes the file CustomerClient.java, providing an interactive test client. Calling the client with its system property based initialization looks like this:

```
java
-Dpride.driver=org.hsqldb.jdbc.JDBCDriver
-Dpride.db=jdbc:hsqldb:test
-Dpride.user=sa
-Dpride.password=root
-Dpride.dbtype=hsql
```

```
-Dpride.logfile=sql.log
quickstart.CustomerClient
```

Play around with the client and then check the working directory. You will find a file sql.log created by PriDE which logs all the SQL statements that resulted from your persistence operation calls. The log file is plain SQL, so if you encounter any unexpected persistence behavior in you application, you can copy the commands from the log and run them from you database's SQL shell. This is a big advantage over the command logging of most other persistence frameworks.

That's it! The tutorial example already introduces the most important basic elements of PriDE. To understand what's going on behind the scenes of the 5 steps above, you will find all aspects explained in detail in the manual.

Entity, Adapter, and Descriptor

The concept of O/R mapping requires three basic building blocks:

- Entity classes which representing data in SQL tables - in the most common usage one entity object represents one record in one SQL table
- Descriptors, describing how the entity classes map to the database
- An adapter which reads data from the database to entities (select) and writes data from entities to the database (insert, update, delete)

There are very different approaches around how to express the descriptor. JPA uses annotations on entity classes, MyBatis uses XML files, and PriDE follows a different approach as you may have seen already from the quick start tutorial. The descriptor is an instance of class pm.pride.RecordDescriptor, i.e. it is code itself. This concept has a few advantages over other approaches.

- It does not clutter the entity classes with database details, so entities can be passed around in the application without violating the information hiding principle. If you are familiar with JPA you may have experienced the problem that mapping annotations can pile up to an annoying amount.
- Its not written in a different language which is always hard to keep in sync with the Java code. This becomes a serious problem when applications grow over time. If you are only working with three database tables, you won't have this problem, of course.
- If it is Java code, it can be tied to any other related Java code by using shared constants for table named and row names and so on. This allows you to easily keep track of dependencies in the code. E.g. if you remove a column from a database table you will remove the appropriate constant in the code and every mentionable IDE will immediately lead you to all the places in the code that do not compile any more. This will include the descriptor as well as all the database queries in the code that refer to that column.
- Descriptors may also be assembled dynamically at runtime. You hopefully will not often run into situations where you need that, but its good to know that there is no limit on that.

A coded descriptor needs to go somewhere in your code, of course. PriDE provides two default patterns for the descriptor placement which are obvious when you think of the building blocks mentioned above: descriptors within adapter classes or descriptors within entity classes.

Descriptors in entities is what you know already from the quick start tutorial. It cases the entities to become their own adapters having their own persistence methods. This is a compact pattern which is suitable for small applications. Therefore you will find it spread over most examples provided with PriDE. The disadvantage is the same one mentioned above with JPA: the entity classes spread knowledge about database mapping information all over the code. Combined with persistence capabilities directly incorporated in entity classes, this is a questionable concept in bigger architectures.

Let's have a look on the more sophisticated pattern of separate adapter classes. You can have a look on the general structure by generating separate classes for the quick start example table. The pure entity class can be generated by the following command:

```
java
-D... see quick start tutorial
pm.pride.util.generator.EntityGenerator CUSTOMER quickstart.Customer -b >
Customer.java
```

The parameter -b tells the generator to create only an entity class without descriptor. The result is an ordinary POJO class:

```
public class Customer implements Cloneable, java.io.Serializable {
    private long id;
    private String name;
    private String firstName;

    public long getId()   { return id; }
    public String getName()   { return name; }
    public String getFirstName()   { return firstName; }

    public void setId(long id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setFirstName(String firstName) { this.firstName = firstName;
}

    public Customer(long id) {
        setId(id);
    }

    public Customer() {}
}
```

The "re-constructor" is an additional constructor getting passed a value for all the attributes making up the entity's primary key. This of interest for find operations.

Generating the corresponding adapter class looks like this:

```
java
-D... see quick start tutorial
pm.pride.util.generator.EntityGenerator CUSTOMER quickstart.CustomerDBA
quickstart.Customer > CustomerDBA.java
```

The first parameter after the table name specifies the class to generate - in this case a class called CustomerDBA in package quickstart. The second parameter is the name of a entity class the adapter should refer to. The result looks like this:

```
public class CustomerDBA extends ObjectAdapter<Customer> {
    public static final String TABLE = "CUSTOMER";
    public static final String COL_ID = "id";
    public static final String COL_NAME = "name";
    public static final String COL_FIRST_NAME = "first_name";

    protected static final RecordDescriptor red = new RecordDescriptor
        (Customer.class, TABLE, null, new String[][] {
            { COL_ID,   "getId",   "setId" },
            { COL_NAME,   "getName",   "setName" },
            { COL_FIRST_NAME,   "getFirstName",   "setFirstName" },
        });

    public RecordDescriptor getDescriptor() { return red; }

    private static String[] keyFields = new String[] { COL_ID };
    public String[] getKeyFields() { return keyFields; }

    CustomerDBA(Customer entity) { super(entity); }
}
```

All what the adapter class has to provide is a RecordDescriptor and an optional list of column names making up the entities primary key. Based on that, the class inherits all entity-related persistence capabilities from class pm.pride.ObjectAdapter. Adapters always operate on an instance of the entity class which must be passed in the adapter’s constructor. Finding a customer by its primary ID looks like this when using separate adapter classes:

```
// Create a customer entity, initialized with a primary key value of 1
Customer customer = new Customer(1);

// Create an adapter based on the entity
CustomerDBA dba = new CustomerDBA(customer);

// Call the adapter's find method to find a customer by primary key 1.
// The primary key value is read from the entity passed in the adapters
constructor
// The result (if any) is written to the same entity
dba.find();
```

As you see, every persistence operation now requires one additional line of code to create the adapter. Especially when you design a multi-threaded application, it is important to know that adapter and entity instances are not supposed to be shared among multiple threads. So creating new instances in every operation is the prefered technique and is usually not a considerable code complication.

If you want to minimize the amount of code, you are free to invent your own adapter concept. Have a look on the base classes pm.pride.ObjectAdapter for the adapter above and pm.pride.MappedObject for the hybrid variant from the quick start tutorial. Both are minimalistic implementations of the mix-in pm.pride.DatabaseAdapterMixin which is the actual provider for all entity-related persistence operations. It is in turn based on the static methods of the class pm.pride.DatabaseAdapter. Using this class or the mix-in you could easily produce a generic adapter being responsible for multiple entity types similar to JPA's EntityManager interface.

One note concerning packages: When you actually use the pattern of separate adapters in a sophisticated architecture, you should consider generating entity and adapter classes in different packages. Only the entity classes should be part of the interface for dependent code while the adapter classes should completely be hidden behind facade components as proposed in the wide-spread repository pattern.

Descriptor structure

The examples for descriptors you have seen so far should already clarify most of their structure. You will see more complicated examples in following chapters of this manual. A descriptor in assembled from the following information:

- The name of the entity class and the name of the database table which the entity class is mapped to. Preferably the table name is not specified as a string-literal but as a reference to a constant representing the table name. If you have a look on the outcome of PriDE's code generator, there are appropriate constants generated and used.
- A reference to the descriptor of a base class. This is of interest when you build up a derivation hierarchy between entity classes as explained in chapter Entity Derivation.
- An attribute description map in form of a two-dimensional string array. The array contains one sub array for every table column resp. entity class attribute consisting of
  - The name of the database column (similar to table names: avoid using string-literals here)
  - The name of the getter method for the corresponding attribute in the entity class
  - The name of the setter method

  The methods are the ones which the adapter is supposed to transport entity attributes to the database via JDBC and vice versa. The getter methods' return type implies which methods the adapter uses to access JDBC statements and result sets and how to translate the values to SQL syntax. Getters are mandatory whereas setters may be null in case of entity types that are never supposed to be written to the database. A typical example for this case are entity classes representing the result of SQL joins (see chapter Joins).

The RecordDescriptor class has a few more constructors concerned with joins and accessing multiple databases, but that’s not important for now. The basic structure described above is what you work with most of the time.

Find and query

The terms “find” and “query” for data retrieval are used in the same sense in PriDE as you may know it from other persistence concepts. Finding means to select data with the expectation to retrieve 1 result or none treating the presence of multiple results as an exception case. The most common example is a selection by primary key. A query means to select data with an unpredictable number of result. PriDE is designed to take “unpredictable” literally and allows to process even millions of results in an efficient way in Java.

Find

Examples for finding a record with PriDE were already part of the quick start tutorial and the chapter about entity, adapter, and descriptor. But let’s go into some details here for a deeper understanding. The important things to know:

- No matter if you are working with hybrid objects or a separation of entity and adapter - PriDE never creates entities for you but expects you to provide them.
- Find operations work like a query-by-example. You provide an entity with all the primary key fields initialized and call the find() method without parameters. This is a method of the entity class itself when using hybrid entities, otherwise it is a method of the corresponding adapter class.
- The result of the find operation is placed in the same entity which you provided the key fields by. The boolean return value of the finder tells the caller if there was actually a matching record found.

When you are working with hybrid entities,