COSMOS

Developer's Guide

Version 1 Release 0

COSMOS

Developer's Guide

Version 1 Release 0

Note
Before using this information and the product it supports, read the information in .

First Edition (June 2008)

This edition applies to version 1, release 1, modification 0 of COSMOS and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Overview

Community Systems Management Open Source (COSMOS) is a generic, extensible, standards-based set of components for a tools platform that software developers can use to create specialized, differentiated, and interoperable offerings of tools for system management

Chapter 2. Creating a Data Manager

This topic lists at a high level the steps needed to create, run, and deploy a Data Manager

Learning a new framework or toolkit can be a daunting task. For most of us, such things are best accomplished by example. In this section, we will walk you through the process of building a sample Management Data Repository (MDR) using the COSMOS SDK. In the process, we hope to help you accomplish the following goals:

- Setting up your development environment for building Management Data Repositories (MDRs)
- Becoming familiar with the tools included with the COSMOS SDK
- Becoming familiar with the underlying framework and APIs that enable you to build MDRs for your existing data store
- Building a working example that shows techniques and best practices for building a CMDBf query service
- Understanding the concepts presented by both COSMOS and CMDBf

The sample MDR you will build contains data for a hypothetical school. This data includes students, teachers, courses, and their relationships. For the duration of this section, we will refer to this sample as the Student-Teacher sample.

To keep things simple, we have provided a sample repository consisting of XML files. While this does not represent a typical production data store, the techniques learned in using the COSMOS framework and tools can be applied to other data stores such as relational databases.

There are also several other example MDRs provided with COSMOS. These examples show how to implement query and registration services with different types of data stores, and with different optional features. They can be downloaded as part of the a package on the COSMOS web site.

Setting up the development environment

This topic explains how to prepare your environment for COSMOS.

COSMOS provides a set of tools to help you build an MDR. These tools are provided as extensions to the Eclipse SDK environment. They are included as part of the COSMOS SDK package.

To begin building the MDR in this Student-Teacher example, you must first install the COSMOS SDK into Eclipse. There are a number of prerequisites for the COSMOS SDK, including Eclipse Webtools and its prerequisites. Please refer to the COSMOS Installation Guide for specifics on how to install the COSMOS SDK and its prerequisites.

Now you can proceed to creating a data manager or MDR project.

Creating the student teacher project

This task describes how to create a project.

This task assumes you have installed the COMOS SDK and its prequisites and you have launched COSMOS.

Follow the instructions below to create an MDR project for the Student-Teacher sample.

1. Click **File > New > Project**
2. Expand **Web** and select **Dynamic Web Project**
3. Click Next and type in org.eclipse.cosmos.example.mdr as the project name.
4. In the `Target Runtime` field, select the Apache Tomcat server where you want to deploy the data manager project. If a Tomcat server is not already defined, click **New...** to select a server runtime environment.
5. After you define and select a target runtime for Apache Tomcat, the configuration ″Default Configuration for Apache Tomcat v5.5″ will be available for selection. Follow the steps below to configure the project:
   a. Select the configuration and click the **Modify...** button. The Project Facets dialog will open.
   b. Select the **Axis2 Web Services** facet category. Axis2 is the web services implementation used by COSMOS to deploy MDRs to a J2EE container.
   c. Under the Systems Management category, select **CMDBf query service** to implement an MDR with a query service. For the purpose of the Teacher-Student sample we will not need the registration service. Any MDR intending to provide a CMDBf registration service would also select the second option under System Management.
   d. Press **OK** to complete the configuration.
6. Click **Next** twice to skip to the MDR Configuration page
7. Enter **org.eclipse.cosmos.example.mdr** as the package name and **StudentTeacherSample** as the MDR name
8. Click **Finish** to create the project

Now, you can configure the project you just created.

Configuring the student teacher project

Put your short description here; used for first paragraph and abstract.

This task assumes you have created the student teacher project as described in the previous section.

Follow the instructions below to configure the student teacher project.

1. Right click the project and select **Properties**.
2. Select **Java Build Path** on the left panel.
3. Select the Libraries tab.
4. Click **Add Variable...,** select **ECLIPSE_HOME - ...** and click on **Extend...**
5. Expand **plugins** and select **org.eclipse.osgi_<<suffix>>**
6. Click **OK** to close the variables dialog.
7. Click **OK** to close the properties dialog.

The student teacher repository

This section will include code for implementing a simple XML-based repository. The content of the repository is contained in an XML file that is processed using a SAX parser. The implementation consists only of two Java classes: a file used to represent the entities of the XML content, and a SAX parser used to load in the XML content into memory.

Before implementing the Java classes, create a file called data.xml under org.eclipse.cosmos.example.mdr/src. The content of the file appears below:

```
<?xml version="1.0" encoding="UTF-8"?>
<school>
 <student>
  <identity firstName="Bob" lastName="Davidson" id="01"/>
 </student>

 <student>
  <identity firstName="Jane" lastName="Ryerson" id="02"/>
 </student>

 <student>
  <identity firstName="Mike" lastName="Lee" id="03"/>
 </student>

 <teacher>
  <identity firstName="Dawn" lastName="Johnson" id="staff01"/>
 </teacher>

 <teacher>
  <identity firstName="Heather" lastName="Reeba" id="staff02"/>
 </teacher>

 <class name="Economics" courseCode="ECM01">
  <students>
   <enrolledStudent idRef="01"/>
   <enrolledStudent idRef="03"/>
  </students>
  <teacher idRef="staff01"/>
 </class>

 <class name="Mathematics" courseCode="MAT01">
  <students>
   <enrolledStudent idRef="01"/>
   <enrolledStudent idRef="02"/>
  </students>
  <teacher idRef="staff02"/>
 </class>

 <class name="Physics" courseCode="PHY01">
  <students>
   <enrolledStudent idRef="02"/>
   <enrolledStudent idRef="03"/>
  </students>
  <teacher idRef="staff02"/>
 </class>
</school>
```

Notice the file contains three students, two teachers, and three classes. Each student and teacher element defines a unique ID using the 'id' attribute and each class indicates the teacher and the enrolled students using the id field.

Let's now create the classes that will load in the data above. Create the following two classes under src/org.eclipse.cosmos.example.mdr.handlers.

1. XMLRepository
2. SchoolXMLHandler

The first file provides an object representation of the entities in data.xml (i.e. student, teacher, and class). The content of XMLRepository appears below.

```
package org.eclipse.cosmos.example.mdr.handlers;

import java.io.IOException;
import java.io.StringWriter;

import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParserFactory;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServicesUtil;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.IXMLWritable;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.IItemConve
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.INodes;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.IRelations
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.QueryOutpu
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IGraphElement;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IGraphElementCollection;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IItem;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IRecord;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IRelationship;
import org.xml.sax.SAXException;

/**
 * This class represents a simple repository that is populated
 * by the data stored in a static XML file.  An implementation of
 * IDataProvider is used to provide a binding to an underlying
 * repository.
 */
public class XMLRepository
{
 /**
  * The MDR ID
  */
 public static final String MDR_ID = "org.eclipse.cosmos.samples.cmdbf.XMLRepository";

 /**
  * The XML file that stores the data
  */
 private static final String XML_DATA = "data.xml";

 /**
  * The namespace of the data
  */
 private static final String SCHOOL_NAMESPACE = "http://school";

 /**
  * The students of the school
  */
 public Student[] students;

 /**
  * The teachers of the school
  */
 public Teacher[] teachers;

 /**
  * The classes of the school
  */
 public ClassSession[] classes;


 /**
```

```
 * Constructor
 *
 * @throws ParserConfigurationException
 * @throws IOException
 * @throws SAXException
 */
public XMLRepository() throws ParserConfigurationException, SAXException, IOException
{
 // Read in the XML file and populate fields accordingly
 SchoolXMLHandler schoolXMLHandler = new SchoolXMLHandler();
 SAXParserFactory saxParserFactory = SAXParserFactory.newInstance();
 saxParserFactory.newSAXParser().parse(XMLRepository.class.getClassLoader().getResourceAsStream(X

 students = schoolXMLHandler.getStudents();
 teachers = schoolXMLHandler.getTeachers();
 classes = schoolXMLHandler.getSchoolClasses();
}


/**
 * Represents an identity, which is consisted
 * of a first name, last name, and an id.
 */
public static class Identity
{
 public String firstName;
 public String lastName;
 public String id;
}

/**
 * An abstract class representing a school
 * member
 */
public abstract static class SchoolMember implements IItemConvertible, IXMLWritable
{
 public Identity identity = new Identity();

 public IItem toItem(INodes parent)
 {
  IItem item = QueryOutputArtifactFactory.getInstance().createItem();
  item.addInstanceId(QueryOutputArtifactFactory.getInstance().createInstanceId(MDR_ID, identity.i
  IRecord record = QueryOutputArtifactFactory.getInstance().createRecord(item, identity.id);
  record.addNamespace("", SCHOOL_NAMESPACE);
  record.setValue(this);
  item.addRecord(record);
  return item;
 }

 private static String valueWithIndent(int indent, String value)
 {
  String tempValue = value;
  StringWriter tabsWriter = new StringWriter();
  tabsWriter.append('\n');
  CMDBfServicesUtil.addIndent(tabsWriter, indent);
  tempValue = tempValue.replace("\n", tabsWriter.toString());
  tempValue = tabsWriter.toString() + tempValue;
  return tempValue;
 }

 public void toXML(StringWriter writer, int indentLevel)
 {
  StringBuffer buffer = new StringBuffer();
  buffer.append ("<" + getElementName() + ">\n");
  buffer.append (" <identity firstName=\"" + identity.firstName + "\" lastName=\"" + identity.las
  buffer.append ("</" + getElementName() + ">");
  writer.write(SchoolMember.valueWithIndent(indentLevel, buffer.toString()) + "\n\n");
```

```
 }

 protected abstract String getElementName();
}

/**
 * Represents a student
 */
public static class Student extends SchoolMember
{
 @Override
 protected String getElementName()
 {
  return "student";
 }

}

/**
 * Represents a teacher of a school
 */
public static class Teacher extends SchoolMember
{
 @Override
 protected String getElementName()
 {
  return "teacher";
 }
}

/**
 * Represents a class with enrolled students
 * and a teacher
 */
public static class ClassSession implements IItemConvertible, IRelationshipConvertible, IXMLWritable
{
 public Student[] students;
 public Teacher teacher;
 public String name;
 public String courseCode;
 public IItem toItem(INodes parent)
 {
  IItem item = QueryOutputArtifactFactory.getInstance().createItem();
  item.addRecord(createRecord(item));
  return item;
 }


 public IRelationship toRelationship(IGraphElementCollection parent)
 {
  IRelationship relationship = QueryOutputArtifactFactory.getInstance().createRelationship();
  relationship.addInstanceId(QueryOutputArtifactFactory.getInstance().createInstanceId(MDR_ID, cours
  relationship.addRecord(createRecord(relationship));
  return relationship;
 }

 private IRecord createRecord(IGraphElement parent)
 {
  IRecord record = QueryOutputArtifactFactory.getInstance().createRecord(parent, courseCode);
  record.addNamespace("", SCHOOL_NAMESPACE);
  record.setValue(this);
  return record;
 }


 public void toXML(StringWriter writer, int indentLevel)
 {
```

```
   StringBuffer buffer = new StringBuffer();
   buffer.append("<class name=\"" + name + "\" courseCode=\"" + courseCode + "\">\n");
   buffer.append(" <students>\n");
   for (int i = 0; i < students.length; i++)
   {
    buffer.append("  <enrolledStudent idRef=\"" + students[i].identity.id + "\"/>\n");
   }
   buffer.append(" </students>\n");
   buffer.append(" <teacher idRef=\"" + teacher.identity.id + "\"/>\n");
   buffer.append("</class>");

   writer.write(SchoolMember.valueWithIndent(indentLevel, buffer.toString()) + "\n\n");
  }
 }
}
```

The second file is a SAX handler required for loading data.xml into memory. The content of SchoolXMLHandler appears below.

```
package org.eclipse.cosmos.example.mdr.handlers;

import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;

import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.ClassSession;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Identity;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Student;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Teacher;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

/**
 * The XML handler for parsing the data that represents
 * a school with students, teacher, and classes
 */
public class SchoolXMLHandler extends DefaultHandler
{
 private Student currentStudent;
 private Teacher currentTeacher;
 private ClassSession currentClass;
 private List<Student> enrolledStudents;

 private Map<String, Student> students;
 private Map<String, Teacher> teachers;
 private List<ClassSession> classes;

 public SchoolXMLHandler()
 {
  students = new Hashtable<String, Student>();
  teachers = new Hashtable<String, Teacher>();
  classes = new ArrayList<ClassSession>();
  enrolledStudents = new ArrayList<Student>();
 }

 @Override
 public void startElement(String uri, String localName, String name, Attributes attributes) throws
 {
  if ("student".equals(name))
  {
   currentStudent = new Student();
  }
  else if ("teacher".equals(name) && currentClass == null)
  {
   currentTeacher = new Teacher();
```

```
 }
 else if ("class".equals(name))
 {
  currentClass = new ClassSession();
  currentClass.name = attributes.getValue("name");
  currentClass.courseCode = attributes.getValue("courseCode");
 }
 else if ("identity".equals(name))
 {
  Identity identity = currentStudent != null ?
    currentStudent.identity : currentTeacher.identity;

  identity.firstName = attributes.getValue("firstName");
  identity.lastName = attributes.getValue("lastName");
  identity.id = attributes.getValue("id");
 }
 else if ("enrolledStudent".equals(name))
 {
  enrolledStudents.add(students.get(attributes.getValue("idRef")));
 }
 else if ("teacher".equals(name))
 {
  currentClass.teacher = teachers.get(attributes.getValue("idRef"));
 }

}

@Override
public void endElement(String uri, String localName, String name) throws SAXException
{
 if ("student".equals(name))
 {
  students.put(currentStudent.identity.id, currentStudent);
  currentStudent = null;
 }
 else if ("teacher".equals(name) && currentClass == null)
 {
  teachers.put(currentTeacher.identity.id, currentTeacher);
  currentTeacher = null;
 }
 else if ("class".equals(name))
 {
  currentClass.students = enrolledStudents.toArray(new Student[enrolledStudents.size()]);
  classes.add (currentClass);
  currentClass = null;
  enrolledStudents.clear();
 }
}

public Student[] getStudents()
{
 return students.values().toArray(new Student[students.size()]);
}

public Teacher[] getTeachers()
{
 return teachers.values().toArray(new Teacher[teachers.size()]);
}

public ClassSession[] getSchoolClasses()
{
 return classes.toArray(new ClassSession[classes.size()]);
}
}
```

This concludes implementing the simple XML repository. The next section will describe how the repository can be extended to provide CMDBf query support.

Providing CMDBf query support

Put your short description here; used for first paragraph and abstract.

As described in the previous section, the CMDBf specification defines a query and a registration service. COSMOS provides a set of reusable, extensible classes and interfaces that can be utilized in implementing such services. This section will cover how COSMOS APIs can be used in providing a CMDBf query implementation for the Student-Teacher example.

Architecture

The process of handling a CMDBf query involves traversing a structure representing the request, processing each individual part, and generating a structure representing the result. COSMOS has already defined structures representing the request and response. There is also generic code in place for traversing the request and invoking handlers for individual part of the CMDBf request. A consumer needs to only provide implementation of such handlers as part of implementing the query service.

A handler is required to process each CMDBf query constraint that an implementation supports. In addition to constraint handlers, the framework also requires two handlers for item templates and relationship templates. Here's the order in which the handlers are invoked.

1. Item/Relationship template handler
2. Instance id constraint handler
3. Record type constraint handler
4. Property value constraint handler

OR

1. Item/Relationship template handler
2. XPath constraint handler

It is the responsibility of the handler factory class to create instances of each handler above. The MDR toolkit automatically generates a handler factory. See src/org.eclipse.cosmos.example.mdr.handlers.QueryHandlerFactory.java for the class generated.

Implementing query handlers

This sample will provide four handlers in total:

- An item template handler
- An instance id constraint handler
- A record type constraint handler
- A relationship handler

Create the following classes for each handler under src/ org.eclipse.cosmos.example.mdr.handlers.

1. ItemTemplateHandler
2. ItemInstanceIdHandler
3. ItemRecordTypeHandler
4. RelationshipTemplateHandler

In addition to the four classes above, create an interface called "ICMDBfSampleConstants" under the same package to contain constants commonly used between handlers. The content of the interface, ICMDBfSampleConstants, appears below.

```
package org.eclipse.cosmos.example.mdr.handlers;

/**
 * Constants used by this CMDBf query sample
 */
public interface ICMDBfSampleConstants
{
 /**
  * The key for the data provider that will be
  * stored in the initialization data of the constraint
  * handlers
  */
 public static final String DATA_PROVIDER = "org.eclipse.cosmos.samples.cmdbf.services.query.ICMDBfSa
}
```

ItemTemplateHandler is there to simply process item templates that do not contain any constraints. The content of the class appears below:

```
package org.eclipse.cosmos.example.mdr.handlers;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServiceException;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.impl.AbstractItemTemplateHandle
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.IItemConve
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.INodes;

/**
 * An item template handler is invoked prior to invoking any constraint handler
 * for item templates.
 */
public class ItemTemplateHandler extends AbstractItemTemplateHandler
{

 /**
  * This method is only invoked if there are no constraints included in an
  * item template.
  *
  * @param nodes The nodes element to append items to
  */
 protected void appendAllItems(INodes nodes) throws CMDBfServiceException
 {
  XMLRepository repo = (XMLRepository)getValue(ICMDBfSampleConstants.DATA_PROVIDER);
  addItems(nodes, repo.classes);
  addItems(nodes, repo.students);
  addItems(nodes, repo.teachers);
 }


 /**
  * A convenient method used to add individual nodes to the
  * INodes instance passed in.
  *
  * @param nodes Container for individual nodes
  * @param itemConvertibles Elements that are convertible to an Item
  */
 private void addItems(INodes nodes, IItemConvertible[] itemConvertibles)
 {
  for (int i = 0; i < itemConvertibles.length; i++)
  {
   nodes.addItem(itemConvertibles[i].toItem(nodes));
  }
 }
}
```

The instance id constraint is used to locate items based on the id attribute of the student/teacher element or the course code defined for a class element. The implementation simply walks through students, teachers, and classes to locate items that match the instance id constraint. The content of ItemInstanceIdHandler appears below:

```
package org.eclipse.cosmos.example.mdr.handlers;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServiceException;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.impl.AbstractItemConstraintH
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.input.artifacts.IConstrain
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.input.artifacts.IInstanceI
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.INodes;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.QueryOu
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IInstanceId;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.SchoolMember;

/**
 * This is the handler class for instance id constraints included
 * in item templates.  Adopters can either provide a direct implementation of
 * IItemConstraintHandler or extend AbstractItemConstraintHandler
 */
public class ItemInstanceIdHandler extends AbstractItemConstraintHandler
{
 @Override
 protected INodes handle(INodes context, IConstraint constraint) throws CMDBfServiceException
 {
  INodes result = QueryOutputArtifactFactory.getInstance().createNodes(context.getId());
  IInstanceIdConstraint instanceIdConstraint = (IInstanceIdConstraint)constraint;
  IInstanceId[] instanceIds = instanceIdConstraint.getInstanceIds();
  for (int i = 0; i < instanceIds.length; i++) {
   if (!XMLRepository.MDR_ID.equals(instanceIds[i].getMdrId()
     .toString())) {
    continue;
   }
   String localId = instanceIds[i].getLocalId().toString();
   XMLRepository repo = (XMLRepository) getValue(ICMDBfSampleConstants.DATA_PROVIDER);
   // Traverse the students
   traverseSchoolMembers(result, localId, repo.students);
   // Traverse the teachers
   traverseSchoolMembers(result, localId, repo.teachers);
   // Traverse the classes
   for (int j = 0; j < repo.classes.length; j++)
   {
    if (localId.equals(repo.classes[j].courseCode))
    {
     result.addItem(repo.classes[j]);
    }
   }
  }
  return result;
 }

 private void traverseSchoolMembers (INodes result, String localId, SchoolMember[] members)
 {
  for (int i = 0; i < members.length; i++)
  {
   if (localId.equals(members[i].identity.id))
   {
    result.addItem(members[i]);
   }
  }
 }
}
```

Similarly, ItemRecordTypeHandler is used to process record type constraints. The handler supports three record types: student, teacher, and class. The content of ItemRecordTypeHandler appears below.

```
package org.eclipse.cosmos.example.mdr.handlers;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServiceException;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.IItemConstraintHandler;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.impl.AbstractItemConstraintHand
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.input.artifacts.IConstraint;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.input.artifacts.IRecordType;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.INodes;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.QueryOutpu
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IGraphElement;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IItem;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IRecord;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.ClassSession;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.SchoolMember;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Student;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Teacher;

/**
 * This is the handler for the record type constraint specified in
 * and item template.  Adopters can either extend AbstractItemConstraintHandler
 * or provide a direct implementation of {@link IItemConstraintHandler}
 */
public class ItemRecordTypeHandler extends AbstractItemConstraintHandler
{
 /**
  * Record type representing student
  */
 private static final String RECORD_TYPE_STUDENT = "student";

 /**
  * Record type representing teacher
  */
 private static final String RECORD_TYPE_TEACHER = "teacher";

 /**
  * Record type representing class
  */
 private static final String RECORD_TYPE_CLASS = "class";


 @Override
 protected INodes handle(INodes context, IConstraint constraint) throws CMDBfServiceException
 {
  IRecordType recordType = (IRecordType)constraint;
  String localName = recordType.getLocalName();
  XMLRepository repo = (XMLRepository)getValue(ICMDBfSampleConstants.DATA_PROVIDER);
  INodes result = QueryOutputArtifactFactory.getInstance().createNodes(context.getId());

  // If the record type constraint includes all items as its context
  if (context.isStartingContext())
  {
   if (RECORD_TYPE_STUDENT.equals(localName))
   {
    addSchoolMembers(result, repo.students);
   }
   else if (RECORD_TYPE_TEACHER.equals(localName))
   {
    addSchoolMembers(result, repo.teachers);
   }
   else if (RECORD_TYPE_CLASS.equals(localName))
   {
    for (int i = 0; i < repo.classes.length; i++)
    {
```

```
     result.addItem(repo.classes[i]);
    }
   }

   return result;
  }

  IGraphElement[] elements = context.getElements();
  for (int i = 0; i < elements.length; i++)
  {
   IRecord[] records = elements[i].getRecords();
   for (int j = 0; j < records.length; j++)
   {
    if (RECORD_TYPE_STUDENT.equals(localName) && records[j].getValue() instanceof Student)
    {
     result.addItem((IItem)elements[i]);
     continue;
    }
    else if (RECORD_TYPE_TEACHER.equals(localName) && records[j].getValue() instanceof Teacher)
    {
     result.addItem((IItem)elements[i]);
     continue;
    }
    else if (RECORD_TYPE_CLASS.equals(localName) && records[j].getValue() instanceof ClassSession)
    {
     result.addItem((IItem)elements[i]);
     continue;
    }
   }
  }

  return result;
 }

 private void addSchoolMembers(INodes result, SchoolMember[] members)
 {
  for (int i = 0; i < members.length; i++)
  {
   result.addItem(members[i]);
  }
 }
}
```

Finally, RelationshipTemplateHandler is used to process one type of relationship: i.e. the relationship between a teacher and a student. Given that there is only one relationship type defined, the input is processed in a relationship template handler. Typically this would be defined in a record type constraint for relationship templates. The class always assumes that the source is a teacher and the target is a student. The relationship is always teacher X teaches student Y. The code for RelationshipTemplateHandler appears below:

```
package org.eclipse.cosmos.example.mdr.handlers;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServiceException;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.IRelationshipTemplateHandler
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.impl.AbstractQueryHandler;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.input.artifacts.IRelations
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.IEdges;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.IQueryR
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.QueryOu
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IItem;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IRecord;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.transform.artifacts.IRelationship;
```

```
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.ClassSession;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Student;
import org.eclipse.cosmos.example.mdr.handlers.XMLRepository.Teacher;

/**
 * Represents the relationship record type handler.  There is only one type
 * of relationship that this handler accepts: "teaches".  The "teaches"
 * relationship can only exist between a teacher A and a student B (i.e.
 * A teaches B).
 * Adopters can either extend AbstractRelationshipConstraintHandler or provide
 * a direct implementation of IRelationshipConstraintHandler
 */
public class RelationshipTemplateHandler extends AbstractQueryHandler implements IRelationshipTempla
{
 public IEdges execute(IQueryResult context, IRelationshipTemplate relationshipTemplate, IItem source
 {
  IEdges results = QueryOutputArtifactFactory.getInstance().createEdges(relationshipTemplate);
  IRecord[] sourceRecords = source.getRecords();
  IRecord[] targetRecords = target.getRecords();

  // Only one record is expected
  if (sourceRecords.length != 1 || targetRecords.length != 1)
  {
   return results;
  }

  if (sourceRecords[0].getValue() instanceof Teacher && targetRecords[0].getValue() instanceof Studen
  {
   XMLRepository repo = (XMLRepository)getValue(ICMDBfSampleConstants.DATA_PROVIDER);
   String teacherId = ((Teacher)sourceRecords[0].getValue()).identity.id;
   String studentId = ((Student)targetRecords[0].getValue()).identity.id;

   ClassSession[] classSessions = findClass (repo, teacherId, studentId);
   for (int i = 0; i < classSessions.length; i++)
   {
    IRelationship relationship = classSessions[i].toRelationship(results);
    relationship.setSourceId(QueryOutputArtifactFactory.getInstance().createInstanceId(XMLRepository
    relationship.setTargetId(QueryOutputArtifactFactory.getInstance().createInstanceId(XMLRepository
    results.addRelationship(relationship);
   }
  }

  return results;
 }

 private ClassSession[] findClass(XMLRepository repo, String teacherId, String studentId)
 {
  List<ClassSession> discoveredClasses = new ArrayList<ClassSession>();
  ClassSession[] classes = repo.classes;
  for (int i = 0; i < classes.length; i++)
  {
   if (teacherId.equals(classes[i].teacher.identity.id))
   {
    Student[] students = classes[i].students;
    for (int j = 0; j < students.length; j++)
    {
     if (studentId.equals(students[j].identity.id))
     {
      discoveredClasses.add(classes[i]);
     }
    }
   }
  }

  return discoveredClasses.toArray(new ClassSession[discoveredClasses.size()]);
 }
}
```

This completes the implementation for all supported query handlers. What remains to be done is the factory class to instantiate instances of each handler. Open the generated class org.eclipse.cosmos.example.mdr.handlers.QueryHandlerFactory and add the following content.

```
package org.eclipse.cosmos.example.mdr.handlers;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServiceException;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.IItemConstraintHandler;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.IItemTemplateHandler;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.IRelationshipTemplateHandler
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.impl.AbstractQueryHandlerFac

/**
 * This factory class is used to create handlers for the selectors
 * that are applicable to the XML repository MDR.  Adopters can either
 * extend AbstractSelectorHandlerFactory or provide a direct
 * implementation of ISelectorHandlerFactory
 */
public class QueryHandlerFactory extends AbstractQueryHandlerFactory
{
 private static QueryHandlerFactory instance;

 /**
  * Make the constructor invisible
  */
 private QueryHandlerFactory()
 {
 }

 public static QueryHandlerFactory getInstance()
 {
  if (instance == null)
  {
   instance = new QueryHandlerFactory();
  }
  return instance;
 }

 @Override
 protected IItemConstraintHandler createItemInstanceHandler()
 {
  return new ItemInstanceIdHandler();
 }

 @Override
 protected IItemConstraintHandler createItemRecordHandler()
 {
  return new ItemRecordTypeHandler();
 }

 @Override
 protected IItemTemplateHandler createItemHandler() throws CMDBfServiceException
 {
  return new ItemTemplateHandler();
 }

 @Override
 protected IRelationshipTemplateHandler createRelationshipHandler() throws CMDBfServiceException
 {
  return new RelationshipTemplateHandler();
 }
}
```

Testing the query service

The easiest way of testing a query service for an MDR is through JUnit tests. This section demonstrates how the query service can be tested independently from other COSMOS components. The code included in this section can easily be incorporated in a JUnit to automatically test the results against an expected outcome. Follow the instructions below to test the query service.

1. Create a query request under `org.eclipse.cosmos.example.mdr/src` called **teaches-relationship.xml**
2. Create a new class called **QueryLauncher** under the same package as the handlers (i.e. `org.eclipse.cosmos.example.mdr.handlers`).

Use the content below for the query request. This file queries for all students that are taught by teacher staff01.

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

<!-- This query selects all students taught by the teacher with -->
<!-- the id: "staff01" -->
<s:query xmlns:s="http://cmdbf.org/schema/1-0-0/datamodel">
 <s:itemTemplate id="teacher">
  <s:instanceIdConstraint>
   <s:instanceId>
    <s:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</s:mdrId>
    <s:localId>staff01</s:localId>
   </s:instanceId>
  </s:instanceIdConstraint>
 </s:itemTemplate>

 <s:itemTemplate id="students">
  <s:recordConstraint>
   <s:recordType namespace="" localName="student"/>
  </s:recordConstraint>
 </s:itemTemplate>

 <s:relationshipTemplate id="reference">
  <s:sourceTemplate ref="teacher"/>
  <s:targetTemplate ref="students"/>
 </s:relationshipTemplate>
</s:query
```

Here's the content for QueryLauncher. The main method simply uses the content of teaches-relationship.xml to submit a CMDBf query:

```
package org.eclipse.cosmos.example.mdr.handlers;


import java.io.IOException;
import java.io.InputStream;
import java.net.URISyntaxException;
import java.util.Hashtable;
import java.util.Map;

import javax.xml.parsers.ParserConfigurationException;

import org.eclipse.cosmos.dc.provisional.cmdbf.services.common.CMDBfServiceException;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.service.impl.CMDBfQueryOperation;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.QueryOutputTransformer;
import org.eclipse.cosmos.dc.provisional.cmdbf.services.query.transform.response.artifacts.IQueryRes
import org.xml.sax.SAXException;

/**
 * The main class used to run the CMDBf queries.  A set of query
 * files exist under the source directory.  Adjust the queryFile field to
 * run the correct CMDBf query.
```

```
 */
public class QueryLauncher
{
 private static final String queryFile = "teaches-relationship.xml";


 public static void main(String[] args) throws ParserConfigurationException, SAXException, IOExcep
 {
  // Run the query
  InputStream query = QueryLauncher.class.getClassLoader().getResourceAsStream(queryFile);
  CMDBfQueryOperation queryOperation = new CMDBfQueryOperation(QueryHandlerFactory.getInstance(),
  Map<String, Object> init = new Hashtable<String, Object>();
  init.put(ICMDBfSampleConstants.DATA_PROVIDER, new XMLRepository());
  queryOperation.initialize(init);
  IQueryResult result = queryOperation.execute(query);

  // Transform and output the query
  InputStream resultStream = QueryOutputTransformer.transform(result);
  byte[] buffer = new byte[1024];
  while (resultStream.available() > 0)
  {
   int chars = resultStream.read(buffer);
   System.out.print(new String(buffer, 0, chars));
  }
 }

}
```

Once executed, the output of the query should be printed out to the console. The output will look something similar to the following code.

```
<cmdbf:queryResult xmlns:cmdbf="http://cmdbf.org/schema/1-0-0/datamodel">
 <cmdbf:nodes templateId="teacher">
  <cmdbf:item>
   <cmdbf:record xmlns="http://school">

    <teacher>
     <identity firstName="Dawn" lastName="Johnson" id="staff01"/>
    </teacher>

    <cmdbf:recordMetadata>
     <cmdbf:recordId>staff01</cmdbf:recordId>
    </cmdbf:recordMetadata>
   </cmdbf:record>
   <cmdbf:instanceId>
    <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
    <cmdbf:localId>staff01</cmdbf:localId>
   </cmdbf:instanceId>
  </cmdbf:item>
 </cmdbf:nodes>
 <cmdbf:nodes templateId="students">
  <cmdbf:item>
   <cmdbf:record xmlns="http://school">

    <student>
     <identity firstName="Mike" lastName="Lee" id="03"/>
    </student>

    <cmdbf:recordMetadata>
     <cmdbf:recordId>03</cmdbf:recordId>
    </cmdbf:recordMetadata>
   </cmdbf:record>
   <cmdbf:instanceId>
    <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
    <cmdbf:localId>03</cmdbf:localId>
   </cmdbf:instanceId>
  </cmdbf:item>
```

```
 <cmdbf:item>
  <cmdbf:record xmlns="http://school">

   <student>
    <identity firstName="Bob" lastName="Davidson" id="01"/>
   </student>

   <cmdbf:recordMetadata>
    <cmdbf:recordId>01</cmdbf:recordId>
   </cmdbf:recordMetadata>
  </cmdbf:record>
  <cmdbf:instanceId>
   <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
   <cmdbf:localId>01</cmdbf:localId>
  </cmdbf:instanceId>
 </cmdbf:item>
</cmdbf:nodes>
<cmdbf:edges templateId="reference">
 <cmdbf:relationship>
  <cmdbf:source>
   <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
   <cmdbf:localId>staff01</cmdbf:localId>
  </cmdbf:source>
  <cmdbf:target>
   <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
   <cmdbf:localId>03</cmdbf:localId>
  </cmdbf:target>
  <cmdbf:record xmlns="http://school">

   <class name="Economics" courseCode="ECM01">
    <students>
     <enrolledStudent idRef="01"/>
     <enrolledStudent idRef="03"/>
    </students>
    <teacher idRef="staff01"/>
   </class>

   <cmdbf:recordMetadata>
    <cmdbf:recordId>ECM01</cmdbf:recordId>
   </cmdbf:recordMetadata>
  </cmdbf:record>
  <cmdbf:instanceId>
   <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
   <cmdbf:localId>ECM01</cmdbf:localId>
  </cmdbf:instanceId>
 </cmdbf:relationship>
 <cmdbf:relationship>
  <cmdbf:source>
   <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
   <cmdbf:localId>staff01</cmdbf:localId>
  </cmdbf:source>
  <cmdbf:target>
   <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
   <cmdbf:localId>01</cmdbf:localId>
  </cmdbf:target>
  <cmdbf:record xmlns="http://school">

   <class name="Economics" courseCode="ECM01">
    <students>
     <enrolledStudent idRef="01"/>
     <enrolledStudent idRef="03"/>
    </students>
    <teacher idRef="staff01"/>
   </class>

   <cmdbf:recordMetadata>
    <cmdbf:recordId>ECM01</cmdbf:recordId>
```

```
    </cmdbf:recordMetadata>
   </cmdbf:record>
   <cmdbf:instanceId>
    <cmdbf:mdrId>org.eclipse.cosmos.samples.cmdbf.XMLRepository</cmdbf:mdrId>
    <cmdbf:localId>ECM01</cmdbf:localId>
   </cmdbf:instanceId>
  </cmdbf:relationship>
 </cmdbf:edges>
</cmdbf:queryResult>
```

Student Teacher client proxy

Put your short description here; used for first paragraph and abstract.

COSMOS provides a remote client proxy for invoking the CMDBf query service. Since this example implements the query service, we can use the query service client to invoke the query operation.

Here is a code snippet for invoking the query service using the query service client. This example loads a query from a file, invokes the GraphQuery operation of the Example MDR query service, and prints out the query response in the console.

```
// EPR of the query service
String queryServieEpr = "http://localhost:8080/axis2/services/ExampleMdrQueryService";

// instantiate the query service client
QueryServiceClient client = new QueryServiceClient(queryServieEpr);

// path to a file that contains a CMDBf query
String filepath = "c:\cosmos-demo\cosmos-client\cmdbfQuery\ExampleMDR";

try {
    // loads the query into a DOM tree
    Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(new File(queryF
    Element elem = doc.getDocumentElement();

    // invokes the query
    Element response = client.graphQuery(elem);

    // Writes the response using a StringWriter
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer trans = tf.newTransformer();
    StringWriter sw = new StringWriter();
    trans.transform(new DOMSource(response), new StreamResult(sw));

    // prints out the response
    System.out.println("Query response:");
    System.out.println(sw.toString());

} catch (CMDBfException cmdbfex) {
    // In case the query operation returns a SOAP fault,
    // here is how you can catch the fault and inspect the fault message
    if (cmdbfex.getCause() instanceof AxisFault) {
        AxisFault axisfault = (AxisFault) cmdbfex.getCause();
        System.out.println("Fault code: " + axisfault.getFaultCode());
        System.out.println("Fault subcode: " + axisfault.getFaultSubCodes());
        System.out.println("Fault message: " + axisfault.getMessage());
        System.out.println("Fault details: " + axisfault.getDetail());
    }
}
```

Adding custom capabilities

Put your short description here; used for first paragraph and abstract.

Besides supporting the CMDBf query and registration services, a data manager can also provide other web services. WTP provide tooling for creating web services. Refer to the Reference section for links to tutorials on how to create web services. After you have created the web service, update the services.xml file to include the definition of the new web service.

Running Student Teacher MDR in eclipse

Eclipse is required to run this MDR.

The example MDR can be tested by running within eclipse. Use these steps to run the Example MDR

1. Right click on the web project, and select **Run As... > Run on Server**
2. Select Tomcat 5.5 server, and click **Next**
3. Click **Finish** on the next screen.

You may want to run the web service in debug mode so that you can step through the code. To run in debug mode, select ″Debug as...″ instead of ″Run As...″ in step 1 above.

Deploying Student Teacher MDR in Tomcat

After you have tested am MDR in eclipse, you can deploy it on a Tomcat server for the production environment. You first need to have Axis2 web archive (version 1.3) installed in tomcat, which can be downloaded from here: http://ws.apache.org/axis2/download/1_3/download.cgi. Then you need to package the web service in axis archive (AAR) format, which has the following directory structure:

```
<data manager name>/lib/
                   /META-INF/
                   /<class files>
```

The root directory is the name of the data manager. It will be used as the axis2 service group name. The lib directory contains all dependency jar files. META-INF directory contains all WSDL and XSD files for the web service definition, as well as the services.xml file. The class files are also packaged in their corresponding Java package directory structure. The Student-Teacher MDR will have the following structure.

```
ExampleMdr/lib/*.jar
          /META-INF/cmdbfQuery.wsdl
                   /QueryService.wsdl
                   /services.xml
                   /cmdbfDatamodel.xsd
          /org/eclipse/cosmos/provisional/example/mdr/ExampleMdr.class
```

This web service directory can then be copied to the axis2 services directory (<tomcat>/webappsaxis2/WEB-INF/services).

Reference

Links to tutorials

These Tutorials are for using WTP to create web services

- Eclipse WTP Tutorials - Creating Bottom Up Web Service via Apache Axis2
- Eclipse WTP Tutorials - Creating Top Down Web Service via Apache Axis2.

Chapter 3. Extending the Web user interface framework

This topic is the introduction to extending the Web user interface.

The COSMOS Web user interface framework is a web-based framework that facilitates the assembly of web components into a single web console application. It includes the following items.

- A set of out-of-the-box web widgets.
- A registration service to discover and share web widget configurations.
- A page template processor.

A web widget is a lightweight, portable component that can be installed and can run within any separate HTML-based web page without requiring a separate compilation.

The focus of the COSMOS UI framework is to provide reusable web widgets to help facilitate the development and validation of Management Data Repositories (MDRs) as defined by the Configuration Management Database Specification Federation specification (CMDBf). An MDR is a CMDBf term that represents a component that contains data about managed resources (e.g. computer systems, application software and building) and process artifacts (e.g. incident records and requests for change forms) and the relationships between them.

This section first covers the overall architecture of the COSMOS UI framework, followed by a set of tutorials that help the user understand basic concepts and extension mechanisms.

What you need to know

Before reading this topic be aware that the COSMOS UI framework uses web methodologies such as AJAX and web technologies such as HTTP, JSON, XML, and Javascript. You need to be quite familiar with these technologies and methodologies. In addition, the framework heavily relies on the DOJO toolkit. You must have a good understanding of the following DOJO toolkit concepts.

- JavaScript Programming With Dojo and Dijit
- Dijit - The Dojo Widget Library.

Web user interface framework design overview

This topic explains the web user interface framework design.

Consider the various configuration scenarios that is possible with the framework. This provides an understanding of the kinds of problems the framework can solve.

Scenario 1: Provide a web console for users to query a Data Manager

In this scenario, a developer would like to assemble a web console made of a set of widget components provided by COSMOS. You want to create a whole new look and feel of the COSMOS UI console in terms of how the widget components are laid out, and how they interact with each other. Furthermore, you want to change certain cosmetic attributes like fonts, background color, and so on.

Scenario 2: Develop a web widget to manage a Data Manager

In this scenario, the existing out-of-the-box web components can not be utilized to interact with the data manager. You want to create your own web widget to visualize the contents of your data manager and submit queries to the data manager. These new custom web widgets are better suited to interact with your data manager.

Scenario 3: Develop a report to visualize the contents of a Data Manager

You want to develop a report that will visualize the data from a particular data manager. These new reports provide a way to visualize the data so that consumers of the data can analyze the information more efficiently.

Examine the overall component architecture of the COSMOS UI framework and explore how the framework meets these requirements for each scenario.

The COSMOS UI framework is comprised of these key components.

- "Pages templates"
- "Widgets" on page 28
- "Widget Configuration files" on page 28
- "Data feeds" on page 31
- Reporting system
- Report templates.

Pages templates

This topic explains the purpose of the pages template component of the COSMOS Web User Interface framework.

A page template is a concept introduced by the COSMOS UI infrastructure. These templates are jsp files that create the structure of the page. You have the freedom to create the layout of the page and import styles sheets to change the look and feel of the page. As a result, the COMOS UI framework exploits the capabilities of JSP pages to define page templates.

A pages folder should be defined under the web application's root directory. Template files and any associated resource files are deployed as sub-folders under the ″pages folder. Let us consider creating and deploying a template file in the COSMOS UI infrastructure. Let us also consider that our web application is named **COSMOSUI**. We would have the following file structure under our web application root directory.

```
COSMOSUI/pages
COSMOSUI/pages/mytemplate/index.jsp
COSMOSUI/pages/mytemplate/css/style.css
```

Note that we have defined a″mytemplate″ sub folder that bundles our template file (i.e. index.jsp) and any other associated files (i.e. styles.css). Users can then load our template file by entering the following url into a browser.

```
http://localhost:8080/COSMOSUI/?page=mytemplate
```

This causes the COSMOS UI infrastructure to load the template file under the COSMOSUI/pages/mytemplate folder.

The framework provides a **Template** bean which is required by each template file to set up bootstrapping code. The bootstrapping code provides the following operations.

- Loads COSMOS web components into the page
- Loads DOJO toolkit into the page
- Initializes the COSMOS web components.

The Template bean also provides helper methods that return the url path to the DOJO toolkit. Let us look at the header section of a typical template file.

Notice that the template file uses the `org.eclipse.cosmos.provisional.dr.ps.common.Template` bean on line 2.

The template file uses the **getDojoBaseUrl** method, on lines 10-12, to get the url location of the dojo toolkit so that the dojo style sheets can be referenced.

Also note that on line 13 the **getPage** method is used to reference the url location where this template file is deployed. Therefore files can be referenced relative to where this template file is deployed.

On line 17 a call is made to the **header** method. This method generates the bootstrapping code that is needed to initialize and load the COSMOS and Dojo widget components.

```
1   <%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"
2   <jsp:useBean id="template" scope="request" class="org.eclipse.cosmos.provisional.dr.ps.common.
3
4   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
5   <html>
6   <head>
7   <title>COSMOS Web User Interface</title>
8
9   <style type="text/css">
10   @import "<%= template.getDojoBaseUrl() %>/dojo/resources/dojo.css";
11  @import "<%= template.getDojoBaseUrl() %>/dijit/themes/tundra/tundra.css";
12  @import "<%= template.getDojoBaseUrl() %>/dojox/grid/_grid/Grid.css";
13  @import "<%= template.getPage() %>/css/style.css";
14
15 </style>
16
17 <%= template.header() %>
18
19 </head>
```

Once the header is constructed you can now construct the body of the template file. Construct the layout of the page within the body and define attachpoints at certain sections in the page. Attachpoints are tags that help the COSMOS UI determine which web widgets should be rendered within certain sections within the page. Consider the following body content.

```
1  <body class="tundra">
2
3   <table style="background-color:#eee" width="100%">
4    <tr><td>
5    <div dojoType="org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer" attach
6   </td></tr>
7   <tr><td>
8   <div dojoType="org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer"" attach
9   </td></tr>
10  </table>
11
12 </body>
```

Notice attachpoints are created by declaring a `org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer` element and setting the *attachPoint* attribute to a tag name. This identifies these section of the page as attach points. As a result any views that are registered to any of these tag names are placed in the particular section within the page.

Widgets

As stated in previous sections, widgets are web components that can be assembled into a unified web console. COSMOS provides a set of out-of-the-box widgets that are outlined under the Web Component Library document. It is possible to extend the existing widgets or create new widgets using the Dojo toolkit. As a result the reader should be familiar with the Dojo widget library and life cycle of the Dojo widgets. It is assumed that the reader is familiar with creating and extending a widget based on the Dojo toolkit programming model.

Let us look at the life cycle of a widget within the COSMOS UI infrastructure when processing a page template.

1. The page template is processed and the COSMOS bootstrapping code is executed. Within this phase the Dojo and COSMOS javascript files are imported into the page.
2. Next an `org.eclipse.cosmos.provisional.dr.ps.components.utility.UIContext` class is instantiated. The UIContext object provides widget factory methods and other convenient methods to submit XML HTTP Requests.
3. Next the declared `org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer`, within the page template, are created with their associated *attachPoint* names. During creating of the WidgetContainer object a call is made to the UIContext widget factory method to create and return all the widgets associated with the *attachPoint* name. It should be noted that each widget created via the factory methods will get a handle to the UIContext object. This allows these widgets to use the convenient methods provided by the UIContext class. The WidgetContainer then appends the newly created widget to its root DOM node resulting in the widget being rendered in the page.

Widget Configuration files

Each web component exposes configuration properties that are stored in configuration files For example, a configuration file may store the table definition for a grid widget, the background color of a button widget, or the url location that contains the contents of a tree widget.

Structure

The COSMOS UI provides a registry that associates a configuration file with a tag value. Page templates and some widget components will utilize the register to get configuration information. For example a page template will define a section within in a page that is associated with a specific tag value. The page template will use this tag value to lookup the web component that is associated with the tag from the registry.

The registry reads a directory made up of configuration files to populate its content. Each entry in the registry is composed of a tag value and a configuration set. The tag value is constructed from the path location of the configuration file. For example, if a configuration file was stored under a `detail` directory the tag

value associated with the configuration file would be *detail*. Similarly, if the configuration file was stored under the `detail\nav` directory the tag value would be named *detail nav*. Notice that the tag value reflects the location of the configuration file and replaces that path separator character with a space.

The registry is located under the views directory of a web application. To get an idea what a typical registry structure would look like, let us consider the following directory structure

```
COSMOSUI\views\_cosmos_context_
COSMOSUI\views\nav
COSMOSUI\views\properties
COSMOSUI\views\detail
COSMOSUI\views\detail\datasource
COSMOSUI\views\detail\nav
COSMOSUI\views\detail\statdatasource
COSMOSUI\views\detail\nav\queryCMDBf
COSMOSUI\views\attributeTable
COSMOSUI\views\attributeTable\cmdbfQueryDetail
COSMOSUI\views\attributeTable\cmdbfQueryDetail\any
COSMOSUI\views\attributeTable\cmdbfQueryDetail\contentSelector
COSMOSUI\views\attributeTable\cmdbfQueryDetail\depthLimit
COSMOSUI\views\attributeTable\cmdbfQueryDetail\expression
COSMOSUI\views\attributeTable\cmdbfQueryDetail\instanceId
COSMOSUI\views\attributeTable\cmdbfQueryDetail\itemTemplate
COSMOSUI\views\attributeTable\cmdbfQueryDetail\prefixMapping
COSMOSUI\views\attributeTable\cmdbfQueryDetail\propertyValue
COSMOSUI\views\attributeTable\cmdbfQueryDetail\recordType
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relany
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relationshipTemplate
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relexpression
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relinstanceId
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relontentSelector
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relprefixMapping
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relpropertyValue
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relrecordType
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relselectedProperty
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relselectedRecordType
COSMOSUI\views\attributeTable\cmdbfQueryDetail\relxpathExpression
COSMOSUI\views\attributeTable\cmdbfQueryDetail\selectedProperty
COSMOSUI\views\attributeTable\cmdbfQueryDetail\selectedRecordType
COSMOSUI\views\attributeTable\cmdbfQueryDetail\sourceTemplate
COSMOSUI\views\attributeTable\cmdbfQueryDetail\targetTemplate
COSMOSUI\views\attributeTable\cmdbfQueryDetail\xpathExpression
COSMOSUI\views\cmdbfQueryDetail
COSMOSUI\views\cmdbfQueryDetail\cmdbfQueryOverview
COSMOSUI\views\cmdbfQueryDetail\cmdbfQueryOverview\item
COSMOSUI\views\cmdbfQueryDetail\cmdbfQueryOverview\relationship
COSMOSUI\views\cmdbfQueryOverview
```

The above directory structure defines a views directory under a web application named **COSMOSUI**. The resulting registry would have the following keys.

```
_cosmos_context_
nav
properties
detail
detail datasource
detail nav
detail statdatasource
detail nav queryCMDBf
attributeTable
attributeTable cmdbfQueryDetail
attributeTable cmdbfQueryDetail any
attributeTable cmdbfQueryDetail contentSelector
attributeTable cmdbfQueryDetail depthLimit
```

```
attributeTable cmdbfQueryDetail expression
attributeTable cmdbfQueryDetail instanceId
attributeTable cmdbfQueryDetail itemTemplate
attributeTable cmdbfQueryDetail prefixMapping
attributeTable cmdbfQueryDetail propertyValue
attributeTable cmdbfQueryDetail recordType
attributeTable cmdbfQueryDetail relany
attributeTable cmdbfQueryDetail relationshipTemplate
attributeTable cmdbfQueryDetail relexpression
attributeTable cmdbfQueryDetail relinstanceId
attributeTable cmdbfQueryDetail relontentSelector
attributeTable cmdbfQueryDetail relprefixMapping
attributeTable cmdbfQueryDetail relpropertyValue
attributeTable cmdbfQueryDetail relrecordType
attributeTable cmdbfQueryDetail relselectedProperty
attributeTable cmdbfQueryDetail relselectedRecordType
attributeTable cmdbfQueryDetail relxpathExpression
attributeTable cmdbfQueryDetail selectedProperty
attributeTable cmdbfQueryDetail selectedRecordType
attributeTable cmdbfQueryDetail sourceTemplate
attributeTable cmdbfQueryDetail targetTemplate
attributeTable cmdbfQueryDetail xpathExpression
cmdbfQueryDetail
cmdbfQueryDetail cmdbfQueryOverview
cmdbfQueryDetail cmdbfQueryOverview item
cmdbfQueryDetail cmdbfQueryOverview relationship
cmdbfQueryOverview
```

As a result, widget components or page templates can lookup the configuration options based on one of the above tag values. Configuration options are defined in two separate files.

- view.jprop
- datamap.jprop

view.jprop

The view.jprop file defines a JSON object that associates a widget to a tag value. The COSMOS UI infrastructure processes the view.jprop file to determine which web components are rendered within a page template. Let us go back and revisit how page templates are created. Consider the following page template.

```
1  <body class="tundra">
2
3   <table style="background-color:#eee" width="100%">
4    <tr><td>
5    <div dojoType="org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer" attachPo
6   </td></tr>
7   <tr><td>
8   <div dojoType="org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer"" attachPo
9   </td></tr>
10  </table>
11
12 </body>
```

Remember that the page template uses the `org.eclipse.cosmos.provisional.dr.ps.components.widget.WidgetContainer` to define *attachPoints*. The values of these attachPoints are used to lookup the configuration options from the registry. Therefore, the above page template defines two attachPoints (i.e. *nav*, and *detail*). When the page template is processed the configuration files associated with the two keys are found from the registry. The web components specified in the configuration files are rendered in the appropriate sections within the page. Consider the following view.jprop file.

```
1 {
2   clazz: "org.eclipse.cosmos.provisional.dr.ps.components.widget.QueryNavigator",
3   query: {nodeClass:'*'},
4   id:"myTree",
5   initQueryHandler: "json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/DomainOut
6   queriesHandler: "json?service=org/eclipse/cosmos/internal/dr/drs/service/handler/common/Queries
7   cmdbfQueryHandler:"json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/CMDBfQuer
8   progressDescription: "Loading Data Managers",
9   publish: ['properties', 'detail']
10 }
```

The *clazz* property defined on line 2 is a special property used by the COSMOS UI to indicate the web component class. The following properties on lines 3-9 are configuration options used to configure the web component. Therefore, the above view.jprop file configures a `org.eclipse.cosmos.provisional.dr.ps.components.widget.QueryNavigator` web component.

datamap.jprop

A datamap.jprop file is also defined in the same folder as the view.jprop file. The datamap.jprop file defines a small registry used by the web component. Consider the following datamap.jprop file.

```
{
   mdrcbe:{
     icon: "mdrcbeIcon",
            expandQuery:{clazz:"org.eclipse.cosmos.provisional.dr.ps.components.utility.BasicQuery
    },
   element:{
     icon: "elementIcon"
    },
   rootElement:{
     icon: "elementIcon"
    }
}
```

The above file defines a registry that contains three tags(i.e. mdrcbe, element and rootElement). A set of configuration options are associated with each key. A widget component would use this registry to configure itself. For example, the `org.eclipse.cosmos.provisional.dr.ps.components.widget.QueryNavigator` web component would use the above registry to determine which icons should be used when rendering particular node data within its tree widget.

Data feeds

Each web component is associated with a set of data feeds that produces configuration and content data. Each data feed produces a specific JSON structure that is consumed by a particular web component.

The COSMOSUI infrastructure provides a framework to create data feeds. Developers are not obligated to utilize this mechanism and are free to create their own data feeds based on other server side frameworks and technologies. COSMOSUI defines a servlet that acts as a gate keeper to data feed requests. This servlet handles the HTTP request and delegates the request to an Outputter class. An Outputter class is a simple java bean that takes in an input map and produces a data feed.

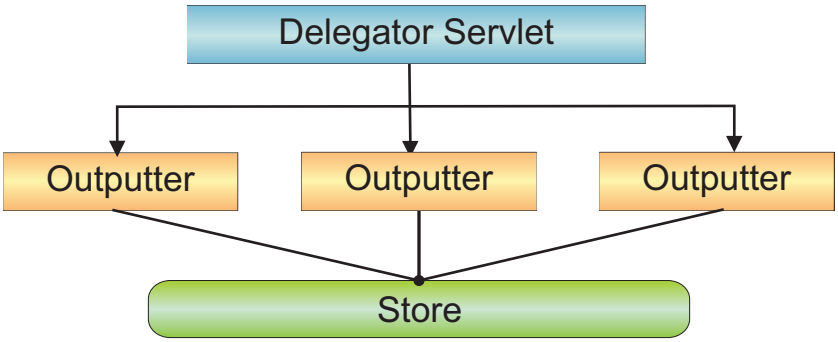

Figure 1. Delegator Servlet

A request delegator servlet receives the request and instantiates a particular outputter that handles the request. The request delegator provides interfaces to deserialize the request into a set of parameters that the outputter understands. The request delegator also provides a global store in which outputters can save state data. Note that the outputter themselves are stateless and can only change the state of the store.

The following sections define the IOuputter, IParameters and IStore interfaces.

IOuputter

```
/**
 * Provides data feeds
 */
public interface IOutputter {

 /**
  * A resolver class that will generate unique ids.  unique ids
  * may be required by the outputter to identify particular items in
  * the generated output.
  * @param idResolver id resolving class
  */
 public abstract void setIdResolver(IIDResolver idResolver);

 /**
  * Writes content to a PrintWriter.  An input map is passed to this method
  * that the render method will use to generate the data feed
  * @param output a PrintWriter that method will write to
  * @param input an input map that contains name value pairs
  * @throws Exception
  */
 public abstract void render(PrintWriter output, IParameters input) throws Exception;

 /**
  * This method is called write after instantiating the outputer.  A persisted storage
  * object is passed that outputters can use to save state
  * @param store a persistent storage object
  * @param parameters an input map that contains name value pairs
  * @throws Exception
  */
 public abstract void initalize(IStore store, IParameters parameters) throws Exception;

}
```

IParameters

```
/**
 * Provides a list of name value map that is used by outputters as input
 * parameters.
 */
```

```
public interface IParameters {
 /**
  * Returns a parameter value with an associated key name
  * @param name - key name
  * @return value of the parameter
  */
 public String getParameter(String name);
}
```

IStore

```
/**
 * Persistent storage used by outputters to save state
 */
public interface IStore {
 /**
  * Returns a value from the store with provided key name
  * @param name - key name
  * @return stored value
  */
 public abstract Object getAttribute(String name);
 /**
  * Stores a value with an associated key name
  * @param name - key name
  * @param value - value to store
  */
 public abstract void setAttribute(String name, String value);
}
```

Data Tagging

Data tagging is an important design notion in the COSMOS UI. Wikipedia provides a definition of meta data tagging.

COSMOS UI distinguishes two types of data tags.

- Data Tags
- Render Tags

Data tags

Data tags provide meta data information about the data. For example, consider the following JSON structure.

```
[
  {type:'cpu', id:'23423511'},
  {type:'mouse' id:'23122'},
  {type:'os' id:'23122'}

]
```

The above information shows three resources that present a cpu, a mouse and an operating system. Data can be tagged in multiple ways. The following shows three options.

Option 1 - categorize data based on hardware/software

```
[
  {type:'cpu', id:'23423511', tag:'hardware'},
  {type:'mouse' id:'23122', tag:'hardware'},
  {type:'os' id:'23122', tag:'software'}
]
```

Option 2 - categorize data based on resource type

```
[
  {type:'cpu', id:'23423511', tag:'cpu'},
  {type:'mouse' id:'23122', tag:'mouse'},
  {type:'os' id:'23122', tag:'os'}
]
```

Note that option 3 uses nested tags to provide meta data information on the resource.

Option 3 - categorize data based on hardware/software and resource type

```
[
  {type:'cpu', id:'23423511', tag:'hardware cpu'},
  {type:'mouse' id:'23122', tag:'hardware mouse'},
  {type:'os' id:'23122', tag:'software os'}
]
```

The COSMOS visualization uses this information to render the data. For example, different icons can be associated with a resource depending on the tag value. Furthermore, if the above data structure is rendered in a tree, the tree widget may show different popup menus when the user selects the node depending on the tag value.

The COSMOS visualizations reads these tags to determine how to render the information.

Render tags

Render tags provide meta data information describing how the data is rendered. Consider the case of a tree widget, table widget and another tree widget.

When the widgets are rendered in the COSMOS UI a tag is associated with each widget.

When data is rendered in each of these widgets, the widget tags the data with their tag value. Consider the following data.

```
[
  {type:'cpu', id:'23423511', tag:'hardware'},
  {type:'mouse' id:'23122', tag:'hardware'},
  {type:'os' id:'23122', tag:'software'}
]
```

If this data is rendered in the tree widget with a tag value of *tree1* the data is tagged as follows.

```
[
  {type:'cpu', id:'23423511', tag:'tree1 hardware'},
  {type:'mouse' id:'23122', tag:'tree1 hardware'},
  {type:'os' id:'23122', tag:'tree1 software'}
]
```

Similarly if this data is rendered in the table widget with a tag value of *table* the data is tagged as follows:

```
[
  {type:'cpu', id:'23423511', tag:'table hardware'},
  {type:'mouse' id:'23122', tag:'table hardware'},
  {type:'os' id:'23122', tag:'table software'}
]
```

These types of rendering tags provides additional meta data information on the data. This is useful in cases where data is sent from one widget to another. For

example, if the table widget receives data from two different widgets it may handle the data differently if the data comes from one widget as opposed to the other widget.

Error handling

The COMSOS UI defines a default error handler object that is defined as a dojo object (for example,`org.eclipse.cosmos.provisional.dr.ps.components.utility.ErrorHandler`). This allows you to extend or replace the error handler by utilizing the dojo programming model.

The error handler utilizes the dojotopic system. This system makes it easy for separate components to communicate without explicit knowledge of one another's internal implementations. The error handler object subscribes to the `org.eclipse.cosmos.provisional.dr.ps.components.utility.ErrorHandler` topic. Web components can publish their error message to this topic to log the error. The following sample shows how this is accomplished.

```
dojo.publish("org.eclipse.cosmos.provisional.dr.ps.components.utility.ErrorHandler",
 [{message:{message:"My Message", detail:"Message details"}, prompt:true, severity:1}]);
```

The above line of code publishes a message object to the error handler topic. The message object has the following structure.

```
{
   //message: Object
   //  object that contains the message information
   message:{
              //message:String
              //   text summary of the message
              message:"",
              //detail:String
              //   text detail of the message
              detail:""
           },
   //prompt:Boolean
   //  set to true if the message should prompt the user with the message, false otherwise.
   prompt:true,
   //severity: Integer
   //  indicates the severity of the message.  The following are values for severity:
   //     1 - ERROR
   //     2 - INFO
   //     4 - WARNING
   //     0 - NONE
   severity:1
}
```

Server Side Logging

The default error handler object provides a means for a server side component to receive error messages from the client. The client uses the post method to send logged messages from the client to the server side error component. The following define the POST arguments sent in the request to the server component

message
a string representation of the message

severity
this is a numeric value 1,2 or 4 representing an error, informational or warning message

Macro Language

This topic explains the macro language that is used by the COSMOS UI infrastructure.

The COSMOSUI infrastructure makes use of a macro language to allow users to externalize strings found in configuration files into a resource bundle. This provides the following benefit.

- supports internationalization for configuration files.
- defines global values that can be applied to configuration files.

Examine the sample configuration file and see how you would use the macro language to externalize strings and use global values.

```
1 {
2   clazz: "org.eclipse.cosmos.provisional.dr.ps.components.widget.QueryNavigator",
3   query: {nodeClass:'*'},
4   id:"myTree",
5   initQueryHandler: "json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/
    DomainOutputter&query=initDM&nodeDecorator=org.eclipse.cosmos.examples.dr.drs.service.outputter.
6   queriesHandler: "json?service=org/eclipse/cosmos/internal/dr/drs/service/handler/common/QueriesO
7   cmdbfQueryHandler:"json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/CMDBfQuery"
8   progressDescription: "Loading Data Managers",
9   publish: ['properties', 'detail']
10 }
```

On line 8 a hard-coded English string specifies the progress indicator description. We want to externalize this string so that we can substitute different sting values based on the locale. We accomplish this by changing the configuration file on line 8 and specifying the {$progressDesciprtion} macro as shown in the next sample.

```
1 {
2   clazz: "org.eclipse.cosmos.provisional.dr.ps.components.widget.QueryNavigator",
3   query: {nodeClass:'*'},
4   id:"myTree",
5   initQueryHandler: "json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/
     DomainOutputter&query=initDM&nodeDecorator=org.eclipse.cosmos.examples.dr.drs.service.outputter
6   queriesHandler: "json?service=org/eclipse/cosmos/internal/dr/drs/service/handler/common/QueriesO
7   cmdbfQueryHandler:"json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/CMDBfQuery"
8   progressDescription: "${progressDescription}",
9   publish: ['properties', 'detail']
10 }
```

Macros are defined in properties files that are deployed in the classpath of the web application. These properties files will be loaded as java resource bundles. As a result, a set of property files can be created that are associated with different locales. Following the example you would define a properties file with the following content.

```
progressDescription = "Loading Data Managers..."
```

You can use this mechanism to define global macros that can be applied to configuration files. Consider a case where you deployed your data feeds on a server named "http://foo:8080. It is cumbersome to change all the configuration files to point to http://foo:8080. However, you can use macros to reference a global configuration value. Consider the following changes to the previous configuration file.

```
1 {
2   clazz: "org.eclipse.cosmos.provisional.dr.ps.components.widget.QueryNavigator",
3   query: {nodeClass:'*'},
4   id:"myTree",
5   initQueryHandler: "${baseURL}json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter/
```

```
     DomainOutputter&query=initDM&nodeDecorator=org.eclipse.cosmos.examples.dr.drs.service.outputt
6   queriesHandler: "${baseURL}json?service=org/eclipse/cosmos/internal/dr/drs/service/handler/com
7   cmdbfQueryHandler:"${baseURL}json?service=org/eclipse/cosmos/internal/dr/drs/service/outputter
8   progressDescription: "${progressDescription}",
9   publish: ['properties', 'detail']
10 }
```

A ${baseURL} macro is defined to reference the base url. You can then define a properties file that contains this type of configuration value as follows:

```
baseURL=htt://foo:8080
```

Globalization

This topic explains the implementation of Globalization in the COMOSUI infrastructure.

The COMSOSUI infrastructure is composed of the following of components that may contain strings that are locale sensitive.

- Web components Data feeds
- Configuration files

The web components provided by the COSMOS UI infrastructure are developed using the Dojo toolkit. Therefore it is expected that the you follow the Dojo globalization porgramming model to externalize locale sensitive strings.

Similarly, when developing data feeds, it is expected strings are externalized using the programming model the data feed is developed under. For example, a data feed would use java resource bundles to externalize strings.

As mentioned in the “Macro Language” on page 36 section, configuration files can externalize their strings based on java resource bundles. This is possible because the configuration files are processed by a java based component. This java based component loads property files based on the *WIDGET_BUNDLE_NAME* context parameter defined in the web.xml file of the web application. The context parameter defines a list of bundles that are loaded before processing the configuration files. The following shows the context parameter definition in a web.xml file. Note that two bundles are specified.

```
<context-param>
 <param-name>WIDGET_BUNDLE_NAME</param-name>
 <param-value>org.eclipse.cosmos.examples.e2e.dr.views.messages,org.eclipse.cosmos.examples.e2e.c
</context-param>
```

Chapter 4. Security requirements

This topic describes COSMOS security.

COSMOS 1.0 will ship with limited Security features. COSMOS 1.0 will support MDRs/Data Managers that require a login-id / password. In this scenario, COSMOS passes on the login-id and password to the MDR/Data Manager along with the query string. Additionally, the COSMOS webUI is secured via a simple login / password authentication feature. In later versions of COSMOS, additional Security capabilities will be added.

Scope of COSMOS security

Security standards supported by COSMOS

How COSMOS enables authentication, authorization, and encryption for its Web UI

How COSMOS enables authentication, authorization, encryption for embedded-approach adopters

Integrating your security provider with COSMOS

Appendix A. References

References

Purpose

This little command copies things.

DITA

Parameters

from *this*
copy from somewhere

to *that*
to somewhere else

Sample

This little sample copies “here” to “there”:

```
copy from here to there
```

Appendix B. Legal

Notices

Glossary

D

Definition. A definition is one of two types of documents supported by SML-IF. A definition document is used to describe the structure and constraints of the model.

Domain model. The root of an SML-IF document. Contains a set of definition an instance documents.

I

Instance. An instance is one of two types of documents supported by SML-IF. An instance document describes a model entity.

S

SML. Service Modeling Language (SML) is an XML-based specification that is used to model complex IT resources, services and systems, including their structure, constraints, policies and best practices. An SML model defines a consistent way for computer networks, applications, servers and other IT resources are described or modeled to assist businesses to manage their services that are built on these. The SML specification leverages Schematron and unique XML Schema extensions to implement these models.

SML-IF. Service Modeling Language – Interchange Format (SML-IF) defines an implementation-neutral interchange format that preserves the content and interrelationships among documents.

SML repository. In the SML tooling, this is a flat-file based directory structure. Each file in the structure contains an SML document. The Import operation creates SML document files based on the top level resources found in an input SML-IF document.

SML validation. Validating an SML model in accordance with the SML specification.

SML-IF validation. Validating an SML-IF document in accordance with the SML-Interchange Format specification.

Index

Printed in USA