Athena Framework Flex Developer's Guide

AthenaSource

Published Mar 2011

Table of Contents

1. Introduction to Athena Framework for Flex

1.1 Overview of Athena Framework for Flex

Athena Framework is a full fledged enterprise object-relational mapping (ORM) framework that employs metadata as mapping configuration. It greatly simplifies rich internet application (RIA) 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 seamless remote object persistence for Adobe Flex and Adobe AIR platforms. This document focuses on **developing Flex based RIA applications with Java back end using Athena Framework.**

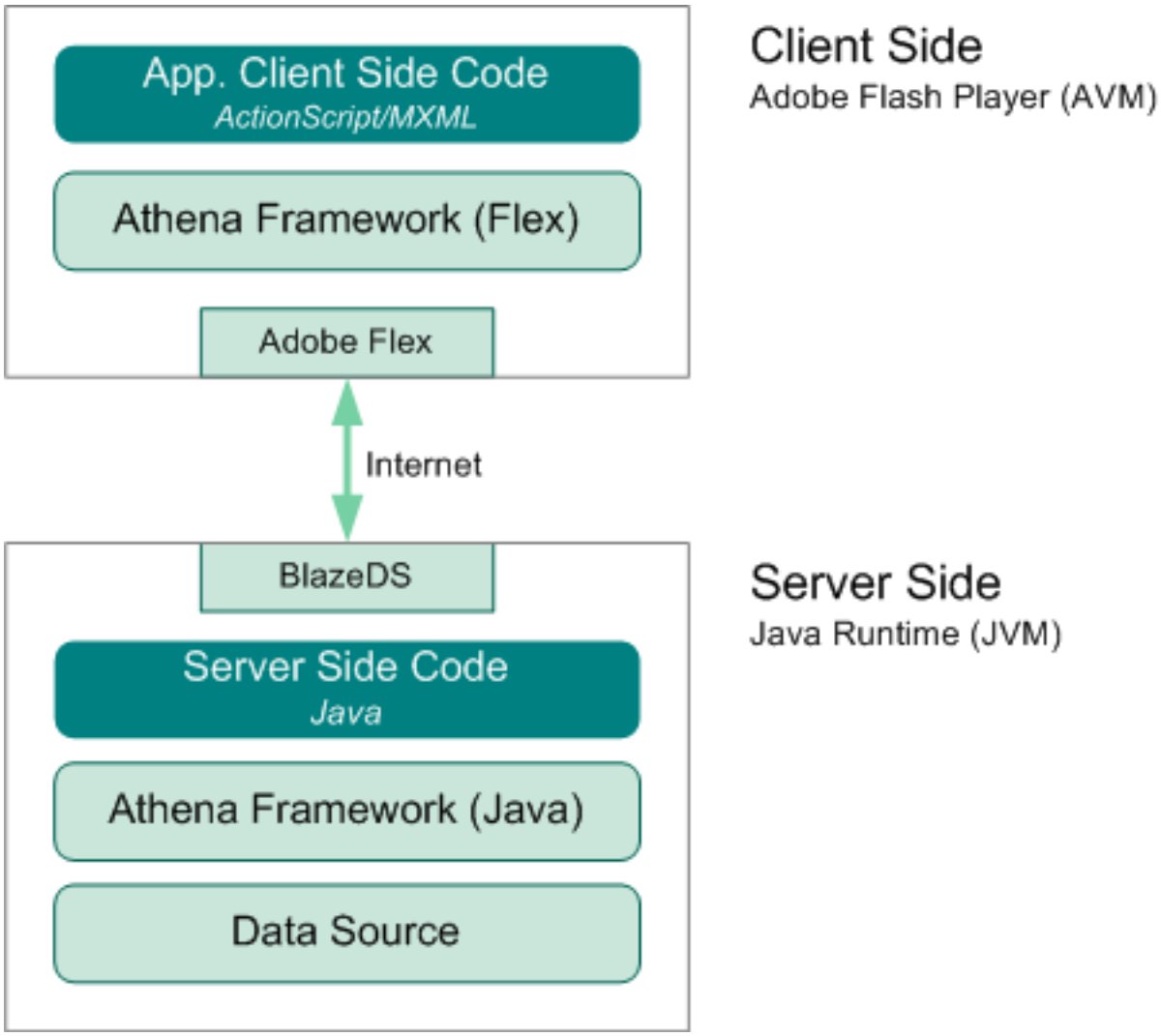
As Athena Framework for Flex has dependency on Athena Framework for Java, you are strongly encouraged to read *Athena Framework for Java* to get basic understanding of the framework first.

Important

Athena Framework for Java provides a detailed guide of Athena Framework. Make sure you read it first.

Architecture

The architecture of a typical Flex application based on Athena Framework is shown below.



Athena Framework uses BlazeDS and Adobe Flex platform to transfer objects between the server side and the client side through marshalling. Developers simply focus on implementing logics and manipulating objects rather than taking care of the low level object remoting details.

1.2 Additional Flex Only Features

Athena Framework for Java lists unique features of Athena Framework. Below are additional features of the framework that are applicable for Flex platform.

Unified Object Model on the Server and on the Client

Traditionally, developers need to develop two different sets of classes to represent the same object model for the server side and for the client side. Such mismatch results confusion and inefficiency. With Athena Framework, you get unified object model on the server and on the

client. During code generation, Athena Console generates entity classes for both Java and Flex. For an entity class, its Java class and Flex class are essentially the same - the only difference is the language syntax. To perform the same operation, you can either do it on the server side or on the client side. For example, to create a Department object and an Employee object, you may use the following code in Java on the server side:

```
Department dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
dept.setNameFull("R & D");
Employee emp = (Employee) uow.createNewInstance(Employee_EO.SYSTEM_NAME);
emp.setFirstName("Alan");
emp.setLastName("Turing");
emp.setDepartment(dept);

uow.flush();
```

Alternatively, you may creates the objects on the client side and remotely save them on the server side:

```
var dept:Department = Department.createNewInstance();
dept.nameFull = "R & D";
var emp:Employee = Employee.createNewInstance();
emp.firstName = "Alan";
emp.lastName = "Turing";
emp.department = dept;

eoService.invokeService("empService", "save", [emp], onSaveSuccess, onSaveError, null);

/** On employee saved success */
protected function onSavedSuccess(e:EventRemoteOperationSuccess):void {
  var savedEmployee:Employee = e.data as Employee;
  trace("Employee saved: ID: " + savedEmployee.employee_ID + ", department: " + »
   savedEmployee.department.nameFull);
}

// --- The corresponding server side Java code: ---
public class EmpService extends AbstractService {
  ...
  public Object saveEmployee(Employee employee) {
     return doPerisist(employee, false);
  }
}
```

In case of editing multiple related objects, manipulating objects on the client side will greatly simplifies the whole process.

Loading Objects through EJBQL to the Client Side

Loading objects through EJQBL on the client side is the same as on the server side except that loading on the client side is done asynchronously. Sample code:

```
var ejbql:String = "SELECT d FROM Department d ORDER BY d.nameFull";
eoService.invokeService("empService", "executeQuery", [ejbql], onQuerySuccess, onQueryError, null);

protected function onQuerySuccess(e:EventRemoteOperationSuccess):void { // listener (success)
  var depts:ArrayCollection = e.data as ArrayCollection;
  trace("Departments loaded, total: " + depts.length);
}

protected function onQueryError(e:EventRemoteOperationError):void { // listener (error)
  trace("Failed to load departments: " + e.message);
}
```

The corresponding server side code:

```
public class EmpService extends AbstractService {
  ...
  public List executeQuery(String ejbql) {
    return eoContext.createSelectQuery(ejbql).getResultList();
  }
  ...
}
```

Once the objects are loaded, you can manipulate them as you wish, e.g., binding them to the UI, modifying them, etc.

Partial Object Loading

Partial objects are objects with only some of their properties loaded. For example, we only need to load the name and other essential properties of the Employee class if we want to display a list of employees on the UI. Only the when user clicks on a certain Employee object, we then fully load the selected object. Partial object loading significantly reduces CPU, memory and network usage. Before diving into the details, you need to understand how *unit of work* works.

Unit of Work Basics

A Unit of Work keeps track of everything you do during a business transaction that can affect the database. `org.athenasource.framework.eo.core.UnitOfWork` is Athena's implementation of the Unit of Work pattern. `UnitOfWork` ensures uniqueness of `EOObject`. **Each database record results maximum one enterprise object in a `UnitOfWork`.**

By default, enterprise objects (instances of entity classes) returned from the server are being put into a `UnitOfWork`. If no `UnitOfWork` is specified when you make the remote call, a new instance of `UnitOfWork` is created for the enterprise objects returned. If the same database record is loaded again to a `UnitOfWork`, the existing corresponding enterprise object will be returned instead of creating a new object. Existing enterprise objects will updated if they are outdated (version lower than db). A partial enterprise object will be updated with full properties when a complete loading is performed.

Partial Object Loading Through EJBQL

To load partial objects, you simple execute EJBQLs with the special `po_` attribute loading properties. For example,

```
var uow:UnitOfWork = new UnitOfWork("myuow");
var ejbql:String = SELECT e FROM Employee e [e.department:S]{po_e='employee_ID, firstName, lastName, »
 department_ID'}";
eoService.invokeService("empService", "executeQuery", [ejbql], onQuerySuccess, onQueryError, uow); // »
 // returned objects will be merged into the specified unit of work.
```

Loading Full Object

If the user selects an employee to view the details, we need to load the full properties of the object:

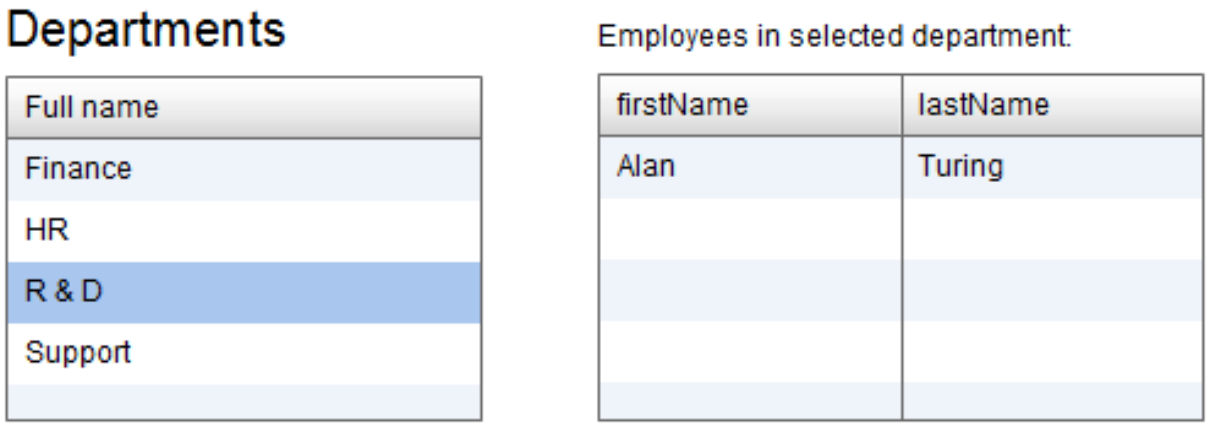
```
eoService.invokeService("empService", "executeQuery", ["SELECT e FROM Employee e WHERE e.employee_ID = »
 " + selectedEmp.employee_ID], onLoadFullEmpSuccess, null, uow); // specifies the same unit of work.
```

Once the remote call returns, a full `Employee` object is returned and merged into the existing `UnitOfWork`. Once all the properties are available, the user can see the full details of the employee.

Automatic Relationship Target Objects Loading

Relationship target objects can be loaded together with the source objects in EJBQL. For example, `"SELECT d FROM Department d [d.employees:S]"` loads all deparments with `Department.employees` relationship target objects (`Employees`). Suppose an object `deparment` with its `employees` relationships unresolved, any access to `deparment.employees` will trigger automatic resolution of the relationship target objects. On the server side, the resolution is performed synchronously. Due to the nature of the Flex application, the resolution is performed asynchronously.

For example, when the user selects a department from the data grid on the left, its employees will be listed on the other data grid through automatic relationship target object loading:



The corresponding code:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009" xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="600" »
   creationComplete="application1_creationCompleteHandler(event)">
  <fx:Script>
    <![CDATA[
      import com.test.Department;
      import com.test.Employee;
      import mx.collections.ArrayCollection;
      import mx.events.FlexEvent;
      import org.athenasource.framework.eo.core.EOService;
      import org.athenasource.framework.eo.core.ioc.EOServiceLocator;
      import org.athenasource.framework.eo.remoting.event.EventEOService;
      import org.athenasource.framework.eo.remoting.event.EventRemoteOperationSuccess;

      var eoService:EOService; // EO service

      [Bindable]
      public var departments:ArrayCollection = new ArrayCollection(); // all departments.

      protected function application1_creationCompleteHandler(event:FlexEvent):void {
        // Initialize eoService
        eoService = new EOService("http://localhost:8080/JavaEmployeeDir/messagebroker/amf", "eo", 2, »
                true, onEoServiceEvent);
        // Set Service Locator
        EOServiceLocator.getInstance().eoService = eoService;
      }

      protected function onEoServiceEvent(event:EventEOService):void { // Called when eo service is »
            // ready.
        if(event.kind == EventEOService.KIND_LOGIN_SUCCESS) {
          trace("Metadata loaded successfully.");
            trace("Loading employees and departments ...");
            eoService.invokeService("empService", "loadDepts", [], onLoadDeptsSuccess, null);
        }else if(event.kind == EventEOService.KIND_LOGIN_ERROR || event.kind == »
```

```
                EventEOService.KIND_META_LOAD_ERROR) {
          trace("Failed to load metadata: " + event.errorMessage);
        }
      }

      protected function onLoadDeptsSuccess(e:EventRemoteOperationSuccess):void {
        trace("Departments loaded successfully.");
        departments = e.data as ArrayCollection;;
      }
    ]]>
  </fx:Script>
  <mx:DataGrid x="10" y="37" id="gridDepts" dataProvider="{departments}" width="195" height="141">
    <mx:columns>
      <mx:DataGridColumn headerText="Full name" dataField="nameFull"/>
    </mx:columns>
  </mx:DataGrid>
  <mx:DataGrid x="252" y="36" id="gridEmps" dataProvider="{gridDepts.selectedItem.employees}" »
     width="248" height="142">
    <mx:columns>
      <mx:DataGridColumn dataField="firstName"/>
      <mx:DataGridColumn dataField="lastName"/>
    </mx:columns>
  </mx:DataGrid>
  <s:Label x="10" y="10" text="Departments" fontSize="20" width="302"/>
  <s:Label x="252" y="16" text="Employees in selected department:"/>
</s:Application>
```

Server side code:

```
public class EmpService extends AbstractService {
  public List loadDepts() {
    EOContext eoContext = createEOContext();
    return eoContext.createSelectQuery("SELECT d FROM Department d ORDER BY »
     d.nameFull").getResultList();
  }
}
```

When the user selects a department from the datagrid on the left, `gridDepts.selectedItem` is set. As the data provider of the data grid for displaying employee list is bound to `gridDepts.selectedItem.employees`, the `employees` relationship of the selected `Department` object is trigger to resolve. Once the target `Employee` objects of the relationship is returned from the server, they are displayed on the data grid.

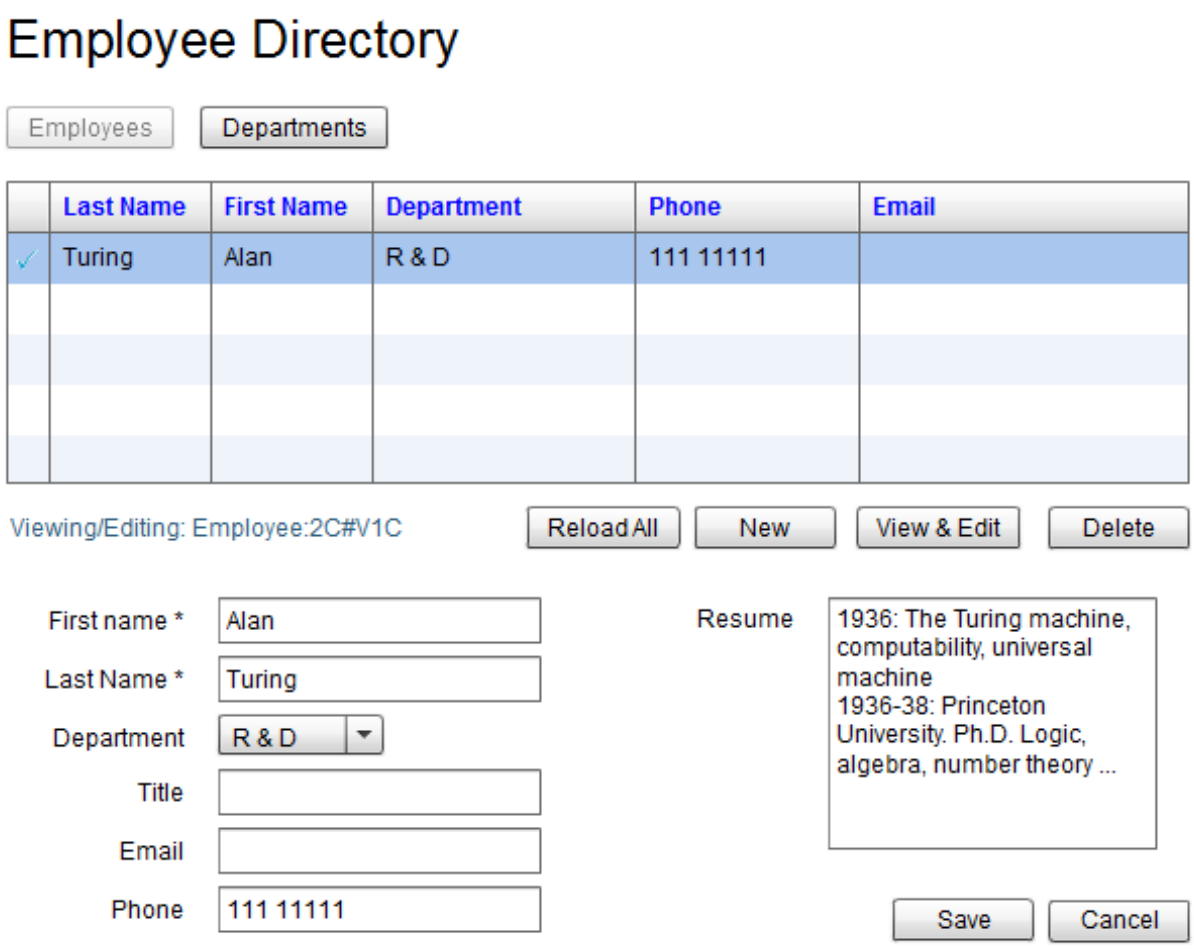
More on relationship resolution will be discuss in later chapters.

2. Get Started

This chapter will walk you through a tutorial to create your first Athena based Flex application.

2.1 Your First Athena Based Flex Application

We'll develop a simple CRUD application for managing employee information as shown below:



Functional Requirements

This application needs to comply the following requirements:

- Loads and displays all employee objects on the grid without "heavy" properties like `resume` which stores large text content (`CLOB`);

- Loads the full employee object with all properties when the user selects an employee and clicks the 'View & Edit' button;
- Saves the employee information to the server when the user clicks the 'Save' button;
- Deletes the employee from the server when the user hits the 'Delete' button.

To avoid complicated setup, we'll use the embedded database Derby in this application. Download it from http://db.apache.org/derby/derby_downloads.html , unzip it and copy `derby-10.x.y.jar` (x, y are the version numbers) to the `WEB-INF/lib` folder.

Alternatively, you may use any other database supported by Athena - MySQL, Oralce or DB2.

Note

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

2.2 Server Side Setup

First, create a Java web project using your favorite IDE.

Project Creation

Create a project and add Athena dependency:

Eclipse

Create a new project File -> New -> Dynamic Web Project, project name: `JavaEmployeeDir`. 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: `JavaEmployeeDir`. 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

Create a new project using maven with the following command:

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

  <!-- BlazeDS configuration starts -->
  <listener>
    <listener-class>flex.messaging.HttpFlexSession</listener-class>
  </listener>
  <servlet>
    <servlet-name>MessageBrokerServlet</servlet-name>
    <servlet-class>flex.messaging.MessageBrokerServlet</servlet-class>
    <init-param>
      <param-name>services.configuration.file</param-name>
      <param-value>/WEB-INF/flex-services-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>MessageBrokerServlet</servlet-name>
    <url-pattern>/messagebroker/*</url-pattern>
  </servlet-mapping>
  <!-- BlazeDS configuration ends -->

</web-app>
```

Note that in `web.xml` you need to add BlazeDS configuration in addition to Athena specific

configuration. `MessageBrokerServlet` performs the actual object transfer function. It requires a service configuration file: `flex-services-config.xml`.

Create flex-services-config.xml

`flex-services-config.xml` provides configuration for `MessageBrokerServlet`. Create a file named `flex-services-config.xml` under the `WEB-INF` folder as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <factories>
    <factory id="eo-factory" class="org.athenasource.framework.flex.FactoryForBlazeDS" />
  </factories>

  <services>
    <service id="remoting-service" class="flex.messaging.services.RemotingService">
      <adapters>
        <adapter-definition id="java-object" »
                class="flex.messaging.services.remoting.adapters.JavaAdapter" default="true" />
      </adapters>
      <default-channels>
        <channel ref="eo-amf" />
      </default-channels>
      <destination id="eo">
        <properties>
          <factory>eo-factory</factory>
        </properties>
      </destination>
    </service>
    <!-- Other services goes here ... -->
  </services>

  <security>
    <login-command class="flex.messaging.security.TomcatLoginCommand" server="Tomcat" />
    <!-- Uncomment the correct app server
      <login-command class="flex.messaging.security.TomcatLoginCommand" server="JBoss">
      <login-command class="flex.messaging.security.JRunLoginCommand" server="JRun"/>
      <login-command class="flex.messaging.security.WeblogicLoginCommand" server="Weblogic"/>
      <login-command class="flex.messaging.security.WebSphereLoginCommand" server="WebSphere"/>
    -->
  </security>

  <channels>
    <channel-definition id="eo-amf" class="mx.messaging.channels.AMFChannel">
      <endpoint url="/JavaEmployeeDir/messagebroker/amf" »
            class="flex.messaging.endpoints.AMFEndpoint" />
    </channel-definition>
    <!-- Other services goes here ... -->
  </channels>

  <logging>
    <!-- target class="flex.messaging.log.ConsoleTarget" level="INFO" --> <!--  must be INFO or »
         above, never ALL or DEBUG. -->
    <target class="org.athenasource.framework.flex.Log4JTargetForBlazeDS" level="INFO"> <!--  »
         must be INFO or above, never ALL or DEBUG. -->
      <properties>
        <prefix>BlazeDS.</prefix>
        <includeDate>false</includeDate>
        <includeTime>false</includeTime>
        <includeLevel>false</includeLevel>
        <includeCategory>true</includeCategory>
      </properties>
      <filters>
        <pattern>Client.*</pattern>
        <pattern>Endpoint.*</pattern>
        <pattern>Message.*</pattern>
        <pattern>Protocol.*</pattern>
        <pattern>Service.*</pattern>
        <pattern>Startup.*</pattern>
        <pattern>Configuration</pattern>
        <pattern>Resource</pattern>
        <pattern>Timeout</pattern>
      </filters>
    </target>
  </logging>

  <system>
    <redeploy>
      <enabled>false</enabled>
    </redeploy>
  </system>
```

```
</services-config>
```

The core purpose of `flex-services-config.xml` is to wire up Athena Framework and BlazeDS. To create other applications, you can simply copy the `flex-services-config.xml` file, and only need to update its `<endpoint>` element's `url` attribute (underlined in above config) to reflect the actual web application path.

Create eo-config.xml

`eo-config.xml` specifies the data source and code generation target folders for Athena Framework. 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="false">
  <datasources>
    <datasource>
      <database-type>Derby</database-type>
      <host>localhost</host>
      <port>-1</port> <!-- '-1' means using the default port -->
      <username>APP</username>
      <password/>
      <db>ABSOLUTE_PATH_TO\JavaEmployeeDir\empdb</db>
      <max-active>10</max-active>
      <max-idle>5</max-idle>
      <max-wait>5000</max-wait>
      <connection-timeout>300</connection-timeout>
    </datasource>
  </datasources>
    <services-declaration>webapp:/WEB-INF/eo-services.xml</services-declaration>
  <property name="java-source-local-dir" value="D:\eclipse\WORKSPACE\JavaEmployeeDir\src"/>
    <property name="flex-source-local-dir" value="D:\flash_builder\WORKSPACE\FlexEmployeeDir\src"/>
</eo-system>
```

Note you must set the following items correctly:

- `<db>` - the absolute path to a folder that Derby will store data. If your project folder is at `C:\projects\JavaEmployeeDir`, you may put `C:\projects\JavaEmployeeDir\empdb` - the folder needs not to exist as Derby can automatically create it.
- `java-source-local-dir` - the absolute path to the Java source folder;
- `flex-source-local-dir` - the absolute path to the Flex source folder. You may fill it after you create your Flex project.

The service declaration file `eo-services.xml` declares service classes and we'll create it later.

2.3 Entity Modeling

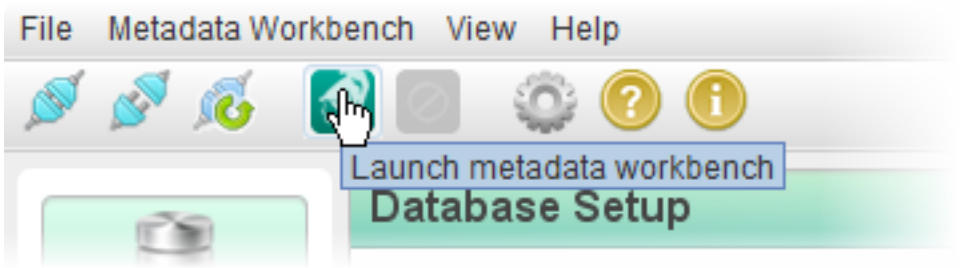
Now, we are ready to create our model.

Initialize Metadata in the Database

Open Athena Console,

1. Click File -> Open EO Config or press 'Ctrl + O' and browse to the `eo-config.xml` file in the Java web project;
2. Click the 'Initialize metadata' button to start the initialization process;
3. 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 or press Ctrl+W:



Create Entities

We need to create the two entities: `Department` and `Employee`.

Department

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

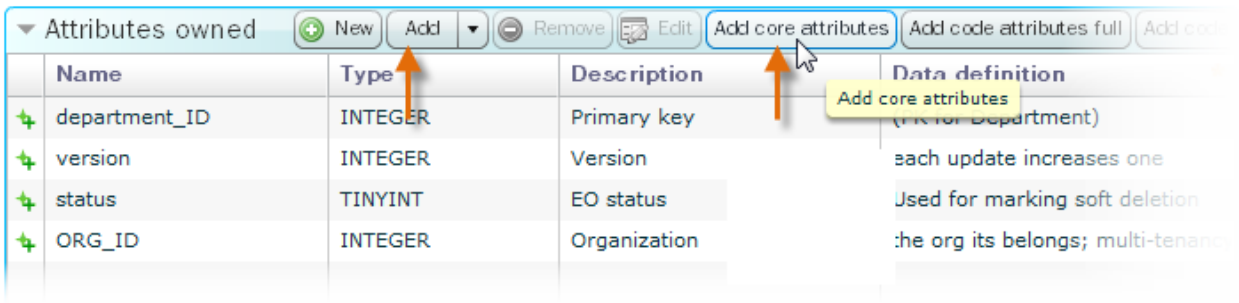
- Entity ID: **101** (Entity IDs between 1 to 100 are reserved for system use only)
- System name: **Department** (Entity's system name will be the class name too)
- Table name: **Data_Department** (You may use any name permitted by the database)
- Package name: **com.test** (The package that the class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

Now, we can add attributes for this 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:**



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 `Department` entity. We need to add an attribute to store the name of the department. 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: **NVARCHAR**
- Precision: **64**

Similarly, add the following attributes:

Table 2.1. Additional Attributes

System name	Display name	Column type	Precision
`description`	`Description`	`NVARCHAR`	`128`

Click the 'Save' button on the Edit Entity page to save the entity with attributes to the server.

Employee

Click on the 'New' button under the 'All Entities' tab of Athena Workbench. On the 'Create new Entity' page, enter the following values in the 'Properties' block:

- Entity ID: **102** (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 brieft description of the entity)

Press the 'Add core attributes' to add four core attributes for the entity.

Then, create the following attributes for the entity:

Table 2.2. Additional Attributes

System name	Display name	Column type	Precision
`firstName`	`First name`	`NVARCHAR`	`32`
`lastName`	`Last name`	`NVARCHAR`	`32`
`email`	`Email`	`VARCHAR`	`128`
`phone`	`Phone`	`VARCHAR`	`32`
`resume`	`Resume`	`CLOB`	`-1`
`department_ID`	`Deparment ID`	`INTEGER`	`-1`

Note that we create an attribute `department_ID` to store the ID of the department that this employee belongs to. Later, we'll use it to set up relationships.

Click the 'Save' button on the Edit Entity page to save the entity with attributes to the server.

Set up Relationships

We are now ready to set up the following relationships:

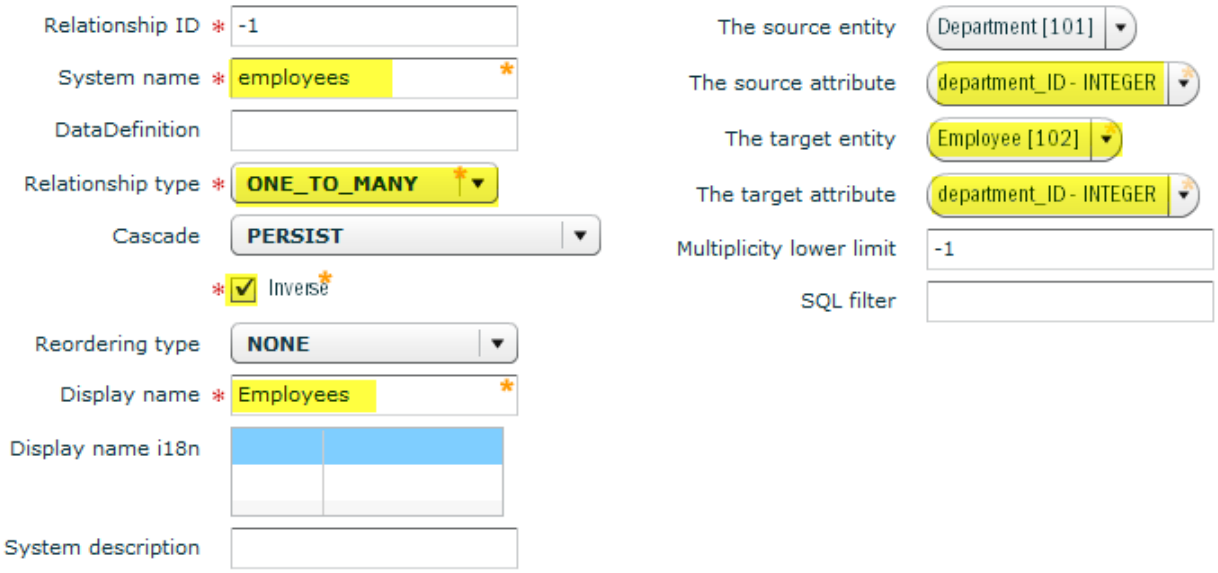
- `Department.employees` - a one-to-many relationship pointing from the `Deparment`

entity to `Employee`.

- `Employee.deparment` - a many-to-one relationship pointing from the `Employee` entity to `Deparment`;

Deparment.employees

Open Athena Metadata Workbench, double click the Department entity to go to the edit page. At the edit page, click the 'New' button on panel 'Relationships owned', and enter the following:



Click 'Save' to go back to the entity edit page. Click 'Save' button at the button of the page to save the entity with the relationship you just created.

Employee.department

Open Athena Metadata Workbench, double click the Employee entity to go to the edit page. At the edit page, click the 'New' button on panel 'Relationships owned', and enter the following:

Table 2.3. Relationship: Employee.department

Property	Value
System name	`department`
Relationship type	`MANY_TO_ONE`
Cascade	`PERSIST`
Inverse	`false` (unchecked)
Display name	`Department`

Property	Value
Source entity	`Employee`
Source attribute	`department_ID`
Target entity	`Department`
Target attribute	`department_ID`

Click 'Save' to go back to the entity edit page. Click 'Save' button at the button of the page to save the entity with the relationship you just created.

2.4 Server Side Coding

The model has been created. Now you can generate Java source code and start coding the server side.

Generate Source Code

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 four classes are generated in the project's source folder:

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

Create the Service Class

A service class extends `org.athenasource.framework.eo.web.service.AbstractService` and its methods can be invoked directly from Flex at the client side.

In this application, `EmpService` (in default package) is the only service class:

```
import java.util.List;

import org.apache.log4j.Logger;
import org.athenasource.framework.eo.core.EOContext;
import org.athenasource.framework.eo.core.UnitOfWork;
import org.athenasource.framework.eo.query.EJBQLSelect;
import org.athenasource.framework.eo.web.service.AbstractService;

import com.test.Department;
import com.test.Employee;
```

```
import com.test.generated.Department_EO;
import com.test.generated.Employee_EO;

public class EmpService extends AbstractService {
  private static final Logger log = Logger.getLogger(EmpService.class);

  /**
   * Load all employees and departments(All employee object is artial).
   * @return an array containg two elements: list of employees and list of departments.
   */
  public Object[] loadData() {
    EOContext eoContext = createEOContext();

    EJBQLSelect select = eoContext.createSelectQuery("SELECT e FROM Employee e »
         [e.department:S]{po_e='employee_ID, firstName, lastName, email, phone, department_ID'}");
    List<Object> listOfEmployees = select.getResultList();

    select = eoContext.createSelectQuery("SELECT d FROM Department d ORDER BY d.nameFull");
    List<Object> listOfDepts = select.getResultList();

    if(listOfDepts.size() == 0) { // Nothing in db yet, fill sample data.
      UnitOfWork uow = createUow();
      Department dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("Finance");
      dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("Support");
      dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("HR");
      dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("R & D");

      Employee emp = (Employee) uow.createNewInstance(Employee_EO.SYSTEM_NAME);
      emp.setFirstName("Alan");
      emp.setLastName("Turing");
      emp.setPhone("111 11111");
      emp.setResume("1936: The Turing machine, computability, universal machine\n" +
        "1936-38: Princeton University. Ph.D. Logic, algebra, number theory ...");
      emp.setDepartment(dept);

      uow.flush();
      uow.close();
      // reload now
      listOfDepts = eoContext.createSelectQuery("SELECT d FROM Department d ORDER BY »
            d.nameFull").getResultList();
      listOfEmployees = eoContext.createSelectQuery("SELECT e FROM Employee e »
            [e.department:S]{po_e='employee_ID, firstName, lastName, email, phone, »
            department_ID'}").getResultList();
    }

    return new Object[] {listOfEmployees, listOfDepts};
  }

  /** Get employee full object. */
  public Object loadFullEmpObject(int empID) {
    String sql = "SELECT e FROM Employee e WHERE e.employee_ID = " + empID;

    EJBQLSelect select = new EJBQLSelect(createEOContext(), sql);
    return select.getSingleResult();
  }

  public Object saveEmployee(Employee employee) {
    Object saveEmployee = doPerisist(employee, false);
    log.info("Employee saved: " + employee);
    return saveEmployee;
  }

  public Object removeEmployee(Employee employee) {
    Object removedEmployee = doPerisist(employee, true);
    log.info("Employee removed: " + employee);
    return removedEmployee;
  }
}
```

Once the service class is created, we need to declare it in `eo-services.xml`.

Declare Service Classes in eo-services.xml

There is a `services-declaration` entry in `eo-config.xml` which points to the service declaration xml file:

```
<services-declaration>webapp:/WEB-INF/eo-services.xml</services-declaration>
```

`eo-services.xml` contains all service classes that can be invoked directly from the Flex at the client side. Our `eo-services.xml` for this application is listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<services version="1.0">
  <!-- Common services -->
  <service name="meta" class="org.athenasource.framework.eo.web.service.MetadataService" »
     description="provides metadata service for applications" />
  <service name="commonEo" class="org.athenasource.framework.eo.web.service.CommonEoService" »
     description="common eo services for applications" />

  <!-- Application specific services -->
  <service name="empService" class="EmpService" description="Employee Service" />
</services>
```

Notice that there are two common services declared along with `empService`. Common services provide metadata loading, default relationship resolution and other utility functions to the Flex client side.

Start Your Server

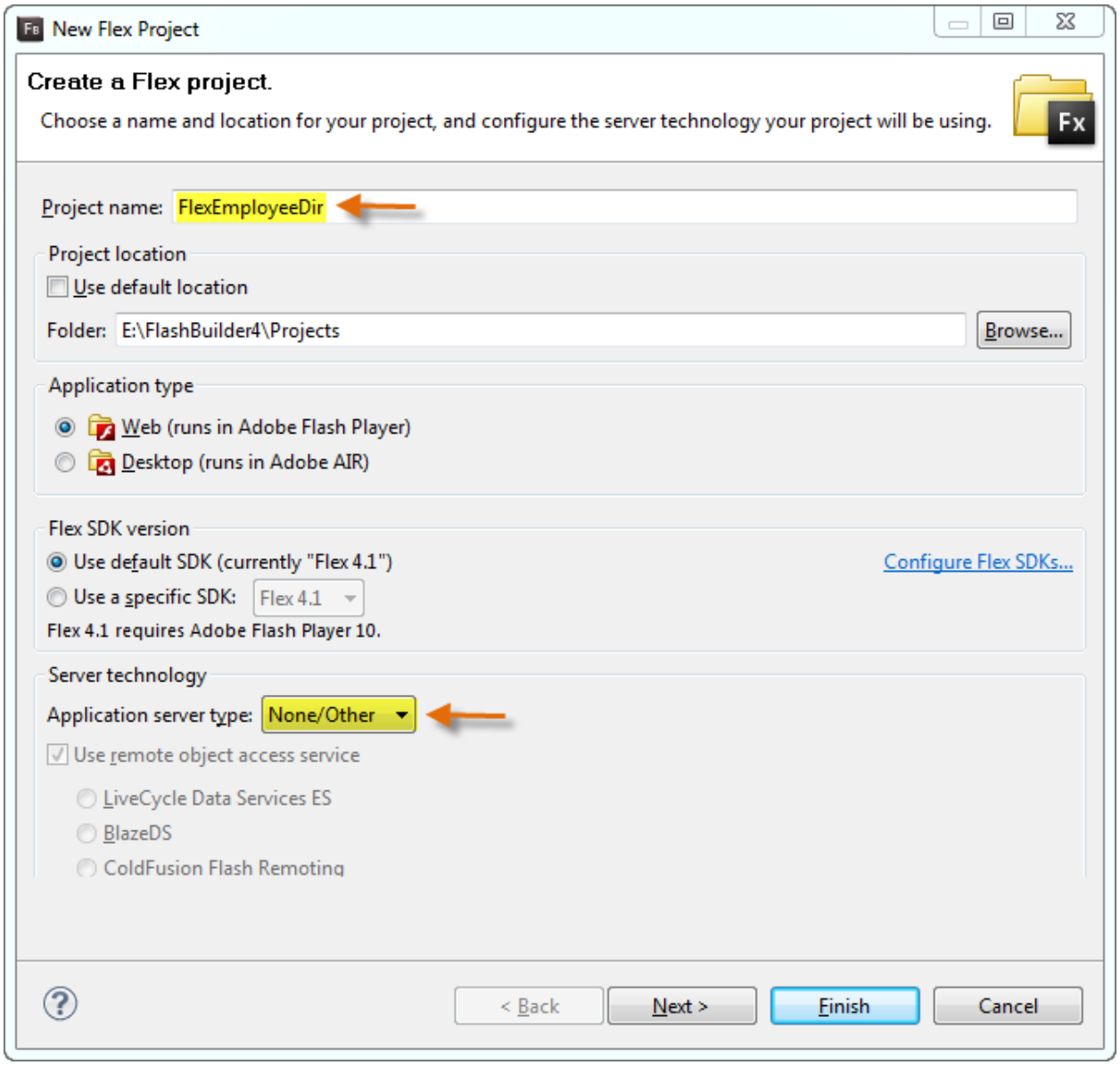
Before starting your server, you should close Athena Console if your database is Derby. Only one application is allowed to access Derby database in embedded mode.

2.5 Client Side Setup

First, create a Flex project using Adobe Flash Builder.

Create a Flex Project

Open Adobe Flash Builder, and create a Flex project as following:



You might notice that we **leave the 'Application server type' as 'None/Other'**. This is because we want to programmatically control the connections to the server side. Some of the benefits are:

- Complicated setup is avoided
- You have total control over the connections to the server side - when to connect/disconnect; and how many **number of concurrent connections** to be used
- You may **connect to multiple servers** instead of only one.

Add Athena Dependency

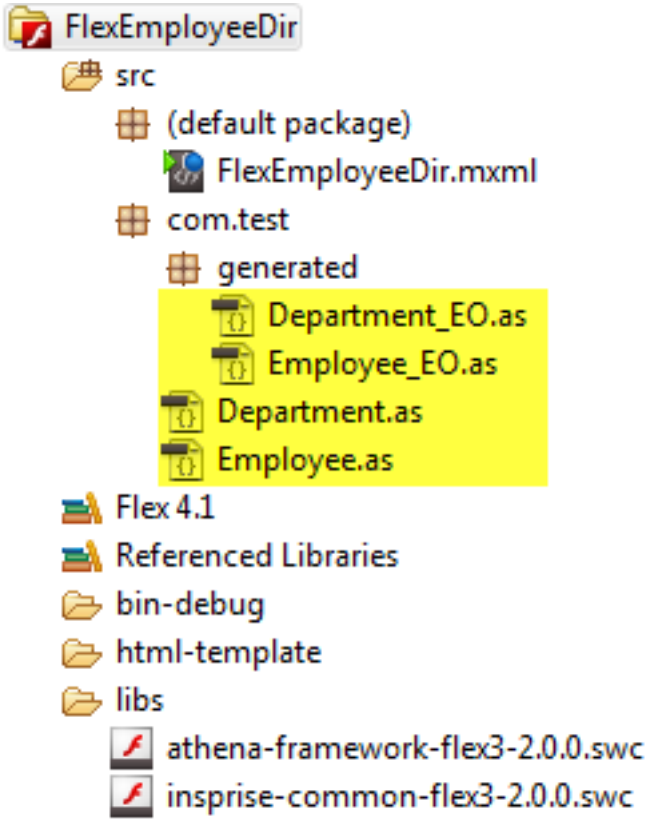
Copy the SWC files from the root folder of the Athena distribution to the `libs` folder of the newly created project and you are ready to code.

Generate Source Code

Once the project is created, Flash Builder creates a source folder, usually located at PROJECT_ROOT/src. You need to update the `flex-source-local-dir` property element in the Java web project's `WEB-INF/eo-config`:

```
<property name="flex-source-local-dir" value="ABSOLUTE_PATH_TO\FlexEmployeeDir\src"/>
```

Stop the Java server if it is running and open 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 four classes are generated in the project's source folder:



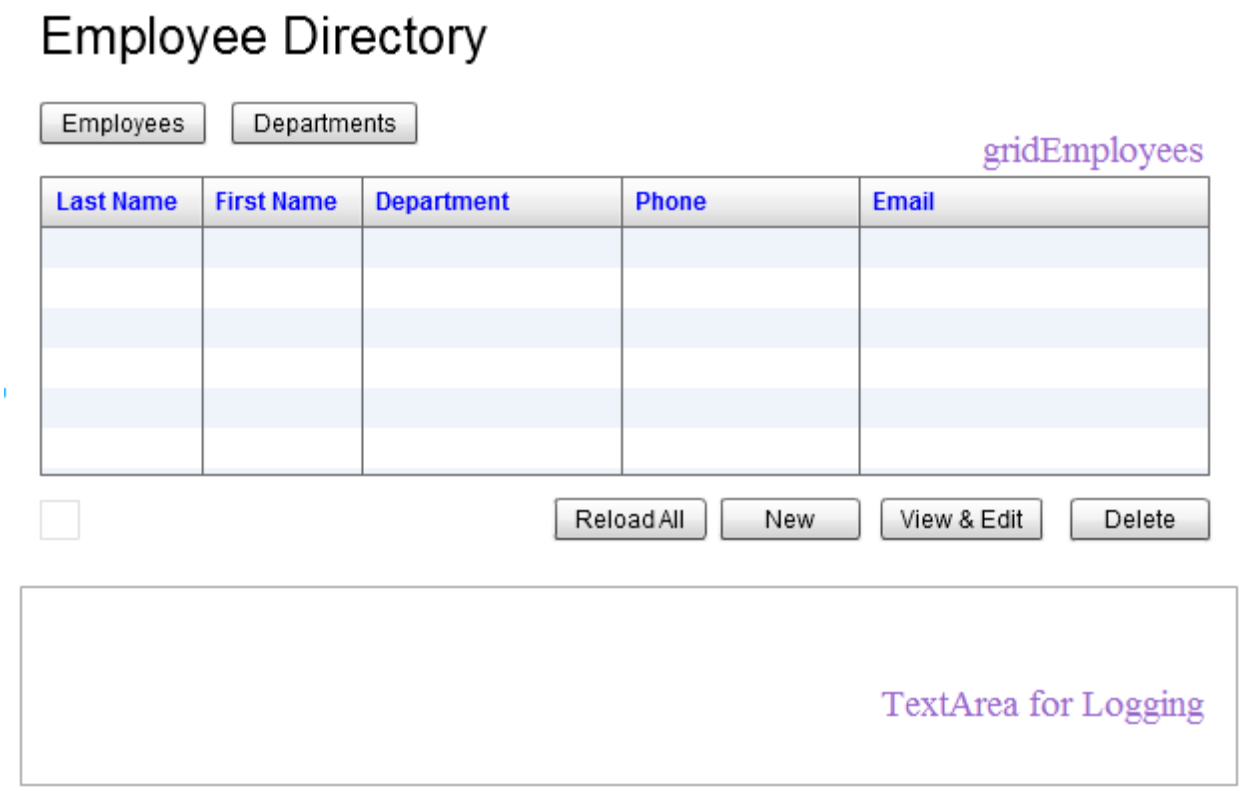
With generated source code in place, we can start to code our application.

2.6 Designing the User Interface

Open `FlexEmployeeDir.mxml` in design mode, and create four states:

StateEmpList (start)

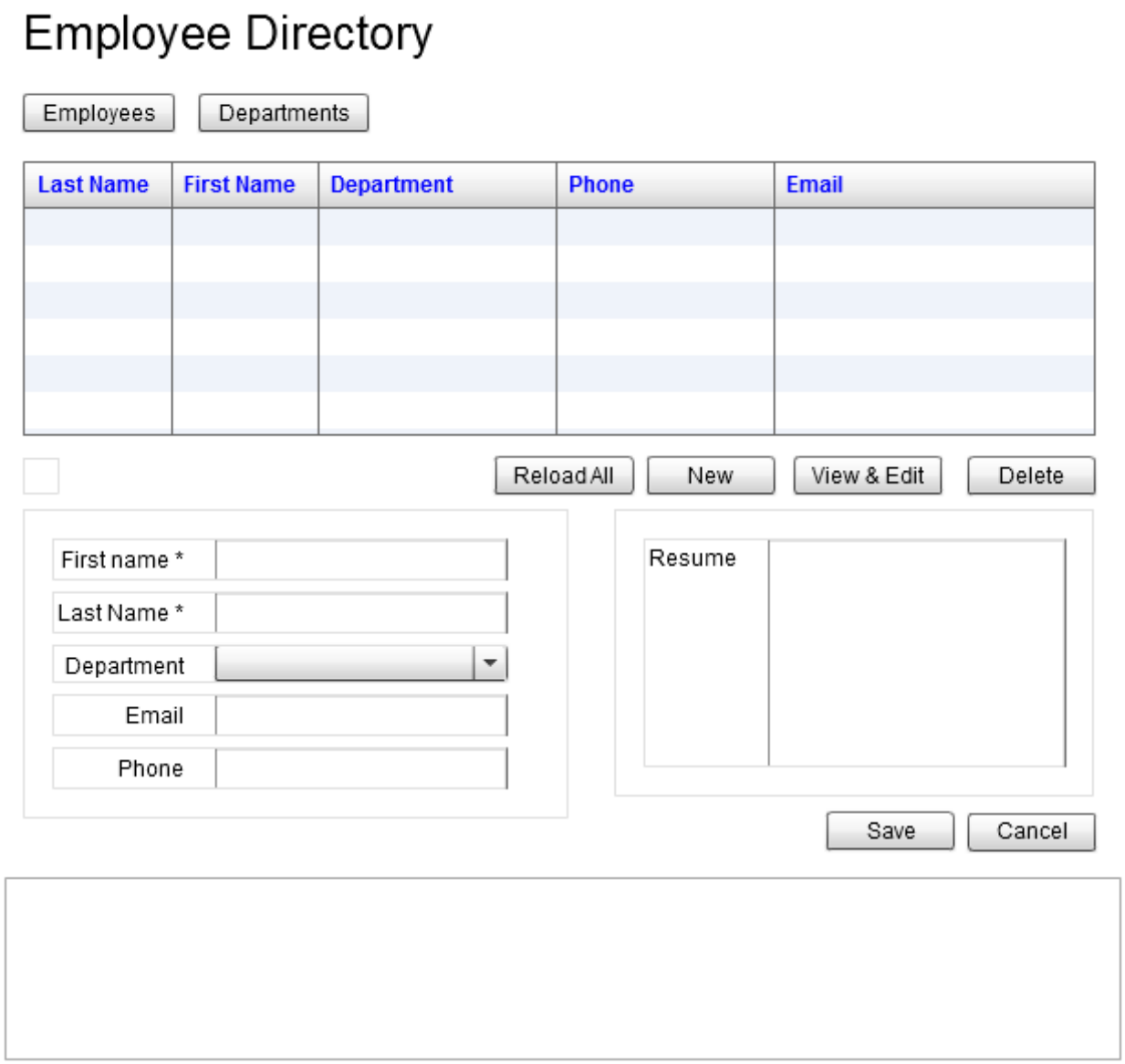
`StateEmpList` is the start state, which displays all the employees in the datagrid as shown below:



A text area for displaying logs is available at the bottom of the UI for all states. It records major actions and events.

StateEmpListWithDetails

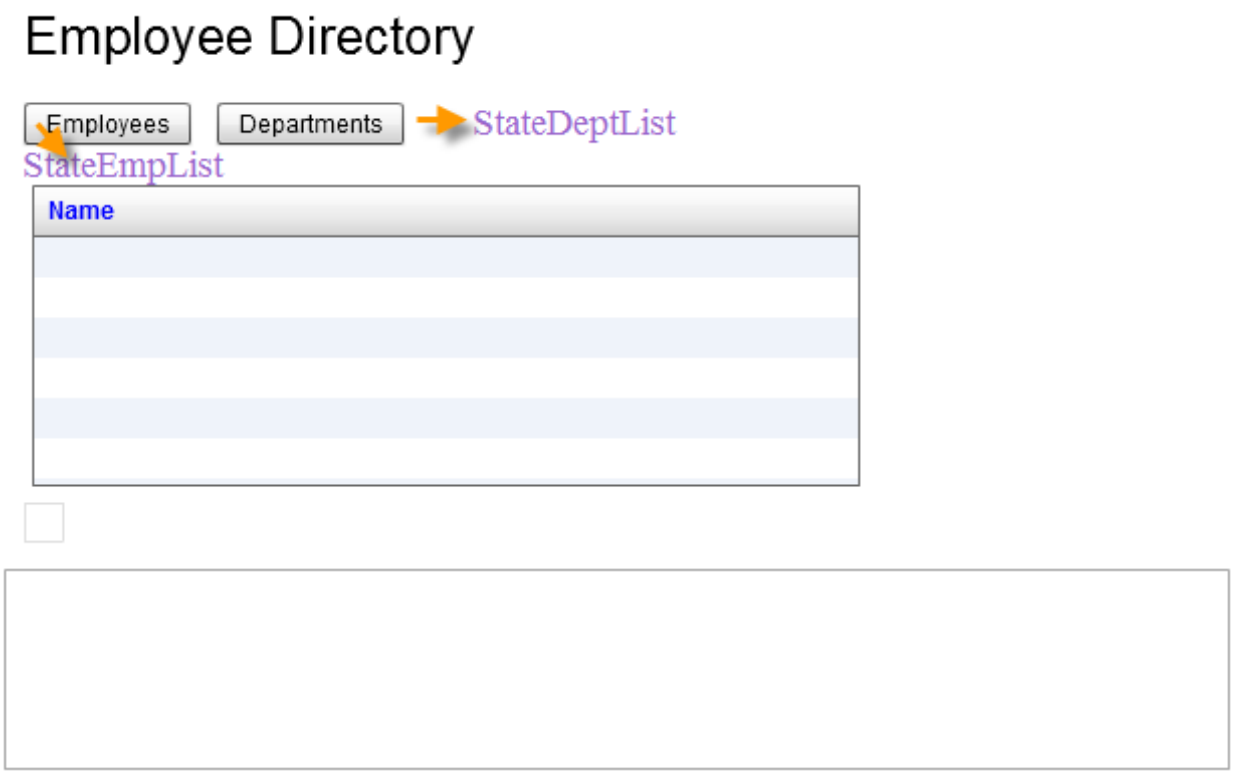
When user clicks 'New' to create a new employee or selects an employ to 'View/Edit', the UI will transit from `StateEmpList` to `StateEmpListWithDetails` as shown below:



`StateEmpListWithDetails` adds employee form and two buttons over `StateEmpList`.

StateDeptList

At at state, when the user clicks on the 'Departments' button at the top, the UI will switch to `StateDeptList`. `StateDeptList` displays all the departments in the data grid as shown below:



When the user clicks on one of the departments on the grid, the UI switches to `StateDeptListWithEmps`.

StateDeptListWithEmps

`StateDeptListWithEmps` displays employees in the selected department as shown below:

UI Code

The corresponding user interface code for the application:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- $Id$ Copyright (c) 2011 athenaframework.org -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009" xmlns:s="library://ns.adobe.com/flex/spark" »
 xmlns:mx="library://ns.adobe.com/flex/mx"
  creationPolicy="all" currentState="StateEmpList" width="630" height="600" height.StateEmpList="414" »
   height.StateDeptList="404"
  creationComplete="onCreationComplete(event)" currentStateChange="onCurrentStateChange(event)">
  <fx:Script>
    <![CDATA[
      ...
      /** EOService */
      protected var eoService:EOService;

      /** Data  */
      [Bindable]
      protected var employeeList:ArrayCollection = new ArrayCollection(); // The employees
      [Bindable]
```

```
      protected var departmentList:ArrayCollection = new ArrayCollection(); // The departments
      ...
    ]]>
  </fx:Script>
  <s:states>
    <s:State name="StateEmpList"/>
    <s:State name="StateEmpListWithDetails"/>
    <s:State name="StateDeptList"/>
    <s:State name="StateDeptListWithEmps"/>
  </s:states>
  <mx:DataGrid id="gridEmployees" dataProvider="{employeeList}" »
     includeIn="StateEmpListWithDetails,StateEmpList" doubleClickEnabled="true" x="20" y="99" »
     height="150" width="586">
    <mx:columns>
      <mx:DataGridColumn dataField="lastName" headerText="Last Name" width="80"/>
      <mx:DataGridColumn dataField="firstName" headerText="First Name" width="80"/>
      <mx:DataGridColumn headerText="Department" dataField="department" »
            labelFunction="lableFunForDGEmployees" width="130"/>
      <mx:DataGridColumn dataField="phone" headerText="Phone"/>
      <mx:DataGridColumn dataField="email" headerText="Email"/>
    </mx:columns>
  </mx:DataGrid>
  <s:Button id="buttonReloadData" click="onButtonClick(event)" »
     includeIn="StateEmpListWithDetails,StateEmpList" label="Reload All" x="277.55" y="260.05" />
  <s:Button id="buttonEmpEdit" click="onButtonClick(event)" label="View &amp;amp; Edit" »
     enabled="true" includeIn="StateEmpListWithDetails,StateEmpList" x="441.15" y="260.05" />
  <s:Button id="buttonEmpDelete" click="onButtonClick(event)" label="Delete" »
     includeIn="StateEmpListWithDetails,StateEmpList" x="535.95" y="260.3"/>
  <mx:Text id="textInfo" color="#32607E" x="19.9" y="261.05"/>
  <mx:Form includeIn="StateEmpListWithDetails" x="20.8" y="289" width="298">
    <mx:FormItem label="First name *">
      <mx:TextInput id="textFirstName"/>
    </mx:FormItem>
    <mx:FormItem label="Last Name *">
      <mx:TextInput id="textLastName"/>
    </mx:FormItem>
    <mx:FormItem label="Department">
      <mx:ComboBox id="comboDepartment" dataProvider="{departmentList}" labelField="nameFull" »
            editable="false" includeIn="StateEmpListWithDetails" itemCreationPolicy="immediate" »
            ></mx:ComboBox>
    </mx:FormItem>
    <mx:FormItem label="Email">
      <mx:TextInput id="textEmail" />
    </mx:FormItem>
    <mx:FormItem label="Phone">
      <mx:TextInput id="textPhone" />
    </mx:FormItem>
  </mx:Form>
  <mx:Form includeIn="StateEmpListWithDetails" x="343.8" y="289" width="262" height="157">
    <mx:FormItem label="Resume" width="100%" height="100%">
      <s:TextArea width="100%" height="100%" id="textResume"/>
    </mx:FormItem>
  </mx:Form>
  <mx:Button includeIn="StateEmpListWithDetails" x="458.75" y="454.15" label="Save" »
     id="buttonEmpSave" click="onButtonClick(event)"/>
  <s:Button includeIn="StateEmpListWithDetails" x="536.2" y="454.7" label="Cancel" »
     id="buttonEmpCancel" click="onButtonClick(event)"/>
  <s:Button label="Employees" id="buttonSwitchToEmpList" click="onButtonClick(event)" x="20" y="62" »
     />
  <s:Button label="Departments" id="buttonSwitchToDeptList" click="onButtonClick(event)" x="116" »
     y="62" />
  <mx:DataGrid id="gridDepartments" dataProvider="{departmentList}" »
     includeIn="StateDeptListWithEmps,StateDeptList" itemCreationPolicy="immediate" x="24" y="103" »
     width="412" height="150">
    <mx:columns>
      <mx:DataGridColumn headerText="Name" dataField="nameFull"/>
    </mx:columns>
  </mx:DataGrid>
  <s:Button includeIn="StateEmpListWithDetails,StateEmpList" label="New" id="buttonEmpNew" »
     click="onButtonClick(event)" x="360.85" y="260.05" />
  <mx:DataGrid x="24" y="304.15" width="412" id="gridDeptEmployees" »
     includeIn="StateDeptListWithEmps" height="150">
    <mx:columns>
      <mx:DataGridColumn headerText="First Name" dataField="firstName"/>
      <mx:DataGridColumn headerText="Name" dataField="lastName"/>
      <mx:DataGridColumn headerText="Phone" dataField="phone"/>
    </mx:columns>
  </mx:DataGrid>
  <mx:TextArea left="10" bottom="10" right="10" id="textLog" color="#6F6666" height="100" »
     borderAlpha="0.6"/>
  <s:Label x="20" y="19" text="Employee Directory" fontSize="26" fontWeight="normal" »
     fontFamily="Arial" />
</s:Application>
```

Now, we can proceed to add code that enables the client side talking with the server side.

2.7 Client Side Coding

Boostrap Client Side EO Service

Athena Framework relies on metadata to work. Once the user interface is created, we immediately load the metadata by creating an `EOService` object:

```
protected function onCreationComplete(event:FlexEvent):void {
  ...
  // Initialize eoService
  eoService = new EOService("http://localhost:8080/JavaEmployeeDir/messagebroker/amf", "eo", 2, true, »
   onEoServiceEvent);
  // Set Service Locator
  EOServiceLocator.getInstance().eoService = eoService;
  ...
}

// eoService Event handler - load employees and departments after the meta data loaded.
protected function onEoServiceEvent(event:EventEOService):void {
  if(event.kind == EventEOService.KIND_LOGIN_SUCCESS) {
  loadEmpAndDept();
  log("Metadata loaded successfully.");
  }else if(event.kind == EventEOService.KIND_LOGIN_ERROR || event.kind == »
   EventEOService.KIND_META_LOAD_ERROR) {
  log("Error: " + event.errorMessage);
    _roInprogress = true;
  }
}
```

The `EOService` class takes the following parameters in its constructor:

- **channelUrl**: in the format of `http://HOST:PORT/AppPath/messagebroker/amf`. We hard code it here. In most cases, you should dynamically pass it through Flash SWF parameter or SharedObject.
- **defaultDestination**: the destination as declared in the Java web project's `flex-services-config.xml`.
- **defaultPoolSize**: number of remote objects to be used for communicating with the server; the larger the number, the sooner queued requests will be sent.
- **initializeNow**: whether to load metadata immediately
- **eoServiceEventHandler_**: listener function to be registered.

Note that once the `EOService` object is created, we immediately register it to the service locator so that it can be accessed from any method of any class.

The `onEoServiceEvent` function is called once loading metadata succeeded or failed. `EOService` optionally supports login, however, here we do not specify any login mechanism. In case of successful metadata loading, `EventEOService.KIND_META_LOAD_SUCCESS` and `EventEOService.KIND_LOGIN_SUCCESS` will be dispatched in order.

Load Objects From the Server

Once `EOService` is initialized, we can start to call methods of the service classes on the server.

```
...
/** Service Names */
protected static const SVC_EMP:String = "empService"; // The remote service class name, as declared in »
 // eo-services.xml (Java web project)
protected static const SVC_EMP_LOAD_DATA:String = "loadData";
protected static const SVC_EMP_LOAD_EMP_FULL:String = "loadFullEmpObject";
protected static const SVC_EMP_SAVE_EMP:String = "saveEmployee";
protected static const SVC_EMP_REMOVE_EMP:String = "removeEmployee";
...
/** UnitOfWork. */
protected var uow:UnitOfWork = new UnitOfWork("myuow");

protected function loadEmpAndDept():void {
  log("Loading employees and departments ...");
  eoService.invokeService(SVC_EMP, SVC_EMP_LOAD_DATA, [], onLoadEmpAndDeptSuccess, onRemoteOpError, »
   uow);
}

/** On employees and departments load success */
protected function onLoadEmpAndDeptSuccess(e:EventRemoteOperationSuccess):void {
  log("Departments and employees loade successfully.");
  var result:Array = e.data as Array;

  employeeList = result[0] as ArrayCollection;
  departmentList = result[1] as ArrayCollection;

  currentState = "StateEmployeesList";
  setInprogressAndRefreshActions(false);
}
...
```

To invoke a method of a registered class on the server, you use `EOService.invokeService` to do so. `EOService.invokeService` takes the following

- **serviceName** the service name as defined in ieo-services.xml (in Java web project)
- **methodName** the name of the method to be invoked on the service class
- **args** arguments to be passed to the service method
- **onServiceSuccess** the function to be called on a successful completion of the invocation
- **onServiceError** the function to be called in case of error during service invocation
- **uow**: the `UnitOfWork` that enterprise objects returned from the server should be merged into
- **orgId** the id of the target org for multi-tenancy applications; use `-1` for non-cloud applications

The corresponding server side Java method to be invoked:

```
public class EmpService extends AbstractService {
  ...
  public Object[] loadData() {
    EOContext eoContext = createEOContext();

    EJBQLSelect select = eoContext.createSelectQuery("SELECT e FROM Employee e »
     [e.department:S]{po_e='employee_ID, firstName, lastName, email, phone, department_ID'}"); // »
     Partial objects
    List<Object> listOfEmployees = select.getResultList();

    select = eoContext.createSelectQuery("SELECT d FROM Department d ORDER BY d.nameFull");
    List<Object> listOfDepts = select.getResultList();

    if(listOfDepts.size() == 0) { // Nothing in db yet, fill sample data.
      UnitOfWork uow = createUow();
      Department dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
```

```
      dept.setNameFull("Finance");
      dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("Support");
      dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("HR");
      dept = (Department) uow.createNewInstance(Department_EO.SYSTEM_NAME);
      dept.setNameFull("R & D");

      Employee emp = (Employee) uow.createNewInstance(Employee_EO.SYSTEM_NAME);
      emp.setFirstName("Alan");
      emp.setLastName("Turing");
      emp.setPhone("111 11111");
      emp.setResume("1936: The Turing machine, computability, universal machine\n" +
        "1936-38: Princeton University. Ph.D. Logic, algebra, number theory ...");
      emp.setDepartment(dept);

      uow.flush();
      uow.close();
      // reload now
      listOfDepts = eoContext.createSelectQuery("SELECT d FROM Department d ORDER BY »
       d.nameFull").getResultList();
      listOfEmployees = eoContext.createSelectQuery("SELECT e FROM Employee e »
       [e.department:S]{po_e='employee_ID, firstName, lastName, email, phone, »
       department_ID'}").getResultList();
    }

    return new Object[] {listOfEmployees, listOfDepts};
  }
  ...
}
```

Note that `Employee` is loaded without its `resume` attribute since `resume` may hold large text content which we do not need when displaying on the datagrid.

Once the server method returns, `employeeList` and `departmentList` are set on the client side. As a result, the two data grids display employees and departments thanks to data binding.

Load Full Objects

When the user selects an employee from the datagrid and clicks the 'View/Edit' button, we need to re-load the selected employee object from the server side it is a partial object without `resume` attribute loaded.

```
/**
 * Try to edit the employee.
 * If the employee object is partial, will load full object first.
 */
protected function editEmployee(employee:Employee):void {
  if(employee == null) {
    return;
  }

  currentState = "StateEmployeeDetail";
  _editingEmployee = employee;

  // update ui using model
  textFirstName.text = employee.firstName == null ? "" : employee.firstName;
  textLastName.text = employee.lastName == null ? "" : employee.lastName;
  comboDepartment.selectedItem = employee.department == null ? null : employee.department;
  textEmail.text = employee.email;
  textPhone.text = employee.phone;

  if(employee.partialObject) {
    loadFullEmployeeObj(employee);
    textResume.text = "Loading...";
    textInfo.text = "Loading partial object...";
  }else {
    log("Editing employee: " + _editingEmployee.toStringOnDisplay);
    textInfo.text = "Viewing/Editing: " + employee.toStringOnDisplay;
    textResume.text = employee.resume;
  }
}

/** Load full employee object. */
protected function loadFullEmployeeObj(empPartial:Employee):void {
  log("The employee object is partial, loading the full object");
```

```
  setInprogressAndRefreshActions(true);
  eoService.invokeService(SVC_EMP, SVC_EMP_LOAD_EMP_FULL, [empPartial.id], onLoadEmpFullSuccess, »
   onRemoteOpError, uow);
}

/** on load full employee object success. */
protected function onLoadEmpFullSuccess(e:EventRemoteOperationSuccess):void {
  var employee:Employee = e.data as Employee;
  setInprogressAndRefreshActions(false);
  if(employee == null) {
    return;
  }
  log("Employee full object load successfully: " + employee.toStringOnDisplay);
  textInfo.text = "Employee full object loaded successfully";
  editEmployee(employee); // Edit the employee now
}
```

The corresponding server side Java method invoked:

```
public class EmpService extends AbstractService {
  ...
  public Object loadFullEmpObject(int empID) {
    String sql = "SELECT e FROM Employee e WHERE e.employee_ID = " + empID;

    EJBQLSelect select = new EJBQLSelect(createEOContext(), sql);
    return select.getSingleResult();
  }
  ...
}
```

When the full object is returned from the server, it is merged into the same `UnitOfWork` with the partial object. The existing partial object becomes a full object as all of its missing attributes are now available.

Save Objects

When the user finishes editing, he or she hits the 'Save' button to save the employee to the server:

```
/** Save Employee. */
protected function saveEmployee(emp:Employee):void {
  if(emp.partialObject) {
    Alert.show("The employee hasn't been loaded completely, please standby ...");
    return;
  }

  // Update model using user input
  _editingEmployee.firstName = textFirstName.text;
  _editingEmployee.lastName = textLastName.text;
  _editingEmployee.department = comboDepartment.selectedItem as Department;
  _editingEmployee.email = textEmail.text;
  _editingEmployee.phone = textPhone.text;
  _editingEmployee.resume = textResume.text;

  if(_editingEmployee.isNew() && _editingEmployee.department != null && »
   _editingEmployee.department.isRelationshipResolved(Department_EO.REL_employees)) {
    _editingEmployee.department.addToEmployees(_editingEmployee); // setup relationship
  }

  if(_editingEmployee.isCommitted() && »
   !_editingEmployee.hasChangesInOwningRelationshipObjects()) {
    log("No change, no need to save: " + _editingEmployee.toStringOnDisplay);
  }else {
    log("Saving Employee: " + _editingEmployee.toStringOnDisplay);
    setInprogressAndRefreshActions(true);
    eoService.invokeService(SVC_EMP, SVC_EMP_SAVE_EMP, [emp], onSavedSuccess, onRemoteOpError, uow);
  }
}

/** On employee saved success */
protected function onSavedSuccess(e:EventRemoteOperationSuccess):void {
  var savedEmployee:Employee = e.data as Employee;
  setInprogressAndRefreshActions(false);
  log("Employee save successfully " +  savedEmployee.toStringOnDisplay);

  if(!Utils.isIn(savedEmployee, employeeList)) {
```

```
    employeeList.addItem(savedEmployee);
  }

  textInfo.text = "Employee saved: " + savedEmployee.toStringOnDisplay;

  currentState = "StateEmployeesList";

  gridEmployees.selectedItem = savedEmployee;
  gridEmployees.verticalScrollPosition = gridEmployees.selectedIndex;
  gridEmployees.validateNow();
  onDGEmpListChanged(null);
}
```

The corresponding server side code:

```
public Object saveEmployee(Employee employee) {
  Object saveEmployee = doPerisist(employee, false);
  log.info("Employee saved: " + employee);
  return saveEmployee;
}
```

The `doPersist` method is convenient method provided by the `AbstractService`. It registers the given object and all the related objects to a newly created `UnitOfWork`, `flush` it and then returns the object.

Delete Objects

When the user selects an employee on the datagrid and hits 'Delete' button, we send a remove request to the server:

```
/** Remove Employee. */
protected function removeEmployee(emp:Employee):void {
  log("Remove employee: " + emp.toStringOnDisplay);
  setInprogressAndRefreshActions(true);
  eoService.invokeService(SVC_EMP, SVC_EMP_REMOVE_EMP, [_currentSelectedEmp], onRemoveSuccess, »
   onRemoteOpError, uow);
}

/** On employee removed successfully. */
protected function onRemoveSuccess(e:EventRemoteOperationSuccess):void {
  var removedEmployee:Employee = e.data as Employee;
  log("Empolyee removed: " + removedEmployee.toStringOnDisplay);
  textInfo.text = "Employee Removed: [" + removedEmployee.toStringOnDisplay + "]";
  currentState = "StateEmployeesList";

  Utils.removeFromArrayCollection(employeeList, removedEmployee);
  if(removedEmployee.department != null) {
    removedEmployee.department.invalidateRelationship(Department_EO.REL_employees);
  }

  onDGEmpListChanged(null);
  setInprogressAndRefreshActions(false);
}
```

Note that once the `Employee` object is deleted from the server, it is removed locally from the employee list and its department object's `employees` relationship is invalidated. Another approach to keep the client synchronized with the server is to reload all the data again.

The corresponding server side Java method invoked:

```
public Object removeEmployee(Employee employee) {
  Object removedEmployee = doPerisist(employee, true);
  log.info("Employee removed: " + employee);
  return removedEmployee;
}
```

Automatic Resolution of Relationships

When the user selects a department on the datagrid displaying all the departments, the selected department's `Deparment.employees` relationship will be resolved automatically:

```
/**
 * On datagrid of departments list changed, display the employees of the department.
 * If the ToMany relationship(department.employees) hasn't resolved, will resolve the relationship »
  first.
 */
protected function onGridDeptSelectionChanged(e:ListEvent):void {
  var _currentDepartment:Department = gridDepartments.selectedItem as Department;

  if(_currentDepartment != null) {
    currentState = "StateDeptEmployees";
    // for logging only.
    if(_currentDepartment.isRelationshipResolved(Department_EO.REL_employees)) { // Relationship has »
     // been resolved.
      textInfo.text = "ToMany relationship alread resolved, Employees: " + »
       _currentDepartment.employees.length;
      log("ToMany relationship alread resolved, Employees: " + _currentDepartment.employees.length);
    }else { // Relationship hasn't resolved, add an event listener
      textInfo.text = "The tomany relationsihp of " + _currentDepartment.toStringOnDisplay + ": »
       department.employees has't resolved, loading...";
      log("The tomany relationsihp of department.employees has't resolved, loading...");
      setInprogressAndRefreshActions(true);
      _currentDepartment.addEventListenerEventRelationshipResolved(onDeptEmployeesResolved);
    }
    // triggers the actual relationship resolution
    gridDeptEmployees.dataProvider = _currentDepartment.employees;
  }else {
    currentState = "StateDeptMgt";
    gridDeptEmployees.dataProvider = null;
  }
}

/** On the employees resolved, employees loaded successfully. */
protected function onDeptEmployeesResolved(e:EventRelationshipObject):void {
  setInprogressAndRefreshActions(false);
  textInfo.text = "ToMany relationship Employees resolved successfully, loaded employees : " + »
   (e.relationshipObject as ToMany).targetObjects.length;
  log("ToMany relationship Employees resolved successfully, loaded employees : " + »
   (e.relationshipObject as ToMany).targetObjects.length);
}
```

On the server side, the Java method `resolveRelationship(eo, relPath)` of the `org.athenasource.framework.eo.web.service.CommonEoService` class is used to resolve the relationship. You may modify this behavior (especially to enforce security) by:

- Configuring your own global relationship resolver method at the client side by set `EOServiceLocator.getInstance().relationshipResolver`; or
- Specifying the relationship resolution service and method for an individual object using `EOObject.setResolveService(serviceName, methodName)`.

Full Source

Please download the sample application project to browse the full source code.

3. Programming Using ActionScript

Athena Framework based applications written in Java run on the server side, while applications written in ActionScript run on the client side. The Athena ActionScript client library closely matches the Java library. For the Java EOObject class, there is an ActionScript version EOObject on the client with almost the same set of methods. For Java UnitOfWork class, there is an ActionScript version UnitOfWork.

The main difference is that **function invocation on the server side happens synchronously and the client side asynchronously**. On the client side, you need to use listener functions to handle function call returns from the server.

Note

For background information on Athena Framework, please refer to *Athena Framework Java Developer's Guide*.

3.1 The Generated Classes

When you hit the 'Generate Classes' button, Athena Console generates classes for Java and Flex. The following are sample classes generated for an entity named *Employee*:

```
[RemoteClass(alias="com.test.Employee")]
public class Employee extends Employee_EO {
  public static function createNewInstance():Employee {
    var instance:Employee = new Employee();
instance.entity = AppContext.getInstance().metaDomain.
  getEntityByName(Employee_EO.SYSTEM_NAME);
    return instance;
  }
} // end class
} // end package

public class Employee_EO extends EOObject {
  public static const COLUMN_COUNT:int = 5;
  public static const SYSTEM_NAME:String = "Employee";

  // Property names
  public static const ATTR_Employee_ID:String = "Employee_ID";
  public static const ATTR_name:String = "name";
  public static const ATTR_age:String = "age";
  public static const ATTR_status:String = "status";
  public static const ATTR_ORG_ID:String = "ORG_ID";

  public static const REL_addresses:String = "addresses";

  [Bindable(event="propertyChange")]
  public function get Employee_ID():int {
    return getValue(0) == null ?  -1 : int(getValue(0));
  }
  public function set Employee_ID(Employee_ID_:int):void {
    setValue(0, Employee_ID_ == -1 ? null : Employee_ID_, true);
  }

  [Bindable(event="propertyChange")]
  public function get name():String {
    var v:String = getValue(1) as String;
    return v;
  }
  public function set name(name_:String):void {
    setValue(1, name_, true);
  }

  [Bindable(event="propertyChange")]
  public function get age():int {
```

```
    return getValue(2) == null ?  -1 : int(getValue(2));
  }
  public function set age(age_:int):void {
    setValue(2, age_ == -1 ? null : age_, true);
  }

  [Bindable(event="propertyChange")]
  public function get status():int {
    return getValue(3) == null ?  -1 : int(getValue(3));
  }
  public function set status(status_:int):void {
    setValue(3, status_ == -1 ? null : status_, true);
  }

  [Bindable(event="propertyChange")]
  public function get ORG_ID():int {
    return getValue(4) == null ?  -1 : int(getValue(4));
  }
  public function set ORG_ID(ORG_ID_:int):void {
    setValue(4, ORG_ID_ == -1 ? null : ORG_ID_, true);
  }

  // ------------------------ Relationships ----------------------

  /**
   * [INVERSE] (Complement rel: Address.employee) Gets addresses
   */
  [Bindable(event="propertyChange")]
  public function get addresses():ToManyArrayCollection {
    return getRelationshipTargetObjectsList("addresses");
  }
  public function addToAddresses(address:EOObject, updateComplementRelationship:Boolean = »
   false):Boolean {
    return addRelationshipTargetObject("addresses", address, updateComplementRelationship);
  }
  public function removeFromAddresses(address:EOObject, updateComplementRelationship:Boolean = »
   false):Boolean {
    return removeRelationshipTargetObject("addresses", address, updateComplementRelationship);
  }

} // end class
} // end package
```

The metadata declared in the Employee class `[RemoteClass(alias="com.test.Employee")]` instructs the Flex framework to map this ActionScript class with the Java class `com.test.Employee` on the server.

3.2 Creates and Persists Enterprise Objects

The code below demonstrates how you create and persist EO's from the client side in ActionScript:

```
var emp:Employee = Employee.createNewInstance();
emp.Employee_ID = 2;
emp.name = "Tim";
emp.age = 40;

var addr1:Address = Address.createNewInstance();
addr1.Address_ID = 203;
addr1.address = "USA";

emp.addToAddresses(addr1, true);

eoService.invokeService("serviceName", "saveEmp", [emp], onSavedSuccess, onRemoteOpError);

private function onSavedSuccess(event:EventRemoteOperationSuccess):void {
  trace("Saved: " + (event.data as Employee).name );
}
```

3.3 Executes EJBQL from the Client-Side

The code below executes EJBQL and displays the returned EO's:

```
eoService.invokeService("serviceName", "executeQuery",
  ["SELECT e FROM Employee e"], onQuerySuccess, onQueryError);

private function onQuerySuccess(event:EventRemoteOperationSuccess):void {
  var emps:ArrayCollection = event.data as ArrayCollection;
  trace("Total number of employees found: " + emps.length);
  for(var i:int = 0; i < emps.length; i++) {
    var emp:Employee = emps.getItemAt(i) as Employee;
    trace("Employee: " + emp.name);
  }
}
```

For the sake of simplicity, the above code handles the success condition only. In practice, you need to handle the failure condition too.

3.4 Resolves Relationships

When you access an unresolved relationship, a placeholder will be returned and a request is sent to the server to request resolve the relationship.

`org.athenasource.framework.eo.core.ioc.CommandResolveService` is used to resolve a relationship object for an EO object. Usually, you do not execute this command explicitly. Instead, you simply query the relationship object:

```
var toManyCol:ToManyArrayCollection = emp.addresses; // Access to to-many relationship.
if(toManyCol.resolved) { // resolved
  doSomething(toManyCol);
}else{ // unresolved
   toManyCol.addEventListenerEventRelationshipResolved(onRelEditFormsResolvedSuccess);
}

private function onRelEditFormsResolvedSuccess(e:EventRelationshipObject):void {
  if(e.relationshipObject.relationship.systemName == “addresses”){
    doSomething(emp.addresses); // now, it is resolved.
  }
}
```

The read access to `emp.addresses` triggers the resolution if it has not been resolved. For a to-one realtionship:

```
var targetEo:EO = eo.toOneRel; // Access to to-many relationship.
if(targetEo == null || targetEo.resolved) {
  // resolved
}else{
  // not resolved
  eo.addEventListenerEventRelationshipResolved(...);
}
```

Control flow for resolving a relationship object

When `emp.addresses` is accessed and the relationship is not resolved, it will pend a relationship resolution request. `CommandResolveService` sends the eo object and the name of the relationship to be resolved to the server. The server replies the eo object with the relationship resolved. This eo object returned from the server is then merged to the UOW that the eo is associated to. The UOW then merges the relationship and dispatches `EventRelationshipResolved` at various levels.

Different levels of event dispatching

For a resolution of a to-many relationship, EventRelationshipResolved is dispatched at three levels: target object list, relationship object and the eo object. While for a to-one relationship, EventRelationshipResolved is only dispatched at two levels: relationship object and the eo object. You can only add listeners to the relationship object and/or the containing eo object for to-one relationships.

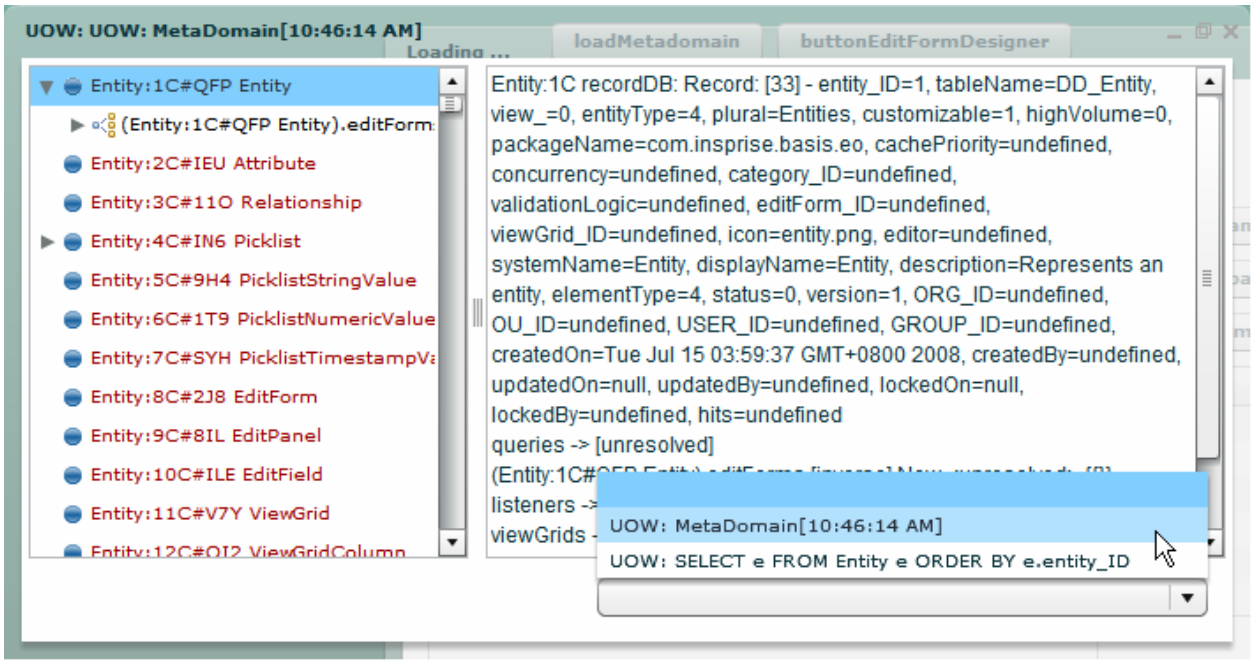
3.5 Merges the result into a UnitOfWork

A UnitOfWork (UOW) organizes a set of EOObjects. It guarantees that for a server object there is one or zero copy existing in the UOW. This uniqueness rule is vital to enterprise computing. A relationship of an EO object can not be resolved unless it is in a valid UOW. We call an EO object associated with a UOW a managed EO object. To make an EO object managed, you need to merge it to a UOW. Merging an EO object to a UOW updates the UOW with new information available in the EO object as well as its related EO objects. If there is already a copy of the object existing in the UOW, the existing copy will be updated and returned; otherwise, the EO object will be managed by the UOW and returned. It’s very important that in the former case, the merging copy should *not* be used again. By default, any call to a service method tries to merge resulting EO objects from the server into a proper UOW on the client side.

If you leave the `uow` parameter to null when calling `EOService.executeService` method, a new UOW will be created to contain the return EO objects from the server. Otherwise, those EO objects will be merged to the UOW passed to the `EOService.executeService` method.

3.6 Use EOObjectBrowser to peek UOW and EOs

EOObjectBrowser can be used to display details about one or more EOs. Further more, all related EOs will be displayed too. EOs from different UnitOfWorks are colored differently.



EOObjectBrowser also makes all current UOWs within the system available for browsing (see the bottom right combo box).

Usage

To explore an EO object, you use

```
EOObjectBrowser.exploreEO(eo:EOObject, parentUI:UIComponent = null): PopupTitleWindow
```

To explore a UOW, you use:

```
EOObjectBrowser.exploreUOW(uow:UnitOfWork, parentUI:UIComponent = null): PopupTitleWindow
```