Stack & Cloud University

Module:	*Advanced CDT Development*
Topic:	*Lab 1 – Create a Service*
Length:	*Approximately 60 minutes*

STATE STREET

WEB AGE solutions inc.

Investment Management | Investment Research and Trading | Investment Servicing

Lab 1 – Create a Service

Introduction

This lab will step you through the process of creating a service and associated unit test. The service will return a list of employee data (e.g. name, position, dept, etc) from a simulated data store, and will implement eSF-based validation of data entitlements. In a subsequent lab, we will build a user-interface for this data, building a complete application.

Student Prerequisites

For the best experience of this lab, you should be familiar with the following:

- Java programming
- Eclipse IDE
- Windows XP

Learning Prerequisites

- Completed the “Advanced CDT” module, Unit 2 – “Services”

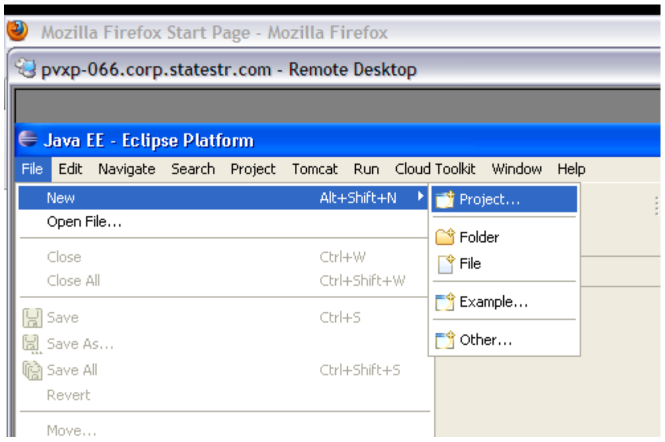
Additional Resources

- For more information about creating Services, visit cdt.statstr.com. Navigate to the “Dev Guide”, and scroll down to the topic, “Services”.
- For more information about the CDT eclipse (version, installation, and troubleshooting), please visit cdt.statestr.com and navigate to the “Dev Guide”, and see the topic, “Getting Started”, and refer to the article, “Installing the CDT environment”.

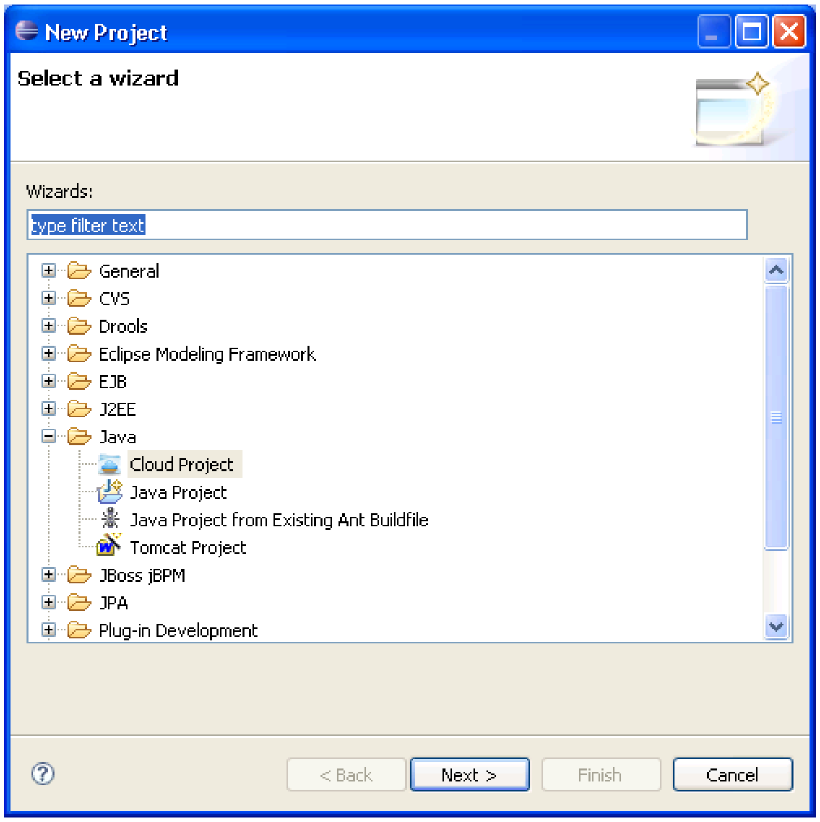
<u>Task: Create a Cloud Project</u>

If necessary, open the CDT Eclipse environment by clicking on the ‘CDT Workspace’ icon on the desktop.
(Shot-4)

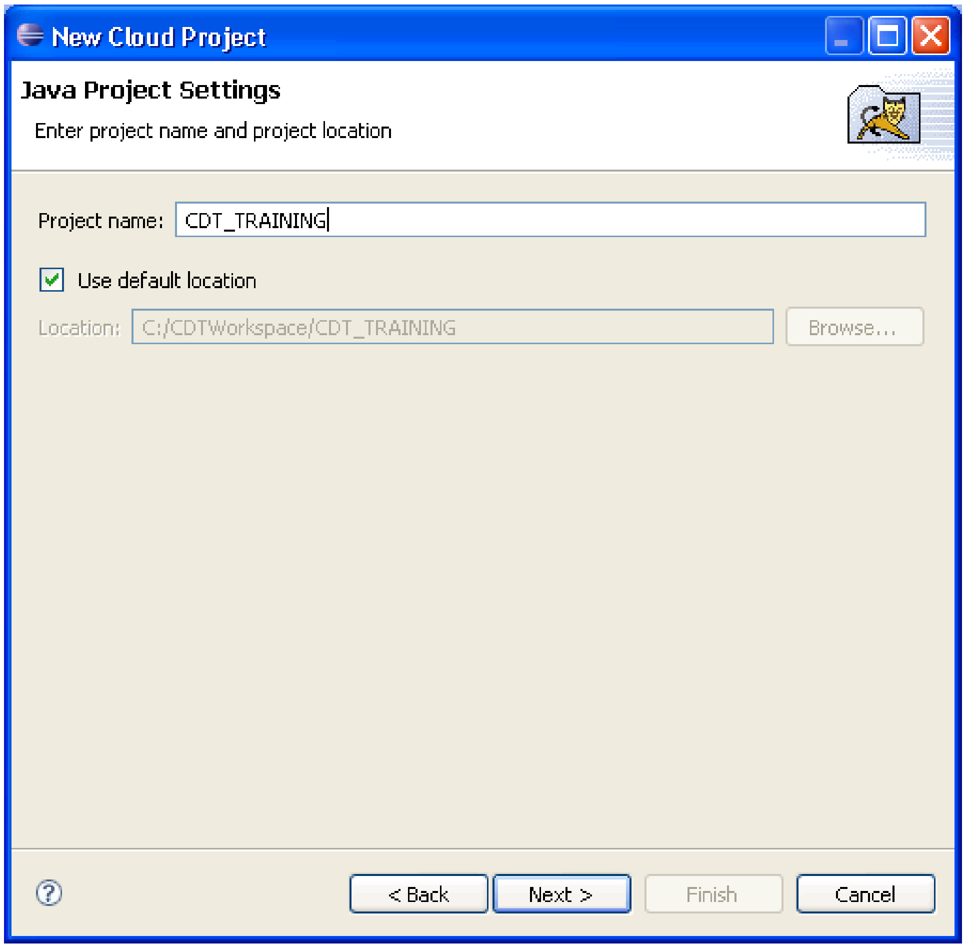
From the Eclipse main menu, select ‘**File -> New -> Project…**’.


Mozilla Firefox Start Page - Mozilla Firefox
pvxp-066.corp.statestr.com - Remote Desktop
Java EE - Eclipse Platform
File Edit Navigate Search Project Tomcat Run Cloud Toolkit Window Help
New Alt+Shift+N
Open File…
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As…
Save All Ctrl+Shift+S
Revert
Move…
Project…
Folder
File
Example…
Other…

When the new project dialog appears, select the ‘Java -> Cloud Project’ node and click ‘**Next >**’.



Enter 'CDT_TRAINING' as the project name, then press '**Next >**'


New Cloud Project

Java Project Settings

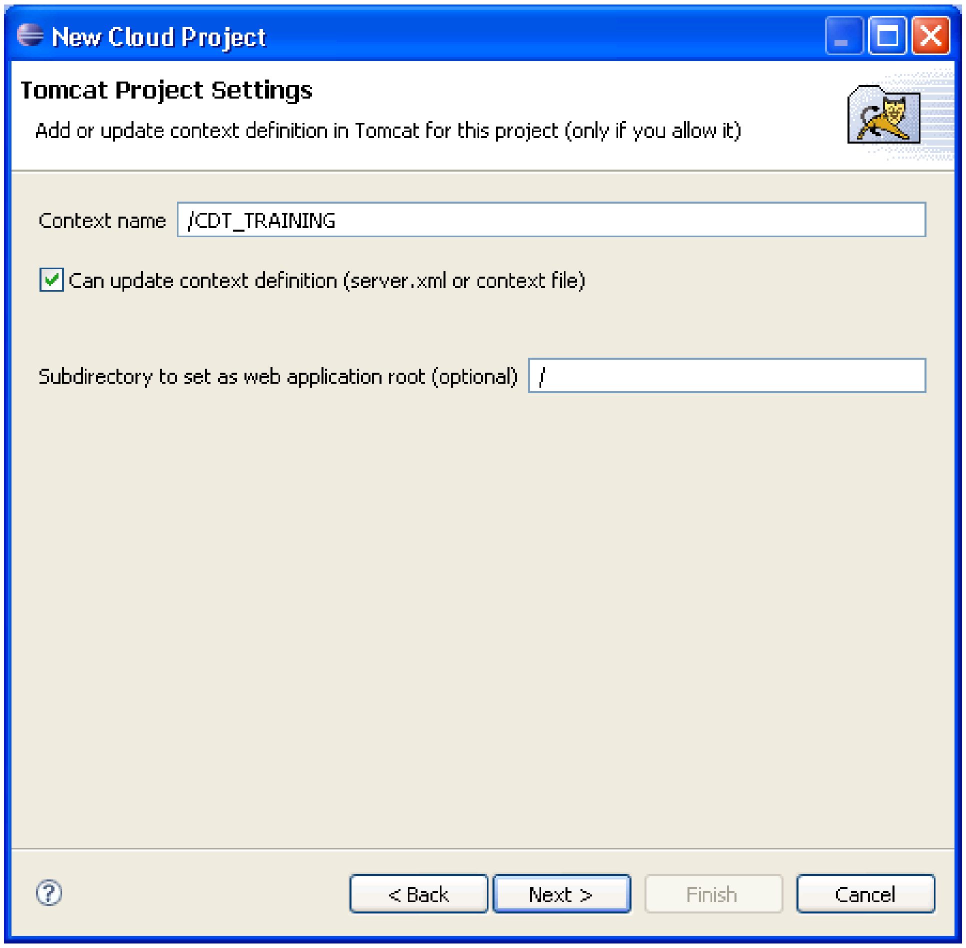
Enter project name and project location

Project name: CDT_TRAINING

☑ Use default location

Location: C:/CDTWorkspace/CDT_TRAINING

Make sure the 'Can update context definition…' box is checked, then press '**Next >**'.


New Cloud Project

Tomcat Project Settings

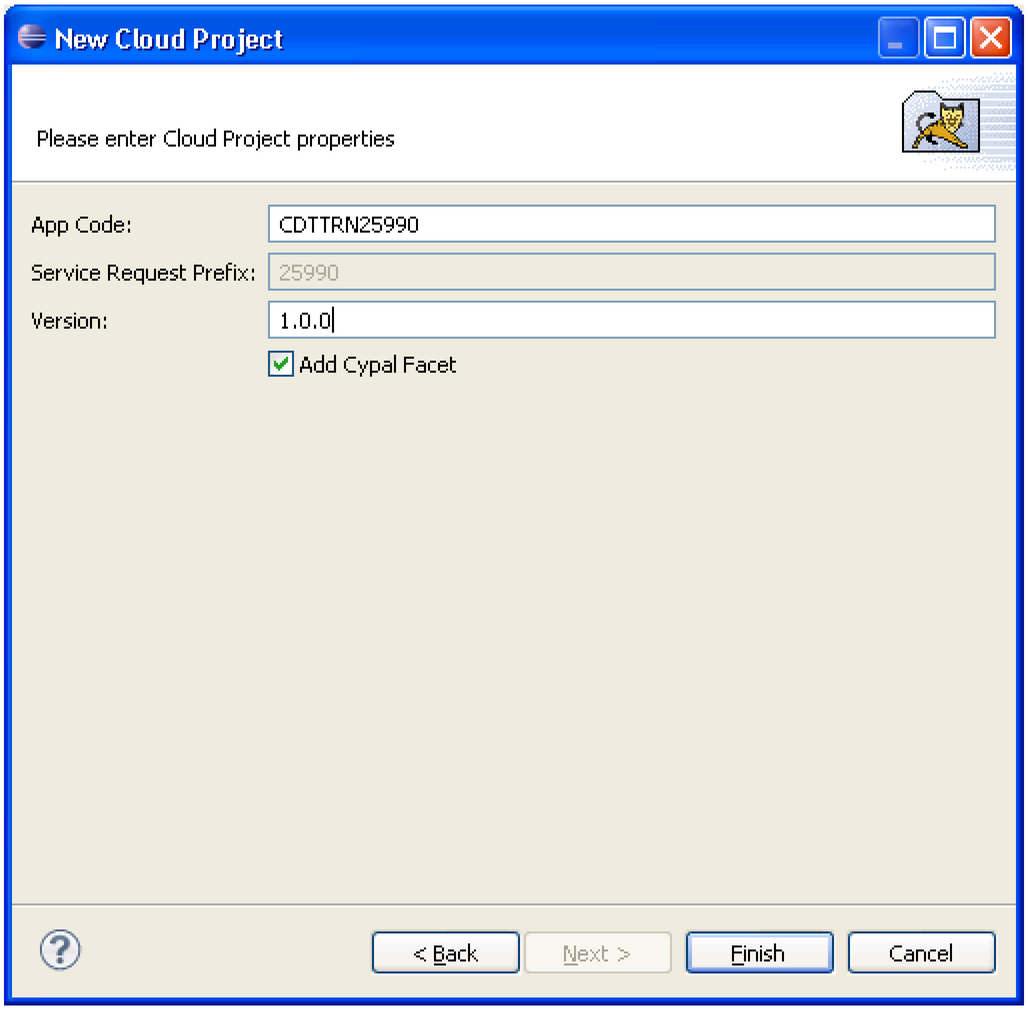
Add or update context definition in Tomcat for this project (only if you allow it)

Context name: /CDT_TRAINING

[✓] Can update context definition (server.xml or context file)

Subdirectory to set as web application root (optional): /

The dialog will prompt for the 'App Code'. For a real service, you would obtain this app code from the cloud team. For our training, we have reserved the app code 'CDTTRN25990'. Enter this code into the New Cloud Project dialog, and enter '0.0.1' as the version, then click '**Finish**'.



Please enter Cloud Project properties

App Code: CDTTRN25990

Service Request Prefix: 25990

Version: 1.0.0

[✓] Add Cypal Facet

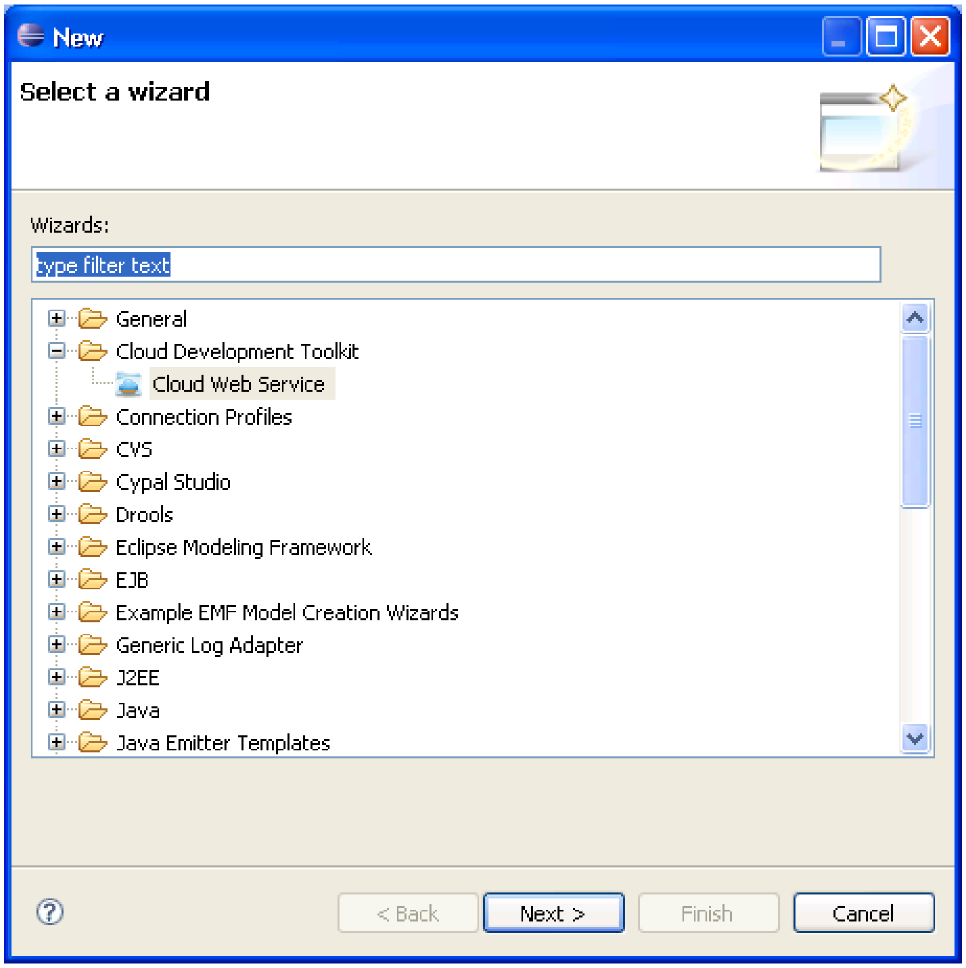
The wizard will create a new project called 'CDT_TRAINING', and register this project in the Tomcat web server's configuration directory.

<u>Task: Create a Service</u>

We will create a service that simulates an employee records data store. We'll start by using the CDT's new service wizard to create a skeleton service and JUnit test case, then we'll write the test code. Finally we'll implement the service so as to pass the tests we write.

Right-click on the 'WEB-INF/src' node in the Project Explorer, then select '**New -> Other…**'. When the dialog appears, select the "Cloud Development Toolkit -> Cloud Web Service" node and click '**Next**

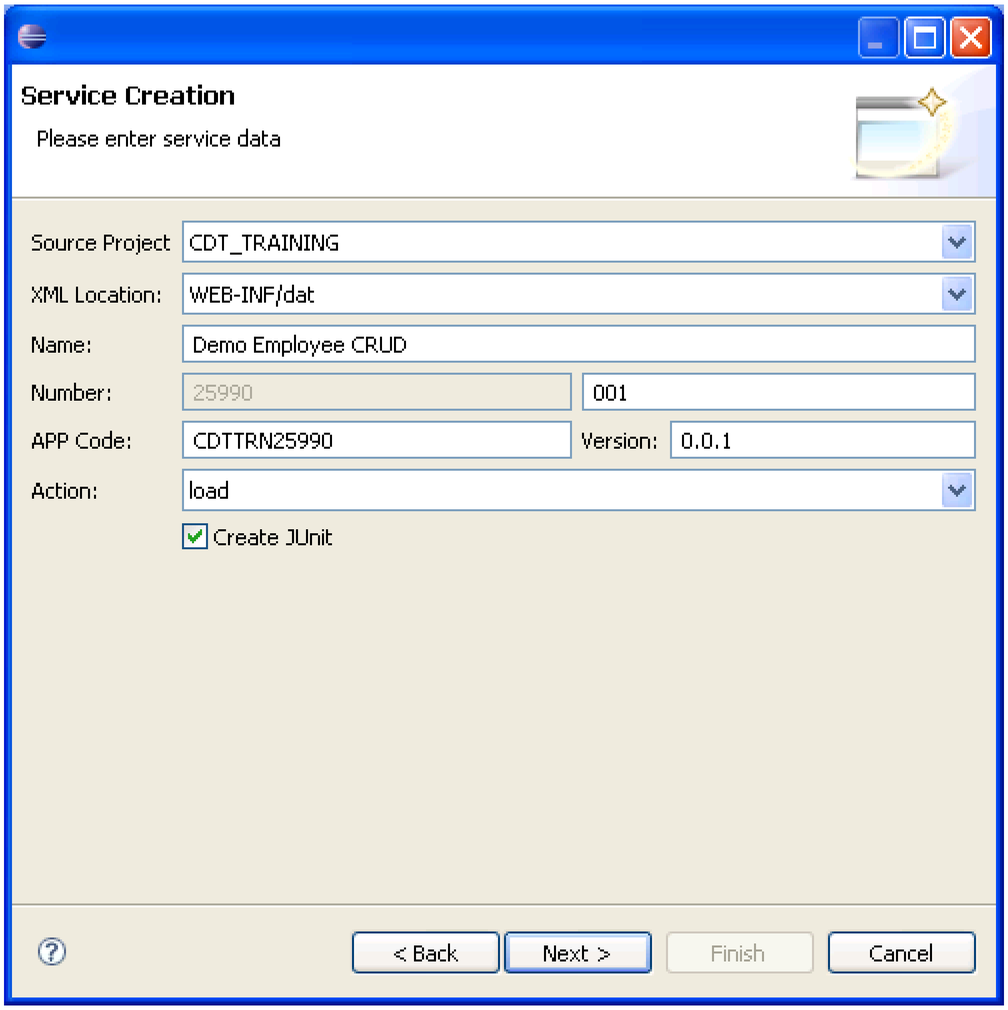
>'.



The system will display the Service Creation dialog. Some of the fields are already filled in using data from the project (e.g. the APP Code). We need to provide a service name and a "request number". The request number for cloud services is constructed from the "request prefix" code that is part of the App ID, plus a 3-5 digit numeric suffix that we'll provide for this service. If an application has more than one service, then we will use a different suffix for each service.

For this first service, enter "Demo Employee CRUD" as the name, and "001" as the number.

Make sure the 'Create JUnit' checkbox is checked. This operation will tell the CDT to create the shell of a JUnit test for the service. Click '**Next >**'



Service Creation

Please enter service data

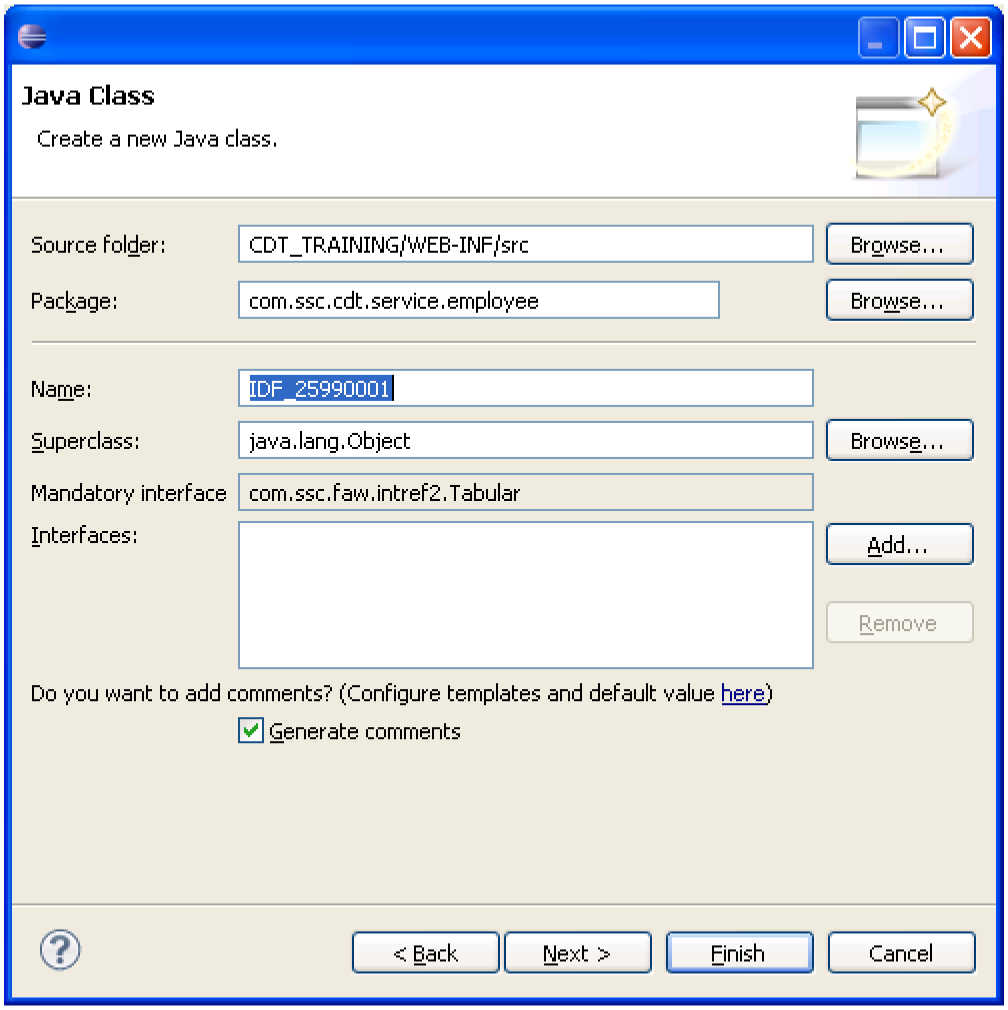
Source Project	CDT_TRAINING		
XML Location:	WEB-INF/dat		
Name:	Demo Employee CRUD		
Number:	25990	001	
APP Code:	CDTTRN25990	Version:	0.0.1
Action:	load		

[✓] Create JUnit

The "Java Class" dialog appears. Here we need to provide a class name and package for the service's action handler. The CDT has already filled in a default name of "IDF_25990001". The meaning of the "IDF" prefix is mainly historical, but the '25990001' part is the request number for the service. This request number is formed from the request prefix portion of the App Code, and the suffix that we provided in the previous dialog. In practice, the cloud controller will use this complete request code to route requests to the correct application in the cloud.

In addition to the class name, we need to provide a package name, which we construct in the usual Java fashion, from the corporate domain name combined with some project-specific name. Enter 'com.ssc.cdt.service.employee' as the package name. Also click 'Generate Comments', and click '**Next**

>'.



Java Class

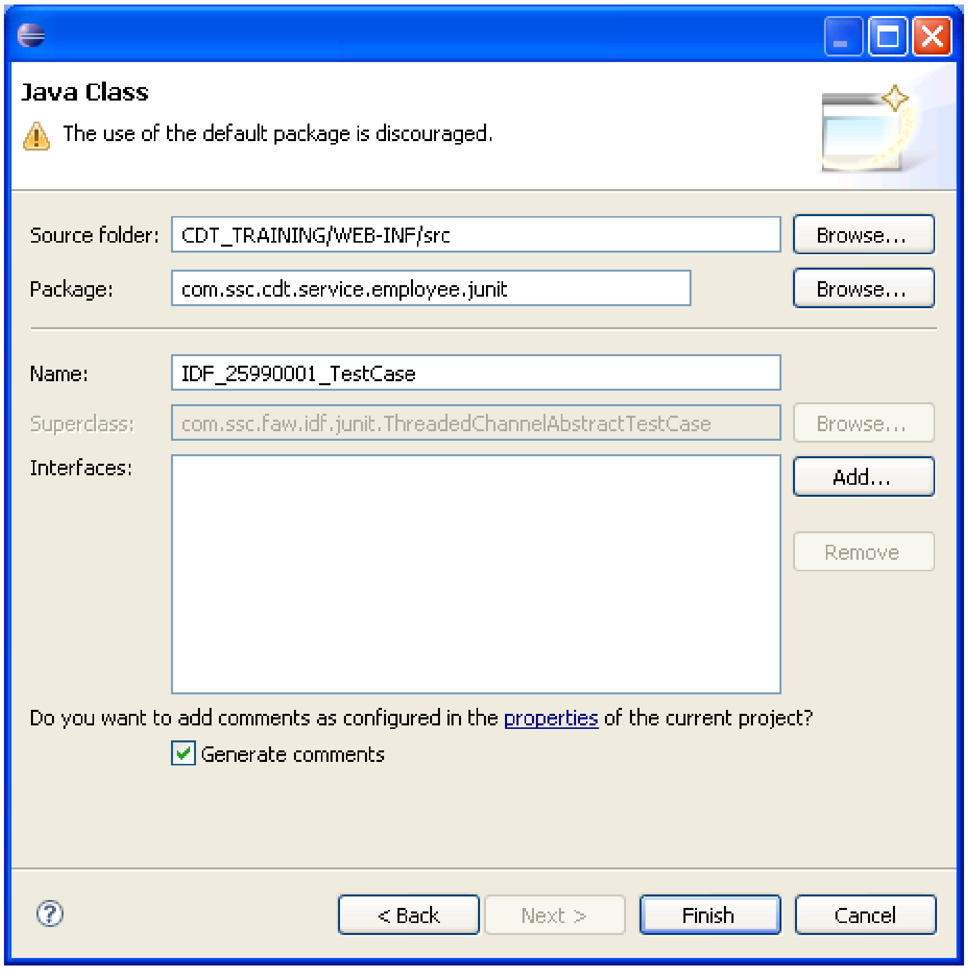
Create a new Java class.

Source folder:	CDT_TRAINING/WEB-INF/src
Package:	com.ssc.cdt.service.employee
Name:	IDF_25990001
Superclass:	java.lang.Object
Mandatory interface	com.ssc.faw.intref2.Tabular
Interfaces:	

Do you want to add comments? (Configure templates and default value here)

[✓] Generate comments

The CDT displays a similar screen for us to enter details of the JUnit test case. The name and package is already filled in; you could change them if desired, but for this exercise we'll leave the defaults. Select the ‘Generate Comments’ checkbox, then click ‘**Finish**’.


Java Class

The use of the default package is discouraged.

Source folder: CDT_TRAINING/WEB-INF/src

Package: com.ssc.cdt.service.employee.junit

Name: IDF_25990001_TestCase

Superclass: com.ssc.faw.idf.junit.ThreadedChannelAbstractTestCase

Interfaces:

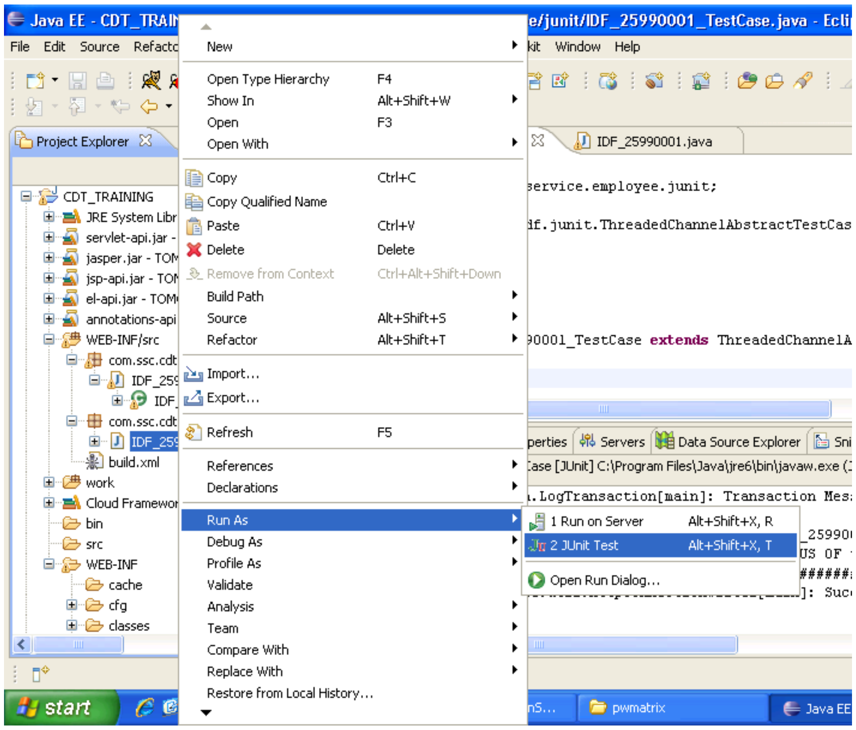
Do you want to add comments as configured in the properties of the current project?

✓ Generate comments

The CDT creates Java classes for the IDF (‘IDF_25990001.java’) and the JUnit test case (IDF_25990001_TestCase.java’), as well as the service descriptor, ‘WEB-INF/dat/IDF_25990001.xml’. You may wish to have a look at these files. At this point, the files are just starting points; we will add code to the Java files and entries to the service descriptor as the lab proceeds.

<u>Task: Execute a Failing JUnit Test</u>

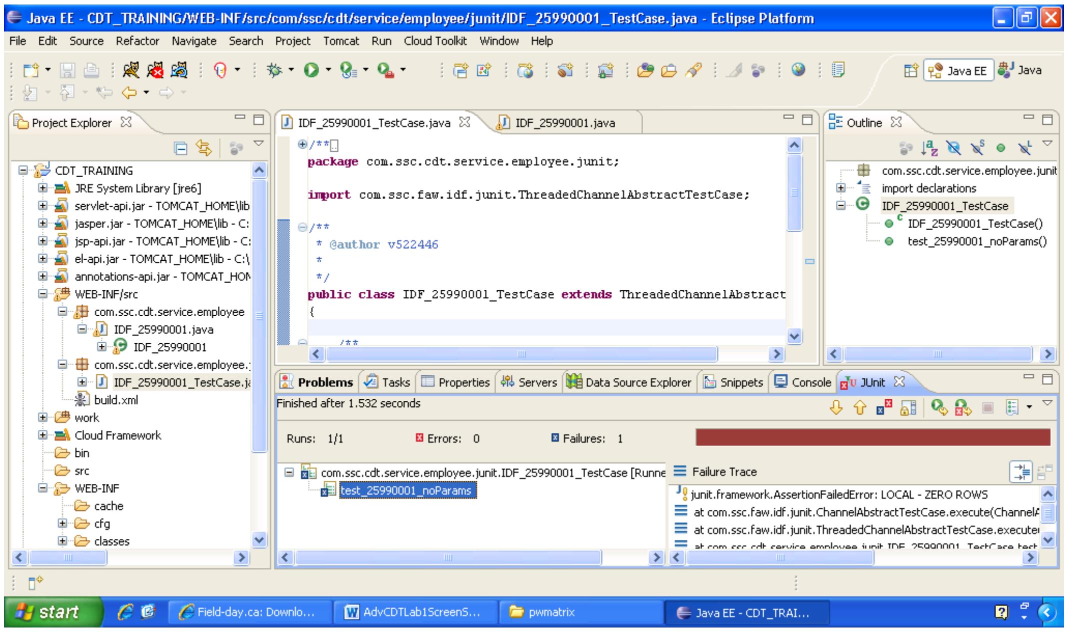
The CDT created an empty JUnit test case that extends from ThreadedChannelAbstractTestCase. Just to make sure that the test environment is setup properly, we’ll execute the JUnit test prior to adding any code. In the Project Explorer panel, right-click on the class ‘com.ssc.cdt.service.employee.junit.IDF_25990001_TestCase’ and then select ‘**Run As -> JUnit Test**’.



```
ervice.employee.junit;
df.junit.ThreadedChannelAbstractTestCas
0001_TestCase extends ThreadedChannelA
```

```
ase [JUnit] C:\Program Files\Java\jre6\bin\javaw.exe (
.LogTransaction[main]: Transaction Mes
_259900
US OF
######
]: Suc
```

The test should execute, and the JUnit TestRunner pane should show in the bottom-right-hand panel in Eclipse. Click on the ‘**JUnit**’ tab to bring it to the front if necessary.


Java EE - CDT_TRAINING/WEB-INF/src/com/ssc/cdt/service/employee/junit/IDF_25990001_TestCase.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Tomcat Run Cloud Toolkit Window Help
Java EE
Java
Project Explorer
IDF_25990001_TestCase.java
IDF_25990001.java
Outline
CDT_TRAINING
JRE System Library [jre6]
servlet-api.jar - TOMCAT_HOME\lib
jasper.jar - TOMCAT_HOME\lib - C:
jsp-api.jar - TOMCAT_HOME\lib - C:
el-api.jar - TOMCAT_HOME\lib - C:\
annotations-api.jar - TOMCAT_HOM
WEB-INF/src
com.ssc.cdt.service.employee
IDF_25990001.java
IDF_25990001
com.ssc.cdt.service.employee.
IDF_25990001_TestCase.j
build.xml
work
Cloud Framework
bin
src
WEB-INF
cache
cfg
classes
com.ssc.cdt.service.employee.junit
import declarations
IDF_25990001_TestCase
IDF_25990001_TestCase()
test_25990001_noParams()
Problems Tasks Properties Servers Data Source Explorer Snippets Console JUnit
Failure Trace
start
Field-day.ca: Downlo...
AdvCDTLab1ScreenS...
pwmatrix
Java EE - CDT_TRAI...

```
/**
package com.ssc.cdt.service.employee.junit;

import com.ssc.faw.idf.junit.ThreadedChannelAbstractTestCase;

/**
 * @author v522446
 *
 */
public class IDF_25990001_TestCase extends ThreadedChannelAbstract
{
    /**
```

```
Finished after 1.532 seconds

Runs:  1/1        Errors:  0        Failures:  1

com.ssc.cdt.service.employee.junit.IDF_25990001_TestCase [Runne
    test_25990001_noParams

junit.framework.AssertionFailedError: LOCAL - ZERO ROWS
at com.ssc.faw.idf.junit.ChannelAbstractTestCase.execute(ChannelA
at com.ssc.faw.idf.junit.ThreadedChannelAbstractTestCase.execute
at com.ssc.cdt.service.employee.junit.IDF_25990001_TestCase.test
```

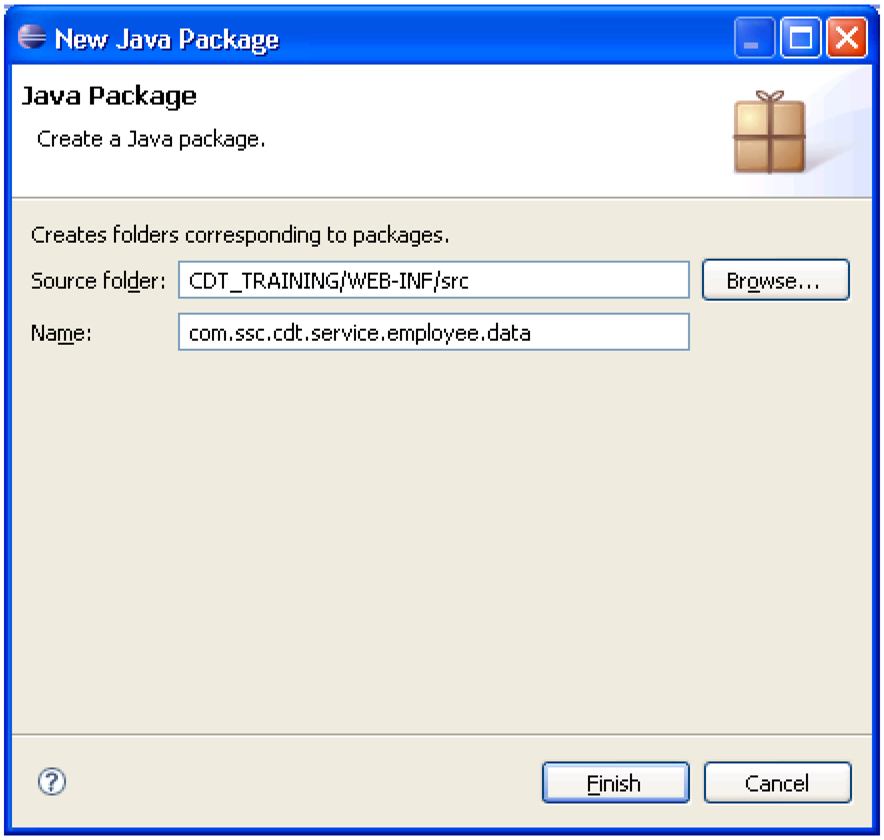
Note that the test shows up as failed (see the Red bar), and that the failure trace shows the reason for failure as “LOCAL-ZERO ROWS”. There is an assertion that built into the ‘ThreadedChannelAbstractTestCase’ that checks that the service returns at least one row.

<u>Task: Create a “dummy” Employee Data Store</u>

Recall that the service is intended to return employee data to the user interface. In order to simplify the task at the beginning, we will start with a “dummy” data store rather than attempting to create and connect to a database. This approach is common with component-oriented systems; once we verify that the service façade and the user interface works, we can approach the data storage system separately. For now, we will use an in-memory set of randomized employee data.

For convenience, the classes required to implement this dummy data store have been provided for you in the supporting files folder.

In the Project Explorer, right-click on the ‘**com.ssc.cdt.service.employee**’ package and select ‘**New -> Package…**’. In the dialog, change the name of the new package to ‘**com.ssc.cdt.service.employee.data**’.



Creates folders corresponding to packages.

Source folder: CDT_TRAINING/WEB-INF/src

Name: com.ssc.cdt.service.employee.data

In Windows Explorer, copy the two files ‘Employee.java’ and ‘EmployeeData.java’ from the lab files folder (recall that you can select multiple files by holding down the ctrl-key while clicking). Go back to Eclipse, right-click on the newly-created package and select ‘**Paste**’ to paste the two files into the new package.

The EmployeeData class will generate a Map object that maps employee ids to instances of the Employee class. Some of the employee fields are randomly generated, but, for testing entitlements, the employee with id ‘1’ will be in department “ITS” with salary ‘40000’.

Note: In order to simplify the code a little, we have used ‘double’ variables to represent the employee’s salaries. Obviously in real code, you would not use a double because of round-off error. The class ‘java.lang.BigDecimal’ would be more appropriate, but formatting and doing math with that class is a little more complicated.

Task: Begin Writing the Test Case

We’re going to use “Test-Driven Development” (TDD). We’ll write the test case for our functionality, and then write the service code to implement the functionality. Basically, the test will perform the following:

- With user set to “MANAGER”, load the list of employees and make sure we get 50 entries (recall that when running locally, we can tell the test case to use a local eSF proxy, which loads its entitlements from a local directory. Inside the test case, we can set the “current user” using the ‘setUser(…)’method).
- With user set to “CLERK”, load the list of employees and make sure that the list only contains employees that a “CLERK” would be authorized to see.
- With user set to “MANAGER”, load employee ‘1’ and check that all the fields are present.
- With user set to “CLERK”, load employee ‘1’ and ensure that the salary is not returned.
- Check that the delete functionality works.
- Check that the update functionality works.

Open the test case, ‘IDF_25990001_TestCase’ by double-clicking it in the Package Explorer.

The CDT created a test method called ‘test_25990001_noParams’ that shows the basic test code. We won’t use it directly, but we’ll take it as the starting point for our first test. Rename this test method to ‘testManagerLoadList’.

The service testing framework requires a few things to be setup, in order to simulate the eSF environment. Before the ‘setParams(…’ line, add the following lines (Note: the code sets the simulated user to “MANAGER” and sets up the eSF proxy to read entitlements from the local files inside the application’s ‘WEB-INF/esf’ directory. This process is explained further in the ‘Security Management’ chapter):

```
setUser(“MANAGER”);

// Must match your app code
System.setProperty(“OEC.APP”, “CDTTRN25990”);

// Must match environment for test.
```

```
System.setProperty("OEC.ENV", "DEV1");

// Use for local and DEV environments.
System.setProperty("ESF.useLocalFile", "true");
```

Have a look at the existing 'setParams(…' line. This line sets the request parameters that will be sent to the service. As you can see, it includes the 'request number', which is formed by the app code's request prefix and a 3-5 digit number of the service developer's choosing. Recall that when we created the service, we used '001' as the request number suffix, so the request number is '25990001'. The 'action' request parameter is used by the client to indicate what type of action (load/update/load list, etc) it is requesting. The IdfDataSet class has a number of constants to represent actions. We'll start by writing the test for the 'load-list' action. The load-list action happens to be '5' but we don't like 'magic' numbers in code, so we'll use the 'String.format(…' method to write out the constant from IdfDataSet.

Change the 'setParams(…' line to read as follows:

```
setParams(String.format("__request=25990001&__action=%d",
IdfDataSet.LOADLIST));
```

Press "ctrl-shift-O" to organize the imports and add the IdfDataSet import.

Add the following after the 'execute(…' line:

```
assertEquals(50, getRowCount());
```

This line checks to see if we had 50 rows returned as expected.

Task: Run the test case

Since we have the test case open, we can right-click anywhere in the window and select '**Run As -> JUnit Test**'.

Not surprisingly, the test fails. It is worth mentioning that it actually still fails in the 'ThreadedAbstractTestCase' assertion that checks to see if zero rows were returned, not on the assertion that we just added. We could shut off that assertion by adding a line that reads 'setIgnoreAsserts(true);' before the 'execute(…)' call, but in this case the "non-zero row count" assertion makes sense, because we expect our service to return at least one row. We will now go on to implement code that makes the test pass.

Task: Write code to satisfy the test case

Open the file 'IDF_25990001.java'

Add an instance variable as follows:

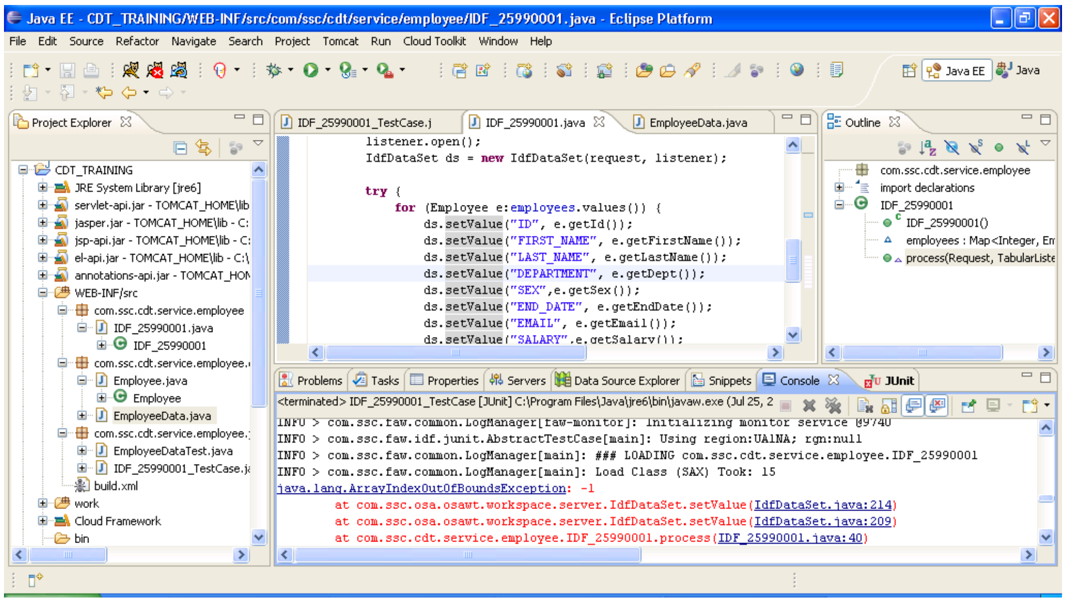
```
Map<Integer, Employee> employees=new EmployeeData();
```

Again, press ctrl-shift-O (letter '0') to add the required import statements for 'Map', 'Employee', and 'EmployeeData'. Make sure that you import the 'Employee' class from the 'com.ssc.cdt.service.employee.data' package.

Alter the contents of the 'process(…' method so it looks like this:

```
    public void process(Request request, TabularListener listener) throws
GenException
    {
      listener.open();
      IdfDataSet ds = new IdfDataSet(request, listener);

      try {
          for (Employee e:employees.values()) {
            ds.setValue("ID", e.getId());
            ds.setValue("FIRST_NAME", e.getFirstName());
            ds.setValue("LAST_NAME", e.getLastName());
            ds.setValue("DEPARTMENT", e.getDept());
            ds.setValue("SEX",e.getSex());
            ds.setValue("END_DATE", e.getEndDate());
            ds.setValue("EMAIL", e.getEmail());
            ds.setValue("SALARY",e.getSalary());
            ds.setValue("START_DATE", e.getStartDate());
            ds.outputRow();
          }
      } catch (Exception ex) {
          ex.printStackTrace();
          ds.setErrorMsg(IdfDataSet.ERROR_MSG, "Failed to load list on
request 25990001:" + ex.getMessage());
      } finally {
            listener.close();
      }
    }
```

Run the JUnit test case. Note that it still fails, but the console shows an exception



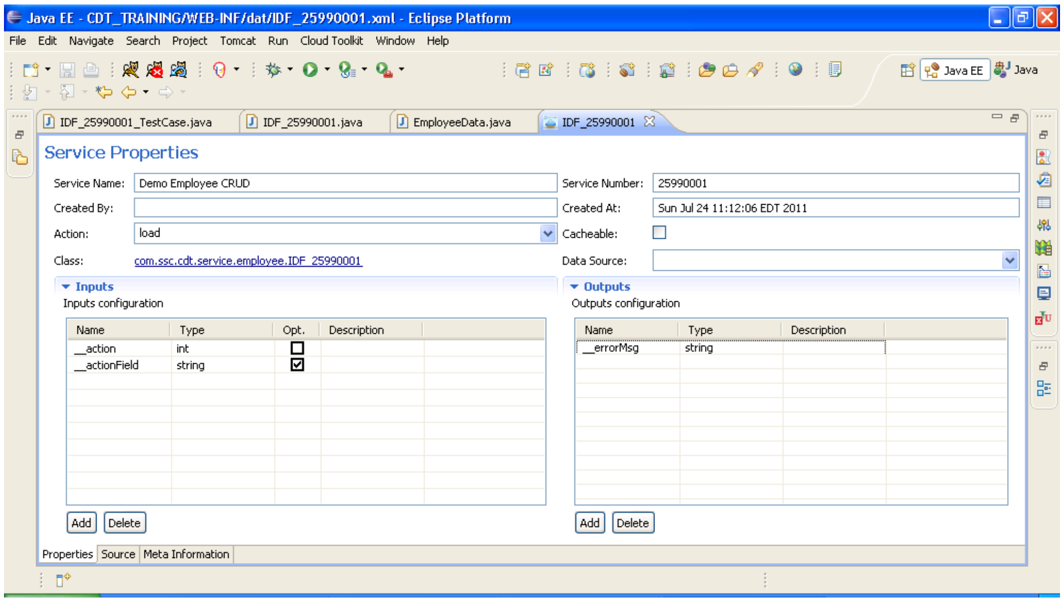
```
        listener.open();
        IdfDataSet ds = new IdfDataSet(request, listener);

        try {
            for (Employee e:employees.values()) {
                ds.setValue("ID", e.getId());
                ds.setValue("FIRST_NAME", e.getFirstName());
                ds.setValue("LAST_NAME", e.getLastName());
                ds.setValue("DEPARTMENT", e.getDept());
                ds.setValue("SEX",e.getSex());
                ds.setValue("END_DATE", e.getEndDate());
                ds.setValue("EMAIL", e.getEmail());
                ds.setValue("SALARY",e.getSalary());
```

```
<terminated> IDF_25990001_TestCase [JUnit] C:\Program Files\Java\jre6\bin\javaw.exe (Jul 25, 2
INFO > com.ssc.faw.common.LogManager[faw-monitor]: Initializing monitor service @9740
INFO > com.ssc.faw.idf.junit.AbstractTestCase[main]: Using region:UAINA; rgn:null
INFO > com.ssc.faw.common.LogManager[main]: ### LOADING com.ssc.cdt.service.employee.IDF_25990001
INFO > com.ssc.faw.common.LogManager[main]: Load Class (SAX) Took: 15
java.lang.ArrayIndexOutOfBoundsException: -1
	at com.ssc.osa.osawt.workspace.server.IdfDataSet.setValue(IdfDataSet.java:214)
	at com.ssc.osa.osawt.workspace.server.IdfDataSet.setValue(IdfDataSet.java:209)
	at com.ssc.cdt.service.employee.IDF_25990001.process(IDF_25990001.java:40)
```

The exception (you might have to scroll around in the console window to find it) is an ArrayIndexOutOfBoundsException. The root cause is that we're trying to write output, but we haven't declared in the service's metadata that the service writes any output. We need to add output fields in the service description file.

Open the file 'WEB-INF/dat/IDF_25990001.xml' by double-clicking on it in the Package Explorer.

Note that under the 'Outputs' heading, the only output defined is for an error message.



Service Properties

Service Name:	Demo Employee CRUD	Service Number:	25990001
Created By:		Created At:	Sun Jul 24 11:12:06 EDT 2011
Action:	load	Cacheable:	☐
Class:	com.ssc.cdt.service.employee.IDF_25990001	Data Source:	

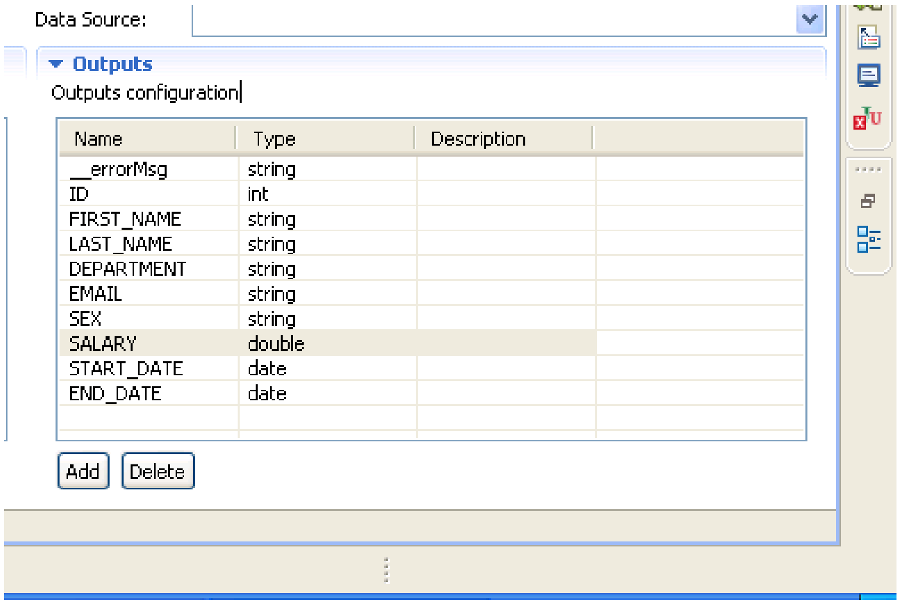
Inputs
Inputs configuration

Name	Type	Opt.	Description
__action	int	☐	
__actionField	string	☑	

Outputs
Outputs configuration

Name	Type	Description
__errorMsg	string	

Use the '**Add**' button and edit the name and type fields to add the following fields:



Name	Type	Description
__errorMsg	string	
ID	int	
FIRST_NAME	string	
LAST_NAME	string	
DEPARTMENT	string	
EMAIL	string	
SEX	string	
SALARY	double	
START_DATE	date	
END_DATE	date	

Save the file by clicking ctrl-S (or use the File menu).

Run the test case again. It should pass, showing a green bar.

Task: Write test code for the next fragment of functionality

We successfully returned a list of entries, but looking at the service code, you notice a few things:

- It only implements the load-list function. In fact it doesn't even check which action was requested.
- It doesn't check entitlements through eSF. This lack of security is clearly in violation of the 10 directives.

As always with TDD, we'll write the test code for the next piece of functionality, then implement the functionality.

Add a new method to the test suite class. Call this method 'testClerkOnlySeesITS()'.

Copy the body of the testManagerLoadList() method into the testClerkOnlySeesITS() method, then change 'setUser("MANAGER")' to 'setUser("CLERK")', so that it looks like:

```
    public void testClerkOnlySeesITS() throws Exception
    {
      setUser("CLERK");

      // Must match your app code.
      System.setProperty("OEC.APP", "CDTTRN25990");

      // Must match environment for test.
      System.setProperty("OEC.ENV", "DEV1");

      // Use for local and DEV environments.
      System.setProperty("ESF.useLocalFile", "true");

      setParams(String.format("__request=25990001&__action=%d",
IdfDataSet.LOADLIST));
      execute(LOCAL | ECHO);
      assertEquals(50, getRowCount());
    }
```

For this test, the assertion at the end of the test (which was copied over from the original test method) is no longer appropriate; we want to check that the clerk only sees rows that are appropriate to their entitlement. That entitlement will be recorded in eSF, and returned by a call to 'eSFProxy.getInstance().getDataValues(…)'. Note: A subsequent module will cover the eSFProxy API in more detail.

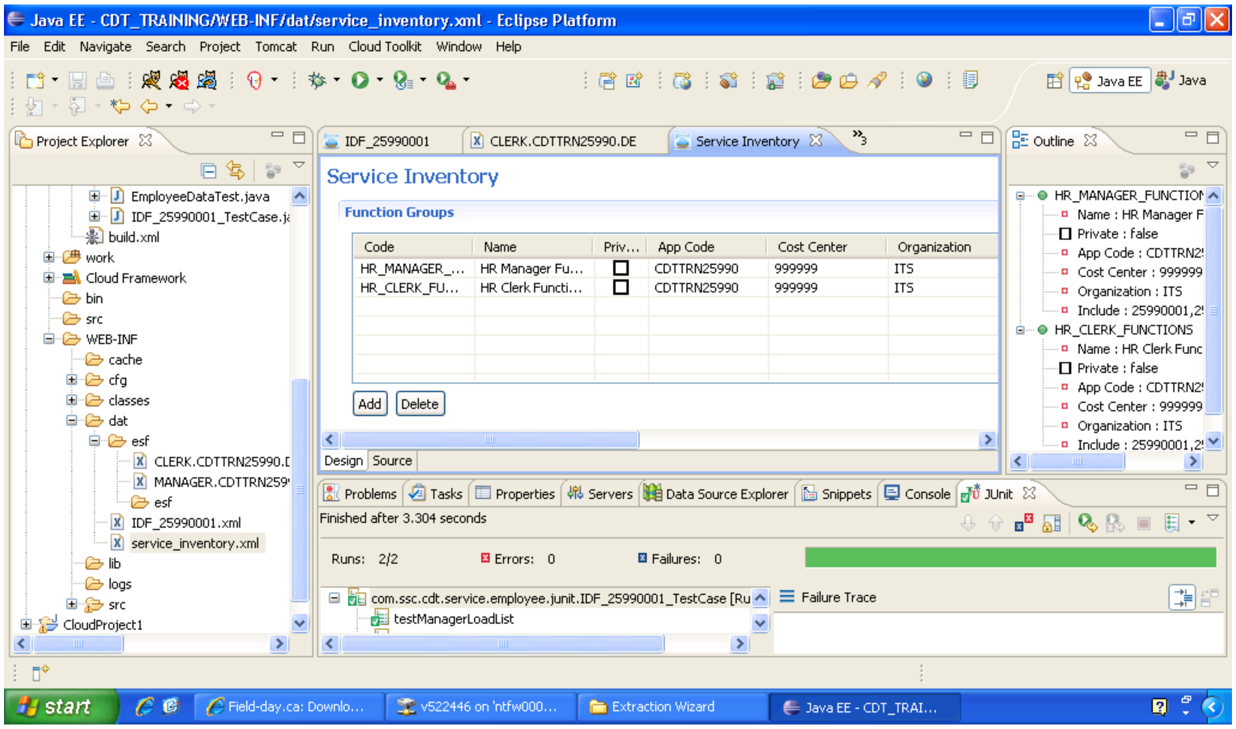
Replace the assertion with the following code:

```
      List<String>
authorizedDepts=Arrays.asList(eSFProxy.getInstance().getDataValues("HR_DEPT")
);
      for (int i=0; i < getRowCount(); i++) {
          String dept=getTestResultData(i, "DEPARTMENT").toString();
          assertTrue("CLERK isn't authorized for " + dept,
authorizedDepts.contains(dept));
      }
```

Organize imports, save the code, and run the test case.

Unsurprisingly, the test fails. The console shows an exception, "CANNOT FIND ESF FILE – clerk.CDTTRN25990.DEV1.xml". As described in the document http://owt.statestr.com/?page_id=4589, we need to define a set of roles for testing with the local eSF proxy. This definition is done with an xml file in your application's 'WEB-INF/dat/esf' directory. For your convenience, the two files we'll need (for the CLERK and MANAGER users) are provided in the 'eSF' folder inside the lab files folder. Copy this folder into 'WEB-INF/dat'.

At the same time, we should update the service inventory file, '/WEB-INF/data/service-inventory.xml'. The service inventory file declares the function groups, functions and app role templates that form the basis for eSF security. The cloud uses this metadata at deployment time to update the service registry. You should decide on the contents this file prior to developing your services. The CDT will have created an empty 'service_inventory.xml' when the project was created, but to save time, we have provided a completed file in the lab files folder. Copy 'service-inventory.xml' from the lab files folder into 'WEB-INF/dat', and then double-click on 'WEB-INF/dat/service-inventory.xml'.



Java EE - CDT_TRAINING/WEB-INF/dat/service_inventory.xml - Eclipse Platform

Service Inventory

Function Groups

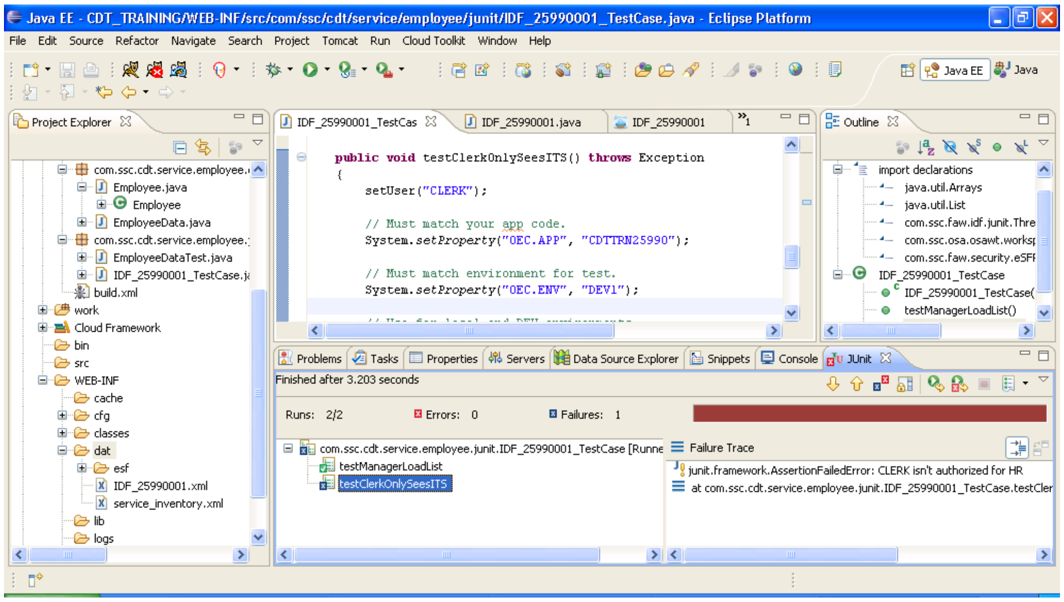
Code	Name	Priv...	App Code	Cost Center	Organization
HR_MANAGER_...	HR Manager Fu...	☐	CDTTRN25990	999999	ITS
HR_CLERK_FU...	HR Clerk Functi...	☐	CDTTRN25990	999999	ITS

```
HR_MANAGER_FUNCTIO
    Name : HR Manager F
    Private : false
    App Code : CDTTRN2
    Cost Center : 999999
    Organization : ITS
    Include : 25990001,2
HR_CLERK_FUNCTIONS
    Name : HR Clerk Func
    Private : false
    App Code : CDTTRN2
    Cost Center : 999999
    Organization : ITS
    Include : 25990001,2
```

```
Finished after 3.304 seconds

Runs:  2/2        Errors:  0        Failures:  0

com.ssc.cdt.service.employee.junit.IDF_25990001_TestCase [Ru
    testManagerLoadList
```

The exact format and meaning of this file is discussed in the document at http://owt.statestr.com/?page_id=4566.

Run the test suite again. The test still fails, but now gives a reason for the failure, "CLERK isn't authorized for HR".



Java EE - CDT_TRAINING/WEB-INF/src/com/ssc/cdt/service/employee/junit/IDF_25990001_TestCase.java - Eclipse Platform

```java
public void testClerkOnlySeesITS() throws Exception
{
    setUser("CLERK");

    // Must match your app code.
    System.setProperty("OEC.APP", "CDTTRN25990");

    // Must match environment for test.
    System.setProperty("OEC.ENV", "DEV1");
```

```
Finished after 3.203 seconds

Runs:  2/2        Errors:  0        Failures:  1

com.ssc.cdt.service.employee.junit.IDF_25990001_TestCase [Runne
    testManagerLoadList
    testClerkOnlySeesITS

Failure Trace
junit.framework.AssertionFailedError: CLERK isn't authorized for HR
at com.ssc.cdt.service.employee.junit.IDF_25990001_TestCase.testCler
```

Now we can go to the service code, 'IDF_25990001.java', and insert the entitlement checking functionality. Similar to the test suite code, the service needs to get the list of authorized departments, then only add the employee to the output if the employee is in one of the authorized department. Modify the 'process(...)' method so it looks like:

```
    public void process(Request request, TabularListener listener) throws
GenException
    {
      listener.open();
      IdfDataSet ds = new IdfDataSet(request, listener);

      try
      {
          // New!
          List<String> authorizedDepts =
Arrays.asList(eSFProxy.getInstance().getDataValues("HR_DEPT"));

          for (Employee e : employees.values())
          {
            // If-statement is new!
            if (authorizedDepts.contains(e.getDept()))
            {
                ds.setValue("ID", e.getId());
                ds.setValue("FIRST_NAME", e.getFirstName());
                ds.setValue("LAST_NAME", e.getLastName());
                ds.setValue("DEPARTMENT", e.getDept());
                ds.setValue("SEX", e.getSex());
                ds.setValue("END_DATE", e.getEndDate());
                ds.setValue("EMAIL", e.getEmail());
                ds.setValue("SALARY", e.getSalary());
                ds.setValue("START_DATE", e.getStartDate());
                ds.outputRow();
            }
          }
      }
      catch (Exception ex)
      {
          ex.printStackTrace();
          ds.setErrorMsg(IdfDataSet.ERROR_MSG, "Failed to load list on
request 25990001:" + ex.getMessage());
      }
      finally
      {
          listener.close();
      }
    }
```

Run the test again. It should now pass, indicating that we are correctly using the local eSF proxy to check the data entitlements (at least as far as making sure the user is allowed to see the record as a whole).

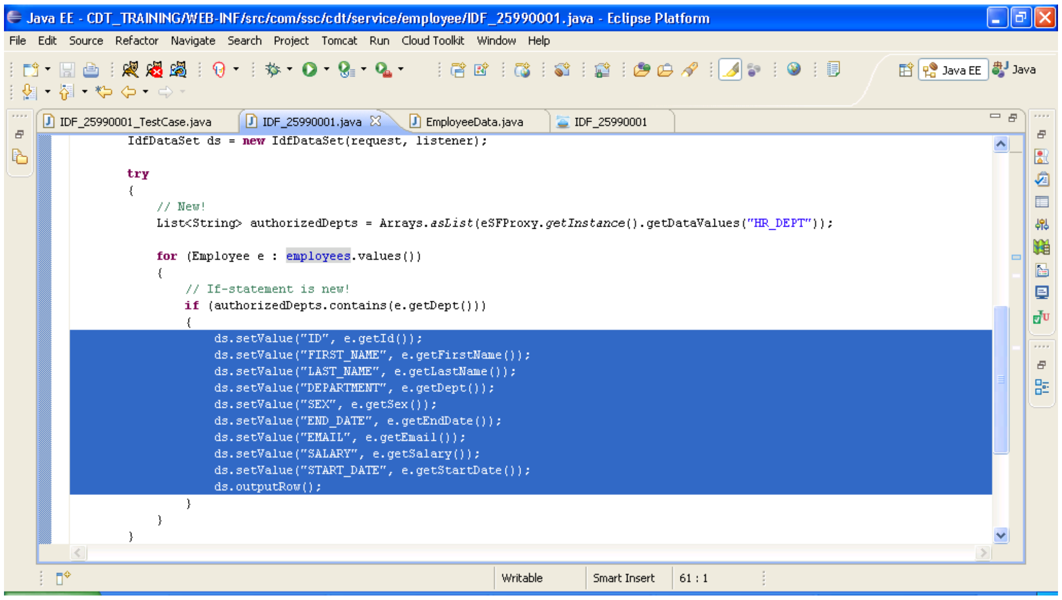
Task: Refactor the Service Class

So far, we have some working functionality, but the service class isn't very readable or extensible, and we haven't yet accounted for the different actions that the client could request (we've only handled the load-list action).

These deficiencies are not exactly errors; TDD says "If you haven't written a test, then the functionality isn't required yet", so we are actually proceeding correctly. It is fairly obvious that we'll need to add some code to handle different actions, but adding that code before the tests are written could be counter-productive. It is perfectly reasonable, however, to refactor the existing code to make it more readable and extensible.

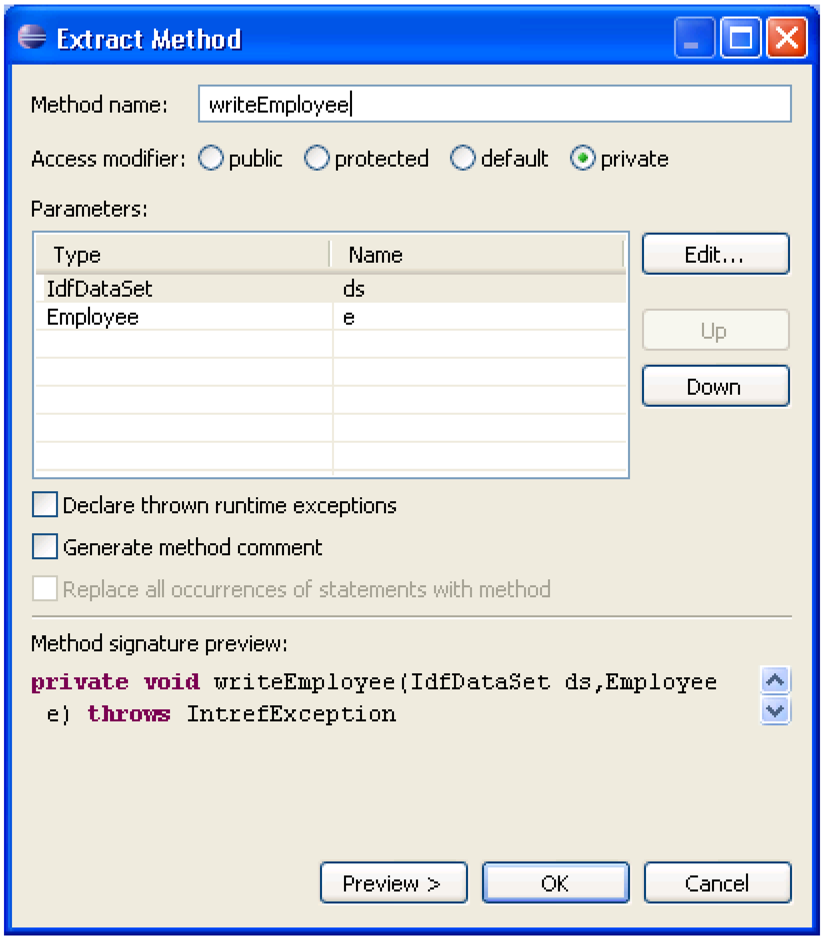
Open (or switch windows to) 'IDF_25990001.java'. Note that in the innermost if-statement of the 'process' method, we have a block of code that takes an Employee instance and writes its contents to the output dataset. This seems like code that we're likely to use elsewhere (for instance in loading an employee by id), so it makes sense to refactor it into a method.

Highlight the lines inside the if statement, from 'ds.setValue(...)' to 'ds.outputRow', inclusive. Right-click in the selected area, and select '**Refactor -> Extract Method**'.


Java EE - CDT_TRAINING/WEB-INF/src/com/ssc/cdt/service/employee/IDF_25990001.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Tomcat Run Cloud Toolkit Window Help
Java EE
Java
IDF_25990001_TestCase.java
IDF_25990001.java
EmployeeData.java
IDF_25990001
Writable
Smart Insert
61 : 1

```
        IdfDataSet ds = new IdfDataSet(request, listener);

        try
        {
            // New!
            List<String> authorizedDepts = Arrays.asList(eSFProxy.getInstance().getDataValues("HR_DEPT"));

            for (Employee e : employees.values())
            {
                // If-statement is new!
                if (authorizedDepts.contains(e.getDept()))
                {
                    ds.setValue("ID", e.getId());
                    ds.setValue("FIRST_NAME", e.getFirstName());
                    ds.setValue("LAST_NAME", e.getLastName());
                    ds.setValue("DEPARTMENT", e.getDept());
                    ds.setValue("SEX", e.getSex());
                    ds.setValue("END_DATE", e.getEndDate());
                    ds.setValue("EMAIL", e.getEmail());
                    ds.setValue("SALARY", e.getSalary());
                    ds.setValue("START_DATE", e.getStartDate());
                    ds.outputRow();
                }
            }
        }
```

In the dialog, enter 'writeEmployee' as the method name, then click '**OK**'.



Method name: writeEmployee

Access modifier: ( ) public ( ) protected ( ) default (● private)

Parameters:

Type	Name
IdfDataSet	ds
Employee	e

[ ] Declare thrown runtime exceptions
[ ] Generate method comment
[ ] Replace all occurrences of statements with method

Method signature preview:

```
private void writeEmployee(IdfDataSet ds,Employee
 e) throws IntrefException
```

Eclipse will extract that functionality into a separate method, leavind a more understandable 'process' method.

Similarly, we can note that all of the code inside the 'try' block is associated with loading the employee list. We can refactor that out to a method called 'writeEmployeeList()'. As above, select the contents of the try block, then right-click and select '**Refactor -> Extract Method**', and enter 'writeEmployeeList' as the method name. This again leaves a more understandable 'process' method.

```
    public void process(Request request, TabularListener listener) throws
GenException
    {
      listener.open();
      IdfDataSet ds = new IdfDataSet(request, listener);

      try
      {
          writeEmployeeList(ds);
      }
      catch (Exception ex)
      {
          ex.printStackTrace();
          ds.setErrorMsg(IdfDataSet.ERROR_MSG, "Failed to load list on
request 25990001:" + ex.getMessage());
      }
      finally
      {
          listener.close();
      }
    }
```

Just to be safe, save everything and run the test suite again. It should still pass.

The IDF is supposed to use the '__action' parameter to decide which CRUD function to execute, but as a first pass, we didn't test this value in our process() method. We are not going to add any further functionality in this lab, but if we were going to check the value, we would alter the process method to do something like this:

```
    public void process(Request request, TabularListener listener) throws
GenException
    {
      listener.open();
      IdfDataSet ds = new IdfDataSet(request, listener);
      int action = ds.Action();
      try
      {
          if (action == IdfDataSet.LOADLIST)
          {
            writeEmployeeList(ds);
          } else if (action==IdfDataSet.LOAD) {
            loadEmployee(ds);
          }
      }
      catch (Exception ex)
      {
          ex.printStackTrace();
          ds.setErrorMsg(IdfDataSet.ERROR_MSG, "Failed equest 25990001:" +
ex.getMessage());
      }
      finally
      {
          listener.close();
      }
    }
```

Note that we would also need to implement the loadEmployee method and add an optional 'ID' input parameter to the service metadata. Naturally, you would write a test for the new functionality first, then implement it.

Review

In this lab, we implemented the basic functionality of a service, after defining Junit tests for it. In reality, to fully implement the service, we would need to write the appropriate tests and functionality to implement update, add-new, data-changed and delete actions. However, our goal was to see the development process, particularly the Junit test mechanism. We also looked at the service metadata and one use of the eSF proxy to verify data entitlements.

Resources

Cloud Development Toolkit: http://cdt.statestr.com

Next Steps

Having completed this lab, you can go on to do “Unit 4 – Developing Interfaces”.