Clear Data Builder

Version 3.1

User Guide

Revision: December 9, 2008

Table of Contents

Contents

Contents

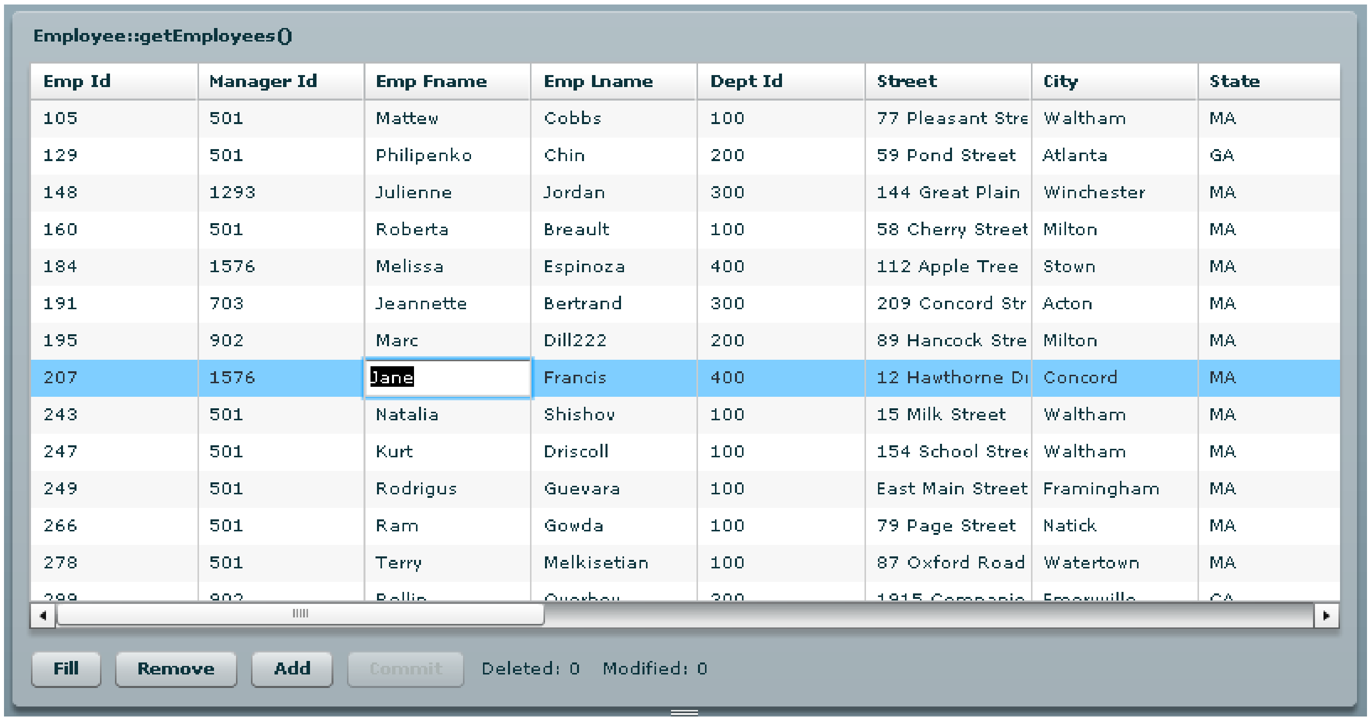
1.0 OVERVIEW

Clear Data Builder Eclipse Plugin (CDB) is a code generator for Flex-Java-DBMS Web applications. As an input, it uses either Java Data Tansfer Object (DTO) or an abstract Java class annotated with SQL statement or stored procedure call. As an output, it generates all required artifacts (Java, XML, MXML, ActionScript) and deploys the Web application with CRUD functionality in the J2EE application server of your choice.

CDB is a rapid application solution for projects that utilize Flex on the front and Java on the server side of your rich Internet application. It can work with the data supplied by Plain Old Java Objects (POJO). While working with POJO, you can configure deployment type to use either commercial server-side component Adobe LiveCycle Data Services ES (LCDS) or the open source Adobe BlazeDS.

For ? quick introduction to the process of creating an SQL-based CRUD application with CDB and BlazeDS, watch a pre-recorded screencast at http://www.myflex.org/demos/CDB_blazeds_db2/CDB_blazeds_db2.html

For ? quick introduction to the process of creating a Java DTO-based CRUD application, watch a pre-recorded screencast at http://www.myflex.org/demos/cdb31/CDB31.html .



Emp Id	Manager Id	Emp Fname	Emp Lname	Dept Id	Street	City	State
105	501	Mattew	Cobbs	100	77 Pleasant Stre	Waltham	MA
129	501	Philipenko	Chin	200	59 Pond Street	Atlanta	GA
148	1293	Julienne	Jordan	300	144 Great Plain	Winchester	MA
160	501	Roberta	Breault	100	58 Cherry Street	Milton	MA
184	1576	Melissa	Espinoza	400	112 Apple Tree	Stown	MA
191	703	Jeannette	Bertrand	300	209 Concord Str	Acton	MA
195	902	Marc	Dill222	200	89 Hancock Stre	Milton	MA
207	1576	Jane	Francis	400	12 Hawthorne Di	Concord	MA
243	501	Natalia	Shishov	100	15 Milk Street	Waltham	MA
247	501	Kurt	Driscoll	100	154 School Stree	Waltham	MA
249	501	Rodrigus	Guevara	100	East Main Street	Framingham	MA
266	501	Ram	Gowda	100	79 Page Street	Natick	MA
278	501	Terry	Melkisetian	100	87 Oxford Road	Watertown	MA
299	902	Rollin	Overbey	200	1915 Companio	Emeryville	CA

In the next sections you'll find detailed instructions on how to try these examples on your own, but first get to know the software that has to be installed on your computer..

1.1 Prerequisites and Supported Servers

1.1.1 Java and Eclipse IDE

During development, CDB requires Java Development Kit (JDK) 1.5 or later, but the version of the JRE used by your application server can be different, for example 1.5.2. It also requires certain features included in Java Development Tools (JDT) plugin, which is installed as a plugin to Eclipse.

CDB 3 requires Eclipse IDE for Java Developers (see http://www.eclipse.org/downloads/) that includes tools for creation of dynamic Web projects and configuring servers.

1.1.2 Application servers supported

CDB supports any Java Servlet engine that can run under JVM 1.5 or later (i.e. Apache Tomcat 6.0 or later) or J2EE application servers such as WebLogic, WebSphere, and others.

1.1.3 Supported DBMS

While CDB can be used with any relational DBMS that has JDBC driver, each DBMS can have its own specifics such as how to store binary large objects or return a data result set from a stored procedure. So far CDB has been tested with MySql Server 5, Oracle, MS SQL 2000, DB2, and MS SQL 2005.

1.2 Required software

The easiest way to learn how to create CRUD applications with CDB 3 is by example. We'll use the following software:

1. Eclipse IDE for Java Developers: http://www.eclipse.org/downloads/
2. Adobe Flex Builder 3 plugin version: http://www.flex.org
3. Adobe BlazeDS 3.0: http://opensource.adobe.com/wiki/display/blazeds/download+blazeds+3
4. Clear Data Builder 3.1 plugin for Eclipse from Farata Systems: http://www.myflex.org/releases/cdb31/site.zip
5. Apache Tomcat 6.0 http://tomcat.apache.org/download-60.cgi
6. Optional. If you are planning to run SQL-based example, install one of the popular DBMS: IBM DB2: http://www-306.ibm.com/software/data/db2/express/ , MySql Server, SQL Server or Oracle.

1.3 Installing the software for the sample application

Eclipse JEE installation is as simple as unzipping it to a folder on your hard disk. You'll need Eclipse JEE version 3.3 or above.

Installing of the plugin version of Flex Builder 3 is also easy, as long as you'll be able to find it online. Adobe is flirting with you girls, by hiding it behind the standalone Flex Builder distribution, but savvy housewives can always find the exact ingredients they are looking for. Just follow the link by the text "Have Eclipse already installed…" on Flex Builder download site.

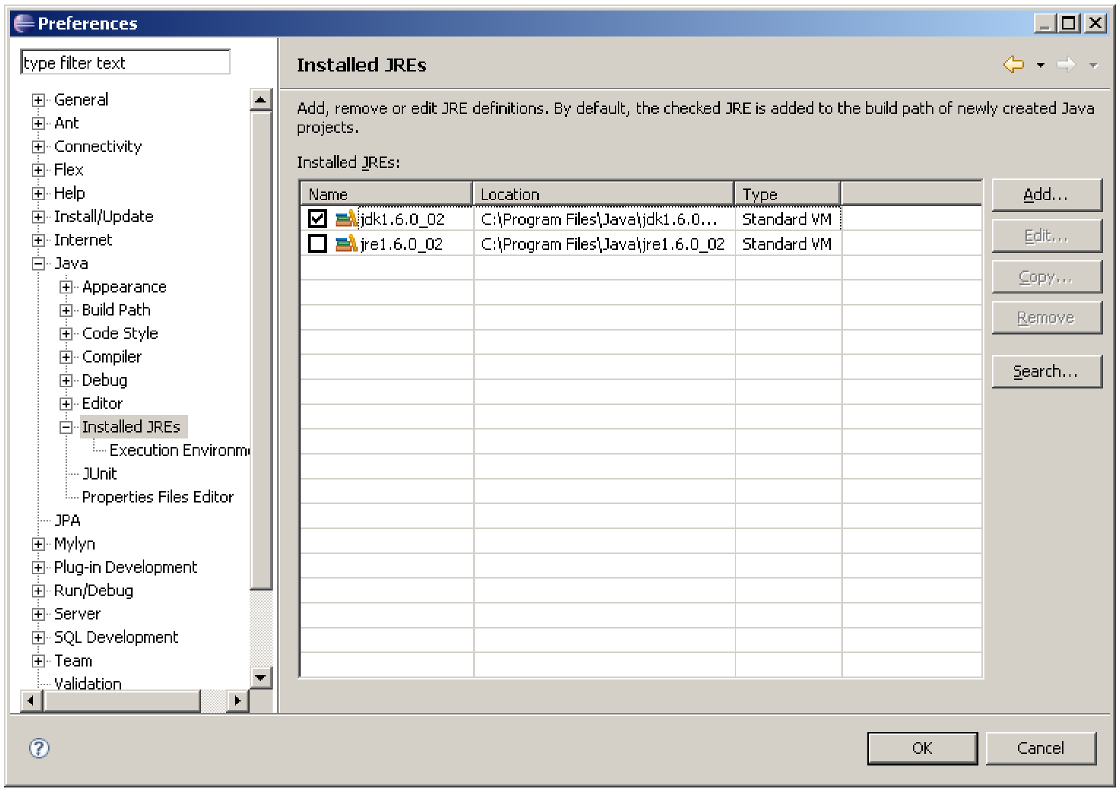
Register at www.myflex.org and get the license to Clear Data Builder (CDB) - it's free. Download CDB 3.1 at http://www.myflex.org/releases/cdb31/site.zip .

Unzip it into any folder on your disk and install CDB plugin by selecting Eclipse menu Help | Software Updates | Find and Install. Select the radio button "Search for new features to install". Press the button New Local Site, give it a name (i.e. Local CDB) and point at the directory where you've unzipped the file site.zip.

Install CDB license in Eclipse by using the Menu Window | Preferences |Flex |My Flex | License Management |Install License. The CDB license goes under the name daoflex, which was a command line open source predecessor of CDB.

4. Download the binary edition of BlazeDS , which is a relatively small (4Mb) zip file. Just unzip it into some folder, say in c:\blazeds .

It goes without saying, that any respected chef has JDK 1.5 or later. Yes, we need JDK, not JRE for today's class . Use Eclipse menu Window | Preferences | Java | Installed JREs and point it to your JDK installation directory, for example:



Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

	Name	Location	Type
☑	jdk1.6.0_02	C:\Program Files\Java\jdk1.6.0...	Standard VM
☐	jre1.6.0_02	C:\Program Files\Java\jre1.6.0_02	Standard VM

5. Download Tomcat 6 - select Windows Service Installer from the Core Downloads section. Run the install accepting all defaults.

2.0 DTO-BASED CODE GENERATION

Starting from version 3.1 CDB has an option of “not knowing” anything about the data persistence layer. It can generate the front end for a CRUD application based on the Java Data Transfer Object (DTO) a.k.a. Value Object. If you are interested in SQL-based code generation, please refer to section 3.0 of this document.

2.1 Creating Sample CRUD Application Based on Java DTO

Pretty often, Flex communicates with POJO that does not directly connects to DBMS using JDBC. The enterprises may use some object-relational frameworks, Web Services or any other means of working with data. This section will cover CRUD generation without the need to use SQL.

In this mode, CDB requires manually written Java class that implements Assembler design pattern similarly to how it’s done in LCDS data management services. The `fill()` method of such a class will return a collection of Java Data Transfer Objects (DTO), and the `sync()` method would receive a collection of `ChangeObjects` that can be examined by Java code to perform data creation, modification or removal.
In this section you’ll see how CDB generates a sample Flex/Java CRUD application based on a provided Java DTO. Since this process is decoupled from the data storage, it’ll be a responsibility of Java developers to write the code for the data persistence and retrieval.

There are several benefits of using CDB for DTO-based code generation :

1. It automatically keeps track of all the changes made by the user and maintains a collection of the `ChangeObjects`.
2. CDB automatically generates Flex code utilizing remoting using AMF protocol. As a part of this process, it automatically generates ActionScript DTO's based on their Java peers (this can be done on any code modification)
3. Test application generated by CDB utilize components from clear.swc component library, which
4. ANT build and deployment scripts are automatically generated for the selected application server.

2.2 Installing and Generating a Sample Application

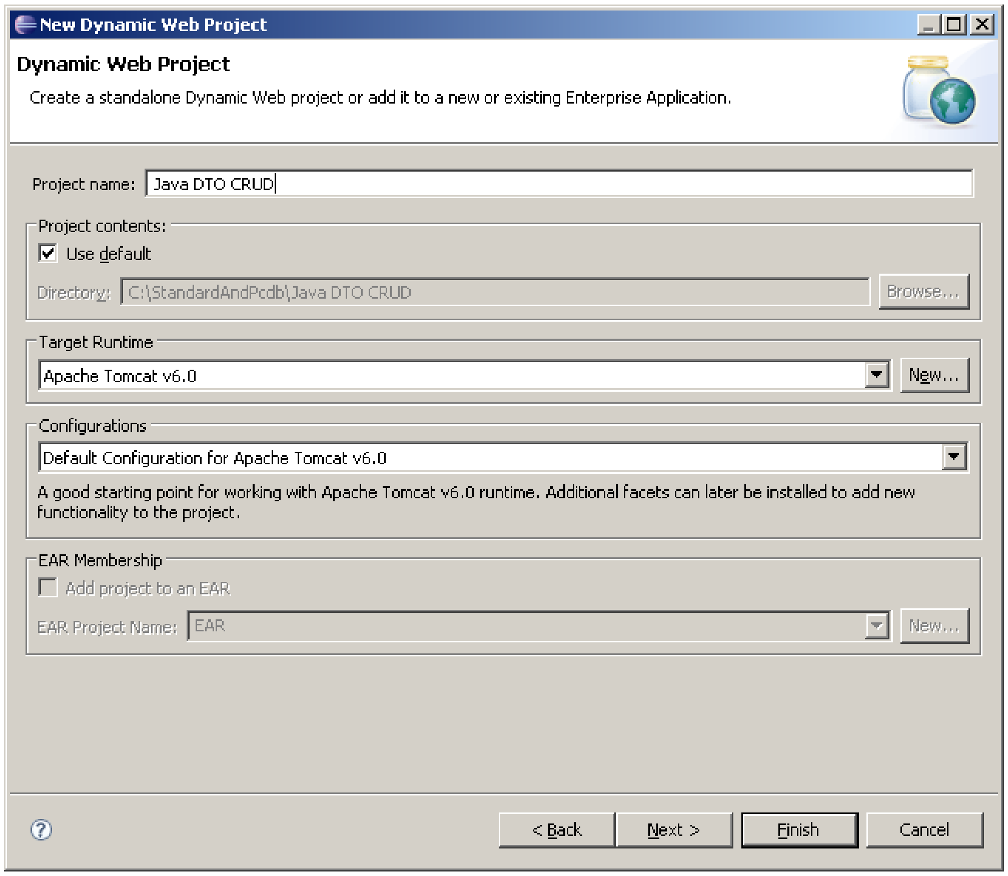
If you have one of the previous versions of CDB, delete all directories and files having "farata" in the name from the eclipse folders plugins and features.

Then, start Eclipse JEE, open the menus Help | Software Updates | Find and Install and select the option Find New Features to install.

On the window "Update Sites To Visit", create a new local site pointing at the folder on your drive where you unzipped the file site.zip. Press the button Finish to complete the install of CDB 3.1.
Follow the steps below to generate a sample

1. Start Eclipse JEE and create a new instance of the Tomcat 6 server. Right-click in the tab Servers, select New, Tomcat 6, and then click on Finish.

2. Create a new Dynamic Web Project in Eclipse using the menus File | New | Other | Web | Dynamic Web Project. Let's name the project Java DTO CRUD. Specify the Target Runtime (we'll use Apache Tomcat 6.0) in the Dynamic Web Project configuration screen:



New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: Java DTO CRUD

Project contents:

[✓] Use default

Directory: C:\StandardAndPcdb\Java DTO CRUD

Target Runtime: Apache Tomcat v6.0

Configurations: Default Configuration for Apache Tomcat v6.0

A good starting point for working with Apache Tomcat v6.0 runtime. Additional facets can later be installed to add new functionality to the project.

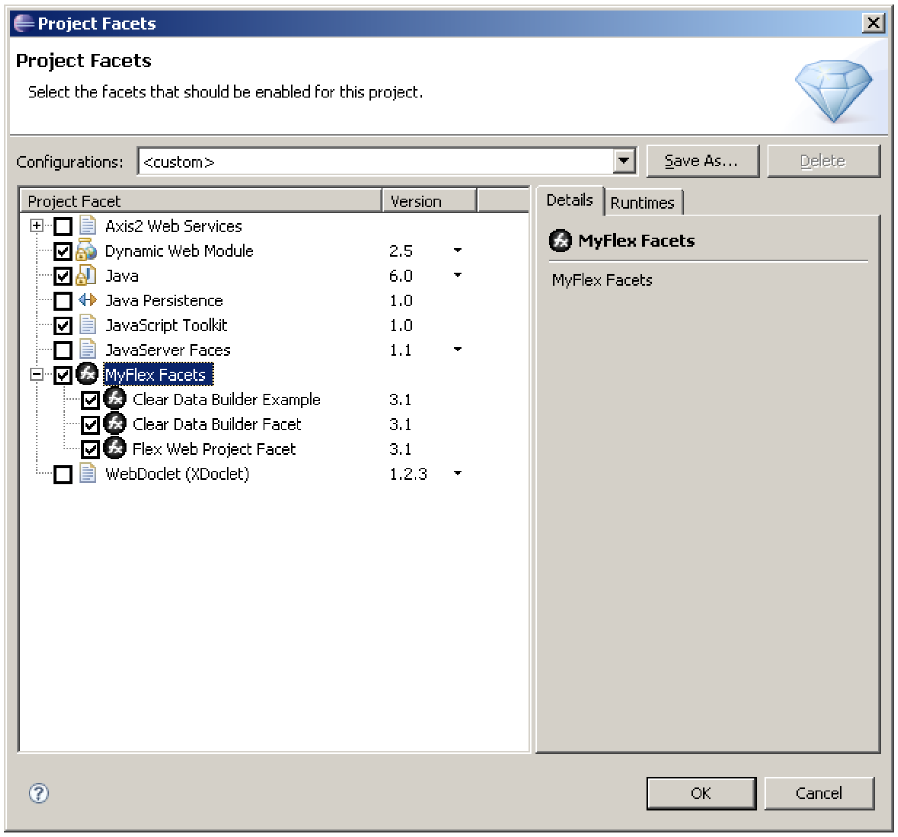
EAR Membership

[ ] Add project to an EAR

EAR Project Name: EAR

Press the button Next.

3. In the next window select MyFlex Facets as shown below:


Project Facets

Select the facets that should be enabled for this project.

Configurations: <custom>

Project Facet	Version
☐ Axis2 Web Services	
☑ Dynamic Web Module	2.5
☑ Java	6.0
☐ Java Persistence	1.0
☑ JavaScript Toolkit	1.0
☐ JavaServer Faces	1.1
☑ MyFlex Facets	
☑ Clear Data Builder Example	3.1
☑ Clear Data Builder Facet	3.1
☑ Flex Web Project Facet	3.1
☐ WebDoclet (XDoclet)	1.2.3

Details: MyFlex Facets

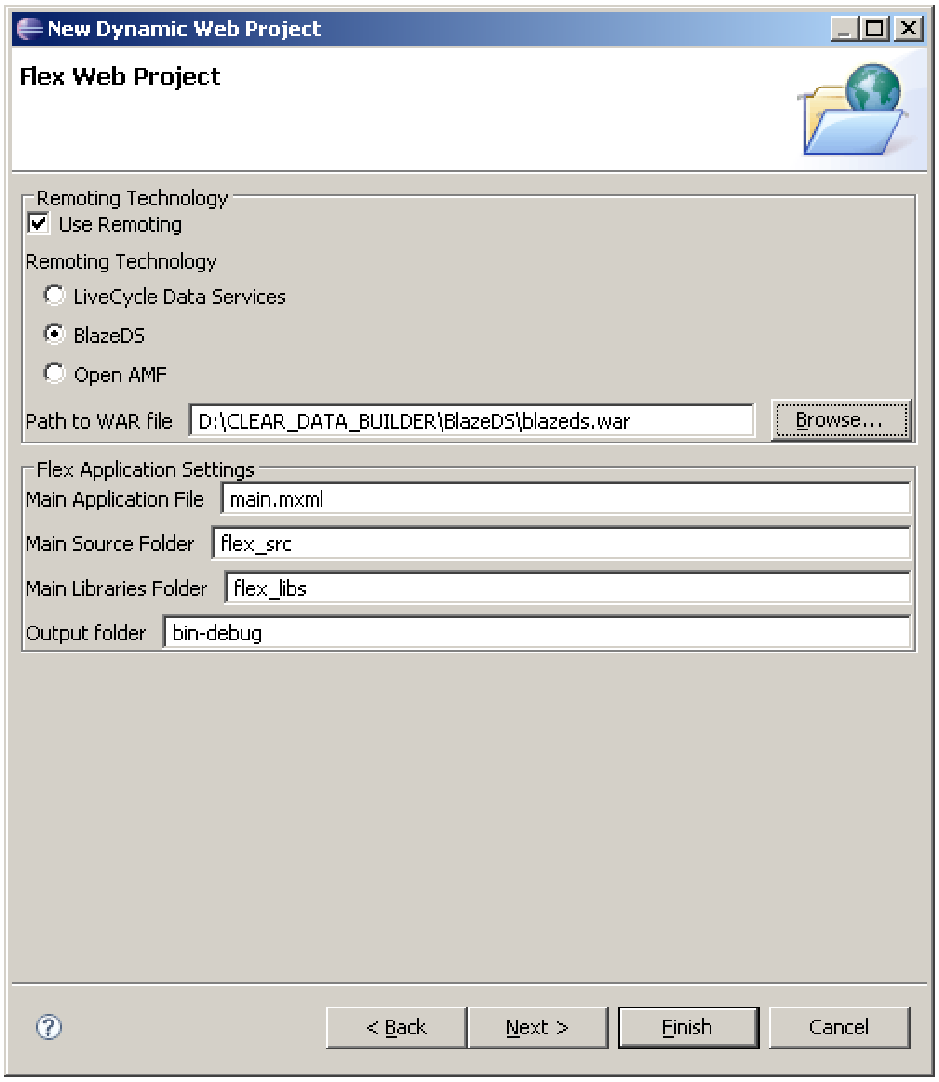
MyFlex Facets

If you don't see CDB facets on this screen, this means you did not install CDB license, which is available for free at http://www.myflex.org .

Press Ok and the button Next.

4. Leave the next screen the suggests Java_DTO_CRUD as a context, WebContent as a content directory and src as a directory for the Java code. Press the button Next.

5. Now you need to specify that we are going to use BlazeDS and where your blazeds.war is located:


New Dynamic Web Project

Flex Web Project

Remoting Technology

☑ Use Remoting

Remoting Technology

○ LiveCycle Data Services

● BlazeDS

○ Open AMF

Path to WAR file: D:\CLEAR_DATA_BUILDER\BlazeDS\blazeds.war

Flex Application Settings

Main Application File: main.mxml

Main Source Folder: flex_src

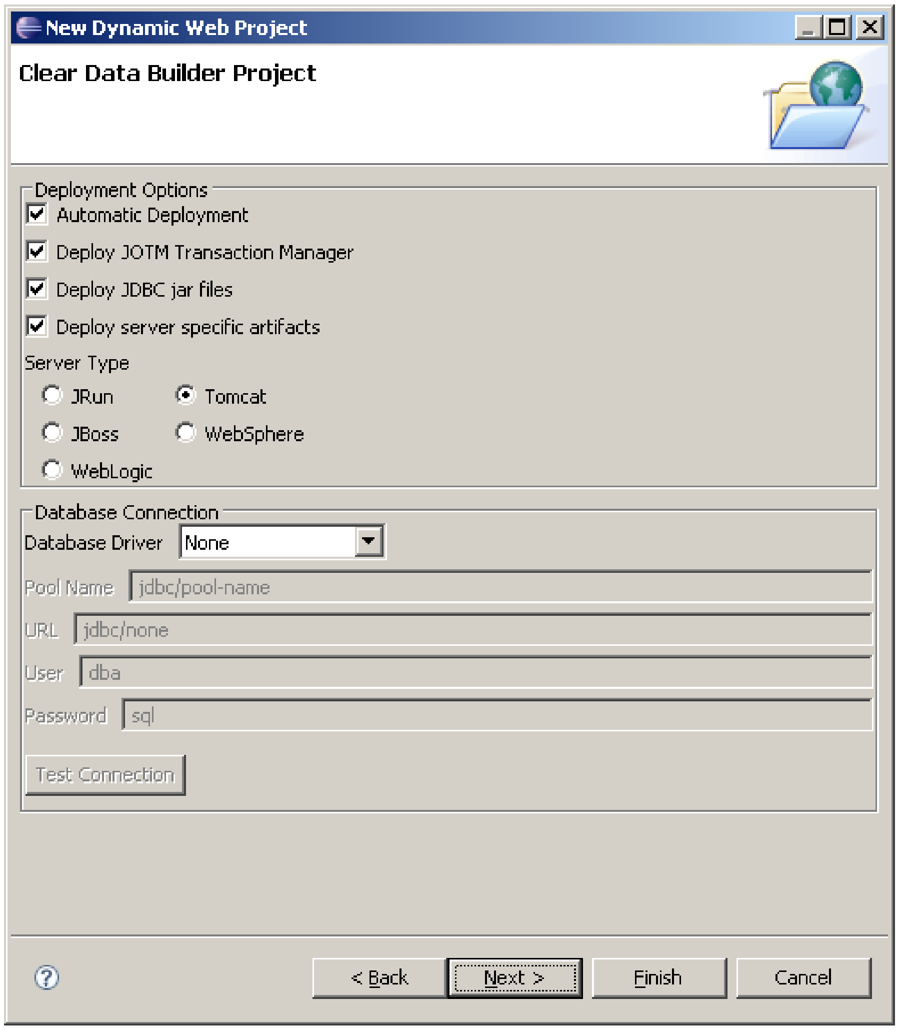
Main Libraries Folder: flex_libs

Output folder: bin-debug

Press the button Next.

6. Finally, we need to specify that the application will be deployed under Tomcat

Note. Since you are not going to use any DBMS in this exercise, keep "None" as a selected database driver:


New Dynamic Web Project

Clear Data Builder Project

Deployment Options

- [✓] Automatic Deployment
- [✓] Deploy JOTM Transaction Manager
- [✓] Deploy JDBC jar files
- [✓] Deploy server specific artifacts

Server Type

- ○ JRun
- ● Tomcat
- ○ JBoss
- ○ WebSphere
- ○ WebLogic

Database Connection

Database Driver: None

Pool Name: jdbc/pool-name

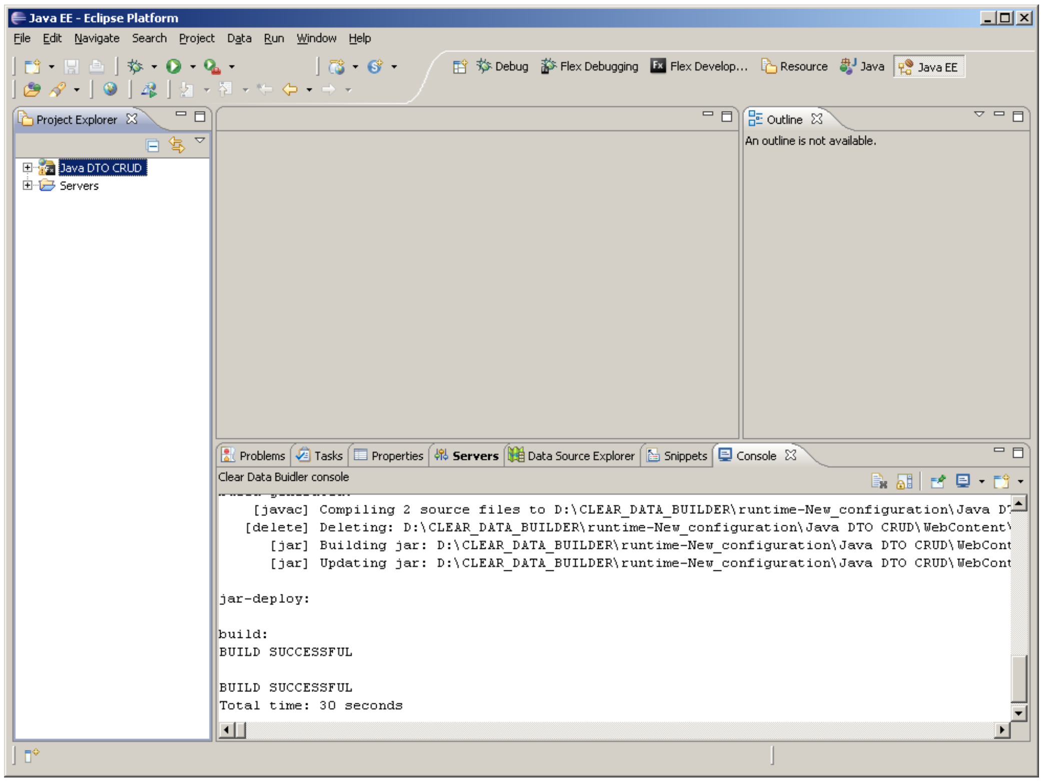
URL: jdbc/none

User: dba

Password: sql

Press Next on this and the next screen that offers to create an example database.

7. Press the button Finish, and the new Dynamic Web Project will be created:


Java EE - Eclipse Platform
File Edit Navigate Search Project Data Run Window Help
Debug Flex Debugging Flex Develop... Resource Java Java EE
Project Explorer
Java DTO CRUD
Servers
Outline
Problems Tasks Properties Servers Data Source Explorer Snippets Console

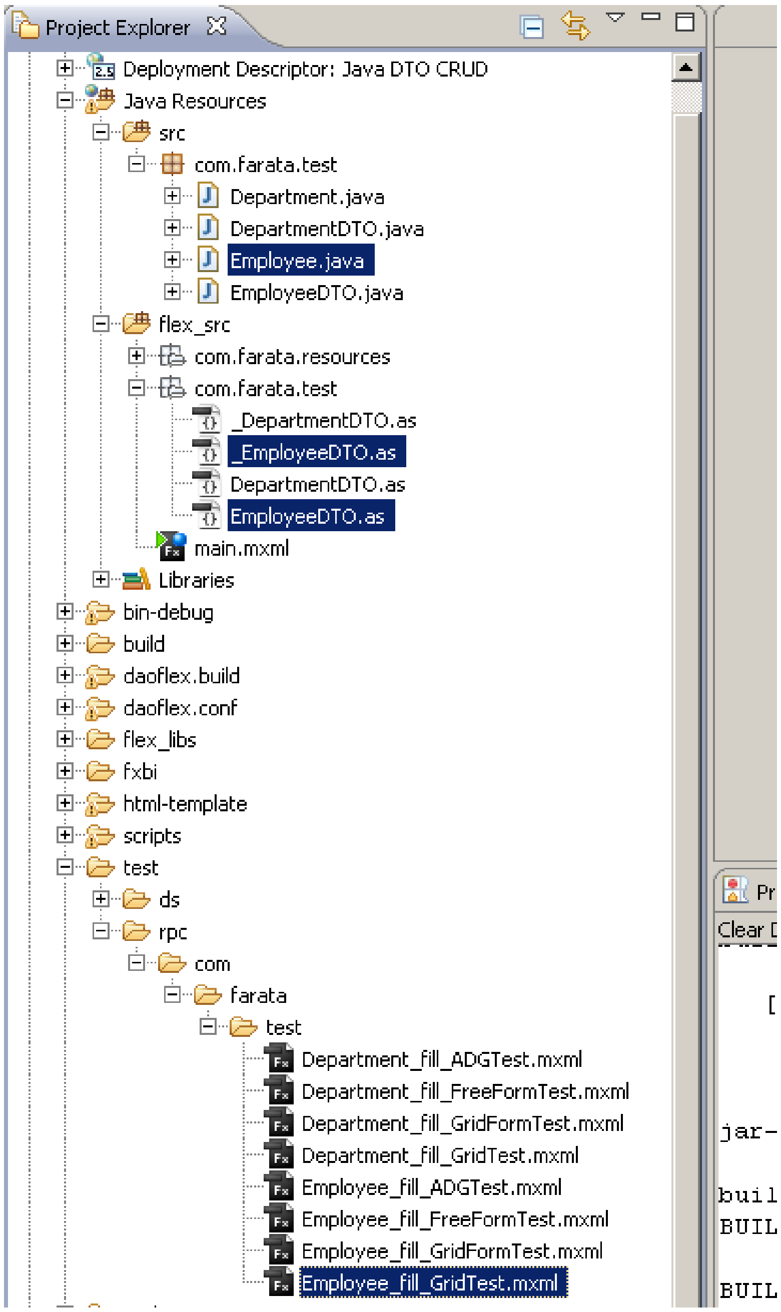
An outline is not available.

Clear Data Buidler console

```
    [javac] Compiling 2 source files to D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Java D
   [delete] Deleting: D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Java DTO CRUD\WebContent\
      [jar] Building jar: D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Java DTO CRUD\WebCont
      [jar] Updating jar: D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Java DTO CRUD\WebCont

jar-deploy:

build:
BUILD SUCCESSFUL

BUILD SUCCESSFUL
Total time: 30 seconds
```

8. Open the Java Resources folder in you project, and you'll find there a number of Java classes generated by CDB. This is an example application with hardcoded data about Employee. The flex_src has a fully functioning Flex application that communicates with Java sample code.

.


Project Explorer

Deployment Descriptor: Java DTO CRUD
Java Resources
src
com.farata.test
Department.java
DepartmentDTO.java
Employee.java
EmployeeDTO.java
flex_src
com.farata.resources
com.farata.test
_DepartmentDTO.as
_EmployeeDTO.as
DepartmentDTO.as
EmployeeDTO.as
main.mxml
Libraries
bin-debug
build
daoflex.build
daoflex.conf
flex_libs
fxbi
html-template
scripts
test
ds
rpc
com
farata
test
Department_fill_ADGTest.mxml
Department_fill_FreeFormTest.mxml
Department_fill_GridFormTest.mxml
Department_fill_GridTest.mxml
Employee_fill_ADGTest.mxml
Employee_fill_FreeFormTest.mxml
Employee_fill_GridFormTest.mxml
Employee_fill_GridTest.mxml

We'll use one of the generated Flex applications called Employee_fill_GridTest.mxml shortly.

9. CDB uses Java DTO as an input for code generation. For illustration purposes, CDB generates so called Java Assembler shown below. In the real-world project, a Java developer would need to write a similar class that includes code for data retrieval and persistence. This is a place to write your code that knows where the data is (WebService, content-management server, the cloud, et al).

The Java developer has to specify the name of the Java DTO that will be used in Flex-Java communication. The sample code below illustrates how to do this using a doclet parameter `transferType` of the **`@daoflex:java`**. See the doclet above right above the method `fill()` of the class `Employee`:

```
package com.farata.test;

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.util.List;

import com.farata.daoflex.IJavaDAO;
import flex.data.ChangeObject;

/**
 * @daoflex:webservice
 */
public class Employee implements IJavaDAO {
        /**
         * @daoflex:java transferType=com.farata.test.EmployeeDTO[]
         */
        public List<EmployeeDTO> fill() {
                // This sample code was generated for illustration purposes.
                // Please write your own code here preparing your objects.

                System.out.println("fill method has executed");

                ArrayList<EmployeeDTO> list = new ArrayList<EmployeeDTO>();
                EmployeeDTO dto = new EmployeeDTO();
                dto.setBene_day_care("N");
                dto.setBene_health_ins("Y");
                dto.setBene_life_ins("Y");
                dto.setBirth_date(new GregorianCalendar(1961, 12, 4).getTime());
                dto.setCity("Waltham");
                dto.setDept_id(100);
                dto.setEmp_fname("Mattew");
                dto.setEmp_id(105);
                dto.setEmp_lname("Cobbs");
                dto.setManager_id(501);
                dto.setPhone("6175553840");
                dto.setSalary(62000);
                dto.setSex("M");
                dto.setSs_number("052345739");
                dto.setStart_date(new GregorianCalendar(1987, 7, 2).getTime());
                dto.setState("MA");
                dto.setStatus("A");
                dto.setStreet("77 Pleasant Street");
                dto.setTermination_date(null);
                dto.setZip_code("02154");
                list.add(dto);

                dto = new EmployeeDTO();
                dto.setBene_day_care("N");
                dto.setBene_health_ins("Y");
                dto.setBene_life_ins("Y");
                dto.setBirth_date(new GregorianCalendar(1967, 10, 30).getTime());
                dto.setCity("Atlanta");
                dto.setDept_id(200);
                dto.setEmp_fname("Chin");
                dto.setEmp_id(129);
                dto.setEmp_lname("Philipenko");
                dto.setManager_id(501);
                dto.setPhone("4045552341");
                dto.setSalary(38500);
                dto.setSex("M");
                dto.setSs_number("024608923");
                dto.setStart_date(new GregorianCalendar(1998, 8, 4).getTime());
                dto.setState("GA");
                dto.setStatus("A");
                dto.setStreet("59 Pond Street");
                dto.setTermination_date(null);
                dto.setZip_code("30339");
                list.add(dto);

                dto = new EmployeeDTO();
                dto.setBene_day_care("N");
                dto.setBene_health_ins("Y");
                dto.setBene_life_ins("Y");
                dto.setBirth_date(new GregorianCalendar(1952, 12, 13).getTime());
                dto.setCity("Winchester");
                dto.setDept_id(300);
                dto.setEmp_fname("Julienne");
                dto.setEmp_id(148);
                dto.setEmp_lname("Jordan");
                dto.setManager_id(1293);
                dto.setPhone("6175557834");
                dto.setSalary(51432);
                dto.setSex("F");
                dto.setSs_number("501704733");
                dto.setStart_date(new GregorianCalendar(1997, 10, 4).getTime());
                dto.setState("MA");
                dto.setStatus("A");
                dto.setStreet("144 Great Plain Avenue");
                dto.setTermination_date(null);
                dto.setZip_code("01890");
                list.add(dto);

                return list;
        }

        public List<ChangeObject> sync(List<ChangeObject> list) {
                for (ChangeObject changeObject : list) {
                        if (changeObject.isCreate()) {
                               doCreate(changeObject);
                        } else if (changeObject.isUpdate()) {
                               doUpdate(changeObject);
                        } else if (changeObject.isDelete()) {
                               doDelete(changeObject);
                        }
                }
                System.out.println("sync method has executed");
                return list;
        }

        private void doCreate(ChangeObject changeObject) {
                // This sample code illustrates how to extract data from the newly
                // created object.
                // Please replace it with your own code.

                System.out.println("doCreate method executing");

                String[] changedNames = changeObject.getChangedPropertyNames();
                EmployeeDTO dto = (EmployeeDTO) changeObject.getNewVersion();
                Class<? extends EmployeeDTO> clazz = dto.getClass();
                for (String changedName : changedNames) {
                        try {
                               String methodName = "get"
                                              + changedName.substring(0, 1).toUpperCase()
                                              + changedName.substring(1);
                               Method method = clazz.getMethod(methodName);
                               Object result = method.invoke(dto);
                               System.out.println("Created: " + changedName + " Value: "
                                              + result);
                        } catch (Exception e) {
                               e.printStackTrace();
                        }
                }
        }

        private void doUpdate(ChangeObject changeObject) {
                // This sample code illustrates how to extract data from the updated
                // object.
                // Please replace it with your own code.

                System.out.println("doUpdate method executing");

                String[] changedNames = changeObject.getChangedPropertyNames();
                EmployeeDTO newDto = (EmployeeDTO) changeObject.getNewVersion();
                EmployeeDTO prevDto = (EmployeeDTO) changeObject.getPreviousVersion();
                Class<? extends EmployeeDTO> clazz = newDto.getClass();
```

```
                for (String changedName : changedNames) {
                        try {
                                String methodName = "get"
                                                + changedName.substring(0, 1).toUpperCase()
                                                + changedName.substring(1);
                                Method method = clazz.getMethod(methodName);
                                Object newResult = method.invoke(newDto);
                                Object prevResult = method.invoke(prevDto);
                                System.out.println("Changed: " + changedName + ", Old Value: "
                                                + prevResult + ", New Value: " + newResult);
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
        }

        private void doDelete(ChangeObject changeObject) {
                // This sample code illustrates how to extract data from the deleted
                // object. Please replace it with your own code.

                System.out.println("doDelete method executing");

                EmployeeDTO dto = (EmployeeDTO) changeObject.getPreviousVersion();
                System.out.println("Deleted: " + dto.getEmp_id());
        }
}
```

The above class `com.farata.test.Employee` has to implement the interface `com.farata.daoflex.IJavaDAO` that declares two functions: `fill()` and `sync()`.

```
package com.farata.daoflex;
import java.util.List;
public interface IJavaDAO
{
     public abstract List fill();
     public abstract List sync(List list);
}
```

The method `fill()` should be used for preparing of list of DTO's. The method `sync()` is for handling data received from the Flex client.

Please note, that Flex code specifies the name of the sync method to call on the server side. The code from the file test\rpc\com\farata\test\Employee_fill_GridTest.mxml demonstrates how this could be done:

```
import com.farata.collections.DataCollection;

[Bindable]
public var collection:DataCollection ;
[Bindable]
private var log : ArrayCollection;

private function onCreationComplete() : void {
        collection = new DataCollection();
        collection.destination="com.farata.test.Employee";
        collection.method="fill";

        collection.syncMethod="sync";

        log = new ArrayCollection();
        collection.addEventListener( CollectionEvent.COLLECTION_CHANGE, logEvent);
        collection.addEventListener("fault", logEvent);
        fill_onClick();
}
```

On the server side, the code of the assembler class goes through the collection of received ChangeObjects, and calls the appropriate function based on the flag set for each object:

```
public List<ChangeObject> sync(List<ChangeObject> list) {
		for (ChangeObject changeObject : list) {
			if (changeObject.isCreate()) {
				doCreate(changeObject);
			} else if (changeObject.isUpdate()) {
				doUpdate(changeObject);
			} else if (changeObject.isDelete()) {
				doDelete(changeObject);
			}
		}
		System.out.println("sync method has executed");
		return list;
	}
```

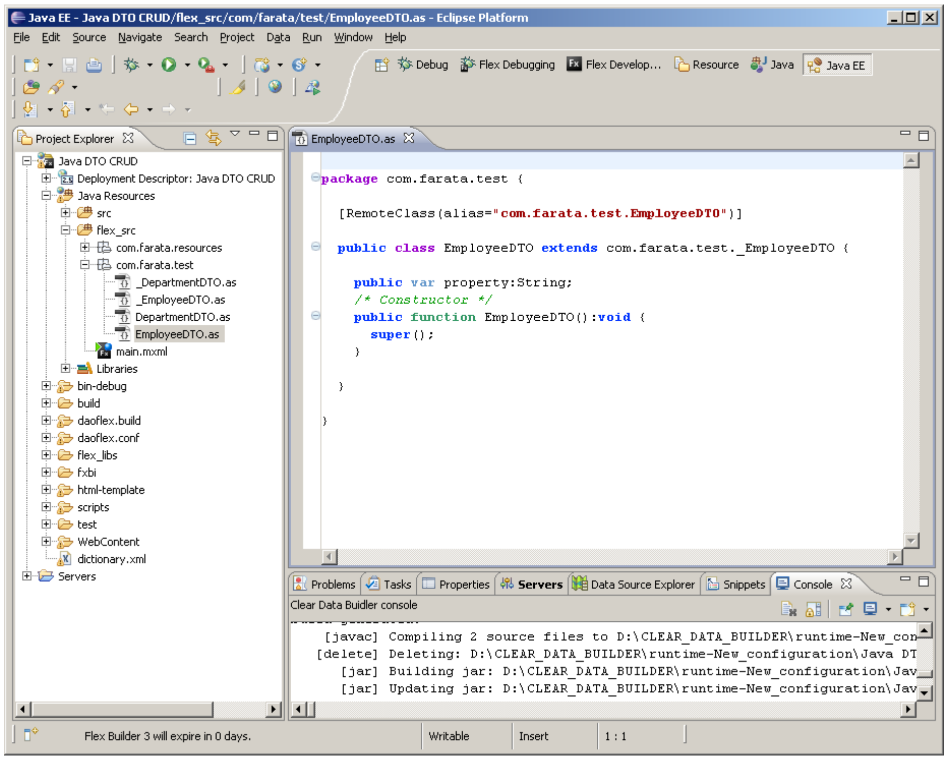
The assembler class `com.farata.test.Employee` uses the DTO class `com.farata.test.EmployeeDTO`:

```
package com.farata.test;
import java.util.Date;
import com.farata.dto2fx.annotations.FXClass;

@FXClass
public class EmployeeDTO {
	public long emp_id;
	public long manager_id;
	public String emp_fname;
	public String emp_lname;
	public long dept_id;
	public String street;
	public String city;
	public String state;
	public String zip_code;
	public String phone;
	public String status;
	public String ss_number;
	public double salary;
	public Date start_date;
	public Date termination_date;
	public Date birth_date;
	public String bene_health_ins;
	public String bene_life_ins;
	public String bene_day_care;
	public String sex;

	public long getEmp_id() {
		return emp_id;
	}
	public void setEmp_id(long emp_id) {
		this.emp_id = emp_id;
	}
	public long getManager_id() {
		return manager_id;
	}
	public void setManager_id(long manager_id) {
		this.manager_id = manager_id;
	}
	public String getEmp_fname() {
		return emp_fname;
	}
	public void setEmp_fname(String emp_fname) {
		this.emp_fname = emp_fname;
	}
	public String getEmp_lname() {
		return emp_lname;
	}
	public void setEmp_lname(String emp_lname) {
		this.emp_lname = emp_lname;
	}
```

```
public long getDept_id() {
        return dept_id;
}
public void setDept_id(long dept_id) {
        this.dept_id = dept_id;
}
public String getStreet() {
        return street;
}
public void setStreet(String street) {
        this.street = street;
}
public String getCity() {
        return city;
}
public void setCity(String city) {
        this.city = city;
}
public String getState() {
        return state;
}
public void setState(String state) {
        this.state = state;
}
public String getZip_code() {
        return zip_code;
}
public void setZip_code(String zip_code) {
        this.zip_code = zip_code;
}
public String getPhone() {
        return phone;
}
public void setPhone(String phone) {
        this.phone = phone;
}
public String getStatus() {
        return status;
}
public void setStatus(String status) {
        this.status = status;
}
public String getSs_number() {
        return ss_number;
}
public void setSs_number(String ss_number) {
        this.ss_number = ss_number;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
public Date getStart_date() {
        return start_date;
}
public void setStart_date(Date start_date) {
        this.start_date = start_date;
}
public Date getTermination_date() {
        return termination_date;
}
public void setTermination_date(Date termination_date) {
        this.termination_date = termination_date;
}
public Date getBirth_date() {
        return birth_date;
}
public void setBirth_date(Date birth_date) {
        this.birth_date = birth_date;
}
public String getBene_health_ins() {
```

```
                return bene_health_ins;
        }
        public void setBene_health_ins(String bene_health_ins) {
                this.bene_health_ins = bene_health_ins;
        }
        public String getBene_life_ins() {
                return bene_life_ins;
        }
        public void setBene_life_ins(String bene_life_ins) {
                this.bene_life_ins = bene_life_ins;
        }
        public String getBene_day_care() {
                return bene_day_care;
        }
        public void setBene_day_care(String bene_day_care) {
                this.bene_day_care = bene_day_care;
        }
        public String getSex() {
                return sex;
        }
        public void setSex(String sex) {
                this.sex = sex;
        }
}
```

While writing your Java DTO, don't forget to add the annotation `@FxClass` of the class `EmployeeDTO`. It's required for generatin the ActionScript class `EmployeeDTO.as`.

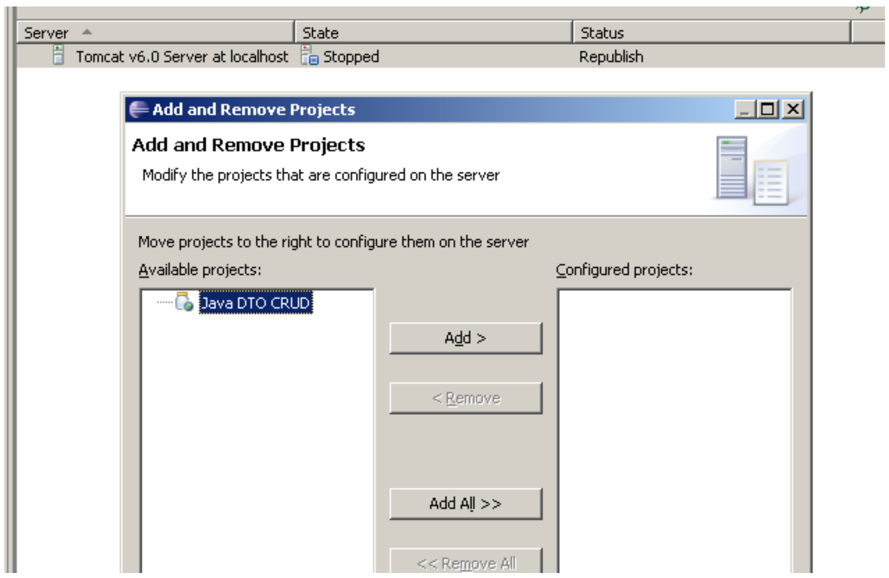


```
package com.farata.test {

  [RemoteClass(alias="com.farata.test.EmployeeDTO")]

  public class EmployeeDTO extends com.farata.test._EmployeeDTO {

    public var property:String;
    /* Constructor */
    public function EmployeeDTO():void {
      super();
    }

  }

}
```

```
    [javac] Compiling 2 source files to D:\CLEAR_DATA_BUILDER\runtime-New_con
   [delete] Deleting: D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Java DT
      [jar] Building jar: D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Jav
      [jar] Updating jar: D:\CLEAR_DATA_BUILDER\runtime-New_configuration\Jav
```

CDB includes yet another plugin DTO2Fx that is described in details at http://flexblog.faratasystems.com/?p=357.

In Eclipse, add your project to Tomcat in the server view. Right-click on the Tomcat Server entry, select Add or Remove projects, and add the project Java DTO CRUD to the Configured projects panel.


Server | State | Status
Tomcat v6.0 Server at localhost | Stopped | Republish

Add and Remove Projects

Add and Remove Projects

Modify the projects that are configured on the server

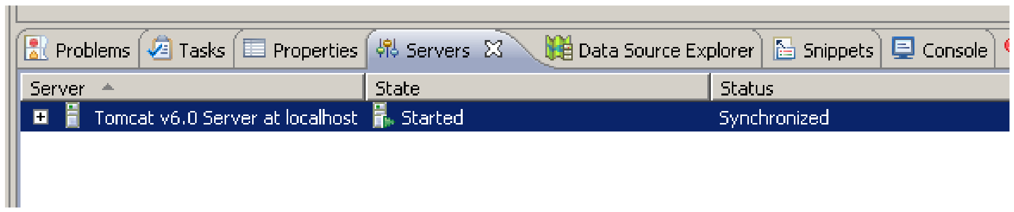
Move projects to the right to configure them on the server

Available projects:

Java DTO CRUD

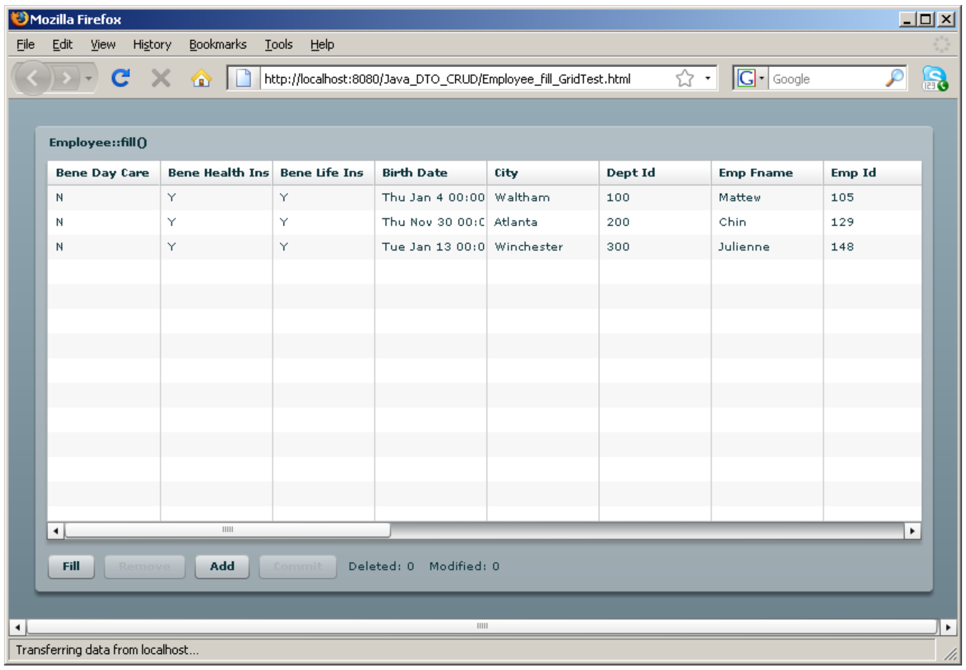
Configured projects:

Right-click again and start the server.



Server	State	Status
Tomcat v6.0 Server at localhost	Started	Synchronized

Switch to the Flex Perspective and copy the file Employee_fill_GridTest.mxml from test/rpc.com.farata.test to flex_src directory, set it as a default application and run it.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/Java_DTO_CRUD/Employee_fill_GridTest.html
Google
Employee::fill()
Fill
Remove
Add
Commit
Deleted: 0
Modified: 0
Transferring data from localhost...

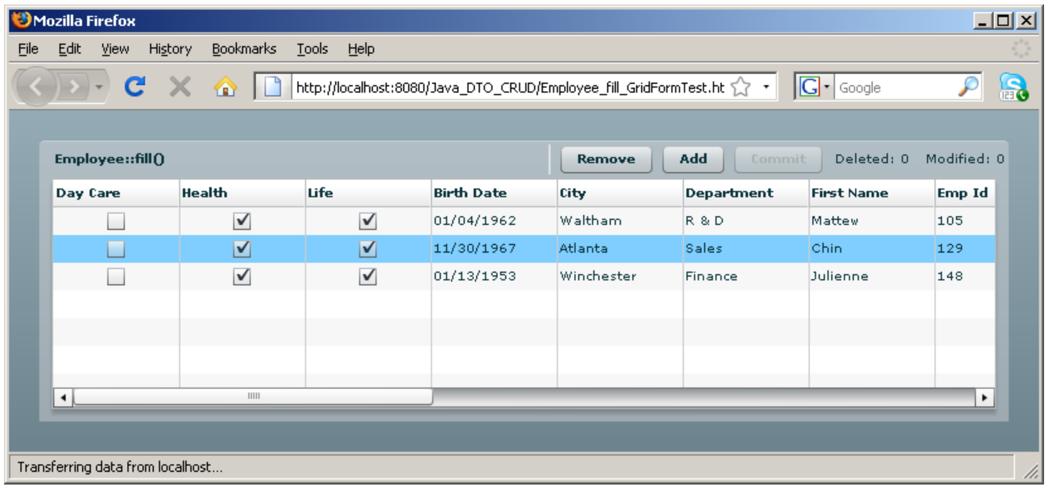
Bene Day Care	Bene Health Ins	Bene Life Ins	Birth Date	City	Dept Id	Emp Fname	Emp Id
N	Y	Y	Thu Jan 4 00:00	Waltham	100	Mattew	105
N	Y	Y	Thu Nov 30 00:0	Atlanta	200	Chin	129
N	Y	Y	Tue Jan 13 00:0	Winchester	300	Julienne	148

The basic Java DTO CRUD application is ready. For testing purposes, remove the first row in the grid, modify Atlanta to New York in the second, and press the button Commit. Eclipse console properly shows that Java code received the changes. Here's how the console view may look like:

```
fill method has executed
doDelete method executing
Deleted: 105
doUpdate method executing
Changed: city, Old Value: Atlanta, New Value: New York
sync method has executed
```

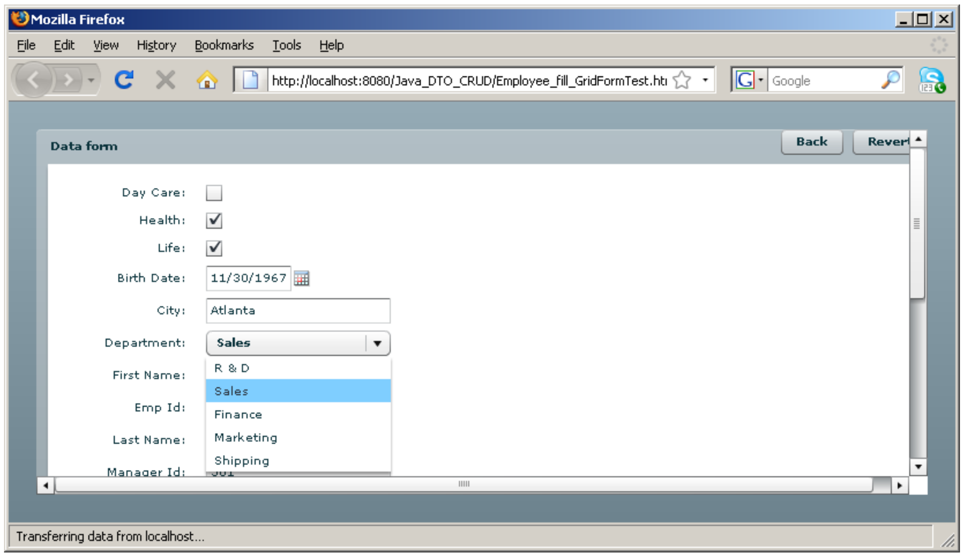
You can easily identify the `System.out.prinln()` statements in the assembler class that resulted in this output.

Try to run another generated test application called Employee_fill_GridFormTest.mxml and you'll see checkboxes used as item renderers:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/Java_DTO_CRUD/Employee_fill_GridFormTest.ht
Google
Employee::fill()
Remove
Add
Commit
Deleted: 0
Modified: 0
Transferring data from localhost...

Day Care	Health	Life	Birth Date	City	Department	First Name	Emp Id
☐	✓	✓	01/04/1962	Waltham	R & D	Mattew	105
☐	✓	✓	11/30/1967	Atlanta	Sales	Chin	129
☐	✓	✓	01/13/1953	Winchester	Finance	Julienne	148

Double-click on a grid row, and you'll see a data form with automatically populated combobox Department (yes this is what `DepartmentDTO` class was for).


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/Java_DTO_CRUD/Employee_fill_GridFormTest.htı
Google
Data form
Back
Revert

Day Care:
Health: ✓
Life: ✓
Birth Date: 11/30/1967
City: Atlanta
Department: Sales
First Name:
Emp Id:
Last Name:
Manager Id:

R & D
Sales
Finance
Marketing
Shipping

Transferring data from localhost...

At this point you should examine the generated Flex code to learn how CDB empowered by rich components from the clear.swc library.

3.0 SQL-BASED CODE GENERATION

In this section you'll see how to create a sample application based on available SQL or a stored procedure. In this case CDB requires configured JDBC connection, a sample database and a simple abstract Java class annotated with the information about your data source, for example:

```
package com.farata;
import java.util.List;

  /**
  * @daoflex:webservice pool=jdbc/db2_sample
  */
  public abstract class Employee {
    /**
    * @daoflex:sql
    *  pool=jdbc/db2_sample
    *  sql=::select * from yf.EMPLOYEE
    *  ::
    *  transferType=EmployeeDTO[]
    *  keyColumns=EMPNO
      *  updateTable=yf.EMPLOYEE
    */
      public abstract List getEmployees();
  }
```

CDB is not only a code generator, but it also builds and deploys all required files into the Web application directory of yuour choice. This application supports Create, Read, Update, and Delete functionality, a.k.a. CRUD. You can use it as a foundation for your next Flex project that needs to communicate with a DBMS via Flex remoting (the AMF protocol).

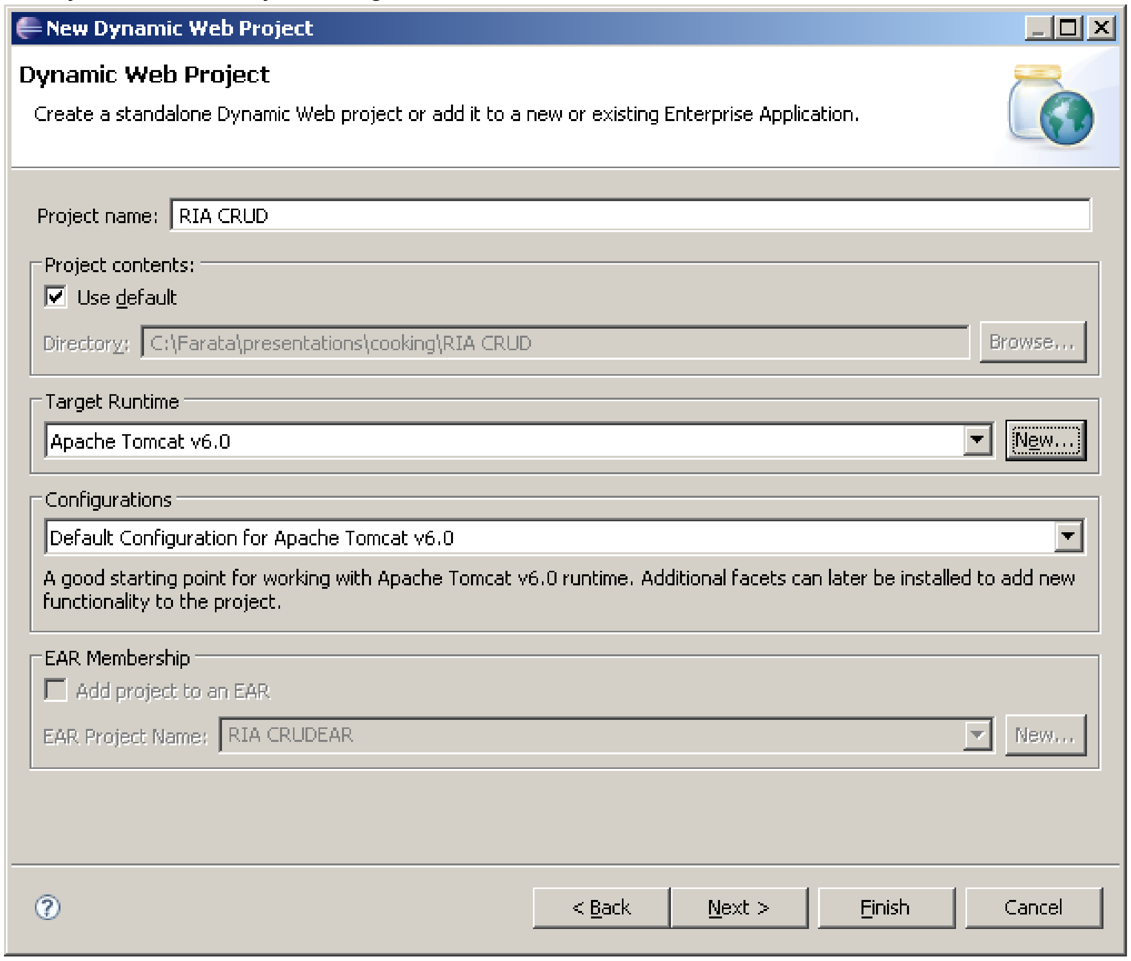
All required source code is generated based on the XSL templates that come with CDB. In addition to these templates, the users of CDB can create their own, if need be.

IBM offers a free version of DB2 Express that works just fine on your PC and we'll use it here. During the installation you'll need to pick a user name and the password (we've selected dba as a user id and sql as a password).

3.1 Creating CRUD application

1. Start Eclipse JEE and create a new instance of the Tomcat 6 server. Right-click in the tab Servers, select New, Tomcat 6, and then click on Finish.

2. Create a new Dynamic Web Project in Eclipse using the menus File | New | Other | Web | Dynamic Web Project. Let's name the project RIA CRUD. Specify the Target Runtime (we'll use Apache Tomcat 6.0) in the Dynamic Web Project configuration screen:


New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: RIA CRUD

Project contents:
☑ Use default
Directory: C:\Farata\presentations\cooking\RIA CRUD

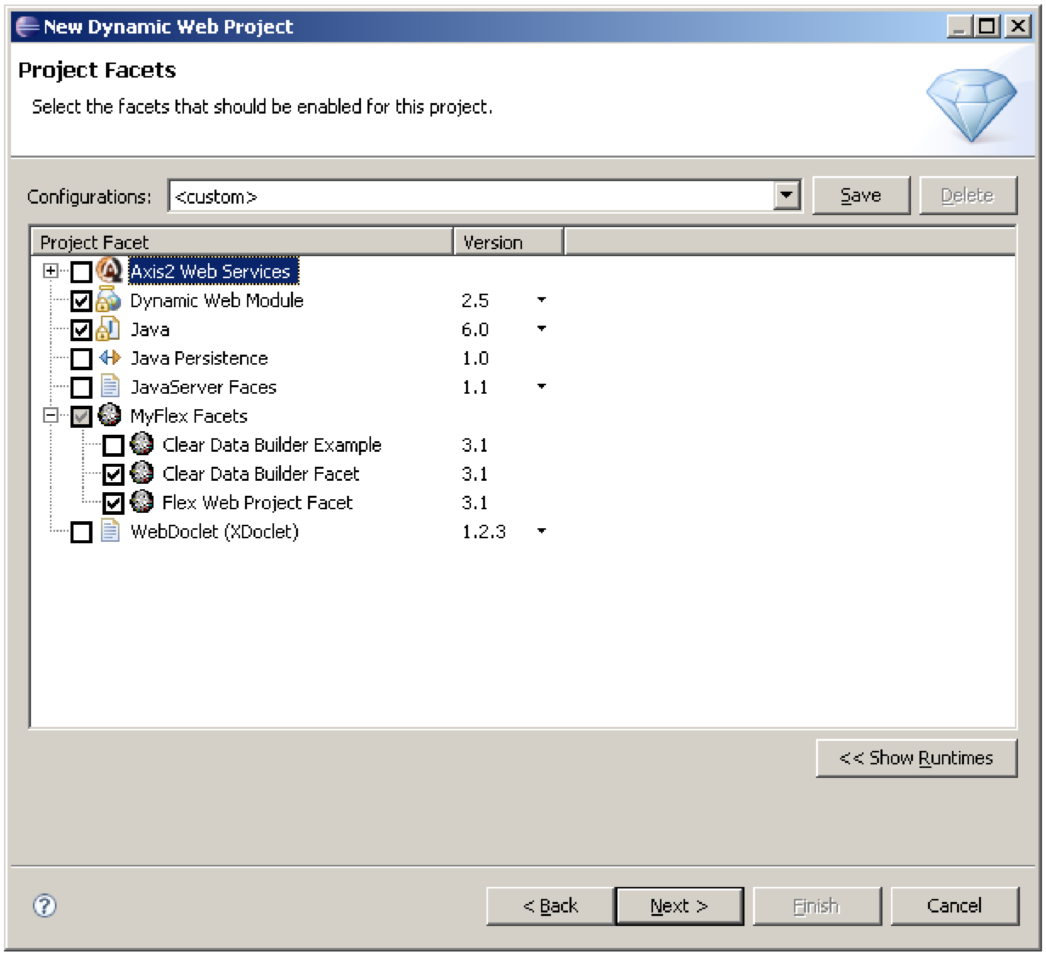
Target Runtime
Apache Tomcat v6.0

Configurations
Default Configuration for Apache Tomcat v6.0
A good starting point for working with Apache Tomcat v6.0 runtime. Additional facets can later be installed to add new functionality to the project.

EAR Membership
☐ Add project to an EAR
EAR Project Name: RIA CRUDEAR

Press the button Next.

3. Select Clear Data Builder and Flex Web Project facets as shown below:


New Dynamic Web Project

Project Facets

Select the facets that should be enabled for this project.

Configurations: <custom>

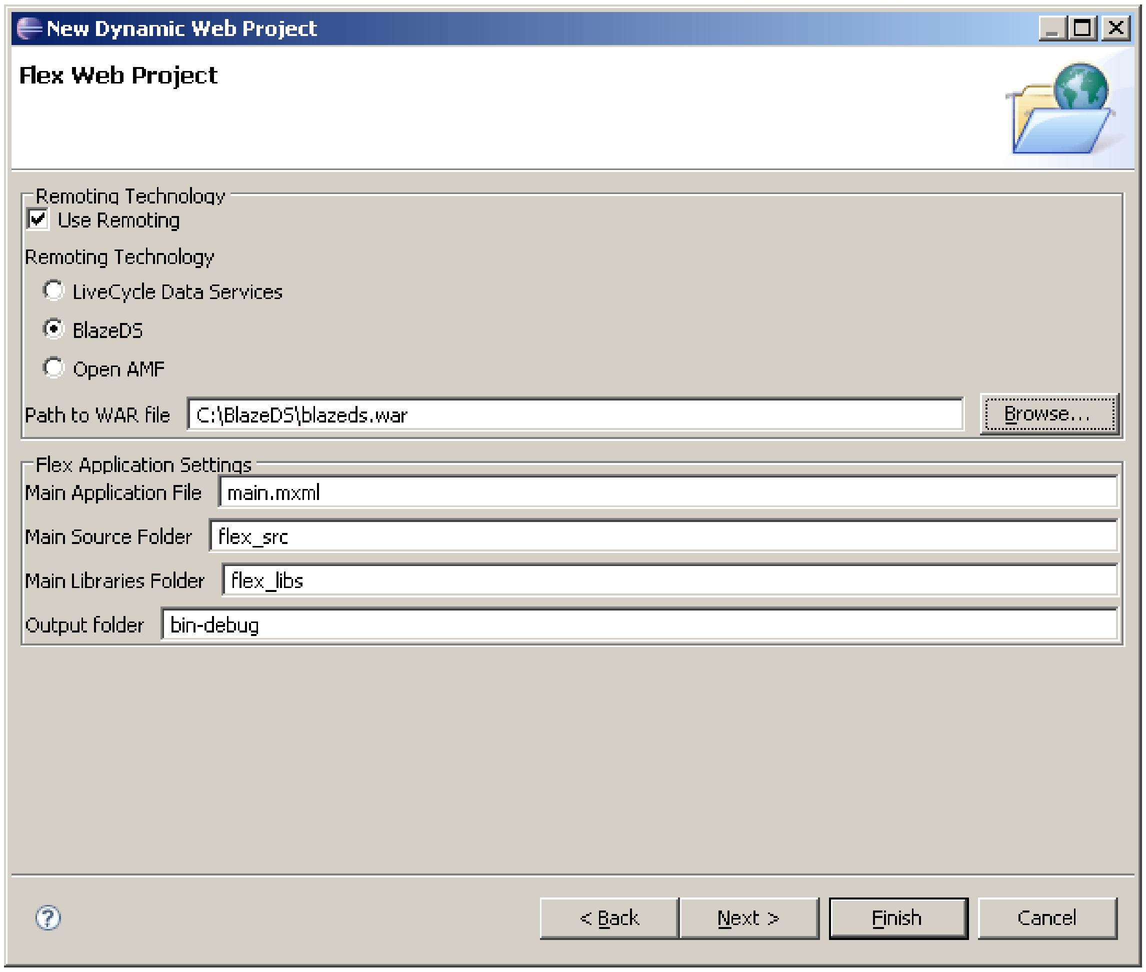
Project Facet	Version
☐ Axis2 Web Services	
☑ Dynamic Web Module	2.5
☑ Java	6.0
☐ Java Persistence	1.0
☐ JavaServer Faces	1.1
☑ MyFlex Facets	
☐ Clear Data Builder Example	3.1
☑ Clear Data Builder Facet	3.1
☑ Flex Web Project Facet	3.1
☐ WebDoclet (XDoclet)	1.2.3

If you do not see CDB facets on this screen, this means you did not install CDB license.

Press Next.

Leave the next screen the suggests RIA_CRUD as a context, WebContent as a content directory and src as a directory for the Java code. Press the button Next.

Now you need to specify that we are going to use BlazeDS and where your blazeds.war is located:



Flex Web Project

Remoting Technology

[✓] Use Remoting

Remoting Technology

- ○ LiveCycle Data Services
- ● BlazeDS
- ○ Open AMF

Path to WAR file: C:\BlazeDS\blazeds.war

Flex Application Settings

Main Application File: main.mxml

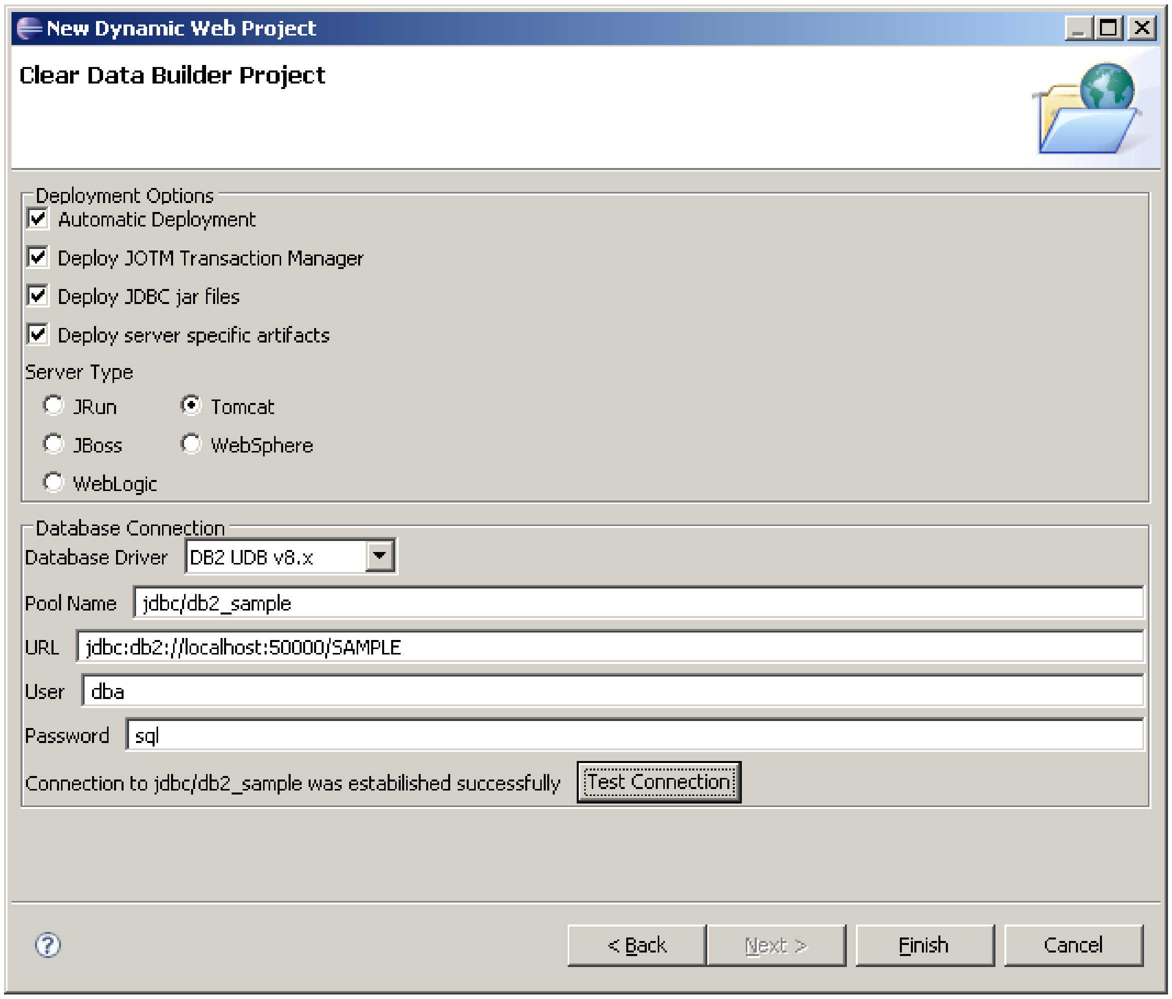
Main Source Folder: flex_src

Main Libraries Folder: flex_libs

Output folder: bin-debug

Press the button Next.

Finally, we need to specify that the application will be deployed under Tomcat, select and configure the database connection . Select the database driver, name the connection pool, and enter the URL of your database. By default, DB2 is running on port 5000 and the name of the sample database is SAMPLE.


New Dynamic Web Project

Clear Data Builder Project

Deployment Options
[✓] Automatic Deployment
[✓] Deploy JOTM Transaction Manager
[✓] Deploy JDBC jar files
[✓] Deploy server specific artifacts

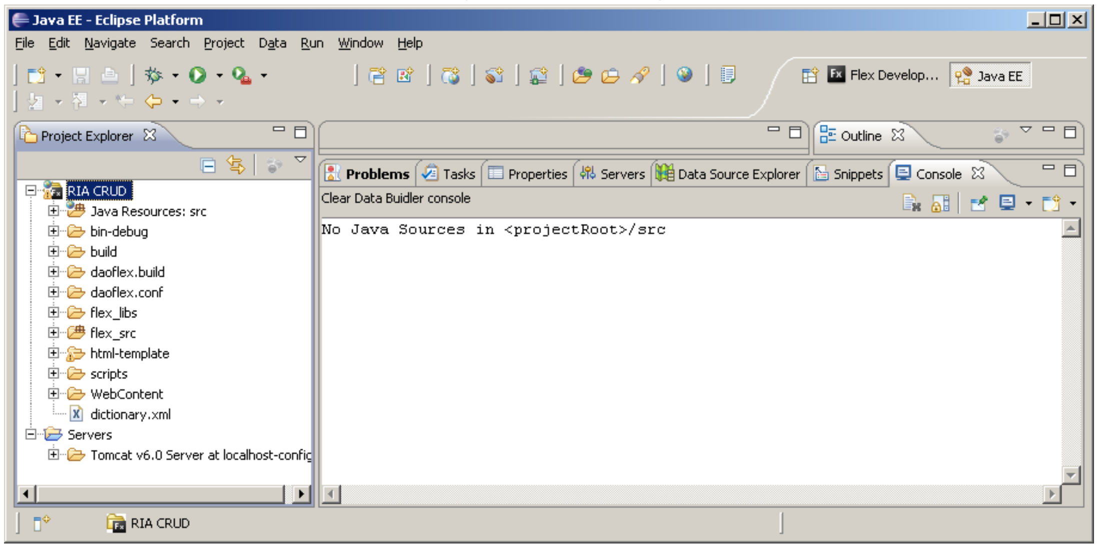
Server Type
JRun [● Tomcat] JBoss WebSphere WebLogic

Database Connection
Database Driver: DB2 UDB v8.x
Pool Name: jdbc/db2_sample
URL: jdbc:db2://localhost:50000/SAMPLE
User: dba
Password: sql

Connection to jdbc/db2_sample was estabilished successfully

Do not forget to press the button Test Connection to ensure that there is no problems in that department.

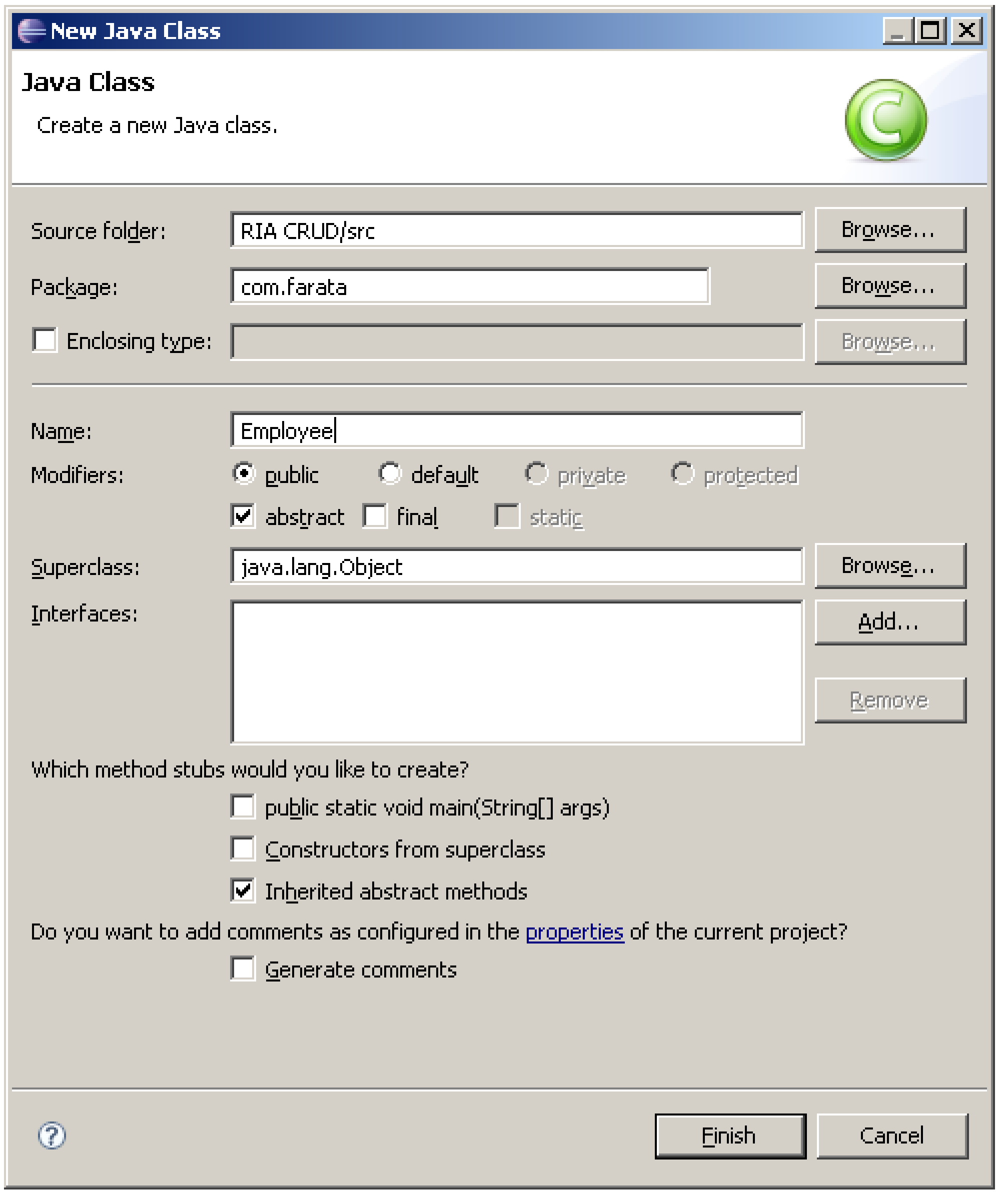
Press the button Finish, and the new Dynamic Web Project will be created:


Java EE - Eclipse Platform
File Edit Navigate Search Project Data Run Window Help
Flex Develop... Java EE
Project Explorer
RIA CRUD
Java Resources: src
bin-debug
build
daoflex.build
daoflex.conf
flex_libs
flex_src
html-template
scripts
WebContent
dictionary.xml
Servers
Tomcat v6.0 Server at localhost-config
Outline
Problems Tasks Properties Servers Data Source Explorer Snippets Console
RIA CRUD

Clear Data Buidler console

```
No Java Sources in <projectRoot>/src
```

Create a new abstract class Employee (right-click on Java Resources). Let's enter com.farata as the package name:


New Java Class

Java Class

Create a new Java class.

Source folder: RIA CRUD/src

Package: com.farata

☐ Enclosing type:

Name: Employee

Modifiers: ● public ○ default ○ private ○ protected

☑ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments as configured in the properties of the current project?

☐ Generate comments

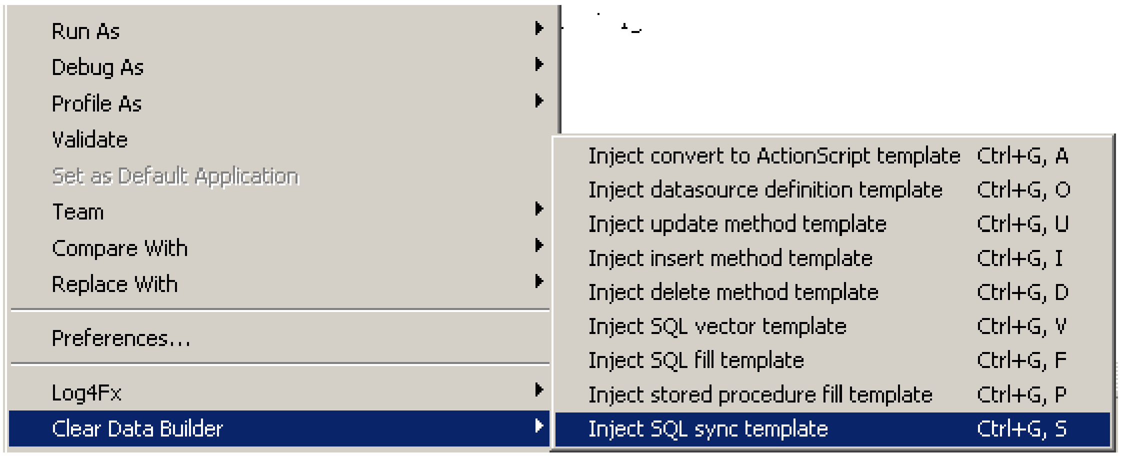
The code of the class Employee will look like this:

```
package com.farata;

public abstract class Employee {

}
```

It’s time to use Clear Data Builder’s code generator, but first we need to declare a method annotated (we use doclets) with our SQL statement that will bring the data to our CRUD application. We’ll need a couple of more lines of code to specify what’s the table to update and what’s the primary key there. The resulting class may look like this:

```
package com.farata;
import java.util.List;

/**
* @daoflex:webservice pool=jdbc/db2_sample
*/
public abstract class Employee {
            /**
            * @daoflex:sql
            *  pool=jdbc/db2_sample
            *  sql=::select * from yf.EMPLOYEE
            *  ::
            *  transferType=EmployeeDTO[]
            *  keyColumns=EMPNO
      *  updateTable=yf.EMPLOYEE
            */
     public abstract List getEmployees();
}
```

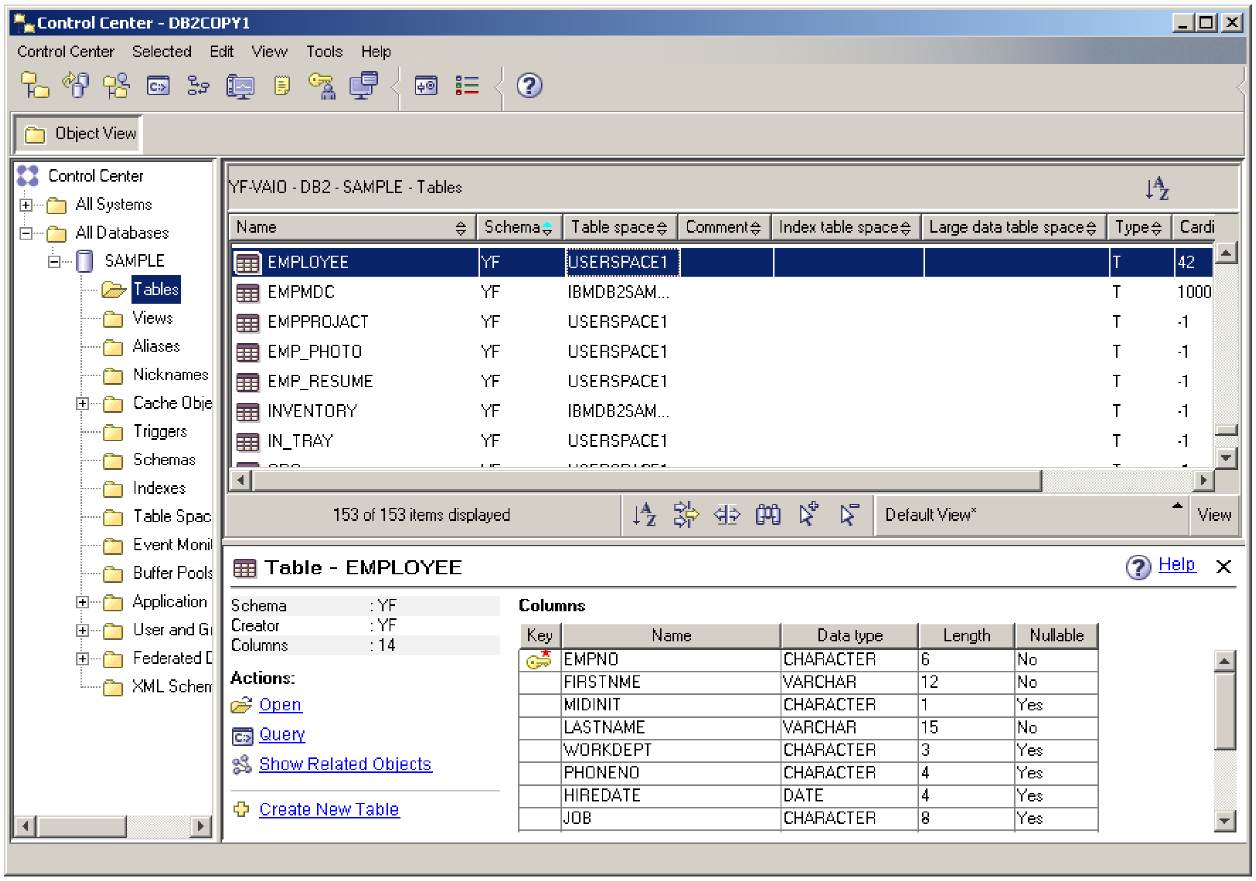
Double colons are used to specify the start and end of the SQL statement. But since it's your first exposure to CDB, right-click inside the curlies in the class Employee, and you'll see a Clear Data Builder's menu.



Run As
Debug As
Profile As
Validate
Set as Default Application
Team
Compare With
Replace With
Preferences...
Log4Fx
Clear Data Builder
Inject convert to ActionScript template Ctrl+G, A
Inject datasource definition template Ctrl+G, O
Inject update method template Ctrl+G, U
Inject insert method template Ctrl+G, I
Inject delete method template Ctrl+G, D
Inject SQL vector template Ctrl+G, V
Inject SQL fill template Ctrl+G, F
Inject stored procedure fill template Ctrl+G, P
Inject SQL sync template Ctrl+G, S

Select Inject SQL sync template if you need a CRUD application or Inject SQL fill template if you are planning to create a read-only application. CDB will insert commented code that will help you to write similar code on your own.

Here's a screenshot of the Control Center of the DB2 SAMPLE database. Please note, that on my PC the table EMPLOYEE was created in the schema YF.



YF-VAIO - DB2 - SAMPLE - Tables

Name	Schema	Table space	Comment	Index table space	Large data table space	Type	Cardi
EMPLOYEE	YF	USERSPACE1				T	42
EMPMDC	YF	IBMDB2SAM...				T	1000
EMPPROJACT	YF	USERSPACE1				T	-1
EMP_PHOTO	YF	USERSPACE1				T	-1
EMP_RESUME	YF	USERSPACE1				T	-1
INVENTORY	YF	IBMDB2SAM...				T	-1
IN_TRAY	YF	USERSPACE1				T	-1

153 of 153 items displayed

Default View*

View

Table - EMPLOYEE

Help

Schema : YF
Creator : YF
Columns : 14

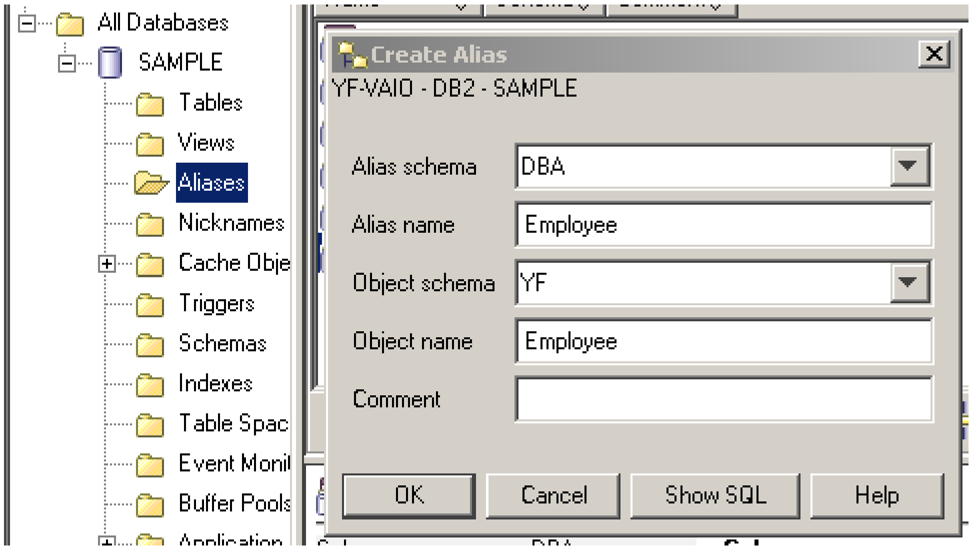
Actions:
Open
Query
Show Related Objects
Create New Table

Columns

Key	Name	Data type	Length	Nullable
🔑	EMPNO	CHARACTER	6	No
	FIRSTNME	VARCHAR	12	No
	MIDINIT	CHARACTER	1	Yes
	LASTNAME	VARCHAR	15	No
	WORKDEPT	CHARACTER	3	Yes
	PHONENO	CHARACTER	4	Yes
	HIREDATE	DATE	4	Yes
	JOB	CHARACTER	8	Yes

Since we configure the database connection pool using dba as a user id, we'd need to fully qualify the table name: select * from yf.employee. Accordingly, the name of the table to update should be also fully qualified.

To eliminate this annoyance, you can use DB2 Control Center and create an alias Employee in the dba schema to point at the table Employee in the schema YF:



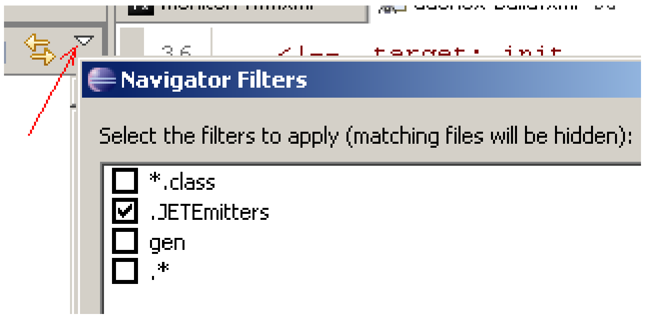
YF-VAIO - DB2 - SAMPLE

Alias schema	DBA
Alias name	Employee
Object schema	YF
Object name	Employee
Comment	

Now we can go to the Eclipse menu Project an perform Clean , which will start the CDB code generation and build process. The Clean process invokes the ANT build script located under the folder daoflex.build. The only proper outcome of this process is BUILD SUCCESSFUL. If you do not see this message, most likely you've done something wrong or in the wrong order.

After this build, the Java DTO and data access classes where generated and deployed in our Tomcat servlet container.

To see the Java code generated by CDB, switch to the Flex Builder perspective, click on a little triangle in the Flex Navigator view, select the Filters option and uncheck the "gen" checkbox as shown below:



Select the filters to apply (matching files will be hidden):

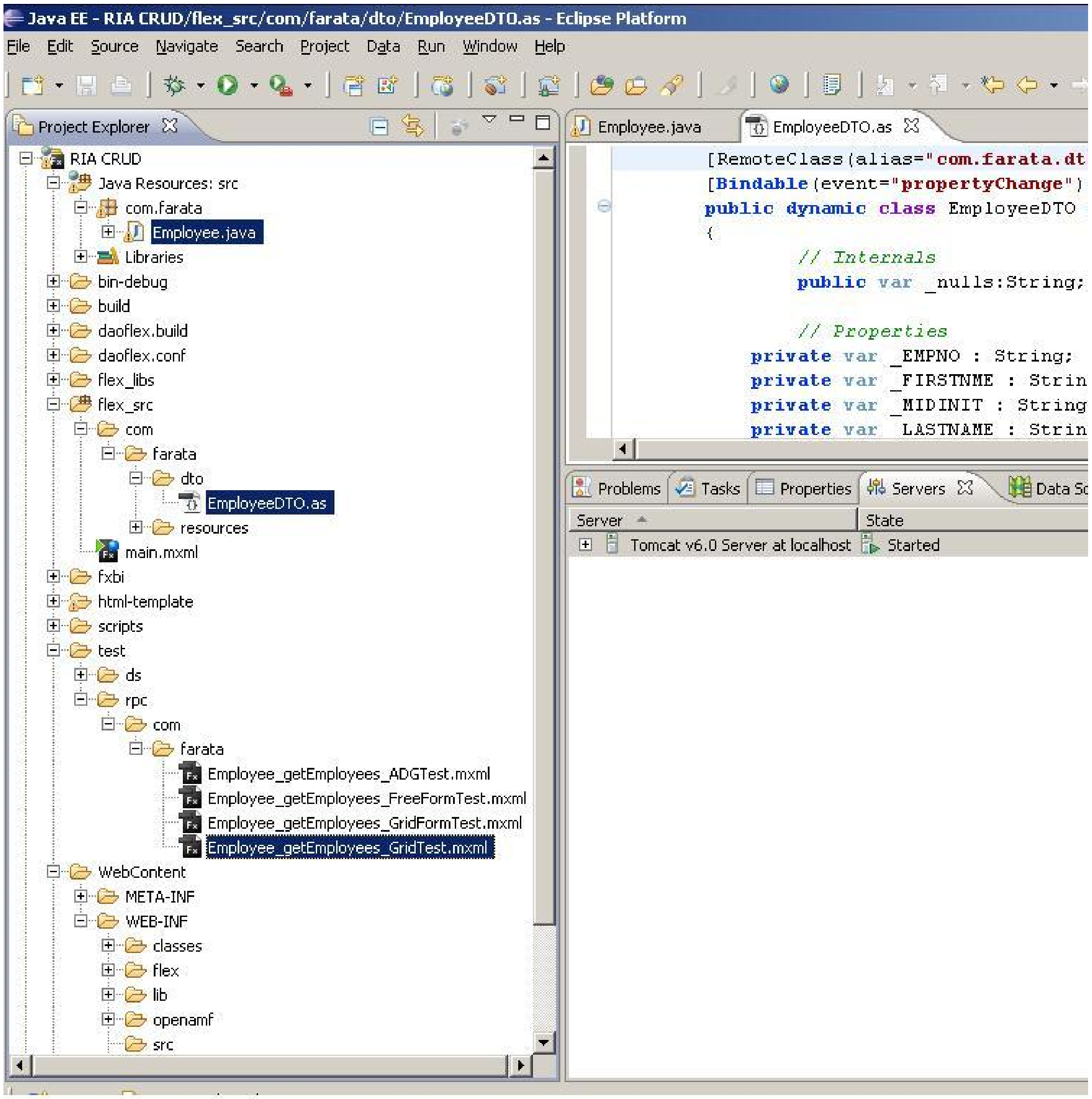
- [ ] *.class
- [x] .JETEmitters
- [ ] gen
- [ ] .*

The generated Java code is located under the folder .daoflex-temp\gen. Re-run the Ant script to regenerate the code. If you do not see this folder immediately, refresh your Eclipse project by pressing F5.

Technically, you do not need keep these source files as they are going to be jar'ed by CDB build process and deployed in the lib directory of your servlet container under WEB-INF\lib in the files daoflex-runtime.jar, services-generated.jar and services-original.jar.

On the client side, CDB has generated the EmployeeDTO.as, an ActionScript peer of the generated EmployeeDTO.java. It's shown on the next screen shot.

Clear Data Builder also generates a number of test Flex applications, which can be used as the front end of our RIA CRUD application.


Java EE - RIA CRUD/flex_src/com/farata/dto/EmployeeDTO.as - Eclipse Platform
File Edit Source Navigate Search Project Data Run Window Help
Project Explorer
Employee.java
EmployeeDTO.as
Problems
Tasks
Properties
Servers
Data Sc

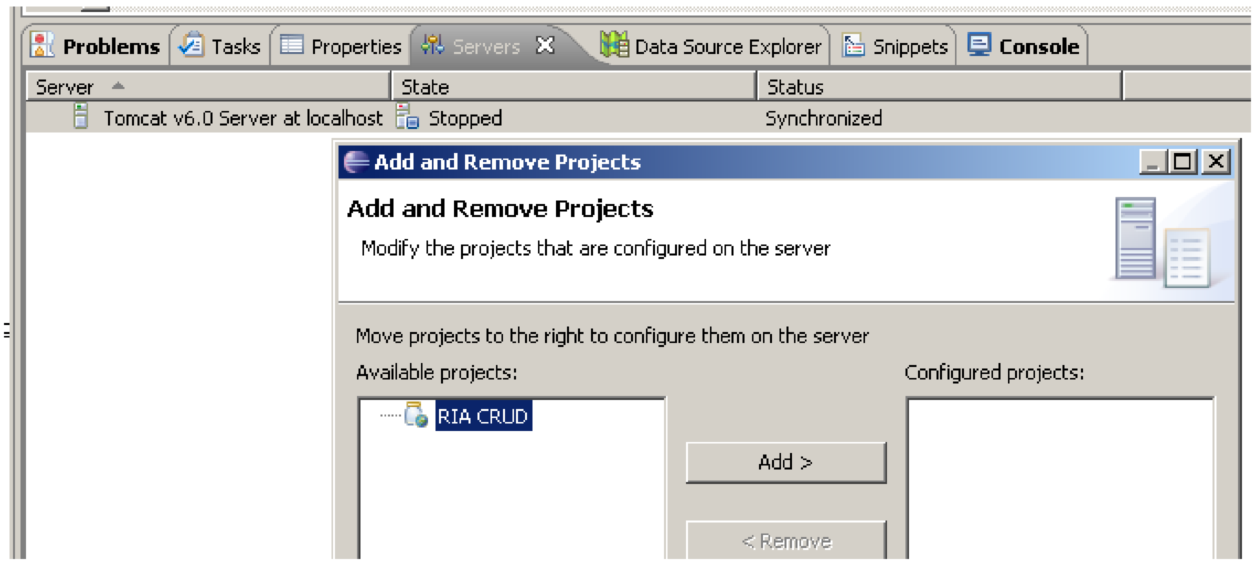
```
RIA CRUD
  Java Resources: src
    com.farata
      Employee.java
    Libraries
  bin-debug
  build
  daoflex.build
  daoflex.conf
  flex_libs
  flex_src
    com
      farata
        dto
          EmployeeDTO.as
        resources
    main.mxml
  fxbi
  html-template
  scripts
  test
    ds
    rpc
      com
        farata
          Employee_getEmployees_ADGTest.mxml
          Employee_getEmployees_FreeFormTest.mxml
          Employee_getEmployees_GridFormTest.mxml
          Employee_getEmployees_GridTest.mxml
  WebContent
    META-INF
    WEB-INF
      classes
      flex
      lib
      openamf
      src
```

```
[RemoteClass(alias="com.farata.dt
[Bindable(event="propertyChange")
public dynamic class EmployeeDTO
{
        // Internals
        public var _nulls:String;

        // Properties
    private var _EMPNO : String;
    private var _FIRSTNME : Strin
    private var _MIDINIT : String
    private var _LASTNAME : Strin
```

Server	State
Tomcat v6.0 Server at localhost	Started

We'll use the one of the generated Flex applications called Employee_getEmployees_GridTest.mxml shortly.

Add your project to Tomcat server: right click on your Tomcat Server, select Add or Remove projects, and add the project RIA CRUD to the Configured projects panel.



Server	State	Status
Tomcat v6.0 Server at localhost	Stopped	Synchronized

Add and Remove Projects

Add and Remove Projects

Modify the projects that are configured on the server

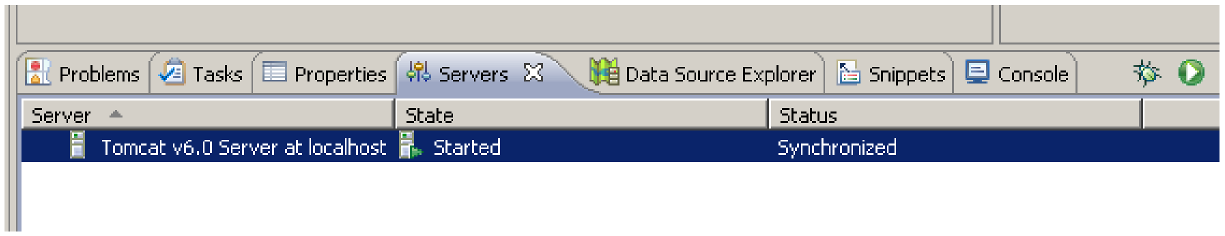
Move projects to the right to configure them on the server

Available projects:

RIA CRUD

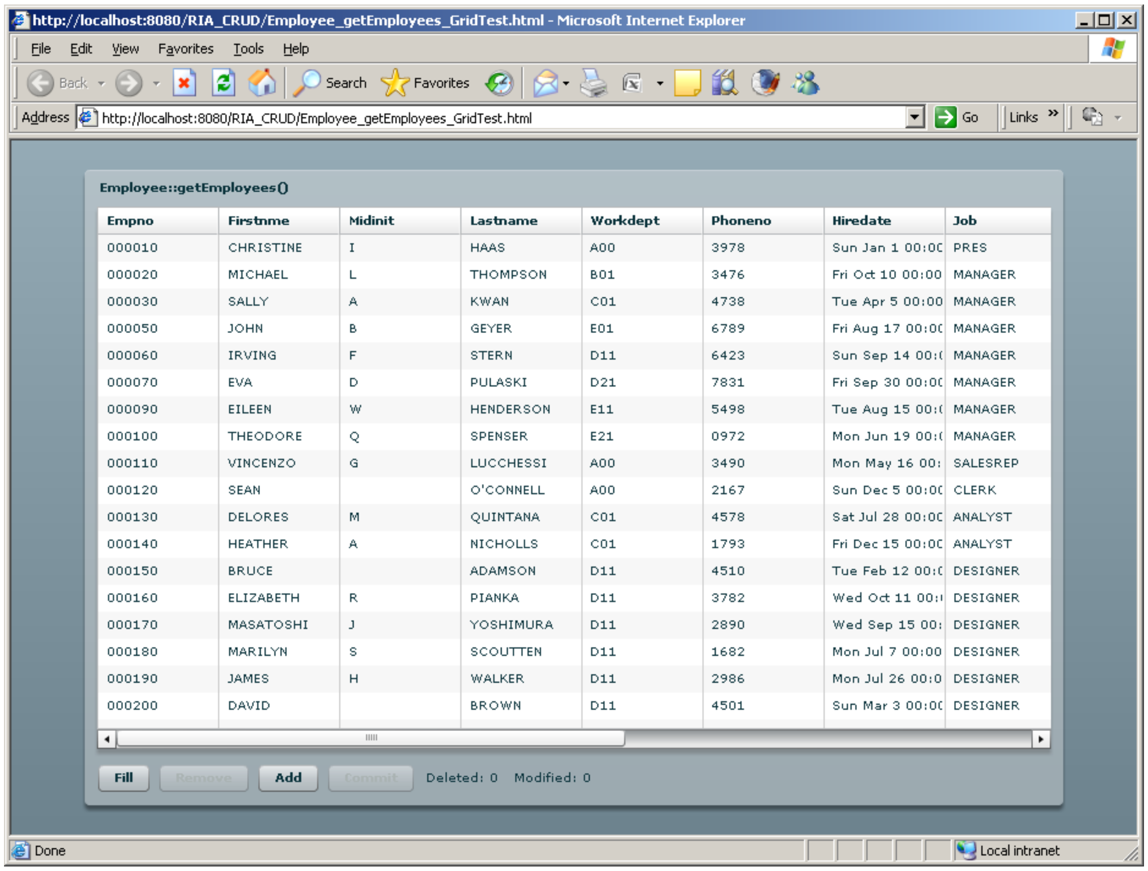
Configured projects:

Start the server by using its right-click menu.



Server	State	Status
Tomcat v6.0 Server at localhost	Started	Synchronized

Switch to the Flex Perspective, and copy the file Employee_getEmployees_GridTest.mxml from test.rpc.com.farata to flex_src directory, set it as a default application, right-click and run it as Flex application.


http://localhost:8080/RIA_CRUD/Employee_getEmployees_GridTest.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/RIA_CRUD/Employee_getEmployees_GridTest.html
Go Links
Employee::getEmployees()
Fill Remove Add Commit
Done
Local intranet

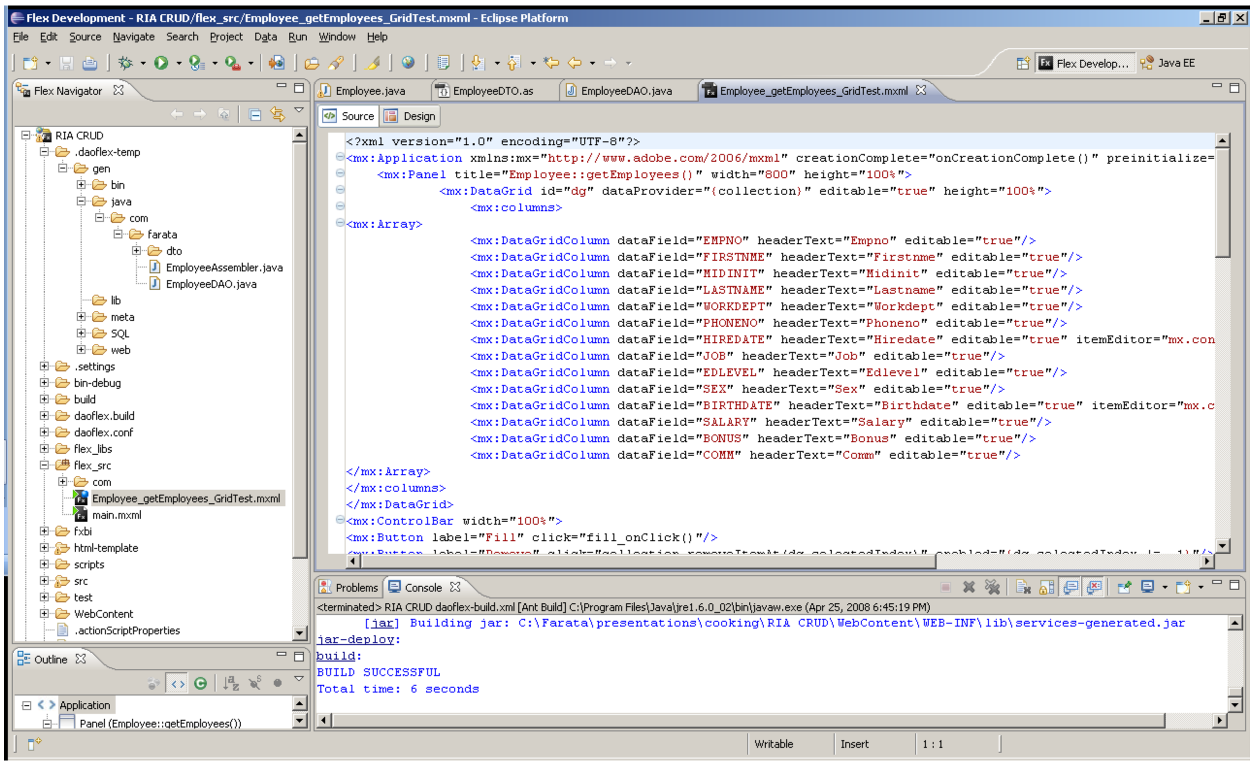
Empno	Firstnme	Midinit	Lastname	Workdept	Phoneno	Hiredate	Job
000010	CHRISTINE	I	HAAS	A00	3978	Sun Jan 1 00:00	PRES
000020	MICHAEL	L	THOMPSON	B01	3476	Fri Oct 10 00:00	MANAGER
000030	SALLY	A	KWAN	C01	4738	Tue Apr 5 00:00	MANAGER
000050	JOHN	B	GEYER	E01	6789	Fri Aug 17 00:00	MANAGER
000060	IRVING	F	STERN	D11	6423	Sun Sep 14 00:0	MANAGER
000070	EVA	D	PULASKI	D21	7831	Fri Sep 30 00:00	MANAGER
000090	EILEEN	W	HENDERSON	E11	5498	Tue Aug 15 00:0	MANAGER
000100	THEODORE	Q	SPENSER	E21	0972	Mon Jun 19 00:0	MANAGER
000110	VINCENZO	G	LUCCHESSI	A00	3490	Mon May 16 00:	SALESREP
000120	SEAN		O'CONNELL	A00	2167	Sun Dec 5 00:00	CLERK
000130	DELORES	M	QUINTANA	C01	4578	Sat Jul 28 00:00	ANALYST
000140	HEATHER	A	NICHOLLS	C01	1793	Fri Dec 15 00:00	ANALYST
000150	BRUCE		ADAMSON	D11	4510	Tue Feb 12 00:0	DESIGNER
000160	ELIZABETH	R	PIANKA	D11	3782	Wed Oct 11 00:0	DESIGNER
000170	MASATOSHI	J	YOSHIMURA	D11	2890	Wed Sep 15 00:	DESIGNER
000180	MARILYN	S	SCOUTTEN	D11	1682	Mon Jul 7 00:00	DESIGNER
000190	JAMES	H	WALKER	D11	2986	Mon Jul 26 00:0	DESIGNER
000200	DAVID		BROWN	D11	4501	Sun Mar 3 00:00	DESIGNER

Deleted: 0 Modified: 0

The basic RIA CRUD application is ready.

To test the database updates, modify, say the last name of an employee. After you tab out of the last name field, the button **Commit** becomes enabled, and you'll be able to apply your changes to the database.
The button **Fill** is for data retrieval from the server side.
The **Add** button is for adding new data.
The button **Remove** becomes enabled when you select a row in the data grid.

The source code of our RIA CRUD application is available, and you can use it as a foundation of your future project just add the required components to the code shown below.



```
<?xml version="1.0" encoding="UTF-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="onCreationComplete()" preinitialize=
    <mx:Panel title="Employee::getEmployees()" width="800" height="100%">
            <mx:DataGrid id="dg" dataProvider="{collection}" editable="true" height="100%">
                <mx:columns>
<mx:Array>
                <mx:DataGridColumn dataField="EMPNO" headerText="Empno" editable="true"/>
                <mx:DataGridColumn dataField="FIRSTNME" headerText="Firstnme" editable="true"/>
                <mx:DataGridColumn dataField="MIDINIT" headerText="Midinit" editable="true"/>
                <mx:DataGridColumn dataField="LASTNAME" headerText="Lastname" editable="true"/>
                <mx:DataGridColumn dataField="WORKDEPT" headerText="Workdept" editable="true"/>
                <mx:DataGridColumn dataField="PHONENO" headerText="Phoneno" editable="true"/>
                <mx:DataGridColumn dataField="HIREDATE" headerText="Hiredate" editable="true" itemEditor="mx.con
                <mx:DataGridColumn dataField="JOB" headerText="Job" editable="true"/>
                <mx:DataGridColumn dataField="EDLEVEL" headerText="Edlevel" editable="true"/>
                <mx:DataGridColumn dataField="SEX" headerText="Sex" editable="true"/>
                <mx:DataGridColumn dataField="BIRTHDATE" headerText="Birthdate" editable="true" itemEditor="mx.c
                <mx:DataGridColumn dataField="SALARY" headerText="Salary" editable="true"/>
                <mx:DataGridColumn dataField="BONUS" headerText="Bonus" editable="true"/>
                <mx:DataGridColumn dataField="COMM" headerText="Comm" editable="true"/>
</mx:Array>
</mx:columns>
</mx:DataGrid>
<mx:ControlBar width="100%">
<mx:Button label="Fill" click="fill_onClick()"/>
```

```
<terminated> RIA CRUD daoflex-build.xml [Ant Build] C:\Program Files\Java\jre1.6.0_02\bin\javaw.exe (Apr 25, 2008 6:45:19 PM)
      [jar] Building jar: C:\Farata\presentations\cooking\RIA CRUD\WebContent\WEB-INF\lib\services-generated.jar
jar-deploy:
build:
BUILD SUCCESSFUL
Total time: 6 seconds
```

The server-side code is deployed under Tomcat server, and the relevant classes are located under the folder WebContent in your project. While generating this project, CDB has added a library component.swc to the build path. It contains the class library called *theriabook* created by Farata Systems. It includes a number of handy components that enhance standard controls of Flex framework and a number of classes simplifying communication with the database layer.

theriabook is a non-intrusive class library allows you to mix and match original Flex components with the ones that come with Flex framework. The difference between *theriabook* and all other Flex frameworks is that

a) it's not a framework but a class library;

b) while other frameworks require you to add some extra code to your application, theriabook makes your codebase smaller.

For example, the following auto-generated code from Employee_getEmployees_GridTest.mxml uses an object DataCollection from theriabook, which is nothing else but a subclass of Flex class ArrayCollection. Look at the code in the onCreationComplete() function below. DataCollection is a smart data-aware class that combines the functionality of Flex ArrayCollection, RemoteObject and some functionality of the Data Management Services that are available in LiveCycle Data Services. Just set the destination and the method to call, and call it's methods fill() or sync(). No need to define the RemoteObject with result and fault handlers, no server-side configuration is required.

```
<mx:Button label="Fill" click="fill_onClick()"/>
<mx:Button label="Remove" click="collection.removeItemAt(dg.selectedIndex)"
enabled="{dg.selectedIndex != -1}"/>
<mx:Button label="Add" click="addItemAt(Math.max(0,dg.selectedIndex+1)) "/>
<mx:Button label="Commit" click="collection.sync()"
enabled="{collection.commitRequired}"/>
…
        import com.theriabook.rpc.remoting.*;
        import com.theriabook.collections.DataCollection;
        import mx.collections.ArrayCollection;
        import mx.controls.dataGridClasses.DataGridColumn;
        import mx.events.CollectionEvent;
        import mx.formatters.DateFormatter;

        import com.farata.dto.EmployeeDTO;

                  [Bindable]
        public var collection:DataCollection ;
        [Bindable]
        private var log : ArrayCollection;

        private function onCreationComplete() : void {
              collection = new DataCollection();
              collection.destination="com.farata.Employee";
              collection.method="getEmployees";
              //getEmployees_sync is the default for collection.syncMethod
              log = new ArrayCollection();
              collection.addEventListener( CollectionEvent.COLLECTION_CHANGE,
                                                               logEvent);
              collection.addEventListener("fault", logEvent);
              fill_onClick();
        }
        private function fill_onClick():void {
              collection.fill();
        }

        private function addItemAt(position:int):void   {
              var item:EmployeeDTO = new EmployeeDTO();
              collection.addItemAt(item, position);
              dg.selectedIndex = position;
        }

                  private function logEvent(evt:Event):void {
              if (evt.type=="fault") {
                    logger.error(evt["fault"]["faultString"]);
              } else {
                    if (evt.type=="collectionChange") {
                          logger.debug(evt["type"] + " " + evt["kind"]);
                    } else {
                          logger.debug(evt["type"]);
                    }
              }
        }
```

Even though this code was auto-generated, nothing stops you from modifying it to your liking. CDB and theriabook library gives your project a jump start and the resulting code base is small – the

Employee_getEmployees_GridTest.mxml is only about 90 lines of code, which includes 10 empty lines and the code for logging!

The other freebee that CDB gives you is an automated ANT process of building and deploying your project both on the client and the server. You do not need to worry about creating compatible Java and ActionSript DTO objects. If the structure of the result set of the server side data is changing, just change the SQL statement in your Java abstract class and re-run the daoflex-build.xml.

3.1.1 The abstract Java class

The abstract Java class that we used in our sample application:

```
package com.farata;
import java.util.List;

/**
* @daoflex:webservice pool=jdbc/db2_sample
*/
public abstract class Employee {
        /**
        * @daoflex:sql
        *  pool=jdbc/db2_sample
        *  sql=::select * from yf.EMPLOYEE
        *  ::
        *  transferType=EmployeeDTO[]
        *  keyColumns=EMPNO
    *  updateTable=yf.EMPLOYEE
        */
   public abstract List getEmployees();
}
```

A large portion of this small Java class consists of comments with tags, which can be automatically created using the hot keys. For example, you can generate a class comment by pressing CTRL + G, O.

To generate comments for a method, use CTRL + G, S. Press CTRL+G to display the list of all available hot keys, which will be explained a little later in this document.

The snippet tags can annotate either entire Java class or a method. Here's a brief explanation of the tags that CDB understands.

@daoflex:webservice means that this class will be used for accessing the database and can be used from a Flex project.

The *pool* parameter refers to the database connection pool that should be used with the selected database profile.

@daoflex:sql means that this method will run SQL statements, which in our case is an SQL query defined between the double colons `sql=::  select * from employee`

```
    *  ::
```

Double colons are used are used as the code block delimiters, similar to open/close parenthesis.

Our method getEmployees() has three parameters: transferType, keyColumns and updateTable.

transferType defines the type of the result returned by your query. You do not have to manually define Java and ActionScript EmployeeDTO classes as they will be generated based on the structure of your database query.

keyColumns defines a unique key(s) of the database table you use.

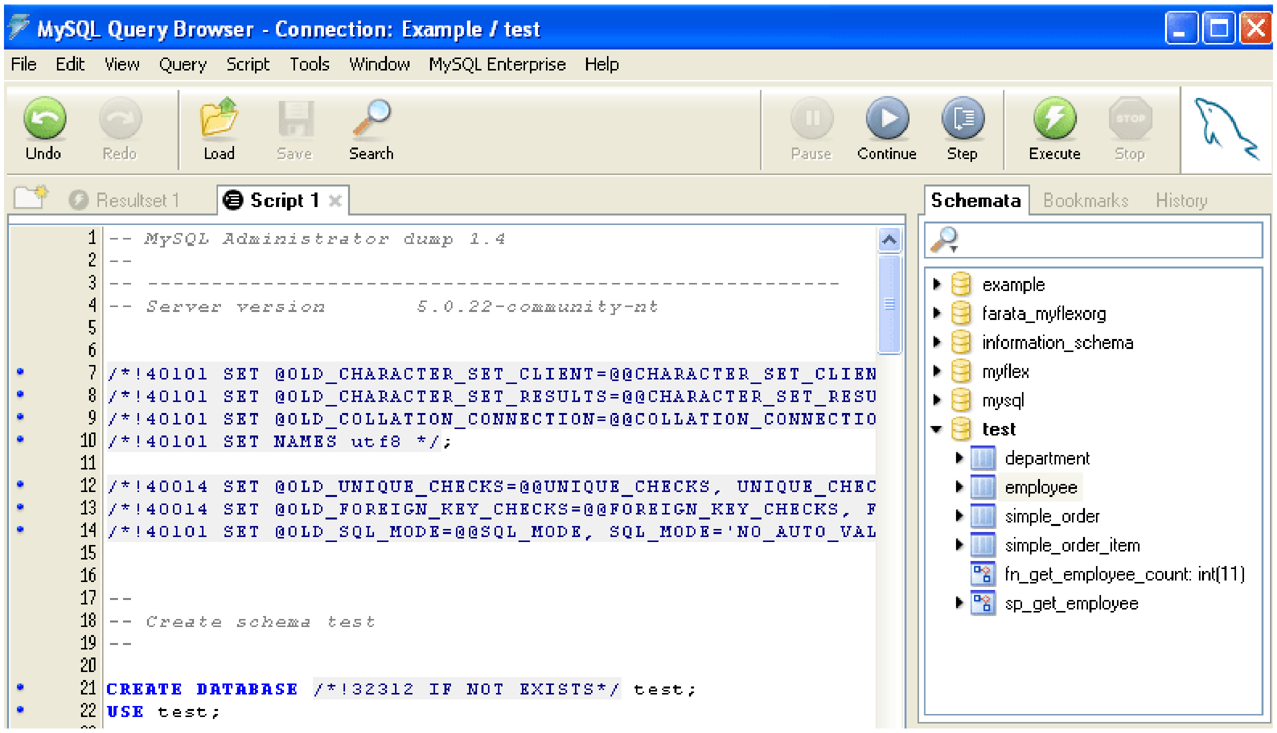
updateTable indicates that you are using a query of the type sync and it includes the name of the table to be updated.

updatableColumns designates the columns that should participate in generated UDPATE and INSERT statements. Not specifying *updatableColumns* is the same as explicitly listing all columns of the result set.

If you've selected the option **Build Automatically**, all required CDB files will be generated each time you modify and save your code. You can also select the option **Build** to perform manual build of the project. After the build is complete, the directory **.generated/java/com/Farata/examples** will contain the class ExampleDAO.java. This class extends and implements all declared methods of the abstract class Example.

Selecting the Eclipse menu Project | Clean also starts the Ant build process.

In this sample application, we did not select the MyFlex facet CDB Example Project. If we did, we'd have an automatically generated example (you would not even need to define the abstract class) by trhe plugin **com.farata.daoflex.examples** that comes with CDB. This plugin has the source codes for all required examples and SQL scripts that create and populate required database tables. During generation of the test example, all required files will be automatically copied into your project's directory. But if you'd like to see the content of these files without generating example projects, you can find all SQL scripts in the directory **<your_eclipse_directory>/plugins/com.farata.daoflex.examples_.../examples/sql/scripts**.



```
-- MySQL Administrator dump 1.4
--
-- ------------------------------------------------------
-- Server version       5.0.22-community-nt


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIEN
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESU
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTIO
/*!40101 SET NAMES utf8 */;

/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHEC
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, F
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VAL


--
-- Create schema test
--

CREATE DATABASE /*!32312 IF NOT EXISTS*/ test;
USE test;
```

The auto-generated example project includes the classes *Employee*, *Order* and *StoredProcedures*, which have the new code sections that did not exist in the project described in the previous section. You'll see the use of stored procedures, functions and an SQL Call Back. Let's take a closer look at these sections.

3.1.2 Stored Procedures

With CDB, you can use stored procedures and functions. You can specify the name of the stored procedure in the parameter *procedure* of the **@daoflex:sql annotation**. To pass argument values to a stored procedure, use the annotation **@daoflex:param** specifying the *name* and the *value*. Besides these two changes, it works as a regular SQL request. Just look at the method *getEmployee()* of the class *StoredProcedures*:

```
/**
* @daoflex:sql
*       pool=jdbc/test
*       procedure=sp_get_employee
*       transferType=EmployeeProcDTO[]
* @daoflex:param
*       name=empId
*       value=105
*/
public abstract List getEmployee(int empId);
```

3.1.3 Functions

Stored functions are called simularly to stored procedures, but you use the parameter *function* instead of *procedure*. Let’s assume that you have the MySQL function defined like this:

```
CREATE FUNCTION fn_get_employee_count() RETURNS int
BEGIN
     declare result int;
     select count(emp_id) into result from employee;
      return result;
END
```

Then you can use the following annotation in you CDB class:

```
/**
* @daoflex:sql
* function = fn_get_employee_count
*/
public abstract Object getEmployeeCount();
```

As a result Java will be returning java.util.Map with the key *outputParam1* containing the return value of the function. On the Flex/ActionScript side, you would obtain the result as ActionScript Object, containing *outputParam1* as the property (you’ll get the object while processing mx.rpc.events.ResultEvent).

3.1.4 SQL Callback

The SQL callback parameter is your key to yet another convenient feature of CDB that allows you to modify the request even after it was called. Let's look at the method *searchEmployees()* of the class *Employee*.

```
/**
* @daoflex:sql
*  sql=::  select * from employee
*  ::
*  transferType=EmployeeDTO[]
*  updateTable=employee
*  keyColumns=emp_id
*  tag=sqlCallback:modifyGetEmployeeSQL
*/
public abstract List searchEmployees(Map context);

protected String modifyGetEmployeeSQL(Map context, String sql) {
      // Whatever you feel like doing with the SQL
      String whereClause = (String)context.get("where");
      if (whereClause != null) {
            return sql + " WHERE " + whereClause;
      } else {
            return sql;
      }

}
```

The *tag=sqlCallback:modifyGetEmployeeSQL* parameter ionstructs CDB to call the method *modifyGetEmployeeSQL* right before execution of the SQL request. This method has to have the same argument list as the *searchEmployees()* plus one more additional argument of type String, which will contain the SQL query specified in the annotation of the method *searchEmployees()*. In our example, this string will contain *select * from employee*. The method *modifyGetEmployeeSQL* has to return the final String of the SQL request that will be executed against your database.

In other words, the method *modifyGetEmployeeSQL()* allows you to change the original request. In this example, a new condition that is specified in the context parameter will be added to the request.

As a reminder, when you remote to the Java method searchEmployees(Map context) from the ActionsScript , Java Map corresponds to the regular ActionScript Object that you would pass as the single parameter of the remote invocation. In our case, you may do it the following way:

```
remoteObject.searchEmployees({where:"lastName like S%"});
```

3.2 Transaction Processing with Clear Data Builder

CDB supports transaction processing: your user can modify the data coming from multiple destinations and CDB provides an APIs that can ensure that all changes are applied as a single unit of work, aka transaction.

Most often then not, synchronizing client data with the server can not be confined to one database table or, in general, to one source of data. Take the simple case: customers place orders, each order can have several items with their own price, quantity etc. Consider this table:

```
create table simple_order_item(
order_id char(32) not null,
item_id char(32) not null,
product_name varchar(32) not null,
quantity int(11) default '1' not null)
```

The order, as a whole, has its own attributes, such as address and date:

```
create table simple_order(
 order_id char(32) PRIMARY KEY not null,
 customer_first_name varchar(32) not null,
 customer_last_name varchar(32) not null,
 order_date datetime default '0000-00-00 00:00:00' not null,
 address varchar(64) not null)
```

Once the order is placed, customer may update it. Customer will expect that order creation as well as order updates follow the transaction or ‘unit of work’ paradigm: it can be completed only entirely, partial update will not happen.

How can we implement trasactions in RIA? One thing is certain: BEGIN and COMMIT (or ROLLBACK) should happen on the server. If our Java DAO is being invoked via DataServices adapter the BEGIN has already happened. For Flex Remoting scenario, we should handle transactions ourselves

3.2.1 BatchGateway destination and BatchService API

When you create a CDB project, it ensures the accessability of the remoting destination called *BatchGateway* that allows you to pass to the Java the sequence of *BatchMember* instances – each a triad of destination name, methodName and method arguments array. To facilitate your work with BatchGateway you should use ActionScript class *com.theriabook.remoting.BatchService*, located in theriabook_daoflex.swc library. This library automatically added to the target Flex project during the CDB project configuration.

Suppose we have two collections - *orders* and *orderItems*. (DataCollection used below is a descendant from ArrayCollection and is included in theriabook_daoflex.swc. Most importantly, DataCollection has methods *fill()* and *sync()* and properties *destination* and *method*):

```
 orders = new DataCollection();
 orders.destination="Order";
 orders.method="getOrders";
 orderItems = new DataCollection();
 orderItems.destination="Order";
 orderItems.method="getOrderItems";
```

Once collections are created, we register them with the BatchService specifying referential integrity priorities along the way. Priorities – in this case 0 and 1 – define order of operations. In our scenario, if there are records to be deleted from both collections than records from *orderItems* will be deleted first. Vice versa, new records will be inserted into *order* first:

```
 batchService = new BatchService();
 batchService.addEventListener(FaultEvent.FAULT, onCommitFault);
 batchService.registerCollection(orders,0);
 batchService.registerCollection(orderItems,1);
```

When it's time to synchronize - perhaps when the user clicks on the "Save" button - we ask BatchService to make the batch and send it to BatchGateway:

```
 var batch:Array = batchService.batchRegisteredCollections();
 batchService.sendBatch(batch);
```

3.2.2 A Sample MXML Application – OrderEntryDemo

Let's look at the sample code from the OrderEntryDemo application, which allows transactional data entry into tables *simle_order* and *simple_order_item*. The application is composed of two panels: OrdersPanel and OrderItemsPanel with controller class – OrderManager, as shown in the listing below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--OrderEntryDemo.mxml -->
<mx:Application
 xmlns:mx="http://www.adobe.com/2006/mxml"
 xmlns="*"
 >
 <OrderManager id="orderManager" />
 <mx:ControlBar>
       <mx:Button label="Fill"  click="orderManager.fillOrders()"  />
       <mx:Button label="Commit"  click="orderManager.commit()"
enabled="{orderManager.commitRequired}" />
 </mx:ControlBar>
 <mx:VDividedBox  >
```

```
        <OrdersPanel id="master" />
        <OrderItemsPanel id="detail" width="100%"/>
 </mx:VDividedBox>
</mx:Application>
```

3.2.3 The Clear Data Builder class for OrderEntryDemo

The source CDB Java class is listed below. It assumes that the name of the datasource is jdbc/theriabook:

```
package com.theriabook.datasource;

import java.util.List;

/**
 * @daoflex:webservice
 *   pool=jdbc/theriabook
 */

public abstract class Order {
       /**
       * @daoflex:sql
       *     sql=::
             select order_id, customer_first_name firstName,
             customer_last_name lastName, order_date,  address
       from  simple_order
          ::
       *     transferType=OrderDTO[]
       *     keyColumns=order_id
       *     updateTable=simple_order
       */

       public abstract List getOrders();

       /**
       * @daoflex:sql
       *     sql=select * from simple_order_item WHERE ORDER_ID=:orderId
       *     transferType=OrderItemDTO[]
```

```
        *     updateTable=simple_order_item
        *     keyColumns=order_id,item_id,product_name
        */

        public abstract List getOrderItems(String orderId);
}
```

3.2.4 OrderEntryDemo: OrdersPanel (Master)

The top panel of the application – OrdersPanel – binds its DataGrid to the collection, a private variable of the component:

```
<mx:DataGrid id="dg" dataProvider="{collection}" …/>
```

Importantly, this variable, upon execution of onCreationComplete(), carries a reference to bindable collection *orders* hosted by OrderManager. In other words, the DataGrid of orders is serving as a pure view or *orders* collection from OrderManager:

```
private function onCreationComplete() : void {
      orderManager = Application.application.orderManager;
      collection = orderManager.orders;
}
```

Buttons “Add” and “Remove” delegate their work to orderManager in the clean MVC style:

```
private function onAdd(position:int):void {
      orderManager.addOrder(position);
      dg.selectedIndex = position;
}
private function onRemove(position:int):void {
      orderManager.removeOrder(position);
}
```

Finally, whenever user is changing the row selection, we communicate to OrderManager the *orderId* that has been selected:

```
<mx:DataGrid id="dg" dataProvider="{collection}"
       change="orderManager.orderId=dg.selectedItem.ORDER_ID">
```

The complete code of OrdersPanel is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- OrdersPanel.mxml -->
<mx:Panel title="Orders"
 xmlns:mx="http://www.adobe.com/2006/mxml"
 creationComplete="onCreationComplete()">

 <mx:DataGrid id="dg" dataProvider="{collection}" editable="true" height="100%"
        change="orderManager.orderId=dg.selectedItem.ORDER_ID">
       <mx:columns>
             <mx:Array>
             <mx:DataGridColumn dataField="ORDER_ID" headerText="Order Id" />
             <mx:DataGridColumn dataField="FIRSTNAME" headerText="First Name" />
             <mx:DataGridColumn dataField="LASTNAME" headerText="Last Name" />
             <mx:DataGridColumn dataField="ADDRESS" headerText="Address"  />
             <mx:DataGridColumn dataField="ORDER_DATE" headerText="Order Date"
             itemEditor="mx.controls.DateField" editorDataField="selectedDate"/>
             </mx:Array>
       </mx:columns>
 </mx:DataGrid>
 <mx:ControlBar width="100%">
       <mx:Button label="Remove" click="onRemove(dg.selectedIndex)"
enabled="{dg.selectedIndex != -1}"/>
       <mx:Button label="Add" click="onAdd(Math.max(0,dg.selectedIndex+1)); "/>
       <mx:Label text="Deleted: {collection.deletedCount}"/>
       <mx:Label text="Modified: {collection.modifiedCount}"/>
 </mx:ControlBar>
 <mx:Script>
 <![CDATA[
 import mx.core.Application;
 import com.theriabook.collections.DataCollection;

 [Bindable]
 private var collection:DataCollection;
 private var orderManager:OrderManager ;

 private function onCreationComplete() : void {
       orderManager = Application.application.orderManager;
       collection = orderManager.orders;
 }
 private function onAdd(position:int):void {
       orderManager.addOrder(position);
       dg.selectedIndex = position;
 }
 private function onRemove(position:int):void {
       orderManager.removeOrder(position);
 }

 ]]>
 </mx:Script>
</mx:Panel>
```

3.2.5 OrderEntryDemo: OrdersItemsPanel (Detail)

The bottom panel of the application – OrderItemsPanel – populates its DataGrid with data from bindable collection *orderItems*, hosted by OrderManager:

```
<mx:DataGrid id="dg" dataProvider="{collection}" editable="true" height="100%"
….
/>
```

```
 [Bindable]private var collection:DataCollection ;
 [Bindable]private var orderManager:OrderManager ;

 private function onCreationComplete() : void {
       orderManager = Application.application.orderManager;
       collection = orderManager.orderItems;
 }
```

Similar to OrdersPanel, buttons “Add” and “Remove” delegate their work to OrderManager:

```
 private function onAdd(position:int):void {
       orderManager.addOrderItem(position);
       dg.selectedIndex = position;
 }

 private function onRemove(position:int):void {
       orderManager.removeOrderItem(position);
 }
```

The complete code of OrderItemsPanel is presented below:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- OrderItemsPanel.mxml -->
<mx:Panel title="OrderItems"
  xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="onCreationComplete()" >
<mx:DataGrid id="dg" dataProvider="{collection}" editable="true" height="100%">
       <mx:columns>
             <mx:Array>
             <mx:DataGridColumn dataField="ORDER_ID" headerText="Order Id" />
             <mx:DataGridColumn dataField="ITEM_ID" headerText="Item Id" />
             <mx:DataGridColumn dataField="PRODUCT_NAME" headerText="Product" />
             <mx:DataGridColumn dataField="QUANTITY" headerText="Quantity" />
             </mx:Array>
       </mx:columns>
 </mx:DataGrid>
 <mx:ControlBar width="100%">
       <mx:Button label="Remove" click="onRemove(dg.selectedIndex);"
enabled="{dg.selectedIndex != -1}"/>
       <mx:Button label="Add" click="onAdd(Math.max(0,dg.selectedIndex + 1)); "
enabled="{orderManager.orderId!=null}"/>
       <mx:Label text="Deleted: {collection.deletedCount}"/>
       <mx:Label text="Modified: {collection.modifiedCount}"/>
 </mx:ControlBar>
 <mx:Script>
       <![CDATA[
       import mx.core.Application;
       import com.theriabook.collections.DataCollection;

    [Bindable]private var collection:DataCollection ;
       [Bindable]private var orderManager:OrderManager ;

       private function onCreationComplete() : void {
             orderManager = Application.application.orderManager;
             collection = orderManager.orderItems;
       }
       private function onAdd(position:int):void {
             orderManager.addOrderItem(position);
             dg.selectedIndex = position;
       }
```

```
        private function onRemove(position:int):void {
              orderManager.removeOrderItem(position);
        }
  ]]>
  </mx:Script>
</mx:Panel>
```

3.2.6 OrderEntryDemo: OrderManager

The OrderManager class is central to the application. It encapsulates *orders* and *orderItems* collections along with the methods that manipulate these collections.The most important here are the methods that populate collections and communicate the changes to the server. Populating is rather trivial, we delegate to the *fill()* method of appropriate collection:

```
        orders.fill();
        orderItems.fill(orderId);
```

To send changed to the server in one, transactional, batch we use the BatchService:

```
        batchService = new BatchService();
        batchService.addEventListener(FaultEvent.FAULT, onCommitFault);

        batchService.registerCollection(orders,0);
        batchService.registerCollection(orderItems,1);

          .  .  .  .
        var batch:Array = batchService.batchRegisteredCollections();
        batchService.sendBatch(batch);
```

How do you determine that changes are pending on a DataCollection? You can use bindable *commitRequired* property. What if you have several DataCollections? Please notice a technique we used to determine if anything needs to be saved on at least one of the DataCollections:

```
        [Bindable]public var commitRequired:Boolean;
        .  .  .
        orders.addEventListener(
              PropertyChangeEvent.PROPERTY_CHANGE, onPropertyChange
        );
        orderItems.addEventListener(
              PropertyChangeEvent.PROPERTY_CHANGE, onPropertyChange
        );
        .  .  .
        public function onPropertyChange(event:PropertyChangeEvent):void {
              if (event.property == "commitRequired") {
                    commitRequired = (
                          orders.commitRequired || orderItems.commitRequired
                    );
              }
        }
```

The complete code of the OrderManager class is presented below:

```
// OrderManager.as
package
{
 import flash.events.EventDispatcher;
 import mx.events.PropertyChangeEvent;
 import mx.rpc.events.FaultEvent;
 import mx.controls.Alert;
 import com.theriabook.collections.DataCollection;
 import com.theriabook.datasource.dto.OrderDTO;
 import com.theriabook.datasource.dto.OrderItemDTO;
 import com.theriabook.remoting.BatchService;

 public class OrderManager
 {

    [Bindable]public var orders:DataCollection ;
    [Bindable]public var orderItems:DataCollection ;
       [Bindable]public var commitRequired:Boolean;

       private var _orderId:String;

       [Bindable]
       public function set orderId(newOrderId:String):void {
             _orderId = newOrderId;
             fillOrderItems(newOrderId);
       }
       public function get orderId():String {
              return _orderId;
        }

        public var batchService:BatchService;
        public function OrderManager(){
              orders = new DataCollection();
              orders.destination="Order";
              orders.method="getOrders";
              orderItems = new DataCollection();
              orderItems.destination="Order";
              orderItems.method="getOrderItems";
              batchService = new BatchService();
              batchService.addEventListener(FaultEvent.FAULT, onCommitFault);

              batchService.registerCollection(orders,0);
              batchService.registerCollection(orderItems,1);

              orders.addEventListener(PropertyChangeEvent.PROPERTY_CHANGE,
onPropertyChange);
              orderItems.addEventListener(PropertyChangeEvent.PROPERTY_CHANGE,
onPropertyChange);
        }

        public function onPropertyChange(event:PropertyChangeEvent):void {
              if (event.property == "commitRequired") {
                    commitRequired = (orders.commitRequired ||
orderItems.commitRequired);
              }
        }
        public function fillOrders():void {
              orders.fill();
        }

        public function fillOrderItems(orderId:String):void {
              orderItems.fill(orderId);
        }

        public function commit():void {
              var batch:Array = batchService.batchRegisteredCollections();
              batchService.sendBatch(batch);
        }

        public function onCommitFault(event:Object):void {
              Alert.show(event.fault.faultString , "Error");
        }

        public function addOrder (position:int):OrderDTO {
              var item:OrderDTO = new OrderDTO();
              item.ORDER_DATE = new Date();
              orders.addItemAt(item, position);
              orderItems.removeAll();
              orderItems.resetState(); // We do not want deletes, don't we?
              return item;
        }
        public function removeOrder(position:int):void {
              orderItems.removeAll();
              orders.removeItemAt(position);
              commit();
        }
        public function addOrderItem (position:int):OrderItemDTO {
              var item:OrderItemDTO = new OrderItemDTO();
              item.ORDER_ID = orderId;
              item.QUANTITY = 1;
              orderItems.addItemAt(item, position);
              return item;
        }
        public function removeOrderItem(position:int):void {
              orderItems.removeItemAt(position);
        }
  }
 }
```

3.3 Clear Data Builder Annotations

This section includes samples of CDB annotations. Technically, you may mix annotated and not annotated classes in one project, although you may want to avoid it for design considerations. If a Java class is not annotated, it's not considered a CDB class. Note: CDB produces two JARs in the *lib* folder of the WebApplication: a JAR with the original classes included in CDB project and a JAR with generated classes, While all Java classes, whether annotated or not, will be included in the first JAR, only annotated classes may - depending on the annotation - produce the output for the second one.

3.3.1 Convert-to-ActionScript Annotation

This annotation tells CDB that public properties of the annotated Java class should be translated into public properties of the ActionScript class located in the target Flex project under the same package structure. In other words, this annotation is used to demarcate DTO (Data Transfer Object) classes.

EXAMPLE:

```
package com.foo.project;
  /**
  * @daoflex:actionscript
  */
public class MyValueObject {
  .  .  .
```

DESCRIPTION:

CDB will create/update ActionScript class com.foo.project.MyValueObject in the Flex project, which has been declared as target Flex Project during the CDB “Configure Build” step. A later change in the original file will be causing regeneration of the ActionScript class.

3.3.2 Define-DataSource Annotation

This annotation defined the JNDI name of the data source that will be harcoded in the generated Java class. Unless overridden on the method level, this data source will apply for all DAO methods in the class.

EXAMPLE:

```
package com.foo.project;
  /**
  * @daoflex:webservice
  * pool=jdbc/theriabook
  */
public abstract class Employee {
  .  .  .
```

DESCRIPTION:

The name 'jdbc/theriabook' will be treated as JNDI data source name for each annotated method of the class.

USAGE:

Use **Clear Data Builder > Configure Build** to setup datasource jdbc/theriabook prior to running **Clear Data Builder > Build** command.

3.3.3 Update Annotation

This annotation is used to execute the single SQL *update* and return integer result of the operation. Note that such *update* may be part of the transaction if you use *BatchService* (see Transaction Processing With Clear Data Builder).

EXAMPLE

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {

  . . .

  /** @daoflex:sql
   *   sql=update employee set salary=salary*1.1 where start_date > :startDate
   *    pool=jdbc/differentdatasource
   */
  public int raiseEmployeeSalary(java.util.Date startDate);
```

DESCRIPTION

Java class created by CDB will contain public method raiseEmployeeSalary(). The method will be 'returning' to ActionScript the actual result of the updated records. Also, this method overrides the value of the datasource to be *jdbc/differentdatasource*.

USAGE

After CDB build, you may access the method via mx:RemoteObject against Flex Remoting destination com.foo.project.Employee.

3.3.4 Insert Annotation

This annotation is used to execute the single SQL *insert* and return integer result of the operation. Note that such *insert* may be part of the transaction if you use *BatchService* (see Transaction Processing With Clear Data Builder).

EXAMPLE

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {


   .  .  .


       /** @daoflex:sql
       *   sql=insert into employee (emp_id, emp_lname, emp_fname) values
(:empId, :lName, :fName)
       */
 public int addEmployee(int empId, String lName, String fName);
```

DESCRIPTION

Java class created by CDB will contain public method addEmployee(). The method will be 'returning' to ActionScript the actual result of the inserted records.

USAGE

After CDB build, you may access the method via mx:RemoteObject against Flex Remoting destination com.foo.project.Employee

3.3.5 Delete Annotation

This annotation is used to execute the single SQL *deletee* and return integer result of the operation. Note that such *delete* may be part of the transaction if you use *BatchService* ((see Transaction Processing With Clear Data Builder).

Example

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {


   .  .  .


        /** @daoflex:sql
         *   sql=delete from employee where end_date < :endDate
         */
 public int deleteEmployee(java.util.Date endDate);
```

Description

Java class created by CDB will contain public method deletedEmployee(). The method will be 'returning' to ActionScript the actual result of the deleted records.

Usage

After the CDB build is complete, you may access the method via mx:RemoteObject against Flex Remoting destination com.foo.project.Employee

3.3.6 Singleton Select Annotation

This annotation is used to execute the singleton SQL *select* and return ActionScript Object. The simplest way to give meaningful names to the fields of the ActionScript Object is by using SQL aliases.

EXAMPLE

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {


  .  .  .

 /**
 * @daoflex:sql
 *  sql=select count(*) EMPLOYEE_COUNT, avg(salary) SALARY_AVG from employee
 */
 public abstract Object getEmployeeMetrics();
```

DESCRIPTION

CDB will build Java class with public method getEmployeeMetrics(), 'returning' private Java class/ActionScript Object with fields EMPLOYEE_COUNT, SALARY.

USAGE

Once you run CDB build, you may invoke getEmployeeMetrics() via mx:RemoteObject, using Flex Remoting destination com.foo.project.Employee.

3.3.7 Stored Function Annotation

This annotation is used to execute the stored function SQL *select* and return ActionScript Object. The simplest way to give meaningful names to the fields of the ActionScript Object is by using SQL aliases.

EXAMPLE

```
        /**
        * @daoflex:sql
        * function = fn_get_employee_count
        */
        public abstract Object getEmployeeCount();
```

DESCRIPTION

As a result Java will be returning java.util.Map with the key *outputParam1* containing the return value of the function. On the Flex/ActionScript side, you would obtain the result as ActionScript Object, containing *outputParam1* as the property (you'll get the object while processing mx.rpc.events.ResultEvent).

USAGE

Once you run CDB build, you may invoke getEmployeeMetrics() via mx:RemoteObject, using Flex Remoting destination com.foo.project.Employee.

3.3.8 SQL Fill Annotation

This annotation is used to execute the SQL *select* returning result set contaning multiple rows of data.

EXAMPLE

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {

  .  .  .

 /**
 * @daoflex:sql
 *  sql=::  select * from employee where start_date < :startDate or
start_date=:startDate
 *  ::
 *  transferType=EmployeeDTO[]
 *  keyColumns=emp_id
 */
 public abstract List getEmployees(Date startDate);
```

DESCRIPTION

CDB will build Java class with public method getEmployees(), 'returning' java.util.List/mx.collections.ArrayCollection of EmployeeDTO instances.

CDB will create - both in Java and ActionScript – class com.foo.project.dto.EmployeeDTO with properties reflecting columns of the result set returned by SELECT query.

To deduct and describe the anticipated result set (properties of the EmployeeDTO) CDB connects to the database and performs real SQL *select* using the value of host arguments (*startDate*) as per the annotation (see example of how to define parameters in Stored Procedure Fill Annotation below) All non-specified arguments default to 'null'.

Note: Clear Data Builder will use 'emp_id' as identity field while configuring FLEX DMS destination

USAGE

1.Once you run CDB build, you may invoke getNewEmployees() via mx:RemoteObject, using Flex Remoting destination com.foo.project.Employee.

2.In particular, you may use com.theriabook.collections.DataCollection from theriabook_daoflex.swc library (that “Configure Clear Data Builder Build” automatically adds to the target Flex project).

This is the sample ActionScript to retrieve the data into DataCollection->ArrayCollection:

```
import com.theriabook.collections.DataCollection;
var collection:DataCollection = new DataCollection();
collection.destination = 'com.foo.project.Employee';
collection.method = 'getEmployees';
collection.addEventListener('result', onResult);  //optional
collection.addEventListener('fault', onFault);  //optional
collection.fill(new Date(1,1,1981));
```

3. Alternatively, you may apply mx:DataService (sync-method will not be available). In this case the name of the destination will be com.foo.project.Employee_getEmployees.

3.3.9 Stored Procedure Fill Annotation

This annotation is used to execute the stored procedure returning result set contaning multiple rows of data.

EXAMPLE:

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {

  . . .

 /**
 * @daoflex:sql
 *  procedure = sp_get_employee
 *  transferType=EmployeeDTO[]
 * @daoflex:param
 *     name=deptId
 *     value=100
 */
 public abstract List getDepartmentEmployeesProc(int deptId);
```

DESCRIPTION:

CDB will build Java class with public method getDepartmentEmployeesProc(), 'returning' java.util.List/mx.collections.ArrayCollection of EmployeeDTO instances.

CDB will create - both in Java and ActionScript - class com.foo.project.dto.EmployeeDTO with properties reflecting columns of the result set returned by stored procedure sp_get_employee.

Important is the value of the input parameter 'deptId' – 100. A stored procedure can return different result sets. To deduct and describe the result set that will be produced during the subsequent run-time call CDB connects to the database and performs real invokation of the stored procedure using the parameters as per the annotation.

All non-specified arguments default to 'null'.

USAGE:

1. Once you run CDB build, you may invoke getDepartmentEmployeesProc() via mx:RemoteObject, using Flex Remoting destination com.foo.project.Employee

2. In particular, you may use com.theriabook.collections.DataCollection from theriabook_daoflex.swc library (that “Configure Clear Data Builder Build” automatically adds to the target Flex project).

This is the sample ActionScript to retrieve the data into DataCollection->ArrayCollection:

```
import com.theriabook.collections.DataCollection;
var collection:DataCollection = new DataCollection();
collection.destination = 'com.foo.project.Employee';
collection.method = 'getDepartmentEmployeesProc';
collection.addEventListener('result', onResult);  //optional
collection.addEventListener('fault', onFault);  //optional
collection.fill(200);
```

3. Alternatively, you may apply mx:DataService (sync-method will not be available). In this case the name of the destination will be com.foo.project.Employee_getDepartmentEmployeesProc.

3.3.10 SQL Sync Annotation

This annotation, similarly to SQL Fill Annotation can be used to execute the SQL *select* returning result set contaning multiple rows of data. Most important distinction is that this annotation also support the synchronization of the table participating in *select* with the changes made by the interactive user. Such changes: inserts of the new rows, update of the old ones, deletes – all all done as one transaction. You can also make the SQL Sync a part of the bigger transaction, using *BatchService* (see section 4.6)

EXAMPLE

```
package com.foo.project;
  /**
  * @daoflex:webservice pool=jdbc/theriabook
  */
public abstract class Employee {


   .  .  .


/**
* @daoflex:sql
*  sql=::    select * from employee
 *                where start_date < :startDate or start_date=:startDate
*  ::
*  transferType=EmployeeDTO[]
*  keyColumns=emp_id
*  updateTable=employee
*  updatableColumns=phone,salary
*/
public abstract List getEmployees(Date startDate);
```

DESCRIPTION

CDB will build Java class with public method getEmployees(), 'returning' java.util.List/mx.collections.ArrayCollection of EmployeeDTO instances.

CDB will create - both in Java and ActionScript – class com.foo.project.dto.EmployeeDTO with properties reflecting columns of the result set returned by SELECT query.

The Java class will also expose method getEmployees_sync(), which accepts List of objects implementing flex.data.ChangeObject interface.

The (generated) method *getEmployee()* will use table *employee* as per *select* statement to populate collection passed down to ActionScript. The generated method *getEmployee_sync()* will apply modification using 'emp_id' as the key in the generated SQL/JDBC. Not all columns will participate in *update* and *insert* operations: as per explicit *updatableColumns* only 'phone' and 'salary' , as well as key – 'emp_id' - will be updated/inserted.

If your select is joining more then one table, you can update – via sync() - only one.

Note: CDB will use 'emp_id' as identity field while configuring FLEX DMS destination.

USAGE

1. Once you run CDB build, you may invoke getEmployees() via mx:RemoteObject, using Flex Remoting destination com.foo.project.Employee

2. In particular, you may use com.theriabook.collections.DataCollection from theriabook_daoflex.swc library (that "Configure Clear Data Builder Build" automatically adds to the target Flex project).

This is the sample ActionScript to retrieve the data into DataCollection->ArrayCollection:

```
import com.theriabook.collections.DataCollection;
var collection:DataCollection = new DataCollection();
collection.destination = 'com.foo.project.Employee';
collection.method = 'getEmployees';
collection.addEventListener('result', onResult);  //optional
collection.addEventListener('fault', onFault);  //optional
collection.fill(new Date(1,1,1981));
This is the sample ActionScript to sync the DataCollection up to the database:

collection.sync();
```

3. Alternatively, you may apply mx:DataService (both fill- and sync- methods will be available). In this case the name of the destination will be com.foo.project.Employee_getEmployees.

Note. If you do not provide name of the table to update and updateable columns you won't be able to send the changes up to the database.

3.4 Clear Data Builder Configuration File and Template Settings

CDB generates files using XSL templates, which help in linking Flex and CDB projects. It also allows you to use your own templates. In this section we'll discuss how to configure standard CDB templates and create your own.

You can configure CDB in different parts of your project. For example, CDB can use *common* settings for the entire project template configurations (defined by you), unless you redefine some of them for a limited number of classes and packages. If you never defined any templates, it'll use the default ones.

You can configure templates at the project, package or a class level. Just right-click on one of these three project elements and select from the menu **Clear Data Builder > Add local template settings**.

If templates are configured at the project level, CDB will create a new file called *daoflex.properties*, and associate it with your project. If you'll create CDB configuration file at the package level, the file *daoflex.properties* will be created in this package directory, and its settings will be applied to all classes of this package. If you create template settings for a class (file), you'll find automatically-generated properties file called <*your_class_name*>*.properties*, which defines templates associated just for the object you right-clicked on.

Creation of the daoflex.properties at the project level overrides CDB default templates settings, whereas defining templates at the package or class level just redefines or compliments project's configuration.

All the files that CDB generates are first being saved in the folder .generated and only then they go to the application server of in the Flex project. The .generated directory is located in the root directory of your CDB project and contains six subdirectories:

- Ø **bin** stores generated comiled java classes (*.class)
- Ø **java** stores generated sources of java classes (*.java)
- Ø **lib** stores generated libraries (*.jar)
- Ø **meta** stores generated metadata about your database and CDB classes (*.xml)
- Ø **SQL** stores generated SQL scripts (*.sql)
- Ø **web** stores generated ActionScript classes (*.as)

By configuring CDB you may affect the content of these directories and create the new ones.

3.4.1 Clear Data Builder Configuration File: daoflex.properties

This section describes the content of the configuration file daoflex.properties, and is applicable to all CDB configuration files regardless of were they were defined. A sample CDB configuration file may look as follows:

```
codegen.dao.template.templates=<dao-template-names>
<dao-template-descriptions>
codegen.bean.template.templates=<bean-template-names>
<bean-template-descriptions>
codegen.java2as.template.templates=<java2as-template-names>
<java2as-template-descriptions>
```

Notice three types of templates dao, bean, java2as.

- **bean-template** is used for generation of the DTO classes for CDB and Flex projects
- **java2as-template** is used for generation of ActionScript classes
- **dao-template** is used for things such as generation of test sample applications and more.

The element <`...-templates-name`> is a comma-separated list of template names, which you want to be used during code generation. For example,

```
codegen.dao.template.templates=ws-metadata,dao
```

You have to define a number of parameters for each template included in this list:

```
codegen.<template-type>.template.<template-name>.<param-name>=<param-value>
…
```

For example, you can define the following parameters for the dao template:

```
codegen.dao.template.dao.name=Dao
codegen.dao.template.dao.src=xsl/Dao.xsl
codegen.dao.template.dao.out=$(javaPath)/$(packagePath)/$(baseName)DAO.java
```

<param-name> can have one of the following values:

- **src** - this required parameter can contain either a name of the source xsl file, that will be used for code generation or a special tag `@bypass`, which will be described later @bypass
- **out** – this required parameter specifies a name of the output file
- **name** – this optional parameter contains the name of the template
- **mode** – this optional parameter is used to pass special commands to this template, and will be described later
- **override** – if the value of this parameter is false, the newly generated output file will not override an existing file, if any. The default value of this parameter is true

Typically, the names of the src files are specified in the following format: **xsl/some_xsl_file.xsl**. If you decide to create your own xsl files, just define the path to a directory that contains xsl file of your CDB project. For example, you your xsl file is located in directory **your_project/xsl/your_xsl.xsl**, the value of src parameter should read **src=xsl/your_xsl.xsl**.

CDB tries to find xsl files in your project directories, and if not found, it uses default templates located in **<your_eclipse_directory>/plugins/com.farata.daoflex.resources_.../resources/**. This means that your own xsl templates override the default ones.

If your templates use the *mode* parameter, its value is gridMethodTest, which allows you to generate not just one file per class, but one file for each method. In other words, if you do not define the parameter mode, then then youir XSL template will be applied for each CDB class just once ans only one output file will be generated.

If you defined mode=gridMethodTest, than you'll get one outpt file for each method of your class. In your XSL you can use modeParam with the value of your method name. For example, this parameter is used in one of the standard templates described in the section 4.6.4 to generate a separate MXML file for each database request that was specified as a method of the CDB class.

3.4.2 Parameters

You can define parameters for reuse in multiple places in the CDB configuration file. For example, the property **out** of the template **dao** can look as follows:

codegen.dao.template.dao.out=$(javaPath)/$(packagePath)/$(baseName)DAO.java

The line above uses three parameters - javaPath, packagePath, baseName. You can also use the same parameters in your xsl. Let's review all available parameters:

3.4.2.1 Common Parameters

debug - this parameter works in pair with an aditional parameter additionalparam: -daoflex-debug='<debug-value>'. You should add it to the target section of the files daoflex-build.xml ? daoflex-build-inc.xml. The <debug-value> can be either true or false. The debug section of the generated ant build file may look as follows:

```
<target name="generate" ...>
        <javadoc classpath="..."
                additionalparam="
                        -daoflex.home='${daoflex.home}'
                        -daoflex.path.webapp='${generated.web.root}'
                        -daoflex.path.java='${generated.java.root}'
                        -daoflex.path.test='${generated.test.root}'
                        -daoflex.path.metadata='${generated.meta.root}'
                        -daoflex-debug='true'
                        -daoflex.path.flex-config='${flex.config.root}'"
                doclet="com.theriabook.daoflex.AntDoclet"
                docletpath="${doclet.path}"
        >
                <packageset dir="${source.java.root}" />
        </javadoc>
        ...
</target>
```

standalone – If this is a standalone version of CDB (not plugin) or not. In case of CDB its value is Boolean.False

webappPath – the path that matches the value of the directory **.generated/web** of your CDB project.

javaPath – the path that matches the **.generated/java** value of your CDB project.

metadataPath – the path that matches the value of **.generated/meta** of your CDB project.

testPath – location of the test MXML folder. You can change the value of this parameter using the window

Clear Data Builder > Generated Paths in the settings of your CDB project.

flexConfigPath – location of the **WEB-INF/flex** directory of your web application.

out – the name of the output file

packagePath – the directory that matched the package name of your class, for example com/farata/examples.

baseParam – the class name (no name extension)

modeParam – this parameter is used for reinvoking XSL template in cascading mode for each method

3.4.2.2 The dao Template

You can use the following paramethers with the template codegen.dao.template:

package – the name of the package where your class is located, for example com.farata.examples.

name – the name of your class

qualifiedName – the full name of the class

3.4.2.3 The bean Template

You can use the following paramethers with the template codegen.bean.template:

package – the name of the package where your class is located, for example com.farata.examples

name – the name of your class

qualifiedName – the full name of the class

viewid – reserved for use in FlexBI component

pool – the name of JDBC connection pool, that can be used either on class or on Method level. This is the pool that you specify as a parameter to @daoflex:webservice (see section 4.7.2).

3.4.3 A Sample Configuration File daoflex.properties:

```
/////////////////////////////////////////////////////////////

codegen.dao.template.templates=ws-metadata,dao,assembler,data-management-
config,dataservice-test,dataservice-formtest,remoting-config,remoting-
test,composition-sql

//

codegen.dao.template.ws-metadata.name=WebServiceMetadata

codegen.dao.template.ws-metadata.src=@bypass

codegen.dao.template.ws-
metadata.out=$(metadataPath)/$(packagePath)/$(baseName).xml


codegen.dao.template.dataservice-test.name=TestApplication (Grid/DataService)

codegen.dao.template.dataservice-test.src=xsl/DataServiceGridTest.xsl

codegen.dao.template.dataservice-
test.out=$(testPath)/ds/$(packagePath)/$(baseName)_$(modeParam)_GridTest.mxml

codegen.dao.template.dataservice-test.mode=gridMethodTest


codegen.dao.template.dataservice-formtest.name=TestApplication
(Grid/DataService)

codegen.dao.template.dataservice-formtest.src=xsl/DataServiceGridFormTest.xsl

codegen.dao.template.dataservice-
formtest.out=$(testPath)/ds/$(packagePath)/$(baseName)_$(modeParam)_GridFormTest
.mxml
```

```
codegen.dao.template.dataservice-formtest.mode=gridMethodTest

codegen.dao.template.dao.name=Dao
codegen.dao.template.dao.src=xsl/Dao.xsl
codegen.dao.template.dao.out=$(javaPath)/$(packagePath)/$(baseName)DAO.java

codegen.dao.template.remoting-test.name=TestApplication (Grid/Remoting)
codegen.dao.template.remoting-test.src=xsl/RemotingGridTest.xsl
codegen.dao.template.remoting-
test.out=$(testPath)/rpc/$(packagePath)/$(baseName)_$(modeParam)_GridTest.mxml
codegen.dao.template.remoting-test.mode=gridMethodTest

codegen.dao.template.remoting-formtest.name=TestApplication (Grid/Remoting)
codegen.dao.template.remoting-formtest.src=xsl/RemotingGridFormTest.xsl
codegen.dao.template.remoting-
formtest.out=$(testPath)/rpc/$(packagePath)/$(baseName)_$(modeParam)_GridFormTes
t.mxml
codegen.dao.template.remoting-formtest.mode=gridMethodTest

codegen.dao.template.assembler.name=Assembler
codegen.dao.template.assembler.src=xsl/Assembler.xsl
codegen.dao.template.assembler.out=$(javaPath)/$(packagePath)/$(baseName)Assembl
er.java

codegen.dao.template.remoting-config.name=remoting-config.xml
codegen.dao.template.remoting-config.src=xsl/remoting-config.xsl
codegen.dao.template.remoting-config.out=$(flexConfigPath)/daoflex/remoting-
config/$(packagePath)/$(baseName).xml
codegen.dao.template.remoting-config.override=true

codegen.dao.template.composition-sql.name=Initial report metadata saved as SQL
INSERT
codegen.dao.template.composition-sql.src=xsl/compositionsql.xsl
codegen.dao.template.composition-
sql.out=$(metadataPath)/../SQL/$(packagePath)/$(baseName).sql

codegen.dao.template.report-mxml.name=Initial report metadata saved as MXML
codegen.dao.template.report-mxml.src=xsl/report-mxml.xsl
codegen.dao.template.report-
mxml.out=$(testPath)/rpc/$(packagePath)/$(baseName)_$(modeParam)_ReportTest.mxml
codegen.dao.template.report-mxml.mode=gridMethodTest
codegen.dao.template.data-management-config.name=Data management config

codegen.dao.template.data-management-config.src=xsl/data-management-config.xsl

codegen.dao.template.data-management-config.out=$(flexConfigPath)/daoflex/data-
management-config/$(packagePath)/$(baseName)_$(modeParam).xml

codegen.dao.template.data-management-config.mode=gridMethodTest

codegen.dao.template.data-management-config.override=true


/////////////////////////////////////////////////////////////
codegen.bean.template.templates=java-dto,as-dto


codegen.bean.template.java-dto.name=Java DTO

codegen.bean.template.java-dto.src=xsl/JavaDTO.xsl

codegen.bean.template.java-dto.out=$(javaPath)/$(packagePath)/$(baseName).java


codegen.bean.template.as-dto.name=ActionScript DTO

codegen.bean.template.as-dto.src=xsl/ActionScriptDTO_IManaged.xsl

codegen.bean.template.as-dto.out=$(webappPath)/$(packagePath)/$(baseName).as


codegen.bean.template.bean-metadata.name=BeanMetadata

codegen.bean.template.bean-metadata.src=@bypass

codegen.bean.template.bean-
metadata.out=$(metadataPath)/$(packagePath)/$(baseName)-bean.xml


/////////////////////////////////////////////////////////////
codegen.java2as.template.templates=


codegen.java2as.template.as-metadata.name=ActionScriptMetadata

codegen.java2as.template.as-metadata.src=@bypass

codegen.java2as.template.as-
metadata.out=$(metadataPath)/$(packagePath)/$(baseName).xml


codegen.java2as.template.classdoc2as.name=ActionScript

codegen.java2as.template.classdoc2as.src=xsl/classdoc2as.xsl

codegen.java2as.template.classdoc2as.out=$(webappPath)/$(packagePath)/$(baseName
).as
```

3.4.4 Default Templates

CDB comes with multiple templates described below.

3.4.4.1 Dao Templates

ws-metadata

This template is used with your java class specified in the annotation **@daoflex**:webservice. This template helps during generation of the xml file containing the meta-data about your class and the database. All other dao and bean templates are applied to the generated xml. This template must be defined when you what to generate at least one more template. By default, the generated xml is located in the following file:

$(metadataPath)/$(packagePath)/$(baseName).xml.

dao

This template generates a descendent of your class that implements all properties defined in your annotations. By default the generated java file is located here: $(javaPath)/$(packagePath)/$(baseName)DAO.java.

assembler

This template is needed to generate the wrappers for DAO classes. Your Flex project will call methods of the class generated with this template.By default the generated java file is $(javaPath)/$(packagePath)/$(baseName)Assembler.java.

data-management-config

This template generates XML files that present a destination that's mapped to a user class. Under the default configuration scenario, this XML would go inside the <services> node of the data-management-config.xml file, located in the WEB-INF/flex folder of your Web application. This template allows you to work with your database using Flex DataService object from the Flex project. By default, the names of these generated files are $(flexConfigPath)/daoflex/data-management-config/$(packagePath)/$(baseName)_$(modeParam).xml.

remoting-config

This template generates XML files with the destination of the user-defined class. Under the default configuration scenario, this XML would go inside the <services> node of the remoting-config.xml file, located in the WEB-INF/flex folder of your Web application. This template allows you to access your database using Flex remoting whith RemoteObject. By default, $(flexConfigPath)/daoflex/remoting-config/$(packagePath)/$(baseName).xml.

dataservice-test, dataservice-formtest, remoting-test, remoting-formtest

These for test templates are used for generation of the test applications that access your database by means of Flex DataService and RemoteObject. You can also use these sample applications to browse and edit database tables defined in annotations. Section 4.8 has instructions on compiling and running these test mxml applications.

3.4.4.2 Bean templates

java-dto

This template is used for generation of the Java DTO class, specified in your annotation as transferType. This class is generated based on the database metadata. The member variables of this class correspond to the fields of the database table. By default, the generated Java class is located over here: $(javaPath)/$(packagePath)/$(baseName).java.

as-dto

This template is used for generation of an ActionScript DTO class specified in your annotation as transferType. This template is similar to java-dto, but it generate the AcrionScript front end peer of the Java DTO. By default, the generated Java class is located over here: $(webappPath)/$(packagePath)/$(baseName).as.

3.4.4.3 ActionScript templates

as-metadata[1]

This template is applied to your Java class defined in the annotation **@daoflex**:actionscript. It generates the xml with metadata about your class and the database. All other dao and bean template are applied to already generated xml. This template should be defined always when you want to generate at least one more templat besides this one. By default, the generated XML is located over here: $(metadataPath)/$(packagePath)/$(baseName).xml.

classdoc2as[2]

This class generates the ActionScript for your Java class. By default, the generated Java class is located over here: $(webappPath)/$(packagePath)/$(baseName).as.

[1] ws-metadata is used for Dao and Bean templates, and as-metadata is used for ActionScript templates.

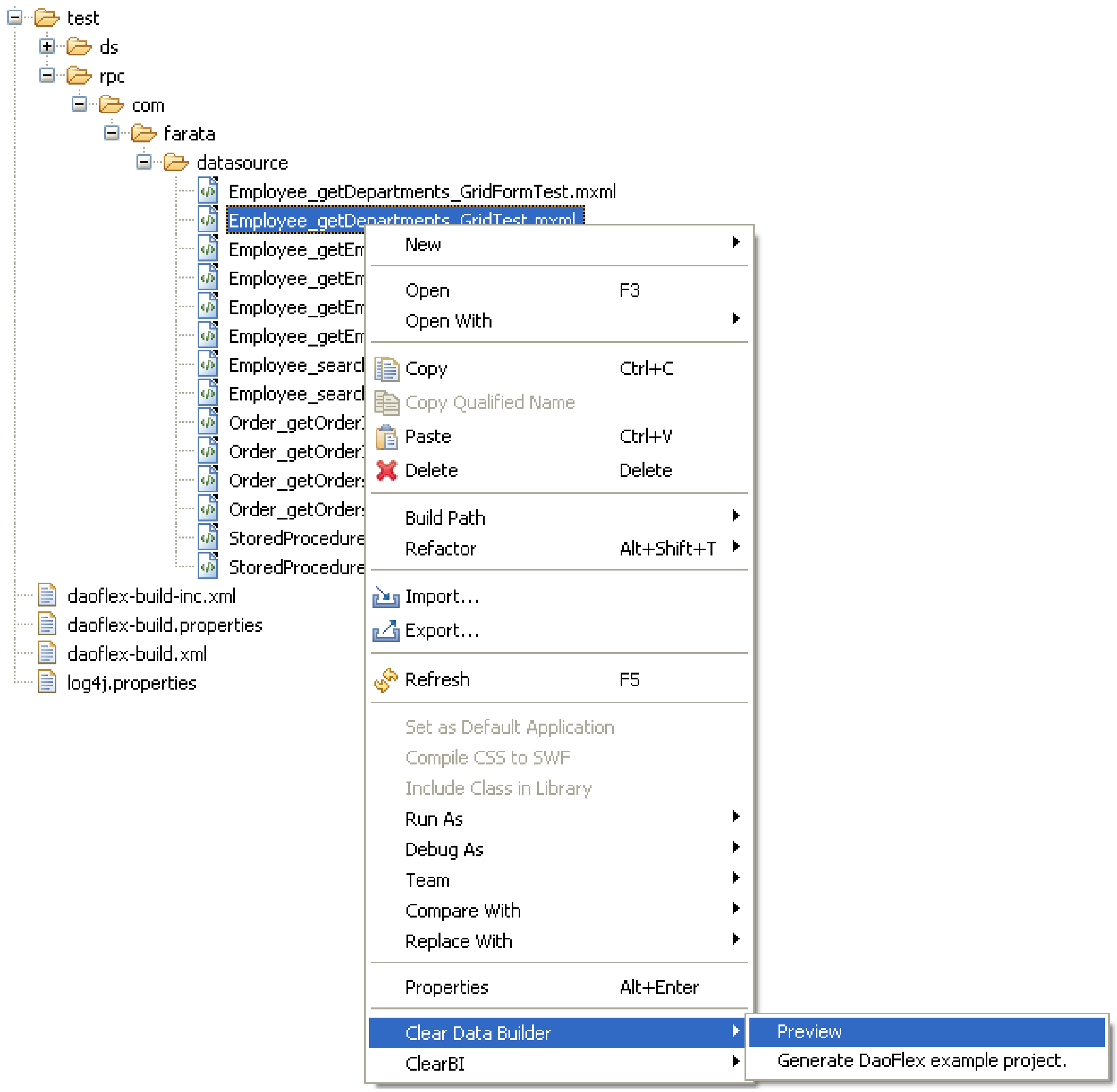
[2] classdoc2as is used for generating ActionScript classes for java classes marked with @daoflex:actionscript, while as-dto classes is used for generating ActionScript classes that are refered as DTO.

3.5 MXML Utilities

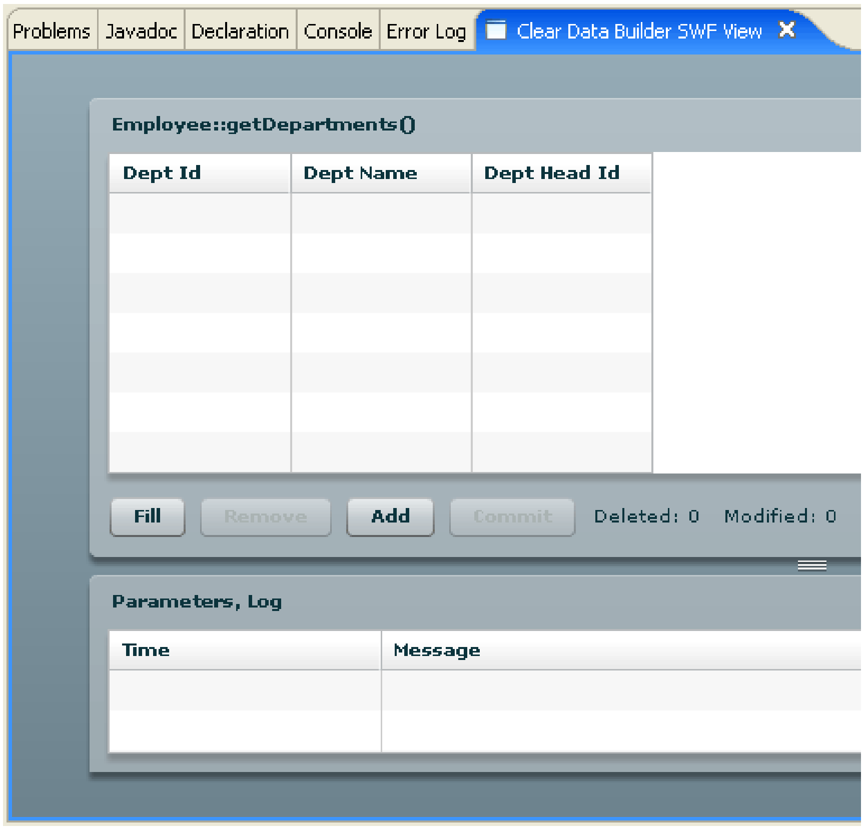
CDB includes utilities for working with standard generated MXML files that were described earlier in section **3.4.4**. These utilities support compilation of MXML files into SWF and running them in a custom Eclipse View. You'll learn how to use these utilities in this section.

1. Open your CDB and right-click on one of the MXML tiles (be default the are located in the directory test).

2. Select **Clear Data Builder > Preview** in the next popup window.



3. Clear Data Builder will compile the MXML file and will start created SWF in a special view Clear Data Builder SWF.



Dept Id	Dept Name	Dept Head Id

Time	Message

4. This SWF can access your database (CRUD) by selected access method – DataService or RemoteObject.

4.0 UNDERSTANDING CLEAR DATA BUILDER

This section is an optional read, and it was included for those developers who want to better understand how Flex Data Management Services work and what's the value proposition of the CDB utility. This information also reveals the of creation of Clear Data Builder itself. This section is an extract from the book written by creators of Clear Data Builder called "Rich Internet Application with Adobe Flex and Java", Sys-Con Media, 2007 (http://www.theriabook.com). This text is used by the permission of the afore-mentioned publisher.

4.1 Flex Data Management Services

LCDS include Remoting and Data Management Services (DMS). Whereas Flex Remoting enables one-way requests, DMS combine one-way requests with the publish/subscribe mechanism so that besides the original result set DMS sends the client live updates produced by other clients working with the same destination. And there's one more dimension in which Data Services depart from Flex Remoting – support for hierarchical collections, but we won't be covering that subject here.

In other words, DMS resolves the task of programming data collaboration: Several users may edit different rows and columns of the "same" DataGrid and see each other's changes automatically pushed by the server. Now, what if they overlap each other's work? In terms of DMS that's called a conflict and the DMS API provides for flexible conflict resolution, which may require the user's intervention.

A DMS destination can be configured for working with the data that is persisted to a data store as well as supporting scenarios that persist the data in the server's memory. To that end, DMS provides Java and ActionScript data adapters that are responsible for reading and updating a persistent data store according to its type. In this chapter we'll focus on use cases involving Java adapters.

4.2 LCDS and Automation: Problem Statement and Solution

LCDS include Flex Remoting and Flex Data Management Services. This section is dedicated to Data Management Services, and we will be referring to them as DMS, omitting the "Management" part. Thus herein DMS should be treated in a "narrow" context, as the counterpart of Flex Remoting and not as LCDS, which encompass both components.

Robust in enabling collaborative manipulation of data, DMS demand a substantial development effort in case of persistent data stores. In particular, you need to build:

• A Java Data Access Object class that implements retrieve, update, delete, and insert of the data

• Java Data Transfer Objects (DTO)

• A matching ActionScript data transfer object class

• A configuration file, which registers identity columns of the result set and, optionally, argument types for every retrieval method and other parameters.

We just mentioned four classes/files containing hard-coded names of the fields and there are more. To function properly, these hard-coded values have to be kept in sync, which is an additional maintenance effort whenever the data structures change.

Instead of this complexity, the main idea of this chapter is not to cover every twist of the DMS API, but rather automate the development effort that DMS takes for granted. We'll start with a "manual," albeit simplified, example of using DataServices. Then we'll introduce you to the methodology of complete code generation based on the pre-written XSL templates and DMS-friendly XML metadata, which will be extracted from the annotated Java abstract classes.

This methodology is fully implemented in Clear Data Builder – an open source utility[3] that's a complementary addition to this book. We'll gradually introduce this tool by leading you through a process of creating the most comprehensive template that generates a complete DataServices Data Access Object (DAO). Finally, we'll

[3] Besides the CDB Eclipse plugin, there is an open source command-line version of the Clear Data Builder utility.

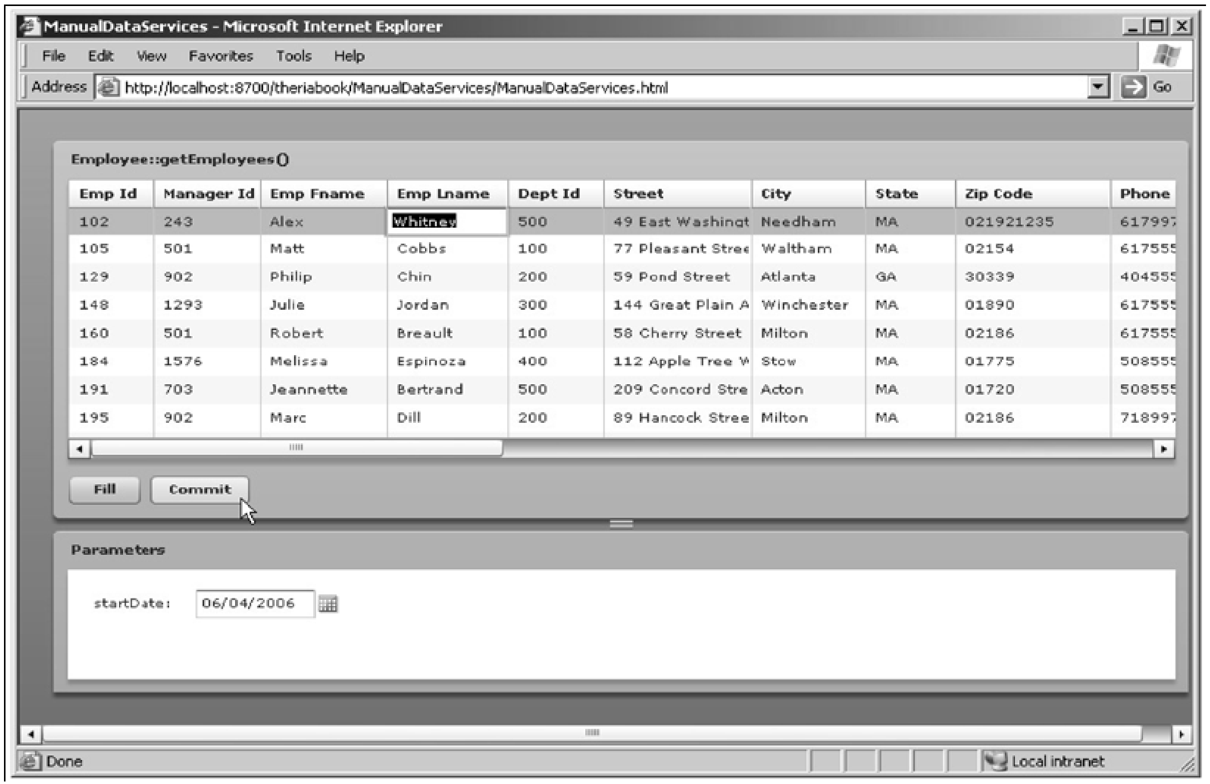
show you how to run and customize Clear Data Builder in your development environment so that writing and synchronizing routine DataServices support classes becomes a task of the Ant building tool and not yours!

4.3 A "Manual" DMS Application

Let’s handcraft the application presented in the screen shot below. This application displays a Panel with a scrollable DataGrid that we consciously did not size in the horizontal dimension, so that all columns can be viewed without shrinking. The database result set is ultimately produced by the following SQL query that will use a bound variable in place of the question mark:

```
select * from employee where start_date < ?
```

There are two buttons below the DataGrid: Fill and Commit. As the names imply, these buttons pull the original data from the database table and submit the data changes back to an DMS destination. A separate Parameters panel permits entering parameters of the back-end method behind the Fill button, which, in our case, is the employee start date:


ManualDataServices - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:8700/theriabook/ManualDataServices/ManualDataServices.html
Go
Employee::getEmployees()
Fill
Commit
Parameters
Done
Local intranet

Emp Id	Manager Id	Emp Fname	Emp Lname	Dept Id	Street	City	State	Zip Code	Phone
102	243	Alex	Whitney	500	49 East Washingt	Needham	MA	021921235	617997
105	501	Matt	Cobbs	100	77 Pleasant Stree	Waltham	MA	02154	617555
129	902	Philip	Chin	200	59 Pond Street	Atlanta	GA	30339	404555
148	1293	Julie	Jordan	300	144 Great Plain A	Winchester	MA	01890	617555
160	501	Robert	Breault	100	58 Cherry Street	Milton	MA	02186	617555
184	1576	Melissa	Espinoza	400	112 Apple Tree W	Stow	MA	01775	508555
191	703	Jeannette	Bertrand	500	209 Concord Stre	Acton	MA	01720	508555
195	902	Marc	Dill	200	89 Hancock Stree	Milton	MA	02186	718997

startDate: 06/04/2006

4.4 Building the Client Application

Let’s build the client application first. The full listing of the application is presented in Listing 4.1. We start with defining the mx:DataServices object (aka ds), which points to the destination “Employee.” Later, when we get to the server components, we’ll discuss mapping this destination to the backing Java class:

<mx:DataService id=”ds” destination=”Employee” fault=”onFault(event)” />

We provide only a rudimentary handler of the fault event, that’s sufficient to keep us aware of any anomalies that may occur along the way. Dynamic referencing of fault and faultString properties will spare us from casting to a specific event:

```
private function onFault(evt:Event):void {
```

```
Alert.show(evt[“fault”][“faultString”], “Fault”);
}
```

Then we define a handler of the application’s onCreationComplete event, where we instantiate a collection to be eventually associated with our mx:DataService object and, most important, set both autoCommit and autoSyncEnabled of the ds to false:

```
private function onCreationComplete() : void {
collection = new ArrayCollection();
ds.autoCommit=false;
ds.autoSyncEnabled=false;
}
```

By setting autoCommit to false we state that all updates have to be batched and explicitly submitted to the server as a single transaction during the ds.commit() call. By setting autoSyncEnabled to false we effectively protect our local instance of data from the delivery of messages caused by other clients connected to destination “Employee.” Setting autoSyncEnabled to false is entirely optional, and we use it to avoid dealing with application specific conflict resolution. In particular, in the handler of the Commit button’s click event you might uncomment the first line to support the “optimistic” way of handling the conflicts:

```
private function commit_onClick():void {
//ds.conflicts.acceptAllClient(); // Optimistic conflict handling, as oppose
to ds.conflicts.acceptAllServer();
ds.commit();
}
```

Last, we have to initiate the population of the local collection with the ds.fill() method, which we do inside the click event handler of the button Fill:

```
private function fill_onClick():void {
ds.release();
ds.fill(collection, param_getEmployees_startDate.selectedDate);
}
```

The scripting portion of the application is completed so let’s build the UI. We create a DataGrid with the dataProvider bound to our collection in Listing 4.1. For brevity’s sake, we didn’t list all the columns here: you’ll have a chance to scrutinize them in the subsequent section of this chapter.

The DataGrid and ControlBar with Fill and Commit buttons are put inside a Panel, with DataGrid’s title bearing the name of the destination and a specific getEmployees method of that destination, which will ultimately be invoked during the ds.fill() call. The second panel, titled Parameters, contains a form with a single item mx:DateField. Both panels are embraced by the VDividedBox.

We’ve included a linkage variable of the data transfer type to ensure that the corresponding Action- Script class (EmployeeDTO) will be linked into the generated SWF file.

```
<?xml version=”1.0” encoding=”UTF-8”?>
<mx:Application xmlns:mx=”http://www.adobe.com/2006/mxml”
backgroundColor=”#FFFFFF”
```

```
creationComplete=”onCreationComplete()”>
<mx:DataService id=”ds” destination=”Employee” fault=”onFault(event)” />
<mx:VDividedBox width=”800” height=”100%”>
<mx:Panel title=”Employee::getEmployees()” width=”800” height=”70%”>
<mx:DataGrid id=”dg” dataProvider=”{collection}” editable=”true” height=”100%”>
<mx:columns><mx:Array>
<mx:DataGridColumn dataField=”EMP_ID” headerText=”Emp Id” />
<mx:DataGridColumn dataField=”MANAGER_ID” headerText=”Manager Id” />
<mx:DataGridColumn dataField=”EMP_FNAME” headerText=”Emp Fname” />
. . . .
</mx:Array></mx:columns>
</mx:DataGrid>
<mx:ControlBar>
<mx:Button label=”Fill” click=”fill_onClick()”/>
<mx:Button label=”Commit” click=”commit_onClick()” enabled=”{ds.commitRe
quired}”/>
</mx:ControlBar>
</mx:Panel>
<mx:Panel title=”Parameters” width=”100%” height=”30%”>
<mx:HBox height=”100%” width=”100%”>
<mx:Form label=”getEmployees()”>
<mx:FormItem label=”startDate:”>
<mx:DateField id=”param_getEmployees_startDate” selectedDate=”{new
Date()}”/>
</mx:FormItem>
</mx:Form>
</mx:HBox>
</mx:Panel>
</mx:VDividedBox>
<mx:Script>
<![CDATA[
import mx.controls.Alert;
import mx.collections.ArrayCollection;
import com.theriabook.datasource.dto.EmployeeDTO;
private var linkage:com.theriabook.datasource.dto.EmployeeDTO = null;
[Bindable]
private var collection : ArrayCollection;
private function fill_onClick():void {
```

```
ds.release();
ds.fill(collection, param_getEmployees_startDate.selectedDate);
}
private function onCreationComplete() : void {
collection = new ArrayCollection();
ds.autoCommit=false;
ds.autoSyncEnabled=false;
}
private function commit_onClick():void {
ds.conflicts.acceptAllClient();
ds.commit();
}
private function onFault(evt:Event):void {
Alert.show(evt[“fault”][“faultString”], “Fault”);
}
]]>
</mx:Script>
</mx:Application>
```

Listing 4.1 The handcrafted DataServices sample application

The application above doesn’t cover all use cases of the DMS API. We tried to keep it as small as possible for one reason: to enable metadata-based code generation. Ultimately, it will be entirely up to you which code you’d elect to generate by modifying the Clear Data Builder templates. Finally, we present the listing of the ActionScript class EmployeeDTO that our collection uses in communicating with the Employee destination:

```
package com.theriabook.datasource.dto
{
[Managed]
[RemoteClass(alias=”com.theriabook.datasource.dto.EmployeeDTO”)]
public dynamic class EmployeeDTO
{
// Properties
public var EMP_ID : Number;
public var MANAGER_ID : Number;
public var EMP_FNAME : String;
public var EMP_LNAME : String;
public var DEPT_ID : Number;
public var STREET : String;
public var CITY : String;
public var STATE : String;
public var ZIP_CODE : String;
public var PHONE : String;
public var STATUS : String;
public var SS_NUMBER : String;
public var SALARY : Number;
public var START_DATE : Date;
public var TERMINATION_DATE : Date;
public var BIRTH_DATE : Date;
public var BENE_HEALTH_INS : String;
public var BENE_LIFE_INS : String;
public var BENE_DAY_CARE : String;
public var SEX : String;
public function EmployeeDTO() {
}
} //EmployeeDTO
}
```

Listing 4.2 The ActionScript DTO class – EmployeeDTO

4.5 Creating Assembler and DTO Classes

The time has come to work on the Java side, which is a rather tedious process, so we'll gradually go top-down.

Our first stop is an Assembler class that the DMS Employee destination should map to. As the Flex documentation suggests, you can implement the methods on your Assembler class in several ways:

• Extend flex.data.assemblers.AbstractAssembler and override the fill(), getItem(), createItem(), updateItem(), and deleteItem() methods as needed.

• Configure these methods via XML definitions against a class that doesn't extend the AbstractAssembler class.

• Combined approach, where methods defined via XML declarations are used if defined, otherwise the AbstractAssembler methods are invoked.

We'll take an XML approach that lets us declare a so-called sync-method. The XML contract of the destination's sync-method prescribes that it accepts a single parameter: a List of flex.data.ChangeObject elements. We find it convenient to control how we want to process data changes. In particular, we'd like to maintain the following order: all deletes, then all updates, and then all inserts. After all, if the user deletes a record for an employee with EMP_ID= 123 and then inserts a new record with EMP_ID=123, we certainly wouldn't want our sync-method to issue the INSERT, followed by DELETE FROM employee WHERE EMP_ID=123 during the batched DMS data modifications.

Let's keep in mind that our ultimate focus is the metadata-based code generation. Should you decide to have your Assemblers as descendants of the AbstractAssembler, you'd have the liberty of modifying the corresponding Clear Data Builder template.

Listing 4.3 presents the complete XML describing the destination Employee. Under the default configuration scenario, this XML would go inside the <services> node of the flex-data-services.xml file, located in the WEB-INF/lib/flex folder of your Web application.

We set com.theriabook.datasource.EmployeeAssembler as the exact name of the class mapped by our destination, with the methods java.util.List getEmployees_fill(java.util.Date dt) and the List getEmployees_sync(List lst) acting as the fill and sync methods, respectively.

You'd be able to configure more than one fill-method, although all of them should operate with the same type of DTO. In the <metadata> node we specified that the EMP_ID property of the DTO has to be considered as a single key, or identity property of the elements distributed by the destination. You could use a generated Universal Unique Identifier (UUID) instead of the real data-store field in place of the identity, which is arguably more flexible, because FDMS didn't support updates to the identity fields when this piece was written.

Even though XML doesn't explicitly declare that the fill-method returns a List or that the syncmethod takes a List, this is a part of the XML contract for Assembler classes in destinations:

```
<destination id=”Employee”>
<adapter ref=”java-dao”/>
<properties>
<source>com.theriabook.datasource.EmployeeAssembler</source>
<scope>application</scope>
<metadata>
<identity property=”EMP_ID”/>
</metadata>
<network>
<session-timeout>0</session-timeout>
<paging enabled=”false”/>
<throttle-inbound policy=”ERROR” max-frequency=”500”/>
<throttle-outbound policy=”ERROR” max-frequency=”500”/>
</network>
<server>
<fill-method>
<name>getEmployees_fill</name>
<params>java.util.Date</params>
</fill-method>
<sync-method>
<name>getEmployees_sync</name>
</sync-method>
</server>
</properties>
</destination>
```

Listing 4.3 The destination Employee for flex-data-services.xml

The structure of the EmployeeAssembler Java class is pretty straightforward. This class delegates the actual data retrieval and update of the data store to the EmployeeDataServiceDAO class, which we'll discuss next:

```
package com.theriabook.datasource;
```

```
import java.util.*;
public final class EmployeeAssembler
{
public EmployeeAssembler()
{ }
public final List getEmployees_fill(Date startDate) {
return new EmployeeDataServiceDAO().getEmployees(startDate);
}
public final List getEmployees_sync(List items) {
return new EmployeeDataServiceDAO().getEmployees_sync(items);
}
}
```

Listing 4.4 EmployeeAssembler.java

Finally, here's the EmployeeDTO class that the EmployeeAssembler-based destination will be operating with. It offers a simplistic approach to UUID generation that has to be replaced by a UUID generator of your choice:

```
package com.theriabook.datasource.dto;
import java.io.Serializable;
public class EmployeeDTO implements Serializable
{
private static final long serialVersionUID = 1L;
public int EMP_ID;
public int MANAGER_ID;
public String EMP_FNAME;
public String EMP_LNAME;
public int DEPT_ID;
public String STREET;
public String CITY;
public String STATE;
public String ZIP_CODE;
public String PHONE;
public String STATUS;
public String SS_NUMBER;
public double SALARY;
public java.util.Date START_DATE;
public java.util.Date TERMINATION_DATE;
public java.util.Date BIRTH_DATE;
public String BENE_HEALTH_INS;
public String BENE_LIFE_INS;
public String BENE_DAY_CARE;
public String SEX;
public String DEPT_NAME;
private String _uid;
private static long _UID = 1L;
public EmployeeDTO() {
_uid = getUUID();
}
public String getUid() {
return _uid;
}
public void setUid(String value) {
_uid = value;
}
public static synchronized String getUUID() {
return “” + _UID++;
}
}
```

Listing 4.5 Listing of the Java DTO class – EmployeeDTO

4.6 Implementing the Fill-Method of the DataServices Data Access Object

Our next stop is the EmployeeDataServicesDAO class, which is responsible for actual data manipulation in the data store. This section will cover its fill-method, and the sync-method will be covered next. We outsource the utility functions of getting a JDBC connection, converting values from java.util.Date to java.sql.Date and vice versa to a handful of classes from the com.theriabook.Clear Data Builder package, which takes less than 200 lines of source code. For brevity’s sake we’ll omit the listings of these classes; you can find the complete source code in the CD accompanying the book (look for the folder TheRiaBook/tools/DaoFlex/dist/runtime/src).

The code for the fill portion of the EmployeeDataServiceDAO is presented in Listing 4.6. By wrapping any Throwable into com.theriabook.Clear Data Builder.DAOException, a descendant of RuntimeException, we avoid unnecessary throws in both the DAO and Assembler implementation, since fatal exceptions will bubble up to the Flex framework classes and show up on the client side as a DataServices fault event. Other than that, most of this code is generic, which is precisely why it’s an excellent candidate for template-based code-generation:

```
public final List /*EmployeeDTO[]*/ getEmployees(java.util.Date startDate){
String sql = “select * from employee where start_date < ?”;
ArrayList list = new ArrayList();
PreparedStatement stmt = null;
ResultSet rs = null;
Connection conn = null;
try{
conn = JDBCConnection.getConnection(“jdbc/theriabook”);
stmt = conn.prepareStatement(sql);
stmt.setDate(1, DateTimeConversion.toSqlDate(startDate));
rs = stmt.executeQuery();
while( rs.next() )
{
EmployeeDTO dto = new EmployeeDTO();
dto.EMP_ID = (rs.getInt(“EMP_ID”));
dto.MANAGER_ID = (rs.getInt(“MANAGER_ID”));
dto.EMP_FNAME = (rs.getString(“EMP_FNAME”));
dto.BIRTH_DATE = DateTimeConversion.toUtilDate(rs.getDate(“BIRTH_DATE”));
. . . . . . . .
list.add(dto);
}
return list;
} catch(Throwable te) {
te.printStackTrace();
throw new DAOException(te);
} finally {
try {rs.close(); rs = null;} catch (Exception e){//your error logging code goes
here}
try {stmt.close(); stmt = null;} catch (Exception e){ //your error logging code
goes here }
JDBCConnection.releaseConnection(conn);
}
}
```

Listing 4.6 Fill-method of the EmployeeDataServiceDAO

4.7 Implementing the Sync-Method of DMS Data Access Object

By definition, the sync-method gets a List of flex.data.ChangeObject elements as an argument. A single ChangeObject can carry the information in an updated original record or, alternatively, a record that is supposed to be deleted or inserted. Since we want to process all changes as a single unit of work, we’ll iterate over the List three times: on the first pass we’ll pick and execute all deletes, then we’ll proceed to all updates, and finally we perform all inserts. This logic is presented in Listing 4.7. Similar to the implementation of the getEmployees() above, our getEmployees_sync() throws only RuntimeExceptions:

```
public final List getEmployees_sync(List items)
{
Connection conn = null;
ChangeObject co = null;
try {
conn = JDBCConnection.getConnection(“jdbc/theriabook”);
Iterator iterator = items.iterator();
while (iterator.hasNext() ) { // Do all deletes first
co = (ChangeObject)iterator.next();
if(co.isDelete()) doDelete_getEmployees(conn, co);
}
iterator = items.iterator();
while (iterator.hasNext()) { // Perform updates
co = (ChangeObject)iterator.next();
if(co.isUpdate()) doUpdate_getEmployees(conn, co);
}
iterator = items.iterator();
while (iterator.hasNext()) { // Finish with inserts
co = (ChangeObject)iterator.next();
if (co.isCreate()) doCreate_getEmployees(conn, co);
}
} catch(DataSyncException dse) {
dse.printStackTrace();
throw dse;
} catch(Throwable te) {
te.printStackTrace();
throw new DAOException(te.getMessage(), te);
} finally {
if( conn!=null ) JDBCConnection.releaseConnection(conn);
}
return items;
}
```

Listing 4.7 Sync-method of the EmployeeDataServiceDao

The next topic is the methods doUpdate…(), doDelete…() and doInsert…().

4.8 Implementing Update, Delete and Insert Methods

Let’s start with doUpdate_getEmployees(). Ultimately, we have to dynamically build a JDBC SQL string for the PreparedStatement. As an example, given a change of salary, phone number, and insurance coverage, we need to build a string UPDATE EMPLOYEE SET SALARY=?, PHONE=?, BENE_HEALTH_INS=? WHERE EMP_ID=?. After that, we have to execute a preparedStatement.setXXX() for all parameters to substitute the question marks.

Conveniently, the creators of DMS enabled the ChangedObject to return an array of all property names that underwent modification, which lets us build a SET clause by iterating over the array returned by the co.getChangedPropertyNames():

```
StringBuffer sql = new StringBuffer(“UPDATE EMPLOYEE SET “);
String [] names = co.getChangedPropertyNames();
for (int ii=0; ii < names.length; ii++) {
sql.append((ii!=0?”, “:””) + names[ii] +” = ? “);
}
```

Now let’s set the values for all the modified fields. Here we’ll take advantage of another function of the ChangeObject – getChangedValues(). This method returns a map of the new values and based on this map and the array names we can execute relevant setXXX() methods against our prepared statement:

```
Map values = co.getChangedValues();
int ii=0;
for (ii=0; ii < names.length; ii++) {
stmt.setObject( ii+1, values.get(names[ii]));
}
ii++;
```

To set the value of the WHERE clause-based parameter EMP_ID, we’ll use another ChangeObject’s method – getPreviousVersion(), which returns a copy of the old DTO:

stmt.setObject(ii++, co.getPreviousValue(“EMP_ID”));

The complete listing of doUpdate_getEmployees() is presented in Listing 4.8:

```
private void doUpdate_getEmployees(Connection conn, ChangeObject co) throws
SQLException{
PreparedStatement stmt = null;
try {
StringBuffer sql = new StringBuffer(“UPDATE EMPLOYEE SET “);
String [] names = co.getChangedPropertyNames();
for (int ii=0; ii < names.length; ii++) {
sql.append((ii!=0?”, “:””) + names[ii] +” = ? “);
}
sql.append( “ WHERE (EMP_ID=?)” );
stmt = conn.prepareStatement(sql.toString());
Map values = co.getChangedValues();
int ii=0;
for (ii=0; ii < names.length; ii++) {
stmt.setObject( ii+1, values.get(names[ii]));
}
ii++;
stmt.setObject(ii++, co.getPreviousValue(“EMP_ID”));
if (stmt.executeUpdate()==0) throw new DataSyncException(co);
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;} catch (Exception e){}
}
}
```

Listing 4.8 Update section of the sync-method of EmployeeDataServiceDao

The implementation of the doDelete_getEmployees() is trivial and is presented in Listing 4.9:

```
private void doDelete_getEmployees(Connection conn, ChangeObject co) throws
SQLException{
PreparedStatement stmt = null;
try {
StringBuffer sql = new StringBuffer(“DELETE FROM EMPLOYEE WHERE (EMP_ID=?)”);
stmt = conn.prepareStatement(sql.toString());
EmployeeDTO item = (EmployeeDTO) co.getPreviousVersion();
stmt.setInt(1, item.EMP_ID);
if (stmt.executeUpdate()==0) throw new DataSyncException(co);
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;
} catch (Exception e){// error processing goes here}
}
}
```

Listing 4.9 The delete section of the sync-method of the EmployeeDataServiceDao

Implementation of the doCreate_getEmployees(), in turn, is based on the ChangeObject’s method getNewVersion(), which returns the copy of the new DTO:

EmployeeDTO item = (EmployeeDTO) co.getNewVersion();

Please note how we rely on Double.isNaN() to distinguish nulls from non-nulls (the alternative and, arguably, more reliable approach applicable to all nullable types would be to supply explicit null indicators as part of the DTO from ActionScript to Java and vice versa):

```
if (Double.isNaN(item.SALARY))
stmt.setNull(13,Types.DOUBLE);
else
stmt.setDouble(13, item.SALARY);
```

The abbreviated listing of doCreate_getEmployees() is presented in Listing 4.10:

```
private ChangeObject doCreate_getEmployees(Connection conn, ChangeObject co)
throws
SQLException{
PreparedStatement stmt = null;
try {
String sql = “INSERT INTO EMPLOYEE “ +
“(EMP_ID,MANAGER_ID,EMP_FNAME,EMP_LNAME,DEPT_ID,STREET,CITY,STATE,ZIP_
CODE,PHONE,STATUS,SS_NUMBER,SALARY,START_DATE,TERMINATION_DATE,BIRTH_DATE,BENE_H
EALTH_
INS,BENE_LIFE_INS,BENE_DAY_CARE,SEX)”+
“ values (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)”;
stmt = conn.prepareStatement(sql);
EmployeeDTO item = (EmployeeDTO) co.getNewVersion();
stmt.setInt(1, item.EMP_ID);
. . . .
stmt.setString(12, item.SS_NUMBER);
if (Double.isNaN(item.SALARY))
stmt.setNull(13,Types.DOUBLE);
else
stmt.setDouble(13, item.SALARY);
stmt.setDate(14, DateTimeConversion.toSqlDate(item.START_DATE));
. . . .
if (stmt.executeUpdate()==0) throw new DAOException(“Failed inserting.”);
co.setNewVersion(item);
return co;
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;} catch (Exception e){}
}
}
```

Listing 4.10 The insert section of the sync-method of the EmployeeDataServiceDao

And this concludes the handcrafting of our DataServices-based example. Now that we’ve been through the whole process, let’s see how it could have been avoided and automated.

4.9 Introducing Metadata

Let’s look at the snippet from the XML file generated by the Clear Data Builder utility – Employee.xml. Please note the name of the Java package – com.theriabook.datasource, the name of the Assembler’s fillmethod – getEmployees(), names on the transferring structures on the Java and ActionsScript side, both pointing to the array of com.theriabook.dto.EmployeeDTO objects, the name of the connection pool – jdbc/theriabook, and the name of the method’s parameter – startDate:

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

```
<WEBSERVICE NAME=”Employee” PACKAGE=”com.theriabook.datasource” TYPE=”DAOFlex” >
<SERVER LANGUAGE=”Java” MODE=”JEE”>
<SQL ACTION=”SELECT”
NAME=”getEmployees” POOL=”jdbc/theriabook” SCOPE=”public”
ASTYPE=”com.theriabook.dto.EmployeeDTO[]” JAVATYPE=”com.theriabook.dto.
EmployeeDTO[]”
>
<PARAM IN=”Y” INDEX=”1” JAVATYPE=”Date” NAME=”startDate”/>
</SQL>
</SERVER>
</WEBSERVICE>
```

Starting at this point, we’ll be working our way through this XML while building the complete XSL stylesheet from scratch. Once we make this stylesheet, it’ll manufacture any DataServiceEmployeeDAO, DataServiceDepartmentDAO, etc. – as long as we have the metadata XMLs like the above one. You’re probably wondering at this point: “What’s the input of the Clear Data Builder that lets it generate this XML?”

The input for Clear Data Builder is an annotated Java class, like the one presented in Listing 4.11:

```
package com.theriabook.datasource;
import java.util.Date;
import java.util.List;
/**
* @DAOFlex:webservice pool=jdbc/theriabook
*/
public abstract class Employee {
/**
* @DAOFlex:sql
* sql=select * from employee where start_date < :startDate or start_date=:start-
Date
* transferType=com.theriabook.dto.EmployeeDTO[]
* updateTable=employee
* keyColumns=emp_id
*/
public abstract List getEmployees(Date startDate);
```

Listing 4.11 Source of the Clear Data Builder code-generation process – An annotated Java class

4.10 Introducing Templates

Let’s put out our first iteration of the SimpleDataServiceDao.xsl stylesheet, which will eventually automatically build DataServiceDao objects for us. We assume that the reader is familiar with XSL basics, but if you haven’t had a chance to use XSL yet, here’s a good place to start: http://www.w3schools.com/xsl/.

We’ll be using XSL transformations to produce not an XML, but a plain text (Java code), so let’s start our stylesheet as follows:

```
<?xml version=”1.0”?>
<xsl:stylesheet
version=”1.0”
xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”
>
<xsl:output omit-xml-declaration=”yes” method=”text”/>
….
</xsl:stylesheet>
```

For the topmost element in our metadata (WEBSERVICE) we’ll print out the value of the package,

required utility imports, declaration of the class, and its constructor (Listing 4.12):

```
<xsl:template match=”/WEBSERVICE”>
/* Generated by SimpleDataServiceDao.xsl */
package <xsl:value-of select=”@PACKAGE”/>;
import java.util.*;
import java.sql.*;
import flex.data.*;
import com.theriabook.DAOFlex.JDBCConnection;
import com.theriabook.DAOFlex.DAOException;
import com.theriabook.DAOFlex.DateTimeConversion;
public final class <xsl:value-of select=”@NAME”/>SimpleDataServiceDAO extends
<xsl:
value-of select=”@NAME”/>
{
public <xsl:value-of select=”@NAME”/>SimpleDataServiceDAO()
{
}
<xsl:apply-templates select=”SERVER[@MODE=’JEE’]/SQL”/>
} //<xsl:value-of select=”@NAME”/>SimpleDataServiceDAO
</xsl:template>
```

Listing 4.12. The first cut of the SimpleDataServiceDao.xsl

We’ll discuss the <xsl:apply-templates select=”SERVER[MODE=’JEE’]/SQL”/> a bit later. For now, let’s look at the output that we’d get if we applied the stylesheet to the Employee.XML:

```
/* Generated by SimpleDataServiceDao.xsl */
package com.theriabook.datasource;
import java.util.*;
import flex.data.*;
import java.sql.*;
import com.theriabook.DAOFlex.JDBCConnection;
import com.theriabook.DAOFlex.DAOException;
import com.theriabook.DAOFlex.DateTimeConversion;
public final class EmployeeDataServiceDAO extends Employee
{
public EmployeeDataServiceDAO()
{
}
} //EmployeeDataServiceDAO
```

Listing 4.13 The output of the transformation per the SimpleDataServiceDao.xsl

The tag <xsl:apply-templates …/> from Listing 4.11 effectively delegates the processing of all nodes that can be located relative to the WEBSERVICE via the XPath expression “SERVER[MODE=’JEE’]/ SQL” to a template that matches “SQL.” Before adding such a template, let’s look a bit deeper at the Employee.xml metadata file.

4.11 Metadata for Input Parameters

A closer look at the metadata produced by the Clear Data Builder utility against the source file Employee.java from Listing 4.10 will reveal two sections with SQL. The first one – WEBSERVICE/SERVICE/SQL/BODY – contains the SQL in its source form, while the other – WEBSERVICE/SERVICE/SQL/BODY/COMPILED – contains the same SQL in a form applicable for the JDBC PreparedStatement. The COMPILED section also contains the result of the matching of the original parameters against the JDBC question mark placeholders:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<WEBSERVICE NAME=”Employee” PACKAGE=”com.theriabook.datasource” TYPE=”DAOFlex”
VERSION=”2.0”>
<SERVER LANGUAGE=”Java” MODE=”JEE”>
<SQL ACTION=”SELECT”
NAME=”getEmployees” POOL=”jdbc/theriabook” SCOPE=”public”
ASTYPE=”com.theriabook.dto.EmployeeDTO[]”
JAVATYPE=”com.theriabook.dto.EmployeeDTO[]”
>
```

```
<PARAM IN=”Y” INDEX=”1” JAVATYPE=”Date” NAME=”startDate”/>
<BODY >
<![CDATA[ select * from employee where start_date < :startDate or
start_date=:start-
Date ]]>
</BODY>
<COMPILED>
<PARAM IN=”Y” INDEX=”1” JAVATYPE=”Date” NAME=”startDate” />
<PARAM IN=”Y” INDEX=”2” JAVATYPE=”Date” NAME=”startDate” />
<BODY>
<![CDATA[ select * from employee where start_date < ? or start_date=? ]]>
</BODY>
</COMPILED>
</SQL>
</SERVER>
</WEBSERVICE>
```

Listing 4.14 A second look at the Employee.xml metadata with input parameters

4.12 Templates for Implementing the Fill Method

Let’s modify the stylesheet to generate the fill-method by adding the template matching the SQL context. The abbreviated form of this template is presented further down in Listing 4.16. First have a look at the code that this template generates:

```
public final List /*com.theriabook.datasource.dto.EmployeeDTO[]*/
getEmployees(java.
util.Date startDate)
{
String sql = “select * from employee where start_date < ? or start_date=?”;
ArrayList list = new ArrayList();
PreparedStatement stmt = null;
ResultSet rs = null;
Connection conn = null;
try {
conn = JDBCConnection.getConnection(“jdbc/theriabook”);
stmt = conn.prepareStatement(sql);
// ....
}
return list;
} catch(Throwable te) {
```

```
te.printStackTrace();
throw new DAOException(te);
} finally
{
try {rs.close(); rs = null;} catch (Exception e){// log your errors here}
try {stmt.close(); stmt = null;} catch (Exception e){ // log your errors here }
JDBCConnection.releaseConnection(conn);
}
}
```

Listing 4.15 The output of the <xsl:template match=”SQL”/> against Employee.xml

Listing 4.16 presents a template that generates the code above. Please note that this template in turn delegates the processing of the input parameters to the named template declareFillParameters as seen in Listing 4.17. The template is abbreviated and we’ll be looking at what’s hidden behind the commented ellipses “//…” in the next section of the chapter:

```
<xsl:template match=”SQL”>
public final List /*<xsl:value-of select=”@JAVATYPE”/>*/ <xsl:value-of
select=”@NAME”/
>(<xsl:call-template name=”declareFillParameters”/>
String sql = “<xsl:value-of select=”COMPILED/BODY”/>”;
public final List /*<xsl:value-of select=”@JAVATYPE”/>*/ <xsl:value-of
select=”@NAME”/
>(<xsl:call-template name=”declareFillParameters”/>)
{
String sql = “<xsl:value-of select=”COMPILED/BODY”/>”;
ArrayList list = new ArrayList();
PreparedStatement stmt = null;
ResultSet rs = null;
Connection conn = null;
try {
conn = JDBCConnection.getConnection(“<xsl:value-of select=”@POOL”/>”);
stmt = conn.prepareStatement(sql);
// . . . .
return list;
} catch(Throwable te) {
te.printStackTrace();
throw new DAOException(te);
} finally
{
try {rs.close(); rs = null;} catch (Exception e){ // log your errors here }
try {stmt.close(); stmt = null;} catch (Exception e){ // log your errors here }
JDBCConnection.releaseConnection(conn);
}
}
</xsl:template>
```

Listing 4.16 This template generates the fill method for each SQL context

Here is the auxiliary template that helped us generate a declaration of parameters for the fill-method. It puts a comma after each parameter, except the latest and narrows the definition of the Date to java.util.Date to avoid ambiguity between java.sql and java.util packages imported at the beginning of the class:

```
<xsl:template name=”declareFillParameters”>
<xsl:for-each select=”PARAM[@IN=’Y’]”>
<xsl:if test=”position()!=1”>, </xsl:if>
<xsl:choose>
<xsl:when test=”@JAVATYPE=’Date’”>java.util.Date</xsl:when>
<xsl:otherwise><xsl:value-of select=”@JAVATYPE”/></xsl:otherwise>
</xsl:choose>
<xsl:text> </xsl:text>
<xsl:value-of select=”@NAME”/>
</xsl:for-each>
</xsl:template>
```

Listing 4.17 The template declareFillParameters, used by <xsl:template match=”SQL”/>

4.13 Completing the Fill Method

Let’s upgrade our stylesheet to a state where it can generate a fully functional fill-method. In the case of the Employee.xml metadata, we’d like to see our template generate the code presented in Listing 4.18:

```
public final List /*com.theriabook.datasource.dto.EmployeeDTO[]*/
getEmployees(java.
util.Date startDate)
{
String sql = “select * from employee where start_date < ? or start_date=?”;
ArrayList list = new ArrayList();
PreparedStatement stmt = null;
ResultSet rs = null;
Connection conn = null;
try {
```

```
conn = JDBCConnection.getConnection("jdbc/theriabook");
stmt = conn.prepareStatement(sql);
stmt.setDate(1, DateTimeConversion.toSqlDate(startDate));
stmt.setDate(2, DateTimeConversion.toSqlDate(startDate));
rs = stmt.executeQuery();
while( rs.next() ) {
com.theriabook.datasource.dto.EmployeeDTO dto = new com.theriabook.data
source.dto.EmployeeDTO();
dto.EMP_ID = (rs.getInt("EMP_ID"));
dto.MANAGER_ID = (rs.getInt("MANAGER_ID"));
dto.EMP_FNAME = (rs.getString("EMP_FNAME"));
dto.EMP_LNAME = (rs.getString("EMP_LNAME"));
dto.DEPT_ID = (rs.getInt("DEPT_ID"));
dto.STREET = (rs.getString("STREET"));
dto.CITY = (rs.getString("CITY"));
dto.STATE = (rs.getString("STATE"));
dto.ZIP_CODE = (rs.getString("ZIP_CODE"));
dto.PHONE = (rs.getString("PHONE"));
dto.STATUS = (rs.getString("STATUS"));
dto.SS_NUMBER = (rs.getString("SS_NUMBER"));
dto.SALARY = (rs.getDouble("SALARY"));
dto.START_DATE = DateTimeConversion.toUtilDate(rs.getDate("START_DATE"));
dto.TERMINATION_DATE = DateTimeConversion.toUtilDate(rs.getDate("TERMINATION_
DATE"));
dto.BIRTH_DATE = DateTimeConversion.toUtilDate(rs.getDate("BIRTH_DATE"));
dto.BENE_HEALTH_INS = (rs.getString("BENE_HEALTH_INS"));
dto.BENE_LIFE_INS = (rs.getString("BENE_LIFE_INS"));
dto.BENE_DAY_CARE = (rs.getString("BENE_DAY_CARE"));
dto.SEX = (rs.getString("SEX"));
list.add(dto);
}
return list;
} catch(Throwable te) {
te.printStackTrace();
throw new DAOException(te);
} finally {
try {rs.close(); rs = null;} catch (Exception e){}
try {stmt.close(); stmt = null;} catch (Exception e){}
```

```
JDBCConnection.releaseConnection(conn);
}
}
```

Listing 4.18 Complete fill-method getEmployees() generated by SimpleDataServiceDao.xsl

To “teach” our stylesheet to produce this code, we’ll replace the try clause generated by the <template match=”SQL”> currently containing:

```
try {
conn = JDBCConnection.getConnection(“<xsl:value-of select=”@POOL”/>”);
stmt = conn.prepareStatement(sql);
// . . . .
return list;
}
```

with the following:

```
try {
conn = JDBCConnection.getConnection(“<xsl:value-of select=”@POOL”/>”);
stmt = conn.prepareStatement(sql);
<xsl:call-template name=”setParameters”/>
rs = stmt.executeQuery();
while( rs.next() ) {
<xsl:variable name=”itemType” select=”substring(string(@JAVA
TYPE), 1, string-length(string(@JAVATYPE))-2)”/> <xsl:value-of
select=”$itemType”/> dto = new <xsl:value-of select=”$itemType”/>();
<xsl:call-template name=”readRecord”/>
list.add(dto);
}
return list;
}
```

As you can see, we’ve delegated the work of calling the setXXX() methods to the named template setParameters and reading the records of the result – set to the named template readRecord. Let’s walk through these templates one at a time.

4.14 Setting JDBC Statement Parameters

While setting the input arguments of the prepared statement, we’ll have to convert the Java types of the parameters listed in the COMPILED node into JDBC types. In particular, we have to convert the Date in our use case to java.sql.Date to accommodate the following lines:

```
stmt.setDate(1, DateTimeConversion.toSqlDate(startDate));
stmt.setDate(2, DateTimeConversion.toSqlDate(startDate));
```

We've centralized this and similar conversions under the named template convertJavaArgument-ToJDBC:

```
<xsl:template name="convertJavaArgumentToJDBC">
<xsl:choose>
<xsl:when test="@JAVATYPE='Date'">DateTimeConversion.toSqlDate(<xsl:value-of
select="@NAME"/>)</xsl:when>
<xsl:when test="@JAVATYPE='Time'">DateTimeConversion.toSqlTime(<xsl:value-of
select="@NAME"/>)</xsl:when>
<xsl:otherwise>
<xsl:value-of select="@NAME"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
```

```
After taking care of the Java-to-JDBC conversion, the template setParameters
becomes easy:
<xsl:template name="setParameters">
<xsl:for-each select="COMPILED/PARAM[@IN='Y']">
stmt.set<xsl:value-of select="@JAVATYPE"/>(<xsl:value-of select="@INDEX"/>,
<xsl:call-template name="convertJavaArgumentToJDBC"/>);</xsl:for-each>
</xsl:template>
```

4.15 Reading the Result Set Record

Reading the result set record would require another look at the metadata. For each SQL annotated method, the Clear Data Builder utility generates a description of the result set. To do that, Clear Data Builder connects to the target database during the generation of the metadata. Connection credentials are expected in the properties file named exactly as the JNDI data source name, and our file is called theriabook. properties and is located in the Clear Data Builder's JDBC folder. Here is the snippet of metadata that represents the description of the result set associated with the getEmployees() method :

```
<SQL … NAME="getEmployees" . . .>
<DATASET>
<FIELDS>
<FIELD key="yes" name="EMP_ID" precision="11" scale="0" tableName="employee"
type="integer" updatable="yes" />
<FIELD name="MANAGER_ID" precision="11" scale="0" tableName="employee"
type="integer"
updatable="yes" />
<FIELD name="EMP_FNAME" precision="20" scale="0" tableName="employee"
type="char"
```

```
updatable=”yes” />
<FIELD name=”EMP_LNAME” precision=”20” scale=”0” tableName=”employee”
type=”char”
updatable=”yes” />
<FIELD name=”DEPT_ID” precision=”11” scale=”0” tableName=”employee”
type=”integer”
updatable=”yes” />
<FIELD name=”STREET” precision=”40” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
<FIELD name=”CITY” precision=”20” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
<FIELD name=”STATE” precision=”4” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
<FIELD name=”ZIP_CODE” precision=”9” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
<FIELD name=”PHONE” precision=”10” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
<FIELD name=”STATUS” precision=”1” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
<FIELD name=”SS_NUMBER” precision=”11” scale=”0” tableName=”employee”
type=”char”
updatable=”yes” />
<FIELD name=”SALARY” precision=”20” scale=”3” tableName=”employee”
type=”decimal”
updatable=”yes” />
<FIELD name=”START_DATE” precision=”10” scale=”0” tableName=”employee”
type=”date”
updatable=”yes” />
<FIELD name=”TERMINATION_DATE” precision=”10” scale=”0” tableName=”employee”
type=”date” updatable=”yes” />
<FIELD name=”BIRTH_DATE” precision=”10” scale=”0” tableName=”employee”
type=”date”
updatable=”yes” />
<FIELD name=”BENE_HEALTH_INS” precision=”1” scale=”0” tableName=”employee”
type=”char”
updatable=”yes” />
<FIELD name=”BENE_LIFE_INS” precision=”1” scale=”0” tableName=”employee”
type=”char”
updatable=”yes” />
<FIELD name=”BENE_DAY_CARE” precision=”1” scale=”0” tableName=”employee”
type=”char”
```

```
updatable=”yes” />
<FIELD name=”SEX” precision=”1” scale=”0” tableName=”employee” type=”char”
updatable=”yes” />
</FIELDS>
</DATASET>
</SQL>
```

Listing 4.19 The description of the result set produced by Clear Data Builder

There is one more point to make before we can look at the implementation of the readRecord template. The data types per DATASET/FIELDS/FIELD nodes are database-specific types, not JDBC ones. For example, we have to apply additional mapping to produce getString() for the char database columns and getDouble() for the decimal ones. This mapping is provided by the named template mapDBtoJDBC (you may need to tweak it a bit for your DBMS):

```
<xsl:template name=” mapDBtoJDBC”>
<xsl:choose>
<xsl:when test=”@type=’boolean’”>Boolean</xsl:when>
<xsl:when test=”@type =’byte’”>Byte</xsl:when>
<xsl:when test=”@type =’byte[]’”>Bytes</xsl:when>
<xsl:when test=”@type =’char’”>String</xsl:when>
<xsl:when test=”@type=’date’”>Date</xsl:when>
<xsl:when test=”@type=’datetime’”>Timestamp</xsl:when>
<xsl:when test=”@type=’decimal’”>Double</xsl:when>
<xsl:when test=”@type=’double’”>Double</xsl:when>
<xsl:when test=”@type=’float’”>Float</xsl:when>
<xsl:when test=”@type=’int’”>Int</xsl:when>
<xsl:when test=”@type=’integer’”>Int</xsl:when>
<xsl:when test=”@type=’lvarchar’”>String</xsl:when>
<xsl:when test=”@type=’money’”>Double</xsl:when>
<xsl:when test=”@type=’nchar’”>String</xsl:when>
<xsl:when test=”@type=’nvarchar’”>String</xsl:when>
<xsl:when test=”@type=’nvarchar2’”>String</xsl:when>
<xsl:when test=”@type=’number’ and @scale=’0’”>Long</xsl:when>
<xsl:when test=”@type=’number’ and @scale!=’0’”>Double</xsl:when>
<xsl:when test=”@type=’numeric’”>Double</xsl:when>
<xsl:when test=”@type=’smallint’”>Int</xsl:when>
<xsl:when test=”@type=’smallfloat’”>Float</xsl:when>
<xsl:when test=”@type=’text’”>String</xsl:when>
<xsl:when test=”@type=’time’”>Time</xsl:when>
<xsl:when test=”@type=’timestamp’”>Timestamp</xsl:when>
```

```
<xsl:when test=”@type=’varchar’”>String</xsl:when>
<xsl:when test=”@type=’varchar2’”>String</xsl:when>
<xsl:otherwise>Object</xsl:otherwise>
</xsl:choose></xsl:template>
```

Listing 4.20 Template mapping database types to JDBC ones

And now, let’s look at the readRecord template, which iterates over all the fields of the result set and generates lines like:

```
dto.EMP_ID = (rs.getInt(“EMP_ID”));
dto.SALARY = (rs.getDouble(“SALARY));
```

In addition, JDBC-related date/time types get converted from java.sql to java.util form:

dto.START_DATE = DateTimeConversion.toUtilDate(rs.getDate(“START_DATE”));

Here is the readRecord template:

```
<xsl:template name=”readRecord”>
<xsl:for-each select=”DATASET/FIELDS/FIELD”>dto.<xsl:value-of
select=”@name”/> = <xsl:choose>
<xsl:when test=”string(@type)=’date’”>DateTimeConversion.toUtilDate</xsl:when>
<xsl:when test=”string(@type)=’datetime’”>DateTimeConversion.toUtilDate</xsl:
when>
<xsl:when test=”string(@type)=’time’”>DateTimeConversion.toUtilDate</xsl:when>
<xsl:when test=”string(@type)=’timestamp’”>DateTimeConversion.toUtilDate</xsl:
when>
<xsl:otherwise></xsl:otherwise>
</xsl:choose>(rs.get<xsl:call-template name=”mapDBtoJDBC”/>(“<xsl:value-of
select=”@name”/>”));
</xsl:for-each>
</xsl:template>
```

This concludes the complete implementation of the fill-method. The sync-method templating comes next.

4.16 Templates for Implementing Sync-Method

We’ll be generating the sync-method using precisely the same metadata that we’ve extracted from the annotated Java class presented in Listing 4.10. We’re particularly interested in the attribute updateTable=employee of the @DAOFlex:sql tag that tells us which table from the SELECT statement should be used as an update target (your select statement can have more than one table, but the current version of the Clear Data Builder can update only one).

Listing 4.21 presents the XML metadata we’ve been working with in this chapter, but it’s the first time that we show the UPDATE node. As expected, the UPDATE node holds the name of the table to update. It also indicates that the WHERE clause of the generated INSERT/DELETE/UPDATE statements should be based on the key fields (as opposed to the other alternatives: modified fields and the combination key-and-modified):

```
<?xml version="1.0" encoding="UTF-8"?>
<WEBSERVICE NAME="Employee" PACKAGE="com.theriabook.datasource" TYPE="DAOFlex" VERSION="
2.0">
<SERVER LANGUAGE="Java" MODE="JEE">
<SQL ACTION="SELECT"
NAME="getEmployees" POOL="jdbc/theriabook" SCOPE="public"
ASTYPE="com.theriabook.dto.EmployeeDTO[]" JAVATYPE="com.theriabook.dto.
EmployeeDTO[]"
>
<PARAM IN="Y" INDEX="1" JAVATYPE="Date" NAME="startDate"/>
. . . .
</COMPILED >
<UPDATE TARGET="EMPLOYEE">
<TABLE NAME="EMPLOYEE" UPDATEMETHOD="key"/>
</UPDATE>
<DATASET><FIELDS>
<FIELD key="yes" name="EMP_ID" tableName="employee" type="integer"
updatable="yes"/>
. . . .
</FIELDS></DATASET>
</SQL>
</SERVER>
</WEBSERVICE>
```

Listing 4.21 The Clear Data Builder metadata with information for the sync-method

Interestingly, the top-level code for the sync-method doesn't really depend on any of this. Indeed, we need to process the entire input List of ChangeObjects. We'll do it in three passes (as we did in a manual mode in Listing 4.7) and execute all DELETEs first, with UPDATEs and INSERTs, but again, the code will be pretty agnostic relative to the underlying SQL. Listing 4.22 presents the template that automatically generates such a sync-method given the content of the <SQL /> metadata node as the context:

```
<xsl:template match="SQL" mode="update">
public final List <xsl:value-of select="@NAME"/>_sync(List items)
{
Connection conn = null;
ChangeObject co = null;
try {
```

```
conn = JDBCConnection.getConnection("<xsl:value-of select="@POOL"/>");
Iterator iterator = items.iterator();
while (iterator.hasNext() ) { // Do all deletes first
co = (ChangeObject)iterator.next();
if(co.isDelete()) doDelete_<xsl:value-of select="@NAME"/>(conn, co);
}
iterator = items.iterator();
while (iterator.hasNext()) { // Proceed to all updates next
co = (ChangeObject)iterator.next();
if(co.isUpdate()) doUpdate_<xsl:value-of select="@NAME"/>(conn, co);
}
iterator = items.iterator();
while (iterator.hasNext()) { // Finish with inserts
co = (ChangeObject)iterator.next();
if (co.isCreate()) doCreate_<xsl:value-of select="@NAME"/>(conn, co);
}
} catch(DataSyncException dse) {
dse.printStackTrace();
throw dse;
} catch(Throwable te) {
te.printStackTrace();
throw new DAOException(te.getMessage(), te);
} finally {
if( conn!=null ) JDBCConnection.releaseConnection(conn);
}
return items;
</xsl:template>
```

Listing 4.22 The template to generate the"top-level" of the sync-method

Now, to make sure this template accompanies the original <xsl:template match="SQL"> we'll modify the latter with the <xsl:apply-templates> as shown in the following snippet:

```
<xsl:template match="SQL">
<xsl:variable name="itemType" select="substring(string(@JAVATYPE), 1, string-
length(st
ring(@JAVATYPE))-2)"/>
public final List /*<xsl:value-of select="@JAVATYPE"/>*/ <xsl:value-of
select="@NAME"/
>(<xsl:call-template name="declareFillParameters"/>)
```

```
{
. . . .
}
<xsl:if test=”@ACTION=’SELECT’ and boolean(UPDATE)”> <xsl:apply-templates
select=”.”
mode=”update”/></xsl:if>
</xsl:template>
```

Since we’re transforming our Employee.xml metadata with the SimpleDataServiceDao.xsl in its current state, the relevant fragment of the output will look like Listing 4.23:

```
public final List searchEmployees_sync(List items)
{
logger.debug(“searchEmployees_sync(...)”);
Connection conn = null;
ChangeObject co = null;
try {
conn = JDBCConnection.getPooledConnection(“jdbc/theriabook”);
Iterator iterator = items.iterator();
while (iterator.hasNext() ) { // Do all deletes first
co = (ChangeObject)iterator.next();
if(co.isDelete()) doDelete_searchEmployees(conn, co);
}
iterator = items.iterator();
while (iterator.hasNext()) { // Proceed to all updates next
co = (ChangeObject)iterator.next();
if(co.isUpdate()) doUpdate_searchEmployees(conn, co);
}
iterator = items.iterator();
while (iterator.hasNext()) { // Finish with inserts
co = (ChangeObject)iterator.next();
if (co.isCreate()) doCreate_searchEmployees(conn, co);
}
} catch(DataSyncException dse) {
dse.printStackTrace();
throw dse;
} catch(Throwable te) {
te.printStackTrace();
throw new DAOException(te.getMessage(), te);
} finally {
```

```
if( conn!=null ) JDBCConnection.releasePooledConnection(conn);
}
return items;
}
```

Listing 4.23 Top-level sync-method code generated by our template

Our next task is to generate doUpdate(), doDelete(), and doInsert() methods.

4.17 Completing the Sync Method

We'll delegate the generation process of the doUpdate(), doDelete(), and doInsert() methods to three correspondingly named templates. We'll alter the template presented in Listing 4.22 as shown below:

```
<xsl:template match="SQL" mode="update">
public final List <xsl:value-of select="@NAME"/>_sync(List items)
{
. . . .
}
<xsl:call-template name="doUpdate" />
<xsl:call-template name="doDelete" />
<xsl:call-template name="doInsert" />
</xsl:template>
```

Let's start with doUpdate. Remember our exercise with handcrafted DataService-based code? That doUpdate() contained a StringBuffer of the UPDATE statement. Given that our XML context is the <SQL> node we can generate the required line as

StringBuffer sql = new StringBuffer("UPDATE <xsl:value-of select="UPDATE/@TARGET"/> SET ");

To produce the WHERE clause, which will enumerate all the key columns in the form "key1=?, key2=?" we can do the following:

```
sql.append( " WHERE (<xsl:for-each select="DATASET/FIELDS/
FIELD[@key='yes']"><xsl:if test="position()!=1"> AND </xsl:if><xsl:value-of
select="@name"/>=?</xsl:for-each>)" );
```

Then, to substitute the "?" symbols with the key values we will employ the similar <xsl:for-each/>: combined with what we learned about the ChangedObject's getPreviousValue() method:

```
<xsl:for-each select="DATASET/FIELDS/FIELD[@key='yes']">
stmt.setObject(ii++, co.getPreviousValue("<xsl:value-of select="@name"/>"));</
xsl:for-each>
```

Ultimately we'll arrive at the text of the doUpdate template as shown in Listing 4.24:

```
<xsl:template name=”doUpdate” >
private void doUpdate_<xsl:value-of select=”@NAME”/>(Connection conn,
ChangeObject co)
throws SQLException{
PreparedStatement stmt = null;
try {
StringBuffer sql = new StringBuffer(“UPDATE <xsl:value-of select=”UPDATE/
@TARGET”/> SET “);
String [] names = co.getChangedPropertyNames();
for (int ii=0; ii &lt; names.length; ii++) {
sql.append((ii!=0?”, “:””) + names[ii] +” = ? “);
}
sql.append( “ WHERE (<xsl:for-each select=”DATASET/FIELDS/
FIELD[@key=’yes’]”><xsl:if test=”position()!=1”> AND </xsl:if><xsl:value-of
select=”@name”/>=?</xsl:for-each>)” );
stmt = conn.prepareStatement(sql.toString());
Map values = co.getChangedValues();
int ii=0;
for (ii=0; ii &lt; names.length; ii++) {
stmt.setObject( ii+1, values.get(names[ii]));
}
ii++;
<xsl:for-each select=”DATASET/FIELDS/FIELD[@key=’yes’]”>
stmt.setObject(ii++, co.getPreviousValue(“<xsl:value-of select=”@name”/>”));</
xsl:for-each>
if (stmt.executeUpdate()==0) throw new DataSyncException(co);
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;} catch (Exception e){}
}
}
</xsl:template>
```

Listing 4.24 The template of the doUpdate() method

The template implementing the doDelete method is very similar, except that, due to the syntax of the DELETE statement it ventures into enumerating the WHERE-hosted keys right off the bat while preparing the StringBuffer:

```
<xsl:template name=”doDelete” >
private void doDelete_<xsl:value-of select=”@NAME”/>(Connection conn,
ChangeObject co)
```

```
throws SQLException{
PreparedStatement stmt = null;
try {
StringBuffer sql = new StringBuffer("DELETE FROM <xsl:value-of select="UPDATE/
@TARGET"/> WHERE (<xsl:for-each
select="DATASET/FIELDS/FIELD[@key='yes']"><xsl:if
test="position()!=1"> AND </xsl:if><xsl:value-of select="@name"/>=?</xsl:for-
each>)");
stmt = conn.prepareStatement(sql.toString());
<xsl:for-each select="DATASET/FIELDS/FIELD[@key='yes']">
stmt.setObject(<xsl:value-of select="position()"/>, co.getPreviousValue("<x
sl:value-of select="@name"/>"));
</xsl:for-each>
if (stmt.executeUpdate()==0) throw new DataSyncException(co);
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;} catch (Exception e){}
}
}
</xsl:template>
```

Listing 4.25 The template of the doDelete() method

We did not list the results of the XSL transformation here because they are literally identical to the handcrafted code we modeled our templates after.

We're almost there. The only remaining part of the SimpleDataServiceDao.xsl to discuss is the doCreate template, which will be covered next.

4.18 The Template for the doCreate() Method

It's time to do another kind of mapping exercise. We've been through such an exercise when reading the ResultSet with the getXXX() functions. Knowing the database type of the result set element we had to determine the proper choice of the getInt(), getString(), etc., and on top of that, perform the conversions like converting java.sql.Date into java.util.Date. On the same note, we have to pick the right setInt(), setString(), and the like, based on the database data type and do the proper conversions.

The named template "mapDBtoJDBC" (see Listing 4.20) comes in handy again. Provided the context is a metadata node <FIELD>, we can generate the proper setXXX() call with the line:

stmt.set<xsl:call-template name="mapDBtoJDBC"/>(arguments);

To perform the type conversion, we'll add another named template, convertJavaColumnToJDBC. It expects to use the value of the database column's type and an expression for conversion. For the date/time/timestamp types it converts the Java value of the DTO property into the corresponding value from the java.sql package leaving other expressions intact:

```
<xsl:template name="convertJavaColumnToJDBC">
<xsl:param name="type"/>
```

```
<xsl:param name=”expression”/>
<xsl:when test=”type=’date’”>DateTimeConversion.toSqlDate(<xsl:value-of
select=”$expression”/>)</xsl:when>
<xsl:when test=”type=’datetime’”>DateTimeConversion.toSqlTimestamp(<xsl:value-of
select=”$expression”/>)</xsl:when>
<xsl:when test=”type=’time’”>DateTimeConversion.toSqlTime(<xsl:value-of
select=”$expression”/>)</xsl:when>
<xsl:when test=”type=’timestamp’”>DateTimeConversion.toSqlTimestamp(<xsl:value-
of
select=”$expression”/>)</xsl:when>
<xsl:otherwise><xsl:value-of
select=”$expression”/></xsl:otherwise></xsl:choose></xsl:
template>
```

Listing 4.26 The template of converting date/time targeting values to java.sql.* values

And here is the last component. In the absence of dedicated null indicators traveling along with the DTO, the only way to set NULL values for types like double and float is to have an if statement similar to:

```
if (Double.isNaN(item.SALARY))
stmt.setNull(4,Types.DOUBLE);
else
stmt.setDouble(4, item.SALARY);
```

Putting it all together, we arrive at the template writeRecord. It starts with converting source expressions like item.BIRTH_DATE, item.START_DATE, etc., into a jdbcValue. Then, depending on the FIELD type, it generates an if-statement like the one above or outputs a setXXX() statement with the code:

```
stmt.set<xsl:call-template name=”mapDBtoJDBC”/>(<xsl:value-of
select=”position()”/>,
<xsl:value-of
select=”$jdbcValue”/>);
```

```
The full listing of the writeRecord template follows:
<xsl:template name=”writeRecord”>
<xsl:for-each select=”DATASET/FIELDS/FIELD”>
<xsl:variable name=”jdbcValue”>
<xsl:call-template name=”convertJavaColumnToJDBC”>
<xsl:with-param name=”type” select=”@type”/>
<xsl:with-param name=”exp”>item.<xsl:value-of select=”@name”/>
</xsl:with-param>
</xsl:call-template>
```

```
</xsl:variable>
<xsl:choose>
<xsl:when test="@type='double' or @type='decimal' or @type='money' or
(@type='number' and @scale!='0') or @type='numeric'">
if (Double.isNaN(item.<xsl:value-of select="@name"/>))
stmt.setNull(<xsl:value-of select="position()"/>,Types.DOUBLE);
else
stmt.setDouble(<xsl:value-of select="position()"/>, item.<xsl:value-of
select="@name"/>);</xsl:when>
<xsl:when test="@type='float' or @type='smallfloat'">
if (Float.isNaN(item.<xsl:value-of select="@name"/>))
stmt.setNull(<xsl:value-of select="position()"/>,Types.FLOAT);
else
stmt.setFloat(<xsl:value-of select="position()"/>, item.<xsl:value-of
select="@name"/>);</xsl:when>
<xsl:otherwise>
stmt.set<xsl:call-template name="mapDBtoJDBC"/>(<xsl:value-of
select="position()"/>, <xsl:value-of select="$jdbcValue"/>);</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</xsl:template>
```

Listing 4.27 The writeRecord template

If you've made it so far, the XSL template for doCreate will look trivial. It builds the SQL INTO clause of the INSERT iterating over the result set fields, and iterates over it again to place an adequate number of comma-separated "?" characters. Then, on calling conn.prepareStatement() it invokes the template writeRecord. That's it.

```
<xsl:template name="doCreate" >
<xsl:variable name="itemType" select="substring(string(@JAVATYPE), 1, string-
length(st
ring(@JAVATYPE))-2)"/>
private ChangeObject doCreate_<xsl:value-of select="@NAME"/>(Connection conn,
ChangeObject co) throws SQLException{he
PreparedStatement stmt = null;
try {
String sql = "INSERT INTO <xsl:value-of select="UPDATE/@TARGET"/> " +
"(<xsl:for-each select="DATASET/FIELDS/FIELD">
<xsl:value-of select="@name"/><xsl:if test="not(position()=last())">,</xsl:if>
</xsl:for-each>)"+
“ values (<xsl:for-each select=”DATASET/FIELDS/FIELD”>?<xsl:if test=”not(positio
n()=last())”>,</xsl:if></xsl:for-each>)”;
stmt = conn.prepareStatement(sql);
<xsl:value-of select=”$itemType”/> item = (<xsl:value-of select=”$itemType”/>)
co.getNewVersion();
<xsl:call-template name=”writeRecord”/>
if (stmt.executeUpdate()==0) throw new DAOException(“Failed inserting.”);
co.setNewVersion(item);
return co;
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;} catch (Exception e){}
}
}
</xsl:template>
```

Listing 4.28 The template of the doCreate() method

Here is the output of the doCreate() template against the Employee.xml metadata:

```
private ChangeObject doCreate_getEmployees(Connection conn, ChangeObject co)
throws
SQLException{
PreparedStatement stmt = null;
try {
String sql = “INSERT INTO EMPLOYEE “ +
“(EMP_ID,MANAGER_ID,EMP_FNAME,EMP_LNAME,DEPT_ID,STREET,CITY,STATE,ZIP_
CODE,PHONE,STATUS,SS_NUMBER,SALARY,START_DATE,TERMINATION_DATE,BIRTH_DATE,BENE_H
EALTH_
INS,BENE_LIFE_INS,BENE_DAY_CARE,SEX)”+
“ values (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)”;
stmt = conn.prepareStatement(sql);
com.theriabook.dto.EmployeeDTO item = (com.theriabook.dto.EmployeeDTO)
co.getNewVersion();
stmt.setInt(1, item.EMP_ID);
stmt.setInt(2, item.MANAGER_ID);
stmt.setString(3, item.EMP_FNAME);
stmt.setString(4, item.EMP_LNAME);
stmt.setInt(5, item.DEPT_ID);
stmt.setString(6, item.STREET);
stmt.setString(7, item.CITY);
stmt.setString(8, item.STATE);
stmt.setString(9, item.ZIP_CODE);
stmt.setString(10, item.PHONE);
stmt.setString(11, item.STATUS);
stmt.setString(12, item.SS_NUMBER);
if (Double.isNaN(item.SALARY))
stmt.setNull(13,Types.DOUBLE);
else
stmt.setDouble(13, item.SALARY);
stmt.setDate(14, DateTimeConversion.toSqlDate(item.START_DATE));
stmt.setDate(15, DateTimeConversion.toSqlDate(item.TERMINATION_DATE));
stmt.setDate(16, DateTimeConversion.toSqlDate(item.BIRTH_DATE));
stmt.setString(17, item.BENE_HEALTH_INS);
stmt.setString(18, item.BENE_LIFE_INS);
stmt.setString(19, item.BENE_DAY_CARE);
stmt.setString(20, item.SEX);
if (stmt.executeUpdate()==0) throw new DAOException(“Failed inserting.”);
co.setNewVersion(item);
return co;
} finally {
try { if( stmt!=null) stmt.close(); stmt = null;} catch (Exception e){}
}
}
```

Listing 4.29 The output produced by the doCreate template

This makes out SimpleDataServiceDao.xsl complete.

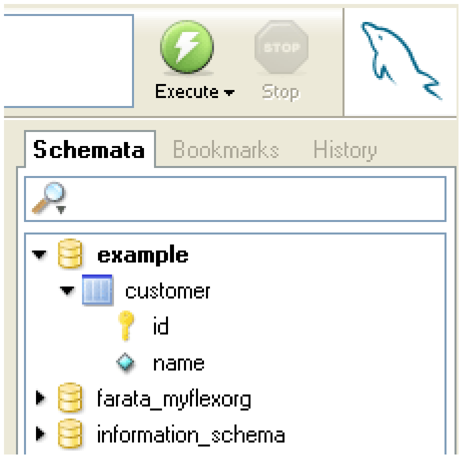
5.0 APPENDIX A. INSTALLING MYSQL SERVER

Download the popular open source database community server MySQL, for example pick the Windows installer at http://dev.mysql.com/downloads/mysql/5.0.html#downloads. Get the file Windows (x86) ZIP/Setup.EXE (if you use Windows) and run the setup.exe, which will install MySql as a service and create MySql menu items in the Windows Start menu. You’ll find MySQL documentation in the directory C:\mysql-5.0.37-win32\Docs

Please pay attention, at the end of install, it’ll ask you about creation of the password for the root user. Do not skip this part to avoid troubles with connecting to MySql Server.

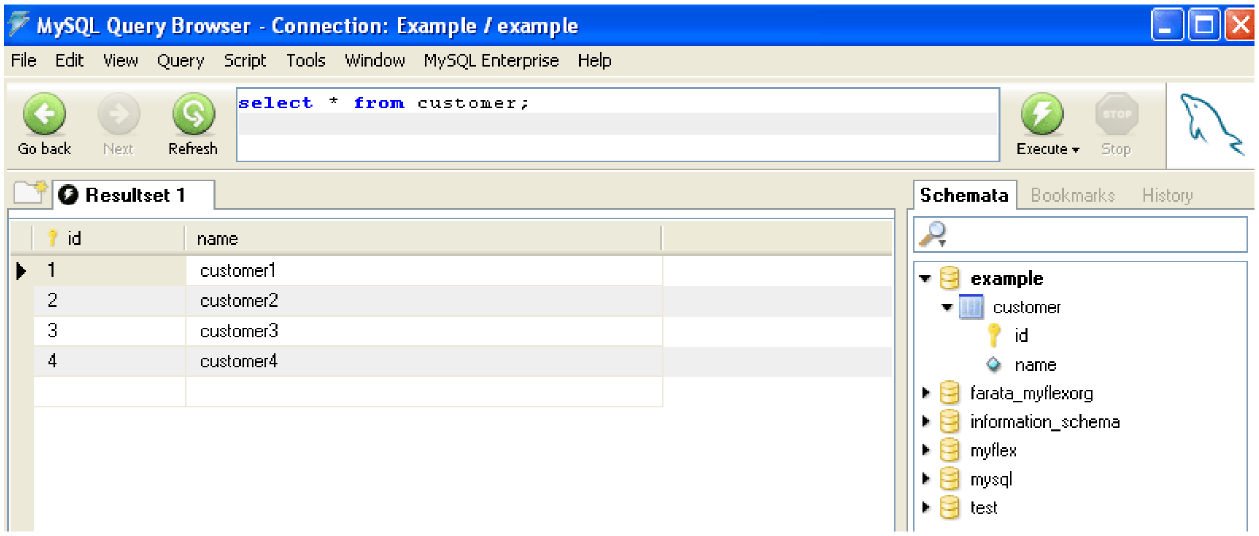
Even though MySQL server allows you to work with the data from a command line, it’s a lot easier to use a GUI client. Download and install MySQL GUI Tools Bundle located at http://mysql.org/downloads/gui-tools/5.0.html . You'll find several new items, and we'll use MySQL Administrator for quick creation of a sample Employees database and MySQL Query Browser for running sample queries.

Start MySql Query browser and use the context menus from the Schemata tab to view existing of create new database schemas:



In MySQL Query Browser open the menu File | Change Default Schema and select test as your default schema.

Enter and execute a test query **select * from customer;** and you should see a result set as shown below:



```
select * from customer;
```

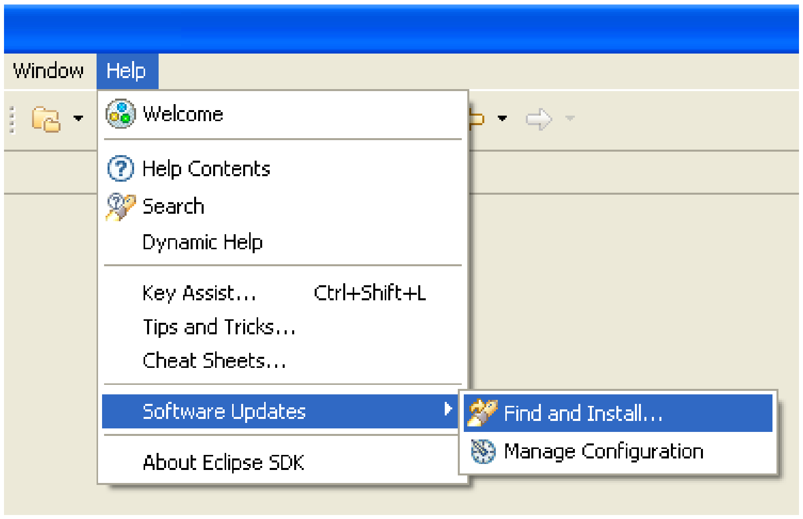
id	name
1	customer1
2	customer2
3	customer3
4	customer4

8. Start MySQL Administrator, select the option User Administration and create a new user: in the User Information tab enter lowercase id *dba* and the password *sql*. Press the tab Schema Privileges, select the test schema and assign to the user all privileges by pressing the button <<. Press the button Apply Changes.

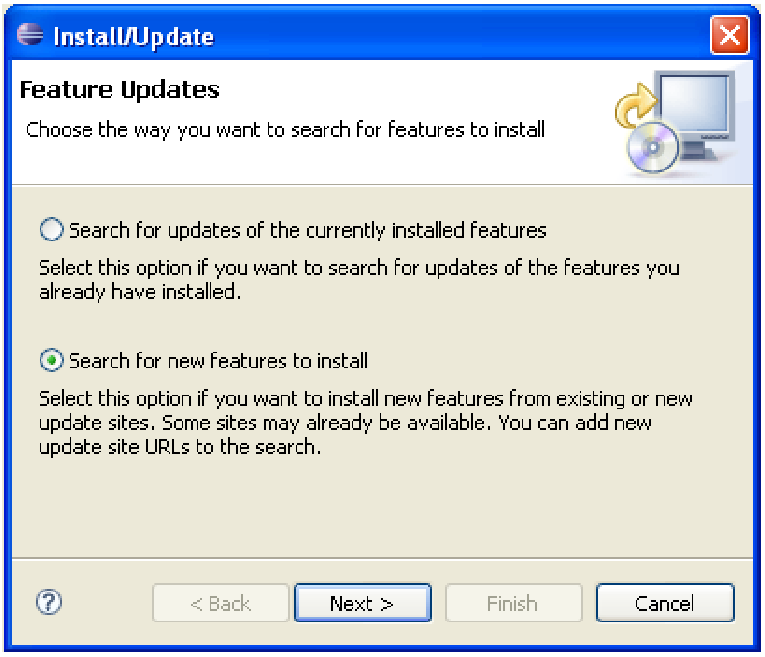
6.0 APPENDIX B. INSTALLING AND CONFIGURING CDB

6.1 Installing CDB 3.1

1. Start Eclipse IDE and select the menus **Help > Software Updates > Find and Install**.



2. Select **Search for new features to install** and press **Next**.


Install/Update

Feature Updates

Choose the way you want to search for features to install

( ) Search for updates of the currently installed features

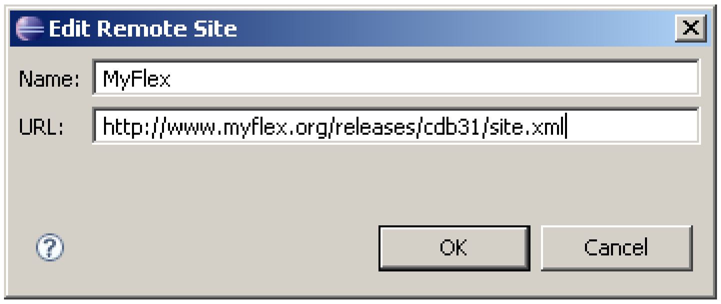
Select this option if you want to search for updates of the features you already have installed.

(●) Search for new features to install

Select this option if you want to install new features from existing or new update sites. Some sites may already be available. You can add new update site URLs to the search.

3. You'll see a popup window with a list of available Web sites with Eclipse-related software. Press the button **New Remote Site...** Enter the name of the site (for example, "myflex.org") and the following URL: http://www.myflex.org/releases/cdb31/site.xml

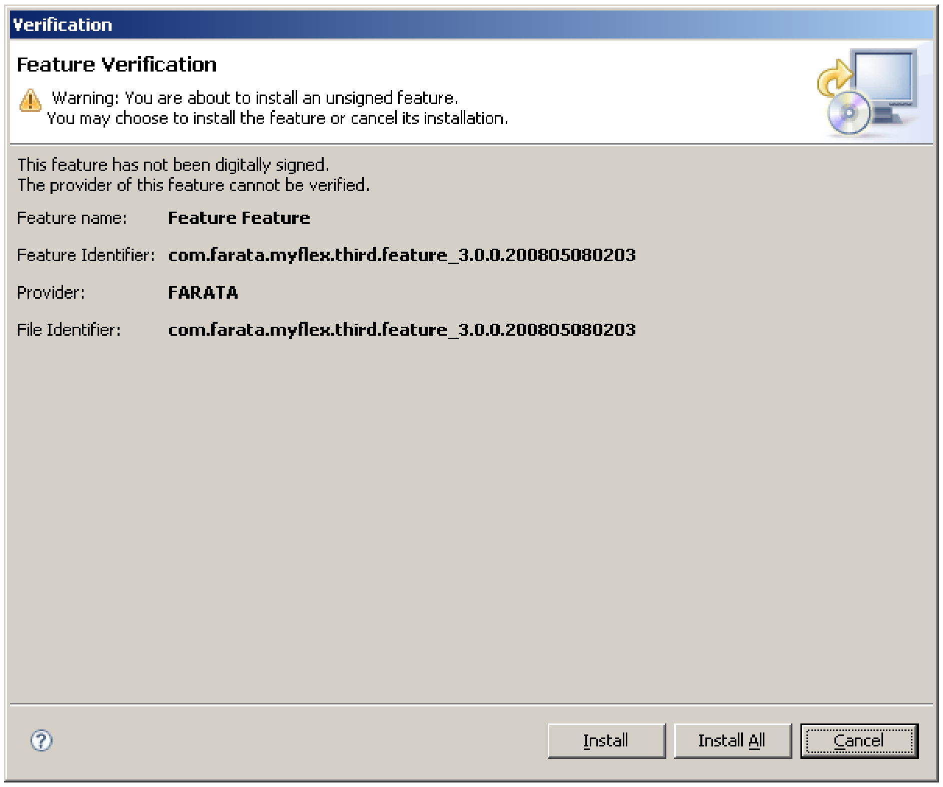
Press **Ok,** and then **Finish**.



Name: MyFlex

URL: http://www.myflex.org/releases/cdb31/site.xml

4. In the popup window shown below, select the checkbox to request install of MyFlex feature and then the button **Next**.. Read the license agreement, and select **"I accept the terms in the license agreements"** and continue the process selecting **Install All** on the verification window.



Feature Verification

Warning: You are about to install an unsigned feature.
You may choose to install the feature or cancel its installation.

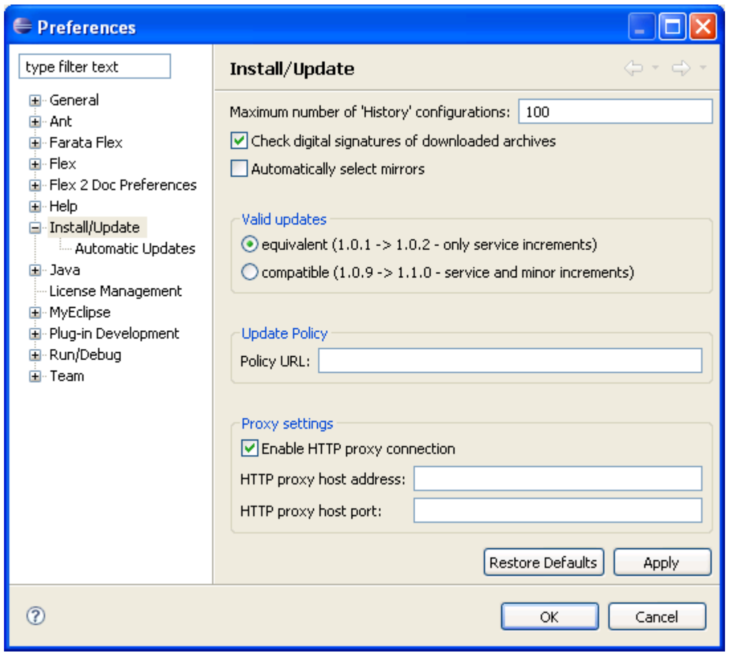
This feature has not been digitally signed.
The provider of this feature cannot be verified.

Feature name:	Feature Feature
Feature Identifier:	com.farata.myflex.third.feature_3.0.0.200805080203
Provider:	FARATA
File Identifier:	com.farata.myflex.third.feature_3.0.0.200805080203

After the installation of CDB is complete, restart Eclipse and install the license file as explained in section **6.3**.

6.2 Enabling Installation of CDB Behind the Firewall

1. If your computer is located behind the firewall, Install/Update from Remote Site will not work, unless you tell Eclipse to use proxy connection. To configure Eclipse for using proxy connection, please go to **Windows > Preferences** and enter the IP address and port of your proxy:



Install/Update

Maximum number of 'History' configurations: 100

[✓] Check digital signatures of downloaded archives

[ ] Automatically select mirrors

Valid updates

(●) equivalent (1.0.1 -> 1.0.2 - only service increments)

( ) compatible (1.0.9 -> 1.1.0 - service and minor increments)

Update Policy

Policy URL:

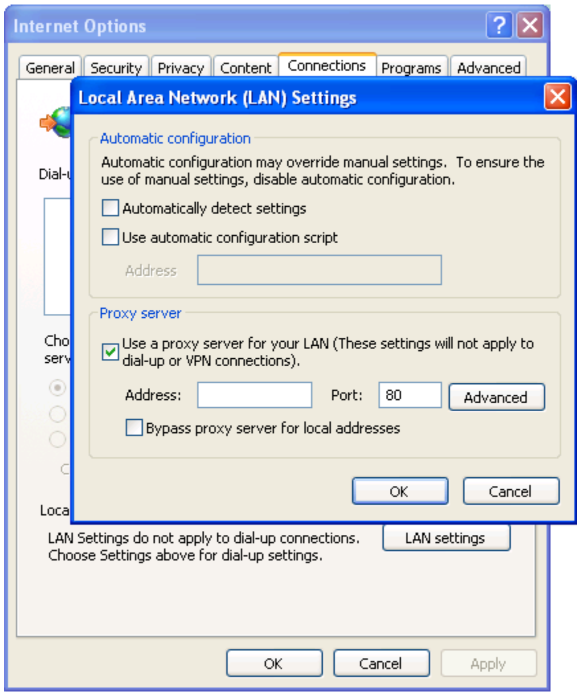
Proxy settings

[✓] Enable HTTP proxy connection

HTTP proxy host address:

HTTP proxy host port:

Note. You may be able to find your proxy by looking up the settings of your browser. Here is where you may find the proxy settings of your Internet Explorer:



Automatic configuration

Automatic configuration may override manual settings. To ensure the use of manual settings, disable automatic configuration.

[ ] Automatically detect settings

[ ] Use automatic configuration script

Address

Proxy server

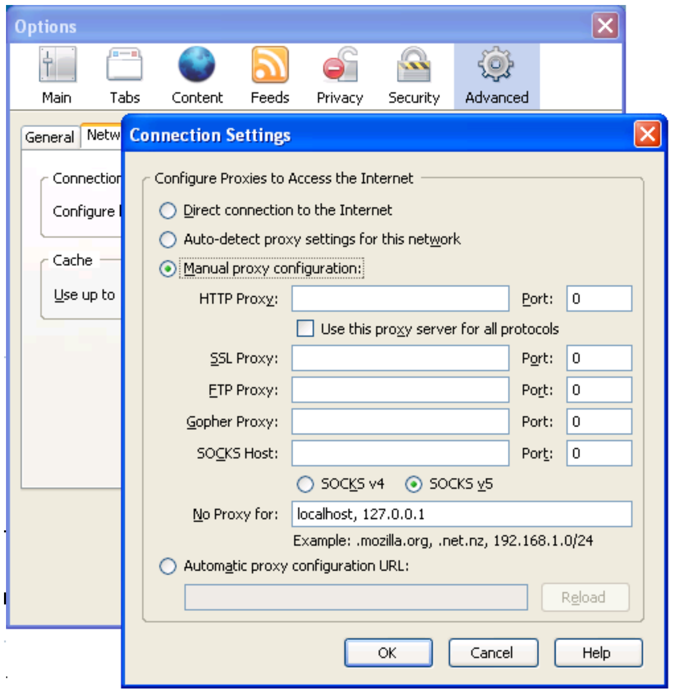
[✓] Use a proxy server for your LAN (These settings will not apply to dial-up or VPN connections).

Address: Port: 80

[ ] Bypass proxy server for local addresses

LAN Settings do not apply to dial-up connections. Choose Settings above for dial-up settings.

Here is how your can lookup settings of your proxy in FireFox:



Connection Settings

Configure Proxies to Access the Internet

- ○ Direct connection to the Internet
- ○ Auto-detect proxy settings for this network
- [✓ Manual proxy configuration:]

HTTP Proxy:		Port:	0
	☐ Use this proxy server for all protocols		
SSL Proxy:		Port:	0
FTP Proxy:		Port:	0
Gopher Proxy:		Port:	0
SOCKS Host:		Port:	0

SOCKS v4 [✓ SOCKS v5]

No Proxy for: localhost, 127.0.0.1

Example: .mozilla.org, .net.nz, 192.168.1.0/24

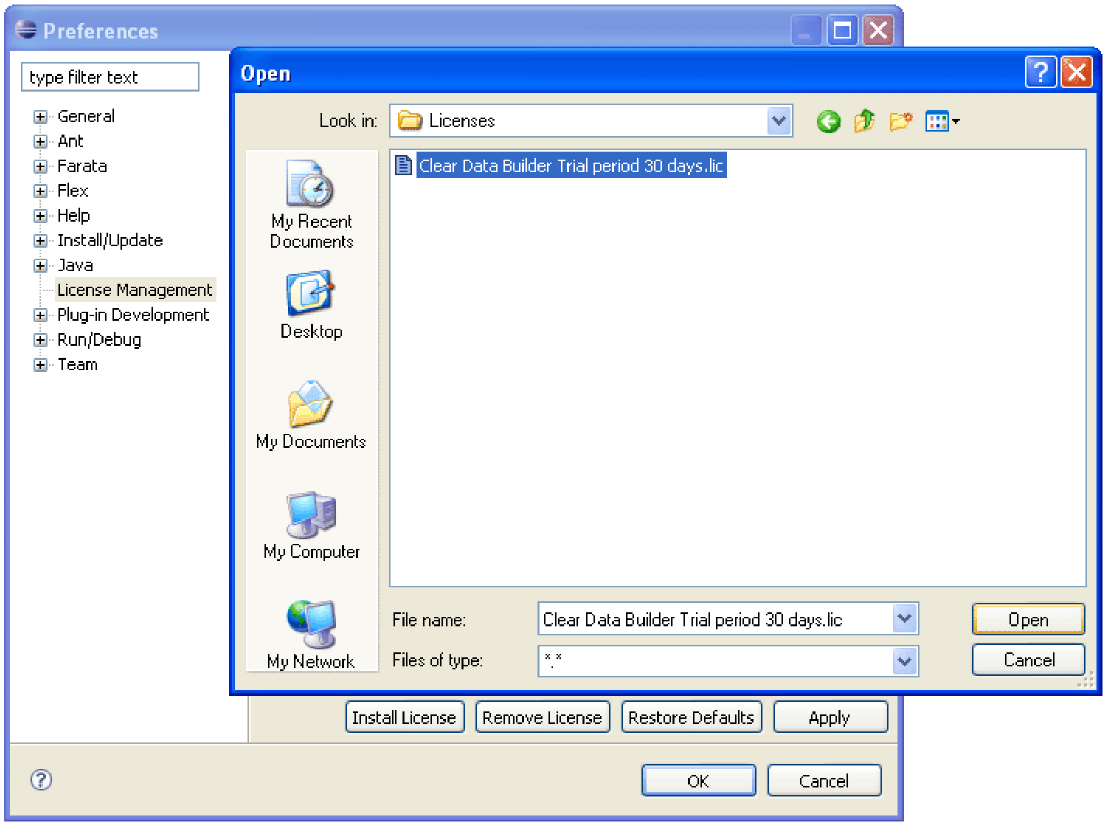
- ○ Automatic proxy configuration URL:

6.3 Installing Clear Data Builder License File

1. Download a free trial or purchase the CDB license at www.myflex.org.

2. Select Eclipse menus **Windows > Preferences**.

3. Select **License Management** from the list on the left. On the popup window press **Install License** and select the downloaded license file.



4. The selected license file becomes your default license. If you did not read the license agreement, please do so by selecting **com.farata.daoflex** in the **Products under license** section. After that, press the button **OK** and restart Eclipse.

7.0 APPENDIX C. CONTACT INFORMATION

To report bugs in Clear Data Builder 3.1, please send an email at support@faratasystems.com. You can also leave a comment under the Contact Us menu at Farata Systems Web site: http://www.faratasystems.com.