eXtended Objects

Dirk Mahler

Table of Contents

This document describes the core concepts of eXtended Objects.

Introduction

Modeling data objects for datastores (i.e. databases) in various domains *often requires common properties (e.g. lastChangedBy, created, name, etc.) to be present in several data types. This is commonly solved by defining abstract base types representing a specific aspect or role and defining these properties here. Concrete data types may derive from these base types if the information is needed. This approach comes to its limits if more than one aspect shall be re-used in one POJO as multiple inheritance is not supported for classes in the Java language.*

eXtended Objects (XO) allows modeling types as interfaces by specifying the required accessor methods (i.e. getter and setter) and composing these types either statically or dynamically. The XO API uses well-known concepts from existing persistence APIs (e.g. JPA, JDO) but extends them with functionality which is enabled by using interfaces to define the data objects, e.g. type migration at runtime. The XO implementation itself is datastore-agnostic, i.e. it does not depend on specific datastore concepts like relational or graph databases, document stores etc. but allows using them. Datastore specific implementations can be provided by implementing the SPI (Service Provider Interface), the reference implementation is based on the graph database Neo4j.

Core Concepts

Static vs. Dynamic Composition

A simple example shall be used to explain the difference between both concepts of composition:

Person.java

```
@DatastoreEntity // A datastore specific annotation marking this interface as an
entity
public interface Person {
  String getName();
  void setName();
}
```

Actor.java

```
@DatastoreEntity
public interface Actor extends Person {
  List<Movie> getActedIn();
}
```

Director.java

```
@DatastoreEntity
public interface Director extends Person {
  List<Movie> getDirected();
}
```

Movie.java

```
@DatastoreEntity
public interface Movie {
  String getTitle();
  void setTitle();

  Director getDirectedBy();
  void setDirectedBy(Director directedBy);

  List<Actors> getActors();
}
```

Using the XO API the following use case can be implemented:

RaidersOfTheLostArk.java

```
public class RaidersOfTheLostArk {

  public static void main(String[] args) {
    // Bootstrap XO, a file META-INF/xo.xml is expected on the classpath
    XOManagerFactory xmf = XO.createXOManagerFactory("movies");

    // Obtain a XOManager (i.e. a connection to the datastore)
    XOManager xm = xmf.createXOManager();

    // Begin a transaction
    xm.currentTransaction.begin();

    // Create a data object representing an actor
    Actor harrison = xm.create(Actor.class);
    harrison.setName("Harrison Ford");

    // Create a data object representing a director
    Directory steven = xm.create(Director.class);
    steven.setName("Steven Spielberg");

    // Create a data object representing a movie
    Movie raiders = xm.create(Movie.class);
    raiders.setTitle("Raiders Of The Lost Ark");

    // Set the relations between all the created data objects
    raiders.setDirectedBy(steven);
    raiders.getActors().add(harrison);

    // Commit the transaction
    xm.currentTransaction.commit();

    // Close the XOManager
    xm.close();

    // Close the XOManagerFactory (on shutdown of the application)
    xmf.close();
  }
}
```

The types Actor and Director use static composition by extending from the type Person. But what happens if a person works in both roles: actor and director? A new type extending from both would be required if static composition was used:

DirectingActor.java

```
@DatastoreEntity
public interface DirectingActor extends Director, Actor {
}
```

The situation gets even more complex if other roles (like screenwriter, editor, etc.) are added to the domain model. Each combination of roles must be represented by such a type. Dynamic composition helps getting around this problem:

TempleOfDoom.java

```
public class TempleOfDoom {

  public static void main(String[] args) {
    XOManagerFactory xmf = XO.createXOManagerFactory("movies");
    XOManager xm = xmf.createXOManager();
    xm.currentTransaction.begin();
    Actor harrison = xm.create(Actor.class);
    harrison.setName("Harrison Ford");

    // Create Steven as composite object using more than one role
    CompositeObject steven = xm.create(Director.class, Actor.class);

    // Use Steven in the role of an actor
    steven.as(Actor.class).setName("Steven Spielberg");

    Movie temple = xm.create(Movie.class);
    temple.setTitle("Temple Of Doom");

    // Use Steven in the role of a director as the "director" property of the type
Movie requires it
    temple.setDirectedBy(steven.as(Director.class));
    temple.getActors().add(harrison);

    // Steven also acted in "Temple Of Doom" (according to IMDB...)
    temple.getActors().add(steven.as(Actor.class));
    xm.currentTransaction.commit();
    xmf.close();
  }
}
```

XO Bootstrapping in OSGi environments

Activator.java

```
public class Activator implements BundleActivator {

  public void start(BundleContext context) throws Exception {

    // performs a lookup in the OSGi service registry
    XOManagerFactory xmf = XOSGi.createXOManagerFactory("movies");

    XOManager xm = xmf.createXOManager();
    xm.currentTransaction.begin();
    Actor harrison = xm.create(Actor.class);
    harrison.setName("Harrison Ford");

    //

    xm.currentTransaction.commit();
    xmf.close();
  }
}
```

Entity Migration At Runtime

There may be situations where an existing data object needs to be migrated to also represent other types. Using the above example the fact that the director also acted in the movie might have been discovered after the data object has been created using the type Director. XO offers a way to perform a migration at runtime and allows adding (or removing) roles (i.e. types):

TempleOfDoom.java

```
public class TempleOfDoom {

  public static void main(String[] args) {
    XOManagerFactory xmf = XO.createXOManagerFactory("movies");
    XOManager xm = xmf.createXOManager();
    xm.currentTransaction.begin();

    // Create Steven as a director
    Director steven = xm.create(Director.class);
    steven.setName("Steven Spielberg");

    Movie temple = xm.create(Movie.class);
    temple.setTitle("Temple Of Doom");
    temple.setDirectedBy(steven);
    xm.currentTransaction.commit();

    // Some days later a fan discovers that Steven also acted in Temple Of Doom
    xm.currentTransaction.begin();
    CompositeObject multiTalentedSteven = xm.migrate(steven, Director.class, Actor
.class);
    temple.getActors().add(multiTalentedSteven.as(Actor.class));

    xm.currentTransaction.commit();
    xmf.close();
  }
}
```

API Overview

eXtended Objects provides an API which re-uses concepts of other frameworks and standards like JPA or JDO:

XOUnit

A configuration defining a datastore configuration, the managed entity and relation types and several settings like validation and concurrency management. Can either be defined as XML descriptor by providing a concrete instance.

XOManagerFactory

Factory for XOManager instances which is configured using a XOUnit.

XOManager

Represents an active session with the datastore, provides

- operations to create, find, delete entities or relations
- a query factory

- access to the associated XOTransation object.

XOTransaction

Allows control of datastore transactions if supported by the underlying datastore.

Query

A user defined query which may be executed against the datastore.

Available Datastore Implementations

Neo4j

https://github.com/buschmais/extended-objects (part of the core distribution)

Titan Datastore

https://github.com/PureSolTechnologies/extended-objects-titan

TinkerPop Blueprints Datastore

https://github.com/BluWings/xo-tinkerpop-blueprints

OrientDb Datastore

https://github.com/BluWings/xo-orientdb