Athena Framework Java Developer's Guide

AthenaSource

Published Mar 2011

Table of Contents

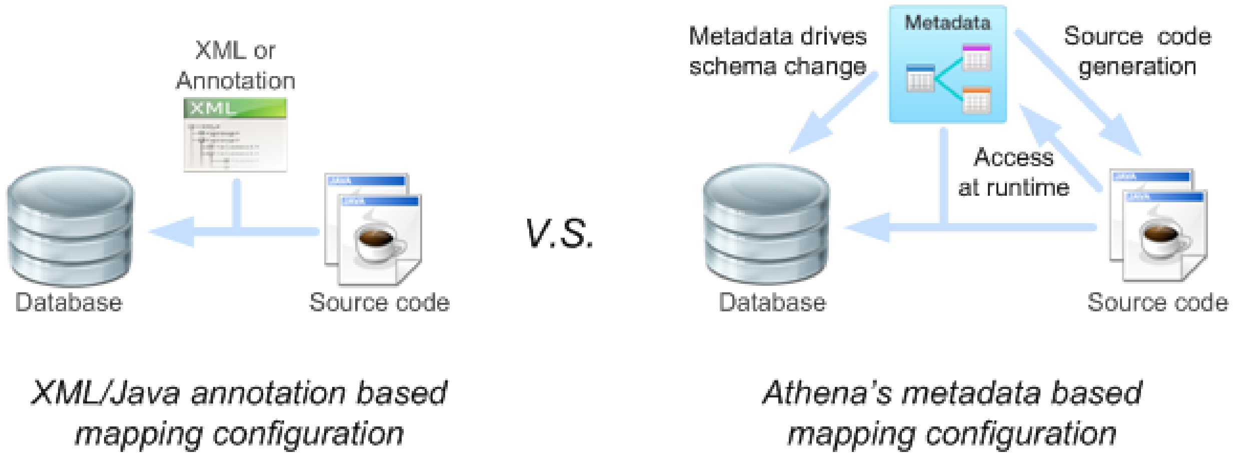
1. Introduction to Athena Framework for Java

1.1 Overview of Athena Framework

Athena Framework is a full fledged enterprise object-relational mapping (ORM) framework that employs metadata as mapping configuration. It greatly simplifies Java web application development by removing the requirement of manual mapping and manual database schema updating. In addition to Java object persistence, Athena provides powerful EJBQL querying execution, comprehensive code generation, built-in cloud ready multi-tenancy support, and optional Flex/Flash object remoting service. Athena can be easily integrated with other libraries like Struts or Spring to provide full stacks of service.

Metadata as the Single Source of Truth

In Athena, metadata refers to the collection of all entities, attributes and relationships in database modeling for the application. Any change made on metadata reflects immediately in the database schema and domain object classes. For example, when you add a new attribute named `fullName` to entity `Employee`, a new column will be automatically inserted to the corresponding table in the database and a new field will be available in the `Employee`'s domain class when you generate the source code. Architectures of traditional XML/Java annotation based ORM frameworks and Athena are compared below:


XML or Annotation
XML
Database
Source code
V.S.
Metadata
Metadata drives schema change
Source code generation
Access at runtime
Database
Source code
XML/Java annotation based mapping configuration
Athena's metadata based mapping configuration

Implementing Changes at the Speed of Thought

Athena realizes true rapid application development by allowing developers to implement changes easily and quickly. Let's say, we need to change `Person.fullName`'s type from `CHAR(100)` to `NVARCHAR(256)`. For those who use traditional ORM frameworks, they need to manually change database table's column type and to update XML or Java annotation mapping configuration, and updates the UI validation code. Such steps are time-consuming and error prone. Athena comes to rescue: you only need to change the attribute type and save it on the user friendly Athena Metadata Workbench. Athena automatically updates the database schema and generate updated source code. Developers' productivity gets significant boost by Athena.

Gaining Total Control of EJBQL Execution

When performing EJBQL queries in Athena, you do not need to guess which relationships will be loaded and which will not be loaded. Instead, you can specify explicitly which relationships will be loaded and how they should be loaded. For example, the query below selects `Employee`s with relationships of `department` and `projects`:

```
SELECT e FROM Employee e [e.department:J, e.projects:S]
```

The relationship prefetch rules in the square brackets specify that relationship `department` shall be resolved through join while `projects` through sub-select query.

Fine-Grained Query Specific Partial Attribute Loading

Some other ORM frameworks allow the developer to specify the fetch policy for an attribute. Athena allows queries instead of the attribute to control the loading behavior. In Athena, you may explicitly specify attributes to be loaded for partial objects:

```
SELECT e FROM Employee e {po_e='nameFull, bornYear'} // Load partial object with two attributes only
```

Developing Multi-Tenancy Cloud SaaS Applications Easily

A multi-tenancy application enable a single instance of the software runs on a server, serving multiple client organizations (tenants). Athena allows you to easily develop shared schema multi-tenancy applications like salesforce.com. To turn an application to multi-tenancy, you simply set the `multitenancy` flag in the configuration. For example, EJBQL `SELECT e FROM Employee e LEFT JOIN FETCH e.dept` results the following native SQLs when `multitenancy` is true and false respectively:

```
SELECT e.employee_ID, e.fullName, e.department_ID, d.department_ID, d.nameFull FROM Employee e LEFT »
 OUTER JOIN Department d ON e.department_ID = d.department_ID

SELECT e.employee_ID, SELECT e.employee_ID, e.ORG_ID, e.fullName, e.department_ID, d.department_ID, »
 d.ORG_ID, d.nameFull FROM Employee e LEFT OUTER JOIN Department d ON e.department_ID = d.department_ID »
 AND d.ORG_ID = 1 WHERE e.ORG_ID = 1
```

As Athena handles multi-tenancy automatically, you can focus on implementing business logic.

Switching Between Soft Deletion and Hard Deletion Quickly

Soft deletion is one of the key requirements for many enterprise applications to ensure data integrity and auditing. Athena allows you to switch between soft deletion and hard deletion quickly using the `deletion-policy` option in the configuration. By default, hard deletion is used. To switch to soft deletion, you simply set deletion policy to `soft`.

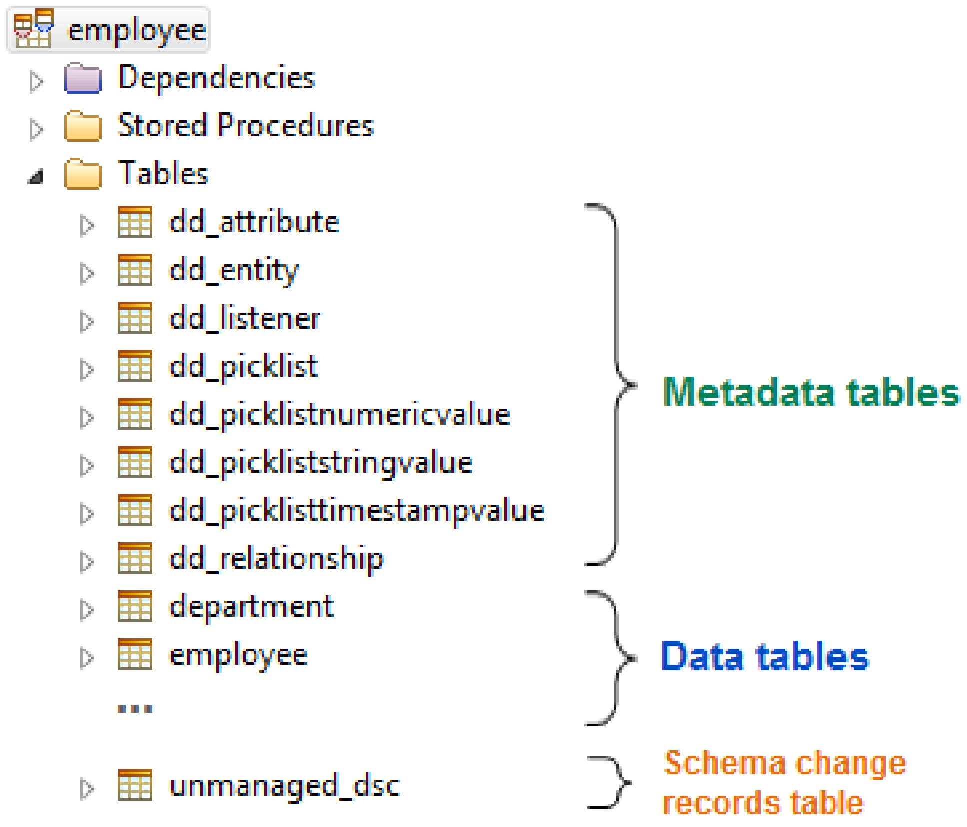
The Ultimate Solution to Database Versioning and Migration

Database migration is heavily involved in development, testing and product upgrading. For example, when you help a customer to upgrade a product from version 1 to version 2, you need to update the jar files as well as the database schema. It's usually required that all existing data should be preserved when the database schema gets updated. It implies that you need to keep track of all code changes as well database schema changes. Normally, you use a revision control system such as Subversion to version code changes. As for database schema changes, many developers use migration scripts to manually version the changes, for example, 001_db_init.sql, 002_emp_table_added.sql, 003_emp_field_name_changed.sql, etc. To migrate a database schema from version M to version N, you simply apply all scripts between M+1 to N.

Using the same idea behind database migration scripts, Athena stores all incremental changes (delta script) to the database schema in the table `unmanaged_dsc`. When you use metadata workbench to make changes, database migration scripts are stored in the table automatically. This automated process significantly reduces the complexity of database versioning and migration.

1.2 Where is Metadata Stored?

Metadata plays the key role in Athena framework. Metadata is stored in the same database as business data as illustrated below:



There are 8 metadata tables that store all kinds of metadata like entities, attributes, relationships and picklists. Names of metadata tables are prefixed by 'dd_' ('dd' stands for legacy data dictionary). As mentioned in the section called “The Ultimate Solution to Database Versioning and Migration”, `unmanaged_dsc` stores incremental database schema changes for database versioning and migration.

Metadata should only be modified through Athena Console and Metadata Workbench.

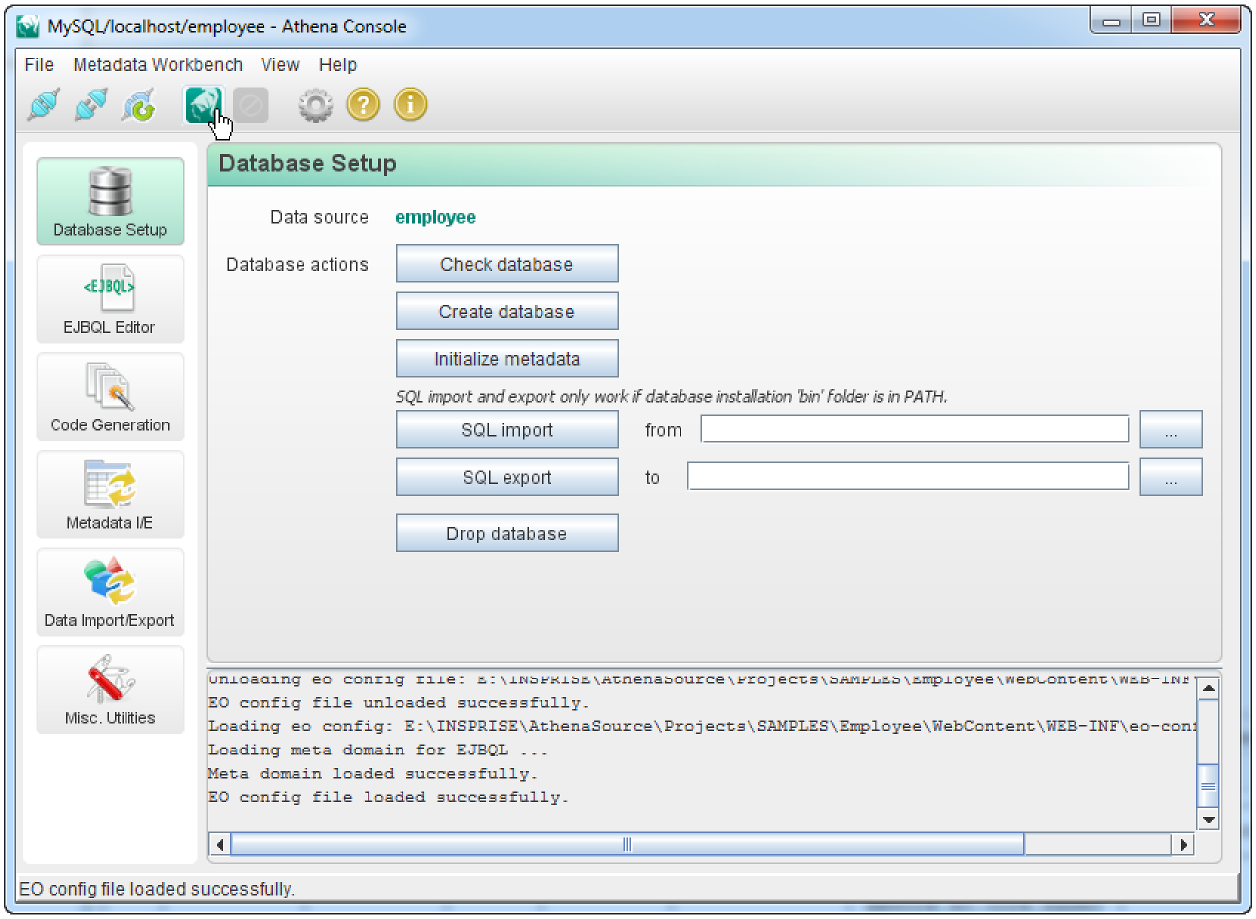
1.3 Athena Console and Metadata Workbench

The runtime of Athena framework is packaged as a few jar files. Besides the runtime, Athena Console and Metadata Workbench are two user friendly tools that help you to manage application configuration and metadata.

You may visit www.athenaframework.org/downloads to download and install Athena Console and Metadata Workbench. Note that Metadata Workbench is bundled in Athena Console.

Athena Console

Athena Console provides utilities for editing application configuration files (usually named as 'eo-config.xml') and performing database operations.



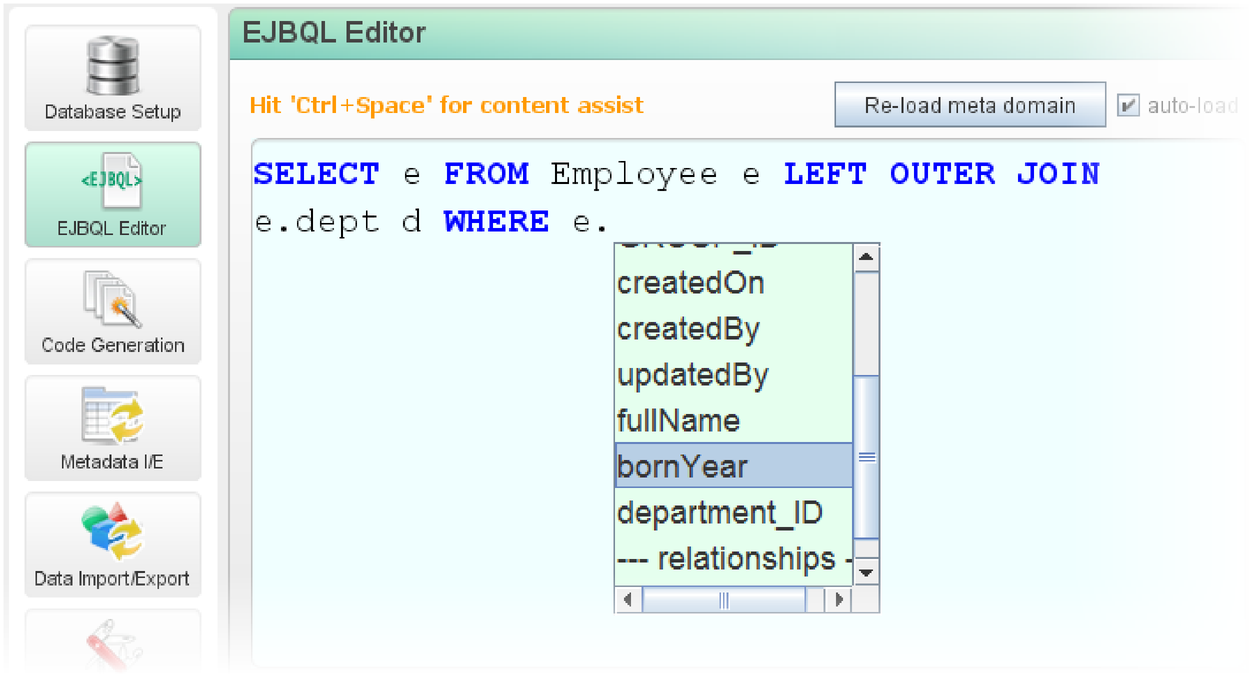
SQL import and export only work if database installation 'bin' folder is in PATH.

```
Unloading eo config file: E:\INSPRISE\AthenaSource\Projects\SAMPLES\Employee\WebContent\WEB-INF
EO config file unloaded successfully.
Loading eo config: E:\INSPRISE\AthenaSource\Projects\SAMPLES\Employee\WebContent\WEB-INF\eo-conf
Loading meta domain for EJBQL ...
Meta domain loaded successfully.
EO config file loaded successfully.
```

EO config file loaded successfully.

EJBQL Editor with Content Assist

Athena Console also offers EJBQL editor content assist which is very popular among Athena developers.



```
SELECT e FROM Employee e LEFT OUTER JOIN
e.dept d WHERE e.
```

Metadata Import and Export

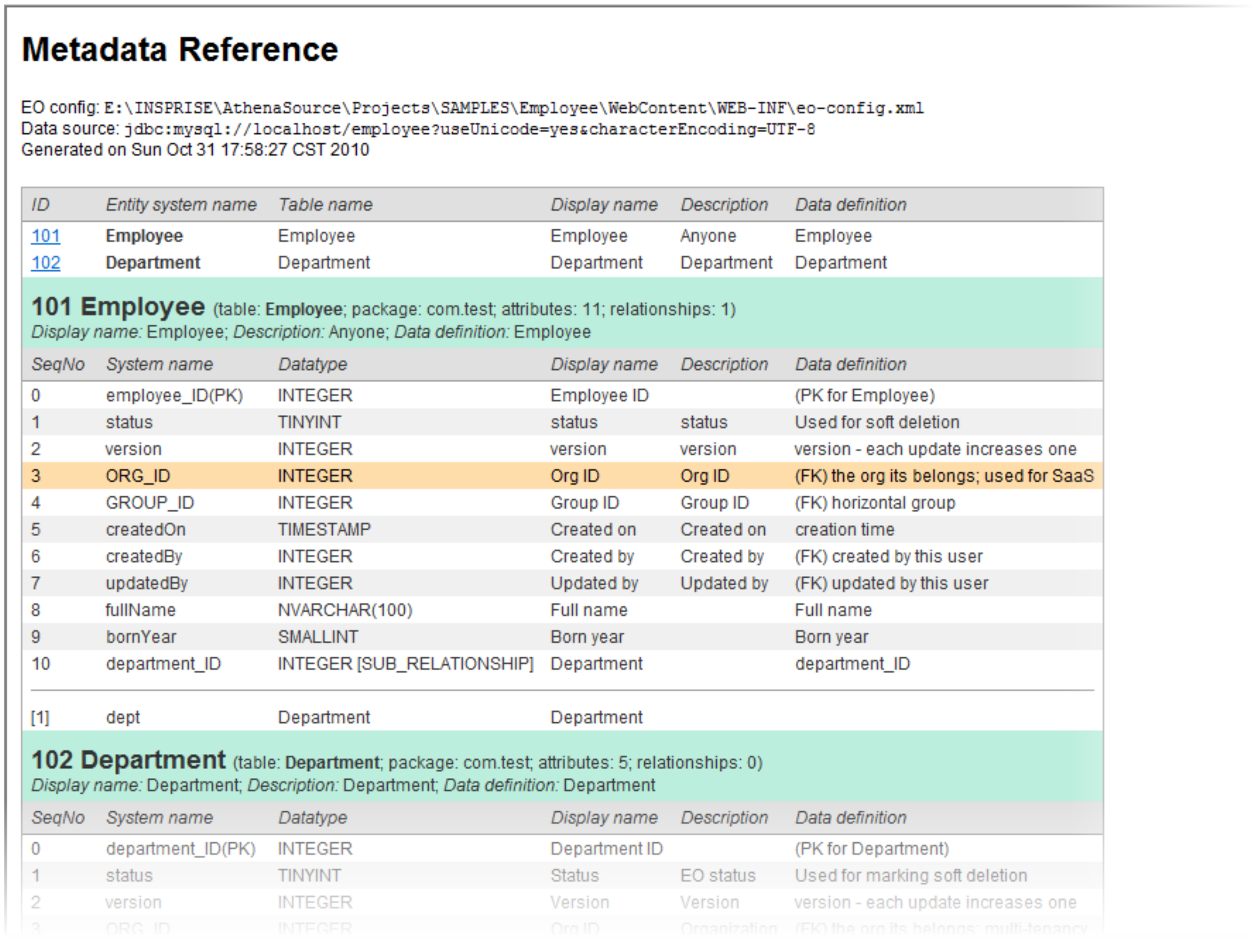
Under the 'Metadata Import/Export' tab, you can import or export metadata. It is useful when you need to reuse some entities from an application to new ones. When metadata is imported, data tables corresponding to imported entities will be constructed automatically.

Source Code Generation

Athena Console allows you to generate source code from metadata. If there is no change on an entity, previously generated source code will not be overwritten so there is no unnecessary commit to the revision control system.

Metadata Reference Document Generation

You can print a copy of metadata reference document for your own reference or to share with your team members. To do so, click the 'Generate' button under Misc. Utilities -> Generate Metadata Document.



Metadata Reference

EO config: `E:\INSPRISE\AthenaSource\Projects\SAMPLES\Employee\WebContent\WEB-INF\eo-config.xml`
Data source: `jdbc:mysql://localhost/employee?useUnicode=yes&characterEncoding=UTF-8`
Generated on Sun Oct 31 17:58:27 CST 2010

ID	*Entity system name*	*Table name*	*Display name*	*Description*	*Data definition*
101	**Employee**	Employee	Employee	Anyone	Employee
102	**Department**	Department	Department	Department	Department

101 Employee (table: **Employee**; package: com.test; attributes: 11; relationships: 1)
Display name: Employee; *Description:* Anyone; *Data definition:* Employee

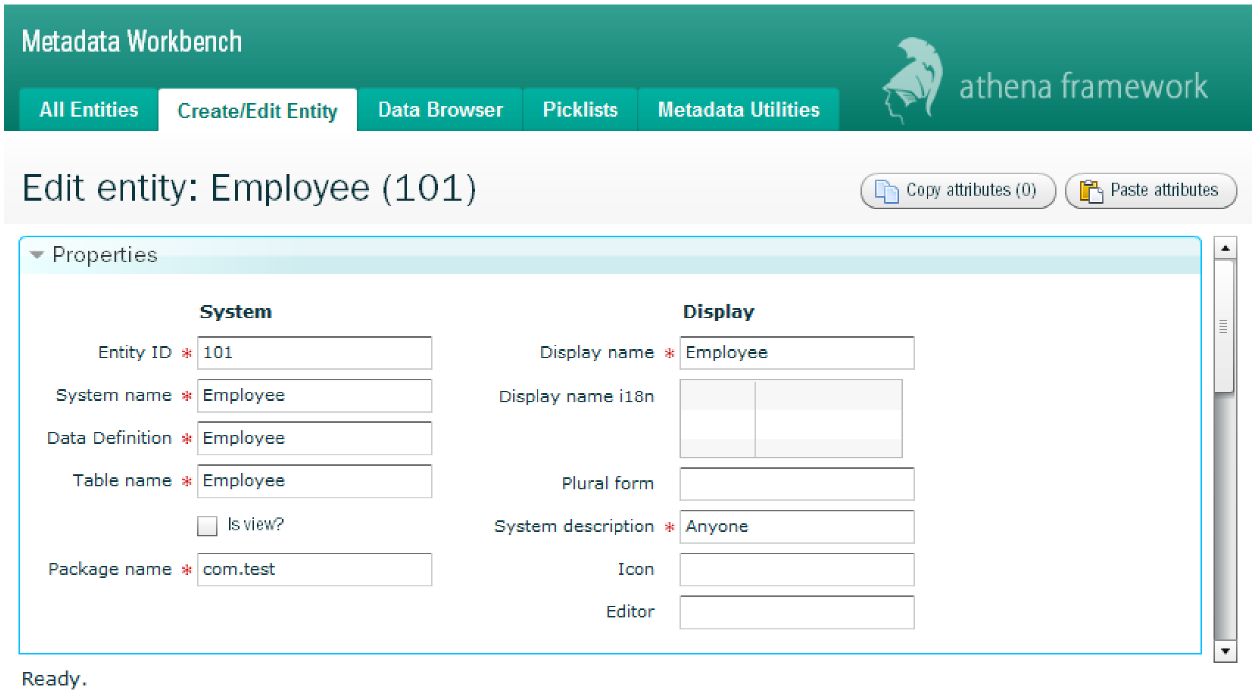
SeqNo	*System name*	*Datatype*	*Display name*	*Description*	*Data definition*
0	employee_ID(PK)	INTEGER	Employee ID		(PK for Employee)
1	status	TINYINT	status	status	Used for soft deletion
2	version	INTEGER	version	version	version - each update increases one
3	ORG_ID	INTEGER	Org ID	Org ID	(FK) the org its belongs; used for SaaS
4	GROUP_ID	INTEGER	Group ID	Group ID	(FK) horizontal group
5	createdOn	TIMESTAMP	Created on	Created on	creation time
6	createdBy	INTEGER	Created by	Created by	(FK) created by this user
7	updatedBy	INTEGER	Updated by	Updated by	(FK) updated by this user
8	fullName	NVARCHAR(100)	Full name		Full name
9	bornYear	SMALLINT	Born year		Born year
10	department_ID	INTEGER [SUB_RELATIONSHIP]	Department		department_ID
[1]	dept	Department	Department		

102 Department (table: **Department**; package: com.test; attributes: 5; relationships: 0)
Display name: Department; *Description:* Department; *Data definition:* Department

SeqNo	*System name*	*Datatype*	*Display name*	*Description*	*Data definition*
0	department_ID(PK)	INTEGER	Department ID		(PK for Department)
1	status	TINYINT	Status	EO status	Used for marking soft deletion
2	version	INTEGER	Version	Version	version - each update increases one
3	ORG_ID	INTEGER	Org ID	Organization	(FK) the org its belongs; multi-tenancy

Metadata Workbench

Once an application configuration files (usually named as 'eo-config.xml') is opened, you can launch Metadata Workbench from Athena Console's menu: Metadata Workbench -> Launch metadata workbench or press Ctrl + W.



Edit entity: Employee (101)

Properties

System		Display	
Entity ID *	101	Display name *	Employee
System name *	Employee	Display name i18n	
Data Definition *	Employee	Plural form	
Table name *	Employee	System description *	Anyone
Is view?	☐	Icon	
Package name *	com.test	Editor	

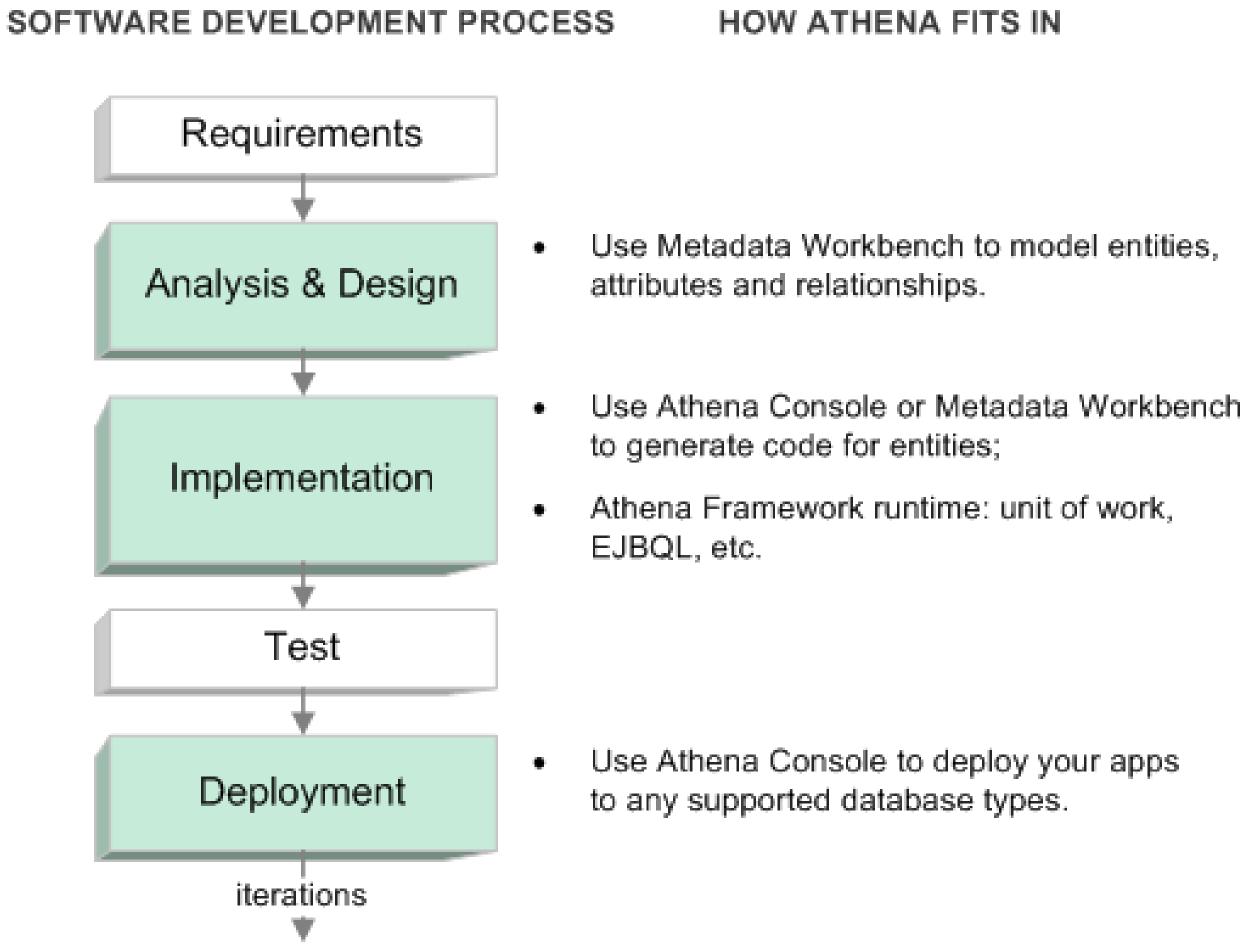
Ready.

Using Metadata Workbench, you can maintain entities, attributes and relationships systematically.

2. Get Started with Athena

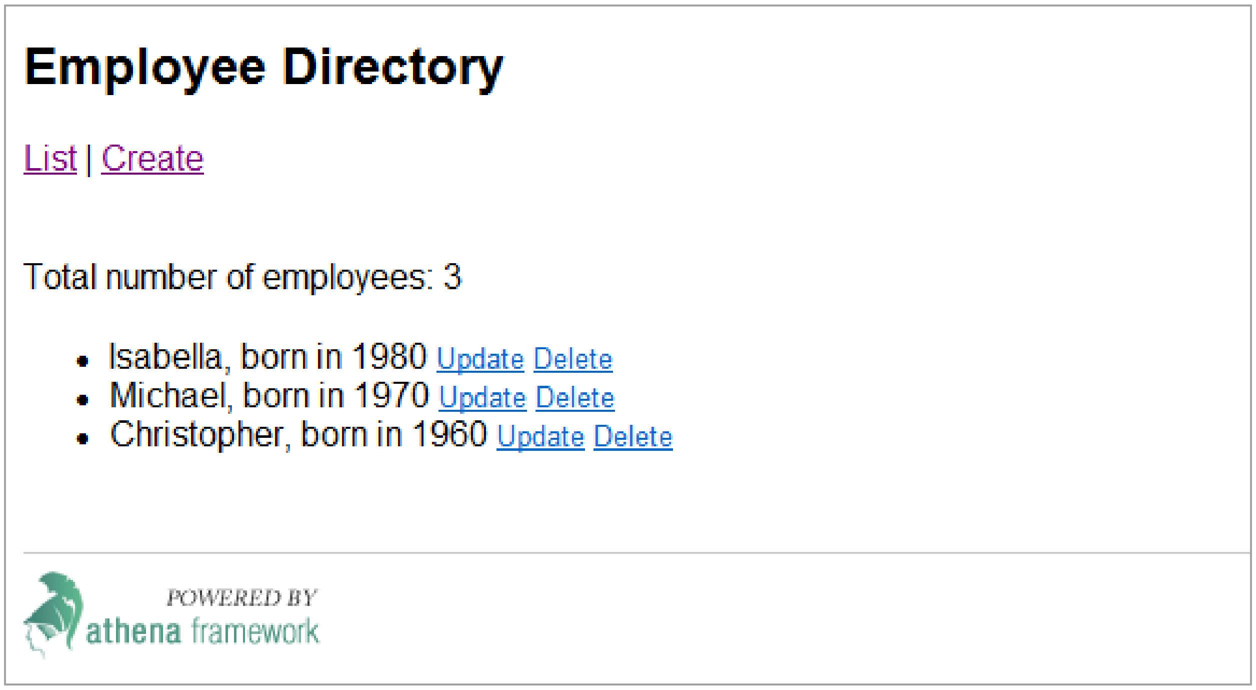
2.1 How Athena Fits in the Software Development Process

Athena Framework and its related tools Athena Console/Metadata Workbench can be used in many stages of software development lifecycle to enhance the development team's productivity. At the analysis and design stage of a typical software development process, the system analyst can use Metadata Workbench to model entities, attributes and relationships captured from the requirement. Developers can quickly jump to implement business logic with source code generated by Athena as a start point. The usual manual relational object mapping chore is eliminated completely. Finally, the software deployer can use Athena Console to deploy the software on any database types supported by Athena (Currently the following databases are supported: DB2, Derby, Oracle, MySQL). For example, developers may use Oracle while the deployer can deploy the software to DB2 simply by importing the metadata into a DB2 database.


SOFTWARE DEVELOPMENT PROCESS
HOW ATHENA FITS IN
Requirements
Analysis & Design
Implementation
Test
Deployment
iterations
• Use Metadata Workbench to model entities, attributes and relationships.
• Use Athena Console or Metadata Workbench to generate code for entities;
• Athena Framework runtime: unit of work, EJBQL, etc.
• Use Athena Console to deploy your apps to any supported database types.

2.2 Your First Athena Application

In this part, we will work through a tutorial to create your first Athena application. Basically, it is a simple CRUD application to record and to show employee information as showed in the screenshot below.



Employee Directory

List | Create

Total number of employees: 3

- Isabella, born in 1980 Update Delete
- Michael, born in 1970 Update Delete
- Christopher, born in 1960 Update Delete

Note

This tutorial comes with a video guide. Please watch it online at http://www.athenasource.org/java/basic-tutorial.php.

Project Setup

First, you need to create a project and add Athena dependency.

Eclipse

Create a new project File -> New -> **Dynamic Web Project**, project name: **EmployeeDir**. Assume PROJECT_ROOT is the root folder of the project:

1. Copy all the jar (jar files in root folder and `lib` folder) files from Athena Framework to `PROJECT_ROOT/WebContent/WEB-INF/lib`.

NetBeans

Create a new project File -> New Project -> Java Web -> **Web Application**, project name: **EmployeeDir**. Assume PROJECT_ROOT is the root folder of the project:

1. Copy all the jar files (jar files in root folder and `lib` folder) from Athena Framework to `PROJECT_ROOT/web/WEB-INF/lib` (create the 'lib' folder first);

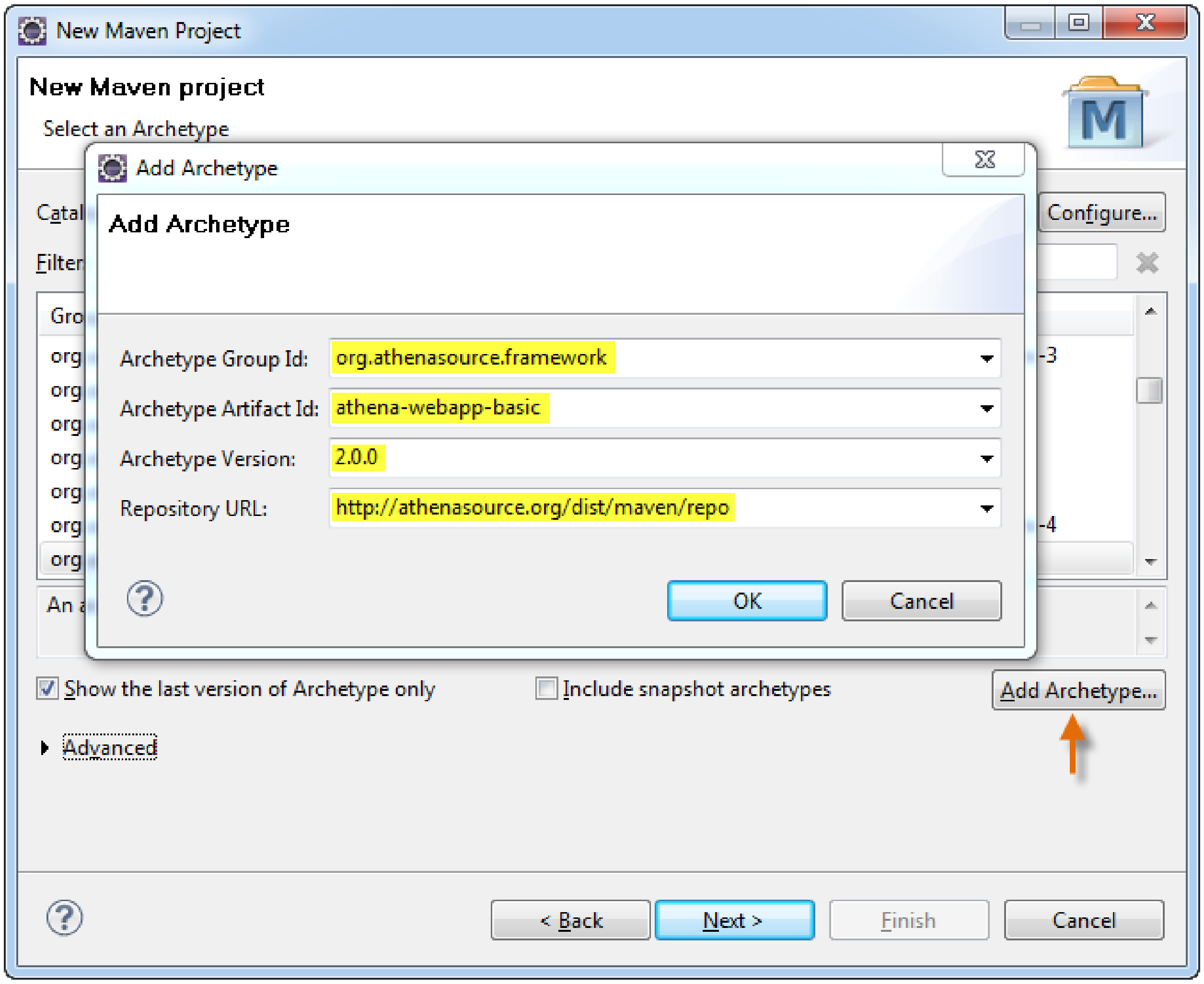
2. Right click on the project, and select 'Properties' to open the Project Properties dialog. Select 'Libraries' on the left panel, then click 'Add JAR/Folder' and browse to `PROJECT_ROOT/web/WEB-INF/lib` and select all the jar files then click 'Open'. Now, Athena Framework runtime has been successfully added to path. Click 'OK' to dismiss the dialog.

Maven

Maven is a build automation tool from Apache. To create a new Athena based project, you use the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.athenasource.framework »
 -DarchetypeArtifactId=athena-webapp-basic -DarchetypeVersion=2.0.0 »
 -DarchetypeRepository=http://athenasource.org/dist/maven/repo -DgroupId=com.test »
 -DartifactId=EmployeeDir -Dversion=1.0-SNAPSHOT
```

Alternatively, you may create the project using any IDE with maven support. For example, you may create the project as following in Eclipse with maven plugin:



Archetype Group Id: org.athenasource.framework
Archetype Artifact Id: athena-webapp-basic
Archetype Version: 2.0.0
Repository URL: http://athenasource.org/dist/maven/repo

Create eo-config.xml

While web.xml is the deployment descriptor file for a Java web application, eo-config.xml is the application configuration file for an Athena based Java web application. In eo-config.xml, you need to define the database source, Java source folder and other settings for the application. A typical eo-config.xml looks like as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<eo-system multitenancy="false" deletion-policy="hard">
  <datasources>
    <datasource>
      <database-type>MySQL</database-type>
      <host>localhost</host>
      <port>-1</port> <!-- '-1' means using the default port -->
      <username>root</username>
      <password>athena</password>
      <db>employeedir</db>
      <max-active>10</max-active>
      <max-idle>5</max-idle>
      <max-wait>5000</max-wait>
      <connection-timeout>300</connection-timeout>
    </datasource>
  </datasources>
  <property name="java-source-local-dir" value="D:\SAMPLES\EmployeeDir\src"/>
</eo-system>
```

You can copy the above code to the project's `WEB-INF` folder (the same folder as `web.xml`) and edit the following elements accordingly:

1. `database type`: valid values are `DB2, Derby, Oracle, MySQL`;
2. `username` and `password`: the credential pair to log into the database or empty string for embedded Derby;
3. `db`: the database name. For Derby, it is the relative or absolute path to a folder. The database does not need to exist now for Derby and MySQL; but must exist for DB2 and Oracle.
4. `java-source-local-dir`: set the `value` to the application project's source folder, usually `PROJECT_ROOT/src`;
5. Optionally, you may need to set `host` and `port`.

Elements `max-active`, `max-idle`, `max-wait` and `connection-timeout` are configurations for the Apache DBCP connection pool, you may leave them unchanged.

Note

If you choose to use DB2 or Oracle, you need to drop the jar file containing the corresponding JDBC driver to `WEB-INF/lib`. Athena Frameworks bundles JDBC drivers for Derby and MySQL, you may remove the jar files containining them if you do not plan to use.

Configure web.xml

At runtime, Athena Framework must load metadata (which is stored in the database) first before you can execute EJBQL. We can add `EOServletContextListener` as a servlet context listener:

```
<context-param>
  <param-name>eo.configuration.file</param-name>
  <param-value>webapp:WEB-INF/eo-config.xml</param-value>
</context-param>

<listener>
  <listener-class>org.athenasource.framework.eo.web.EOServletContextListener</listener-class>
</listener>
```

Note that the context listener needs a parameter named `eo.configuration.file` to retrieve the URL for the application's `eo-config.xml`. Athena extends normal URL protocols to support URL links for resources in the webapplication with prefix of `webapp:`. Alternatively, you may use `file:///PATH/eo-config.xml`, but it is less portable than `webapp:` version.

To enable any class to use Athena's EO service (enterprise object service, which is the core interface for Athena Framework. It will be covered in details later), we need to add `EOServletFilter` as a servlet filter:

```
<filter>
  <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
  <filter-class>org.athenasource.framework.eo.web.EOServletFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The complete `web.xml` is listed below:

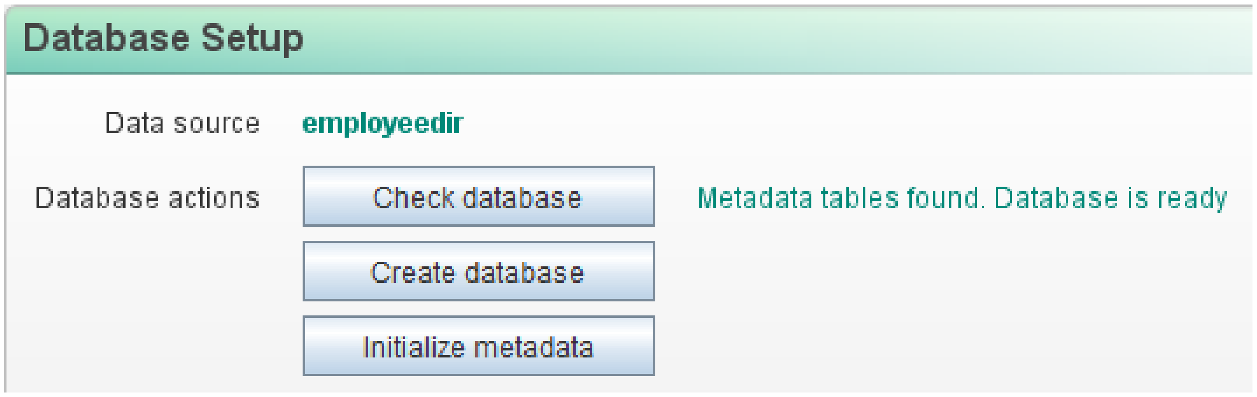
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" »
 xmlns="http://java.sun.com/xml/ns/javaee" »
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee »
     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>EmployeeDir</display-name>
  <welcome-file-list>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>

  <!-- Athena configurations -->
  <context-param>
    <param-name>eo.configuration.file</param-name>
    <param-value>webapp:WEB-INF/eo-config.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.athenasource.framework.eo.web.EOServletContextListener</listener-class>
  </listener>

  <filter>
    <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
    <filter-class>org.athenasource.framework.eo.web.EOServletFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

</web-app>
```

Initialize Metadata in the Database

Note

The GUI tool Athena Console is required. Please download and install it from http://www.athenasource.org/downloads.

As metadata is stored in database, you need to initialize metadata setup (create metadata tables and fill in initialization records) in the database first. Open Athena Console, and do the following under the Database Setup tab:



Metadata tables found. Database is ready

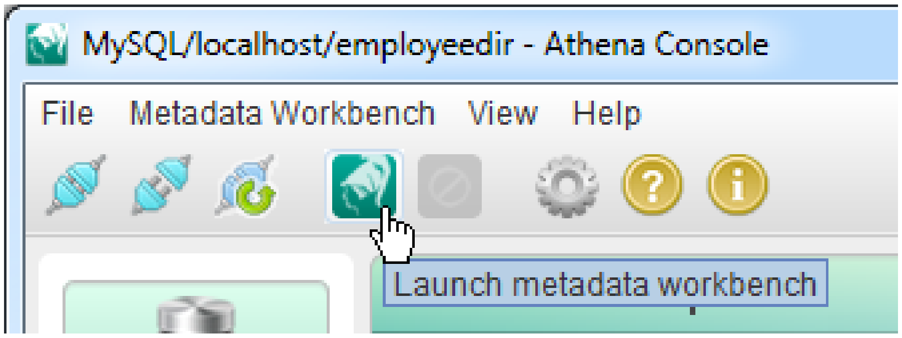
1. Click File -> Open EO Config or press 'Ctrl + O' and browse to the `eo-config.xml` file in the project;

2. Skip this step if you are using non-MySQL database. Click 'Create database' if the database does not exist;

3. Click the 'Initialize metadata' button to start the initialization process;

4. Click the 'Check database' button, and you should get 'Metadata tables found. Database is ready' message indicating metadata has been initialized successfully.

Once metadata has been initialized, you may proceed to create and update entities in the Metadata Workbench. To launch Metadata Workbench, you can simply click the toolbar button:


MySQL/localhost/employeedir - Athena Console
File
Metadata Workbench
View
Help
Launch metadata workbench

Create Entities

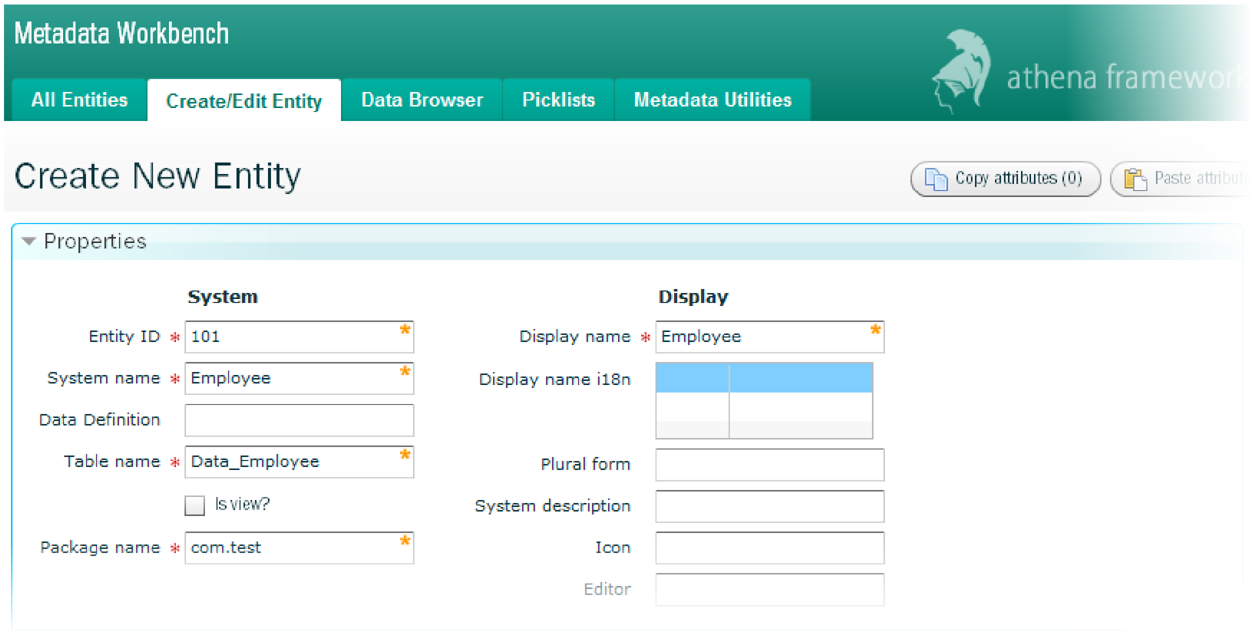
To create a new entity, you need to click on the 'New' button under the 'All Entities' tab.

Define Entity Properties

On the 'Create new Entity' page, enter the following values in the 'Properties' block:

- Entity ID: **101** (Entity IDs between 1 to 100 are reserved for system use only)

- System name: **Employee** (Entity's system name will be the class name too)
- Table name: **Data_Employee** (You may use any name permitted by the database)
- Package name: **com.test** (The package that the class belongs to; you may use any valid package name)
- Display name: **Employee** (A brief description of the entity)



Create New Entity

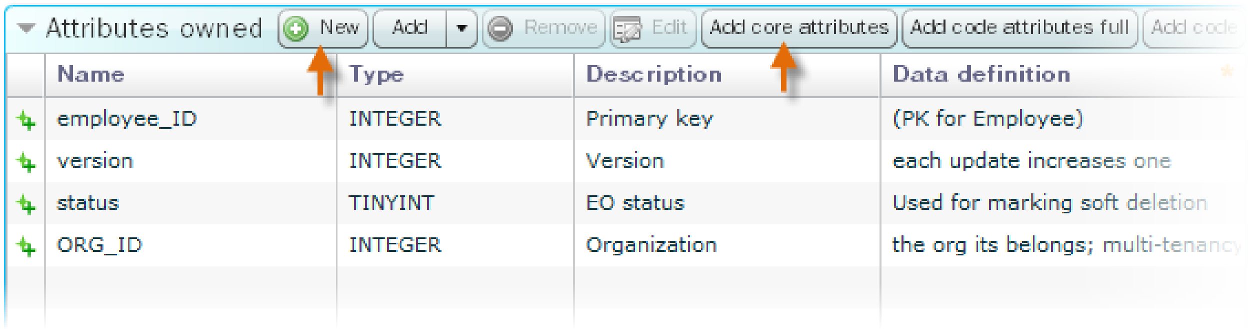
Properties

System		Display	
Entity ID *	101	Display name *	Employee
System name *	Employee	Display name i18n	
Data Definition			
Table name *	Data_Employee	Plural form	
	☐ Is view?	System description	
Package name *	com.test	Icon	
		Editor	

To add attributes to the entity, you can either press 'New' button to create new attributes or use the 'Add ...' quick add tools to add common attributes in the 'Attributes owned' block.

Add Core Attributes

Core attributes are required by the Athena Framework. Before adding normal attributes, you should press the 'Add core attributes' to add four core attributes for the entity:



Name	Type	Description	Data definition
employee_ID	INTEGER	Primary key	(PK for Employee)
version	INTEGER	Version	each update increases one
status	TINYINT	EO status	Used for marking soft deletion
ORG_ID	INTEGER	Organization	the org its belongs; multi-tenanc

Core attributes:

- **x_ID** - Required. The first attribute must be an integer based primary key with auto increment.
- **version** - Required. The revision number is used by the unit of work to update obsoleted data for EOs.
- **status** - Recommended. If you plan to use soft deletion, this attribute must be present.
- **ORG_ID** - Recommended. If you plan to use multi-tenancy, this attribute must be present.

Primary key: In Athena, each entity has an integer based primary key and the primary key is always the first attribute. The primary key is auto increment so that you do not have to specify an implicit value all the time. Such kind of primary key is called a surrogate key as opposite to a natural key.

Note

If you do not plan to use soft deletion or multi-tenancy, you may select the `status` attribute or the `ORG_ID` attribute and press the 'Remove' button.

Add More Attributes

We can now add normal attributes to the `Employee` entity. First, we need to add an attribute to store the name of an employee. Press the 'New' button, and input the following values on the create new attribute page:

- Attribute ID: (please keep unchanged, handled by the workbench automatically)
- Belonging entity ID: (please keep unchanged, handled by the workbench automatically)
- System name: **nameFull**
- Display name: **Full name**
- Column type: **VARCHAR**
- Precision: **200**

Click 'Save' to return to the create entity page, then click 'New' under the attributes owned block to add another attribute to store born year of an employee:

- System name: **bornYear**
- Display name: **Born year**
- Column type: **INTEGER**

- Precision: **-1**
- Minimum value: **1900** (default value -13 means minimum is not applicable)
- Maximum value: **2000** (default value -13 means maximum is not applicable)

Click 'Save' on the new entity page and our first entity has been created.

To add new attributes or edit existing attributes for an entity, you can simply double click the entity under the 'All entities' tab and repeat the process.

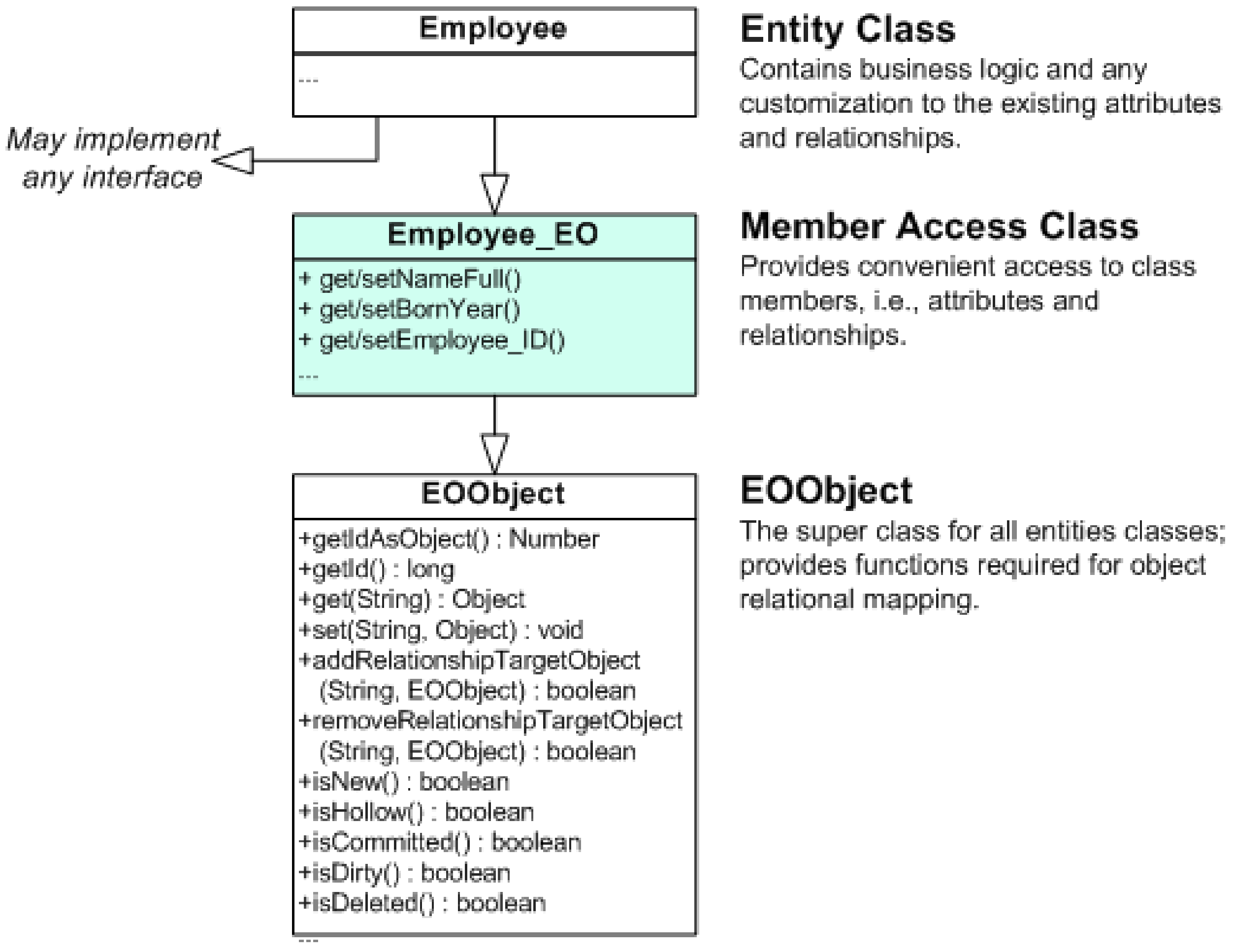
Generate Source Code

Traditionally, you need to code the data transfer object classes and/or data access object classes to make ORM work. Athena has changed all that. Instead of manually coding and duplicating database schema information, you can simply generate classes for each entity using Athena Console.

To generate classes, select the 'Code Generation' tab in Athena Console and click the 'Generate classes' button. Once the code generation is done, refresh the project folder in your IDE, and you'll find the following two classes are generated in the project's source folder:

- `com.test.Employee` (Entity class)
- `com.test.generated.Employee_EO` (Member access class)

We refer the `Employee` class as the entity class. You should put business logic code into it. Entity classes are only generated if they do not exist. The `Employee_EO` class provides convenient access methods to entity members - attributes and relationships. You can access those methods in `Employee` since it extends `Employee_EO`. `Employee_EO` is always re-generated in case of any change of entity's attributes or relationships. The `EOObject` class is the super class for all entity classes, which provides functions required for object relational mapping. The relationships of the classes can be illustrated in below class diagram:


Employee

May implement
any interface
Employee_EO
+ get/setNameFull()
+ get/setBornYear()
+ get/setEmployee_ID()

EOObject
+getIdAsObject() : Number
+getId() : long
+get(String) : Object
+set(String, Object) : void
+addRelationshipTargetObject
(String, EOObject) : boolean
+removeRelationshipTargetObject
(String, EOObject) : boolean
+isNew() : boolean
+isHollow() : boolean
+isCommitted() : boolean
+isDirty() : boolean
+isDeleted() : boolean

Entity Class
Contains business logic and any customization to the existing attributes and relationships.
Member Access Class
Provides convenient access to class members, i.e., attributes and relationships.
EOObject
The super class for all entities classes; provides functions required for object relational mapping.

Once you have the entity classes generated, you can start to implement the application.

Implement CRUD Opearations

In this application, we'll put all the code into a single JSP file. Create a file named `main.jsp` under the web content folder in your project and copy the code below then save it. You may run the application in any servlet container now.

```
<%@page import="java.io.PrintWriter"%>
<%@page import="org.athenasource.framework.eo.core.UnitOfWork"%>
<%@page import="com.test.Employee"%>
<%@page import="java.util.List"%>
<%@page import="org.athenasource.framework.eo.query.EJBQLSelect"%>
<%@page import="org.athenasource.framework.eo.core.EOService"%>
<%@page import="org.athenasource.framework.eo.core.context.EOThreadLocal"%>
<%@page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@page import="org.athenasource.framework.eo.core.EOObject"%>
<%@page import="com.test.generated.Employee_EO"%><html>
<body style="font-family: arial; ">   <h2>Employee Directory</h2>

<p><a href="?action=LIST">List</a> | <a href="?action=FORM_CREATE">Create</a> »
 </p><br>
<%
String action = request.getParameter("action");
if(action == null || action.trim().length() == 0) { // if no action specified, use 'LIST'.
  action = "LIST";
}
```

```
if("LIST".equalsIgnoreCase(action)) { // List all employees
  EOService eoService = EOThreadLocal.getEOService();
  EJBQLSelect selectEmployees = eoService.createEOContext().createSelectQuery("SELECT e FROM Employee »
     e");
  List<Object> employees = selectEmployees.getResultList();

  out.println("<p>Total number of employees: " + employees.size() + "</p>");
  out.println("<ul>");
  for(int i=0; i < employees.size(); i++) {
    Employee employee = (Employee)employees.get(i);
    out.println("<li>" + employee.getNameFull() + ", born in " + employee.getBornYear());
    out.println(" <a href='?action=FORM_UPDATE&empid=" + employee.getEmployee_ID() + "'><font »
         size=-1>Update</font></a>");
    out.println(" <a href='?action=DELETE&empid=" + employee.getEmployee_ID() + "' »
         onClick=\"return confirm('Are you sure to delete this employee?')\"><font »
         size=-1>Delete</font></a>");
  }
  out.println("</ul>");

}else if("CREATE".equalsIgnoreCase(action)) { // Create a new one
  UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
  Employee newEmp = (Employee)uow.createNewInstance(Employee.SYSTEM_NAME);
  uow.persist(newEmp);
  try {
    newEmp.setNameFull(request.getParameter("fullname"));
    newEmp.setBornYear(Integer.parseInt(request.getParameter("bornyear")));
    uow.flush();
    uow.close();
    out.println("Employee created successfully. ID: " + newEmp.getEmployee_ID());
  }catch(Throwable t) {
    out.println("Failed to create employee due to exception: <pre>");
    t.printStackTrace(new PrintWriter(out));
    out.println("</pre>");
  }

}else if("UPDATE".equalsIgnoreCase(action)) { // Update an existing one
  UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
  EJBQLSelect selectEmp = uow.createSelectQuery("SELECT e FROM Employee e WHERE e.employee_ID = ?1");
  selectEmp.setParameter(1, Integer.parseInt(request.getParameter("empid")));
  Employee emp = (Employee)selectEmp.getSingleResult();
  if(emp == null) {
    out.println("Employee not found, id: " + request.getParameter("empid"));
  }else{
    try {
      emp.setNameFull(request.getParameter("fullname"));
      emp.setBornYear(Integer.parseInt(request.getParameter("bornyear")));
      uow.flush();
      uow.close();
      out.println("Employee data updated successfully, id: " + emp.getEmployee_ID());
    }catch(Throwable t) {
      out.println("Failed to create employee due to exception: <pre>");
      t.printStackTrace(new PrintWriter(out));
      out.println("</pre>");
    }
  }

}else if("DELETE".equalsIgnoreCase(action)) { // Update an existing one
  UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
  Employee emp = (Employee)uow.find(Employee_EO.SYSTEM_NAME, »
     Integer.parseInt(request.getParameter("empid")), null, null);
  if(emp == null) {
    out.println("Employee not found, id: " + request.getParameter("empid"));
  }else{
    try {
      uow.remove(emp);
      uow.flush();
      uow.close();
      out.println("Employee data deleted successfully, id: " + emp.getEmployee_ID());
    }catch(Throwable t) {
      out.println("Failed to delete employee due to exception: <pre>");
      t.printStackTrace(new PrintWriter(out));
      out.println("</pre>");
    }
  }

}else if("FORM_CREATE".equalsIgnoreCase(action)) { // display form for create
%>
<form action="">
<input type="hidden" name="action" value="CREATE" />
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_nameFull).getDisplayName() %>:
<input name="fullname" type="text" size="20" />
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_bornYear).getDisplayName() %>:
<input name="bornyear" type="text" size="4" />
<input type="submit" value="Create">
</form>
<%
} else if("FORM_UPDATE".equalsIgnoreCase(action)) { // display form for update
```

```
  Employee emp = null;
  EJBQLSelect selectEmp = EOThreadLocal.getEOService().createEOContext().createSelectQuery("SELECT e »
     FROM Employee e WHERE e.employee_ID = ?1");
  selectEmp.setParameter(1, Integer.parseInt(request.getParameter("empid")));
  emp = (Employee)selectEmp.getSingleResult();
%>
<form action="">
<input type="hidden" name="action" value="UPDATE" />
<input type="hidden" name="empid" value="<%= request.getParameter("empid") %>" />
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_nameFull).getDisplayName() %>:
<input name="fullname" type="text" size="20" value="<%= emp.getNameFull() %>"/>
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_bornYear).getDisplayName() %>:
<input name="bornyear" type="text" size="4" value="<%= emp.getBornYear() %>"/>
<input type="submit" value="Update">
</form>
<%
}else if(action == null || action.trim().length() == 0) {
  out.println("Welcome.");
}else{
  out.println("Unsupported action: " + action);
}
%>
<p align="left" style="padding-top: 20px;">
<hr style="width: 100%; color: #ccc; height: 1px;"/>
<a href="http://www.athenaframework.org" target='_blank'>
<img src="http://www.athenaframework.org/_img/logo/logo-poweredby.png" align="left" hspace="0" »
 vspace="1" border="0">
</a></p>
</body>
</html>
```

Above is the complete code list of `main.jsp`. If you have used other ORM frameworks before, you may find it familiar. Below briefly explains code snippets in `main.jsp` and detailed concepts will be elaborated in later chapters.

Use EOThreadLocal.getEOService() to Obtain EOService

`EOService` is the core interface to Athena Framework. To obtain it, you use the following statement:

```
EOService eoService = EOThreadLocal.getEOService();
```

Execute EJBQL and Get Results

If you need to load objects for read only, you can use `EOContext`. If you need to load objects and later modify them, you use `UnitOfWork`. Both `EOContext` and `UnitOfWork` represent an object repository that ensures no two objects mapping to the same database record row. They are equivalent to Java Persistence API's EntityManager or Red Hat JBoss Hibernate's Session.

To execute EJBQL to get a list of objects:

```
EOContext eoContext = eoService.createEOContext();
EJBQLSelect selectEmployees = eoContext.createSelectQuery("SELECT e FROM Employee e");
List<Object> employees = selectEmployees.getResultList();
```

To execute EJBQL to get a single object:

```
EJBQLSelect selectEmp = eoContext.createSelectQuery("SELECT e FROM Employee e WHERE e.employee_ID = »
 ?1");
selectEmp.setParameter(1, 1);
Employee emp = (Employee)selectEmp.getSingleResult();
```

Update Objects Through UnitOfWork

The `UnitOfWork` tracks every changes made to objects maintained in it. You may load objects through UnitOfWork and modify them then save the changes in a transaction using `UnitOfWork.flush()`.

```
UnitOfWork uow = eoService.createUnitOfWork();
EJBQLSelect selectEmp = uow.createSelectQuery("SELECT e FROM Employee e WHERE e.employee_ID = ?1");
selectEmp.setParameter(1, Integer.parseInt(request.getParameter("empid")));
Employee emp = (Employee)selectEmp.getSingleResult();
emp.setNameFull("New Name"));
uow.flush();
```

To delete an object:

```
UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
Employee emp = (Employee)uow.find(Employee_EO.SYSTEM_NAME, »
 Integer.parseInt(request.getParameter("empid")), null, null);
uow.remove(emp);
uow.flush();
```

Access Metadata Information at Runtime

In `main.jsp`, we use the following code to retrieve the display name of the `nameFull` attribute:

```
EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_nameFull).getDisplayName()
```

For large applications with complicated user interfaces, metadata information can be used to ensure the consistence of the presentation. To obtain the `Entity` object using its system name, you use:

```
Entity entity = eoService.getEntity(Employee_EO.SYSTEM_NAME);
List<EOObject> attributes = entity.getAttributesOwned();  // all the attributes owned by the entity
List<EOObject> relationship = entity.getRelationshipOwned(); // all the relationship owned
```

This chapter scratched the surface of Athena Framework by walking through the development of the sample application. The next part will introduce features of Athena Framework in details.

3. Architecture and Concepts

3.1 Architecture

As a mature object relational mapping and persistence framework, Athena provides an elegant solution to object-relational impedance mismatch that enables developers to work with high level objects instead of manipulating SQL statements. Figure 3.1, “Architecture of Athena Framework” illustrates core components in the Athena Framework.

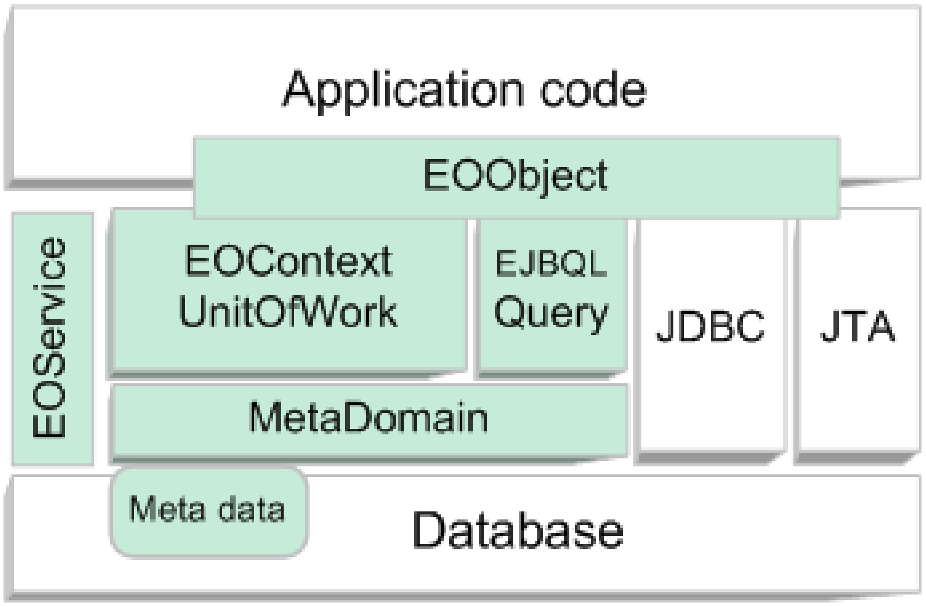


Figure 3.1. Architecture of Athena Framework

Overview of each components in the diagram:

EOService (`org.athenasource.framework.eo.core.EOService`)
: Core interface to access features of the Athena Framework. An EOService connects to only one database instance. In most cases, you use only one EOService. However, you are free to construct and use many EOServices at the same time to access different database instances.

MetaDomain (`org.athenasource.framework.eo.core.MetaDomain`
: Represents all the metadata defined, including entities, attributes and relationships. MetaDomain is loaded by EOService. At runtime, you may query MetaDomain to obtain meta information.

EOObject (`org.athenasource.framework.eo.core.EOObject`)
: The super class for all generated entity classes. EOObject represents an instance of an entity. Each EOObject is mapped to a single row in the table for the entity in the database.

EOContext (`org.athenasource.framework.eo.core.EOContext`)
: EOContext represent a unit of work for manipulating a set of entity enterprise objects. It guarantees that there is maximum one EOObject corresponding to any database record row in the context. EJQBQL querying must be carried in a particular EOContext to avoid duplicating EOObjects for the same database record row. EOContext is for querying only as it can not persist EOObjects.

UnitOfWork (`org.athenasource.framework.eo.core.UnitOfWork`)
: Extends EOContext by allowing persistence.

EJBQLSelect (`org.athenasource.framework.eo.query.EJBQLSelect`)
: Athena implementation's query similar to JEE's Enterprise Java Beans Query Languages. A EJBQLSelect performs in a context and returns EOObjects or primitive values as results.

4. Entity Relationship Modeling

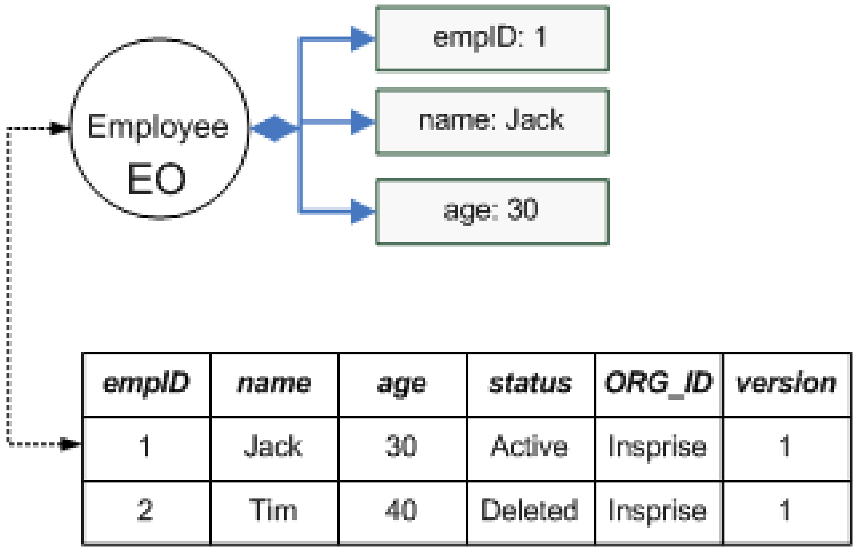
4.1 Core Concepts

Enterprise Objects

Enterprise object (EO) plays the key role in the Athena Framework. An EO is a managed object by Athena Framework. Features of enterprise objects over normal objects include:

- EO’s are manageable elements in the Athena Framework –Athena performs object relational mapping (ORM) to persist EO’s.
- EO’s are accountable – Athena automatically updates various field for an EO if certain attributes are available. For example, `version` to record revision number and `ORG_ID` to record organization that the EO belonging to for multi-tenancy applications.
- EO’s can be soft deleted – If the deletion policy in an application configuration is set to soft, the EO will be soft deleted when the user choose to delete an EO. Athena marks the EO's `status` field as deleted state. This feature allows recovery from accidental deletion and enables auditing.

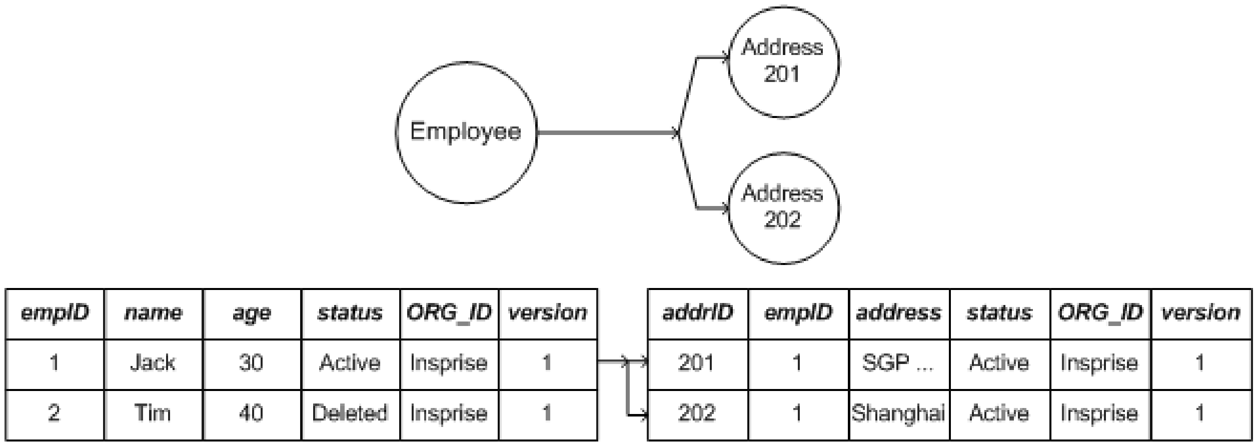
Figure 4.1, “Mapping of EOObjects to Database Records” illustrates the mapping between enterprise objects and database records.



empID	*name*	*age*	*status*	*ORG_ID*	*version*
1	Jack	30	Active	Insprise	1
2	Tim	40	Deleted	Insprise	1

Figure 4.1. Mapping of EOObjects to Database Records

Athena does not only map individual enterprise objects to database records, but it also can map complex networks of enterprise objects through relationships as illustrated in Figure 4.2, “Mapping of Networks of EOObjects to Database Records”



empID	*name*	*age*	*status*	*ORG_ID*	*version*
1	Jack	30	Active	Insprise	1
2	Tim	40	Deleted	Insprise	1

addrID	*empID*	*address*	*status*	*ORG_ID*	*version*
201	1	SGP ...	Active	Insprise	1
202	1	Shanghai	Active	Insprise	1

Figure 4.2. Mapping of Networks of EOObjects to Database Records

How does Athena know to map EOObjects to database records? It uses metadata that is comprised of entities, attributes and relationships.

If you are familiar with other ORM frameworks, you may skip the following part.

Entities and Attributes

We refer to a certain class of enterprise objects as an **Entity**. For example, in the example given in last section, we have two entities – Employee and Address. In Java, an Entity is a class. In database domain, an Entity is represented by a table. An **Attribute** is a property owned by an Entity. For example, the Employee entity has an attribute named “age”. You can think an attribute as a Java property in the class or a column in the database table.

Relationships

A **Relationship** represents a connection or association between two entities. For example, for the association between the Employee entity and the Address entity, there are two relationships: relationship 1 - an employee may own many addresses (one-to-many); relationship 2: an address belongs to one employee (many-to-one). Such an association is called **bidirectional**.

For a pair of relationships for the same association, we call the relationship arising from the entity that stores the relationship information (i.e., has a foreign key to the target entity) as the **owning** relationship and the other one as the **inverse** relationship. Java represents relationships using object references, and database represents relationships using foreign keys.

We refer Entity, Attribute and Relationship collectively as the metadata.

Owning and Reverse Relationships

When you define relationships, you must take extra care on whether they are owning or reverse. If relationships are not set properly, persistence may not work.

For example, in the diagram below, we have two relationships:

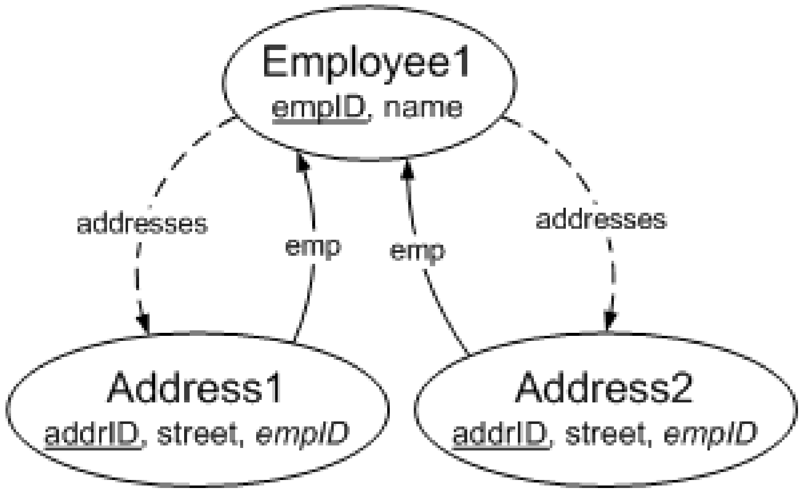


Figure 4.3. Employee-Address Relationship Graph

- **emp** – many-to-one; source entity: Address; target entity: Employee – this is the **owning** side since the source entity Address contains a FK (e.g., a field named empID) to store the information for this relationship.

- **addresses** - one-to-many; source entity: Employee; target entity: Address – this is the **inverse** side.

Both owning and inverse relationships are used by the `UnitOfWork` to persist by reachability - i.e., when you call `uow.persist(o)`, all other objects connected directly or indirectly through owning or inverse relationships are persisted too. However, only the owning relationships determine the persistence order. You get the dependency graph by keeping the owning relationships and eliminating the inverse relationships (see Figure 4.4, “Employee-Address Dependency Graph”).

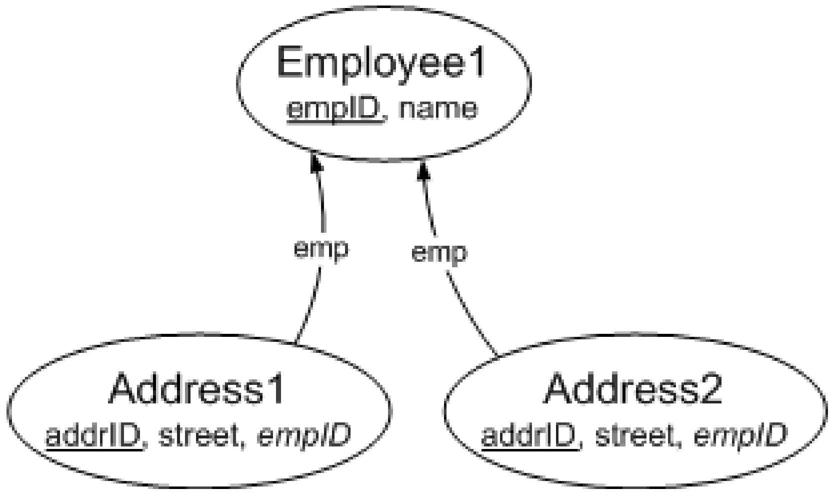


Figure 4.4. Employee-Address Dependency Graph

When UnitOfWork processes the dependency graph, it starts persisting those objects that do not depend on others. For example, in this case, Employee1 will get persisted before Address1 and Address2. From the database perspective, Employee1 must be INSERTed first since both Address1 and Address2 need the primary key of Employee1.

What if you set the relationships incorrectly – emp as the inverse and addresses as the owning? In that case, the two arrows in the dependency graph will be reversed and as a result, Address1 and Address2 are persisted before Employee1. In such case, the foreign key field for both Address1 and Address2 will contain invalid value (-1) since at the time of INSERTing Address1 & 2, the primary key value for Employee1 is unknown.

To summary, **for any relationship, the owning side is always points to one and the inverse side may point to one or many. In any case, the source entity for an owning relationship contains a foreign key to the target entity of the owning relationship.**

This section has provided theory of entity relationships. Next section will show you detailed modeling procedure with Metadata Workbench.

4.2 Modeling with Metadata Workbench

To launch Metadata Workbench, you should start Athena Console, open the `eo-config.xml` and click the Metadata Workbench button.

Define Entities

To create a new entity, press the 'New' button under the 'All entities' tab, and you'll be brought to the entity editing page. To edit an entity, simply double click on the entity in the entity list.

On the Properites block, you need to specify values for the following properties:

Table 4.1. Entity Properties

Property	***Constraint***	***Value***	***Description***
Entity ID	Required	Integer value larger than 100	IDs between 1 to 100 are reserved for Athena system use.
System name	Required	Valid class name	System name will be used as the class name during code generation
Data definition	Optional	String	Documents data source
Table name	Required	Valid table name	You may use the same value as system name or use a different one
Package name	Required	Valid package name	The package that entity class belongs to, e.g, `com.test`
Display name	Required	String	Descriptive name of the entity; you may use it on the user interface of your applications.
Display name i18n	Optional	String	Optional internationalized descriptive names that may be used on the UI of your applications.
Plural form, icon, editor	Optional	String	Optional. You may access these values at runtime and display them using your own logic.
System description	Recommended	String	Long description in additon to display name. Usually you can use display name as label and description as the tooltip on UI.

Once you finished specifying properties of the entity, you can then proceed to define attributes and relationships. There are two ways to define attributes.

Quick-Add Common Attributes

You may use the quick-add buttons as shown in Figure 4.5, “Quick-Add Buttons” to add common used attributes for the entity.

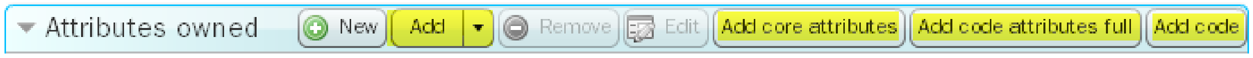


Figure 4.5. Quick-Add Buttons

The following common attributes can be added through quick-add buttons:

Table 4.2. Common Attributes

System name	Constraint	Description	Add core?	Add core full?	add Code?
Entity_ID	Rquired	Primary key	Yes	Yes	No
version	Required	Used to record revision number	Yes	Yes	No
status	Recommended	Used for soft deletion	Yes	Yes	No
ORG_ID	Recommended	Used for multi-tenancy	Yes	Yes	No
createdOn	Optional	Creation time	No	Yes	No
createdBy	Optional	User id of the creator	No	Yes	No
updatedOn	Optional	Last updated on	No	Yes	No
updatedBy	Optional	id of the last update user	No	Yes	No
entityCode	Optional	Natural key if any	No	No	Yes

Note that `createdOn`, `createdBy`, `updatetdOn`, and `updatedBy` will be automatically set by the framework if they are present. If you want to add one by one, you can use the "Add" drop down list.

In most cases, you can just *add core attributes* and proceed to define custom attributes.

Define Attributes

Quick-add can help you to add common attributes; however for the rest of the attributes, you need to define them one by one. To create a new attribute, click *New* button on the attributes block, and the screen will transit to the attribute editing form.

Properties of the attribute are categorized into three groups:

- **System** - core properties;
- **Column Definition** - properties related to the definition of the corresponding column in the database;
- **UI and Validation** - user interface and validation related properties

List of properties to define the attribute:

Table 4.3. Attribute Properties

Property	***Constraint***	***Value***	***Descrption***
System			
Attribute ID, Belonging entity ID	<do not change>	Keep these values unchanged	Metadata Workbench will handle them.
System name	Required	Valid Java field name	System name will be used as the Java variable name as well as column name in the database. The workbench validates the input against existing attribute names and SQL reserved words.
Data definition	Optional	String	Documents data source
Display name	Required	String	Descriptive name of the entity; you may use it on the user interface of your applications.
Display name i18n	Optional	String	Optional

Property	*Constraint*	*Value*	*Descrption*
			internationalized descriptive names that may be used on the UI of your applications.
Plural form, icon, editor	Optional	String	Optional. You may access these values at runtime and display them using your own logic.
System description	Recommended	String	Long description in additon to display name. Usually you can use display name as label and description as the tooltip on UI.
Element type	Optional	Integer	The type of element. For your own reference only.
Default value	Optional	String	The default value to be returned in case database returns `null`. Athena automatically cast the string value to a proper type.
Lazy load, Searchable, Sortable, Totalable, Translatable, Is natural key?	Optional	Boolean	Various flags. For your own reference only.
Column Definition			
Column type	Required	Integer	Column data type
Sub type	Optional	Integer	Optional indicator for user interface rendering and validation
Primary key	-	Boolean	Only the first attribute can be the primary key. Uncheck it for all attributes other than the first attribute.

Property	***Constraint***	***Value***	***Descrption***
Nullable	Optional	Boolean	Whether the column allows `null` values
Precision	-	Integer	The length for column type CHAR, NCHAR, VARCHAR, and NVARCHAR; enter `-1` for other types
Scale	-	Integer	The scale for column type DECIMAL; enter `-1` for other types
Auto-increment	Optional	Boolean	On some databases, only the primary key can be auto-incremented.
Auto-increment from, Increment step	-	Integer	Usually, you should keep these values `1, 1` unchanged.
Default column exp.	Optional	String	Default column expression to be used in column definition
Indexed column	Optional	Boolean	Whether the column should be index.
Unique column	Optional	Boolean	Whether column values must be unique.
UI and Validation			
Hidden	Optional	Boolean	Indicator for the UI side that it should not show this attribute
Mandatory	Optional	Boolean	Whether this attribute always needs an non-empty value
Display format	Optional	String	Display format string that can be used to display on the UI
Editor	Optional	String	Editor information for this attribute
Minimum value, Maximum value	Optional	Double	Min./max values for numeric types. `-13`

Property	*Constraint*	*Value*	*Descrption*
			means ignore the setting.
Regex	Optional	String	Regular expression used to valid string value

Copy Attributes

You can copy one or more attributes from an entity to another one. To do so, select attributes to be copied in the source entity, and click the "Copy attributes" button on the top of the page as illustrated in Figure 4.6, “Copy Attributes”. Click the "Past attributes" button to paste the attributes into the editor for the target entity then save it.

Figure 4.6. Copy Attributes

Define Relationships

the section called “Relationships” describes that relationships define which entities are associated with which other entities. In a relationship, the maximum number of instances of an entity that can be associated with an instance of another entity is referred as **cardinality**. There are three basic cardinality types:

One to One

One instance of entity A can be associated with at most one instance of entity B, and vice verse.

Assume that an employee can only have one corporate email account, we have an one-to-one relationship between `Employee` and `EmailAcct` as illustrated in Figure 4.7, “Employee-EmailAccount Relationship”.

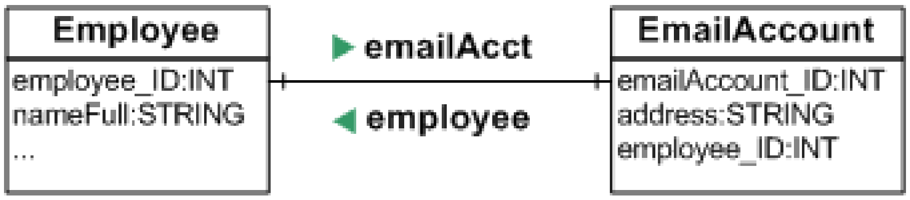
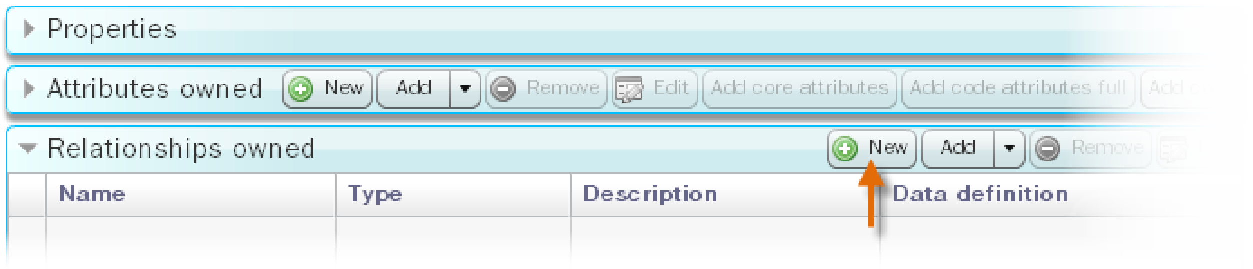


Figure 4.7. Employee-EmailAccount Relationship

To create a relationship, you click the 'New' button on the relationship block of the entity editor:



First, create the Employee and EmailAccount entities if they do not exist yet. Then create a relationship named *emailAcct* on Employee and a relationship named *employee* on EmailAccount:

Table 4.4. Relationships: Employee.emailAcct and EmailAccount.employee

Property	**Employee.emailAcct**	**EmailAccount.employee**
System name	"emailAcct"	"employee"
Display name	"Email account"	"Employee"
Relationship type	ONE_TO_ONE	ONE_TO_ONE
Cascade	ALL	PERSIST
Inverse	true	false
Source entity	Employee	EmailAccount
Source attribute	employee_ID	employee_ID
Target entity	EmailAccount	Employee
Target attribute	employee_ID	employee_ID

Cascade defines the persistence/deletion behavior of the target instance when the source instance gets deleted which is explained in the section called “Cascade Rules”. If you are unsure which cascade type to choose, you can safely use PERSIST.

Programming One to One Relationships

The following code illustrates how to create instances of entities and to form a one-to-one relationship:

```
public class TestRelationships {

  EOService eoService;

  public TestRelationships() throws IOException { // constructor
    EOConfiguration eoConfiguration = new EOConfiguration(new »
         File("WebContent/WEB-INF/eo-config.xml").toURI().toString());
    eoService = new EOService(eoConfiguration);
    eoService.loadMetadomain();
  }

  public void test1to1Create() {
    Employee emp = (Employee) »
```

```
        eoService.getMetaDomain().getEntity(Employee_EO.SYSTEM_NAME).newInstance();
    EmailAccount account = (EmailAccount) »
        eoService.getMetaDomain().getEntity(EmailAccount_EO.SYSTEM_NAME).newInstance();

    emp.setNameFull("John Smith");

    account.setAddress("johh.smith@athenaframework.org");
    account.setEmployee(emp);
    emp.setEmailAcct(account);

    UnitOfWork uow = eoService.createUnitOfWork();
    uow.persist(account); ❶
    uow.flush();

    System.out.println("Employee created: id = " + emp.getEmployee_ID() + "; its email account id: " + »
        emp.getEmailAcct().getEmailAccount_ID());
  }
```

Note that ❶ will not only persist `account` but also `emp` since the EmailAccount.employee relationship's cascade policy is PERSIST.

```
  public void test1to1Load() {
    EJBQLSelect select = eoService.createEOContext().createSelectQuery("SELECT e FROM Employee e »
        [e.emailAcct:J]");
    List<Object> employees = select.getResultList();
    for(int i=0; i < employees.size(); i++) {
      Employee emp = (Employee) employees.get(i);
      System.out.println(emp.getNameFull() + ", email address: " + (emp.getEmailAcct() == null ? »
              "(null)" : emp.getEmailAcct().getAddress()));
    }
  }
```

Above code loads Employee with relationship Employee.emailAcct. Below code removes email accounts with address ending with the specific term.

```
  public void test1to1Delete() { // remove all email account with address ending with »
     // 'athenaframework.org'
    UnitOfWork uow = eoService.createUnitOfWork();
    EJBQLSelect select = uow.createSelectQuery("SELECT e FROM Employee e [e.emailAcct:J]");
    List<Object> employees = select.getResultList();
    for(int i=0; i < employees.size(); i++) {
      Employee emp = (Employee) employees.get(i);
      if(emp.getEmailAcct() != null && »
              emp.getEmailAcct().getAddress().endsWith("athenaframework.org")) {
        uow.remove(emp.getEmailAcct());
                 emp.setEmailAcct(null);
      }
    }

    uow.flush();
  }
```

One to Many

One instance of entity A can be associated with many instances of entity B. One instance of entity B can be associated with at most one instance of entity A.

An employee can own multiple addresses, so we have an one-to-many relationship between `Employee` and `Address` as illustrated in Figure 4.8, “Employee-Address Relationship”.

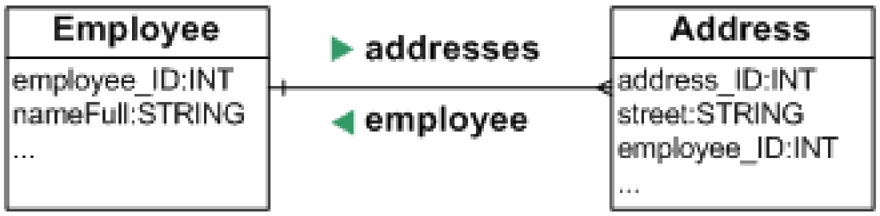


Figure 4.8. Employee-Address Relationship

First, create the Address entity if it does not exist yet. Then create a relationship named *addresses* on Employee and a relationship named *employee* on Address:

Table 4.5. Relationships: Employee.addresses and Address.employee

Property	**Employee.addresses**	**Address.employee**
System name	"addresses"	"employee"
Display name	"Addresses"	"Employee"
Relationship type	ONE_TO_MANY	MANY_TO_ONE
Cascade	ALL	PERSIST
Inverse	true	false
Source entity	Employee	Address
Source attribute	employee_ID	employee_ID
Target entity	Address	Employee
Target attribute	employee_ID	employee_ID

Relationships `Employee.addresses` and `Address.employee` forms a bidirectional relationship (refer to the section called “Relationships”).

Programming

The following code lists creating objects and forming relationships:

```
public void test1toMCreate() {
  Employee emp = (Employee) eoService.getMetaDomain().getEntity(Employee_EO.SYSTEM_NAME).newInstance();
  Address addr = (Address) eoService.getMetaDomain().getEntity(Address_EO.SYSTEM_NAME).newInstance();

  emp.setNameFull("John Smith");

  addr.setAddress("No. 1, Athena Street");

  emp.addToAddresses(addr, true); ❶
  // addr.setEmployee(emp);

  UnitOfWork uow = eoService.createUnitOfWork();
  uow.persist(emp);
  uow.flush();
}
```

❶ does not only set up Employee.addresses but also sets up the bidirectional complement Address.employee (updateComplementRelationship is true). It is equivalent to:

```
emp.addToAddresses(addr, false);
addr.setEmployee(emp);
```

Athena finds a relationship's complement by switching the source attribute with the target attribute. For example, Employee.addresses (source attribute: Employee.employee_ID; target attribute: Address.employee_ID) has a complement relationship Address.employee (source attribute: Address.employee_ID; target attribute: Employee.employee_ID).

The code below illustrates how to one-to-many relationships:

```
public void test1toMLoad() {
  EJBQLSelect select = eoService.createEOContext().createSelectQuery("SELECT e FROM Employee e »
     [e.addresses:J]");
  List<Object> employees = select.getResultList();
  for(int i=0; i < employees.size(); i++) {
    Employee emp = (Employee) employees.get(i);
    System.out.println(emp.getNameFull() + ", addresses: " + emp.getAddresses());
  }
}
```

If a relationship of an object has been not loaded (i.e., unresolved), any access to it will trigger the resolving procedure.

Removing one-to-many relationships is illustrated below:

```
public void test1toMDelete() { // remove the first address for the employee named 'John Smith'
  UnitOfWork uow = eoService.createUnitOfWork();
  EJBQLSelect select = uow.createSelectQuery("SELECT e FROM Employee e WHERE e.nameFull LIKE 'John »
     Smith'");
  Employee emp = (Employee) select.getSingleResult();

  List<EOObject> addresses = emp.getAddresses();
  if(addresses.size() > 0) {
    emp.removeFromAddresses(addresses.get(0), true);
    uow.remove(addresses.get(0));
  }

  uow.flush();
}
```

Many to Many

One instance of entity A can be associated with many instances of entity B, and vice verse.

A many-to-many relationship can be implemented using a junction table. For example, we have a many-to-many relationship between `Employee` and `Project` with `EmpProj` as the junction table, as illustrated in Figure 4.9, “Employee-Project Relationship”.

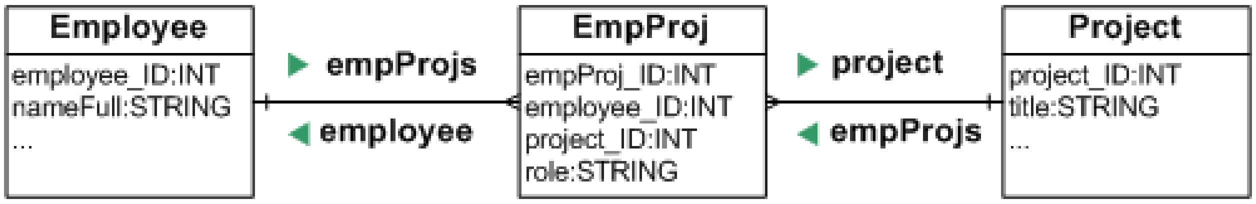


Figure 4.9. Employee-Project Relationship

First, create the Project and EmpProj entities if they do not exist yet. Then create a relationship named *emailAcct* on Employee and a relationship named *employee* on EmailAccount:

Table 4.6. Relationships: Employee.empProjs, EmpProj.employee, EmpProj.project and Project.empProjs

Property	Employee.empProjs	EmpProj.employee	EmpProj.project	Project.empProjs
System name	"empProjs"	"employee"	"project"	"empProjs"

Property	Employee.empProjs	EmpProj.employee	EmpProj.project	Project.empProjs
Display name	"Project association"	"Employee"	"Project"	"Employee assocation"
Relationship type	ONE_TO_MANY	MANY_TO_ONE	MANY_TO_ONE	ONE_TO_MANY
Cascade	ALL	PERSIST	PERSIST	ALL
Inverse	true	false	false	true
Source entity	Employee	EmpProj	EmpProj	Project
Source attribute	employee_ID	employee_ID	project_ID	project_ID
Target entity	EmpProj	Employee	Project	EmpProj
Target attribute	employee_ID	employee_ID	project_ID	project_ID

Programming

Setup a many-to-many relationship:

```
public void testM2MCreate() {
  Employee emp = (Employee) eoService.getMetaDomain().getEntity(Employee_EO.SYSTEM_NAME).newInstance();
  Project proj = (Project) eoService.getMetaDomain().getEntity(Project_EO.SYSTEM_NAME).newInstance();
  EmpProj empProj = (EmpProj) »
     eoService.getMetaDomain().getEntity(EmpProj_EO.SYSTEM_NAME).newInstance();

  emp.setNameFull("John Smith");
  proj.setTitle("Project A");

  empProj.setEmployee(emp);
  empProj.setRole_("Leader");
  empProj.setProject(proj);

  UnitOfWork uow = eoService.createUnitOfWork();
  uow.persist(empProj);
  uow.flush();
}
```

Load a many-to-many relationship:

```
public void testM2MLoad() {
  EJBQLSelect select = eoService.createEOContext().createSelectQuery("SELECT e FROM Employee e »
     [e.empProjs.project:J]");
  List<Object> employees = select.getResultList();
  for(int i=0; i < employees.size(); i++) {
    Employee emp = (Employee) employees.get(i);
    System.out.println(emp.getNameFull() + ", projects: ");
    for(int j=0; j < emp.getEmpProjs().size(); j++) {
      System.out.println("\t" + ((EmpProj)emp.getEmpProjs().get(j)).getProject().getTitle());
    }
  }
}
```

Remove a many-to-many relationship:

```
public void testMtoMDelete() { // remove project leaders in Project A
  UnitOfWork uow = eoService.createUnitOfWork();
  EJBQLSelect select = uow.createSelectQuery("SELECT e FROM Employee e WHERE e.empProjs.project.title = »
     'Project A' [e.empProjs.project:J]");
  Employee emp = (Employee) select.getSingleResult();

  for(int i = 0; i < emp.getEmpProjs().size(); i++) {
    if(((EmpProj)emp.getEmpProjs().get(i)).getRole_().equals("Leader")) {
      uow.remove(emp.getEmpProjs().get(i));
    }
  }
  uow.flush();
}
```

Cascade Rules

A cascade rule defines the persistence/deletion behavior of the target instance when the source instance gets deleted. There are three basic cascade types:

- **ALL** - Target instances should be persisted or deleted if the source instances is persisted or deleted. In relationship `Employee.emailAcct`, the target instance (EmailAccount) should be persisted or deleted if the source instance (Employee) gets persisted or deleted.
- **PERSIST** - Target instances should be persisted when the source entity is persisted; they should not be deleted if the source instance gets deleted. In relationship `EmailAccount.employee`, the target instance (Employee) should not be deleted if the source instance (EmailAccount) gets deleted.
- **DELETE** - Target instances should not be persisted when the source entity is persisted; they should be deleted if the source instance gets deleted.

5. Programming Athena Framework

The previous chapter explains entity relationship modeling in details. In this chapter, you'll learn essential runtime components that enable you to access metadata and to manipulate objects.

5.1 EOConfiguration and EOService

EOService is the core interface for you to access the Athena Framework; EOConfiguration specifies the data source and other settings for the EOService.

The XML Format of EOConfiguration

While you can manually create EOConfiguration (org.athenasource.framework.eo.core.EOConfiguration), usually you should initialize it from an XML file as the below format:

```
<?xml version="1.0" encoding="UTF-8"?>
<eo-system multitenancy="false" deletion-policy="hard">
  <datasources>
    <datasource>
      <database-type>MySQL</database-type>
      <host>localhost</host>
      <port>-1</port> <!-- '-1' means using the default port -->
      <username>root</username>
      <password>athena</password>
      <db>employeedir</db>
      <max-active>10</max-active>
      <max-idle>5</max-idle>
      <max-wait>5000</max-wait>
      <connection-timeout>3000</connection-timeout>
    </datasource>
  </datasources>
  <property name="java-source-local-dir" value="D:\SAMPLES\EmployeeDir\src"/>
</eo-system>
```

Configuration entries:

Table 5.1. Core Configuration Items of EO Configuration

Attribute/Element	Constraint	Description
multitenancy	Optional	Default is `false`. Set to `true` to enable multitenancy.
deletion-policy	Optional	Default is `hard`. Set to `soft` to use soft deletion, i.e. update the record status as deletion instead of perform SQL deletion.
datasource	Required	Currently one and only one datasource must to be declared.
database-type	Required	Valid values are `DB2,`

Attribute/Element	Constraint	Description
		`Derby, Oracle, MySQL`. More database types will be supported in future releases.
db	Required	the database name. For Derby, it is the relative or absolute path to a folder. When you use Athena Console to initialize the database, it does not need to exist now for Derby and MySQL; but must exist for DB2 and Oracle.
max-active	Required	The max number of connections in the pool.
max-idle	Required	The max number of connections to retain in the pool.
max-wait	Required	The max time in milliseconds to wait for a connection. A negative number means no limit.
connection-timeout	Required	Number of milliseconds an idle connection should be discarded. -1 means forever.
property: java-source-local-dir	Required	The target directory that generated Java source code should be written to.

By convention, you should name the configuration xml as eo-config.xml and put it into the WEB-INF folder (Java web applications). Once you have the eo-config.xml ready, you can initialize metadata and create entities and attributes as illustrated in previous chapters. In order to access any of the features provided by the Athena Framework at runtime, you need to obtain the EOService - the core interface to access all kinds of features of the Athena Framework.

Obtaining EOService

Java Web Application

To make EOService accessible from any method of any class in a Java web application, you only need to:

1. Register `org.athenasource.framework.eo.web.EOServletContextListener` as a servlet context listener in web.xml. When the web application starts, EOServletContextListener will create the EOService object and store its reference in the servlet context.
2. Register `org.athenasource.framework.eo.web.EOServletFilter` as a servlet filter that filters all urls. When a request is received, EOServletFilter will retrieve the EOService object from the servlet context and put it into thread local storage.

The corresponding configuration in web.xml:

```
<!-- Athena configurations starts -->
<context-param>
   <param-name>eo.configuration.file</param-name>
   <param-value>webapp:/WEB-INF/eo-config.xml</param-value>
</context-param>
<listener>
   <listener-class>org.athenasource.framework.eo.web.EOServletContextListener</listener-class>
</listener>

<filter>
  <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
  <filter-class>org.athenasource.framework.eo.web.EOServletFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Once the configuration is done, you may access EOService from any method of any class as the following code snippet illustrates:

```
EOService eoService = EOThreadLocal.getEOService();
```

Non-Web Applications

If you are developing a non-web Java application or you need more than one EOService in a web application, you can manually create EOService as following:

```
EOConfiguration eoConfig = new EOConfiguration(new File("PATH/TO/eo-config.xml").toURI().toString());
EOService eoService = new EOService(eoConfig);
eoService.loadMetadomain(); // only need to load once.
EOThreadLocal.setEOService(eoService); // Bind to thread local
// access eoService now ...
```

The code above may take a few seconds to execute. Once the EOService object is created, you should store its reference to somewhere that can be easily accessed.

Once EOService is obtained, you may use it to access metadata and to perform all kinds of operations on enterprise objects.

5.2 Accessing Metadata

At runtime, you may access any metadata information. A typical usage is to display entities or attributes' description fields as labels on the user interface. Once the EOService object is created and metadata is loaded through `loadMetadomain`, you can obtain the `MetaDomain` object that represents all metadata information (entities, attributes, and relationships, etc.):

```
MetaDomain metaDomain = eoService.getMetaDomain();
```

Querying Entities

List All Entities, Attributes, and Relationships

The below code print all entities, attributes and relationships:

```
MetaDomain metaDomain = eoService.getMetaDomain();
List<IEntity> entities = metaDomain.getAllEntities(false);
for(IEntity ientity : entities) {
  Entity entity = (Entity)ientity;
  if(entity.isCoreEntity()) {
    continue; // exclude system entities.
  }
  System.out.println("Entity: " + entity.getSystemName() + ", id: " + entity.getId());
  List<Attribute> attributes = entity.getAttributes();
  for(Attribute attribute : attributes) {
    System.out.println("\tAttribute: " + attribute.getSystemName() + ", " + attribute.getTypeInfo());
  }
  List<Relationship> relationships = entity.getRelationships();
  for(Relationship rel : relationships) {
    System.out.println("\tRelationship: " + rel.getSystemName() + ", " +
      rel.getSourceEntity().getSystemName() + "." + rel.getSourceAttribute().getSystemName() + "->" +
      rel.getTargetEntity().getSystemName() + "." + rel.getTargetAttribute().getSystemName());
  }
}
```

Sample output:

```
Entity: Employee, id: 101
  Attribute: employee_ID, INTEGER
  Attribute: version, INTEGER
  Attribute: status, TINYINT
  Attribute: ORG_ID, INTEGER
  Attribute: nameFull, VARCHAR(200)
  Attribute: bornYear, INTEGER
  Relationship: addresses, Employee.employee_ID->Address.employee_ID
  Relationship: emailAcct, Employee.employee_ID->EmailAccount.employee_ID
  Relationship: empProjs, Employee.employee_ID->EmpProj.employee_ID
Entity: Address, id: 102
  ...
```

Search Entities

Find an entity by system name:

```
Entity entity = (Entity) eoService.getMetaDomain().getEntity("Employee");
```

Find an entity by ID:

```
Entity entity = (Entity) eoService.getMetaDomain().getEntity(101);
```

Search Attributes

Find an attribute by system name:

```
Attribute attribute = entity.getAttributeBySystemName("nameFull");
```

Find an attribute by column index:

```
Attribute attribute = entity.getAttribute(4);
```

Search Relationships

Find a relationship by system name:

```
Relationship relationship = entity.getRelationship("addresses");
```

Reloading Metadata

While developing your application, you may frequently add and update entitiy, attributes and relationships. In such case, instead of restarting the application, you can reload metadata:

```
eoService.loadMetadomain(true);
```

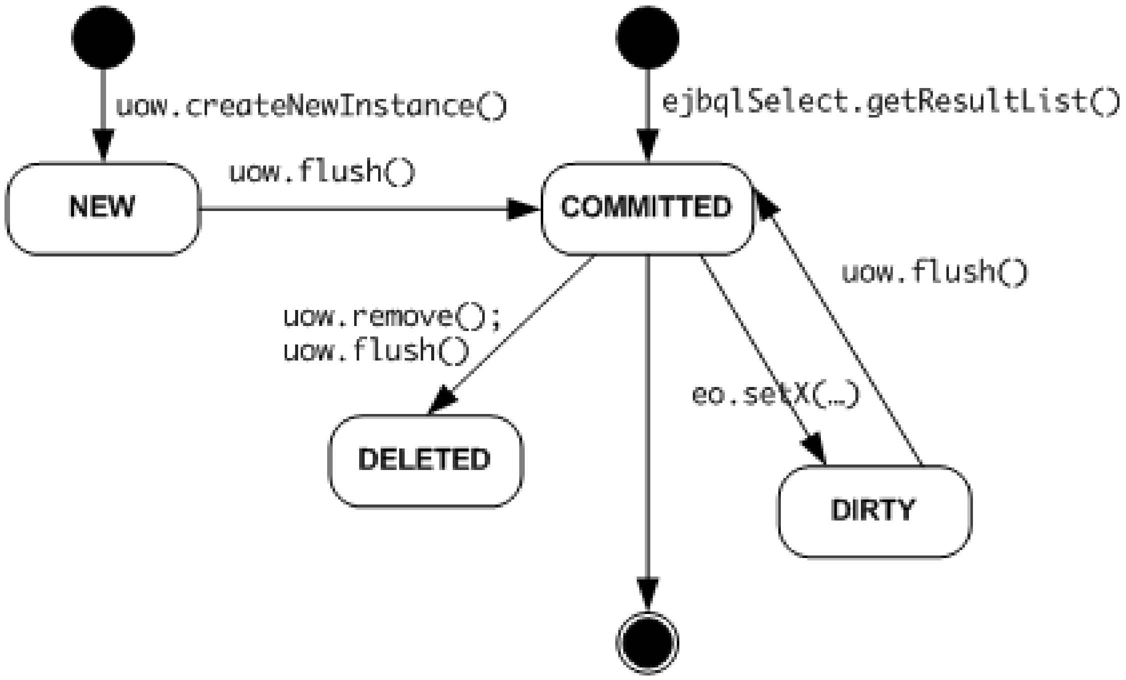
Note that you should abandon all EOContext, UnitOfWork and enterprise objects associated with previous versions of metadata once the metadata is reloaded.

5.3 Enterpise Object Manipulation

As a full-fledged object relational mapping framework, Athena enables the developer to work with high level objects instead of low level SQL statements. In Athena, a `UnitOfWork` is an object container of `EOObject`'s which are objects loaded from the database. UnitOfWork and EOObject allow the developer to manipulate and persist objects easily.

EOObject States

An EOObject may go through various states in its lifecycle:



NEW
: When an EOObject is created, it is in NEW state. At this moment, it has not been persisted to the database (there is no corresponding database record for it).

COMMITTED
: The EOObject has a matching record in the database.

DIRTY
: The EOObject has been modified in memory thus it has different property values with its matching record in the database. The change has not been persisted to the database yet.

DELETED
: The corresponding database record of a DELETED EOObject has been deleted (either hard or soft).

To check the state of an EOObject, you can use the following methods:

```
EOObject.isNew()
EOObject.isCommitted()
EOObject.isDirty()
EOObject.isDeleted(()
```

An EOObject can be *partially* loaded or fully loaded (default). A partial EOObject has one or more of its properties with missing values. More on partial EOObjects will be discussed later.

UnitOfWork and EOContext

A *Unit of Work* keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.
—Patterns of Enterprise Application Architecture, Martin Fowler

`UnitOfWork` is Athena's implementation of the Unit of Work pattern. In short, a `UnitOfWork` is a container of `EOObject`s that tracks object changes and persists such changes to the database when is flushed.

UnitOfWork ensures uniqueness of EOObject. Each database record results maximum one EOObject in a UnitOfWork. Let's say if you load Employee with id 100 in the unit of work. In a second query loading all employees, the same EOObject will be returned instead of a new EOObject being created for Employeed with id 100. This rule does not only keep the object graph consistent but also removes the confusion of object updating and improves the performance.

UnitOfWork tracks changes and commits them upon one single method call. Fetch ten employee records, modify some records, and delete a few - to commit such changes you need to write tedious SQL statements in traditional applications. In Athena, you just call `uow.flush()` and Athena will generates corresponding SQL statements and executes them in a transcation.

While `EOService` is a threadsafe object shared by all threads, **`UnitOfWork` are thread unsafe object that can be created and discarded whenever needed.** To create a `UnitOfWork`:

```
UnitOfWork uow = eOService.createUnitOfWork();
```

EOContext is the super class of `UnitOfWork` and it is a read-only version of `UnitOfWork`. You use EOContext if you only need to query records but not to modify them. To create a `EOContext`:

```
EOContext context = eoService.createEOContext();
```

Querying Objects

You may use EJBQL statements to query objects explicitly or through implicit relationship loading.

General EJBQL Querying

Direct execution of EJBQL queries is the most powerful and flexible query method. Sample code:

```
EOContext context = eoService.createEOContext();
EJBQLSelect select = context.createSelectQuery("SELECT e FROM Employee e WHERE e.nameFull LIKE »
 'Jack%'");
List<Object> emps = select.getResultList();
System.out.println("Total number of employees found: " + emps.size());
```

```
for (Iterator iterator = emps.iterator(); iterator.hasNext();) {
  Employee emp = (Employee) iterator.next();
  System.out.println("Emp: " + emp.getNameFull());
}
```

To create a `EJBQLSelect` object, you use:

```
EJBQLSelect select = EOContext.createSelectQuery(...)
```

To execute an EJQBL statement:

```
List<Object> resultList = select.getResultList(); // expecting multiple objects
Object result = select.getSingleResult(); // expecting zero or one object to be returned
```

Athena's EJBQL implementation supports advanced features like relationship loading and partial object. You may refer to the EJBQL chapter for more details.

Find EOObject by ID

To find a EOObject in a `EOContext`, you may use the `find()` method as following:

```
Employee emp = (Employee)context.find(Employee_EO.SYSTEM_NAME, 100, null, null); // find emp with id »
 // 100
```

The `find` method returns the EOObject with the given id from the context or load from the database if it does not exist in the context.

Loading Through Relationships

When you navigate relationships of an EOObject, the relationships will be automatically resolved. The code below finds an employee through EJBQL and loads the addresses associated with the employee through relationship.

```
EJBQLSelect select = context.createSelectQuery("SELECT e FROM Employee e {result_first='1', »
 result_max='1'}"); // returns max 1 employee.
Employee emp = (Employee) select.getSingleResult();
if(emp != null) {
  List<EOObject> addrs = emp.getAddresses();
  for(int i = 0; i < addrs.size(); i++) {
    System.out.println("Address " + i + ": " + ((Address)addrs.get(i)).getAddress());
  }
}
```

Internally, Athena generates and executes proper EJBQL to resolve relationships.

Creating Objects

To create an instance of certain type of EOObject, you use `UnitOfWork.createNewInstance`, then you use `UnitOfWork.flush` to persist the new object. The code below creates an Employee instance and an Address instanace:

```
UnitOfWork uow = eoService.createUnitOfWork();
Employee emp = (Employee) uow.createNewInstance(Employee_EO.SYSTEM_NAME);
emp.setNameFull("Jane Smith");
emp.setBornYear(1970);

Address addr = (Address) uow.createNewInstance(Address_EO.SYSTEM_NAME);
addr.setAddress("No. 1 New City Street");
```

```
emp.addToAddresses(addr, true); // wire up the relationship.

uow.flush();

System.out.println("New employee created. ID: " + emp.getEmployee_ID());
```

`uow.flush()` obtains a database connection, generates proper SQL statements and commits the changes in one transaction.

Updating and Deleting Objects

Update an EOObject

To update an EOObject, you can simply call its `setXXX` method and then call `UnitOfWork.flush` to commit the changes. The following code snippet finds a Employee instance and updates one of the properties:

```
UnitOfWork uow = eoService.createUnitOfWork();
EJBQLSelect select = uow.createSelectQuery("SELECT e FROM Employee e {result_first='1', »
 result_max='1'}");
Employee emp = (Employee) select.getSingleResult();
if(emp != null) {
    System.out.println("Updating name for Employee with id: " + emp.getEmployee_ID());
  emp.setNameFull(emp.getNameFull() + " New");
}
uow.flush();
```

In above code, only one object is updated. When `uow.flush()` is executed, all changes made on all objects managed by the UnitOfWork will be persisted.

Delete an EOObject

To delete an EOObject, you simply register it using `UnitOfWork.remove` and then call `UnitOfWork.flush` to update the database. The code below finds an Employee instance and deletes it.

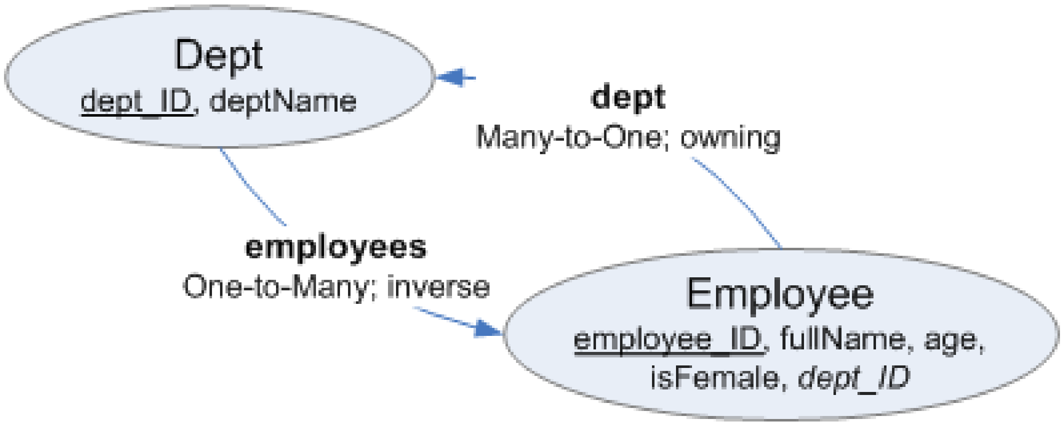
```
UnitOfWork uow = eoService.createUnitOfWork();
EJBQLSelect select = uow.createSelectQuery("SELECT e FROM Employee e {result_first='1', »
 result_max='1'}");
Employee emp = (Employee) select.getSingleResult();
if(emp != null) {
  System.out.println("Deleting Employee with id: " + emp.getEmployee_ID());
  uow.remove(emp);
}
uow.flush();
```

When an EOObject is being deleted, target objects of its relationships with cascade deletion will be deleted too. Deletion of one EOObject instance could result deletion of many related objects through relationships. For more details, please refer to the Entity Relationship Modeling chapter.

6. EJBQL Reference

JSR220 Enterprise JavaBeans 3.0 defines EJB QL, which is an object query language provided for navigation across a network of objects. Athena Framework provides an extended version of EJB QL for object querying. In this document, unless explicitly specified, EJB QL or EJBQL refer to the Athena extended version instead of the EJB QL defined in the EJB specification.

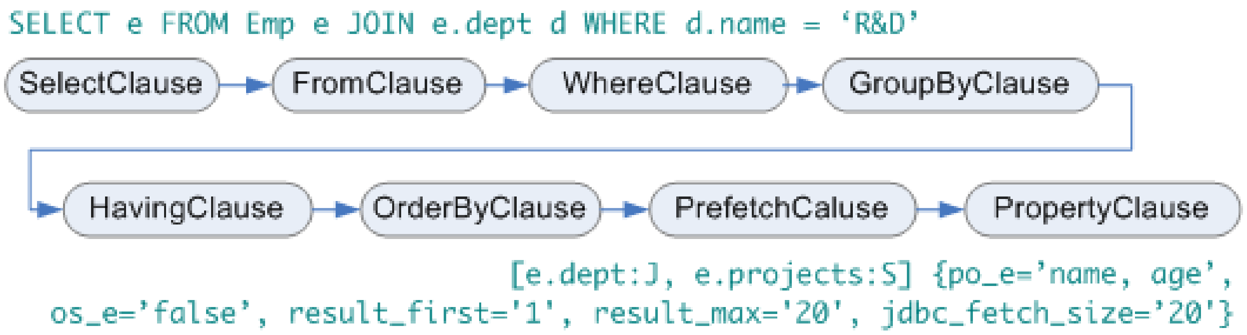
We'll use the following entities and relationships to illustrate the usages of various EJBQL's.



6.1 EJBQL Basics

Query Statement Syntax

The query statement syntax is illustrated below:



```
SELECT e FROM Emp e JOIN e.dept d WHERE d.name = 'R&D'
```

```
[e.dept:J, e.projects:S] {po_e='name, age', os_e='false', result_first='1', result_max='20', jdbc_fetch_size='20'}
```

The following sections introduce many techniques. For the sake of brevity, the examples use only the corresponding techniques, however, please feel free to combine all the techniques to enjoy the full power of Athena EJBQL.

Execution of an EJBQL

You can use `EOContext` or `UnitOfWork` to create an EJBQL statement and then execute it. Sample code:

```
EJBQLSelect select = context.createSelectQuery(ejbql);
List list = select.getResultList();
// Object object = select.getSingleResult(); // If zero or one record is expected
```

Basic Select Query

Syntax: **SELECT varName FROM EntityName varName WHERE conditions**

Example EJBQL: `SELECT d FROM Dept d WHERE d.dept_ID < 10`

Actual SQL Executed: // SQL generated varies across databases.

```
SELECT d.dept_ID, d.status, d.version, d.deptName FROM TT_Dept d WHERE d.dept_ID < 10 AND d.status »
 <> 4
```

Result:

```
java.util.ArrayList #0
  [0] [Dept:3C]-EOContext:qjX #1
    employees -> [unresolved]
  [1] [Dept:4C]-EOContext:qjX #2
    employees -> [unresolved]
```

Note: in the translated SQL, Athena appends d.status <> 4 to indicate selecting only the objects that are not marked as deleted (soft deletion). For an application with soft deletion, when an object gets deleted, its corresponding record in the database will be marked as deleted by setting its status field to 4 instead of real deletion.

Warning: When there is a JOIN with a to-many relationship (regardless how you specify the join – JOIN or […]), you should not use ORDER BY and/or result-first/result-max as the EJBQL engines use un-ordered full records to wire up the relationships. If you use ORDER BY and/or result-first/result-max, you may get incomplete relationship lists (for example a Department with only some of its employees). If you need to sort the result, consider to sort the result in Java. Alternatively, you can fetch the to-many relationship using subselect.

6.2 Relationship and Prefetch

Join Through Relationships

Used when: **you need to JOIN two entities for filtering purpose**

Syntax: … **FROM EntityName varName [[INNER] | LEFT] JOIN varName.relName varName2 WHERE** …

There are two kinds of JOINs: *left join* – when LEFT JOIN is specified; *inner join* – when INNER JOIN or the only word JOIN is specified.

Example EJBQL: `SELECT DISTINCT d FROM Dept d JOIN d.employees e WHERE e.fullName LIKE '%Jack%'` - find all departments that have at least one employee whose name contains Jack.

Actual SQL Executed:

```
SELECT d.dept_ID, d.status, d.version, d.deptName FROM TT_Dept d INNER JOIN TT_Emp e ON d.dept_ID = »
 e.dept_ID WHERE e.fullName LIKE '%Jack%' AND d.status <> 4 AND e.status <> 4
```

Result:

```
java.util.ArrayList #0
    [0] [Dept:4C]-EOContext:KxE #1
    employees -> [unresolved]
```

Note: the DISTINCT keyword filters out duplicated occurrences of objects.

Relationship Prefetch Using Join

Used when: **you need to prefetch relationships for selected objects (the number of average related objects per object is smaller)**

Syntax: … **FROM EntityName varName … [varName.relName:J, …]**

Example EJBQL: `SELECT DISTINCT d FROM Dept d WHERE d.dept_ID < 10 [d.employees:J]`

Actual SQL Executed:

```
SELECT d.dept_ID, d.status, d.version, d.deptName, e.employee_ID, e.status, e.version, e.fullName, »
 e.age, e.isFemale, e.dept_ID FROM TT_Dept d LEFT OUTER JOIN TT_Emp e ON d.dept_ID = e.dept_ID WHERE »
 d.dept_ID < 10 AND d.status <> 4 AND e.status <> 4
```

Result:

```
java.util.ArrayList #0
    [0] [Dept:3C]-EOContext:PgX #1
    employees -> (target objects: 1)
      [Employee:4C]-EOContext:PgX #2
        dept -> [Dept:3C]-EOContext:PgX #1
    [1] [Dept:4C]-EOContext:PgX #3
    employees -> (target objects: 2)
      [Employee:5C]-EOContext:PgX #4
        dept -> [Dept:4C]-EOContext:PgX #3
      [Employee:6C]-EOContext:PgX #5
        dept -> [Dept:4C]-EOContext:PgX #3
```

Note: When the number of average related objects for an object is very large, prefetch using JOIN will result huge amount of redundant data output from the database server. In such cases, you should use prefetch with select.

You can **prefetch multi-level relationships**, e.g, `d.employees.addresses`.

You can **prefetch any number of relationships** in a single select statement, e.g.,`[d.employees:J, d.projects:S, …]`

Prefetch Using Select

Used when: **you need to prefetch relationships for selected objects**

Syntax: **… FROM EntityName varName … [varName.relName:S, …]**

Example EJBQL: `SELECT DISTINCT d FROM Dept d WHERE d.dept_ID < 10 [d.employees:S]`

Actual SQL's Executed:

```
SELECT d.dept_ID, d.status, d.version, d.deptName FROM TT_Dept d WHERE d.dept_ID < 10 AND d.status »
 <> 4

SELECT D.dept_ID, e.employee_ID, e.status, e.version, e.fullName, e.age, e.isFemale, e.dept_ID FROM »
 TT_Dept D LEFT OUTER JOIN TT_Emp e ON D.dept_ID = e.dept_ID WHERE D.dept_ID IN (3, 4) AND D.status »
 <> 4 AND e.status <> 4 ORDER BY D.dept_ID
```

Result:

```
java.util.ArrayList #0
    [0] [Dept:3C]-EOContext:lFk #1
    employees -> (target objects: 1)
      [Employee:4C]-EOContext:lFk #2
        dept -> [Dept:3C]-EOContext:lFk #1
    [1] [Dept:4C]-EOContext:lFk #3
    employees -> (target objects: 2)
      [Employee:5C]-EOContext:lFk #4
        dept -> [Dept:4C]-EOContext:lFk #3
      [Employee:6C]-EOContext:lFk #5
        dept -> [Dept:4C]-EOContext:lFk #3
```

Pick the right prefetch strategies:

Table 6.1. Prefetch Strategy Comparison

	Prefetch using Join	**Prefetch using Select**
SQL Performance	only one SQL statement is need	Two SQL statements are needed
Redundancy	Potential huge redundant data	No redundant data excepted the ID
Used when	Number of selected objects and related objects is small.	Large amount of objects.

6.3 Partial Objects

Used when: **you only need a small set of attributes of objects**. For example, if there are more than a hundred attributes for Employee and you only need to display the name and age for each employee. Obvious, it's waste if you retrieve all the attributes especially when you need retrieve a large number of Employees. In such cases, you can retrieve partial objects.

Syntax: **… [varName.relName:J, …] {po_varNameOrPath=’attr1, attr2…’; …}**

Example EJBQL: `SELECT DISTINCT d FROM Dept d WHERE d.dept_ID < 10 [d.employees:J] {po_d.employees='fullName, age'}`

Actual SQL Executed:

```
SELECT d.dept_ID, d.status, d.version, d.deptName, e.employee_ID, e.fullName, e.age, e.version, »
 e.status FROM TT_Dept d LEFT OUTER JOIN TT_Emp e ON d.dept_ID = e.dept_ID WHERE d.dept_ID < 10 AND »
 d.status <> 4 AND e.status <> 4
```

Result:

```
java.util.ArrayList #0
    [0] [Dept:3C]-EOContext:PVP #1
    employees -> (target objects: 1)
      [Employee:4C]-EOContext:PVP(PartialObject) #2
        dept -> [Dept:3C]-EOContext:PVP #1
    [1] [Dept:4C]-EOContext:PVP #3
    employees -> (target objects: 2)
      [Employee:5C]-EOContext:PVP(PartialObject) #4
        dept -> [Dept:4C]-EOContext:PVP #3
      [Employee:6C]-EOContext:PVP(PartialObject) #5
        dept -> [Dept:4C]-EOContext:PVP #3
```

After the Employee objects are retrieved, you can access their fullName and age attributes. For all attributes other than the explicit loading attributes and default loading attributes (PK ID, version and status), access to them will result exception.

The following methods are available for enquiring status of a partial object:

- `boolean isPartialObject()`
- `boolean isAttributeLoaded(int attrIndex)`
- `boolean isAttributeLoaded(String attrName)`

Note when use po_PATH to specify partial loading, PATH must be either in the SELECT clause or it is prefect using JOIN. If PATH refers to a prefetch using SELECT, you’ll get error – in that case, please consider to split the select into separate ones.

6.4 Querying Scalar Values

Select Attribute Values

Used when: **you need to get attribute values instead of objects**

Syntax: SELECT varName.attrName … FROM EntityName varName …

Select one attribute only

Example EJBQL #1: **SELECT d.deptName FROM Dept d**

Actual SQL Executed:

```
SELECT d.deptName FROM TT_Dept d WHERE d.status <> 4
```

Result:

```
java.util.ArrayList #0
    [0] Sales
    [1] Engineering
```

Select multiple attributes

Example EJBQL #2: **SELECT d.dept_ID, d.deptName FROM Dept d**

Actual SQL Executed:

```
SELECT d.dept_ID, d.deptName FROM TT_Dept d WHERE d.status <> 4
```

Result – each record is an array (of type Object[]) of attribute values:

```
java.util.ArrayList #0
    [0] [Ljava.lang.Object; #1
      [0] 3
      [1] Sales
    [1] [Ljava.lang.Object; #2
      [0] 4
      [1] Engineering
```

Mixed selection of attributes and objects

Example EJBQL #3: **SELECT d.deptName, e FROM Employee e JOIN e.dept d**

Actual SQL Executed:

```
SELECT d.deptName, e.employee_ID, e.status, e.version, e.fullName, e.age, e.isFemale, e.dept_ID FROM »
 TT_Emp e INNER JOIN TT_Dept d ON e.dept_ID = d.dept_ID WHERE e.status <> 4 AND d.status <> 4
```

Result – each record is an array containing the dept. name and the employee object.

```
java.util.ArrayList #0
    [0] [Ljava.lang.Object; #1
      [0] Sales
      [1] [Employee:4C]-EOContext:7ON #2
    [1] [Ljava.lang.Object; #3
      [0] Engineering
      [1] [Employee:5C]-EOContext:7ON #4
    [2] [Ljava.lang.Object; #5
      [0] Engineering
      [1] [Employee:6C]-EOContext:7ON #6
```

Aggregate Functions

Used when: **you need to get information from a collection of input values**

Syntax: `AVG/MAX/MIN/SUM(varName.attrName) COUNT(varName[.attrName])`

Use AVG/MAX/MIN/SUM

Example EJBQL #1: **SELECT d.deptName, AVG(e.age) FROM Employee e JOIN e.dept d GROUP BY e.dept_ID –** returns dept name and average age for its employees

Actual SQL Executed:

```
SELECT d.deptName, AVG(e.age) FROM TT_Emp e INNER JOIN TT_Dept d ON e.dept_ID = d.dept_ID WHERE »
 e.status <> 4 AND d.status <> 4 GROUP BY e.dept_ID
```

Result: // getResultList()

```
java.util.ArrayList #0
    [0] [Ljava.lang.Object; #1
      [0] Sales
      [1] 30.0000
    [1] [Ljava.lang.Object; #2
      [0] Engineering
      [1] 30.0000
```

Use COUNT

COUNT differs to other aggregate function in the way that it allows both entity and attribute as the argument. If you pass the entity, the entity’s primary keys will be used for counting.

Example EJBQL: **SELECT COUNT(e) FROM Employee e JOIN e.dept d WHERE d.dept_ID = 3 –** counts the total number of employees in a given dept

Actual SQL Executed:

```
SELECT COUNT(e.employee_ID) FROM TT_Emp e INNER JOIN TT_Dept d ON e.dept_ID = d.dept_ID WHERE d.dept_ID »
 = 3 AND e.status <> 4 AND d.status <> 4
```

Result: // getSingleResult()

```
1
```

6.5 Pagination

Used when: **you need to get only a subset of the result records**

Syntax: … `ORDER BY varName.attrName { result_first='oneBasedIndex', result_max='count'}`

Example EJBQL: **SELECT e FROM Employee e ORDER BY e.fullName {result_first='1', result_max='2'} –** returns the first two records found

Actual SQL Executed:

```
SELECT e.employee_ID, e.status, e.version, e.fullName, e.age, e.isFemale, e.dept_ID FROM TT_Emp e WHERE »
 e.status <> 4 ORDER BY e.fullName LIMIT 0, 2
```

Result:

```
java.util.ArrayList #0
    [0] [Employee:5C]-EOContext:SBx #1
    dept -> [unresolved]
    [1] [Employee:6C]-EOContext:SBx #2
    dept -> [unresolved]
```

When using pagination, you should always add an ORDER BY clause. Otherwise, the returned records could be in random order.

6.6 Using Input Parameters

Used when: **you need to execute similar EJBQL's which differs in one or two parameters**

Syntax: … `:namedParam … ?positionalParam`

There are two different types of input parameters – named parameters and positional parameters. A named parameter is designated by colons (:) followed by a string while a positional input parameter is designated by a question mark (?) followed by an integer – the first input parameter is ?1 and the second is ?2. Before you call getSingleResult/getResultList methods, you should use setParameter methods to set values for the parameters first.

Example of Named Input Parameters

```
EJBQLSelect select = context.createSelectQuery(
  "SELECT d.deptName FROM Dept d WHERE d.dept_ID = :id");
select.setParameter("id", 3);
Object object = select.getSingleResult();
System.out.println("Result: " + EOUtils.traceEO(object, 0, null, null));

select.setParameter("id", 4);
object = select.getSingleResult();
System.out.println("Result: " + EOUtils.traceEO(object, 0, null, null));
```

Rewrite Using Positional Input Parameters

```
EJBQLSelect select = context.createSelectQuery(
  "SELECT d.deptName FROM Dept d WHERE d.dept_ID = ?1");
select.setParameter(1, 3);
Object object = select.getSingleResult();
System.out.println("Result: " + EOUtils.traceEO(object, 0, null, null));

select.setParameter(1, 4);
object = select.getSingleResult();
System.out.println("Result: " + EOUtils.traceEO(object, 0, null, null));
```

6.7 Other Optimization Techniques

Used when: **you need to retrieve a large number of records**

Syntax: … `{ … , jdbc_fetch_size='expectedSize'}`

Example EJBQL: **SELECT e FROM Employee e ORDER BY e.fullName {result_first='1', result_max='1000', jdbc_fetch_size='250'}**

The JDBC fetch size gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed. For large queries that return a large number of objects you can configure the row fetch size used in the query to improve performance by reducing the number database hits required to satisfy the selection criteria.

Most JDBC drivers default to a fetch size of 10, so if you are reading 1000 objects, increasing the fetch size to 256 can significantly reduce the time required to fetch the query's results. The optimal fetch size is not always obvious. **Usually, a fetch size of one half or one quarter of the total expected result size is optimal.** Note that if you are unsure of the result set size, incorrectly setting a fetch size too large or too small can decrease performance.

6.8 Organization Specific Querying

Used when: **you need to explicitly specify organization specific data querying**

Syntax: … `{ … , os_varNameOrPath='true/false'}`

Example EJBQL: **SELECT e FROM Employee e {os_e='false'}**

Athena has built-in support for multi-tenant hosting. Each business, company or institute is called as an 'organization'. Data can be classified into two types – cross-organization and organization-specific. Most of the metadata are cross-organization (shared by all organizations) and application data are usually organization-specific (belonging to certain organization). When a user logs on the system, the organization she belongs can be easily retrieved. For anonymous browsing, the organization id is obtained through the client configuration (in HTML code for SWF and static configuration file for AIR).

By default, data of core entities (Entity, Attribute, Relationship, Picklist, etc.) are treated as cross-organization and data of other entities (User, Org, …) are treated as organization-specific. When querying data from organization-specific entities, the generated SQL always includes a WHERE clause 'path.ORG_ID = …' for filtering purpose.

To modify the above behavior, you can explicitly specify the organization-specific property for entities to be queried.

Example EJBQL without os_ spec: **SELECT c FROM Career c**

Actual SQL Executed:

```
SELECT c.career_ID, c.status, c.version, c.ORG_ID, ... FROM ES_Career c WHERE c.status <> 4 AND »
 c.ORG_ID = 7
```

Result:

```
java.util.ArrayList #0
```

Example EJBQL with os_ spec: **SELECT c FROM Career c {os_c='false'}**

Actual SQL Executed:

```
SELECT c.career_ID, c.status, c.version, c.ORG_ID, ... FROM ES_Career c WHERE c.status <> 4
```

Result:

```
java.util.ArrayList #0
  [0] [Career:1C]-EOContext:BSr #1
  [1] [Career:2C]-EOContext:BSr #2
```

7. Transactions

Transactions ensure data integrity of your application. If you only use one database in your application, you may skip this chapter as `UnitOfWork` takes cares most of the work. However, if you need to update two or more databases from different vendors simultaneously, you have to use proper transactions to maintain data integrity. Athena offers flexible and straightforward options for you to manage any kind of transactions easily.

In this chapter, we'll use a typical bank transaction as an example: transferring $100 from account A to account B.

7.1 Unit Of Work: JDBC Transactions

Used when: **there is one database to be updated**

One of the transaction's ACID properties, atomicity, is defined as "a transaction is an indivisible unit of work". In Athena, `UnitOfWork` represents an unit of work. Typically, you query objects, update them, and call `UnitOfWork.flush()` to make all changes of these objects in one database transaction.

Pseudo code to perform the bank transfer:

```
Account A = uow.find('A');
Account B = uow.find('B');
A.setBalance(A.getBalance() - 100);
B.setBalance(B.getBalance() + 100);
uow.flush();
```

When `UnitOfWork.flush()` is called, Athena will finds all the dirty objects and generates proper DML statements and then executes in one database transaction.

7.2 JTA Transactions

Used when: **there is two or more databases to be updated simultaneously**

JTA stands for Java Transaction API. JTA allows the developer to perform distributed transactions across multiple databases from different vendors.

In the same bank transfer example, assume account A and account B are stored in different databases. The pseudo code for perform the transaction using JTA:

```
DataSource ds1 = ...;
DataSource ds2 = ...;

UserTransactionManager utm = new UserTransactionManager();
utm.init();
utm.begin();

Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();

try {
  uow1.flush(conn1);
  uow2.flush(conn2);

  conn1.close();
  conn2.close();

  utm.commit();
} catch (Throwable t) {
  utm.rollback();
}

utm.close();
```

The code above invokes the `UnitOfWork.flush(Connection)` method that uses the connection passed to perform DML statements. If no connection is passed, a connection is obtained from the data source defined in the eo-config.xml.

7.3 Container Managed Transactions

Container managed transactions are less flexible than bean managed transactions; however, they are easy to declare and require less code. A container managed transaction begins when an enterprise bean method starts and commits before the method ends. You use a transaction attribute to control the scope of a transaction.

The code below illustrates how the bank transfer is performed in a container managed transaction:

```
@TransactionAttribute(REQUIRED)
public void doTransfer() {
  initCtx = new InitialContext();
  DataSource ds = (DataSource)initCtx.lookup("DATASOURCE JNDI NAME");
  Connection conn = ds.getConnection();

  Account A = uow.find('A');
  Account B = uow.find('B');
  A.setBalance(A.getBalance() - 100);
  B.setBalance(B.getBalance() + 100);
  uow.flush(conn);

  conn.close();
}
```

Again, `UnitOfWork.flush(Connection)` is invoked instead of `UnitOfWork.flush()`. The method in above code is tagged with a REQUIRED transaction attribute, which means the container should starts a new transaction is the caller client is not associated with any transaction.

8. Multitenancy

Nowdays, cloud computing has been widely adopted by individuals and large enterprises for agility, scalability and low cost. Cloud computing can be loosely categorized into:

- **Infrastructure as a Service (IaaS)** provides operation systems, storage, network, and other infrastructure services. Amazon Web Service is good example, which offer EC2 (computing instances), S3 (storage), etc.
- **Platform as a Service (PaaS)** provides software platform that allows developers to build applications on it without worrying about the underlying infrastructure. Force.com and Google App Engine are examples of this category.
- **Software as a Service (SaaS)** provides various applications to end users, also known as on-demand software. Examples include salesforce.com, NetSuite, and Google Apps.

Athena Framework can help developers build both Platform as a Service and Software as a Service cloud applications. Being an open source framework with no dependency on bloated libraries, Athena provides a simple and clean solution for you to build SaaS applications quickly.

8.1 Overview

One of the key requirements for SaaS architecture design is multitenancy. Multitenancy allows a single instance of the application to serve multiple organizations virtually isolated from one another. Multi-Tenant Data Architecture by Microsoft is one of the most quoted articles on multitenancy. It discusses three distinct approaches to implement multitenancy:

- `separated databases` (store tenant data in separated databases, e.g., using DB1 and DB2 to store data for org 1 and org 2 respectively);
- `separated schemas` (store tenent data in separated tables of the same database, e.g., using TBL1 and TBL2 tables to store data for org 1 and org 2 respectively); and
- `shared schema` (store data for all tenants in the same sets of tables).

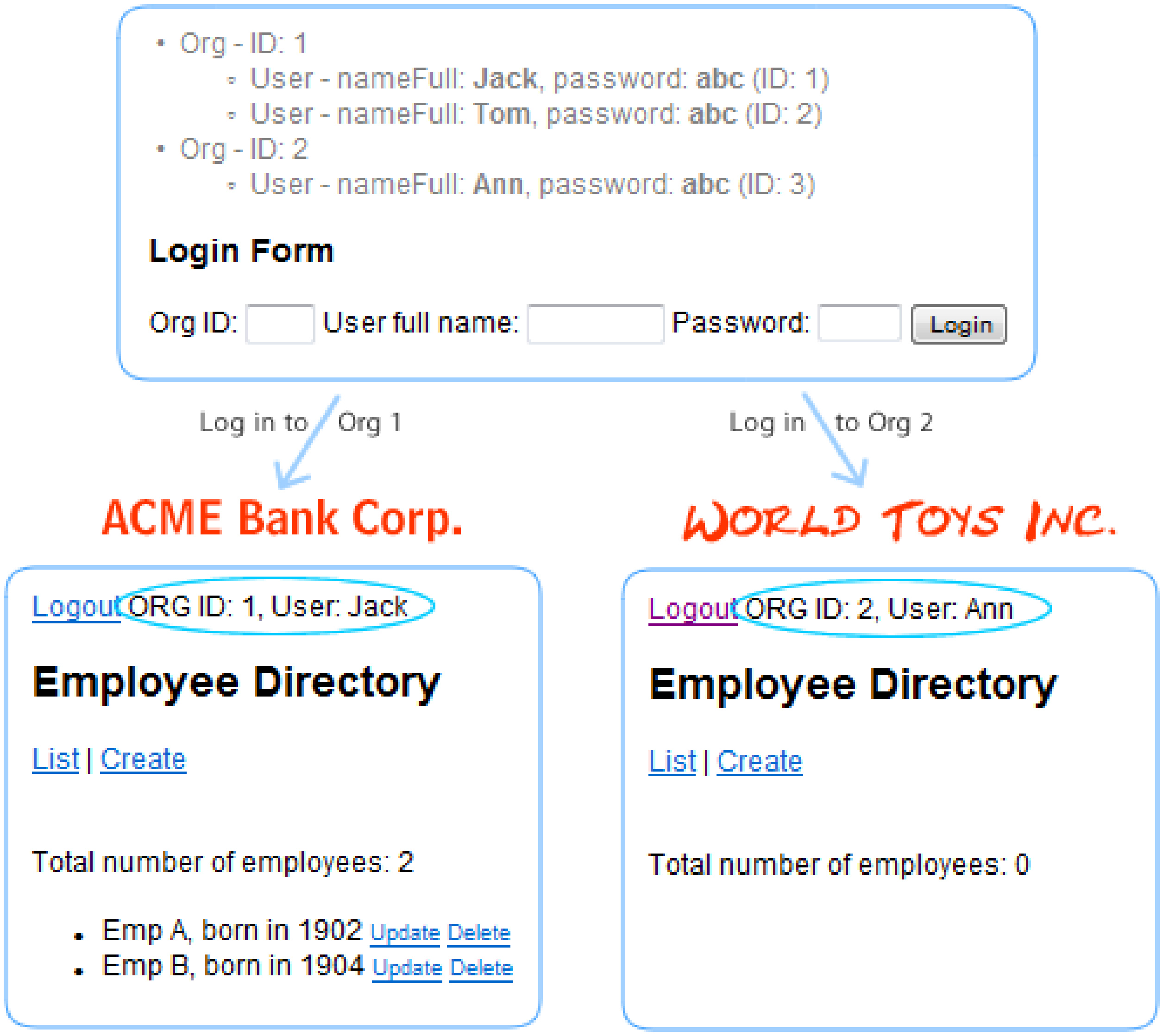
Separated databases and separated schemas make application maintenance a nightmare: to upgrade the application, you need to upgrade N databases or tables - N is the total number of tenants. Shared schema comes to rescue - you only need to update only one database. However, of the three approaches, shared schema has the highest initial cost.

Athena Framework significantly reduces your initial investment on shared schema. With Athena, creating a multitenant application only requires two additional steps over tranditional applications:

1. Set the `multitenancy` flag to true in `eo-config.xml`;
2. Create an entity to represent organizations and corresponding utility to maintain organizations.

As Athena Framework has built-in support for multitenancy, you can easily convert any traditional applications into SaaS applications with little effort.

Chapter 2 walks you through a tutorial creating an employee directory application. In this chapter, we'll create the same application, except that it supports multitenancy. As illustrated in the picture below, users at different organizations will see data for his or her organization after logged in.



- Org - ID: 1
  - User - nameFull: **Jack**, password: **abc** (ID: 1)
  - User - nameFull: **Tom**, password: **abc** (ID: 2)
- Org - ID: 2
  - User - nameFull: **Ann**, password: **abc** (ID: 3)

Login Form

Org ID: User full name: Password: Login

Logout ORG ID: 1, User: Jack

Employee Directory

List | Create

Total number of employees: 2

- Emp A, born in 1902 Update Delete
- Emp B, born in 1904 Update Delete

Logout ORG ID: 2, User: Ann

Employee Directory

List | Create

Total number of employees: 0

8.2 Project Setup and Application Coding

First, create a project in your favorite IDE.

Project Creation

Create a project and add Athena dependency:

Eclipse

Create a new project File -> New -> Dynamic Web Project, project name: `EmployeeDirMT`. Assume PROJECT_ROOT is the root folder of the project:

1. Copy all the jar files (jar files in root folder and `lib` folder) from Athena Framework to `PROJECT_ROOT/WebContent/WEB-INF/lib`.

NetBeans

Create a new project File -> New Project -> Java Web -> Web Application, project name: `EmployeeDirMT`. Assume PROJECT_ROOT is the root folder of the project:

1. Copy all the jar files (jar files in root folder and `lib` folder) from Athena Framework to `PROJECT_ROOT/web/WEB-INF/lib` (create the 'lib' folder first);
2. Right click on the project, and select 'Properties' to open the Project Properties dialog. Select 'Libraries' on the left panel, then click 'Add JAR/Folder' and browse to `PROJECT_ROOT/web/WEB-INF/lib` and select all the jar files then click 'Open'. Now, Athena Framework runtime has been successfully added to path. Click 'OK' to dimiss the dialog.

Maven

Create a new project using maven with the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.athenasource.framework »
 -DarchetypeArtifactId=athena-webapp-basic -DarchetypeVersion=2.0.0 »
 -DarchetypeRepository=http://athenasource.org/dist/maven/repo -DgroupId=com.test »
 -DartifactId=EmployeeDirMT -Dversion=1.0-SNAPSHOT
```

Modify web.xml

In most cases, the IDE generates a `web.xml` under the `WEB-INF` folder. Modify it as following to add Athena support:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" »
 xmlns="http://java.sun.com/xml/ns/javaee" »
 xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee »
     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>EmployeeDirMT</display-name>
  <welcome-file-list>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>

  <!-- Athena configuration starts -->
  <context-param>
    <param-name>eo.configuration.file</param-name>
    <param-value>webapp:/WEB-INF/eo-config.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.athenasource.framework.eo.web.EOServletContextListener</listener-class>
  </listener>

  <filter>
    <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
    <filter-class>org.athenasource.framework.eo.web.EOServletFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>org.athenasource.framework.eo.web.EOServletFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
    <!-- Athena configuration ends -->

</web-app>
```

Create eo-config.xml

Create a `eo-config.xml` as following under the `WEB-INF` directory:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<eo-system deletion-policy="hard" multitenancy="true">
  <datasources>
    <datasource>
      <database-type>MySQL</database-type>
      <host>localhost</host>
      <port>-1</port> <!-- '-1' means using the default port -->
      <username>root</username>
      <password>athena</password>
      <db>employeedirMT</db>
      <max-active>10</max-active>
      <max-idle>5</max-idle>
      <max-wait>5000</max-wait>
      <connection-timeout>300</connection-timeout>
    </datasource>
  </datasources>
  <property name="java-source-local-dir" value="D:\eclipse3.4.1\WORKSPACE\EmployeeDirMT\src"/>
</eo-system>
```

Note that you should set `multitenancy` to true and configure the data source and java source folder accordingly.

Create Entities For The Application

Open Athena Console application:

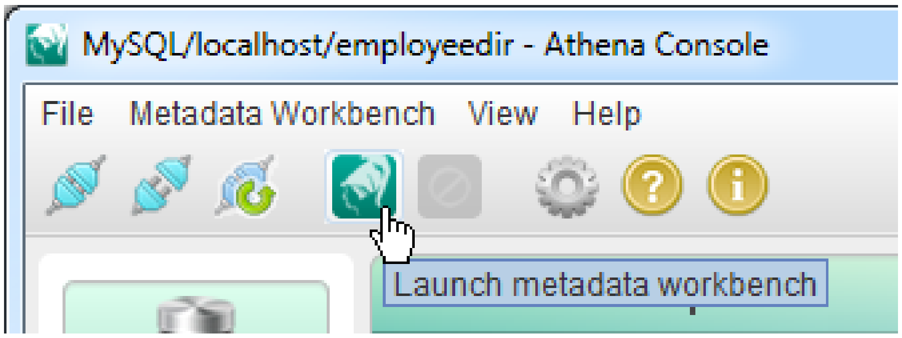
Initialize Metadata in the Database

1. Click File -> Open EO Config or press 'Ctrl + O' and browse to the `eo-config.xml` file in the project;
2. Skip this step if you are using non-MySQL database. Click 'Create database' if the database

does not exist;

3. Click the 'Initialize metadata' button to start the initialization process;
4. Click the 'Check database' button, and you should get 'Metadata tables found. Database is ready' message indicating metadata has been initialized successfully.

Once metadata has been initialized, you may proceed to create and update entities in the Metadata Workbench. To launch Metadata Workbench, you can simply click the toolbar button:


MySQL/localhost/employeedir - Athena Console
File
Metadata Workbench
View
Help
Launch metadata workbench

Create Entities

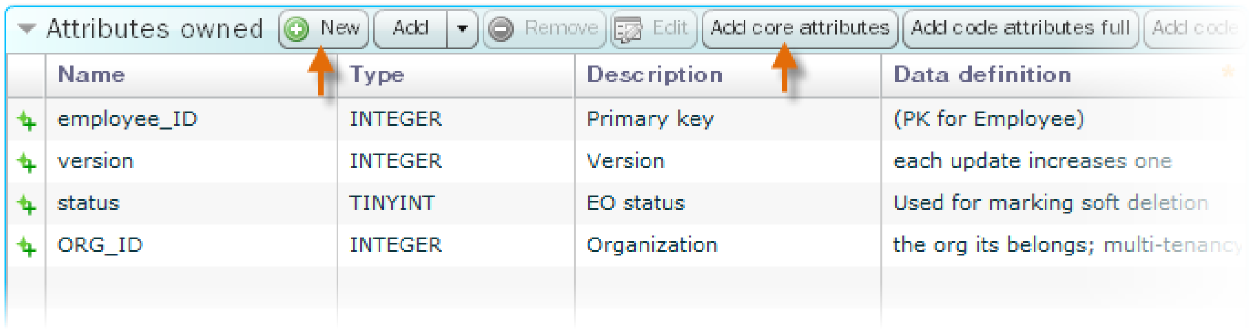
First, we need to create the `Employee` entity.

To create a new entity, you need to click on the 'New' button under the 'All Entities' tab. On the 'Create new Entity' page, enter the following values in the 'Properties' block:

- Entity ID: **101** (Entity IDs between 1 to 100 are reserved for system use only)
- System name: **Employee** (Entity's system name will be the class name too)
- Table name: **Data_Employee** (You may use any name permitted by the database)
- Package name: **com.test** (The package that the class belongs to; you may use any valid package name)
- Display name: **Employee** (A brief description of the entity)

To add attributes to the entity, you can either press 'New' button to create new attributes or use the 'Add ...' quick add tools to add common attributes in the 'Attributes owned' block.

Core attributes are required by the Athena Framework. Before adding normal attributes, you should press the 'Add core attributes' to add four core attributes for the entity:



	Name	Type	Description	Data definition
	employee_ID	INTEGER	Primary key	(PK for Employee)
	version	INTEGER	Version	each update increases one
	status	TINYINT	EO status	Used for marking soft deletion
	ORG_ID	INTEGER	Organization	the org its belongs; multi-tenancy

Core attributes:

- **x_ID** - Required. The first attribute must be an integer based primary key with auto increment.
- **version** - Required. The revision number is used by the unit of work to update obsoleted data for EOs.
- **status** - Recommended. If you plan to use soft deletion, this attribute must be present.
- **ORG_ID** - Recommended. If you plan to use multi-tenancy, this attribute must be present.

We can now add normal attributes to the `Employee` entity. We need to add an attribute to store the name of an employee. Press the 'New' button, and input the following values on the create new attribute page:

- Attribute ID: (please keep unchanged, handled by the wqorkbench automatically)
- Belonging entity ID: (please keep unchanged, handled by the workbench automatically)
- System name: **nameFull**
- Display name: **Full name**
- Column type: **VARCHAR**
- Precision: **200**

Generate Source Code

To generate source code for the entity created, select the 'Code Generation' tab in Athena Console and click the 'Generate classes' button. Once the code generation is done, refresh the project folder in your IDE, and you'll find the following two classes are generated in the project's source folder:

- `com.test.Employee` (Entity class)

• `com.test.generated.Employee_EO` (Member access class)

Coding JSP

The code for the core JSP page `main.jsp`:

```
<%
if(EOThreadLocal.getUserContext() == null) { // not in session, redirect to login page.
  response.sendRedirect("login.jsp");
  return;
} else {
  out.println("<a href='login.jsp?action=logout'>Logout</a> ");
  EOThreadLocal.setOrgId(EOThreadLocal.getUserContext().getUser().getOrgId()); // set org id.
  out.println("ORG ID: " + EOThreadLocal.getOrgId() + ", User: " + »
     EOThreadLocal.getUserContext().getUser().getNameFull());
}
%>
<%@page import="java.io.PrintWriter"%>
<%@page import="org.athenasource.framework.eo.core.UnitOfWork"%>
<%@page import="com.test.Employee"%>
<%@page import="java.util.List"%>
<%@page import="org.athenasource.framework.eo.query.EJBQLSelect"%>
<%@page import="org.athenasource.framework.eo.core.EOService"%>
<%@page import="org.athenasource.framework.eo.core.context.EOThreadLocal"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" »
 "http://www.w3.org/TR/html4/loose.dtd">

<%@page import="org.athenasource.framework.eo.core.EOObject"%>
<%@page import="com.test.generated.Employee_EO"%>
<%@page import="org.athenasource.framework.eo.core.EOContext"%><html>
<body style="font-family: arial; ">
<h2>Employee Directory</h2>

<p><a href="?action=LIST">List</a> | <a href="?action=FORM_CREATE">Create</a> </p>
<br>

<%

String action = request.getParameter("action");
if(action == null || action.trim().length() == 0) { // if no action specified, use 'LIST'.
  action = "LIST";
}

if("LIST".equalsIgnoreCase(action)) {

  EOService eoService = EOThreadLocal.getEOService();
  EJBQLSelect selectEmployees = eoService.createEOContext().createSelectQuery("SELECT e FROM Employee »
     e");
  List<Object> employees = selectEmployees.getResultList();

  out.println("<p>Total number of employees: " + employees.size() + "</p>");
  out.println("<ul>");
  for(int i=0; i < employees.size(); i++) {
    Employee employee = (Employee)employees.get(i);
    out.println("<li>" + employee.getNameFull() + ", born in " + employee.getBornYear());
    out.println(" <a href='?action=FORM_UPDATE&empid=" + employee.getEmployee_ID() + "'><font »
         size=-1>Update</font></a>");
    out.println(" <a href='?action=DELETE&empid=" + employee.getEmployee_ID() + "' »
         onClick=\"return confirm('Are you sure to delete this employee?')\"><font »
         size=-1>Delete</font></a>");
  }
  out.println("</ul>");

}else if("CREATE".equalsIgnoreCase(action)) {

  UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
  Employee newEmp = (Employee)uow.createNewInstance(Employee.SYSTEM_NAME);
  uow.persist(newEmp);
  try {
    newEmp.setNameFull(request.getParameter("fullname"));
    newEmp.setBornYear(Integer.parseInt(request.getParameter("bornyear")));
    uow.flush();
    uow.close();
    out.println("Employee created successfully. ID: " + newEmp.getEmployee_ID());
  }catch(Throwable t) {
    out.println("Failed to create employee due to exception: <pre>");
    t.printStackTrace(new PrintWriter(out));
    out.println("</pre>");
  }

}else if("UPDATE".equalsIgnoreCase(action)) {
  UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
  EJBQLSelect selectEmp = uow.createSelectQuery("SELECT e FROM Employee e WHERE e.employee_ID = ?1");
  selectEmp.setParameter(1, Integer.parseInt(request.getParameter("empid")));
  Employee emp = (Employee)selectEmp.getSingleResult();
  if(emp == null) {
    out.println("Employee not found, id: " + request.getParameter("empid"));
  }else{
    try {
      emp.setNameFull(request.getParameter("fullname"));
      emp.setBornYear(Integer.parseInt(request.getParameter("bornyear")));
      uow.flush();
      uow.close();
      out.println("Employee data updated successfully, id: " + emp.getEmployee_ID());
    }catch(Throwable t) {
      out.println("Failed to create employee due to exception: <pre>");
      t.printStackTrace(new PrintWriter(out));
      out.println("</pre>");
    }
  }

}else if("DELETE".equalsIgnoreCase(action)) { // delete
  UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();
  Employee emp = (Employee)uow.find(Employee_EO.SYSTEM_NAME, »
     Integer.parseInt(request.getParameter("empid")), null, null);
  if(emp == null) {
    out.println("Employee not found, id: " + request.getParameter("empid"));
  }else{
    try {
      uow.remove(emp);
      uow.flush();
      uow.close();
      out.println("Employee data deleted successfully, id: " + emp.getEmployee_ID());
    }catch(Throwable t) {
      out.println("Failed to delete employee due to exception: <pre>");
      t.printStackTrace(new PrintWriter(out));
      out.println("</pre>");
    }
  }

}else if("FORM_CREATE".equalsIgnoreCase(action)) { // display form for create
%>

<form action="">
<input type="hidden" name="action" value="CREATE" />
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_nameFull).getDisplayName() %>
: <input name="fullname" type="text" size="20" />
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_bornYear).getDisplayName() %>
: <input name="bornyear" type="text" size="4" />
<input type="submit" value="Create">
</form>

<%
} else if("FORM_UPDATE".equalsIgnoreCase(action)) { // display form for update
  Employee emp = null;

  EJBQLSelect selectEmp = EOThreadLocal.getEOService().createEOContext().createSelectQuery("SELECT e »
     FROM Employee e WHERE e.employee_ID = ?1");
  selectEmp.setParameter(1, Integer.parseInt(request.getParameter("empid")));
  emp = (Employee)selectEmp.getSingleResult();
%>

<form action="">
<input type="hidden" name="action" value="UPDATE" />
<input type="hidden" name="empid" value="<%= request.getParameter("empid") %>" />
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_nameFull).getDisplayName() %>
: <input name="fullname" type="text" size="20" value="<%= emp.getNameFull() %>"/>
<%= EOThreadLocal.getEOService().getEntity(Employee_EO.SYSTEM_NAME)
  .getAttributeBySystemName(Employee_EO.ATTR_bornYear).getDisplayName() %>
: <input name="bornyear" type="text" size="4" value="<%= emp.getBornYear() %>"/>
<input type="submit" value="Update">
</form>

<%
}else if(action == null || action.trim().length() == 0) {
  out.println("Welcome.");
} else {
  out.println("Unsupported action: " + action);
}

%>

<p align="left" style="padding-top: 20px;">
<hr style="width: 100%; color: #ccc; height: 1px;"/>
<a href="http://www.athenaframework.org" target='_blank'>
<img src="http://www.athenaframework.org/_img/logo/logo-poweredby.png" alt="Powered by Athena »
 Framework" align="left" hspace="0" vspace="1" border="0">
</a></p>
```

```
</body>
</html>
```

You might notice that the multitenancy version of `main.jsp` is the same as the non-multitenancy version except the code inserted at the top (highlighted in **bold** font).

The code at the top checks whether a user is logged in. If the user has logged in, the user's org id will be available and all the EJQBL querying and upating to the database in the page will use this org id. Otherwise, it will redirect to the login page.

At this moment, `main.jsp` is not working. We need to set up org and to create a login page.

8.3 Org, User Entities and Admin Page

In this section, we'll create two new entities: Org (represents an organization, i.e., a tenant) and User (represents a user in an org), and code a JSP page to maintain them.

Create Entity: Org

Open Athena Workbench, and create a new entity with the following values in the 'Properties' block:

- Entity ID: **201**
- System name: **Org** (Entity's system name will be the class name too)
- Table name: **Sys_Org** (You may use any name permitted by the database)
- Package name: **com.sys** (The package that the class belongs to; you may use any valid package name)
- Display name: **Org** (A brieft description of the entity)

Add Core Attributes

Now, press the 'Add core attributes' to add core attributes for the entity. You might notice the following four attributes are added: `org_ID`, `version`, `status`, and `ORG_ID`. We now have duplication. **Remove the last attribute `ORG_ID`** by selecting it and then clicking the 'Remove' button.

Add Another Attribute

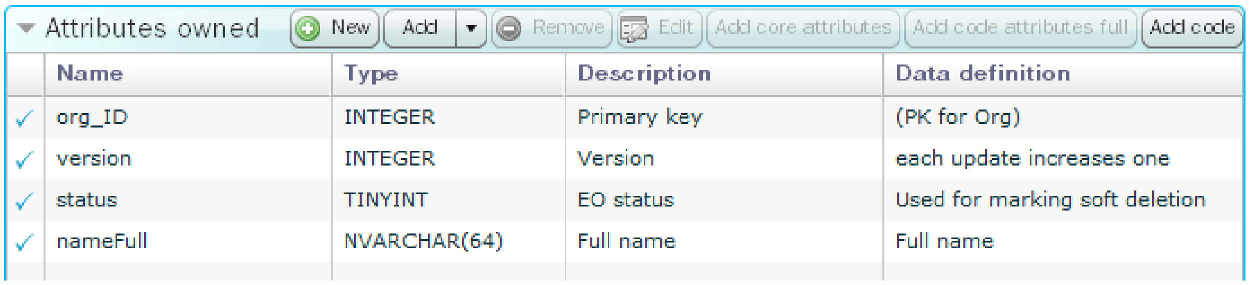
Click on the 'New' button on 'Attributes owned' panel, and create a new attribute to store the name of the org:

Table 8.1. Additional Attribute

System name	Display name	Column type	Precision
`nameFull`	`Full name`	`NVARCHAR`	`64`

`nameFull` - system name: `nameFull`, display name: `Full name`, column type: `NVARCHAR`, precision: `64` (leave values for all other properties unchanged). Click 'Save' button, and then click 'Cancel' to return to the entity page. Click 'Save' button at the end of the entity page to save the entity with attributes to the database.

The attributes of Org is illustrated as below:



	Name	Type	Description	Data definition
✓	org_ID	INTEGER	Primary key	(PK for Org)
✓	version	INTEGER	Version	each update increases one
✓	status	TINYINT	EO status	Used for marking soft deletion
✓	nameFull	NVARCHAR(64)	Full name	Full name

Create Entity: User

Open Athena Workbench, and create a new entity with the following values in the 'Properties' block:

- Entity ID: **202**
- System name: **User** (Entity's system name will be the class name too)
- Table name: **Sys_User** (You may use any name permitted by the database)
- Package name: **com.sys** (The package that the class belongs to; you may use any valid package name)
- Display name: **User** (A brief description of the entity)

Add Core Attributes

Now, press the 'Add core attributes' to add core attributes for the entity. You might notice the following four attributes are added: `user_ID`, `version`, `status`, and `ORG_ID`.

Add Other Attributes

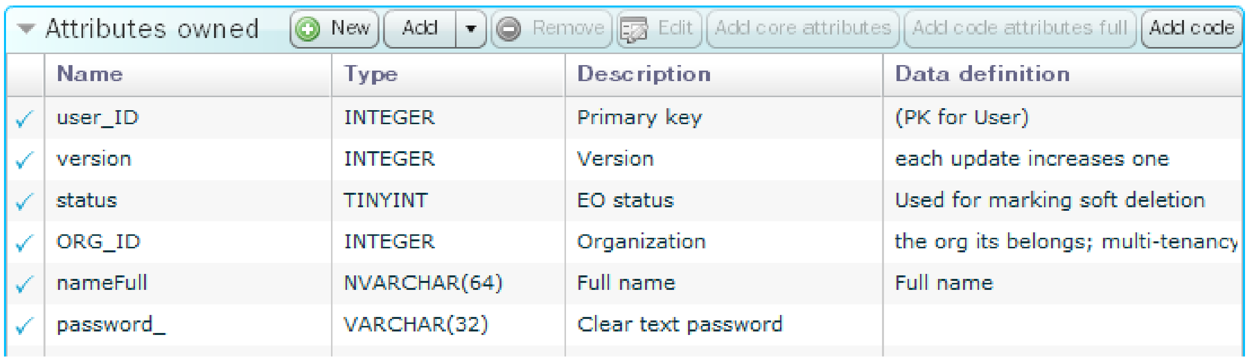
Click on the 'New' button on 'Attributes owned' panel, and create two new attributes to store the user name and password:

Table 8.2. Additional Attributes

System name	***Display name***	***Column type***	***Precision***
`nameFull`	`Full name`	`NVARCHAR`	`64`
`password_`	`Password`	`VARCHAR`	`32`

Click 'Save' button on the attribute editor page, and then click 'Cancel' to return to the entity page. Click 'Save' button at the end of the entity page to save the entity with its attributes to the database.

The attributes of `User` is illustrated as below:



	Name	Type	Description	Data definition
✓	user_ID	INTEGER	Primary key	(PK for User)
✓	version	INTEGER	Version	each update increases one
✓	status	TINYINT	EO status	Used for marking soft deletion
✓	ORG_ID	INTEGER	Organization	the org its belongs; multi-tenancy
✓	nameFull	NVARCHAR(64)	Full name	Full name
✓	password_	VARCHAR(32)	Clear text password	

Generating and Modifying Source Code

To generate classes for the entities, select the 'Code Generation' tab in Athena Console and click the 'Generate classes' button. Once the code generation is done, refresh the project folder in your IDE, and you'll find the following four classes are generated in the project's source folder:

- `com.sys.Org` (Org's entity class)
- `com.sys.generated.Org_EO` (Org's member access class)
- `com.sys.User` (User's entity class)
- `com.sys.generated.User_EO` (User's memeber access class)

In order to plug the two entities into the Athena Framework, `Org` needs to implement the `org.athenasource.framework.eo.core.baseclass.ext.IOrg` interface and `User` to implement `org.athenasource.framework.eo.core.baseclass.ext.IUser`. `IOrg` and `IUser` declare methods for Athena Framework to obtain the org id.

Org implements IOrg

Modify `Org.java` as following:

```
package com.sys;

import org.athenasource.framework.eo.core.baseclass.ext.IOrg;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.sys.generated.Org_EO;

// You may modify this file. It will not be overwritten in subsequent code generations.
/**
 * Represents a Org  - Organization
 */
public class Org extends Org_EO implements IOrg {

  static final Logger log = LoggerFactory.getLogger(Org.class);

  public Org() {
    super();
  }

  public int getOrgID() {
    return getOrg_ID();
  }

}
```

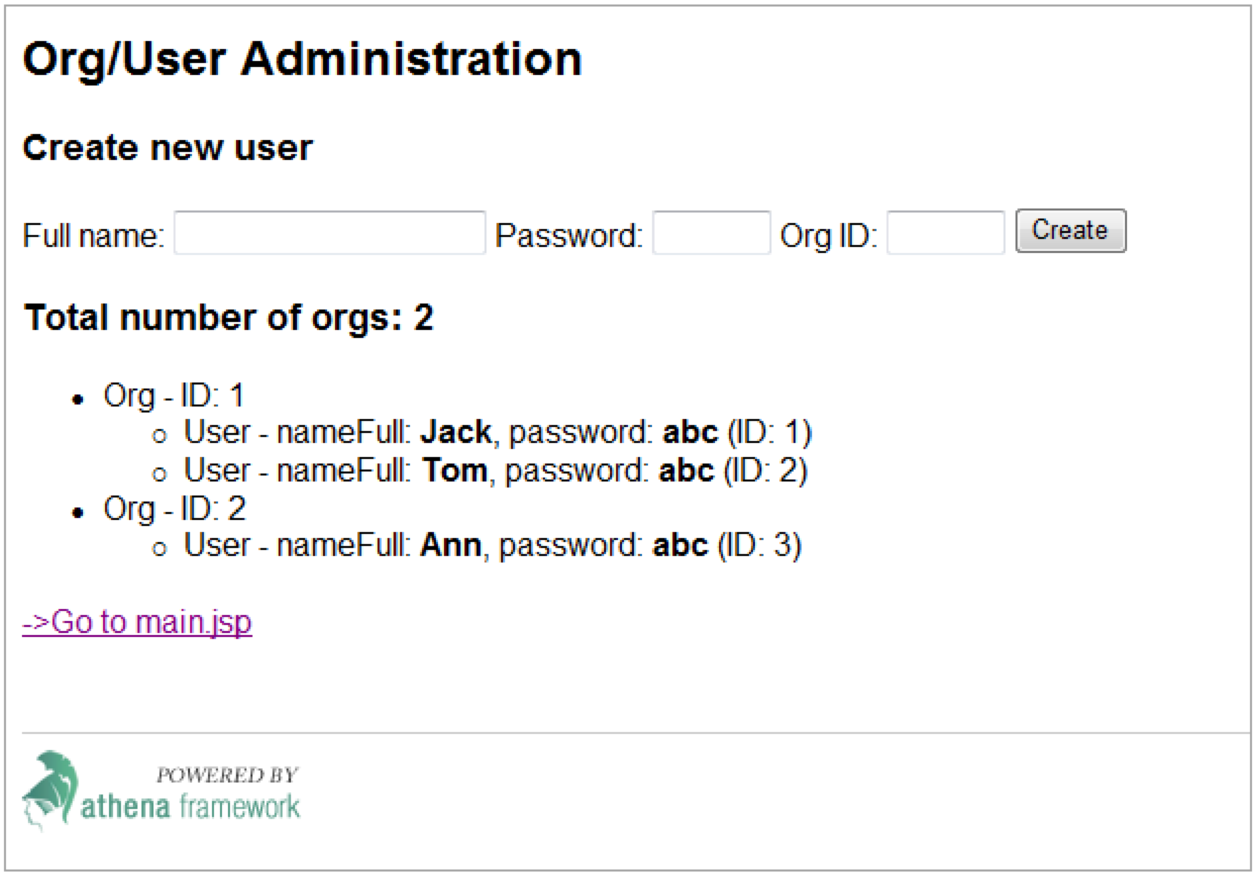
User implements IUser

Modify `User.java` as following:

```
package com.sys;

import org.athenasource.framework.eo.core.baseclass.ext.IUser;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.sys.generated.User_EO;

// You may modify this file. It will not be overwritten in subsequent code generations.
/**
 * Represents a User
 */
public class User extends User_EO implements IUser {

  static final Logger log = LoggerFactory.getLogger(User.class);

  public User() {
    super();
  }

   public int getOrgId() {
    return getORG_ID();
  }

  public int getOuId() {
    return 0;
  }

  public int getUserId() {
    return getUser_ID();
  }
}
```

Code the Admin JSP Page

We need to code an admin page in order to maintain orgs and users. The user interface of the admin page is illustrated below:



Org/User Administration

Create new user

Full name: Password: Org ID: Create

Total number of orgs: 2

- Org - ID: 1
  - User - nameFull: **Jack**, password: **abc** (ID: 1)
  - User - nameFull: **Tom**, password: **abc** (ID: 2)
- Org - ID: 2
  - User - nameFull: **Ann**, password: **abc** (ID: 3)

->Go to main.jsp

To create a new user, you input the name, password and the org id then click 'Create'. If the org with the given id does not exist, it will be created automatically.

The full source code of `admin.jsp`:

```
<%@page import="java.io.PrintWriter"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" »
 "http://www.w3.org/TR/html4/loose.dtd">

<%@page import="org.athenasource.framework.eo.core.EOService"%>
<%@page import="org.athenasource.framework.eo.core.context.EOThreadLocal"%>
<%@page import="org.athenasource.framework.eo.query.EJBQLSelect"%>
<%@page import="java.util.List"%>
<%@page import="com.sys.Org"%>
<%@page import="com.sys.User"%>
<%@page import="org.athenasource.framework.eo.core.UnitOfWork"%>
<%@page import="com.sys.generated.User_EO"%>
<%@page import="com.sys.generated.Org_EO"%><html>
<body style="font-family: arial; ">
<h2>Org/User Administration</h2>

<%
String action = request.getParameter("action");

if("CREATE".equalsIgnoreCase(action)) {
  try {
    // validate input parameters
    int orgId = Integer.parseInt(request.getParameter("orgid"));
    if(orgId <= 0) {
      throw new IllegalArgumentException("Org id must be greater than 0 - your input was: " + orgId);
    }

    String nameFull = request.getParameter("fullname");
    if(nameFull == null || nameFull.trim().length() == 0) {
      throw new IllegalArgumentException("Please provide the full name of the user");
    }
    String password = request.getParameter("password");
    if(password == null || password.trim().length() == 0) {
      throw new IllegalArgumentException("Please provide the password for the user");
    }

    UnitOfWork uow = EOThreadLocal.getEOService().createUnitOfWork();

    EJBQLSelect selectExistingOrg = uow.createSelectQuery("SELECT o FROM Org o WHERE o.org_ID = ?1 »
         {os_o='false'}");
    selectExistingOrg.setParameter(1, orgId);
    Org existingOrg = (Org) selectExistingOrg.getSingleResult();
    if(existingOrg == null) { // org with the given id does not exist, create it now.
      Org org = (Org) uow.createNewInstance(Org_EO.SYSTEM_NAME);
      org.setOrg_ID(orgId);
      org.setNameFull("<untitled>");
    }

    User user = (User) uow.createNewInstance(User_EO.SYSTEM_NAME);
    user.setNameFull(nameFull);
    user.setPassword_(password);
    user.setORG_ID(orgId);

    uow.flush();
    out.println("<h3 style='color: green;'>New user created successfully.</h3>");
  }catch(Throwable t) {
    out.println("<pre style='color: red'>");
    t.printStackTrace(new PrintWriter(out));
    out.println("</pre>");
  }
}

%>

<h3>Create new user</h3>
<form action="">
<input type="hidden" name="action" value="CREATE" />
<%= EOThreadLocal.getEOService().getEntity(User_EO.SYSTEM_NAME)
  .getAttributeBySystemName(User_EO.ATTR_nameFull).getDisplayName() %>
: <input name="fullname" type="text" size="20" />
<%= EOThreadLocal.getEOService().getEntity(User_EO.SYSTEM_NAME)
  .getAttributeBySystemName(User_EO.ATTR_password_).getDisplayName() %>
: <input name="password" type="text" size="4" />
 Org ID: <input name="orgid" type="text" size="4" />
<input type="submit" value="Create">
</form>

<%
// list orgs and users
EOService eoService = EOThreadLocal.getEOService();
EJBQLSelect selectOrgs = eoService.createEOContext().createSelectQuery("SELECT o FROM Org o ORDER BY »
 o.org_ID {os_o='false'}");
List<Object> orgs = selectOrgs.getResultList();

out.println("<h3>Total number of orgs: " + orgs.size() + "</h3>");
out.println("<ul>");
for(int i=0; i < orgs.size(); i++) {
  Org org = (Org)orgs.get(i);
  out.println("\n<li>Org - ID: " + org.getOrg_ID());
  out.println("\n<ul>");
  EJBQLSelect selectUsers = eoService.createEOContext().createSelectQuery("SELECT u FROM User u WHERE »
     u.ORG_ID = ?1 ORDER BY u.user_ID {os_u='false'}");
  selectUsers.setParameter(1, org.getOrg_ID());
  List<Object> users = selectUsers.getResultList();
  for(int u = 0; u < users.size(); u++) {
    User user = (User) users.get(u);
    out.println("\n<li>User - nameFull: <b>" + user.getNameFull() + "</b>, password: <b>" + »
         user.getPassword_() + "</b> (ID: " + user.getId() + ")</li>");
  }
  out.println("\n</ul></li>");
}
out.println("\n</ul>");

%>

 <a href='main.jsp'>-&gt;Go to main.jsp</a>

<p align="left" style="padding-top: 20px;">
<hr style="width: 100%; color: #ccc; height: 1px;"/>
<a href="http://www.athenaframework.org" target='_blank'>
<img src="http://www.athenaframework.org/_img/logo/logo-poweredby.png" alt="Powered by Athena »
 Framework" align="left" hspace="0" vspace="1" border="0">
</a></p>
</body>
</html>
```

There are a few points you should take note:

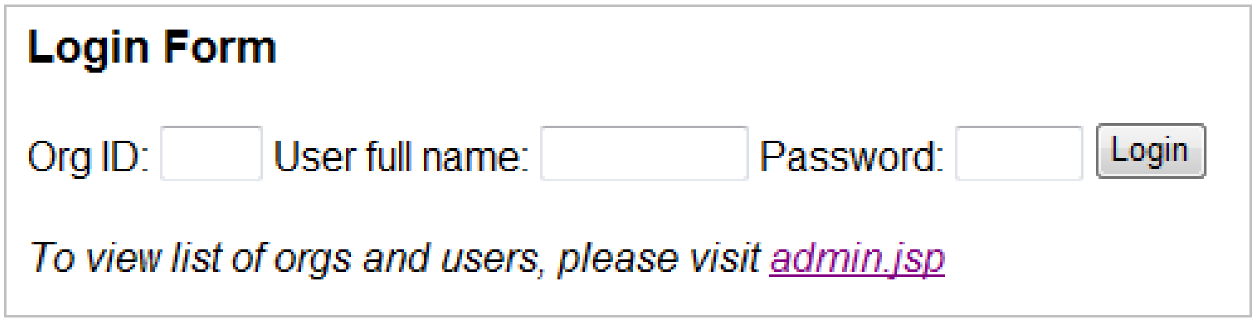
- **Execute org independent EJBQL query by specifying '{os_VARNAME='false'}.** To retrieve data across multiple orgs, you need set os (stands for Org-Specific) to false in EJBQL queries;
- **Explictly set org id by using eo.setORG_ID** (e.g. user.setORG_ID). In multitenant applications, an eo's `ORG_ID` attribute will be set automatically by the framework. To override this behavior, you can explicitly set the org id.

Use `admin.jsp` to add a few users and orgs and we are ready to login.

8.4 Login Page and UserContext Binding

The login page `login.jsp` should authenticate the user, create a `UserContext` object and bind it to session.

The user interface of the login page:


Login Form

Org ID: User full name: Password: Login

To view list of orgs and users, please visit admin.jsp

Full source code of `login.jsp`:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" »
 "http://www.w3.org/TR/html4/loose.dtd">
<%@page import="org.athenasource.framework.eo.query.EJBQLSelect"%>
<%@page import="org.athenasource.framework.eo.core.context.EOThreadLocal"%>
<%@page import="com.sys.User"%>
<%@page import="org.athenasource.framework.eo.core.context.UserContext"%>
<%@page import="org.athenasource.framework.eo.web.BindingConstants"%>
<%@page import="java.io.PrintWriter"%><html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Login</title>
</head>
<body style="font-family: arial; ">
<%
String action = request.getParameter("action");

try {
  if("LOGIN".equalsIgnoreCase(action)) {
    EJBQLSelect selectUser = EOThreadLocal.getEOService().createEOContext().createSelectQuery("SELECT u »
         FROM User u WHERE u.nameFull = ?1 AND u.password_ = ?2 AND u.ORG_ID = ?3 {os_u='false', »
         result_first='1', result_max='1'}");
    selectUser.setParameter(1, request.getParameter("fullname"));
    selectUser.setParameter(2, request.getParameter("password"));
    selectUser.setParameter(3, Integer.parseInt(request.getParameter("orgid")));
    User user = (User)selectUser.getSingleResult();
    if(user == null) { // unable to find the user record
      out.println("<h3 style='color: red'>No match for Org ID/Full name/password. Please try »
             again.</h3>");
    } else { // ok, login
      UserContext uc = new UserContext(user);
      session.setAttribute(BindingConstants.ATTRIBUTE_USERCONTEXT, uc);
      out.println("<h3 style='color: green'>User logged in to org (org id: " + user.getORG_ID() + ") »
             successfully.</h3>");
      out.println("<p>You may visit <a href='main.jsp'>main.jsp</a> now.</p>");
    }
  } else if("LOGOUT".equalsIgnoreCase(action)) { // logout.
    session.setAttribute(BindingConstants.ATTRIBUTE_USERCONTEXT, null);
    out.println("<h3 style='color: green'>User logged out successfully.</h3>");
  }
} catch(Throwable t) {
  out.println("<pre style='color: red'>");
  t.printStackTrace(new PrintWriter(out));
  out.println("</pre>");
}
%>
<h3>Login Form</h3>
<form action="">
<input type="hidden" name="action" value="login" />
Org ID: <input name="orgid" type="text" size="1" />
User full name: <input name="fullname" type="text" size="8" />
Password: <input name="password" type="password" size="4" />
<input type="submit" value="Login">
</form>
<p><i>To view list of orgs and users, please visit <a href='admin.jsp' »
 target=_blank>admin.jsp</a></i></p>
</body>
</html>
```

In above code, once the user is authenticated successfully, a `UserContext` object is created and bound to the context. In the user's subsequent requests to any page, the UserContext object will be available in `EOThreadLocal` thanks to `EOServletFilter`:

```
package org.athenasource.framework.eo.web;
...
public class EOServletFilter implements Filter {
    ...
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws »
     IOException, ServletException {
    HttpSession session = ((HttpServletRequest)request).getSession();
    ServletContext servletContext = session.getServletContext();
    // set thread context.
    EOThreadLocal.setEOService((EOService) »
         servletContext.getAttribute(BindingConstants.ATTRIBUTE_EOSERVICE));
     EOThreadLocal.setUserContext((UserContext) »
        session.getAttribute(BindingConstants.ATTRIBUTE_USERCONTEXT));
    EOThreadLocal.setHttpSession(session);
    EOThreadLocal.setServletRequest(request);

    chain.doFilter(request, response);
  }
    ...
}
```

When a EJBQL is executed, it will obtain the org id from `EOThreadLocal`. `UnitOfWork` also obtain user id from `EOThreadLocal` to set the `ORG_ID` attribute for any newly created objects.

Now, we have finished coding everything. You may access `main.jsp` to try out the employee directory application in multitenant mode.