Lift Microservice Template – Database Guide and Tutorial

[draft1]

# Goal

* Guide to integrating the microservice template with various databases

# Scope

* Lift Mapper (with mySql)
* Squeryl-Record (with MongoDB)
* Slick
* Phantom with Cassandra

This version includes only Lift Mapper. Future versions will include more.

# Lift's Mapper

## Concept

The Mapper is an Object Relational Mapping (ORM) system for integrating database to the code. It is used with relational databases, such as MySQL, PostgreSQL, or H2, and allows rows in database to be represented as Scala objects.

## Features

* Quick implementation, intuitive syntax
* JDBC driver backed
* 1-to-many and many-to-many relationship
* Foreign Key declaration
* Baked in validation
* Schemifier to create tables and columns according to the coded model
* Integrated with Lift snippets; can generate HTML forms

## Cautions

* Erroneous commands and query failures will throw an exception
* Suffers from usual ORM problems (N+1 problem, no complex/custom query available, etc.)
* Is synchronous and blocking
* Some Column type not supported out of the box, can implement yourself

## Recommended Use Case

* Low to Medium size
* Low to Medium relationship complexity
* Mostly basic queries
* Quick implementation/prototyping

## Tutorial

### Setup

For official tutorial, see here: <http://exploring.liftweb.net/master/index-8.html> , and here: <https://www.assembla.com/wiki/show/liftweb/Mapper>. Caution: a bit old and outdated, but will mostly work.

To get started using Mapper, we will need to create a database and declare a connection manager. We will use MySQL database for this tutorial. Feel free to try it on any other supported relational database.

First, create a new empty MySQL database. This guide will assume the database is in localhost and in default port.

Add the following dependency to the project in build.sbt:

"mysql" % "mysql-connector-java" % "5.1.36"

Next, we will declare a connection manager for Lift project. Add the following code to Boot.scala's boot method. Replace the database name, username and password for the database connection.

import net.liftweb.mapper.{ DB, StandardDBVendor, DefaultConnectionIdentifier }

if (!DB.jndiJdbcConnAvailable\_?) {

val vendor =

new StandardDBVendor(

"com.mysql.jdbc.Driver",

"jdbc:mysql://localhost/DATABASE\_NAME",

Full("USERNAME"),

Full("PASSWORD")

)

LiftRules.unloadHooks.append(vendor.closeAllConnections\_! \_)

DB.defineConnectionManager(DefaultConnectionIdentifier, vendor)

}

What the code does is declaring a default connection manager that will connect to the specified URL, using the specified driver, username, and password, then set it to the DB object.

### Model

Now, we can work on Mapper itself. For the purpose of this tutorial, we will create a simple student – course data and relationship. A student will have an id, name, and age. A course will have an id a name, and status.

Declaring a student model could be done like this. Put it in, for example, Student.scala

package code.model

import net.liftweb.mapper.\_

class Student extends Mapper[Student] {

def getSingleton = Student

object id extends MappedLong (this)

object name extends MappedString (this, 255)

object age extends MappedInt (this)

}

object Student extends Student with MetaMapper[Student]

There are two components here, Mapper and MetaMapper. The Mapper is the definition of each instance of the model, while the MetaMapper is a handler of the whole model. We'll see how they work in a while.

The Student class contains 2 objects, which are the fields inside the model that we want, being a MappedField of particular data type, such as in this case, MappedLong, MappedString. By the way, the second argument for MappedString is for the maximum length of the text field. The required method getSingleton provides a way to access the controlling MetaMapper.

So what we have here is a basic definition of the model, but we can do better. We can add index to id and make it the primary key field of the model.

class Student extends LongKeyedMapper[Student] {

def primaryKeyField = id

def getSingleton = Student

object id extends MappedLongIndex (this)

object name extends MappedString (this, 255)

object age extends MappedInt (this)

}

object Student extends Student with LongKeyedMetaMapper[Student]

Here, we change the extended trait to LongKeyedMapper, which means the model has a primary key, and it is of a Long type. Then, we need to tell which field is the primary key, via the method primaryKeyField, and make that field indexed by extending from MappedLongIndex instead. Notice the MetaMapper also changed to LongKeyedMetaMapper.

Since the id field is used very often, the library comes with a way to quickly create the field, like the following.

class Student extends LongKeyedMapper[Student] with IdPK {

def getSingleton = Student

object name extends MappedString (this, 255)

object age extends MappedInt (this)

}

object Student extends Student with LongKeyedMetaMapper[Student]

The trait IdPK helps with all the necessary modelling of the primary key. Declaring primaryKeyField and the field id is no longer necessary.

So we have a simple model of Student in the code, but we don't have any database tables to back it up yet. Let's fix that by adding this code in Boot.scala's boot method, after the database connection declaration.

import net.liftweb.mapper.Schemifier

import code.model.\_

Schemifier.schemify(true, true, Schemifier.infoF \_, Student)

This command will create or update the tables for the Student model upon server start. You could work without the Schemifier, but you will have to manually match the database table to the code model.

### Using the model

The query on this model is done through the MetaMapper.

> Val allStudents: List[Student] = Student.findAll()

> Val student3: Box[Student] = Student.find(3)

> Val john: Student = Student.create.name("john").age("18")

> john.save

> val johnId = newStudent.id.get

Notice that we do not create a new Student instance using new. Doing so will create problems.

The commonly used methods in KeyedMetaMapper are:

* find: return a single instance of the model, wrapped in a Box. You can put in QueryParams or, in case of LongKeyedMetaMapper, a Long of the primary key to find.
* findAll: return all instances of the model that match the supplied QueryParams, or all if none supplied, wrapped in a List.
* create: return a new, unsaved instance of the model with default values.
* count: return a Long count of the number or rows in the database that matches the supplied QueryParams.

### QueryParam

A QueryParam is a Scala representation of the condition clauses in the SQL, in other words, the things you type in after WHERE. There are a few types of QueryParams available. Here are some examples.

// find all students, order by age

> Student.findAll(OrderBy(Student.age, Ascending))

// find all students aged exactly 18

> Student.findAll(By(Student.age, 18L))

// find all students using the LIKE comparison

> Student.findAll(Like(Student.name, "jo%"))

// find all students whose name matches exactly any item in the list

> Student.findAll(ByList(Student.name, List("john", "mary", "juan")))

// you can also use several QueryParams together

> Student.findAll(

OrderBy(Student.age, Ascending),

By(Student.age, 18L))

For complete list of QueryParams, check the ScalaDoc here: <http://liftweb.net/api/26/api/#net.liftweb.mapper.package> . Look for classes that extend QueryParam and object members that extend nothing (many of these are QueryParam creators).

This is also a good time to create a shortcut model for the purpose of this tutorial; add the following method to the model handler.

object Student ... {

...

//do not use `head` in production code, Leave the Box as is

def findByNameUnsafe(s: String): Student = find(By(name, s)).head

}

### Defining Relationships

#### One to many

Next, let's add the course model. This is done in the same way as defining Student model. Save the following code to Course.scala

package code.model

import net.liftweb.mapper.\_

class Course extends LongKeyedMapper[Course] with IdPK {

def getSingleton = Course

object name extends MappedString(this, 255)

object status extends MappedBoolean(this)

}

object Course extends Course with LongKeyedMetaMapper[Course] {

def findByNameUnsafe(s: String): Course = find(By(name, s)).head

}

And modify Schemifier in Boot to include the Course model:

Schemifier.schemify(true, true, Schemifier.infoF \_, Student, Course)

Now, for example, we want to add relationship between the two models. Let's say a student can enrol in a single course. In other words, one course is related to many students. Here is how we can declare that relationship; add this field inside the Student model:

class Student ... {

...

object course extends MappedLongForeignKey(this, Course)

}

This new (concrete) field that extends MappedLongForeignKey declares that the field is a Long type and refers to the primary key field of the Model specified in the argument, in this case Course. By default, this constraint is not enforced at the database level.

Now, you can set and get a Student's course as if it is a normal field.

> val historyCourse = Course.create.name("history").status(true).saveMe

> val historyCourseId = historyCourse.id.get

//previously created

> val john = Student.findByNameUnsafe("john")

> john.course(historyCourseId).save

> val johnCourse: Box[Course] = john.flatMap{ \_.course.foreign}

The extra method that the MappedLongForeignKey has is foreign which returns a Box of the instance the key refers to. In the example above, student1Course will be a Full box with the same course as the historyCourse inside.

Next, we will make each instance of the Courses able to refer to all Students that enrol under it.

Make the following changes to the class Course

class Course

extends LongKeyedMapper[Course]

with OneToMany[Long, Course] {

...

object enroled\_students extends MappedOneToMany(Student, Student.course)

}

Note, the trait OneToMany is required, and the type refers to the model's key and the model type respectively.

The object enroled\_students is not a true field in a sense that it is backed by a column in the same table. Instead it will get the singleton Student (the 1st arg in MappedOneToMany), ask it to find all instance whose field course (the 2nd arg) matches its own primary key. Here's an example:

// Find the previously created history course

> val historyCourse = Course.findByName("history")

// create a new student, notice we don't set the course for this example

> val mary = Student.create.name("mary").age(16).saveMe

// add Mary to the history course

> historyCourse.enroled\_students += mary

> historyCourse.save

// all the students enroled in history course

> val historyStudents: Buffer[Students] = historyCourse.enroled\_students

// Mary's `course` field should be set to history Courses's primary key

> mary.reload

> mary.course.get

Notice that the result of the enroled\_students is a mutable Buffer. You can append and remove members of said Buffer and save it to manage the relationship.

#### Many to many

Let's say we change the requirement so that each student can enrol in many courses. In other words, the relationship of student and course is many to many.

First, let's clear the one to one relationship. Remove object course from class Student and the trait OneToMany and object enroled\_students from class Course. What you should have left now is two unrelated models.

To define the many-to-many relationship, we'll need another model to hold the relation. Let's call it StudentCourse, and add some functions to it

class StudentCourse extends Mapper[StudentCourse] {

def getSinglton = StudentCourse

object student = MappedLongForeignKey(this, Student)

object course = MappedLongForeignKey(this, Course)

}

object StudentCourse extends StudentCourse with MetaMapper[StudentCourse]{

def findByStudent(s:Student): List[StudentCourse] =

findAll(By(StudentCourse.student, s))

def findByCourse(c:Course): List[StudentCourse] =

findAll(By(StudentCourse.course, c))

}

Don't forget to add this model to the Schemifier in Boot as well.

What we have for each row are primary keys of a student and a course. Notice we are using a simple Mapper without a primary key for this model, as it is not required. You can add it if it is suitable for your use case.

You can now create relationship via this model.

> val math = Course.create.name("math").status(false)

//previously created instances

> val history = Course.findByNameUnsafe("history")

> val john = Student.findByNameUnsafe("john")

> val mary = Student.findByNameUnsafe("mary")

// add john to history course

> StudentCourse.create

.student(john.id.get)

.course(history.id.get)

.save

// add john to math course

> StudentCourse.create

.student(john.id.get)

.course(math.id.get)

.save

// add mary to math course

> StudentCourse.create

.student(mary.id.get)

.course(math.id.get)

.save

// find john's courses

> val johnCourses: List[Course] = for {

sc <- StudentCourse.findByStudent(john)

c <- sc.course.foreign

} yield c

// find all students in math course

> val mathStudents: List[Student] = for {

sc <- StudentCourse.findByCourse(math)

s <- sc.student.foreign

} yield s

While this works, it is rather unwieldy, and we can improve it by making each Student and Course able to refer to each other directly.

Make the following changes to the class Student

class Student extends ... with ManyToMany[Student] {

...

object courses extends MappedManyToMany(

StudentCourse,

StudentCourse.student,

StudentCourse.course,

Course)

}

The class MappedManyToMany takes 4 arguments, the first states which model stores the relationship, the second is the field that refers to the owner model's key, the third is the target model's key, and the forth the singleton of the target model.

The same thing could be done on the class Course

class Course extends ... with ManyToMany[Course] {

...

Object students extends MappedManyToMany(

StudentCourse,

StudentCourse.course,

StudentCourse.student,

Student)

}

The usage is the same as MappedOneToMany

> val john = Student.findByNameUnsafe("john")

> val history = Course.findByNameUnsafe("history")

> val math = Course.findByNameUnsafe("math")

> john.courses ++= Seq(history, math)

> john.save

> val johnCourses: Buffer[Course] = john.courses

> val mathStudents: Buffer[Student] = math.students

### Validation

The Mapper library provides a way to validate the fields in models. All mapped fields have a method called validations which is a list of functions that evaluate the field's value. For example, if we want the name of the students to be at least 1 and at most 255 characters long (to prevent exception when writing over the database column's size), we can add the following.

class Student ... {

...

object name extends ... {

override def validations =

valMinLen(1,"name must be at least 1 characters") \_ ::

valMaxLen(255, "name must be at most 255 characters") \_ ::

super.validations

}

...

}

The methods valMinLen and valMaxLen are methods provided by the library. You can create your own validation functions. For example, let's say the age of the students must be over 13 years old.

import net.liftweb.util.FieldError

class Student ... {

...

object age extends ... {

private def ageOver13: Int => List[FieldError] = value => value match{

case v if v > 13 => Nil // pass validation

case \_ => FieldError(this, "Student's age must be above 13") :: Nil

}

override def validations = ageOver13 :: super.validations

}

...

}

In order to use the validations, you must manually call the method validate on the instance. Calling only save or saveMe will **not** go through validations. Conversely, this means you can save while ignoring the validation rules. Here's a simple example.

> val newStudent = Student.create.name("").age(3)

> val newStudentErrors: List[FieldError] = newStudent.validate

> newStudentErrors.length match {

case 0 => newStudent.save

case \_ => handleFailedStudentValidation(newStudentErrors)   
 // do this yourself, e.g. report the failure

}

// you still can do this at your own risk. The instance will save

> newStudent.save

### LifeCycle callback

MetaMappers have simple hook methods for you to implement actions. Here are some examples: beforeSave, afterSave, beforeDelete, afterDelete. They will be run at various points in the query lifecycle as the names suggest. However, you should note the order might not be exactly as you expect. Do read this: <http://stackoverflow.com/questions/4036014/lift-mapper-aftercreate-before-aftersave>

To use, override those methods inside the MetaMappers.

object Course extends Course with LongKeyedMetaMapper[Course] {

...

override def beforeSave = logCourseCreate \_ :: super.beforeSave

private def logCourseCreate(c: Course): Unit = logger.info(

"New course created" + c.name.get)

}

### Optimising Queries

Like all ORM system, Mapper tends to make a lot of small queries. The N+1 problem exists here too, but there is a way to mitigate this. When making find queries that involve the foreign key, you can add a QueryParam called PreCache to load the related instances early and avoid multiple query calls.

// only when field Student.course is a MappedLongForeignKey

> val students = Student.findAll(Order(Student.age, Ascending),

PreCache(Student.course))

> students.flatMap{ s: Student => s.course.foreign.name }

Sometimes, it's better to just write your own SQL conditions. This could be done by using BySql. You **NEED** to sanitize the strings if taken from user input. Use caution.

> Students.findAll(BySql(

"age between ? and ?",

IHaveValidatedThisSQL("YourName","2016-05-13"),

10, 19))

BySql supports parameters. Use ? as shown, and put the parameter values as the 3rd argument onwards in order. The class IHaveValidatedThisSQL does nothing but to show who to give credit to when query is running well or to blame when trouble hits the fan.

Sometimes, it's best to write the whole SQL yourself. You can do this by using DB.runQuery. Great care must be taken to prevent injection, as there is no escaping measure built in.

> import net.liftweb.mapper.DB

> val sql =

"""SELECT c.name as 'course', count(s) as 'count' FROM course AS 'c'

JOIN student AS 's' on s.course = c.id

WHERE c.status = 1 AND s.age = ?

GROUP BY c.name

ORDER BY c.name ASCENDING

"""

> val age = 15

> val result:(List[String], List[List[String]]) = DB.runQuery(sql, age)

The result returns a tuple of Lists. The first List in the tuple is the columns. The second List is the result rows; each row is a List of String whose elements are the value of the columns.

Example result from previous query:

( List("course", "count"),

List("english", "14") ::

List("math", "5") :: Nil

)