

Katari user manual

January 1, 2012

Contents

1	General Overview	5
1.1	Welcome to Katari	5
2	Getting started	7
2.1	Starting a Katari project	7
2.2	Creating your first module	9
3	General Architecture	15
3.1	Architecture	15
3.1.1	Modules	15
3.1.2	Module initialization	16
3.1.3	Spring configuration	17
3.1.4	Currently supported modules	19
3.1.5	Layers	20
3.2	Tools and libraries	20
3.2.1	Core libraries	21
3.2.2	Frameworks	21
3.2.3	Support libraries	21
3.2.4	Support tools	21
4	Features provided by Katari	23
4.1	Menu support	23
4.1.1	Introduction	23
4.1.2	Menus in modules	24
4.1.3	Configuration	24
4.2	Serving static content	25
4.2.1	Hot modification of resources	25
4.2.2	Production vs development	26
5	Internationalization	27
5.1	Introduction	27
5.2	Design	28
5.3	Writing a localized application	28
5.3.1	Create a jar with translated messages.	29

5.3.2	Add the translated messages to the global messages files	29
5.4	Internationalizing you module	29
6	Editable pages module	31
6.1	Editable pages design	31
6.2	Using this module	31
6.3	Configuration	31
6.3.1	Configuring the site name	32
6.3.2	Configuring FCKEditor	32
6.4	The user interface	33
6.5	Next steps	33
7	Search module	35
7.1	Search design	35
7.2	How a search is performed	35
7.3	Security considerations	35
7.4	Preparing a module to index its content	36
7.5	Using this module	38
7.6	Configuration	38
7.7	The user interface	39
7.8	Modules integrating search	39
7.9	Next steps	39
8	Reports module	41
8.1	Reports design	41
8.2	Using this module	41
8.3	Design	41
8.3.1	Domain model	41
8.3.2	Entities responsibilities	42
8.4	Use case scenarios	42
8.4.1	Report upload	42
8.4.2	Report execution	43
9	Katari Social	45
9.1	Shindig	45
9.1.1	Activities	45
9.1.2	Gadget rendering proxy	45
9.2	The gadget container	45
9.3	Using this module	45
9.4	Design	46
9.4.1	Domain model	46
9.4.2	Entities responsibilities	47
9.5	Configuration	47

10 Task scheduling with quartz	49
10.1 Introduction	49
10.2 Using this module	49
11 Managing js modules	51
11.1 Js modules	51
11.2 Using this module	51
11.3 Packaging javascript	51

Chapter 1

General Overview

1.1 Welcome to Katari

Katari is a java based software platform that will help you develop web applications faster. It was born as an internal project at Globant where was used successfully for dozens of projects. We are opening it to the community and make it freely available under the Apache Software License 2.0.

Katari is built upon the most well known open source frameworks:

- Spring.
- Spring-mvc.
- Hibernate.
- Acegi security (now Spring security).
- Freemarker.
- Sitemesh.

But you don't have to spend a week or more properly integrating and configuring them. With Katari you get a sample application up and running in a couple of minutes. This sample application includes:

- Secured and non-secured content. A login page allows you to access secured content.
- A nice styled site, where all styling element are under your control.
- A security aware menu structure.
- Sample pages that can be edited on line with the help of FCKEditor.

When you start a project with Katari you will also find a strong reliance on best practices:

- Code is well documented, consistently formatted and easy to read.
- You will find testing (unit and integration) already in place, with a nice approach to integrating tests that hits the database.
- Katari modular approach will make it easier to develop guided by business driven requirements.

You can get started in Katari with the help of the provided maven archetype (yes, it uses maven). The getting started guide will give you instructions on how to start with Katari.

Chapter 2

Getting started

2.1 Starting a Katari project

To demonstrate how to start a project from scratch with Katari, We will be creating a sample application called acme-coyote. But first you need:

- java-1.5 or above.
- maven-2.2.1 or above. It will probably work from maven 2.1. Maven 2.0.x packaged an old version of jtidy classes that clashes with jetty.
- mysql-5.x.

Katari provides a maven archetype to make it easy to start a new project:

```
mvn archetype:generate -DarchetypeArtifactId=katari-archetype \
-DarchetypeGroupId=com.globant.katari -DarchetypeVersion=0.42 \
-DarchetypeRepository="http://katari.globant.com/nexus/content/repositories/globant" \
-Drepository="http://katari.globant.com/nexus/content/repositories/globant"
```

This will ask for the following information:

- the project friendly name
- the group id
- the artifact id
- the project version.

For consistency, we recommend the groupId is of the form *your domain.artifactId*.

You can also create the project in batch mode, passing all the information in the command line. To see it in action, lets create the coyote application for our client acme:

```
mvn archetype:generate -DarchetypeArtifactId=katari-archetype \
-DarchetypeGroupId=com.globant.katari -DarchetypeVersion=0.42 \
-DarchetypeRepository="http://katari.globant.com/nexus/content/repositories/globant" \
-Drepository="http://katari.globant.com/nexus/content/repositories/globant" \
-DinteractiveMode=false \
-DfriendlyName="Acme tools for the coyote" \
-DgroupId=com.globant.acme.coyote -DartifactId=acme-coyote
```

This creates a full fledged application with the following features:

- Menu management.
- Security.
- A login page.
- A report management module.
- A module that allows users to edit html pages.

(Note: if you are behind a proxy, you may be hit by bug <http://jira.codehaus.org/browse/ARCHETYPE-202>: maven will freeze for a minute or two trying to download archetype-catalog.xml. Add `-DarchetypeCatalog=local` to speed things up.)

Once the archetype finishes, you will find a directory called `acme-coyote`. There you will find:

```
pom.xml
dev
|-- acme-coyote-style
|   |-- src
|   |   |-- main
|   |   |-- site
|   |-- pom.xml
|-- acme-coyote-web
|   |-- src
|   |   |-- main
|   |   |   |-- java
|   |   |   |-- resources
|   |   |   |-- sql
|   |   |   |-- webapp
|   |   |       |-- index.jsp
|   |   |       |-- WEB-INF
|   |   |           |-- applicationContext.xml
|   |   |           |-- applicationContextRuntime.xml
|   |   |-- web.xml
|   |-- site
|   |-- pom.xml
|-- src
|   |-- main
|   |   |-- assemble
|   |   |-- config
|   |-- site
|   |   |-- apt
|   |   |   |-- index.apt
|   |   |-- *.apt
|   |-- site.xml
|-- pom.xml
|-- README
```

The structure follows Globant CMMi standards, with a `dev` top level module. Below that, the archetype creates two modules: `web` and `style`.

The `dev/README` provides the necessary information to getting started with the project. So follow the instructions under the 'Quick start guide' title and you will have a running Katari application hitting a database.

When you run `jetty`, it will tell you in which port it is running (usually port 8088). So open in the browser `http://localhost:8088/acme-coyote-web`, and you will see the home page for anonymous users.

If you click the login link, you can log in using `admin` for the username and `admin` for the password, as show in the login page.

After a successful login you will see the katari home page for logged in users. You notice that you now have additional menus, all accesible to the roles that the admin user has. If you now navigate to the home

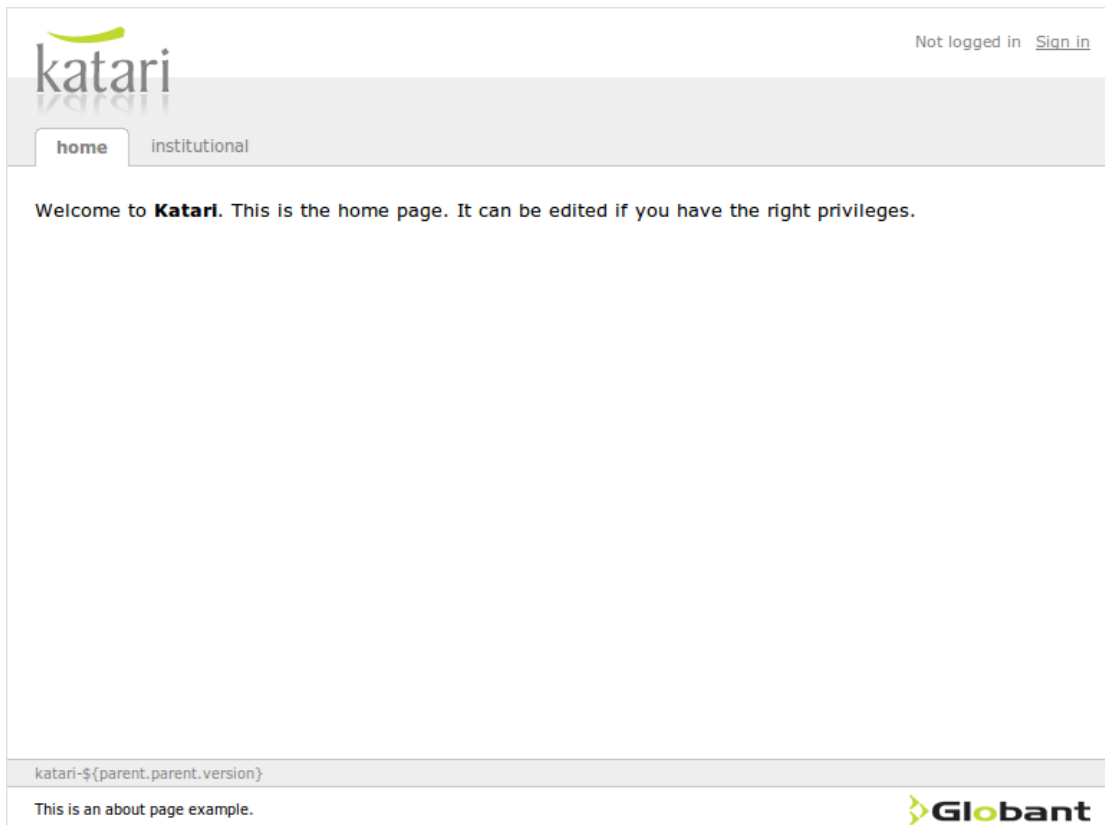


Figure 2.1: Katari home page for anonymous users.

page, you will also see a new icon at the top right corner. This icon, shown when you have the correct roles, allows users to edit the page.

Now you have access to new menu entries due to the administrator privileges granted to admin.

2.2 Creating your first module

We will now create a module named `hello` in the previous project. Katari provides a maven archetype to create modules. Go to the dev directory created in the previous step and run:

```
mvn archetype:generate -DarchetypeArtifactId=katari-module-archetype \
  -DarchetypeGroupId=com.globant.katari -DarchetypeVersion=0.42 \
  -DarchetypeRepository="http://katari.globant.com/nexus/content/repositories/globant" \
  -Drepository="http://katari.globant.com/nexus/content/repositories/globant" \
  -DinteractiveMode=false \
  -DfriendlyName="Acme tools for the coyote - hello" \
  -DgroupId=com.globant.acme.coyote \
  -DartifactId=acme-coyote-hello -DmoduleName=hello
```

This creates a new maven module under the directory `acme-coyote-hello`. This directory now contains a maven project with `acme-coyote-hello` artifactId. The directory structure created by the archetype is:

```
acme-coyote-hello
|-- pom.xml
`-- src
    |-- main
    |   |-- java/com/globant/acme/coyote/hello/view
```




Figure 2.3: Katari home page for logged in user.

```

http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.globant.com/schema/katari
http://www.globant.com/schema/katari/katari.xsd">

```

```

<bean id="hello.module" class="com.globant.katari.core.web.ConfigurableModule">
  <property name="entryPoints">
    <!--
      The servlet mappings. It maps the spring DispatcherServlet to *.do.
    -->
    <bean class="org.springframework.beans.factory.config.MapFactoryBean">
      <property name="sourceMap">
        <map>
          <!-- The spring-mvc controller servlet -->
          <entry key=".*\.do">
            <bean class="com.globant.katari.core.web.ServletAndParameters">
              <constructor-arg index="0">
                <bean class="com.globant.katari.core.web.DispatcherServlet" />
              </constructor-arg>
              <constructor-arg index="1">
                <map>
                  <entry key="contextConfigLocation">
                    <value>
                      classpath:/com/globant/acme/coyote/hello/view/spring-servlet.xml
                    </value>
                  </entry>
                </map>
              </constructor-arg>
            </bean>
          </entry>
        </map>
      </property>
    </bean>
  </property>
</bean>

```

```

        </entry>
    </map>
</property>
</bean>
</property>
</bean>
</beans>

```

This file is a standard spring bean definition file. It must contain a bean named *module-name.module*, of a type that implements Module. The most common implementation is ConfigurableModule, that is a simple pojo with setters for all the properties.

This looks complex, but is simply a servlet configuration equivalent to what you would do in web.xml, but in spring style. This entryPoints property of the hello.module bean is a map of regular expressions to servlets. This example maps all requests ending in .do to the DispatcherServlet of spring. This DispatcherServlet is configured with the servlet parameter contextConfigLocation set to the full classpath to spring-servlet.xml. The spring-servlet.xml file is the standard bean definition file that contains the spring controllers.

When a user makes a request to *context-path/module/hello/whatever.do*, this request is routed to the hello module, the *whatever.do* fragment is matched with the defined entry points for that module and finally routed to the corresponding servlet.

This module declares a spring dispatcher servlet configured in:

```
src/main/resources/com/globant/acme/coyote/hello/view/spring-servlet.xml
```

This file declares a spring controller (HelloController) that listens to the /hello.do request:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:katari="http://www.globant.com/schema/katari"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.globant.com/schema/katari
    http://www.globant.com/schema/katari/katari.xsd">

  <!-- freemarker config -->
  <bean id="freemarkerConfig" class="com.globant.katari.core.web.FreeMarkerConfigurer"
    p:templateLoaderPath='classpath:/com/globant/acme/coyote/hello/view/'
    p:debugPrefix=' ../acme-coyote-hello/src/main/resources'
    p:debug-ref='debugMode' />

  <!-- View resolvers can also be configured with ResourceBundles or XML files.
  If you need different view resolving based on Locale, you have to use the
  resource bundle resolver. -->
  <bean id="viewResolver"
    class="com.globant.katari.core.web.FreeMarkerViewResolver"
    p:prefix="/" p:suffix=".ftl"
    p:exposeSpringMacroHelpers="true"
    p:exposeRequestAttributes="true" />

  <bean name='/hello.do'
    class='com.globant.acme.coyote.hello.view.HelloController' />
</beans>

```

As you can see, this file looks like a standard spring-mvc dispatcher servlet configuration file. The only things to notice are related to the freemarkerConfig bean. This bean is from a katari package instead of spring's. This is really optional (you can use the one provided by spring), but Katari's implementation

adds a nice feature: Katari searches ftl files from the file system if debug mode is enabled. This makes it very convenient to edit ftl and view the result in the browser on the fly. This feature is enabled with the debug and debugPrefix properties. debugPrefix is the path relative to the web application where katari will search for the ftl files.

This file must be included in your main application context, webapp/WEB-INF/applicationContext.xml. When the application loads, it finds and initializes all beans implementing the Module interface.

So, what is a module for Katari? A module is a vertical cut in an application, comprising all the typical layers: view (controllers and views, in spring jargon), application layer (facades or commands), domain layer (repositories, entities, domain services, etc, in Domain Driven Design parlance), and the persistence or integration layer (usually hibernate).

A module can be packaged in its own jar and reused in other applications. The Katari core takes care of integrating the module features.

The module archetype creates a hello world page.

Our hello world page will have the view and application layers. Create a new package for both layers (application and view) under the hello package.

The archetype creates a controller named HelloController in the view package that extends AbstractController.

This class implements handleRequestInternal and returns a ModelAndView instance for the 'hello' view:

```
package com.globant.acme.coyote.hello.view;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

/** Spring MVC controller to show a hello message.
 * /
public class HelloController extends AbstractController {

    /** Forwards the request to the hello view.
     *
     * {@inheritDoc}
     */
    @Override
    protected ModelAndView handleRequestInternal(final HttpServletRequest
        request, final HttpServletResponse response) throws Exception {
        ModelAndView mav = new ModelAndView("hello");
        return mav;
    }
}
```

The generated view, a freemarker template, is called hello.ftl:

```
<html>
  <head>
    <title>Hello</title>
  </head>

  <body>
    <h3>Hello</h3>
  </body>
</html>
```

The archetype created a new module but it is not yet registered in your application. You need to add the correct dependencies in your webapp, and register the module in katari.

To declare the dependencies, add to dev/pom.xml somewhere in the dependencyManagement section:

```
<dependency>
  <groupId>com.globant.acme.coyote</groupId>
  <artifactId>acme-coyote-hello</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

And in dev/acme-coyote-web, somewhere in the dependencies section:

```
<dependency>
  <groupId>com.globant.acme.coyote</groupId>
  <artifactId>acme-coyote-hello</artifactId>
</dependency>
```

This is just maven standard dependency management information. It is a good practice to centrally declare the versions of all dependencies in your modules, so you don't accidentally mix different versions for the same library. This is what the dependencyManagement element of dev/pom.xml is for.

To register the module in katari, open the applicationContext.xml under webapp/WEB-INF and add a line of the form:

```
<!-- The hello module. It is a simple almost empty module. -->
<katari:import module="com.globant.acme.coyote.hello"/>
```

Now, compile and run the app:

```
mvn jetty:run
```

Go to the app, login and hit /module/hello/hello.do. You will see the expected hello text.

There is no link in the app that points to the new page. Let's add a menu entry for this. Open the module.xml file and add a new property menuBar to the hello.module bean:

```
<property name='menuBar'>
  <!-- The menu bar. -->
  <katari:menuBar>
    <katari:menuNode name='Hello'>
      <katari:menuItem name='SayHello' link='hello.do' />
    </katari:menuNode>
  </katari:menuBar>
</property>
```

restart jetty (mvn jetty:run) and see the result.

But to say hello you don't want to be authenticated. Lets change that. Edit again the module.xml and add the following property to the hello.module bean:

```
<property name="urlToRoleMapper">
  <bean class="com.globant.katari.core.security.StaticUrlToRoleMapper"
    id='report.urlToRoleMapper'>
    <constructor-arg index="0">
      <map>
        <entry key="/**/*" value="IS_AUTHENTICATED_ANONYMOUSLY" />
      </map>
    </constructor-arg>
  </bean>
</property>
```

This is saying that to access all urls in the module you do not need to be authenticated (the IS_AUTHENTICATED_ANONYMOUSLY role).

Restart jetty and access the application. You will see the hello menu entry without the need to login.

Chapter 3

General Architecture

3.1 Architecture

This chapter describes Katari general architecture.

3.1.1 Modules

Katari applications are designed with modularity in mind, so the main concept in Katari is a module. Each module is a reusable independent 'vertical cut' in the functionality, spanning from the UI down to persistence.

Modules are contained in a Katari application, that serves as a module container. The module container provides common services to all modules, like request dispatching, menu registration, security infrastructure, and more.

The full stack of each module (view, application and domain) can be independently packed in a jar and reused in many applications.

The following figure gives a global overview of how a Katari application is structured.

Modules are plugged with the help of spring. There are three 'points of contact' between modules: the view, the domain and the database.

1. View

At the view layer, the application consists of a servlet that dispatches the request to the corresponding module. This servlet listens to all requests that go to *context/module*. From there, the servlet dispatches the request to the corresponding module. The modules are mapped with the following url: *context/module/module-name*.

For example, if the application is deployed at localhost:8098, the main module servlet gets all requests that go to:

```
http://localhost:8098/acme-coyote/module.
```

The user module gets mapped to the following url:

```
http://localhost:8098/acme-coyote/module/user.
```

The following figure describes how a request is dispatched to the destination servlet in a module, considering a request to `<http://localhost:8098/acme-coyote/module/user/editUser.do>`.

As you can see, the main entry point for a servlet request is the `SpringBootstrapServlet`. This is a servlet that simply passes the request to another servlet. The target server is configured in the spring application context, under a bean named, by default, `'katari.moduleContainer'`. This bean is

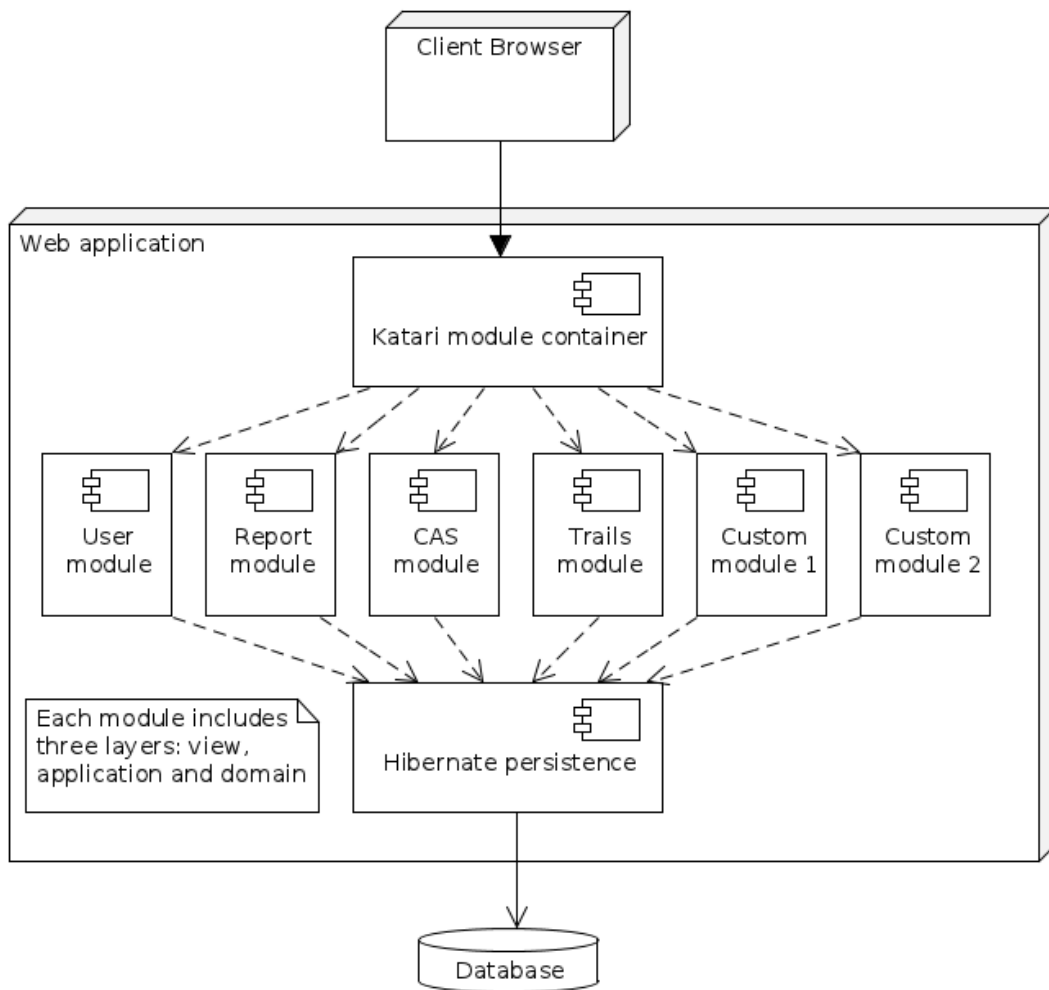


Figure 3.1: High level module view

an instance of `ModuleContainerServlet` whose responsibility is to know all the modules registered in an application and dispatch the request to the corresponding module.

2. Domain

Each module defines its own domain model. We recommend following Evans' recommendations laid out in *Domain Driven Design*. You will find entities, value objects, repositories, factories and services in modules implemented by us.

All modules share the same classloader, so a module can use any public class from another module. With the help of spring, a module can export a service that can be used by any other module.

3. Database

There is one database for all the modules. The database configuration is injected in every module through spring.

3.1.2 Module initialization

When you write a module, you need to provide a spring bean that implements the `Module` interface. You usually use an instance of `ConfigurableModule` so you don't need to implement it yourself. The module

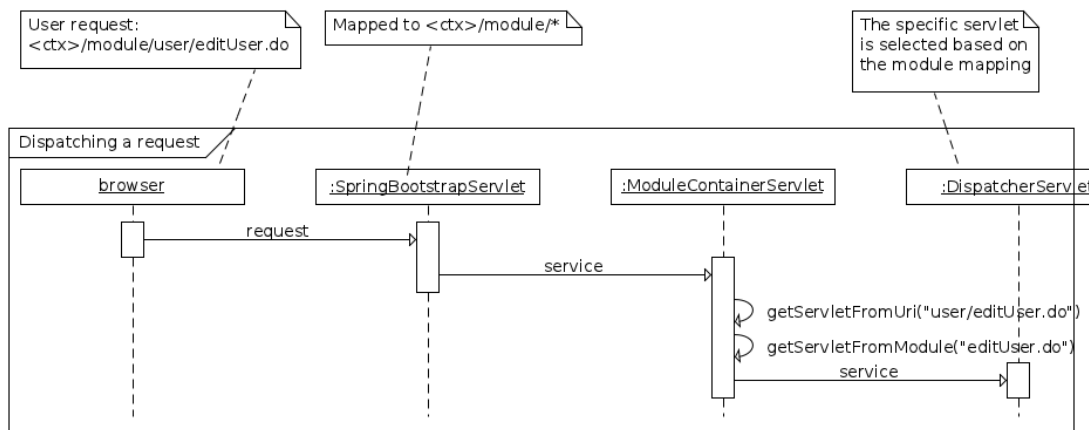


Figure 3.2: Module dispatching

interface defines:

1. An init lifecycle operation

```
void init (final com.globant.katari.core.web.ModuleContext context);
```

2. A destroy lifecycle operation.

```
void destroy();
```

When the spring application context starts up, a special Katari provided bean factory post-processor (ModuleBeanPostProcessor) takes care of initializing all the modules declared in the application context. This post-processor creates an instance of ModuleContext for each module and calls init. The ModuleContext is the point of contact between the module and its container. It allows the module to declare:

1. The module servlet and mappings (entry points).
2. The module web filter and mappings.
3. The module web context loader listeners and mappings.
4. The menu entries.
5. Security configuration.
6. The module weblets.

3.1.3 Spring configuration

As mentioned earlier, Katari uses spring as its DI framework. Katari makes heavy use of spring to split the application into modules. A normal Katari application has a main bean definition file usually called applicationContext.xml, located in the WEB-INF directory of the webapp. This file imports the module spring files, called module.xml (for example /com/globant/katari/editablepages/module.xml) with the katari:import element, and overrides some other beans to configure the modules (for example, the data source connection information.)

A typical applicationContext.xml file looks like this:

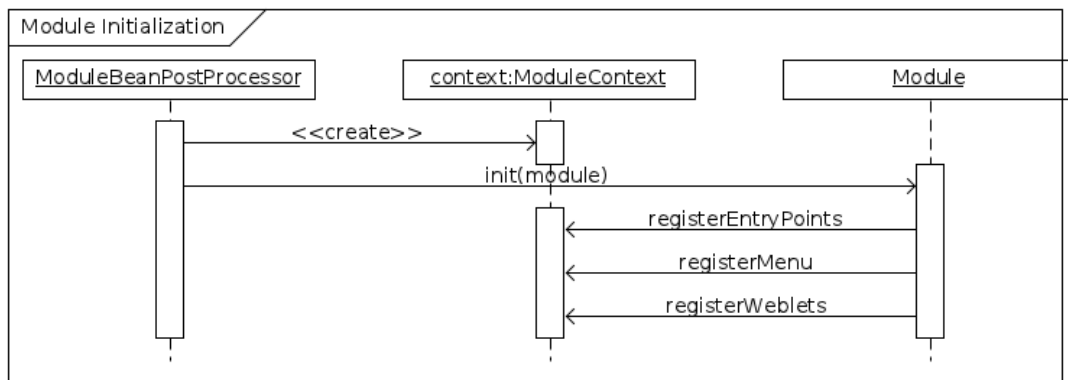


Figure 3.3: Module initialization

```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:katari="http://www.globant.com/schema/katari"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.globant.com/schema/katari
    http://www.globant.com/schema/katari/katari.xsd">

  <!-- The katari global beans. -->
  <import resource="classpath:/com/globant/katari/core/applicationContext.xml"/>

  <!-- The home module. -->
  <katari:import module="com.globant.katari.sample.home"/>

  <!-- The classic menu module. -->
  <katari:import module="com.globant.katari.menu.classic"/>

  .... more modules ....

  <!-- The hibernate properties, They are referenced from the hibernate bean. -->
  <bean id="katari.hibernateProperties"
    class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="properties">
      <props>
        <prop key="hibernate.dialect">
          org.hibernate.dialect.MySQL5InnoDBDialect
        </prop>
      </props>
    </property>
  </bean>

  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
    destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    ... more datasource properties ....
  </bean>

  <!-- The katari data source as an alias to dataSource. -->
  <alias name="dataSource" alias="katari.dataSource"/>

```

```

... more configuration beans ...

</beans>

A sample module.xml file is:

<beans ... >

  <bean id="editable-pages.module"
        class="com.globant.katari.core.web.ConfigurableModule">

    <property name='entryPoints'>
      <!-- The servlet mappings. It maps the spring DispatcherServlet to *.do.
      -->
      <bean
        ...
      </property>

  </bean

  <bean class=' com.globant.katari.core.web.ListFactoryAppender'>
    <constructor-arg value='katari.persistentClasses' />
    <constructor-arg>
      <list>
        <value>com.globant.katari.editablepages.domain.Page</value>
      </list>
    </constructor-arg>
  </bean>

  ... other beans ...

</beans>

```

Given that beans defined in module.xml files share the same spring bean factory, every bean must comply to a strict naming convention:

- Beans defined by katari start with "katari." prefix.
- Beans defined by modules start with "module."

This convention guarantees that beans do not accidentally override each other.

3.1.4 Currently supported modules

Katari includes the following modules:

katari-ajax A module that exposes yui client-side widgets.

katari-classic-menu Provides simple button menu bars.

katari-report Allow users to upload and manage jasper reports.

katari-cas Integrates CAS as a single sign-on for authentication with all the services within the application.

katari-local-login Integrates the normal ACEGI login with a local database as means of authentication.

There are also a couple of sample modules that are included in the main web application:

- The **style** module: provides a unified look and feel for all the modules in the application.

- The user module: this is basically a user and role crud module.
- The time reporting module: this is a sample module that serves as the core end user functionality in the basic example.

3.1.5 Layers

Katari modules use a traditional layered model, consisting of the following four layers:

View It includes the servlet based front controllers (like spring-mvc controllers or struts actions) and the view templates (like freemarker templates).

Application The application layer adapts the client requests performed through the view to the domain. This layer is responsible of converting the request data (usually strings or ids) to domain objects, delegate the request to the corresponding object in the domain and transaction demarcation. The Katari samples and modules use a command based approach in this layer.

The view layer is permitted to call just one method per request en the application layer, plus additional idempotent methods when rendering the view.

Domain Where the main business logic is implemented.

Persistence The persistence layer consists of hibernate without any wrapping code. You should not see any package related to this layer in the project.

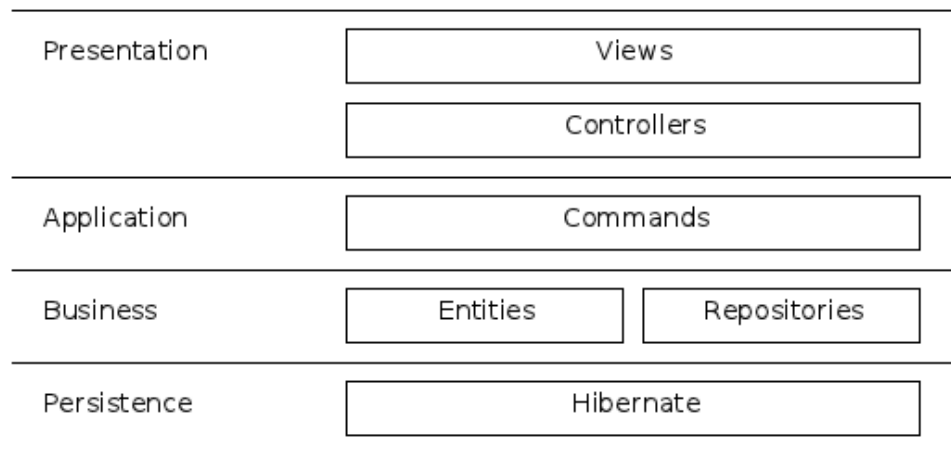


Figure 3.4: The Katari layers

The view and the application layer depends on the domain layer: classes from the domain layer can be imported in the view and/or application layers.

Katari modules do not need to follow this layering schema, but it is our recommended approach.

3.2 Tools and libraries

Katari integrates the following libraries and tools:

3.2.1 Core libraries

Acegi Security Acegi Security is an open source project that provides comprehensive authentication and authorisation services for enterprise applications. Acegi Security can authenticate using a variety of pluggable providers.

<http://www.acegisecurity.org>

Spring Spring is a layered Java/J2EE application platform for inversion of control. It includes a lightweight container, an abstraction layer for transaction management and integration with persistence frameworks.

<http://www.springframework.org>

Spring MVC Spring MVC is a flexible and highly configurable web application framework, which accommodates multiple view technologies.

<http://www.springframework.org>

Hibernate Hibernate is a powerful, high performance object/relational persistence and query service.

<http://www.hibernate.org>

Freemarker FreeMarker is a "template engine"; a generic tool to generate text output (anything from HTML to autogenerated source code) based on templates.

<http://freemarker.org>

SiteMesh SiteMesh is a web-page layout and decoration framework to aid in creating large sites consisting of many pages for which a consistent look/feel is required.

<http://opensymphony.com/sitemesh>

3.2.2 Frameworks

CAS CAS (Central Authentication Server) is an authentication system originally created by Yale University to provide a trusted way for an application to authenticate a user.

<http://www.ja-sig.org/products/cas>

3.2.3 Support libraries

Log4j Log4j allows the developer to control which log statements are output with arbitrary granularity.

<http://logging.apache.org/log4j>

Jakarta commons The Commons is an Apache project focused on all aspects of reusable Java components.

<http://commons.apache.org>

Easymock EasyMock provides Mock Objects for interfaces in JUnit tests by generating them on the fly using Java's proxy mechanism.

<http://www.easymock.org>

JUnit JUnit is a simple framework for writing and running automated tests.

<http://www.junit.org>

3.2.4 Support tools

Maven Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

<http://maven.apache.org>

Chapter 4

Features provided by Katari

4.1 Menu support

4.1.1 Introduction

Katari has support for declarative menus. Modules declare menu entries that Katari merges together. The menus can be defined using a special syntax in spring.

The responsibility of drawing a menu is delegated to a module, it is not part of Katari's core. The module `katari-classic-menu` is one possible module that draws the menu in the page.

This is the structure of a menu declaration:

```
<katari:menuBar name='menubar-name' >

  <katari:menuNode name='menu-name' display='display text'
    tooltip='tooltip text' >

    <katari:menuItem name='menu-name-1' display='display text'
      tooltip='tooltip text' link='entry-1.do' />

    <katari:menuItem name='menu-name-2' display='display text'
      tooltip='tooltip text' link='entry-2.do' />

  </katari:menuNode>

</katari:menuBar>
```

It consists of a hierarchy of `menuNode` elements with `menuItem` elements, that can appear anywhere in the hierarchy. The attributes of each element are:

- **name:** it applies to elements `menuNode` and `menuItem`. This is a name that identifies the node in the current level. It cannot contain spaces.
- **display:** it applies to elements `menuNode` and `menuItem`. This is the text that appears as the label of the menu. This is optional. If not specified, Katari uses the menu name.
- **tooltip:** an option popup help text for the menu. It applies to elements `menuNode` and `menuItem`.
- **link:** only applies to `menuItem`. This is the link of the menu. It is relative to the module home. This link can contain `$variablename`, that get replaced when the link is generated. The currently supported variables are of the form `$modulename.module` (`modulename` is the name of some other module), to create links to other modules.

Each menu entry is only shown only if the user has access to the corresponding link.

4.1.2 Menus in modules

As said in the previous section, modules can add menu items to the global menu bar. This global menu bar is simply a katari menu bar declared in spring application context under the name 'katari.menuBar'.

By default, katari provides an empty menu bar. You can also define your own. Each module can add additional entries by setting the menuBar property in it's module definition:

```
<bean id="user.module" class="com.globant.katari.core.web.ConfigurableModule">
  ...
  <property name='menuBar'>
    <!-- The menu bar. -->
    <katari:menuBar>
      <katari:menuNode name='Administration'>
        <katari:menuNode name='Users'>
          <katari:menuItem name='List' link='users.do' />
          <katari:menuItem name='New' link='createUser.do' />
        </katari:menuNode>
      </katari:menuNode>
    </katari:menuBar>
  </property>
  ...
</bean>
```

When you include this module in your application context, katari merges the module's menu bar with the 'katari.menuBar' bean.

4.1.3 Configuration

Katari menus can be configured by deciding what modules can add menu entries to the global menu bar. To this end, add:

```
<util:list id='katari.moduleMenusToIgnore'>
  <value>user</value>
</util:list>
```

Each entry in the list is a regular expression that, if matches a module name, makes the menu bar for that module ignored by katari. You can exclude the menu bars from all modules with:

```
<util:list id='katari.moduleMenusToIgnore'>
  <value>.*</value>
</util:list>
```

This makes katari ignore all menu bars from the modules, and gives you total control on the content of your menu bar:

```
<katari:menuBar name='katari.menuBar'>

  <katari:menuNode name='menu-name' display='display text'
    tooltip='tooltip text'>

    <katari:menuItem name='menu-name-1' display='display text'
      tooltip='tooltip text' link='entry-1.do' />

    <katari:menuItem name='menu-name-2' display='display text'
      tooltip='tooltip text' link='entry-2.do' />

  </katari:menuNode>

</katari:menuBar>
```

4.2 Serving static content

Katari has a servlet that can serve static content from the classpath. With this servlet, you can serve javascript, css, images, html, anything that is not modified while the application is running.

The servlet is `com.globant.katari.core.web.StaticContentServlet`. It can be used in your module with:

```
<entry key="/asset/*.*">
  <bean class="com.globant.katari.core.web.ServletAndParameters">
    <constructor-arg index="0">
      <bean class="com.globant.katari.core.web.StaticContentServlet"/>
    </constructor-arg>
    <constructor-arg index="1">
      <map>
        <!-- WARNING: Be careful with the value staticContentPath:
        if it starts with '/' it won't work under Tomcat.
        -->
        <entry key="staticContentPath"
          value="com/globant/katari/style/asset"/>
        <entry key="debug" value-ref='debugMode' />
        <entry key="debugPrefix"
          value="../katari-style/src/main/resources"/>
        <entry key="requestCacheContent" value="true"/>
        <entry key="mimeType_png" value="image/png"/>
        <entry key="mimeType_js" value="text/javascript"/>
      </map>
    </constructor-arg>
  </bean>
</entry>
```

This example shows how to configure the servlet to serve png and js resources. When the client performs a request to `[your module]/asset/script.js`, the static content servlet will search for a file named `script.js` under the classpath entry `com/globant/katari/style/asset` and send it to the client with the mime type `text/javascript`.

To protect the application from serving unexpected content (like a `.class` file), this servlet only serves files that end in one of the specified mime types.

4.2.1 Hot modification of resources

There are two additional parameters: `debug` and `debugPrefix`. These parameters enable developers to edit static resources without the need to redeploy the application, even if the resource is served from a different module.

The standard directory layout in Katari is a flat structure, one artifact per directory. So all the jars that compose your application are siblings of the web artifact. With this assumption, the static content servlet can be configured to load the resources directly from the file system, from a path relative to the war:

For example:

```
dev
  katari-classic-menu
  katari-style
  katari-sample
```

With this layout, the static content servlet from `katari-style` can be configured to load the resources from file setting `debugPrefix` to `../katari-style/src/main/resources`, so that all modifications to resources under the style module can be immediately seen from the web application.

4.2.2 Production vs development

If you paid attention to the configuration, the debug parameter was set with:

```
<entry key="debug" value-ref='debugMode' />
```

This means that the value of debug is set from another bean, called debugMode. This bean is defined by Katari. It is a string wrapper that wraps, by default a false value. The katari.properties file located in test/resources overrides this bean setting the debug mode to true. Be sure not to copy this file into production environments.

Chapter 5

Internationalization

5.1 Introduction

When an application needs to be prepared for different cultures (countries, languages, etc) we are talking about internationalization (i18n) and localization (l10n).

- i18n means modifying or designing the program so that it can handle potentially multiple languages.
- l10n means adding a new language.

We don't want to go to an in depth discussion of internationalization here, but these are the main aspects to take into account when you want your application translated to many languages:

- Deployment: will there be one installation per language? Or one installation must support all the languages?
- Content scope: what will be translated? Static pages, user modifiable content, both?
- Translation process: what is the process by which a new language is integrated in the system?
- Culture scope: which languages are intended to be supported? Latin based? Oriental languages? Right to left? Must images be translated due to cultural differences? What about colors?
- Language selection: what mechanisms will be used to allow the final user to change the language?

Once these (and surely many more) are taken into account, the i18n process must also consider all the technical aspects, of which the most important is related to the byte vs string difference: you must be sure that bytes are correctly translated to strings and viceversa. Under java, this issue surfaces when the application has to move data to or from the virtual machine so you must configure the encoding to use in such translations. Places where you must pay attention:

- The database must be correctly configured. This includes the connection string, that under some drivers it specifies the encoding to use.
- Reading data from the user's browser.
- Writing data to the user's browser.
- Sending an email.
- Reading or writing to a file.
- Property files.

5.2 Design

The way Katari chooses to implement internationalization allows flexibility in many aspects:

- Each module writer can decide to support i18n or not. This will not be enforced by Katari.
- Katari uses mainly resource messages and freemarker templates. The freemarker templates are used in sitemesh and in spring-mvc. Localization is uniform, ie, the localization of a sitemesh template and a spring-mvc page are similar: you can use @message macro in spring-mvc managed views and in freemarker decorators.
- A module can be i18n but be localized to just one locale. But it is possible to add additional locales without rebuilding the module, just adding a new jar with the translation.
- If you are building a web application, you can override any message you choose in any locale.
- Katari provides a way to let users choose which locale to use.
 - Resources can be loaded from the file system, as supported today by Katari for static content. This allows you to see your translation changes in debug mode with just a refresh in your browser.

5.3 Writing a localized application

If you want to internationalize your application, you must decide what locales you will support and which is the default one. The default locale is used when the user provides a locale that you not support.

To declare the supported locales, add this to your main application context (usually in src/main/webapp/WEB-INF/applicationContext.xml in your web module):

```
<util:set id='katari.supportedLocales'>
  <value>en</value>
  <value>es</value>
</util:set>
```

Of course, you can add as many locales as you want.

The default locale is configured in the same applicationContext file:

```
<bean id="katari.defaultLocale" class="java.util.Locale">
  <constructor-arg index='0' value='en' />
</bean>
```

This would be enough if all the modules you use already support the languages you want.

To select a language, just add lang=en (or any other locale) to any request. A filter will pick that parameter and change the locale accordingly. You can add a locale selection input in any form in your application, and the locale chosen by the user will be stored in a cookie. If the locale selected is not in the list of supported locale, Katari will pick your default locale.

If a module in your application has already been internationalized, but it has not been translated to a language you need, you have two choices:

- Create a jar file with the translated messages.
- Add the translated messages to the global messages files (see below).

5.3.1 Create a jar with translated messages.

If you have to translate a module you do not own, this is the most 'community friendly' way: create a jar file with the translated messages. You need to know what is the base name for the messages file of the module you are translating (this base name is, by convention, *module-package-name/lang/messages*). Look in the module documentation for this base name. Now, translate the main properties file of the module and place it in:

```
<module-package-name>/lang/<locale>/messages.properties
```

For example, suppose you want to add the portuguese translation for the katari-local-login module. Place the translated file in the package:

```
com/globant/katari/login/local/lang/pt/messages.properties
```

Create a jar named katari-local-login-lang_pt (it could be any other name, but this is a good one), with this one file and add it as a dependency of your web application.

One complication of this approach is that loading the translation from the file system in debug mode does not work. You will need to build the jar and restart your web application any time you want to see the effect of a change in the translation.

5.3.2 Add the translated messages to the global messages files

Every web application may have its own message source. Katari searches for translated messages first in this message source. This allows application writers to override any message from any module, and provide in their application translations that are not yet available in the module.

This message source usually references properties files in the WEB-INF/lang directory of the war file.

A sample message source, defined in the applicationContext.xml of the webapp, is:

```
<!-- Overrides katari message source. -->
<bean id="katari.messageSource"
      class="com.globant.katari.core.spring.KatariMessageSource"
      p:basename="WEB-INF/lang/messages"
      p:debugPrefix='../katari-sample/src/main/webapp'
      p:debug-ref='debugMode' >
  <constructor-arg index='0' value='en' />
</bean>
```

All attributes are self-explained. The debugPrefix is the prefix to add to the base name to find the properties files in the file system in debug mode. This allows developers to see the result of their work without restarting the application.

The constructor argument is the default locale to select in case no locale was provided by the client.

Next, you need to know the message keys you need to translate. You need to find in the original module the corresponding properties files. Then add to your WEB-INF/lang/messages_locale.properties all the keys prefixed with the module name. For example, katari-local-login has a message key username.label. To translate to portuguese, add a key:

```
local-login.username.label= ....
```

to WEB-INF/lang/messages_pt.properties

5.4 Internationalizing you module

Depending on your module, you will need to provide translations for your spring mvc components or for components outside spring mvc (like menu entries).

For the components outside spring mvc, you need to register a KatariMessageSource for your module:

```

<property name="messageSource">
  <!-- Message source for this module, loaded from localized "messages_xx"
  files.
  Note: as this message source is not named messageSource, it is not used as
  the default message source for the application context. -->
  <bean id="local-login.messageSource"
    class="com.globant.katari.core.spring.KatariMessageSource"
    p:basename="classpath:/com/globant/katari/login/local/lang/messages"
    p:debugPrefix='../katari-local-login/src/main/resources'
    p:debug-ref='debugMode'>
    <constructor-arg index='0' value='local-login' />
    <constructor-arg index='1' ref='katari.messageSource' />
  </bean>
</property>

```

This goes inside your ConfigurableModule bean.

For spring mvc, you need to add something similar to your spring-servlet.xml:

```

<!-- Message source for this context, loaded from localized "messages_xx"
files -->
<bean id="messageSource"
  class="com.globant.katari.core.spring.KatariMessageSource"
  p:basename="classpath:/com/globant/katari/login/local/lang/messages"
  p:debugPrefix='../katari-local-login/src/main/resources'
  p:debug-ref='debugMode'>
  <constructor-arg index='0' value='local-login' />
  <constructor-arg index='1' ref='katari.messageSource' />
</bean>

```

Future versions of katari will allow you to reuse the same bean definitions for both cases. For now, just define to beans that use the same base names for the message files.

The previous examples were stolen from katari-local-login. Change the values for your module.

Now create your properties files for the locales you support.

Chapter 6

Editable pages module

6.1 Editable pages design

This module provides pages that can be edited by a user with the appropriate permissions.

The main concept in this module is the page. A page belongs to a site. Pages have a name and a content. They can be referenced from other pages using [content-path]/module/editablepages/[page-name], or from other editable pages simply using the page name.

The module has two modes: a user mode and an editor mode. The user mode is the standard way of navigating through the pages. The editor mode shows a toolbar where editors can edit, create, remove, modify, revert and publish pages.

When an editor creates or modifies a page, the new content is only visible in editor mode. The editor must publish the page to be viewable by the general public.

For editors, the module also provides a list of pages with their status. This is useful to identify 'dirty' pages (pages with modifications not yet published).

There are two special roles relevant to this module: EDITOR and PUBLISHER. To create, edit or remove a page, a user must have the EDITOR role. The PUBLISHER role is necessary to publish a page. Users with the ADMINISTRATOR role can perform both types of activities.

If the search module is available, this module registers the pages to be indexed by the search module.

6.2 Using this module

To include this module in a Katari based application, simply put a line of the form:

```
<katari:import module="com.globant.katari.editablepages"/>
```

This module also provides a weblet that can be used to show editable page fragments. For this, use:

```
<@katari.weblet "editable-pages" "page" "[name]"/>
```

where [name] is the name of the page you want to include.

6.3 Configuration

There are two configuration options: the site name and the FCKEditor configuration.

6.3.1 Configuring the site name

The site name under a collection of related pages can be stored. With this option you can have multiple sites on the same database under different tomcat instances.

To set the site name, define a bean in your main application context (after the module import) like the following:

```
<bean id="editable-pages.siteName"
      class="com.globant.katari.core.spring.StringHolder"
      p:value='mySite' />
```

6.3.2 Configuring FCKEditor

Most of FCKEditor configuration option can be set in a javascript file that contains entries of the form:

```
FCKConfig.ImageUploadAllowedExtensions = ".(jpg|gif|jpeg|png)$" ;
```

This javascript file must be accessible from the editor page. To configure the location of this javascript file you must define a bean in your application context like this:

```
<bean id='editable-pages.fckEditorConfiguration'
      class='com.globant.katari.editablepages.view.FckEditorConfiguration'
      p:configurationUrl='/module/mymodule/asset/fckconfig.js'
/>
```

You can customize other aspects of the editor setting the following properties in the previous spring bean:

```
<bean id='editable-pages.fckEditorConfiguration'
      class='com.globant.katari.editablepages.view.FckEditorConfiguration'
      p:configurationUrl='/module/mymodule/asset/fckconfig.js'
      p:editorAreaCss='/module/decorator/css/fck-editorarea.css'
      p:toolbarSet='myToolbar'
      p:width='100%'
      p:height='500' />
```

editorAreaCss: The url for the css to style the content of the editor. This url must be relative to the context path. If you don't specify this parameter, the editor loads /module/decorator/css/fck-editorarea.css. Make sure to make this file available in that location if you do not change the editorAreaCss parameter.

configurationUrl: The url (relative to the context path) for the fck config javascript file. See the FckEditor documentation related to the configuration file (fckconfig.js).

toolbarSet: The toolbar available to use in the editor. The possible toolbars are defined in the js file under configurationUrl. If that parameter is not defined, the only toolbar available is called EditablePagesMain.

height: The height of the area containing the editor.

width: The width of the area containing the editor.

See the javadoc for FckEditorConfiguration in the katari-editablepages module for more information on configuration options.

6.4 The user interface

If a user has one of the relevant roles, the module automatically goes into editor mode. In this mode, each page is shown with a pencil icon in its upper left corner. Clicking in this pencil shows the editor toolbar.

As a future enhancements, we can provide an 'edit mode' button that shows the 'pencil' menu and the yet to be published content.

6.5 Next steps

The current integration with the search module shows the first 100 characters of the page content, including html tags. This should be revisited to extract only the text data from the html, and possibly integrate the compass highlighter.

Chapter 7

Search module

7.1 Search design

This module provides a site-wide, security aware, search facility with the help of compass, an indexing and search engine based on lucene that can be easily integrated with hibernate.

The aim of this module is to provide final users a way to search for content across the whole application. Users enter a search condition and the module shows a list with the result of the search. Users can interact with each of the elements in the result, performing different actions. There is a default action (view), and modules can add additional actions.

For example, if the client searches for a user name, the resulting list shows the name of the user, some details of the user, a link to show the full information of the user, and a link allowing the client to modify the user.

7.2 How a search is performed

The index module will find objects, but will not know how to handle them. It doesn't know anything about what it is being indexed, other modules provide the content. Each module must provide an adapter that wraps the result of the search in an object that holds all the information that the search module needs to present the result of the search on the screen.

The full process is shown in the following diagram

The main entry point for the search process in the application layer is the SearchCommand. This command is created by spring and initialized by the spring controller with the query posted by the user. On the execute operation, the search command delegates all the work to the IndexRepository.

The IndexRepository knows how to perform the search, with the help of compass. Once compass returns all the objects that matched the query, the IndexRepository finds, for each resulting object, the corresponding SearchAdapter instance. Those SearchAdapter instances are provided by the modules, and know how to create a SearchResultElement from the domain object found by compass. This SearchResultElement has all the information needed by the search module to show the result of the search.

Finally, the IndexRepository returns to the command a SearchResult, that contains a list of SearchResultElement instances.

7.3 Security considerations

The search module has implemented a simple security schema: it will only return to the user the elements of types that the user is allowed to view. To determine which elements the user is allowed to view, modules need to implement the getViewUrl operation in the SearchAdapter interface. This url must be

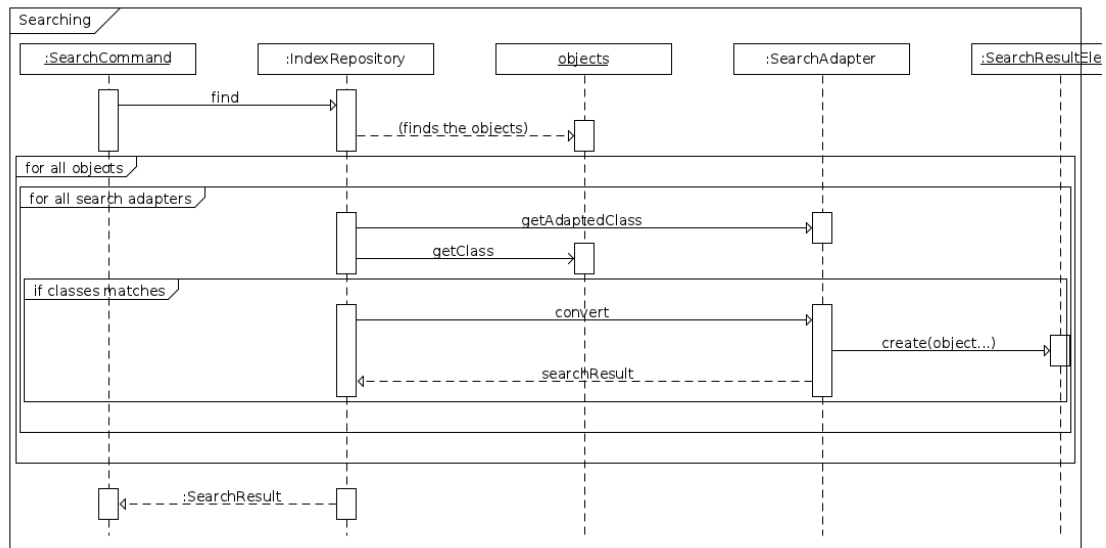


Figure 7.1: How a search is performed

independent of the specific object that will be viewed. The search module will use katari standard security mechanism (using acegi security) to check if the logged on user has access to that url.

7.4 Preparing a module to index its content

A module that wants to integrate with the search module must provide two things:

1. A list of entities that must be indexed.
2. A list of adapters that adapt each entity to the requirements of the search module.

The indexed entities must be mapped by hibernate, and annotated with compass indexing annotations (for example, `Searchable`, `SearchableId` and `SearchableProperty` in the package `org.compass.annotations`).

This is a fragment of an annotated entity:

```

@Entity
@Table(name = "users")
@Searchable
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", nullable = false)
    @SearchableId
    private long id = 0;

    @Column(name = "name", nullable = false, length = 50)
    @SearchableProperty
    private String name;

    @Column(name = "email", nullable = false, length = 50)
    @SearchableProperty
    private String email;
  
```

```
.  
. .  
}
```

And this is a sample adapter for the previous object:

```
public class UserSearchAdapter implements SearchAdapter {  
  
    public SearchResultElement convert(final Object o, final float score) {  
  
        User user = (User) o;  
  
        ArrayList<Action> actions;  
        actions = new ArrayList<Action>();  
  
        actions.add(new Action("Edit", null,  
            "userEdit.do?id=" + user.getId()));  
  
        StringBuilder description = new StringBuilder();  
        description.append("User - name: " + user.getName());  
        description.append("; email: " + user.getEmail());  
  
        return new SearchResultElement("User", user.getName(),  
            description.toString(), "/module/user/user.do?id=" + user.getId(),  
            actions, score);  
    }  
  
    public String getViewUrl() {  
        return "/module/user/user.do";  
    }  
  
    public Class getAdaptedClass() {  
        return User.class;  
    }  
}
```

As you can see, the adapter provides:

- A view url, for the specific object, returned in the convert operation.
- A description of the object, suitable to be show to the user.
- A list of urls representing actions that the user can perform (for example, edit).
- A generic view url. This is used to check if the user can access the type of object.
- Which class this adapter adapts.

The last step to index the module content is to register the objects and the adapters in the search module. For that, add the following lines in your module.xml:

```
<bean class='com.globant.katari.core.web.ListFactoryAppender'>  
    <constructor-arg value='search.indexableClasses' />  
    <constructor-arg value='true' />  
    <constructor-arg>  
        <list>  
            <value>com.globant.katari.myModule.domain.User</value>  
        </list>  
    </constructor-arg>
```

```

</bean>

<bean class=' com.globant.katari.core.web.ListFactoryAppender' >
  <constructor-arg value=' search.adapters' />
  <constructor-arg value='true' />
  <constructor-arg>
    <list>
      <bean class="com.globant.katari.myModule.domain.UserSearchAdapter"/>
    </list>
  </constructor-arg>
</bean>

```

These two beans add the domain object (User) and its corresponding adapter (UserSearchAdapter) to the search.indexableClasses and search.adapters collections, defined in the search module. In this case, if the search module is not installed, nothing will happen. This is indicated by the 'true' value as the second constructor parameters in both beans.

7.5 Using this module

To include this module in a Katari based application, simply put a line of the form:

```
<katari:import module="com.globant.katari.search"/>
```

Importing module.xml will automatically index and search on all the entities provided by other modules that support indexing.

This module also provides a weblet that can be used to show the search input text and its button. To use it, add the following to your decorator (usually header.dec):

```
<@katari.weblet "search" "search" />
```

7.6 Configuration

The only configurable options provided by this module are the location of the main and temporary lucene indexes. This is the default configuration:

```

<!-- Overridable bean to define the file system location to store the lucene
index. -->
<bean id=' search.compassEngineConnection'
  class=' com.globant.katari.core.spring.StringHolder'
  p:value='target/compass/' />

```

It is recommended to make this location configurable from the properties file. To do this, add to your applicationContext.xml:

```

<!-- Externalizes the location of the compass files for mirroring and search.
-->
<bean id=' search.compassEngineConnection'
  class=' com.globant.katari.core.spring.StringHolder'
  p:value-ref=' indexLocation' />

<!-- The value of this bean is intended to be overridden in the properties
file. -->
<bean id=' indexLocation'
  class=' com.globant.katari.core.spring.StringHolder'
  p:value='target/compass' />

```

Compass can perform search and mirroring separately from batch indexing. The `search.compassEngineConnection` configuration defines the place where compass performs the search and mirroring operations. The `search.compassIndexEngineConnection` is used for batch indexing. Once the indexing process is finished, compass copies the new index to `search.compassEngineConnection`.

The default location for the index operation is `target/compass-tmp`. As with `compassEngineConnection`, it is recommended to make this property configurable from the properties file:

```
<!-- Externalizes the location of the compass files for indexing. -->
<bean id=' search.compassIndexEngineConnection'
      class=' com.globant.katari.core.spring.StringHolder'
      p:value-ref=' indexTempLocation' />

<!-- The value of this bean is intended to be overridden in the properties
      file. -->
<bean id=' indexTempLocation'
      class=' com.globant.katari.core.spring.StringHolder'
      p:value=' target/compass-temp' />
```

These bean definitions (`compassEngineConnection` and `compassIndexEngineConnection`) are adequate to be configured by a spring `PropertyOverrideConfigurer`. So you can add to your property file entries like this:

```
indexLocation.value=/home/tomcat/target/index
indexTempLocation.value=/home/tomcat/target/index-temp
```

7.7 The user interface

The entry point of the search module is the previous weblet. This weblet shows an input field and a search button. When you click on the search button, you are taken to the search results page. This page contains the same input field and search button, plus a list with the result of the search.

Each element of the list has a link to the 'view' location, and an optional list of actions.

7.8 Modules integrating search

For now, the sample user module and the editable pages module index their content through this module. More to come ...

7.9 Next steps

There are still a couple of things that would be nice to add here:

- Integrate the highlighter.
- Define a more fine grained security model, for example, define roles per object, an owner, etc.
- Provide support to index things that are not hibernate mapped domain objects.
- Improve the search result page.

Chapter 8

Reports module

8.1 Reports design

The report module for Katari allows the user to build and execute reports. The report is built using a visual tool and then uploaded into Katari, to be executed from a web interface on a later date.

The reports may be generated using the WYSIWYG tool iReports, where the designer can customise the look and feel of the report. The query to generate the report is written using SQL language inside iReports.

The report file is uploaded into Katari using a web interface. The user is prompted to map the types used in the report parameters, so they can be displayed correctly in a web form.

At a later time, a user with the correct role (specified upon the report creation) can execute the report and print the output into a PDF.

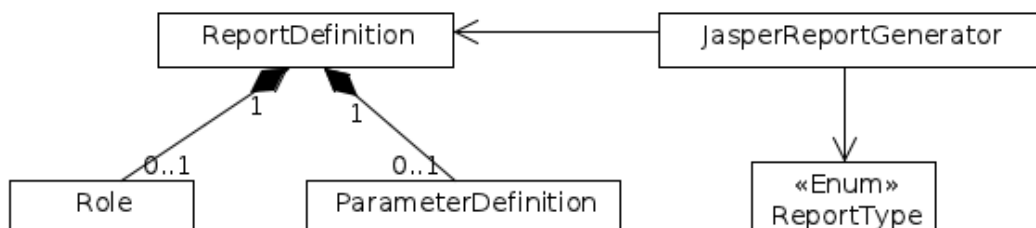
8.2 Using this module

To include this module in a Katari based application, simply put a line of the form:

```
<katari:import module="com.globant.katari.report"/>
```

8.3 Design

8.3.1 Domain model



8.3.2 Entities responsibilities

ReportDefinition The report definition represents the blueprint of a report, with the intended layout, query and constraints. The actual contents of the report are opaque to the programmer, who only knows the relevant parts of the report. The report definition also specifies which roles can execute it.

ParameterDefinition The parameter definition is the description of a variable constraint in the report. The constraint is created in definition time, and will be mapped for example to a constraint in the WHERE clause of a SQL statement. In runtime, the actual value of the parameter is specified, and the report is run with the appropriate constraint.

Role The roles that can execute a report, and are taken from the DB.

JasperReportGenerator The Jasper generator is used to execute a report definition which was designed using iReports. The execution will need an actual report definition, and the actual values for the parameters specified by such definition. The output of the generator will depend on the ReportType, and will be a PDF, an HTML, etc.

8.4 Use case scenarios

8.4.1 Report upload

1. The User creates a report using Jasper iReports WYSIWYG tool
2. The User selects 'new report', fills in the information, selects the file he has just created for upload and submits the form.
3. The System creates the Report Definition and uploads it to the database.

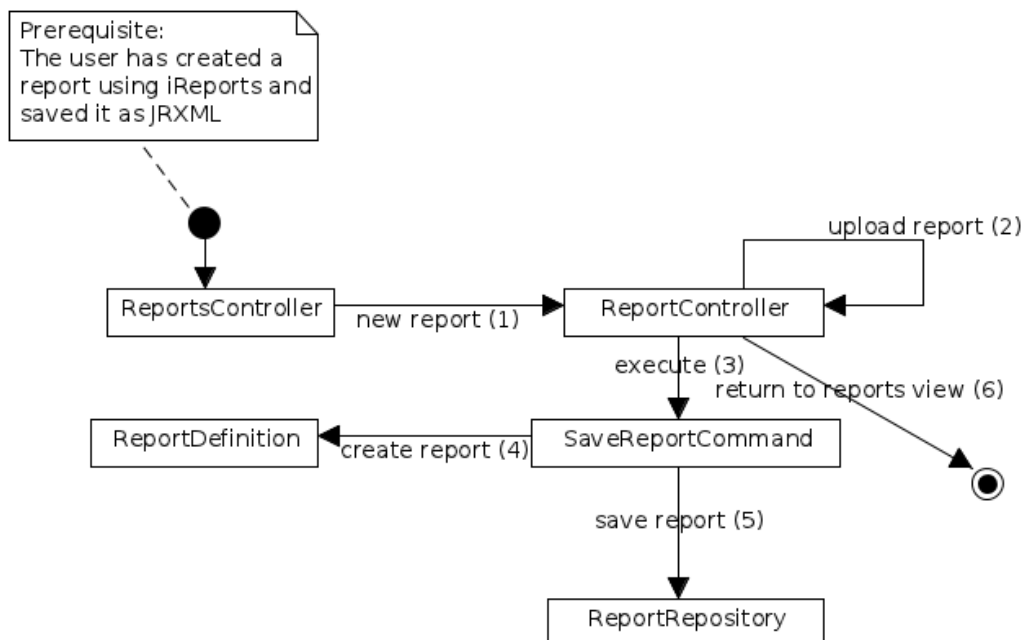


Figure 8.1: The report upload scenario

8.4.2 Report execution

1. The User selects a report previously uploaded.
2. The System shows a screen with the defined parameters for the selected report, and prompts the user to complete them.
3. The User completes the report parameters by which the final report will be filtered and submits the form.
4. The System retrieves the report definition from the database.
5. The System generates the report according to the definition and the actual parameters given by the User.
6. The System prompts the User to download the generated report.

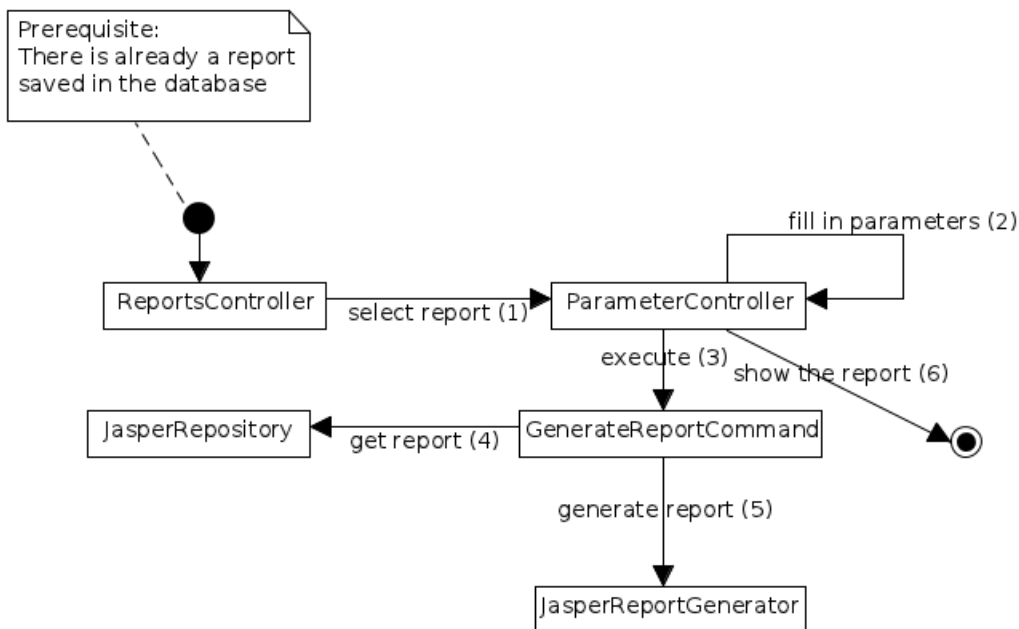


Figure 8.2: The report execute scenario

Chapter 9

Katari Social

9.1 Shindig

9.1.1 Activities

Katari provides `KatariActivityService`, an implementation of shindig `ActivityService`, and the related Activity entities: `KatariActivity` and `KatariMediaItem`. These entities are all mapped with hibernate.

As you can see, Katari uses the convention of prefixing 'Katari' to the name of the shindig interface that it implements.

`http://localhost:8098/katari-sample/module/shindig/social/rest/activities/1/@self/1?sortBy=title`

Without the sortBy:

Problem accessing `/katari-sample/module/shindig/social/rest/activities/1/@self/1`. Reason:
could not resolve property: topFriends of: `com.globant.katari.shindig.domain.KatariActivity`

9.1.2 Gadget rendering proxy

`http://localhost:8098/katari-sample/module/shindig/gadgets/ifr?url=http://www.labpixies.com/campaigns/todo/todo.xml`

`http://localhost:8098/katari-sample/module/shindig/gadgets/ifr?url=http://www.labpixies.com/campaigns/minesweeper/minesweeper.xml`

9.2 The gadget container

This module provides an open social compliant gadget container.

9.3 Using this module

To include this module in a Katari based application, put in your application context file (usually `src/main/webapp/WEB-INF/applicationContext.xml`), a line of the form:

```
<katari:import module="com.globant.katari.shindig"/>
```

Shindig needs to know where the application is running, so you need to configure the host name, port number, and context path. Override `shindig.hostAndPort` and `shindig.contextPath`:

```
<bean id='shindig.hostAndPort'  
      class='com.globant.katari.core.spring.StringHolder'
```

```

    p:value='localhost:8088' />

<bean id='shindig.contextPath'
    class='com.globant.katari.core.spring.StringHolder'
    p:value='/sample-web' />

```

This makes the gadget container available for your application. You can make it available in any page you want. But first, you must make some gadgets available to show in it. In this version you must manually insert the gadgets into the applications table:

```

insert into applications (id, title, url) values (2, 'Activities',
    'http://localhost:8098/katari-sample/module/gadget/ActivityTest.xml');

```

This makes the activities gadget available in the application directory, a page where you can select gadgets and include them in your page.

To include the gadget container in your page, include this in it:

```

<div id='custom-gadgets'>
  <div class='gadgetContainerTools'>
    <a href='${baseweb}/module/gadgetcontainer/directory.do \
      ?returnUrl=/module/institutional/dashboard.do&gadgetGroupName=main' >
      Add ...</a>
    </div>
    <!-- will hold open social gadgets. -->
  </div>
  <script type='text/javascript'>

    $(document).ready(function() {
      katari.social.renderGadgetGroup('custom-gadgets', 'main');
    });
  </script>

```

As you can see, you need to create an html container (custom-gadgets in this case) to hold the gadgets. Then you initialize a GadgetGroup object with data coming from gadgetcontainer/getGadgetGroup.do.

The full signature of the renderGadgetGroup function is:

```

katari.social.renderGadgetGroup = function(container, groupName, ownerId);

```

The ownerId is the owner of the gadget group you want to render. If not specified, it uses the gadget group owner by the viewer.

This example also adds a button to the gadget directory (gadgetcontainer/directory.do) to add new elements to the page.

9.4 Design

9.4.1 Domain model

The gadget container organizes gadgets in 'gadget groups', that contains a collection of 'gadget instances'. Each gadget instance has a reference to an application, basically the url for the gadget xml specification, and the views the application supports.

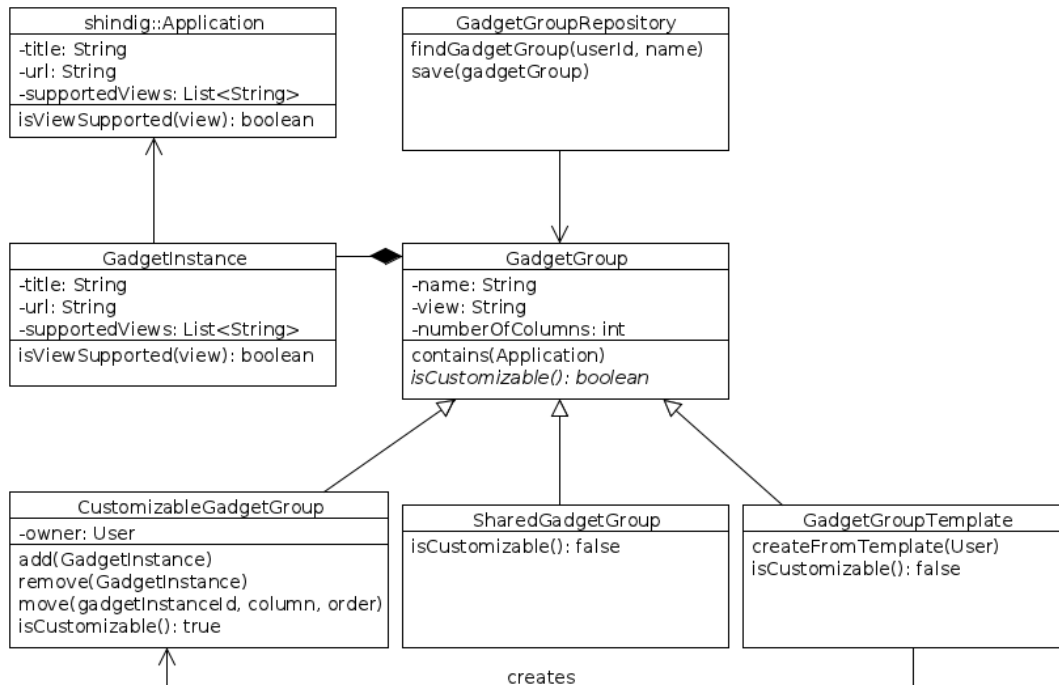
You can display gadget groups in a page.

There are three types of gadget groups: shared (the same gadget group is seen by all users), non shared (each user has his own group) or template.

Non shared gadget groups can be configured by the user, adding, moving or removing gadgets from the group.

Non-shared gadget groups can be created on demand: if a user tries to access a non existing non-shared gadget group, then katari creates one for him, from a gadget group template with the same name.

Each gadget group has a view. A gadget group can contain only applications with the default view, or applications that support the gadget group's view.



9.4.2 Entities responsibilities

Application This comes from the shindig module. It knows everything that is needed to render a gadget in the browser. An application has a list of supported views.

GadgetGroup The central concept in the gadget container, a gadget group is a collection of applications positioned on the page. It defines a view, and can only contain applications that support the gadget group's view, or the default view.

GadgetInstance An element that holds the position of a gadget in the gadget group.

GadgetGroupRepository The entry point to find gadget groups.

9.5 Configuration

If `debugMode.value` is set to `true` in the configuration property file, then the container does not cache the gadget xml spec and does not compress the social javascript files. This makes it easier to debug the javascript files and hot redeploy the gadgets.

You can also allow users to see other people's gadgets. You must implement the interface `ViewerOwnerRestriction` and declare it in your application context as:

```

<bean id="gadgetcontainer.viewOwnerRestriction"
      class="com.globant.gadgets.MyViewerOwnerRestriction" />
  
```


Chapter 10

Task scheduling with quartz

10.1 Introduction

This module provides task scheduling with the help of quartz.

10.2 Using this module

To include this module in a Katari based application, put in your application context file (usually src/main/webapp/WEB-INF/applicationContext.xml), a line of the form:

```
<katari:import module="com.globant.katari.quartz"/>
```

To add a task to schedule, add to your module.xml:

```
<bean id=' search.reindexCommand'  
  class=' com.globant.katari.search.integration.ReindexScheduledCommand' >  
  <constructor-arg index="0" ref="search.indexRepository"/>  
</bean>  
  
<bean class=' com.globant.katari.core.web.ListFactoryAppender' >  
  <constructor-arg value=' quartz.schedulerTasks' />  
  <constructor-arg>  
    <list>  
      <bean class="com.globant.katari.quartz.domain.CronTrigger">  
        <constructor-arg index='0' ref="quartz.daily"/>  
        <constructor-arg index='1' value="search.reindexCommand"/>  
      </bean>  
    </list>  
  </constructor-arg>  
</bean>
```

This is an example taken from the search module. It declares a bean `search.reindexCommand` and adds it to the `quartz.schedulerTasks`, with a cron trigger to be run daily.

The class `ReindexScheduledCommand` implements `ScheduledCommand`, from the `com.globant.katari.quartz.domain` package.

Chapter 11

Managing js modules

11.1 Js modules

This module is used to support packing javascript in jar files, making it possible to manage them with maven, deploying to repositories and tracking dependencies and versions.

11.2 Using this module

To include this module in a Katari based application, simply put a line of the form:

```
<katari:import module="com.globant.katari.jsmodule"/>
```

Then, add the dependencies to you pom file:

```
<dependency>
  <groupId>com.globant.jslib</groupId>
  <artifactId>jquery</artifactId>
  <version>1.4.2</version>
</dependency>
```

And finally, import the javascript file like this:

```
<script type='text/javascript'
  src='${baseweb}/module/jsmodule/com/globant/jslib/jquery/jquery.js'>
```

Future versions of this module will provide compression, bundling and dynamic dependency management.

11.3 Packaging javascript

To package a javascript library in a jar file, you just create a jar with the js files in it, in a standard java package format. Then you need to add a meta file META-INF/katari-resource-set describing where in the classpath to search for the files. For example, for jquery:

```
staticContentPath=/com/globant/jslib/jquery
mimeType.js=text/javascript
debugPrefix=../jquery/src/main/resources
```