Vizrt Community Expansion
Developer Guide
3.8.0.130433

Escenic Content Engine™

Technical Support

For technical support and the latest news of upgrades, documentation, and related products, visit the Vizrt web site at www.vizrt.com.

Last Updated

13.09.2012

Table of Contents

1   Introduction

Welcome to the Vizrt Community Expansion's guide for template developers. In this guide, we will first give you an overview of the different modules in Community Engine. We will then go on to describe some of the concepts that may be new to you when starting off with it. The guide will then continue with walking through a list of common use cases you want to accomplish with Community Engine, such as creating a blog, uploading images and displaying a tag cloud.

The guide then rounds off with a number of chapters describing some of the modules in depth, such as using the Third Party API and the Dashboard content moderation module.

2 Concepts

As mentioned in the introduction, this guide assumes that you have already read the Escenic Content Engine Developer Guide and understand the concepts presented in there such as; Struts actions & forms, JSP, the various publication resources, section parameters and so on.

This chapter will describe a number of concepts that are unique to the Community Engine and that the developer creating Community templates should understand.

2.1   Overview of the Community Engine Modules

Community Engine contains several modules providing a whole range of functions for building community web sites. These modules different access points such as Java APIs, JSP `tag` libraries, Struts `action` classes and Javascript methods for accessing the DWR AJAX framework. The JSP tag libraries are fully documented in **Vizrt Community Expansion Taglib Reference**. Turn to the API JavaDoc included with Community Engine distribution to use the plugin API and the Struts `Action` classes.

You can find examples of most of the features provided by all modules in `community-demo.war` and we recommend that after reading through this guide and trying out some (or all) of the cookbook examples, that you turn to the `community-demo.war` webapp for examples on how to use the remaining set of functions available in Community Engine.

- The **Auth** module provides security related features. You can apply role based security constraints on the HTTP requests and your JSP pages. See further details in **section 2.3**.
- The **Community** module provides Community Engine specific user information (as opposed to normal user information and related functions to these already available in Content Engine). Using the Struts `action` classes and tag library provided in this module, it is easy to implement friendship and community group related features such as making friends, joining the groups, retrieving friends and group members.
- The **User Content** module deals with creating, validating, editing and removing user generated content such as blogs, image, videos. Using the Struts `action` classes provided in this module, it is easy to implement such features. Apart from this, user creation and user login features are also included here.
- The **Messaging** module provides messaging for the Community Engine users. Sending messages and retrieval of the message notification are all part of this.
- The **Qualification** module qualifies user generated content. Rating and flagging functionality can be implemented using this module. It also qualifies a Community Engine user based on his content's qualification. The qualification of user and his content can be retrieved using the qualification tag library.

- The **Tag** module lets you tag content, create tag clouds, retrieve tagged content, find similar tags.

- The **Statistics** module deals with the action history of the Community Engine user. Based on the action history, this module can provide statistics on user and user generated content such as "most popular blogs".

- The **3rd Party Content** module: Using this component, it is possible add content from 3rd party providers, such as Flick, Last.fm, Picasaweb and Twitter, to a user's profile page.

- The **SSO** module enables **Single Sign On** login on VCE sites. With this feature, you can log into an VCE site using your 3rd party ID provider credentials instead of the VCE site's.

- The **Dashboard** is a web application included with Community Engine which provides functionality to moderate user generated content. Using the Dashboard, you can accept or reject blogs, comments, images and videos created by users (or editorial staff) on your web site.

- **Captcha** support is provided for several action classes in Community Engine to ensure that only human can initiate requests to those actions. It is possible to configure VCE to use different captcha providers.

2.2　Direct Web Remoting

Community Engine uses a library called **DWR** to make Javascript talk to the server side Java API, much the same way you would do with a AJAX library such as jQuery. The difference with DWR, is that it maps everything to beautiful Java objects on the server side.

In order to use Community Engine DWR support, make sure that your publication's webapp context (i.e. `WEB-INF`) contains `dwr.xml` and `community-plugin-beans.xml`. For further details, see section **chapter 5**. Furthermore, you must have the following `Spring` configuration in your `web.xml`:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/community-plugin-beans.xml</param-value>
</context-param>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

Once you have configured Community Engine DWR support correctly, you may browse the classes that are available from JavaScript using the DWR servlet on your web site here: http://yoursite.com/your-publication/dwr/

To provide security when using DWR, Community Engine provides a publication resource `/escenic/plugin/community/security` file that defines the prerequisites for using the DWR methods. The resource must be present for each publication running Community Engine. Please see **section 2.3.4** for a further description of this resource.

2.3 Security

The security module in Community Engine is in place to secure actions from unauthorized users. This module checks whether the logged in user has enough privileges to perform the desired HTTP request. These privileges are determined from the user's ownership on the content items and what permissions he/she has on the sections.

2.3.1 Roles & Permissions

The Community Engine security module uses a model of roles and permissions. This is designed in a way that the requests can be secured without writing any security related code. User privileges can be defined by simply manipulating the roles and permissions in the Web Studio and using these in your JSP templates.

A `Permission` is a privilege on a section that a user should have for performing an action on the section or on the contents of the section. The permission required to perform a request or an action can be configured in `security` publication resource. See further details on configuring `security` publication resource in **section 2.3**.

Users can have one or more `role`s on a section. Each role grants one or more permissions to the user.

The mapping of role and permission is called can be done in the authorisation matrix in Web Studio. Here, you define the permissions for a role.

Community Engine provides a tag library to access the users' roles and permissions and thereby implementing your desired security model on your web site. This information can be used to show or hide a part of the page according to the role/permission of the user. For instance, to see a group's message board, the user would need to have the `GROUP MEMBER` role in the group section. For further details on using the security related tags, see **Vizrt Community Expansion Taglib Reference: Chapter 3: auth**.

2.3.2 Defining and Assigning Roles & Permissions

When Community Engine is installed, a component called `Community Authorisation` can be found on the Escenic Web Studio home page. There are several pages for this component for defining and assigning roles & permissions:

- `Permissions`: All the permissions available for a publication are listed here. To add more permissions, simply click on the **New permission** link.
- **Roles**: All the roles available for a publication are listed here. To add more roles, click on the **New role** link.
- **Authorisation matrix** : This is where permissions can be assigned to different roles. Use the check boxes to define permissions for each role.
- **User management**: Roles can be assigned to users here. Enter the user name or user ID on the page and click on **Get User Roles**. A list of roles assigned to the user will show up. Roles can also be added from this page.

Simply select a role from the combo box, put a section ID and click on add. It is also possible to assign roles to user on no section, which means this role applies to user for all section.

These are some scenarios where roles are assigned to the user automatically:

- When a user signs up, he gets the role 'MEMBER'
- When a user signs up, he gets the role 'SECTION OWNER' on his profile section
- When a user creates a group, he gets the role 'SECTION OWNER' on the group's section

2.3.3 Using the Security Filter and security Publication Resource

To use the security module, you have to add the filter `com.ndc.auth.filter.SecurityFilter` to your web application's `web.xml` file:

```
<filter>
  <filter-name>securityFilter</filter-name>
  <filter-class>com.ndc.auth.filter.SecurityFilter</filter-class>
  <init-param>
    <param-name>login</param-name>
    <param-value>/login.jsp</param-value>
  </init-param>
  <init-param>
    <param-name>error</param-name>
    <param-value>/error.jsp</param-value>
  </init-param>
  <init-param>
    <param-name>unauthorized</param-name>
    <param-value>/unauthorized.jsp</param-value>
  </init-param>
</filter>
```

The security filter checks the logged in user's privileges or permissions on content items and sections based on the parameters found in the HTTP request.

The security rules can be configured in `/escenic/plugin/community/security` resource. You can find a working example of the `security` resource here: `$VCE_HOME/misc/contrib/publication/META-INF/escenic/publication-resources/escenic/plugin/community/security`. It is a good starting point for building your own configuration. Here is an example of a `security` element inside `security` resource file:

```
<security>
  <action pattern="message/text/add"/>
  <action pattern="group/membership/request" user="true"/>
  <action pattern="blog/save" author="true"/>
</security>
```

The child elements of the `security` element are `action` and `ajax`. Both of these accept a `pattern` element. Each `action` element describes one rule for security checking. The value of the `pattern` attribute is an **Ant** like pattern string which will match the last part of the request URI (for Struts actions, this is the action name). Here are some examples of how to use asterisk (*) in pattern strings for wildcard matching.

Pattern	Matched Action Name
`delete`	delete
`delete*`	delete
	deleteBlog
`*Blog`	createBlog
	deleteBlog
`delete*Image`	deleteImage
	deleteAllImage
	deleteAlbumImage

The `action` elements can also have `user` and `author` attributes. It can also contain `permission` elements.

See details on the syntax of the `security` resource in **chapter 10**. You can validate your `security` resource with the RELAX NG schema file `$VCE_HOME/documentation/schemas/community-security.rng`. Here are some examples of security rule definitions in the `security` resource:

Description	Required Request Parameter	Action Tag Syntax	Fails If
To check whether a logged in user is performing the action	(none)	`<action pattern="message/text/add"/>`	No user object could be found in the session, no user is logged in
To check whether the user himself is performing the action that deals with the user's personal state. This type of configuration can be used for requests where the action is performed on the user with ID specified by the `userId` request parameter. Only this user should be able to perform the action.	`userId`	`<action pattern="group/membership/request" user="true"/>`	The value of the request parameter `userId` is the same as the ID of the logged in user
Check whether the logged in user is the author of the article which is dealt with in the action.	`articleId`	`<action pattern="blog/save" author="true"/>`	The logged in user is not the author of the article found by the value of the request parameter `articleId`
Check whether the logged in user has a certain permission on a section	`sectionId homeSectionId`	`<action pattern="/home/add" author="true"> <permission>addContent</permission>`	The logged in user does not have the permission

Description	Required Request Parameter	Action Tag Syntax	Fails If
		`permission> </action>`	(required for this action) on the section specified by the request parameter `sectionId` or `homeSectionId`
Check whether the logged in user has permission on the article	`articleId`	`<action pattern="ReportArticle"> <permission>report</permission> </action>`	The logged in user does not have the permission (required for this action) on the home section of the article specified by the request parameter `articleId`.

2.3.4 Securing DWR

Community Engine provides security functionality so that it is possible to use the `security` publication resource to secure the DWR calls.

To secure an DWR AJAX call an `ajax` element needs to be used instead of `action` element. This `ajax` element must have a `pattern` attribute which corresponds to what you find on the operation overview on the `/dwr/` servlet page.

Similar to the `action` element, the `ajax` element accepts the parameters `user` and `author` and may contain `permission` elements. Here is an example on how to use the `ajax` element.

```
<ajax pattern="TagPluginAjax.addTag" user="true">
  <permission>rate</permission>
</ajax>
```

One difference between checking HTTP requests and DWR calls is that for DWR calls, the security module uses AJAX's method arguments instead of the request parameters.

2.3.5 Securing DWR on Weblogic

The HTTPOnly flag is used to prevent client side scripts to read the cookies (provided the browser supports the HTTP only cookie extension). By default, cookies are set to HTTP Only in version 11g of Weblogic. Unfortunately, the

current stable release of DWR does not support HTTPOnly JSESSIONID cookies and we therefore have to turn this feature off.

This implies that XSS checking for DWR must be disabled. This can be done by adding the following init- parameter to the dwr-invoker servlet.

```
<init-param>
    <param-name>crossDomainSessionSecurity</param-name>
    <param-value>false</param-value>
</init-param>
```

Recently, DWR has received support for HttpOnly, see this issue: Add support for HttpOnly cookies. This feature will be a part of their upcoming 3.0 release and should then make the described configuration step redundant.

2.4   ESI

Getting the dynamic bit of your community website to scale is "easily" done with Edge Side Includes, enabling you to have different caching policy for different fragments. Varnish, Akamai, Oracle Webcache and Squid 3 all support this and we recommend that all community web sites utilise this technology.

Using ESI may imply that you will have to strongly structure your template set, creating JSPs/JSPFs based on their cacheability (and not only functionality). Thus, it pays off to consider using ESI (or not) early on in the project. Please see the `Vizrt Community Expansion Performance Guide` for more information on ESI and how to apply caching rules from your JSP templates.

2.5   Configuring the VCE Publication

In this section, we will go through what needs to be done for each ECE publication that wants to use the Community Engine.

2.5.1   The Qualification module

Qualification plugin provides flagging functionality which is used by visitors to the site to indicate that the article has inappropriate contents. When a given article has received a flagging threshold, it is moved to a section called `flagged`. This section can be configured using `qualification.flaggedSectionUniqueName` in `feature` publication resource.

If you are using the flagging functionality , it is necessary to modify the articletype resource of your publications. For each articletype that is allowed to be flagged add this parameter to its articleType:

```
<parameter name="com.ndc.qualification.allowFlagging" value="true"/>
```

You may also want to alter the publication feature property that governs how many flags an article needs before it is moved to the `flagged` section:

```
qualification.flaggingThreshold=3
```

Editorial staff may view and revoke all flagged articles using the Dashboard interface, e.g. http://myserver:8080/dashboard/content/?content-type=allFlaggedContent

2.5.2   The Statistics module

Specific notes on installing and using the statistics module.

Per default, all posting actions are recorded. If this is not desired, it is possible to turn this off globally or per publication. To set this for all publications, edit the global configuration layer, e.g.:

```
$ vi /etc/escenic/engine/common/com/escenic/statistics/Initial.properties
```

If the file does not exist, create the directories and file, then add:

```
$class=com.ndc.statistics.api.StatisticsPluginProperties
recordCommentsEnabled=false
```

To set this just for one publication, edit the publication's **feature** properties and set:

```
statistics.recordCommentsEnabled=false
```

The VCE Statistics Module records most of user actions which can be displayed indicating what a user has been doing. As of VCE 2.6-15, it uses an event/listener approach to record comments created with the Forum plugin.

The **IOEscenicForumEventFilter** (records an action when a user create/modifies a comment) is enabled by default. If the feature of recording comment action is not desired then it can be disabled in all ECE instances.

The Nursery path for this component is: **/com/escenic/statistics/filter/IOEscenicForumEventFilter**. To disable this component, create a **properties** file at the Nursery path mentioned above and add the following line:

```
enabled=false
```

The database that stores statistics information can be really big. Regular cleanup of old statistics data that are not interesting can help the database to perform better. For instance, one year old login history of a community user may not be used in the features of a community site.

VCE provides a service that deletes old statistics data for configured action types. The component responsible for doing the cleanup is **/com/escenic/community/statistics/ActionHistoryCleaner**. This can configured to specify how old action history of a specific action type should be deleted. The example configuration can be found in **$VCE_HOME/misc/siteconfig/com/escenic/community/statistics/ActionHistoryCleaner**.

2.5.3   The User Content module

2.5.3.1   Configuration for User Content Section

The Community Engine needs the following parameter to be set on the root section. It specifies where the user profile articles are stored.

```
usercontent.userProfile.uniqueName=profile
```

User content articles (e.g. blog and groupProfile) and images are added to the user's profile section. The content items (articles and images) are also added to another section which must be configured by setting a section parameter for each content type. These parameters can either be set on the root section or on the user's home section. The former approach will make all users inherit the same settings whereas the latter gives user specific sections.

Below is an example of section parameters for a publication having three kinds of user content types ("imageFile", "blog" and "groupProfile"):

```
usercontent.imageFile.uniqueName=images
  usercontent.blog.uniqueName=blogs
  usercontent.groupProfile.uniqueName=profile
```

2.5.3.2   Configuring Content Type Parameter

This module requires a number of content type parameters for registering a community user. Here is the usage of the parameters that should be added to the content type which is used as the user profile.

```
<content-type name="userInfo">
    ...
    <parameter name="com.escenic.community.articleType" value="userProfile"/>
    <parameter name="com.escenic.articleType" value="profile"/>
    <parameter name="neo.xredsys.service.article.attribute" value="true"/>
    <parameter name="com.ndc.usercontent.sectionNameField" value="username"/>
    <parameter name="com.ndc.usercontent.usernameField" value="username"/>
    <parameter name="com.ndc.usercontent.firstNameField" value="firstName"/>
    <parameter name="com.ndc.usercontent.surNameField" value="surName"/>
    <parameter name="notification.fellow.action.email" value="email-notification"/>
    <parameter name="notification.fellow.action.message" value="message-notification"/>
    ...
  </content-type>
```

Parameter	Description
`com.escenic.community.articleType`	The value `userProfile` tells VCE to use this content type to create user profile content.
`com.escenic.articleType`	The value `profile` tells ECE that this content type is a user profile.
`com.ndc.usercontent.sectionNameField`	The value is the name of a content type field which should be used as the name of the user section.
`com.ndc.usercontent.usernameField`	The value is the name of a content type field which should be used as the username of the user.
`com.ndc.usercontent.firstNameField`	(Optional) The value is the name of a content type field which should be used as the first name the user.

Parameter	Description
	Default value is the the value of `com.ndc.usercontent.usernameField`.
`com.ndc.usercontent.surNameField`	(Optional) The value is the name of a content type field which should be used as the surname the user. Default value is the the value of `com.ndc.usercontent.usernameField`.
`com.ndc.usercontent.emailAddressField`	(Optional) The value is the name of a content type field which should be used for as the email address of the user..
`notification.fellow.action.email`	(Optional)The value is the name of the `content type field` which should be used as the means of notification when other users has performed the same action on an content item that he/she has performed on before. If this field is enabled, the user will receive e-mails.
`notification.fellow.action.message`	(Optional)The value is the name of the `content type field` which should be used as the means of notification when other users has performed the same action on an content item that he/she has performed on before. If this field is enabled, the user will receive internal messages.

2.5.3.3   Configuring User Generated Text Filtering

The action classes used for generating contents for user filters the value of the content fields. In case of a normal text field, all the special characters to define HTML markup such as &, <, > are converted into entities and saved to database. Rich text field is dealt differently. They are cleaned by using the JTidy library which converts the rich text field value into well formed XHTML. But it is also possible not to use JTidy for cleaning. In that case, only &, <, > are converted into entities. To configure VCE to use this basic cleaning, copy `$VCE_HOME/misc/siteconfig/com/escenic/community/RichTextFieldCleaner.properties` to your local config and modify following:

```
tidyEnabled=false
```

Rich text field value is also looked for unwanted HTML tags and attributes to secure from XSS(Cross-site scripting) vulnerabilities. This way a user cannot put some JavaScript code or a <form> tag. By default, following tags and attributes are configured for the filtering:

```
filterElements=script,form,iframe
filterAttributes=onload,onunload,onchange,onsubmit,onreset,onselect,onblur,onfocus,onkeydown,onkeypress,
\
    onkeyup,onclick,ondblclick,onmousedown,onmousemove,onmouseover,onmouseout,onmouseup
```

You can change the values of `filterElements` and `filterAttributes` in `com/escenic/community/RichTextFieldCleaner.properties` in you local config. You can also disable filtering of tags and attributes by setting the value of `filterEnabled` property to `false` .

2.5.4   Captcha Support

Escenic Content Engine comes with built-in feature to add Captcha to different actions. Please see the **Advanced Developer Guide** for instructions on how to setup Captcha.

3 Common Functions

The chapter explains some functions that are common to most community sites. To help developers create similar functions, VCE provides a generic set of Struts actions that can be customised to fit the developers' needs.

3.1   Creating Articles

The section explains how a developer can implement a feature that allows a user to create an arbitrary article.

```
<form-bean name="articleForm" type="com.ndc.usercontent.struts.actions.forms.ArticleForm" />

<action path="/content/add"
        name="articleForm"
        scope="request"

 parameter="articleType=blog;allowSetPublishDate=true;allowSetActivateDate=true;allowSetExpireDate=true"
        type="com.ndc.usercontent.struts.actions.save.SaveArticle">
    <forward name="success" path="/create-content-success.jsp" />
    <forward name="error" path="/create-content.jsp" />
</action>
```

Please note the **parameter** attribute of the action declaration. The **articleType** defines what content type can be created using the action. An example JSP template that allows the user to fill in some information about the article and create it can be as follows:

```
<community:user id="user"/>
<html:form action="/content/add">
  <html:hidden property="articleType" value="blog"/>
  <html:hidden property="homeSectionId" value="${user.section.id}"/>
  <html:hidden property="state" value="published"/>
  <html:hidden property="image" value=""/>
  <html:hidden property="articleId"/>

  <html:text property="field(TITLE)" />
  <html:text property="field(BODY)" />

  <html:hidden property="errorUrl" value="error-url" />
  <html:submit value="Create"/>
</html:form>
```

Form Property	Description
articleType	The content type of the article to be created. Note that, if the form property is not present, the content type of the article will be of the configured value of the **articleType** parameter, configured by the **parameter** attribute of the action declaration.
homeSectionId	The home section of the article to be created. On an VCE site, it is usually set to the ID of the home section of the requesting user
state	The state of the article to be created. Set it to **published** if the blogs created by users are to be published on the site by default.
field(field-name)	These are the article fields that you will probably want the user to fill in. For example, a blog

Form Property	Description
	may have a title and a body field. The fields are configured in the content type of the publication. So, if a blog needs three fields i.e. "Title", "Sub title" and "Body", the content type needs to declare that and the JSP template can have an entry additional to the example given above. `<html:text property="field(SUBTITLE)" />`
`publicationId`	The ID of the publication in which the article is to be created. If not specified, the current publication is used.
`successUrl`	The URL where the user should be directed to if the article is created successfully. If the form property is not present, the user is directed to the URL of the newly created article.
`publishDate`	Corresponds to the publish date of the article to be created. The format of the date must be **yyyy-MM-dd'T'HH:mm:ss**
`activateDate`	Corresponds to the activation date of the article to be created. The format of the date must be **yyyy-MM-dd'T'HH:mm:ss**
`expireDate`	Corresponds to the expiration date of the article to be created. The format of the date must be **yyyy-MM-dd'T'HH:mm:ss**
`errorUrl`	The URL where the user should be directed to if any error occurs while trying to creating the blog.
`allowSetPublishDate`	The property is configured through the parameter attribute of the declared action. It enables the **publishDate** attribute in the article form.
`allowSetActivateDate`	The property is configured through the parameter attribute of the declared action. It enables the **activateDate** attribute in the article form.
`allowSetExpireDate`	The property is configured through the parameter attribute of the declared action. It enables the **expireDate** attribute in the article form.

3.2   Validating Fields

There are a few built in validators available for template developers so that they can easily configure a content type field to get validated whenever it is saved. These validators can be configured for a content type field by defining certain predefined parameters. Below is a table describing the available validation schemes.

Parameter name	Value	Struts message key	Description
`com.ndc.usercontent.constraint`	`notEmpty`	`validate.error.empty`	Validates the field

Parameter name	Value	Struts message key	Description
			ensuring that it is not empty
com.ndc.usercontent.constraint	isEmail	validate.error.isEmail	Validates the field ensuring that the input value is an email address
com.ndc.usercontent.constraint	minLength	validate.error.minLength	Validates the field ensuring that the input value has the minimum number of characters. Example: `minLength(10)`
com.ndc.usercontent.constraint	maxLength	validate.error.maxLength	Validates the field ensuring that the input value has at most the configured number of characters. Example: `maxLength(10)`
com.ndc.usercontent.constraint	isNumber	validate.error.isNumber	Validates that the field value is a number
com.ndc.usercontent.constraint	isPhoneNumber	validate.error.isNumber	Validates the field ensuring that the input value is a valid phone number
com.ndc.usercontent.constraint	userNameFormat	validate.error.userNameFormat	Ensures that the input value consists of only letters and digits

Parameter name	Value	Struts message key	Description
com.ndc.usercontent.constraint.dateInPast	dateInPast	validate.error.dateInPast	Ensure that the input date value is of a past date. The format must be: `2006-11-04T10:46:29`
com.escenic.community.constraint.regex	A valid java regular expression that can be compiled with: `java.util.regexf.Pattern`	validate.error.regex	Validates the input value with the configured regular expression

4 Cookbook

The chapter explains how a developer can create various Vizrt Community Expansion (VCE) components. Note that, most of the functions related to user activity is implemented using the Struts 1.x framework. Thus, some knowledge of the Struts framework is required to work with VCE.

Furthermore, before using this cookbook, make sure that you have configured the publication to use the modules. You can find details on publication configuration in **section 2.5**

4.1   User Registration

Perhaps the first component that comes to mind on a community site is the registration of new users. VCE is distributed with built-in features to allow template developers to write the user registration feature in minimal effort.

The struts configuration for the user registration action can be as follows:

```
<form-bean name="userProfileForm" type="com.ndc.usercontent.struts.actions.forms.UserProfileForm" />

<action path="/user/profile/add"
        name="userProfileForm"
        scope="request"
        parameter="userProfile"
        type="com.ndc.usercontent.struts.actions.save.SaveUserProfile">
  <forward name="success" path="/?view=signupSuccess" redirect="true"/>
  <forward name="error" path="/index.jsp?view=signup" />
</action>
```

An example template that will allow a new user on the site to register his information can be as follows:

```
<html:form action="/user/profile/add">
  <html:messages id="messages" message="true">
    <bean:message  name="messages"/><br/>
  </html:messages>

 <html:hidden property="articleType" value="${articleType.name}"/>
 <html:hidden property="homeSectionId" value="${homeSectionId}"/>
 <html:hidden property="state" value="${state}"/>
 <html:hidden property="image" value=""/>
 <html:hidden property="articleId"/>

  <html:hidden property="errorUrl" value="/signup.jsp" />

  Username: <html:text property="field(username)" /><br/>
  Email: <html:text property="field(emailaddress)" /><br/>
  First name: <html:text property="field(firstname)" /><br/>
  Family name: <html:text property="field(surname)" /><br/>
  Telephone: <html:text property="field(telephone)" /><br/>
  <html:submit value="Register"/>
</html:form>
```

On some community sites, you may be required to display the password of the user after registration. Set `redirect="false"` in action forwoard in struts configuration file to show the password in the forwarded page. This can be easily achieved through the following JSP template:

```
<form-bean name="userProfileForm" type="com.ndc.usercontent.struts.actions.forms.UserProfileForm" />
```

```
<action path="/user/profile/add"
        name="userProfileForm"
        scope="request"
        parameter="userProfile"
        type="com.ndc.usercontent.struts.actions.save.SaveUserProfile">
  <forward name="success" path="/?view=signupSuccess" redirect="false"/>
  <forward name="error" path="/index.jsp?view=signup" />
</action>


<logic:present name="userProfileForm">
 <bean:define id="password" name="userProfileForm" property="password" type="String"/>
  Password: ${password}<br/>
</logic:present>
```

The **com.ndc.usercontent.struts.actions.forms.UserProfileForm** class is a subclass of the
com.ndc.usercontent.struts.actions.forms.ArticleForm. For details on form properties of **ArticleForm**, please have a look at: **section 3.1**. Apart from the article-form properties, the user-form adds the following properties:

Form property	Description
articleId	The profile article ID of the community-user. If set, the profile identified by the ID is updated with the new information.
password	The property is only used to display the password. See the example JSP above that renders the password.

4.2   User Login

One of the most important functions of a community site is that it should identify the requesting user. VCE comes with built-in functionality that allows a developer to implement user login in a very short time.

The Struts configuration for the login action may look as follows:

```
<form-bean name="signinForm" type="com.ndc.usercontent.struts.actions.forms.SigninForm" />

<action path="/auth/login"
  name="signinForm"
  scope="request"
  validate="true"
  input="/index.jsp"
  type="com.escenic.profile.presentation.struts.LoginAction">
  <forward name="success" path="/auth/community/login.do"/>
  <forward name="error" path="/"/>
</action>

<action
  path="/auth/community/login"
  scope="request"
  name="signinForm"
  validate="true"
  input="/index.jsp"
  parameter="userProfile"
  type="com.ndc.usercontent.struts.actions.login.Login">
  <forward name="error" path="/" />
</action>
```

Since VCE works as a plug-in on top of ECE, we need to delegate control from the first login action to the second one. The first action, **/Login**, makes the

user log into ECE and the second action **/auth/community/login** makes the user log into VCE.

To add SSO support to your login component, please have a look at: **chapter 7**

The JSP template that renders the form to the user may look as follows:

```
<html:form action="/auth/login">
  <div>
    <label for="userName">User name:</label>
    <html:text tabindex="1" property="userName" size="10"/>
    <label for="userPassword">Password:</label>
    <html:password tabindex="2" property="password" size="10"/>
    <br/>
    <html:checkbox property="savePassword">Remember me</html:checkbox>
    <br/>
    <html:submit value="Login" tabindex="3" />
    <html:hidden property="targetUrl" value="/profile/"/>
    <html:hidden property="errorUrl" value="/profile/"/>
  <div>
</html:form>
```

Form Property	Description
userName	The user name of the requesting user
password	The password that identifies the requesting user
savePassword	Indicates if the login action should remember the user or not. If selected, the user will remain authenticated for a certain period of time.
targetUrl	The URL (absolute or relative to the publication) where the requesting user should be directed to after a successful login. If not specified, the action tries to find the configured forward named **target**. Note that, in our example configuration, we have not configured any such forward. In which case, the user is redirected to his or her profile section.
errorUrl	The URL (absolute or relative to the publication) where the requesting user should be directed to if the login attempt fails. On failure, the login action binds the relevant error messages to the request scope so that the template developer can display information relevant to the failed login attempt.
	userName: contains error information related to the user name. Usually, the error property is present with the action error key "no_userName" to indicate that the user has not entered any user name. The error can be displayed using: **<html:errors property="userName"/>**,
	password: contains error information related to the password field. Usually, the error property is present with action error key "no_password" to indicate that the user has not entered any password. The error can be displayed using: **<html:errors property="password"/>**

Form Property	Description
	login_failed: indicates that the login attempt failed due to unknown user name or invalid password. The error can be displayed using: `<html:errors property="login_failed"/>`
	login_failed_account_expired: indicates that the login attempt failed because the user's account expired. The error can be displayed using: `<html:errors property="login_failed_account_expired"/>`
	login_failed_other: indicates that the login attempt failed because the user's credentials expired. The error can be displayed using: `<html:errors property="login_failed_other"/>`

4.3 Remember me

When a user chooses to be remembered, one cookie is set on the user's computer. This cookie contains three important pieces of information: **userId**, **publicationId** and **auth token**. The validity of the auth token is two weeks. If user login to the site within two weeks then a new **auth token** will set for next two weeks for that user. The expiry time of the cookie is one year, after this time the cookie will be deleted. However, it also gets deleted when the user decides to log out.

The web configuration to allow **remember me** functionality as follows:

```
<filter>
  <filter-name>cookieFilter</filter-name>
  <filter-class>
 com.ndc.usercontent.filter.UserCookieFilter
  </filter-class>
  <init-param>
    <param-name>profileArticleType</param-name>
    <param-value>userProfile</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>cookieFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

4.4 Creating Blogs

The feature can be implemented by customising the generic functionality explained in **section 3.1**

The Struts configuration to create a blog may look as follows:

```
<form-bean name="blogForm" type="com.ndc.usercontent.struts.actions.forms.ArticleForm" />

<action path="/blog/add"
        name="blogForm"
        scope="request"
        parameter="articleType=blog"
```

```
        type="com.ndc.usercontent.struts.actions.save.SaveArticle">
  <forward name="success" path="/create-blog-success.jsp" />
  <forward name="error" path="/create-blog.jsp" />
</action>
```

An example JSP that allows the user to fill up some form fields to create a blog can be as follows:

```
<community:user id="user"/>
<html:form styleId="addBlogForm" action="/blog/add">
  <html:hidden property="articleType" value="blog"/>
  <html:hidden property="homeSectionId" value="${user.section.id}"/>
  <html:hidden property="state" value="published"/>

  <html:text property="field(TITLE)" />
  <html:text property="field(BODY)" />

  <html:hidden property="errorUrl" value="error-url" />
  <html:submit value="Create"/>
</html:form>
```

The form and action classes used here are the same classes used to create general articles. For an explanation of the form fields, please have a look at: **section 3.1**

4.5 Creating Comments

VCE uses the Forum plug-in for comments. It even supports hierarchies of comments in cases where a user should be able to comment on another comment made to the article.

The only difference from a normal ECE site that uses the Forum plug-in, is that VCE provides its own Struts action **com.escenic.forum.struts.presentation.CommunityInsertPostingAction** which extends forum plug-in's **com.escenic.forum.struts.presentation.AuthenticatedInsertPostingAction**. The **CommunityInsertPostingAction** creates the comments under the requesting users' home section.

Fore more on the Forum plug-in, please have a look at the **Forum Plug-in Guide** shipped with the Forum plug-in distribution.

4.6 Uploading Photos & Videos

VCE comes with built-in functionality to allow developers to write features such as uploading images and videos.

Uploading media files is done by using VCE's Struts actions. An example Struts configuration for uploading images follows:

```
<form-bean name="uploadForm" type="com.ndc.usercontent.struts.actions.forms.UploadForm" />

<action path="/photo/upload"
        name="uploadForm"
        scope="request"
        parameter="articleType=photo"
        type="com.ndc.usercontent.struts.actions.upload.MediaContentUpload">
        <forward name="error" path="/" />
</action>
```

An example JSP template that uploads the image can be as follows:

```
<community:user id="user"/>
<html:form enctype="multipart/form-data" action="/photo/upload" method="post">
  <html:hidden property="homeSectionId" value="${user.section.id}"/>

  <html:hidden property="errorUrl" value="/upload-images.jsp"/>
  <html:hidden property="state" value="published"/>
<%--
  <html:hidden property="articleType" value="photo"/>
--%>
  <label>Select photo</label>
  <html:file property="file" />

  <label>Caption</label>
  <html:text property="field(caption)" />
  <label>Description</label>
  <html:textarea property="field(description)"> </html:textarea>
  <html:submit value="Upload"/>
</html:form>
```

Note the form encoding type **enctype="multipart/form-data"**. This specifies that the form will upload the data in a multi part format. This must be specified in the example JSP since this is not the default behaviour.

Form property	Description
articleType	In our example, the **articleType** property is commented out. This means that the value of the property is retrieved from the value of the **parameter** attribute configured in the declared action. If the parameter does not contain this declaration, the action will try resolve the content type by using the value specified in the **articleType** form property. If none are present, the system will throw an exception.
state	The state of the image to be created. If the property is not present, the state will be **draft**. We strongly recommend developers to explicitly specify the state since the default may change in future releases of VCE.
file	The property is the actual image data and the reason why the form encoding needs to be set to **enctype="multipart/form-data"**.
field(field-name)	These are the image content type fields that you will probably want the user to fill in. Note that the fields are configured through the content type publication resource. In the demo web application, the configured fields are caption and description.
homeSectionId	The home section of the image to be created. On an VCE site, it is usually set to the ID of the home section of the requesting user
publicationId	If this property is present, the image is created in the publication identified by the ID. If the property is not present, the image is created in the publication of the requesting user.
successUrl	Our example does not contain this form property. If it is present, the user is directed to the specified

Form property	Description
	URL after the image is uploaded successfully. If the property is not present in the form (which is the case of our example), then the user is directed to the URL of the newly uploaded image.
errorUrl	If this property is present, the user is directed to the URL using the value of the property if an error occurs while trying to upload the image. If the form property is not present, then the user is directed to the URL configured in the action through the forward named **error**.
	Note that the error messages are bound to the property named **error**. They can be displayed using:
	`<html:messages id="message" message="true" property="error">` `    error-message: <%=message%> ` `</html:messages>`
	The message keys of the error messages saved by the upload action in the request scope are as follows: **upload.error.filesize** and **upload.error.filenotsupported**.

The video upload feature works the same way as the image upload. The same Struts action can be used with the name of the video content type. An example struts configuration follows:

```
<form-bean name="uploadForm" type="com.ndc.usercontent.struts.actions.forms.UploadForm" />

<action path="/video/upload"
        name="uploadForm"
        scope="request"
        parameter="articleType=video"
        type="com.ndc.usercontent.struts.actions.upload.MediaContentUpload">
  <!--
    Required by the action since it will fail if neither a successUrl is defined
    in the form, nor a success forward is declared in the action declaration
  -->
  <forward name="success" path="/"/>
</action>
```

Note the value of the **parameter** attribute. It restricts the action into creating only content type of **video**. An example JSP template that allows a user to upload the video follows:

```
<html:form enctype="multipart/form-data" styleId="ImageUpload" action="/video/upload" method="post">
  <div>
    <html:hidden property="homeSectionId" value="${section.id}"/>
    <%--
      We don't need to specify successUrl since if no url is defined, the user will be redirected to
      the URL of the newly created resource.
    --%>
    <html:hidden property="errorUrl" value="${section.url}?view=uploadVideo"/>
    <html:hidden property="state" value="published"/>

    <label>Select Video</label>
    <html:file property="file" styleId="file"/>

    <label for="field-title">Title</label>
    <html:text property="field(title)" styleId="field-title"/>
```

```
    <label for="field-description">Description</label>
    <html:textarea property="field(description)" styleId="field-description"></html:textarea>
    <p/>
    <html:submit styleId="uploadButton" value="Upload Video"/>
  </div>
</html:form>
```

The form properties are the same as in the image upload form. Please note that, the same form bean of type **com.ndc.usercontent.struts.actions.forms.UploadForm** is used to upload videos.

4.7   Rating Content

VCE uses the DRW framework to allow developers to implement this feature.

The Java class that provides this feature is **com.ndc.qualification.ajax.QualificationPluginAjax**. Developers can access various methods of this class using DWR. Please have a look at the JavaDoc of the class shipped with the distribution to get an overview of the methods that can be invoked.

The method we are the most interested in has the signature: **StarRating submitStarRating(MetaData, Integer)**. Now, to be able to invoke this method using Javascript in the requesting user's browser, we need to perform the following steps:

1.   **Define the bean in the Spring context**. This can be achieved by adding the following XML segment in the spring bean definition XML (Sample configuration can be found in **$VCE_HOME/misc/contrib/ publication/WEB-INF/community-plugin-beans.xml**).

```
<bean id="qualificationPluginAjaxBean"
 class="com.ndc.qualification.ajax.QualificationPluginAjax" />
<bean id="qualificationPluginAjaxInterceptor"
 class="com.ndc.auth.filter.ajax.SecurityInterceptor" />
<bean id="qualificationPluginAjax" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="qualificationPluginAjaxBean" />
  <property name="interceptorNames">
    <idref local="qualificationPluginAjaxInterceptor" />
  </property>
</bean>
```

Note that the first bean declaration is of the class we are interested in. If we had declared the bean as follows: **<bean id="qualificationPluginAjax"**

```
class="com.ndc.qualification.ajax.QualificationPluginAjax"/>
```
and had left the other two bean declarations out, it would have worked as well. However, we then would not have any security constraints on the requesting user. Therefore, we have added the security interceptor to intercept method calls made to our bean **qualificationPluginAjaxBean** and named the proxy class **qualificationPluginAjax**.

2. **Allow method invocation through DWR**. Since we will be calling the method **submitStarRating**, we need to enable this method by adding the following XML segment in the DWR configuration XML (in the demo web application it would be **/WEB-INF/dwr.xml**).

```xml
<create creator="spring" javascript="QualificationPluginAjax">
  <param name="beanName" value="qualificationPluginAjax" />
  <include method="submitStarRating" />
  <include method="submitFlagging" />
  <include method="submitFavorite" />
  <include method="deleteFavorite" />
</create>
```

We have now enabled invocation of the above methods through DWR. Note the **beanName** property. The value is **qualificationPluginAjax** which refers to the proxy class that we declared earlier. Also note the value of the **javascript** attribute of the **create** element. DWR will bind the **javascript** variable needed to invoke the method with the name **QualificationPluginAjax**.

3. **Include the Javascript and invoke rating**. To do this, our JSP first needs to refer to the JS file that declares the **QualificationPluginAjax** methods.

```html
<script type='text/javascript'
  src='<util:valueof param="publication.url"/>dwr/interface/QualificationPluginAjax.js'></script>
```

Once the JS is included, we can write JS code as follows:

```javascript
function submitStarRating(articleId, userId, rating, element) {
 try {
  var meta = { userId:userId, articleId:articleId, sectionId:null, publicationId:null };
  QualificationPluginAjax.submitStarRating( meta, rating,
     {
       callback : function(rating) {
         submitStarRatingCallBack( rating, element );
       },
       exceptionHandler : submitStarRatingException
     }
   );
 } catch( ex ) {
  // JavaScript exception
 }
 return false;
}

function submitStarRatingCallBack(rating, element) {
 if (rating != null) {
  alert("you rated: " + rating);
 }
}

function submitStarRatingException(errorString, exception) {
 alert("unable to rate article, errror-message: " + exception.javaClassName);
}
```

The line of our interest is:
`QualificationPluginAjax.submitStarRating(meta, rating, ...)`. This call will:

1. Serialize the JS variable and send the request to the server
2. The DWR servlet on the server side will create the necessary Java objects
3. The DWR servlet will invoke the Java method **`submitStarRating`**
4. The DWR servlet will send response to the invoking JS code with the return value of the Java method

4.8   Adding Tags to Content and Creating Tag Clouds

Community Engine provides DWR functionality to add tags
to a content. The Java method implementing this feature is
`com.ndc.tag.ajax.TagPluginAjax#addTag()`.

To call methods on the **`TagPluginAjax`** class through DWR, a JavaScript file needs to be included in your JSP file. This JavaScript file contains JavaScript methods generated by DWR to call methods of **`TagPluginAjax`** class.

```
<script type='text/javascript'
  src='<util:valueof param="publication.url"/>dwr/interface/TagPluginAjax.js'></script>
```

If you look into http://yoursite:port/your-publication/dwr/interface/
TagPluginAjax.js, you will find a method **`TagAjaxAccess.addTag`** which allows you to add a tag to a content item.

Once you have tagged a content item, you probably will want to show this tag on the content page without reloading the page. Here is a JavaScript method **`submitTag`** which will call the **`TagAjaxAccess.addTag`** method and the callback method **`submitTagCallBack`** which will add the newly added tag to the given **`element`** of the HTML of your content page.

```
function submitTag(tagName, articleId, sectionId, userId, element) {
  try {
    var meta = { userId:userId,
                 articleId:articleId,
                 sectionId:null,
                 publicationId:null };
    TagAjaxAccess.addTag(meta, tagName,
                          {
                            callback:function(tag) {
                              submitTagCallBack(tag, element);
                            }
                          }
    );
  } catch (ex) {
    // handling the exception
  }
}

function submitTagCallBack(tag, element) {
  if (tag != null) {
    var elementHTML = $(element).html() + "&lt;span&gt;" + tag[0].name + "&lt;/span&gt;";
    $(element).html(elementHTML);
  }
}
```

Now, if you want to create a tag cloud, you can use the tag **`<tag:cloud/>`**. Here is an example script which will create a tag cloud with the top 20 tags

sorted alphabetically. The text size of the tag labels will be based on the tag popularity.

```
<tag:cloud id="tags" max="20" sizeMax="20" sizeMin="10" />

<c:forEach items="${tags}" var="tag">
  <a style="font-size: ${tag.tagSize}px" href="${your-tagpage-rul}/${tag.name}">
    ${tag.name}(${tag.frequency})
  </a>
</c:forEach>
```

For more details on the tag library provided by Community Engine for tagging functionality, see **Vizrt Community Expansion Taglib Reference: Chapter 9: tag** and **<stats:tagSuggest/>**, **<stats:tagPopularityList/>** in **Vizrt Community Expansion Taglib Reference: Chapter 8: stats**.

4.9   Adding Captcha Fields

To use captcha validation support in the actions, the **checkCaptcha** parameter needs to be set to **true** in the Struts action configuration. When this parameter is set, the action enforces captcha validation. For example, if we want to add captcha validation to the **/blog/add** action which is an instance of the class **com.ndc.usercontent.struts.actions.save.SaveArticle**, we need the following configuration.

```
<action path="/blog/add"
    name="articleForm"
    parameter="articleType=blog;checkCaptcha=true"
    type="com.ndc.usercontent.struts.actions.save.SaveArticle">
        <!-- Other action configuration stuff -->
</action>
```

Once the above configuration is in place, we can write JSP segment as part of an HTML form to show the captcha challenge interface. Please see **Advanced Developer Guide** for instructions.

When the user's input captcha code is wrong, the action will direct the user to the configured error page (which in common case is the same page where the user is filling the form). Have a look at **section 3.1** to get an overview on how to configure the error URL. Here is the JSP fragment to show captcha error in error page:

```
<logic:messagesPresent message="true" property="CAPTCHA">
  <p class="fieldError">
    <html:messages id="error" message="true" bundle="Validation" property="CAPTCHA">
      <bean:write name="error"/><br />
    </html:messages>
  </p>
</logic:messagesPresent>
```

The following VCE Struts actions come with built-in support for captcha verification:

- **com.ndc.usercontent.struts.actions.save.SaveArticle**

- **com.ndc.usercontent.struts.actions.save.SaveGroupProfile**

- **com.ndc.usercontent.struts.actions.save.SaveUserProfile**

4.10   Making Friends and Enemies

Perhaps one of the reasons behind the popularity of the community sites is their ability to let users make new friends. VCE comes with built-in functionality to let developers implement this feature on their community site.

4.10.1   Requesting Friendships

The feature is provided by Struts framework action. The Action can be declared as follows:

```
<form-bean name="friendshipForm" type="com.ndc.community.struts.actions.forms.FriendshipForm"/>

<action path="/friendship/request"
        name="friendshipForm"
        scope="request"
        type="com.ndc.community.struts.actions.friendship.RequestFriendship">
        <forward name="error" path="/friendship-error.jsp" />
        <forward name="success" path="/" redirect="true" />
</action>
```

An example JSP template to allow a user to invite any other user can be as follows:

```
<html:form action="/friendship/request"  method="post">
  <community:user id="currentUser"/>
  <community:user id="targetUser" userId="${param.userId}"/>

  <html:hidden property="userId" value="${currentUser.id}"/>
  <html:hidden property="friendId" value="${targetUser.id}"/>

  <html:hidden property="successUrl" value="${targetUser.section.url}" />
  <html:hidden property="errorUrl" value="${targetUser.section.url}?view=requestFriendship"/>

  <!-- Rest of the form fields related to textMessage content-type -->
  <html:hidden property="articleType" value="textMessage"/>
  <html:hidden property="state" value="published"/>
  Title: <html:text property="field(TITLE)"/><br/>
  Message: <html:text property="field(BODY)"/><br/>
  <html:submit>Request</html:submit>
</html:form>
```

The form properties above are explained as follows:

Form Property	Description
userId	The ID of the requesting user who wants to be friends. Needless to say, the property is required.
friendId	The ID of the user with whom the requesting user wants to be friends with. The property is also required.
articleType	The name of the content type of the message content that will be send to the requested user (identified by friendId).
state	Usually set to published since you will want the requested user see the message.
field(field-name)	The form properties are field names of the content-type. In our case, the textMessage content-type contains only two fields, Title and Body in the demo web-app.

Form Property	Description
`successUrl`	The URL to which the user should be directed to when the friendship request becomes successful.
`errorUrl`	The URL to which the user should be directed to if the friendship request results in an error. The errors are saved into request scope with different property names as follows:
	`RequestFriendship` : if no **friendId** is given.
	`error` : if the two users are already friends or an internal error has occured.
	`field(field-name)` : the field for which validation error has occured.

4.10.2 Accepting Friendship Requests

This section explains how to implement the functionality to allow a user to accept friendship requests. The functionality is yet again, provided by Struts framework actions. The Action declaration can be as follows:

```
<form-bean name="friendshipForm" type="com.ndc.community.struts.actions.forms.FriendshipForm"/>

<action path="/friendship/accept"
        name="friendshipForm"
        scope="request"
        type="com.ndc.community.struts.actions.friendship.AcceptFriendship">
        <forward name="error" path="/accept-friendship-error.jsp" />
        <forward name="success" path="/" redirect="true" />
</action>
```

The action definition and form properties for this feature is almost identical to that of the **Request Friendship** feature. But first, the user needs to see who has requested for friendship. An example JSP that will allow the user to see this is as follows:

```
<div>
  <community:user id="currentUser"/>
  <h3>Waiting Friendship Request(s)<h3/>
  <h3>(Wants to be friends with ${currentUser.article.fields.userName})</h3>
  <community:friends id="candidateFriends" name="currentUser" type="candidate"/>
  <logic:iterate id="candidate" name="candidateFriends" type="com.ndc.presentation.PresentationUser">
    <p>
      <util:valueof param="candidate.article.fieldElement(USERNAME)"/>
    </p>
  </logic:iterate>
</div>
```

Once the user can see who requested for his friendship, he can be presented a page where he can accept the request and perhaps send a response message back to user who requested his friendship. An example JSP template to accomplish this can be as follows:

```
<html:form action="/friendship/accept" method="post">
  <community:user id="currentUser"/>
  <community:user id="targetUser"/>

  <html:hidden property="friendId" value="${targetUser.id}"/>
  <html:hidden property="userId" value="${currentUser.id}"/>

 <html:hidden property="successUrl" value="${currentUser.section.url}"/>
 <html:hidden property="errorUrl" value="{targetUser.section.url}"/>
```

```
<!-- Rest of the form fields related to textMessage content-type -->
<html:hidden property="articleType" value="textMessage"/>
Title: <html:text property="field(TITLE)"/><br/>
Message: <html:text property="field(BODY)"/><br/>

<html:submit>Accept</html:submit>
</html:form>
```

Almost all the properties are the same as **Request Friendship** form. The error properties are probably worth mentioning since they are different.

- **AcceptFriendship** : if the message could not be send.
- **error** : if any internal error occurs while accepting the friendship.

4.10.3   Rejecting Friendship Requests

On a community site, a user may not wish to accept all friendship requests to keep some privacy. VCE lets a developer implement this functionality through yet another Struts framework action.

The action declaration can be as follows:

```
<form-bean name="friendshipForm" type="com.ndc.community.struts.actions.forms.FriendshipForm"/>

<action path="/friendship/decline"
        name="friendshipForm"
        scope="request"
        type="com.ndc.community.struts.actions.friendship.DeclineFriendship">
        <forward name="error" path="/decline-friendship-error.jsp" />
        <forward name="success" path="/" redirect="true" />
</action>
```

An example JSP that can allow a user to decline a friendship request can be as follows:

```
<html:form action="/friendship/decline">
  <community:user id="currentUser"/>
  <community:user id="requestingUser" userId="${param.userId}"/>

  <html:hidden property="userId" value="${currentUser.id}"/>
  <html:hidden property="friendId" value="${requestingUser.id}"/>
  <html:submit value="Decline"/>
</html:form>
```

4.10.4   Removing Friends

A community user may desire to remove someone from his friend list. This can be easily implemented through the following Struts action and JSP template:

```
<form-bean name="friendshipForm" type="com.ndc.community.struts.actions.forms.FriendshipForm"/>
<action path="/friendship/resign"
        name="friendshipForm"
        scope="request"
        type="com.ndc.community.struts.actions.friendship.ResignFriendship">
        <forward name="error" path="/" />
        <forward name="success" path="/" redirect="true" />
</action>
```

An example JSP template can be as follows:

```
<html:form action="/friendship/resign">
  <community:user id="currentUser"/>
  <community:user id="targetUser" userId="${param.userId}"/>
  <html:hidden property="userId" value="${currentUser.id}"/>
  <html:hidden property="friendId" value="${targetUser.id}"/>
  <html:submit value="Remove"/>
```

```
</html:form>
```

It may be desirable that the removed person does not get any notification of removal. A different Struts Action can be used to achieve this feature.

```
<form-bean name="friendshipForm" type="com.ndc.community.struts.actions.forms.FriendshipForm"/>
<action path="/friendship/remove"
        name="friendshipForm"
        scope="request"
        type="com.ndc.community.struts.actions.friendship.RemoveFriendship">
        <forward name="error" path="/" />
        <forward name="success" path="/" redirect="true" />
</action>
```

The action expects the same set of form properties and thus, the same JSP template can be used with the name of the action.

```
<html:form action="/friendship/remove">
  <community:user id="currentUser"/>
  <community:user id="targetUser" userId="${param.userId}"/>
  <html:hidden property="userId" value="${currentUser.id}"/>
  <html:hidden property="friendId" value="${targetUser.id}"/>
  <html:submit value="Remove"/>
</html:form>
```

4.10.5   Showing Friendship Notifications

The user should get a notification when his friendship request is accepted or rejected and may be when he is removed from someone's friend list. Here is an example script on how to show the notification that someone has accepted the friendship request.

```
<community:friendsNotifications id="acceptingUsers" name="communityUser" type="Accepted" />

<c:forEach var="acceptor" items="${acceptingUsers}" >
  <div>
    ${acceptor.article.title}
    <html:form action="/notification/friendship/clear">
      <html:hidden property="userId" value="${me.id}"/>
      <html:hidden property="friendId" value="${acceptor.id}"/>
      <html:submit value="Clear"/>
    </html:form>
  </div>
</c:forEach>
```

Similar notifications can be shown for rejection or removal of friendship using different values for the `type` attribute. For further details on this tag and other tags related to friendship, see **Vizrt Community Expansion Taglib Reference: Chapter 4: community**.

Once the user has read the notification, he will want to remove it. This way, he does not have a homepage stuffed with a lot of notifications. The script above showed how to clear the friendship notification using the Struts action **/ notification/friendship/clear**. The action class that serves this purpose is **com.ndc.community.struts.actions.friendship.ClearFriendshipNotification**. The form used for the action is **com.ndc.community.struts.actions.forms.FriendshipForm**

Form Property	Description
userId	The ID of the user who gets the notification.

Form Property	Description
`friendId`	The ID of the user whose action (acceptance, rejection or removal) has created the notification.

4.11   Creating Groups

One of the most common features provided by community sites is the users' ability to form groups together. VCE comes with built-in feature to support the concept of user groups.

A user group basically consists of a group profile content, a section and an owner of the group. To create a group, we will need a content type with the following parameter:

```
<parameter name="com.escenic.community.articleType" value="groupProfile"/>
```

The feature is implemented using the Struts framework. Hence we will need some Struts action declarations before we can write the template to let a user create a group. An example Struts configuration can be as follows:

```
<form-bean name="groupProfileForm"
  type="com.ndc.usercontent.struts.actions.forms.GroupProfileForm" />

<action path="/group/profile/add"
        name="groupProfileForm"
        scope="request"
        parameter="groupProfile"
        type="com.ndc.usercontent.struts.actions.save.SaveGroupProfile">
        <!-- Error and Success url are set in the template -->
</action>
```

Note that, the form class
com.ndc.usercontent.struts.actions.forms.GroupProfileForm is a subclass of the article form class
com.ndc.usercontent.struts.actions.forms.ArticleForm, so the form properties of the group form is almost identical to the properties of the article form.

An example JSP template that will let a user create a group can be as follows:

```
<community:user id="user"/>
<html:form styleId="addGroupForm" action="/group/profile/add">
  <html:hidden property="articleType" value="groupProfile"/>
  <html:hidden property="homeSectionId" value="${user.section.id}"/>
  <html:hidden property="state" value="published"/>

  Group name: <html:text property="field(name)"/><br/>
  Description: <html:text property="field(description)" /><br/>

  <html:hidden property="errorUrl" value="error-url" />
  <html:submit value="Create"/>
</html:form>
```

The **name** field is used as the group name. This is also used for group profile section url.

4.12 Joining and Leaving Groups

There can be two different workflow for users to join the groups. The first supposes that Charlie is the administrator of a group and Irene wants to join the group. Irene requests the membership of the group. Charlie gets the notification that Irene wants to join. Now, Charlie can accept her group membership or reject the request. Once Charlie has accepted the request, Irene is a member of the group.

In an alternative workflow, Charlie invites Irene to join the group. Irene can reject the invitation or accept it. Once Irene has accepted the invitation, she is a member of the group.

When Irene is a member of the group, she can leave the group. Alternatively, Charlie, being a group administrator, can remove Irene from the group.

Community Engine provides Struts `action` classes for implementing all these activities of Charlie and Irene described above. A tag library is also provided to show the notifications that Charlie and Irene should receive. For further details on group notifications, see in section **section 4.12.3**.

It is very important that all the group related actions are secured using Community Engine's `SecurityFilter` (i.e. through configuration rules in `security` publication resource). Example of rules for securing group membership related actions can be found in `security` resource file inside the `community-demo.war`.

4.12.1 User's Group Membership Actions

Community Engine provides a Struts `action` class `com.ndc.community.struts.actions.groupmembership.RequestGroupMembership` to perform a group membership request. Here is the Struts configuration for it:

```
<form-bean name="groupMemberForm" type="com.ndc.community.struts.actions.forms.GroupMembershipForm" /
>

<action path="/group/membership/request"
      name="groupMemberForm"
      scope="request"
      type="com.ndc.community.struts.actions.groupmembership.RequestGroupMembership">
  <forward name="error" path="/group-request-error.jsp" />
  <forward name="success" path="/group-request-success.jsp" redirect="true" />
</action>
```

Here is an example JSP script to use this `action` in group's section page

```
<community:group id="group" name="section"/>

<html:form styleId="groupMemberForm" action="/group/membership/request">
  <html:hidden property="groupId" value="${group.id}" />
  <html:hidden property="userId" value="${user.id}" />
  <html:submit value="Join This Group"/>
</html:form>
```

Form Property	Description
`groupId`	The ID of the group to be joined.

Form Property	Description
`userId`	The ID of the user who is making the request. Normally, this property refers to the ID of the user that performs an action on his group membership.
`memberId`	Not used in this `RequestGroupMembership action`. Normally, this property is not used for the `action` classes where the user performs change on his group membership.

There are other user initiated `action`s which can be used in the similar way.

- `com.ndc.community.struts.actions.groupmembership.AcceptGroupInvitation` - the user accepts an invitation to the group.

- `com.ndc.community.struts.actions.groupmembership.DeclineGroupInvitation` - the user declines an invitation to the group.

- `com.ndc.community.struts.actions.groupmembership.ResignGroupMembership` - the user quits the group.

4.12.2   Group Administrator's Actions

`com.ndc.community.struts.actions.groupmembership.AcceptGroupRequest` can be used for accepting a group membership request from a user. Here is the struts configuration to use this `action` class:

```
<form-bean name="groupMemberForm" type="com.ndc.community.struts.actions.forms.GroupMembershipForm" />

<action path="/group/request/accept"
        name="groupMemberForm"
        scope="request"
        type="com.ndc.community.struts.actions.groupmembership.AcceptGroupRequest">
  <forward name="error" path="/group-request-error.jsp" />
  <forward name="success" path="/group-request-success.jsp" redirect="true" />
</action>
```

Here is an example JSP script to let the group administrator accept the group membership requests:

```
<community:groupMembers id="candidates" name="group" type="candidate"/>

<c:forEach items="${candidates}" var="candidate">
  <html:form styleId="groupMemberForm" action="/group/request/accept">
    <html:hidden property="groupId" value="${group.id}" />
    <html:hidden property="memberId" value="${candidate.id}" />
    <html:submit value="Accept"/>
  </html:form>
</c:forEach>
```

Form Property	Description
`groupId`	The ID of the group to be joined.
`userId`	Not used in this `AcceptGroupRequest action`. Normally, this property is not used for the `action` classes where the group administrator performs change on user's group membership.
`memberId`	The ID of the user who's group member ship request is being processed. Normally, this property

Form Property	Description
	refers the ID of the user that group administrator deals with.

There are other group administrator initiated actions which can be used in the similar way.

- **com.ndc.community.struts.actions.groupmembership.InviteGroupMembership** - A group member or a group administrator invites an user to the group.

- **com.ndc.community.struts.actions.groupmembership.DeclineGroupRequest** - the group administrator declines a group membership request from an user.

- **com.ndc.community.struts.actions.groupmembership.RemoveGroupMembership** - the group administrator removes a user from the group.

4.12.3   Group Notifications

When Charlie accepts Irene's group membership request, Irene should be notified that her request has been accepted by the group administrator. VCE provides a tag library to retrieve this type of notifications.

Here is an example script on how to show the notification that group membership request has been accepted.

```
<community:user id="communityUser"/>
<community:groupsNotifications id="acceptedGroups" name="communityUser" type="Accepted"/>

<c:forEach var="acceptedGroup" items="${acceptedGroups}">
  Membership Accepted to group ${acceptedGroup.article.title}
  <html:form styleId="groupMemberForm" action="/notification/group/clear">
    <html:hidden property="groupId" value="${acceptedGroup.id}"/>
    <html:hidden property="userId" value="${communityUser.id}"/>
    <html:submit value="Clear"/>
  </html:form>
</c:forEach>
```

Similarly, the group administrator can see the notifications of acceptances or rejections of the group invitations he has sent. There is a tag **<community:groupMembersNotifications/>** to retrieve these notifications. For further details on group related tags, see **Vizrt Community Expansion Taglib Reference: Chapter 4: Community**.

Once the user has read the notification, he will want to remove that notification. The script above showed how to clear the group notifications using the Struts **action /notification/ group/clear**. The Struts **action** class serving this purpose is **com.ndc.community.struts.actions.groupmembership.ClearGroupMemberNotification** which uses the form class **com.ndc.community.struts.actions.forms.GroupMembershipForm**.

Form Property	Description
userId	The ID of the user whose notification is to be cleared.

Form Property	Description
`groupId`	The ID of the group for which notification is to be cleared.

4.13 User Karma

A typical use case is that you want to give active contributors to your community elevated rights as they reach a certain level. Typically, on a news site, you want to reward active bloggers and if you are creating the next World of Warcraft, you want to add functions to advance users' avatars through the epic sets.

Both of these scenarios are possible using VCE's reputation module.

Before this feature can be used, it needs to be enabled through publication feature resources.

```
qualification.userQualificationEnabled=true
qualification.initialUserGroup=Rookie
qualification.autoGroupChangeEnabled=true
```

Once these features are enabled, the publication will need a set of qualification groups with rating ranges based on which community users can be grouped. As an example, a community site may have some qualification groups defined as follows:

- `Rookie`: All users with qualification value [0, 1] belong to the group.
- `Regular`: All users with qualification value [1, 3] belong to the group.
- `Pro`: All users with qualification value [3, 5] belong to the group.

VCE automatically updates the users' qualification when someone rates an article. A user's current qualification group can be presented on the site as follows:

```
<%@ taglib uri="http://www.escenic.com/taglib/escenic-qualification" prefix="qual" %>
...
<community:user id="communityUser" name="section"/>
<div>
  <qual:userQualification id="qual" userId="communityUser.id"/>
  Group name: ${qual.group.name}
</div>
...
```

On many community sites, the best writers/authors of the site are listed on the front page. A feature like this can be implemented with the VCE taglibs as follows:

```
<qual:userQualifications id="userQualifications" groupName="Pro"/>
<div>
Group: ${qualGroup.name}<br/>
  <ul>
    <logic:iterate id="qualifiedUser" name="userQualifications"
                 type="com.ndc.qualification.api.domain.UserQualification">
      <community:user id="communityUser" userId="${qualifiedUser.userId}"/>
      <li>${communityUser.article.title}</li>
    </logic:iterate>
  </ul>
</div>
```

The above JSP segment will render all authors of the community belonging to the qualification group `Pro`.

4.14 Adding an Avatar to a User Profile

On a community site, it is very common for the users to have avatars or profile photos. On an VCE site, this functionality can be provided easily by relating an image content item to the user profile article of the user. Here is the preferred way to do this:

Create a relation type in your user profile content type definition. This relation type will only be used to related the profile image content items. Here is the sample configuration for creating a relation type named 'avatar':

```
<relation-type-group name="profile-relation">
  <relation-type name="avatar">
    <ui:label>Avatar</ui:label>
  </relation-type>
</relation-type-group>

<content-type name="userProfile">
  [...]
  <ref-relation-type-group name="profile-relation"/>
</content-type>
```

Use `com.escenic.community.actions.RelateContentAction` struts action class to provide the user a way to relate his image content items to his profile article. Use the relation type name(avatar) in the form property. Please see the action class javadoc for details.

For displaying profile images, show the related contents of 'avatar' relation type of the user profile article. In most of the cases, you may want to show the last profile image. Here is an example JSP fragment to show the profile image of the logged in user in any page:

```
<community:user id="vceUser"/>
<c:set var="avatarImageItems" value="${vceUser.article.relatedElements.avatar.items}"/>
<c:set var="lastAvatarIndex" value="${fn:length(avatarImageItems) -1}"/>

<c:if test="${lastAvatarIndex != -1}">
  <img src="${avatarImageItems[lastAvatarIndex].content.fields.ALTERNATES.value.thumb.href}"
 alt="User:">
</c:if>
<!-- Here 'ALTERNATES' is the name of the representation field of the image content -->
<!-- Here 'thumb' is the representation name of the image content -->
```

Note that, you can remove the image content relation from the user profile article using the action class `com.escenic.community.actions.RemoveContentRelationAction`. Please see the javadoc for details.

4.15 Enabling VizIWYG Functionality to the Demo Web Application

The demo publication that ships with VCE has VizIWYG functionality integrated with it. The user only needs to enable the functionality by adding some section

parameters required by VizIWYG. Two section parameters need to be set on the sections for which the user wants to enable VizIWYG.

The following parameter specifies the location to the VizIWYG web service that should be running if VizIWYG is installed properly. Typically, this parameter will be set on the root section of the publication.

```
viziwyg.webservice.url=http://SERVER:PORT/viziwyg-ws
```

The following parameter enables or disables VizIWYG for a section and its sub sections.

```
viziwyg.enabled=true
```

For detailed information on how to install, configure and use VizIWYG, please refer to the **VizIWYG Plug-in Guide**.

4.16   Fellow user activity

Fellow user activity means when a user performs an action on a content item then some activities are listed for the users (fellow users) who have performed the same action on the same cotnent item previously. By default when an action is performed on a content item, an action history entry is added for the user performing the action. As well as an entry is added for each of the fellow users.

Community Engine provides a way of managing this type of activities. This section shows you how to show these activities and how to configure to send notifications.

4.16.1   Showing the fellow user activities

Fellow user activities can be show using stats:actionList tag. Here is a sample script for doing this.

```
<stats:actionList id="feedList" type="fellowuserrating, fellowuserfavorites, fellowusercommenting"
        max="10" user="userProfile"/>
<ul>
<c:forEach items="${feedList}" var="feedAction">
  <c:set var="actionKey" value="${feedAction.actionType.shortKey}"/>
  <community:user id="fellowUser" userId="${feedAction.fellowUserId}"/>
  <li>
  <c:choose>
    <c:when test="${actionKey == 'fellowuserrating'}">
      <article:use articleId="${feedAction.articleId}">
      <a href="${fellowUser.section.url}">${fellowUser.article.fields.userName}</a> also rated <a
 href="${article.url}">${article.fields.title}</a>
      </article:use>
    </c:when>
    <c:when test="${actionKey == 'fellowuserfavorites'}">
      <article:use articleId="${feedAction.articleId}">
      <a href="${fellowUser.section.url}">${fellowUser.article.fields.userName}</a> also likes <a
 href="${article.url}">${article.fields.title}</a>
      </article:use>
    </c:when>
    <c:when test="${actionKey == 'fellowusercommenting'}">
      <article:use articleId="${feedAction.articleId}">
      <a href="${fellowUser.section.url}">${fellowUser.article.fields.userName}</a> also commented on
 <a href="${article.url}">${article.fields.title}</a>
      </article:use>
    </c:when>
  </c:choose>
```

```
    </li>
</c:forEach>
</ul>
```

For further details on this tag, see **Vizrt Community Expansion Taglib Reference: Chapter 8: stats:actionList**

4.16.2   Sending notification

There are two ways provided by Community Engine to send notifications.

• Notification by sending internal message

• Notification by sending e-mail

To enable this you have to add some parameters to the content-type which is used as the user profile

```
<content-type name="userInfo">
    ...
  <parameter name="notification.fellow.action.email" value="email-notification"/>
  <parameter name="notification.fellow.action.message" value="message-notification"/>
    ...
 </content-type>
```

The enabled field will be used to send notifications. If both of the fields are enabled then both internal message and e-mail will be sent.

5 Publication Context Configuration Files

The following files in a publication's webapp context (i.e. WEB-INF) has VCE related configuration:

- dwr.xml
- community-plugin-beans.xml
- struts-config-community.xml
- struts-config-qualification.xml
- struts-config-statistics.xml
- struts-config-usercontent.xml
- struts-config-forum.xml
- struts-config-messaging.xml
- struts-config-search.xml
- struts-config.xml
- web.xml

We have provided sample files of these in **$VCE_HOME/misc/contrib/ publication/WEB-INF**. The reason for these not to be included automatically when you re-build your web application with Assembly Tool, is that these may be edited to suit your needs or amended to your existing configuration.

Start with **web.xml.add** where you will see how the other files are called (the finished file must of course be called **web.xml**). Make your adjustments and copy the files to your webapp's WEB-INF, e.g.:

```
$ cd /opt/escenic/community-engine/misc/contrib/
$ cp -r publication publication-orig
$ emacs publication/WEB-INF/*
$ cp -r publication/WEB-INF/* /path/to/myproject/src/main/webapp/WEB-INF/
```

In addition to these files, the publication resource **/escenic/plugin/ community/security** must be present. A sample resource file can be found in **$VCE_HOME/misc/contrib/publication/META-INF/escenic/publication- resources/escenic/plugin/community/security**.

You can now develop your web application as normal and run the Escenic Assembly Tool to generate the finished WAR that can deployed on the application server. Please see the **Escenic Content Engine Installation Guide** for instructions on how to run the Assembly Tool.

6 3rd Party Content

3rd party content fetching functionality requires some configurations before using it. Please see details in **Vizrt Community Expansion Installation Guide: Section 9 3rd party content**

6.1   Default 3rd Party Services

The VCE relates to two content types for dealing with 3rd party content: `application-service` and `application`. The former for defining the what 3rd party applications should be available to the users of the web site and the latter for the user when adding a particular service to his or her profile.

Have a look at the content-type definition inside the community-demo WAR (`community-engine/wars/community-demo.war`), file which comes with the VCE distribution to see how these two content types are defined.

If you use the VCE demo WAR, these two content types are already available to you and you will also get the following application services defined:

- The Flickr picture community, http://flickr.com/
- The Picasa online image gallery, http://picasaweb.com
- The Last FM music community, http://last.fm
- The Youtube online video sharing community, http://www.youtube.com
- The Twitter microblogging service, http://twitter.com

To add 3rd party application fetching to your own publication: First add the two content types to your definition and then create similar application service content items; either by manually following the steps described below in Content Studio, or by using the `content` XML available in the demo WAR file.

6.2   Adding a New Third Party Application Service

To add another application service, open Escenic Content Studio and create a new `applicaiton-service` article. Be sure to fill in the following parameters correctly:

6.2.1   Name

Name of the service that will be used as a key for both background processes and template developers. Write it in lowercase with dashes ("-") to separate words. Take heed to choose a good name that is unique among your app service definitions.

6.2.2   URL

The URL of the service. Indicate the user specific parameter values on the form `http://myservice/?variable=${user-specific-value}`
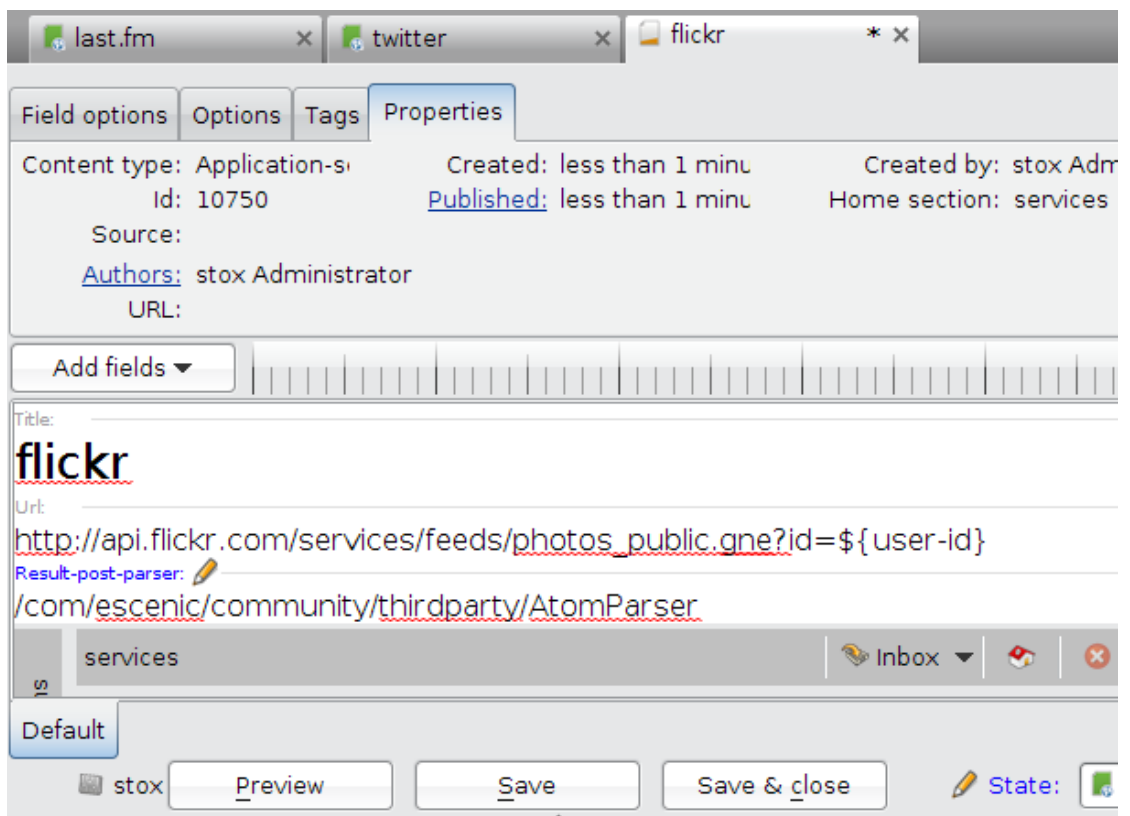
When the user adds this application to his/her profile, he/she will get prompted for a parameter called `user-specific-value`. You may have as many parameters as you like.

6.2.3   Post Result Parser

The content returned from the 3rd party service must normally be parsed before presenting it on the website. If you leave this field blank, the content from the service is displayed as is when accessing it from the JSP using `${article.fields.lastResult}` or `<article:field field="lastResult"/>`

The post result parser may be a Nursery component or just a class that resides in the application server's classpath. The only requirement for the parser is that it implements the `com.escenic.community.thirdparty.fetcher.ThirdPartyApplicationParser` interface.

Below you can see a screen shot where we add an application service content item to fetch http://flickr.com feeds:

Be sure to add the article to the `thirdparty/services` section and change the state of the article to "published" for VCE to pick it up.

6.3 Displaying All 3rd Party Apps for a User

This piece of code illustrates how to list all the applications a given user has added:

```
<article:list id="userApps"
  includeArticleTypes="<%= ThirdPartyConstants.APP_ARTICLE_TYPE %>"
  sectionUniqueName="${section.name}"
  includeSubSections="true"
  all="true"
/>

<c:forEach var="userApp" items="${userApps}">
  <div class="user-app">
    <h2>${userApp.title}</h2>
    <div class="user-app-contents">
      ${userApp.fields.lastResult}
    </div>
  </div>
</c:forEach>
```

6.4 Displaying All 3rd Party Apps Available

```
<article:list
  id="appArticles"
  includeArticleTypes="<%= ThirdPartyConstants.APP_SERVICE_ARTICLE_TYPE %>"
  sectionUniqueName="services"
  all="true"
/>
<ul>
  <c:forEach var="appArticle" items="${appArticles}">
    <li>
      add
      <a href="${section.url}?${pageContext.request.queryString}&addUserAppId=${appArticle.id}">
        ${appArticle.title}
      </a>
    </li>
  </c:forEach>
</ul>
```

6.5 Adding a 3rd Party App

Use the `com.escenic.community.forms.AddUserAppForm` and `com.escenic.community.actions.AddUserAppAction` Struts to add 3rd party apps to a given user. To create the form, you will also find `com.escenic.community.thirdparty.ThirdPartyUtil.getRequiredParameters(urlStrin` useful.

6.6 com.escenic.community.forms.AddUserAppForm

The form has the following parameters:

- `sectionId` : specifies the section ID to which the application article will be added. This is optional. By default, the home section of the logged in user is used.

- `appServiceId` : specifies the ID of the application-service article. The application-service article defines what the newly created application article will be an instance of.

- **parameter(param-name)** : specifies the application service article's parameter value where **param-name** specifies the name of the parameter. For example, if the URL configured in the application-service article is: **http://www.example.com/profile/${user-name},** then **user-name** is a required parameter. Thus, the form parameter should be **parameter(user-name)**. Likewise, the form must contain entries for all the parameters defined in the application service article URL.

- **successURL** : the URL where the user should be redirected to if adding the application to the user is successful.

- **errorURL** : the URL where the user should be redirected to if adding the application to the user fails.

7 SSO Support

7.1   Community Engine SSO Support

Community Engine now support Single Sign On (SSO). You can use a third party authentication provider (e.g. Facebook, Google, Yahoo) with Community Engine so that existing users of those providers can log into a web site built on Community Engine without going through (yet another) registration process.

7.2   Configuration

In order to make the Community Engine work with a third party authentication provider, you need to configure VCE with a set of providers.

7.2.1   Configure VCE to Work with Facebook

Please follow these steps to make VCE work with Facebook.

1. Create a `Facebook` application on the Facebook developer site: http://www.facebook.com/developers
2. Note the `Application ID` and the `Application Secret`
3. Go to the `Facebook Integration` tab
4. Update the `Canvas URL` with the URL of your publication, e.g. http://community-example.com/.
5. Create a Nursery configuration file in the Nursery path: `/com/escenic/community/sso/FacebookProvider.properties`
6. Add the following information to the Nursery configuration file created in the last step:

```
applicationID = the-application-ID-found-in-step-two
applicationSecret = the-application-secret-found-in-step-two
```

7.2.2   Configure VCE to Work with Google and Yahoo

Login function with Google and Yahoo is done using `com.escenic.community.sso.OpenIDProvider` and configured out of the box. No extra configuration is required for using these.

7.2.3   Configure VCE to Work with OpenID Providers

Copy the Nursery configuration file `$VCE_HOME/misc/siteconfig/com/escenic/community/sso/OpenIDProvider.properties` to your configuration layer in the Nursery path: `/com/escenic/community/sso/OpenIDProvider.properties`. A number of OpenID providers are added to this configuration. You can add more providers once you have tested that the OpenIDProvider works fine for your candidate provider. You can also remove some of the providers if you do not want to support them.

7.3   Using the SSO Login Functionality

In order to login using the SSO feature of the Community Engine, you need to configure VCE to use the `com.escenic.community.actions.SSOLogin` action class.

```
<form-bean
  name="signinForm"
  type="com.escenic.community.forms.SSOSignInForm" />

<action
  path="/auth/login"
  name="signinForm"
  scope="request"
  input="/index.jsp"
  type="com.escenic.community.actions.SSOLogin">
  <forward name="success"
           path="/auth/community/login.do" />
</action>

<action
  path="/auth/community/login"
  scope="request"
  name="signinForm"
  parameter="userProfile"
  type="com.ndc.usercontent.struts.actions.login.Login">
  <forward name="error" path="/" />
</action>
```

The `com.escenic.community.forms.SSOSignInForm` has the following properties:

- `providerId` : the configured SSO provider ID. Note that the ID of the provider can be retrieved using the `com.ndc.community.api.CommunityPlugin#getSSOProviderList` or `com.ndc.community.api.CommunityPlugin#getSSOProvider` method. Please see the JavaDoc for more information. Note that, if the providerId given is `-1`, the SSOLogin will proceed with the Community Engine user authentication.

- `userName` : the username of a VCE user. This property is not required when a SSO provider is used (i.e provider id is not `-1`). It is only for logging in using regular Community Engine user.

- `password` : the password of the VCE user identified by the `userName` property mentioned above. This property is not required when a SSO provider is used (i.e provider id is not `-1`). It is only for logging in using regular Community Engine user.

- `openid_identifier` : the URL or XRI chosen by the user as their OpenID identifier. It is not required when `openIDProviderIdentifier` is configured for the OpenIDProvider.

- `successUrl` : the URL where the user should be redirected to if the user successfully signs in.

- `errorUrl` : the URL where the user should be redirected to if the user can not sign in successfully.

Please note that, if the `providerId` is provided (i.e. if the user is using SSO), the other properties are ignored. Only if the providerId is set to `-1`, the rest of the properties are used by VCE.

8 User Qualification

The term `Qualification` is used in a sense of rating articles.

Whenever an article published by a user is qualified, the user gets reputed which is called user qualification. Its possible to create groups for user qualification and let users be part of a group based on his qualification.

As an example a publication may have four `UserQualificationGroup` defined:

1. Novice Writer - having score 1 to 2
2. Average Writer - 2 to 3
3. Good Writer - 3 to 4
4. Pro Writers - 4 to 5

Now if the articles of a user are qualified 2.5 on an average, then the user belongs to the qualification group Average Writer (no. 2). If the articles published by the user gets higher qualification, and the qualification of the user reaches 3 or above, he gets promoted to the next user qualification group Good Writer (no. 3) and so on.

8.1 Enabling User Qualification

User qualification groups can be created from Escenic Web Studio. Choose the `Community qualification` component and go to `Group management`. Clicking on the `Add group` link will display a form to create a new user qualification group.

To enable user qualification, the following two publication feature properties must be set:

```
qualification.userQualificationEnabled = true
qualification.initialUserGroup = A_QUALIFICATION_GROUP_NAME
```

You can use the following publication feature properties to configure user qualification module:

*   **`qualification.userQualificationTimeSpan`**
*   **`qualification.autoGroupChangeEnabled`**
*   **`qualification.minimumVotesForGroupChange`**
*   **`qualification.initialUserGroup`**
*   **`qualification.excludeGroupsFromAutoChange`**
*   **`qualification.userQualificationScoringFormula`**

See details in .

8.2 Managing User Qualification Group

Go to the `Community qualification` component in Escenic Web Studio and select the `User management` link. To get a list of users of an user qualification

group, choose a group from `view a group` list . You can also choose a group from `view recommended status changes` combo box to see a list of users the selected group who are recommended for a different user qualification group.

To change the user qualification group of a user, click on the user ID of a user from the list above. A combo box with available user qualification groups will occur. Select a group and click the `Move user` button.

9 Publication Feature Properties

Vizrt Community Expansion supports the following feature properties:

- **auth.MEMBER_NAME** : allows the publication to be configured with a member role name, default is **Member**. The authorisation module contains a set of roles which are assigned to users based on their actions. The **Member** role is used when a user becomes a member of a group.

- **auth.SECTION_OWNER_NAME** : default is **Section Owner**. This role is used to grant a user owner authority on a **Section**.

- **auth.GROUP_OWNER_NAME** : default is **Group Owner**. This role is used to grant a user owner authority on an VCE group.

- **auth.GROUP_MEMBER_NAME** : default is **Group Member**. This role is used to grant a user member authority on a VCE group.

- **auth.PARTNER** : default is **Partner**.

- **qualification.flaggingThreshold** : configures the minimum number of times an article must be flagged before it is moved to the flagging section. The article state is also set to **Draft** and **expireDate** property of the article is set to the current time once it reaches the threshold. The flagging section is identified by the unique name set for the feature **qualification.flaggedSectionUniqueName**

- **qualification.flaggedSectionUniqueName** : configures the unique name of the section used to contain flagged articles, default is **flagged**. Note that such a section must exist in ECE.

- **qualification.userQualificationEnabled** : enables/disables the qualification plug-in which decides whether user qualification should be allowed or not. Default is **false**, meaning it is disabled.

- **qualification.userQualificationTimeSpan** : configures the number of days to consider when a user's qualification is considered to retrieve qualification group, default is 30 (meaning 30 days).

- **qualification.autoGroupChangeEnabled** : default is **false** This property indicates if a user should be promoted/demoted automatically based on his user qualification.

- **qualification.minimumVotesForGroupChange** : default is **Zero** Configures the minimum number of votes a user must have to be promoted/demoted automatically based on his qualification.

- **qualification.initialUserGroup** : default is none. The initial group which a user must belong to when he first starts getting qualified.

- **qualification.excludeGroupsFromAutoChange** : a comma , separated **String** of group names, default is none. When a reputed user's qualification changes to a value which falls in the range of some qualification group other than his current group, he can be automatically moved to the new qualification group if VCE is configured to do so through the property: **qualification.autoGroupChangeEnabled**. But this property excludes automatic change of groups. This ensures that the user must not be automatically promoted/demoted to a group that is unwanted.

- **qualification.userQualificationScoringFormula** : configures a class which is used to calculate a user's score. Default is **com.ndc.qualification.plugin.util.AverageScoringFormula** which calculates the score by a simply dividing total rating (of all votes) by total number of votes.

    The configured class must implement the interface: **com.ndc.qualification.api.domain.util.ScoringFormula**

- **message.deleteRecordsOnDeletionUserEnabled** : dictates if all messages send by the user who is being deleted should be removed from VCE or not.

- **usercontent.maxUploadFileSize** : configures the maximum size of an image that can be uploaded. This applies to legacy images. The size is specified in bytes. Default size is 3 Megabytes (3 x 1024 x 1024 bytes).

- **usercontent.<content-type>.file.maxsize** : configures the maximum size of a media content item that can be uploaded. This applies for media contents which has a binary field for the media file. The size is specified in bytes. Default size is 3 x 1024 x 1024 bytes. To increase the max size of uploaded photographs, you would do:

```
# setting maximum file size of a image file for photo
# content type to 5MB
usercontent.photo.file.maxsize=5242880
```

10 community-security

Namespace URI

The namespace URI of the **community-security** schema is **http://schema.escenic.com/2010/community/security**.

Root Element

The root of a **community-security** file must be a **security** element.

10.1 action

Each **action** element describes one rule for security checking. It has one required attribute **pattern**, and, two optional attributes **user** and **author**.

Syntax

```
<action
    pattern="text"
    user="(true|false)"?
    author="(true|false)"?
 >
    <permission>...</permission>?
 </action>
```

Attributes

pattern="*text*"
    The **pattern** attribute is common to both **action** and **ajax** elements. The value of this attribute is an Ant like pattern string. Examples include **delete***, ***Blog, delete*Image**.

user="(true|false)" (optional)
    The **user** attribute is common to both **action** and **ajax** elements. This is a boolean attribute, and, the default value is **false**. If it is set to **true**, it will check if the given user is a logged in user.

author="(true|false)" (optional)
    The **author** attribute is common to both **action** and **ajax** elements. This is a boolean attribute, and, the default value is **false**. If it is set to **true**, it will check if the logged in user is the author of the given content item.

10.2 ajax

To secure DWR calls, an **ajax** element needs to be used. Similar to the **action** element, this element also has a required attribute **pattern**, and two optional attributes **user** and **author**.

Syntax

```
<ajax
    pattern="text"
```

```
    user="(true|false)"?
    author="(true|false)"?
>
    <permission>...</permission>?
</ajax>
```

Attributes

pattern="*text*"
> The **pattern** attribute is common to both **action** and **ajax** elements. The value of this attribute is an Ant like pattern string. Examples include **delete*, *Blog, delete*Image**.

user="(true|false)" (optional)
> The **user** attribute is common to both **action** and **ajax** elements. This is a boolean attribute, and, the default value is **false**. If it is set to **true**, it will check if the given user is a logged in user.

author="(true|false)" (optional)
> The **author** attribute is common to both **action** and **ajax** elements. This is a boolean attribute, and, the default value is **false**. If it is set to **true**, it will check if the logged in user is the author of the given content item.

10.3   permission

The **permission** element is common to both **action** and **ajax** elements. The value of this element is a string describing a permission for a certain action.

Syntax

```
<permission>
    text
</permission>
```

10.4   security

The **security** tag has two elements: **action** and **ajax**. Both of these elements are optional. A **security** element can have any number of **action** and **ajax** elements inside it. It has no attribute.

Syntax

```
<security>
    <action>...</action>*
    <ajax>...</ajax>*
</security>
```