



Roma

<Meta> Framework

Handbook

v 2.1

www.romaframework.org

Document Disclaimer

Information contained in this manual is subject to change without notice. Please check the web site (see Download Information) to make sure you have the most recent information.

Download Information

For download this document go to <http://www.romaframework.org> and go to the **Documentation** page.

Trademarks and Service Marks:

- **Java** is a trademark of Sun Microsystems Inc.
- **JDO** is a trademark of Sun Microsystems Inc.
- **Microsoft, Windows, and Windows NT** are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
- **UNIX** is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.
- **MacOsX** is a registered trademark of Apple Inc.





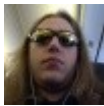


All brand names and product names are trademarks or registered trademarks of their respective companies.

This handbook has been written by many hands, all volunteers.

Surely you'll find grammatic errors, layout issues, parts of code not updated with the last changes. Please have patience. The reason is that we're all coders and most of us are not master with the English language :-)

If you'd like to improve this handbook feel free to send a message in the Roma Meta Framework forum.

Authors

Luca Garulli		Main ideator and supervisor of the project. He developed the first prototype in the 2005. He's the author of the modules: Core, Console wizards , Persistence-Jpox, Persistence-Datanucleus , Admin , Users , Scripting-Java6 , Frontend and View-Echo2
Luigi Dell'Aquila		Roma developer since July 2007. Designer and (co)author of the following Roma modules: Workflow Tevere Engine and GUI, Janiculum View Aspect , Portlet Project, Chart jFreeChart, Service CXF , Etl XPath, Semantic Jena
Giordano Maestro		Roma developer since July 2007. Designer and (co)author of the following Roma modules: Workflow Tevere Engine and GUI, Janiculum View Aspect , Reporting Aspect , Service CXF , Designer Module
Emanuele Tagliaferri		Developer of most of the features of Roma 2.x, specially regarding the Core. Author of Serialization Aspect and DWR Service Module. Contribution in Core, Janiculum View Aspect , REST Service
Luca Molino		Roma developer since 2006 and author of Messaging Module , Workflow Pojo, Portal Solo. Contribution to the modules: View-Echo2 , Persistence-Jpox, Core, Etl-Xpath and Admin
Marco De Stefano		Contribution to the modules View-Echo2 , Core and Admin
Paolo Spizzirri		Author of the Scheduler Aspect and Scheduler Quartz module

[Marco
Gentile](#)



Author of check, install and update [Console wizards](#)

Contents

- 1 Before to read.....11**
- 2 Introduction.....12**
 - 2.1 Decoupling Solution For Seamless Integration..... 12
 - 2.2 Fast Development..... 12
 - 2.3 Advanced Technology..... 13
 - 2.4 Semantic Web..... 14
- 3 Get and install.....15**
 - 3.1 Pre-requirements..... 15
 - 3.2 Install from binaries..... 15
 - 3.1 Install from sources..... 15
- 5 Create your first application.....18**
 - 5.1 Create the project..... 18
 - 5.2 Project scaffolding..... 18
 - 5.3 Import the project in Eclipse IDE..... 19
 - 5.4 Start the application..... 20
 - 5.5 Write the domain classes..... 23
 - 5.6 Generate CRUD for the domain entity classes..... 25
 - 5.7 Debug the application using Eclipse..... 27
 - 5.8 More about CRUDs..... 28
- 6 The Meta-Framework concept.....30**
 - 6.1 Aspects..... 30
 - 6.2 Aspects Overview..... 31
 - 6.3 Modules..... 32
 - 6.4 Modules Overview..... 32
 - 6.5 Accessing to Aspects and Components..... 36
 - 6.6 DDD Domain Driven Development..... 37
 - 6.7 Enrich you model using annotations..... 37
 - 6.8 Change the meta-model at run-time..... 40
- 7 Console wizards.....42**
 - 7.1 Console create command..... 43
 - 7.2 Console info command on projects..... 44
 - 7.3 Console list command..... 44

7.4 Console switch command.....	45
7.5 Console get command.....	45
7.6 Console set command.....	46
7.7 Console add command.....	46
7.8 Console crud command.....	47
7.9 Console check command.....	47
7.10 Console install command.....	48
7.11 Console upgrade command.....	48
7.12 Console uninstall command.....	49
7.13 Console info command on modules.....	49
7.14 Console update command.....	50
7.15 Console remove command.....	51
7.16 Console JSP command.....	52
8 CRUD.....	<u>53</u>
8.1 How CRUD works.....	59
8.2 Customize a CRUD.....	61
9 Conventions (over configuration).....	<u>68</u>
9.1 Extension by Composition.....	68
9.2 Override default behaviour using the events.....	71
10 Best Practices.....	<u>73</u>
10.1 Separate domain classes from presentation classes.....	73
10.2 Dirty approach: dirty you hands when required.....	74
11 Virtual Objects.....	<u>75</u>
12 Controller.....	<u>79</u>
12.1 Controller in action.....	80
12.2 Transparent binding of field values.....	80
13 Flow Aspect.....	<u>82</u>
13.1 Display a POJO.....	82
13.2 Go back.....	83
13.3 Access to the user history.....	83
13.4 Action Confirm.....	83
14 View Aspect.....	<u>85</u>
14.1 How Roma renders POJOs.....	85
14.2 Area concept.....	86
14.3 Create your form layout.....	87
14.4 Screen and Area concepts.....	88

14.5 Configurable Screens.....	88
14.6 Component placement in screens.....	89
14.7 RenderingResolver: how to select the component to render.....	90
14.8 Rendering modes.....	91
14.9 Layout mode.....	98
14.10 Order the fields and actions.....	100
14.11 Form Validation.....	102
14.12 Annotations.....	102
14.13 Push commands.....	108
14.14 Implementation: Echo2.....	111
14.15 Implementation: Janiculum.....	116
15 Validation Aspect.....	<u>126</u>
16 I18N Aspect.....	<u>131</u>
16.1 Interface.....	131
16.2 Implementations.....	131
17 Persistence Aspect.....	<u>134</u>
17.1 Choose the right strategy.....	134
17.2 Atomic strategy.....	135
17.3 Change strategy for a single operation.....	135
17.4 Java Interface.....	136
17.5 Query Patterns.....	137
17.6 Implementations.....	141
18 Hook Aspect.....	<u>144</u>
18.1 Scope.....	144
18.2 Change the behaviour at run-time.....	145
18.3 Collect results.....	146
19 Reporting Aspect.....	<u>147</u>
19.1 Architecture.....	148
19.2 Render Types.....	148
19.3 How to push a report.....	151
19.4 Template Generation.....	152
19.5 Template Manager.....	153
19.6 Implementations.....	153
20 Security Aspect.....	<u>155</u>
20.1 Main concepts.....	155
20.2 Annotations.....	155
20.3 The Secure interface.....	156

20.4 Available implementations.....	157
21 Logging Aspect.....	<u>158</u>
21.1 Example of usage.....	158
21.2 Messages conventions.....	159
21.3 The Loggers.....	160
21.4 Annotations reference.....	160
21.5 Implementations.....	161
22 Scripting Aspect.....	<u>162</u>
22.1 Interface.....	162
22.2 Usage.....	162
22.3 Advance topics.....	163
22.4 Implementations.....	164
23 Monitoring Aspect.....	<u>165</u>
23.1 Implementations.....	165
24 Semantic Aspect.....	<u>166</u>
24.1 Introduction.....	166
24.2 The Semantic Aspect purpose.....	166
24.3 Interface.....	167
24.4 Implementations.....	171
25 Workflow Aspect.....	<u>172</u>
25.1 Implementations.....	173
26 ETL.....	<u>174</u>
26.1 What is ETL.....	174
26.2 Etl Aspect architecture.....	174
26.3 Defining a custom ETL procedure.....	175
26.4 A custom ETL: from JDBC to JDBC.....	182
26.5 Validating imported data.....	184
27 Session Aspect.....	<u>185</u>
27.1 Interface.....	185
27.2 Usage Example.....	185
27.3 Get current logged user account.....	186
27.4 Implementations.....	187
28 Scheduler.....	<u>188</u>
28.1 Write a Job.....	188
28.2 Scheduler Control Panel.....	190

28.3 Creating a new job instance.....	190
28.4 Implementations.....	192
29 Portlet.....	<u>193</u>
30 Chart.....	<u>194</u>
30.1 Introduction.....	194
30.2 What kind of objects can be represented as charts.....	194
30.3 What kind of charts can be represented.....	194
30.4 Installing the chart module.....	196
30.5 Rendering a chart.....	197
31 Service Aspect.....	<u>202</u>
31.1 How to expose a service.....	202
31.2 Implementations.....	203
31.3 REST Service.....	210
32 Serialization Aspect.....	<u>215</u>
32.1 Java Interface.....	215
32.2 Serialization Inspection Strategy.....	215
32.3 Serialization Format Strategy.....	216
32.4 Example.....	216
33 Admin module.....	<u>217</u>
33.1 Info and InfoCategory.....	217
33.2 Realms.....	217
33.3 Plug-ins.....	217
33.4 Active sessions.....	218
33.5 Application Logs.....	218
33.6 Database.....	218
33.7 Environment.....	219
33.8 What is an Environment.....	219
33.9 Use Environment objects.....	219
33.10 Admin module to handle environment objects.....	220
33.11 How to access Admin functionalities.....	220
34 Users module.....	<u>221</u>
34.1 Dependencies.....	221
34.2 Standard Login.....	221
34.3 Activity Log.....	223
34.4 Accounts.....	224
34.5 Profiling system.....	224
34.6 Base Group.....	225

34.7 LDAP Authentication.....	225
34.8 Users Security Aspect	226
35 Designer module.....	228
35.1 Open the Wizard.....	230
35.3 Edit the Xml Annotation.....	231
36 Messaging module.....	232
36.1 Dependences.....	232
36.2 Display Messaging menu inside your application menu.....	232
37 Token Authentication module.....	233
37.1 Example usage.....	233
38 Tips & Tricks.....	235
38.1 Cascade events.....	235
38.2 Get text input from the user at the fly.....	236
38.3 Create a wizard.....	237
38.4 Show a collection field as editable using the TableWrapper component....	239
38.5 Expand a collection into the form using the RowsetWrapper component...	240
38.6 Handle collection as Master and Details.....	241
38.7 Create a tabbed CRUD Instance implementation.....	242
39 About Romulus.....	243
40 License.....	244

1 Before to read

Symbols used in the document:



Warning, reading this box is very important



Additional information to know



Web resource

All the code is wrapped inside a gray block:

```
if( true ){  
}
```

All commands to execute are wrapped inside a gray block with the prompt at the beginning:

```
> roma module add
```



In this guide we'll always refer to the scripts as .bat. If you're using a Unix/Linux/MacOSX Operative System use the script with the same name but with extension “.sh”.

2 Introduction

Roma allows you to develop enterprise level Java applications with low effort following a real **Domain Driven Design** approach. It's a new way to conceive the application: anything is a **POJO**, from the GUI forms to the persistent objects.

Using the Meta-Framework approach you can integrate the latest breaking technology in your application without modifying your domain and application logic because they are really [decoupled](#).

Roma provides you automatic support for every layer and aspect for your application, from dynamic web user interface and persistence, to report functionalities, portlet development and semantic technologies.

Roma is **Open Source** and commercial friendly [licensed](#).

2.1 Decoupling Solution For Seamless Integration

Every day a new tool or framework appears on the market. Using a technology means to have a **marriage** with it, very often forever. What happens if you want to change the DBMS you're using or the engine that persist your objects? Or again your web framework?

Divorces, you know, are costly. Many technologies don't follow standards so the migration cost is huge. Often it's cheaper to rewrite it from scratch.

Roma is the only “[meta](#)” framework. Using Roma you're definitely free from the technology you're using. How it works? Roma provides very generic behaviour interfaces called “[Aspects](#)”. Aspects enclose the most common use cases. Using the tool and framework through the Aspect allow to leave your code clean from the technology.

But if I need to dirt my hands using the technology directly, can I do it with Roma? Sure. You can always reach the implementation under the hood. Just remember you had to migrate that pieces of code in case tomorrow you'll change the technology. In the documentation you can find some best practice to make migration the easiest possible.

Abstracting the technologies let you to have the development team aware of their. This can be translated as **minor skill requested**, therefore **less cost**.

2.2 Fast Development

I know. So many times you thought that applications were quite similar. You're right! Technologies are horizontal and you can use the same ones in very different contexts. What really changes is the **Domain**. The Domain is the real value of your application, because you've analysed it, studied it and shared it with the client.

Roma lets you focus on the domain and treat the rest as a minor detail. It's a pure [DDD \(Domain Driven Design\)](#) approach.

How does it work ?

You start off by defining the entities/classes of your domain. You can write them by using an IDE of choice or you could have your preferred UML tool do the class generating for you.

Once your model is well documented, let Roma worry about the rest. I'm talking about the presentation, the persistence of your objects, the internationalization, etc. They can all be auto-determined figured out by analysing your domain model.

Since the Java language syntax is not enough to represent the complexity of your domain (think of all kinds of relationships) you could have the need to enrich your model by adding same details using [Annotations](#). Annotation can be under the form of [Java Annotation](#) (introduced with Java version 5) or [Xml Annotation](#).

By using [annotations](#) you can tell Roma to hide a field in your form or to render it in a different way, to express the kind of relationship between two entities, and much more.

Since you're not coupling your application to any specific technology and your domain is defined as a collection of **POJO's**, you can generate a fully featured web application in a short time and without writing one single line of HTML or SQL. Aspect implementations will deal with it. But remember you always own great powerful in customization if you need.

This greatly reduces development time and allows your team to focus on the domain, **without them having to be skilled up** on every single technology you're using.

Roma is the framework of choice of **hundreds of developers** who employ it, as we speak, to build **very complex applications** in a fraction of the time in comparison to conventional technologies.

2.3 Advanced Technology

Roma aims to provide integration with the most advanced technologies in the market. Each module cooperates with the available modules using the common Aspect interface mechanism. An Aspect is a Java Interface describing the behaviour of a set of modules. Think to the Persistence Aspect: it defines the most common operation against persistent objects like: load, update, delete, query, etc.

All the modules and your business logic will collaborate with the Aspects in generic way using the interfaces. The Aspect implementation will worry about the rest. Continuing with the example above you can plug a [JDO](#) or [JPA](#) module as implementation of the Persistence Aspect to works with your repository in transparent way.

Each component is handled by the [IoC](#) container. By default Roma uses the [Spring](#)

[Framework](#).

2.4 Semantic Web

Web 3.0 has become a reality; the number of web sites that provide semantic information is growing every day; semantic languages such as RDF, OWL and SPARQL are now well-established.

Is your application ready for this revolution? What do you have to do to expose your application domain as semantic web data? And what about importing in your application data coming from semantic web sources?

With Roma Meta-Framework you can expose your application data as semantic information without effort. You can also read semantic information from the Web and translate it in your application domain. Roma knows how to transform your POJOs into Semantic Web syntax such as RDF and how to bind semantic information in your application domain. You don't have to know RDF, you just have to write POJOs, adding them annotations with semantic information such as predicates, and Roma Semantic Aspect will do the trick.

3 Get and install

Roma Meta-Framework can be obtained from the web site, as a binary distribution or from Sourceforge SVN repositories, as a source distribution.

3.1 Pre-requirements

Roma Meta-Framework is a Java framework. In order to implement applications with Roma you have to install on your computer a Java compiler. Although most of Roma components are compiled and work with Java 5, in order to take advantage of all Roma functionalities, you should have Java 6 SDK installed. Go to [Sun's Java Home page](#) to download the right SDK.

3.2 Install from binaries

First [download a Roma binary distribution](#). You will find many packages, one for each module. For a quick start you just need to download the WebWizard file containing most useful modules to develop Web applications.

Then follow these steps:

- Extract the downloaded archive to a directory in your file system
- Set the **ROMA_HOME** environment variable to the Roma's binary installation directory just created.

3.1 Install from sources

3.1.1 Get sources

First of all you need to get the sources in order to compile its. We suggest to get always the latest sources from [SVN](#) since official releases are published every 3-4 months.

3.1.1.1 From Source distribution

1. [Download a sources distribution](#)
2. Extract the downloaded archive to a directory in your file system

3.1.1.1 By checking out from Subversion repository

1. Open a shell and goto in the path where you want to download the sources
2. Assure **SubVersion binary directory** is in the system path

3. Type:

```
svn co https://romaframework.svn.sourceforge.net/svnroot/romaframework/trunk
romaframework
```



You can use simpler tools than console SVN such as [Tortoise SVN](#). To know more information about **Sub Version** take a look to the [Roma's SourceForge SVN page](#)

3.1.1 Compile sources

3.1.1.1 Before compile

Before to compile and install the latest version of Roma assure to have the following tools:

Java5+

Java SDK 5.0+. Previous releases don't work since Roma uses Java5 features such as [Annotations](#) and Typed Collections. You can download it [here](#).

Java6+ for some additional modules

If you use additional modules such as [Scripting-Java6](#) please use JDK 6 since it needs advanced feature contained in Java 6.



Please assure to download the **JDK** (Java Development Kit) and not JRE (Java Runtime Environment)

Apache Ant

Apache Ant version 1.7.0 or major. You can download [here](#).

3.1.1.2 Compile it all

Follow these steps:

1. Change current directory to the Roma's **dev** directory and execute:

```
> ant install
```

2. Wait for building and when finished set the **ROMA_HOME** environment path

to the **dist** directory just created by the compilation process (it's at the same level of **dev** directory).

3.1.1 Test the installation

After the download and install test if anything is ok by changing current directory to the **ROMA_HOME** and executing the ROMA console:

4.1.1.1 Under Linux/Unix systems

```
> cd $ROMA_HOME  
> roma.sh
```

If you are using a bash console, maybe you would like to enable command auto completion (pressing <tab> key) in Roma Console. In Roma source distribution (under /dev/common/bash/ directory) you can find a script called “roma”. To enable auto completion you just have to copy this script on your system under /etc/bash_completion.d/ directory.

4.1.1.2 Under Windows systems

```
> cd %ROMA_HOME%  
> roma
```

If the installation succeeded, the help will appears.

5 Create your first application

Before we talk about Roma architecture, conventions and so much theory let's create our very first simple application. Then you can go deeper by reading the rest of the document learning advanced features and internal behaviour.

5.1 Create the project

Roma comes with a [console tool](#) to help in generating stuff. Assure to be in the %ROMA_HOME% (%ROMA_HOME for Linux/Unix/MacOSX os) directory or just set once for ever the \$ROMA_HOME directory in the PATH.

Now type the command '[roma project create](#)' (roma.sh for Linux/Unix/MacOSX users). The full syntax for the create wizard will be shown. See below:

```
ROMA Framework CONSOLE [http://www.romaframework.org]

Use this syntax:

roma project create <project-type> <project-name> <src-root-package>
[<project-path>]
  where: <project-type>      is the type of project between types locally
  available                  (see below)
        <project-name>      is the name of project to create
        <src-root-package>  Is the root package for sources
        <project-path>      is the path where to place the project scaffolding;
                           if not specified, the current path is taken

Project types locally available (discovered in the $ROMA_HOME/modules
directory):

- simple
- web
- webready

Example:

roma project create webready blog org.test.blog C:/temp
```

5.2 Project scaffolding

Now let's create our very first project using the Roma console. Type as follow:

```
> roma project create webready blog org.test.blog C:/temp
```

Your brand new application will be waiting for you at the specified directory ([C:/temp/blog](#) in this case). We decided to create an application of type **webready** because this wizard install the modules that a classic Web Application needs.

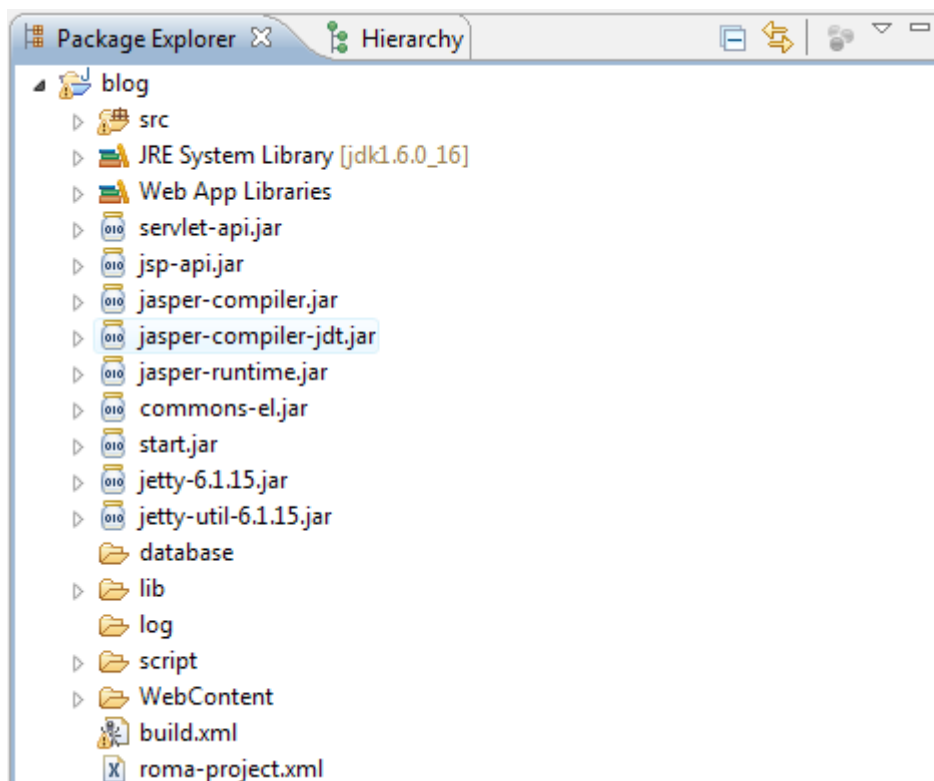
5.3 Import the project in Eclipse IDE

The create wizard generates the appropriate [Eclipse](#) meta-data files so that your project is ready to be imported into the IDE.



If you use another IDE you can create a new project and point it to the project directory.

1. Open [Eclipse](#) IDE
2. Select the menu *File* → *Import*
3. Select *Existing Projects into Workspace*
4. Change the root directory to '*C:/temp/blog*'
5. Select "blog" application between the project listed
6. Click on *Finish*



If you're using a recent Eclipse distribution the default settings are compatible with the Roma configuration, otherwise you need to follow these steps in order to be sure to use the right settings:

1. Right Click on the *Project* just imported and then *Properties*
2. Go to *Java Build Path* item and then select the *Libraries* tab
3. Assure **JDK 5.0** or **superior** is the used compiler.
4. Assure in the project properties the *Java Compiler* tab has *Compiler compliance level* ≥ 5.0



[Eclipse](#) warning: Some version of [Eclipse](#) has a strange bug: the project can't see the libraries under **WEB-INF/libs** until you close and reopen the project. If you notice this issue just close and reopen the project



In some configuration the project won't start unless you add the **Jasper Libraries**.

5.4 Start the application

The project just created is ready to be started. Before to write any line of code let's go to see your new application.

Go to the directory where is located the Blog project and execute the file *script/jetty-start.bat* if you're using Microsoft Windows® as Operative System.



In this guide we'll always refer to the scripts as `.bat`. If you're using a Unix/Linux/MacOsX Operative System use the script with the same name but with extension `".sh"`.

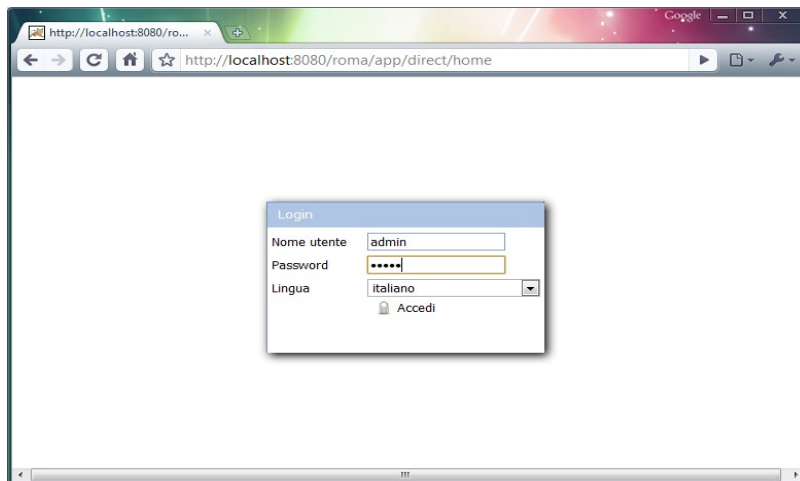
When the script starts a new Console window will be opened containing the log of the application.

```
C:\Windows\system32\cmd.exe
ger.SchemaConfigurationLoader.SchemaReLoader.ScreenConfigurationLoader.RenderingModeManager.AreaModeManager.LayoutModeManager.Login
gSpec.ConsoleLogger.ValidationAspect.SerializationAspect.SerializationInspectionStrategy.SerializationFormatStrategy.JsonFormatStr
ategy.JsonFullFormatStrategy.HookAspect.AuthenticationAspect.Echo2ViewAspect.Echo2SessionAspect.WebResourceResolver.LookAndFeelManag
er.ComponentFactory.DelayMessageManager.ScreenManager.RenderingResolver.FieldContainerFactory.FrontendModule.DataNucleusPersistenceM
odule.PersistenceAspect.TxPersistenceAspect.NoTxPersistenceAspect.OIDManager.JDOMetadataResourceResolver.RestServiceModule.UsersModu
le.UsersModuleGUIHook.UsersSecurityAspect.DBLogger.RealnRepository.ActivityLogRepository.BaseAccountRepository.BaseProfileRepository
.BaseGroupRepository.MessageSource.ApplicationConfiguration.AspectManager.ApplicationInstaller; root of factory hierarchy
00:57:26.283 WARN Extension Point 'org.jpox.store_mapping' not registered, but plugin '' defined in file:/C:/temp/blog/WebContent/
WEB-INF/lib/jpox-lob.jar refers to it.
00:57:26.283 WARN Extension Point 'org.jpox.store_datastoremapping' not registered, but plugin '' defined in file:/C:/temp/blog/Web
bContent/WEB-INF/lib/jpox-lob.jar refers to it.
00:57:26.290 WARN Extension Point 'org.jpox.store_mapping' not registered, but plugin '' defined in file:/C:/temp/blog/Web
bContent/WEB-INF/lib/jpox-lob.jar refers to it.
00:57:26.297 INFO Mlog clients using log4j logging.
00:57:26.992 INFO Initializing c3p0-0.9.1.2 [built 21-May-2007 15:04:56; debug? true; trace: 10]
00:57:27.135 INFO Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource [acquireIncrement -> 1, acquireRetryAttempt
s -> 30, acquireRetryDelay -> 1000, autoCommitOnClose -> false, automaticTestTable -> null, breakAfterAcquireFailure -> false, check
outTimeout -> 0, connectionCustomizerClassName -> null, connectionTesterClassName -> com.mchange.v2.c3p0.impl.DefaultConnectionTeste
r, dataSourceName -> z8Kf1t85y4vb50u8preo113cd27, debugUnreturnedConnectionStackTraces -> false, description -> null, driverClass ->
org.hsqldb.jdbcDriver, factoryClassLocation -> null, forceIgnoreUnresolvedTransactions -> false, identityToken -> z8Kf1t85y4vb50u8
preo113cd27, idleConnectionTestPeriod -> 600, initialPoolSize -> 1, jdbcUrl -> jdbc:hsqldb:file:database/testdb, maxAdministrativeT
askTime -> 0, maxConnectionAge -> 0, maxIdleTime -> 600, maxIdleTimeExcessConnections -> 0, maxPoolSize -> 6, maxStatements -> 0, ma
xStatementsPerConnection -> 0, minPoolSize -> 1, numHelperThreads -> 3, numThreadsAwaitingCheckoutDefaultUser -> 0, preferredDestQue
ry -> null, properties -> {user=*****}, propertyCycle -> 0, testConnectionOnCheckin -> false, testConnectionOnCheckout -> false, un
returnedConnectionTimeout -> 0, usesTraditionalReflectiveProxies -> false]
00:57:28.112 INFO Loading i18n bundles from path: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n
00:57:28.112 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\admin_messages_en.properties
00:57:28.113 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\admin_messages_it.properties
00:57:28.113 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\default_messages_en.properties
00:57:28.113 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\default_messages_it.properties
00:57:28.113 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\domain_messages_en.properties
00:57:28.113 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\domain_messages_it.properties
00:57:28.805 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\users_messages_en.properties
00:57:28.805 INFO Loading bundle: C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\i18n\users_messages_it.properties
00:57:28.805 INFO DirectServiceModule Registered service 'direct' binded to class: DirectServiceImpl (class:org.test.blog.service.
DirectServiceImpl)
00:57:28.950 WARN Stylesheet defined in 'C:\temp\blog\WebContent\WEB-INF\classes\org\test\blog\views\style\application.stylesheet'
is invalid.
00:57:29.464 INFO Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource [acquireIncrement -> 1, acquireRetryAttempt
s -> 30, acquireRetryDelay -> 1000, autoCommitOnClose -> false, automaticTestTable -> null, breakAfterAcquireFailure -> false, check
outTimeout -> 0, connectionCustomizerClassName -> null, connectionTesterClassName -> com.mchange.v2.c3p0.impl.DefaultConnectionTeste
r, dataSourceName -> z8Kf1t85y4vb50u8preo113c53a8, debugUnreturnedConnectionStackTraces -> false, description -> null, driverClass ->
org.hsqldb.jdbcDriver, factoryClassLocation -> null, forceIgnoreUnresolvedTransactions -> false, identityToken -> z8Kf1t85y4vb50u8
preo113c53a8, idleConnectionTestPeriod -> 600, initialPoolSize -> 1, jdbcUrl -> jdbc:hsqldb:file:database/testdb, maxAdministrativeT
askTime -> 0, maxConnectionAge -> 0, maxIdleTime -> 600, maxIdleTimeExcessConnections -> 0, maxPoolSize -> 6, maxStatements -> 0, ma
xStatementsPerConnection -> 0, minPoolSize -> 1, numHelperThreads -> 3, numThreadsAwaitingCheckoutDefaultUser -> 0, preferredDestQue
ry -> null, properties -> {user=*****}, propertyCycle -> 0, testConnectionOnCheckin -> false, testConnectionOnCheckout -> false, un
returnedConnectionTimeout -> 0, usesTraditionalReflectiveProxies -> false]
00:57:29.537 WARN [SchemaClassReflection.readFields] Cannot find the annotation XML and/or Java annotation @CoreClass(entity=X.class)
for class: class org.romaframework.frontend.domain.entity.ComposedEntityInstance'. Since it's a ComposedEntity implementation an
notation is required to expand the entity correctly.
00:57:29.561 INFO ----- MEMORY USAGE -----
00:57:29.565 INFO HEAP INIT MEMORY: 33.554.431 bytes
00:57:29.567 INFO HEAP USED MEMORY: 35.478.184 bytes
00:57:29.568 INFO HEAP COMMITTED MEMORY: 71.892.992 bytes
00:57:29.570 INFO HEAP MAX MEMORY: 517.013.504 bytes
00:57:29.571 INFO -----
00:57:29.573 INFO NON HEAP INIT MEMORY: 19.136.512 bytes
00:57:29.575 INFO NON HEAP USED MEMORY: 23.005.832 bytes
00:57:29.578 INFO NON HEAP COMMITTED MEMORY: 23.724.032 bytes
00:57:29.581 INFO NON HEAP MAX MEMORY: 117.440.512 bytes
00:57:29.582 INFO -----
00:57:29.586 INFO [RomaWebFilter.init] Startup completed in 00:04.300.
2009-12-11 00:57:29.653::INFO: Started SelectChannelConnector@0.0.0.0:8080
```

The last messages tell you that the startup was executed without problems and that the Application listen to the localhost (127.0.0.1), port 8080.

Open your preferred Internet Browser at this location:

<http://localhost:8080/roma/app/direct/home>



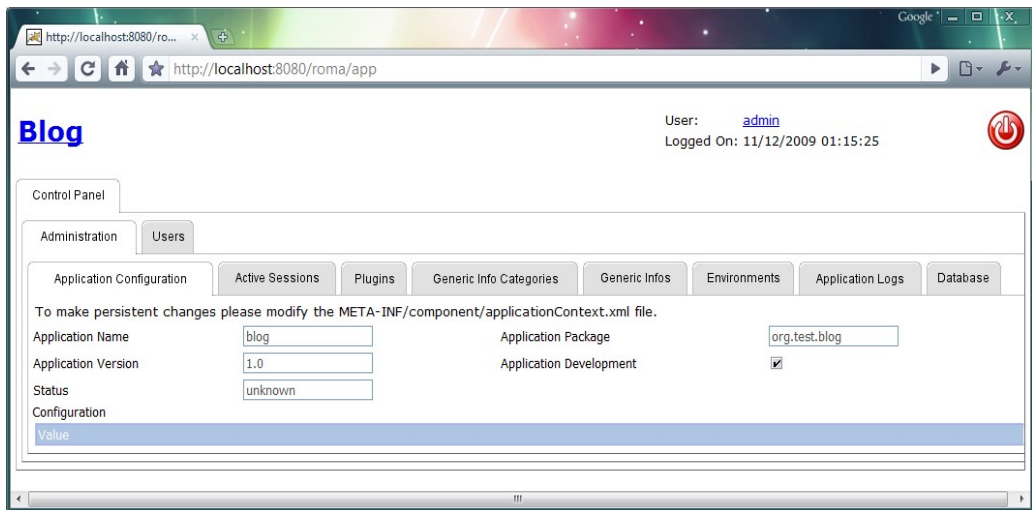
As you can see Roma supports the localization of messages in different languages. In this case the Italian language settings are used (see the [I18N Aspect](#)).

The first time the application starts, some auto configuration is done automatically. One of these regards the [Users Module](#). In particular Roma creates two default users:

- "admin" with password "admin" of profile "Administrator" and
- "user" with password "user" of profile "Basic"

To access to the application login as "admin" and password "admin".

The Homepage looks like the following image:



The Administrator profile allow to access to a huge Control Panel to setup the applications. To know in deep all the features go to [Admin Module](#) and [Users Module](#).

To stop the application type CTRL+C in the console window or just close the window.

5.5 Write the domain classes

Now let's go to create the class that represents the Blog entity. Remember that in Roma the most important thing is the Domain. Other aspects are secondary and driven by the Domain Model. Create the following classes under the **org.test.blog.domain.entity** package (Right click on src/org.test.blog.domain.entity and the click on "New Class").

Class Blog.java

```
package org.test.blog.domain.entity;

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

public class Blog {
    private String    name;
    private String    title;
    private String    author;
    private List<Post> posts = new ArrayList<Post>();

    public String getAuthor() {
        return author;
    }
}
```

```

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Post> getPosts() {
        return posts;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}

```

This class represents a Blog entity. Note that Blog class is a simple POJO (or also a JavaBeans, but we have not created constructor yet). It's a good practice to set the attributes as private and generate Getters and Setters methods for them. Eclipse generates Getter/Setter automatically by Right Clicking on the source window, selecting "Source" and then on "Generate Getters and Setters". Select all attributes and then click on "Ok".

Class Post.java

```

package org.test.blog.domain.entity;

import java.util.Date;

public class Post {
    private Date    postedOn;
    private String  title;
    private String  content;

    @Override
    public String toString() {
        return title != null ? title : "new";
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}

```



```

    }

    public Date getPostedOn() {
        return postedOn;
    }

    public void setPostedOn(Date postedOn) {
        this.postedOn = postedOn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}

```

This class represents a Post entity that is part of a Blog. Note that the class Blog contains a collection of references to Post objects. This relationships will be saved in the database transparently by Roma.



Remember to initialize any POJO's collections or they cannot handle any values!

The `toString()` method tells to Roma what to display in the list component. The object will be displayed in short form as the String returned by `toString()` method, in this case the post title.

5.6 Generate CRUD for the domain entity classes

At this point we can generate the CRUD using another wizard (to know more look at [generation of the CRUD](#)). The generation of the CRUD must be executed for each domain classes we need to have a CRUD. Open a console and go into the directory where Roma is installed and type this:

```
roma project crud org.test.blog.domain.entity.Blog
```

Now refresh the project in Eclipse IDE and a new package will appears: **org.test.blog.view.domain.blog**. Under this package there are the class created by the wizard ready to be used. The wizard analyses the domain class Blog and register that class as persistent.

For this reason, starting by now, we need to use the ANT compilation after or in place of the classic compilation of Eclipse. This assures the enhancement of the byte

code for the persistent classes. To know more look at the [Persistence Aspect](#).

To change the default mapping for the database you can act into the file **package.jdo** located in the source entity domain package. In our example is **src/org.test.blog.domain.entity**.

To compile the project you can do it using the console by changing the directory where is placed your project and just type:

```
ant
```

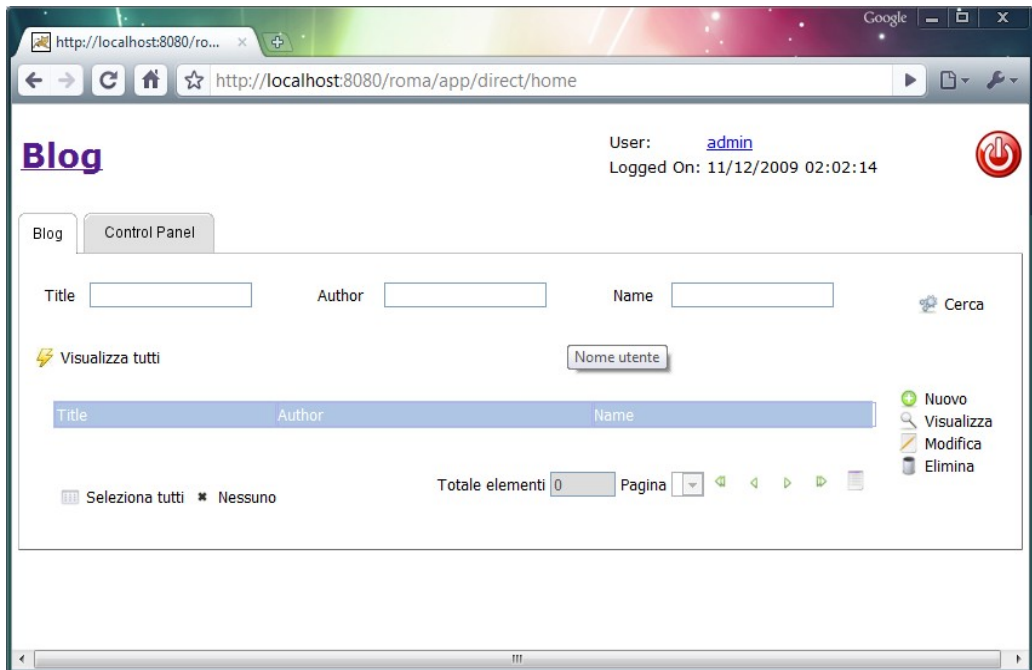
Apache Ant will find the **build.xml** configuration to compile our application. This file was generated by the first wizard when we created the project.

If you want to use Eclipse to execute ANT, right click on the **build.xml** file of your project and select Run As → Ant Build.

Every time you launch the persistence compilation assure that there are no problem on enhancing. The following message assure us that the Blog class is ready to be automagically persistent:

```
ENHANCED (PersistenceCapable) : org.test.blog.domain.entity.Blog
```

Now relaunch the application again (see above), re login and the result looks like the following image:



A new Tab was created containing the CRUD for the domain class Blog. From this page we can Create, Read, Update and Delete objects of class Blog.

5.7 Debug the application using Eclipse

Now configure a Run Task in Eclipse. This is needed only the first time:

1. Select the menu *Run* → *Open Debug Dialog...*
2. Select Java Application
3. Click on New button
4. Select in the Project field the blog project
5. Insert in the Main class field the value **org.mortbay.start.Main**. (Note: Versions before 1.2.0 must use **org.mortbay.jetty.Server** instead)
6. Select the Arguments tab
7. Write **src/jetty.xml** in the Program arguments field

8. Write `-Djetty.webapp=WebContent` in the VM arguments field
9. Click Apply button and then Run
10. Now open the browser at this address:
`http://localhost:8080/roma/app/direct/home`

Every time you want to re-start the application go to *Window -> Open Perspective -> Debug*, right click on the running instance of Jetty and select "Terminate and Relaunch". If you want to debug your application, right click on the running instance of Jetty and select "Terminate". Then click on the "Debug" button on the Eclipse Toolbar to start it in the debugger.

Once running in your browser press the "Edit Blogs" button and you will see the Blog entity automatically rendered by Roma.

The "Posts" field shows a list of posts. The "Add", "View", "Modify", "Remove" buttons create and manipulate posts in the list. "Author", "Title", and "Name" fields are global for the blog with the respective meaning.

Note that we have written no HTML code but our application it's web based and works totally using Ajax technology. Furthermore anything is Internationalized (I18N) based on the Locale setting of the browser connected (in the screenshot above is Italy).

Obviously you can change the graphic layout by:

- changing the *.stylesheet files in src directory
- adding more [Java Annotations](#) or [Xml Annotations](#) to change the behaviour
- create a custom Desktop

5.8 More about CRUDs

In the previous sections we have seen how to generate CRUD. What is a [CRUD](#)? Yes, [CRUD](#) is another buzzword and it stands for: **Create Read Update Delete**. These are the operations you often execute against your business objects.

The most of applications need to store information in persistent way. Very often the repository is a **Relational DBMS**. With Roma doesn't matter the kind of DBMS selected or the brand of DBMS since its **Persistence Aspect** uses the *JDO 2.0* technology. JDO is a standard to works with Objects in heterogeneous repositories. It can be also an Object Database such as *Orient ODBMS*.

The JDO implementation bundled with Roma distribution is [Persistence-Datanucleus](#) (that is the evolution of the [JPOX project](#) that is the Reference Implementation of the [JDO 2.0 standard](#)). You can use [Persistence-Datanucleus](#) with all its [supported](#)

Datastore.

As you can see we specify the class of objects that Roma will place in the database. Current release need to specify always the attribute **detachable="true"** for each classes. [Datanucleus](#) will auto-discover the class structure to gather fields information. The only thing to specify is the embedded instance for collection and map types.

In the snippet above, we have declared that **posts** attribute is a collection and it contains objects of type **Post**.

CRUDs in Roma are themselves **POJO** classes.

Roma's CRUD is divided in two parts:

- **Runtime library**, some classes inside
- **roma-core** library that handle CRUD events
- **Generated classes** by the wizard "crud".

You can change the sources generated adding more properties, actions or changing the default behaviour inherited by **CRUDMain**, **CRUDSelect** and **CRUDInstance** classes placed in the runtime library.

Now let's play with the CRUD creating new blogs, update them, delete someone, and so on; for more information about CRUD customization refer to the chapter [Customize a CRUD](#).

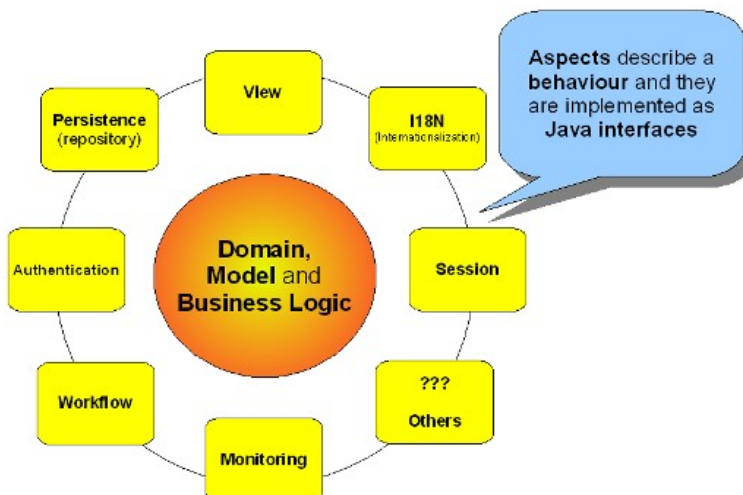
6 The Meta-Framework concept

6.1 Aspects

Roma Meta-Framework architecture is based on Aspects: an Aspect represents an aspect of the application, such as view or persistence. An Aspect is then implemented by one or more Modules. The aspect is the concept upon which Roma implements the Meta-Framework architecture: Roma Aspects are implemented as Java interfaces and using these interfaces you can write your application without being aware of the underlying implementation, so you can replace the current module that implements an Aspect with another one that uses a different technology without re-writing the application logic.

In this chapter we present a quick overview of Roma Aspects, describing their purpose. In the following chapters we will describe each Aspect in detail.

Architecture Atom I: Aspects



15/11/06

2

6.2 Aspects Overview

Aspect name	Description
Authentication Aspect	Is responsible of user authentication, with password encryption, group and realm hierarchies etc.
Flow Aspect	Handles the application flow
I18N Aspect	Manages application localization
Logging Aspect	Is responsible of application logs
Monitoring Aspect	Allows business objects to be monitored from outside the application
Persistence Aspect	Covers the persistence of objects. Its task is to allow object manipulation in persistent way.
Reporting Aspect	Allows to generate reports from POJOs
Security Aspect	Allows to enable/disable show/hide fields and actions of POJOs depends on user, role and profile
Service Aspect	Exposes POJOs as services
Scheduler Aspect	Allows to schedule events in Roma applications
Scripting Aspect	Adds scripting capabilities to Roma applications
Semantic Aspect	Generates semantic (Web 3.0) information from POJOs
Session Aspect	Handles user sessions
Serialization Aspect	Serialize objects and meta data to XML, JSON or native
Validation Aspect	Validates POJO with rules expressed in annotations or by using a custom one.
View Aspect	Provides view functionalities to the application
Workflow Aspect	Adds workflow functionalities to the application

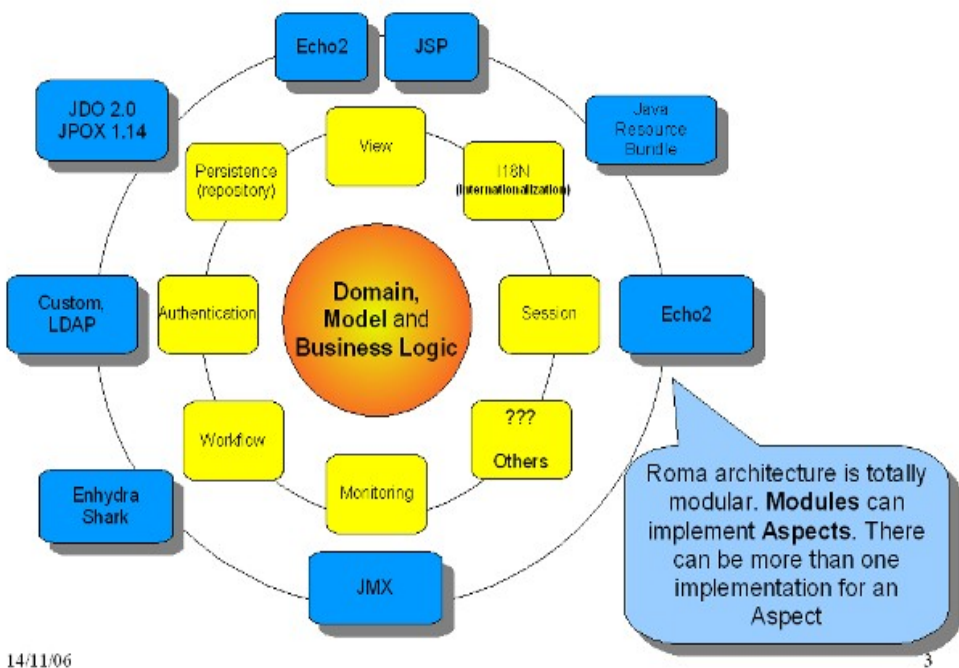
6.3 Modules

Modules are components that provide functionalities to Roma applications. A Module can implement one or more Aspects. In this chapter we present an overview of Roma modules available in the official Roma distribution and that are supported by Roma development group. Anyway the Meta-Framework approach or Roma allows everyone to develop a new Roma Module and to plug it in Roma applications.

6.3.1 Future plans

We're investigating on the native support for OSGi technology directly by the Roma core.

Architecture Atom II: Implementations



14/11/06

3

6.4 Modules Overview

Module name	Implemente	Dependenci	Description
-------------	------------	------------	-------------

	d Aspects	es	
Admin	-	-	Provides basic administration functionalities, like user profiling, application log and session monitoring etc.
Chart-JFreeChart	-	-	Enriches View Aspect and Reporting Aspect with the ability of rendering POJOs as charts. This module is based on JFreeChart library.
Core	Core I18N Flow Aspect Validation Security	-	Is the core module of all Roma applications. It contains Aspect interfaces, application controller and other basic functionalities. All Roma Modules depend on Core Module. This module also contains basic implementations of Flow Aspect and I18NAspect
Designer	-	Frontend	Allows to customize applications at runtime, changing POJO configurations in respect of enabled aspects.
ETL-XPath	-	-	Provides the ability to execute Spring XML based procedures to import data in Roma applications. This module is based on Spring IoC and Reverspring.
Frontend			It's the base module for all the View Aspect implementations.
Messaging	-	-	Adds mail and messaging functionalities to Roma applications, with complete email GUI and SMTP/POP3 support. This module is based on JavaMail technology.
Monitoring-JMX	Monitoring	-	Adds monitoring functionalities to Roma applications. This module is based on JMX technology.

Monitoring-MX4J	Monitoring	-	Adds monitoring functionalities to Roma applications. This module is based on MX4J technology.
Persistence-Datanucleus	Persistence	Persistence-JDO	Provides persistence capabilities to the application. This module is based on DataNucleus technology.
Portal-Solo	-	-	Provides simple portal functionalities for Roma web applications
Reporting-JR	Reporting	-	This is an implementation of Reporting Aspect based on JasperReports library.
Scheduler-Quartz	Scheduler	-	Adds to Roma applications the ability to schedule events. This module is based on Quartz scheduler.
Scripting-Java6	Scripting	-	Adds scripting capabilities to Roma applications. This module is based on Java6 Rhino JavaScript engine.
Semantic-Jena	Semantic	-	Allows to export POJOs as RDF feeds. This module is being developed under the Romulus Project, so it is not hosted in Roma repository.
Service CXF	Service	-	Exposes POJOs as web services. This module is based on Apache CXF technology.
Users	Security Logging	Admin	Provides users authentication functionalities.
View-Echo2	View Session	Web	Adds view capabilities to Roma applications, rendering POJOs as Echo2 AJAX pages
View-Janiculum	View Session	Web	Adds view capabilities to Roma applications, rendering POJOs as HTML pages. This module is being developed under the Romulus Project, so it is not

			hosted in Roma repository.
Web	View	-	This is an abstract implementation of the View Aspect based on HTTP. Web based View Aspect implementations such as View-Echo2 and View-Janiculum rely on this module.
Workflow Tevere Engine	Workflow	-	Adds complex workflow capabilities to the application. This module is being externalized as an independent project called TevereFlow.
Workflow Tevere GUI	Workflow	-	A web-based, AJAX GUI for Workflow Tevere Engine. This module is being externalized as an independent project called TevereFlow.

6.5 Accessing to Aspects and Components

Roma allows to access to each registered component and aspect by Roma class.

Get by **class type**. Convention wants each aspect's implementation is registered in the IoC system with the name of the interface:

```
Roma.component(MyComponent.class).doSomething(employee)
```

Get the component by the **registered name**:

```
Roma.component("MyComponent")
```

Get an aspect by class type or registered name, same as component's access but works only with Roma Aspects:

```
Roma.aspect("FlowAspect")  
  
Roma.aspect(FlowAspect.class).forward(new HomePage())
```

Get the component in **context**. Useful for [PersistenceAspect](#) to work with the same transaction:

```
Roma.context().component(PersistenceAspect.class).updateObject(employee)
```

Roma and **RomaFrontend** helper classes allow also direct access to the some aspects:

- `Roma.session()` --> `SessionAspect`
- `Roma.persistence()` --> `PersistenceAspect` (Atomic Persistence Aspect)
- `Roma.i18n()` --> `I18NASPECT`
- `Roma.validation()` --> [Validation Aspect](#)
- `Roma.scripting()` --> [Scripting Aspect](#)
- `Roma.context().persistence()` --> `PersistenceAspect` contained in Context.

- `RomaFrontend.flow()` --> [Flow Aspect](#)
- `RomaFrontend.view()` --> [View Aspect](#)
- `RomaFrontend.reporting()` --> [Reporting Aspect](#)

It's always suggested to use the shorter form, example:

```
Roma.context().persistence().updateObject(employee);

RomaFrontend.flow().back();

RomaFrontend.view().pushCommand(new
RedirectViewCommand("http://www.google.com"));
```



Please remember to use `Roma.context().persistence()` instead of `Roma.persistence()` to have the transaction automatically committed by the Roma engine. Use `Roma.persistence()` only if you want to have the full transaction control by your own.

6.6 DDD Domain Driven Development

Using Roma you place the domain to the centre of your development.

To know more about DDD methodology take a look at [DDD resources](#).

6.6.1 DDD Resources

- [Free E-Book Domain Driven Design Quickly](#)

6.7 Enrich your model using annotations

The Roma development is fully focused on domain. But since Java language is not enough to express complex domain model you need to use an enriched one.

Roma provides annotations under the form of [Java Annotations](#) and [Xml Annotations](#). Remember that on conflict [Xml Annotations](#) win.

Below an example of the same annotation in Java and XML.

ContactInstance.java:

```
public class ContactInstance{
    @ViewField(render = ViewConstants.RENDER_SELECT, selectionField =
```

```
"entity.type")
private List<ContactType> types;
}
```

ContactInstance.xml:

```
<?xml version="1.0"?>
<class>
  <fields>
    <field name="types">
      <aspects>
        <view render="select" selectionField="entity.type"/>
      </aspects>
    </field>
  </fields>
</class>
```

6.7.1 Annotations table

Below the table of available annotations divided in Aspects.

Aspects	Classes	Fields (properties)	Actions (methods)
Core	Core Class	Core Field	
Flow			Flow Action
Hook		Hook Field	Hook Action
I18N		I18n Field	
Logging	Logging Class	Logging Field	Logging Action
Monitoring	Monitoring Class	Monitoring Field	Monitoring Action
Security	Security Class	Security Field	Security Action
Semantic			
Service	Service Class		
Validation		Validation Field	Validation Action
View	View Class	View Field	View Action

6.7.2 Java Annotations

(<http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>). You can "annotate" classes, attributes and methods with the following syntax:

```
@<Annotation Name>[(<attribute> = <value>*)]
```

Where the annotation name is such composited:

<AspectName>Class: to annotate the **class** under the aspect <AspectName>

<AspectName>Field: to annotate the **field** under the aspect <AspectName>

<AspectName>Action: to annotate the **action** under the aspect <AspectName>

<AspectName>Event: to annotate the **event** under the aspect <AspectName>

6.7.2.1 Use XML Annotation instead

Remember you can always use the [Xml Annotations](#) instead of [Java Annotations](#) to maintain things separated. [Xml Annotations](#) are also used to build the [Virtual Objects](#).

6.7.2.2 Example

Below a complete example:

```
@ViewClass(render="popup")
@MonitoringClass(enabled=AnnotationConstants.TRUE)
class MyWindow{
    @ViewAction(validate=AnnotationConstants.TRUE)
    public void save()
    {
        saveAllInfoToDatabase();
    }

    @ViewField(label="Window position")
    public int getPosition()
    {
        return 4;
    }

    @FlowAction(next="WizardStep1")
    public void next()
    {
    }

    @ViewField(label="Name of the window")
    private String name;
}
```

6.7.3 XML Annotations

[Java Annotations](#) are a new feature of Java 5 platform (<http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>). You can

"annotate" classes, attributes and methods with the following syntax:

Someone prefer to separate the code from the meta model. For this reason you can use XML Annotation instead of [Java Annotations](#). In the case Roma finds a [Java Annotation](#) and a XML Annotation, the XML Annotation wins.

A XML Annotation must be written in a file with the same name of the class you want to apply the annotation but with **.xml** suffix. Example: if you want to define a XML Annotation for the class **Customer.java** you had to define the XML Annotation in a file called **Customer.xml**. By convention the XML Annotation file resides in the same package of Java class, but Roma find it just if is in the domain class-path.

Writing the XML Annotation assure it **validates against the XML Schema**: <http://www.romaframework.org/schema/v2/roma.xsd>.

Xml Annotations are also used to build [Virtual Objects](#), namely classes entirely defined using a descriptor instead of a real Java implementation.

6.7.3.1 Example

Below a complete example:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <actions>
    <action name="search">
      <aspects>
        <view label="$default.refresh" />
      </aspects>
    </action>
    <action name="send" />
    <action name="forward" />
    <action name="create">
      <aspects>
        <view visible="false" />
      </aspects>
    </action>
  </actions>
</class>
```

6.8 Change the meta-model at run-time

You can change the meta-model at run-time in the following ways:

- If you're in **debug mode** change the Java source and save it. Roma will catch the changes and will reload it.
- By changing the **XML Annotation** and save the file. Roma will catch the changes and will reload it.

- Via APIs. Example:

```
ObjectContext.getInstance().setClassFeature( this, ViewAspect.class,  
                                             ViewClassFeatures.LABEL, "My  
playlist" );  
  
ObjectContext.getInstance().setFieldFeature( this, ViewAspect.class,  
                                             "account", ViewActionFeatures.VISIBLE,  
false );  
  
ObjectContext.getInstance().setActionFeature( this, ViewAspect.class,  
                                             "save", ViewActionFeatures.ENABLED,  
false );
```

7 Console wizards



The [Romulus project](#) has developed IDE plug-ins for Eclipse and Net-Beans.

Roma comes with a console tool able to interact with the developer. The Roma console uses wizards to execute any operations. Wizards are based on Ant Build file and such custom commands written in Java. To invoke the console:

In Windows systems open the "Prompt of commands" and type:

```
roma
```

In Linux/Unix systems open a shell and type:

```
roma.sh
```

Assure you have the roma distribution in your path or change the current directory to the roma distribution before to type the command above. The result will be something like this:

```
ROMA Framework CONSOLE v.2.0 [http://www.romaframework.org]
```

```
Please specify the wizard to use between the following wizards discovered in  
the classpath:
```

```
- get [<variable-name>]  
- module add <module-name> [-p<project-name>]  
- module check [new]  
- module info <module-name>  
- module install <module-name> [<module-version>]  
- module uninstall <module-name>  
- module upgrade [<module-name> [<module-version>]]  
- project create <project-type> <project-name> <src-root-package> [<project-  
path>]  
- project crud <domain-class> [<output-class>] [-p<project-name>]  
- project info [-p<project-name>]  
- project list  
- project remove [-p<project-name>]  
- project switch <project-name> [<project-path>]  
- project update [-p<project-name>] [<module-name> [<module-version>]]  
- set <variable-name> '<variable-value>'
```

```
Example: roma project create web blog org.test.blog C:/temp
```

7.1 Console create command

This command create a new project.

7.1.1 Syntax

Syntax: `roma project create <project-type> <project-name> <src-root-package> [<project-path>]`

Where: `<project-type>` is the type of project between types locally availables (see below)

`<project-name>` is the name of project to create

`<src-root-package>` Is the root package for sources

`<project-path>` is the path where to place the project scaffolding; if not specified, the current path is taken. Optional

Project types are themselves modules installed under the modules directory. Currently the project types that came with the official distribution are:

`simple` Create a simple Java project

`web` Create a Web Java project

`webready` Create a Web Java project ready to be launched. It contains some modules in order to start developing in a flash.

`portlet` Create a Portlet project, ready to be deployed on a portal server (e.g. Liferay). The new project will contain a portlet whose name is the name of the project (to add new portlets see Portlet wizard)

This project type is hosted on Romulus repository.

7.1.2 Usage

```
roma project create webready blog com.mycompany.blog C:/temp
```

Creates a new project called "blog" of type webready under the directory [C:/temp/blog](#) using the Java package com.mycompany.blog.

7.2 Console info command on projects

This command shows the information about the current project.

7.2.1 Syntax

Syntax: `roma project info [-p<project-name>]`

Where: `<project-name>` is the name of project to show its info. Optional

7.2.2 Usage

```
roma project info
```

Shows the information about current project.

```
roma project info -pdemo
```

Shows the information about the "demo" project.

7.3 Console list command

This command displays all the project configured in Roma. This command reads the file placed under `$ROMA_HOME/config/projects.xml`. Feel free to change it by hand, but respects the XML formatting.

7.3.1 Syntax

Syntax: `roma project list`

7.3.2 Usage

```
roma project list
```

Displays all projects configured.

7.4 Console switch command

This command switch the current project to another one. The project to set as current can be already configured or not. If it's already configured you can just specify the name of the project to switch, otherwise the command needs the path of the project.

Use the switch command to change a project path.

7.4.1 Syntax

Syntax: roma project switch <project-name> [<project-path>]

Where: <project-name> is the name of project to create

 <project-path> is the path where to place the project scaffolding; if not specified, the current path is taken. Optional

7.4.2 Usage

```
roma project switch demo
```

Switch to the 'demo' project.

```
roma project switch blog C:/temp/blog
```

Switch to the 'blog' project located under [C:/temp/blog](#) directory. If the project is already configured it will be updated with the new path, otherwise a new entry will be created.

7.5 Console get command

This command returns one or all configuration values.

7.5.1 Syntax

Syntax: roma get [<variable-name>]

Where: <variable-name> is the name the variable. Optional

7.5.2 Usage

```
roma get
```

Displays all the configured variables.

```
roma get author
```

Displays the author variable.

7.6 Console set command

This command set the value of a variable.

7.6.1 Syntax

Syntax: `roma set <variable-name> '<variable-value>'`

Where: `<variable-name>` is the name the variable
 `<variable-value>` is the value of the variable

7.6.2 Usage

```
roma set author 'Luke Skywalker (luke.skywalker@starwars.com)'
```

Set the author variable to "Luke Skywalker (luke.skywalker@starwars.com)" so all generated code will contain such author in JavaDoc comments.

7.7 Console add command

This command adds a module in a user project. If no project is specified the current will be taken.

7.7.1 Syntax

Syntax: `roma module add [[<module-name>] [-p<project-name>]]`

Where: `<module-name>` is the name the module to add. Optional
 `<project-name>` is the name of the configured project. Optional

7.7.2 Usage

```
roma module add
```

Displays all the modules installed.

```
roma module add web-jetty
```

Add the module web-jetty to the current project.

```
roma module add persistence-datanucleus -pblog
```

Add the [Persistence-Datanucleus](#) module in the project configured as "blog".

7.8 Console crud command

This command generate a [CRUD](#) in a user project. If no project is specified the current will be taken. Roma will generate a set of files all under the package called as `<application-package>.view.<domain-class>` unless you specify the `<output-class>` parameter.

7.8.1 Syntax

Syntax: `roma project crud <domain-class> [<output-class>] [-p<project-name>]`

Where: `<domain-class>` the domain class where to generate the CRUD

`<output-class>` the package where to place classes. By default is `<application-package>.view.<domain-class>`

`<project-name>` is the name of the configured project. Optional

7.8.2 Usage

```
roma project crud org.romaframework.blog.domain.Blog
```

Generates the CRUD classes for the domain class Blog.

7.9 Console check command

This command checks for new version of installed modules. It will search in all the configured repositories placed under `$ROMA_HOME/config/repositories.xml` file.

7.9.1 Syntax

Syntax: `roma module check [new]`

Where: `new` Check for new modules

7.9.2 Usage

```
roma module check
```

Check for a new version of all installed modules

```
roma module check new
```

Check for all the modules not installed in the local distribution.

7.10 Console install command

This command install a new module in the local distribution.

7.10.1 Syntax

Syntax: roma module install <module-name> [<module-version>]

Where: <module-name> Name of the new module to install

 <version-name> Version to install. Optional, default is the latest one

7.10.2 Usage

```
roma module install scheduler-quartz
```

Install the latest version of "[Scheduler-Quartz](#)" module.

```
roma module install scheduler-quartz 1.1.0
```

Install the version 1.1.0 of "[Scheduler-Quartz](#)" module.

7.11 Console upgrade command

This command upgrade modules of own local distribution.

7.11.1 Syntax

Syntax: roma module upgrade [<module-name> [<module-version>]]

Where: <module-name> Name of the module to upgrade

 <version-name> Version to upgrade. Optional, default is the latest one

7.11.2 Usage

```
roma module upgrade
```

Upgrade the latest version of all module.

```
roma module upgrade scheduler-quartz
```

Upgrade the latest version of "[Scheduler-Quartz](#)" module.

```
roma module upgrade scheduler-quartz 1.1.0
```

Upgrade the version 1.1.0 of "[Scheduler-Quartz](#)" module.

7.12 Console uninstall command

This command uninstall a module of local distribution.

7.12.1 Syntax

Syntax: roma module uninstall <module-name>

Where: <module-name> Name of the module to uninstall

7.12.2 Usage

```
roma module uninstall scheduler-quartz
```

Uninstall the “[Scheduler-Quartz](#)” module.

7.13 Console info command on modules

This command shows the information about a module in the local distribution.

7.13.1 Syntax

Syntax: roma module info <module-name>

Where: <module-name> is the name of module which you want to analyze

7.13.2 Usage

```
roma module info web-jetty
```

Shows the information about the "web-jetty" module.

7.14 Console update command

This command update a Roma project with a new module installed locally.

7.14.1 Syntax

Syntax: `roma project update [-p<project-name>][<module-name> [<module-version>]]`

Where: `<project-name>` is the name of the configured project. Optional

`<module-name>` Name of the module to update

`<module-version>` Module version to update in the project

7.14.2 Usage

```
roma project update persistence-jpox
```

Updates the module 'persistence-jpox' of the current project to the version available locally.

```
roma project update admin 1.0.3
```

Updates the module [Admin](#) of the current project to the version 1.0.3.

```
roma module upgrade scheduler-quartz 1.1.0
```

Upgrade the version 1.1.0 of "[Scheduler-Quartz](#)" module.

```
roma module install view-echo2
```

Install the latest version of "[View-Echo2](#)" module never installed before.

```
roma module install scheduler-quartz 1.1.0
```

Install the version 1.1.0 of "[Scheduler-Quartz](#)" module never installed before.

7.15 Console remove command

This command remove a Roma project from project list. This command reads the file placed under \$ROMA_HOME/config/**projects.xml**.

7.15.1 Syntax

Syntax: roma project remove [-p<project-name>]

Where: <project-name> is the name of the project to remove from project list

7.15.2 Usage

```
roma project remove -pdemo
```

Remove project “demo” from project list.

7.16 Console JSP command

This command generates Java Server Pages (JSP) to customize how a POJO is rendered. It works along with the [View-Janiculum](#) module (not the [View-Echo2](#)). The generated JSP will have all class members expanded. It uses the [View-Janiculum](#) module JSP tag library.



This is a 3rd party contribution from the [Romulus project](#).

7.16.1 Syntax

Syntax: `roma jsp <domain-class> | * [-p<project-name>]`

Where: `<project-name>` is the name of the configured project. Optional
 `<domain-class>` the domain class where to generate the JSP. To generate JSP for all the classes under a such package use `<package>.*`

7.16.2 Usage

```
roma jsp org.test.domain.Airport
```

Generates the JSP called `Airport.jsp` under the directory `<project-path>/WebContent/dynamic/base/view/Airport.jsp`.

```
roma jsp org.test.domain.*
```

As above, but it generates a JSP for each class found in the package `org.test.domain`.

8 CRUD

CRUD stands for **Create Read Update Delete** and represents the very common operations you can execute to entities. Many pieces of applications are composed by CRUD forms where you can handle entities.

Roma Framework supports CRUD natively, allowing fast development of CRUD against domain classes.

To explain the CRUD creation and customization we are going to use a bank application domain.

Class Person.java

```
package org.romaframework.test.presentation.domain;

import java.util.Date;

public class Person {
    protected String name;
    protected Date birthDate;

    public Person() {
    }

    public Person(String name, Date birthDate) {
        this.name = name;
        this.birthDate = birthDate;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public boolean equals(Object o){
        if (o instanceof Person){
            Person person = (Person)o;
            boolean equals = name.equals(person.getName());
            equals = equals && created.equals(person.getBirthDate());
            return equals;
        }
        return false;
    }
}
```

```

    public int hashCode(){
        return name.hashCode() + birthDate.hashCode();
    }
}

```

Class BankAccount.java

```

package org.romaframework.test.presentation.domain;

import java.util.Date;
import java.util.List;

public class BankAccount {

    protected Person owner;
    protected List<BankTransaction> transactions;
    protected Integer currentAmount;
    protected Date created;

    public Integer getCurrentAmount() {
        return currentAmount;
    }

    public void setCurrentAmount(Integer currentAmount) {
        this.currentAmount = currentAmount;
    }

    public Date getCreated() {
        return created;
    }

    public void setCreated(Date opened) {
        this.created = opened;
    }

    public Person getOwner() {
        return owner;
    }

    public void setOwner(Person owner) {
        this.owner = owner;
    }

    public List<BankTransaction> getTransactions() {
        return transactions;
    }

    public void setTransactions(List<BankTransaction> transactions) {
        this.transactions = transactions;
    }

    public boolean equals(Object o){
        if (o instanceof BankAccount){
            BankAccount account= (BankAccount)o;
            boolean equals = owner.equals(account.getOwner());
            equals = equals && created.equals(account.getCreated());
            return equals;
        }
    }
}

```

```
    retrurn false;
}

public int hashCode(){
    return owner.hashCode + created.hashCode();
}
}
```

Class BankTransaction.java

```
package org.romaframework.test.presentation.domain;

import java.util.Date;

import org.romaframework.module.admin.domain.Info;

public class BankTransaction {

    protected Date        when;
    protected Float        amount;
    protected String       cause;

    @ViewField(visible = false)
    protected BankAccount account;

    protected Info type;

    public Info getType() {
        return type;
    }

    public void setType(Info type) {
        this.type = type;
    }

    public BankAccount getAccount() {
        return account;
    }

    public void setAccount(BankAccount account) {
        this.account = account;
    }

    public Date getWhen() {
        return when;
    }

    public void setWhen(Date when) {
        this.when = when;
    }

    public Float getAmount() {
        return amount;
    }

    public void setAmount(Float amount) {
        this.amount = amount;
    }

    public String getCause() {
        return cause;
    }

    public void setCause(String cause) {
        this.cause = cause;
    }
}
```



```

public boolean equals(Object o){
    if (o instanceof BankTransaction){
        BankTransaction transaction = (BankTransaction)o;
        boolean equals = account.equals(transaction.getAccount());
        equals = equals && when.equals(transaction.getWhen());
        return equals;
    }
    return false;
}

public int hashCode(){
    return account.hashCode() + when.hashCode();
}
}

```



Best practice: Avoid to use primitives Java types for fields (int, float, etc..) because they can't be null. Even though Roma works well with literals, the CRUDs will search using the [QueryByExample](#). It uses the NULL values to exclude fields in the search. So if you need to generate CRUD against domain entity classes use the Object types such as Integer, Float, etc.



Best practice: Avoid to assign values in the object construction (constructors and/or field declaration). Roma creates an empty object as filter in the CRUD page by calling the empty constructor of domain entity class. If you initialize some fields at this point the CRUD filter will result not empty (as usual) but with the field already filled.

To auto-generate a CRUD for a domain class you can use the Roma console. Below an example of how to generate CRUD for BankAccount class.

```

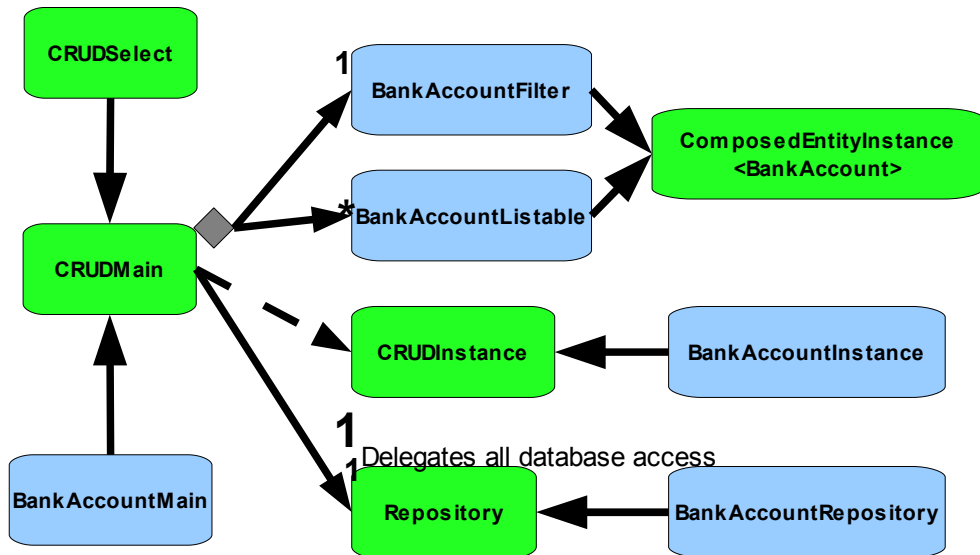
> ./roma.sh project crud
org.romaframework.test.presentation.domain.BankAccount

```

This will generate in your project the following Java source files.

BankAccountMain.java: Is the "main" class of the CRUD. Is composed by a filter of type BankAccountFilter and a result of type List. Executing the search the filter will be used to filter persistent instances and returning the result in "result" property (of type List).

BankAccountFilter.java: Is the class that handle the filter that read the search



method.

BankAccountInstance.java: Is the class that handle creation, updating and reading of selected instances.

BankAccountListable.java: Is showed inside the result table.

BankAccountSelect.java: As for BankAccountMain, but is showed in a popup and contains the additional actions: Select and Cancel.



The **BankAccountFilter**, **BankAccountInstance**, **BankAccountListable** are wrappers of the class **BankAccount** (based on the Composed Entity pattern), so if you add or remove a field from a domain object you **don't** need to generate the CRUD again.

To display the entry point of your brand new CRUD you should display a **BankAccountMain** instance, using the flow to call the BankAccount's CRUD-Main class.

```
@FlowAction(next = BankAccountMain.class)
public void showBankAccountManager() {}
```

This piece of code allow to choose the Posts to insert in the collection **BestPostsOfTheYear.posts**.

8.1 How CRUD works

By default the CRUD entry point form (**CRUDMain** class) is composed by:

- The **Filter**, an instance of the generated *BankAccountFilter* class.
- The **Result**, as a collection of instances of generated class **BankAccountListable** class.

The filter is for search purpose: every filled fields of the filter instance will acts as filter condition on the search query. Now all query conditions use the **AND** operator, but we're working on it. This is a simple implementation of **Query By Example pattern**.

For example if I fill the **currentAmmount** field with the value "100" and then click Search, the **CRUDMain** class will compose the query in JDOQL: **currentAmmount == '100'**.

You can express complex queries using [JDOQL](#) or [SQL](#) directly, but the Query By Example satisfy the most common needs: execute a query specifying filters on some fields. Using the Query By Example you don't have to write JDOQL/SQL code.

Result contains the query result filled from the execution of the method **CRUDMain.search()** method invoked by clicking the "Search" button.

BlogFilter, as the other classes generated by the wizard, extends the class **ComposedInstance<BankAccount>**; Roma handles the *BankAccountFilter* as a class that extends *Blog*, but using the **Composition pattern**.

Why this? Because in this way **BankAccount** objects can be retrieved and stored in the database without conversion. If *BankAccountFilter* inherits *BankAccount* directly, the persistence layer recognize *BankAccountFilter* class as **Non Persistent**.

Instead Roma handles everything transparently by retrieving and storing the instance contained inside the class extends **ComposedInstance**.

8.1.1 CRUD Working modes

Each `CRUDInstance` derived subclass contains the protected field “**mode**” that tells to the `CRUDInstance` how to work. The working mode supported are:

Mode	Value	Description	Callback
Embedded	0	This is the default one: The <code>CRUDInstance</code> class is used to display embedded objects. No save operations occurs since they should happen in the container object.	
Create	1	Used when the object is created. Called by <code>CRUDMain.create()</code> . The save operation will create the new entity object into the repository.	On displaying the callback <code>onCreate()</code> is called. Override it to handle your own initialization.
Read	2	Used when the object is read. Called by <code>CRUDMain.read()</code> . By default all fields are disabled. No save operation will be displayed.	On displaying the callback <code>onRead()</code> is called. Override it to handle your own initialization.
Update	3	Used when the object is created. Called by <code>CRUDMain.update()</code> . The save operation will updated the entity object to the repository.	On displaying the callback <code>onUpdate()</code> is called. Override it to handle your own initialization.
Custom	9	This is a special mode to use the <code>CRUDInstance</code> implementation with a custom behaviour different by the previous ones.	



Working modes are defined as constants in the interface:

`org.romaframework.frontend.domain.crud.CRUDWorkingMode`

To intercepts all the working modes your `CRUDInstance` subclass should override all the callbacks in this way:

[Class `BankAccountInstance.java`](#)

```

public class BankAccountInstance extends CRUDInstance<BankAccount> {
    @Override
    public void onCreate() {
    }

    @Override
    public void onRead() {
    }

    @Override
    public void onUpdate() {
    }
}

```

8.2 Customize a CRUD

8.2.1 Change the GUI

From the initial generation of the **BankAccount** CRUD we get this result:

First of all we want to remove the transactions in the filter, so we open the **BankAccountFilter.xml** and we modify it obtaining:

```

<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
        xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
    <fields>
        <field name="entity">
            <aspects>
                <view layout="expand" />
            </aspects>
            <class>
                <fields>
                    <field name="transactions">
                        <aspects>
                            <view visible="false" />
                        </aspects>
                    </field>
                </fields>
            </class>
        </field>
    </fields>
</class>

```

```

        </aspects>
      </field>
    </fields>
  </class>
</field>
</fields>
</class>

```

Then we modify the BankAccountListable.xml to set change the fields order and hide the transactions column:

```

<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <fields>
    <field name="entity">
      <aspects>
        <view layout="expand" />
      </aspects>
      <class>
        <aspects>
          <view explicitElements="true"></view>
        </aspects>
        <fields>
          <field name="owner"></field>
          <field name="created"></field>
          <field name="currentAmount"></field>
        </fields>
      </class>
    </field>
  </fields>
</class>

```

8.2.2 Adding other search conditions

In our example we have in the filter the field currentAmount, and the basic crud uses it to generate the condition `currentAmount == '$inserted_value'`; this kind of search isn't useful for the bankAccounts object, it would be better to have the a condition like `currentAmount >= '$inserted_value'`.

So we modify the BankAccountFilter adding two properties, the amountLessThan and amountMoreThan:

```

package org.romaframework.test.presentation.view.domain.bankaccount;

import org.romaframework.aspect.view.annotation.ViewClass;
import org.romaframework.core.entity.ComposedEntityInstance;
import org.romaframework.test.presentation.domain.BankAccount;

@ViewClass(entity = BankAccount.class, label = "")
public class BankAccountFilter extends ComposedEntityInstance<BankAccount> {
    private Float amountMoreThan;
    private Float amountLessThan;
}

```

```

public BankAccountFilter() {
    this(new BankAccount());
}

public BankAccountFilter(BankAccount iBankAccount) {
    super(iBankAccount);
}

public Float getAmountMoreThan() {
    return this.amountMoreThan;
}

public void setAmountMoreThan(Float amountMoreThan) {
    this.amountMoreThan = amountMoreThan;
}

public Float getAmountLessThan() {
    return amountLessThan;
}

public void setAmountLessThan(Float amountLessThan) {
    this.amountLessThan = amountLessThan;
}
}

```

We hide the currentAmount that is no more used as filter condition changing the XML and finally, we override the search method of the BankAccountMain CRUD:

```

...
<field name="currentAmount">
    <aspects>
        <view visible="false" />
    </aspects>
</field>
...

```

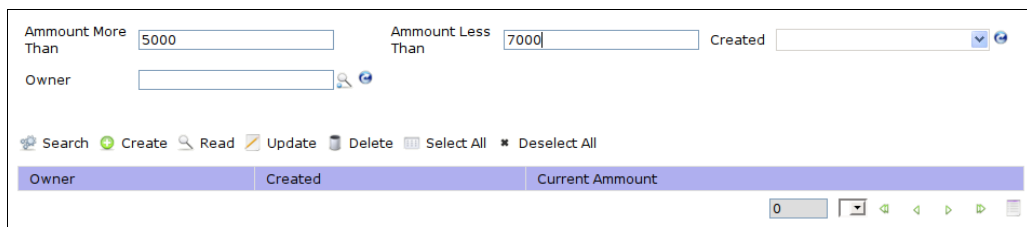
```

@Override
public void search() {
    QueryByFilter query = new QueryByFilter(BankAccount.class);

    BankAccountFilter filter = getFilter();
    if (filter.getAmountMoreThan() != null) {
        query.addItem("currentAmount", QueryByFilter.FIELD_MAJOR_EQUALS,
            filter.getAmountMoreThan());
    }
    if (filter.getAmountLessThan() != null) {
        query.addItem("currentAmount",
            QueryByFilter.FIELD_MINOR_EQUALS,
            filter.getAmountLessThan());
    }
    super.searchByExample(query);
}

```

After this changes to the code we get this result:



Amount More Than: 5000 Amount Less Than: 7000 Created: [dropdown]

Owner: [input]

Search Create Read Update Delete Select All Deselect All

Owner	Created	Current Ammount
		0

8.2.3 Order the result

Than if you want to order the result you can add to the previous code this line of code:

```
query.addOrder( "created", QueryByFilter.ORDER_ASC );
```

So the result will be ordered by the creation date.

8.2.4 Adding pre and post operations events

Sometimes we need to perform a set of operations when we are going to create, update, view or save the object, we have all we need to intercept this events in the BankAccountInstance class:

```
@Override
public void onCreate() {
}

@Override
public void onRead() {
}

@Override
public void onUpdate() {
}

public void validate() throws ValidationException {
}

public void save() {
    ...
    super.save();
}
```

As you can see we also have a method validate, that should be used only if the normal validation through annotations cannot reach the target because we need a really complex validation condition.

First of all we will add the date field to the BankAccountInstance:

```
protected Date lastUpdate;

public Date getLastUpdate() {
    return lastUpdate;
}

public void setLastUpdate(Date lastUpdate) {
    this.lastUpdate = lastUpdate;
}
```



Reminder: we don't need to generate the crud again after this change.

Then we change the BankAccountInstance as follows:

```
private static final String VALIDATION_ERROR_MESSAGE = "Last update date cannot be before creation date";

@Override
public void onCreate() {
    entity.setCreated(new Date());
}

@Override
public void onRead() {
}

@Override
public void onUpdate() {
    entity.setLastUpdate(new Date());
}

public void validate() throws ValidationException {
    if (getMode() == CRUDInstance.MODE_CREATE) return;
    if (entity.getCreated().compareTo(entity.getLastUpdate()) > 0) {
        throw new ValidationException(this, "lastUpdate",
            VALIDATION_ERROR_MESSAGE, null);
    }
}

public void save() {
    if (entity.getTransactions() != null) {
        for (BankTransaction transaction : entity.getTransactions()) {
            transaction.setAccount(entity);
        }
    }
    super.save();
}
```

As you can see when we create or update an object we set the creation and update date, during the save we test for the update date major than the creation and we set on all the transactions added the currentBankAccount.

Finally we change the **BankAccountInstance.xml** to render this fields as label, so the users cannot change them:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
      xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <fields>
    <field name="entity">
      <class>
        <aspects>
          <view explicitElements="true" />
        </aspects>
        <fields>
          <field name="owner"></field>
          <field name="created">
            <aspects>
              <view render="label" />
            </aspects>
          </field>
          <field name="lastUpdate">
            <aspects>
              <view render="label" />
            </aspects>
          </field>
          <field name="currentAmmount">
            </field>
          <field name="transactions">
            <aspects>
              <view render="table" />
            </aspects>
          </field>
        </fields>
      </class>
    </field>
  </fields>
</class>
```

8.2.5 How to get different CRUDs

Sometime you may need to define different GUI for the management of Object, so what happen if you need to CRUD of the same object type but using different instance, “listable” or filter class? Or how can I have on the same crud a different instance class for creation and modification?



Best practice: Don't create a new CRUD for each extension but extend existent CRUD in order to inherit CRUD customizations.

In this case you all you need to do is to define your own object and override the CRUDMain constructor, so we added a constructor to BankAccountMain:

```
public BankAccountMain(Class<? extends ComposedEntity<?>> listClass, Class<?>
createClass, Class<?> readClass, Class<?> editClass) {
    super(listClass, createClass,
        BankAccountInstanceForVisualization.class, editClass);
}
```

As you can this constructor accept the class to be used as “listable”, for creation, read and update, so you can replace, in the example above we changed the visualization class.

8.2.6 CRUDHelper

If you need advanced controls and manage of a CRUD there is the CRUDHelper Class, there are three usefull method to show and bind a crud with a POJO:

```
public static Bindable show(Bindable iSelectObj, Object iSourceObject, String
iSourceFieldName){}

public static <T extends Bindable> T show(Class<T> iClass, Object
iSourceObject, String iSourceFieldName, Object iCaller) {}

public static <T extends Bindable> T show(Class<T> iClass, Object
iSourceObject, String iSourceFieldName) {}
```

You can use this methods to bind a field to a CRUDSelect object in this way:

```
public class BankAccountMigration {
    private ArrayList<BankAccount> accounts;

    public void selectAAccountForMigration() {
        CRUDHelper.show(BankAccountSelect.class, this, "accounts");
    }
}
```

or if you need a to initialize a CRUDSelect with some value on the filter you can use the other version of the method:

```
public class BankAccountMigration {
    private ArrayList<BankAccount> accounts;

    public void selectAAccountForMigration() {
        BankAccountSelect select = new BankAccountSelect();
        select.getFilter().setAmmountMoreThan(300000);
        CRUDHelper.show(select, this, "accounts");
    }
}
```

9 Conventions (over configuration)

"Convention over configuration" is the a new approach brought by Ruby On Rails framework that preach the knowledge of conventions over writing a lot of configuration files. Roma uses some conventions to get the application configuration short, but anything is modifiable by changing the configuration files.

This is the reason because is fundamental to know the Roma basic conventions: learn them and the development process will speed up!

9.1 Extension by Composition

The best way to extend a domain class is not to inherit it, but using the **Composite Pattern**. Roma Framework provide the interface **ComposedEntity< T >** and its extensions as:

- **EntityPage< T >**
- **CRUDInstance< T >**

Below a common use case: extend domain **Profile** class adding the **confirmPassword** property useful to make a double check on password input. **confirmPassword** must not reside in the domain class since is concerned only to the [View Aspect](#)!

Example:

```
public class Profile {  
    private String name;  
  
    @ViewField( render = "password" )  
    private String password;  
}
```

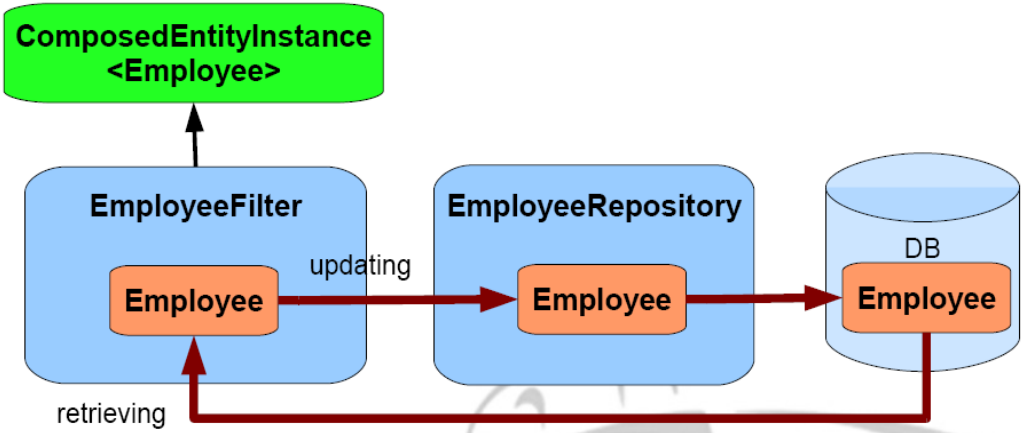
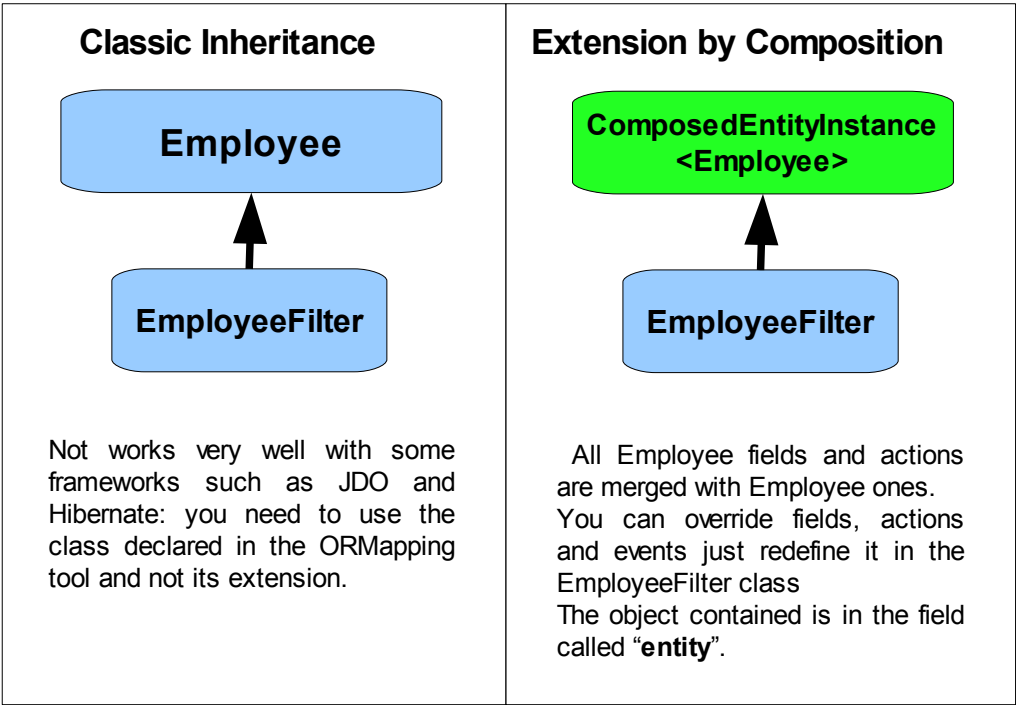
This was the base entity class and below the extension using the composition:

```
@ViewClass( entity = Profile.class )  
public class ProfileInstance extends ComposedInstance<Profile> implements  
CustomValidation {  
  
    private String confirmPassword;  
  
    public void validate() {  
        if( !confirmPassword.equals( entity.getPassword() ) )  
            throw new ValidationException( this, "confirmPassword", null );  
    }  
}
```

Roma will treat the **ProfileInstance** class as a class that extends the **Profile** class.

9.1.1 Advantages

- The entity object is passed among layers using the **ValueObject** pattern
- No cloning



9.2 Override default behaviour using the events

The Roma approach "**Convention over Configuration**" allows to save much time on development. However often you need to override the default behaviour. That's why you can write some **callbacks** or intercept and override an **events**. Event definition

All events in Roma must have the following syntax:

```
public void on[<Field>][<Operation>]() {  
    ...  
}
```

Where:

- **Field** (optional) is the name of the field to catch the event. If the field is not present, the event is assigned to the class.
- **Operation** (optional) wants to specify the associate operation to catch. Often you'll need it to override the collection behaviour. Take a look to the examples forward.

9.2.1 Predefined Events

9.2.1.1 Callback onShow()

You can write a callback called by Roma before an object is displayed. This is specially useful for objects reused more times, since the constructor is not more invoked after the first time.

To write the callback just implement the **ViewCallback** interface and the **onShow()** method. Example:

```
public class CarForm implements ViewCallback {  
    public void onShow(){  
        // SHOW A MODAL POPUP TO CONTINUE  
        Roma.flow().forward( new MessageOk( "message", "Warning!", null,  
                                              "Click OK to continue" ) );  
    }  
}
```

9.2.1.2 Callback onDispose()

You can write a callback called by Roma when an object is not more visible. This is specially useful to free resources you don't need any more.

To write the callback just implement the **ViewCallback** interface and the **onDispose()** method. Example:

```
public class CarForm implements ViewCallback {  
    public void onDispose(){  
        socket.close();  
    }  
}
```

```
}
```

9.2.2 Override Collection behaviour (List and Tables)

Roma renders **Collection** fields as **Lists** or **Tables** provided by **View**, **Add**, **Remove** and **Update** buttons to work with elements contained inside of it.

If you want to override the default management you had to write an **event** with the operation to override. Example:

```
public class CarForm {
    private Set<Person> owners;

    public void onOwnersAdd(){
        // SHOW THE CRUD SELECT INSTEAD TO CREATE A NEW INSTANCE
        CRUDHelper.show( PersonSelect.class, this, "owners" );
    }
}
```

The code above shows the Selection CRUD of Person instances and binds the user selection (if any) to current class (**this**), in the field **owners**.

For any **Collections** you can override the default behaviour by defining the supported events:

- on<Field>View()
- on<Field>Add()
- on<Field>Remove()
- on<Field>Edit()

10 Best Practices

10.1 Separate domain classes from presentation classes

It's a really good practice to divide classes in own aspect-package. A common use case is to place the **REAL** domain class in **<application-package>.domain** and all the presentation classes in **<application-package>.view.domain**.

<application-package>.domain should contain only the basic domain classes that are common to the application and all the aspects.

<application-package>.domain.entity should contain the persistent classes stored in the database.

Example:

```
package it.demo.domain.entity;

public class Profile {

    private String name;

    @ViewField( visible = AnnotationConstants.FALSE )
    private City city;
}
```

```
package it.demo.view.domain.profile;

@ViewClass( entity=Profile.class )
public class ProfileInstance extends ComposedInstance<Profile> implements
ViewCallback {
    // OMITTED GETTER AND SETTER, IN YOUR APP WRITE THEIR!
    @ViewField( selectField = "entity.city", render = "select" )
    private City[] cities;

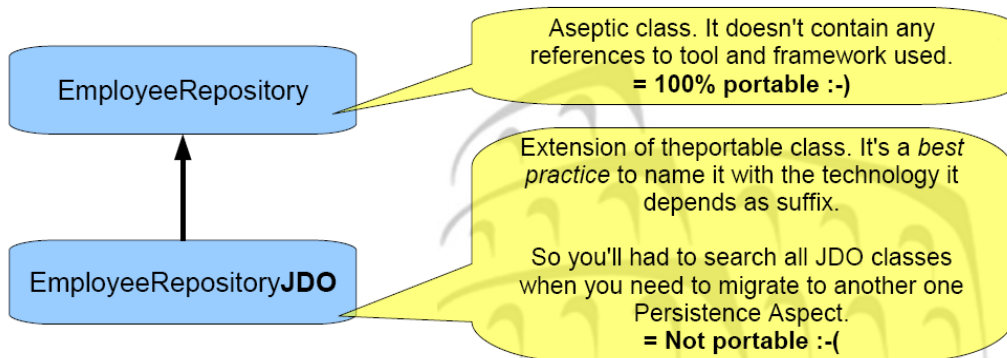
    public ProfileInstance(){
        cities = Roma.context().persistence().query( new
QueryByExample( City.class, null ) );
    }
}
```

ProfileInstance extends the **Profile** class by [Containing Mode](#). It shows a combo box component with all cities to select. Selection will be bound to real city property (contained in **entity** property of **ProfileInstance**, see **ComposedInstance.java**).

In this way the presentation concerns are separated from others and reside in its package.

10.2 Dirty approach: dirty your hands when required

Roma wants to cover the **most common use cases**. It's utopian to imagine covering all user requirements since they are so many and frameworks can be so much different between them.



So if you need to access directly to the tool and framework in the background you had to know that you can do it. Just remember that piece of code will be not portable across implementation when, and if, you'll decide to migrate to another one.

11 Virtual Objects

Roma Framework is strictly based on the POJO Java architecture. Anything is a POJO from the view to the persistence. But in some circumstances it could be better having a lighter way to hide some fields in a form or adding a new action to an existent POJO.

This is the reason because Roma (starting from version 2.0.3) supports the Virtual Objects. Virtual Objects are treated just real Java Objects but they are declared using a descriptor. Now only XML descriptor is supported but in future it could be planned JSON or other ways.

Virtual Objects, in effects, are declared using the same XML Annotation syntax with minor additions to handle field types, inheritance and other stuff that Roma can't knows since the XML descriptor is not backed by a real class file.

Cross language

Virtual object's business logic is written inside the <action> tag using the [Scripting Aspect](#). This means that Virtual Objects are totally decoupled by the Java technology and they can be written in any language supported by the [JSR 223](#).

Hot changes high productivity

Virtual Objects are loaded at run-time and reloaded every time the file is saved. This boost up the productivity since it never requires to restart the JVM!

Indeed changing the signature of a class (add and remove attributes and methods) require always the JVM to restart. This is the reason why the Virtual Objects are strongly suggested to being used to build forms to display.

Web IDE

It's planned for the end of 2009 a powerful graphic editor on the Web to change the virtual objects directly from the running application without the need of an IDE.

Not Just Forms

Even though Virtual Objects makes the difference used in the [View Aspect](#), they can be improve the productivity in any Aspect of Roma. Think for example to a service written entirely in Javascript just creating the XML descriptor and annotating it with the [Service Aspect](#).

Below an example of a Virtual Object that extends the Roma's CustomLogin standard POJO used to make the login. Note that by default the language setted is the [Scripting Aspect](#) is JavaScript but you can use any language installed using the [JSR 223 implementations](#).

MyLogin.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd" extends="CustomLogin">
  <aspects>
    <view>
      <form>
        <area name="main" type="grid" size="1">
          <area name="help" />
          <area name="fields" type="grid" size="2" />
          <area name="actions" type="row" />
        </area>
      </form>
    </view>
    <scripting>me.field("help", "Please insert your account." )</scripting>
  </aspects>
  <fields>
    <field name="help" type="String">
      <aspects>
        <view render="html" />
      </aspects>
    </field>
    <field name="additionalCode" type="String">
      <aspects>
        <validation required="true" min="5" max="5" />
      </aspects>
    </field>
  </fields>
  <actions>
    <action name="login">
      <aspects>
        <scripting>
          if( me.field("additionalCode") == "12345" ) me.parent.login();
        </scripting>
      </aspects>
    </action>
    <action name="close">
      <aspects>
        <scripting>
          org.romaframework.frontend.RomaFrontend.view().close(me.pojo)
        </scripting>
      </aspects>
    </action>
    <action name="standardLogin">
      <aspects>
        <scripting>
          print('Goto standard login...')
          org.romaframework.frontend.RomaFrontend.flow().forward("ProjectLogin")
        </scripting>
      </aspects>
    </action>
  </actions>
</class>
```

```
</action>
</actions>
</class>
```

In this example the Virtual Object called “MyLogin” makes the following changes to the class CustomLogin that extends:

- Add a new field called “help” defined as a String and displayed as HTML on top of all fields (see area definition)
- Add a new field called “additionalCode” defined as a String. This field define some validation rules: it's mandatory and the length must be of 5 characters
- In the constructor (see the scripting declaration inside the class) the field help is initialized with a welcome message.
- The login action is overridden to check if the virtual field “additionalCode” is equals to “12345”. Only in this case calls the super.login(). Not the use of “me.parent” that returns the super class.
- Add a new action called “close” to close the form. Note the use of “me.pojo” to return the real POJO in Roma.
- Add a new action called “Standard Login” that show the classic Login form.

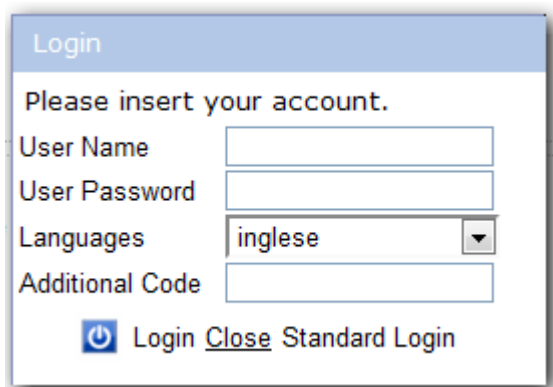
You can use a Virtual Object such as normal POJO in Roma. To display the Virtual Object just created write:

```
public void test() {
    RomaFrontend.flow().forward(new VObject("MyLogin"), "screen:popup");
}
```

or just:

```
public void test() {
    RomaFrontend.flow().forward("MyLogin");
}
```

This is what is displayed:

A screenshot of a 'Login' dialog box. The title bar is light blue and contains the word 'Login'. The main area is white and contains the text 'Please insert your account.' followed by four input fields: 'User Name', 'User Password', 'Languages' (a dropdown menu showing 'inglese'), and 'Additional Code'. At the bottom, there is a blue power button icon, the text 'Login', a blue underlined 'Close' button, and the text 'Standard Login'.

Login


Please insert your account.

User Name

User Password

Languages ▼

Additional Code

 Login Close Standard Login

12 Controller

Roma binds the POJO directly to the form generated. The relationship is 1-1: if you change the content of a field in the browser, it will be updated in you Java object and vice versa. When you push any button, the action associated will be executed.

Fast to write, easy to debug, simple to handle!

default:GRID(2)			
Name	<input type="text" value="Luca"/>	Surname	<input type="text" value="Garulli"/>
City	<input type="text" value="Luca"/>	Web	<input type="text" value="zioncity"/>

notes:CELL
Notes <input type="text" value="Zion's citizen"/>

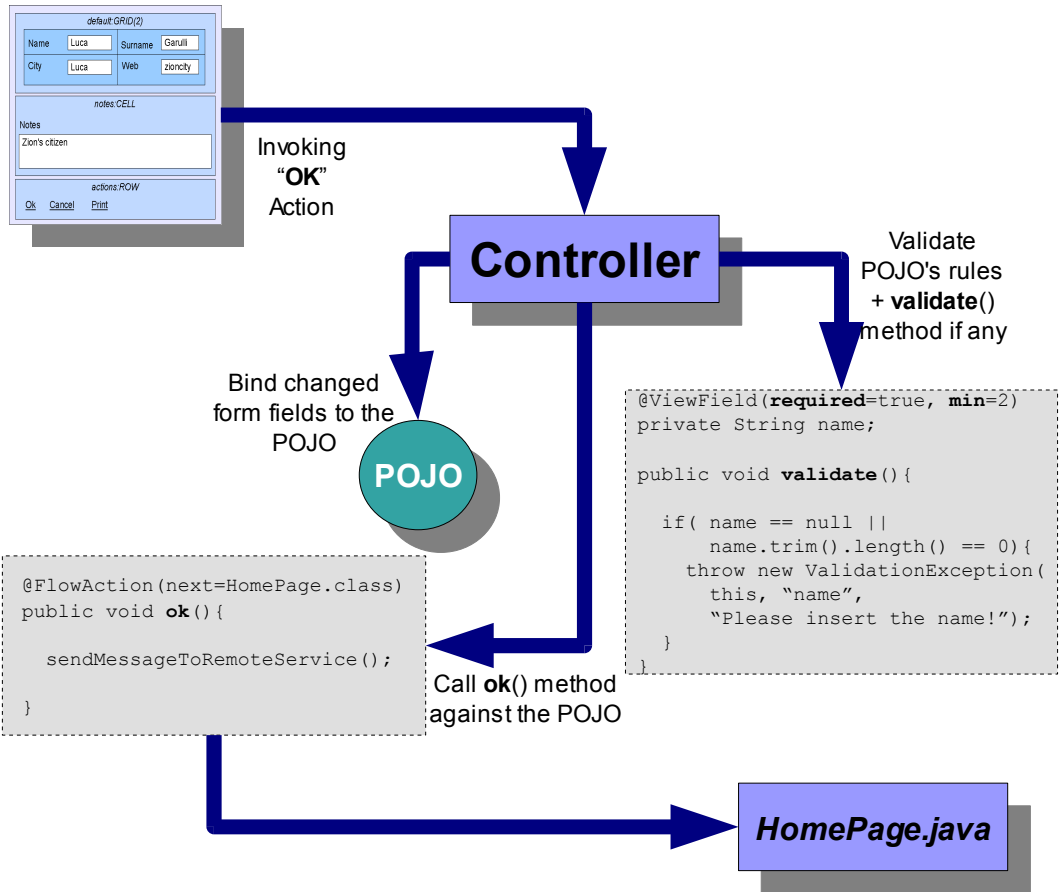
actions:ROW
<input type="button" value="Ok"/> <input type="button" value="Cancel"/> <input type="button" value="Print"/>

```
public class Customer{  
  
    private String name;  
    private String  
surname;  
    private String city;  
    private String web;  
  
    private String notes;  
  
    public void ok(){}  
    public void cancel(){}  
    public void print(){}  
}
```

12.1 Controller in action

12.2 Transparent binding of field values

Roma controller acts between Forms and POJOs. Every time the user change some



field values Roma binds changes to the POJO. This usually happens before an actions is called.

You can avoid to make automatic binding by setting the “bind” annotation to false (use Java/XML annotation):

```
@ViewAction(bind=AnnotationConstants.FALSE)
public void reload(){
    ...
}
```

If you change some field values of your POJO you had to tell it to Roma in order to refresh the updated values on the form when the Controller receives the control

again. Use this:

```
public void reload(){
    name = "";
    ObjectContext.getInstance().fieldChanged(this, "name");
}
```

13 Flow Aspect

The Flow Aspect worries about the application flow. When you pass from a form to another one, the Flow Aspect is involved.



Before version 1.1 (or SVN version 3593) the Flow Aspect is uniquely responsible for the application flow, while previously you could bypass it using directly the [View Aspect](#). Now calls as `ViewAspect.show()` and `ViewAspect.close()` are strongly discouraged in favour of `FlowAspect.forward()` and `FlowAspect.back()`.

13.1 Display a POJO

You can display a POJO using annotations or via code. When possible use annotations in order to change the application flow without change application code.

13.1.1 Via Annotations

13.1.1.1 Using XML Annotation

CompanyInstance.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <actions>
    <action name="save">
      <aspects>
        <flow next="EmployeeInstance" position="form://body" />
      </aspects>
    </action>
  </actions>
</class>
```

13.1.1.2 Using Java Annotation

CompanyInstance.java

```
public class CompanyInstance {
    ...
    @FlowAction( next=EmployeeInstance.class, position="form://body")
    public void save() {
        ...
    }
}
```

```
...  
}
```

13.1.2 Via Code

```
public void save() {  
    EmployeeInstance form = new EmployeeInstance();  
    Roma.component(FlowAspect.class).forward( form, "form://body" );  
}
```

13.2 Go back

To go back to the previous form just use `FlowAspect.back()` method:

```
public void cancel() {  
    Roma.component(FlowAspect.class).back();  
}
```

13.3 Access to the user history

To access to the user's history use the `getHistory()` methods.

13.4 Action Confirm

Sometimes clicking a button is a critical action, and you may need to ask the user to confirm before proceeding. Roma [Flow Aspect](#) provides a declarative way to implement such a behaviour. You just have to annotate your action with `@FlowAction`, declaring that it requires confirmation:

```
class MyClass{  
  
    @FlowAction( confirmRequired = AnnotationConstants.TRUE )  
    public void delete() {  
        //some critical code...  
    }  
    ...  
}
```

And add to `i18n` properties files an entry specifying the message that will appear in the confirm popup:

```
MyClass.delete.confirmMessage      Are you sure you want to delete this object?
```

The result of this is that Roma will display a message popup with “yes” and “no” keys, and the action will be executed only if the user clicks “yes”.

14 View Aspect

View aspect has the difficult mission to provide a GUI (Graphical User Interface) to the application. Roma can be used to build any kind of application. However most of users use Roma as a RAD to build Web Applications.

View aspect mainly aims to render POJOs automatically, without the need to explicitly define GUIs. Obviously you can override default behaviour by enhancing the class model using [Java Annotations](#) or [XML Annotations](#).

14.1 How Roma renders POJOs

Roma can render each POJO automagically. How it works? It's very easy:

- Each **fields** (or Java properties) are rendered basis on their nature. For example Date properties as Calendar, String as Text Fields, etc.
- Each **actions** (or Java public-void methods) are rendered as links.

You have 1-1 relationship between your POJO and your GUI.

Roma uses the concept of [Area](#) to place graphical components in the right place. By default each form is divided in two areas: "**fields**" and "**actions**". If exist custom areas are displayed after the default one, but you can change the order.

main:COLUMN			
<i>default:GRID(4)</i>			
Name	<input type="text" value="Luca"/>	Surname	<input type="text" value="Garulli"/>
City	<input type="text" value="Luca"/>	Web	<input type="text" value="zioncity"/>
<i>notes:COLUMN</i>			
Notes			
<input type="text" value="Zion's citizen"/>			
<i>actions:ROW</i>			
<u>O</u> k	<u>C</u> ancel	<u>P</u> rint	

14.2 Area concept

To have a form layout really portable across technologies and frameworks we can't use HTML to define the layout neither the Swing or the SWT way. We need a new generic way to tell to Roma the aspect our form should be have.

Areas aim to divide your form in multiple parts. Each one must describe the way it place the component: the “**area rendering mode**”. By default is the **placeholder**, namely an area type that replace itself with the component rendered (and only one). You can use all the modes between the supported ones:

Area Rendering Mode	Components min-max	Behaviour	Description
Placeholder	1	Replace the current component	The Default. Similar to cell, but on creation is empty. <pre><area name="foo" /></pre>
Cell	1	Replace the current component	As the Placeholder, but a empty component is created. <pre><area name="top_dx" type="cell" /></pre>
Column	1-N	Add the component to the bottom	Place the components vertically as in a column. <pre><area name="main" type="column" /></pre>
Row	1-N	Add the component to the right	Place the components horizontally as in a row. <pre><area name="actions" type="row" /></pre>
Grid	1-N	Add the component to the right until column size is reached, then in a new row	Place the components in a grid. The number of columns are taken by the "size" attribute. <pre><area name="fields" type="grid" size="4" /></pre>

Areas only describe the layout form. To handle the presentation aspect much deeper

you need to work with the [View Aspect](#) implementation in the behind. If you're using [View-Echo2](#) module you had to work with the XML stylesheet files contained under the package `<your-apppackage>.view.style`.

Roma team chosen to not generalize features like component width, heights, colours, etc. since they are so many and so much technology-related that it's much simpler to work directly with the [View Aspect](#) implementation way it offers.



*Note: Remember that your application will be portable across technologies and frameworks. It can run as **Web Application** or as **Swing Desktop** application. But GUI details will be lost and need to be translated in the technology you want to use!*

14.3 Create your form layout

Starting from release 1.0rc3 Roma allows to specify custom form layout in easy way by [Xml Annotations](#).

To define the form layout write the areas in a file called `<Class>.xml` where **Class** is the class name you want change the default layout. When Roma try to discover the layout to use, it follows the class inheritance. So you can create form layouts in super-classes and all extensions will use it.

Under the package domain of your application you can find the **Object.xml** file. Since **Object** is the base class of all classes in Java, it will be used if sub-classes doesn't declare a layout form. By default **Object.xml** contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <aspects>
    <view>
      <form>
        <area name="main" type="column">
          <area name="title" />
          <area name="fields" type="grid" size="2" />
          <area name="actions" type="row" />
        </area>
      </form>
    </view>
  </aspects>
</class>
```

This is the explanation:

- main: sub-areas will be placed in **vertical** ("column" mode)
- fields: **2 columns**: label + field
- actions: will be placed in horizontal ("row" mode)

14.3.1 Fields area

Is the default [area](#) where the fields of a POJO are placed.

Object.xml contains the definition of the "fields" [area](#) as a grid component. You can specify how many columns is composed the grid to optimize the layout.

For each field are always rendered two graphical objects: the label and the field content. If you define an empty label it will be not displayed and only the field content will.

The field order is specified by [Xml Annotation](#) or in the [Java Annotation](#) called **orderFields**:

```
@CoreClass( orderFields = "name surname city" )
```

14.3.2 Actions area

Is the default [area](#) where the actions of a POJO are placed.

The order followed is expressed in [Xml Annotation](#) or in the [Java Annotation](#) called **orderActions**:

```
@ViewClass( orderActions = "ok cancel print" )
```

14.4 Screen and Area concepts

Roma renders POJO inside a screen. A screen is a set of [areas](#). [Areas](#) inside screens are the same of Form [areas](#). In this way you can compose very complex GUI playing with screens and form layouts.

By convention POJO are rendered always in the last screen [area](#) used, or "body" if null.

14.5 Configurable Screens

Starting from v. 0.9.9 Roma allows to use screens configured via XML. First of all assure your **applicationContext-core.xml** file contains the following services:

```
<bean id="ScreenManager"
      class="org.romaframework.aspect.view.echo2.screen.Echo2ScreenManager"
      singleton="true"/>
```



```
<bean id="ScreenConfigurationLoader"
      class="org.romaframework.aspect.view.screen.config.ScreenConfigurationLoader"
      singleton="true"/>
```

By convention, place you screen files under **[you package].view.domain.screen**. However Roma will search it inside all configured domains packages.

14.5.1 Example

main-screen.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<screen xmlns="http://www.romaframework.org/xml/roma"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
        xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.orienttechnologies.com/roma/schema/roma-view-screen.xsd>
  <area name="mainPanel" type="column">
    <area name="header" />
    <area name="topmenu" />
    <area name="centralPanel" type="row">
      <area name="leftMenu" />
      <area name="body" />
    </area>
    <area name="log" />
    <area name="footer" />
  </area>
</screen>
```

To use a configurable XML screen, write (generally in HomePage class or when you want to change the screen):

```
Screen screen = Roma.component(ScreenManager.class).
    getScreen( "main-screen");

ObjectContext.getInstance().setScreen(screen);
```

14.6 Component placement in screens

Once you have defined the screen layout you can easily place Roma component by specifying it through **layout** feature: by **Java Annotation** or by **[XML Annotation](#)**:

Example using Java annotations:

```
@ViewClass( layout = "screen://body" )
public class HomePage{
  ...
  @ViewField( layout = "screen:centralPanel/leftMenu" )
```

```
protected MyMenu menu = new MyMenu();
...
}
```

In the example above we have used `screen://body` to express the body [area](#) wherever is it. Instead for the menu component we want to render it in `screen:centralPanel/leftMenu` telling the `leftMenu` [area](#) under `centralMenu` (XPath-like syntax) .

See the [screen layout mode](#) for more information.

14.7 RenderingResolver: how to select the component to render

Roma select automatically the right graphical component for each action and field reading the type. Obviously you can always overwrite default behaviour using [Annotations](#) in every single action/type you want. However sometimes can be useful change the default behaviour.

You can do it by changing the mapping between types and components in the bean called [RenderingResolver](#) placed in the file:

[src/META-INF/components/applicationContext-view-echo2.xml](#):

```
<bean id="RenderingResolver" singleton="true"
      class="org.romaframework.aspect.view.echo2...ConfigurableRenderingResolver">
  <property name="configuration">
    <map>
      <entry key="action"           value="link" />
      <entry key="object"          value="object" />

      <entry key="java.lang.String" value="text" />
      <entry key="java.lang.Integer" value="text" />
      <entry key="java.lang.Float"   value="text" />
      <entry key="java.lang.Double"  value="text" />
      <entry key="java.lang.Boolean" value="check" />
      <entry key="java.util.Date"    value="date" />
      <entry key="java.util.Collection" value="list" />
      <entry key="java.util.Set"      value="list" />
      <entry key="java.util.Map"      value="table" />
    </map>
  </property>
</bean>
```

The Echo2 ViewAspect will search for the right component to place in this component. By convention the entry with key "action" is the component to use for actions; "object" for embedded objects.

14.8 Rendering modes

Rendering modes are the way Roma can render objects, fields and methods. By default the [View Aspect](#) delegates to the [RenderingResolver](#) component to resolve it. You can override the default behaviour by assigning the rendering you want by using the [annotation](#) (Java/XML).

In this section we are going to describe all the possible rendering modes currently supported by Roma [View Aspect](#). Some rendering modes can be applied to classes, fields, methods (actions) or some of the three.

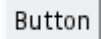
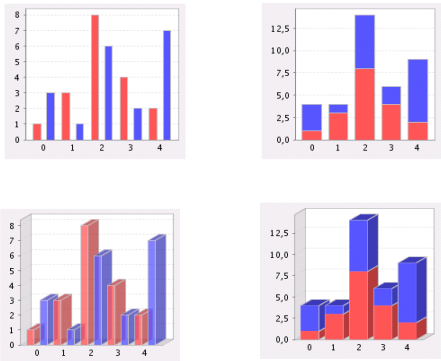

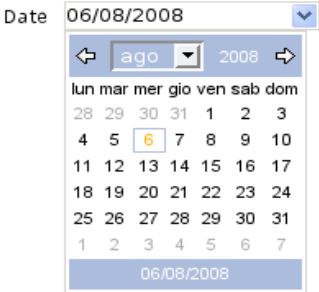

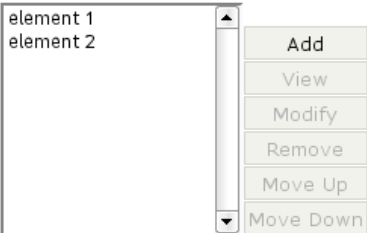
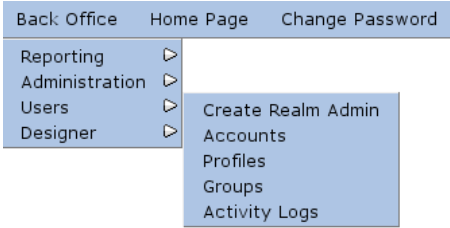
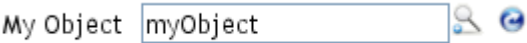
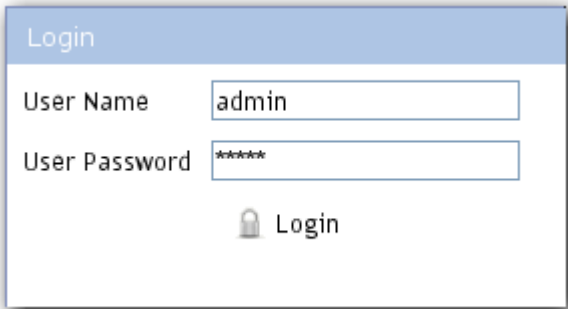

Rendering Mode	Applies to	Description
Default	Classes, Fields, Actions	<p>If no rendering mode is defined for a particular class, field or method, Roma will render it in a default way. Only public fields and methods can be represented by Roma in the user interface. Anyway, public fields can be explicitly hidden. The default behaviour for rendering fields depends on the field type: String, Integer, Float, Double, int, float, double fields are rendered in <i>Text</i> rendering mode. Boolean and boolean fields are rendered in <i>Check</i> rendering mode. Date fields are rendered in <i>Date</i> rendering mode. Collections are rendered in <i>List</i> rendering mode. Fields that don't match the previous requirements are rendered in <i>Objectlink</i> rendering mode.</p> <p>Methods that are not getters or setters, whose name doesn't start by "on", that don't take parameters, are rendered in <i>Button</i> rendering mode.</p>
Accordion	Fields	Accordion is a vertical two-level menu, where, selecting a first level element, its sub-elements list is expanded, and other sub-elements lists are shrinked.
Button	Actions	<p>This rendering mode shows a button having as label the method name or the declared label. In some implementations a button can be enriched with an image.</p> 

Chart	Fields	<p>Chart rendering mode is not implemented by default, but can be enabled adding a module that implements the <code>ChartRendering</code> mode (see <code>Error: Reference source not found</code>). The current implementation of the Charting module allows to render Collections and Maps as different types of charts, such as pie charts, line charts, bar charts etc.</p> 
Check	Fields (Boolean)	<p>Check is a simple check box (checked if the field value is <i>true</i>).</p> 
Colset	Fields (Collection)	<p>This rendering mode renders a collection as a set of columns. For each column is expanded an element of the collection.</p>
Date	Fields (Date)	<p>Date rendering mode shows a complete interactive calendar</p> 

Datetime	Fields	<p>Date rendering mode shows a complete interactive calendar with time</p> 
Html	Fields	<p>When a field is rendered as Html, its value is inserted in the page without being interpreted. If its value contains HTML tags, these will be inserted in the page and interpreted by the browser.</p>
Image	Fields (String, java.net.URL)	<p>A field can be rendered as an image if its value is the file name of an image that is present in the package <code>[applicationPackage].view.image</code></p>
Imagebutton	Fields(String, java.net.URL)	<p>An ImageButton is a normal Image, but can be clicked. The method that is invoked when the user clicks on the image is the corresponding “on” method (e.g. if the field name is <i>myField</i>, the method that will be invoked, if it exists, is <i>onMyField()</i>).</p>
Label	Fields	<p>The field is rendered as a text label. The represented value is formatted string with the format indicated in the <i>format</i> attribute of <i>ViewField</i>, if this is not set use the <i>toString()</i> value of the object.</p> <div data-bbox="528 1258 1184 1382"> <pre>@ViewField(format="0.00", render=ViewConstants.RENDER_LABEL) private Long number = new Long(1);</pre> </div> <p style="text-align: center;">Number 1.00</p> <div data-bbox="528 1487 1184 1610"> <pre>@ViewField(format="MM/yyyy", render=ViewConstants.RENDER_LABEL) private Date date = new Date;</pre> </div> <p style="text-align: center;">Date 07/2009</p>

Link	Fields, Actions	The field is rendered as a clickable text label. If the rendering mode is applied to a Field, the method that is invoked when the user clicks on the link is the corresponding “on” method (e.g. if the field name is <i>myField</i> , the method that will be invoked, if it exists, is <i>onMyField()</i>).
List	Fields (Collection)	<p>Collections can be rendered as lists of elements. In the list is displayed, for each element, its <i>toString()</i> value. This rendering also provides buttons to add, remove, view, edit and move up/down elements (only for ordered Collections). The setter method relative to the field indicated as “selectionField” will be called at every selection. The getter, instead, will be called to know the initial value to pre-select in the component.</p> 
Menu	Classes, Fields	<p>This rendering mode allows to render objects as hierarchical menus. Fields in a menu can be rendered as sub-menus. Methods will be rendered as links.</p> 
Objectlink	Fields	<p>An object that is rendered as an ObjectLink is represented as a clickable element. The displayed value is the <i>toString()</i> value of the object; when clicking on the element, the corresponding object is expanded (by default as a popup).</p> 
Objectembedded	Fields	An object that is rendered as ObjectEmbedded is expanded inside the view of its parent object.

		<p>Password <input type="password" value="*****"/></p> <p>For collections types every single object inside the collection will be rendered as Objectembedded using the parent area defined. So if you want to distribute all the objects contained in the collection into a grid, specify a new area called as the field of type grid and use this rendering mode.</p>
Password	Fields	Password rendering mode is a simple text box that shows points or asterisks (the classic obfuscated characters) instead of the real text.
Popup	Classes	 <p>This rendering mode renders an object as a modal popup window.</p>
Progress	Fields (integers between 0 and 100)	This rendering mode renders an object as a progress bar. The minimum value of the bar is 0, the maximum value is 100. The value of the field should be inside this interval. NOTE: this rendering mode is not yet implemented.
Radio	Fields	<p>Collections and arrays can be represented as a group of radio buttons. The selected value is set on the field indicated in the "selectionField" attribute of "ViewField" annotation if declared, otherwise no selection will be allowed annotation. The setter method relative to the field indicated as "selectionField" will be called at every selection. The getter, instead, will be called to know the initial value to pre-select in the component.</p> <p><input checked="" type="radio"/> one <input type="radio"/> two <input type="radio"/> three</p>
Richtext	Fields (String)	This rendering mode displays a rich text editor.


Rowset	Fields (Collection)	This rendering mode renders a collection as a set of rows. For each row is expanded an element of the collection.
Select	Fields (Collection)	<p>Collections and arrays can be represented as a select box. The selected value is set on the field indicated in the "selectionField" attribute of "ViewField" annotation if declared, otherwise no selection will be allowed. The setter method relative to the field indicated as "selectionField" will be called at every selection. The getter, instead, will be called to know the initial value to pre-select in the component.</p> 
Tab	Fields (Map)	Maps can be represented as tabbed panes: each entry in the map represents a different tab, whose name is the entry key and whose content is the entry value. The selected value is set on the field indicated in the "selectionField" attribute of "ViewField" annotation if declared, otherwise no selection will be allowed. The setter method relative to the field indicated as "selectionField" will be called at every selection. The getter, instead, will be called to know the initial value to pre-select in the component.
Table	Fields (Collection)	Collections can be rendered as tables. Each row contains a value in the map; each column represents a property of the beans contained in the collection. The selected value is set on the field indicated in the "selectionField" attribute of "ViewField" annotation if declared, otherwise no selection will be allowed. The setter method relative to the field indicated as "selectionField" will be called at every selection. The getter, instead, will be called to know the initial value to pre-select in the component.
Tableedit	Fields (Collection)	Collections can be rendered as editable tables. Each row contains a value in the map; each column represents a property of the beans contained in the collection. Cells in the table are editable. The selected value is set on the field indicated in the "selectionField" attribute of "ViewField" annotation if declared, otherwise no selection will be allowed. The setter method relative to the field indicated as "selectionField" will be called at every selection. The getter, instead, will be called to know the initial value to pre-select in the component.
Text	Fields (String)	This rendering mode renders an object as a text field.

Textarea	Fields (String)	This rendering mode renders an object as a text area.
Time	Fields	This rendering mode renders a text field containing the hour in the format chosen in l18N configuration.
Tree	Fields (implements TreeNode)	This rendering mode renders a tree structure as an expandable tree.
Upload	Fields (Stream)	<p>This rendering mode allows to upload a file as a <code>org.romaframework.core.domain.type.Stream</code>.</p> <p>When the user upload the file, the event <code>on<Field></code> will be called.</p> <p>Example:</p> <pre> class UploadTest { @ViewField(render="upload") private Stream stream; public void onStream() { // READ THE UPLOADED FILE stream.load(); OutputStream out = new FileOutputStream("logo.png"); out.write(stream .getBuffer()); out.close(); } } </pre>
URL	Fields	<p>Is strictly derived by Html rendering mode but uses the field value as an URL to load the content at run-time. It's very useful to ensemble form with pieces of HTML files, JSP or any Web content.</p> <p>Example: http://www.romaframework.org/.</p> <p>Use <code>\${application}</code> as prefix in the value to replace it with the application context path.</p> <p>Example: <code>\${application}/static/header.htm</code>.</p> <p>Use <code>\${session}</code> to call the same application but maintaining the Http Session.</p> <p>Example: <code>\${session}http://localhost/test/dynamic/print.jsp</code>. \$</p>

		<p>Use both to share objects between your business POJOs and JSPs through the Http session.</p> <p>Example: <code>\${application}\${session}/dynamic/print.jsp</code>.</p>
--	--	--

14.9 Layout mode

Layout mode is the way a component will be placed. By default every single field is placed inside its own [area](#) if defined (just create one and assign the same name of the field), otherwise will go in the “fields” [area](#). Every single action will follow the same behaviour but if not found the own [area](#), Roma will place it in the “actions” [area](#). You can override this by using annotation and specifying a type between these:

Layout Mode	Applies to	Description
Default	Classes, Fields, Actions	<p>If no layout mode is defined for a particular class, field or method, Roma will use this as the default.</p> <p>For classes they will be placed in the last area used, otherwise in the “screen:/body” area.</p> <p>For fields they will be added to the “form://fields” area if any, otherwise will be created a new one.</p> <p>For actions they will be added to the “form://actions” area if any, otherwise will be created a new one</p>
Form	Classes, Fields, Actions	<p>Allow to place components inside areas of the current POJO. Use the XPath like syntax to express nested areas. Example: “form:/main/bottom”</p>
Block	Fields, Actions	<div>  <p>We strongly suggest to avoid block layout mode in favour of “form” rendering using areas.</p> </div> <p>A new block will be created. If you want to assign a name to the block use the syntax “block:foo” where “foo” is the name of the new block just created. Use the same block to group fields at run-time.</p> <p>Example:</p> <pre>@ViewField(layout = "block:areal")</pre>

		<p>If more than one field/action declare the same custom area (they have the same block name), then the fields/actions will be displayed in succession based on the order declared.</p> <p>Example using Java Annotations:</p> <pre> @CoreClass(orderFields = "name surname city web", orderActions = "ok cancel print") public class Customer{ private String name; private String surname; private String city; private String web; @ViewField(layout = "block:notesArea") private String notes; public void ok(){} public void cancel(){} public void print(){} } </pre>
Screen	Classes, Fields, Actions	<p>Allow to place components inside areas of the current screen. Use the XPath like syntax to express nested areas. Example: "screen:/main/bottom".</p> <p>See the Component placement in screens for more information.</p>
Expand	Fields	Expand all the fields of embedded object inside current fields
Popup	Classes	Layout objects as Popups

Example using [Java annotations](#):

```

public class Task{
    @ViewField( render = "richtext", layout = "form://bottom" )
    private String description;
}

```

14.9.1 XPath like syntax

Layout supports a XPath like syntax to express the node of the tree to use. Only a very subset of XPath language is used:

Syntax	Applies to	Description
--------	------------	-------------

/		Root node
//		The first node found searching recursively from root

If you want to display in the "body" area whatever is in the three use:

```
screen://body
```

In this way the layout will work also after refactoring in the area tree.

If you want to be sure to display the component in the area called "body" under the main node use:

```
screen:/main/body
```

Use the same syntax also in form layout mode.

14.10 Order the fields and actions

The Java language doesn't define a standard way to order properties and methods. You can write a Java class placing properties and methods in order, but this information is not written in the byte code once compiled (the .class file). For this reason the Reflection doesn't return properties and action following the order you wrote in Java source file.

For this reason we need an easy way to specify ordering of fields and action using the [Annotations](#).

Usually in Roma every [Java Annotation](#) finds the same declaration in the [XML Annotation](#). Not this time. Using the [Java Annotation](#) the definition is this.

Foo.java:

```
@CoreClass( orderFields="aa bb cc dd ee" orderActions="ok cancel" )
public class Foo{
    ...
}
```

And using the [XML Annotation](#) the order is taken from the declaration order.

Foo.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
```

```

    xsd:schemaLocation="http://www.romaframework.org/xml/roma
    http://www.romaframework.org/schema/v2/roma.xsd">
<fields>
  <field name="aa">
  <field name="bb" />
  <field name="cc" />
  <field name="dd" />
  <field name="ee" />
</fields>
<actions>
  <action name="ok"/>
  <action name="cancel"/>
</actions>
</class>

```

14.10.1 Ordering embedded instances

If you need to specify the ordering for fields and actions inside an embedded property you can follow three ways:

1. Use the [Java Annotation](#) in both the sources of containing and contained classes. But every time the class will be displayed will be taken that ordering.
2. Use only the [XML Annotation](#) for both the sources of containing and contained classes. But every time the class will be displayed will be taken that ordering.
3. Use only the [XML Annotation](#) for only the containing class and expand the embedded property in the definition.



The best way is using the 3rd approach in order to control fields and actions ordering in every single class.

Example on using the 3rd approach. The Zoo class [extends by composition](#) the Foo class declared previously.

Zoo.java:

```

public class Zoo extends ComposedEntityInstance<Foo> {
  private String a;
  private Integer z;
}

```



Remember that every class that extends the **ComposedEntityInstance** class has the **"entity"** property with the object that extends. For more information see: [Extension by Composition](#).

Zoo.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
<fields>
  <field name="a" />
  <field name="entity"> <!-- THE EMBEDDED INSTANCE -->
    <class>
      <fields>
        <field name="aa" />
        <field name="bb" />
        <field name="cc" />
        <field name="dd" />
        <field name="ee" />
      </fields>
      <actions>
        <action name="ok"/>
        <action name="cancel"/>
      </actions>
    </class>
  </field>
  <field name="z" />
</fields>
</class>
```

There are no limits about the level of depth in declaration of embedded instances.



Is not yet supported to declare annotations recursively on **collection properties**. Future version will.

14.11 Form Validation

See the [Validation Aspect](#) to validate forms using Annotations or a custom mode.

14.12 Annotations

View annotations allow to define how POJOs, fields and methods have to be rendered as displayed components.

14.12.1 @ViewClass

This annotation can be placed on a class declaration and describes how instances of this class have to be rendered. The interface of the annotation is the following:

```
public @interface ViewClass {  
    String label() default AnnotationConstants.DEF_VALUE;  
    String description() default AnnotationConstants.DEF_VALUE;  
    Class<?> entity() default Object.class;  
    String render() default AnnotationConstants.DEF_VALUE;  
    String layout() default AnnotationConstants.DEF_VALUE;  
    String style() default AnnotationConstants.DEF_VALUE;  
    byte explicitElements() default AnnotationConstants.UNSETTED;  
    byte hideRequiredColumn() default AnnotationConstants.UNSETTED;  
}
```

We report a brief description of the annotation properties:

Property	Description	Allowed values
label	Is the label that is assigned to instances of this class when they are displayed (e.g. if the instance is displayed as a popup window, the label is the heading of the window). The default label of an object is the class simple name, with a space before each capital letter	custom String values
entity	this property has to be used when defining ComposedEntity classes and represents the entity type.	Object.class (domain classes)
render	Represents the default rendering of instances of this class (e.g. popup).	one of the ViewConstants.RENDER_* constants
layout	Where to place the form	one of the ViewConstants.LAYOUT_* constants or "screen:<screenarea>"
style	Customized style name	
explicitElements	if set to true, only elements having ViewField(visible=AnnotationConstants	AnnotationConstants.TRUE,

	.TRUE) annotation are displayed. By default it is set to false.	AnnotationConstants.FALSE
--	---	---------------------------

14.12.2 @ViewField

This annotation can be placed on a field, getter or setter declaration and describes how the corresponding field has to be rendered. The interface of the annotation is the following:

```
public @interface ViewField {

    String label() default AnnotationConstants.DEF_VALUE;
    String description() default AnnotationConstants.DEF_VALUE;
    byte required() default AnnotationConstants.UNSETTED;
    byte visible() default AnnotationConstants.UNSETTED;
    byte enabled() default AnnotationConstants.UNSETTED;
    String render() default AnnotationConstants.DEF_VALUE;
    String layout() default AnnotationConstants.DEF_VALUE;
    String style() default AnnotationConstants.DEF_VALUE;
    String match() default AnnotationConstants.DEF_VALUE;
    int min() default DEF_MIN;
    int max() default DEF_MAX;
    String selectionField() default AnnotationConstants.DEF_VALUE;
    byte selectionMode() default AnnotationConstants.UNSETTED;
    String format() default AnnotationConstants.DEF_VALUE;

    public static final int DEF_MIN = Integer.MIN_VALUE;
    public static final int DEF_MAX = Integer.MAX_VALUE;

}
```

We report a brief description of the annotation properties:

Property	Description	Allowed values
label	it is the label that appears near the field value on the view. By default the label of a field is the field name, with the first capital letter and a space before each upper case letter.	custom String values
visible	Defines whether a field has to be visible or not. Default value is true, but becomes false if the class declares "explicitElements = true"	AnnotationConstants.TRUE, AnnotationConstants.FALSE
enabled	Defines whether a field has to be enabled for user modification or not. Default value is true.	AnnotationConstants.TRUE, AnnotationConstants.FALSE

		SE
render	Defines the type of rendering of the field (see 14.8)	one of the ViewConstants.RENDER_* constants
layout	defines the position of this field in the form	one of the ViewConstants.LAYOUT_* constants or "screen:<screenarea>"
style	Customized style name	
selectionField	it applies to collection and map fields rendered as selection fields (selection, radio box, list etc.). Declares the field that has to contain the value (or the index) of the currently selected value(s)	a valid field name for the current class.
selectionMode	it applies to list fields rendered as selection fields (selection, radio box, list etc.). It declares if the selection field has to be populated with the selected item or with the index of the selected item in the list.	ViewFieldFeatures.SELECTION_MODE_VALUE, ViewFieldFeatures.SELECTION_MODE_INDEX
format	It applies to date and numeric field and define the format to print the value, support i18n. This format is supported only for date and label render mode.	custom String values

14.12.3 @ViewAction

This annotation can be placed on methods having void return type and no arguments. It describes how the corresponding method has to be rendered. The interface of the annotation is the following:

```
public @interface ViewAction {
    String label() default AnnotationConstants.DEF_VALUE;
    String description() default AnnotationConstants.DEF_VALUE;
    String actionName() default AnnotationConstants.DEF_VALUE;
    byte visible() default AnnotationConstants.UNSETTED;
    String type() default AnnotationConstants.DEF_VALUE;
    String render() default AnnotationConstants.DEF_VALUE;
    String layout() default AnnotationConstants.DEF_VALUE;
    String style() default AnnotationConstants.DEF_VALUE;
}
```

```

    byte bind() default AnnotationConstants.UNSETTED;
    byte enabled() default AnnotationConstants.UNSETTED;
    byte validation() default AnnotationConstants.UNSETTED;
}

```

We report a brief description of the annotation properties:

Property	Description	Allowed values
label	it is the label that represents the action on the screen. By default the label of an action is the method name, with the first capital letter and a space before each upper case letter (See also).	custom String values
visible	Defines whether a method has to be visible or not. Default vaule is true, but becomes false if the class declares "explicitElements = true"	AnnotationConstants.TRUE, AnnotationConstants.FALSE
render	Defines the type of rendering of the action (see 14.8)	one of the ViewConstants.RENDER_* constants
layout	defines the position of this action in the form	one of the ViewConstants.LAYOUT_* constants or "screen:<screenarea>"
style	Customized style name	
bind	Bind all the fields changed in the form before to call this action. Default is TRUE	AnnotationConstants.TRUE, AnnotationConstants.FALSE
enabled	Defines whether an action has to be enabled for user modification or not. Default value is true.	AnnotationConstants.TRUE, AnnotationConstants.FALSE

14.12.4 XML layout

In XML annotations you can define the layout of a particular POJO as nested grids of areas. Each area can contain one or more actions and fields of the POJO. If POJO does not explicitly define a form, it inherits the **Object.xml** form. By convention a field is positioned (by priority order):

- in the area named as its "ViewField.layout" attribute
- in the area named as the field
- in the area named "fields"

By convention an action is positioned (by priority order):

- in the area named as its "ViewAction.layout" attribute
- in the area named as the method
- in the area named "actions"

Areas can be defined as:

- column: each element in the area is placed below the previous one
- row: each element in the area is placed on the right of the previous one
- grid: each element is placed in cell of the grid (if you define this kind of area you have to declare the number of the columns of the grid, like size="<number>")
- placeholder: a simple placeholder for an element (this is the default area type)

Here we report a some example of how to create hierarchical positioning of fields and actions on a form:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <aspects>
    <view>
      <form>
        <area name="main" type="column">
          <area name="fields" type="grid" size="2" />
          <area name="actions" type="row" />
        </area>
      </form>
    </view>
  </aspects>
```

```
</class>
```

This is the Object.xml form and is the default form that a POJO uses if it does not define its own form. In this form all the fields are placed in the "fields" area, that is a grid of two columns (remember that if the label of a field is not empty, it will occupy a cell); the actions are placed in a row positioned under the fields (because the main area, that contains fields and actions is a column).

You can define your form in a MyClass.xml file (if your class is named MyClass, obviously) to make a more detailed positioning of the fields:

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <aspects>
    <view>
      <form>
        <area name="main" type="column">
          <area name="mainFieldsArea" type="column">
            <area name="myField1" type="row">
            <area name="myField2" type="row">
            <area name="fields" type="grid" size="4">
          </area>
          <area name="actions" type="row" />
        </area>
      </form>
    </view>
  </aspects>
</class>
```

In this form we refined our representation, defining that the fields named "myField1" and "myField2" have to be placed in a column and before all the other fields. We also defined that the other fields have to be placed in a grid of four columns.

14.13 Push commands

The [View Aspect](#) allows to push commands in order to execute actions such as download a file, redirect to a page, and so on. Push command is the only way to works with sessions different by current.

14.13.1 Download file

14.13.1.1 Download a file using an InputStream as source

If your source is a `java.io.InputStream` implementation then use the `DownloadStreamViewCommand` command.

Example:

```
ViewCommand cmd = new DownloadStreamViewCommand(stream, "readme.txt",  
"text/plain");  
  
Roma.component(ViewAspect.class).pushCommand(cmd);
```

14.13.1.2 Download a file using an Reader as source

If your source is a `java.io.Reader` implementation then use the `DownloadReaderViewCommand` command instead.

Example:

```
ViewCommand cmd = new DownloadReaderViewCommand(reader, "readme.txt",  
"text/plain");  
  
Roma.component(ViewAspect.class).pushCommand(cmd);
```

14.13.2 Open a new window

If you want to open a new window just use the `OpenWindowViewCommand` push command.

Example:

```
ViewCommand cmd = new OpenWindowViewCommand("http://www.newurl.com");  
  
Roma.component(ViewAspect.class).pushCommand(cmd);
```

14.13.3 Redirect to a location

If you want to send a redirect to your Web Browser you had to use the `RedirectViewCommand` push command.

Example:

```
ViewCommand cmd = new RedirectViewCommand("http://www.newurl.com",  
null );  
  
Roma.component(ViewAspect.class).pushCommand(cmd);
```

14.13.4 Refresh a form

If you want to refresh a form in a session you had to use the `RefreshViewCommand` push command. The first argument is the `SessionInfo` telling to Roma the session where the form is contained. The second one is the Form to refresh.

Example:

```
ViewCommand cmd = new RefreshViewCommand( colleagueSession, form );
```

```
Roma.component(ViewAspect.class).pushCommand(cmd);
```

14.13.5 Show a form

If you want to show a form in another user session you had to use the `ShowViewCommand` push command. The first argument is the `SessionInfo` telling to Roma the session where the form is contained. The second one is the Form to display and last is the location area where to display.

Example:

```
ViewCommand cmd = new ShowViewCommand( colleagueSession, warningForm,
    "screen:popup" );

Roma.component(ViewAspect.class).pushCommand(cmd);
```

14.14 Implementation: Echo2

The first implementation of the [View Aspect](#) is based on Echo2 Web Framework (<http://echo.nextapp.com/site/echo2>). Echo2 can create web 2.0 interfaces without writing client-side scripting or pages, but just writing Java server-side code.

14.14.1 Get the underlying Echo2 components of the XML Configurable Screen

From any point in the Roma based application to obtain the root element of the xml screen:

```
ConfigurableScreen screen = (ConfigurableScreen)
ObjectContext.getInstance().getDesktop();
AreaInstance rootArea = screen.getRoot();
```

Starting from the root element you can navigate the entire tree using a XPath-like syntax:

```
AreaInstance myArea = rootArea.searchNode( "//MyAreaName" );
Component c = myArea.getComponent();
```

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<screen xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.orienttechnologies.com/roma/schema/roma-view-screen.xsd>
  <area name="main" type="column">
    <area name="header" />
    <area name="dashboard" type="dashboard" size="2">
      <area name="menubar" />
      <area name="userSession" />
      <area name="body" />
      <area name="help" />
    </area>
    <area name="footer" />
  </area>
</screen>
```

To obtain the “body” cell of the desktop:

```
ConfigurableScreen cScreen = (ConfigurableScreen)
ObjectContext.getInstance().getDesktop();

AreaInstance area = cScreen.getRoot();
```

```
AreaInstance bodyArea=area.searchNode( "//body" );  
Component bodyEcho2Component= bodyArea.getComponent();
```

14.14.2 Using JSP

You can use JSP technology together with Echo2 forms. The simpler way is to redirect the browser to the page using the [Push Redirect command](#) or by opening a new popup window to the JSP page using the [Push Open Window](#) command.

14.14.2.1 Exchange information between Business Logic and JSP

Since JSPs run in separate context than Echo2 you need to store the information you want to exchange in the Http Session (From the JSP you cannot use the Session Aspect).

From the Business Logic side use the Session Aspect in this way:

```
Roma.session().setProperty("account", myAccount );  
  
ViewCommand cmd = new  
RedirectViewCommand("http://localhost:8080/shop/dynamic/Shop.jsp",  
null );  
  
Roma.component(ViewAspect.class).pushCommand(cmd);
```

From the JSP side read the information needed by accessing directly to the Http Session in this way:

```
<html><head></head><body>  
...  
<%  
BaseAccount account = (BaseAccount) session.getAttribute("account");  
%>  
...  
</body></html>
```



Avoid to store too many objects inside the Http Session to avoid the overload of the Servlet Container. To get the object stored from the Business Logic side and remove from the Http Session (to free memory space) in one shot use the `session.removeAttribute()` instead of the `session.getAttribute()`.

14.14.3 Stylesheets

[View-Echo2](#) implementation comes with proprietary stylesheet format. It's a XML document based on the XML Schema: <http://www.romaframework.org/schema/echo2-stylesheet.xsd>.

All the stylesheet files are placed under **<user-package>.view.style**. The default stylesheet file is called **default.stylesheet** and contains the default styles for all base components. By convention each module has a own stylesheet file.

To create a new style for a component use this format:

```
<style name="<target>" [base-name="<base-target>"]  
  type="<component>">  
  <properties>  
    <...>  
  </properties>  
</style>
```

Where:

- **target** is the name of the target style. Use the:
 - "Object" to propagate the style to all POJOs, since Object is the base class of all Java classes.
 - The class name to apply to all the members of the class (fields and actions). Example: "Customer"
 - The class name plus member to restrict the style just to that member. Member can be a field or an action. Example: "Customer.name" or "Customer.print".
 - Special names as:
 - *ComposedComponent.lookup* (type="nextapp.echo2.app.Button") for the button used to link objects.
 - *ComposedComponent.reset* (type="nextapp.echo2.app.Button") for the button used to reset the link to object.
- **base-target** is the name of the style defined where to inherit properties. Use this to extend the properties of a component. Usually the base-name id "Object" to use the default one.
- **component** is the full class name of the component we are interested to define the style properties. It must be a Echo2 component. You can use your own Echo2 components or the standard well known:

Component's class	Description
-------------------	-------------

<code>echopointng.ContainerEx</code>	The base component for all containers
<code>nextapp.echo2.app.text.TextComponent</code>	It's the base component for Text Field and Text Area
<code>nextapp.echo2.app.TextField</code>	Text Field component
<code>echopointng.TextAreaEx</code>	Text Area component
<code>nextapp.echo2.app.Label</code>	Label component
<code>nextapp.echo2.app.Column</code>	Column place components
<code>nextapp.echo2.app.Row</code>	Row place components
<code>nextapp.echo2.app.Grid</code>	Grid to place components
<code>nextapp.echo2.app.Button</code>	Button component. Generally used for actions
<code>nextapp.echo2.app.ListBox</code>	List Box component
<code>nextapp.echo2.app.SelectField</code>	Select component
<code>nextapp.echo2.app.Table</code>	Table component
<code>nextapp.echo2.app.WindowPane</code>	Popup window
<code>org.romaframework.aspect.view.echo2.component.ComposedComponent</code>	Link Component. It's the link to other objects. Generated using the render="objectlink"
<code>org.romaframework.aspect.view.echo2.component.DynaDate</code>	Dynamic Date component
<code>echopointng.DateField</code>	Base DateField component
<code>nextapp.echo2.extras.app.MenuBarPane</code>	Menu Bar component
<code>org.romaframework.aspect.view.echo</code>	Tab Pane component

<i>2.component.DynaTabbedPane</i>	
<i>echopointng.ImageIcon</i>	Image Icon component

Example:

```

<!-- APPLY TO THE FIELD "MIN" OF CLASS "UserField" THE TEXT
FIELD
    WIDTH TO 40 PIXELS -->
<style name="UserField.min" base-name="Object"
    type="nextapp.echo2.app.text.TextComponent">
    <properties>
        <property name="width" value="40px" />
    </properties>
</style>

<!-- APPLY TO ALL BUTTON COMPONENTS THE FONT ARIAL 12 -->
<style name="Object" type="nextapp.echo2.app.Button">
    <properties>
        <property name="font">
            <font typeface="Arial" size="12px" />
        </property>
    </properties>
</style>

```

14.15 Implementation: Janiculum

This implementation is being implemented under the Romulus project (<http://www.ict-romulus.eu>).

This module doesn't use any Web Framework in the behind, but translates directly the generic view concepts of Roma in HTML pages using XHTML and CSS2 standards. The reason no Web Framework has been used for rendering resides on the previous experience with [View-Echo2](#). Echo2 is a very powerful web framework with great features. Unfortunately the development is decreased year-over-year and doesn't allow to use the CSS technology in order to customize the GUI.

This is also the main reason we decided to create a parallel implementation to the [View Aspect](#). The main purpose of this module is to allow a very wide range of view customization, using well known technologies like HTML and CSS.

This module implements the [View Aspect](#) and Session Aspect.

14.15.1 Architecture

Janiculum has to face many problems to generate a proper view starting from POJOs: first of all it has to map POJOs on screen/form areas to reproduce Roma form layout. Then it has to create an association between view elements and POJOs. To do this, Janiculum has to:

- generate page structure from Roma Screen and Form definitions
- generate HTML fragments for each POJO/field/action
- associate an unambiguous id to each POJO/field/action, so that request parameters coming from user agents can be bound to the right element on the server side.
- perform the right conversion between request parameters (that are strings) and POJO field values.
- recognise which action has been performed on the user agent and invoke the corresponding method on the server side POJOs

To perform these operations, Janiculum relies on some important components:

- **Transformers:** Transformers are components that are responsible of transforming POJO elements in XHTML fragments. A transformer has to write in the HTML some information that is essential both to allow the binder to obtain values from the page form request and to allow the user to customize the page with stylesheets. These informations are mainly form element names and html element classes. Janiculum contains different types of transformers, one for each type of rendering mode.
- **Binders:** A binder, that is associated to a transformer, is the component that knows how to receive request parameters, translate them in the right format and set them in the POJO attributes on the server.
- **Screen JSP:** Roma Screen is the highest level representation of a view of the application. In Janiculum each screen is associated with a JSP, whose name is exactly the same as the xml screen name, but with ".jsp" suffix. In this page are contained the JavaScript and CSS import for the pages, the html form tag to send requests to the server and other html content that can be customized by the user. In the Screen JSP are contained the roma:screenArea tags that indicate where to put Roma screen areas in the page.
- **POJO JSP:** Janiculum can use a custom JSP to represent POJOs that are instances of a particular class. The basic behaviour in rendering a POJO is to look for a JSP whose name is the same as the POJO class name (with the right upper case characters and ".jsp" suffix). If this JSP exists the Aspect will use this to represent the POJO, otherwise it will climb the class hierarchy until it finds a suitable JSP (by default Roma provides an Object.jsp file for

default rendering). In POJO JSP files are contained roma:class, roma:field and roma:action tags that allow the user to customize the view putting in the desired position each field/action of the POJO.

- **Roma tag library:** as stated before, Janiculum provides a tag library that allows to compose screen JSP and POJO JSP to obtain custom views.

14.15.2 Basic usage

As each Roma [View Aspect](#) implementation, Janiculum performs completely automatic POJO rendering by default. It means that you can write your domain and application logic without worrying about the presentation layer: Roma will provide you a basic XHTML/CSS view of your POJOs. To obtain this behaviour you just have to use Roma as you did with other [View Aspect](#) implementations, writing rendering information in Roma Annotations or in Roma XML Annotations.

14.15.3 Customization through CSS

As default rendering of your POJOs generated by Janiculum is plain XHTML code, you can add your own CSS to the pages to modify both element appearance and positioning.

Janiculum generates html element "class" and "id" attributes in a deterministic way, so you can rely on these attributes to write your CSS.

About "id" attribute, it is composed appending:

- area names that recursively contain the element (separated by "_")
- an underscore followed by the Class name of the element or of the POJO that contains it (if it is a field or an action)
- an underscore followed by the field or action name (only if the element corresponds to a field or an action)
- a suffix that depends on the sub-element of the corresponding html (e.g. "_item_<n>" for list and table items)

Here we report a list of "class" attribute values associated with different attribute rendering modes and parts of generated HTML.

Class name	Rendered object
cell, column, grid, placeholder, row	these names are applied to <div> elements that represent screen and form areas. The class name corresponds to the area type.
accordion, button, chart,	these names are applied to <div> elements that

check, colset, date, datetime, decimal, grid, html, image, imagebutton, number, label, link, list, menu, objectlink, objectembedded, password, popup, progress, radio, richtext, rowset, select, tab, table, tableedit, text, textarea, time, tree, upload,	contain a field or an action of a POJO. The class name corresponds with the current rendering of the field or action
<RENDERING_MODE>_label	this class is applied to <label> elements that represent labels of a field of a POJO. <RENDERING_MODE> corresponds to the rendering type of the attribute (ex. if a field has "select" rendering model, its label will have "select_label" class). This label is the first element inside the <div> that represents a POJO field.
<RENDERING_MODE>_content	this class is applied to the elements that represent values of a field of a POJO. <RENDERING_MODE> corresponds to the rendering type of the attribute. This element is generally the second element (after the "label") inside the <div> that represents a POJO field.
[list table]_selection	this class is applied to checkboxes used to select list or table rows
[list table]_actions	this class is applied to the <div> that contains table or list operation buttons
[list table]_actions_[add edit view remove up down]	these classes are associated to buttons that represent corresponding list and table actions

14.15.4 Global (application level) CSS

As Janiculum always uses a screen JSP for screen rendering, you can modify this JSP adding there your CSS references. These CSS will be loaded in all the pages that share the same screen definition (so, probably, in all your application).

As you can see, Roma provides a default screen JSP, named "main-screen.jsp" that corresponds to the "main-screen.xml" screen definition. In this JSP you can find by default one of the following tags:

- **<roma:css>** - this tag adds to the page a reference to the Roma servlet that

automatically generates positioning css.

- **<roma:inlinecss>** - this tag adds inside the page (in-line) the positioning CSS generated by Roma (this tag has to be put always at the end of the page)

If you want to change default Roma behaviour about CSS, you can remove these tags, so that Roma CSS will not be added to the page any more, or you simply can add your CSS to the page near these tags, so that your custom CSS goes to enrich Roma default.

14.15.5 Class level CSS

As Roma relies on <ClassName>.jsp files to represent POJOs (see customization through JSP), if you want to add a fragment of CSS only to pages that contain a POJO that is an instance of a particular class, you just have to put your CSS in the corresponding JSP file.

This is as simple as writing in your MyDomainClass.jsp file the following

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="/WEB-INF/roma.tld" prefix="roma"%>
<%@ page buffer="none" %>

<link REL="stylesheet" TYPE="text/css" href="myStyle.css">

<roma:class />
```

14.15.6 Customization through HTML/JSP

If css customization is not enough for your application, you can modify html that is generated by Roma for each different class.

By default Janiculum, to render a POJO, looks for a JSP whose name is <class-name>.jsp, where <class-name> is the name of the class of the current POJO. If this JSP does not exist, Roma goes up in the class hierarchy until it finds a suitable JSP; Janiculum by default provides an "Object.jsp" that is used when no specific JSP is found for a particular class. These JSP files have to be put in the directory of your application called "/dynamic/custom/view".

The best way to start writing your custom JSP is to generate it using the [JSP command console](#).

In your custom JSP you have to use Roma taglibs, that generate for you the html code that represents POJO fields and actions. Around Roma tags you can put your custom html code. Let's analyse how to use Roma taglib: when Roma tries to render a POJO, it calls the corresponding JSP (Object.jsp if no custom JSP is defined) passing it the POJO as the "current object"; in the page there should be Roma tags that can process the pojo:

- **<roma:class>** - this tag displays the whole pojo with its default rendering. It is the easiest way to represent a POJO. Using this tag you can just put before and after the default representation your custom html code. This field is used by default by Object.jsp
- **<roma:field name="fieldName" part="partName">** - this tag displays a field of the current POJO. The "name" attribute (mandatory) has to contain the name of the POJO attribute. The "part" attribute (optional) can contain one of the following: "all" to display the whole field; "label" to display the l18Nized label of the field, "content" to display the field content; if you omit the "part" attribute, it will be treated as "all" value.
- **<roma:action name="actionName">** - this tag displays an action of the current POJO. the "name" attribute (mandatory) has to contain the name of the POJO action.

Here we report a very basic example of how to customize a class with two fields and an action using a JSP.

Example:

Person.java

```
public class Person{
    private String firstName;
    private String surname;

    // add here getters and setters

    public void clickThis(){
        // do something
    }
}
```

you can add a JSP with the same name as the class (Person.jsp), defining in detail the HTML that has to be generated for instances of this class and the order and position of each field and action.

Person.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="/WEB-INF/roma.tld" prefix="roma"%>
<%@ page buffer="none" %>

Employee name: <roma:field name="firstName" part="content" />

Here I write my html code <roma:field name="surname" />

And here I write more html code <roma:romaAction name="clickThis" />
```

```
<%  
    //here I write my scriptlet  
%>
```

Now each time you render an instance of Person, this JSP will be used and you will obtain a completely customized result, with custom code generated by Roma for each field/action of the POJO embedded in your totally customized html. As this page is parsed by a JSP compiler you can also put in it custom JSP scriptlets and tags from third party tag libraries.

Besides POJO JSP customization, you can also customize screens. Roma provides a basic "main-screen.jsp" screen JSP representation, that is associated to "main-screen.xml" screen. If you want to add your own screen you also have to provide the default JSP for this screen, where the JSP name has to match exactly the xml screen file name. In a screen you can put your custom screen areas with the tag `<roma:screenArea name="areaName">`. The name attribute defines the name of the area in the xml screen definition.

If you want to display the whole screen, you just have to put the tag with the root area as name (`<roma:screenArea name="/">`), this way you can customize your screen adding before and after the main area your html code. If you want to put your html between different areas you have to put more `<roma:screenArea >` tags, one for each screen area that you want to render.

14.15.7 Customization of automatic behaviour

If you want to go really deep in the customization, you can override the default field and action generation behaviour writing your own Transformers and Binders.

To do this you have to keep in mind some important rules:

- a Transformer has to implement the Transformer interface (see Javadoc of Janiculum). This interface has only three methods:
 - `transform(renderable)`: is invoked when the aspect has to render an `HtmlViewRenderable` (that is a wrapper of a POJO) and returns the generated HTML
 - `transformPart(renderable, part)`: is invoked when the aspect has to render a part of a renderable (part can be "label", "content", "raw", "actions" or "all") and returns the generated HTML
 - `getBinder()`: has to return the Binder associated to this Transformer
- a Transformer has to generate html code that produces request parameters that can be parsed by the corresponding Binder
- a Transformer has to override the `toString()` method to return the name of the rendering mode that the Transformer implements.

- a Binder implements the `HtmlViewBinder` interface and is responsible of binding request data to the corresponding POJO. Its interface has only one method: `bind(HtmlViewRenderable renderable, Map<String, String> values)`. Through this method the binder receives the renderable that wraps the POJO and the request values that are associated with this POJO (all the parameters whose name starts with the id of the POJO as calculated by `HtmlViewRenderable.getId()` method).
- The `HtmlViewRenderable` objects passed to Binders and Transformers are generally `HtmlViewContentComponent` or `HtmlViewActionComponent` instances.
- To let Roma Janiculum be aware of your Transformer you have to add it to the `HtmlViewTransformerManager` bean in the `applicationContext-janiculum.xml` file of your application (removing the transformer that implemented the same rendering mode of your one).

14.15.8 FreeMarker Transformers

The default implementation of Transformers is based on FreeMarker template engine. A FreeMarker transformer is composed of two parts: a Java class, that is responsible of retrieving information such as element ids and html classes, and a FreeMarker template, that is only responsible of the rendering of the corresponding HTML.

Default FreeMarker transformer templates are placed in the application directory `WEB-INF/freemarker/transformers/ajax`. Each template is named as `<rendering-mode>.ftl`, where `<rendering-mode>` is the name of the field rendering mode (e.g. "button", "table" etc.).

A simple way to modify the html fragments generated for a single rendering mode is to just modify the corresponding `.ftl` template. This template can also be modified at run-time, because it will be reloaded after modification.

Actually Janiculum comes with two different sets of transformers,

- one in the folder `WebContent/WEB-INF/freemarker/transformers` that does not contain JavaScript code and is intended for accessible application development
- one in the folder `WebContent/WEB-INF/freemarker/transformers/ajax` that contains full JavaScript and AJAX support; this set of transformers is selected by default

If you want to change your set of transformers or you want to configure Janiculum to use your own set of transformers you just have to reconfigure the bean named "FreeMarkerTemplateManager" in the Janiculum Spring configuration (`applicationContext-janiculum.xml`).

14.15.9 Html events

One of the main features of Janiculum, that allow you to take full control of your page design, is the ability to declare in your Java classes events that can be bound to html javascript events.

Take this very simple case into consideration: you have a text field and you want to execute some code when the user changes its content. In a normal html page you would write something like this:

```
<input type="text" name="myTextField" onchange="doSomething()">
```

And then you should implement your “doSomething()” function to make an AJAX call to the server.

In Janiculum you can obtain the same result simply by convention, declaring a java method composing the field and the event names:

```
class Person{
    String name;

    public String getName(){
        return name;
    }

    public void setName(String iName){
        this.name=iName;
    }

    public void onNameChange(){
        //do something
    }
}
```

Displaying this class, janiculum will add in the html page an “onChange” event on the “name” field; this event will automatically produce an AJAX call to the server that will invoke the “onNameChange” method and will update the page with changes produced by the method invocation.

To use this mechanism you only have to take care of following this convention: the event method must:

- be public
- have “void” return type
- have no arguments

- be named “on<FieldName><Eventname> (first capital of field name and event name)

This way you can intercept all html events, such as:

- onclick
- onchange
- onmouseover
- onmouseout
- onmouseup
- onmousedown
- onblur
- onfocus
- onkeypressed
- onkeyup
- onkeydown
- etc.

15 Validation Aspect



This Aspect was introduced starting from 2.0 version. Before this release validation was covered by the [View Aspect](#). In facts most of annotations are moved from [View Aspect](#) to this one. Assure to move all the old annotations to use the Validation Aspect.

Roma allow to validate any POJOs as long as forms (since they are POJO themselves). There are two kinds of validations:

- **Field Validation**, by using annotations
- **Custom Validation**, by implementing an interface and checking what you want in your POJO.

15.1.1 Field Validation

The most common validation rules are implemented to field level by using Java 5 annotations or XML descriptor:

- **required**, if a field is required before to process actions
- **min**, specifies the minimum length for String fields and minimum value for numeric fields
- **max**, specifies the minimum length for String fields and maximum value for numeric fields
- **match**, specifies the Regular Expression to express complex rules

The example below shows how to set these constraints by using [Java Annotations](#):

```
public class User {
    @ValidationField(min = 3, max = 8)
    private String name;

    @ValidationField(min = 18, max = 130)
    private String age;

    @ValidationField(render="password", required = AnnotationConstants.TRUE)
    private String password;

    @ValidationField(match="^[_a-z0-9-]+(\\.[_a-z0-9-]*)*@[a-z0-9-]*(\\.[_a-z0-9-]*)*$")
    private String email;
}
```

Or the same using [XML Annotations](#):

```
<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.orienttechnologies.com/roma/schema/v2/roma.xsd">
  <fields>
    <field name="name">
      <aspects>
        <validation min=3 max=8 />
      </aspects>
    </field>
    <field name="age">
      <aspects>
        <validation min=3 max=8 />
      </aspects>
    </field>
    <field name="password">
      <aspects>
        <validation render="password" required="true" />
      </aspects>
    </field>
    <field name="email">
      <aspects>
        <validation match="^[_a-z0-9-]+(.[_a-z0-9-]*)*@[a-z0-9-]*(.[_a-z0-9-]*)*$" />
      </aspects>
    </field>
  </fields>
</class>
```

15.1.2 Custom Validation

When things became more complex you may need to use the **Custom Validation**. Roma checks if POJOs implement the CustomValidation interface. If yes, after passed the **Field Validation**, Roma controller invokes **validate()** method on POJO.

If in class contains fields with validation annotations and it implements also the CustomValidation, then Roma will check the validation rules expressed using the annotations and then will call the validate() method.

The method **validate()** in the user's class must throw a **ValidationException** or a **MultivalidationException** exceptions if respectively one or multiple validation rules are violated. **ValidationException** constructor accepts the following parameters:

- **Object iInstance**, where you make the rule checking
- **String iFieldName**, is the name of field that violated the rule
- **String iMessage**, the message you want to show. Prefix the message content with \$ character to use I18N. Example: "\$MyMessage.error".
- **String iRefValue**, the reference value if any, otherwise NULL

The example below shows the validation rules for a User data input:

```
import org.romaframework.core.validation.CustomValidation;
import org.romaframework.core.validation.ValidationException;

public class User implements CustomValidation {
    @ValidationField(min = 3, max = 8)
    private String name;

    @ValidationField(render="date", required = AnnotationConstants.TRUE)
    private Date birthDate;

    @ValidationField(required = AnnotationConstants.TRUE)
    private String phone;

    private boolean agree;

    public void validate() throws ValidationException {
        if (birthDate > new Date() )
            throw new ValidationException(this, "birthDate", "Invalid date!", null);

        if (!phone.startsWith("+"))
            throw new ValidationException(this, "phone", "Please insert the
international prefix. Example: +39", null);

        if (!agree)
            throw new ValidationException(this, "agree", "You must agree to
continue", "true");
    }
}
```

15.1.3 Throw multiple exceptions in one shot

If you need to signal to the user **ALL** rules violated in one shot (without signal one by one) you should use the `MultiValidationException` instead of `ValidationException` used before.

Example:

```
public void validate() throws ValidationException {
    MultiValidationException e = new MultiValidationException();

    if (birthDate > new Date() )
        e.addException( ValidationException(this, "birthDate", "Invalid date!",
null) );

    if (!phone.startsWith("+"))
        e.addException( new ValidationException(this, "phone", "Please insert the
international prefix. Example: +39", null) );

    if (!agree)
        e.addException( new ValidationException(this, "agree", "You must agree to
continue", "true") );

    if( e.hasExceptions())
        throw e;
}
```


15.1.4 Change the Look & Feel

On exception errors, a form is displayed by default reporting all exception raised. You can change the look and feel by [View-Echo2 stylesheet](#) document: **default.stylesheet**.

Field in error are marked with a red border. Also this setting is contained in the stylesheet:

```
<style name="field-error" base-name="default"
type="nextapp.echo2.app.text.TextComponent">
  <properties>
    <property name="insets" value="1px" />
    <property name="border">
      <border color="#ff0000" size="1px" style="solid" />
    </property>
  </properties>
</style>
```

15.1.5 @ValidationField

This annotation can be placed on a field, getter or setter declaration and describes how the corresponding field has to be rendered. The interface of the annotation is the following:

```
public @interface ValidationField {

    byte    required()          default AnnotationConstants.UNSETTED;
    String  match()             default AnnotationConstants.DEF_VALUE;
    int     min()               default DEF_MIN;
    int     max()               default DEF_MAX;

    public static final int DEF_MIN = Integer.MIN_VALUE;
    public static final int DEF_MAX = Integer.MAX_VALUE;

}
```

We report a brief description of the annotation properties:

Property	Description	Allowed values
required	it is used for form validation, a field with "required" set to true has to contain a value, otherwise the form submission will throw a validation exception. Default value is false.	AnnotationConstants.TRUE, AnnotationConstants.FALSE
match	Regular expression against to validate	See the syntax of Regular Expressions, i.e: http://www.regular-expressions.info

min	it is used for form validation. Applies to numeric fields and defines the minimum allowed value	an integer value
max	it is used for form validation. Applies to numeric fields and defines the maximum allowed value	an integer value

16 I18N Aspect

The I18NAspect is responsible of the localization of Roma applications. The main user of the I18NAspect is the [View Aspect](#), that uses it to translate class, field and action labels in the current locale.

16.1 Interface

The interface of I18NAspect is the following:

```
public interface I18NAspect extends Aspect {

    public static final String ASPECT_NAME = "i18n";
    public static final String VARNAME_PREFIX = "$";
    public static final String CONTEXT_SEPARATOR = ".";
    public static final String DEFAULT_VALUE_KEY = "$content";

    public String getLabel(SchemaObject iObject, String iElementName,
                          String iElementLabel);

    public DateFormat getDateFormat();
    public DateFormat getDateTimeFormat();
    public DateFormat getDateTimeFormat(Locale iLocale);
    public DateFormat getTimeFormat();
    public DateFormat getTimeFormat(Locale iLocale);
    public DateFormat getDateFormat(Locale iLocale);

    public String resolveString(Class<?> iObjectClass,
                              String iText, Object... iArgs);
    public String resolveString(String iText, Object... iArgs);

    public String getString(String iText);
    public String getString(String iText, Locale iLocale);

}
```

Its methods allow to:

- retrieve date and time format objects for the current locale
- obtain the translation of an object field or action label
- obtain a localized value from a resource bundle

16.2 Implementations

The current implementation of the I18NAspect is part of the **core** module and is based on Java resource bundles. It allows you to declare localized translations of

class, field and action labels and to define parametrized strings

The resource files are text files with extension ".properties"; these files have to be placed in the package named `<application-package>.domain.i18n`. The extension can be preceded by the code that identifies a locale.

A typical configuration of I18N files is the following

- `default_messages.properties` (contains the default messages)
- `default_messages_EN.properties` (contains the messages in English language)
- `default_messages_FR.properties` (contains the messages in French language)
- `default_messages_IT.properties` (contains the messages in Italian language)
- etc.

These files are simple text files; each row is composed by two parts: the `class.property.attribute` and the message

a very simple example of I18N files is the following:

`default_messages.properties:`

<code>Object.name.label</code>	First Name
<code>Object.surname.label</code>	Surname
<code>Person.label</code>	Person

`default_messages_IT.properties:`

<code>Object.name.label</code>	Nome
<code>Object.surname.label</code>	Cognome
<code>Person.label</code>	Persona

The syntax is very simple: the first element is the class name whose the element applies, the second is the property or the method name and the third is:

- "label" if the row applies to the element label
- "hint" if the row applies to the hint that appears when you pass on the element

You can also define class labels as `<Class-name>.label`; these labels will be represented as popup titles if the object is rendered as a popup.

I18NAspect uses class inheritance to find the appropriate labels, so if you define a label on Object class, it will be applied to all the classes unless you define a specific label for your class or another class in its hierarchy.

You can also add custom key/value couples in the resource bundle, without respecting the Class.field.label syntax; to retrieve the localized value of the label you just have to invoke the getString(String) method on the I18NAspect.

17 Persistence Aspect

The Roma's Persistence aspect does a magic art: makes Java objects persistent and retrieve their from database in transparent way. Most of the job is served by the underlying technology. Roma acts as a controller that orchestrate all the aspects together.

17.1 Choose the right strategy

To achieve automatic persistent with Roma you must to learn few, simple rules.

JDO engine can be used in three ways:

- **Transaction Strategy:** Objects are managed by the same JDO PersistenceManager implementation. You can update objects in transparent way inside a Transaction. JDO, at commit time, writes only attributes changed to the database. It handles lazy loading transparently.
- **No Transaction Strategy:** No transaction is started. Objects are managed by a JDO PersistenceManager implementation. You cannot update objects in transparent way but only access on its in read-mode. It handles lazy loading transparently.
- **Atomic Strategy:** Uses a single transaction for each operation. Is not always the best way to use the JDO engine since it's CPU and Memory consuming because it copies objects and it not allows lazy loading. The advantage of detaching over other modes is that objects can be modified and reattached in a subsequent operation.

Strategy	Persistence Aspect implementation	Transaction aware	Object access mode	Allow lazy-loading
Atomic	AtomicPersistenceAspect	Transaction begins and commits at each operation	Objects are detached	No
Tx	TxPersistenceAspect	Transaction begins when the Persistence Aspect is taken and is shared among all operations. Call commit() method to change updates	Objects are managed by a PersistenceManager	Yes

		persistently		
NoTx	NoTxPersistenceAspect	Transaction begins and commits at each operation	Objects are managed by a PersistenceManager	Yes

17.2 Atomic strategy

To use the Atomic strategy you need to set **detachable** attribute to **true** at every persistent class you want to use by following **Atomic** strategy. Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC
  "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
  "jdo.dtd">
<jdo>
  <package name="org.romaframework.test.presentation.domain">
    <class name="Employee" detachable="true">
      <field name="company" default-fetch-group="true"></field>
    </class>
    <class name="Company" detachable="true"></class>
  </package>
</jdo>
```

17.3 Change strategy for a single operation

17.3.1 Select the strategy on Query execution

You can override the strategy of Persistence Aspect used on query execution by defining the new strategy by **setStrategy()** method on Query object:

```
QueryByFilter query = new QueryByFilter( Employee.class );
query.setStrategy( PersistenceAspect.STRATEGY_TRANSIENT );
List<Employee> employees = db.query( query );
```

17.3.2 Select the strategy on object loading

You can override the strategy of Persistence Aspect used on loading an object by using the **load()** method passing strategy as last parameter:

```
Employee employee = db.loadObject( employee, "full",
PersistenceAspect.STRATEGY_TRANSIENT );
List<Employee> employees = db.query( query );
```

17.4 Java Interface

```
public interface PersistenceAspect extends Aspect {  
    public String getOID(Object iObject) throws PersistenceException;  
    public <T> T loadObject(T iObject, String iMode) throws PersistenceException;  
    public <T> T loadObject(T iObject, String iMode, byte iStrategy)  
        throws PersistenceException;  
    public <T> T loadObjectByOID(String iOID, String iMode)  
        throws PersistenceException;  
    public <T> T loadObjectByOID(String iOID, String iMode, byte iStrategy)  
        throws PersistenceException;  
    public <T> T createObject(Object iObject) throws PersistenceException;  
    public <T> T createObject(Object iObject, byte iStrategy)  
        throws PersistenceException;  
    public <T> T updateObject(Object iObject) throws PersistenceException;  
    public <T> T updateObject(Object iObject, byte iStrategy)  
        throws PersistenceException;  
    public Object[] updateObjects(Object[] iObjects) throws PersistenceException;  
    public Object[] updateObjects(Object[] iObjects, byte iStrategy)  
        throws PersistenceException;  
    public void deleteObject(Object iObject) throws PersistenceException;  
    public void deleteObjects(Object[] iObjects) throws PersistenceException;  
    public <T> List<T> query(Query iQuery) throws PersistenceException;  
    public <T> T queryOne(Query iQuery) throws PersistenceException;  
    public boolean isObjectLocallyModified(Object iObject)  
        throws PersistenceException;  
    public byte getStrategy();  
    public byte getTxMode();  
    public void setTxMode(byte txMode);  
    public boolean isActive();  
    public void commit();  
    public void rollback();  
    public void close();  
}
```

- The **createObject()** method make an instance persistent.

- **updateObject()** update the object in the repository. There is also the version that accepts an array of objects to be updated in the same transaction.
- **deleteObject()** delete the object in the repository. There is also the version that accepts an array of objects to be deleted in the same transaction.
- **query()** executes a query using the own Persistence Aspect implementation language. For example JDOQL for JDO implementation, EJBQL for EJB3 implementation and so on. Make attention that using directly query make the object not portable. So let's use the Custom Inheritance best practice?
- **queryByExample()** follow the QBE (Query By Example) pattern. It executes a query passing a class and an object as filter. The object filter fields different from NULL will concur to the where clause of query.

17.5 Query Patterns

Roma provides the Persistence Aspect module to handle in very simple way the most common cases:

- create objects
- update modified objects
- delete objects
- query using the pattern Query By Example
- query using the pattern Query By Filter
- query using a custom predicate

Before to use the Persistence Aspect you need to get a reference to it:

```
PersistenceAspect db = Roma.context().persistence();
```

Now you can create objects, update their, delete someone and more. If you are using JDO as persistence technology, to make query you can use **JDOQL**. **JDOQL** is the query language defined by the **JDO standard**. To know more look at JDO Resources. Just remember that JDOQL is strongly based on...Java! Yes, you can execute queries as you are writing an if condition as predicate.

Below an example used in the login module.

```
List<Account> result = db.query(new
    QueryByText(Account.class, "name == '" + iUserName + "'"));
```

As you can see the equals operator is as for Java language: `==`. Again, you can find tons of docs about JDOQL in JDO Resources.

17.5.1 Query by example

The auto-generated CRUD works with an instance of the domain class as filter. Taking the **Blog** example, the "**Post**" class should be something like this (getter and setter are omitted):

```
class Post
{
    private String title;
    private Date   postedOn;
    private String content;
}
```

When Roma generates the CRUD, the method **PostMain.search()** will execute a query among account instances and filtering by title, **postedOn** and **content** properties if they are valued.

This is the simpler **Query By Example**¹ implementation [Error: Reference source not found], but has many limits:

- You cannot query range of values (think to the range of blog posting, from-to)
- You cannot specify other query constraints
- By default not-null properties in the filter object are involved in the query. Primitive types cannot be null and are so always involved unless you hide that properties in the form (visible="false")
- You cannot navigate across object properties

So these are some reason because you may need the **Query By Filter pattern**².

17.5.2 Query by filter

The custom filter is implemented by the **QueryByFilter** class. It allows to specify query predicate in independent way of the Persistence Aspect implementation.

1 In the rest of the document named as *QBE*

2 In the rest of the document named as *QBF*

You can use it in CRUD by overriding the `search()` method (in the example above `PostMain.search()`):

```
@Override
public void search() {
    QueryByFilter dynaFilter =
        new QueryByFilter(QueryByFilter.PREDICATE_AND);

    if (filter.getEntity().getContent() != null) dynaFilter.addItem("content",
        QueryByFilter.FIELD_LIKE, filter.getEntity().getContent());

    searchByFilter(dynaFilter); }
```

With this code we have changed the default CRUD search behaviour and now the "content" field in the filter works using the LIKE operator. Note we have built `QueryByFilter` instance using `QueryByFilter.PREDICATE_AND` (default) telling that all conditions are in AND mode: search all instances where ALL conditions are satisfied. You can use `QueryByFilter.PREDICATE_OR` if you want that just one condition is satisfied.

If you want more powerful you need to use the specific query language of Persistence Aspect implementation: the **Query By Text pattern**. If you are using the *JDO* [Persistence-Datanucleus](#) or *JPOX* implementations you can use *JDOQL* or *SQL*, but the query is not portable across other Persistence Aspect implementations.

On resuming generally:

- 40% of your CRUD can fits well with default QBE
- another 40% probably needs the advanced QBF with custom filters
- the remaining 20% may need ad-hoc queries using full Query Language power of Persistence Aspect implementation (JDOQL, SQL, etc.) and the class `QueryByText`.

If you need to insert other fields in you filter form, you need to extend the CRUD Filter class. In the example above the generated class is `PostFilter`. We need to query all Posts written in a range of dates. So let's add two new properties to the `PostFilter` class (getter/setter are omitted):

```
@ViewClass(entity = Post.class)
public class PostFilter extends ComposedEntityInstance<Post> {
    private Date rangeFrom;
    private Date rangeTo;

    public PostFilter() {
        super(new Post());
    }

    public PostFilter(Post iPost) { super(iPost); }
}
```

You'll see the new entries in the form since ROMA automatically renders thier. But they are not yet envolved in query. Let's modify the QueryByFilter above:

```
@Override
public void search() {
    QueryByFilter dynaFilter = new QueryByFilter(QueryByFilter.PREDICATE_AND);
    if (filter.getEntity().getContent() != null)
        dynaFilter.addItem("content", QueryByFilter.FIELD_LIKE,
            filter.getEntity().getContent());

    // SET DATE RANGE
    if (filter.getRangeFrom() != null) dynaFilter.addItem("postedOn",
        QueryByFilter.FIELD_MAJOR_EQUALS, filter.getRangeFrom());
    if (filter.getRangeTo() != null) dynaFilter.addItem("postedOn",
        QueryByFilter.FIELD_MINOR_EQUALS, filter.getRangeTo());

    searchByFilter(dynaFilter);
}
```

Now the two fields are used in the query to display only posts written in the range of date specified. Remember to set the **postedOn** property to *visible="false"* to remove it by the form.

Now let's think to the Blog class. It should be something like this:

```
class Blog
{
    private String name;
    private Author author;
    private ArrayList<Post> posts;
}

class Author { private String name; private City city; }

class City { private String name; private String zip; }
```

If you'd like to search all Blogs with authors that live in Rome, write this filter:

```
@Override
public void search() {
    QueryByFilter dynaFilter = new QueryByFilter();
    dynaFilter.addItem("author.city.name", QueryByFilter.FIELD_EQUALS,
        "Rome");

    searchByFilter(dynaFilter);
}
```

ROMA uses the power of the OR-Mapping tool, in this case [Datanucleus](#)/JDO 2.

When you execute a query you can specify the mode of execution. In current JDO 2.0 implementation modes are mapped with JDO 2 fetch groups. In JDO you can define what fields to fetch on loading. The best approach is to define only the common fields with `default-fetch-group="true"` and let to specific modes to fetch other fields only on demand.

You can define the fetch group in the `package.jdo` file in this way:

```
<fetch-group name="full">
  <field name="levels"></field>
</fetch-group>
```

To use the full JDO fetch group you had to set the mode:

```
QueryByFilter dynaFilter = new QueryByFilter();

dynaFilter.setMode( "full" );
dynaFilter.addItem( "author.city.name", QueryByFilter.FIELD_EQUALS, "Rome");

Roma.context().persistence().query(dynaFilter);
```



Remember that "full" mode is always setted from CRUD when load object to view or update.

17.6 Implementations

17.6.1 DataNucleus

[DataNucleus](#) is an OR-Mapping tool compliant with JDO 2.1 and JPA 2 standards. We choose it in place of the more famous Hibernate because it supports the more complete JDO standard and has a pluggable Datastore architecture to support even ODBMS and other repository types.

JDO stands for Java Data Objects and it's a Sun Microsystems's standard for the persistence in Java. JDO is orthogonal and is not just relegated to RDBMS but to any kinds of storage: RDBMS, ODBMS, XML files, etc.

The `persistence-datanucleus` module is the implementation of the Persistence aspect using [DataNucleus](#) tool.



DataNucleus is the evolution of the JPOX project.

17.6.1.1 Persistence Mapping

Since Roma works always in JDO detach mode, you had to define what fields you need to access because in detach mode the JDO lazy-loading cannot be applied.

By default JDO 2.0 fetches all the fields of primitive and simple types (int, Integer, String, Date, etc.). Complex types such as relationships and collections/maps need to declare the fetch mode to be used in your application. Watch this example:

```
public class Employee {  
    private String name;  
    private Integer age;  
    private Date bornOn;  
    private Company company;  
}
```

All fields but **company** will be fetched on object retrieving. If you access to company attribute an `JDOException` will be thrown since the field was not fetched and lazy loading cannot be used on detached objects (see <http://www.datanucleus.org/products/accessplatform/jdo/fetchgroup.html> to know more about this).

You had to define the **company** field in your package.jdo file (under your application domain package).

Example:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects  
Metadata 2.0//EN" "jdo.dtd">  
<jdo>  
  <package name="com.mycompany.test">  
    <class name="Employee" detachable="true">  
      <field name="company" default-fetch-group="true"/>  
    </class>  
  </package>  
</jdo>
```

17.6.1.2 JDO resources

Roma's first Persistence Aspect implementation use the [DataNucleus tool](#). DataNucleus is the reference implementation choose by Sun Microsystems for JDO 2.0 standard. DataNucleus is very customizable tool and allow fine grained tuning.

- [JDO 2.0 home page](#)
- [DataNucleus](#)

- JDOQL
 - [DataNucleus's JDOQL page](#)
 - JDOQL tutorial by Robin Roos and published on TheServerSide site:
 - [Part I](#)
 - [Part II](#)
 - JDOQL tutorial in Italian language, translated from TheServerSide articles above:
 - [Part I](#)
 - [Part II](#)
 - [SolarMetric JDOQL manual \(PDF\)](#)

18 Hook Aspect

The Hook Aspect aims to write listeners of POJO events in simple way. The Hook Aspect allows the developer to extend an existent application in a flash. The Hook Aspect is based on the well known principles of [Aspect Oriented Programming](#), AOP by now. The AOP supports the Object oriented paradigm. The two approaches can be used together to get the best of both.

You can hook any action of any POJO at runtime as well as read/write of any fields.

Using the Hook Aspect you can change and customize your application behaviour in a *non-invasive mode*.

18.1 Scope

Each hook belong to a scope. Scope can be:

- **SESSION**, means that all the registered POJOs that are part of the current User Session will be waked up when the event is raised
- **APPLICATION**, means that the POJOs are components declared into the IoC container (Spring IoC).



Using the scope **APPLICATION** you must assure that the POJO loaded by Spring IoC use the annotation **@CoreClass(loading=EARLY)** on top of the class declaration (look at the example below). This is the only way to assure to activate the Roma configuration when the application starts.

Below a table of the hooks available for the action:

@HookAction	Description
onBeforeAction	Invoked just before an action is called
onAroundAction	Invoked in place of the original action
onAfterAction	Invoked just after an action is called
onBeforeFieldRead	Invoked just before a field is read
onAroundFieldRead	Invoked in place of the original read. The action must return the value

	to return in place of the original
onAfterFieldRead	Invoked just after a field is read
onBeforeFieldWrite	Invoked just before a field is written
onAroundFieldWrite	Invoked in place of the original write. The action must return the value to assign to the field in place of the original
onAfterFieldWrite	Invoked just after a field is written

@HookField	Description
field	Copy the content of the field hooked everytime changes

18.2 Change the behaviour at run-time

In this example you need to avoid further login into the application for a while. The classic path would be to change the code or to write a parameter to switch on/off this feature.

Using a hook makes all things much easier. Please note that the hook **hookAroundAction** means in-place-of. This action will be called in place of the original one.

ApplicationControl Example

```
@CoreClass(loading = LOADING_MODE.EARLY)
public class ApplicationControl {
    @HookAction(hookAroundAction="CustomLogin.login",
scope=HookScope.APPLICATION)
    public void hook() {
        // DISABLE THE LOGIN
        System.out.println("Application in mantainance: login disabled");
        RomaFrontend.flow().forward(
            new MessageOk("login", "Login temporary disabled").
                setMessage("Login is temporary disabled, please try in few
minutes"),
            "screen:popup:alert");
    }
}
```

In the same way you can customize strategies, algorithms, navigation paths all by using the Hook Aspect.

18.3 Collect results

This is an example of a simple profiler that record in a Map in memory the times the actions are called.

Remember that to use the scope “application” you need to instantiate the component as singleton by the IoC container (Spring IoC at this time). Add this line at the end of applicationContext.xml file of your application:

```
<bean id="ProfilerDemo" singleton="true"
      class="com.orienttechnologies.moobilis.server.view.domain.ProfilerDemo" />
```

And create a class in this way:

ProfilerDemo Example:

```
import java.util.HashMap;

import org.romaframework.aspect.hook.annotation.HookAction;
import org.romaframework.aspect.hook.annotation.HookScope;

@CoreClass(loading = LOADING_MODE.EARLY)
public class ProfilerDemo {

    protected HashMap<String, Long> totalTimes = new HashMap<String,
Long>();

    @HookAction(hookAfterAction = "*", scope = HookScope.APPLICATION)
    public void profileAllActions() {
        Long times = totalTimes.get("allActions");
        if (times == null)
            times = new Long(0);
        times = times + 1;
        totalTimes.put("allActions", times);
    }

    @HookAction(hookAfterAction = "**Login*", scope = HookScope.APPLICATION)
    public void profileAllLoginActions() {
        Long times = totalTimes.get("loginActions");
        if (times == null)
            times = new Long(0);
        times = times + 1;
        totalTimes.put("loginActions", times);
    }
}
```

Use the **early loading** mode to assure to configure the class when Roma starts

Use the wildcard to hook to multiple actions in one shot

19 Reporting Aspect

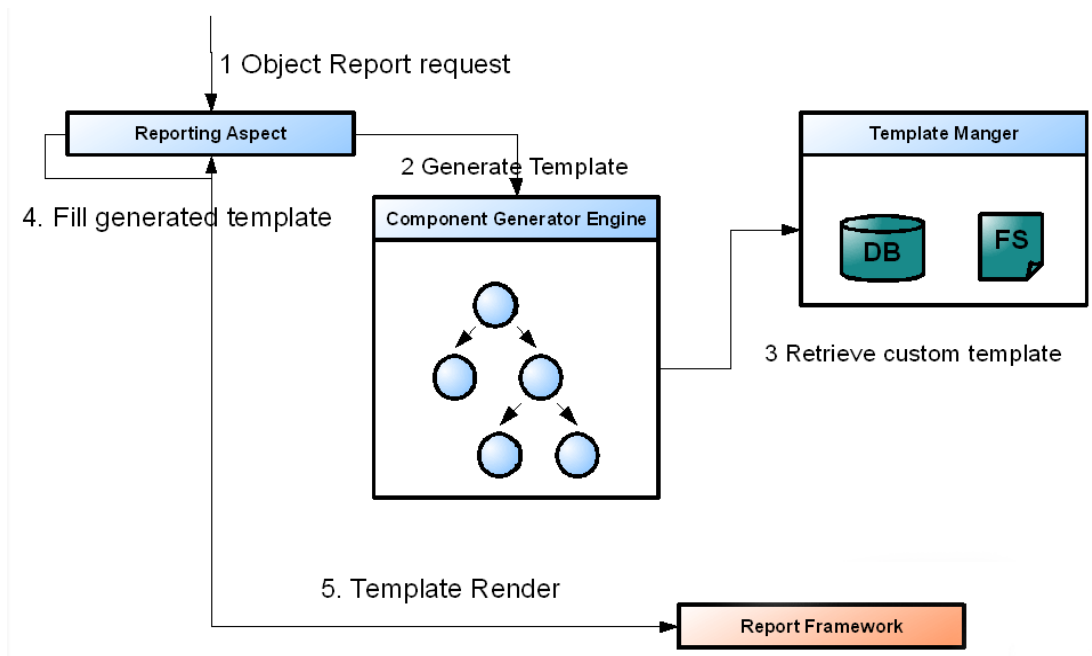
The main responsibility of the reporting aspect is to generate a report using the same annotations of the the [View Aspect](#), render it in different formats, and generate a template for the customization of the report.

```
public class ReportingAspect {  
    public String[] getSupportedTypes();  
  
    public void createTemplate(Object iExample) throws  
ReportingException;  
  
    public void createDynaTemplate(Object iExample) throws  
ReportingException;  
  
    public String getDocumentType(String renderType);  
  
    public byte[] render(Object iToRender, String iRenderType,  
SchemaObject iUserSchema);  
  
    public byte[] renderCollection(Collection<?> collection, String  
iRenderType, SchemaObject iUserSchema);  
}
```



When you have a reporting aspect module imported you will have a report button on every CRUD and every instance, that button will show what you see on the screen.

19.1 Architecture



19.2 Render Types

The [Reporting Aspect](#) supports most of the render types of the [View Aspect](#), and some of them with some limitation. That is because for the creation of a report the aspect generate first a template used by the framework and it cannot be dynamic as a view.


Other render type of the [View Aspect](#) will be changed in a simple text render because we don't need the interaction buttons as the original component.





The reporting mode as a set of annotation that are the same of the [View Aspect](#), they should be used only to make changes to the original layout, if no Reporting annotation is defined the Aspect will use the view annotation defined

In the table there is a list of supported types, the last column describe if a render needs the generation trough the the aspect of a normal template or a dynaTemplate

(See next section):

Rendering Mode	Applies to	Description	Template Type
Default	Classes, Fields, Actions	If no render is defined for a component on the report will be showed the label of the field and the toString() value of the object (Text render)	Normal
Accordion	Fields	Text render applied	Normal
Button	Actions	Nor rendered	Normal
Chart	Fields	Render the chart using the chart aspect, it may be applied to the same object types of the Char Module.	Normal
Check	Fields (Boolean)	Text render applied (shows the boolean value)	Normal
Colset	Fields (Collection)	Not supported	Normal
Date	Fields (Date)	Text render applied	Normal
Datetime	Fields	Text render applied	Normal
Html	Fields	Text render applied <div data-bbox="454 1199 1059 1367">  <p>The HTML could be rendered without conversions, it depends on the framework used for the render.</p> </div>	Normal
Image	Fields (String)	Render the image. All the images for the report must be in the package <code>{applicationPackage}.reporing.image</code>	Normal
Imagebutton	Fields	Not rendered	Normal
Label	Fields	Text render applied	Normal

Link	Fields, Actions	Not rendered	Normal
List	Fields (Collection)	Render a table with the toString() value of the contained Pojo	Normal
Menu	Classes, Fields	Text render applied	Normal
Objectlink	Fields	Text render applied	Normal
Objectembe dded	Fields	Render the object embedded <div>  <p>If an object contains itself as object embedded only tree level will be render, otherwise there StackOverflow problems</p> </div>	Normal
Password	Fields	Text render applied <div>  <p>It will show the passwords, hide it with reporting annotation</p> </div>	Normal
Popup	Classes	Default render applied	Normal
Progress	Fields (integers between 0 and 100)	Not supported yet	Normal
Radio	Fields	List render applied	Normal
Richtext	Fields (String)	Not rendered	Normal
Rowset	Fields (Collection	As for the view aspect it will expand any object in the list	Dyna

)		
Select	Fields (Collection)	Text render applied	Normal
Tab	Fields (Map)	Table render applied	Normal
Table	Fields (Collection , Maps)	Collections can be rendered as tables. Each row contains a value in the map; each column represents a property of the beans contained in the collection. It can render also Map as a table of key.toString(), value.toString();	Normal
Tableedit	Fields (Collection)	Table applied	Normal
Text	Fields (String)	This rendering mode renders an object as a text field.	Normal
Textarea	Fields (String)	Text render applied	Normal
Time	Fields	Text render applied	Normal
Tree	Fields (implements TreeNode)	Not supported	Normal
Upload	Fields (Stream)	Nor rendered	Normal

19.3 How to push a report

To create a report you only need to create an annotated pojo with all the information (Or you can use a pojo annotated for the view and modify it with reporting annotations), generate and edit the template and than push the report download in this way:

```
ReportingAspect reporting = RomaFrontend.reporting();
StaticA1Report reportPage = new StaticA1Report(instance, other, direction,
                                                testFileName, titolo);
```

```
byte[] report = reporting.render(reportPage,
                                ReportingConstants.DOCUMENT_TYPE_PDF, null);

RomaFrontend.view().pushCommand( new DownloadStreamViewCommand(
                                new ByteArrayInputStream(report),
                                FILENAME + _ + DOT +
                                ReportingConstants.DOCUMENT_TYPE_PDF,
                                ReportingConstants.DOCUMENT_TY
                                PE_PDF));
```

19.4 Template Generation

The aspect can generate a template dynamically, after the template is generated you can download it using the template manager GUI, and edit with the particular framework tool.



By convention if a template with the name of class is contained in the class package it will be used for the report generation instead to create a new one

The template generation generate a template with the class name, and a set of sub-templates named class_field, so you can customize the render of a single field.



Some render type need a **dyna template generation**, for example the **rowset** render expand any object contained in a collection, it can expand any type of object so the generation of the template need an example object to generate a template for any Class type in the list.

EmployeeMain Carica

Application Domain Only ☒

Available Class Names

AbstractDataViewListEditor
AdvancedMenu
BackOfficeMenu
Campaign
CampaignConfiguration
CampaignConfiguration.entity
CampaignFilter
CampaignFilter.entity
CampaignInstance
CampaignInstance.entity
CampaignListable
CampaignListable.entity

 Edit  Salva

19.5 Template Manager

The template manager is a GUI implemented in the reporting to manage the layout of a report at runtime.

It is composed by a Form for the selection of the class:

Upload File

Template Files	File Name
	EmployeeMain.jrxml
	EmployeeMain_filter.jrxml
	EmployeeMain_result.jrxml

Indietro

all we have to do is to select the class, save and then edit it; the edit mask shows all the generated templates, and clicking on one of them we can download it and edit we the framework tool.

19.6 Implementations

19.6.1 Jasper Reports Module

This module uses the well known Jasper Reports by JasperSoft company. It's an open source project mainly divided in two parts:

- Server side libraries, used from the module

- iReport-nb-3.5.3, a 100% Java client (runs using Swing GUI components) to edit report templates.

This module implements the [Reporting Aspect](#) interface.

19.6.2 Template Management

When a template is generated using the the Template manager it will be created form the base layout of the **Object.template.jrxml** template in the package `${applicationPackage}.reporting`.

The file **Object.subtemplate.jrxml** will be used for the generation of the sub-reports template.

So if we want to add an text logo to any report of the application we can modify the **Object.template.jrxml** and add a static text on the header section and any template generated will have the text defined

19.6.3 Jasper report tips

19.6.3.1 How can be an image added to a report?

To add an image to a report there is a trick that have to be used because when an image type is added into a template you cannot import the image serialized.

So you have to set the type as `InputStream` and set the field expression as :

```
JRDesignHelper.getImage("${logo.png}")
```



The image must be in the package `${applicationPackage}.reporting.image`

20 Security Aspect

20.1 Main concepts

The Security Aspect is intended to allow developers to define access rules on POJOs, fields, actions and events.

20.2 Annotations

Roma Security Aspect has four annotations, one for classes, one for fields, one for actions and one for events:

20.2.1 @SecurityClass

- **readRoles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to view all the fields of this class, otherwise they will be hidden
- **writeRoles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to modify all the fields of this class, otherwise they will be disabled
- **executeRoles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to execute all the actions of this class, otherwise they will be disabled

20.2.2 @SecurityField

- **readRoles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to view this field, otherwise it will be hidden
- **writeRoles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to modify this field, otherwise it will be disabled

SecurityField rules override SecurityClass rules.

20.2.3 @SecurityAction

- **roles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to execute this action, otherwise it will be disabled

SecurityAction rules override SecurityClass rules.

20.2.4 @SecurityEvent

- **roles:** accepts a String or an array of Strings; if this rule matches the user will be allowed to execute this action, otherwise it will be disabled

SecurityEvent rules override SecurityClass rules.

20.3 The Secure interface

Security annotations are good for compile-time definition of filters on attributes, but sometimes you want to define rules on objects (e.g. sometimes you want to filter POJOs in a list based on the POJO content).

To do so you can use the **Secure** interface. This interface is really very simple:

```
public interface Secure {  
    public boolean canRead();  
    public boolean canWrite();  
}
```

The **canRead()** method declares if in the current user session this POJO can be displayed or not.

The **canWrite()** method declares if in the current user session this POJO can be displayed or not.

Here is a very basic example usage of Secure interface:

```
public class Car implements Secure {  
    private String      name;  
    private BaseAccount owner;  
  
    public boolean canRead(){  
        return owner.equals(Roma.session().getActiveSessionInfo().getAccount());  
    }  
  
    public boolean canWrite(){  
        return owner.equals(Roma.session().getActiveSessionInfo().getAccount());  
    }  
}
```

In this case, instances of Car can be read and modified only by their owner.

In case you have a field that is a collection of Cars, the collection will be filtered by Roma before being displayed.

20.4 Available implementations

- [Users Security](#) in [Users module](#)

21 Logging Aspect

The logging aspect is used to log a message to an output stream, it can be configured through annotations so that a message is logged any time a method is invoked.

```
public interface LoggingAspect extends Aspect {
    public static final String ASPECT_NAME = "logging";
    public void registerLogger(Logger iLogger);
    public void removeLogger(Logger iLogger);
    public void log(int level, String category, String mode, String
message);
}
```

21.1 Example of usage

```
public class HomePage {

    @ViewField(render = "menu", layout = "screen:menu")
    protected MainMenu menu = new MainMenu();

    public HomePage() {}

    public MainMenu getMenu() {
        return menu;
    }

    @LoggingAction(level = LoggingConstants.LEVEL_INFO, category = "Test", mode =
LoggingConstants.MODE_CONSOLE, post = "@{user} entered on @{date}", enabled =
AnnotationConstants.TRUE)
    public String enter() {
        ....
    }
}
```

In the example we want to log any access to the application, so we define a message "@{user} entered on @{date}" configured to be executed after the invocation, and anytime a user invoke the we will have on the console the result:

```
level: 1
category: Test
message: admin entered on Fri Oct 10 13:00:28 CEST 2008
```



Remember that the logging aspect doesn't realize an enhancement of the classes so the log is printed only for method that are used to create the application GUI.

Or we can write the message on the i18n file properties, in that case the code will change in this way

```
@LoggingAction(level = LoggingConstants.LEVEL_INFO, category = "Test", mode =
LoggingConstants.MODE_CONSOLE, post = "@{i18n}", enabled =
AnnotationConstants.TRUE)
public String enter() {
    ....
}
```

21.2 Messages conventions

A message configured by annotation or xml as a set of conventions for the replacement of message part with values of common interest,

- **@{me}**: replaced with the value of the object where the method is invoked or the field is changed, it can use XPath syntax to retrieve values in the object for example **@{me.name}**
- **@{returned}** replaced with the value of the returned object by the method, as for **@{me}** it can use XPath syntax.
- **@{who}** replaced with the value with the information about who is logging the message, for example **[before method invocation]**
- **@{where}** replaced with the the information about where the log message was generated, for example **[PersonMain.search]**
- **@{exception}** replaced with the exception name, it can use the XPath syntax to retrieve other informations
- **@{user}** replaced with the current user name
- **@{i18n}** all the message must be composed only by this string, in this case we are configuring message as taken from the i18n file property, the key on this file for the message will be **<ClassName>.<methodName>.[pre,post,exception]**

21.3 The Loggers

The logging aspect uses a set of loggers to log the messages, this classes extend the AbstractLogger class in the Roma core package and implement the Logger interface

```
public interface Logger {  
    public String[] getModes();  
    public void print(int iLevel, String iCategory, String message);  
}
```

Were the getModes() return the list of modes managed by the logger, so can also define your custom mode in the application.

The logger implementations are added to the application through IOC, so remember to add your bean to the IOC file of the framework and to extends the AbstractLogger that contains the code to register the logger in the logging aspect.

21.4 Annotations reference

Parameter	Default	Description	Applied to
level	LEVEL_INFO	The level of the message	Field, Action
category	DEFAULT_CATEGORY	The category of the log message	Field, Action
mode	MODE_CONSOLE	The OutputStream used to save the log message	Class, Field, Action
enabled	FALSE		Class, Field, Action
pre	null	The message that is printed before a method invocation	Action
post	@{who} @{where}	The message that is printed after a method invocation or a field is changed	Action, Field
exception	@{who} @{where} @{exception}	The message printed if an exception is thrown	Action,Field

exceptionsToLog	{Throwable}	The exceptions list that generate the log message	Action,Field
-----------------	-------------	---	--------------

21.5 Implementations

21.5.1 Bundled core logging

The core module comes with a gateway implementation in bundle. This gateway redirects logging to the [ActivityLog APIs](#) of [Users Module](#) if installed or to the [Apache Commons Logging APIs](#).

22 Scripting Aspect

The Scripting Aspect's aim is to add server-side scripting capabilities to Roma applications. Server-side Scripting is becoming a very common practice in web development because it allows rapid development and quick code modifications without the need to compile and deploy.

22.1 Interface

The Scripting Aspect interface is really very simple: it just allows to evaluate scripts and to know which languages are supported by the current implementation.

```
public interface ScriptingAspect {
    public Object evaluate(String language, Reader script, Map<String, Object>
context) throws ScriptingException;

    public Object evaluate(String language, String script, Map<String, Object>
context) throws ScriptingException;

    public Object evaluate(String script, Map<String, Object> context) throws
ScriptingException;

    public List<String> getSupportedLanguages();

    public String getDefaultLanguage();
}
```

22.2 Usage

Through this interface you can evaluate a script in one of the languages supported by the implementation.

A very simple example of how to use Scripting Aspect is the following:

```
Map<String, Object> context = new HashMap<String, Object>();
context.put("age", 32);

Object scriptResult = Roma.scripting().evaluate("print($age)", context);
```

This way you execute your script in the default language, with the given input. You also can explicitly declare which is the language of the script, with the following call:

```
Object result = Roma.scripting().evaluate("JavaScript", script,
context);
```

This method can call an `UnsupportedLanguageException` if the language you declare is not supported by the aspect implementation.

As you can see, you can pass a context to provide and get back values to/from the script. Context is a simple Map of `<String, Object>`, the keys of this map can be used in the script to reference the corresponding values as variables.

22.3 Advance topics

22.3.1 Callbacks

You can listen to the events that are fired right before and right after the execution of a script by implementing the interface: `org.romaframework.aspect.scripting.ScriptingAspectListener`:

```
public interface ScriptingAspectListener {
    public Reader onBeforeExecution(String iLanguage, Reader iScript,
        Map<String, Object> iContext);

    public Object onAfterExecution(String iLanguage, Reader iScript, Map<String, Object> iContext, Object iReturnedValue);
}
```

In this example it puts the repository component that manages the employee instances in the context with name “employees” to being referenced by the script itself.

After the execution check if a null is returned. If yes return instead an empty string.

```
public class DBScriptInject implements ScriptingAspectListener {
    public Reader onBeforeExecution(String iLanguage, Reader iScript, Map<String, Object> iContext)
    {
        iContext.put("employees", Roma.component(EmployeeRepository.class));
        return iScript;
    }

    public Object onAfterExecution(String iLanguage, Reader iScript, Map<String, Object> iContext, Object iReturnedValue)
    {
        return iReturnedValue == null ? "" : iReturnedValue;
    }
}
```

22.4 Implementations

22.4.1 Java6 Scripting Module

This module leverages on the Java6 Virtual Machine to use the [JSR 223](#) scripting facilities. Java6 Virtual Machine comes also with JavaScript Rhino interpreter in bundle. [JSR 223](#) supports several languages. To know more information about [JSR 223](#) features see here: <http://scripting.dev.java.net/>.

Java6 Scripting Module comes with in bundle the following language bindings:

- **JavaScript** using Java6 Rhino interpreter
- **Java** allowing to execute Java code loaded at run-time

This module implements the [Scripting Aspect](#) interface.

23 Monitoring Aspect

Monitoring aspect allow business object to be monitorable externally to the application using the [JMX](#) standard. With the JMX Monitoring Aspect, by example, you can access to your business object from your Application Server JMX console.

23.1 Implementations

Monitoring-jmx and monitoring-mx4j.

24 Semantic Aspect

24.1 Introduction

The Semantic Aspect arises from the collaboration of Roma Meta-Framework development team in Romulus project (<http://www.ict-romulus.eu>).



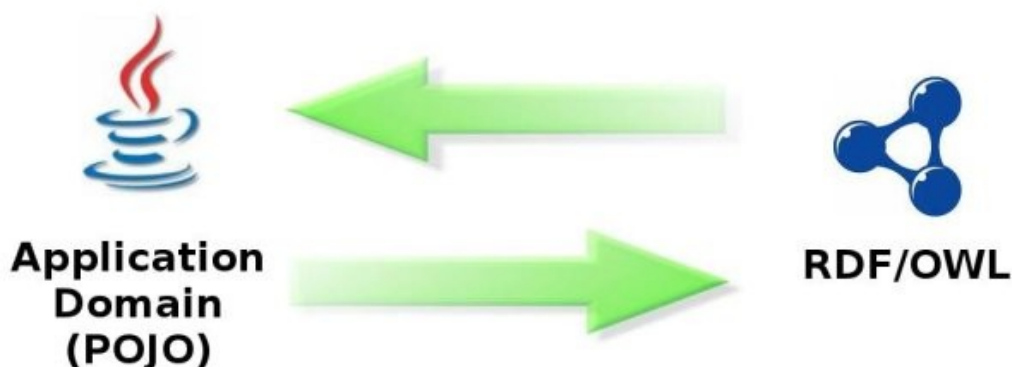
WHEN WRITING THIS DOCUMENT, THE SEMANTIC ASPECT INTERFACE IS STILL UNDER DEFINITION, SO IT MAY CHANGE IN THE FUTURE REVISIONS.

24.2 The Semantic Aspect purpose

Semantic Aspect aims to integrate POJO based applications with Web 3.0 (semantic) technologies. As defined in Romulus analysis documents, its functionalities can be resumed as follows:

1. Semantic content generation: Generating semantic representation (RDF) and ontologies (RDFS/OWL) from POJOs. This is the main functionality of this module.
2. Semantic queries on the system: The module should support querying the system with semantic web query languages, such as SPARQL, to retrieve information directly from the relational database of the application. These queries should be performed taking into account the semantic representation of the application domain, as defined in Semantic content generation.
3. Automatic Domain generation from existing ontology: The module should provide facilities to automatically generate Java classes from ontologies.

Some of these (mainly point 1) are strictly required from the implementation, others are desiderata that are not yet been analysed in detail, so the aspect definition can change in the future in case that some of these are intrinsically not achievable.



24.3 Interface

24.3.1 Annotations

A POJO, to be rendered as semantic information, needs some additional data that can be expressed with Java annotations or XML annotations.

24.3.1.1 SemanticClass Annotation

This annotation declares that a instances of a Class can be represented as subjects and/or objects in semantic information. It contains the following data:

- **subjectPrefix:** the prefix of the URI that identifies instances of this class
- **subjectId:** the name of the property that contains the suffix of the URI of instances of this class.

If subjectId and subjectPrefix are not declared, instances of this class will be represented as anonymous subjects. If they are declared, the URI of the POJO will be calculated as the concatenation of the **subjectPrefix** and the value of the object's property named as **subjectId** value.

24.3.1.2 SemanticField Annotation

A field of a POJO represents a relation between the POJO and another object, so, in semantic representation a field represents a predicate. SemanticField annotation contains the following data:

- **predicate:** the URI of the predicate that corresponds to the field.

a SemanticField defines a triple where the subject is the the POJO representation as defined by SemanticClass, the predicate is the one defined in the predicate attribute and the object is the value of the field (that can be the semantic URI of the

value if it has a `SemanticClass` annotation or a `String` representation of the object).

24.3.1.3 XML Annotations

XML annotations can be used instead of Java annotations. Roma priority rules state that if you write both annotations and XML annotations for a class or a field, the XML annotations are taken into consideration.

XML annotation names of attributes are the same as annotation attributes.

24.3.1.4 SemanticAspect Interface

This class is the access point of the `SemanticAspect`; it contains methods to

- transform POJOs (and all the tree of objects connected through `SemanticField` relations) in semantic representation
- transform POJOs in ontologies
- create `SemanticModel` instances
- transform semantic query (SPARQL or other languages) in a format that can be executed by `Roma PersistenceAspect`

For a complete description of its methods please refer to Javadoc.

24.3.1.5 SemanticModel Interface

This class describes a semantic model. It can be used to incrementally create a semantic representation, adding it POJOs.

Its main methods are:

- `addObject(Object pojo)`: adds semantic representation of a POJO to the model. When adding a POJO to the model, all the tree of objects connected through `SemanticField` relations is imported.
- `getRepresentation()`: returns a semantic representation (`String`) of the current model
- `getUnderlyingImplementation()`: returns an implementation-specific instance of the model

24.3.1.6 SemanticConstants class

This class contains some constants containing well known URIs and can be used in `SemanticField` annotations to define predicates (VCARD, RSS etc.).

For a complete description of this class please refer to Javadoc.

24.3.2 Example of usage

Here we describe with some code examples how to use Roma Semantic Aspect to generate RDF content.

Suppose your application contains employees personal data and it is represented with the Person class.

Person.java

```
public class Person {
    private String personId;
    private Person manager;
    private String fullName;
    private String email;
    private String role;

    public Person(String personId) {
        super();
        this.personId = personId;
    }

    //add here getters and setters
}
```

Your boss asks you to publish employee information on the web as RDF. With Roma SemanticAspect you just have to add Semantic* annotations to your Person class:

```
@SemanticClass(subjectPrefix = "http://www.mycompany.com/personnel/",
subjectId="personId")
public class Person {
    private String personId;

    @SemanticField(predicate = "http://www.mycompany.com/semantic/manager")
    private Person manager;

    @SemanticField(predicate = SemanticConstants.VCARD_FN)
    private String fullName;
    @SemanticField(predicate = SemanticConstants.VCARD_EMAIL)
    private String email;
    @SemanticField(predicate = SemanticConstants.VCARD_ROLE)
    private String role;

    public Person(String personId) {
        super();
        this.personId = personId;
    }

    //add here getters and setters
}
```

And then use SemanticAspect to translate POJOs in RDF:

```

Person luca = new Person("LGA");
luca.setFullName("Luca Garulli");
luca.setEmail("luca.garulli@assetdata.it");
luca.setRole("CTO");

Person luigi = new Person("LDE");
luigi.setFullName("Luigi Dell'Aquila");
luigi.setEmail("luigi.dellaquila@assetdata.it");
luigi.setRole("Software Architect");
luigi.setManager(luca);

SemanticAspect sa = Roma.component(SemanticAspect.class);
String result = sa.getSemanticRepresentation(luigi);
System.out.println(result);

```

The result is the printing on the “out” stream of the following XML:

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:j.0="http://www.mycompany.com/semantic/"
  xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#">
  <rdf:Description rdf:about="http://www.mycompany.com/personnel/LDE">
    <vcard:FN>Luigi Dell'Aquila</vcard:FN>
    <vcard:ROLE>Software Architect</vcard:ROLE>
    <vcard:EMAIL>luigi.dellaquila@assetdata.it</vcard:EMAIL>
    <j.0:manager>
      <rdf:Description rdf:about="http://www.mycompany.com/personnel/LGA">
        <vcard:FN>Luca Garulli</vcard:FN>
        <vcard:ROLE>CTO</vcard:ROLE>
        <vcard:EMAIL>luca.garulli@assetdata.it</vcard:EMAIL>
      </rdf:Description>
    </j.0:manager>
  </rdf:Description>
</rdf:RDF>

```

Sometimes you are asked to create this functionality without modifying existing code, so you cannot add annotations to Person class. In this case you can use XML annotations instead of classic Java annotations: you just have to put in the same package of Person.java a file named Person.xml containing the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<class xmlns="http://www.romaframework.org/xml/roma"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
  xsd:schemaLocation="http://www.romaframework.org/xml/roma
http://www.romaframework.org/schema/v2/roma.xsd">
  <aspects>
    <semantic subjectPrefix="http://www.mycompany.com/personnel/"
      subjectId="personId" />
  </aspects>
  <fields>
    <field name="manager">
      <aspects>
        <semantic

```

```

        predicate="http://www.mycompany.com/semantic/manager" />
    </aspects>
</field>
<field name="fullName">
    <aspects>
        <semantic
            predicate="http://www.w3.org/2001/vcard-rdf/3.0#FN" />
        </aspects>
    </field>
<field name="email">
    <aspects>
        <semantic
            predicate="http://www.w3.org/2001/vcard-rdf/3.0#EMAIL" />
        </aspects>
    </field>
<field name="role">
    <aspects>
        <semantic
            predicate="http://www.w3.org/2001/vcard-rdf/3.0#ROLE" />
        </aspects>
    </field>
</fields>
</class>

```

Obviously, when writing XML annotations you cannot use SemanticConstants class to declare predicates, so you have to write full predicate URIs.

24.4 Implementations

24.4.1 Semantic-Jena

An implementation based on Jena Semantic Web Framework (<http://jena.sourceforge.net/>) is under development. Currently this implementation only supports RDF and N-TRIPLE generation from POJOs. It does not yet support query translation and ontology generation.

This implementation is being developed under the Romulus Project (<http://www.ict-romulus.eu>), so you can find it at the following SVN:

<https://ict-romulus.svn.sourceforge.net/svnroot/ict-romulus/trunk/roma-modules/semantic-jena>

25 Workflow Aspect

The workflow aspect provide a generic interface for the interaction with a workflow engine.

```
public interface WorkflowModule extends WorkflowModuleLocal,
WorkflowModuleService {
    public String getEngineName();
    public String getEngineVersion();
    public String getEngineDescription();
    public long[] getWorkflowInstances();
    public long[] getRunningWorkflowInstances();
    public long[] getPausedWorkflowInstances();
    public long startWorkflowDefinition(String iDefinitionName, Map<String,
Object> iParameters) throws WorkflowException;

    public Map<String, String> getWorkflowDefinitionParameters(String
iDefinition);

    public void stopWorkflowInstance(long iWorkflowInstanceID) throws
WorkflowException;

    public void pauseWorkflowInstance(long iWorkflowInstanceID) throws
WorkflowException;

    public void resumeWorkflowInstance(long iWorkflowInstanceID) throws
WorkflowException;

    public void signalActivityCompleted(long iActivityInstanceID, Map<String,
Object> iOutputValues, String exitCode)
        throws WorkflowException;

    public Statuses getWorkflowInstanceStatus(long iWorkflowInstanceID) throws
WorkflowException;

    public long[] getActivityInstances(long iWorkflowInstanceID) throws
WorkflowException;

    public long[] getWorkflowInstances(String iDefinitionName);

    public String getWorkflowDefinitionNameByWorkflowInstance(long
iWorkflowInstanceID);

    public Object getEngineScopeParameter(String paramName) throws
WorkflowException;

    public void setEngineScopeParameter(String paramName, Object value) throws
WorkflowException;

    public Object getDefinitionScopeParameter(String definitionName, String
paramName) throws WorkflowException;

    public void setDefinitionScopeParameter(String definitionName, String
paramName, Object value) throws WorkflowException;

    public Set<String> getEngineScopeParameterNames() throws WorkflowException;

    public Set<String> getDefinitionScopeParameterNames(String definitionName)
```

```

throws WorkflowException;

    public Object getInstanceScopeParameter(long workflowInstanceId, String
paramName) throws WorkflowException;

    public void setInstanceScopeParameter(long workflowInstanceId, String
paramName, Object value) throws WorkflowException;

    public Set<String> getInstanceScopeParameterNames(long workflowInstanceId)
throws WorkflowException;

    public void startup();

    public void shutdown();

    public abstract void signalActivityCompleted(long iActivityId,
Map<String, Object> iOutputValues, String exitCode,
TransactionPartecipant partecipant) throws
WorkflowException;
}

```

25.1 Implementations

25.1.1 Tevere Flow module

[Tevere Flow](#) is an Open source project released with Apache 2.0 license. Tevere Flow is a workflow engine based on the command Pattern with a 2.0 web GUI. Go to [Tevere Flow documentation site](#) to know more.

This module implements the [Workflow Aspect](#) interface.

26 ETL

26.1 What is ETL

Writing a new application often requires importing data from existing data sources. These data sometimes come from another application, and the original format is different from the one of the new application domain.

To make the import procedure automatic, we designed a module that provides the infrastructure to extract data from an external data source, transform them and import them in the new application.

The name ETL was born in the field of data warehousing, where multiple relational data sources have to be merged and rearranged to populate a data mart, that often is not properly relational.

ETL stands for:

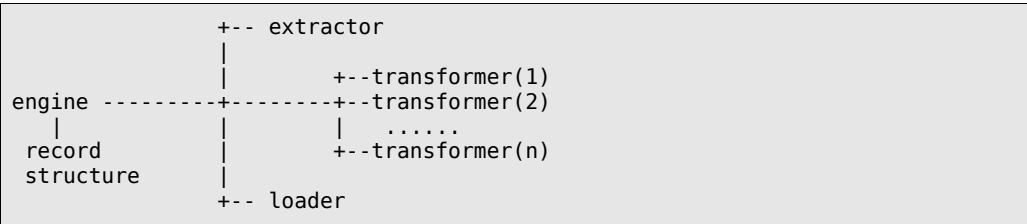
- Extraction: the operation of retrieving data from existent data sources
- Transformation: the operation of filtering and transforming data to make them compatible with our needs
- Loading: the operation of inserting data in our application.

26.2 Etl Aspect architecture

Our ETL architecture relies on a tree-step architecture based on extractors, transformers and loaders and on an engine that co-ordinates these three operations.

Every ETL procedure is described through an XML file where we declare which is the extractor implementation to use (and its configuration), what are the transformations to be executed on the extracted data and the order they have to be executed, and what is the loader implementation that will make data loading in Roma POJO's.

The structure may be represented as follows :



The record structure is defined in the descriptor and allows the user to configure the association between extracted data and elements of our application domain.

each module of the procedure must implement a defined interface that allows it to communicate with the other modules.

The ETL execution proceeds as follows:

- the engine asks the extractor to extract one element in the original data source
- the extractor returns a record containing the extracted data
- the engine invokes the first transformer passing it the record.
- the transformer transforms the record
- the engine proceeds invoking each transformer in the defined order
- when all the transformations have been executed the engine gives the record to the loader that uses it to populate our application domain
- the engine proceeds asking the extractor for the next record and the operation continues until there are no more data to be extracted

26.3 Defining a custom ETL procedure

26.3.1 The ETL descriptor

An ETL descriptor is a structure that contains information about an ETL procedure. In our implementation the `EtlDefinition` POJO contains data about the ETL like its name, description and associated ETL procedure.

The ETL descriptor is a Spring xml file that represents an `EtlDefinition`.

26.3.2 The Record

A record contains information about an extracted data element and is modified during the ETL project to adapt data structure for loading. The record also contains the rules to bind extracted data to attributes of data objects to be loaded.

During the ETL procedure only one record should exist, and should be cleaned and recycled at each iteration (for efficiency purposes).

In ETL configuration we have to configure the record defining the data structure that will bring data from extraction to loading and the binding rules.

A record is made of record fields that represent single data attributes. A record field contains essential information for ETL procedures, in particular:

- the "name" attribute, that indicates the name of the attribute and is normally referred by the extractor
- the "bind" attribute, that indicates the name of the property of the destination object
- the "currentValue" attribute, that contains the data extracted and transformed during the procedure

A very simple example is given here. In this example we suppose we want to extract employees data from a dataset that contains elements with two properties, called respectively "extractedName", "extractedSurname" and "extractedDepartment" and that we want to insert them directly in a Employee object with "name", "surname" and "department" attributes.

```
<bean class="org.romaframework.aspect.etl.imp.Record" id="record">
  <property name="fields">
    <list>
      <bean class="org.romaframework.aspect.etl.imp.RecordField" id="field1">
        <property name="name">
          <value>extractedName</value>
        </property>
        <property name="bind">
          <value>name</value>
        </property>
      </bean>
      <bean class="org.romaframework.aspect.etl.imp.RecordField" id="field2">
        <property name="name">
          <value>extractedSurname</value>
        </property>
        <property name="bind">
          <value>surname</value>
        </property>
      </bean>
      <bean class="org.romaframework.aspect.etl.imp.RecordField" id="field3">
        <property name="name">
          <value>extractedDepartment</value>
        </property>
        <property name="bind">
          <value>department</value>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

26.3.3 defining extraction strategy

An extractor must implement the EtlExtractor interface:


```
public interface EtlExtractor extends Enumeration<Record>{
    public void endExtraction();
    public void beginExtraction(CommandContext context);
    public boolean hasMoreElements() throws IllegalStateException;
    public Record nextElement() throws IllegalStateException;
    public Record getRecord();
    public void setRecord(Record record);
}
```

The method:

```
public void beginExtraction(CommandContext context);
```

Is invoked by the ETL engine before starting the procedure.

The method :

```
public void endExtraction();
```

Is invoked by the etl engine before closing the procedure.

These two methods should respectively open and close the datasource.

The methods:

```
public boolean hasMoreElements() throws IllegalStateException;
public Record nextElement() throws IllegalStateException;
```

Are used to extract all the data elements in the original data source.

To define an extraction strategy we have to implement this interface. Suppose we want to extract data from a CSV file: we have to implement a new class like the following:

```
package org.test;

public class CSVEtlExtractor implements EtlExtractor{

    private String fileName;
    private List<String> fieldNames=new ArrayList<String>();
    private String separator=",";
    private String stringDelimiter="\\";

    Record record;

    public void beginExtraction(CommandContext context) {
        // open input stream from file
    }

    public void endExtraction() {
        //close input stream
    }
}
```

```

}

public boolean hasMoreElements() throws IllegalStateException{
    //check if the stream has more rows
}

public Record nextElement() throws IllegalStateException{
    //read next row
    //parse the row
    //populate the record return record;
}

public void setRecord(Record record){
}
//add here getters and setters for private fields
}

```

Now we have to add an instance of this class to the descriptor.

We have to add to the descriptor the bean representing an instance of our extractor:

```

<bean class="org.test.CSVEtlExtractor">
    <property name="fileName" value="myFile.csv"/>
    <property name="separator" value=","/>
    <property name="stringDelimiter" value="'"'/>
    <property name="skipRows" value="1"/>
    <property name="record">
        <ref bean="record"/> <!-- this refers to the record previously described
-->
    </property>
    <property name="fieldNames">
        <list>
            <value>extractedSurname</value>
            <value>extractedName</value>
            <value>extractedDepartment</value>
        </list>
    </property>
</bean>

```



For common data source types (CSV and JDBC) we provided extractor implementations in Roma Framework ETL module, so you don't have to re-implement these extractors.

26.3.4 defining transformation strategy

A transformer must implement the EtlTransformer interface:

```

public interface EtlTransformer {
    public void transform(Record record, CommandContext context);
}

```

In an ETL procedure can define zero or more transformers that are invoked in the order they are declared.

The two parameters are respectively the record that has to be modified by the transformer and a context that can contain custom data useful for various purposes (e.g. the `EtlLoaderImpl` class adds to the context a persistence aspect referring to the destination persistence layer).

The ETL engine invokes the transform record passing it the record extracted by the extractor and modified by previous transformers. The transformer has to modify that record according to its purpose.

To add a transformer to the procedure we have to add it to the "transformers" list of the ETL procedure (next described).

For this example we present a very simple transformer, that transforms a department name in a `Department` object:

```
public class EtlTransformerString2Department implements EtlFieldTransformer{
    public void transform(Record record, CommandContext context){
        if(value==null) return null;
        String departmentName=getValueFromBinding(record, "department");
        PersistenceAspect db = (PersistenceAspect)
            context.getParameter(EtlContextConstants.PERSISTENCE_ASPECT);
        Department dept=loadFromDb(departmentName, db);
        if (dept==null) dept=new Department();
        return dept;
    }

    private String getValueFromBinding(Record record, String binding){
        List<RecordField> fields=record.getFields();
        for(RecordField field:fields){
            if(binding.equals(field.bind) return field.value;
        }
        //if not found return default!
    }

    private Department loadFromDb(String departmentName, PersistenceAspect db){
        //load from db the Department corresponding to this name (if it exists)
    }
}
```

26.3.5 Defining load procedure

A loader must implement the `EtlLoader` interface:

```
public interface EtlLoader {
    public void loadRecord(Record record);
    public void beginLoading(CommandContext context);
    public void endLoading();
    public void setFactory(EtlObjectFactory factory);
    public EtlObjectFactory getFactory();
}
```

A simple `EtlLoader` implementation (`EtlLoaderImpl`) is provided by `RomaFramework` ETL module. This implementation just needs an object factory that returns instances of the objects to be loaded. The loader then, starting from the bindings defined in the record, assigns the values extracted to the attributes of the object.

Here we report a simple `EtlLoader` definition in the descriptor. The `EtlSimpleObjectFactory` is a factory implementation that just instantiates an object of the class given as "className" attribute:

```
<bean class="org.romaframework.aspect.etl.loader.impl.EtlLoaderImpl"
  id="loader">
  <property name="factory">
    <bean
      class="org.romaframework.aspect.etl.objectfactory.EtlSimpleObjectFactory"
      id="factory">
      <property name="className" value="org.test.etl.domain.Employee" />
    </bean>
  </property>
</bean>
```

26.3.6 The import procedure description

Let's put the pieces together: an ETL descriptor is made this way:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-lazy-init="true" default-autowire="no">
  <bean class="org.romaframework.aspect.etl.definition.EtlDefinition"
    id="definition">
<!-- the id MUST be "definition"-->
    <property name="name" value="myEtl" />
    <property name="description">
      <value>My ETL description</value>
    </property>

    <property name="importProcedure">
      <bean
        class="org.romaframework.aspect.etl.procedure.EtlProcedure"
        id="procedure">
        <property name="extractor">
          <bean class="org.test.CSVetlExtractor">
            <property name="fileName" value="myFile.csv" />
            <property name="separator" value="," />
            <property name="stringDelimiter" value="'" />
            <property name="skipRows" value="1" />
            <property name="record">
              <ref bean="record" />
            </property>
          </bean>
        </property>
        <property name="fieldNames">
          <list>
            <value>extractedSurname</value>
            <value>extractedName</value>
            <value>extractedDepartment</value>
          </list>
        </property>
      </bean>
    </property>
  </bean>
```

```

    <property name="transformers">
      <list>
        <bean id="transformer1"
          class="org.test.etl.transformers.EtlTransformerString2Department" />
      </list>
    </property>

    <property name="loader">
      <bean
        class="org.romaframework.aspect.etl.loader.impl.EtlLoaderImpl"
        id="loader">
        <property name="factory">
          <bean id="factory"
            class="org.romaframework.aspect.etl.objectfactory.EtlSimpleObjectFacto
ry">
            <property name="className"
              value="org.test.etl.domain.Employee" />
          </bean>
        </property>
      </bean>
    </property>
  </bean>
</property>
</bean>

<bean class="org.romaframework.aspect.etl.imp.Record" id="record">
  <property name="fields">
    <list>
      <bean
        class="org.romaframework.aspect.etl.imp.RecordField" id="fiel1">
        <property name="name" value="extractedName" />
        <property name="bind" value="name" />
      </bean>
      <bean
        class="org.romaframework.aspect.etl.imp.RecordField" id="field2">
        <property name="name" value="extractedSurname" />
        <property name="bind" value="surname" />
      </bean>
      <bean
        class="org.romaframework.aspect.etl.imp.RecordField" id="field3">
        <property name="name" value="extractedDepartment" />
        <property name="bind" value="department" />
      </bean>
    </list>
  </property>
</bean>
</beans>

```

Now to invoke this procedure we just have to pass this descriptor to a Spring XmlBeanFactory, retrieve the EtlDefinition and launch the procedure:

```

BeanFactory factory = new XmlBeanFactory(new FileSystemResource(fileName));
EtlDefinition definition=(EtlDefinition)factory.getBean("definition");
definition.getImportProcedure().doImport();

```

26.4 A custom ETL: from JDBC to JDBC

This example demonstrates how to write a simple ETL procedure that extracts data from a JDBC source and inserts them in a different JDBC database. The XML contains a lot of comments that should be clear enough to explain how it works.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-lazy-init="true" default-autowire="no">
  <bean class="org.romaframework.aspect.etl.definition.EtlDefinition"
    id="definition">
    <property name="description">
      <!-- description of ETL procedure (plain text) -->
      <value>extract from JDBC and insert to JDBC</value>
    </property>
    <property name="importProcedure">
      <ref bean="bean0" />
    </property>
    <property name="name">
      <!-- the name used by EtlAspect to reference this ETL procedure -->
      <value>jdbc2jdbcExample</value>
    </property>
  </bean>
  <bean class="org.romaframework.aspect.etl.imp.Record" id="bean2">
    <property name="fields">
      <list>
        <bean
          class="org.romaframework.aspect.etl.imp.RecordField" id="bean3">
          <property name="name">
            <!-- name of the column on the source table,
              as in fieldNames of the extractor -->
            <value>name</value>
          </property>
          <property name="bind">
            <!-- name of the column on the destination table
              (if null, this field is not inserted in the destination)-->
            <value>nome</value>
          </property>
        </bean>
        <bean
          class="org.romaframework.aspect.etl.imp.RecordField" id="bean4">
          <property name="name">
            <!-- name of the column on the source table,
              as in fieldNames of the extractor -->
            <value>surname</value>
          </property>
          <property name="bind">
            <!-- name of the column on the destination table
              (if null, this field is not inserted in the destination) -->
            <value>cognome</value>
          </property>
        </bean>
      </list>
    </property>
  </bean>
  <bean class="org.romaframework.aspect.etl.procedure.EtlProcedure"
    id="bean0">

    <!-- EXTRACTOR: extracts data from a source -->
```

```

<property name="extractor">
  <bean
class="org.romaframework.aspect.etl.domain.extractor.impl.jdbc.JdbcEtlExtract
or"
    id="bean1">
      <property name="driver"><!-- driver for source db connection -->
        <value>net.sourceforge.jtds.jdbc.Driver</value>
      </property>
      <property name="connectionURL"><!-- url for source db connection -->
        <value>jdbc:jtds:sqlserver://localhost:1433;DatabaseName=dbName</value>
      </property>
      <property name="username">
        <value>SqlServerUser</value><!-- username for source db connection -->
      </property>
      <property name="password">
        <value>SqlServerPassword</value><!-- password for source db connection
-->
      </property>
      <property name="query">
        <!-- query to be executed to extract data -->
        <value>select name, surname from employee</value>
      </property>
      <!-- fields to be read from query result and to be inserted in the record
("name" in RecordField)-->
      <property name="fieldNames">
        <list>
          <value>name</value>
          <value>surname</value>
        </list>
      </property>

      <property name="record">
        <ref bean="bean2" />
      </property>
    </bean>
  </property>
  <!-- TRANSFORMERS: transform each record extracted from source
(they are executed in the order they are declared)-->
  <property name="transformers">
    <list>
      <bean
class="org.romaframework.aspect.etl.domain.transformer.impl.script.EtlTransfor
merScriptInterpreter">
        <property name="script"><!-- a simple script, transforms the name to
lower case -->
          <value>
            importClass(org.romaframework.aspect.etl.imp.RecordField);
            var field=record.getFieldById("name");
            if(field.currentValue != null){
              field.currentValue =
                field.currentValue.toLowerCase(); }
          </value>
        </property>
      </bean>
    </list>
  </property>
  <!-- LOADER: inserts data in the destination -->
  <property name="loader">
    <bean
class="org.romaframework.aspect.etl.loader.jdbc.EtlJdbcLoaderImpl"
id="bean5">

```

```

    <property name="driver">
      <value>com.mysql.jdbc.Driver</value><!-- driver for destination db
connection -->
    </property>
    <property name="connectionURL">
      <!-- url for destination db connection -->
      <value>jdbc:mysql://localhost:3306/mySqlDbName</value>
    </property>
    <property name="username">
      <value>MySQLUser</value><!-- username for destination db connection -->
    </property>
    <property name="password">
      <value>MySQLPassword</value><!-- password for destination db connection
-->
    </property>
    <property name="tableName">
      <value>impiegato</value><!-- name of table where to execute updates or
inserts -->
    </property>
    <property name="atomic">
      <!-- if false each record is committed separately,
      else all the inserts and updates are made in the same transaction -->
      <value>true</value>
    </property>

    <!-- these fields are used as a filter for updates:
      if this list is empty, the loader executes an "insert";
      if this list contains at least one element,
      the loader performs a "select" using as filter the
      columns indicated in this list, with corresponding
      values in the corresponding record fields;
      if the query returns at least one result,
      an update is executed using the same filter,
      otherwise an insert is executed
    -->
    <property name="searchKeysForUpdate">
      <list>
        <value>cognome</value><!-- as in "bind" of record field -->
      </list>
    </property>
  </bean>
</property>
</bean>
</beans>

```

26.5 Validating imported data

TODO

27 Session Aspect

The session aspect handles user sessions in the application. In the user session the application can save user or navigation information under the form or POJOs.

This aspect is really very simple, it has some methods to get and disable the current session, to set and retrieve properties in the current session and to get the current locale.

27.1 Interface

Here we report the Session Aspect interface. For a full reference please read the Javadoc.

```
public interface SessionAspect extends Aspect {

    public SessionInfo getSession(Object iSystemSession);
    public SessionInfo addSession(Object iSession);
    public SessionInfo removeSession(Object iSession);
    public void destroyCurrentSession();
    public void destroyCurrentSession(Object iSystemSession);

    public SessionInfo getActiveSessionInfo();
    public Collection<SessionInfo> getSessionInfos();

    public Locale getActiveLocale();

    public Object getProperty(Object iSession, String iKey);
    public Object getProperty(String iKey);
    public void setProperty(Object iSession, String iKey, Object iValue);
    public void setProperty(String iKey, Object iValue);
}
```

27.2 Usage Example

In your application you can obtain an instance of the Session Aspect writing the following code:

```
SessionAspect sessionAspect = Roma.session();
```

The typical operations that you will perform on the Session Aspect will be to set and retrieve session properties. These properties will be kept in the session until the users logs off. The session scope is similar to the one of HttpSession.

To set a property in the current user session you have to write the following code:

```
sessionAspect.setProperty("myProperty", myPojo);
```

and to retrieve the property write the following:

```
MyClass myPojo = (MyClass)sessionAspect.getProperty("myProperty");
```

To disable the current session you can write:

```
sessionAspect.destroyCurrentSession();
```

This way all the properties that were added to the session will be lost.

You can also access the underlying implementation of the session (i.e. an `HttpSession`) with the following code:

```
SessionInfo romaSession = sessionAspect.getActiveSessionInfo();  
HttpSession httpSession = (HttpSession)romaSession.getSession();
```

Remember that the system session type depends on the aspect implementation.

27.3 Get current logged user account

To get the active session use:

```
Object account = Roma.session().getSessionInfo().getAccount();
```

If you're using Roma [Users](#) module you can cast the account object to the `BaseAccount` class:

```
BaseAccount account = (BaseAccount) currAccount;  
String userName = account.getName();
```

In just one line:

```
String name = ((BaseAccount) Roma.session().getSessionInfo().  
              getAccount()).getName();
```

27.4 Implementations

Currently there are two known implementations of the Session Aspect: one is contained in the [View-Echo2](#) module and is based on Echo2 session; the other one is contained in the [View-Janiculum](#) aspect and is based on HttpSession.

28 Scheduler

The goal of the Scheduler Aspect is to manage background tasks, which are fired by the application, based on rules defined by user. The scheduler aspect is a powerful tool to write and manage a new job to schedule.

The scheduler Aspect behaviour is defined by the interface `org.romaframework.aspect.scheduler.SchedulerAspect`:

```
public interface SchedulerAspect extends Aspect {

    public static final String NAME = "scheduler";

    public static final String EVENT_STATUS_BLOCKED = "Blocked";
    public static final String EVENT_STATUS_ERROR = "Error";
    public static final String EVENT_STATUS_COMPLETED = "Completed";
    public static final String EVENT_STATUS_PAUSED = "Paused";
    public static final String EVENT_STATUS_NORMAL = "Normal";
    public static final String EVENT_STATUS_NOT_SCHEDULED = "Not Scheduled";
    public static final String EVENT_STATUS_EXECUTING = "Executing";

    public SchedulerEvent createEvent() throws SchedulerException;

    public Date schedule(SchedulerEvent iTrigger) throws SchedulerException;

    public void unSchedule(SchedulerEvent iTrigger) throws SchedulerException;

    public void executeNow(SchedulerEvent iTrigger) throws SchedulerException;

    public void pauseJob(SchedulerEvent iTrigger) throws SchedulerException;

    public void unpauseJob(SchedulerEvent iTrigger) throws SchedulerException;

    public String getLastEventExcecution(String name);

    public Date getNextEventExcecution(String name);

    public void setJobsImplementation(Map<String, Command> jobsImplementation);

    public Map<String, Command> getJobsImplementation();

    public List<String> getImplementationsList();

    public String getEventState(String name);

    public Object getUnderlyingComponent();
}
```

28.1 Write a Job

Now we can write our first job to be scheduled. The class, representing the job, has to implement the `org.romaframework.core.command.Command` interface class, as in

the following example:

```
package it.assetdata.test.scheduler;

import java.util.Date;
import org.romaframework.core.command.Command;
import org.romaframework.core.command.CommandContext;

public class FirstJob implements Command {

    public Object execute(CommandContext iContext) {

        System.out.println("Job instance executed at " + (new Date()));
        return null;
    }
}
```

To put into practice, all you have to do is to fill the "execute" method, generated by IDE, with your business logic; you can write everything you want: change status to objects, verify expiry dates, access and write into a database. Also you can pass optional values at runtime, through the CommandContext input parameter. At the end of execution, the method can return a generic object, but most of cases it will return null value. In our example, the job simply writes date and time of its execution on console output and return null.

As last step, you have to tell to Scheduler Aspect that a new job is available to be scheduled; so you have to modify the "applicationContext-scheduler-<impl>.xml" where <impl> is the Scheduler Aspect Implementation. In this example we always will use [Quartz](#), so the file is: "applicationContext-scheduler-quartz.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="FirstJob" class="it.assetdata.test.scheduler.FirstJob"
        singleton="true">
    </bean>

    <!--Scheduler application context -->
    <bean id="SchedulerAspect"
        class="org.romaframework.module.schedulerquartz.QuartzSchedulerAspect"
        singleton="true">
        <property name="jobsImplementation">
            <map>
                <entry>
                    <key><value>First Job</value></key>
                    <ref bean="FirstJob"/>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

As you can see, two new items have been added:

- a bean definition with its id and implementation class
- a new element, with key and reference to new bean, into the map managed by Scheduler.

28.2 Scheduler Control Panel

At this point, the Scheduler Aspect knows which jobs can be scheduled; now a way to create instances of jobs with their own firing rules is required. Roma Framework helps you in this task, offering a scheduler control panel; you can call it from your application menu. To obtain it, you have to add the following piece of code in the MainMenu class or in any other class that implements a submenu:

```
public void scheduler() {
    RomaFrontend.flow().forward(new QuartzSchedulerEventMain());
}
```

In the following image, you can see a Scheduler Control Panel example:

Name	Rule	Implementation	Context	Status	Start Time	Last Execution	Next Execution
Time Marker	0 52 21 ? * *	First Job	{}	Normal	07/12/2009 21:50:22	08/12/2009 11:18:52	08/12/2009 21:52:00
Second Instance	0 0 14 ? * TUE,WED,THU	First Job	{}	Normal	08/12/2009 16:00:45	Not yet executed	09/12/2009 14:00:00

You can make search of scheduled job instances by name, stop and resume one of them, run an immediate shot, create or delete a job instance, observe the status of running ones.

28.3 Creating a new job instance

Starting from Scheduler Control Panel, clicking on “new” button, you can access to job instance creation panel:

Back Office Scheduler Home Page Change Password Logout

Name Start Time 08/12/2009 Implementations First Job

Date rule Cron expression

Day	Selected
Sunday	<input type="checkbox"/>
Monday	<input type="checkbox"/>
Tuesday	<input type="checkbox"/>
Wednesday	<input type="checkbox"/>
Thursday	<input type="checkbox"/>
Friday	<input type="checkbox"/>
Saturday	<input type="checkbox"/>

Time 0 : 0 : 0

Context

Key Value

You can select the job implementation from the list of available ones, define a name for the instance and the starting time. The panel offers a standard helpful tab to define basic firing rules; if you need to define more complex rules, you can select “cron expression” tab and write your own rules, according to the syntax.

Back Office Scheduler Home Page Change Password Logout

Name Start Time 08/12/2009 Implementations First Job

Date rule Cron expression

Rule

Rule Syntax Help

```

+----- seconds (0 - 59)
| +----- minute (0 - 59)
| | +----- hour (0 - 23)
| | | +----- day of month (1 - 31)
| | | | +----- month (1 - 12)
| | | | | +----- day of week (0 - 6) (Sunday=0 or 7)
| | | | | *
* * * * *

```

Field Name	Allowed Values	Allowed Special Characters
Seconds	0-59	, - * /
Minutes	0-59	, - * /
Hours	0-23	, - * /
Day-of-month	1-31	, - * ? / L W C
Month	1-12 or JAN-DEC	, - * /
Day-of-Week	1-7 or SUN-SAT	, - * ? / L C #
Year (Optional)	empty, 1970-2099	, - * /

Examples:

```

0 15 10 ? * *      Fire at 10:15am every day
0 0/5 14,18 * * ?   Fire every 5 minutes starting at 2pm and ending at 2:55pm, AND fire every 5
                    minutes starting at 6pm and ending at 6:55pm, every day

```

28.4 Implementations

28.4.1 Quartz Scheduler

This is the default implementation of the Scheduler Aspect and it's distributed as roma-scheduler-quartz module. It's based on the well known job scheduling system “[Quartz enterprise job scheduler](#)” and that abstracts the user from it.

28.4.1.1 Install

At the command prompt, we have to type the following command to add Quartz Scheduler implementation to our Application:

```
roma module add scheduler-quartz
```

This command adds roma-scheduler-quartz.jar and quartz-all-x.x.x.jar in /WebContent/WEB-INF/lib directory and “applicationContext-scheduler-quartz.xml” in /src/META-INF/components directory of our project scaffolding. The file “applicationContext-scheduler-quartz.xml” contains the definition of our scheduler and, particularly, the list of the available jobs. In the following example, you can find an example of this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="no" default-dependency-check="none" default-lazy-
init="false">
  <!--Scheduler application context -->
  <bean id="SchedulerAspect"
    class="org.romaframework.module.schedulerquartz.QuartzSchedulerAspect"
    singleton="true">

    <property name="jobsImplementation">
      <map />
    </property>

  </bean>
</beans>
```

28.4.1.2 Configure

As you can see, no job is defined and put in the list of available ones.

29 Portlet

To create a Roma Portlet project you just have to launch a command from Roma console:

```
roma project create portlet <PortletName> <PortletPackage>  
<ProjectPath>
```

Roma will create the project scaffolding.

The project will contain the following:

- <MyPortletName>.java: the main POJO that will be displayed by the portlet
- <MyPortletName>Portlet.java: the portlet that will be deployed
- <MyPortletName>ApplicationConfiguration.java: that manages user session start and stop
- <MyPortletName>.jsp: the jsp corresponding to the POJO
- <MyPortletName>.xml: the xml annotations of the POJO

To deploy your portlet project in a Portal Server you just have to launch the Ant target called “install” in the build.xml file of the project; this will create a deployable .war file in the /dist directory.

To add a new portlet to an existing Roma Portlet project, just type the following command in Roma console:

```
roma project portlet <MyPortletName>
```

Where <MyPortletName> is the name of the new portlet.

This wizard will add to your project some classes:

The wizard will also update configuration files in your project.

30 Chart

30.1 Introduction

Roma Chart jFreeChart module is intended to represent POJO's as charts of different types, like pie charts, bar charts, line charts etc. Roma Chart jFreeChart module is based on jFreeChart Open Source project (<http://sourceforge.net/projects/jfreechart>).

30.2 What kind of objects can be represented as charts

Chart jFreeChart module allows to represent as charts the following types of objects:

- *Collection<Number>*
- *Collection<Collection<Number>>*
- *Map<Comparable<? >, Number>*
- *Map<Comparable<? >, Map<Comparable<? >, Number>*
- *ChartRenderer*
- *ChartRepresentable*

In the future it should be possible to also display arrays.

30.3 What kind of charts can be represented

With Roma Chart jFreeChart you can represent the following types of chars:

- Area charts and stacked area charts (figure 1)
- Bar charts and stacked bar charts, 2D and 3D (figure 2)
- 2D and 3D line charts (figure 3)
- 2D and 3D pie charts (figure 4)
- Spider charts (figure 5)

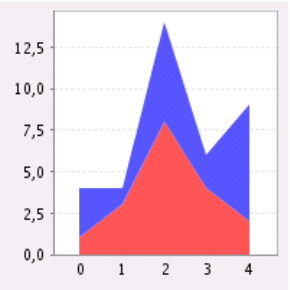
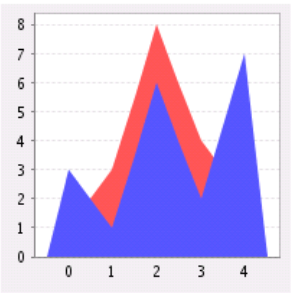


Figure 1: Area charts

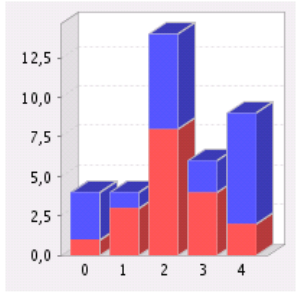
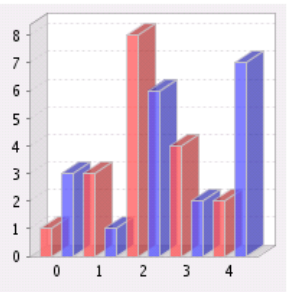
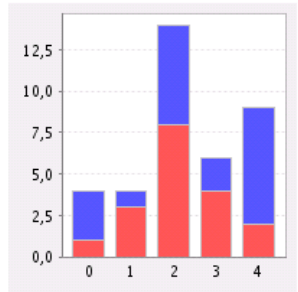
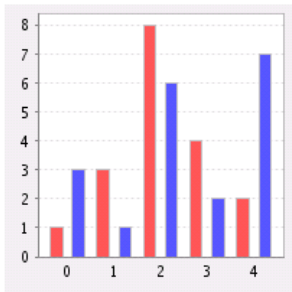


Figure 2: Bar charts

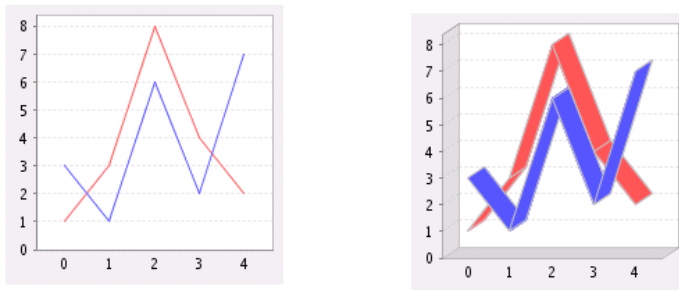


Figure 3: Line charts

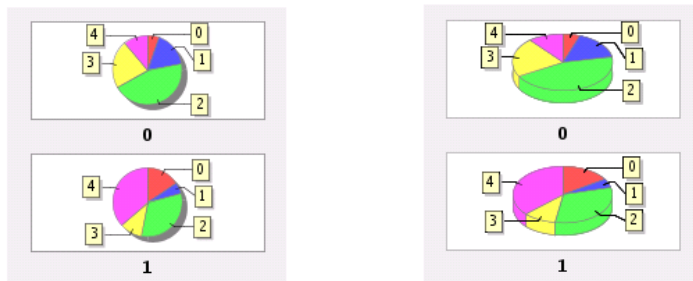


Figure 4: Pie charts

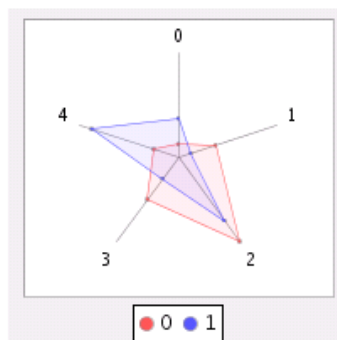


Figure 5: Spider chart

30.4 Installing the chart module

To use Chart you have to install it and to register it in your project, as for all Roma modules. To install the Roma Chart jFreeChart module, just download the binary

distribution of the module and copy it in your . When you have your module installed, you can add it to your project, launching the following command in a shell:

```
> roma module add chart-jfreechart
```

30.5 Rendering a chart

When you have the module installed, you can render POJO's as charts, simply declaring `render` as the rendering type for your object. In the following we show a simple example that demonstrates how to render a list of integers as a chart:

```
...
public class MyClass{

    @ViewField(render = ViewConstants.RENDER_CHART)
    private List<Integer> myChart;

    public MyClass(){
        myChart = new ArrayList<Integer>();
        myChart.add(1);
        myChart.add(5);
        myChart.add(3);
    }
    // TODO add here getters and setters
    ....
}
```

30.5.1 Maps and Collections

As we have seen in the first example, it is possible to use Collections and Maps to build simple charts. Here we briefly describe the behaviour of Roma Chart module when a Collection or a Map is rendered as a chart.

Rendering *Collection<Number>*

A *Collection<Number>* is rendered as a 3D bar chart, where the first element of the collection is the element labelled on the chart, the second element is labelled as 1 and so on. The height of the bars is the value of each element.

Rendering *Collection<Collection<Number>>*

A *Collection<Collection<Number>>* is rendered as a grouped 3D bar chart where the first group represents the first elements of each "internal" collection and so on; each sub-collection is rendered in a different colour.

Rendering *Map<Comparable<? >, Number>*

A *Map<Comparable<? >, Number>* is rendered as a 3D bar chart, where each entry is used as an element of the chart, whose label is the `toString()` of the key and the value is the value bound to the key.

Rendering *Map<Comparable<? >, Map<Comparable<? >, Number>>*

A *Map<Comparable<? >, Map<Comparable<? >, Number>>* is represented as a 3D bar chart that is the composition of all the sub-charts of the inner maps. Each key of the external map is used to identify a subset of the chart (in a different colour).

30.5.2 The ChartRenderer class

Maps and Collections are sufficient to render simple charts. However, if you want more control on the parameters of the chart (height, width, type, orientation etc.) you should use ChartRenderer. With a simple example we are going to show how to represent a complex data set as a spider chart, and how to set its dimensions.

```
...
public class MyClass{

    @ViewField(render = ViewConstants.RENDER_CHART)
    private ChartRenderer myChart;

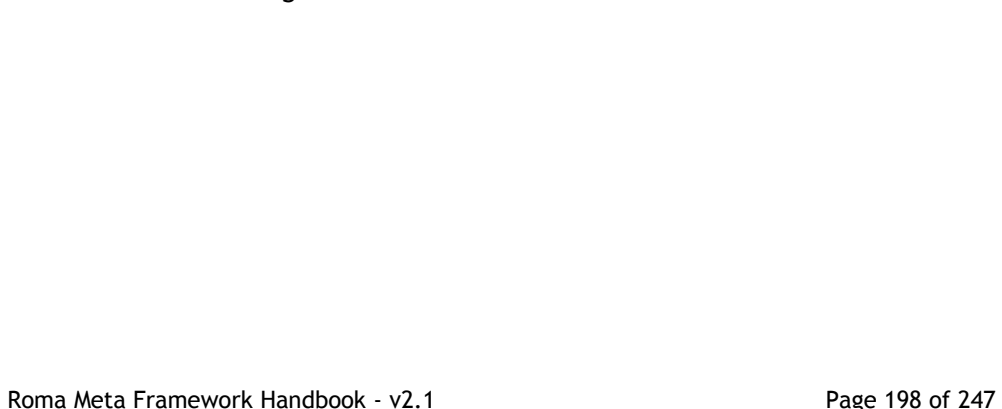
    public MyClass(){
        myChart = new ChartRendererImpl();

        myChart.setPoint("x", "a", 1);
        myChart.setPoint("x", "b", 2);
        myChart.setPoint("x", "c", 3);
        myChart.setPoint("y", "a", 4);
        myChart.setPoint("y", "b", 1);
        myChart.setPoint("y", "c", 2);

        EditableRenderOptions options = EditableRenderOptions.newInstance();
        options.setWidth(300);
        options.setHeight(300);
        options.setChartType(RenderOptions.CHART_TYPE_SPIDER);

        myChart.setOptions(options);
    }
    // TODO add here getters and setters
    ....
}
```

the result is the following:



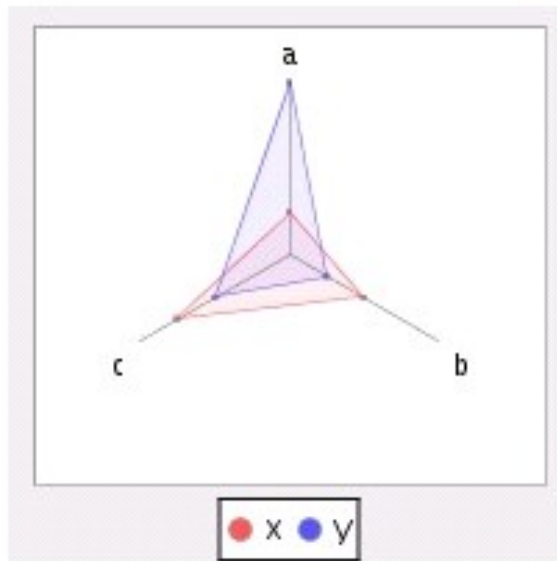


Figure 25.6: example of spider chart

30.5.3 Using custom Chart types

When you need a huge customization on the chart displayed, you can use all the power of the underlying implementation by registering your own. In the following code the implementation `A1CustomBarChartRender` will be registered with name "A1":

```
ChartRendererImpl.registerCustomChartType("A1", new A1CustomBarChartRender());
```

To use it set that name in the `EditableRenderOptions` instance:

```
...
public class MyClass{

    @ViewField(render = ViewConstants.RENDER_CHART)
    private ChartRenderer myChart;

    public MyClass(){
        myChart = new ChartRendererImpl();

        myChart.setPoint("x", "a", 1);
        myChart.setPoint("x", "b", 2);
        myChart.setPoint("x", "c", 3);
        myChart.setPoint("y", "a", 4);
        myChart.setPoint("y", "b", 1);
        myChart.setPoint("y", "c", 2);

        EditableRenderOptions options = EditableRenderOptions.newInstance();
```

```

        options.setWidth(300);
        options.setHeight(300);
        options.setChartType("A1");    // --- SET THE CUSTOM CHART TYPE

        myChart.setOptions(options);
    }
    // TODO add here getters and setters
    ....
}

```

And this is the implementation of the EditableRenderOptions class taken as example:

```

public class A1CustomBarChartRender implements ChartTypeRenderer {

    public void render(Dataset dataset, RenderOptions options,
        OutputStream outputStream) {

        JFreeChart chart = ChartFactory.createBarChart(options.getChartTitle(),
            options.getXLabel(), options.getYLabel(),
            (CategoryDataset) dataset, options.getOrientation(),
            options.isShowLegend(), options.isShowTooltip(), false);

        final CategoryPlot plot = chart.getCategoryPlot();

        final BarRenderer renderer = (BarRenderer) plot.getRenderer();

        final GradientPaint gpValues = new GradientPaint(0.0f, 0.0f, Color.green,
            0.0f, 0.0f, Color.lightGray);

        renderer.setSeriesPaint(0, gpValues);
        renderer.setSeriesPaint(1, Color.BLUE);

        final IntervalMarker target = new IntervalMarker(-0.01, 0.01);
        target.setPaint(Color.RED);
        plot.addRangeMarker(target, Layer.FOREGROUND);

        CategoryAxis domainAxis = plot.getDomainAxis();
        domainAxis.setMaximumCategoryLabelLines(3);
        domainAxis.setCategoryLabelPositionOffset(10);

        try {
            ChartUtilities.writeChartAsPNG(outputStream, chart, options.getWidth(),
                options.getHeight());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

30.5.4 The ChartRepresentable interface

It is possible to render your POJO's as charts implementing the ChartRepresentable interface:

```

public interface ChartRepresentable {
    public ChartRenderer generateGraph();
}

```



```
}
```

The interface method must return the chart representation of your POJO. This object will be represented instead of your POJO if you declare for that `render = ViewConstants.RENDER_CHART`.

30.5.5 The ChartViewer class

The ChartViewer class is a simple cockpit that allows you to manipulate a chart representation through a simple GUI, allowing you to modify its dimensions, type, orientation etc.

30.5.6 Using Chart Module to build images

The Roma Chart jFreeChart module can also be used to create images that can be manipulated programmatically. To obtain an image from a ChartRenderer object you can proceed as follows:

```
ByteArrayOutputStream outputStream = new  
ByteArrayOutputStream( );  
ChartRenderer renderer = getMyRenderer(); //generate a chart  
renderer.render(outputStream);  
byte[] image = outputStream.toByteArray();
```

The result is an array representation of a PNG image. If you want to obtain a chart from a Collection or from a Map, you can use the ChartHelper class, that contains static methods to render images from Maps and Collections.

31 Service Aspect

The service aspect has the responsibility to expose a simple pojo as a [web service](#), create a client for a web service, invoke a remote service, and start all the defined services.

As for all the other aspects the configuration of the aspect could be done by POJO annotation or XML annotation.

```
public interface ServiceAspect extends Aspect {
    public <T> T getClient(Class<T> iInterface, String iUrl);

    public List<Object> invokeDynamicService(String serviceURL, String
                                                operationName,
                                                List<Object> inputs)
        throws UnsupportedOperationException;

    public List<ServiceInfo> listOperations(String serviceURL)
        throws UnsupportedOperationException;
}
```

31.1 How to expose a service

The generation of a web service from a POJO is performed by the definition of an interface, the service aspect will expose in the service all the the methods of the defined interface:

```
public interface HelloWorld {
    public String greet(Person person);
}

// Mark the pojo as service:
@ServiceClass(interfaceClass = HelloWorld.class, serviceName =
"Hello")
public class HelloWorldImpl implements HelloWorld {

    public String greet(Person person) {
        String s = "Hi, " + person.getName();
        return s;
    }

    public String insult(Person person) {
        String s = person.getName() + " is stupid!"
        return s;
    }
}

// Get the client
HelloWorld client = Roma.component(ServiceAspect.class)
                        .getClient(HelloWorld.class,
                        "http://applicationPath/services/Hello"
);
```

The example shows all you have to do to expose a service and to create a client, as you can see defining `interfaceClass = HelloWorld.class` we expose only the method in `HelloWorld` interface so we avoid to create a service that could insult persons.

31.2 Implementations

31.2.1 Service CFX

Framework	Author	Added
Apache CXF	Giordano Maestro, Luigi dell'Aquila	February 2008

31.2.1.1 @ServiceClass

Property	Description	Allowed values
<code>interfaceClass</code>	The interface that the service expose	Interface Class
<code>serviceName</code>	The name of the exposed service. The final address for the invocation of will be <code>http://applicationPath/services/Hello</code>	String value

31.2.1.2 @Callback (Roma [Service CXF](#) JSONP Support)

Property	Description	Allowed values
<code>paramCallback</code>	The parameter name of the JSONP callback function	String value

CXF Service Aspect, by default, listen to the address with this format:

```
http://<host>[:<port>]/services/<service>/<method>[/param]*
```

Example:

```
http://localhost:8080/roma/services/info/profile/luca
```

The `@Callback` annotation is needed by the CXF Service Aspect to know the parameter name (in the URL path or in the Request query parameters) that contains the callback function requested for the JSONP call.

31.2.1.3 JAX-RS Support

The Apache [Service CXF](#) framework supports JAX-RS (JSR-311), Java API for RESTful Web Services, that allows roma users to configure Rest Services by simply annotate the class with standards JAX-RS Java annotations.

The Roma [Service CXF](#) adds to that feature two more Mime type handlers: CSV and JSONP, not supported by default in CXF framework.

31.2.1.4 How to expose a RESTful Web Service

To expose your Service you need only to annotate the interface or the class that implements your operations. All the annotation (excluding the `@Callback` annotation) are JSR-311 standart Java annotations.

For all next examples we'll use this POJO entity class:

```
public class TestEntity {  
    protected Integer    id;  
    protected String     name;  
    protected String     surname;  
    public Integer getId() {  
        return id;  
    }  
    public void setId(Integer id) {  
        this.id = id;  
    }  
    public String getNome() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getSurname() {  
        return surname;  
    }  
    public void setSurname(String surname) {  
        this.surname = surname;  
    }  
}
```

31.2.1.5 Standard JAX-RS annotations

@Path annotation is applied to resource classes or methods. The value of @Path annotation is a relative URI path and follows the URI Template format.

The method of the request can be configured, JAX-RS specification defines a number of annotations such as @GET, @PUT, @POST and @DELETE.

@Produces annotation is used to specify the format of the response, CXF support some format like application/json, application/xml and text/plain. Roma supports also application/json and application/csv types.

@PathParam is the name of the parameter of the URL which will be passed to the method.

@Context will get the parameter from the Context, it can be used with HttpSession, HttpServletRequest, HttpServletResponse, ServletContext, UriInfo.

@XmlRootElement is the root name of the entity that will be transformed in JSON. If exists the formatter will add an additional level on the JSON Object.

Without any annotation we'll have an output like:

```
{ "id": "...",  
  "name" : "...",  
  "surname" : "..."}  
}
```

With the annotation @XmlRootElement(name = "entity") we'll have an output like:

```
{ "entity" : { "id": "...",  
              "name" : "...",  
              "surname" : "..."} }  
}
```

For other informations about the annotations, their meanings, configuration, usage in CXF framework see the [CXF JAX-RS documentation](#).

31.2.1.6 Example

Service interface

```
public interface TestJSONPService {  
    public TestEntity getJSONEntityById(@PathParam("id") Integer id);  
    public TestEntity getJSONPEntityByIdAndPathParam(@PathParam("id") Integer id);  
    public TestEntity getJSONPEntityByIdAndQueryParam(@QueryParam("id") Integer id);  
    public TestEntity getXMLEntityById(@PathParam("id") Integer id);  
}
```

```

    public TestEntity getCSVEntityById(@PathParam("id") Integer id);
}

```

Service implementation

```

@ServiceClass(interfaceClass = TestJSONPService.class, aspectImplementation =
    CxfServiceAspect.class)
@Path("/entities/")
public class TestJSONPServiceImpl implements TestJSONPService {

    @GET
    @Path("/JSONEntityById/{id}")
    @Produces("application/json")
    public TestEntity getJSONEntityById(@PathParam("id") Integer id){
    }

    @GET
    @Path("/JSONPEntityById/{callback}/{id}")
    @Produces("application/jsonp")
    @Callback(paramCallback = "callback")
    public TestEntity getJSONPEntityByIdAndPathParam(@PathParam("id") Integer id)
    {
        ...
    }

    @GET
    @Path("/JSONPEntityById/")
    @Produces("application/jsonp")
    @Callback(paramCallback = "callback")
    public TestEntity getJSONPEntityByIdAndQueryParam(@QueryParam("id") Integer
    id){
        ...
    }

    @GET
    @Path("/XMLEntityById/{id}")
    @Produces("application/xml")
    public TestEntity getXMLEntityById(@PathParam("id") Integer id){
        ...
    }

    @GET
    @Path("/CSVEntityById/{id}")
    @Produces("application/csv")
    public TestEntity getCSVEntityById(@PathParam("id") Integer id){
        ...
    }
}

```

This class exposes a REST service that returns the TestEntity in various types: JSON, JSONP, XML and CSV.

Supposing that your application base path is <http://localhost:8080/roma> and the application path is <http://localhost:8080/roma/app> we can reach the previous

service by calling the URLs:

```
http://localhost:8080/roma/services/entities/JSONEntityById/1  
http://localhost:8080/roma/services/entities/JSONEntityById/2
```

This will execute the operations contained in "getJSONEntityById(id)" method with id values 1 and 2, and the browser will get it as a JSON object.

```
http://localhost:8080/roma/services/entities/JSONPEntityById/callbackFunction/  
1  
  
http://localhost:8080/roma/services/entities/JSONPEntityById/callbackFunction/  
2
```

or

```
http://localhost:8080/roma/services/entities/JSONPEntityById?  
id=1&callBack=callbackFunction  
  
http://localhost:8080/roma/services/entities/JSONPEntityById?  
id=1&callBack=callbackFunction
```

This will execute operations with id values 1 and 2, and the callback function name as "callbackFunction".

The first two URLs will call the method in "getJSONPEntityByIdAndPathParam(id)" that gets the informations from the URL path, the other two URLs will call the method "getJSONPEntityByIdAndQueryParam(id)" that gets the informations from the Request query parameters. The service will return to the browser the JSON object with the JSONP calling of "callbackFunction" script.

```
http://localhost:8080/roma/services/entities/XMLEntityById/1  
http://localhost:8080/roma/services/entities/XMLEntityById/2
```

This will execute the operations contained in "getXMLEntityById(id)" method with id values 1 and 2, and the browser will get it as a XML object.

```
http://localhost:8080/roma/services/entities/CSVEntityById/1  
http://localhost:8080/roma/services/entities/CSVEntityById/2
```

This will execute the operations contained in "getCSVEntityById(id)" method with id values 1 and 2, and the browser will get it as a CSV object.

31.2.2 Service DWR

Framework	Author	Added
DWR	EmanueleTagliaferri	August 2009

31.2.2.1 @ServiceClass

Property	Description	Allowed values
interfaceClass	The interface that the service expose	Interface Class
serviceName	The name of the exposed service. The final address for take the javascript to call service will be <code>http://applicationPath/dwr/interface/serviceName.js</code>	String value

The service DWR allow to expose a service on client side as javascript, you must define an interface to be exposed, make an implementation and annotate the implementation.

31.2.2.2 Example:

Define interface:

```
public interface MyDWService {  
    public String sayHello();  
    public String sayHalloTo(String target);  
}
```

Write service Implementation:

```
@SeviceClass(serviceName="MyDWService",interfaceClass=MyDWService.  
class)  
public class MyDWServiceImpl implements MyDWService {  
    public String sayHello(){  
        return "Hello ";  
    }  
    public String sayHalloTo(String target){  
        return "Hello "+target;  
    }  
}
```



```
}  
}
```

Write the client page:

```
<html>  
  <head>  
    <script type="text/javascript"  
src="dwr/interface/MyDWRService.js">  
    </script>  
    <script type="text/javascript" src="dwr/engine.js"> </script>  
  </head>  
  <body>  
    <input type="button" onclick="alert(MyDWRService.sayHello())"  
value="Say Hello">  
    <input type="button"  
onclick="alert(MyDWRService.sayHelloTo('john'))" value="Say Hello to john">  
  </body>  
</html>
```

31.3 REST Service

Roma web projects allows a REST access to user-defined Services. It means that by mapping a POJO with a `@ServiceClass` annotation his methods can be called from the external using an URL.

To do that you had to create a class with the methods you need, define the service name (by `@ServiceClass` Roma annotation) map the `RestServiceFilter` in `web.xml`, call it by the URL and Roma will do the rest.

The URL syntax is very simple: assuming that your base application URI is <http://localhost:8080/roma/> your REST Service URL would be like this:

[<serviceName>/<operation>/<parameter1>/<parameter2>/.../<parameterN>](http://localhost:8080/roma/)



It's fundamental to order the parameters in the same way they are defined in the service method.

Usually you should use this feature to forward the application flow to a specific POJO, otherwise, if you only use that to perform operations you will be redirected to the current shown POJO (like a page refresh).

RestServiceFilter configuration

The filter configuration needs just two parameters:

1. the application web path (ex. /app)
2. the url pattern where you want to call the REST services

31.3.1 Examples

RestTestClass

```
@ServiceClass(serviceName="service")
public class MyRestClass{

    public void firstOperation(){
        //do something
    }

    public void secondOperation(String text){
        //do something
    }
}
```

```

public void thirdOperation(String text, int number){
    //do something
}

public void fourthOperation(int number , String text){
    //do something
}

}

```

web.xml

```

<filter>
    <filter-name>RestServiceFilter</filter-name>
    <filter-
class>org.romaframework.web.service.rest.RestServiceFilter</filter-class>
    <init-param>
        <param-name>applicationWebPath</param-name>
        <param-value>app</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>RestServiceFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>

```

31.3.1.1 Service calling without parameters

To call the service you have created you only to enter the correct URL:

```

http://localhost:8080/roma/service/firstOperation/

```

That URL calls the method firstOperation in our MyRestClass.

31.3.1.2 Service calling with one parameter

This URL calls the method secondOperation in our MyRestClass passing the String “textStringParameter” as parameter.

```

http://localhost:8080/roma/service/secondOperation/textStringParameter

```

31.3.1.3 Service calling with multiple parameters

This URL calls the method thirdOperation in our MyRestClass passing the String “textStringParameter” as first String parameter and the Integer 10 as second number parameter.

```

http://localhost:8080/roma/service/thirdOperation/MyText/10

```

This URL calls the method fourthOperation in our MyRestClass passing the Integer 10 as first String parameter and the String “textStringParameter” as second Number

parameter.

```
http://localhost:8080/roma/service/fourthOperation/10/MyText
```

31.3.1.4 Wrong urls

This URL is incorrect because the REST Service need the operation name that has to be executed.

```
http://localhost:8080/roma/service/
```

This URL is incorrect because the method defined with name secondOperation needs a parameter.

```
http://localhost:8080/roma/service/secondOperation/
```

This URL calls the method fourthOperation in our MyRestClass, passing the String “textStringParameter” as first String parameter and the Integer 10 as second number parameter, and obviously is incorrect because the method needs the first parameter to be an Integer and in this case “MyText” can’t be converted to an Integer.

```
http://localhost:8080/roma/service/fourthOperation/MyText/10
```

31.3.1.5 Real world example

Example with Blog application: Rest Service is used to retrieve Blogs by blog name and post name.

We’re assuming that the web.xml configuration is the one previously defined.

BlogService class

```
@ServiceClass(serviceName = "blogService", interfaceClass = Object.class)
public class BlogService {

    public void blog(String iName) {
        BlogRepository repo = Roma.component(BlogRepository.class);
        BlogInstance page = new BlogInstance();
        page.setMode(CRUDInstance.MODE_READ);
        page.setEntity(repo.getBlogByName(iName));
        RomaFrontend.flow().forward(page);
    }

    public void post(String iPostTitle) {
        BlogRepository repo = Roma.component(BlogRepository.class);
        BlogInstance page = new BlogInstance();
        page.setMode(CRUDInstance.MODE_READ);
    }
}
```

```

        page.setEntity(repo.getBlogByPostTitle(iPostTitle));
        RomaFrontend.flow().forward(page);
    }
}

```

31.3.1.6 Display the blog called “BlogTest”

<http://localhost:8080/roma/go/blogService/blog/BlogTest>

The screenshot shows a web application interface with a navigation bar at the top containing the following links: Back Office, Blog, Home Page, Change Password, and Logout. Below the navigation bar, there is a form with the following fields and controls:

- Title:** A text input field containing the value "Blog di prova".
- Posts:** Two radio button options: "nessuno" (selected) and "Titolo".
- Author:** A text input field containing the value "Me".
- Name:** A text input field containing the value "BlogTest".
- Buttons:** An "Annulla" button with a red circular icon is located at the bottom left of the form.

31.3.1.7 Display the Blog that contains a post called “Titolo”

<http://localhost:8080/roma/blogService/post/Titolo>

This screenshot is identical to the previous one, showing the same web application interface with the navigation bar and the form fields: Title (Blog di prova), Posts (nessuno selected), Author (Me), Name (BlogTest), and the Annulla button.

31.3.1.8 Display the Blog that contains a post called “fefas”

<http://localhost:8080/roma/blogService/post/fefas>

Back Office

Blog

Home Page

Change Password

Logout

Title

Blog di prova 2

Posts

☐ Nessuno

☐ dsadfe

☐ fefas

Author

sempre me

Name

BlogTest2

 Annulla

32 Serialization Aspect

The serialization aspect allows to serialize and deserialize object instances to and from different formats using different serialization strategies.

32.1 Java Interface

```
public interface SerializationAspect extends Aspect {

    public byte[] serialize(Object toSerialize);

    public void serialize(Object toSerialize, OutputStream outputStream);

    public byte[] serialize(Object toSerialize, String inspectionStrategy,
String formatStrategy);

    public void serialize(Object toSerialize, OutputStream outputStream, String
inspectionStrategy, String formatStrategy);

    public Object deserialize(byte[] data);

    public Object deserialize(InputStream inputStream);

    public Object deserialize(byte[] data, String inspectionStrategy, String
formatStrategy);

    public Object deserialize(InputStream inputStream, String
inspectionStrategy, String formatStrategy);

    public void deserialize(Object toFill, byte[] data);

    public void deserialize(Object toFill, InputStream inputStream);

    public void deserialize(Object toFill, byte[] data, String
inspectionStrategy, String formatStrategy);

    public void deserialize(Object toFill, InputStream inputStream, String
inspectionStrategy, String formatStrategy);

    public void addInspectionStrategy(SerializationInspectionStrategy
strategy);

    public void addFormatStrategy(SerializationFormatStrategy strategy);
}
```

32.2 Serialization Inspection Strategy

The Serialization Inspection Strategy determines which field must be include in the serialization phase and how much deep must be the inspection of the object to serialize. At this time are available two inspection strategies:

- Full inspection, that reads all the fields of an object and go in deep for each field recursively
- View inspection, that determines wich field to include using the View Features of object, for example it avoid the serialization of hidden fields.

All serialization inspection strategies ignore Java *transient* field.

32.3 Serialization Format Strategy

The Serialization Format Strategy determines how to format the data when it writes. At this time are available two serialization strategies:

- JSON
- Native format, namely Java Serialization

32.4 Example

```
public class EmployeeSerializerTest {
    private Employee employee;

    public void serialize(){
        OutputStream outputStream = new FileOutputStream("employee.json");

        Roma.aspect(SerializationAspect.class).serialize(
            employee,
            outputStream,
            SerializationConstants.INSPECTION_FULL,
            SerializationConstants.FORMAT_JSON)
    }

    public void deserialize(){
        InputStream inputStream = new FileInputStream("employee.json");

        Roma.aspect(SerializationAspect.class).deserialize(
            employee,
            inputStream,
            SerializationConstants.INSPECTION_FULL,
            SerializationConstants.FORMAT_JSON)
    }
}
```


33 Admin module

Admin module comes as a set of utility classes.

33.1 Info and InfoCategory

Often is applications you need to declare categories of attributes (think about the default values in a select box); to do this, Roma Admin module provides two simple classes: the Info, that represents single values and the InfoCategory, that represents a category of Infos. Roma Admin module provides GUIs to manage Info and InfoCategory instances and advanced optimization such as caching.

To leverage Info functionalities in your application code, you can use the InfoHelper class, that provides methods to create and retrieve Info and InfoCategory instances.

33.2 Realms

A realm is partition of application domain data that can be accessed exclusively by a defined set of user accounts. Admin module provides classes and user interfaces to create different realms in your applications.

33.3 Plug-ins

Modules		
Aspects		
Name	Class Name	Defined In Module
core	org.romaframework.aspect.core.CoreAspect	
semantic	org.romaframework.module.semantic.jena.JenaSemanticAspect	
reporting	org.romaframework.aspect.reporting.jr.JRReportingAspect	
view	org.romaframework.aspect.view.echo2.Echo2ViewAspect	
session	org.romaframework.aspect.session.echo2.Echo2SessionAspect	
persistence	org.romaframework.aspect.persistence.jdo.JDOTxPersistenceAspect	
flow	org.romaframework.aspect.flow.impl.POJOFlow	
i18n	org.romaframework.aspect.i18n.rb.I18NAspectResourceBundleImpl	
authentication	org.romaframework.module.users.UsersAuthentication	
Startup Shutdown		

Roma applications are composed by a set of Roma modules that can be installed when they are needed. Some of these modules can be configured at run time. Admin module provides a GUI to manage installed modules, start them up, shut them down and configure them.

33.4 Active sessions

Admin module provides user interfaces (see figure Error: Reference source not found) to control the Session Aspect of the currently running application. Through these interfaces you can

- retrieve information about active sessions
- send messages to a logged user (the message is displayed as a pop-up on the target user interface)
- kill a user session

View Refresh Send Message Shutdown Select All Deselect All

Sessions

Source	Account	Created	Last Accessed
127.0.0.1:37238	admin	Fri Aug 08 12:27:25 CEST 2008	Fri Aug 08 12:42:36 CEST 2008

33.5 Application Logs

Admin module provides user interfaces to inspect application logs. This interface allows to export log files generated by the application to analyse them.

33.6 Database

Admin module provides a GUI that allows you to use the application Persistence Aspect to perform custom queries straight on the application database. Of course this is not a feature that has to be exposed to every user, but with the application profiling (see [Users Module](#)) you can quickly hide it to all the users but the system administrators.

Limit Result Size

Command

select * from Info

Output

Command executed correctly.

Execute Clear

Result

INFO_ID	CATEGORY_INFOCATEGORY_ID_OID	TEXT	VALUE
1	1	Active	0
2	1	Inactive	0
3	1	Suspended	0
4	2	System	0
5	2	Login	0
6	2	Administration	0

33.7 Environment

Often you need to handle **variables** in your application such as:

- **Flags**, to enable or disable functionalities
- **Counters**, to save statistics values
- **Numbers or range of values**, i.e.: "Number of items to display in a page"

Where to place this variables? In your code as constants? Or in a separate property/XML file? Maybe in Spring XML configuration file?

The solution could be using the **Environments**.

33.8 What is an Environment

An **Environment** is a container of variables. You can handle multiple environments. A variable is itself a POJO of type **EnvironmentItem**:

```
public class Environment {
    protected String          name;
    protected String          description;
    protected Map<String, EnvironmentItem> parameters;
}

public class EnvironmentItem {
    protected String key;
    protected String description;
    protected String value;
    protected byte type;

    public static byte TYPE_STRING      = 0;
    public static byte TYPE_INTEGER     = 1;
    public static byte TYPE_FLOAT       = 2;
    public static byte TYPE_BOOLEAN     = 3;
}
```

33.9 Use Environment objects

You can use Environments by **EnvironmentHelper** class.

```
// GET AN ENVIRONMENT VARIABLE
String loadFactor =
EnvironmentHelper.getInstance().getCurrent().getParameterValue("loadFactor");

// SET AN ENVIRONMENT VARIABLE
EnvironmentHelper.getInstance().getCurrent().setParameterValue("loadFactor",
"10.3");
EnvironmentHelper.getInstance().getCurrent().save();
```

33.10 Admin module to handle environment objects

If you've included the Admin Menu in your application, click to **Admin Menu** → **Environments**, select an environment (main is the default), and click on **Update**:

Environment variables are grouped in separated tab by the prefix name. To group variables assign as name `<prefix>.<name>`.

33.11 How to access Admin functionalities

Admin module contains a simple Menu class that contains the links to all Admin functionalities. This Menu is by default injected in the menu bar of Roma WebReady projects; if you created a different project and you want to add AdminMenu to your menu bar you just have to add the following code to your MainMenu class

```
protected AdminMenu administration = new AdminMenu();
public AdminMenu getAdministration() {
    return administration;
}
```

34 Users module

Users module is based on [Admin](#) module where it depends and comes as a set of domain classes to handle users, account and groups.

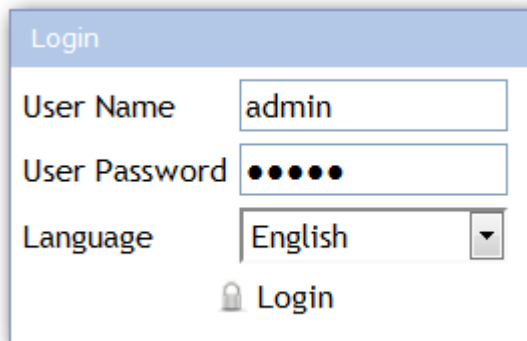
34.1 Dependencies

- [Admin module](#)

34.2 Standard Login

If you want to use the accounting system of the Users module you must assure to have the BaseAccount object in your session. The simpler way to reach this is using the Login form to enter in your application.

Below the default Login form:



Starting from version 2.0.0 you can select the language before to login and entire application will use this locale setting.

34.2.1 Set the login form as the first POJO to display

Make the **Login** POJO the entry point for your application, by changing the source `<application-package>.CustomApplicationConfiguration`:

```
public class CustomApplicationConfiguration implements
ApplicationConfiguration {
    ...
}
```

```

    * Callback called on every user connected to the application
    */
    public void startUserSession() {
        ObjectContext.getInstance().show(new CustomLogin());
    }
    ...
}

```

34.2.2 Realm-based login

If you need to divide your application in different [Realms](#), then let to your CustomLogin class implementing the "RealmLogin" class instead of Login.

34.2.3 Write your own Login behaviour

CustomLogin classes is your own Login class implementation that inherit Domain-Users's base Login class. Create a CustomLogin class (usually under the package <application-package>.view) that implement the **onSuccess()** method like this below:

```

@ViewClass(label = "Login", render = "popup", layout = "popup")
public class CustomLogin extends Login {
    @Override
    protected void onSuccess() {
        // LOG HERE THE LOGON IF YOU NEED TO TRACE IT

        // SHOW THE HOME PAGE
        ObjectContext.getInstance().show(new HomePage());
    }
}

```

You can redefine the **onError()** method if you want to handle login errors in a way different from that standard (It just throws a **UserException**).

34.3 Activity Log

You can trace any user or system activity using the Activity Log. The Activity Log is a Log Aspect implementation that stores in the database the log entries.

34.3.1 Example logging all the login succeeded

```
@Override
protected void onSuccess() {
    Roma.aspect(LoggingAspect.class).log(LoggingConstants.LEVEL_INFO,
                                         LoggingConstants.MODE_DB,
                                         ActivityLogCategories.CATEGORY_LOGIN,
                                         "User logged");









    super.onSuccess();
}
```

This statement tell to the activity logger to create an entry for the current account using the level "LEVEL_INFO", category "Login" and note "User logged".






All the logged activities can be browsed using the Activity Log menu item of USERS menu. Below a screenshoot.

Range From  Range To  Account  

Levels Categories Notes



 Search  New  Read  Update  Delete  Report  Select All  Deselect All

When	Account	Notes	Category	Level
11/03/2009	admin	Logout	Login	1
05/03/2009	admin	Logout	Login	1
05/03/2009	admin	Logout	Login	1
05/03/2009	admin	Logout	Login	1
05/03/2009	admin	Logout	Login	1
24/02/2009	admin	Logout	Login	1
19/01/2009	admin	Logout	Login	1
08/01/2009	admin	Logout	Login	1
17/12/2008	admin	Logout	Login	1
17/12/2008	admin	Logout	Login	1
17/12/2008	admin	Logout	Login	1
17/12/2008	admin	Logout	Login	1
15/12/2008	admin	Logout	Login	1
05/12/2008	admin	Logout	Login	1
07/11/2008	admin	Logout	Login	1

Total Items Page     

34.4 Accounts

34.5 Profiling system

Mode Allow all but ▾ Home Page ProjectBackOfficeHom Parent Administrator  

Realm shop Name Back office Notes

Available Class Names


- CRUDEntity
- CRUDInstance
- CRUDMain
- CRUDPaging
- ClassesPanel
- ComposedEntityInstanc
- CustomLogin
- Customizable
- EntityPage
- HomePage

Available Members

- < Class >
- search()
- logout()
- userName()
- logo()
- footer
- searchText
- userLabel

Functions

Key	Value
HomePage.logo	<input type="checkbox"/>




 Add

Add

View

Modify

Remove

 Save  Cancel  Report

The most powerful thing in the Users Module is the profiling system. It has a very flexible way to define profiles:

- Each profile defines the **mode**, namely the way functions play:
 - **Allow all but**: you define only the exception. Use this when exception are less than all functionalities.
 - **Deny all but**: you define only what the profile can access. Use this when functions are much less than the total.
 - Each profile can define a **Parent profile** where to inherit the functionalities allowed and denied.
- Functionalities are expressed as a simple name in the format below:
 - **.field**: access or not to the field of the entity
 - **.field.enabled**: enabled or disabled the field
 - **.action**: access or not to the action of the entity
 - **.action.enabled**: enabled or disabled the action

You can profile from the Class level up to each fields/actions.

34.5.1 How to use it?

If you want to disable the Control Panel item from the main menu, just insert the function with name "MainMenu.controlPanel" (that is the action name that call the sub menu) and with allow equals to false.

If you'd like to hide the employee's salary field to the "Visitor" profile, then set it with mode "Allow all but" and insert the function "Employee.salary" with allow equals to false. That is.

Roma controller will check for each POJO's fields and actions if the current user profile is allowed to display.

More examples:

- Employee.save() = false: the profile will not see the save() action
- Employee.name.enabled = false: the profile will see the name of the employee but it could not modify it

34.6 Base Group

BaseGroup is a way to grouping accounts together. A BaseAccount can be part of multiple BaseGroups.

For example you can assign a group to all divisions in your company.

34.7 LDAP Authentication

If you already have an LDAP (or ActiveDirectory) server, you can configure Roma applications to use it for authentication.

All you have to do is to modify "applicationContext-core.xml" and replace the default AuthenticationAspect bean with the LDAP based implementation. Here is the code you have to put in the Spring configuration:

```
<bean id="AuthenticationAspect"
      class="org.romaframework.module.users.LdapAuthentication"
      singleton="true">

  <property name="domain" value="<domain-name>" />
  <property name="ldapHost" value="ldap://<host-name>" />
  <property name="searchBase" value="your AD root, e.g. dc=mydomain,dc=org" />

  <property name="singleSessionPerUser" value="false" />

  <property name="accountBinder">
    <bean class="org.romaframework.module.users.SimpleAccountBinder"/>
    <!-- override this for a new strategy of binding an LDAP account to a
         BaseAccount -->
  </property>
</bean>
```

```
</property>
</bean>
```

To configure it properly you have to replace the property values (domain, ldapHost, searchBase) with your server connection data.

LDAP authentication module relies on an “AccountBinder” to map an LDAP username to a BaseAccount instance. The default implementation (SimpleAccountBinder) looks in the database for a BaseAccount whose name is the same as the one user by LDAP, so if you want to use it you must have a BaseAccount for every user in your LDAP server.

If you need to write your custom logic to map LDAP accounts with BaseAccount you can implement the AccountBinder interface and replace the SimpleAccountBinder with your own custom implementation.

34.8 Users Security Aspect

This is an implementation of the Security Aspect with roles based on the Users module.

To use the User Security Aspect you must insert into the roles a special syntax to define if a user can execute an action or see a field.

Role Definition: to define a role will be used an syntax composed by two parts separated by character ":".

The first part identifies the category of the role:

- user, to identify a single account
- group, to identify a group of users/accounts
- profile, to identify all the user of the specified profile

The second part is a regular expression that must match with the value of category. If the regular expression doesn't match with the value of the category, the action or the field will be hidden and any attempts to invoke the action will throw a SecurityException.

34.8.1 Examples

To specify that an action can be executed by all the users you should use:

```
SecurityAction(roles={"user:.*"})
```

To specify that an action can be executed only by the users part of group "group1" you should use:

```
SecurityAction(roles={"group:group1"})
```

To specify that an action can be executed only by the user with profile "Administrator" you should use:

```
SecurityAction(roles={"profile:Administrator"})
```

To specify that a field can be read by any authenticated user and modified only by the users with profile "Administrator" you should use:

```
SecurityField(readRoles={"user:.*"}, writeRoles={"profile:Administration"})
```

To specify that all field of a class can be read by any authenticated users but written only by users with profile "Administrator" and the action of the class can be executed only by users of profile "Administrator" you should use:

```
SecurityClass(readRoles={"user:.*"}, writeRoles={"profile:Administration"},  
executeRoles={"profile:Administration"})
```

34.8.2 Full Java Example

```
public class Employee {  
    @SecurityField( readRoles = {"profile:Basic"}, writeRoles =  
{"profile:Administrator"} )  
    private String      name;  
  
    @SecurityField( readRoles = {"profile:Administrator"} )  
    private float       salary;  
  
    @SecurityAction( roles = {"profile:Administrator"} )  
    public void delete(){  
        ...  
    }  
}
```

35 Designer module

Designer is a graphic tool that help to tune your application by changing the meta data information at run-time, directly in your live application.

In order to use the designer module add the module in your application:


```
roma module add designer
```

Once installed the designer will enrich each single form only if the component `ApplicationConfiguration` has the property `applicationDevelopment` setted to `true` that is the default. Remember to set this attribute to `false` when you install your application in production to avoid changes of application by the final users.

You can also enable/disable it at run-time logging in as Administrator:

Blog

User: [admin](#)
Logged On: 2013/12/15 15:01:42



Enable/Disable the designer at run-time

Blog Control Panel

Administration Designer Users

Application Configuration Active Sessions Plugins Generic Info Categories Generic Infos Environments Application Logs Database

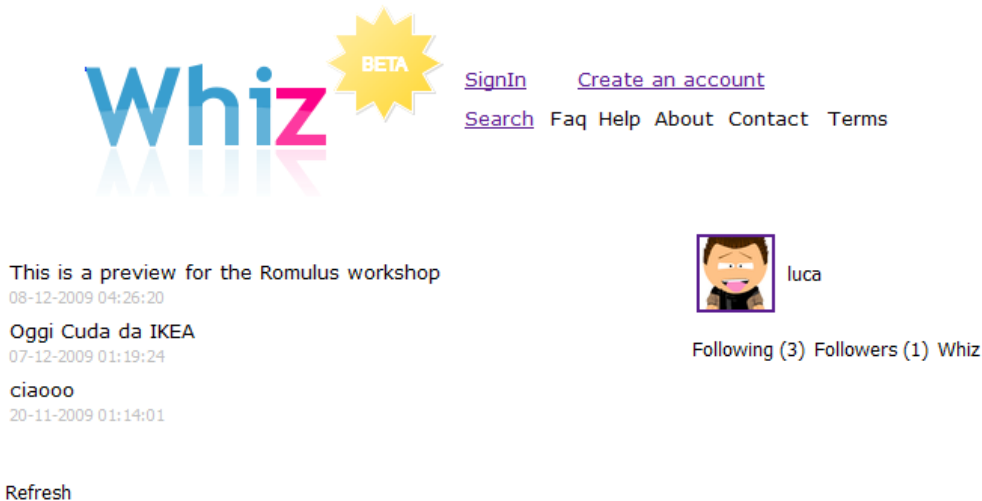
To make persistent changes please modify the META-INF/component/applicationContext.xml file.

Application Name	<input type="text" value="blog"/>	Application Package	<input type="text" value="org.test.blog"/>
Application Version	<input type="text" value="1.0"/>	Application Development	<input checked="" type="checkbox"/>
Status	<input type="text" value="unknown"/>		

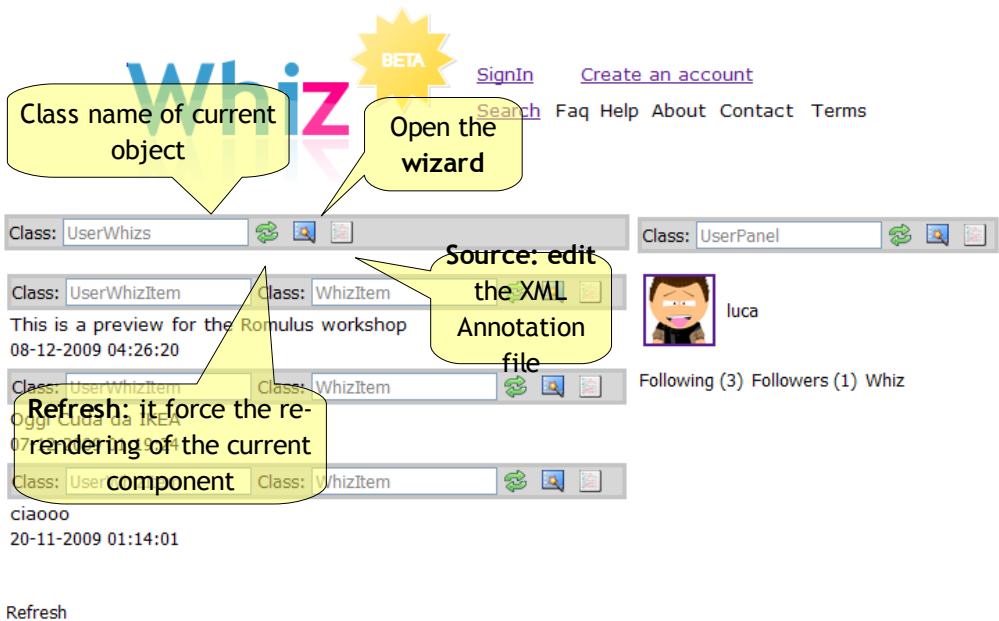
Configuration

Value

This is a Web Application developed with Roma before the installation of the designer module:



This is the same page, but with the designer enabled:

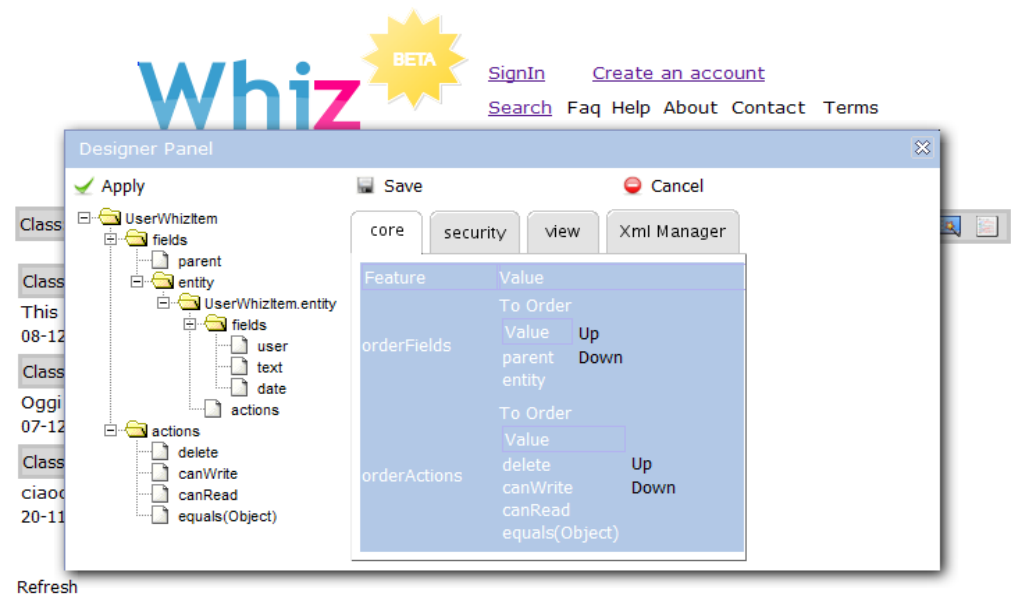


As you can see each single component (remember that they are always POJOs!) has a toolbar on top of it.

35.1 Open the Wizard

By clicking on the wizard button, a popup window will be opened containing all the meta data of the class.

Below the popup opened by clicking on the wizard button on the UserWhizItem object:



The left panel of the wizard contains the object structure as fields and actions (future development will allow also the edit of events). On the right side there is the detail of the node selected on the left tree.

In the picture above we can change the order of the fields. By selecting the aspect tab we can change other features such as validation and security constraints, view attributes, etc.

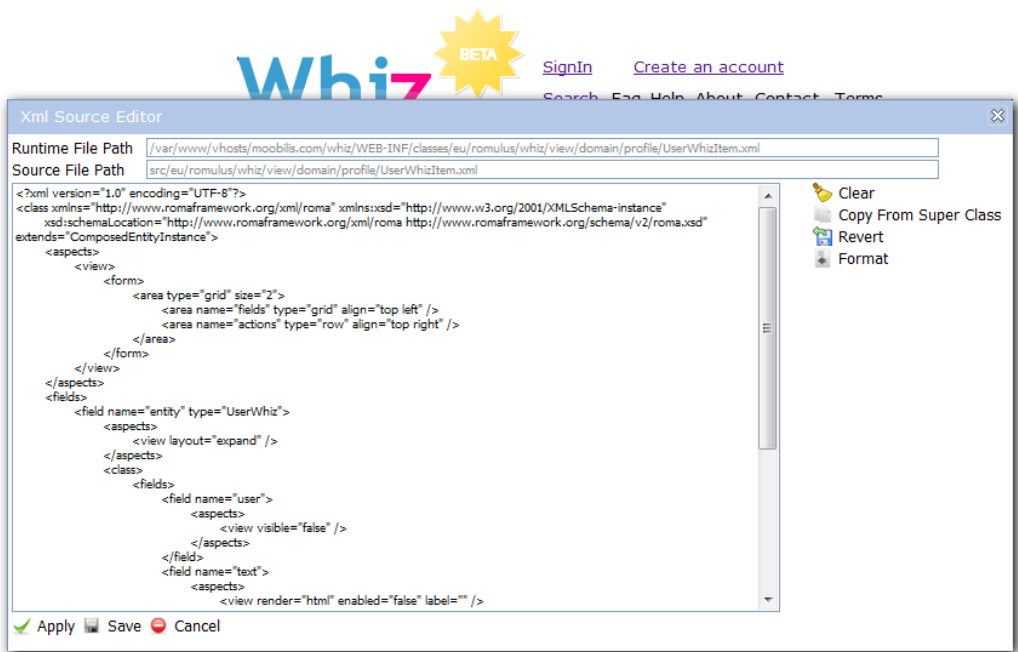
All the changes happen in real-time and are propagated to all the opened sessions.

35.2

35.3 Edit the Xml Annotation

By clicking on the source button, a popup window will be opened containing the XML file of the class.

Below the popup opened by clicking on the source button of the UserWhizItem object:



All the changes happen in real-time and are propagated to all the opened sessions.

36 Messaging module

The messaging module allow to exchange messages between users of your application. Users are BaseAccount classes, so it needs the [Admin](#) and [Users](#) Modules. Messages are sent and received without a mail server since anything play internally using the database.

36.1 Dependences

- [Users module](#)

36.2 Display Messaging menu inside your application menu

Link the class `org.romaframework.module.messaging.view.domain.menu.MessagingMenu` inside your `MainMenu`.

Example:

```
@ViewClass(render = "menu")
public class MainMenu {
    ...
    private MessagingMenu msgMenu = new MessagingMenu();

    public MessagingMenu getMsgMenu() {
        return msgMenu;
    }
    ...
}
```


37 Token Authentication module

This module is a very simple implementation of an application based session.

You can use it to log in the system using Roma [Users](#) module credentials (BaseAccount with username and password), obtaining a token that can be used to retrieve the user session over multiple calls. This module is particularly useful for service implementations that require user authentication.

37.1 Example usage

To install this module in your application you just have to type the following command in Roma console:

```
> roma module add authentication-token
```

After that you can use the module API to log in, store and retrieve parameters from the current session.

Typical usage of this module is explained by this very simple code snippet

```
TokenAuthenticationModule auth =  
Roma.component(TokenAuthenticationModule.class);  
  
String mySessionToken =  
auth.login("myRealName", "myUsername", "myPassword");  
  
auth.setProperty(mySessionToken, propertyName, propertyValue);
```

After a while, you can retrieve the property from the session with the same syntax:

```
TokenAuthenticationModule auth =  
Roma.component(TokenAuthenticationModule.class);  
  
String myProperty = auth.getProperty(mySessionToken, propertyName);  
  
auth.logout(mySessionToken);
```

The module automatically handles session expiration: each session has a session timeout (that can be configured through Spring applicationContext file) that gets renewed each time a user makes an operation on the session (get/setProperty) or can be manually renewed using the *TokenAuthenticationModule.renew(token)* method.

38 Tips & Tricks

38.1 Cascade events

Sometimes forms need to be dynamics. A typical use case is the cascade selection done with Select field components. In our example we have 2 select: Countries and Cities. Cities depends by the selection of the first one.

Example:

```
public class CitySelector {
    @ViewField(render="select", selectionField="country")
    protected List<Country> countries;
    @ViewField(visible=AnnotationConstants.FALSE)
    protected Country country;

    @ViewField(render="select", selectionField="city")
    protected List<City> cities;
    @ViewField(visible=AnnotationConstants.FALSE)
    protected City city;

    public void setCountry(Country c) {
        cities = loadCitiesOfCountry(c); // BUSINESS METHOD
        city = null;
        Roma.fieldChanged(this, "cities");
    }
}
```



Remember to provide **Getter** and **Setter** for all the envolved properties. For the purpose of demonstration in this piece of code are omitted.

As you can see the interaction happens into the setter method of Country. Roma calls that method when the user changed the selection of "countries" select field because "country" property was signaled in the "selectionField" annotation.

After any changes remember to signal to Roma the properties are changed. In this case cities and "cities".

38.2 Get text input from the user at the fly

Use the `TextEditor` class and register the current pojo as listener for the response. The response is the `String` text the user has inserted.



Example:

```
public class Foo implements MessageResponseListener {

    public void ask() {
        Roma.component(FlowAspect.class).forward(
            new TextEditor("import", "Import curricula", this, ""));
    }

    public void responseMessage(Message message, Object response) {
        System.out.println((String) response);
    }
}
```

To customize the layout act in the stylesheet if you're using the [Echo2 View Aspect implementation](#).

Example:

```
<style name="TextEditor" base-name="Message"
type="nextapp.echo2.app.WindowPane">
    <properties>
        <property name="width" value="560px" />
        <property name="height" value="320px" />
    </properties>
</style>

<style name="TextEditor" base-name="Object" type="echopointng.TextAreaEx">
    <properties>
        <property name="width" value="500px" />
        <property name="height" value="200px" />
    </properties>
</style>
```

38.3 Create a wizard

Many applications need wizards to help users to do some tasks. Roma comes with 2 utility classes to help in building them:

1.The wizard container base class:

```
org.romaframework.frontend.domain.wizard.FormWizard
```

2.And the interface to implement to create a single step wizard:

```
org.romaframework.frontend.domain.wizard.FormWizardStep
```

Below an example of a wizard that send an activation code before to create a new account. The first step ask for an email address and on "next" button sends the email with the activation code. The second one ask the activation code just sent and create the account on "next". The CreateAccountWizard2 will create the account.

CreateAccountWizard.java

```
public class CreateAccountWizard extends FormWizard<BaseAccount> {  
    public CreateAccountWizard() {  
        super(new FormWizardStep[] { new CreateAccountWizard1(),  
                                     new CreateAccountWizard2(),  
                                     new CreateAccountWizard2() });  
    }  
}
```

In this case the wizard container just declare the steps in order: **CreateAccountWizard1** and **CreateAccountWizard2**. You can override all the button such as cancel() to define custom behavior.

38.3.1 Step 1

Insert a valid email address where we can send you the activation code. Once received cut and paste the activation code in the next page to continue.

Your Email

 Cancel  Begin  Back  Next  Finish

CreateAccountWizard1.java

```
public class CreateAccountWizard1 extends  
FormWizardAbstractStep  
                                implements CustomValidation  
{  
  
    @ValidationField(required = AnnotationConstants.TRUE)
```

```

protected String yourEmail;

@Override
public boolean onNext() {
    try {
        // GENERATE A UNIQUE HASH AS ACTIVATION CODE
        String activationCode =
Roma.component(AuthenticationAspect.class).encryptPassword(yourEmail +
System.currentTimeMillis());

        SystemMailConfiguration mailCfg = ((CustomApplicationConfiguration)
Roma.component(ApplicationConfiguration.class))
            .getEmailConfiguration();






        MailMessage msg = new MailMessage();
        msg.setToAccounts(yourEmail);
        msg.setSubject("[Test] Account verification");
        msg.setText("Validation code: " + activationCode );
        EmailMessageHelper.send(mailCfg, msg, null);
    } catch (Exception e) {
        log.error("[CreateProjectWizard1.next] Error on sending email", e);
        throw new ValidationException(this, "yourEmail", e.getMessage(), null);
    }
    return true;
}
}

```

38.3.2 Step 2

Insert the activation code received in the email we just sent you and click "Next" to continue. If you have not received any email or the address inserted is wrong, press the "Begin" button to restart.

Activation Code

 Cancel
  Begin
  Back
  Next
  Finish

CreateAccountWizard2.java

```

public class CreateAccountWizard2 extends FormWizardAbstractStep
    implements CustomValidation {
    @ValidationField(required = AnnotationConstants.TRUE, min = 24, max = 24)
    protected String activationCode;

    @Override
    public boolean onNext() {
        if(!activationCode.equals(container.getStep(CreateProjectWizard1.class).
            getActivationCode()))
            throw new ValidationException(this, "activationCode",
                "Activation code wrong. Retype correctly or press
the \"Back\" button to resend a new activation code");

        // CREATE THE ACCOUNT IN THE NEXT STEPS
    }
}

```

38.4 Show a collection field as editable using the TableWrapper component

The **TableWrapper** class handle collections in fair way by letting to the user to work on instances acting on the cells. You can add and remove instances inside the collections. The **TableWrapper** emulates a true Collection in Roma supporting also [events](#) such as **on<Field>Add()** when add a new item and **on<Field>Remove()** when remove an item.

This type shows the items using the **<Class>Listable** class allowing easy customization of columns.

Before this class you can obtain the same goal by using the **RENDER_TABLEEDIT** mode on the collection and by writing a lot of code for the management of the Wrapped collection of Listable types.

Skills	Levels
Linguaggio Java	Di base
Linguaggio C++	Sufficiente
Grails	Buono

Aggiungi Elimina

Collection items are editable directly inside the table

Actions to add and remove items

Example:

```
public class CurriculumInstance extends CRUDInstance<Curriculum> {  
    @ViewField(render = ViewConstants.RENDER_OBJECTEMBEDDED)  
    public TableWrapper<GenericValue> contacts;  
    // GETTER WAS OMITTED FOR BREVITY  
  
    public Curriculum() {  
        contacts = new TableWrapper<GenericValue>(GenericValue.class,  
                                                    entity.getContacts());  
    }  
  
    public void onContactsAdd(){  
        // CUSTOM EVENT WHEN ADD IS PRESSED  
    }  
}
```

This component is a logical component so it works against all [View Aspect](#) implementations.

38.5 Expand a collection into the form using the RowsetWrapper component

The **RowsetWrapper** class works exactly as **TableWrapper**, but it doesn't use the **RENDER_TABLEEDIT** rendering mode but the **RENDER_ROWSET**, expanding the content of the collection directly to the form.

Results

<input type="checkbox"/>	Subject	MS Windows	Comment		Vote	Excellent	Teacher	Gates Bill
<input type="checkbox"/>	Subject	Marketing	Comment		Vote	Bad	Teacher	Gates Bill
<input type="checkbox"/>	Subject	English	Comment		Vote	Excellent	Teacher	Capriati Joseph
<input type="checkbox"/>	Subject	English	Comment		Vote	Good	Teacher	Capriati Joseph
<input type="checkbox"/>	Subject	Mac os X	Comment		Vote	Bad	Teacher	Jobs Steve

Add Remove

Collection items are editable directly inside the table

Actions to add and remove items

Example:

```
public class ReportInstance extends CRUDInstance<Report> {  
    protected RowsetWrapper<Result> results = new  
        RowsetWrapper<Result>(Result.class, this,  
"entity.results");  
    public RowsetWrapper<Result> getResults() {  
        return results;  
    }  
}
```

This component is a logical component so it works against all [View Aspect](#) implementations.

38.6 Handle collection as Master and Details

The **MasterDetailWrapper** class handle collection in fair way by letting to the user to work with the detail of the selected instance. You can add and remove instances inside the collections. The **MasterDetailWrapper** emulates a true Collection in Roma supporting also [events](#) such as **on<Field>Add()** when add a new item and **on<Field>Remove()** when remove an item.

This type shows the items using the **<Class>Listable** class allowing easy customization of columns and the **<Class>Instance** for the detail.

Before this class you can obtain the same goal by using the **RENDER_TABLE** mode on the collection and by writing a lot of code for the management of the Wrapped collection of Listable types plus the management of the selected as detail.



Example

```
public class CurriculumInstance extends CRUDInstance<Curriculum> {
    @ViewField(render = ViewConstants.RENDER_OBJECTEMBEDDED)
    protected MasterDetailWrapper<GenericValue> experiences;

    public Curriculum() {
        experiences = new MasterDetailWrapper<Experience>(Experience.class, this,
                                                         "entity.experiences");
    }

    public MasterDetailWrapper<GenericValue> getExperiences() {
        return experiences;
    }
}
```

This component is a logical component so it works against all [View Aspect](#) implementations.

38.7 Create a tabbed CRUD Instance implementation

Very often you need to separate all your fields in multiple tabs. Roma comes in help by providing the `CRUDTabbedInstance<T>` class to use in place of `CRUDInstance<T>`.

38.7.1 Example

Class TaskInstance.java

```
public class TaskInstance extends CRUDTabbedInstance<Task> {  
  
    public TaskInstance() {  
        getTabs().addPage("General"), new GeneralTaskTab();  
        getTabs().addPage("History"), new HistoryTaskTab();  
        getTabs().addPage("Comments"), new CommentsTaskTab();  
    }  
}
```

Class GeneralTaskTab.java

```
public class GeneralTaskTab extends CRUDInstance<Task> {  
}
```

Class HistoryTaskTab.java

```
public class HistoryTaskTab extends CRUDInstance<Task> {  
}
```

Class CommentsTaskTab.java

```
public class CommentsTaskTab extends CRUDInstance<Task> {  
}
```

And in each XML file make visible or invisible the fields and actions you want per single tab page.

39 About Romulus



Romulus is a EU project focused on researching on novel methods for increasing productivity and reliability of web software development, in particularly, focused on Java web development.

It is based on two mature open source projects, Roma and LIFERAY, which will be extended according to this proposal needs and following an open source project development methodology.

Some important new modules for Roma Meta-Framework are being developed under the Romulus project, such as:

- Janiculum, plus the wizard-jsp
- ATP4Romulus, to execute a lot of tests in any Roma applications
- Semantic Aspect, to export the domain in semantic way
- Tevere Workflow, a transactional Workflow engine
- MyCocktail, a Web Mashup Editor
- IDE4Romulus, the Eclipse and NetBeans plugins
- Check, Install and Update [Console wizards](#)

Information about Romulus project can be found on Romulus web site <http://www.ict-romulus.eu>.

40 License

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of

the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
- b. You must cause any modified files to carry prominent notices stating that You changed the files; and
- c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those

notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or

Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

1. END OF TERMS AND CONDITIONS