IBM Rational Team API Tutorial for ClearQuest

Legal Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation North Castle Drive
Armonk, NY 10504-1785
U.S.A.
For license inquiries regarding double-byte (DBCS) information,
contact the IBM Intellectual Property Department in your country or send inquiries,
in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law.

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department BCFB
20 Maguire Road
Lexington, MA 02421
U.S.A.
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs

Trademarks

Contents

Overview

This tutorial provides code examples that use the Rational Team API to implement some common ClearQuest use cases. The use cases include:

- List user databases
- Run a Query and view the result set
- Get a Record and list its fields
- Modify a Record
- Create a Record

Although you should be able to cut and paste the code in this document into compilable Java, it has not been compiled and may contain typographical errors that prevent it from being compiled. Complete, compilable versions of these examples can be found in the samples Eclipse project of the archive distributed with the documentation for the Rational Team API.

What interfaces to import

To access ClearQuest through the Rational Team API, you must import the ClearQuest-specific interfaces. Since these are all in the same package, you can use a wildcard to include all of the ClearQuest-specific interfaces that you will need:

```
import com.ibm.rational.wvcm.stp.cq.*;
```

You will also need the classes from the WVCM package that implement the basic mechanisms of the API. The remaining interfaces in the WVCM package deal specifically with source configuration management resources and are not used when working exclusively with ClearQuest.

```
// WVCM classes to import for ClearQuest applications
import javax.wvcm.PropertyNameList;
import javax.wvcm.ProviderFactory;
import javax.wvcm.ResourceList;
import javax.wvcm.PropertyNameList.NestedPropertyName;
import javax.wvcm.PropertyNameList.PropertyName;
import javax.wvcm.ProviderFactory.Callback;
import javax.wvcm.ProviderFactory.Callback.Authentication;
import javax.wvcm.WvcmException;
```

You will also use many of the interfaces in the STP (software team package) package that specify the extensions to WVCM used by the Rational Team API.

```
// Common Rational Team API classes to import for ClearQuest
import com.ibm.rational.wvcm.stp.StpTeamProvider;
import com.ibm.rational.wvcm.stp.StpQuery;
import com.ibm.rational.wvcm.stp.StpReleasableIterator;
import com.ibm.rational.wvcm.stp.StpRowData;
import com.ibm.rational.wvcm.stp.StpQuery.DisplayField;
import com.ibm.rational.wvcm.stp.StpException;
import com.ibm.rational.wvcm.stp.StpProperty;
import com.ibm.rational.wvcm.stp.StpResource;
import com.ibm.rational.wvcm.stp.StpProperty.MetaPropertyName;
import com.ibm.rational.wvcm.stp.StpChangeContext;
import com.ibm.rational.wvcm.stp.StpRequestList;
import com.ibm.rational.wvcm.stp.StpLocation;
import com.ibm.rational.wvcm.stp.StpRequestList.TargetProperties;
```

Finally, since the sample applications use the Swing GUI, you also must include a number of the Swing, AWT, and Java utility classes.

```
// other utility classes to import for this tutorial
import java.lang.reflect.InvocationTargetException;
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Arrays;
import java.util.Comparator;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableModel;
import javax.swing.JComboBox;
import javax.swing.JFileChooser;
```

Get a provider

The first thing any Team API application needs to do is get a Provider object. This object implements the StpTeamProvider interface and thereby connects the other Team API interfaces to the implementation provided by the Team API library.

There is only one Provider implementation class within the Team API library. It is named by the literal StpTeamProvider.CLIENT_HOSTED_PROVIDER. Pass this to the ProviderFactory.createProvider method to obtain an instance of the Rational Team API Provider class.

ProviderFactory.createProvider also needs to be given a Callback object from which the instantiated Provider can obtain Authentication objects. The Authentication object provides the provider with the credentials needed to authenticate the user as a ClearQuest user before performing operations in a database (such as changing the state of a record or modifying field values). The use of a Callback object that you supply gives your application full control over the acquisition of user credentials (username and password), which may be hardwired into the application, solicited from the user on start up, or the first time it is needed.

Since a Provider object will be needed by all of our examples, we define a getProvider method in our Utilities class to be used by all applications. The Callback used requests user name and password when the user first tries to access a database and then continues to reuse those credentials as long as they are acceptable.

```
/**
  * A simple Authentication object in which the username and (plain-
  * text) password obtained from the user is cached.
  */
    static class UnPw implements Authentication {
        /**
         * Constructs an Authentication object
         *
         * @param unpw
         *            A String[] containing the username and password.
         */
        UnPw(String[] unpw) { m_data = unpw; }

        public String loginName() { return m_data[0]; }

        public String password()
           { return m_data.length > 1 ? m_data[1] : ""; };

        /** The cached credentials */
        private String[] m_data;
    }

/**
 * Constructs an instance of the Team API provider with or without an
 * authentication Callback.
 *
 * @param authenticated
 *       If true, credentials will be requested from the user (via a
 *       pop-up dialog) the first time a repository is accessed.
 *       Subsequent accesses to a repository will attempt to reuse the
 *       provided credentials, requesting new credentials from the user
 *       only if the last-obtained credentials are invalided.
 *       If false, no callback is provided, so only operations that
 *       require no authentication will be allowed.
 * @return The instantiated Provider object
 * @throws Exception
 *            If the Provider could not be instantiated
 */
static StpTeamProvider getProvider(boolean authenticated)
throws Exception
```

```
{
         try {
        Callback callback = !authenticated ? null
            : new ProviderFactory.Callback() {
            private UnPw m_unpw;

            public Authentication getAuthentication(String realm,
                                                    int retryCount) {
                // Try to reuse last credentials on each new repository
                if (m_unpw != null && retryCount == 0)
                     return m_unpw;

                String unpw = JOptionPane
                    .showInputDialog("Enter Username '+' Password for "
                    + realm + " [" + retryCount + "]", "admin+");

                if (unpw == null || unpw.length() == 0)
                    throw new IllegalAccessError("User cancelled request");

                if (unpw.startsWith("@")) {
                    File file = new File(unpw.substring(1));

                    try {
                        FileReader reader = new FileReader(file);
                        char[] buf = new char[100];
                        int count = reader.read(buf);

                        unpw = new String(buf, 0, count);
                        reader.close();
                    } catch (Throwable t) {
                        exception(null,
                                  "Reading password file " + unpw,
                                  t);
                    }
               }

               return m_unpw = new UnPw(unpw.split("\\+", -2));
           }
       };

       // Instantiate a Provider
       return (StpTeamProvider) ProviderFactory
          .createProvider(StpTeamProvider.CLIENT_HOSTED_PROVIDER, callback);
    } catch (InvocationTargetException ite) {
       System.out.println("*** " + ite.getLocalizedMessage());

       StpException ex = (StpException) ite.getTargetException();
       Exception[] nested = ex.getNestedExceptions();

       for (int i = 0; i < nested.length; ++i)
           System.out.println("***  " + nested[i].getLocalizedMessage());

       throw ex;
    }
}
```

Since the Rational Team API reports all errors by throwing an StpException, we include in this Utilities class a method that formats the information in such an exception into a meaningful message and displays it in a Swing dialog.

```
    private static Object messages(Throwable ex)
    {
        String msg = ex.getLocalizedMessage();

        if (msg == null || msg.length() == 0)
            msg = ex.toString();

        if (ex instanceof StpException) {
            Exception[] nested = ((StpException)ex).getNestedExceptions();

            if (nested != null && nested.length > 0) {
                Object[] msgs = new Object[nested.length];

                for (int i=0; i < msgs.length; ++i)
                    msgs[i] = messages(nested[i]);

                return new Object[]{msg, msgs};
            }
        } else if (ex.getCause() != null) {
            return new Object[]{msg, new Object[]{messages(ex.getCause())}};
        }

        return msg;
    }

    static void exception (Component frame, String title, Throwable ex)
    {
        JOptionPane.showMessageDialog(frame, messages(ex), title,
                               JOptionPane.ERROR_MESSAGE);
    }
}
```

List available user databases

The following example lists all user databases (UserDb objects) available from the Provider by calling the CqProvider.userDbFolderList() method. The list of available databases can be used by a client application as a starting point for working with ClearQuest resources.

In this example, each user database is identified by a combination of its database set name (CqDbSet.DISPLAY_NAME) and its user database name (CqUserDb.DISPLAY_NAME). (A database set is sometimes called a configuration or a schema repository). This is the canonical way to identify a user database in the Rational Team API. The complete syntax is “**@**<*db-set*>/<*user-db*>”.

```
/**
 * List the user databases visible in the current context
 */
```

```
public class FindDatabases {

   public static void main(String[] args) throws Exception
   {
      try {
         // Request a list of the CQ databases known to our provider;
         CqProvider provider = Utilities.getProvider(false);
         ResourceList databases = provider.userDbFolderList(DB_PROPS);

         // List the returned information
         for (Iterator dbs = databases.iterator(); dbs.hasNext(); ) {
            CqUserDb userDb = (CqUserDb)dbs.next();
            System.out.println (userDb.getDbSet().getDisplayName() + "/"
                          + userDb.getDisplayName());
         }
      } catch(Throwable ex) {
         ex.printStackTrace();
      } finally {
         System.exit(0);
      }
   }

   // Properties to be displayed for User Databases and Database Sets
   static final PropertyNameList DB_PROPS =
      new PropertyNameList(new PropertyName[] {
         CqUserDb.DISPLAY_NAME,
         CqUserDb.DB_SET.nest(new PropertyName[] {
               CqDbSet.DISPLAY_NAME})
      });
}
```

Run a query

The following code is a Swing application that

1. allows the user to select a database from those available,
2. allows the user to select a query from the available queries in the selected database, and then
3. displays the rows of the result set in a table.

The selection and execution of the query takes place in the run method. The display of the result set happens in the showResults method. Here is the complete example:

```
public class ExecuteQuery {

   static void run(String title, CqProvider provider, Viewer viewer)
   throws WvcmException
   {
      // Request a list of the available databases from the provider
      ResourceList databases = provider.userDbFolderList(null);
      // Display the list and let the user select one to login to
      CqUserDb userDb = (CqUserDb)JOptionPane.showInputDialog
         (null, "Choose a Database to Explore", title,
          JOptionPane.INFORMATION_MESSAGE,
          null, databases.toArray(new Object[]{}), databases.get(0));

      if (userDb == null) System.exit(0);

      // Obtain a list of all queries in the selected database
      userDb = (CqUserDb)userDb.doReadProperties
         (new PropertyNameList(new PropertyName[]{CqUserDb.ALL_QUERIES}), null);

      // Convert the list to a sorted array for use in the selection dialog
      StpQuery[] queries =
         (StpQuery[])userDb.getAllQueries().toArray(new StpQuery[]{});

      Arrays.sort(queries, new Comparator(){
              public int compare(Object arg0, Object arg1)
              { return arg0.toString().compareTo(arg1.toString()); }});

      // Present the list of queries to the user and allow the user to select one
      g_query = (StpQuery)JOptionPane.showInputDialog
                  (null, "Choose a Query to Execute",
                   "All Queries in " + userDb.location().string(),
                   JOptionPane.INFORMATION_MESSAGE, null,
                   queries, queries[0]);

      // Retrieve detailed information about the selected query from the database
      g_query = (StpQuery)g_query.doReadProperties(QUERY_PROPERTIES, null);

      // Execute the query (without parameter)
      StpReleasableIterator results =
         g_query.doExecute(null, 1, Long.MAX_VALUE, true);

      // If the query executed properly, internalize the data and prepare it for display
      if (results.hasNext()) {
          // Column information accessed from the viewer
         g_columns = g_query.getDisplayFields();
         // Row information accessed from the viewer
         g_cell = new StpRowData[Integer.parseInt(results.next().toString())];

         while (results.hasNext()) {
              StpRowData row = (StpRowData)results.next();
              g_cell[(int)row.getRowNumber()-1] = row;
         }

         // Display the query result data
         showResults(g_query.getUserFriendlyLocation().toString(), viewer);
      }
   }

   // Properties of the selected query that we will be using.
   static final PropertyNameList QUERY_PROPERTIES =
      new PropertyNameList(new PropertyName[]{
         StpQuery.DISPLAY_FIELDS,
         StpQuery.DYNAMIC_FILTERS,
         StpQuery.USER_FRIENDLY_LOCATION        });

   // Data made accessible to the GUI components for display
   static StpQuery g_query;
```

```
    static StpRowData[] g_cell;
    static DisplayField[] g_columns;

        /**
      * Displays the result set (in g_cell) in a table.
      * @param title The title string for the result set window
      * @param viewer A Viewer instance to be used for a detailed
      * display of a single resource of the result set. May be null,
      * in which case the option to display a single resource is not
      * presented.
  */
  static void showResults(String title, final Viewer viewer) {
        // Define the table model for the JTable window; one column for each
        // query display field and one row for each row of the query result set.
        TableModel dataModel = new AbstractTableModel() {
            public int getColumnCount() { return g_columns.length; }
            public int getRowCount() { return g_cell.length;}
            public Object getValueAt(int row, int col)
                { return g_cell[row].getValues()[col]; }
            public String getColumnName(int col)
                { return g_columns[col].getLabel(); }
        };

        // Construct the query result window with an optional button for displaying the
        // record in a selected row (used in the View Record and Modify Record examples)
        JFrame frame = new JFrame(title);
        final JTable table = new JTable(dataModel);
        JPanel panel = new JPanel(new BorderLayout());

        if (viewer != null) {
            JButton button = new JButton("Open");

            panel.add(button, BorderLayout.SOUTH);
            button.addActionListener(new ActionListener(){
                    public void actionPerformed(ActionEvent arg0)
                    {
                        int[] selected = table.getSelectedRows();

                        for (int i =0; i < selected.length; ++i)
                            try {
                                viewer.view((Record)g_cell[selected[i]]
                                                        .getResource(g_query));
                            } catch (WvcmException ex) { ex.printStackTrace(); }
                    }
                });
        }

        panel.add(new JScrollPane(table), BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setContentPane(panel);
        frame.setBounds(300, 300, 600, 300);
        frame.setVisible(true);
    }

      /**
       * A simple interface for an object that will display a Record
```

```
        * resource. (Used by extensions to the ExecuteQuery example.)
       */
      static interface Viewer {
         JFrame view(CqRecord resource);
      }

      public static void main(String[] args) throws Exception
      {
         try {
            run("Execute Query", Utilities.getProvider(), null);
         } catch(Throwable ex) {
            Utilities.exception(null, “Execute Query”, ex);
            System.exit(0);
         }
      }
   }
```

The list of available databases is constructed as in the previous example. And the list is presented to the user for selection of a single database to login to.

After the user selects a user database, the ALL_QUERIES property of that database is read into the application. The value of this property is a ResourceList of Query proxies. This list is sorted on the location of the query and presented to the user for selection of a single query to execute.

For selection of the database and selection of the query, the same general-purpose Swing method, JOptionPane.showInputDialog, is used. The input is the array of the proxies to select from and the result is the selected proxy. The proxies’ toString() method is used to generate the list displayed to the user. The toString() method of a proxy generates an image of the location field of the proxy, i.e. Resource.location().string().

After the user selects a query, a number of its properties are read from the database. The DISPLAY_FIELDS property contains information about each column of the result set and will be used to label the columns in the final display. DYNAMIC_FILTERS contains any leaves of the filtering expression that the user should fill in before executing the query. A dynamic filter is sometimes also called a query parameter.

This application ignores the DYNAMIC_FILTERS property and will exhibit strange behavior or perhaps fail if the selected query has dynamic filters. A more robust implementation would examine this property and obtain the missing data from the user before executing the query. This is left as an exercise to the reader.

When executing the query (via StpQuery.doExecute), the sample code requests a count of the total number of records found. This Long value is the first element of the StpResponseIterator returned by StpQuery.doExecute and must be stripped off before processing the rest of the result set. It is used to pre-allocate an array of StpRowData into which the rows of the result set can be stored for use by the GUI table display code (ExecuteQuery.showResults).

The second parameter to ExecuteQuery.showResults (named viewer) is not used in this sample, but will be used in the next example to allow the user to select a row of the result set and display the associated record.

View a Record

The following code extends the previous ExecuteQuery example (Run a query) by adding the ability to view all the fields of a record returned by a query, not just the ones in the query's display fields.

ExecuteQuery.run is invoked so that the user can select and execute a query. In this example, the ExecuteQuerty.run method is provided with a real instance of ExeuteQuerty.Viewer called ViewRecord.Viewer. This causes an "Open" button to appear in the result set display, which when clicked invokes the Viewer.view method. The Viewer.view method (defined below) is passed a proxy for the resource associated with the selected row of the result set. The proxy is obtained using the StpRowData.getResource() method, which in this case returns a CqRecord proxy since that is the type of resource underlying ClearQuest StpRowData.

The substance of this example is, therefore, embodied in the Viewer.view method:

```
// View the current state of a record
public class ViewRecord {

// the following class extends a class defined in the previous section
   static class Viewer implements ExecuteQuery.Viewer {
      Viewer(CqProvider provider)
      {
         m_provider = provider;
      }

      /**
       * @see com.ibm.rational.stp.client.samples.ExecuteQuery.
       * Viewer#view(com.ibm.rational.wvcm.stp.cq.CqRecord)
       */
      public JFrame view(CqRecord record)
      {
         if (record != null) try {
            record = (CqRecord)record.doReadProperties(RECORD_PROPERTIES, null);
            return showRecord("View: ", record, null);
         } catch (WvcmException ex){
            ex.printStackTrace();
         }
      }

     /**
       * Displays the content of an Attachment resource in a text window.
       * @param attachment An Attachment proxy for the attachment to be
       * displayed.
       */
      public void view(CqAttachment attachment)
      {
         if (attachment != null) {
```

```
      try{
         File file = File.createTempFile("attach", "tmp");

         attachment.doReadContent(null, file.getAbsolutePath());
         BrowserDataModel
            .showFile(attachment.getDisplayName().toString(), file);
      } catch(Throwable ex)
      {
         ex.printStackTrace();
      }
   }
}

RecordFrame showRecord(String title, CqRecord record, JComponent[] future)
throws WvcmException {
   final StpProperty.List fields = record.getAllFieldValues();
   // Define a table model in which each row is a property of the record resource
   // and each column is a meta-property of the property, such as its name, type,
   // and value;
   TableModel dataModel = new AbstractTableModel() {
      public int getColumnCount() { return fieldMetaProperties.length; }
      public int getRowCount() { return fields.size();}
      public Object getValueAt(int row, int col)
         {
            try {
               Object val = ((StpProperty)fields.get(row))
            .getMetaProperty((MetaPropertyName)fieldMetaProperties[col].getRoot());

               if (val instanceof CqRecord)
                  return ((CqRecord)val).getUserFriendlyLocation()
                     .getName();
               else if (val instanceof CqAttachmentFolder)
                  return ((CqAttachmentFolder)val)
                     .getAttachmentList().size()
                     + " attachments";
               else
                  return val;

            } catch(Throwable ex) {
               if (ex instanceof StpException) {
                  return ((StpException)ex).getStpReasonCode();
                } else {
                   String name = ex.getClass().getName();
                   return name.substring(name.lastIndexOf(".")+1);
                }
            }
         }
      public String getColumnName(int col)
         { return fieldMetaProperties[col].getName(); }
   };

   // Define the display layout
   final JTable table = new JTable(dataModel);
   final JPanel panel = new JPanel(new BorderLayout());
   final JPanel buttons = new JPanel(new FlowLayout());
   final JButton button = new JButton("View");
final RecordFrame frame =
    new RecordFrame(title + record. getUserFriendlyLocation ().toString(),
                    table, fields);

// Add a button for viewing a selected record or attachment field
buttons.add(button, BorderLayout.SOUTH);
button.setEnabled(false);
button.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent arg0)
        {
            int[] selected = table.getSelectedRows();

            for (int i =0; i < selected.length; ++i) {
                int row = selected[i];
                if (isAttachmentList(fields, row)) {
                    view(selectAttachment(frame, fields, row, "View"));
                } else {
                    view(getRecordReferencedAt(fields, row));
                }
            }
        }
    });

// Add more buttons (used by later examples)
if (future != null)
    for(int i = 0; i < future.length; ++i) buttons.add(future[i]);

table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Ask to be notified of selection changes and enable the view button
// only if a record-valued field or attachment field is selected
ListSelectionModel rowSM = table.getSelectionModel();
rowSM.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        if (!e.getValueIsAdjusting()){
            int[] selected = table.getSelectedRows();
            button.setEnabled(false);

            for (int i=0; i <selected.length; ++i)
                if (getRecordReferencedAt(fields, selected[i]) != null
                    || isAttachmentList(fields, selected[i])) {
                    button.setEnabled(true);
                    break;
                }
        }
    }
});

panel.add(new JScrollPane(table), BorderLayout.CENTER);
panel.add(buttons, BorderLayout.SOUTH);
frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
frame.setContentPane(panel);
frame.setBounds(g_windowX += 10, g_windowY += 10, 600, 300);
frame.setVisible(true);

return frame;
```

```
        }

        protected CqProvider m_provider;
    }

    /** Additional information desired for attachments */
static final PropertyNameList VALUE_PROPERTIES =
        new PropertyNameList(new PropertyName[] {
            StpResource.USER_FRIENDLY_LOCATION,
            CqAttachmentFolder.ATTACHMENT_LIST.nest(
                new PropertyName[] {
                    CqAttachment.DISPLAY_NAME,
                    CqAttachment.FILE_NAME,
                    CqAttachment.FILE_SIZE,
                    CqAttachment.DESCRIPTION})});

    /** The field meta-properties to be requested and displayed */
    static final PropertyName[] fieldMetaProperties =
        new PropertyName[]{
            CqFieldValue.NAME,
            CqFieldValue.REQUIREDNESS,
            CqFieldValue.TYPE,
            CqFieldValue.VALUE.nest(VALUE_PROPERTIES)};

    /**
     * The PropertyNameList to use when reading data from a record to be
     * displayed by this viewer. Note the level of indirection used to request
     * the meta-properties of the fields in the ALL_FIELD_VALUES list rather
     * than those meta-properties of the ALL_FIELD_VALUES property itself.
     */
final static PropertyNameList RECORD_PROPERTIES =
        new PropertyNameList(new PropertyName[]{
            CqRecord.USER_FRIENDLY_LOCATION,
            CqRecord.ALL_FIELD_VALUES
                .nest(new PropertyName[]{StpProperty.VALUE.nest(fieldMetaProperties)})
        });

    /**
     * Examines the property value of a field and, if it references a record,
     * returns a proxy for the referenced record. Otherwise it returns null.
     * @param fields The Property.List to examine.
     * @param row The index of the element in the list to examine.
     * @return A Record proxy if the field references a record; null otherwise
     */
static CqRecord getRecordReferencedAt(StpProperty.List fields, int row)
    {
        try {
            CqFieldValue field = (CqFieldValue)fields.get(row);

            if (field.getType() == ValueType.RESOURCE
                && field.getValue() instanceof CqRecord)
                return (CqRecord)field.getValue();
        } catch (WvcmException ex) { ex.printStackTrace(); }

        return null;
    }
    /**
     * Whether or not the indicated field is an attachment field.
     * @param fields The Property.List to examine.
     * @param row The index of the element in the list to examine.
     * @return true iff the field at the given index is an attachment field
     */
    static boolean isAttachmentList(StpProperty.List fields, int row)

    {
        if (row >= 0) try {
            CqFieldValue field = (CqFieldValue)fields.get(row);

            return field.getType() == ValueType.ATTACHMENT_LIST;
        } catch (WvcmException ex) { ex.printStackTrace(); }

        return false;
    }

/**
     * Presents to the user a list of the attachments associated with a
     * specified field of a record and allows the user to select one.
     * @param frame The parent frame for the dialog generated by this method.
     * @param fields The Property.List to examine.
     * @param row The index of the element in the list to examine.
     * @param op A string identifying the operation that will be performed on
     * the selected attachment.
     * @return An Attachment proxy for the selected attachment; null if the
     * user chooses to make no selection.
     */
static CqAttachment
    selectAttachment(Component frame, StpProperty.List fields, int row, String op)
    {
        CqFieldValue field = (CqFieldValue)fields.get(row);

        try {
            CqAttachmentFolder folder = (CqAttachmentFolder)field.getValue();
            ResourceList attachments =
                (ResourceList)folder.doReadProperties(ATTACHMENT_PROPERTIES, null)
                    .getProperty(CqAttachmentFolder.ATTACHMENT_LIST);

            if (attachments.size() > 0) {
                CqAttachment attachment = (CqAttachment)
                    JOptionPane.showInputDialog
                    (frame, "Choose an Attachment to " + op,
                     op +" Attachment", JOptionPane.INFORMATION_MESSAGE,
                     null, attachments.toArray(new Object[]{}),
                     attachments.get(0));

                return attachment;
            }
         } catch(Throwable t)
            { Utilities.exception(frame, op + " Attachment", t);}

        return null;
    }
```

```
  /**
   * The attachment properties to be displayed in the attachment selection
   * list generated by {@link #selectAttachment}.
   */
  final static PropertyNameList ATTACHMENT_PROPERTIES =
     new PropertyNameList(new PropertyName[]{
        CqAttachmentFolder.ATTACHMENT_LIST.nest(new PropertyName[]{
           CqAttachment.DISPLAY_NAME,
           CqAttachment.FILE_NAME,
           CqAttachment.FILE_SIZE,
           CqAttachment.DESCRIPTION})
     });

  /**
   * The main program for the ViewRecord example. Instantiates a Provider and
   * then invokes the ExecuteQuery example, passing in a version of Viewer
   * that displays fields of a ClearQuest record.
   * @param args not used.
   */

public static void main(
     String[] args)
  {
     try {
        CqProvider provider = (CqProvider)Utilities.getProvider();
        ExecuteQuery.run("View Record", provider, new Viewer(provider));
     } catch(Throwable ex) {
        Utilities.exception(null, "View Record", ex);
        System.exit(0);
     }
  }

   /**
   * An extension of JFrame for the record field display,
   * exposing to clients the JTable component of the frame and
   * the field list that is being displayed in the table.
   */
  static class RecordFrame extends JFrame
  {
     RecordFrame(
        String title,
        JTable table,
        StpProperty.List fields)
     {
        super(title);

        m_table = table;
        m_fields = fields;
     }

     JTable m_table;
     StpProperty.List m_fields;
     private static final long serialVersionUID = 1L;
  }
```

```
        /** X offset for the next window to be displayed */
        private static int g_windowX = 200;

        /** Y offset for the next window to be displayed */
        private static int g_windowY = 200;
    }
```

Within this example, the ViewRecord.Viewer is used not only to display a record returned by a query, but also to view a record referenced by a selected field of a record and to view a file attached to a field of a record. The user may use this feature to browse through references from one record to another.

ViewRecord.view(CqRecord) reads all fields from the record in the database and the meta-properties of each field used by the viewer and passes the populated proxy to the showRecord method. The viewer uses the ALL_FIELD_VALUES property of a ClearQuest record to get a list of all the fields in the record. For each field, the NAME, REQUIREDNESS, TYPE and VALUE meta-properties are requested. Note that these meta-property requests are nested under another VALUE meta-property request so that these meta-properties are obtained for the *value* of the ALL_FIELD_VALUES property and not for the ALL_FIELD_VALUES property itself. (See the declaration of RECORD_PROPERTIES, VALUE_PROPERTIES, and fieldMetaProperties.)

In case the field is an attachment field, the ATTACHMENT_LIST property of the value is also requested. (If the field is not an attachment field, this property request will fail, but since this property is accessed only if the field is an attachment, this failure will not result in an exception.)

ViewRecord.showRecord uses the same Swing GUI components and structure as ExecuteQuery.showResults only the content of the rows and columns of the table differ. In this case, each row is a field of the record, which is expressed as a CqFieldValue object. Each column is a meta-property of the field. The generic StpProperty.getMetaProperty interface is used to fetch each meta-property value from the CqFieldValue/StpProperty structure for each field. With two exceptions, the toString() method for each meta-property value is relied upon for generating an image of the meta-property in the record view. For record resources, just the USER_FRIENDLY_LOCATION property is displayed to reduce the clutter in the output. For attachment fields, only the number of attachments is displayed, not each attachment name.

The display of field types RESOURCE_LIST, JOURNAL, STRING_LIST, and STRING might also be given special attention. This is left as an exercise for the reader.

When an attachment field is selected and the “view” button is clicked, selectAttachment is called and its result is passed to ViewRecord.view(CqAttachment). The selectAttachment method uses JOptionPane.showInputDialog again to present to the user a list of the attachments associated with the selected field. The value of an attachment field is an attachment folder resource. The attachments associated with the field are bound members of that attachment folder.

ViewRecord.view(CqAttachment) uses CqAttachment.doReadContent to read the attached file from the database into a temporary file and then calls a utility method (all Swing code) to display the file to the user.

The ViewRecord.Viewer should also support the display of RESOURCE_LIST values in a separate window. But this, too, has been left as an exercise for the reader.

Modify a record

The next code example allows a user to update the record he is viewing. This example extends the ViewRecord example by extending the ViewRecord.Viewer class to support a combo box for selecting an action and an additional “Edit” button for initiating the selected action on the record. The editRecord method of this Viewer orchestrates the viewing and updating of a record opened for update. These new GUI components are introduced into the ViewRecord.Viewer display using the “future” argument to showRecord.

The view(CqRecord) method now reads the LEGAL_ACTIONS property of the record in addition to the ALL_FIELD_VALUES property. The value of the LEGAL_ACTIONS property is a list of Action proxies for the actions that may legally be applied to the record in its current state. This list is used to populate a combo box control, from which the user may select an action prior to clicking the edit button. Some types of actions, such as SUBMIT and RECORD_SCRIPT_ALIAS are not supported by this example and are, therefore, not added to the combo box control even if present in the set of legal actions.

```
public class EditRecord {
   static class Viewer extends ViewRecord.Viewer {
      Viewer(CqProvider provider)
      {
         super(provider);
      }

      public void view(CqRecord selected)
      {
         try {
            final JComboBox choices = new JComboBox();
            final JButton start = new JButton("Edit");

            final CqRecord record =
                  (CqRecord)selected.doReadProperties(RECORD_PROPERTIES, null);
            Iterator actions = record.getLegalActions().iterator();

            while (actions.hasNext()){
               CqAction a = (CqAction)actions.next();
               if (a.getType() != CqAction.Type.IMPORT
                   && a.getType() != CqAction.Type.SUBMIT
                   && a.getType() != CqAction.Type.BASE
                   && a.getType() != CqAction.Type.RECORD_SCRIPT_ALIAS)
                  choices.addItem(a);
            }
```

```
final JButton add = new JButton("Attach");
final JButton remove = new JButton("Detach");

final ViewRecord.RecordFrame frame =
   showRecord("View: ", record, choices.getItemCount()==0? null:
            new JComponent[]{choices, start, add, remove});
```

The EditView.Viewer also supports an “Attach” and a “Detach” button for adding and removing the attached files of an attachment field.

The “Attach” button presents a standard Swing file-chooser dialog to the user and allows him to select the file to be attached. To attach the file to the record, CqAttachment.doCreateGeneratedResource() is used, passing it the name of the file selected by the user. Before this method can be invoked a CqAttachment proxy addressing the proper location must be constructed. The folder in which the attachment resource will reside is the value of the field to which it is attached—this is the field currently selected in the record view and so it is obtained from the frame object returned by ViewRecord.showRecord. A unique name for the attachment resource is generated from the current time of day. This name is merely a placeholder since doCreateGeneratedResource() is free to change the name of the resource to suit its needs.

```
add.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent arg0)
   {
      if (!ViewRecord.isAttachmentList(frame.m_fields,
                    frame.m_table.getSelectedRow()))
         return;

      JFileChooser chooser = new JFileChooser();

      if (chooser.showOpenDialog(frame) ==
            JFileChooser.APPROVE_OPTION) {
         try {
            String filename =
               chooser.getSelectedFile().getAbsolutePath();
            CqFieldValue field = (CqFieldValue)frame.m_fields.get
               (frame.m_table.getSelectedRow());
            StpLocation aLoc = (StpLocation)
               ((CqAttachmentFolder)field.getValue()).location()
                  .child("new" + System.currentTimeMillis());
            CqAttachment attachment =
               record.cqProvider().cqAttachment(aLoc);

            attachment = attachment
               .doCreateGeneratedResource(null, filename,
                              null, StpChangeContext.UNORDERED);

            JOptionPane.showMessageDialog(frame,
              "Added '" + filename + "' as " + attachment);

            frame.dispose();
            view(record);
        } catch(Throwable t)
            { Utilities.exception(frame, "Add Attachment", t);}
    }}});
```

The “Detach” button uses ViewRecord.selectAttachment to get from the user the identity of the attachment to be removed in the form of an CqAttachment proxy. The doUnbind method of this proxy is then invoked to remove the attachment from the database.

```
remove.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0)
    {
        if (!ViewRecord.isAttachmentList(frame.m_fields,
                        frame.m_table.getSelectedRow()))
            return;

        try {
          CqAttachment attachment = ViewRecord.selectAttachment
              (frame, frame.m_fields, frame.m_table.getSelectedRow(), "Remove");

          if (attachment != null) {
              attachment.doUnbind();
              frame.dispose();
              view(record);
          }
        } catch(Throwable t)
            { Utilities.exception(frame, "Remove Attachment", t);}
        }
    }
);
```

The “Edit” button fetches the selected CqAction proxy from the combo box and passes it and the record proxy for the current viewer to the edit method, which will actually initiate the editing of the record. If the CqAction.Type of the selected action is DUPLICATE, then the id of the duplicated record must be supplied with the action. This value is requested from the user and is placed into the Action’s argument map as the value of the argument named “original”.

```
start.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0)
    {
        CqAction action = (CqAction)choices.getSelectedItem();

        try {
            if (action.getType() == CqAction.Type.DUPLICATE) {
                String id = JOptionPane.showInputDialog
                    (frame, "Enter ID of duplicated record");

                if (id == null) return;

                action.argumentMap(new Hashtable());
                action.argumentMap()
                    .put("original",
                                    record.cqProvider()
                                        .cqRecord((StpLocation)record
                                        .location().parent().child(id)));
                    }

                    edit(record, action);
                    frame.dispose();
                } catch (Exception ex) {
                    Utilities.exception(frame, "Duplicate Action", ex);
                }
            }
        });
    } catch (WvcmException ex){
        Utilties.exception(null, “View Record”, ex);
    }
}
```

The edit method is invoked when the user clicks the “Edit” button in the record viewer. It is passed a proxy for the record to be edited and a proxy for the action under which the edit is to be performed.

Note that no properties need to be defined by the Action proxy used to start a ClearQuest action—only its location and its argument map (if needed) must be defined. To form the location for an action, you need to know its <*name*>, the <*record-type*> of the record it’s going to be used with, and the <*database*> and <*db-set*> where the record is located. The location is then “**cq.action:**<*record-type*>/<*name*>**@** <*db-set*>/<*database*>”; e.g., “cq.action:Defect/Assign@ 7.0.0/SAMPL”

In this example application, a new change context is used for each edit operation. This allows multiple edits to proceed simultaneously without interfering with one another. So, the first thing the edit method does is get a new provider to use for this independent change context. Then it forms a proxy for the record to be edited in this new change context. This step is important because it is the proxy that determines the change context in which an edit will take place. If the input proxy were used instead of this new proxy then the edit would happen in the change context that proxy is associated with and not in the change context that was just created.

Using the new proxy, CqRecord.doStartAction is invoked to begin the edit operation. An StpRequestList is passed to the doStartAction method. It contains the names of the properties and meta-properties needed for editing the record in the form of a PropertyNameList. When the method returns, the StpRequestList will contain a proxy for the record that was opened for edit and that proxy will contain the requested properties. The properties could also be obtained using record.doReadProperties (with the same PropertyNameList) after the doStartAction returns, but that would be less efficient as it would require another roundtrip to the repository for the same data.

```
public void edit(CqRecord selected, CqAction action)
```

```
{
   try {
      CqProvider context = (CqProvider)m_provider.doOpenContext(null);
      CqRecord record = context.cqRecord((StpLocation)selected.location());
      StpRequestList wantedProps =
         new StpRequestList(new TargetProperties(RECORD_PROPERTIES));
      // make the record editable in the change context
      record.doStartAction(action, wantedProps, StpChangeContext.HOLD);
      editRecord("Edit: ", (CqRecord)wantedProps.getResource(), selected);
   } catch (WvcmException ex){
      Utilities.exception(null, "Start Action", ex);
   }
}
```

Viewer.editRecord is quite similar to showRecord. The primary difference is that it defines TableModel.isCellEditable and TableModel.setValue.

```
    JFrame editRecord(
         String title,
         final CqRecord record,
         final CqRecord selected
         ) throws WvcmException
     {
         final JFrame frame =
            new JFrame(title + record. getUserFriendlyLocation().toString());
         JPanel panel = new JPanel(new BorderLayout());
         JPanel buttons = new JPanel(new FlowLayout());
         final JButton show = new JButton("View");
         final JButton add = new JButton("Add");
         final JButton remove = new JButton("Remove");
         final JButton cancel = new JButton("Cancel");
         final JButton deliver = new JButton("Deliver");
         final Property.List fields = record.getAllFieldValues();
         TableModel dataModel = new AbstractTableModel() {
            public int getColumnCount() { return fieldMetaProperties.length; }
            public int getRowCount() { return fields.size();}
            public String getColumnName(int col)
            { return fieldMetaProperties[col].getName(); }
            public Object getValueAt(int row, int col)
               {
                  try {
                     return ((StpProperty)fields.get(row))

.getMetaProperty((MetaPropertyName)fieldMetaProperties[col].getRoot());
                  } catch(Throwable ex) {
                     if (ex instanceof StpException) {
                        return ((StpException)ex).getStpReasonCode();
                      } else {
                        String name = ex.getClass().getName();
                        return name.substring(name.lastIndexOf(".")+1);
                      }
                  }
               }
```

TableModel.isCellEditable returns **true** only for the VALUE column and only if the row belongs to a field whose REQUIREDNESS is not READ_ONLY.

```
    public boolean isCellEditable(int row, int col) {
        if (matches(fieldMetaProperties[col], StpProperty.VALUE)) {
            CqFieldValue field = (CqFieldValue)fields.get(row);

            try {
                return field.getRequiredness()
                        != CqFieldDefinition.Requiredness.READ_ONLY;
            } catch (WvcmException ex) {
                Utilities.exception(frame, "Field Requiredness", ex);
            }
        }

        return false;
    }
```

TableModel.setValueAt sets the new field value into the CqRecord proxy associated with the display. First, the CqFieldValue structure is updated using its initialize() method. This method accepts a string representation for most types of fields and will convert the string into the appropriate data type for the field.

Once the CqFieldValue has been updated, it is set into the CqRecord proxy associated with the PropertyName for the field. This step is necessary so that the new field value gets written to the database when the record is committed. Without this step, the new field value would remain only in the CqFieldValue object that is part of the ALL_FIELD_VALUES property. The ALL_FIELD_VALUES property is not writeable, so the change would never get written to the database. By copying the modified CqFieldValue directly to the proxy entry for the field, that field becomes an "updated" property and the updated value will be written to the database by the next "do" method that is executed on the proxy.

```
    public void setValueAt(Object aValue, int row, int col)
    {
        if (matches(fieldMetaProperties[col], StpProperty.VALUE)) {
            CqFieldValue field = (CqFieldValue)fields.get(row);

            field.initialize(aValue);
            record.setProperty(field.getPropertyName(), field);
        }
    }
};
final JTable table = new JTable(dataModel);

table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
show.setEnabled(false);
add.setEnabled(false);
remove.setEnabled(false);

// Ask to be notified of selection changes.
```

```
ListSelectionModel rowSM = table.getSelectionModel();
rowSM.addListSelectionListener(new ListSelectionListener() {
   public void valueChanged(ListSelectionEvent e) {
      if (!e.getValueIsAdjusting()){
         int[] selected = table.getSelectedRows();
         show.setEnabled(false);
         add.setEnabled(false);
         remove.setEnabled(false);
         for (int i=0; i <selected.length; ++i)
            if (ViewRecord.getRecordReferencedAt(fields, selected[i]) != null) {
               show.setEnabled(true);
            } else if (ViewRecord.isAttachmentList(fields, selected[i])) {
               show.setEnabled(true);
               add.setEnabled(true);
               remove.setEnabled(true);
            }
      }
   }
});
```

The “View” button is the same as the “View” button in a ViewRecord.Viewer.

```
buttons.add(show);
show.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent arg0)
      {
         int[] selected = table.getSelectedRows();

         for (int i =0; i < selected.length; ++i) {
            int row = selected[i];
            Record record =
               ViewRecord.getRecordReferencedAt(fields, row);

            if (record != null) {
               view(record);
            } else if (ViewRecord.isAttachmentList(fields, row)) {
               view(ViewRecord.selectAttachment(frame,
                                       fields, row,
                                       "View"));
            }
         }
      }
   });
```

The “Deliver” button commits the modified record to the database by calling doDeliver() on the record, which, in turn, performs a doDeliver on its change context. After the delivery succeeds, the Viewer is taken down, the change context used for the edit is closed and then a new Viewer is brought up on the original proxy. Since the view() method rereads properties from the database, this new view will display the updated values.

```
buttons.add(deliver);
deliver.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0)
    {
      try {
        int mode = frame.getDefaultCloseOperation();

        record.doDeliver(null, StpChangeContext.UNORDERED);
        frame.dispose();
        record.cqProvider().doCloseContext();
        view(selected).setDefaultCloseOperation(mode);
      } catch (WvcmException ex) {
        Utilities.exception(frame, "Deliver failed", ex);
      }
    }
  });
```

The “Cancel” button abandons the edit operation using the doRevert method of the record proxy. As with the “Deliver” button, the Viewer and the change context are closed and a new Viewer is instantiated for the original proxy.

```
buttons.add(cancel);
cancel.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0)
    {
      try {
        int mode=frame.getDefaultCloseOperation();
        record.doRevert(null);
        frame.dispose();
        record.cqProvider().doCloseContext();
        // In a record-create context, there will be no
        // object to revert to, so just quit
        if (mode == JFrame.EXIT_ON_CLOSE)
          System.exit(0);
        view(selected);
      } catch (WvcmException ex) {
        Utilities.exception(frame, "Cancel failed", ex);
      }
    }
  });
```

The “Attach” and “Detach” buttons are the same as those for the ViewRecord.Viewer, except that the delivery order argument is HOLD instead of UNORDERED. This will defer committing the addition or deletion of the attachment until the entire record is committed via the “Deliver” button.

```
buttons.add(add);
add.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent arg0)
  {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showOpenDialog(frame);

    if (returnVal == JFileChooser.APPROVE_OPTION) {
```

```
                    try {
                      String filename =
                           chooser.getSelectedFile().getAbsolutePath();
                       CqFieldValue field =
                           (CqFieldValue)fields.get(table.getSelectedRow());
                       StpLocation aLoc = (StpLocation)
                           ((StpFolder)field.getValue()).location()
                               .child("new" + System.currentTimeMillis());
                       CqAttachment attachment =
                           record.cqProvider().cqAttachment(aLoc);

                       attachment = attachment
                           .doCreateGeneratedResource(null, filename,
                                                      null, StpChangeContext.HOLD);

                       JOptionPane.showMessageDialog(frame,
                          "Added '" + filename + "' as " + attachment
                          + " (pending delivery)");
                    } catch(Throwable t)
                        { Utilities.exception(frame, "Add Attachment", t);}
                }}});

        buttons.add(remove);
        remove.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent arg0)
            {
                try {
                   CqAttachment attachment = ViewRecord.selectAttachment
                       (frame, fields, table.getSelectedRow(), "Remove");

                   if (attachment != null)
                       attachment.doUnbind(null);
                } catch(Throwable t)
                    { Utilities.exception(frame, "Remove Attachment", t);}
                }
            }
       );

        panel.add(new JScrollPane(table), BorderLayout.CENTER);
        panel.add(buttons, BorderLayout.SOUTH);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setContentPane(panel);
        frame.setBounds(300, 300, 600, 300);
        frame.setVisible(true);

        return frame;
    }
}

/** The following field meta-properties are requested and displayed */
static PropertyName[] fieldMetaProperties = ViewRecord.fieldMetaProperties;

final static PropertyNameList RECORD_PROPERTIES =
         new PropertyNameList(new PropertyName[]{
        CqRecord.USER_FRIENDLY_LOCATION,
        CqRecord.LEGAL_ACTIONS.nest(new PropertyName[]{
         CqAction.USER_FRIENDLY_LOCATION,
         CqAction.DISPLAY_NAME,
         CqAction.TYPE}),
      CqRecord.ALL_FIELD_VALUES.nest(new PropertyName[]{
         StpProperty.VALUE.nest(fieldMetaProperties)})
   });

public static void main(
   String[] args)
{
   try {
      Provider provider = Utilities.getProvider();
      ExecuteQuery.run("Edit Record", provider, new Viewer(provider));
   } catch(Throwable ex) {
      Utilities.exception(null, "Edit Record", ex);
      System.exit(0);
   }
}
```

In this example, all of the edited field values are kept on the client until the user selects the “Deliver” button. Each field modified by the user marks the corresponding property in the record proxy as having been updated. These updated properties are written to the server as the first step of the deliver operation. A problem with this approach is that the new field values are not checked until the final delivery, which could fail if the values are not appropriate for the schema being used.

The GUI could perform a record.doWriteProperties each time a field is modified, which would give the user immediate feedback, but might also be terribly inefficient depending on the communication protocol between the client and the server. The GUI could also provide an “Update” button, which would cause all accumulated updates to be written to the server, without actually delivering them. This would give the schema an opportunity to examine the values and report back errors. These modifications are left to the reader and which approach is taken would depend on the intended use of this application.

Create a record

In this final example, the user is allowed to create a record. Initial dialogs allow the user to select the database in which to create the record and the type of record to create. After the record is created, the user is presented with the EditRecord dialog. In that dialog, the user can set mandatory or optional fields and then deliver the new record to the database.

```
public static void main(String[] args)
{
   try {
      CqProvider provider = Utilities.getProvider();
      Viewer viewer = new Viewer(provider);
      ResourceList databases = provider.userDbFolderList(null);
      CqUserDb userDb = (CqUserDb) JOptionPane
         .showInputDialog(null,
                     "Choose a Database for the New Record",
                     "Create Record",
                    JOptionPane.INFORMATION_MESSAGE,
                    null,
                    databases.toArray(new Object[] {}),
                    databases.get(0));

if (userDb == null) System.exit(0);

userDb = (CqUserDb) userDb
    .doReadProperties(new PropertyNameList(new PropertyName[] {
                           CqUserDb.RECORD_TYPE_SET
                              .nest(RECORD_TYPE_PROPERTIES)}),
                     null);

// Read the list of all record types from the selected database
// and remove from that list those record types that are not
// submittable.
ResourceList rTypes = userDb.getRecordTypeSet();
Iterator types = rTypes.iterator();

while (types.hasNext()) {
    CqRecordType type = (CqRecordType)types.next();

    if (!type.getIsSubmittable())
        types.remove();
}

// Present the list of submittable record types to the user for
// selection
CqRecordType recordType = (CqRecordType) JOptionPane
    .showInputDialog(null,
                    "Choose the type of record to create",
                    "All Record Types in "
                        + userDb.location().string(),
                    JOptionPane.INFORMATION_MESSAGE,
                    null,
                    rTypes.toArray(new CqRecordType[] {}),
                    rTypes.get(0));

if (recordType == null) System.exit(0);

// The EditRecord dialog expects the editable record to be in
// its own change context, so create one for this purpose. It
// will be destroyed by the EditRecord dialog
CqProvider context = (CqProvider) recordType.cqProvider()
    .doOpenContext(null);

// The actual name for the new record is determined by the
// schema. All that is needed here is a "suggested" location
// that makes the record a member of the specified record type.
CqRecord record = context.cqRecord((StpLocation) recordType
    .getUserFriendlyLocation().child("new"));

// Create the record. Don't try to deliver it since mandatory
// fields may need to be set by the user before delivery will
// succeed.
record = (CqRecord) record
```

```
          .doCreateGeneratedResource(null /* default submit action */,
                               new StpRequestList
                                  (new TargetProperties
                                      (RECORD_PROPERTIES)),
                               StpChangeContext.HOLD);

       /*
        * After delivering the created record to its database, the
        * EditRecord dialog will want to redisplay it in its own viewer.
        * This requires a proxy for the record in the initial change
        * context. We need to create this "original" proxy after the fact
        * because we don't have a valid location for the new record until
        * after it has been created. Need to use the stable location
        * because, in some cases, the user-friendly location can change
        * when field values are changed.
        */
       CqRecord selected = recordType.cqProvider()
          .cqRecord(record.getStableLocation());

       // With the new record created in the change context, the process
       // proceeds in the same fashion as editing a record. Mandatory
       // fields must be supplied by the user and then it can be delivered.
       viewer.editRecord("Create Record", record, selected)
          .setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    } catch (Throwable ex) {
       ex.printStackTrace();
       Utilities.exception(null, "Create Record", ex);
       System.exit(0);
    }
 }

 /** The record type properties read prior to creating a record */
 final static PropertyNameList RECORD_TYPE_PROPERTIES =
    new PropertyNameList(new PropertyName[]{
       CqRecordType.USER_FRIENDLY_LOCATION,
       CqRecordType.IS_SUBMITTABLE,
       CqRecordType.DISPLAY_NAME
    });
```