# ActiveJDBC == ActiveRecord for Java

By Igor Polevoy

August 2010

# Who the heck is Igor Polevoy?

and why should we trust him?

- Developer like you
- Architect
- Teach Java/Rails at DePaul IPD
- Battle-beaten on IT projects
- Currently coding away at Productive Edge

# ..but why?

- Hibernate
- JDO
- JPA
- iBatis
- Castor
- Torque
- JPersist
- …20 more??

# Dissatisfaction

- Hibernate: very powerful, but complicated, hard to setup, people have theories how it works, feels like a straight jacket, hard to learn
- JPA – different configuration on every platform, not as feature reach as Hibernate, feels like a black box.
- Most of all: getters/setters
- Not OO!

# ActiveRecord just feels right

- DHH is a great API designer
- AR simple to use, yet provides a level of sophistication
- Does not try to solve every problem under sun, but covers 80% of what you need; you can get your work done faster…much faster!
- Complicated DB schemas ==usually== wrong design

# Who are parents of invention?

If necessity is a mother

then…

laziness is its father!

# ActiveJDBC Design principles

- Should infer metadata from DB
- Should be very easy to work with
- Should reduce amount of code to a minimum
- No configuration, just conventions
- Conventions are overridable in code
- No need to learn another language
- No need to learn another QL - SQL is sufficient
- Code must be lightweight and intuitive, should read like English
- No sessions, no "attaching, re-attaching"
- No persistence managers.
- No classes outside your own models.
- Models are lightweight, no transient fields
- No proxying. What you write is what you get (WYWIWYG :))
- Should have the least possible resistance to startup a project
- No useless getters and setters (they just pollute code). You can still write them if you like.
- No DAOs and DTOs - this is mostly junk code anyway

# ActiveJDBC is thin

- The main design principle:
  - thin!!!
  - 4 – 5 methods on stack traces
- Fast: simple non-scientific performance test: read 50K records from DB:
  - 2 times slower than JDBC
  - 40% times faster than Hibernate
  - no cache enabled
- No dependencies...almost
  - Slf4j
  - OSCache (only if needed)
- Size ~ 100k jar

# Surrogate PKs

- Same as ActiveRecord
- Easy with MySQL, PostgreSQL
- Harder with Oracle (but doable, more SQL, see website)

# Writing Models (Entities)

Fully functional model:

```
public class Person extends Model {}
```

- Maps to table PEOPLE
- Know all necessary details about this table from database
- Immediately ready for CRUD operations

# Select

```
List<Person> johns =

  Person.where(“last_name =‘John’”);

//iterate:
for(Person p: people){
  log(p);
}
```

# Select with associations

```
class Address extends Model{}
//table ADDRESSES has column person_id


Person p = Person.findFirst(“ssn = ?”,”123-34-
   5678”);

List<Address> addresses = p.getAll(Address.class);
```

Same API exactly for One to Many, Many to Many and Polymorphic associations!

# Select with Fluent Interfaces

```
List<Person> people =
        Person.where(“age > 21”)
        .limit(40)
        .offset(20)
        .orderBy("age asc");
```

# Create new data

```
Person p = new Person();
p.set("first_name", "John");
p.set("last_name", "Doe");
p.set("dob", "1935-12-06");
p.saveIt();
```

# Create shortcuts (method chaining)

```
Person p = new Person();
    p.set("name", "John")
     .set("last_name","Doe")
     .set("dob", "1935-12-06")
     .saveIt();
```

# Create class shortcut

```
Person.createIt("first_name",
  "Sam", "last_name", "Margulis",
  "dob", "2001-01-07");
```

```
Arguments: name, value, name1,
  value1, etc., reads like
  English
```

# Validations

```
public class Person extends Model {
 static{
   validatePresenceOf("first_name","last_name";
  }
}
```

# Validation, fluent interfaces style

```
public class Percentage extends Model {
    static{
        validateNumericalityOf("total")
        .allowNull(true)
        .greaterThan(0)
        .lessThan(100)
        .onlyInteger()
        .message("incorrect 'total'");
    }
}
```

# save() and saveit()

- Same as `save` and `save!` in ActiveRecord

# One to Many

**User:**

```
public class User extends Model {}
```

**Address:**

```
public class Address extends Model {}

List<Address> addresses =
  user.getAll(Address.class);
```

Condition: table **ADDRESSES** needs to have column **user_id**

# Many to many

Model for table DOCTORS:

```
public class Doctor extends Model {}
```

Model for table PATIENTS:

```
public class Patient extends Model {}
List<Patient> patients =
  doctor.getAll(Patient.class);
```

**or:**

```
List<Doctor> doctors =
  patient.getAll(Doctor.class);
```

- Condition: have table **DOCTORS_PATIENTS** with columns **doctor_id**, **patient_id**

# Adding/removing associated entries

**Adding:**

```
user.add(address);          //one to many
doctor.add(patient);        //many to many
article.add(tag);           //polymorphic
```

**Removing:**

```
user.remove(address);       //one to many
doctor.remove(patient);     //many to many
article.remove(tag);        //polymorphic
```

# Auto-generated fields

- `created_at` – set at creation time
- `updated_at` – set at update time, as well as batch updates
- `id` (can override name, value generated by DB)

# Caching

```
@Cached
public class Library extends Model {}
```

Supports efficient clustering (courtesy OSCache)

# Life-cycle callbacks

```
void beforeSave();
void afterSave();
void beforeCreate();
void afterCreate();
void beforeDelete();
void afterDelete();
void beforeValidation();
void afterValidation();
```

# Callback example

```
public class User extends Model{
    public void beforeSave(){
        set("password" encryptPassword());
    }
    private String encryptPassword(){
        //do what it takes
        return encrypted;
    }
}
```

# Lazy/Eager

ActiveJDBC is lazy by default.

If you need to load eager, use `include()`:

```
List<User> users =
    User.findAll().orderBy("id")
        .include(Address.class);
```

Addresses are loaded eagerly.

Include works for:

- One to many children
- One to many parents
- Many to many

# Logging

- Uses SFL4J – hookup a log system *de jur*
- By default will not log anything
- Will log everything at INFO level:

```
java -Dactivejdbc.log
com.acme.YourProgram
```

- Will log whatever matches regexp:

```
-Dactivejdbc.log=your-reg-exp
```

- Can change matching expressionat run time

# Questions