Ujorm User Guide

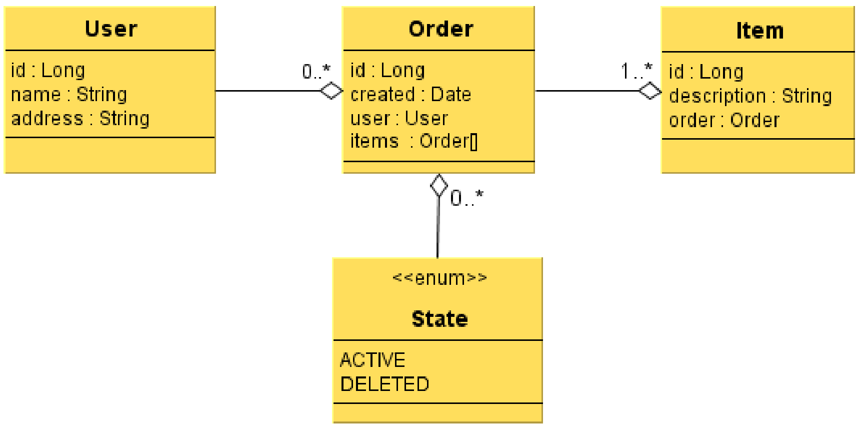
Document License: Creative Commons BY-ND ,
Author: Pavel Ponec, ponec@ujorm.com
Spring Integration by Tomas Hampl
Version: Ujorm 1.00 (October, 2010)
PDF format:
Thank you for your comments and suggestions.

Quick Start

Ujorm is a lightweight ORM framework for **effective** development of Java applications on a relational database. Some key features are:

- type safe database queries
- relation mapping the Java code rather
- internal cache with a protection against overload storage
- no entity states
- incremental database update using DDL statements
- lazy initialization is supported
- simple and reliable implementation solution
- integration support for the Spring Framework

For fast acquisition recommend reading a tutorial built on the annotated source code. In tutorial you can found entities from following class diagram:



Note: UjoProperties are in diagrams formally described using the attributes of entity.

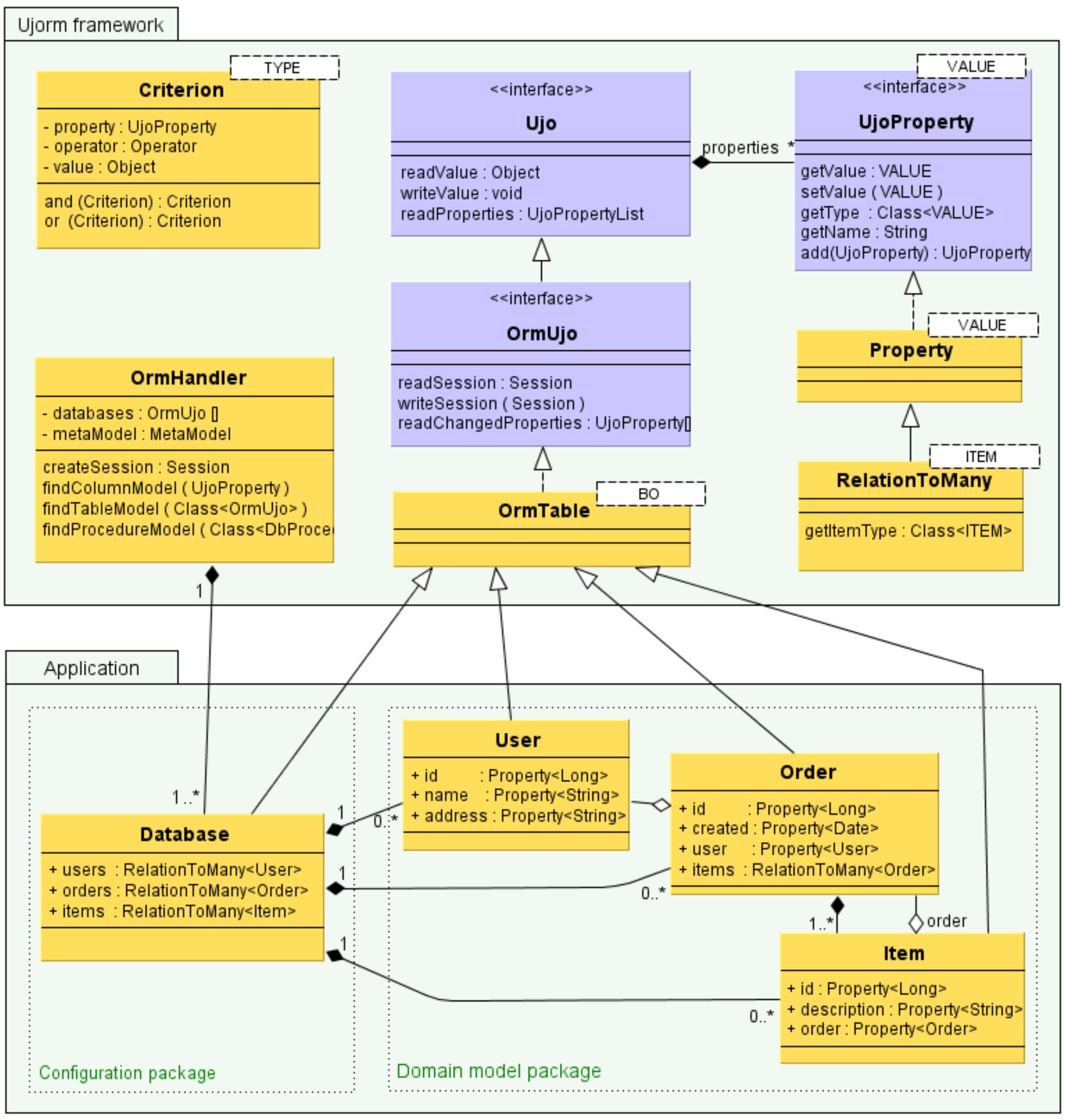
Because Ujorm connected from Maven repository (repo1) is compiled without debugging information, I suggest to you to join the Ujorm JavaDoc to your project, so that you get full names of method parameters and more useful information from API.

When you use the Maven POM file, take the following dependency:

```
<dependency>
     <groupId>org.ujoframework</groupId>
     <artifactId>ujo-orm</artifactId>
     <version>1.00</version>
</dependency>
```

Architecture

The following class diagram clarifies relation between Ujorm framework classes and application domain classes. The diagram is simplified and slightly modified to show only the relevant entities. A more detailed descriptions can be found, if necessary, in the Ujorm JavaDoc .



Notes to graph:
UjoProperty attributes described in the domain objects are always **static constants,** which provide methods for writing and reading values to the Ujo object. Only this diagram is hinted for better illustration the implementation of UjoProperties in domain objects, in other diagrams of this document is used the same formal registration as for a regular POJO attributes.

Entity Description:

- Ujo - interface of an object containing the values. The object provides a list of their properties. A kind of data storage determines the implementation, for example OrmTable uses internally an array of objects.
- UjoProperty - immutable object containing meta-data of the property, which provides two methods for reading and writing values to Ujo object. Two UjoProperties could be joined using the method add(), the result is stored in a new immutable instance.
- Property - default implementation of UjoProperty
- RelationToMany - extended default implementation of UjoProperty designed for the data type: UjoIterator<item>

- OrmUjo - an interface designed for persistent application domain objects. Each object contains the session and provides a list of modified properties, more informations are available later in this document
- OrmTable - default implementation of OrmUjo interface
- OrmHandler - owner of the meta-model, which provides an efficient methods to search for specific parts of the meta-model. Meta-model is locked after initialization and can't be changed later.
- Database - the class of an application package containing a description of the persistent domain classes and stored procedures. For different DB connections, you can define multiple classes of type Database
- Criterion - the fundamental condition for the evaluation database Query. Criterion objects can be joined into a tree by operators and () and or () .
- User, Order, Item - example of using application domain objects, Please notice that in object can be used RelatinToMany property like in class Database. This RelatinToMany always returns UjoIterator as a result of a new database query.

Example of application the code

The following code example obtains instances of OrmHandler class and its configuration using the Database class, the instance is retained for. It creates an instance of Order class Items and stores in a database. The save() and commit() operations can throw an runtime exception so there is necessary to catch the exception and close the used Session.

```
OrmHandler handler = new OrmHandler();
handler.loadDatabase(Database.class);

Order order = new Order();
order.setDate(new Date());
order.setDescr("John's order");

Item item1 = new Item();
item1.setOrder(order);
item1.setDescr("Yellow table");

Item item2 = new Item();
item2.setOrder(order);
item2.setDescr("Green window");

Session session = OrmHandler.getInstance()
                  .createSession();
session.save(order);
session.save(item1);
session.save(item2);

session.commit();
session.close();
```

The second example contains usage a query using the Criterion joined by two elementary conditions. Missing operator is replaced by the EQUALS' one. Notice how is created composite 'UjoProperty'. The composite 'property' can be stored as an additional static constants of domain object. Prepared query is contained in the 'Query' object a can be simply used in 'for-each' command. In the end we have to close database 'session' again.

```
Criterion<Item> cn1, cn2, criterion;

cn1 = Criterion.where( Item.ID, Operator.GE, 1L );
cn2 = Criterion.where( Item.ORDER.add(Order.DESCR)
                     , "John's order" ); // Equals to
criterion = cn1.and(cn2);

Session session = handler.createSession();
Query<Item> items = session.createQuery(criterion)
         .orderBy ( Item.ORDER.add(Order.CREATED)
                  , Item.ID.descending() );
System.out.println( "Row count: " + items.getCount());

for (Item item : items) {
    String descr = item.getDescr();
    Date created = item.getOrder().getDate();
    System.out.println( created + " : " + descr );
}
session.close();
```

In the last example is used information from meta-model, which may contain useful information such as validation of the object.

```
MetaColumn col = (MetaColumn)
            handler.findColumnModel(Order.DESCR);

 StringBuilder msg = new StringBuilder()
     .append("Length : " + col.getMaxLength() + '\n')
     .append("NotNull: " + col.isMandatory()  + '\n')
     .append("PrimKey: " + col.isPrimaryKey() + '\n')
     .append("DB name: " + col.getFullName()  + '\n')
     ;
 System.out.println(msg);
```

How to use the Criterion ?

The criterion is a base element to build an condition. The Criterion has an interesting feature: it can be used immediately for the evaluation of an object - such as the validator.

For next examples create some Criterions:

```
final Order order = new Order();
order.setId(100L);
order.setDescr("my order");
order.setDate(new Date());

Criterion<Order> crnId
```

```
   = Criterion.where(Order.id, 100L);
Criterion<Order> crnDescr
   = Criterion.where(Order.descr, "another");
Criterion<Order> crnCreated
   = Criterion.where(Order.created
   , Operator.LE
   , new Date());
Criterion<Order> crn = null;
```

The samples of the use:

```
// Simple condition: Order.id=100
crn = crnId;
assert crn.evaluate(order);

// Compound condition:
// Order.id=100 or Order.descr='another'
crn = crnId.or(crnDescr);
assert crn.evaluate(order);

// Compound condition with parentheses:
// Order.created<=today()
// and (Order.descr='another' or Order.id=100)
crn = crnCreated.and(crnDescr.or(crnId));
assert crn.evaluate(order);

// Another condition:
// (Order.created<=today() or Order.descr='another')
//  and Order.id=100
crn = (crnCreated.or(crnDescr)).and(crnId);
// ... or simple by a native priority:
crn =  crnCreated.or(crnDescr).and(crnId);
assert crn.evaluate(order);
```

Further examples of solved tasks can be found at this link .

States of persistent object

Persistent class is obtained by extending the abstract class OrmTable or implementation of OrmUjo interface. Persistent object **doesn't** contain invasive states known from specifications of 'JPA/Hibernate', but each persistent object of OrmUjo type contains two other attributes:

- Session - is needed to obtain a 'DB connection' in late values approach, the process is called 'lazy-loading'. If an **attribute of** the persistent object is another persistent object, then in each session reading is 'session' propagated from the reference to the relational object. For this reason, if you restore the 'session', there is sufficient to setup only the first object of the relational hierarchy to the current Session.

- UjoProperty[] - contains a list of 'Properties' **registered** during the assigned session. This list provides more efficient updating of the database by 'UPDATE' command, because the command is compiled only for the modified value of the 'Properties'. Other values are ignored while updating. Note that if a session isn't assigned to the object, there is registered no value changes.

Function of OrmUjo object is just "data transporter", without any restrictions you can do activities such as:
- object obtained using the 'update' can be used (after changing ID) for insert
- a new session can be assigned to object anytime (no other dependency)
- object can be assigned with foreign key to force LazyLoading - using 'Ujo.writeValue()'
- to change the session of ManyToOne type (update of foreign key) is sufficient to assign a new instance of an object containing an real Primary Key
- when using Criterion, the value of the filter (parameter) can be a new instance of OrmUjo object that contains the required Primary Key

Note: both of these attributes aren't transferred by Java **serialization,** which means that after the restoration from serialization, they will be logically empty.

Mapping of data types

Mapping of data types means adding JDBC type and element name to persistent Ujo property. Depending on the parameter mapping can be divided into two major groups:
- mapping of session - mapping to another persistent object of **OrmUjo** type
- mapping of value - other assignments of selected object types

More detailed description is here:

Mapping of values

- **default** mapping is implemented for objects of these types: String, Boolean, Byte, Character, Short, Integer, Long, Float, Double, BigDecimal, BigInteger, byte [], java.util.Date, java.sql.Date, java.sql. Time, java.sql.Timestamp, java.sql.Blob, java.sql.Clob, Enum
- **explicit** mapping is given in '@Column' annotation or via an XML configuration file
- via **interface:** any object that implements the interface 'ValueExportable' is automatically mapped to 'VARCHAR'. That interface can also implement the 'Enum' type.
- specific requirements for mapping can be implemented by overlay **TypeService** class

Mapping of session M:1

- mapping on primary key: for mapping is used standard property, only in class name appears another persistent class of object (implementing OrmUjo). As a relational column will be automatically used the primary key of a relational table.
- explicit mapping: used if the referenced column is not the primary key. In this case, you can use a 'RelationToOne' class that lets you put relational column with a second parameter in the following format:

  ordProperty = RelationToOne.newInstance (Order.class, **Order.sid)**

I recommend referenced column mark with a unique **index** for fast searching and data integrity, some databases require an index explicitly.

Mapping of session 1:M

This type of mapping offers (in way of writing code), easy access to the database query results. Any other value invoking generates a new database query. This type

of property doesn't affect the creation or modification of database structure.
Example of use:

```
items = newRelation (Item.class)
```

Mapping of session 1:1

From the perspective of a solution this is a special case of session M: 1, where degree number M can be limited by appropriate unique index. An alternative solution can be using a relational table as primary key (see example below).

Mapping of relation M:N

The binding is implemented using two relations M:1 & 1:N, which are mapped on the other (relation) table.

Lazy loading

Late loading of objects is a process that generally allows more efficient transfer of data between the database and the client. Late loading is implemented for each attribute that contains other entity, which is, in the database model, mapped by a foreign key.

In late loading of an object Ujorm starts new database query that uses a database connection stored in **opened session.** However, because the entity **can't** open the closed session is itself, so LazyLoading on closed session shows a run-time exception:

```
new IllegalStateException ("The session is closed")
```

Such exemption can be avoided by choosing a suitable solution:

- before invoking lazy to assign a open session to object:
  `OrmUjo.writeSession(session)`
- before invoking lazy to update the attributes of an object from database:
  `Session.reload (ujo)`
- load the required lazy property right after obtaining the object using method according to the model:
  `item.getOrder();`
- retrieve all the lazy property to the desired depth by method:
  `OrmTools.loadLazyValuesAsBatch(...)`
- sometimes it is not necessary to load the entire entity, just a foreign key, for which there is no need to have an opened session. This key can be obtained by command:
  `OrmTable.readFK (property)`

Note: Ujorm core doesn't use this method and isn't included in OrmUjo interface. The method is available in the default implementation of OrmTable.

Sometimes you need to get lazy attribute(s) with a single SQL query using JOIN for more database query optimization, or to use some specific operators in your database. In this case, you can use:

- user's query called "Ujorm native query" mapped to a new entity, or
- database VIEW - mapped to the new entity

Session cache

Ujorm uses caching objects in session in order to limit duplicate database queries. Cache behavior has a simple rule:

- lifetime of cache has range (SCOPE) of database transactions. After executing a commit / rollback the cache will be released
- in cache would get only objects of foreign key when loading lazy (lazy loading), in other cases the object is always loaded from the database
- while updating the object, cache doesn't sync and so for elimination of potential conflicts you can empty the cache by clearCache ()
- cache storage is protected against memory overload, it means that the garbage collector can release part of the cache at any time

From cache side: Ujorm doesn't have exclusive access to the database, the registration will only need to align the allocation of primary keys. With parameter CACHE_POLICY you can disable the cache for whole meta-model or you can disable releasing the cache with the garbage collector.

To optimize performance rather recommend to keep the cache of **logic units** at the level of application logic (eg, Articles) than maintaining a cache of database rows.

DDL support

Ujorm can edit the database schema using DDL updates of its meta-model. Meta-model contains a description of the database's real structure, which is compiled from class names, UjoProperties names, their annotations and eventually via the XML configuration file. When loaded into memory, meta-model is locked so any future modifications aren't possible. Updating the database is following these rules:

- updates are carried out according to a meta-model right after its compilating in the OrmHandler object
- update is done only for **database items:**
  - new table
  - new columns
  - new indexes
  - db schema is created only in the first update
- item (column, index) created in the database framework is no longer changed, so you will have to be handle migrations on **your own**, for example:
  - change the column NULL to NOT NULL (or back)
  - change in the index UNIQUE to NON-UNIQUE (or back)
  - change the primary key of table, etc.
- database table must contain at least the primary key
- Ujorm ignores non-mapped database objects (tables, columns, indexes)

Parameter ORM2DLL_POLICY can change behavior of update to one of the following approaches:
- update ban
- allow only the first creation of the db model
- allow incremental changes (default)

General use of Ujorm parameters is described in the tutorial.

Entity inheritance

Ujorm doesn't support the concept of inheritance in the JPA. The main reason is that implementation of inheritance into the existing Ujorm architecture simply doesn't fit.

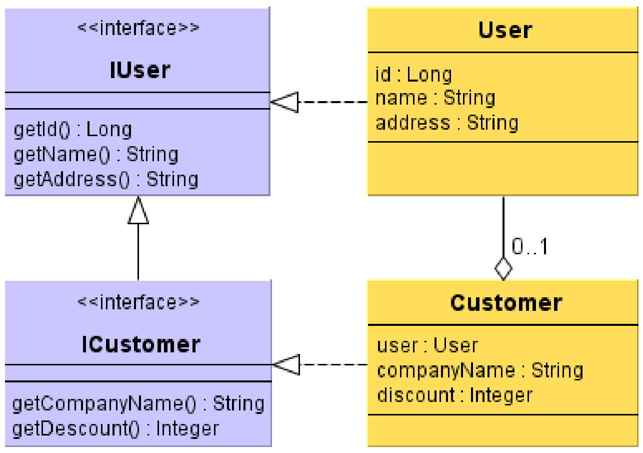
Easy availability of inheritance in ORM tempting to light-headed use, which can cause performance problems on production data, so I recommend rather avoid the inheritance.

However, if required, Ujorm allows **to emulate** inheritance - using design pattern of **aggregation and interface** (interface).

In draft of solution the inheritance are used entities:

- User [id, login, password, name]
- Customer [user, company, discount]

The proposed solution is based on IUser interface, that implements User and Customer objects. Both entities, User and Customer, shares the same ID. We can start using the advantages of inheritance, becaues instance of type **ICustomer** can be written to the data type **IUser.** Descriptions of **setters** have been skipped to simplify the diagram.



Notice that the 'Customer' has a **primary key** of **User** type and shares the same ID value:

```
public class Customer extends OrmTable<Employee> implements IUser {
     @Column(pk=true, value="id")
     public static final UjoProperty<Customer, User> user
                      = newProperty(User.class);
     ...
}
```

Implementation of the method **Customer.getLogin ()** might look like this:

```
@Override
public String getLogin() {
     return get(user).getLogin();
}
```

In operations such as INSERT and UPDATE, we have to take care of each entity separately. See the example of coding to save the 'Customer':

```
 Session session = handler.getSession();
 session.save(customer.getUser());
 session.save(customer);
 session.commit();
```

Example of a full implementing the inheritance of the 'Customer' class can be found here and its use here.

Primary key auto increment

Ujorm supports automatic generation of primary key using **UjoSequencer** class. The solution was influenced primarily by requirements for high **reliability** and smooth **portability** of implementation between database of different providers. Allocation of primary keys is following these rules:

- Primary key (PK) will be assigned automatically only if the value of 'UjoProperty' is NULL. It is possible to assign own values of PK without changing any configuration.
- Each table has its own set of primary keys, about which take care its own instance of the UjoSequencer class. Class UjoSequencer contains primary key values in an auxiliary database table **ujorm_pk_support,** here is a description of columns:
    - **id** (db type VARCHAR) - table name including schema format
    - **seq** - the value last assigned sequence
    - **cache** - UjoSequencer can allocate a number of consecutive numbers, which then allocates from memory to increase performance. Enabled cache may cause:
        - discontinuous series of keys in case of restart Ujorm
        - violation of time ascendance in the case of multiple Ujorm clients

    This behavior can be eliminated by disabling the cache using value 1. Value of cache can be modified correctly also in application program (SQL command).
    - **maxvalue** - if a column 'seq' exceeds the maximum value, Ujorm shows an exception
- allocating the primary key of Ujorm is used independent DB connection, so that any rollback of transaction wouldn't cause the return already assigned primary keys

A common question is how to use native support of database sequence? The solution lies in creating own implementation (child) of UjoSequencer class. Class of this child must be registered in the parameters Ujorm before creating a meta-model. The only requirement of solution is that the acquisition of a new primary key (e.g. using JDBC) must be completed before INSERT command. Removing this condition is in the development plan of Ujorm.

Database procedures

Ujorm supports since version 0.94 invoking database stored **procedures** and **functions** (the procedure). Each procedure is described by class extending an abstract DbProcedure class, which implements the Ujo. Important is an order of UjoProperties in the class, which must match the order of parameters in the procedure. If the procedure **have no return type,** the first constant UjoProperty in order will be **Void.class** type**.** The first UjoProperty is OUTPUT type, others are INPUT type. The default type of properties you can override by annotation *@Parameter (input = true, output = true).* Database procedures need to be **registered** in the class of databases like the database table:

```
public static final Property <Database,MyProcedure> procedure
        = NewInstance (MyProcedure.class);
```

We create an instance of the procedure, fill the INPUT type parameters and invoke with 'call' method with parameter of opened session:

```
MyProcedure procedure MyProcedure = new ().
procedure.set (MyProcedure.DATE, new Date ());
String result = procedure.call (session);
```

Complete example you can find in tutorial. Instances of the procedure isn't connected with a particular session and such instance can be used can frequently with different parameters (only one-thread access). Ujorm version 1.00 does not support the return type of parameter ResultSet.

Transaction control

All database DML statements can be seen as temporary until they are committed. Ujorm supports commitment with the Ujorm Session.commit() . Method Session.rollback() is used to remove operations performed since the previous commit or rollback. Use this method when an exception occurs or when the program detects some error condition or error in the data.

For a parallel transaction use a next session.

See a JDBC 2.0 documentation for more information.

API extention

In development, after time we can meet the requirement of implementation of specific function or features of SQL language. For this type of requirement can be created "own child" of **SqlDialect** class (or some of its implementation), because this class provides several important activities:

- creates a specific SQL queries for different CRUD type databases
- defines the syntax of each type of operator in the WHERE phrase
- creates DDL commands according to a meta-model
- provides DB connection (or other parameters for DB connection)
- provides a list of SQL keywords

Any of the attributes above can be easily modified by creating a 'child', new class is registred by annotation in the Database class type. Overlapping the SqlDialect class can solve one of the following requirements:

- expansion and optimization of SQL queries
- mapping of specific data type of SQL language
- completion of own operators for Criterion class has two steps:
    1. create a child of ValueCriterion class, extended with specification of a custom type
    2. overlay metodu SqlDialect.getCriterionTemplate () implementing the extended operator
- creating own DB connection or a its modification (adding parameters)
- add or remove a reserved SQL keywords

TypeService

class provides a mapping of application values to the JDBC API. Methods of class can be overlaied with their own implementations. Registration is done by annotation in the 'Database' type class.

UjoSequencer
As stated earlier, this class allows the implementation of your own algorithm for assigning a unique ID to INSERT command. Registration is done by annotation in the 'Database' type class.

Supported databases

Ujorm framework offers prepared dialects for the following databases:

- PostgreSQL (8.3)
- MySQL (5.0)
- Derby (10.5)
- Oracle (8.0)
- H2 (1.2.x)
- HSQLDB (1.8.x)
- Firebird (jaybird 2.1.6)

As mentioned in the previous chapter, support for other databases can be implemented by overwriting the **SqlDialect** class or one of its 'childs'. The new class has to be registred in application class of the 'Database' type.

Why UjoProperty constants have lowercase letters?

In the examples mentioned in this document are the names of the static field of UjoProperties often written in lower case - contrary to normal convention. The reason is to shorten the code when creating UjoProperty, but also seems to be useful when small letters more correspond to instance variables (fields). Lowercase letters are proven to be better, but the using the capital letters in your project won't create a technical problem.

Full specification of UjoProperty can be set explicitly, including a **name**, by this pattern:

```
public static final Property <BO, Long>  id = newProperty('id', Long.class);
```

In short writing is name of 'Property' taken from name of field:

```
public static final Property <BO, Long> id = newProperty (Long, class);
```

Terms used in UJO Framework:

- UjoProperty **field** - the name of the static constants of UjoProperty in the implementation of Ujo
- UjoProperty **name** - a logical indication that returns 'UjoProperty.getName()' as the String type

Use of UJO objects

This chapter describes the important features of the UJO object, which have an effect on the correct behavior:

- each **attribute of entity** has its own unique UjoProperty instance
- for reading and writing values are used these methods:
    - UjoProperty.getValue (..)
    - UjoProperty.setValue (..)

  or other methods that invoke tham internally, for example:
    - UjoMiddle.get (..)
    - UjoMiddle.set (..)

  or more comfortable **getters** and **setters,** that invoke (internally) one of the methods above
- method 'UjoProperty.getValue()' **changes** values NULL to the default value UjoProperty (which is NULL if it isn't set otherwise). Beware, the method 'Ujo.readValue()' doesn't do any replacement of default values.
- for reading and writing values **is not recommended** to invoke **untyped** methods
    - Ujo.readValue ()
    - Ujo.writeValue ()

  However, these two methods are suitable for the **overlaying** in prepose to implement the various conversions, validations and other requirements.
- for default value of 'UjoProperty' is suitable only **steady object** (immutable object)
- Ujo objects can be generally inherited, inheritance of OrmUjo objects isn't supported directly
- more information about Ujo Core can be found here .

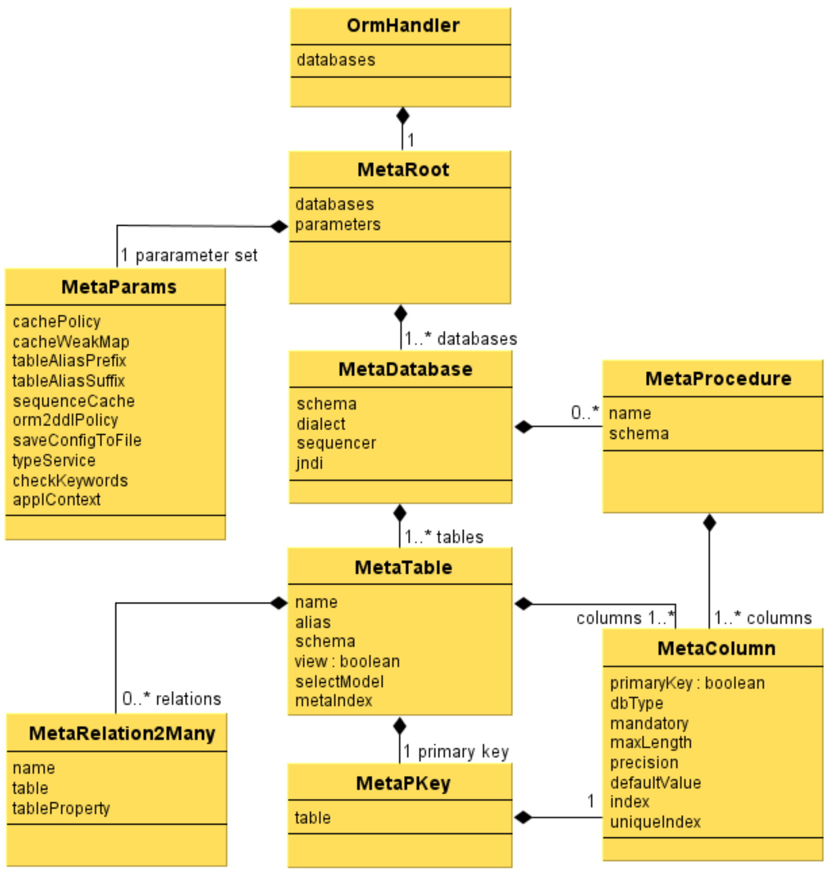
Best Practices

- common ancestor of your domain objects can simplify behavior changes of all children. An example might be a generic implementation of method 'toString ()', or implementation of methods 'equals (..)', 'hashCode ()', etc.
- service for processing data from DB to Query can easily complement other Criterion. An example of use might be implementation of limited access to data based on context (role) of logged user
- for access to the values of UJO objects proved using the methods:
    - UjoMiddle.get (property)
    - UjoMiddle.set (property, value)

Meta-Model

Meta-model is created according to the content of Ujo properties of Database class and its tables. The names of tables, view and procedure are derived from names of Ujo property. Most features of the model can be overlayed by annotations placed directly in property or in name of the class. Freatures set by annotation can be also overlayed by configuration XML file, which can optionally modify only selection of a meta-model.

After creating a meta-model in memory, the model is locked and you can't write into meta-model anymore, at runtime, however, you can obtain useful information from it. Slightly simplified class diagram meta is in the following picture:



Entity Description:

- OrmHandler is a class providing meta-model, in the application is recommended to keep only one instance of this class. Class offers several methods to effectively assistive selected parts of meta-model. OrmHandler is the only entity in the diagram, which isn't a child of an abstract AbstractMetaModel class (implementing the interface of Ujo).
- MetaRoot creates the base of meta-model and allows you to read an XML configuration file
- MetaParams contains general Ujorm parameters that can be also modify by the XML configuration file
- MetaDatabase contains a database connection, a list of tables (or views) and the list of database procedures. It also includes a default schema.
- MetaTable represents the database table that takes the schema from 'MetaDatabase', but you can configure (with annotations) also a different scheme. Logical attribute 'view' differs view from the tables (you can't white into 'view' and it doesn't affect the updating of the DB).
- MetaPKey - each table contains one primary key
- MetaRelation2Many - represents a 1: M type database session
- MetaColumn - class represents the database column, or stored procedure parameters.

Integrating Spring Framework

The Spring Framework can automate the creation and release Ujorm session and manage a transaction management using a special module called 'orm-spring'. In a Maven project use the next dependency in your POM file:

```
<dependency>
   <groupId>org.ujoframework</groupId>
   <artifactId>ujo-spring</artifactId>
   <version>1.00</version>
</dependency>
```

How to get a Ujorm Session? The procedure for obtaining the session will vary according to the code that calls the session:

- aroundSessionAOP - calls from jobu or other internally
- Open Session in View - using HTTP Request

AroundSessionAOP is currently defined in the current spring-beans.xml poincut
org.application.administration.jobs ..(..)*

Open Session in View is defined in the **web.xml** .

How to use the session Manualy?

```
UjoSessionFactoryFilter ujoSessionFactoryFilter
   = (UjoSessionFactoryFilter)wac.getBean("ujoSessionFactory");
ujoSessionFactoryFilter.openSession();
...
ujoSessionFactoryFilter.closeSession();
```

How to use a Manual control session and transaction - using ORM handle?

```
OrmHandler handler = (OrmHandler) wac.getBean("ormHandler");
Session s = handler.createSession() ;
...
s.commit();
s.close();
```

See the official Spring documentation for more information.

Performance

Ujorm is very fast in comparison to its competitors, see the benchmark table to detail information. I hope that you will not be angry if I compare Ujorm performance with the popular Hibernate ORM framework. The result of the benchmark is listed in the next table. Each action represents an database operation without any optimization.

Action	Ujorm 1.00 [sec]	Hibernate 3.3.1 [sec]	Ratio
meta data	0.33	1.53	4.65

single select	0.35	0.58	1.67
empty select	1.24	151.17	121.85
multi select	21.78	168.01	7.71
insert	11.12	12.30	1.11
delete	101.18	208.28	2.06
update	8.04	3.90	0.49

Action description:

- meta data - time to loading meta-model and to verify database model (auto-update option is enabled)
- single select - the one big select for all order items (with a condition)
- empty selects - 2000 selects with the empty result - where a condition contains different parameter values
- multi select - many different selects to emulate a server application or statements with no optimization
- insert - insert 2000 orders, 14000 items and 1 user
- delete - execution many statements to delete all table rows
- update - modify and save a loaded BO

Environment description:

- date of performance: 2010-10-15
- test through the 2000 orders and 14,000 items
- against the database PostgreSQL 8.3 (on the same computer)
- values are an average of 5 measurements
- DB indexes was insignificant
- UJO Framework release 1.00
- Hibernate 3.3.1.GA + EhCachProvider
- Java version "1.6.0_14"
- default Java run-time parameters
- Intel Dual Core, 2GB RAM
- Windows XP + SP3
- benchmark code is available on the SourceForge

More information

- JavaDoc the detail API descriptions with some examples of use
- Home Page
- Presentation (intro slide show), includes inter alia
  - short motivation
  - Key Features
  - Compared with competitors (Hibernate iBatis)
- Examples of using the source code
- Blog
- Wiki
- FAQ

Acknowledgements

Thanks to developer team at Effectiva Solutions for their valuable comments on the

technical content of this document.