ULC

Add-on Packages

Guide

Canoo RIA-Suite 9.0

canooria-suite

Canoo Engineering AG
Kirschgartenstrasse 5
CH-4051 Basel
Switzerland
Tel: +41 61 228 9444
Fax: +41 61 228 9449
ulc-info@canoo.com
http://riasuite.canoo.com

DISCLAIMER OF WARRANTIES

This document is provided “as is”. This document could include technical inaccuracies or typographical errors. This document is provided for informational purposes only, and the information herein is subject to change without notice. Please report any errors herein to Canoo Engineering AG. Canoo Engineering AG does not provide any warranties covering and specifically disclaim any liability in connection with this document.

TRADEMARKS

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

All other trademarks referenced herein are trademarks or registered trademarks of their respective holders.

Contents

1 Add-on packages

RIA-Suite comprises ULC core and add-on packages. The add-on packages are extensions of ULC to provide features like

- external application integration
- creating charts
- creating graphs
- browser component for displaying HTML5+JavaScript
- creating office documents
- server push
- integration of ULC applications in the Spring framework

The following sections describe features of the add-on packages.

2 ULC Application Integration

This package enables integration of external ULC, Swing and JavaFx applications into a ULC application (host application). By integration it is meant that:

- the UI of the external application is displayed in the window of the host application,
- the host application can start and stop the external application
- the host and the hosted application can exchange messages.

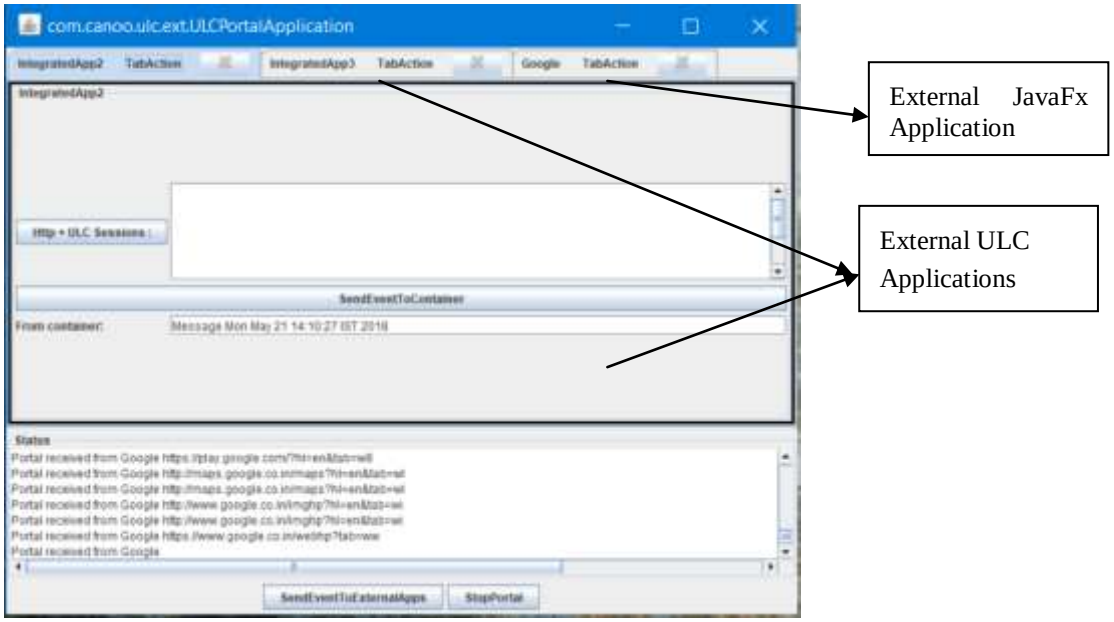
Thus the host ULC application can start and stop external ULC, Swing and JavaFx applications and run the external applications within its window. In this way a developer can create a portal application which can host external ULC, Swing and JavaFx applications.

The idea behind providing ULC Application integration is that the developers can integrate new UI technologies like JavaFx and HTML5/JavaScript in their ULC applications and display these UIs within a window of their existing ULC application. An HTML5/JavaScript based UIs can run within the JavaFx Webview which can be displayed inside a window of a ULC application. On the server side, external applications can be integrated with an existing ULC application by making use of the *ULCHazelcastEventBus* which is provided as part of the *ULCServerPush* package. By using the event bus two applications can exchange messages to update each other's state and notify the ULC client through *ULCServerPush*.

2.1 Host and hosted applications

For integration of an external ULC application into a host ULC application, ULC provides two types of classes: one set of classes for the host application and the second for the external application.

In order to be hosted by an ULC application, an External ULC Application must use *ULCExternalApplicationRootPane* as its top level window instead of a *ULCFrame*. The ULC application has to use *ULCExternalApplicationContainer* to host an external ULC application that has *ULCExternalApplicationRootPane* as its root window.



The following sections describe the classes to be used by the host and the External ULC applications.

2.2 Classes for the Host ULC Application

This section describes the classes to be used by the host ULC application.

2.2.1 ULCExternalApplicationsManager

A host ULC application uses *ULCExternalApplicationsManager* to obtain information from the client side about all the External Applications available for hosting. This information is made available to the host ULC application on the server side in the form of *ExternalApplicationMetaData*.

For each External Application to be hosted in a ULC application, there must exist an implementation of *IExternalApplication* on the classpath of the client side of the host application. All the implementations of *IExternalApplication* must be specified in the file:

com.ulcjava.ext.applicationintegration.client.externalapplication.IExternalApplication

which should be placed in the *META-INF/services* directory on the client side of the host ULC application. For example:

```
com.ulcjava.applicationintegration.sample.portalapp.IntegratedApp2
com.ulcjava.applicationintegration.sample.portalapp.IntegratedApp3
com.ulcjava.integration.javafx.WebApplication
```

In the above example *IntegratedApp2* and *IntegratedApp2* are external ULC applications and the *WebApplication* is a JavaFx application. All of them implement the *IExternalApplication* interface which specifies attributes of an External Application such as uniqueName, description, url to icons etc. These attributes are collected by *ULCExternalApplicationsManager* from the client side and made available to the host application on the server side such that the host application can know which External Applications are available for hosting and can present the list to the user to select.

```
ULCExternalApplicationsManager integratedApplicationManager =
                                new ULCExternalApplicationsManager();
integratedApplicationManager.addListener(
    new ExternalApplicationsMetaDataLoadedListener(){
    @Override
    public void externalApplicationsMetaDataLoaded(
        List<ExternalApplicationMetaData> externalApplicationsMetaDataList)
    {
       System.out.println("applicationsUpdated()");
       for (ExternalApplicationMetaData app :
                     externalApplicationsMetaDataList) {
            textArea.append(app.getName() + "\n");
       }
     }
});
integratedApplicationManager.loadExternalApplicationsMetaData();
```

2.2.2 ULCExternalApplicationContainer

To host an external application a ULC application should use *ULCExternalApplicationContainer* to hold the content of the external application. The external application can be a ULC, Swing, JavaFx, or a Web application running in JavaFx Webview – applications which have *JComponent* as their top most container.

There should be one *ULCExternalApplicationContainer* for each external application to be hosted/integrated. The External Application should implement *IExternalApplication* on the client side. For an External Application that is a ULC Application the top most container should be a *ULCExternalApplicationRootPane*.

Each External Application is identified by a unique name. When creating the *ULCExternalApplicationContainer* you pass the unique name of the External Application and optionally some properties that can be passed to the External Application when it is started from the host application. It is also possible to pass an *IOnExternalApplicationInitializedHandler* that is notified by the hosted external application when it is initialized.

Invoking the *startExternalApplication()* method of *ULCExternalApplicationContainer* class calls the *startExternalApplication* method of the *IExternalApplication* on the client side. The External Application starts and returns its root container (a *JComponent* instance) which is placed in the client side proxy of *ULCExternalApplicationContainer*. The *startExternalApplication* method of the External Application can then notify the *onExternalApplicationInitialized* method of the *ExternalApplicationContext*. Here is how you host an External Application in your ULC application:

1. Create *ULCExternalApplicationContainer*.
2. Pass the unique name of the External Application that will be hosted/contained in this container. The name of the external application must be the one specified in the *IExternalApplication* on the client side.
3. Pass the *HashMap* for startup time properties that you may want to pass to the External Application
4. Pass the handler that will be notified when External Application has been started on the client side. Also pass arguments for external application launcher and the external application.

```
IOnExternalApplicationInitializedHandler
    onExternalApplicationInitializedHandler =
        new IOnExternalApplicationInitializedHandler() {
          @Override
```

```
        public void onExternalApplicationInitialized
                        (ULCExternalApplicationContainer panel)
        {
           System.out.println(
             "ApplicationPanel Notified that Integrated App is
             Initialized");
       // Send An event to the external application
           panel.fireEvent("From Host", "Message for App1 ");
         }
      };
// Arguments to be passed to the ULC client launcher for external ULC
// application
Map<String, String> launcherArgs = new HashMap<>();
launcherArgs.put("keep-alive-interval", "99");

// properties to be passed to the external application when it is
// initialized on server. These properties can be accessed using
// ULCExternalApplicationRootPane.getStartProperties() in the external
//  application
Map<String, String> startArgs = new HashMap<>();
startArgs.put("startArg", "start");

ULCExternalApplicationContainer applicationPanel =
   new ULCExternalApplicationContainer("IntegratedApp1", launcherArgs,
                   startArgs, onExternalApplicationInitializedHandler);
```

5. Add a listener to listen to events fired by External Application.

```
applicationPanel.addExternalApplicationEventListener(
       new IExternalApplicationEventListener() {
       @Override
       public void onExternalApplicationEvent(
                           ExternalApplicationEvent event) {
              System.out.println("Host Received : " + event);
      }
 });
```

6. Add a handler to handle errors in the External Application

```
applicationPanel.addErrorHandler(new  IExternalApplicationErrorHandler()
{
   @Override
   public void onExternalApplicationError(
       ExternalApplicationError error) {
          String errMsg = error.getErrorType() + " error in " +
                       applicationName + " " + error.getErrorMessage();
           fStatus.append(errMsg);
       }
});
```

7. Upload the *ULCExternalApplicationContainer*

```
applicationPanel.upload();
```

8. Start the External Application. This will start the External Application from client side. The External Application's root container (must be a *JComponent*) will be placed in *UIExternalApplicationContainer* on the client side. When the External Application has started it will notify the host container that it has been initialized.

```
applicationPanel.startExternalApplication();
```

9. Place the *ULCExternalApplicationContainer* in a window of the host ULC application (like any other component)

```
frame.add(applicationPanel, ULCBorderLayoutPane.CENTER);
```

10. From the *ULCExternalApplicationContainer* events can be sent to the External Application

```
applicationPanel.sendEventToExternalApplication(
                                        "ToApp", "Message from Container");
```

2.3 Classes for an External (hosted) Application

This section describes the client and server side classes to be implemented by an External Application.

2.3.1 IExternalApplication

This is a client-side interface. Each External Application must have an implementation of *IExternalApplication* available on the class path of the ULC client of your host application. An External Application is identified by a unique name as specified in *IExternalApplication*:

```
public class IntegratedApp1Launcher implements IExternalApplication {
   public String getUniqueName() {
      return "IntegratedApp1";
   }
   ...
   public JComponent startExternalApplication(final
                                ExternalApplicationContext context) {
    // Launch the External Application and get its root container.
    // For an external ULC application it will be the client side
    // component of ULCExternalApplicationRootPane.
    // For a JavaFx application it should JFXPanel.
    // This will be placed in ULCExternalApplicationContainer's client proxy
    }
 }
```

All External Application's *IExternalApplication* implementations should be specified in the following file in *resources/META_INF/services* directory on the client side as:

```
com.ulcjava.ext.applicationintegration.client.externalapplication.IExternalApp
lication
```

which contains the respective *IExternalApplication* implementation class names:

```
com.ulcjava.integration.IntegratedApp1Launcher
```

ExternalUlcApplicationLauncher

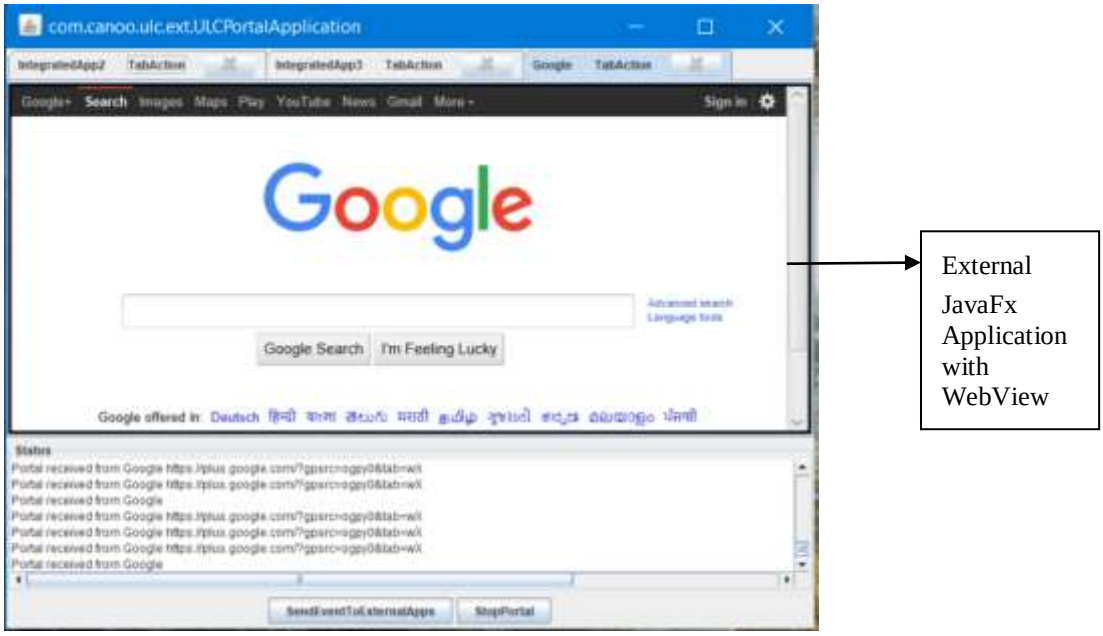
This is a default implementation of *IExternalApplication* to start an external ULC application. This class extends the *CustomizableStandaloneLauncher* which is the launcher of ULC client. You need to pass the URL of the external application to this class.

AbstractExternalJavaFXApplication

This is an implementation of *IExternalApplication* to start an external JavaFx applications.

JavaFXWebViewApplication

This is an implementation of *IExternalApplication* that extends *AbstractExternalJavaFXApplication*. It allows starting of an external JavaFx application that has a WebView component thus enabling running of a Web Application as an external application in a ULC application.



2.3.2 ULCExternalApplicationRootPane

This is a server side class for an External ULC Application. In an external application that is a ULC application, *ULCExternalApplicationRootPane* class must be used as the root container. Normally, a ULC application uses a *ULFrame* as its top most window. However, if you wish to host or run that ULC application as an external application in a host ULC application, then instead of using *ULCFrame* you should use

ULCExternalApplicationRootPane as the top most container for the application's UI components. The external application will be hosted in your ULC Application in a *ULCExternalApplicationContainer*.

Requirements for an External ULC Application :

- The client side jars of the External Application must be available on the class path of the ULC client of the ULC Application which is going to host those external applications.
- The ULC version used by the Host ULC Application and the External ULC Application must be the same.
- It is recommended that the Look and Feel used by the Host and all the External ULC Applications is the same.

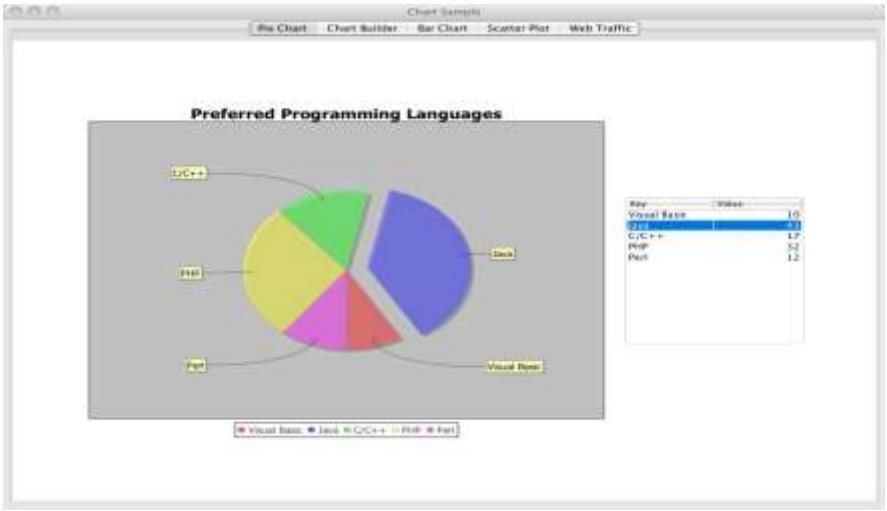
An example of an External ULC Application:

```
public class IntegratedApp2 extends AbstractApplication {
  @Override
  public void start() {

    // Create a listener that gets notified when External Application
    // has been started by the Host ULC Application
    IExternalApplicationInitializedHandler fOnInitializedHandler =
       new IExternalApplicationInitializedHandler() {
          public void onInitialized(
                    ULCExternalApplicationRootPane rootPane) {
             System.out.println("Integrated App : is Initialized.");
             integratedApp.sendEventToContainer(
             "From IntegratedApp", "IntegartedApp : I am Initialized.");
          }
     };

     // Create the root pane for the external application and pass the
     // InitializationHandler created above.

     final ULCExternalApplicationRootPane integratedApp =
                 new ULCExternalApplicationRootPane(fOnInitializedHandler);

     // Add a listener for events from the host
     // ULCExternalApplicationConatiner

     integratedApp.addContainerEventListener(
                 new ContainerToExternalApplicationEventHandler() {
         public void onContainerEvent(ExternalApplicationEvent event) {
             System.out.println("Integrated App Received : " + event);
         }
     });

     // create UI components of the External Application
     final ULCButton button = new ULCButton("Integrated App");
     button.addActionListener(new IActionListener() {
         public void actionPerformed(ActionEvent event) {
            // Send an event to the container that is running this
           // application as external application
```

```
            integratedApp.sendEventToContainer("From Integrated App",
                                                "Msg from Integrated App");
        }
    });

    // add the UI components of the External Application to the root pane.
    integratedApp.add(button, ULCBorderLayoutPane.CENTER);
  } // end of start
} // end of class
```

3 ULC Chart

The ULC Chart package provides support for charting in ULC. This package contains a basic integration of the well-known open-source JFreeChart library into ULC. This package enables the creation of various charts like pie chart, bar chart, histogram and scatter plot amongst others.



Unlike other ULC widgets, this ULC JFreeChart integration is meant to be extended on the client side to take advantage of the full JFreeChart API. For example, one can extend *UIAbstractChartPanel* class in order to create any kind of chart from the JFreeChart library.

For more information please have a look at JFreeChart API and ULC chart package Javadoc. In addition, you can download the sources of ULC Chart demo application from http://ulc.canoo.com/demos/index.html.

3.1 How to create a chart

To create a chart use *ULCChartPanel* and specify in the constructor the full path to the client-side UI half object like this:

```
private ULCChartPanel<IPieDataset<String, Integer>> createChartPanel() {
   ULCChartPanel<IPieDataset<String, Integer>> chartPanel =
            new    ULCChartPanel<IPieDataset<String, Integer>>
                           ("com.ulcjava.sample.chart.client.PieChartDemo");
   chartPanel.setDataset(createDefaultPieDataset());
   ULCChartSelectionModel chartSelectionModel =
                                      chartPanel.getChartSelectionModel();
   chartSelectionModel.setSelectionMode(
                                  ULCChartSelectionModel.MULTIPLE_SELECTION);
   chartPanel.setPreferredSize(new Dimension(600, 500));
   return chartPanel;
}
```

The *createPieDataset()* method shown below returns an *IDataSet* implementation that is suitable for the pie chart being created in the above snippet:

```
private static DefaultPieDataset<String, Integer> createDefaultPieDataset() {
       DefaultPieDataset<String, Integer> dataset =
                                    new DefaultPieDataset<String, Integer>();
       dataset.setValue("Visual Basic", 10);
```

```
        dataset.setValue("Java", 43);
        dataset.setValue("C/C++", 17);
        dataset.setValue("PHP", 32);
        dataset.setValue("Perl", 12);
        return dataset;
    }
```

The selection model that maintains the chart's selection state can be accessed with *ULCChartPanel*'s method *getChartSelectionModel()*.

On the client side, the UI half-object typically extends *UIAbstractChartPanel* like this:

```
/**
 * An {@link UIAbstractChartPanel} extension to make a <i>Pie Chart</i> demo.
 */
public class PieChartDemo extends UIAbstractChartPanel {

   private PiePlotSelectionRenderer createPlotSelectionRenderer(PiePlot plot)
   {
      …
   }

   private JFreeChart createFreeChart() {
      return ChartFactory.createPieChart("Preferred Programming Languages",
      getPieDataset(), true, true, false);
   }

   private PieDataset getPieDataset() {
      return (PieDataset)getBasicChartPanel().getDataset();
   }

   private void configurePiePlot(PiePlot piePlot) {
      …
   }

   @Override
   protected void postInitializeState() {
      super.postInitializeState();
      BasicChartPanel basicChartPanel = getBasicChartPanel();
      JFreeChart freeChart = createFreeChart();
      PiePlot piePlot = (PiePlot)freeChart.getPlot();
      basicChartPanel.setChart(freeChart);
      configurePiePlot(piePlot);

      basicChartPanel.setPlotSelectionRenderer(
                              createPlotSelectionRenderer(piePlot));
   }
}
```

Note that in the above snippets we created data on the server side while the chart itself was created from JFreeChart on the client side.

3.2 How to create a chart without having to write client code

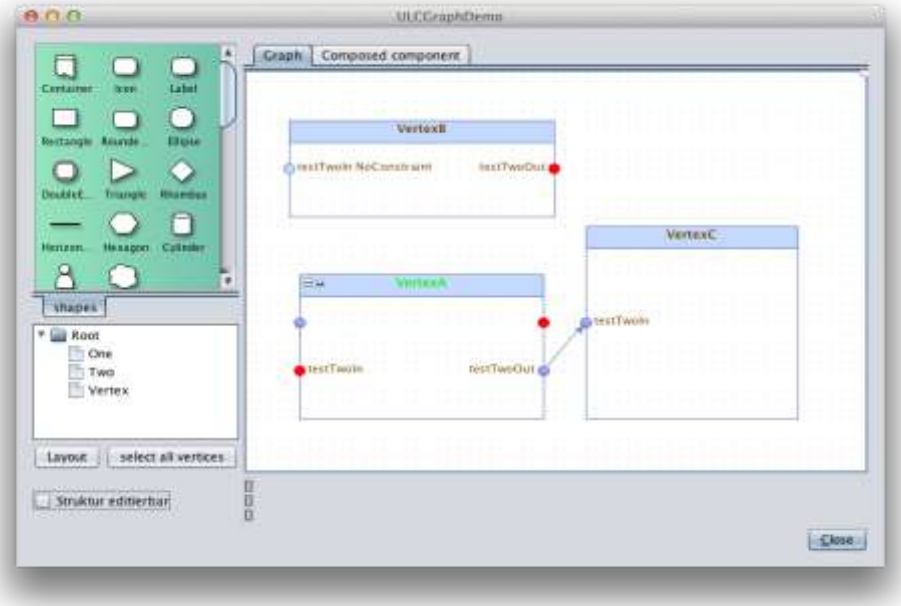
The easiest way to create a chart is to use *com.ulcjava.ext.chart.servert.ChartBuilder*. This class is a convenience class for making charts without any client code. However, you have to keep in mind that it is not possible to configure everything using it, i.e. you cannot access the full JFreeChart API when using a *ChartBuilder*.

```
private ULCChartPanel<IPieDataset<String,Integer>>
   createChartPanel(ChartBuilder chartBuilder) {
      ULCChartPanel<IPieDataset<String, Integer>> chartPanel =
          new ULCChartPanel<IPieDataset<String,
                                  Integer>>(createChartBuilder());
```

```
    chartPanel.setDataset(
                    PieChartSampleApplication.createDefaultPieDataset());
    ULCChartSelectionModel chartSelectionModel =
                                        chartPanel.getChartSelectionModel();
    chartSelectionModel.setSelectionMode(
                                ULCChartSelectionModel.MULTIPLE_SELECTION);
    chartPanel.setPreferredSize(new Dimension(600, 500));
     return chartPanel;
}

private ChartBuilder createChartBuilder() {
     ChartBuilder chartBuilder = new ChartBuilder();
     chartBuilder.setChartType(ChartType.PIE_CHART);
     return chartBuilder;
}
```

4 ULC Graph

The ULC Graph package supports creation and display of interactive diagrams and graphs. ULC Graph is based on *JGraph* (http://www.jgraph.com/) and extends it to be able to use it with ULC.



ULC Graph provides the following layout options:

- hierarchical layout for directional graphs, such as workflow and BPM.
- organic layout positions for diagrams like computer networks, mapping overlays and UML diagrams.
- tree layout for organizational charts

The following features are supported:

- Vertex
- Ports
- Edges
- Constraints
- Pallet with predefined shapes

```
ULCGraph graph = new ULCGraph();
 try {
      Vertex vertexA = new Vertex("Vertex A");
      vertexA.setTitle("Vertex A");
      vertexA.setStyle("Rectangle");
      vertexA.setRectangle(new Rectangle(50, 30, 100, 100));

      Vertex vertexB = new Vertex("Vertex B");
      vertexB.setTitle("Vertex B");
      vertexB.setStyle("Rectangle");
      vertexB.setRectangle(new Rectangle(50, 170, 100, 100));

      Port portA = new Port("Port A", PortType.OUT,
                            PortAlignment.BOTTOM, "",
                            "Port A");
      Port portB = new Port("Port B", PortType.IN, PortAlignment.TOP,
                            "", "Port B");
      vertexA.addPort(portA);
```

```
            vertexB.addPort(portB);

            graph.addVertex(vertexA);
            graph.addVertex(vertexB);

            Edge edge = new Edge("Edge A", portA, portB);
            graph.addEdge(edge);
        } catch (DuplicateIdException e) {
        } catch (InvalidStateException e) {
        } catch (InvalidEdgeException e) {
        }

        ULCGraphComponent graphComponent = new ULCGraphComponent(graph);
        return graphComponent;
```

For more information please have a look at ULC Graph package Javadoc. In addition, you can download the sources of ULC Graph demo application from http://ulc.canoo.com/demos/index.html.

5 ULC FxBrowser

The ULC FxBrowser package contains *class ULCFxBrowser* which provides an HTML rendering engine as a ULC component. *ULCFxBrowser* is based on the JavaFX WebView component. *ULCFxBrowser* provides the following features:

- Based on WebKit, an open source web browser engine.
- Render HTML content from local and remote URLs.
- Access to browser state (title, location, content, navigation state).
- Setting HTML-content of the browser.
- Listeners for hooking into browser events and state changes.
- Executing JavaScript.
- Calling server-side methods from within JavaScript

The Web Engine component is run asynchronously. Therefore, any JavaScript calls are queued until the browser page is loaded.

Access to the content of a website in the ULC application on the server-side can be quite expensive. By default, this state is not mirrored on the server side. Tracking this state can be switched on with *ULCFxBrowser.setContentTracking(true)*.

For more information please have a look at ULCFxBrowser package Javadoc and demos.

5.1 Development setup for ULC FxBrowser

The RIA-Suite project generator will do the required set up if you chose the ULC Fx-Browser package as one of the extensions while running the generator (see section 2.3).

In case you did not use the generator for creating your RIA-Suite application project, you need to include in the build path of your project the libraries from the following sub-directories of the directory **<Canoo RIA-Suite Install Dir>/ext/ULCFxBrowser/client and /server** and **jfxrt.jar from the JRE**.

5.2 How to create a simple browser

The main browser class is *ULCFxBrowser*. Just instantiate it and navigate to the desired URL. Following code snippet illustrates this:

```
ULCFxBrowser browser = new ULCFxBrowser();
browser.navigate("http://www.google.com/");
```

5.3 How to deploy a browser based application

The ULCFxBrowser requires *JavaFx runtime* which is part of Java 1.7. No additional third party libraries are needed. However, note that the *JavaFX jars* are not loaded by default, therefore *"javafx-runtime"* tag is needed in the deployment descriptor.

JNLP Deployment - sample jnlp file :

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" xmlns:jfx="http://javafx.com" codebase="$$codebase"
href="$$name">
```

```
  ...

 <resources>
 <j2se version="1.7.0_16+" />
 <jfx:javafx-runtime version="2.2+" />
 <jar href="ULCFxBrowser-client.jar" />
 ...<other client side jars>
 </resources>

 <application-desc main-
class="com.ulcjava.environment.jnlp.client.DefaultJnlpLauncher">
         <argument>url-string=$$context/ulc</argument>
     </application-desc>
 </jnlp>
```

Applet Deployment:

ULCFxBrowser is based on JavaFX and there is difference in the way the JavaFX applets are deployed from normal applets. The traditional way of defining applet tag will not work anymore. Two files are required for applet deployment as shown below:

1. **JNLP file :** Specifies the applet class and the required resources as described below:

   ULCFxBrowserDemoApplet.jnlp:

```
 <?xml version="1.0" encoding="utf-8"?>
 <jnlp spec="1.0+" xmlns:jfx="http://javafx.com"
codebase="$$codebase"   href="ULCFxBrowserDemoApplet.jnlp">
  ...

 <resources>
 <j2se version="1.7.0_16+" />
 <jfx:javafx-runtime version="2.2+" />
 <jar href="ULCFxBrowser-client.jar" />
 ...<other client side jars>
 </resources>

 <applet-desc   name="ULCFxBrowserDemo Applet"
 main-class=
"com.ulcjava.ext.fxbrowser.client.ULCFxBrowserAppletLauncher"
              width="800" height="600" >
       <param name="url-string" value="$$context/ulc"/>
       <param name="centerimage" value="true"/>
      <param name="boxorder" value="false"/>
</applet-desc>
</jnlp>
```

2. **JSP file** which contains the applet.

   ULCFxBrowserDemo.jsp which has the following contents:

   - *dtjava.js* javascipt is used to deploy the applet.
   - "*hideSplashScreen()*" javascript function is called by *ULCFxBrowserAppletLauncher"t*o hide the splash screen after loading the application.
   - Parameter '*url*' points to 'ULCFxBrowserDemoApplet.jnlp
   - Parameter 'placeholder' points to *div* tag 'ULCFxBrowserDemo-applet' the placeholder for the applet tag by the javascript.

For more details please see Deploying JavaFx Applications and How to deploy Swing applications in which JavaFX is embedded.

```
<%
    String appName="ULCFxBrowserDemo";
%>
<html>
....
<script type="text/javascript" src="http://java.com/js/dtjava.js"></script>
<script type="text/javascript">
function hideSplashScreen() {
 dtjava.hideSplash('<%=appName%>Applet')
}
function embedApp() {
 dtjava.embed(
{
  id: '<%=appName%>Applet',
  url: '<%=appName%>Applet.jnlp',
  placeholder: '<%=appName%>-applet',
  width : '100%',
  height : '100%'
},
{
  javafx : '2.2+',
  toolkit: 'swing'
},
{
 onGetSplash: (new dtjava.Callbacks()).onGetSplash
}
);
}
<!-- Embed Swing application into web page after page is loaded -->
 dtjava.addOnloadCallback(embedApp);
</script>
        <style type="text/css">
            html, body, #<%=appName%>-applet, #<%=appName%>Applet-app  {
                margin-bottom: 0px;
                margin-left: 0px;
                margin-right: 0px;
                margin-top: 0px;
                height: 100%;
                width: 100%;
                overflow: hidden;
            }
        </style>
    </head>
    <body>
        <div id="<%=appName%>-applet">
        </div>
    </body>
</html>
```

5.4 How to Call Server side methods from JavaScript

1. Create a class that has methods to be invoked from JavaScript:

```
public class JavaScriptToJavaTest {

 public void testInvokeULC () {
 }

 public void testInvokeULCWithParam(String param)
 }
}
```

2. Register the instance of above class as *InvocationTarget* with *ULCFxBrowser*:

```
ulcFxBrowser.addInvocationTarget(new JavaScriptToJavaTest())
```

3. Directly Invoke server-side method by executing a JavaScript on *ULCFxBrowser*:

```
ulcFxBrowser.setContent("<html><body>Test</body></html>");
ulcFxBrowser.executeScript("ulc.invokeULC('testInvokeULC');");
```

4. Directly Invoke server-side method with parameter by executing a JavaScript on *ULCFxBrowser*:

```
ulcFxBrowser.setContent("<html><body>Test</body></html>");
ulcFxBrowser.executeScript("ulc.invokeULCArgs(
       'testInvokeULCWithParam','test');");
```

5. Invoke server-side method by executing a JavaScript on *ULCFxBrowser* that sets the title property:

```
ulcFxBrowser.setContent("<html><body>Test</body></html>");
ulcFxBrowser.executeScript(
             "document.title='@invokeULC:testInvokeULC';");
```

6. You can customize the the JavaScriptToJava handling by extending the *UIFxBrowser* and registering a custom *JavaScriptToJavaHandler* :

```
public class MyUIFxBrowser extends UIFxBrowser{

   @Override
   protected Map<String, Object> createJavaScriptToJavaHandlers() {
      Map<String, Object> handlers =
                           super.createJavaScriptToJavaHandlers();
      handlers.put("myHandler", new CustomJavaHandler());
      return handlers;
    }
 }

 public class CustomJavaHandler{

    public void call(String methodname, String param) {
       SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                 invokeJavasciptToULCTargets(methodName, parameter);
            }
```

```
        });
    }
}
```

Call the custom handler's method

```
ulcFxBrowser.setContent("<html><body>Test</body></html>");
ulcFxBrowser.executeScript("myHandler.call('test');");
```

6 ULC Office Integration

This package enables ULC applications to generate downloadable content in Word, Excel (using Apache POI) and PDF(with iText 2.1.7 LGPL) formats. The package provides the base ground for creating basic documents, however it's extensible enough to let you build your own formatters, support other file formats even.

For more information please have a look at ULC Office Integration package Javadoc. In addition, you can download the sources of ULC Office Integration demo application from http://ulc.canoo.com/demos/index.html.

6.1 Development setup for ULC Office Integration

The ULC project generator will do the required set up if you selected the ULC Office Integration package as one of the Available Extensions (see section **Error! Reference source not found.**).

In case you did not use the generator for creating your ULC Application project, you need to include in the build path of your project the libraries from the following sub-directories of the directory **<Canoo RIA-Suite Install Dir>/ext/ULCOfficeIntegration**: **server**, and **server/third_party**.

6.2 How to use the ULC Office Integration

The are are two classes that make this package work as expected: *IResourceProvider* and *DownloadManager*. The former defines a resource available on the server while the latter contains the mechanism to manage resources and send them to the client. There are two types of *IResourceProvider*:

- *FileResourceProvider* – points to an existing file.
- *MemoryResourceProvider* – points to a resource available in memory only.

The ULC Office Integration package provides both types of resource providers for each of its supported formats.

There is one additional step you must follow before deploying an application to the server: you must configure a *ResourceDownloadServlet* on the deployment descriptor <your-project>/WebContent/WEB-INF/lib/web.xml. This servlet should point to the "/download/*" path, as seen in the following configuration

```
<servlet>
    <servlet-name>ResourceDownloadServlet</servlet-name>
    <servlet-class>
      com.ulcjava.ext.officeintegration.server.ResourceDownloadServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ResourceDownloadServlet</servlet-name>
    <url-pattern>/download/*</url-pattern>
</servlet-mapping>
```

6.3 How to use the ULC Office Integration (Word)

This package comes with a pair of classes (one for each mode of operation: file and in memory) that can generate a Word file out of a blank template. This means there will be

no special formatting applied to the document. The following example demonstrates how a resource can be created and then presented to the client for download

```
String text = … // initialized elsewhere
IResourceProvider resource = new SimpleWordFileProvider(text);
String handle = DownloadManager.getInstance().put(resource);
try {
    DownloadManager.getInstance().showDocument(handle);
} catch (IOException ioe) {
    LOG.sever("Could not show document with handle " + handle);
}
```

The *DownloadManager* will return a resource handle whenever a resource is registered with it. This way a resource can be downloaded several times without needing to create a new instance of it.

The package enables you to build your own *IResourceProvider* subclasses should the default formatting prove too inadequate for your needs. Two abstract classes can be used as a basis for Word documents: *AbstractWordFileResource* and *AbstractWordMemoryResource*.

6.4 How to use the ULC Office Integration (Excel)

This package comes with a pair of classes (one for each mode of operation: file and in memory) that can generate an Excel. Opposed to the Word support explained in the previous section, there are no concrete implementations for Excel documents, given that the data to be exposed may proceed from different sources and formats. Creating concrete implementations of Excel resources is a task left to the application developer.

There are two base abstract classes that can be used to create concrete implementations, similarly to Word support: *AbstractExcelFileProvider* and *AbstractExcelMemoryProvider*.

Once a concrete implementation is available, it can be registered with *DownloadManager* and used in the same way as explained in the Word section; that is, the following snippet could be a valid usage of JDBC aware Excel resource provider:

```
ResulSet rs = … // initialized elsewhere
IResourceProvider resource = new JDBCExcelFileProvider(rs);
String handle = DownloadManager.getInstance().put(resource);
try {
    DownloadManager.getInstance().showDocument(handle);
} catch (IOException ioe) {
    LOG.sever("Could not show document with handle " + handle);
}
```

Please take note that this package does not provide an implementation of *JDBCExcelFileProvider* as such class requires specific knowledge of the underlying data source and its schema. Application developers are tasked with create such provider if needed.

6.5 How to use the ULC Office Integration (PDF)

Lastly, the PDF integration classes work in the same way as the Word support explained in section 5.4.3, the main difference being that they create their output in PDF.

Application developers are highly encouraged to use the base abstract classes (*AbstractPdfFileResource* and *AbstractPdfMemoryResource*) in order to create their own PDF formatters.

6.6 How to print documents

The following classes facilitate printing functionality :

- *ULCPrintService* class enables printing on the client side. *PDF* and *Text* documents can printed directly without native application dependency. It opens the native print dialog. For other type of documents, printing is done by the native desktop printing facility provided by *java.awt.Desktop.* This uses the associated application's print command. Usage:

```
IResourceProvider resourceProvider=new AbstractPdfMemoryProvider() {
    protected void doWithDocument(Document doc) throws DocumentException {
        ... // Generate PDF content
    }
};
ULCPrintService.getInstance().pr int(resourceProvider, printStatusHandler);
```

- *ULCPDFPreviewPane* class is a *ULCComponent* that enables preview and print of *PDF* documents within a ULC Application. Usage :

```
ULCPDFPreviewPane fPreviewPane = new ULCPDFPreviewPane();
ULCBorderLayoutPane contentPane = new ULCBorderLayoutPane();
contentPane.add(fPreviewPane,ULCBorderLayoutPane.CENTER);
fPreviewPane.setTitle("sample.pdf");
fPreviewPane.show(resourceProvider);
```

7 ULC ServerPush and Distributed EventBus

The *ULC ServerPush* package provides *HTTP* based server push feature to *ULC* applications. The implementation is based on the concept of long polling.

An external data/event/message source such as *JMS*, distributed Event Bus like Hazelcast, background batch processes, etc., generate data/messages/events which a ULC application needs to pick up and update the *ULC* client without waiting for the next roundtrip from the *ULC* client.

A *ULC* client and a *ULC* application operate in request response mode. The client sends/initiates a request and the ULC application on the server updates the client. However, if the *ULC* application wants to update the *ULC* client it has to wait for the next roundtrip from the client. A *ULCPollingTimer* could be used by a *ULC* application to initiate roundtrips from the client. However, this has two drawbacks:

1. unnecessary or wasted round-trips when there are no updates on server.
2. the response time of the client to events on the server is limited by the frequency of polling.

ULCServerPush enables updates to the *ULC* Client as and when the data is available on the server side and most importantly without the client having to initiate a roundtrip on its own, for example through a *ULCPollingTimer*. Thus, *ULCServerPush* is a replacement to *ULCPollingTimer* when it comes to updating the *ULC* client.

The server push solution involves the following components: *ULCServerPush*, *UIServerPush* and *ULCServerPushServlet*.

7.1 How to use ULCServerPush

1. Instantiate *ULCServerPush* object and optionally specify the relative path of the *ULCServerPushServlet* **.**

```
ULCServerPush serverPush = new ULCServerPush("/serverPushServlet");
```

2. Register a listener for *ULCServerPushEvent*. This listener will update the ULC client.

```
class ChatEventListener implements IServerPushEventListener{
public void handleEvent(ULCServerPushEvent event){
// handle event here : for example take the data out of event
// and update the widget
String chatMessage = (String)event.getData();
          fChatTextArea.append(chatMessage);
      }
}
serverPush.addServerPushEventListener(new ChatEventListener());
```

3. Data/events from the external data/event source on the server side need to be pushed to the *ULCServerPush*. Typically processes running in the background wrap the external data/event in *ULCServerPushEvent* and push it to *ULCServerPush*.

   In the following example a *JMS MessageListener* pushes a message wrapped in

ULCServerPushEvent to *ULCServerPush* object.

```
class IncomingMessageHandler implements MessageListener {
   public void onMessage(Message message) {
       try {
          ChatMessage chatMessage =
              (ChatMessage)((ObjectMessage)message).getObject();
          serverPush.push(
             new ULCServerPushEvent(EVENT_MESSAGE_RECEIVED, chatMessage));
       } catch (JMSException e) {
          e.printStackTrace();
       }
   }
}
...
chatQueueConsumer.setMessageListener(new IncomingMessageHandler());
```

4. Configure the web.xml file to include the *ULCServerPushServlet* as follows:

```
<servlet>
   <servlet-name>ULCServerPushServlet</servlet-name>
   <servlet-class>
       com.ulcjava.ext.serverpush.server.ULCServerPushServlet
   </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>ULCServerPushServlet</servlet-name>
   <url-pattern>/serverPushServlet</url-pattern>
</servlet-mapping>
```

7.2 How does ULC ServerPush work

1. *ULCServerPush* specifies the URL of the *ULCServerPushServlet* to its *UIServerPush* proxy on the client.
2. *UIServerPush* on the client long polls the *ULCServerPushServlet* and waits for response.
3. *ULCServerPushServlet* detects whether any *ULCServerPushEvent* events have been pushed by the external data source to the *UIServerPush* object and returns response to *UIServerPush* on the client.
4. *UIServerPush* on the client initiates the ULC server roundtrip asynchronously to *ULCServerPush*.
5. *ULCServerPush* notifies the *ULCServerPushEvent* to the registered *IServerPushEventListener*s.
6. The *IServerPushEventListener*s then handles the event and update the ULC widgets in the application.

7.3 Distributed EventBus

ULC ServerPush has a class *UlcHazelcastEventBus* which provides integration with *Hazelcast* Distributed EventBus (see https://hazelcast.org/use-cases/messaging/). This enables :

a) An ULC application to publish events to the *Hazelcast* eventbus in order to communicate with external applications

b) An ULC application to subscribe to events published by external applications and push them to the ULC ServerPush.

Here is how to use the *UlcHazelcastEventBus*:

- There is only one instance of *UlcHazelcastEventBus* class per *ULCSession*.
- Create an instance of *IUlcHazelcastEventBusConfiguration* which has:
    - Specifications for *ULCServerPush* such as ULCServerPush servlet endpoint and long-polling interval
    - Specifications for *Hazelcast* such as *Hazelcast* server host name, port, group name, and connection attempt, period and timeout.

```
// Override methods of DefaultUlcHazelcastEventBusConfiguration
// to provide your own configuration
DefaultUlcHazelcastEventBusConfiguration configuration =
                        new DefaultUlcHazelcastEventBusConfiguration();
```

- Create and get an instance of *UlcHazelcastEventBus*:

```
// Creates Hazelcast Client instance and a ULCServerPush instance.
// Wires the instance such that messages on Hazelcast are pushed to
// ULCServerPush
UlcHazelcastEventBus.createInstance(configuration);

// Subsequently access the event bus using
UlcHazelcastEventBus.getInstance();
```

- After creating the event bus, start it to initiate server push

```
UlcHazelcastEventBus.getInstance().startEventBus();
```

- Define a common Topic in which the subscriber and publisher are both interested.

```
Topic<String> TOPIC1 = Topic.create("Topic1");
```

- The publisher publishes a MessageEvent for the Topic:

```
String data = "My Message";
DefaultMessageEvent message =
       new DefaultMessageEvent<String>(TOPIC1,
                              System.currentTimeMillis(), data);
message.addMetadata(MessageEventConstants.SENDER_ID,
    (String)ApplicationContext.getAttribute(ULCSession.ULC_SESSION_ID));
UlcHazelcastEventBus.getInstance().publish(TOPIC1, message);
```

- A subscriber subscribes to the Topic by registering a *MessageEventListener*. Based on the *MessageEventContext* in the message, the listener can decide if it wants to process or ignore the message:

```
UlcHazelcastEventBus.getInstance().subscribe(TOPIC1,
    new MessageEventListener<String>() {
    @Override
    public void onMessageEvent(MessageEvent<String> message) {
      MessageEventContext context = message.getMessageEventContext();
      String senderID =
       (String)context.getMetadata().get(MessageEventConstants.SENDER_ID);
      // don't process message if from same source
      if ( !senderID.equals( (String) ApplicationContext.getAttribute
          (ULCSession.ULC_SESSION_ID))) {
          textField.setText(textField.getText() + " : " +
                            "Frame1 Received : " +
                 message.getMessageEventContext().getTopic().getName()
                  + " : " + message.getData());
      }
    }
});
```

- A subscriber unsubscribes to the Topic by deregistering a MessageEventListener:

```
UlcHazelcastEventBus.getInstance().unsubscribe(TOPIC1,
                                                messageEventListener);
```

- Stop the event bus using to stop ULCServerPush and Hazelcast

```
UlcHazelcastEventBus.getInstance().stopEventBus();
```

- The *HazelcastClient* which is part of *UlcHazelcastEventBus* is configured with *EventStreamSerializer*. *EventStreamSerializer* serializes *MessageEvent* as a *JSON* so that external publishers and consumers on Hazelcast Event Bus can send/receive messages in *JSON* format to a ULC application.

8 ULC SpringIntegration

This package enables running of ULC applications as managed applications in Spring framework and Spring Boot. This way all the advantages of Spring framework such as dependency injection, declarative programming through aspects, and POJO programing can be made available to a ULC application. Integration with Spring Boot enables faster development and debug cycle as the application can be run in client-server mode without explicitly deploying it in an application server.

8.1 Modules of ULC SpringIntegration

There are five modules of ULC SpringIntegration:

- ULCSpringIntegration-server

  This module has classes that enable a ULC application class to be managed as a bean within the Spring framework. The annotation *UlcEntryPoint* that identifies a ULC application (*IApplication* implementation) as a Spring managed bean and its processors are part of this module. This module also has classes to configure and initialize ULC application servlets. Moreover, this module has classes that define scope for injectable Spring Components in a managed ULC application. There are 3 types of scopes:

  1. @SessionScope : this is Spring's HttpSession scope and the injected component will have only one instance per HttpSession.
  2. @ULCApplicationScope : the injected component will have only one instance across many instances of the ULC application running on server.
  3. @ULCSessionScope : the injected component will have only one instance within a ULCSession.

  The dependencies for this module are as follows:

```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>${ulc.group}</groupId>
        <artifactId>ulc-core-server</artifactId>
        <version>${ulc.version}</version>
    </dependency>
    <dependency>
        <groupId>${ulc.ext.group}</groupId>
        <artifactId>ULCServerPush-server</artifactId>
        <version>${ulc.serverpush.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.reflections</groupId>
        <artifactId>reflections</artifactId>
        <version>0.9.10</version>
    </dependency>
```

```
</dependencies>
```

- ULCSpringIntegration-boot-server

  This module has classes that enable configuring and running of a ULC Application as a managed application in Spring Boot. This way a ULC application can be run in client-server mode without explicit deployment in an application server.

  The dependencies for this module are as follows:

```
<dependencies>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>ULCSpringIntegration-boot-server</artifactId>
        <version>${project.version}</version>
    </dependency>
    <dependency>
        <groupId>commons-beanutils</groupId>
        <artifactId>commons-beanutils</artifactId>
        <version>1.8.3</version>
    </dependency>
 </dependencies>
```

- ULCSpringIntegration-client

  This module has class *UlcSpringApplicationClientLauncher* to launch ULC client that connects to the ULC application running in Spring framework.

  The dependencies for this module are as follows:

```
<dependencies>
       <dependency>
         <groupId>${ulc.group}</groupId>
         <artifactId>ulc-core-client</artifactId>
         <version>${ulc.version}</version>
     </dependency>
     <dependency>
         <groupId>${ulc.ext.group}</groupId>
         <artifactId>ULCServerPush-client</artifactId>
         <version>${ulc.serverpush.version}</version>
         <scope>runtime</scope>
     </dependency>
  </dependencies>
```

- ULCSpringIntegration-development

  This module has classes that enable a ULC application to run as a Spring managed application but in ULC's *DevelopmentRunner*. This enable ULC client and server to run in same JVM but at the same time as managed application in Spring.

  This module has the following dependencies:

```
<dependencies>
   <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>ULCSpringIntegration-server</artifactId>
      <version>${project.version}</version>
      <exclusions>
      <exclusion>
      <groupId>${ulc.group}</groupId>
```

```
        <artifactId>ulc-core-server</artifactId>
        </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>ULCSpringIntegration-client</artifactId>
        <version>${project.version}</version>
        <exclusions>
        <exclusion>
        <groupId>${ulc.group}</groupId>
        <artifactId>ulc-core-client</artifactId>
        </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>${ulc.group}</groupId>
        <artifactId>ulc-core-development</artifactId>
        <version>${ulc.version}</version>
    </dependency>
    <dependency>
          <groupId>javax.servlet</groupId>
          <artifactId>javax.servlet-api</artifactId>
          <version>3.0.1</version>
          <scope>runtime</scope>
      </dependency>
</dependencies>
```

- ULCSpringIntegration-project-archetype

  This module creates a Maven archetype that enables developers to create a Maven based project modules for a ULC Application that can be run in Spring Boot.

8.2 ULC application with Spring Boot

To run a ULC application as a managed application in Spring you simply have to annotate the application with :

com.ulcjava.ext.springintegration.server.managed.UlcEntryPoint and specify a unique name for the application as parameter to the annotation :

```
/**
 * Sample ULC Application which can run as managed application inside
 * Spring boot container
 */
@UlcEntryPoint("SampleULCApplication")
public class SampleULCApplication extends AbstractApplication {
    @Override
    public void start() {
    …
    }
}
```

The unique name of the application is used in the ULC application *URL* by a ULC client launcher when it connects to the ULC application on the server. ULC SpringIntegration provides a client side class to launch the ULC client and make it connect to a ULC application running in Spring. U*lcSpringApplicationClientLauncher* class is an extension of *CustomizableStandaloneLauncher* and is used as follows:

```
public class ClientStarter {
    public static void main(String[] args) {
       UlcSpringApplicationClientLauncher.startApplication(
                 "http://localhost:4085/ulc", "SampleULCApplication");
    }
}
```

The *pom.xml* for the ULC Spring Boot application project on server side has the following specifications:

```
<dependencies>
      <dependency>
         <groupId>${ulc.ext.group}</groupId>
         <artifactId>ULCSpringIntegration-boot-server</artifactId>
         <version>${ulc.springintegration.version}</version>
      </dependency>
      <dependency>
         <groupId>${ulc.group}</groupId>
         <artifactId>ulc-deployment-key</artifactId>
         <version>9.0</version>
      </dependency>
      <dependency>
         <groupId>${ulc.ext.group}</groupId>
         <artifactId>ULCApplicationIntegration-server</artifactId>
         <version>9.0</version>
      </dependency>
   </dependencies>
   <build>
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>1.2.2.RELEASE</version>
            <configuration>
            <mainClass>com.ulc.test.test_springi.ServerStart</mainClass>
            </configuration>
         </plugin>
      </plugins>
   </build>
```

The *main* class to start ULC Spring Boot application looks as follows:

```
@SpringBootApplication
@Import(UlcSpringBootConfiguration.class)
public class ServerStart extends SpringBootServletInitializer {

    protected SpringApplicationBuilder configure(
              SpringApplicationBuilder application) {
        return application.sources(ServerStart.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(ServerStart.class, args);
    }

   /**
    * If required, specify a HttpSessionListener
    */
   @Bean
    public ServletListenerRegistrationBean <HttpSessionListener>
       sessionListener() {
          return new ServletListenerRegistrationBean<HttpSessionListener>
                 (new MySessionListener());
    }
```

```
}
```

You need to specify *application.properties* file in *resources* directory as follows:

```
server.port=4085   // port for embedded Tomcat in Speing Boot
# Following are used when ULCSpringServletIntializer is initialized
ulc.endpoint.configure=true // use configuration from params
ulc.endpoint.register=true  // register ULC application servlet
ulc.push.endpoint.register=true // register ULC ServerPush servlet
server.session.timeout=1800
```

Also you need to specify the *ULCApplicationConfiguration.xml* in ***resources*** directory as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<ulc:ULCApplicationConfiguration
        xmlns:ulc="http://www.canoo.com/ulc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.canoo.com/ulc
ULCApplicationConfiguration.xsd ">
    <ulc:applicationClassName>
 com.ulcjava.ext.springintegration.server.managed.ManagedApplicationWrapper
    </ulc:applicationClassName>
    <ulc:java>
        <ulc:forceAllPermissions/>
    </ulc:java>
    <ulc:keepAliveInterval>300</ulc:keepAliveInterval>
    <ulc:clientLogLevel>FINE</ulc:clientLogLevel>
    <ulc:serverLogLevel>FINE</ulc:serverLogLevel>
</ulc:ULCApplicationConfiguration>
```

Note the **`ulc:applicationClassName`**. *ManagedApplicationWrapper* is the class that wraps the ULC application class (*IApplication*) to be managed by Spring. The class is identified in spring framework by the name parameter specified in *@UlcEntryPoint* annotation.

8.3 ULC application with Spring packaged in a *war* file

This section describes how you can deploy a ULC application integrated with Spring as a *war* file in an application server.

As far as the ULC application is concerned you just need to annotate it with *@UlcEntryPoint* as described in section 8.2. Also you can use the same U*lcSpringApplicationClientLauncher* class on the client side to launch the ULC client and connect it to the ULC application deployed in the application server.

```
public class ClientStarter_SampleAppsOnSpringServerTest {
   public static void main(String[] args) {
       UlcSpringApplicationClientLauncher.startApplication(
          "http://localhost:8080/ULCSpringApplication/ulc",
          "SampleApplication");
    }
}
```

On the server side you place the *ULCApplicationConfiguration.xml* in *resources* directory just as descibed in section 8.2. The name of the ULC application class in *ULCApplicationConfiguration.xml* should be *ManagedApplicationWrapper*.

The *web.xml* file should be placed in *src/main/webapp* directory. In *web.xml* file one can specify parameters for configuring ULC Spring Servlet as in addition to the normal specifications like the *servlet* and *servlet-mapping*:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   version="3.0">
   <context-param>
      <param-name>ulc.endpoint.configure</param-name>
      <param-value>false</param-value>
   </context-param>
   <!-- a user defined attribute that can be injected in ULC application-->
   <context-param>
      <param-name>attr.myAttribute</param-name>
      <param-value>ULCSpringIntegration-Sample</param-value>
   </context-param>
   <context-param>
      <param-name>ulc.springConfig</param-name>
      <param-value>
      com.ulcjava.ext.springintegration.server.test.MySpringConfiguration
      </param-value>
   </context-param>
   <context-param>
      <param-name>ulc.springbootstrap</param-name>
      <param-value>true</param-value>
   </context-param>
   <servlet>
      <description>ULC Servlet for Application1</description>
      <servlet-name>ServletContainerAdapter </servlet-name>
      <servlet-class>
         com.ulcjava.container.servlet.server.ServletContainerAdapter
      </servlet-class>
   </servlet>
   <servlet>
      <description>
         Easy deployment servlet
      </description>
      <servlet-name>EasyDeploymentServlet</servlet-name>
      <servlet-class>
         com.ulcjava.easydeployment.server.EasyDeploymentServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet>
      <servlet-name>ULCServerPushServlet</servlet-name>
      <servlet-class>
         com.ulcjava.ext.serverpush.server.ULCServerPushServlet
      </servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name> ServletContainerAdapter</servlet-name>
      <url-pattern>/ulc</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>EasyDeploymentServlet</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>ULCServerPushServlet</servlet-name>
      <url-pattern>/serverPushServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

On the server side you can also specify a configuration class to specify scan path for Spring components that can be instantiated your ULC application:

```
@Configuration
@ComponentScan({"com.ulcjava.ext.springintegration.server.test.common"})
public class MySpringConfiguration {
}
```

The *pom.xml* for the ULC Spring application project on server side has the following specifications:

```
<dependencies>
       <dependency>
          <groupId>${project.groupId}</groupId>
          <artifactId>ULCSpringIntegration-server</artifactId>
          <version>${project.version}</version>
       </dependency>
       <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
            <scope>compile</scope>
        </dependency>
       <dependency>
          <groupId>${ulc.group}</groupId>
          <artifactId>ulc-deployment-key</artifactId>
          <version>${ulc.license.version}</version>
       </dependency>
</dependencies>
<build>
       <plugins>
          <plugin>
             <groupId>org.apache.maven.plugins</groupId>
             <artifactId>maven-war-plugin</artifactId>
             <version>2.6</version>
             <configuration>
                <failOnMissingWebXml>false</failOnMissingWebXml>
             </configuration>
          </plugin>
       </plugins>
</build>
```

8.4 ULC application with Spring in *DevelopmentRunner*

To run a ULC application annotated with *@UlcEntryPoint* with ULC's *DevelopmentRunner* use the class *UlcSpringDevelopmentRunner* as follows :

```
@Configuration
/* Configuration to define scope of Scanning injectable components */
@Import(MySpringConfiguration.class)
public class SampleULCApplicationDevelopmentStart {

   public static void main(String[] args) {
              UlcSpringDevelopmentRunner.start(
          SampleULCApplicationDevelopmentStart.class,
          SampleULCApplication.class); // @ULCEntryPoint class
   }
}


/* Configuration to define scope of Scanning injectable/Spring
 * managed components
 */
```

```
@Configuration
/* Sepcify the package where spring managed components can be found */
/* In current example the components are in the server module */
@ComponentScan("com.ulc.test.test_springi;")
public class MySpringConfiguration {
}
```

The *pom.xml* for the ULC Spring Development project on server side has the following dependency specifications:

```
<dependencies>
    <dependency>
        <groupId>${ulc.ext.group}</groupId>
        <artifactId>ULCSpringIntegration-development</artifactId>
        <version>${ulc.springintegration.version}</version>
    </dependency>
</dependencies>
```

Make sure to add dependencies such that other client side and server side jars of your application are available on the classpath.

8.5 Creating a ULC Spring Boot project

ULC Spring Integration includes *ULCSpringIntegration-project-archetype* that enables you to create a ULC Spring Boot template project. The project has modules: client, development and server.

You can install *ULCSpringIntegration-project-archetype* jar file *ULCSpringIntegration-project-archetype-<version>.jar* that is provided with the RIA-Suite release in your local maven repository using the Maven command : *mvn install:install*.

Once the archetype is installed, you can generate a sample project using the Maven command :

>mvn archetype:generate

-DarchetypeCatalog=local

-DarchetypeGroupId=com.canoo.ulc.ext

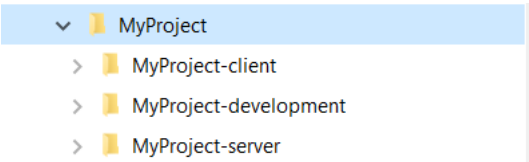
-DarchetypeArtifactId=ULCSpringIntegration-project-archetype

-DarchetypeVersion=9.0

-DgroupId=com.myproject

-DartifactId=MyProject

The above command will generate the project as follows:



MyProject is the main maven project with 3 modules : client, development and server.

The projects have sources and *pom.xml*s for a sample ULC application with Spring Boot integration.