

ULC

Application

Development

Guide

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

Copyright 2000-2010 Canoo Engineering AG
All Rights Reserved.

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

Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc. All other trademarks referenced herein are trademarks or registered trademarks of their respective holders.

Contents

1	Overview	8
2	Getting started tutorial	9
2.1	Install Canoo RIA Suite	9
2.2	Generate an application skeleton.....	10
2.3	Extend application skeleton with a bean	13
2.4	Change the Swing Look & Feel.....	15
2.5	Change the application logo.....	16
2.6	Change table column ordering	17
2.7	Add business logic to the tool bar	17
2.8	Change form layout.....	19
2.9	Add validation to the form	20
2.10	Deployment.....	20
3	Set up	22
3.1	How to install Canoo RIA Suite	22
3.2	About licenses.....	22
3.2.1	License key location	22
3.2.2	Using the ULC license manager to manage ULC licenses.....	22
3.2.3	How to upgrade to new a version of Canoo RIA Suite	23
3.2.4	Deployment license	24
3.3	How to setup an Eclipse project.....	24
3.3.1	As an Eclipse WST project.....	24
3.3.2	As a simple Java project	24
3.4	How to create and build an application skeleton	25
3.5	How to run sample applications	25
4	Application Development	27
4.1	Development Environment.....	27
4.1.1	How to run ULC applications in Development Environment.....	27
4.1.2	How to configure DevelopmentRunner.....	27
4.1.3	Graphical User Interface for Development Environment	31
4.2	Application Main Classes	32
4.2.1	How to implement a ULC Application	32
4.2.2	IApplication.....	33
4.2.3	AbstractApplication.....	34
4.2.4	ApplicationContext	34
4.2.5	ApplicationFramework's Application and ApplicationContext ...	35
4.2.6	ApplicationFramework's SingleFrameApplication	35

4.3	Main windows	36
4.3.1	How to define a main window.....	36
4.3.2	How to access the main window	38
4.3.3	How to add an Action to the main window.....	38
4.3.4	How to enable / disable Actions	39
4.3.5	How to give early startup feedback	40
4.3.6	How to show popup windows and message alerts.....	40
4.4	Layout Containers.....	40
4.4.1	Various layout options in ULC	40
4.4.2	How to use ULCBoxPane.....	41
4.5	Components.....	45
4.6	How to format and validate displayed/entered values	45
4.6.1	How to visualize formatting and validation errors	46
4.7	How to enable / disable components	46
4.8	Events and Event Delivery Modes	46
4.8.1	Event Delivery Modes	47
4.8.2	Model Update Modes	47
4.9	Special features of ULCTable	48
4.9.1	How to define a table based on a list of Java beans	48
4.9.2	How to enable table sorting.....	49
4.9.3	How to enable table filtering.....	49
4.10	Forms.....	50
4.10.1	How to bind a form to a bean	50
4.10.2	How to define the form layout.....	51
4.10.3	How to embed a form in a window	52
4.10.4	How to add validation code to a form	52
4.11	Resource bundles	53
4.11.1	How to internationalize	54
4.11.2	How to use images.....	54
4.11.3	How to use more than one resource bundle.....	54
4.11.4	How to define styles	55
4.12	Client Access	56
4.12.1	How to install a File Service	57
4.12.2	Choosing a File Located on the Client Machine	57
4.12.3	Reading from a File Located on the Client Machine.....	58
4.12.4	Writing to a File Located on the Client Machine	59
4.12.5	Showing a Document in a Web Browser	60
4.12.6	Accessing the Client Environment.....	61
4.12.7	Accessing the Client's UI Defaults	61
4.12.8	How to send messages from server to client.....	62
4.13	Container Access.....	63
4.13.1	Servlet Container Access	63
4.13.2	EJB Container Access.....	64
4.14	Drag & Drop	64
4.14.1	Drag & Drop Operation Overview.....	65
4.14.2	Configuring DnD.....	65

4.14.3	Transfer Data	66
4.14.4	DataFlavor.....	66
4.14.5	IDnDData	66
4.14.6	TransferHandler	67
4.15	Creating Charts.....	68
4.15.1	How to create a chart	69
4.15.2	How to create a chart without having to write client code.....	70
4.16	Creating Animation Effects.....	70
4.16.1	How to create a simple animation	71
4.16.2	How to use multiple targets and animation triggers.....	72
4.16.3	How to use custom TimingTargets	72
4.17	Creating extended Visual Effects	73
4.17.1	How make a component translucent	73
4.17.2	How to paint gradient and textured backgrounds	73
4.17.3	How to create gradient paints.....	74
4.17.4	How to create image paints.....	74
4.17.5	How to display a component with rounded corners	75
4.18	Best Practices	77
4.18.1	Use callback functions.....	77
4.18.2	Avoid using static variables	77
4.18.3	Avoid sharing proxies between sessions.....	77
4.18.4	Components cannot have multiple parents	77
4.18.5	Share objects whenever possible.....	77
4.18.6	Use uniform renderer components	77
4.18.7	Avoid asynchronous event handling.....	78
4.18.8	Thread-Safety	78
4.18.9	Serializable session.....	78
4.18.10	Session termination.....	78
4.18.11	Reuse components	78
4.18.12	Simulate Low Bandwidth	78
4.18.13	Use the GUI Provided with DevelopmentRunner	79
4.18.14	Check reachability through firewalls and proxies	79
4.18.15	How to code long tasks	79
4.18.16	How to implement server push functionality	80
4.18.17	Handling database connection pooling.....	80
4.18.18	How to call native code on the client	81
4.18.19	Memory Management	81
4.19	How to extend ULC	82
4.19.1	How to display a customized dialog on application failure.....	82
4.19.2	How to customize user feedback during server roundtrip.....	83
4.20	How to customize the failure behavior of servlet connector	84
4.20.1	Customize all connector commands.....	84
4.20.2	Customize a specific connector command	85

5 Add-on packages 86

5.1	Using ULC Web Integration	86
5.1.1	Development setup for ULCBrowser	86

5.1.2	How to create a simple browser	86
5.1.3	How to deploy a browser based application	87
5.2	ULC Table Plus for creating complex tables	88
5.2.1	Development setup for ULCTableScrollPane	88
5.2.2	How to use the ULCTableScrollPane	89
5.2.3	How to create multi-line headers for a table	90
5.2.4	How to create AutoFilterTableHeaders for a table	91
5.3	ULC Enterprise Portal-Integration	92
5.3.1	How to use the ULC Enterprise Portal-Integration	92
5.3.2	How to integrate multiple applets with distinct sessions	93
5.3.3	How to use user parameters to pass data to the ULC applet ...	94
5.4	ULC Office Integration.....	94
5.4.1	Development setup for ULC Office Integration.....	94
5.4.2	How to use the ULC Office Integration	94
5.4.3	How to use the ULC Office Integration (Word)	95
5.4.4	How to use the ULC Office Integration (Excel)	95
5.4.5	How to use the ULC Office Integration (PDF)	95
6	Debugging and Testing	97
6.1	How to enable logging.....	97
6.1.1	Logging Classes.....	97
6.1.2	Configuring the Client-Side (UI Engine) Log Manager	97
6.1.3	Configuring the Server-Side (ULC) Log Manager	98
6.1.4	How to analyze the client-server communication	99
6.1.5	How to setup a debugger	101
6.1.6	How to setup a profiler	101
6.2	How to write unit tests	101
6.3	How to perform load tests	101
7	Deployment	102
7.1	EasyDeployment.....	102
7.2	How to use EasyDeployment without using generator	103
7.3	How to run your application inside a browser.....	103
7.4	How to integrate the applet into your own application	105
7.5	How to run your application outside a browser	105
7.6	How to integrate into the client environment	106
7.7	How to secure ULC applications	106
7.8	How to sign the application's libraries with your certificate.....	106
7.9	How to handle HTTP session timeout	107
7.10	How to do Authentication and Authorization	107
7.11	Deploying a ULC application in standalone/offline mode	107
7.12	Google App Deployment.....	107

8	The ULC Application Configuration	110
8.1	ULCApplicationConfiguration.xml	110
8.2	The Application Properties file	114

1 Overview

Canoo RIA Suite is based on the ULC technology and provides components for building Rich Internet Applications (RIA) in Java. Use this standard Java based library to develop rich, responsive graphical user interfaces (GUIs) for enterprise web applications within Java EE and Java SE infrastructures.

To understand the concepts behind ULC please refer to the [ULC Architecture Guide](#). For detailed information on ULC API please see [ULC API](#) documentation. The [ULC Deployment Guide](#) describes various modes of deploying the ULC client on the desktop and a ULC application on the sever. Finally, the [ULC Extension Guide](#) discusses how the ULC widget set can be extended.

This document focuses on the application developer and covers the whole development lifecycle ranging from installation, setup, coding, debugging, and testing to deployment.

Organization

This document is organized into the following chapters:

Chapter 2 helps you to get started with ULC by stepping you through a tutorial.

Chapter 3 describes how to install and set up ULC in your IDE.

Chapter 4 describes how to develop applications using ULC.

Chapter 5 describes features provided by add-on packages.

Chapter 6 describes how to debug and test a ULC application.

Chapter 7 describes how to deploy a ULC application.

Chapter 8 describes the various ways to configure a ULC application.

2 Getting started tutorial

The tutorial in this chapter covers more than the traditional “Hello World” application. After completing the tutorial you will have a ready to use end-to-end application that manages beans in a database.

You will need Eclipse to work through the tutorial. Eclipse can be downloaded from <http://www.eclipse.org/downloads/>. Please make sure that you download the **package for Java EE developers**. The tutorial cannot be executed with package for simple Java developers. Moreover, you will require Java **JRE version 1.5** or higher.

2.1 Install Canoo RIA Suite

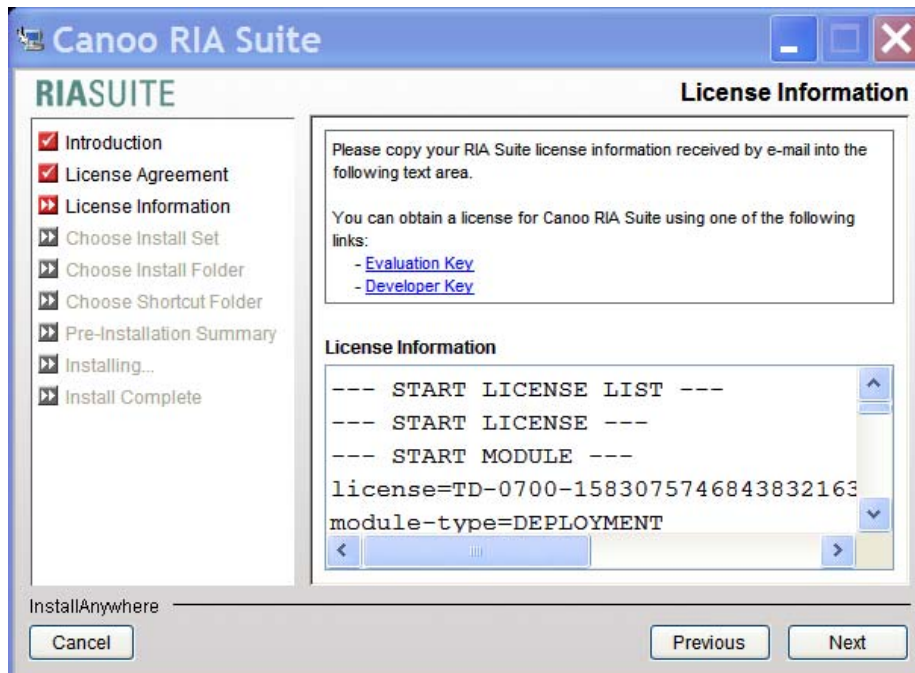
In this step you will install Canoo RIA Suite onto your computer.

To use ULC you will first need a valid license key. There are two kinds of license keys: evaluation license keys and developer license keys. Evaluation license keys are valid for 30 days and you can get one from [Canoo Customer Portal](http://www.canoo.com/kurt)

Just register with the portal and fill in the form on the website to order evaluation key.

Upon submitting your evaluation license key request, you will receive an email from Canoo that contains your evaluation license key. With this evaluation license key you are now ready to download and run the latest ULC installer from <http://www.canoo.com/kurt>.

The installer will guide you through the installation steps. When you are asked to enter the license key, copy the text from ---START LICENSE LIST--- to ---END LICENSE LIST--- within the license key (that you received by mail) and paste it into the installer.

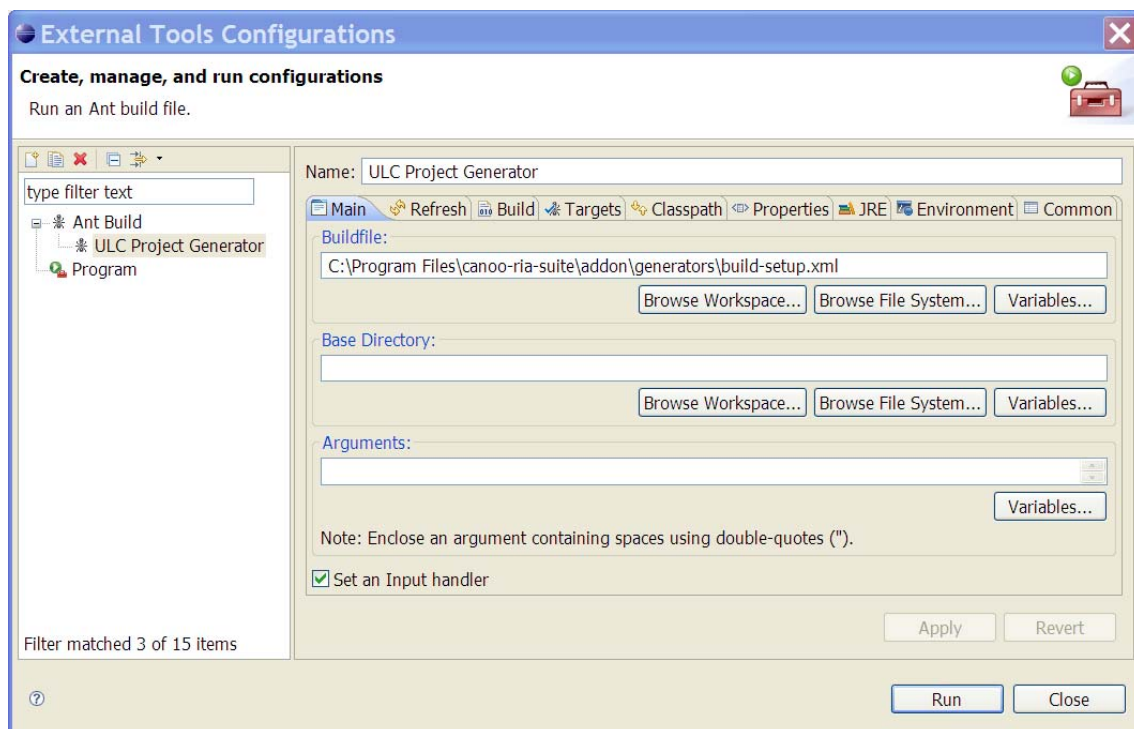


When the installer finishes, you have installed Canoo RIA Suite on your computer. The installer will create the required license files in the *<User Home Directory>/ulc-<version>* directory. This directory will contain development and deployment licenses for ULC Core and any other extension packages whose license keys were contained in the pasted text.

2.2 Generate an application skeleton

Now that you have successfully installed Canoo RIA Suite, you are ready to generate a ULC application skeleton.

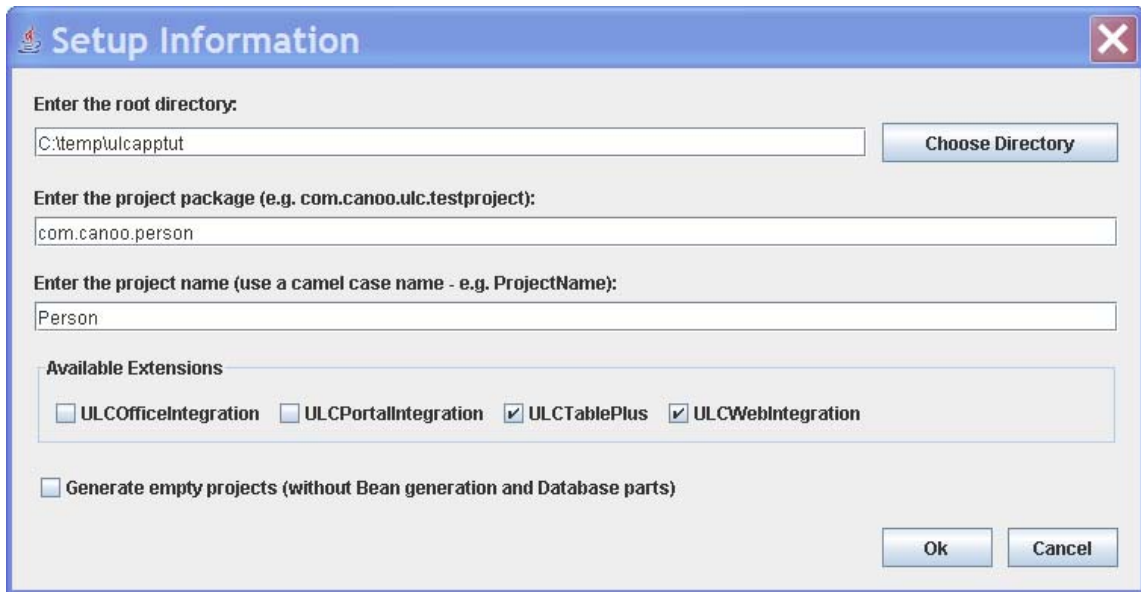
Start Eclipse and configure the ULC project generator. To do so, select the **Run > External Tools > Open External Tools Dialog...** menu item. Select **Ant Build** in the tree on the left of the external tools dialog. Click the leftmost button in the tool bar on top of the tree to create a new launch configuration. Enter a meaningful name for the new launch configuration - **ULC Project Generator** - and click **Browse File System...** to choose the **<Canoo RIA Suite Install Dir>/addon/generators/build-setup.xml** build file from your ULC installation.



Select the **Common** tab and check the **External Tools** check box in the **Display in favorites menu** section of the dialog. Click **Apply** to save the launch configuration and click **Close** to close the external tools dialog. The ULC project generator is now configured and you are ready to start it. Select the **Run > External Tools > ULC Project Generator...** menu item to start the ULC project generator. The project generator opens a dialog and asks you to enter the following:

- **Root directory:** Directory into which the generator generates the application skeleton. Later we will import the generated application skeleton into your Eclipse workspace.
- **Project package name:** Identifies the package below which the generator generates all classes.
- **Project name:** Name which the generator uses to build Eclipse project names.
- **Available Extensions:** Choose the extension packages you wish to use in your ULC application in addition to ULC Core. The generator will then include the required libraries for the selected packages in the build path of the generated projects

- **Generate empty project:** You have the option to generate a project with or without the ability to generate Bean and Database related code.



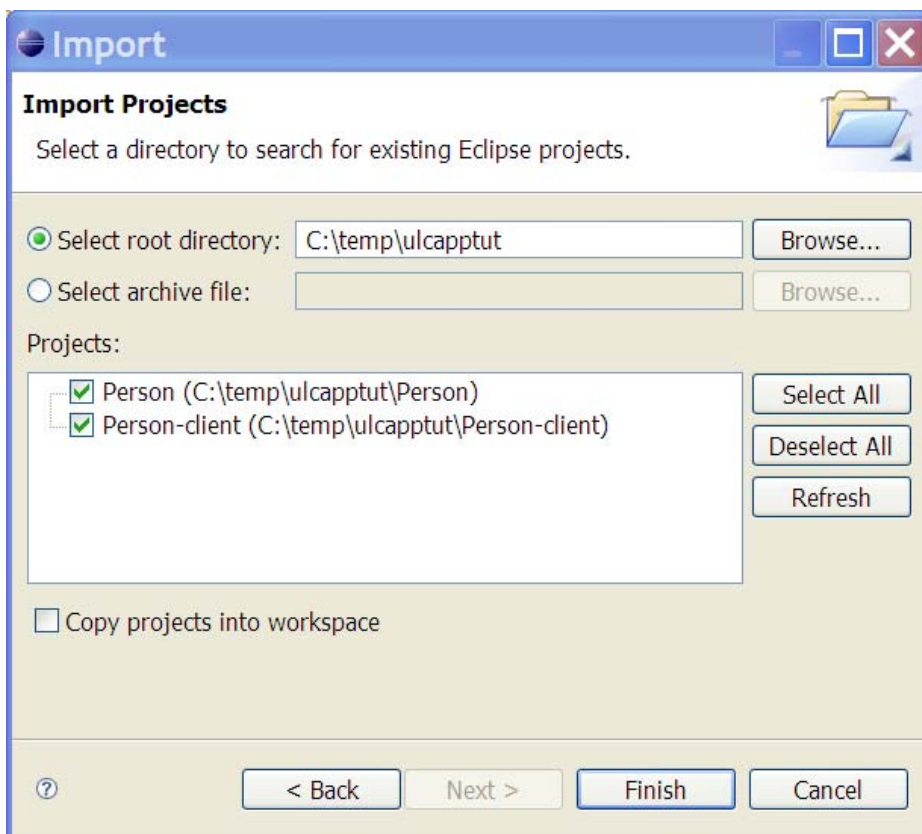
The 'Setup Information' dialog box is used to configure the project generation. It contains the following fields and options:

- Enter the root directory:** A text field with 'C:\temp\ulcapptut' and a 'Choose Directory' button.
- Enter the project package (e.g. com.canoo.ulc.testproject):** A text field with 'com.canoo.person'.
- Enter the project name (use a camel case name - e.g. ProjectName):** A text field with 'Person'.
- Available Extensions:** A group box containing four checkboxes: 'ULCOfficeIntegration' (unchecked), 'ULCPortalIntegration' (unchecked), 'ULCTablePlus' (checked), and 'ULCWebIntegration' (checked).
- Generate empty projects (without Bean generation and Database parts):** An unchecked checkbox.
- Buttons:** 'Ok' and 'Cancel' at the bottom right.

Click **Ok** to close the dialog and to generate the Eclipse projects.

The project generator generates two projects one for the client-side code (**Person-client**) and one for the server-side code (**Person**). Normally you will code your ULC application in the server-side **Person** project. However, in case there is a need to extend ULC widgets such that it requires some client-side code, then such code will have to be part of the **Person-client** project.

The next step is to import the generated Eclipse projects into your Eclipse workspace.



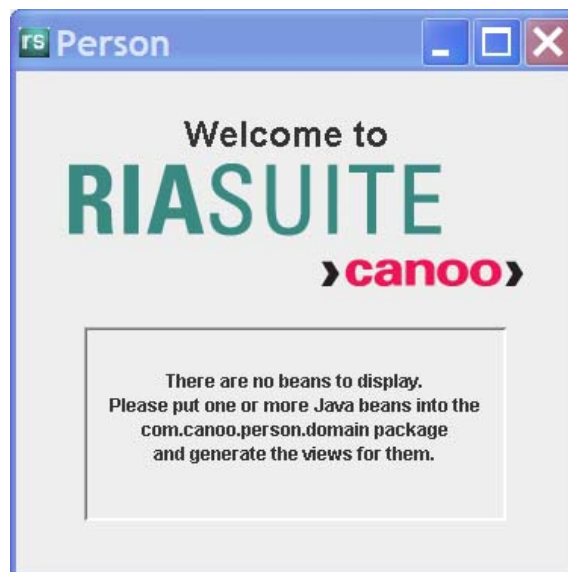
The 'Import Projects' dialog box is used to import existing Eclipse projects. It contains the following fields and options:

- Import Projects:** A title bar with a folder icon.
- Select a directory to search for existing Eclipse projects.** A text field with 'C:\temp\ulcapptut' and a 'Browse...' button.
- Select root directory:** A radio button (selected) next to the text field 'C:\temp\ulcapptut' and a 'Browse...' button.
- Select archive file:** A radio button (unselected) next to an empty text field and a 'Browse...' button.
- Projects:** A list box containing two items: 'Person (C:\temp\ulcapptut\Person)' and 'Person-client (C:\temp\ulcapptut\Person-client)', both with checked checkboxes. To the right of the list box are buttons: 'Select All', 'Deselect All', and 'Refresh'.
- Copy projects into workspace:** An unchecked checkbox.
- Buttons:** '< Back', 'Next >', 'Finish', and 'Cancel' at the bottom.

Select the **File > Import...** menu item to open the import wizard. Select **General > Existing Projects into Workspace** in the tree and click **Next**. Enter the root directory you provided to the ULC project-generator in the previous step as root directory for the import wizard. Type tab to remove focus from the **Select root directory** text field. As a result you should now see two Eclipse projects in the **Projects** list, one for the ULC client-side code (**Person-client**) and one for the server-side, ULC application code (**Person**). Check both projects. Make sure that the **Copy projects into workspace** check box is not checked. Click **Finish** to finally import the projects into your Eclipse workspace.

Have a look at the **Person** and **Person-client** projects in the workspace. The **Person** project has the skeletal ULC application **com.canoo.person.PersonApplication** and the ULC application configuration file **ULCApplicationConfiguration.xml**. It also has a class **com.canoo.person.extension.ULCBlinkingButton** which extends **ULCButton**. The **Person-client** project has the client side code for **ULCBlinkingButton**, namely **com.canoo.person.extension.UIBlinkingButton**. How to extend ULC is described in chapter 4. At this stage in the tutorial you don't need to worry about extending ULC.

The generated application skeleton is now part of your Eclipse workspace and is ready to run. Select the **Run > Run History > Person** menu item to run the application skeleton. This displays a welcome dialog that informs you that you successfully generated the application skeleton and that you are now ready to extend the application skeleton with a bean. You can learn more about this in the next section.



2.3 Extend application skeleton with a bean

In this section you will extend the application skeleton with a bean. The extended application skeleton will allow you to create new bean instances and search for, modify and delete existing ones. And as a bonus everything is persisted in a database.

First step is to define the bean in the **com.canoo.person.domain** package in the **Person** project. For example create the following bean in that package:

```
package com.canoo.person.domain;

import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;

    private String firstName;
    private String lastName;
    private Date dateOfBirth;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(Date dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }
}
```

The things that make this bean special are the *javax.persistence* annotations. These annotations are important for the persistence framework to work:

- **@Entity**: means that the bean should be persisted.
- **@Id**: means that this property identifies an instance, i.e. it will hold the primary key.
- **@GeneratedValue**: means that the value of this property will be generated by the persistence framework.

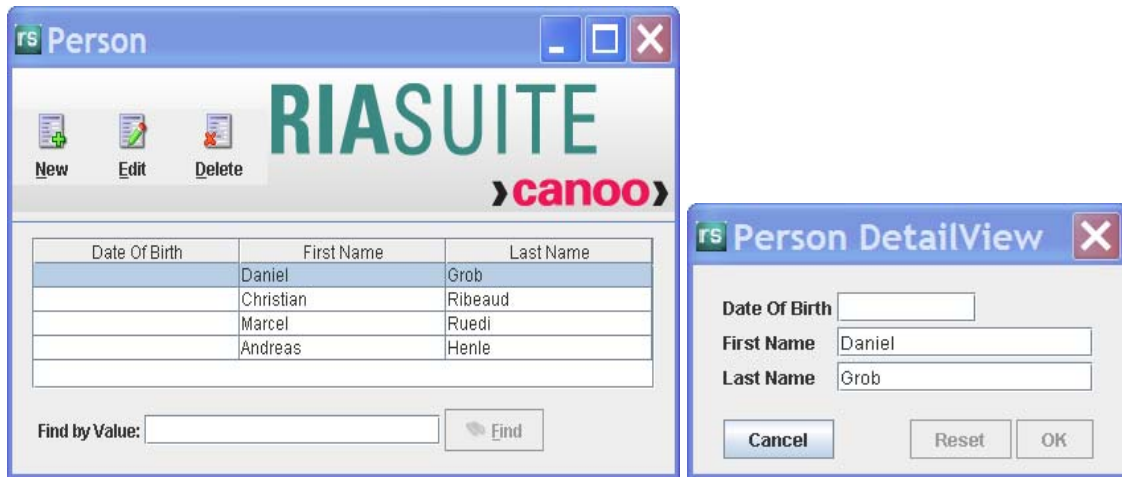
The persistence layer of the application skeleton is based on the Java Persistence API. We won't cover all aspects of the persistence framework in this tutorial. If needed, you can find more information about the annotations and the persistence framework in general at java persistence site <http://java.sun.com/javaee/technologies/persistence.jsp>.

After defining and annotating the bean you can now generate bean views and persistence binding using the **generator.xml Ant build script**. For locating the ULC resources a **ulc.home** property is specified in the **generator.properties** file. This property may have to be adapted if the generator is used on different environments

To create bean views and persistence bindings, Select the **Run > External Tools > Person generate-bean-views** menu item. After the bean views generator has terminated, you will find the following files in your project:

com.canoo.person	for the application
PersonApplication.java	the application class
resources/PersonApplication.properties	resource bundle for application class
resources/icons/*	icons for application class
src/ULCApplicationConfiguration.xml	The configuration file for ULC application
src/ULCApplicationConfiguration.xsd	The XML schema for ULC configuration
com.canoo.person.view	for views
PersonFormModel.java	model for form
PersonForm.java	builder that builds a form for your bean
PersonDetailView.java	detail dialog containing form
PersonMasterView.java	main window part for your bean, i.e. table
resources/PersonFormModel.properties	resource bundle for form model
resources/PersonForm.properties	resource bundle for form
resources/PersonMasterView.properties	resource bundle for main window part
com.canoo.person.domain	for domain classes
Person.java	your bean
com.canoo.person.persistence	for persistency / services
PersonDAO.java	database access for your bean
test/com.canoo.person	for test
PersonMasterViewTest.java	a test class for the application main window
src/META-INF	for the JPA configuration
persistence.xml	configuration file for the JPA based persistence framework

To run the application, select the **Run > Run History > Person** menu item. The welcome message that we saw in the previous run has been replaced by a bean management application that supports all relevant operations. The main view of the bean management application contains a tool bar, a table, and a search field. Using the actions in the tool bar, you can create, modify, and delete beans. The table displays all existing beans and the search field at the bottom can be used to locate beans in the table.



The generation of the bean view and the persistence binding can be executed for any new bean that is added to the project. By default the already generated view and persistence code is not changed. To regenerate all code property **overwriteExistingBeanViews** in **generator.xml** should be set to true.

2.4 Change the Swing Look & Feel

In this step you will improve the general look of the application.

ULC uses a central configuration file to manage several application wide settings. One of these settings affects the installed Swing Look & Feel which is used to render the user interface. Simply change the Swing Look & Feel to improve the general look of the application.

Open the **ULCApplicationConfiguration.xml** file inside Eclipse. Place the caret after the **ulc:applicationClassName** element and use code completion to add a **ulc:lookAndFeel** element. Use code completion again to add an embedded **ulc:lookAndFeelClassName** element that holds the class name of a Swing Look & Feel, e.g. **org.jvnet.substance.skin.SubstanceBusinessBlueSteelLookAndFeel**.

The resulting configuration file should look 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.canoo.person.PersonApplication
  </ulc:applicationClassName>
  <ulc:lookAndFeel>
    <ulc:lookAndFeelClassName>
      org.jvnet.substance.skin.SubstanceBusinessBlueSteelLookAndFeel
    </ulc:lookAndFeelClassName>
  </ulc:lookAndFeel>
</ulc:ULCApplicationConfiguration>
```

Select again the **Run > Run History > Person** menu item to start the application. With one simple configuration option you have dramatically changed the general look of the application!



2.5 Change the application logo

In this step you will change the logo displayed in the upper right corner of your application.

ULC supports resource bundles to externalize resources from the source code, e.g. texts, icons, key strokes, etc. Resource bundles are locale aware text files containing key value pairs. So changing resources means changing text files instead of changing and compiling source code.

Among various other things, the generated application reads the application icon from such a resource bundle. Open the **PersonApplication.properties** file inside the **com.canoo.person.resources** package in Eclipse. Search the **logo.Label.icon** key and change its value from **ria_suite.png** to **canoo.png**.

The resource bundle file should now look as follows:

```
Application.title=Person
...
logo.Label.icon=${resource.images.path}/canoo.png
...
```

Select the **Run > Run History > Person** menu item to start the application and see the effect.



2.6 Change table column ordering

In this step you will change the ordering of the table columns in the main view.

The generator generates code that adds a table column for each property in your bean and orders these table columns randomly inside the table. This is the best that the generator can do but is usually not what you want for application. However, it is easy to change this generated code.

Open the **com.canoo.person.view.PersonMasterView** class inside Eclipse and locate the **configureTableBinding()** method. As the method name indicates, this method is responsible for the configuration of the table binding. Inside this method you find one line of code for each table column. The order of the statements represents the order of the table columns in the table. As an example, delete the code line that creates the table column for the **dateOfBirth** property and change the order of the remaining code lines so that the **lastName** table column is first and the **firstName** is the last one.

The **configureTableBinding()** method should then look as follows:

```
protected void configureTableBinding(TableBinding<Person> tableBinding) {
    tableBinding.addColumnBinding("lastName",
                                   getResourceMap().getString("lastName.columnHeader"));
    tableBinding.addColumnBinding("firstName",
                                   getResourceMap().getString("firstName.columnHeader"));
}
```

Select the **Run > Run History > Person** menu item to start the application and admire the new table column ordering.



2.7 Add business logic to the tool bar

In this step you will add a new action to the tool bar in the main view. This new action will invoke some business logic.

ULC's Application Framework uses annotated methods as actions. So first thing to do is to implement a method that executes your business logic. Then you annotate this method with the **@Action** annotation. The **@Action** annotation can take an optional

enabledProperty argument. The value of this argument identifies a bound property from the annotated method's class that is used to enable and disable the action. As an example add the following method to the **PersonMasterView** class:

```
@Action(enabledProperty = "selectionAvailable")
public void hideNames() {
    int index = getSelectedBeanIndex();
    if (index >= 0) {
        Person bean = fBeanList.get(index);

        char[] hiddenFirstName = new char[bean.getFirstName().length()];
        Arrays.fill(hiddenFirstName, '*');
        bean.setFirstName(String.valueOf(hiddenFirstName));

        char[] hiddenLastName = new char[bean.getLastName().length()];
        Arrays.fill(hiddenLastName, '*');
        bean.setLastName(String.valueOf(hiddenLastName));

        fBeanList.set(index, bean);
        PersonDAO instance = PersonDAO.getInstance();
        instance.mergeElement(bean);
    }
}
```

The name of this action is same as the method name, i.e. **hideNames**. The action can be executed only if the **selectionAvailable** property of the action method's class (**PersonMasterView**) is true.

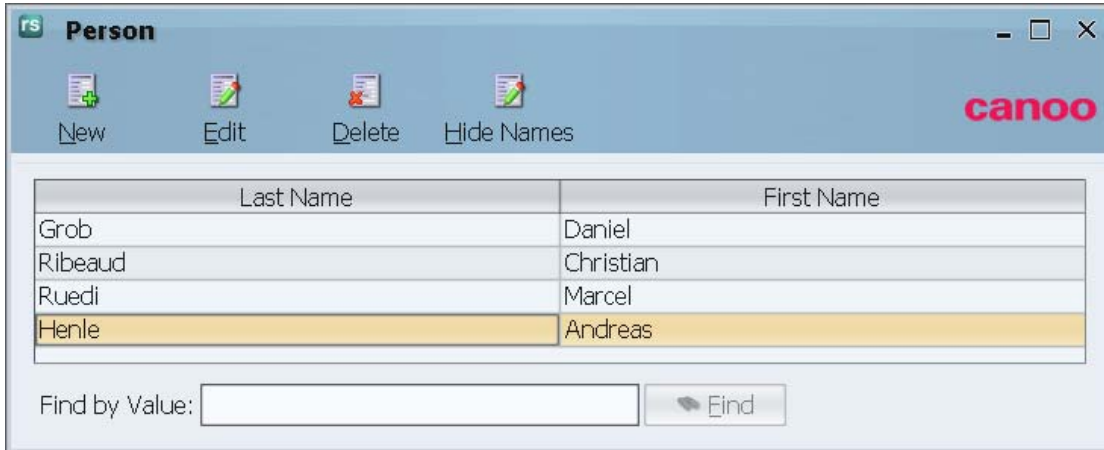
Now that you have defined the action logic, you want to add it to the tool bar. The tool bar actions are configured in the **PersonMasterView.getToolBar()** method. Add the **hideNames** action in the corresponding place.

```
public ULCToolBar getToolBar() {
    return new ToolBarFactory(getActionMap()).createToolBar(
        "newAction", "modifyAction", "removeAction", "hideNames");
}
```

This is all you have to do to display the action in the tool bar. However, the corresponding button in the toolbar just displays the name of your action, i.e. **hideNames**, but no icon, no tool tip, and no mnemonic. You can change the button text and define other action properties in the resource bundle of the action method's class. In our case this is the **PersonMasterView.properties** resource bundle which can be found in the *com.canoo.person.view.resources* package. All action properties are prefixed with **actionName.Action**, which is **hideNames.Action** in our example. Add the following lines to the bundle:

```
Title=Person
...
hideNames.Action.text = &Hide Names
hideNames.Action.icon = icons/form_blue_edit.png
hideNames.Action.shortDescription = Hides the selected person's names.
hideNames.Action.longDescription = Hides the selected person's names. Hiding means ...
```

Run the application again to see the result as shown below:



The screenshot shows a window titled "Person" with a toolbar containing icons for "New", "Edit", "Delete", and "Hide Names". The "canoo" logo is in the top right corner. Below the toolbar is a table with two columns: "Last Name" and "First Name". The table contains the following data:

Last Name	First Name
Grob	Daniel
Ribeaud	Christian
Ruedi	Marcel
Henle	Andreas

Below the table is a search bar labeled "Find by Value:" with a text input field and a button labeled "Find".

2.8 Change form layout

In this step you will change the form layout in the detail dialog.

The generator generates code that adds an input field for each property in your bean to the form in the detail dialog. This is the best that the generator can do but is usually not what you want for your application. However, it is easy to change this generated code.

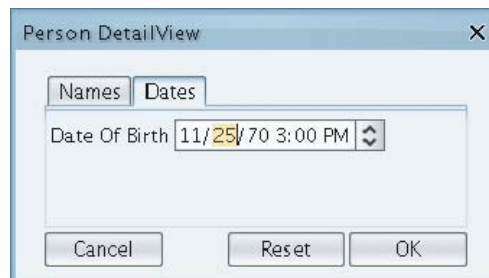
Open the **com.canoo.person.view.PersonForm** class inside Eclipse and locate the **initForm()** method. As the method name indicates, this method is responsible for initializing the form layout. Inside this method you will find one line of code for each input field in the form. The order of the statements represents the order of the input fields in the form. With these statements you specify:

- Kind of input field, e.g. text field, slider, spinner, combo box.
- Configuration of the input field, e.g. width of text field, min / max values of spinner.
- Layout configuration, e.g. let the input field grow or not, display a label or not.
- Communication configuration, e.g. input triggers a round trip or not.

When having a large number of input fields, the form allows you to group them in several tabs. As an example, let's place the **firstName** and the **lastName** property on one tab and the **dateOfBirth** property on a second one. In addition let's change the input field type for the **dateOfBirth** property into a **spinner**.



The screenshot shows a dialog titled "Person: DetailView" with two tabs: "Names" and "Dates". The "Names" tab is selected, showing input fields for "First Name" (containing "Daniel") and "Last Name" (containing "Grob"). At the bottom are buttons for "Cancel", "Reset", and "OK".



The screenshot shows the same dialog "Person: DetailView" but with the "Dates" tab selected. It shows a "Date Of Birth" spinner field displaying "11/25/70 3:00 PM". At the bottom are buttons for "Cancel", "Reset", and "OK".

The **initForm()** method should then look as follows:

```
protected void initForm() {
    setColumnWeights(0f, 0f, 1f);

    startTab("Names");
    addTextField("firstName").columns(15);
    addTextField("lastName").columns(15);

    startTab("Dates");
    addSpinner("dateOfBirth").notGrowing(false);
}
```

2.9 Add validation to the form

In this step you will add validation to the form in the detail dialog.

To add validation code, open the **com.canoo.person.view.PersonFormModel** class. This class provides all specifications for the form. Among this is the specification of validation on an input field. Locate the **createValidators()** method in this class. This method defines the validation to be performed for each input. Let's add a validator that checks if a name field starts with a digit or not.

The **createValidators()** method should then look as follows:

```
protected IValidator[] createValidators() {
    return new IValidator[]{new PropertyValidator<String>("firstName", "lastName") {
        public String validateValue(String value) {
            if (value == null || value.trim().length() == 0) {
                return null;
            }

            char firstCharacter = value.charAt(0);
            if (Character.isDigit(firstCharacter)) {
                return "name field must not start with digit";
            }

            return null;
        }
    }};
}
```



2.10 Deployment

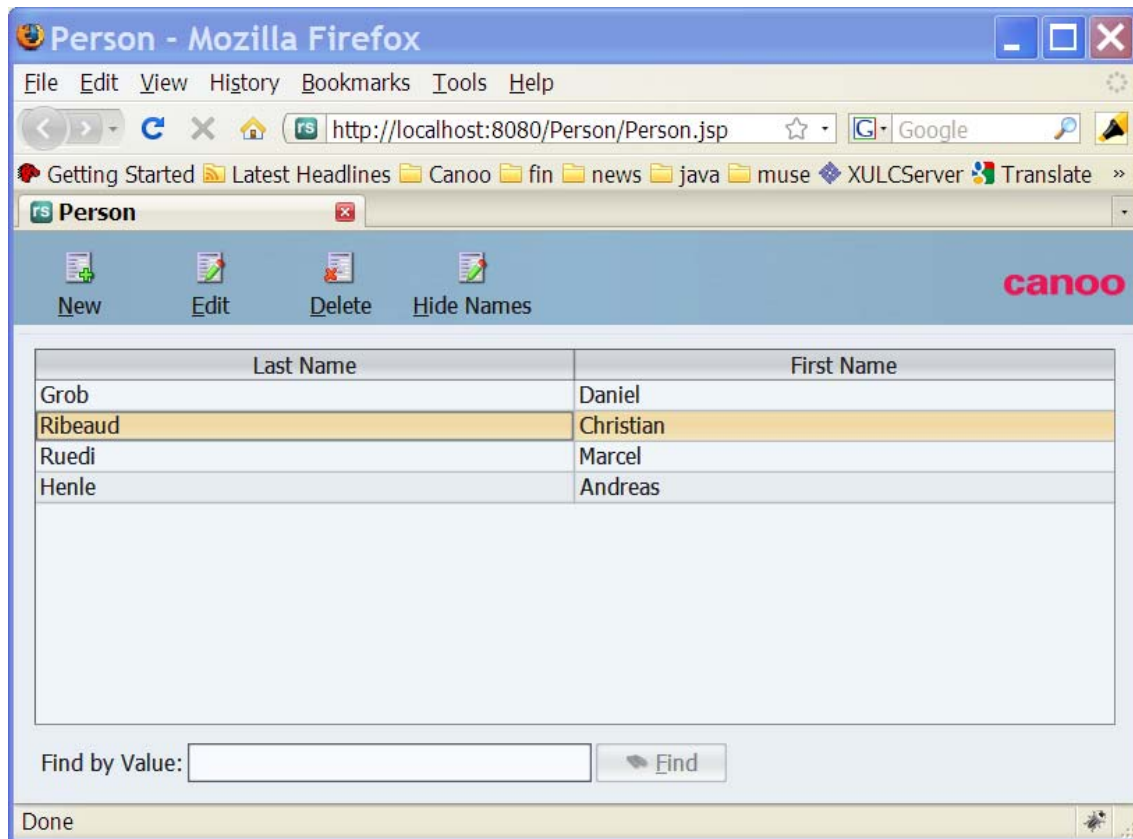
In this step you will deploy the application on a server and access the application from within your browser.

The generator that generated the Eclipse project in one of the first steps prepared everything for you. You get deployment capabilities for free. To deploy your application you need to create a WAR file and copy it to an appropriate deployment folder of your application server. You can export the WAR file of your application to

the deployment directory of your application server from within Eclipse. Make sure that your application server supports Java Servlet 2.4 standard.

The generator also created a file **copy-to-GoogleApp.xml** and launch configuration **Person-copy-to-GoogleApp.launch** that executes the XML file with Ant. It enables deployment of the ULC Application into Google's App Engine (<http://code.google.com/appengine/>).

To deploy and run your ULC application as a servlet in a servlet container such as Tomcat, on your project's right-click menu select the **File > Export...** menu item to start the export wizard. Select **Web > WAR file** as export destination in the tree and click **Next >**. Fill the **Destination** text field with the deployment directory of your application server (for example **webapps** directory of Tomcat) and click **Finish** to finally create the WAR file. You can then start your application server and direct your browser to the URL of your deployed ULC application as shown in the picture below:



Alternatively, instead of explicitly exporting and deploying a war file, you can right click on the **Person** project and choose **Run As > Run On Server** to deploy, start server and bring up the application in a browser. To be able to do this you should have configured Eclipse to start Tomcat as a Server.

3 Set up

3.1 How to install Canoo RIA Suite

See section 2.1 *Install Canoo RIA Suite*.

3.2 About licenses

This section provides information about how ULC and extension licenses are managed.

3.2.1 License key location

The installer will create the required license files in the *<User Home Directory>/ulc-<version>* directory.

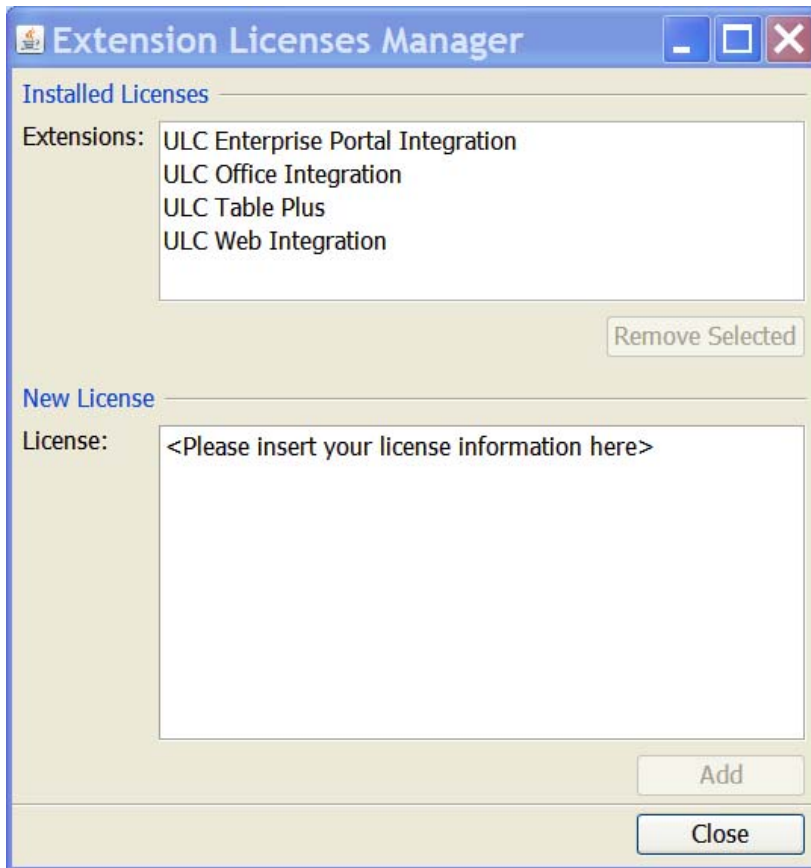
3.2.2 Using the ULC license manager to manage ULC licenses

After a Canoo RIA Suite installation, you may use the license manager in the following situations:

1. To upgrade one or more licenses. After having evaluated the software with an evaluation license, you ordered a commercial one and want to upgrade your installation.
2. To activate one or more extensions. You bought an extension and want to activate it in your Canoo RIA Suite installation.
3. To deactivate one or more extensions. After having tested an extension and because the evaluation period is over, you want to deactivate it in your Canoo RIA Suite installation.

The license manager is located in *<Canoo RIA Suite Install Dir>/tool* directory and is started by double clicking or executing the **ulc-license-manager.jar**.

The picture below shows the license manager tool. In the upper panel, you see the extensions already activated. If an extension is not displayed, this means that it has not been activated. If an extension is displayed in red, this probably means that its validation period is over but the extension has not been deactivated yet.



To remove one or more extensions, just select it (or them) and click the Remove Selected button. If you have a new license for your Canoo RIA Suite installation, just copy/paste the content of the mail you got from Canoo in the lower panel and click the Add button. Your licenses will get upgraded and corresponding extensions will get activated. Once the license manager has finished adding the new license, the upper panel is automatically updated.

Note that the license manager does not upgrade the libraries in projects already existing. It only upgrades your Canoo RIA Suite licenses. To use the new libraries (e.g. libraries belonging to an extension such as ULC Table Plus) in already existing projects, you will have to import them. All activated extension libraries are located in **<Canoo RIA Suite Install Dir>/ext** directory. To be able to properly use the new extension libraries in your already existing projects, you will have to overwrite the **ulc-deployment-key.jar** library as well.

When you are finished with the license manager, click the Close button to exit the application. This application logs important operations or steps in a file called **manager.log**. This file could be used for debugging purposes. It gets overwritten by any new application start.

3.2.3 How to upgrade to new a version of Canoo RIA Suite

If you purchased a ULC developer key within 90 days before a new release of ULC is made available, or if you have a valid ULC Premium Support license associated with your developer key, you are entitled to a free upgrade key.

You can upgrade your current developer key at <http://www.canoo.com/kurt>. If you are entitled to a free upgrade key, a new developer key will be sent to your email address.

You can copy and paste the new license key during installation of the new version of Canoo RIA Suite.

3.2.4 Deployment license

To deploy a ULC application, the .jar file *<ULC Install Directory>/license/ulc-deployment-key.jar* must be available on the classpath of the ULC application deployment. Unlike the developer license, the deployment license is free of charge and freely distributable with your ULC application.

3.3 How to setup an Eclipse project

The following sections describe two ways of creating projects within Eclipse to be able to develop ULC applications.

3.3.1 As an Eclipse WST project

The easiest way to create a ULC project with a skeletal application within Eclipse is to use the generator, as described in section 2.2 *Generate an application skeleton*, to generate a ULC project. The generator generates twin projects - one for writing ULC application code and the other for writing optional ULC client side extension code. Both of these projects are configured as Eclipse Web Standard Tools projects which enable easy deployment and execution of a ULC application in an application server.

3.3.2 As a simple Java project

Alternatively, you could setup a simple Java project for developing core ULC application without add-on packages and client extensions. For this you need to create a Java project which uses Java 1.5 (or higher). You will need to put the libraries from *<Canoo RIA Suite Install Dir>/all/lib* folder on the class path of the project. The following table describes the required libraries:

ulc-core-development.jar	Needed to run the application in the development environment.
ulc-core-server.jar ulc-core-client.jar	Client and server jar to be used in the deployment environment.
commons-beanutils-1.8.0.jar commons-collections-3.2.1.jar commons-logging-1.1.1.jar	Needed for the ULC application framework.
jstl.jar standard.jar jdk5tools.jar jnlpservlet.jar	Needed for the easy deployment features.
Jemmy.zip jetty.jar (v6.1.16) jetty-util.jar (v6.1.16) junit.jar	Needed for the test framework.

servlet-api.jar (v2.5)	
------------------------	--

Add the required jars to the class path of the project. Attach the corresponding source stubs (e.g., **<Canoo RIA Suite Install Dir>/all/src/ulc-core-development-src.jar**) to the jar file in **Java Build Path** to enable code completion in the editor. In addition, specify the Javadoc location for ULC jars as **<Canoo RIA Suite Install Dir>/doc/apidoc**.

3.4 How to create and build an application skeleton

Use the generator (see Chapter 2 Getting started tutorial) to create an application skeleton.

To perform the compile, packaging and bundling tasks for the application, manually or in a build infrastructure, a build script is created with the application skeleton. The **build.xml** file is located in the root folder of the generated project. The build results of the compile and packaging task are stored in the **build** folder. The WAR package for the server deployment is located in the **dist** folder.

3.5 How to run sample applications

We encourage you to have a close look at the samples available as part of the release in **<Canoo RIA Suite Install Dir>/sample** directory. Try modifying, adding new features and experimenting with the samples to learn more about ULC. To do this you will have to set up your Java IDE for ULC development as described in section 3.3.2. You should then import the samples including their sources, resources, and libraries into a ULC project within your IDE. The following sample applications with complete source code are part of the release:

- *Online Shop*: multi-step user interaction superior to HTML
- *Team Members*: simple example of master-detail view
- *ULC Set*: ULC standard widget set (equivalent to SwingSet)
- *Pie*: example showing how to extend the ULC standard widget set
- *ULC Drag-and-Drop Set*: Drag-and-drop with ULC
- *Hello*: Hello World

However, it is not necessary to import sample applications into your IDE in order to have a first look at them. You can run them without setting them up in your IDE. To run samples, use the preconfigured Jetty Servlet engine provided with the release:

- Open the **Start** or **Program** menu and select **Canoo RIA Suite > Sample Applications > Start Sample Application Server** to start the server. Alternately, you can execute **<Canoo RIA Suite Install Dir>/sample/startServer.bat** (or **startServer.sh**).
- Open the **Start** or **Program** menu and select **Canoo RIA Suite > Sample Applications > Sample Applications Index** to open the sample index page.

Alternately, you can open **<Canoo RIA Suite Install Dir>/sample/startClient.html** in a browser.

- Select a sample application in the **Sample Applications Index** page and click **Start as Applet** to view the applet version or **Start as JNLP** to view the Java Web Start version.

As an alternative, you may also deploy a sample application as a Servlet in a Servlet engine of your choice (e.g. freely available Jakarta Tomcat servlet engine <http://jakarta.apache.org/tomcat/>). The client will run as an applet or using Java Web Start (<http://java.sun.com/products/javawebstart/>) and connects to the application using the HTTP protocol.

To deploy the sample as a Servlet:

- Install the application as a Servlet. Each sample is packaged as a web application archive (.war) file containing the application classes and all necessary libraries. The archive contains HTML files to start the client and the classes for the ULC client (to be downloaded as an applet or using Java Web Start).
- Deploy the web application archive file *<sample name>.war* from the *sample/<sample name>/webapp* directory into your Servlet container. See your Servlet container's instructions on how to deploy web applications (for the Jakarta Tomcat Servlet engine, simply copy the file into the *webapps* directory).
- Start your Servlet engine, browse to *http://localhost:<port>/<sample name>/index.html* and click on the applet hyperlinks or on the JNLP hyperlink. For the latter to work, you must have Java Web Start or any other JNLP client installed.

4 Application Development

This chapter describes various aspects of ULC application development. For detailed information on the ULC API please see <Canoo RIA Suite Install Dir>/doc/apidoc/index.html or the website [ULC API](#).

Note that the source code snippets in this chapter are for illustration purpose only; they are not necessarily part of the sample code in the ULC release.

4.1 Development Environment

ULC applications and clients can be deployed in various ways; the server part can be run in a servlet container or an EJB container (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)), the client part can be run standalone, as an applet, or using a JNLP client such as Java Web Start. In a production environment the client and server parts will typically run on different computers.

At development time, however, application developers typically like to run everything locally in order to keep the deployment process as lightweight as possible. The ULC development environment package allows both client and server to run within the same Java Virtual Machine (JVM).

4.1.1 How to run ULC applications in Development Environment

The class `com.ulcjava.base.development.DevelopmentRunner` provides a runtime environment for the client as well as for the server part. The *DevelopmentRunner* first starts the server, and then connects the client to it. Both client and server parts run inside the same virtual machine.

As with all other deployment scenarios, server and client half objects communicate through the ULC communication infrastructure. Since the two parts run in the same JVM, the communication can be implemented in a straightforward way by the ULC framework where the ULC requests between client and the server are exchanged using request queues.

The development environment package provides ways to simulate all relevant deployment and runtime aspects that a ULC application developer must be aware of, such as *init* and *user* parameters, and connection characteristics. For convenience, a graphical user interface is provided that can be used to control these aspects. Moreover, this user interface provides the means to investigate the communication between the server and client half objects.

The *DevelopmentRunner* is suited to provide a ULC developer with early feedback on how an application will perform under expected network conditions. Note that the applet module provides an *AppletDevelopmentRunner* that should be used whenever a ULC application uses the *ULCAppletPane* component. It is based on the standard *DevelopmentRunner* and displays the applet pane in a separate applet viewer window.

4.1.2 How to configure DevelopmentRunner

The developer can configure the following parameters when using the *DevelopmentRunner*. Some important parameters are:

Parameter	Description
Application main class	The application main class implementing the <i>com.ulcjava.base.application.IApplication</i> interface.
Init parameters	The init parameters that are provided to the application. At deployment time these parameters are specified through (server-side) container configuration.
User parameters	The user parameters that are passed to the application. At deployment time these user parameters are provided by the (client-side) launcher component.
Connection characteristics	Connection characteristics used for simulation of transport delays. A range of typical configurations is provided. Additionally, the developer can define custom configurations.
Log level	Log level used on this application. The developer can change the log level used in the development environment.

To configure these parameters, the *DevelopmentRunner* class provides four usage modes that are described in the following sections. The *DevelopmentRunner* can be configured in one of the following ways:

- Using the *ULCApplicationConfiguration.xml* file
- Programmatically using setter methods.
- Using program startup parameters.
- The developer can use a graphical user interface that also provides a monitor visualizing information on client-server communication. Moreover, this user interface can be used to pause and resume the application and to test passivation and activation of application sessions.

Configuration using *ULCApplicationConfiguration.xml*

ULC Application:

```

public class Hello extends AbstractApplication implements Serializable {
    public void start() {
        ULCLabel label = new ULCLabel("Hello World!");
        label.setHorizontalAlignment(ULCLabel.CENTER);
        label.setPreferredSize(new Dimension(140, 40));

        ULCTable frame = new ULCTable("Hello Sample");
        frame.setDefaultCloseOperation(ULCTable.TERMINATE_ON_CLOSE);
        frame.add(label);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        DevelopmentRunner.run();
    }
}

```

ULCApplicationConfiguration.xml file:

```
<?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.sample.hello.Hello
  </ulc:applicationClassName>
  <ulc:clientLogLevel>INFO</ulc:clientLogLevel>
  <ulc:serverLogLevel>INFO</ulc:serverLogLevel>
</ulc:ULCApplicationConfiguration>
```

Programmatic Configuration

The *DevelopmentRunner* class provides the following static methods to programmatically configure and start it:

Method	Description
<i>setApplicationClass(Class)</i>	Sets the application main class implementing the <i>com.ulcjava.base.application.IApplication</i> interface.
<i>setInitParameters(Properties)</i>	Sets the init parameters as defined by the key-value pairs in the supplied <i>java.util.Properties</i> object.
<i>setUserParameters(Properties)</i>	Sets the user parameters as defined by the key-value pairs in the supplied <i>java.util.Properties</i> object.
<i>setConnectionType(ConnectionType)</i>	Sets the connection type used for the simulation of transport delays to the supplied <i>com.ulcjava.base.development.ConnectionType</i> object.
<i>setLogLevel(Level)</i>	Sets the log level of the application. The argument has to be an instance of <i>com.ulcjava.base.shared.logging.Level</i> . The default is set to <i>com.ulcjava.base.shared.logging.Level.WARNING</i> .
<i>setUseGui(Boolean)</i>	If set to <i>true</i> , the graphical user interface of the <i>DevelopmentRunner</i> will be opened when <i>run</i> is called (see Section 4.1.3).
<i>run()</i>	Starts the <i>DevelopmentRunner</i> using the currently defined configuration.

Typically you will need to create a development launcher class which configures and starts the *DevelopmentRunner* for your specific application. This is done in the *main()*

method of the launcher. The following code shows a typical example of such a development launcher.

```
public class TeamMembersDevelopmentLauncher {  
    public static void main(String[] args) throws Exception {  
        DevelopmentRunner.setApplicationClass(TeamMembers.class);  
        DevelopmentRunner.run();  
    }  
}
```

Configuration Using Program Arguments

In addition to the methods for programmatic configuration, the *DevelopmentRunner* class offers a *main()* method to support passing program arguments:

Program argument	Description	Required
-applicationClass <i>application class</i>	The fully qualified application class name (e.g., <i>com.ulcjava.sample.hello.Hello</i>).	Yes Optional if <i>-useGui</i> is specified.
-initParameter <i>key=value</i>	Defines an init parameter. The format of the key-value argument must be as in Java property files.	No May be repeated.
-userParameter <i>key=value</i>	Defines a user parameter. The format of the key-value argument must be as in Java property files.	No May be repeated.
-connectionType <i>connection type</i>	Defines the connection type to be used. Must be one of the following: UNLIMITED, LAN10M, LAN1M, DSL512K, DSL256K, DSL112K, MODEM56K, MODEM28K, or custom.	No Default is UNLIMITED
-logLevel <i>logLevelName</i>	Sets the log level.	No Default is WARNING.
-useGui	If present, the graphical user interface will be opened (see 4.1.3: Graphical User Interface).	No
-reloadClasses (only applicable together with the -useGui option)	If present, classes will be reloaded when the Start button in the GUI is pressed. Classes to be excluded from reloading can be defined in a file <i>excluded.properties</i> , located anywhere on the classpath (see 4.1.3: Graphical User Interface).	No

The following command line shows how to start the *Hello* application using the *DevelopmentRunner*; additionally two init parameters are defined and the connection type is configured to be a 28K modem connection.

```
java com.ulcjava.base.development.DevelopmentRunner
  -applicationClass com.ulcjava.sample.hello.Hello
  -initParameter db-url=jdbc:mysql://somehost/test
  -initParameter mail-host=someotherhost
  -connectionType MODEM56K
```

Programmatic configuration and configuration using program arguments may be combined as illustrated by the following example. The example defines the application main class programmatically. All other parameters are taken from the program parameters. Note that program arguments will overwrite programmatic configuration if there are conflicts.

```
public class TeamMembersDevelopmentLauncher {
    public static void main(String[] args) throws Exception {
        DevelopmentRunner.setApplicationClass(TeamMembers.class);
        DevelopmentRunner.main(args);
    }
}
```

4.1.3 Graphical User Interface for Development Environment

The graphical user interface (GUI) of the *DevelopmentRunner* can be used to conveniently enter all configuration parameters. Using the provided controls, an application session can be started and stopped from this GUI. Moreover, an application session can be paused and resumed as well as passivated and activated.

In addition, the GUI provides information on the communication between client and server and offers the possibility to dump ULC requests to a console for debugging purposes.

When the **Class reloading** checkbox is selected, all application classes are reloaded during the next startup of the application. To reload, the *DevelopmentRunner* uses a custom class loader that sits in front of the system class loader. Classes in the *com.ulcjava.base* packages are always excluded from class reloading. In addition, classes defined in an *excluded.properties* file are loaded by the system class loader and are hence also excluded from class reloading. Class and package entries in this file must obey the following syntax:

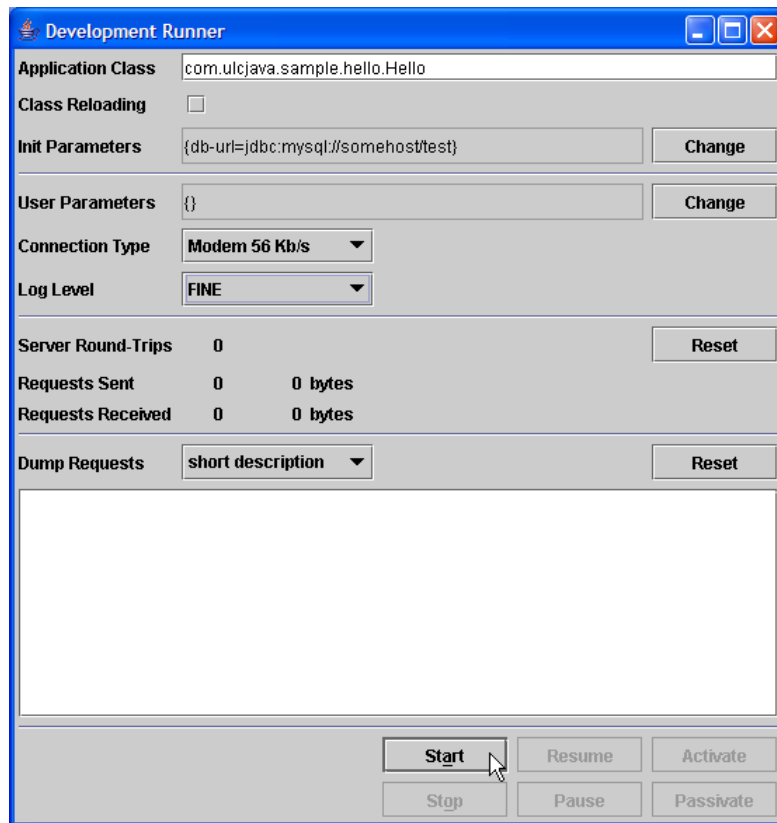
```
excluded.<nr>=<package or class name>
```

Example:

```
excluded.0=org.*
excluded.1=com.weblogic.*
excluded.2=com.acme.StartupClass
excluded.3=com.oracle.*
```

There is a default *excluded.properties* file located in the *com.ulcjava.base.development* package that contains the most common classes that need to be loaded by the system class loader. Place a custom *excluded.properties* file on the *DevelopmentRunner* classpath before the default, either in the same package or in the default package.

Below is a screenshot of the *DevelopmentRunner* GUI.



Development Runner GUI

The graphical user interface is started in either of the two ways described in section 4.1.2. Init and user parameters that are already defined at GUI startup can be changed before the application is started.

4.2 Application Main Classes

The next section describes the main classes that are used to implement a ULC application.

4.2.1 How to implement a ULC Application

Implementing the *com.ulcjava.base.application.IApplication* interface creates a ULC application. The *IApplication* interface defines the methods that control the lifecycle of a ULC application.

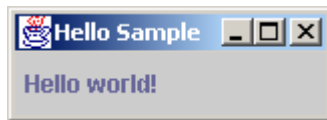
In accordance with the long tradition of developers around the world, let us start ULC programming by writing the classic “Hello World” application. The following code shows a fully functional ULC application that displays a frame containing a single label. Additionally, the application is configured to terminate when the user closes the frame.

```

public class Hello extends AbstractApplication {
    public void start() {
        ULCFrame frame = new ULCFrame("Hello");
        frame.setDefaultCloseOperation(ULCFrame.TERMINATE_ON_CLOSE);
        frame.add(new ULCLabel("Hello world!"));
        frame.setVisible(true);
    }
}

```

The following picture shows a screenshot of the running application. Note that this application is also provided as a sample application with the release.



For more technical information on how applications are created and managed, please see the chapter on *Server Architecture* in the [ULC Architecture Guide](#).

4.2.2 IApplication

The `com.ulcjava.base.application.IApplication` interface is the central interface that ULC applications must implement. For every connecting client, a new instance of the class implementing this interface is created by the underlying runtime container adapter (e.g., servlet container adapter).

Features

- The `start()` method is the first method that is called after a new instance of the implementation has been created. Applications build up their first view within this method and make the view visible.
- The `stop()` method is the last method that is called before a session is terminated. Applications typically use this method to clean up and release any resources used for this session.
- The `passivate()` method is called before the session is passivated. This method only needs to be implemented for applications that are deployed in server containers that support session passivation (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)). In this method the application is supposed to release any resources (e.g., DB connections) or transient data that should not be serialized.
- The `activate()` method is called after the server session has been activated again. It must be implemented for applications that are deployed in server containers that support passivation, `activate()` should reopen resources or restore transient state.
- The `handleMessage()` method is called when the server session receives a message from the client session. This mechanism can be used when implementing custom client launcher classes to notify the application about special client-side events (see the example in the chapter on *Interacting with the Enclosing HTML Page* in the [ULC Deployment Guide](#)).
- The `pause()` method is called when the client sends pause message to the ULC application. The server side ULC session will remain active to be resumed later.
- The `resume()` method is called when the client sends a resume message to the previously paused ULC application.

4.2.3 AbstractApplication

The *com.ulcjava.base.application.AbstractApplication* class is a default implementation of the *com.ulcjava.base.application.IApplication* interface.

Features

- The *AbstractApplication* class provides empty implementations of all methods except the *start()* method. Many ULC applications will only need to implement the *start()* method.

4.2.4 ApplicationContext

The *com.ulcjava.base.application.ApplicationContext* class provides access to the application's runtime environment. ULC applications can use this class to access the environment in a container-independent way. Note that all methods are static, so the ULC framework forwards the call to the corresponding application instance.

Features

- The *terminate()* method is used to terminate the current session. A call to this method will close all open windows, call the *stop()* method on the *IApplication* implementation, and terminate the session.
- The *getApplication()* method returns the actual *IApplication* instance. This method may be used to easily access the main application instance from anywhere in the application code.
- The *setAttribute()* and *getAttribute()* methods may be used to easily access objects from anywhere in the application code. The *getAttributeNames()* method returns the names of all attributes of the ULC application. The *removeAttribute()* method removes an attribute from the ULC application.
- The *getInitParameter()* method returns the value of the named initialization parameter as defined by the deployment. The *getInitParameterNames()* method returns the names of all initialization parameters of the ULC application. These calls are forwarded to the underlying runtime container (e.g., Servlet container); the actual declaration of these init parameters at deployment time is therefore container-dependent.
- The *getUserPrincipal()* method returns a *java.security.Principal* object containing the name of the current authenticated user. If the user has not been authenticated, the method returns *null*. A call to this method is forwarded to the underlying runtime container, the setup of the authentication configuration is container-dependent.
- The *isUserInRole()* method returns a boolean indicating whether the authenticated user is included in the specified logical role. If the user has not been authenticated, the method returns *false*. A call to this method is forwarded to the underlying runtime container, definition of roles and role membership is container-dependent.
- The *getServerContainerType()* method answers the type of server container, namely Development, Servlet, EJB, or Local.
- Method *addRoundTripListener(IRoundTripListener listener)* for the application to get notified when a roundtrip begins and ends.
- There are methods to find out version information of ULC libraries being used by the application.

The following code snippet shows how to use the *ApplicationContext* to terminate the application.

```
quitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        ApplicationContext.terminate();
    }
});
```

The following example demonstrates how to access init parameters defined at deployment time.

```
String dbUrl = ApplicationContext.getInitParameter("db-url");
```

The following snippet shows how can an application register to get notified on each roundtrip:

```
public class MyApplication extends AbstractApplication {

    public void start() {
        ApplicationContext.addRoundTripListener(new IRoundTripListener() {
            public void roundTripDidStart(RoundTripEvent event) {
                System.out.println("Round trip has started...");
            }

            public void roundTripWillEnd(RoundTripEvent event) {
                System.out.println("Round trip will end...");
            }
        });
    }
    ...
}
```

4.2.5 ApplicationFramework's Application and ApplicationContext

The *com.ulcjava.applicationframework.application.Application* class implements *IApplication* and defines a simple lifecycle: initialize, startup, ready, and shutdown. The *start()* method implementation calls

- *initialize()* method: for initialization before the GUI becomes visible
- *startup()* method: for constructing and showing the first screen of the GUI
- *ready()* method: called after initial GUI has become visible on the client side

The *stop()* method calls the *shutdown()* method which may contain code for clean up before application termination.

The *createApplicationContext()* method creates ApplicationFramework's *ApplicationContext* that provides access to global objects such as *ResourceManager*, *ActionManager*, and methods of *com.ulcjava.base.application.ApplicationContext*.

4.2.6 ApplicationFramework's SingleFrameApplication

The *com.ulcjava.applicationframework.application.SingleFrameApplication* extends the *Application* class to provide a ready to use ULC Application skeleton with full life cycle and a main window (with menu bar, tool bar, status bar, and content pane) whose type (applet or frame) is automatically determined by the client deployment environment (browser or JWS). It uses applications *ResourceMap* for injecting properties for the entire component hierarchy starting with the rootpane.

```
public class Hello extends SingleFrameApplication {
    protected ULComponent createStartupMainContent() {
        return new ULCLabel("Hello UltraLightClient");
    }
}
```

```
    public static void main(String[] args) {  
        DevelopmentRunner.main(args);  
    }  
}
```

4.3 Main windows

Every application needs a main window. ULC provides classes like *ULCFrame* and *ULCAppletPane* to be used for the main window of an application depending on the client deployment environment. However, ULC's ApplicationFramework provides ready to use base classes such as *SingleFrameApplication* that do the necessary setup and implement all the low level details. Normally, there is no reason to not extend from one of these base classes.

4.3.1 How to define a main window

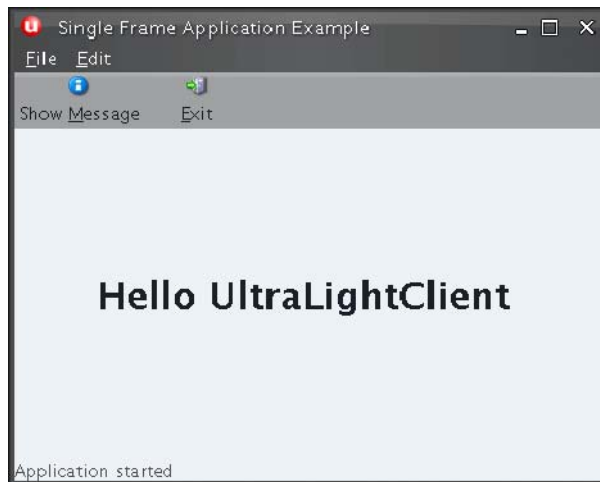
The current version of the application framework contains exactly one such base class: **com.ulcjava.applicationframework.application.SingleFrameApplication**. This base class provides the following services:

- A standard window layout with menu bar, tool bar, status bar, and main content area
- A predefined application life cycle.
- Respects the browser, i.e. when the application runs inside a browser, the main window is embedded inside the browser, otherwise the main window gets its own window.
- Action management
- Resource injection

The following snippet shows ULC's version of the "Hello World" application:

```
public class HelloWorldApplication extends SingleFrameApplication {  
    @Override  
    protected ULComponent createStartupMainContent() {  
        return new ULCLabel("Hello UltraLightClient");  
    }  
  
    // The main method to run the application in the DevelopmentRunner  
    public static void main(String[] args) {  
        Application.run(args);  
    }  
}
```

The example application code following the picture below overwrites the *SingleFrameApplication*'s factory methods to add components to the main content area and to create the application's menu bar, tool bar and status bar. The action management feature of the application framework is used to link menu and tool bar items to application methods. For example, the *createToolBar("showMessage", "quit")* method maps *showMessage()* and *quit()* (a predefined action in *Application* class) actions to the tool bar buttons.



```
public class SingleFrameApplicationExample extends SingleFrameApplication {
    private static final String[] EDIT_MENU =
        new String[] {"&Edit", "copy", "paste", "-", "selectAll"};
    private static final String[] FILE_MENU =
        new String[] {"&File", "exit"};

    @Override
    protected ULComponent createStartupMainContent() {
        ULLabel label = new ULLabel("Hello UltraLightClient");
        label.setName("main.content.Label");
        return label;
    }

    @Override
    protected ULMenuBar createStartupMenuBar() {
        return buildMenuBar(FILE_MENU, EDIT_MENU);
    }

    @Override
    protected ULComponent createStartupToolBar() {
        ULToolBar toolBar = buildToolBar("showMessage", "quit");
        toolBar.setBackground(Color.gray);
        return toolBar;
    }

    @Override
    protected ULComponent createStartupStatusBar() {
        ULLabel statusBarLabel = new ULLabel();
        statusBarLabel.setName("status.Label");
        return statusBarLabel;
    }

    @Action
    public void showMessage() {
        new ULAlert("Message", "Snippet Message", "OK").show();
    }

    // The main method to run the application in the DevelopmentRunner
    public static void main(String[] args) {
        Application.run(args);
    }
}
```

Many properties of the window and the widgets are injected using the *SingleFrameApplicationExample.properties* file by way of resource bundles:

```
Application.title=Single Frame Application Example

main.Frame.size=450,360
main.Frame.iconImage=icons/ulcicon.png

main.content.Label.horizontalAlignment=#{ULLabel.CENTER}
main.content.Label.font=-bold-28
```

```
status.Label.text=Application started
status.Label.foreground=darkGray
status.Label.opaque=true

quit.Action.text=&Exit
quit.Action.icon=icons/exit.png

showMessage.Action.text=Show &Message
showMessage.Action.icon=icons/information.png
```

4.3.2 How to access the main window

The *SingleFrameApplication* uses a suitable root pane depending on the client runtime environment. If the client runs the application inside a browser it uses a *ULCApletPane*, otherwise a *ULCFrame*. There are several ways to access these root panes:

1. Injecting properties using the ResourceMap, as seen in the example above **main.Frame.size=450,360** sets the size of the *ULCFrame*. The key prefix for the *ULCFrame* is **main.Frame**, for the *ULCApletPane* it is **main.AppletPane**.
2. Overwrite the **initFrame()** and/or **initAppletPane()** methods to set the initial values.
3. Use the **getRootPane()**, **getAppletPane()**, **getFrame()** methods to access the application's root pane.

4.3.3 How to add an Action to the main window

The application framework provides an API for action management. Creating an action is nothing more than annotating a public method with the **@Action** annotation. This defines an action with the method name as action name. Use resource bundles to change the action text and define other action properties such as icon, mnemonic, accelerator, and tool tip text. The action properties in the resource bundles are all prefixed with **actionName.Action**:

```
sayHello.Action.text=Say &Hello
sayHello.Action.icon=information2.png
sayHello.Action.accelerator = shortcut H
sayHello.Action.shortDescription = Say Hello
sayHello.Action.longDescription = Says hello to the user in an extra dialog
```

The above excerpt from a resource bundle defines the action properties for the *sayHello* action. To specify a mnemonic, just prefix the mnemonic with an ampersand.

Use the **ActionManager** class of the application framework to access an action. However, it is often not necessary to deal with action instances directly in your code. For common tasks, the application base classes provide convenience methods that take as arguments action names instead of actions. Examples are the *buildMenuBar()* method to create the menu bar and the *buildToolBar()* method to create the tool bar.

```

public class ActionSnippet extends SingleFrameApplication {
    @Override
    protected ULCMenuBar createStartupMenuBar() {
        return buildMenuBar(new String[] { "&File", "sayHello" });
    }

    @Override
    protected ULCComponent createStartupToolBar() {
        ULCToolBar result = buildToolBar("sayHello");
        result.setBackground(Color.gray);

        return result;
    }

    @Override
    protected ULCComponent createStartupMainContent() {
        IAction sayHelloAction = getAction("sayHello");
        ULCButton sayHelloButton = new ULCButton(sayHelloAction);

        IAction sayHelloEnabledAction = getAction("sayHelloEnabled");
        ULCheckBox sayHelloEnabledCheck = new ULCheckBox(sayHelloEnabledAction);

        ULCBoxPane result = new ULCBoxPane(1, 0);
        result.add(sayHelloButton);
        result.add(sayHelloEnabledCheck);

        return result;
    }
    ...
}

```

4.3.4 How to enable / disable Actions

The `@Action` annotation can take an optional **enabledProperty** argument. The value of this argument identifies a bound property from the annotated method's class that you then can use to enable and disable the action. Whenever the enabled property changes, the action receives the property change event and enables or disables itself accordingly.

```

@Action(enabledProperty = "sayHelloEnabled")
public void sayHello() {
    new ULCAalert("Message", "Hello UltraLightClient", "OK").show();
}

public void setSayHelloEnabled(boolean sayHelloEnabled) {
    boolean oldSayHelloEnabled = this.sayHelloEnabled;
    this.sayHelloEnabled = sayHelloEnabled;

    // fire property change event
    firePropertyChange("sayHelloEnabled", oldSayHelloEnabled, sayHelloEnabled);
}

```

4.3.5 How to give early startup feedback

The application framework first invokes the **startup()** method of your application. The **startup()** method uses various factory methods to fill the parts of the user interface. All of these factory methods have a **createStartup** prefix for example **createStartupContent()**, **createStartupToolBar()**, etc.

Typically, you display only the initial user interface in the **startup()** method. You should not do any time consuming action in the **startup()** method else the user will have to wait before the first screen appears. Showing the initial GUI gives the user an early impression that the application is up and running.

After the execution of the **startup()** method, the framework invokes the **ready()** method in a separate server round-trip. You should do time consuming initialization actions such as database lookups / heavy computation stuff inside the **ready()** method.

```
public class StartupReadySnippet extends SingleFrameApplication {
    @Override
    protected ULCComponent createStartupMainContent() {
        return new ULCLabel("Loading...");
    }

    @Override
    protected void ready() {
        doVeryComplexDatabaseQuery();

        setMainContent(new ULCLabel("Hello UltraLightClient"));
    }

    @Override
    protected void initFrame(ULCFrame frame) {
        super.initFrame(frame);

        frame.setSize(600, 400);
    }
    ...
}
```

4.3.6 How to show popup windows and message alerts

ULC provides the class *ULCDialog* for use as a popup window. Dialogs can be modal and non-modal. For displaying message alerts, ULC provides the class *ULCAAlert*.

4.4 Layout Containers

In many cases, the target machine on which the UI Engine will run is not known at development time. Nevertheless, the layout of a user interface should be visually appealing in different environments. Explicit placement of widgets does not work well in this situation because the exact widget sizes are not known in advance. In addition, the precise horizontal and vertical alignment of widgets and the specification of the resize behavior is a tedious task. ULC provides layout management as an integral part, based on a hierarchical and high-level layout description rather than on the explicit placement of widgets. This layout management handles resize behavior automatically. The layout mechanism of AWT and Swing has the same goals.

4.4.1 Various layout options in ULC

ULC supports four layout container types:

- **Basic:** The basic layout containers align widgets in a row/column fashion. These are *ULCBoxPane*, *ULCGridBagLayoutPane*, *ULCGridLayoutPane*, *ULCBorderLayoutPane*, *ULCFlowLayoutPane*, and *ULCBoxLayoutPane*.
- **Stacked:** The stacked layout containers pile page components on top of each other in such a way that only the topmost page is visible. These are *ULCCardPane*, and *ULCTabbedPane*.
- **Special-purpose:** The special-purpose layout containers provide layout management on a higher abstraction level. These are *ULCScrollPane*, *ULCSplitPane*, and *ULCToolBar*.
- **MDI:** MDI layout containers provide layout management by handling a set of internal frames. This is *ULCDesktopPane*.
- **Null:** *ULCLayeredPane* has no layout manager and hence every added component must have bounds.

The widgets *ULCBoxPane* and *ULCGridBagLayoutPane* have the notion of a *cell* and *cell alignment*. A cell is a space which can be empty or can accommodate a single widget. The size of a cell is always equal to or greater than the minimum size of the enclosed widget (if any). If the cell is larger than the preferred size of the widget, the weight and the cell alignment specifies what to do with the extra space. Possible options are to expand the widget until it fills the cell completely or to align the widget within its cell:

	Left	Center	Right	Expand
Top				
Center				
Bottom				
Expand				

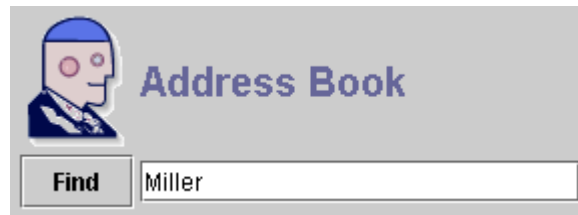
4.4.2 How to use *ULCBoxPane*

The *ULCBoxPane* layout component is based on a few fundamental concepts. Combining these layout strategies recursively enables sophisticated layouts and automatic resize behavior. There are many ways to solve a given layout problem and in general it is not intuitively clear which approach is the least complex and the most appropriate. The following paragraphs outline an overall strategy for designing and implementing a layout using the *ULCBoxPane* layout concepts. In addition, we provide some guidelines and tips for cases with several alternative solutions.

Planning the Layout

Mentally, visualize the layout as a grid. Decide which components will be placed in each grid cell. Decide how that component should size itself when the parent view is resized. For almost every layout a resize strategy has to be defined: which elements should grow if the containing window is enlarged? What happens in a localized version of the layout when a supported language has different string length requirements?

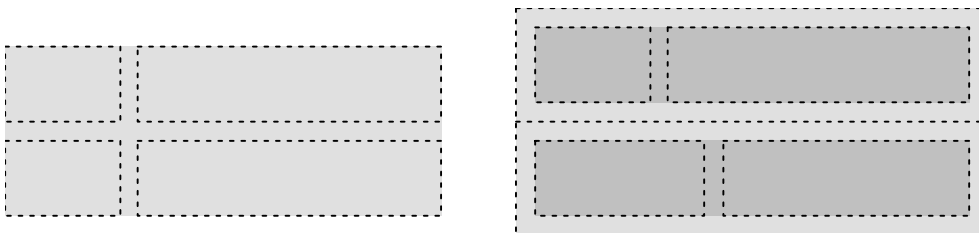
Another requirement is aesthetics: how do we keep controls nicely grouped and aligned and not spaced too far apart even if everything may resize drastically?



Nested boxes or a two-dimensional box?

Start from Top to Bottom

Identify the top-level elements or groups of elements. Decide whether to use a horizontal, vertical or two-dimensional *ULCBoxPane* layout. Sometimes this is not as simple as it seems. For example: is the top-level layout of figure above a 2x2 box (see figure below left) or is it a vertical box with two rows each containing a horizontal box (see figure below right)?



Possible layout structures

Sometimes an answer can be found by asking whether elements should be kept aligned across horizontal rows. For example, should the label **Address Book** line up with the **Find** text field? If yes, a 2x2 box has to be used. If not (which appears to be more realistic in this case) it is better to nest independent horizontal boxes within one vertical box.

Because constructing box hierarchies is simpler than rearranging them, build and verify the complete box layout before filling in and adjusting the settings of all the non-layout widgets.

Avoid Deep Nesting

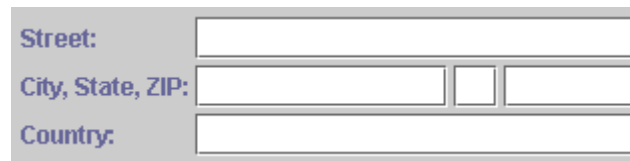
Always try to keep your layout structures simple. The most important way to achieve this goal is by avoiding structures nested too deeply. Using two-dimensional boxes instead of nesting boxes simplifies the layout. But this goal sometimes conflicts with a pleasing layout because the two-dimensional box forces elements into a rigid grid and thereby aligns elements, which need not or should not align (see figure below). Below we describe how spanning makes the use of two-dimensional boxes much more flexible.

Use Spanning

Spanning allows to merge adjacent cells to form larger cells. This is useful if you want to align components with different size requirements within an overall grid. Without spanning, the width and/or height of rows and columns are determined by the widest or

tallest element. With spanning, exceptionally large elements can grow into neighboring cells without making their containing rows or columns too wide or tall.

One way to implement the following figure is to nest a 1x3 box (for city, state, and zip) inside the overall 3x2 box. Spanning allows the use of a single overall 3x4 box (with spanning of street and country field) and avoids nesting altogether.



Using spanning

Spanning is a very powerful mechanism. You can use a grid layout, which improves the overall aesthetics because elements are aligned and their spacing is more uniform. On the other hand you are not forced to make everything the same size because you can span elements across multiple grid cells. In addition, grid spanning makes it easy to have elements with different sizes to always use multiples of some base cell size, which is visually more appealing than having elements with various unrelated sizes (this concept is called a typographic grid).

When using the *ULCBoxPane*, make sure that for each column, at least one component has its vertical constraint property set to `EXPAND` and for each row, at least one component has its horizontal constraint property set to `EXPAND`. Otherwise, your UI might look odd when extra space is available.

A Simple Address Form Using Nested Boxes

Find below the code to build the form displayed in the figure above using nested boxes:

```
// Create a box with 3 rows and 2 columns
ULCBoxPane addressBox = new ULCBoxPane(2, 3);

// add the label for Street
addressBox.add(new ULCLabel("Street:"));

// add the field for Street and specify that it must be at
// least 20 columns (characters) wide.
// we also specify the alignment here by choosing the option
// EXPAND_CENTER which defines that this field should expand in the
// horizontal direction when the parent expands to take up all the
// room allocated to it and it should remain vertically
// centered.
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(20));

// add the label for City state zip
addressBox.add(new ULCLabel("City, State, ZIP:"));

// create a Horizontal box to contain the City, State, Zip fields
ULCBoxPane cityStateZipBox = new ULCBoxPane(false);

// add the field for the city and specify that it must be at least 10
```

```

// columns wide and expand horizontally when the parent expands.
cityStateZipBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(10));

// add the field for the state and request that it must be at least 2
// columns wide and expand horizontally
cityStateZipBox.add(new ULCTextField(2));

// add the field for the zip and set it to be at least 5 columns wide
// and expand horizontally
cityStateZipBox.add(new ULCTextField(5));

// now add the cityStateZipBox to the parent addressBox and request
// that it expands horizontally
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, cityStateZipBox);

// add the label and field for the Country
// expand horizontally
addressBox.add(new ULCLabel("Country:"));
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(20));

// Finally we add the addressBox to the parent box/frame and set
// the frame to be visible to display the form. (code not shown)

```

Note that in all *add()* operations of the above example we have not specified the row and column position at which the component should be added. The *ULCBoxPane* widget automatically adds parts starting from the top left corner and adds the widgets in each column until it reaches the last column and then moves to the next row and so on. The order of adding the parts in this case is critical. If for any reason you are unable to build the form in a top-down, left-right order you can use the extended *set()* API of *ULCBoxPane* which takes the row and column within the box at which the component should be added.

The address form demonstrated the use of nested boxes as well as the cell alignment within the box.

A Simple Address Form Using Spanning

Let us implement the same form using spanning instead of nested boxes. The corresponding code snippet looks like this:

```

// Create a box with 4 columns instead
ULCBoxPane addressBox = new ULCBoxPane(4, 2);

addressBox.add(new ULCLabel("Street:"));
// let the field for street span 3 cells
addressBox.add(3, ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(20));

addressBox.add(new ULCLabel("City, State, ZIP:"));
// no nested box
// these fields are added to the box holding the complete address instead
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(10));
addressBox.add(new ULCTextField(2));
addressBox.add(new ULCTextField(5));

```

```
addressBox.add(new ULCLabel("Country:"));
// let the field for country span 3 cells as well
addressBox.add(3, ULCTextField.BOX_EXPAND_CENTER, new ULCTextField(20));
```

Note that we created a *ULCBoxPane* with only 2 rows but we are actually using 3 rows. This seems wrong and though it is not good practice it works because the *ULCBoxPane* automatically handles additional rows. If we were spanning in the horizontal direction and exceeded the configured column size it would not work since the box would wrap around at the last column and attempt to span to the next row. In fact, experienced ULC developers usually create a box using the following API:

```
// Create a box with n rows and 4 columns
ULCBoxPane addressBox = new ULCBoxPane(4, 0);
```

Since the box will automatically wrap around, we do not need to concern ourselves with the actual number of rows added.

4.5 Components

This section gives an overview of various ULC components with which a developer can construct rich GUIs. For more details see <Canoo RIA Suite Install Dir>/doc/apidoc for packages **com.ulcjava.base.application.***.

Buttons: *ULCButton*, *ULCCheckBox*, *ULCRadioButton*.

Text display and editing: *ULCLabel*, *ULCTextField*, *ULCTextArea*, *ULCPasswordField*.

Menus: *ULCMenu*, *ULCMenuItem*, *ULCPopupMenu*.

Visualizing/choosing values: *ULCProgressBar*, *ULCSlider*, *ULCSpinner*, *ULCComboBox*.

Choosing color: *ULCColorChooser*.

Viewing Data: *ULCList*, *ULCTable*, *ULCTableTree*, *ULCTree*.

Forms: *AbstractFormBuilder*.

Charts: *ULCChartPanel*, *ChartBuilder*.

Viewing HTML: *ULCHtmlPane*

Each component in general has attributes and events. An event is sent to the server to be processed by an event listener only if the component has a registered event listener. Please see corresponding Javadoc for more details.

4.6 How to format and validate displayed/entered values

ULC provides data type widgets for formatting values displayed on a label or text field, and for validating (syntactic) of values entered in a text field. These widgets, although defined on the server side, execute code exclusively on the client side thus saving on client-server roundtrip. Some of the widgets are: *ULCNumberDataType*, *ULCRegularExpressionDataType*, *ULCStringDataType*, *ULCPercentDataType* and *ULCDateDataType*.

Note that application logic specific semantic validation has to be coded within a component's event listener in the ULC application on the server side.

4.6.1 How to visualize formatting and validation errors

ULC provides *ULCAbstractErrorManager* and *ULCDefaultErrorManager* classes for managing and visualizing *ErrorObjects* for *ULCComponents*. *ErrorObjects* can be created on the server or on the client side. ULC DataTypes can be configured with an *ErrorManager* in order to display formatting and validation errors. The following snippet and picture show how to validate a date value:

```
Map errorMessages = new HashMap();
errorMessages.put(ErrorCodes.ERROR_CODE_BAD_FORMAT, "Invalid format");
ULCDefaultErrorManager errorManager =
    new ULCDefaultErrorManager(errorMessages, Color.pink,
        BorderFactory.createLineBorder(Color.red, 1));
fDateField = new ULCTextField(10);
fDateField.setDataType(new ULCDateDataType(errorManager));
```



4.7 How to enable / disable components

ULC provides widgets for enabling/disabling components. The enablers are configured in the ULC application on the server side but execute purely on client side thus optimizing on client-server roundtrips. *ULCComponent* has a method *setEnabled(IEnabler)*. Some ULC components and models which are capable of value change (e.g., *ULCTextField*, *ULCListSelectionModel*, etc.) are *IEnablers*. In addition, ULC enables logical combination of enablers through widgets like *ULCAndEnabler*, *ULCOrEnabler*, *ULCNotEnabler*, and *ULCXorEnabler*. *ULCMandatoryAndEnabler* can be used to indicate mandatory components. Use *ULCHasChangedEnabler* for enabling/disabling on value change based on multiple enabler components.

4.8 Events and Event Delivery Modes

ULC's event model is based on the same principles as the standard Java (AWT and Swing) event model. Action events, such as button click, scrolling, focus change, selection change, defined by widgets are handled in the ULC application on the server. The ULC application framework does not give access to low level events like mouse events. Instead, low level events are either handled locally on the client side (to avoid a server roundtrip) or converted to a semantic event like an action event, and then sent to the application on the server side. Events are sent from the client to the server only if there is a registered listener for this event on the server side.

Due to the distributed nature of ULC applications, event delivery and data updates from client to server may take a considerable amount of time. For this reason, ULC minimizes network traffic whenever possible. It also keeps the user interface as responsive as possible, e.g., dynamic loading of data does not block the user interface. When loading data (e.g., table rows), ULC uses asynchronous communication between the UI Engine and the ULC application. This allows ULC applications to stay responsive even in a wide area network where response times and network latency vary greatly.

Event dispatching (i.e., sending an event to the server and calling the corresponding listeners) is handled synchronously (i.e., the UI blocks till the roundtrip is over) by default in order to ensure data and user interface consistency. To increase the usability and responsiveness of a user interface, application developers can configure specific event listeners to be called asynchronously. However, developers must be aware that users might continue to interact with the user interface on the client while the event is dispatched to the server. The *ClientContext* class provides an API to further configure the event delivery modes for an application's widgets and models.

4.8.1 Event Delivery Modes

When sending events, the user interface must be blocked to ensure state consistency between client and server. The ULC framework does not completely block the client (e.g., repainting of the user interface is still performed) but merely blocks any user input (either by mouse or keyboard) until the server roundtrip triggered by the event has been completed and all side effects on the server and the respective updates on the client have been accomplished. In most situations, the user will not be aware of the fact that the user interface has been blocked for a short period of time.

To further improve the usability of a user interface, ULC widgets can be configured to deliver certain events in an asynchronous way. With such a configuration, users are able to continue to interact with the user interface while the event is being delivered to the application. Appropriate usage of this feature can significantly improve the usability. However, the application developer must ensure consistency by application-specific means.

Using the *ClientContext.setEventDeliveryMode()* method, events of a specific type and for a specific component can be configured to be delivered synchronously (i.e., the user interface is blocked) or asynchronously (i.e., the user can continue to interact with the user interface). By default, events are delivered synchronously.

The following example shows how to register an *ICollectionChangedListener* on the *ULCListSelectionModel* of a *ULCTable* and how to configure the *ULCListSelectionModel* to deliver *ListSelectionEvents* asynchronously to the server side.

```
table.getSelectionModel().addListSelectionListener(new StatusBarUpdater());
ClientContext.setEventDeliveryMode(table.getSelectionModel(),
                                   UlcEventCategories.LIST_SELECTION_EVENT_CATEGORY,
                                   UlcEventConstants.ASYNCHRONOUS_MODE);
```

4.8.2 Model Update Modes

Whenever tables, table trees, and trees are configured to provide in-place editing, the corresponding models (*ITableModel*, *ITableTreeModel*, and *ITreeModel*) are updated by default in *UlcEventConstants.DEFERRED_MODE*, i.e., the update is sent with the next event. To send the model update event immediately, choose *UlcEventConstants.SYNCHRONOUS_MODE*. In order to minimize network traffic *UlcEventConstants.DEFERRED_MODE* has been made default for the above model updates, i.e., these models are updated only when other events are sent to the server side (e.g., when the user clicks a button). When the user clicks on a button, the model is first updated, and then the application's action listeners are called.

Using the *ClientContext.setModelUpdateMode()* method, model updates for a specific model can be configured to be delivered to the server immediately or deferred (i.e., the update is sent with the next event).

The following example configures an *ITableModel* to be updated immediately after the user edits the corresponding table.

```
ClientContext.setModelUpdateMode(table.getModel(),
                                UlcEventConstants.SYNCHRONOUS_MODE);
```

4.9 Special features of ULCTable

The following sections describe some special features of *ULCTable*.

4.9.1 How to define a table based on a list of Java beans

Besides the conventional way of defining a table model to organize access to the table data (see *ITableModels* in the ULC API documentation), the *TableBinding* class provides the ability to bind a list of Java beans directly to a *ULCTable*. The access to the bean data is handled transparently through a table model that is internally created by the *TableBinding* class.

Each bean in the bean list is mapped to a single row in the table, the bean properties are mapped to the columns, and the property values of the beans represent the cells of the table row. The mapping of bean properties to table columns is done by *ColumnBinding* objects, which are created for each bean property automatically if the *autoCreateColumnFromBean* property of *TableBinding* is set, otherwise they have to be added manually for each property that should be represented by a table column.

The following sample code illustrates the usage of the *TableBinding* class together with a *ULCTable*:

```
// create a list of beans for e.g. through a database query
List<Person> persons = createPersonList();
ULCTable table = new ULCTable();

TableBinding tableBinding =
    TableBinding.createTableBindingFromBeanList(persons, table, Person.class, true);

tableBinding.addColumnBinding("lastName",

getResourceMap().getString("lastName.columnHeader").preferredWidth(150);
tableBinding.addColumnBinding("firstName",

getResourceMap().getString("firstName.columnHeader").preferredWidth(150);
// Suppose there is a boolean property and you would like to have CheckBox cell
renderer
tableBinding.addColumnBinding("vip",

getResourceMap().getString("vip.columnHeader").columnType(Boolean.class);
```

As shown in the sample code, the configuration of *ColumnBinding* objects is done by chaining method calls. These methods from the *ColumnBinding* class return the modified *ColumnBinding* object.

Internally the given list of Java beans is mapped to an *ITableModel* instance that is set on the specified *ULCTable*.

If no sorter or filter mechanism is set on the *ULCTable*, the order of the table rows is defined by the order of the beans in the given list.

If the list of Java beans provided while creating the *TableBinding* instance is an instance of *ObservableList*, the underlying table model will react on changes on this list. If a bean is removed, added, or changed, the *ULCTable* will be updated.

If the properties of the bound Java beans are bounded properties, the *ULCTable* will react on property changed events and will update the appropriate table cells.

4.9.2 How to enable table sorting

Setting a *TableRowSorter* for a given *ITableModel* enables *ULCTables* to sort and filter table data.

The *TableRowSorter* provides a mapping between the model row indices and the view row indices. A sort action is either triggered by a user interaction or if the underlying model gets updated.

The following sample demonstrates the usage of a *TableRowSorter*:

```
// create an instance of ITableModel
ITableModel tableModel = createTableModel();

ULCTable table = new ULCTable();

TableRowSorter tableRowSorter = new TableRowSorter(tableModel);
table.setRowSorter(tableRowSorter);
```

The *TableRowSorter* provides the possibility to enable or disable sorting of specified column and to set user defined comparators for the columns.

The specific comparator for a sort action on a selected column is chosen by the following rules:

1. If a user defined comparator is set for the selected column, this comparator is returned.
2. If the class returned by *getColumnClass* is *String*, a comparator is used that is returned by *Collator.getInstance*.
3. If the class returned by *getColumnClass* implements comparable, a default comparator is used to compare the objects of this column.
4. If no rule is matching, a comparator is used that is returned by *Collator.getInstance* and *String.valueOf()* is called on each object.

4.9.3 How to enable table filtering

Filtering on *ULCTables* is provided by *TableRowFilters*. The *TableRowFilter* has to be set on a *TableRowSorter*, which has been set on a given table.

The following sample shows how to enable filtering on a *ULCTable*:

```
ITableModel tableModel = createTableModel();
ULCTable table = new ULCTable(tableModel);

TableRowSorter tableRowSorter = new TableRowSorter(tableModel);
tableRowSorter.setRowFilter(TableRowFilter.regexFilter(".*ulc.*", new int[] {0, 1}));
table.setRowSorter(tableRowSorter);
```

The specified *TableRowFilter* filters all rows, which do not contain the string “ulc” in the first or second column.

The *TableRowFilter* class provides the following predefined filters which can be created using the appropriate factory methods:

- *AndFilter*
- *DateFilter*
- *NotFilter*
- *NumberFilter*
- *OrFilter*
- *RegexFilter*

The following sample shows a user defined *TableRowFilter* that filters all rows not containing the character 'A':

```
private class MyTableRowFilter extends AbstractFilter {
    MyTableRowFilter(int[] columns) {
        super(columns);
    }

    protected boolean include(RowEntry rowEntry, int index) {
        // returns the model value at the specified index for this row
        Object value = rowEntry.getValue(index);
        if (String.valueOf(value).contains("A")) {
            return true;
        }
        return false;
    }
}
```

4.10 Forms

The form component enables a bean to be viewed in a form. It takes care of the following aspects which need to be handled if a business object (bean) is bound to a user interface:

- layout of the components bound to the bean attributes
- value binding to the bean attributes
- input validation
- status feedback: missing or invalid input, dirty state (i.e. not yet stored input)
- calculation of dependent values (e.g. find corresponding city to entered ZIP)

The following sections explain in detail the various steps required to implement a form.

4.10.1 How to bind a form to a bean

To bind a Java bean to a form, the bean has to be adapted by a subclass of *FormModel*. The *FormModel* maintains the following presentation state per property:

- **error** - holds an *ErrorObject*.
- **enabled** - a boolean value that reflects the enabled or disabled state of a property.
- **mandatory** - a boolean value that reflects if the property is mandatory.
- **readonly** - a boolean value that reflects if the property is readonly.

Subclasses of *FormModel* can override two methods to handle the presentation state: the *initState()* method to set the initial state, and the *updatePresentationState()* method to change the presentation state after a property is set. The *calculate()* method is called after each property change. Override this method to update calculated fields. The following sample code shows how to bind a form to a bean and how to implement the *calculate()* method:

```
PersonFormModel personFormModel = new PersonFormModel(person);
PersonFormBuilder personFormBuilder = new PersonFormBuilder(personFormModel);

public static class PersonFormModel extends FormModel<Person> {
    public PersonFormModel(Person bean) {
        super(bean);
    }

    protected void calculate() {
        Person person = getBean();
    }
}
```

```

        if (hasChanged("zip")) {
            String city = toCity(person.getZip());
            setProperty("city", city);
        }
    }

    private String toCity(String zip) {
        return "Knichttown";
    }
}

```

4.10.2 How to define the form layout

A form is implemented by extending the *AbstractFormBuilder* class and passing the form model that wraps the Java bean as constructor argument. The *AbstractFormBuilder* provides the whole infrastructure that is needed to create the user interface container and to handle the layout of individual UI components.

Implement *AbstractFormBuilder.initForm()* to add and configure the UI components like shown in the following sample:

```

public static class PersonFormBuilder extends AbstractFormBuilder<PersonFormModel> {
    public PersonFormBuilder(PersonFormModel person) {
        super(person);
    }

    protected void initForm() {
        String generalTabText = getResourceMap().getString("general.Tab.text");
        String addressTabText = getResourceMap().getString("address.Tab.text");

        startTab(generalTabText);
        addTextField("firstName");
        addTextField("lastName").append(true); // append = add on same line
        addTextField("company");

        startTab(addressTabText);
        addTextField("street");
        addTextField("zip");
        addTextField("city").append(true); // append = add on same line
        addTextField("country");
    }
}

```

The form builder loads the label texts from its resource bundle. The label text resource keys are prefixed with *propertyName.Label*, e.g. *firstName.Label*. Of course it is possible to set other label properties via the resource bundles. See section 4.11 *Resource bundles* for more information on what is possible with resource bundles.

The resource bundle for our sample looks as follows:

```

general.Tab.text=General
firstName.Label.text=First Name, Last Name:
lastName.Label.text=
company.Label.text=Company:

address.Tab.text=Address
street.Label.text=Street:
zip.Label.text=ZIP, City:
city.Label.text=
country.Label.text=Country

```

This results in the following form layout. Note that the first name and last name text fields are rendered on the same row:



4.10.3 How to embed a form in a window

Use the *AbstractFormBuilder.getFormPane()* method to get the form's user interface element. You can use this user interface element wherever you like. However, the application framework comes with a ready to use form container that provides the following features:

- Standard **Reset** and **OK** buttons at the bottom
- The standard buttons are enabled only if there is user input and if this input is valid
- Possibility to add custom behavior to the standard buttons
- Possibility to add custom buttons

Do the following to embed the form container into a dialog:

```
PersonFormModel personFormModel = new PersonFormModel(person);
PersonFormBuilder personFormBuilder = new PersonFormBuilder(personFormModel);
BeanFormDialog<PersonFormModel> personDialog =
    new BeanFormDialog<PersonFormModel>(personFormBuilder);
dialog.setContentPane(personDialog.getContentPane());
```

The resulting dialog looks like the one in section 4.10.2 *How to define the form layout*.

4.10.4 How to add validation code to a form

A standard requirement for forms is user input validation. The application framework uses validators to validate user input. Validators are reusable classes that can validate a specific aspect on your bean. Usually, you either implement or use an existing validator for each user input validation requirement.

Define the validators to use in the *AbstractFormModel.createValidators()* method:

```
public static class PersonFormModel extends FormModel<Person> {
    public PersonFormModel(Person bean) {
        super(bean);
    }

    protected IValidator[] createValidators() {
        return new IValidator[] {
            new NameValidator("firstName"),
            new NameValidator("lastName")
        };
    }
}
```

The validator interface contains just a single *validate(FormModel)* method that the form model executes for each user input. The task of the validator is to check the validity of the form model. If the validator finds invalid or inconsistent data in the form model then it has to notify this back to the form model by setting a corresponding error via the form model's *setError()* method.

The application framework comes with a convenience validator that limits validation to single properties as opposed to the validation of the whole form model. Extend the *PropertyValidator* to validate a single property:

```
public static class NameValidator extends PropertyValidator<String> {
    private NameValidator(String propertyName) {
        super(new String[] { propertyName });
    }

    public String validateValue(String value) {
        char firstChar = value.charAt(0);
        if (Character.isDigit(firstChar)) {
            return "Name must not start with digit.";
        }

        return null;
    }
}
```

The validator in the previous code snippet validates a single property. The property to validate can be specified when constructing the validator. In the *validateValue()* method the validator either returns null, which means that the user input is valid, or the error text.

By default, the form component signals invalid user input with a red background and a red border around the corresponding user interface element. The error message is displayed as tool tip text when the user hovers the mouse over the user interface element for some time.



Note that the feedback for user input errors that are validated on the client side only is similar to the feedback for user input errors that are validated on the server-side in the form model. For more information on client-side only user input validation see the JavaDoc for *ULCTextField.setDataType()*.

4.11 Resource bundles

The following sections describe the use of resource bundles within ULC.

4.11.1 How to internationalize

ULC's application framework provides an easy to use infrastructure to deal with resource bundles. Name your user interface elements, and the application framework auto-injects arbitrary property values from resource bundles into your user interface elements. And to make things even easier, the ULC application framework creates default names for forms and tables so that you don't have to do anything at all in your application code.

Your application code looks as follows:

```
ULCLabel helloLabel = new ULCLabel();
helloLabel.setName("hello.Label");
```

HelloApplication.properties file:

```
hello.Label.text=Hello World!      # default text for the hello label
hello.Label.icon=icons/hello.png   # default icon for the hello label
```

HelloApplication_de.properties file:

```
hello.Label.text=Hallo Welt!       # German text for the hello label
hello.Label.icon=icons/hello_de.png # German icon for the hello label
```

The previous sample sets the name of a label to *hello.Label*. At application startup, the application framework now either injects into the label a text of **Hallo Welt!** (for the German locale) or a text of **Hello World!** (for all other locales).

4.11.2 How to use images

Besides strings, the resource bundle infrastructure of ULC's application framework handles resources like images, colors, fonts etc.

4.11.3 How to use more than one resource bundle

For real world projects you usually want to modularize your resource bundles. A typical approach is to have a resource bundle per class in your application. And corresponding to your class hierarchies you want to have resource bundles that extend or specialize other resource bundles. This functionality is already implemented in ULC's application framework.

Such groups of resource bundles are managed by the application framework's *ResourceMap* class. The *ResourceMap* class implements a chain of responsibility. If a resource cannot be found in the current *ResourceMap* then it is searched in the parent's *ResourceMap*.

Use the *ResourceManager* class to create *ResourceMap* chains for your class hierarchies. The *ResourceManager* class searches the resource bundle in the *resources* subpackage of the *class* package, e.g.

- com.canoo.A class => com/canoo/resources/A.properties file
- com.canoo.foo.B class => com/canoo/foo/resources/B.properties file

The *ApplicationContext* provides access to the *ResourceManager* method to get a *ResourceMap*. The basic method to create *ResourceMap* hierarchies is

```
public <T> ResourceMap getResourceMap(Class<? extends T> startClass, Class<T>
endClass, ResourceMap parent)
```

This retrieves the `ResourceMap` built from the `startClass.properties` along the class hierarchy to `endClass.properties`. The third parameter denotes the `ResourceMap` that is set as parent of the `endClass` resource map.

Additionally there are two convenience methods:

```
ResourceMap applicationResourceMap = getContext().getResourceMap();
```

Returns resource map of the application class hierarchy, from the running application class's properties up to `Application.properties`.

```
ResourceMap bClassUpToAClassResourceMap = getContext().getResourceMap(B.class, A.class);
```

Returns resource map of the class hierarchy, from `B.properties` up to `A.properties` with the application resource map as parent. If the `B` class is a direct subclass of the `A` class, then searching a resource in the `BClassUpToAClassResourceMap` resource map means first search the resource in the `B.properties` file and if not found then search in the `A.properties` file continuing in the running application's property file and up to finally `Application.properties`.

```
ResourceMap complexResourceMap = getContext().getResourceMap(X.class, Y.class, bClassUpToAClassResourceMap);
```

Builds an even more complex map from `X.properties` → `Y.properties` → `B.properties` → `A.properties` → `MyApplication.properties` → `Application.properties`.

4.11.4 How to define styles

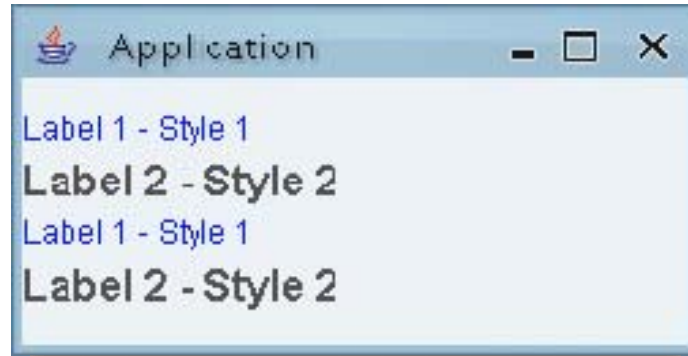
Use resource bundles to ensure a uniform look of your user interface elements. To do so define styles inside the resource bundles and link components to styles by setting a synthetic style property.

```
# link components to styles
label1.Label.style=style1
label2.Label.style=style2
label3.Label.style=style1
label4.Label.style=style2

# define style1
style1.font=Arial-PLAIN-12
style1.foreground=blue

# define style2
style2.font=Arial-BOLD-16
style2.foreground=darkGray
```

The previous resource bundle links *label1* and *label3* to *style1*, and *label2* and *label4* to *style2*, whereas *style1* is blue and *style2* is large dark grey. This results in the following output:



As a bonus you can define default property values for arbitrary user interface components in your resource bundles. Default property values have a key of *componentClassName.PropertyName*, e.g. *ULCLabel.background*.

The evaluation order for resource bundle properties is as follows:

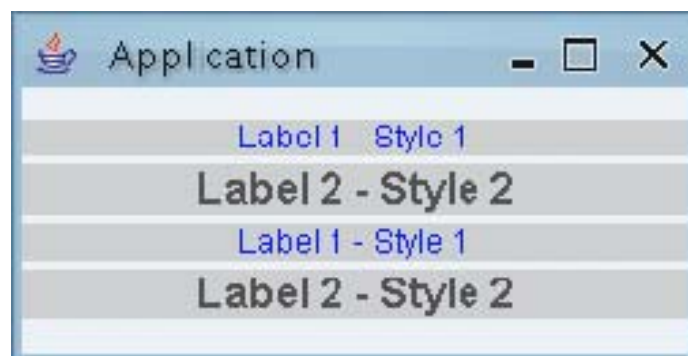
1. Instance properties => *componentName.propertyName*
2. Style properties => *styleName.propertyName*
3. Default properties => *componentClassName.propertyName*

Adding the following lines to the previous resource bundle:

```
# define ULCComponent default properties
ULCComponent.opaque=true

# define ULCLabel default properties
ULCLabel.horizontalAlignment=#{ULCLabel.CENTER}
ULCLabel.background=lightGray
```

Results in the following picture:



4.12 Client Access

This section describes how to access the client machine. The necessary methods are provided as static members of the *ClientContext* class.

4.12.1 How to install a File Service

You can use `ULCApplicationConfiguration.xml` file to specify the class that provides file service (see *File Service* in the [ULC Architecture Guide](#)) for transferring files between the client and the server:

```
<?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:applicationName>
    com.canoo.person.view.PersonApplication
  </ulc:applicationName>
  <ulc:fileService>
    <ulc:standardFileService>AllPermissions</ulc:standardFileService>
  </ulc:fileService>
</ulc:ULCApplicationConfiguration>
```

The above example configures the *AllPermissionFileService* as the standard file service. The *AllPermissionsFileService* has no file access restrictions but requires that the client runs outside of the client environment's sandbox. How you disable the sandbox depends on the client environment:

- **Applet**
Sign all the client-side jar files.
- **Java Web Start**
Sign the all the client-side jar files and instruct Java Web Start to run your application outside of the sandbox with help of the all-permissions element in the JNLP file.

4.12.2 Choosing a File Located on the Client Machine

Choosing a file located on the client machine is a two-step operation. First the static method *chooseFile()* in the *ClientContext* class has to be called. This method has to be provided with an *IFileChooseHandler* argument. In a second step the ULC framework invokes a hook in the provided *IFileChooseHandler* when the selection has been completed. Salient features:

- The *IFileChooseHandler* interface provides hooks for successful selections and for error handling.
- The user chooses the file with a file chooser dialog that can be configured with a *FileChooserConfig* object.
- When choosing a file with a file chooser dialog, a *ULCComponent* is configured as the parent for the file chooser dialog
- The chosen filename is provided by the on success hook of the *IFileChooseHandler*

Not all client environments support client-side file operations. For instance, applets running in the default applet sandbox are not allowed to read and write files.

In the following code snippet the application asks the user to choose a text file located on the client machine. The only selectable files are those with extension ".txt".

```
// parent for the file chooser
ULCFrame frame = new ULCFrame();
// ...

FileChooserConfig fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Choose a text file");
```

```

fcConfig.addFileFilterConfig(new FileChooserConfig.FileFilterConfig(
new String[]{"*.txt"}, "text files (*.txt)"));

ClientContext.chooseFile(new IFileChooseHandler() {
    public void onSuccess(String filePath) {
        System.out.println("Chosen text file is: " + filePath);
        // return the file path
    }

    public void onFailure(int reason, String description) {
        // show an alert informing the operation has failed
    }
}, fcConfig, frame);

```

In the next example the application asks the user to choose a directory (only directories are shown).

```

fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Choose a directory");
fcConfig.setSelectionMode(FileChooserConfig.DIRECTORIES_ONLY);

ClientContext.chooseFile(new IFileChooseHandler() {
    public void onSuccess(String filePath) {
        System.out.println("Chosen directory is: " + filePath);
        // return the file path
    }

    public void onFailure(int reason, String description){
        // show an alert informing the operation has failed
    }
}, fcConfig, frame);

```

4.12.3 Reading from a File Located on the Client Machine

Reading from a file located on the client machine is a two-step operation. The first step is to initiate the client file download by calling the static *loadFile()* method in the *ClientContext* class. This method has to be provided with an *IFileLoadHandler* argument. In a second step the ULC framework invokes a hook in the provided *IFileLoadHandler* when the download has been completed. Salient features:

- The *IFileLoadHandler* interface provides hooks for successful downloads and for error handling. For successful downloads the file data can be accessed by means of a provided input stream.
- The file to be downloaded can be specified as a client file pathname. Alternatively, the application lets the user choose the file with a file chooser dialog that is configured with a *FileChooserConfig* object. With a file chooser dialog only the file selection mode *FileChooserConfig.FILES_ONLY* is allowed.

Not all client environments support client-side file operations. For instance, applets running in the default applet sandbox are not allowed to read and write files.

In the following example the application attempts to load a property file located on the client machine.

```

ClientContext.loadFile(new IFileLoadHandler() {
    public void onSuccess(InputStream in, String filePath) {
        Properties p = new Properties();
        try {
            p.load(in);
        } catch (IOException e) {
        }
        // process properties
    }

    public void onFailure(int reason, String description) {
        // ignored - the default properties will be used
    }
}

```

```
}, "C:\\myapp\\myapp.properties");
```

In the next example the application asks the user to choose a file containing some report. The only selectable files are those with extension ".rep".

```
// parent for the file chooser
ULCFrame frame = new ULCFrame();
// ...

FileChooserConfig fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Choose a report to be loaded");
fcConfig.addFileFilterConfig(new
    FileChooserConfig.FileFilterConfig(new String[]{"*.rep"},
                                        "report files (*.rep)"));

ClientContext.loadFile(new IFileLoadHandler() {
    public void onSuccess(InputStream in, String filePath) {
        // read the report from the input stream and process it
    }

    public void onFailure(int reason, String description) {
        // show an alert informing the operation has failed
    }
}, fcConfig, frame);
```

4.12.4 Writing to a File Located on the Client Machine

Writing to a file located on the client machine is a three-step operation. The first step is to initiate the client file upload by calling the static *storeFile()* method in the *ClientContext* class. This method has to be provided with an *IFileStoreHandler* argument. To collect the data to be uploaded, the ULC framework invokes a hook in the provided *IFileStoreHandler*. Finally, another hook in *IFileStoreHandler* is called when the upload has finished. Salient features:

- The data to be uploaded has to be written to an output stream in the *prepareFile()* hook of the *IFileStoreHandler*. The method *prepareFile()* may throw an exception if the data cannot be written. In this case, the upload process will be terminated.
- The *IFileStoreHandler* interface provides hooks for successful uploads and for error handling.
- The file to be written to can be specified as a client file pathname. Alternatively the application can let the user choose the file with a configurable file chooser dialog.

Not all client environments support client-side file operations. For instance, applets running in the default applet sandbox are not allowed to read and write files.

In the following example the application attempts to save a property file on the client machine:

```
try {
    ClientContext.storeFile(new IFileStoreHandler() {
        public void prepareFile(OutputStream data) throws Exception {
            fProperties.store(data, "myapplication properties");
        }

        public void onSuccess(String filePath) {
            // bring up an alert reporting the success
        }

        public void onFailure(int reason, String description) {
            // bring up an alert reporting the failure
        }
    }, "C:\\myapp\\myapp.properties");
} catch (Exception e) {
    // handle any exceptions that may have occurred, e.g., in prepareFile()
}
```

In the next example the application asks the user to choose a place and filename under which an exported report will be saved:

```
// parent for the file chooser
ULCFrame frame = new ULCFrame();
// ...

FileChooserConfig fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Where would you like to save the report?");

try {
    ClientContext.storeFile(new IFileStoreHandler() {
        public void prepareFile(OutputStream data) throws Exception {
            // write report data to the 'data' stream
        }

        public void onSuccess(String filePath) {
            // bring up an alert reporting the success
        }

        public void onFailure(int reason, String description) {
            // bring up an alert reporting the failure
        }
    }, fcConfig, frame);
} catch (Exception e) {
    // handle any exceptions that may have occurred, e.g., in prepareFile()
}
```

4.12.5 Showing a Document in a Web Browser

IBrowserService is used to access a web browser on the client's machine. This service can be controlled by an application to display documents in a browser, e.g., an HTML page or, if an appropriate plug-in is installed, a PDF document. The second use case for the browser service is the help facility of ULC components.

Showing a given URL in a web browser can be achieved by calling the static method *ClientContext.showDocument()*. How the web browser is accessed and what URLs are accepted depends on the configuration of the client environment, i.e., on the platform and on the installed browser service. Using the *AppletBrowserService* or the *JnlpBrowserService*, for instance, only valid protocols (i.e., protocols for which there is a protocol handler installed) are allowed in the document's URL.

Irrespective of the installed browser service, the security manager installed on the client (e.g., a jnlp or applet security manager) might disallow security-critical operations such as reading local files. Salient features:

- Shows a given URL in a web browser.
- One variant of the *showDocument()* method defines the frame target name as defined in the HTML 4.01 specification¹. This target argument is only interpreted in the case of the UI Engine running as an applet; it is ignored otherwise.
- It is not possible to track whether displaying a document in the browser was successful or not.
- *AllPermissionsBrowserService* can be used to overcome the security manager constraints.

The following example outlines the specification of browser service in **ULCApplicationConfiguration.xml** and usage of *ClientContext* to display a document in a web browser.

¹ <http://www.w3.org/TR/html4/>

Specification:

```
<ulc:browserService>
    <ulc:standardBrowserService>
        AllPermissions
    </ulc:standardBrowserService>
</ulc:browserService>
```

Code:

```
final ULCTextField urlField = new ULCTextField(30);

urlField.addActionListener(new IActionListener() {
    public void actionPerformed(ActionEvent e) {
        ClientContext.showDocument(urlField.getText());
    }
});
```

4.12.6 Accessing the Client Environment

The *ClientContext* class also provides methods to access information about the client system environment. Salient features:

- *getAddress()* returns the client IP address
- *getAvailableFontFamilyNames()* returns an array containing the names of all font families.
- *getHost()* returns the client host name
- *getLocale()* returns the client locale
- *getLookAndFeelName()* returns the client's look and feel name
- *getScreenHeight()*, *getScreenResolution()*, *getScreenWidth()* return the client screen attributes
- *getSystemColor(String key)* returns the system color for the specified key. See the *SystemColor* class for all valid keys.
- *getSystemPropertyNames()* returns an array containing the names of all client system properties, use *getSystemProperty(String)* to retrieve individual values.
- *getTimeZone()* returns the client time zone.
- *getUserParameterNames()* returns an array containing the names of all user parameters, use *getUserParameter(String)* to retrieve individual values.

ULC applications can be easily localized by using the standard Java internationalization support. The resource bundle is retrieved based upon the connecting client's *Locale* object:

```
Locale clientLocale = ClientContext.getLocale();
ResourceBundle myResources =
    ResourceBundle.getBundle("MyResources", clientLocale);
...
ULCButton okButton =
    new ULCButton(myResources.getString("okButtonLabel"));
```

4.12.7 Accessing the Client's UI Defaults

It is possible for ULC applications to access the client's most important UI defaults for which there is no API available (in particular, all default borders, colors, fonts, insets, and integer-valued properties). Access to these default values is especially valuable when creating customized cell renderers for tables, trees, etc. but they also prove useful for adjusting and fine-tuning an application's appearance. Note that a ULC application cannot change these defaults.

Initially, i.e., as soon as the connection from the client to the ULC application is established, all colors, fonts, insets, and integer-valued UI defaults are transferred to the

application. In this way, they are immediately available to the application and can already be used for the very first user interface window. Therefore, any changes to the UI defaults have to be made by the launcher *before* the connection is initiated. Salient features:

- `getBorder(String key)` returns the UI default border for the specified key.
- `getColor(String key)` returns the UI default color for the specified key.
- `getFont(String key)` returns the UI default font for the specified key.
- `getInsets(String key)` returns the UI default insets for the specified key.
- `getInt(String key)` returns the UI default integer values for the specified key.

The key is made up of the property prefix defined in the swing UI classes and the property name separated with a dot (*propertyPrefix* + . + *propertyName*, e.g. *CheckBox.foreground*), or just property name if there is no component name (e.g. *activeCaption*).

In the following code snippet, a custom renderer is installed on a *ULCComboBox*. A label is used as the rendering component to right-align the *Double* values in the drop-down list. However, since a label does not change its background and foreground colors when it is selected, we set these values explicitly according to the client's default selection foreground and background colors of *ULCComboBox*. There is no API available on the *ULCComboBox* to retrieve these default values, therefore we use the static methods in the *ClientContext* class.

```
Double[] values = getAvailableValues();
ULCComboBox comboBox = new ULCComboBox(values);
comboBox.setRenderer(new NumberRenderer());
...
private static class NumberRenderer implements IComboBoxCellRenderer {
    private ULCLabel fLabel;

    public NumberRenderer() {
        fLabel = new ULCLabel();
        fLabel.setHorizontalAlignment(ULCLabel.RIGHT);
    }

    public IRendererComponent getComboBoxCellRendererComponent(
        ULCComboBox comboBox, Object value, boolean isSelected, int row) {

        fLabel.setForeground(isSelected ?
            ClientContext.getColor("ComboBox.selectionForeground") :
            comboBox.getForeground());
        fLabel.setBackground(isSelected ?
            ClientContext.getColor("ComboBox.selectionBackground") :
            comboBox.getBackground());
        return fLabel;
    }
}
```

4.12.8 How to send messages from server to client

The message service *IMessageService* is used to respond to messages sent from the application using the *sendMessage()* API of the *ClientContext*. There is no default implementation provided in the standard release as there are a wide variety of possible implementations, ranging from controlling client components, starting and controlling other client sessions to interacting with the client environment, e.g., using LiveConnect and JavaScript for multichannel applications.

IMessageService implementation:

```
public class MyMessageService implements IMessageService {
    public void handleMessage(String msg) {
```

```

        System.out.println("Got message from server : " + msg);
    }
}

```

IMessageService specification in **ULCApplicationConfiguration.xml**:

```

<ulc:messageServiceClassName>
    MyMessageService
</ulc:messageServiceClassName>

```

Sending message from ULC application:

```

public class MyApp extends SingleFrameApplication {
    @Override
    protected void startup() {
        super.startup();
        ClientContext.sendMessage("message from server");
    }

    public static void main(String[] args) {
        DevelopmentRunner.run();
    }
}

```

4.13 Container Access

The *ApplicationContext* class (section 4.2) provides an API to access container runtime information that is independent of the actual underlying container that the ULC application is running in.

In addition to these standard services, a specific implementation of a container adapter may offer additional services and access to container specific information.

Important note: A standard ULC application should not use container specific APIs, only applications with special requirements are supposed to do so. As these APIs are limited to applications being deployed and run in specific containers, you must be aware that using this API restricts deployment to such containers only. Also, an in-depth knowledge of the underlying container and ULC architecture is required, as incorrect manipulation of the objects returned by this API may interfere with the ULC runtime system.

4.13.1 Servlet Container Access

The ULC servlet container adapter offers an API to access servlet container specific runtime information. For instance, this API can be used to access the underlying *HttpSession* object that a ULC application session resides in. Using this approach, HTML-ULC multichannel applications can be written to present the same session data simultaneously in the ULC application client and in a web browser. Salient features:

The *com.ulcjava.container.servlet.application.ServletContainerContext* class provides access to the following information:

- The *javax.servlet.http.HttpSession* object the ULC application session resides in.
- The *javax.servlet.ServletContext* object corresponding to the web application that the ULC servlet container adapter was configured to be part of.
- The HTTP servlet request and response object of the current server roundtrip.

The following example shows a servlet displaying a session attribute and a ULC application that changes the same attribute directly in the *HttpSession* object. At deployment time it must be ensured that both the servlet and the ULC application (i.e., the servlet container adapter servlet) share the same session. More specifically, they

must be part of the same web application and it must be ensured that session tracking information is sustained between the UI Engine (running as applet) connecting to the servlet engine and the servlet rendering the HTML page.

The displaying servlet can be implemented as follows:

```
public class MultiChannelServlet extends HttpServlet {
    protected void doGet(HttpServletRequest httpRequest,
        HttpServletResponse httpResponse)
        throws IOException {
        httpResponse.setContentType("text/html");

        PrintWriter out = httpResponse.getWriter();
        out.println("<HTML><BODY>");
        out.println("Text: " +
            httpRequest.getSession().getAttribute("text"));
        out.println("</BODY></HTML>");
    }
}
```

Using the *ServletContainerContext* class, the ULC application can directly access the attributes of the *HttpSession* that the application is running in:

```
public class MultiChannelApplication extends AbstractApplication {
    private ULCTextField fField;
    public void start() {
        ULCAppletPane contentPane = ULCAppletPane.getInstance();
        ULCBoxPane boxPane = new ULCBoxPane(true);

        String text = (String)ServletContainerContext.
            getHttpSession().getAttribute("text");
        fField = new ULCTextField(text, 20);
        boxPane.add(fField);

        ULCButton saveButton = new ULCButton("Save");
        saveButton.addActionListener(new IActionListener() {
            public void actionPerformed(ActionEvent event) {
                ServletContainerContext.getHttpSession().
                    setAttribute("text", fField.getText());
            }
        });
        boxPane.add(saveButton);
        contentPane.add(boxPane);
        contentPane.setVisible(true);
    }
}
```

4.13.2 EJB Container Access

The EJB container adapter implementation currently does not offer additional container specific services. Future releases may add such functionality.

4.14 Drag & Drop

ULC provides support for Drag & Drop operations. A Drag & Drop operation is started with a drag gesture (e.g. click an object and drag the mouse) on the drag source component. The dragged object can subsequently be dropped on a drop target component.

The following ULC components support Drag & Drop operations: *ULCLabel*, *ULCList*, *ULCPasswordField*, *ULCTable*, *ULCTableTree*, *ULCTextArea*, *ULCTextField*, and *ULCTree*.

4.14.1 Drag & Drop Operation Overview

When a Drag & Drop operation is started with an appropriate drag gesture, the ULC component that acts as the drag source creates a drag data object that describes the component-specific items that were selected at the point the drag started. While the selected items are dragged from the drag source towards the drop target (the mouse button is pressed and the mouse is being moved), the drag data object created by the drag source is stored within a *Transferable* instance. In addition, the cursor type of the mouse pointer is continuously being updated depending on the current possible drop target and the actions supported both by the drag source and the drop target.

As soon as the dragged items are being dropped on a ULC component that acts as the drop target, the drop target creates a drop data object that contains a description of the location where the drop occurred. The *Transferable* instance is completed with the drop data object and is sent to the server. On the server-side, the appropriate *TransferHandler* instance that is registered with each ULC component that supports Drag & Drop operations is being notified. First, the *importData()* method of the *TransferHandler* for the drop target ULC component is called to import the data provided by the drag source. During this step, items dragged from the drag source to the drop target can be added to the drop target ULC component. Second, the *exportDone()* method of the *TransferHandler* for the drag source is called to complete the data export. During this step, items moved from the drag source to the drop target can be removed from the drag source ULC component. The actual implementation of the *importData()* and *exportDone()* method depends on the application-specific *TransferHandler* implementation.

4.14.2 Configuring DnD

Drag & Drop support is configured using the following methods, located on each ULC component that supports Drag & Drop:

Method	Description
<i>setDragEnabled(boolean)</i>	Enables or disables the drag operation support for a specific ULC component. If enabled, a Drag & Drop operation can be started from the ULC component. If not set otherwise, a default implementation of <i>TransferHandler</i> is used to control the Drag & Drop operation. If disabled, dragged items from other ULC components can still be dropped onto the ULC component that has drag support disabled.
<i>setTransferHandler(TransferHandler)</i>	Sets the <i>TransferHandler</i> instance that handles data export on drag and data import on drop and thus configures the behavior of the component concerning Drag & Drop operations. For further information on the <i>TransferHandler</i> .

ULC provides a default implementation of the *TransferHandler* for each component that supports Drag & Drop. The default implementation is limited to the exchange of plain text data. The operations supported by each component are listed in the following table:

Component	Drag (Data Export)	Drop (Data Import)
<i>ULCLabel</i>	text/plain	-
<i>ULCList</i>	text/plain	-
<i>ULCPasswordField</i>	-	text/plain
<i>ULCTable</i>	text/plain	-
<i>ULCTableTree</i>	text/plain	-
<i>ULCTextArea</i>	text/plain	text/plain
<i>ULCTextField</i>	text/plain	text/plain
<i>ULCTree</i>	text/plain	-

This default behavior also applies to Drag & Drop operations between a specific ULC application and an external application.

The following example shows how to turn on Drag & Drop support for a *ULCList* component. The default implementation for the *TransferHandler* is used.

```
ULCList list = new ULCList(new String[]{"one", "two", "three"});
list.setDragEnabled(true);
```

4.14.3 Transfer Data

Data is transferred between the drag source and the drop component using the *Transferable* class. A *Transferable* is a container that contains data objects for specific data.

4.14.4 DataFlavor

A *DataFlavor* instance acts as a key for the associated transfer data. ULC uses the following types of *DataFlavor*:

Data Flavor	Description
DRAG_FLAVOR	DataFlavor that is used to describe the data exported by the ULC component acting as drag source. The data associated with this DataFlavor contains the drag source ULC component and the items selected for dragging.
DROP_FLAVOR	DataFlavor that is used to describe the data exported by the ULC component acting as drop target. The associated data contains the drop target ULC component and information on the location where the item has been dropped.

4.14.5 IDnDData

During a Drag & Drop operation, data exported by the drag source and imported by the drop target is represented as an instance/subclass of *IDnDData*. Both the ULC component that acts as drag source and the component acting as drop target provide an *IDnDData* object. Conceptually, the *IDnDData* objects contain the following information:

- For the drag source, the *IDnDData* object contains information on the drag source component, and on the items that were selected on and dragged from the drag source. The items selected for dragging are described by component-specific data

(such as selected indices, rows, columns, tree paths, text positions or text parts) that is contained in the *IDnDDData* object created by the drag source ULC component.

- For the drop target, the *IDnDDData* contains information on the drop target component and location where the dragged items have been dropped. The drop location is described by component-specific data (such as selected index, row, column, tree path, or text positions) that is contained in the *IDnDDData* object created by the drop target ULC component.

ULC uses the following implementations of the *IDnDDData* interface:

IDnDDData	Component	Structure
DnDLabelData	<i>ULCLabel</i>	<ul style="list-style-type: none"> • Component • label text
DnDTableData	<i>ULCTable</i>	<ul style="list-style-type: none"> • component • selected rows • selected columns
DnDTableTreeData	<i>ULCTablTree</i>	<ul style="list-style-type: none"> • component • selected tree paths • selected columns
DnDTextData	<i>ULCTextArea</i> <i>ULCTextField</i> <i>ULCPasswordField</i>	<ul style="list-style-type: none"> • component • text position • text
DnDTreeData	<i>ULCTree</i>	<ul style="list-style-type: none"> • component • selected tree paths
DnDExternalData	external applications	<ul style="list-style-type: none"> • text represenation of external data
DnDFileListData	operating system	<ul style="list-style-type: none"> • list of file paths

4.14.6 TransferHandler

The *TransferHandler* is used to configure the behavior of a specific ULC component during Drag & Drop operations. It is also responsible for handling data import and export. Typically, each ULC component has its own instance of *TransferHandler*. The *TransferHandler* class defines the following methods:

Method	Description
<i>getSourceActions()</i>	<p>Returns the Drag & Drop actions supported by the ULC component when acting as drag source. Valid actions are:</p> <ul style="list-style-type: none"> • <i>TransferHandler.ACTION_NONE</i> • <i>TransferHandler.ACTION_COPY</i> • <i>TransferHandler.ACTION_MOVE</i> • <i>TransferHandler.ACTION_LINK</i> <p>or a combination of them.</p>
<i>getTargetActions()</i>	<p>Returns the Drag & Drop actions supported by the ULC component when acting as drop target. Valid actions are:</p>

	<ul style="list-style-type: none"> • <i>TransferHandler.ACTION_NONE</i> • <i>TransferHandler.ACTION_COPY</i> • <i>TransferHandler.ACTION_MOVE</i> • <i>TransferHandler.ACTION_LINK</i> or a combination of them.
<i>importData(...)</i>	Called by the ULC framework whenever a drop operation on a specific ULC component occurs. This method is used to import dragged data into the ULC component acting as drop target.
<i>exportDone(...)</i>	Called by the ULC framework whenever a drop operation has completed. This callback method is used to modify the ULC component acting as drag source, e.g. for deleting moved items.

The following example shows how to turn on and configure the Drag & Drop support with a custom *TransferHandler* implementation for a *ULCList*.

```
public class MyTransferHandler extends TransferHandler {
    public int getSourceActions(ULCComponent component) {
        // allow copy and move
        return TransferHandler.ACTION_COPY | TransferHandler.ACTION_MOVE;
    }

    public boolean importData(ULCComponent target, Transferable t) {
        // get transfer data exported by the drag source
        // (here, it must be a ULCList component)
        Object dragData = t.getTransferData(DataFlavor.DRAG_FLAVOR);
        DnDListData listDragData = (DnDListData)dragData;

        // get source list and the indices of dragged list items
        ULCList sourceList = listDragData.getList();
        int[] draggedListIndices = listDragData.getIndices();

        // get transfer data exported by the drop target
        // (must be a ULCList component)
        Object dropData = t.getTransferData(DataFlavor.DROP_FLAVOR);
        DnDListData listDropData = (DnDListData)dropData;

        // get index of the item at the drop location
        int targetListIndex = listDropData.getIndices()[0];

        // ... handle data import

        // return whether import was successful or not
        return true;
    }

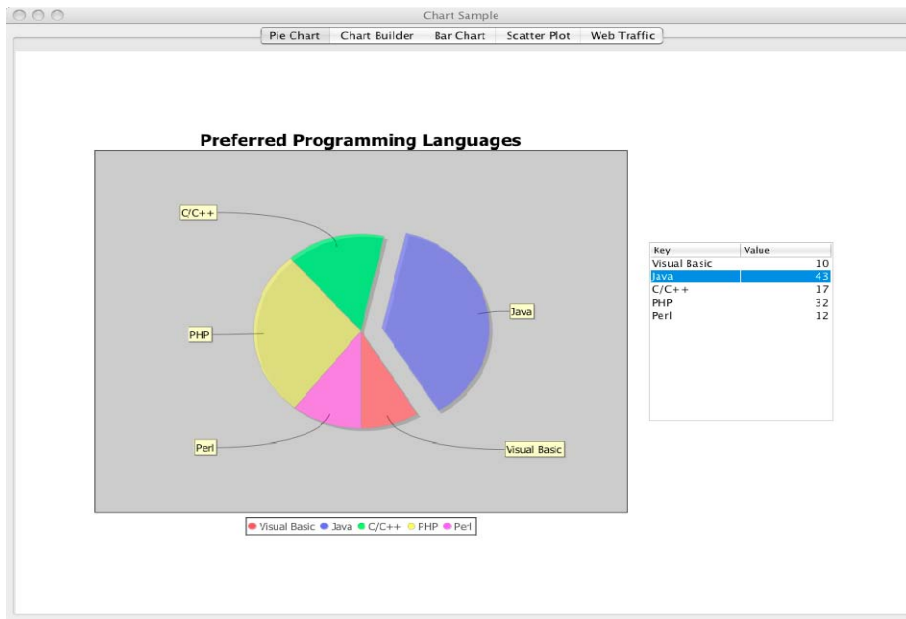
    public void exportDone(ULCComponent src, Transferable t, int action) {
        // check if items have been moved from drag source to
        // drop target and remove items from the drag source list.
        if (action == TransferHandler.ACTION_MOVE) {
            Object dragData = t.getTransferData(DataFlavor.DRAG_FLAVOR);
            DnDListData listDragData = (DnDListData)dragData;
            ULCList sourceList = listDragData.getList();
            int[] movedListIndices = listDragData.getIndices();

            // ... remove items from the list
        }
    }
}
```

4.15 Creating Charts

Package *com.ulcjava.base.server.chart* 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, the ULC release contains a few sample charting applications.

4.15.1 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.

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

The easiest way to create a chart is to use *com.ulcjava.base.server.chart.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 ULChartPanel<IPieDataset<String,Integer>>
createChartPanel(ChartBuilder chartBuilder) {
    ULChartPanel<IPieDataset<String, Integer>> chartPanel =
        new ULChartPanel<IPieDataset<String, Integer>>(createChartBuilder());
    chartPanel.setDataset(PieChartSampleApplication.createDefaultPieDataset());
    ULChartSelectionModel chartSelectionModel =
        chartPanel.getChartSelectionModel();
    chartSelectionModel.setSelectionMode(ULChartSelectionModel.MULTIPLE_SELECTION);
    chartPanel.setPreferredSize(new Dimension(600, 500));
    return chartPanel;
}

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

4.16 Creating Animation Effects

ULC provides animation support thanks to the package *com.ulcjava.base.server.animation*. This is an integration of the core concepts from the [TimingFramework](#), a library for making Java animation and timing-based control easier. This package enables to animate a property of a component. An animation is executed

completely on the client side but it is set up and configured from the server side. Since the animation exclusively takes place on the client side, the targeted properties should belong to the UI half-object or to its underlying basic object.

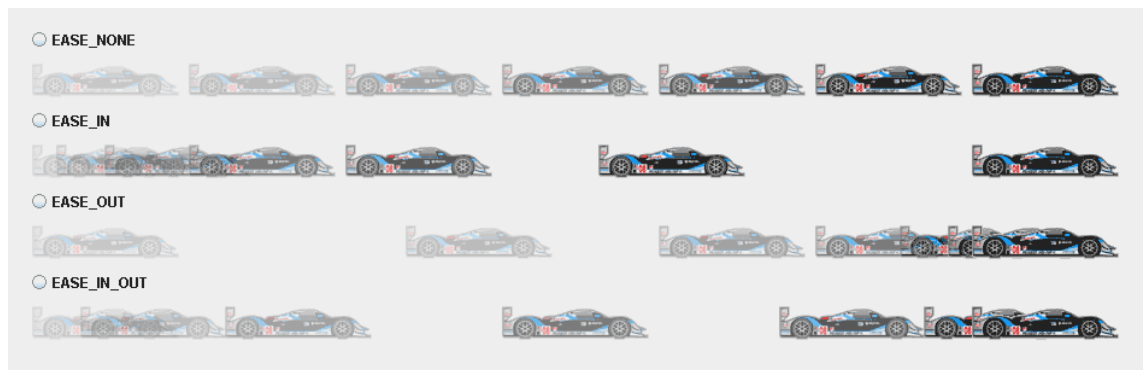
The [TimingFramework](#) has built-in support for interpolation between numeric properties (e.g. position, size, slider value, and such), colors, or some shapes. It also supports non-linear interpolation, infinite repeat count, or multiple targets. Moreover, triggers allow for starting animations based on actions, mouse events, focus events, and events from other animations. The latter enables to chain multiple animations.

For more information, please have a look at the [TimingFramework API](#) and the ULC animation package Javadoc. An animation demo is also provided in the ULCSample application.

4.16.1 How to create a simple animation

The main animation class is *ULCAimator*, which sets the main properties for animations :

- Duration: An integer value expressed in milliseconds. As a convenience, one may use the *Duration* class constants *SLOW*, *NORMAL* and *FAST*.
- Interpolator: The [TimingFramework](#) provides an *Interpolator* interface which is implemented for accelerating and decelerating behaviors. Once again, there is a convenience class called *Easing*, which predefines the following interpolation constants: *EASE_NONE* (linear), *EASE_IN* (accelerating), *EASE_OUT* (decelerating), and *EASE_IN_OUT* (both accelerating at the start and decelerating at the end). The following picture depicts various Easing options.



- Target: In order to animate a property of a component, the *ULCAimator*'s constructor also takes a *ULCComponent*, a property name, and a series of parameters which are the values for this property over time duration.

Other optional properties can be set prior to starting the animation, such as *repeatCount* (which can be *INFINITE* or even fractional), *repeatBehavior* (*LOOP* or *REVERSE* when cycles repeat), or *endBehavior* (*HOLD* the final value or *RESET*).

The example below shows how to create an animation that lasts for 5 seconds, interpolates linearly, and changes a label's background color from red to green. In addition, it will also repeat three times and run in reverse-mode every two cycles.

```
ULCLabel label = new ULCLabel();  
label.setOpaque(true);
```

```

label.setHorizontalAlignment(ULCLabel.CENTER);
label.setVerticalAlignment(ULCLabel.CENTER);
label.setText("Explicit Animation");

final ULCAnimator animator = new ULCAnimator(Duration.SLOW, Easing.EASE_NONE, label,
"background", Color.red, Color.green);
animator.setRepeatBehavior(RepeatBehavior.REVERSE);
animator.setRepeatCount(3);
ULCButton button = new ULCButton("Animate");
button.addActionListener(new IActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        if (!animator.isRunning()) {
            animator.start();
        } else {
            animator.stop();
        }
    }
});

```

Please note that animations do not start automatically. To control the animation of a *ULCAnimator*, you can use methods like *start()*, *stop()*, *pause()*, *resume()*, or *cancel()* or call the *isRunning()* method to get its current state.

4.16.2 How to use multiple targets and animation triggers

The first example showed how to create and run a simple animation. However, animations can also have multiple targets, and they can be started automatically based on some events. To do this, you will use either the *addActionTrigger()*, *addMouseTrigger()*, *addFocusTrigger()*, or *addTimingTrigger()* methods from *ULCAnimator*. The example below shows how to create an animation that lasts for 1 second, ends smoothly, and changes a label's foreground color from white to black. This animator is triggered only when a *CLICK* event happens on the button. In addition, this animation also changes a slider's position.

```

ULCLabel label2 = new ULCLabel();
label2.setOpaque(true);
label2.setHorizontalAlignment(ULCLabel.CENTER);
label2.setVerticalAlignment(ULCLabel.CENTER);
label2.setText("Trigger Animation");
label2.setBackground(Color.lightGray);

ULCSlider slider = new ULCSlider(ULCSlider.HORIZONTAL, 0, 100, 50);

final ULCAnimator animator2 = new ULCAnimator(Duration.NORMAL, Easing.EASE_OUT,
label2, "foreground", Color.white, Color.black);

animator2.addTarget(slider, "value", 0, 100);

animator2.addMouseTrigger(button, MouseEvent.CLICK, false);

```

4.16.3 How to use custom TimingTargets

The easiest way to create animations is to use so-called property setters which are created internally when you provide a component, a property name and corresponding parameters to a *ULCAnimator*'s constructor or *addTarget()* method. However, you can also create custom targets that could perform various actions when the animation runs. To do this, you may extend both the *ULCAbstractAnimationTimingTarget* and the *UIAbstractAnimationTimingTarget*, and create a basic object that will either implement the *TimingTarget* interface or extend the equivalent abstract class *TimingTargetAdapter* from the TimingFramework. These provide methods for listening to animation events such as *begin()*, *end()*, *repeat()*, and *timingEvent()*.

4.17 Creating extended Visual Effects

ULC enables some exciting visual effects for its components such as translucency, gradients, textures and rounded corners. They all rely on the [JXLayer](#) library, a “universal decorator” for Swing components. Concretely, it enables to perform additional painting before and/or after the component paints itself.

In order to support these effects, the new properties translucency, paints and corner radius along with their getters and setters have been added to *ULCComponent* and are therefore available for all of its subclasses as well.

Please note that you cannot use any extended visual effects on *ULCComponents* whose basic object is not a *JComponent* (e.g. *ULCFrame*).

When applying extended visual effects on a component added to a scroll pane’s viewport, please also consider applying the visual effect to the scroll pane instead. It might, for example, be more desirable to have not only the table body semi-transparent but also its header and the scrollbars.

For more information, please have a look at the *ULCComponent* Javadoc.

4.17.1 How make a component translucent

Any *ULCComponent* can be made translucent using the *setTranslucency()* method. Valid values for the *alpha* parameter are within the closed interval [0, 1] with 0 meaning completely translucent and 1 meaning completely opaque. All children (contained components) of this component will also be affected by this effect.

Applying translucency to a component can be done with a single method call:

```
myUlcComponent.setTranslucency(0.5f);
```

4.17.2 How to paint gradient and textured backgrounds

Gradients and Textures can be created using the *setPaints()* method, which can take one or several arguments of two types: *GradientPaint* and *ImagePaint*. Both of these types implement the *ComponentPaint* interface that is expected by the *setPaints()* method. This means that you can compose multiple and various paints to achieve advanced background effects. Note that the order in which the backgrounds are painted is as follows :

- First, the standard background color is painted if required
- Then the extended backgrounds are painted in the exact order of the arguments of the *setPaints()* method. The last *ComponentPaint* argument is therefore always the topmost one.

The following example composes two semi-transparent gradients, one is vertical and the other is horizontal, along with a repeated semi-transparent image pattern. Moreover, it features a thick semi-transparent border.

```
ULCIcon image = new ULCIcon(getClass().getResource("/pattern.png"));
ULCBoxPane box = new ULCBoxPane();
box.add(ULCBoxPane.BOX_EXPAND_EXPAND, label);
box.setBackground(Color.yellow);
box.setPaints(
    new GradientPaint(new Color(0, 255, 255, 0), Color.cyan, Orientation.HORIZONTAL),
    new GradientPaint(new Color(255, 0, 0, 0), Color.red, Orientation.VERTICAL),
    new ImagePaint(image, ImagePaintRepetition.BOTH)
);
box.setBorder(new ULCLineBorder(new Color(255, 0, 255, 127), 20));
```



4.17.3 How to create gradient paints

As seen in the previous example, applying gradients can be done via the `setPaints()` method and the `GradientPaint` class. There are several constructors for this class that enable creation of two-color linear gradients in any direction. The following snippet creates four gradient paints:

```
// Paint gradients on ULCBoxPanels
box1.setPaints(new GradientPaint(Color.white, Color.black, Orientation.HORIZONTAL));
box2.setPaints(new GradientPaint(0, 0, Color.red, 100, 100, Color.gray));
box3.setPaints(new GradientPaint(0, 0, Color.red, 100, 100, Color.gray, true));
box4.setPaints(new GradientPaint(0, 0, Color.red, .2f, 0, Color.gray,
                                GradientPaintScale.PERCENTAGE, true));
```

The first gradient is horizontal and fades from white to black over the total width of the component. For vertical gradients, you may use the enum `Orientation.VERTICAL`. The second gradient is oblique along a line from point A(0px, 0px) to point B(100px, 100px). The same holds for gradient no. 3 which in addition is “cyclic”, meaning it fades alternately from its first color to its second color then back to its first color, etc.

Note that gradients no. 2 and no. 3 use “absolute” positioning in pixels. If you want to use “relative” positioning, use the enum `GradientPaintScale.PERCENTAGE` as in the case of gradient no. 4. This makes it fade from red to gray between 0 and 20% of its total width and also makes it cyclic.

4.17.4 How to create image paints

Just like `GradientPaints`, `ImagePaints` can be displayed in multiple ways depending on their alignment, their repetition across space, their scale, and their offset position:

```
box1.setPaints(new ImagePaint(fImage));
```

This component has the background image painted from the top-left corner, with no scaling, no repetition, and no offset.

```
box2.setPaints(new ImagePaint(image, ImagePaintAlignment.TOP_CENTER));
box3.setPaints(new ImagePaint(image, ImagePaintAlignment.TOP_CENTER,
    new Point(20, 20))
);
```

The second component has the background image painted from the top-center position, as specified by the enum `ImagePaintAlignment`. The third component the background image is with an offset of 20 pixels along both directions as compared to the second component.

```
box4.setPaints(new ImagePaint(image, ImagePaintRepetition.BOTH));
box5.setPaints(new ImagePaint(image, ImagePaintRepetition.X, new Point(0, 40)));
```

In this fourth example, the background image is positioned in the top-left corner, and is tiled like a mosaic on both directions as specified by the enum `ImagePaintRepetition`. In the fifth example, the background image is repeated only along the X-axis and additionally it is moved down by 40 pixels.

```

box6.setPaints(new ImagePaint(fImage,
    ImagePaintScaleType.BY_FACTOR, 2f,
    ImagePaintScaleType.IN_PROPORTION, 1f)
);

```

In the last example, the component has its background image scaled such that it will be painted twice its original width and that it will fill the exact height of the component as specified by the enum *ImagePaintOneDimScale*. This enum supports absolute target size, relative target size ‘in proportion’ and a ‘by factor’ scaling of the source image size. Scaling the image with locked aspect ratio is possible by using the alternative enum *ImagePaintTwoDimScale*.

To sum up, there are multiple constructors that allow you to paint image textures with simple behaviors as well as more complex ones.

You may set the alignment using the *ImagePaintAlignment* enum, the repeating behavior using the *ImagePaintRepetition* enum, the independent scaling of the image along X- and Y-axis using the *ImagePaintOneDimScale* enum and finally the scaling of the image with locked aspect ratio using the *ImagePaintTwoDimScale*.

4.17.5 How to display a component with rounded corners

ULCComponents can be displayed with rounded corners by using the *setCornerRadius()* method which takes a positive integer value for the desired radius. The component’s background, its border, and its children will then be “cropped” with nice anti-aliased rounded corners.



If child components being cropped in the corners does not have the desired look, you might provide your container with an additional *ULCEmptyBorder* of the desired insets. The optimal inset value is then equal to the round value of $\text{cornerRadius} * (1 - \sqrt{2} / 2)$.

You can have a *ULCLineBorder* with rounded corners by specifying the corner radius in its constructor. In addition, a rounded cornered *ULCLineBorder* can be displayed with anti-aliasing.

The following snippet shows how to add rounded corners using optimal insets and a rounded line border.

```

ULCIcon image = new ULCIcon(getClass().getResource("/pattern.png"));
ULCBoxPane box = new ULCBoxPane();
box.add(ULCBoxPane.BOX_EXPAND_EXPAND, new ULCLabel("ULC Rounded Corners"));

box.setBackground(Color.yellow);
box.setPaints(
    new GradientPaint(new Color(0, 255, 255, 0), Color.cyan, Orientation.HORIZONTAL),
    new GradientPaint(new Color(255, 0, 0, 0), Color.red, Orientation.VERTICAL),
    new ImagePaint(image, ImagePaintRepetition.BOTH)
);

int radius = 20;
box.setCornerRadius(radius);

int inset = (int) Math.round(radius * (1 - Math.sqrt(2) / 2));

```

```
box.setBorder(new ULCompoundBorder(  
    new ULCLineBorder(new Color(255,0,255,127), 20, radius),  
    new ULEmptyBorder(inset, inset, inset, inset)  
));
```

4.18 Best Practices

This section describes best practices that might be helpful to ULC application developers. In addition, it points out some issues to avoid when developing with ULC.

4.18.1 Use callback functions

Notification is always done through callback functions, which are realized as listener interfaces. It is not possible to stop somewhere in your server-side code and wait until the user triggers something and to only then continue the server-side code on the next line. Your code got triggered as a reaction to a user request and we have to return a response to finish the round-trip. This is different from a pure Swing application.

4.18.2 Avoid using static variables

Avoid the use of static variables because these are shared by all application sessions. The last session changing the variable wins. In most cases, this is not desired. This is especially true for ULC proxies such as `ULCIcon`, a common mistake of developers new to ULC and server-side programming, see also the following best practice.

As with server-side programming in general, static variables should only be used in case of constants, and even there constant references to mutable objects should be avoided as well.

Use the `ApplicationContext.setAttribute()` / `getAttribute()` API to statically access the session global state.

4.18.3 Avoid sharing proxies between sessions

ULC uses a unique identifier (*object id*) to address each proxy on the client and on the server side, respectively. Since this identifier is only unique within the current session, sharing proxies across different sessions is not allowed. As a consequence, no references to ULC proxies must be kept in static variables (s. best practice above). Note, that all framework classes with the prefix “*ULC*” are subclasses of *ULCProxy*.

4.18.4 Components cannot have multiple parents

Analogous to Swing, the same instance of a *ULCComponent* cannot have more than one parent, hence must not be added to different panes (containers) at the same time.

4.18.5 Share objects whenever possible

Share objects to keep the server session size low, e.g. models, icons, borders.

A large server session size will cause problems in a clustered environment when the session has to be exchanged between nodes of the cluster.

But keep in mind that icons, being ULC proxies, must not be shared between sessions (s. above).

4.18.6 Use uniform renderer components

Currently, you cannot modify the text in the renderer of any widget. Therefore, make sure your widget's model returns the text to be displayed. As an alternative, provide a decorator model that converts your data to its UI representation.

ULC frequently invokes the callback methods of renderers. Therefore you should be very careful to implement efficient callbacks, e.g. it is better to configure the renderer components with existing objects rather than with new objects.

Try to use uniform renderer components, e.g. avoid color gradients in tables. Non-uniform renderer components result in a lot of traffic as ULC cannot optimize communication in such a case.

4.18.7 Avoid asynchronous event handling

Do not use asynchronous event handling, except when you are aware of its consequences.

4.18.8 Thread-Safety

Do not access ULC components except from within the ULC thread. Otherwise, the behavior of your application is not specified. It is not possible to perform operations on ULC objects from another thread and there is no equivalent of the Swing method *SwingUtilities.invokeLater()* in ULC. All the code must be executed as part of processing a client request in the widget's event handlers. To simulate *invokeLater*-like behavior, use the *ULCPollingTimer* class. Using this class, the UI Engine can poll the ULC application at a specified polling interval. The non-ULC code can then be given access to the ULC objects in the *IActionListener* for the polling event. However, the appropriate *fire* methods should be called in the *actionPerformed()* method of the registered listener to initiate a server roundtrip.

4.18.9 Serializable session

Keep your session serializable in order to allow proper integration into Servlet and EJB application servers. This means that all objects having your *IApplication* instance as an ancestor should be serializable and any objects put in the *ApplicationContext* should be serializable as well. See the *Serializable Check* contribution on the [ULC Code Community website](#).

4.18.10 Session termination

Always terminate a session by calling *ApplicationContext.terminate()*. Never use *System.exit()*.

4.18.11 Reuse components

Keep the server and client memory footprint low by reusing ULC components like *ULCFrame*, *ULCWindow*. Hold these components in a pool and reconfigure them before display. Keep your pool in the *ApplicationContext*.

4.18.12 Simulate Low Bandwidth

Discover client-server traffic and communication bottlenecks already during development time. This can be achieved by launching ULC's *DevelopmentRunner* using the *-useGui* program parameter and changing the communication speed to a lower rate.

4.18.13 Use the GUI Provided with DevelopmentRunner

Watch the number of server roundtrips for a user operation. You can reduce this by bunching requests if your GUI allows you to compromise on responsiveness. For instance, by sending all the data from all the fields in a form.

Watch the sizes of the requests. Use this data to optimize the data transfer between ULC application and the UI Engine.

Simulate low bandwidth to discover client-server traffic and communication bottlenecks already during development time.

4.18.14 Check reachability through firewalls and proxies

Make sure your application can be accessed from remote clients. If you fail to access your application, this might be due to firewall or proxy settings on the client side or server side. Use the *Ping* tool available on the [ULC Code Community website](#) to narrow down the problem.

4.18.15 How to code long tasks

ULC follows the request-response paradigm. All requests originate from the client and are processed on the server mainly as part of event listeners. You must not perform long tasks in event listeners because that will lead to a long roundtrip and blocked and unresponsive GUI till the roundtrip is completed. To get around this problem you can make use of *ULCPollingTimer*.

Either you can accomplish a long task step-by-step in the *ULCPollingTimer*'s *ActionListener* or you can start a long task in a separate thread and have its status checked using a *ULCPollingTimer*. In the latter case, where you are using a separate thread from the ULC application, you must take care that you do not access ULC components from that thread (see best practice about Thread-Safety).

The following snippet shows how a long task is performed in a separate thread and monitored by *ULCPollingTimer*:

```
public class ULCSHowProgressSnippet extends AbstractApplication {
    private ULCFRAME frame;
    private boolean fIsTaskOver;

    public void start() {

        com.ulcjava.base.application.ApplicationContext.setAttribute("key", "value");
        frame = new ULCFRAME("ULCSHowProgressSnippet");
        frame.setSize(100, 100);
        frame.setLocation(100, 100);

        final ULCTextArea textArea = new ULCTextArea();
        textArea.setLineWrap(true);

        frame.getContentPane().add(textArea, ULCSBorderLayoutPane.CENTER);

        final ULCPollingTimer timer = new ULCPollingTimer(1000, null);
        timer.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                if (!fIsTaskOver) {
                    textArea.append(". ");
                } else {
                    timer.stop();
                    textArea.append("!");
                }
            }
        });

        final ULCSButton button = new ULCSButton("DoWork");
```

```

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                button.setEnabled(false);
                firstTaskOver = false;
                textArea.setText("");
                timer.start();
                doWork();
            }
        });

        frame.getContentPane().add(button, ULCSouthBorderLayoutPane.SOUTH);
        frame.setDefaultCloseOperation(ULCFrame.TERMINATE_ON_CLOSE);
        frame.setVisible(true);
    }

    private void doWork() {
        Thread worker = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 30; i++) {
                    try {
                        Thread.sleep(1000);

                        System.out.println(ULCSession.currentSession().getAttribute("key"));
                    } catch (InterruptedException e) {
                    }
                }
                firstTaskOver = true;
            }
        });
        worker.start();
    }

    public static void main(String[] args) {
        DevelopmentRunner.setApplicationClass(ULCShowProgressSnippet.class);
        DevelopmentRunner.run();
    }
}

```

See also the *Progress Pane* contribution on the [ULC Code Community website](http://ulc.codecommunity.com).

4.18.16 How to implement server push functionality

ULC is based on the JEE Request-Response paradigm. All requests originate from the client and the server responds to them. This means there is no direct support for server-side push.

However, push functionality can be simulated with client-side polling by using the *ULCPollingTimer* class. *ULCPollingTimer* creates a Swing timer on the client side, which will generate requests at specified time intervals. Then in the registered *IActionListener* code, check the global state and update each client session. Each time an action event is fired on the client side, a server roundtrip will be initiated. Hence, the *ULCPollingTimer* class can be used to poll the server for data updates. See the Javadoc of *ULCProgressBar* for sample implementation.

ULC polling scales much better than refreshing an HTML page, since ULC only sends changed data from the server to the client. In addition, the user interface remains stable, whereas with HTML the user interface is completely rebuilt. Having said this, for really high frequency updates the ULC client can even be extended (after all it is pure Java) to implement real push functionality.

4.18.17 Handling database connection pooling

ULC is basically a GUI library, which takes care of client/server partitioning. Developers define the application GUI using ULC widgets and implement the application logic in the event handlers of the widgets. The connection from the application to services, databases, etc. depends on the developer's preferences and to

some extent on the container in which the ULC application is deployed. The code related to connections to databases and services can either be implemented in the container initialization-related code or in the application logic.

As far as database connection pooling is concerned, one solution would be to model the pool as a singleton. The connections will be established when the first client session gets the singleton. On the other hand, if you would like to do the initialization before starting the first client session then you need to look into container facilities. ULC requires the Servlet 2.4 specification.

4.18.18 How to call native code on the client

The UI Engine is a standard Java application and can be adapted and extended to specific needs. Therefore native calls via JNI are possible as well. The native code DLL or library, if not available on the client, can be distributed using JNLP.

ULC provides hooks to exchange custom messages between client and server. For example, using the *ClientContext.sendMessage()* API, a ULC application may send messages to the client which may delegate it to the native library. To react to such messages a client must have an *IMessageService* installed. This can be done by calling *ClientEnvironmentAdapter.setMessageService(IMessageService messageService)*. This custom messaging mechanism could be used for notification of the completion of an asynchronous call.

4.18.19 Memory Management

ULC's memory management is server centric because ULC maintains the state of the GUI on the server; client is just a renderer of the server-side GUI state.

In ULC all memory management is done from the server side. When the server side garbage collector (gc) runs, it removes collectable objects from the server side object registry and sends a message to the client to remove those objects from the client side registry in the next roundtrip after the gc run. Since server side memory is larger than client side memory, often times, server gc runs too late before client side memory runs out. The client will run out of memory at some time unless gc is forced on the server and there are unreferenced widgets that can be collected on the server side.

A server side proxy (ULCXXX) becomes eligible for collection when there are no references to it, e.g., it does not belong to any container (say ULCFrame) or the container itself is eligible for collection (say Frame is disposed).

There can be two reasons for client side out-of-memory exception:

1. there is a memory leak in your application
2. the garbage collector does not run

To prevent client side out-of-memory error, first you should make sure (with help of a profiler) that you don't have a memory leak in your application. If you don't have a memory leak in your application then you can do one of the following:

- Increase the max heap size on the client

This can be specified in the **ULCApplicationConfiguration.xml** file:

```
<ulc:java>
  <ulc:initialHeapSize>128</ulc:initialHeapSize>
  <ulc:maxHeapSize>512</ulc:maxHeapSize>
</ulc:java>
```

- Do not create too many new proxies in your application

Reuse proxies, e.g. *ULCButtons*, *ULCLabels*, *ULCBorders*, *ULCIcons*, and whole windows.

- Regularly force a server-side garbage collector run from your application: `System.gc()`, though this may not be an acceptable solution.
- A component itself does not use that much memory on the client side. Therefore when the client runs out of memory then usually this is not because of too many component proxies. The issue is that some components hold large caches on the client side: *ULCTree*, *ULCTable*, *ULCList*, *ULCTableTree*.

So a first workaround should always be to invalidate all client side caches for non used components. The following code snippet does this for table trees:

```
private void disposeClientCaches(ULCComponent component) {
    if (component instanceof ULCTableTree) {
        ULCTableTree tableTree = (ULCTableTree) component;

        ITableTreeModel model = tableTree.getModel();
        if (model instanceof AbstractTableTreeModel) {
            AbstractTableTreeModel abstractModel =
                (AbstractTableTreeModel) model;

            abstractModel.structureChanged();
        }
    }

    if (component instanceof ULCContainer) {
        ULCContainer container = (ULCContainer) component;

        for (ULCComponent child : container.getComponents()) {
            disposeClientCaches(child);
        }
    }
}
```

4.19 How to extend ULC

In section 2.2 we saw how the generator generated two projects: **Person** and **Person-client**. The **Person** project has a class **com.canoo.person.extension.ULCBlinkingButton** which extends **ULCButton**. The **Person-client** project has the client side code for **ULCBlinkingButton**, namely **com.canoo.person.extension.UIBlinkingButton**. Please have a look at these two classes to understand how a ULC widget can be extended to add client-side code to it.

You can find detailed instructions on how to extend ULC in the <Canoo RIA Suite Install Dir>/doc/ULCExtensionGuide or on website [ULC Extension Guide](#). That document describes basic concepts about extending ULC widgets (both visual and non-visual). It also describes how to implement custom events, custom coders, custom model adapters, and custom cell renderers and editors.

Please also look at the [ULC Code Community](#) website listing many useful ULC extensions.

4.19.1 How to display a customized dialog on application failure

Create a class that implements *ISessionErrorHandler* in the client project.

```
package com.ulc.myproject;

import javax.swing.JOptionPane;

import com.ulcjava.base.client.UISession;
```

```
import com.ulcjava.base.client.launcher.ISessionErrorHandler;

public class MyErrorDialog implements ISessionErrorHandler {

    public void handleSessionError(UISession session, Throwable reason) {
        JOptionPane.showConfirmDialog(null,
            "The session has terminated because of " + reason.getMessage(),
            "Application Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Insert a `<ulc:clientSessionErrorHandlerClassName>` element into the **ULCApplicationConfiguration.xml**:

```
<ulc:clientSessionErrorHandlerClassName>
    com.ulc.myproject.MyErrorDialog
</ulc:clientSessionErrorHandlerClassName>
```

Now whenever there is an uncaught exception in the ULC Application, the ULC client will display the custom dialog before terminating the application.

4.19.2 How to customize user feedback during server roundtrip

ULC client blocks the UI for an event delivery or a server method invocation that happens to be synchronous. During such a server roundtrip, the ULC client displays a wait cursor indicating that the server roundtrip is in progress and the UI is blocked. Sometimes there is a requirement to customize the blocked input feedback, i.e., to indicate blocked UI state by other means than the wait cursor. ULC enables customizing of the user feedback during a synchronous server roundtrip by way of *com.ulcjava.base.client.IUserFeedbackStrategy*.

The developer can implement *com.ulcjava.base.IUserFeedbackStrategy* and set it on *com.ulcjava.base.client.InputBlocker* class in the ULC client launcher. By default, ULC uses the *DefaultUserFeedbackStrategy* which displays the wait cursor.

When providing your own implementation of *IUserFeedbackStrategy*, you must make sure that the class is available on the class path of the ULC client.

The following code demonstrates how to implement and set an *IUserFeedbackStrategy*:

```
public class CustomUserFeedbackStrategyLauncher {
    public static void main(String[] args) throws UnsupportedOperationException {
        PlasticLookAndFeel.setMyCurrentTheme(new TeamMembersTheme());
        UIManager.setLookAndFeel(new PlasticLookAndFeel());

        InputBlocker.setUserFeedbackStrategy(new CustomUserFeedbackStrategy());

        DevelopmentRunner.setApplicationClass(TeamMembers.class);
        DevelopmentRunner.setConnectionType(new ConnectionType("VERY SLOW",
            "Very slow connection type", 0, 2000));
        DevelopmentRunner.main(args);
    }

    private static class CustomUserFeedbackStrategy implements IUserFeedbackStrategy {
        public void stopBlockingFeedback(UISession session) {
            UIRootPane[] rootPanes =
                (UIRootPane[])session.getRegistry().findAssignables(UIRootPane.class);
            for (int i = 0; i < rootPanes.length; i++) {
                rootPanes[i].getBasicRootPane().getGlassPane().setVisible(false);
            }
        }

        public void giveUserInputBlockedFeedback(UISession session, AWTEvent event) {
            UIRootPane[] rootPanes =
                (UIRootPane[])session.getRegistry().findAssignables(UIRootPane.class);
            for (int i = 0; i < rootPanes.length; i++) {
```

```

        UIRootPane rootPane = rootPanes[i];
        rootPane.getBasicRootPane().setGlassPane().setBackground(
            new Color(255, 0, 0, 50));
    }
}

public void startBlockingFeedback(UISession session) {
    UIRootPane[] rootPanes =
        (UIRootPane[])session.getRegistry().findAssignables(UIRootPane.class);

    for (int i = 0; i < rootPanes.length; i++) {
        JLabel label = new JLabel("Waiting...", JLabel.CENTER);
        label.setFont(label.getFont().deriveFont(Font.BOLD, 48));
        label.setForeground(new Color(100, 100, 100, 150));

        JPanel glassPane = new JPanel(new BorderLayout());
        glassPane.setOpaque(true);
        glassPane.setBackground(new Color(100, 100, 100, 50));
        glassPane.add(label);

        UIRootPane rootPane = rootPanes[i];
        rootPane.getBasicRootPane().setGlassPane(glassPane);
        rootPane.getBasicRootPane().getGlassPane().setVisible(true);
    }
}
}
}

```

4.20 How to customize the failure behavior of servlet connector

By default each connector command tries up to 3 times to perform its job. When it fails, then it waits 1s before trying another time. When the maximum number of retries is reached, then it asks the user via a confirm dialog whether he wants to start over the whole process or not.

Two scenarios of customization are imaginable:

- You want to specify for every connector command a certain behavior to follow.
- Only for a specific command you want to apply a certain strategy.

4.20.1 Customize all connector commands

Create your own `IConnectorCommandFailureStrategyFactory` class in your client extension jar. For example:

```

package test.ulc.launcher;

import javax.swing.JOptionPane;

import com.ulcjava.container.servlet.client.ConnectorCommand;
import com.ulcjava.container.servlet.client.DefaultConnectorCommandFailureStrategy;
import com.ulcjava.container.servlet.client.IConnectorCommandFailureStrategy;
import com.ulcjava.container.servlet.client.IConnectorCommandFailureStrategyProvider;

public class FailureStrategyProvider implements
    IConnectorCommandFailureStrategyProvider {
    public IConnectorCommandFailureStrategy createConnectorCommandFailureStrategy
        (ConnectorCommand connectorCommand) {
        DefaultConnectorCommandFailureStrategy strategy
            = new DefaultConnectorCommandFailureStrategy() {
            protected int showConfirmDialog(ConnectorCommand connectorCommand) {
                return JOptionPane.showConfirmDialog
                    (null, "try again ?", "Failure !!!", JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE);
            }
        };
        strategy.setConfirmRetrying(true);
        strategy.setMaxRetryCount(5);
        strategy.setSleepTime(100);
        return strategy;
    }
}

```

```
    }  
}
```

Specify your class in the application configuration:

```
<ulc:connectorCommandFailureStrategyProviderClassName>  
    test.ulc.launcher.FailureStrategyProvider  
</ulc:connectorCommandFailureStrategyProviderClassName>
```

4.20.2 Customize a specific connector command

Override the factory method of corresponding connector command in your *ServletConnector* extension and customize the *IConnectorCommandFailureStrategy* implementation you get by invoking *getConnectorCommandFailureStrategy* method:

```
(ServletConnector)  
...  
protected AbstractSendRequestsCommand createSendRequestsCommand(Request[] requests) {  
    AbstractSendRequestsCommand sendRequestCommand = new SendRequestsCommand(fUrl,  
        fRequestPropertyStore, getSessionId(), requests, fDataStreamProvider,  
        fCarrierStreamProvider, fCoderRegistry, fConnectorCommandFailureStrategyFactory);  
    ((DefaultConnectorCommandFailureStrategy) sendRequestCommand.getConnectorCommandFail  
        ureStrategy()).setMaxRetryCount(10);  
    return sendRequestCommand;  
}
```

5 Add-on packages

The following sections describe features from add-on packages. These features are not part of ULC Core; they have to be purchased separately.

5.1 Using ULC Web Integration

The ULC Web Integration package contains *ULCBrowser* which provides an HTML rendering engine as a ULC component.

ULCBrowser is based on the [JxBrowser](#) component. It provides the following features:

- Rendering engines: Mozilla (Gecko), Safari (Webkit), IE
- Navigation to a URL (optionally with post data)
- Access to browser state (title, content, history, rendering engine type)
- Access to the browser history and 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 [JxBrowser](#) component is run asynchronously. Therefore, it is necessary to use a navigation listener to be sure that the entire content of the browser has been rendered and the browser is again in a consistent state.

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

Not all methods are supported by [JxBrowser](#) across all native rendering engines. The resulting exception is ignored on the client side and *ULCBrowser* returns *null*, e.g. *ULCBrowser.getAllHistory* is currently only supported with the Mozilla browser.

Using a *ULCBrowser* component in an applet deployment scenario is not recommended (works only for specific host browser/embedded browser combinations and is not stable at all).

5.1.1 Development setup for ULCBrowser

The ULC project generator will do the required set up if you selected the ULC Web Integration package as one of the Available Extensions (see section 2.2).

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/ULCWebIntegration:** **client**, **client/third_party**, **server**, and **server/third_party**.

5.1.2 How to create a simple browser

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

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

Have a look at *BrowserType* for the rendering engines currently supported.

5.1.3 How to deploy a browser based application

The ULCBrowser comes with following client libraries:

- *jxbrowser-client.jar*
- *log4j-client.jar*
- *slf4j-api-client.jar*
- *slf4j-log4j-client.jar*
- *tuxpack-client.jar*
- *winpack-client.jar*
- *xulrunner-linux-client.jar*
- *xulrunner-mac-client.jar*
- *xulrunner-windows-client.jar*
- *engine-gecko-client.jar*
- *engine-ie-client.jar*
- *engine-webkit-client.jar*
- *jniwrap-native-client.jar*
- *MozillaGlue-client.jar*
- *MozillaInterfaces-client.jar*
- *mshtml-client.jar*

Because the libraries needed on client side heavily depend on the operating system running on the client, it makes sense to reduce the resources which must be downloaded by the client as following (**JNLP** example, client libraries located in public directory called *lib*):

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.5+" codebase="$$codebase" href="$$name">
...
<resources>
  <jar href="lib/log4j-client.jar" />
  <jar href="lib/jxbrowser-client.jar" />
  <jar href="lib/jniwrap-native-client.jar" />
  <jar href="lib/engine-gecko-client.jar" />
  <jar href="lib/MozillaGlue-client.jar" />
  <jar href="lib/MozillaInterfaces-client.jar" />
  <jar href="lib/slf4j-api-client.jar" />
  <jar href="lib/slf4j-log4j-client.jar" />
</resources>
<resources os="Windows">
  <jar href="lib/engine-ie-client.jar" />
  <jar href="lib/mshtml-client.jar" />
  <jar href="lib/winpack-client.jar" />
  <jar href="lib/xulrunner-windows-client.jar" />
</resources>
<resources os="Mac OS X">
  <jar href="lib/engine-webkit-client.jar" />
  <jar href="lib/xulrunner-mac-client.jar" />
</resources>
<resources os="Linux">
  <jar href="lib/tuxpack-client.jar" />
  <jar href="lib/xulrunner-linux-client.jar" />
</resources>
...
</jnlp>
```

Additionally, you should increase the maximum heap size of the client. Mac OS X clients should moreover specify following Java VM arguments:

```
-d32 -Dcom.apple.eawt.CocoaComponent.CompatibilityMode=false
```

Finally, you will end with following **JNLP** tags:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.5+" codebase="$$codebase" href="$$name">
...
<resources os="Mac OS X">
  <j2se version="1.5+" href="http://java.sun.com/products/autodl/j2se" initial-heap-
size="64m" max-heap-size="256m" java-vm-args="-d32 -
Dcom.apple.eawt.CocoaComponent.CompatibilityMode=false" />
</resources>
<resources os="Windows">
  <j2se version="1.5+" initial-heap-size="64m" max-heap-size="256m"/>
</resources>
<resources os="Linux">
  <j2se version="1.5+" initial-heap-size="64m" max-heap-size="256m"/>
</resources>
...
</jnlp>
```

5.2 ULC Table Plus for creating complex tables

The ULC Table Plus package enables sophisticated tables for business applications. The ULC Table Plus package extends the functionality of the *ULCTable* by some of the key features of the [JIDE Grids](#) package which is provided as Swing components by Jide Software company.

Together with the *ULCJideTable* component a new scroll pane component the *ULCTableScrollPane* is introduced. The *ULCTableScrollPane* the standard scroll pane with the possibility to display parts of *ITableModel* to different areas of the scroll pane.

The following features are provided by the *ULCJideTable*

- Nested columns headers
- Auto filter in Column header
- Column auto-resize and selection with popup menu
- Row height resizing and column resizing in table (not only in the header)
- Auto-completion in cells
- table disable
- multi-column sorting
- auto-start editing

The following features are provided by the *ULCTableScrollPane*

- Row header and footer
- Table footer
- Convenience methods for to configure the underlying *ULCJideTables*

5.2.1 Development setup for ULCTableScrollPane

The ULC project generator will do the required set up if you selected the ULCTablePlus package as one of the Available Extensions (see section 2.2).

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/ULCTablePlus: **client**, **client/third_party**, **server**, and **server/third_party**.

5.2.2 How to use the ULCTableScrollPane

The screenshot shows a window titled 'ULCTablePlusDemoApp' with a menu bar (New, Edit, Delete, Switch to Nested Header) and a toolbar. The main area contains a table with columns: No., Task, Billable, Project, 1 May '10, 2 May '10, 3 May '10, 4 May '10, 5 May '1, and Sum. The table has 21 rows of data. Red numbers 1 through 6 are overlaid on the table to indicate specific regions: 1 is on the 'Task' column header, 2 is on the main data area, 3 is on the 'Sum' column header, 4 is on the bottom row header, 5 is on the bottom data area, and 6 is on the bottom right corner.

ULCTableScrollPane with six table areas

The above picture shows a usage scenario for a *ULCTableScrollPane*. The different parts of the *ULCTableScrollPane* in the above picture are described below:

1. **RowHeaderTable**: a table that contains a static part at the beginning of the main table
2. **MainTable**: the main table which is scrollable if necessary
3. **RowFooterTable**: a table which contains a static part at the end of the main table
4. **ColumnFooterRowHeaderTable**: a table which contains a static part at the beginning of the footer table
5. **ColumnFooterTable**: the footer table at the bottom of the main table which is scrollable if necessary
6. **ColumnFooterRowFooterTable**: a table which contains a static part at the end of the footer table

The tables placed in the *ULCTableScrollPane* are extended *ULCTables* - the *ULCJideTables* - but these tables have to be created by the component itself. These underlying *ULCJideTables* are accessible via appropriate getters. The client state of the table is available through the provided API. Note that *ULCTableScrollPane* uses JIDE table internally on the client side and does not expose JIDE table's API on the server side. As a result of adapting the complex API of the JIDE Grids components, the *ULCJideTables*, in contrast to *ULCTables*, do not support lazy loading of row data at present.

The following example shows creation of a *ULTableScrollPane* that contains beside the main table a row header table, a row footer table and a column footer:

```
private ULComponent createTableScrollPane() {
    fHeaderColumns = new int[] {0, 1, 2, 3, 4, 5};
    fFooterColumns = new int[] {13};
    fMainTableModel = new MainTableModel();
    fFooterTableModel = new FooterTableModel(fMainTableModel);
    fScrollPane = new ULTableScrollPane(fMainTableModel, fFooterTableModel,
                                       fHeaderColumns, fFooterColumns, true);

    customizeTable();
    return fScrollPane;
}
```

The upper part (*RowHeaderTable*, *MainTable* and *RowFooterTable*) share one *ITableModel*. The definition of columns that belong to different tables is done by adding the column indices to the *headerColumns* or to the *footerColumns*. Similarly the lower part tables (*ColumnFooterRowHeaderTable*, *ColumnFooterTable* and *ColumnFooterRowFooterTable*) share an *ITableModel* among them.

5.2.3 How to create multi-line headers for a table

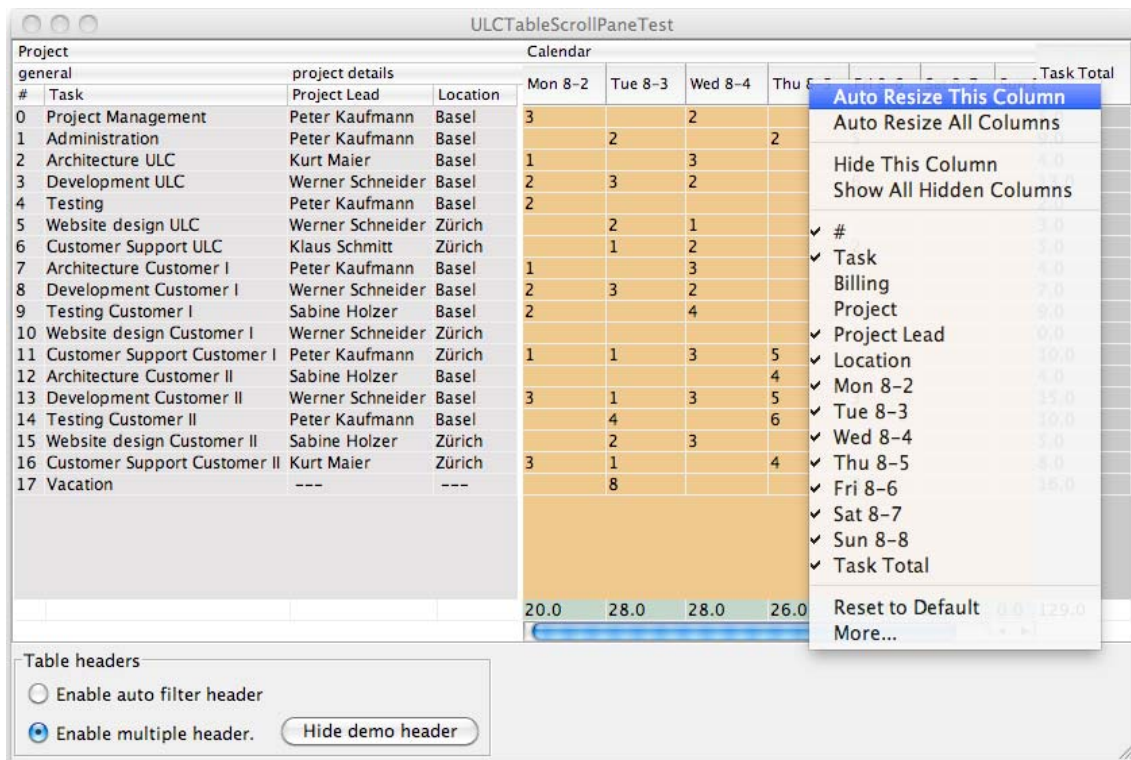
The *ULTableScrollPane* offers extended possibilities for sorting and filtering and it provides an API to set multi-line headers on tables.

A multi-line header consists of a hierarchical organized collection of *ULTableColumnGroups*. The following code demonstrates how to create a multi-line header:

```
private void addMultiLineHeader() {
    fRowHeaderColumnHeader = new ULTableColumnGroup("Project");
    ULTableColumnModel rowHeaderColumnModel =
        fScrollPane.getRowHeaderTable().getColumnModel();
    ULTableColumnGroup first = new ULTableColumnGroup("general");
    first.add(rowHeaderColumnModel.getColumn(0));
    first.add(rowHeaderColumnModel.getColumn(1));
    first.add(rowHeaderColumnModel.getColumn(2));
    ULTableColumnGroup second = new ULTableColumnGroup("project details");
    second.add(rowHeaderColumnModel.getColumn(3));
    second.add(rowHeaderColumnModel.getColumn(4));
    second.add(rowHeaderColumnModel.getColumn(5));
    fRowHeaderColumnHeader.add(first);
    fRowHeaderColumnHeader.add(second);

    fMainColumnHeader = new ULTableColumnGroup("Calendar");
    ULTableColumnModel columnModel =
        fScrollPane.getMainTable().getColumnModel();
    for (int i = fHeaderColumns.length; i <= fFooterColumns[0]; i++) {
        fMainColumnHeader.add(columnModel.getColumn(i));
    }

    fScrollPane.getMainTable().addColumnGroup(fMainColumnHeader);
    fScrollPane.getRowHeaderTable().addColumnGroup(fRowHeaderColumnHeader);
}
```



ULCTableScrollPane with multi line header and auto generated utility pop up menu

5.2.4 How to create AutoFilterTableHeaders for a table

The *ULCJideTable* provides the possibility to set a auto filter table header. The following example shows how to create such a header:

```
public void configureAutoFilterHeader() {
    ULCJideTable mainTable = fScrollPane.getMainTable();
    ULCJideTable rowHeaderTable = fScrollPane.getRowHeaderTable();
    ULCJideTable rowFooterTable = fScrollPane.getRowFooterTable();

    mainTable.setNestedTableHeader(false);
    rowHeaderTable.setNestedTableHeader(false);
    rowFooterTable.setNestedTableHeader(false);

    fScrollPane.setAutoFilterTableHeader(true);

    mainTable.setShowFilterName(true);
    mainTable.setShowFilterNameAsToolTip(true);
    mainTable.setShowFilterIcon(true);
}
```

No.	Task	Billable	Project	1 May '10	2 May '10	3 May '10	Sum
	(All)		Siemens Prototype				29
	(Custom...)		Siemens Prototype				28
	Administration Canoo		Canoo internal	6			28
	Architecture IBM 1		Telekom Complete				29
	Architecture Telekom 1		Telekom Complete				20
	Architecture UBS 1		Telekom Complete				23
	Architecture ULC		Telekom Complete	7			14
	Customer Support IBM 1		Ria Suite	6			27
	Customer Support UBS 1		Ria Suite				22
	Customer Support ULC		Ria Suite	7			23
	Development IBM 1		Ria Suite				27
	Development Siemens 1		IBM Workflow				17
	Development Telekom 1		IBM Workflow	6			26
	Development UBS 1		IBM Workflow	3			25
	Development ULC		IBM Workflow	4			25
			IBM Workflow	5			10
				45	0		487

ULTableScrollPane with configured auto filter table header

5.3 ULC Enterprise Portal-Integration

This package enables ULC applications to run inside an enterprise application portal. Many enterprises are using a JEE Application Server or a JSR-186 compliant portal server to bundle their business application into integrated employee workspaces. A ULC application can be integrated seamlessly in such a workspace as an applet.

This package allows ULC applications to be run in an enclosing Web application without losing its state on a page reload and between visits of the page. This allows the user to work with a ULC application, temporarily navigate away to another application and come back to the ULC application and resume his/her work therein.

The ULC Enterprise Portal-Integration binds the *ULCSession* to the *HttpSession* of the Web Container it is integrated to. The classes of the package are handling the session start and resume.

The libraries of the package are in the <Canoo RIA Suite Install Dir>/ext/ULCPortalIntegration folder.

On the client side the **ULCPortalIntegration-client.jar** contains the package *com.ulcjava.ext.portalintegration.client* with the **EnterpriseAppletLauncher** replacing the **DefaultAppletLauncher**. The **EnterpriseAppletLauncher** uses the **EnterpriseServletConnector** which creates a **CreateEnterpriseSessionCommand** in order to create a connection to the ULC application when the applet starts.

On the server side the **ULCPortalIntegration-server.jar** contains the package *com.ulcjava.ext.portalintegration.server* with the **EnterpriseServletContainerAdapter** and **EnterpriseServletContainerAdapterHelper** replacing the standard *ServletContainerAdapter* and *ServletContainerAdapterHelper*. They use the **CreateEnterpriseSessionCommand** to create an **EnterpriseULCSession** which allows reconnection on a page reload.

5.3.1 How to use the ULC Enterprise Portal-Integration

To enable the ULC Enterprise Portal-Integration for your application follow these steps:

-
1. Generate a new project as described in 3.4. Check the `ULCPortalIntegration` box from the available extensions.²

2. In the configuration file `<your-project>/src/ULCApplicationConfiguration.xml` add the following tag

```
<ulc:appletLauncherClassName>
com.ulcjava.ext.portalintegration.client.EnterpriseAppletLauncher
</ulc:appletLauncherClassName>
```

3. In the deployment descriptor `<your-project>/WebContent/WEB-INF/lib/web.xml` you must configure the **EnterpriseServletContainerAdapter** for the ULC application :

```
<servlet>
  <servlet-name>ServletContainerAdapter</servlet-name>
  <servlet-class>
    com.ulcjava.ext.portalintegration.server.EnterpriseServletContainerAdapter
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ServletContainerAdapter</servlet-name>
  <url-pattern>/ulc</url-pattern>
</servlet-mapping>
```

If your application does not inherit from `com.ulcjava.applicationframework.application.SingleFrameApplication`, you should overwrite its `resume` method and call `setVisible(true)` on the application's rootpane.

5.3.2 How to integrate multiple applets with distinct sessions

If you provide a ULC application in a portlet, you might want to place several independent instances of the portlet.

To allow different applets of the same ULC application within an HTTP session, you can set an arbitrary string as a client id to discriminate the different sessions.

- If you are using the `ulcApplet.tag` in your `jsp` you can define the `clientId` attribute:

```
<h:ulcApplet title="Title" clientId="Quote1"
style="left: 0px; top: 20px; width: 300px; height: 210px;" >
</h:ulcApplet>
```

- Otherwise you should add the client id as parameter with the name "client-id" to the `object/embed/applet` tag

```
<param name="client-id" value="Quote1">
```

² or integrate the **ULCPortalIntegration-client.jar** and **ULCPortalIntegration-server.jar** into your project if you don't use the generator.

5.3.3 How to use user parameters to pass data to the ULC applet

With the ULC Enterprise Portal-Integration the user parameters are passed to the application each time the session is resumed.

This can be used to pass parameters from the enclosing web application into the ULC application.

Evaluate the user parameter in the resume method of your application to react on its value.

5.4 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.

5.4.1 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 2.2).

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**.

5.4.2 How to use the ULC Office Integration

There 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>
```

5.4.3 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 proves to inadequate for your needs. Two abstract classes can be used as a basis for Word documents: *AbstractWordFileResource* and *AbstractWordMemoryResource*.

5.4.4 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:

```
ResultSet 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);
}
```

5.4.5 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 Debugging and Testing

6.1 How to enable logging

Logging provides valuable information during development and debugging. Logging settings are configured in the ULC configuration file. This configuration file is named *ULCApplicationConfiguration.xml* and is located in the root package of your application's class path.

```
<?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:applicationName>
      com.canoo.person.view.PersonApplication
    </ulc:applicationName>
    <ulc:clientLogLevel>
      ALL
    </ulc:clientLogLevel>
    <ulc:serverLogLevel>
      OFF
    </ulc:serverLogLevel>
  </ulc:ULCApplicationConfiguration>
```

The above example configures the full client-side logging and disables server-side logging. The client-side logging is printed to the client-side console, e.g. the Applet console, the server-side logging is printed to the server-side console, e.g. the application server log file.

6.1.1 Logging Classes

ULC provides a simple log mechanism containing the three classes *Level*, *Logger* and *LogManager*. The *Level* class defines a set of standard logging levels. A *Logger* object is used to log messages for a specific system or application component. *LogManager* is the abstract base class for log managers in ULC. It manages the logger objects for a ULC application. *SimpleLogManager* is the default log manager used for ULC applications. More information about these classes can be found in the ULC API documentation.

By default the *SimpleLogManager* is used to log information, both on the client side (UI Engine) and on the server side (ULC). ULC itself logs information at the following levels:

Level SEVERE: Log critical errors and events that stop the session.

Level WARNING: Log errors that do not stop the session.

Level INFO: Log events that occur once for a session.

Level FINE: Log events that occur once for a server round-trip.

Level FINER: Log events that occur once for a request.

Level FINEST: Log additional information such as contents of a request.

To be able to see the information logged by ULC, configure the logging level of the current log manager. See the example provided in Chapter 6.1.3. By default the level is set to WARNING.

6.1.2 Configuring the Client-Side (UI Engine) Log Manager

The log level for the UI Engine can be set by providing it as a parameter to the launcher.

For a *DefaultAppletLauncher*, the developer can set the *log-level* parameter for the UI Engine applet in the web page:

```
<jsp:plugin
    type="applet"
    code="com.ulcjava.environment.applet.client.DefaultAppletLauncher.class"
    archive="lib/ulc-applet-client.jar,lib/ulc-base-client.jar,
        lib/ulc-servlet-client.jar">
    <jsp:params>
        <jsp:param name="url-string" value="<%= applicationUrl %>" />
        <jsp:param name="keep-alive-interval" value="900" />
        <jsp:param name="log-level" value="WARNING" />
    </jsp:params>
</jsp:plugin>
```

For a *DefaultJNLPLauncher*, specify an argument in the JNLP file:

```
<application-desc
    main-class="com.ulcjava.environment.jnlp.client.DefaultJnlpLauncher">
    <argument>$$context/hello.ulc</argument>
    <argument>900</argument>
    <argument>WARNING</argument>
</application-desc>
```

Another way of enabling logging is through the *ULCApplicationConfiguration.xml* file:

```
<ulc:clientLogLevel>ALL</ulc:clientLogLevel>
```

6.1.3 Configuring the Server-Side (ULC) Log Manager

A developer can configure the log manager in the application's *start()* method. Select:

- the desired granularity of log information by setting the appropriate level.
- the IO stream where the logger output will be directed.

```
if (LogManager.getLogManager() instanceof SimpleLogManager) {
    SimpleLogManager manager =
        (SimpleLogManager) LogManager.getLogManager();
    manager.setLevel(Level.ALL);
    manager.setLogStream(System.out);
}
```

Another way of enabling logging is through the *ULCApplicationConfiguration.xml* file:

```
<ulc:serverLogLevel>ALL</ulc:serverLogLevel>
```

Alternatively, the log level can be provided as an argument to the container, in which a ULC application is running. For instance, if the ULC application is running as a Servlet then the log level can be specified as a parameter to the Servlet in the **web.xml** file as follows:

```
<servlet>
    <servlet-name>ApplicationApplet</servlet-name>
    <servlet-class>
        com.ulcjava.container.servlet.server.ServletContainerAdapter
    </servlet-class>
    <init-param>
        <param-name>application-class</param-name>
```

```
<param-value>com.ulcjava.sample.hello.HelloApplet</param-value>
</init-param>
<init-param>
  <param-name>log-level</param-name>
  <param-value>WARNING</param-value>
</init-param>
</servlet>
```

Changing the logging configuration in the above described ways affects all ULC sessions running in the same “environment”. E.g., if a ULC application runs as a Servlet *A*, then all ULC sessions associated with *A* have the same log level and the same log stream. However, if there exists a second ULC application, which runs as Servlet *B* in the same web container, then the logging configuration of the ULC sessions of *B* are not affected by changing the logging configuration for *A*. In short, *every Servlet hosting a ULC application offers a separate logging configuration scope* for the ULC sessions it controls. If a ULC application is deployed as an EJB, then *every instance of that EJB offers a separate logging configuration scope too*.

The different logging configuration scopes only affect the logging that happens *within a ULC session* (including all ULC proxies contained in the session). They do not affect the logging configuration on the environment level, in which the ULC sessions are hosted. E.g., the logging information provided by the Servlet instance that contains a ULC session is outside of these scopes. The same holds for an EJB instance, which contains a ULC session. In other words, logging messages that happen on the servlet execution level or the EJB execution level (outside a ULC session) are not affected by the logging configuration.

The logging configuration on the environment level has a separate, global scope, which can be set by means of a properties file. In the Servlet case, the global scope is the entire web application (usually represented by a **war** file). In the EJB case, the global scope is the entire EJB application (usually represented by an **ear** file).

The properties file to set the global scope is called **ulclog.properties** and may be deployed as part of a ULC web application. In the Servlet case, it must be accessible by the ULC Servlet and should be placed under the following path of a corresponding **war** file:

```
WEB-INF/classes/ulclog.properties
```

In the EJB case, the properties file should be placed in the **jar** file containing your ULC application. (The jar file itself is typically contained in an **ear** file.)

The properties file may have one (optional) property entry:

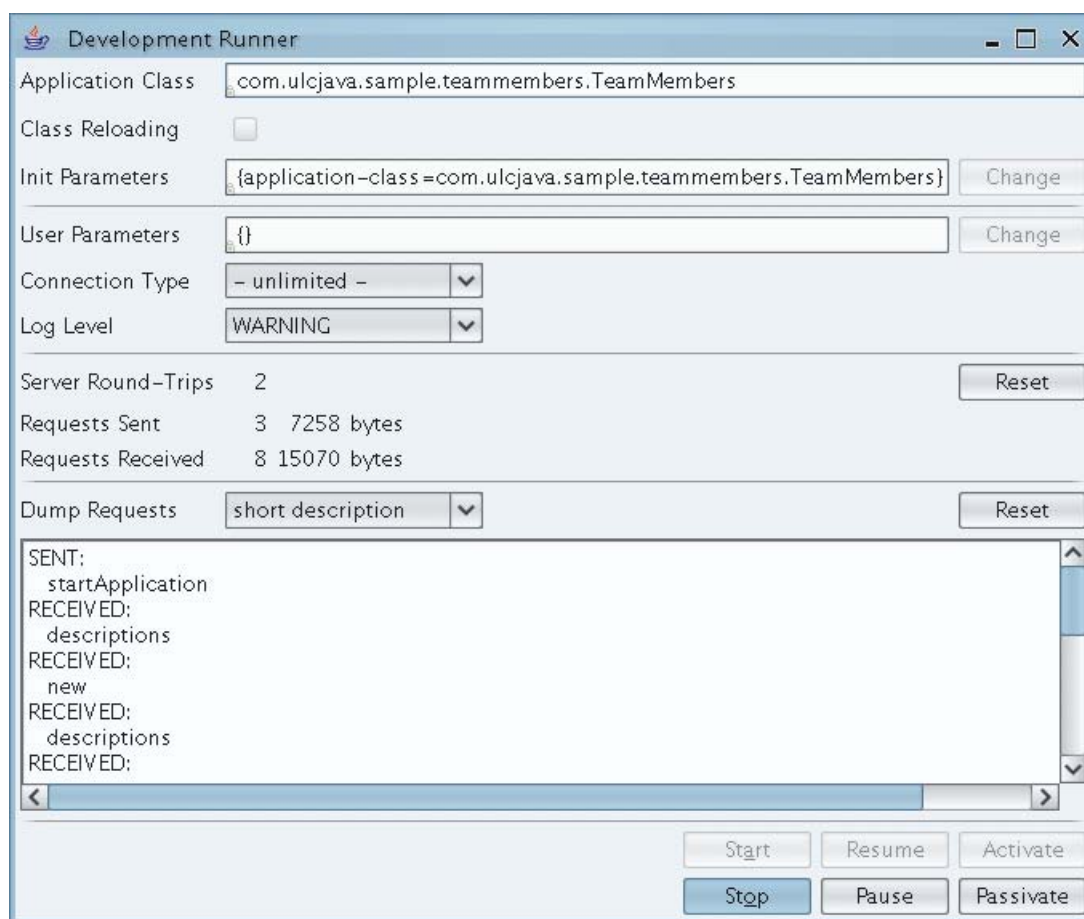
- **log-level** defines the log level for the resulting logging configuration

If the properties file or its property entry does not exist, then the default log level **WARNING** is used.

6.1.4 How to analyze the client-server communication

You can view the requests and responses exchanged between the ULC client and server for the purpose of debugging and gaining insight into your application. There are two ways to analyze the client-server communication.

One way, in development mode when running your application using the *DevelopmentRunner* (See Section 4.1.3), is to turn on the *DevelopmentRunner*’s user interface: *DevelopmentRunner.setUseGui(true)*.



The above screen shot was taken after starting the TeamMembers sample application that is part of the release and that can found in the sample/teammembers directory. It shows that starting the TeamMembers sample application required 2 server round-trips. In total the client-side sent 3 requests and 7'258 bytes, and the server-side responded with 8 requests and 15'070 bytes. In the text area at the bottom of the user interface you can see the request details.

The second way is to enable logging and interpret the logging output. ULC supports the following log levels:

- **SEVERE**
Critical errors that stop the application, e.g. NullPointerException in your application.
- **WARNING**
Errors that do not stop the application.
- **INFO**
Events that occur once for an application, e.g. application start.
- **FINE**
Events that occur once for a server round-trip, e.g. start of a server round-trip.
- **FINER**
Events that occur once for a request.
- **FINEST**
Additional information, e.g. HTTP header fields.

Please note that interpreting the logging output requires some experience with the ULC communication protocol. You can find more information on this in the [ULC Extension Guide](#) that comes with the release.

6.1.5 How to setup a debugger

With a debugger you can look into your application at runtime. As ULC is pure Java, it is easy to debug your application from within the IDE.

For example, in Eclipse, for the generated application from chapter 2 select the **Run > Debug History > Person** menu item to start the Person application inside the debugger.

6.1.6 How to setup a profiler

With a profiler you can find performance bottlenecks and memory issues in your application. As ULC is pure Java, you can use your regular Java profiler. The documentation of your Java profiler will tell you how to run a Java application inside the profiler.

6.2 How to write unit tests

ULC provides a JUnit and Jemmy based test framework for writing tests for ULC applications. You can find detailed instructions on how to write unit tests for your ULC application in the **ULCTestFrameworkGuide** that comes with the release. You can find this guide in the <Canoo RIA Suite Install Dir>/doc/addon folder or on the ULC web site [ULC TestFramework Guide](#).

6.3 How to perform load tests

To perform load tests for your ULC application you can use the ULC Load companion product. You can find more information about **ULC Load** on the ULC web site:

- <http://www.canoo.com/webexperts/products/ulc/#packages>
- <http://ulc.canoo.com/developerzone/ULCLoadUserGuide.pdf>

7 Deployment

This chapter describes various ways in which ULC client and ULC application can be deployed in production environments. For more details on deployment please see **<Canoo RIA Suite Install Dir>/doc/ULCDeploymentGuide**.

A ULC application has two deployment units: an application-independent presentation engine called the UI Engine on the client, and the ULC application running on the server. The UI Engine does not contain any application-specific code.

Client-Side Deployment:

The UI Engine is Java code packaged in various Java archives (jar files). It consists of two parts: the UI Engine launcher and the UI Engine core. Both parts are customizable and extensible. The launcher creates an execution environment and starts the UI Engine. The UI Engine can be deployed and started in either of the following modes:

- The launcher can be a standalone Java application creating a secure sandbox for the UI Engine. This mode requires explicit installation of the UI Engine on the client. The installation process might be supported by an installation program such as InstallAnywhere.
- The launcher can be an applet, running the UI Engine within a browser.
- Java Web Start is a lean way to both deploy the UI Engine and start ULC applications.

Server-Side Deployment:

A ULC application may be deployed as a servlet in a servlet container or as a stateful session bean in an EJB container.

The required protocol for the servlet container is HTTP(S). This is the standard deployment scenario for applications with users in the Internet since HTTP(S) is accepted by most firewalls.

An EJB container offers more sophisticated application management features, such as support for activation/passivation. The protocol with an EJB container might be RMI, IIOP or T3. For Intranet scenarios, this may be an alternative to servlet container deployment.

7.1 EasyDeployment

ULC comes with convenience Servlets that return default HTML pages and JNLP files that start your application. With EasyDeployment there is:

- No need to adapt deployment descriptors
- No need to write custom Applet tags or JNLP files
- No need to write HTML welcome pages

The EasyDeployment servlets provide for signing, pack200, and version based downloads.

Have a look at the **web.xml** file in the directory **<Canoo RIA Suite Install Dir>/addon/generators/templates/projecttemplates/server/WebContent/WEB-INF**.

This file specifies all the servlets used by the EasyDeployment feature:

- ConfigPropertiesDownloader servlet delivers the client specific configuration from the **ULCApplicationConfiguration.xml** file.

- ClientJarDownloader servlet supports versioning based downloads and pack200 of ULC client side jars. It also does auto packing and auto signing. The client jar files (placed in WEB-INF/lib and ending in -client.jar) are prepared by a servlet context listener you can find in the easy deployment module: com.ulcjava.easydeployment.server.ClientJarPreparationListener. The client jar files are signed if a keystore is specified in the jarsigner.properties file.
- ResourceDownloader servlet serves images from the application resource bundles.
- IndexServlet servlet dispatches to applet or jnlp environment depending on the tag **defaultClientEnvironment** in **ULCApplicationConfiguration.xml**. This tag can have two values:
 - **Applet** : redirect to the URL configured in the **applet-redirect** init-parameter of the IndexServlet
 - **JNLP** : redirect to the URL configured in the **jnlp-redirect** init-parameter of the IndexServlet
- JnlpDownloader servlet creates the .jnlp for ULC using the values specified in ULCApplicationConfiguration.xml and resources from the resource bundles. Some of the values used from the **ULCApplicationConfiguration.xml** file are:
 - **java** : version => java version to use, vmArgs => java virtual machine arguments, initialHeapSize => initial heap size of the java virtual machine, maxHeapSize => maximum heap size of the java virtual machine, forceAllPermissions => starts the client environment outside of the sandbox
 - **desktopIntegration** :
 - **shortcut** : desktop => create a desktop shortcut, menu => create a application menu shortcut,
 - **association** : extension => associate the application with this file extension, mimeType => associate the application with this mime type

7.2 How to use EasyDeployment without using generator

If you cannot use the application generator, you may still like to use some of the EasyDeployment features. Look at the files in the directory **<Canoo RIA Suite Install Dir>/addon/generators/templates/projecttemplates/server/WebContent/WEB-INF** that you would like to use and replace the tokens, @projectname@, @rootpackage@ and @rootpackagepath@ with values that are matching your ULC project's data in Eclipse. You need also to add the jars from **<Canoo RIA Suite Install Dir>/addon/easydeployment/lib** to the **WEB-INF/lib** directory of the project:

- standard.jar and jstl.jar are used for the jsp and the ulcApplet.tag
- tools.jar is used for the signing of the jars.

7.3 How to run your application inside a browser

ULC comes with a servlet that dispatches a request to the context root to a jsp page that renders the necessary HTML page. Just point your browser to e.g. <http://localhost:8080/TeamMember-server/> to run your application inside a browser.

The ULC project generator provides all the files necessary to run your application inside a browser:

- *appletPage.jsp*

In *addon/generators/templates/projecttemplates/server/WebContent* directory

Java Server Page that renders an HTML page that displays the ULC Applet. It uses the *ulcApplet* tag.

- *ulcApplet.tag*

In *addon/generators/templates/projecttemplates/server/WebContent/WEB-INF/tags* directory. Custom tag file that renders the html snippet that includes the applet. It has the following attributes:

title	Name of the applet.
style	Style for the OBJECT and EMBED tags. This is used to control the size of the applet.
scriptToCall	Name of a javascript function that is called after the applet has started.
clientId	Identifier to discriminate different applets for the same ULC application when using the ULC Enterprise Portal-Integration package.

- *web.xml*

In *addon/generators/templates/projecttemplates/server/WebContent/WEB-INF/tags* directory. It configures all necessary Servlets and Listeners to run your application inside or outside a browser.

- *ulc-easydeployment-server.jar*

In *addon/easydeployment/lib* directory. It is also integrated into the **ulc-core-server.jar**. This jar file contains the servlets and utility code for EasyDeployment feature:

- *jnlp.sample.servlet.JnlpDownloadServlet*³

Helper Servlet that provides your application's client-side Java classes. The servlet handles the different kinds of download requests the JNLP specification defines and delivers pack200 compressed jar files to clients that are requesting them.

- *com.ulcjava.container.servlet.server.servlets.ResourceDownloader*

Helper Servlet that provides resources from the application. This is used to access the application's icons from the html page or the java web start environment. The servlet provides access only to resources for which a mime-mapping exists.

- *com.ulcjava.container.servlet.server.servlets.ConfigPropertiesDownloader*

Helper Servlet that provides your application's ULC configuration file.

- *com.ulcjava.container.servlet.server.IndexServlet*

Helper Servlet that dispatches a request to either a html page that contains the applet for your ULC application or the jnlp file that runs your application with Java Web start, depending on the value of the *defaultClientEnvironment* element

³ See <http://java.sun.com/j2se/1.5.0/docs/guide/javaws/developersguide/downloadervletguide.html>

in the `ULCApplicationConfiguration.xml` file. Two init parameter are used to define the dispatching targets :

```
<init-param>
  <param-name>applet-redirect</param-name>
  <param-value>/@projectname@.jsp</param-value>
</init-param>
<init-param>
  <param-name>jnlp-redirect</param-name>
  <param-value>@projectname@.jnlp</param-value>
</init-param>
```

- *com.ulcjava.container.servlet.server.ServletContainerAdapter*
Servlet that provides your ULC application.
- *com.ulcjava.container.servlet.server.servlets.ClientJarPreparationListener*
Uses the configurable `IClientResourceHandler` to prepare the client jar files on server startup.
- *com.ulcjava.easydeployment.server.DefaultClientResourceHandler*
Prepares the client jars for downloading by performing the following:
 1. All `*-client.jar` files in the `/WEB-INF/lib` directory are copied to the document root.
 2. A pack200-gzip compressed version of the jar file is created in the document root.
 3. If a keystore is specified within the `WEB-INF/jarsigner.properties` file, it is used to sign the client jar files.

7.4 How to integrate the applet into your own application

Use the `ulcApplet.tag` in your own jsp or adapt the generated `ProjectName.jsp` to your needs. The tag can be found in directory `<Canoo RIA Suite Install Dir>/addon/generators/templates/projecttemplates/server/WebContent/WEB-INF/tags`.

7.5 How to run your application outside a browser

The recommended way to run your application outside a browser is to run it inside the Java Web Start environment (see http://en.wikipedia.org/wiki/Java_web_start for more information).

The easiest way to provide your application for java web start is to set the value of the `defaultClientEnvironment` element in the `ULCApplicationConfiguration.xml` file to `JNLP`.

```
<?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:applicationName>
    com.canoo.TeamMember
  </ulc:applicationName>
  <ulc:defaultClientEnvironment>JNLP</ulc:defaultClientEnvironment>
</ulc:ULCApplicationConfiguration>
```

Clients that are accessing <http://localhost:8080/TeamMember-server/> will now get the jnlp file that launches the application. The JNLP file is provided by the `com.ulcjava.container.servlet.server.servlets.JNLPDownloader`.

As you can see in the `web.xml`, this Servlet is mapped to the `/<ProjectName>.jnlp` URL pattern. If you point your browser to `http://localhost:8080/TeamMember-`

server/TeamMember.jnlp the application is started with Java Web Start independent of the configuration.

This allows each client to choose the way the application is run by choosing the appropriate URL.

<http://localhost:8080/TeamMember-server/TeamMember.jnlp> for outside the browser

<http://localhost:8080/TeamMember-server/TeamMember.jsp> for inside the browser.

7.6 How to integrate into the client environment

Client desktop integration can be controlled by the entries in the ULCApplicationConfiguration.xml and the application's properties file (see Chapter 8). Use the <desktopIntegration> element of the configuration file to set shortcuts and file associations. The <java> element lets you control the parameter of the client vm.

forceAllPermissions tag of java element will make the ULC client to run outside of the sandbox with help of the all-permissions element in the JNLP file.

Various keys in the *Application.properties* are used to retrieve localized resources for the application as for example the application's title and icons.

7.7 How to secure ULC applications

Similar to a web browser, ULC offers a variety of measures to meet stringent security requirements. Basically, it is recommended that the UI Engine is run in a sandbox. A sandbox is provided automatically when you run the UI Engine as an applet or through Java Web Start. In addition, protocols like HTTP(S) or IIOPS provide encryption and authentication.

The UI Engine can also be adapted to different security environments. In a trusted environment it may run outside the sandbox if it is restricted to connect to trusted hosts only. The UI Engine can also be signed by a trusted entity and restricted to connect to specific hosts only. Finally, using the Java Security Architecture, restricted access to security-sensitive functionality (e.g., access to portions of the file system) can be enabled.

7.8 How to sign the application's libraries with your certificate

In the generated application's WebContent/WEB-INF directory is the signjar.properties file that contains all the needed data to automatically sign the client jar files.

signjar.keystore	File name of the keystore. The file must be in WebContent/WEB-INF. If this entry is removed or commented, no signing takes places.
signjar.alias	Required: Alias of the key to sign the jars with.
signjar.storepass	Required: password of the keystore
signjar.keypass	Password of the key. If missing the signjar.storepass is used.

To use your own certificate you must save the keystore file with the certificate into the WebContent/WEB-INF directory and adapt the signjar.properties file.

7.9 How to handle HTTP session timeout

The session timeout of the web application must be configured to be compatible with the server environment (e.g., load balancing mechanisms) and with the client configuration (e.g. using “*keep-alive*” requests). Setting *keep-alive-interval* to be less than session timeout interval will make sure that `HttpSession` does not time out when a ULC client is idling due to lack of user interaction.

To be able to detect session timeout you may specify in the **web.xml** the class `com.ulcjava.container.servlet.server.HttpSessionListener` which notifies ULC applications after the `HttpSession` has been destroyed (e.g. because of a HTTP session timeout). When the `HttpSession` is destroyed, `HttpSessionListener` gets `ULCSessions` from the `SessionStore` and calls on it `sessionDestroyed()` which in turn calls `IApplication.stop()` thus terminating the ULC application. The installation of `HttpSessionListener` requires servlet specification 2.3 and the following listener tag added to the `web.xml` of the ULC application:

```
<DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <listener>
    <listener-class>
      com.ulcjava.container.servlet.server.HttpSessionListener
    </listener-class>
  </listener>

  <servlet>
    ...
  </web-app>
```

7.10 How to do Authentication and Authorization

`ApplicationContext.getUserPrincipal()` method returns authentication information provided by the Servlet container. The ULC Servlet container adapter returns the information as provided by the `getUserPrincipal()` method of `javax.servlet.http.HttpServletRequest`. Please see your Servlet container’s documentation for information on how to set up *user principal* support.

`ApplicationContext.isUserInRole()` returns a boolean value indicating whether the authenticated user is included in the specified logical role. Roles and role membership must be specified in the deployment descriptor.

7.11 Deploying a ULC application in standalone/offline mode

Please see chapter 3 of `<Canoo RIA Suite Install Dir>/doc/ULCDeploymentGuide`.

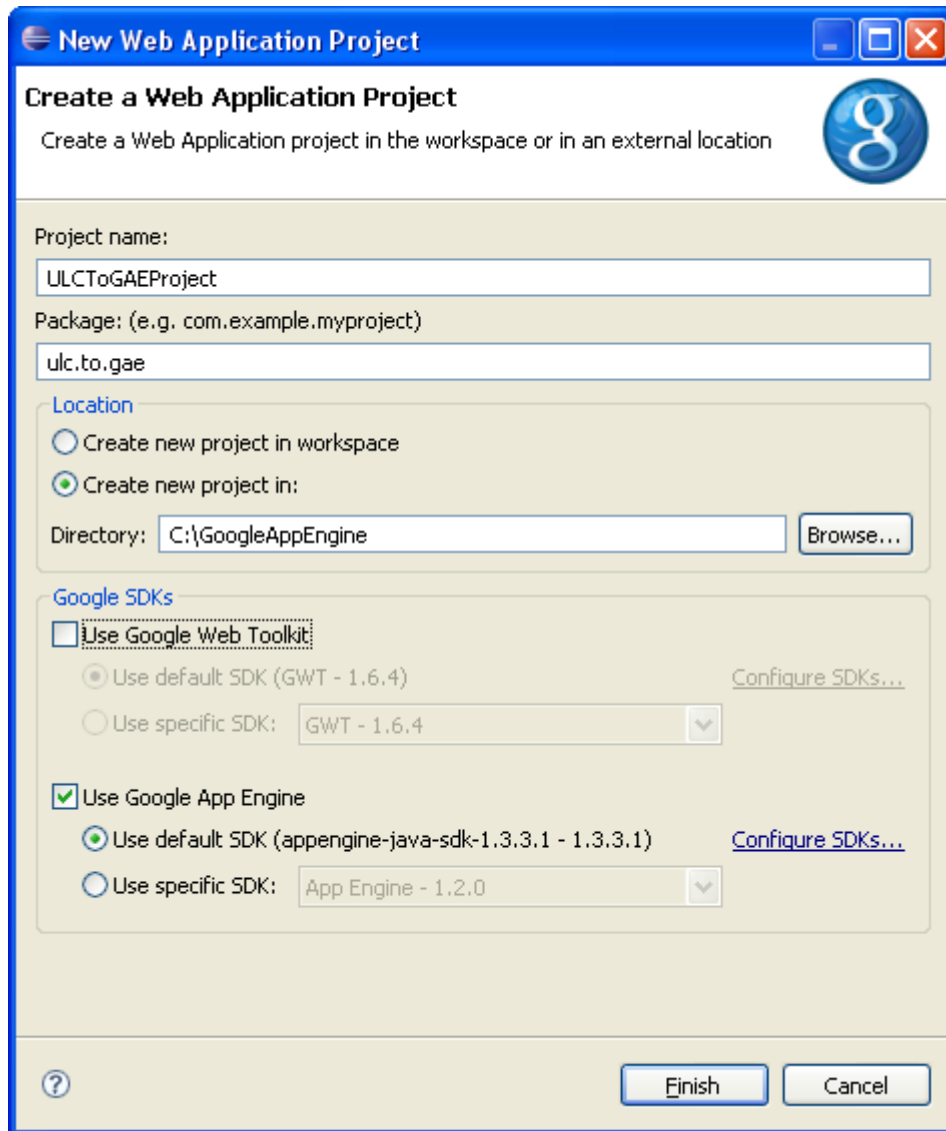
7.12 Google App Deployment

ULC applications run in a standard JEE Web Container, therefore they can be deployed into the [Google App Engine](#) (GAE).

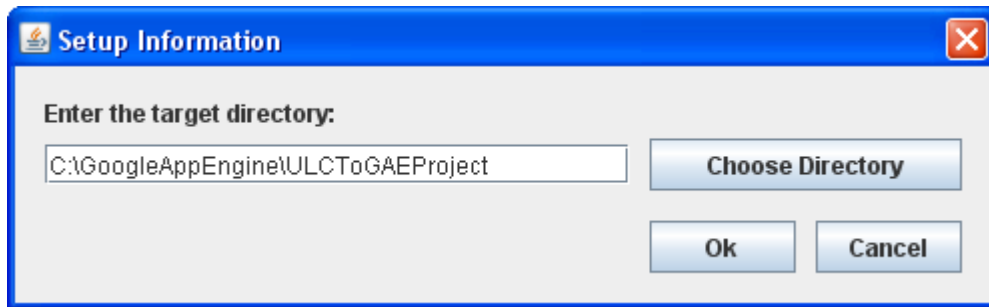
To work with the GAE you must [Sign up](#) for an App Engine account and install the [Google Plugin for Eclipse](#). (See <http://code.google.com/appengine/>)

The GAE Environment has some limitations which the developer has to respect and the project must be package in a certain way. A project that is created with the ULC Project generator (see 2.2) contains an ant script that copies the ULC project into a GAE project and packs it to be ready for upload.

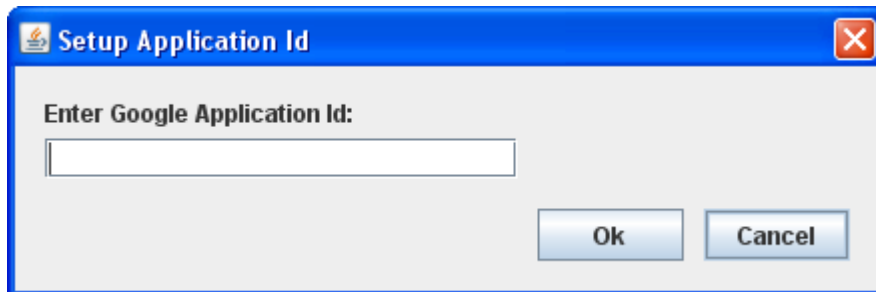
1. Create a ULC project (see 2.2) and write the application.
2. Create a new Web Application Project for the Google App Engine.




3. Select the **Run > External Tools > *ProjectName*-copy-to-GoogleApp** menu item to package and copy your project into the Google App Engine Project created in the second step. You will be asked for the path to the Google App Engine project



and the Google Application ID.



4. Now you can test the Google App Engine Project by launching it locally. Use the button  to deploy it the Google App Engine.
- 5.

Please refer to the [Google App Engine for Java](http://code.google.com/appengine/docs/java/overview.html) (<http://code.google.com/appengine/docs/java/overview.html>) documentation for further information.

8 The ULC Application Configuration

8.1 ULCAApplicationConfiguration.xml

ULCAApplicationConfiguration.xml file can be used to configure ULC client as well as the ULC application on the server. All the tags are optional, except **applicationClassName**.

applicationClassName	The full class name of the implementation of com.ulcjava.base.application.IApplication to be started.
defaultClientEnvironment	Defines the client environment in which to run the application using the ContextRoot URL. Allowed values are Applet and JNLP . If not defined Applet is used.
keepAliveInterval	The keep alive interval in seconds.
clientLogLevel	Sets the log level on the client side. In an integrated environment, like the DevelopmentRunner, this value is ignored. Allowed values are ALL , CONFIG , SEVERE , FINE , FINER , FINEST , INFO , OFF and WARNING .
serverLogLevel	Sets the log level on the server side. In an integrated environment, like the DevelopmentRunner, this value is the only one used to set the log level. Allowed values are ALL , CONFIG , SEVERE , FINE , FINER , FINEST , INFO , OFF and WARNING .
lookAndFeel	Defines which Look and Feel is going to be used on the client.
standardLookAndFeel	Select one of the standard Look and Feels that are provided by the UIManager. Allowed values are SYSTEM and CROSSPLATFORM .
lookAndFeelClassName	The full class name of the Look and Feel that is going to be used on the client.
modelAdapterProviderClassName	The full class name of the implementation of the com.ulcjava.base.server.IModelAdapterProvider interface to be used on the server.
dataStreamProviderClassName	Defines which implementation of com.ulcjava.base.shared.IDataStreamProvider is to be used on each side.
connectorCommandFailureStrategyProviderClassName	Defines the implementation of com.ulcjava.container.servlet.client.IConnectorCo

	mmandFailureStrategyProvider that is instantiated on the client to handle communication errors. The class must have an accessible no-arg constructor.
clientErrorHandlerClassName	Defines the implementation of com.ulcjava.base.client.launcher.IErrorHandler that is instantiated on the client to present the error that causes the session to terminate to the user. The class must have an accessible no-arg constructor.
coderRegistryProvider	Defines which implementation of the com.ulcjava.base.shared.ICoderRegistryProvider interface is going to be used on each side. This element should be used only when you want to use customized coders in place of predefined coders provided by ULC. It is recommended that you use the <i>coders</i> element which allows to register coders without writing a coder registry provider class.
clientCoderRegistry-ProviderClassName	The full class name of the implementation of com.ulcjava.base.shared.ICoderRegistryProvider to be used on the client side.
serverCoderRegistry-ProviderClassName	The full class name of the implementation of com.ulcjava.base.shared.ICoderRegistryProvider to be used on the server side.
carrierStreamProvider	Defines which implementation of the com.ulcjava.base.shared.ICarrierStreamProvider interface is going to be used on either side.
standardCarrierStreamProvider	Select one of the standard implementations of the com.ulcjava.base.shared.ICarrierStreamProvider interface. Allowed values are TRIVIAL , ZIP and BASE64 .
carrierStreamProviderClassName	The full class name of the implementation of the com.ulcjava.base.shared.ICarrierStreamProvider interface to be used on both sides.
messageServiceClassName	The full class name of the implementation of the com.ulcjava.base.client.IMessageService interface to be used on the client side.
browserService	Defines which implementation of the com.ulcjava.base.client.IBrowserService is to be used on the client side.
standardBrowserServices	Select one of the standard implementations of the com.ulcjava.base.client.IBrowserService interface. Allowed values are AllPermissions , Applet and JNLP .

browserServiceClassName	The full class name of the implementation of com.ulcjava.base.client.IBrowserService to be used on client side.
fileService	Defines which implementation of com.ulcjava.base.client.IFileService is to be used on the client side.
standardFileService	Selects one of the standard implementations of the com.ulcjava.base.client.IFileService interface. Allowed values are AllPermissions and JNLP .
fileServiceClassName	The full class name of the implementation of com.ulcjava.base.client.IFileService to be used on client side.
desktopIntegration	Defines the application meta data used for the JNLP file's information tag. The information data that can be internationalized has to be defined in the <ApplicationClassName>.properties file.
association	A hint to the JNLP client that it wishes to be registered with the operating system as the primary handler of certain extensions and a certain mime-type.
mimeType	The mime-type for which the application wishes to be registered as primary handler.
extensions	A list of file extensions (separated by spaces) for which the application wishes to be registered as primary handler.
shortcut	A hint for the JNLP client which shortcuts should be created.
desktop	Used in the JNLP file as hint to create a shortcut on the client's desktop.
menu	Hint to create a menu item on the client's start menu. If the shortcut should be added to a menu group, the menu group's title must be provided in the <ApplicationClassName>.properties with the key deployment_parameter.menu_group.
Java	Definitions for the client java runtime.
version	The java version required to run the application. Legal values are 1.5, 1.5+, If not specified 1.5+ is used for the jnlp file.
vmArgs	Arguments passed to the client vm.
initialHeapSize	Indicates the initial size of the Java heap in Megabyte.
maxHeapSize	Indicates the maximum size of the Java heap in Megabyte.

forceAllPermissions	Forces the all permissions security on the client .
coders	Additional Coder definition to be registered with the CoderRegistries on the client and server side.
asymmetricCoder	Use this for classes that are represented differently on the client and server side (e.g java.awt.Color and com.ulcjava.base.application.util.Color). This defines the classes and coders on each side.
clientCoder	This defines the classes and coder on the client side. See the coder type at the end of the table.
serverCoder	This defines the classes and coder on the server side. See the coder type at the end of the table.
symmetricCoder	Use this to register a coder for classes that are used on the server and client side (e.g. Date.class and DateCoder). See the coder type at the end of the table.
serverSessionProviderClassName	Class name of the session provider class, a factory that creates the ULCSession on the server side. The configured class must inherit from com.ulcjava.base.server.DefaultSessionProvider.
jnlplauncherClassName	The full class name of the Launcher that is run if the application is started with Java Web Start.
appletLauncherClassName	The full class name of the Launcher that is run if the application is started as an Applet.
clientResources	
directory	The directory where client resources are originally located. Default value is 'WEB-INF/lib' (relative to the servlet container context).
pattern	The pattern which will identify client resources in specified client resource directory. Default value is '*-client.jar' (* = any string, ? = any character).
handlerClassName	The <i>IClientResourceHandler</i> implementation class name. Specifies how/where to store/access the client resource. Default value is <i>com.ulcjava.easydeployment.server.DefaultClientResourceHandler</i> : client resources are located on the file system and are accessible via '/' (relative to servlet container context).

The coder type is defined as having one or more class elements and a coder element:

class	The full qualified name of the class that is serialized / deserialized by the coder.
coderClassName	The full qualified name of the class that serializes / <u>deserializes</u> objects of the class defined by the <i>class</i> element. The coder class

	must implement the IStreamCoder interface and must have either a default constructor or a constructor that takes a Class object as parameter. If the coder has a constructor with a Class object parameter the Class that the coder is registered for is used to create the coder instance.
--	---

8.2 The Application Properties file

The properties file for a generated application contains some keys that are used by the easy deployment servlets. The value in brackets denotes the element of the information element of the JNLP file that is set to the property.

Application.title	The title of the application.
Application.vendor	The vendor of the application.
Application.homepage	URL pointing to where more information on this application can be found.
Application.description	Short description of the application.
Application.icon.16	The path to an image file relative to the document root. Used to represent the application in the browser. See http://en.wikipedia.org/wiki/Favicon for more information.
Application.icon.32 Application.icon.64	The path to an image file relative to the document root. Used to represent the application on the desktop. The number at the end denotes the size of the image.
Application.splash	The path to an image file relative to the document root. Used as splash screen for the application.
Application.menu_group	Name of the start menu group where a shortcut for the application is created.