

ActiveObjects

Basic Concepts

The underlying principle behind ActiveObjects is that ever-popular catch phrase: convention rather than configuration. With ActiveObjects, you create the abstract Java class interfaces using standard Java data object conventions (`get`, `set`, `is`, etc...) and ActiveObjects handles the wiring to the database. In fact, ActiveObjects will even generate the database-specific schema to correspond with the specified Java class model.

From a developer standpoint, using ActiveObjects is as simple as the following:

```
public interface Person extends Entity {
    public String getFirstName();
    public void setFirstName(String name);

    public String getLastName();
    public void setLastName(String name);
}

// ...
EntityManager manager = new EntityManager("jdbc:mysql://localhost/ao_test", "user", "password")

Person frank = manager.create(Person.class);
frank.setName("Frank");
frank.setName("Smith");
frank.save();
```

The code specified above is fully functional; there are no unspecified configuration files. In fact, the only missing step would be to create the corresponding database schema. ActiveObjects can handle this step too, using a feature called migrations:

```
// ...
manager.migrate(Person.class);
```

This will execute DDL something like the following (assuming MySQL as in the URI above):

```
CREATE TABLE person (
    id INTEGER AUTO_INCREMENT NOT NULL,
    firstName VARCHAR(255),
    lastName VARCHAR(255),
    PRIMARY KEY (id)
);
```

With migrations, you don't need to worry about making changes to your schema, or keeping your schema in sync with your entity model in code. Supposing you add a `getAge():int` method (and its corresponding setter) to the `Person` entity in the above example. If the `person` table has already been

created, it probably contains data. It would be ugly and painful to write scripts to generate a new version of the table, migrate the old data over and delete the old table. Instead of this mess, all that is necessary is another call to the `migrate` method (as shown above). This time, instead of generating the full table from scratch, ActiveObjects will inspect the existing schema and determine that the only necessary action is the addition of the `age` field. To this end, it will execute *only* the following DDL statement:

```
ALTER TABLE person ADD COLUMN age INTEGER;
```

All of the data has been retained, no ugly scripts were written and the entity model in code is once again in sync with the database schema. This powerful functionality allows for things such as refactoring your entity model, adding constraints and indexes and so on all without adversely effecting your database.

Switching databases is as easy as changing the JDBC URI value in the EntityManager constructor. There is never any need to be concerned about the ins and outs of database specifics or to write any SQL by hand. In fact, you can use ActiveObjects and take full advantage of database functionality without ever once opening a database client or even understanding the basics of SQL or RDBMS.

In fact, any Java developer can use ActiveObjects, even if they have no experience whatsoever in databases or their underlying concepts. The framework is designed so that it “feels” natural to any Java developer, not requiring any knowledge of relational databases.

Internals

While the ActiveObjects API does allow for some rather unusual usage patterns (such as the construction of entity definitions as pure Java interfaces), the code to support this API isn’t all that complex. Much of the heart and soul of the ActiveObjects library revolves around the concept of reflective instance proxying.

Instance proxying is a lesser known part of the Java reflection API which allows a specified instance of the `InvocationHandler` class to intercept all method calls to a contrived instance of a specified class type. This is how the `EntityManager` can dynamically create an instance of an arbitrary `Entity` sub-interface. In fact, the instance created by `EntityManager` is extremely special in that it will dynamically parse all method calls, attempting to proxy the relevant calls down to the database. This is how a call to an accessor within an entity instance will immediately correspond to a call to the database.

The instance proxy `InvocationHandler` for ActiveObjects is the `EntityProxy` class. `EntityProxy` has a single method - `invoke` - which receives all method calls to the corresponding `Entity` instance. `EntityProxy` then applies a set of rules to the method name, method annotations and method parameters to determine if it is a data accessor method. If the method should correspond to a database function, `EntityProxy` handles the retrieval of a database connection and the execution of the appropriate SQL.

For more details on ActiveObjects’s implementation, refer to the [javadocs](#).

Example Usage

The first step in building an application which uses ActiveObjects is to define the entities. This is done in pure Java (no XML configuration). The entity definitions are database non-specific:

```
// Person.java
public interface Person extends Entity {
    public String getName();
    public void setName(String name);

    public int getAge();
    public void setAge(int age);
}
```

```

        @SQLType(Types.CLOB)
        public String getComment();

        @SQLType(Types.CLOB)
        public void setComment(String comment);

        public Family getFamily();
        public void setFamily();

        @ManyToMany(PersonToPerson.class)
        public Person[] getPeople();
    }

    // Family.java
    public interface Family extends Entity {
        public String getName();
        public void setName(String name);

        @OneToMany
        public Person[] getPeople();
    }

    // PersonToPerson.java
    public interface PersonToPerson extends Entity {
        public Person getPersonA();
        public void setPersonA(Person person);

        public Person getPersonB();
        public void setPersonB(Person person);
    }

```

Once we have the entity definitions, the DDL statements can be auto-generated for whatever database you may be using. This is done using migrations. Here's what the DDL would look like as generated for the MySQL database:

```

CREATE TABLE family (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(255),
    PRIMARY KEY (id)
) ENGINE=InnoDB;

CREATE TABLE person (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(255),
    age INTEGER,
    comment TEXT,
    familyID INTEGER,
    PRIMARY KEY (id),
    CONSTRAINT fk_person_familyID FOREIGN KEY (familyID) REFERENCES family(id)
) ENGINE=InnoDB;

CREATE TABLE personToPerson (
    id INTEGER NOT NULL AUTO_INCREMENT,
    personAID INTEGER,

```

```

    personBID INTEGER,
    PRIMARY KEY (id),
    CONSTRAINT fk_personToPerson_personAID FOREIGN KEY (personAID) REFERENCES person (id),
    CONSTRAINT fk_personToPerson_personBID FOREIGN KEY (personBID) REFERENCES person (id)
) ENGINE=InnoDB;

```

You'll notice the usage of convention to determine field names, primary and foreign keys, etc...

Of course, an ORM layer isn't very useful unless you actually *use* it. Here's some sample code showing some basic CRUD operations with our new model:

```

// ...

// retrieves an EntityManager relevant to the specified URI
// if available (i.e. the classpath set appropriately), the connection will be pooled
EntityManager manager = new EntityManager("jdbc:mysql://localhost/ao_test", "user", "password")

Family family = manager.create(Family.class);
family.setName("Spiewak");
family.save();

Person me = manager.create(Person.class);
me.setName("Daniel Spiewak");
me.setAge(27);
me.setComment("I love databasing");
me.setFamily(family);
me.save();

Person you = manager.create(Person.class);
you.setName("Joe Blow");
you.setAge(23);
you.setComment("Guess who?");
you.setFamily(family);
you.save();

PersonToPerson relation = manager.create(PersonToPerson.class);
relation.setPersonA(me);
relation.setPersonB(you);
relation.save();

family.getPeople();           // ...returns new Person[] {you, me}
you.getPeople();             // ...returns new Person[] {me}

/*
 * notice, this is the first use of SQL in the whole example, and
 * it's a contrived usage at that
 */
Family[] families = manager.findWithSQL(Family.class, "familyID", "SELECT DISTINCT familyID FRO

// returns any person with age greater than or equal to 18
Person[] overAge = manager.find(Person.class, "age >= ?", 18);

/*
 * notice the varargs parameters, as well as the direct use of an
 * entity instance without worrying about the ID value

```

```

*/
Person[] inFamilyOver21 = manager.find(Person.class, "age >= ? AND familyID = ?", 21, family);

```

As you see, much of the framework is designed to completely encapsulate the developer from the complexities of SQL and even from the precise specifics of the schema. The only place where this rule is broken is in the `find` methods, which are designed to leverage the full power of SQL, rather than completely shelter the developer. .. :mode=rest:

Supported Databases

Most major databases are currently supported by ActiveObjects. This data is only current to the **0.8** stable release.

Database	URI Protocol	Support
Derby	<ul style="list-style-type: none"> • jdbc:derby • jdbc:derby:// 	Stable
HSQLDB	<ul style="list-style-type: none"> • jdbc:hsqldb • jdbc:hsqldb:// 	Well tested and very stable
MS SQL Server 2005 and 2007	<ul style="list-style-type: none"> • jdbc:sqlserver:// • jdbc:jtds:sqlserver:// 	Stable
MySQL	<ul style="list-style-type: none"> • jdbc:mysql:// 	Well tested and very stable
Oracle	<ul style="list-style-type: none"> • jdbc:oracle:thin • jdbc:oracle:oci 	Under development
PostgreSQL	<ul style="list-style-type: none"> • jdbc:postgresql:// 	Well tested and stable

Why ActiveObjects?

Possibly the most important question to answer regarding ActiveObjects is to explain: why another ORM? The Java ORM framework genre is completely dominated by the JBoss framework, Hibernate. In fact, Hibernate is so widely used and respected that it's even been ported to .NET (NHibernate). So, if there's already a widely respected, widely used and mature framework which seems to satesfy all important use-cases, why throw another into the mix?

The answer really comes back to complexity. Hibernate is an incredibly complex framework. Granted, it is complex mainly because it is powerful, but sometimes - most of the time even - the required use-case is very simple and doesn't require all of Hibernate's complexities. ActiveObjects is designed from the ground up to be as easy to use as possible, with a bare minimum of configuration. In fact, to date ActiveObjects doesn't even have a single hook which takes an XML (or any other format) configuration file. Any and all configuration is either guessed from code, or easily set in code through the discrete use of annotations.

There is an increasing trend in the industry towards "convention over configuration." This is most reflected in RAD frameworks such as Ruby on Rails. Much of the accepted industry pundits agree that writing 20 lines of code is much easier than 20 lines of code and 150 lines of XML configuration (go figure). ActiveObjects follows this practice as much as possible in its implementation. In fact, much of the inspiration for the framework comes from Rails's excellent ActiveRecord ORM (hence, the name). The exception to this inspiration would be that ActiveObjects does *not* impose English pluralization rules by default (though it is capable of such functionality when it is desirable).

In fact, ActiveObjects strives so hard to be a simple and easy-to-use persistence framework that some functionality (such as distributed transactions) has been simply omitted. The reasoning behind this is that 99% of use-cases do not call for such extreme measures. If your project does require such complex behavior within your ORM, you should be using Hibernate. It's as simple as that. **ActiveObjects is not intended to supplant Hibernate.** Rather, its goal is to be an easier and lighter alternative for the many common scenarios which do not call for all of Hibernate's awesome power.

In short, ActiveObjects attempts to make database development simple and fun again. It abstracts the developer from the intricacies of the database schema and the particulars of how to access it. Using ActiveObjects, the only thing the developer needs to worry about is the high-level concept of object-oriented design and data encapsulation.