Table of Contents

Changes:

5/10/2012 : updated for mapperdao 1.0.0-rc15 or newer

MapperDao tutorial

We will create domain classes for a product catalogue and map it to the database tables using mapperdao. We will start with a simple domain model and tables and extend it as it goes.

Downloading the source files

Please download the source files from

https://code.google.com/p/mapperdao-examples/source/checkout

This tutorial goes through parts of the implementation of the productcatalogue example.

Setting up the database connection and initializing mapperdao

For an online shop, we need to store product information into the database and be able to load those into domain classes. Our classes consist of a Product class which stores information for products, a Category class which stores information about the category that the product belongs to and the Attribute class which stores product attributes.

But initially we need a DataSource to configure mapperDao. For this example we will use apache dbcp to create a connection pool and the DataSource:

http://mvnrepository.com/artifact/commons-dbcp/commons-dbcp/1.4

```
val properties = new Properties
properties.load(getClass.getResourceAsStream("/jdbc.properties"))
val dataSource = BasicDataSourceFactory.createDataSource(properties)
```

This will configure a database connection pool according to the settings in jdbc.properties file. An example properties file follows:

```
driverClassName=org.postgresql.Driver
url=jdbc:postgresql://localhost/productcatalogue
username=productcatalogue
password=secret
```

We can now configure mapperDao. We need a list of all entities and then we can configure it using one of the convenient factory methods. In the following example we configure it for ProductEntity only and using the postgresql database driver. We will declare ProductEntity later on but for now here is the configuration:

```
val entities = List(ProductEntity)
val (jdbc, mapperDao, queryDao,txManager) = Setup.postGreSql(dataSource,
entities)
```

jdbc:Jdbc can be used to do low level queries/updates to the database. mapperDao:MapperDao can

be used to insert/update/select/delete entities and queryDao to query for entities. txManager:PlatformTransactionManager takes care of transactions. The 'val entities' is a list of our entities, currently just ProductEntity.
For more information on configuring mapperdao, please see
https://code.google.com/p/mapperdao/wiki/SetupDaos

MapperDao, QueryDao and txManager are low level data access components. Later on we'll see how we can create dao's using those along with a few mixin traits.

Creating a simple model

Our initial model will start simple, lets map just the Product entity and later on we'll do more than that.

```
case class Product(title: String, description: String)
```

So far we only got a product title and a description. We can map this entity to 1 table. Here is the ddl for postgresql:

```
create table Product (
     id serial not null,
     title varchar(100) not null,
     description text not null,
     primary key(id)
)
```

Please note that the mapping will work for all supported databases but in this tutorial the ddl's will be for postgresql. For other databases, please see productcatalogue/ddl folder of the productcatalogue example: http://code.google.com/p/mapperdao-examples/

Also note that the table contains an autogenerated surrogate id but our domain class doesn't. The id is a property of the database. Only a persisted entity contains an id. And that's also how mapperdao works. When an entity is loaded from the database, it will acquire the id via the SurrogateIntId trait. (or SurrogateLongId for entities with surrogate long id but please note that this mechanism is extensible and new traits can be used). Alternatively, if the id is not autogenerated, then it can be a field of the domain class but for this example we will use autogenerated id's. The id will be available whenever an entity is inserted or loaded from the database.

Please consult mapperdao's wiki for more information regarding id's, autogenerated id's and sequences :

https://code.google.com/p/mapperdao/wiki/TableOfContents
https://code.google.com/p/mapperdao/wiki/DiffBetweenNaturalAndSurrogateKeys

Lets now map the domain class to the database table. It's time to create the ProductEntity. MapperDao uses scala code to do the mapping. The mapping can be an object or a class and it is the equivalent of xml files that other ORM tools use. It maps all the columns and also contains a constructor method which is responsible for constructing instances of the entity.

```
import com.googlecode.mapperdao._

object ProductEntity extends Entity[Int,SurrogateIntId, Product] {
	val id = key("id") autogenerated (_.id)
	val title = column("title") to (_.title)
	val description = column("description") to (_.description)

	def constructor(implicit m:ValuesMap) = new Product(title,
description) with Stored {
		val id: Int = ProductEntity.id
	}
}
```

The entity extends Entity[Int,SurrogateIntId,Product]. This means that the entity is of type [Product] but when loaded it is of type [Product with SurrogateIntId]. SurrogateIntId adds the 'id' field to the domain class. The primary key is of type Int.

The first mapping is the mapping of id column. It is an autogenerated key and it maps to Product.id column and Product.id field of [Product with SurrogateIntId]. The field value is retrieved by the _.id function.

```
val id = key("id") autogenerated (_.id)
```

Next, the title column is mapped against Product.title field.

```
val title = column("title") to (_.title)
```

After the mapping of the description, the constructor definition follows:

```
def constructor(implicit m:ValuesMap) = new Product(title, description)
with Stored {
	val id: Int = ProductEntity.id
}
```

This method is used to construct instances of the Product class when it is loaded from the database. The implicit m:ValuesMap is a map that contains all the database values and it is implicitly used to convert the title and description columns to their String values. We could also explicitly convert them, i.e. m(title) would return a String and even further we could i.e. do m(title).toLowerCase or any other data transformation.

The constructor creates a new Product with Stored. Stored is a type alias and for this example it is SurrogateIntId and provides the id field which gets it's value from the autogenerated id column. Later on when we insert the instance with

```
val inserted=mapperDao.insert(ProductEntity,Product(“title”,”desc”))
```

inserted.id will contain the autogenerated id. And that's a good thing because when we create in-memory instances of Product class, those don't contain a valid database id and hence it is better that we don't have to provide dummy id's. But as soon as we persist them, mapperdao returns the same entity but with SurrogateIntId (or SurrogateLongId) mixed in, which again is correct since now the entity is persisted and hence contains an autogenerated id.

The entity object seems a bit verbose compared to java ORM tools. But this allows us a great deal of flexibility later on as we can refer to columns via their mapping, i.e. in queries we can refer to ProductEntity.title or even better if we 'val p=ProductEntity' then we can refer to it via p.title. Also the constructor gives us a great deal of functionality as we can transform the data as we wish, i.e. m(title).toLowerCase, order related data according to their id, convert database columns to objects, intern and so on. We will see some of those capabilities later on in this tutorial.

Now we are ready to use mapperdao to insert, select, update and delete the Product entity:

```
val inserted=mapperDao.insert(ProductEntity,Product("title","desc"))
println(inserted.id)
val updated=
     mapperDao.update(ProductEntity,inserted,Product("newv","newv"))
val selected=mapperDao.select(ProductEntity,updated.id)
mapperDao.delete(ProductEntity,inserted)
```

MapperDao provides insert,select,update,merge and delete methods but those are low level. Effectively we'll need to create dao for CRUD, querying and other data operations on each one of our entities. We can use mapperDao and queryDao to create custom dao's but also we can take advantage of the library's mixin traits which provide create, retrieve, update, merge, delete (CRUD) methods and utility methods i.e. all() which retrieves all entities from the database. Those mixins can be transactional or non-transactional, with the transactional one been the most common case because it will update the entity and all related entities in a transaction:

```
abstract class ProductsDao
     extends TransactionalSurrogateIntIdCRUD[Product]
     with SurrogateIntIdAll[Product] {
     val entity = ProductEntity
}
```

This not only creates a dao that can do CRUD but also can do queries (via the SurrogateIntIdAll trait) and paginate the data. Of course it is not hard at all to replicate this functionality using queryDao and mapperDao, but the traits certainly help.

These are some of the methods that ProductsDao aquire from the mixins:

```
// inserts a product into the database and returns a
// Product with SurrogateIntId
create(product):Product with SurrogateIntId
createBatch(products):List[Product with SurrogateIntId]

// updates a product – if product is mutable
update(product with IntId): Product with SurrogateIntId
updateBatchMutable(products):List[Product with SurrogateIntId]

// updates oldProduct to it's newProduct value – for immutable classes
update(oldProduct,newProduct): Product with SurrogateIntId
updateBatch(List[(oldProduct,newProduct)]): List[Product with
SurrogateIntId]

// merges (reads the entity and updates it to newProduct)
```

```
merge(newProduct,id)

// retrieve a product by it's id
retrieve(pk:Int):Product with SurrogateIntId

// delete a product
delete(Product with SurrogateIntId):Product

// delete a product by it's id
delete(id:Int):Unit

// returns all products
all:List[Product with SurrogateIntId]

// returns the number of all products
countAll

// returns a page of products
page(pageNumber,rowsPerPage):List[Product with SurrogateIntId]
```

It is worth having a peek on the source of those traits. Each of these methods is not more than 2 lines long and gives a good idea on how mapperDao and queryDao can be used to create extra functionality. In addition, there are other traits for the most common primary keys but also all of them inherit from CRUD and TransactionalCRUD and hence all kinds of dao's can be implemented using those traits.

Now we can instantiate our dao and inject it's dependencies.

```
val productsDao = new ProductsDao {
     val (mapperDao, queryDao, txManager) = (md, q, txM)
}
```

Using this dao, crud operations are easy. Inserting, selecting and deleting is a breeze with mapperdao. For insert, we just need to provide a new instance of the entity along with all related entities. The dao will insert the entity and also insert/update any related entities as necessary, all transactionally due to the TransactionalCRUD mixin. Same goes for retrieving data, mapperdao will fetch all related entities from the database, but please note that this is configurable, i.e. only some related entities might be configured to be inserted, updated or retrieved, or even all/some of the related entities can be lazy-loaded. Please look at https://code.google.com/p/mapperdao/wiki/ConfigurableCRUD and https://code.google.com/p/mapperdao/wiki/LazyLoading for more information.

A note about transactions: mapperdao is based on spring framework's jdbc/transactions API. By default the transactional mixin traits are configured to create a transaction, if no transaction was already available. I.e. if a transaction was already created (say, on the service layer) then that transaction will be active when the dao methods are executed. If no transaction was created then a new one will be created. Please look at https://code.google.com/p/mapperdao/wiki/Transactions for more information. The non-transactional mixin traits will not create a transaction but will use one if available.

Updating entities is a bit trickier. MapperDao support immutable entities. And because immutable entities can't change, mapperdao allows them to be updated by providing a new instance of the entity with the new values. For case classes that can be easy, i.e. product.copy(title=”new title”) will create a copy of the product entity with a different title. For mutable entities mapperdao is very similar to other ORM libraries, i.e. if the entity is mutated then a productsDao.update(entity) will do the update.

Lets see an example of some operations:

```
val inserted=productsDao.create(Product("p1","desc"))
println(inserted.id)
val updated=productsDao.update(inserted,Product("new title","new desc"))

val selected=productsDao.retrieve(updated.id)
// selected is now Product("new title","new desc")
val list=productsDao.page(1,10)
// list:List[Product with SurrogateIntId] now contains the 1st page of
// products, each page having 10 items.
```

MapperDao contains strong support for pagination in all supported databases. We can fetch a page of all the data for an entity or a page of data on that same entity.

Back to queryDao now, we can also do all short of queries using the query DSL.

```
import com.googlecode.mapperdao.Query._

val q=select from ProductEntity where ProductEntity.title like "jean"
queryDao.query(q) // returns List[Product with SurrogateIntId]

// … or
(
     select
     from ProductEntity
     where ProductEntity.title like "jean%"
).toList(queryDao) // returns List[Product with SurrogateIntId]
```

It might be better to alias ProductEntity so that our query is more readable:

```
import com.googlecode.mapperdao.Query._

// create the 'pe' alias
val pe=ProductEntity

val q=select from pe where pe.title like “jean”

// … or an even more readable version of the same:
val q=(
     select
     from pe
     where pe.title like “jean%”
)
val list=queryDao.query(q) // returns List[Product with SurrogateIntId]
```

Extending the domain model by adding attributes to products

All our infrastructure is now in place and we can extend our domain model to include extra information for the products. For a start, our products have attributes, an attribute been a key/value pair. Each product can have 1 or more attributes and attributes are shared by products. i.e. many products can have the attribute colour=red. We'll create this as a many-to-many relationship. Lets see our extended domain model:

```
case class Product(
     title: String,
     description: String,
     attributes: Set[Attribute]
}

case class Attribute(name: String, value: String)
```

Ok, now our product contains Set of attributes. We would need to add the relevant tables to our database schema in order to create this many-to-many relationship.

```
create table Attribute (
     id serial not null,
     name varchar(100) not null,
     value varchar(100) not null,
     primary key(id)
)

create table Product_Attribute (
     product_id int not null,
     attribute_id int not null,
     foreign key (product_id) references Product(id) on delete cascade,
     foreign key (attribute_id) references Attribute(id) on delete
cascade
)
```

Now we're ready to change the mappings to include this new relationship.

```
object ProductEntity extends Entity[Int,SurrogateIntId, Product] {
	val id = key("id") autogenerated (_.id)
	val title = column("title") to (_.title)
	val description = column("description") to (_.description)
	val attributes = manytomany(AttributeEntity) to (_.attributes)

	def constructor(implicit m:ValuesMap) = new Product(title,
		description,  attributes) with Stored {
		val id: Int = ProductEntity.id
	}
}
```

Please notice the mapping of attributes using the AttributeEntity. A manytomany relationship between Product and Attribute entities is declared and maps to _.attributes. If we don't specify column and join table names, mapperdao will map them using convention, this means the mapping will be via Product_Attribute table and product_id, attribute_id columns. Please see the wiki on how to customize those.

Finally, the new Attribute class needs it's own mapping:

```
object AttributeEntity extends Entity[Int,SurrogateIntId, Attribute] {
	val id = key("id") autogenerated (_.id)
	val name = column("name") to (_.name)
	val value = column("value") to (_.value)
	def constructor(implicit m:ValuesMap) = new Attribute(name, value)
with Stored {
		val id: Int = AttributeEntity.id
	}
}
```

It will be useful to create a dao for attributes. This way we can add methods to get or create an attribute if it doesn't already exist into the database. That will come handy since we then can easily retrieve/create attributes.

```
abstract class AttributesDao extends
          TransactionalSurrogateIntIdCRUD[Attribute] {
     protected val entity = AttributeEntity

     protected val queryDao: QueryDao
     import Query._ // import the query DSL

     // alias for queries
     private val a = AttributeEntity

     // returns Attribute with SurrogateIntId
     def getOrCreate(name: String, value: String) = {
          val q = (
               select from a
               where
               a.name === name
               and a.value === value
          )
          queryDao.querySingleResult(q).
               getOrElse(create(Attribute(name, value)))
     }
}
```

Please note that we import Query._ which contains the DSL for querying. We then create 'a', an alias for AttributeEntity that we use later on in our query. Our query tries to find if there is already a row with the same name and value and if yes, it loads it. If no, it creates the attribute and by calling create(), it stores it in the database. create() is method provided by the TransactionalSurrogateIntIdCRUD trait. The return value of create(Attribute(name, value)) is an Attribute with SurrogateIntId. Hence if the attribute originally exists or doesn't exist, it will be available in the database when this method returns.

Now we are ready to add and manipulate some products with attributes into our database.

```
// create 2 attributes
val a1 = attributesDao.getOrCreate("performance", "fast")
val a2 = attributesDao.getOrCreate("security", "extra secure")

// create a product that has those 2 attributes.
// ProductsDao uses the Transactional trait and hence all
// inserts will be within a transaction
val p = productsDao.create(Product("test1", "desc1", Set(a1, a2)))

// time to remove a2 from product.
val u1 = productsDao.update(p, Product(p.title, p.description, Set(a1)))

// now add a new attribute. This new attribute, will be added to
// the product but also inserted into the database in 1 single
// transaction.
val u2 = productsDao.update(p, Product(p.title, p.description,
Set(a1,Attribute(“performance”,”slow”)))
```

Finally, having our entities sorted out, we can do various queries, always importing Query._ so that the query DSL is in scope:

```
// aliases
val p = ProductEntity
val attr = AttributeEntity

// get all : List[Product with SurrogateIntId]
val l=queryDao.query(select from p)

// get a List[Product with SurrogateIntId] containing 20 items starting
// from the 10nth row
val l=queryDao.query(
     QueryConfig(offset = Some(10), limit = Some(20)),
     select from p)

// find all red products by joining p=>p.attributes=>attr
val l=queryDao.query(
     select from p
     join (p, p.attributes, attr)
     where attr.value === "red'")

// get all products but order them according to their
// attribute values (asc) and product id's (desc)
val l = queryDao.query(
     select from p
     join (p, p.attributes, attr)
     orderBy (attr.value, asc, p.id, desc)
)
```

Adding categories to our products

We will now improve our domain classes by adding categories to the products. A TV will belong to the TV category or say to the 30' TV category. There will be a number of categories and each product will belong to one or more of them. This can be mapped with a many-to-many relationship. But also each category can have a parent category. For example category 46" TV's will have TV's as it's parent category. This can be mapped as a many-to-one relationship between categories and itself. Since there will be root categories without parents, the parent category is optional. Hence, the parent category will be an Option[Category].

For a start, lets write the ddl for the new tables:

```
create table Category (
     id serial not null,
     name varchar(100) not null,
     parent_id int,
     primary key(id),
     foreign key(parent_id) references Category(id) on delete cascade
)

create table Product_Category (
     product_id int not null,
     category_id int not null,
     primary key(product_id,category_id),
     foreign key(product_id) references Product(id) on delete cascade,
     foreign key(category_id) references Category(id) on delete cascade
)
```

Lets now modify our domain model. Not only we will add a List of Categories to our Product class, but also each Category will have a parent:Option[Category] parent category.

```
case class Product(
          val title: String,
          val description: String,
          val attributes: Set[Attribute],
          val categories: List[Category]
}

case class Attribute(val name: String, val value: String)

case class Category(
     val name: String,
     val parent: Option[Category]
)
```

It is now time to map the domain classes to the tables.

```
object ProductEntity extends Entity[Int,SurrogateIntId, Product] {
     val id = key("id") autogenerated (_.id)
     val title = column("title") to (_.title)
     val description = column("description") to (_.description)
     val attributes = manytomany(AttributeEntity) to (_.attributes)
     val categories = manytomany(CategoryEntity) to (_.categories)

     def constructor(implicit m) = new Product(title, description,
attributes, categories) with Stored {
          val id: Int = ProductEntity.id
     }
}

object CategoryEntity extends Entity[Int,SurrogateIntId, Category] {
     val id = key("id") autogenerated (_.id)
     val name = column("name") to (_.name)
     val parent = manytoone(CategoryEntity) foreignkey "parent_id"
                    option (_.parent)

     def constructor(implicit m) = new Category(name, parent) with
Stored {
          val id: Int = CategoryEntity.id
     }
}
```

So, initially we added the many-to-many declaration to the ProductEntity mapping:

```
val categories = manytomany(CategoryEntity) to (_.categories)
```

Then we used the 'val categories' to construct the Product instances. It is converted to a List[Category] via the 'implicit m:ValuesMap' variable which contains the values retrieved from the database:

```
def constructor(implicit m) = new Product(title, description,
attributes, categories) with SurrogateIntId …
```

We could implicitly do the conversion and some times this is preferable, i.e. if we would like the categories list to be sorted:

```
def constructor(implicit m) = new Product(title, description,
attributes, m(categories).toList.sortWith(_.name<_.name) …
```

Finally we declared the CategoryEntity itself making sure the parent category is mapped as a many-to-one but is also optional.

```
val parent = manytoone(CategoryEntity) foreignkey "parent_id" option
(_.parent)
```

The foreign key is “Category.parent_id” column and the name doesn't follow the default convention (it should have been “category_id”). Hence we need to explicitly define the foreign key. Also, since we map to an Option[Category], we use option(_.parent) to do the mapping.

Now we are ready to create CategoryDao. Except from the CRUD methods that will be mixed in by the TransactionalCRUD trait, we also need methods to create hierarchies of categories.

```
abstract class CategoriesDao extends
     TransactionalSurrogateIntIdCRUD[Category] {
     protected val entity = CategoryEntity
     protected val queryDao: QueryDao

     import Query._

     // alias to category entity
     private val c = entity

     def getOrCreate(name: String,
          parent: Option[Category]): Category = {
               val q = (
                    select
                    from c
                    where c.name === name
                    and c.parent === parent.getOrElse(null)
               )
          queryDao.querySingleResult(q).getOrElse(
               create(Category(name, parent)))
          }

     def createHierarchy(categories: List[String]): Category =
          {
               def createHierarchy(categories: List[String],
                    parent: Option[Category]): Category =
                    {
                         val newParent = getOrCreate(
                              categories.head, parent)
                         if (categories.tail.isEmpty)
                              newParent
                         else
                              createHierarchy(
                                   categories.tail,
                                   Some(newParent))
                    }
               createHierarchy(categories, None)
          }
}
```

The getOrCreate method gets (from the database) or creates a category. The createHierarchy takes a list of Strings and returns the persisted category or creates the hierarchy in the database. i.e. createHierarchy(List(“TV”,”46' TVs”)) will create Category(“46' Tvs”,Some(Category(“TV”,None))) and persist it.

Now this dao comes very handy:

```
val cat1 = categoriesDao.createHierarchy(List("Operating Systems",
"iMac"))

val cat2 = categoriesDao.createHierarchy(List("Operating Systems",
"iMac", "Leopard"))

val categories = List(cat1)

val p1 = productsDao.create(Product("test1", "desc1", Set(),
categories))

val updated=productsDao.update(p1,Product("test1", "desc1",
Set(),List(cat1,cat2))
```