

CloudView CV23

Mashup Programmer

Table of Contents

Mashup Programmer.....	4
What's New?.....	5
Packaging Custom Components.....	6
About Developing Custom Features.....	6
Installing a Development Environment.....	6
Developing Components with the CloudView Eclipse Plugin.....	7
Which Mashup components can you develop?.....	7
Why use it?.....	8
Generate WAR Files for Heavy Customization.....	8
Requirements.....	9
Generate a standard WAR file.....	9
Generate a standalone WAR File.....	10
Upgrading to a Newer Version.....	11
Customizing the Mashup UI.....	13
Understanding the SDK Architecture.....	13
Using Developer Tools.....	14
Monitor the Developer Area status.....	15
Develop and work on a non-packaged Mashup UI.....	15
Switch to Debug mode.....	15
Use the Mashup UI Debug tools.....	15
Customizing Style Sheets (CSS).....	17
Customizing the Mashup UI Language.....	17
About Internationalization features.....	17
Set default language.....	18
Add a new language.....	18
Manage I18N for multiple applications.....	19
Develop with I18N.....	19
Enforcing the Application Language.....	20
Creating Widgets.....	21
Widget architecture.....	22
Widget manifest.....	23
Create a widget.....	29
Create a widget template.....	30
Implement how to display subwidgets.....	31
Update widgets with Mashup Ajax Client.....	32
Troubleshooting.....	33
Creating Collaborative Widgets Using Storage Service.....	34
Creating Collaborative Widgets Using Storage Service.....	35
Storage type scopes.....	37
Common operations.....	38
How clients communicate with the Storage Service.....	39
Creating Feeds.....	44
Using the Eclipse plugin.....	45
Abstract class Feed.....	45
Sample Feed.....	46
Creating Triggers.....	47
About Feed and Design Triggers.....	47
Mashup UI interface.....	49
Mashup API interface.....	50
Creating Controllers.....	53
Create and package a controller.....	53
Reference JSP in a controller.....	54
Managing URL Rewriting.....	55
Enable URL rewriting.....	56
Configure URL rewriting.....	56

Implementing custom layout or templates as plugins.....	58
Using the Mashup API.....	59
About the Mashup API.....	59
Choosing between the Mashup API and the Search API.....	59
What you can do.....	59
How to choose between the two APIs.....	60
Using the Mashup API Java client.....	61
Where is the Mashup API endpoint?.....	61
How to configure a proxy.....	62
How to send security tokens to a secured Search API.....	62
How to configure failover.....	62
How to configure the max number of concurrent connections to the distant host.....	63
How to configure the stale connection check.....	63
How to configure a socket read timeout.....	63
Using the HTTP Mashup API.....	64
Using the Atom Output Format.....	65
namespace.....	65
link element.....	65
entry element.....	65
meta element.....	66
feed element.....	66
facet element.....	66
category element.....	66
Creating Parallel Requests.....	67
Configuring Hits Enrichment.....	69
About the Administration API.....	72
Administration Methods.....	72
Default Services Locations.....	73
Configuration System.....	73

Mashup Programmer

The Mashup framework is designed to build highly modular front-end applications. The Exalead CloudView Mashup Programmer's Guide describes how to develop your own applications and to go beyond the various possibilities offered by Mashup Builder. It gives examples illustrating the most common customization but does not cover all the possibilities of the framework which are manifold.

Audience

This guide is mainly destined to software programmers or users with a few programming skills.

Further reading

You might need to refer to the following guides:

Guide	for more details on
Mashup Builder	building the front-end of your search application.
Widget Reference	Mashup Builder widget descriptions.
Programmer	Exalead CloudView customization.
Configuration	indexing and search concepts, as well as advanced functionality.

What's New?

There are no enhancements in this release.

Packaging Custom Components

This section describes how to set up your environment and package custom components.

[About Developing Custom Features](#)

[Installing a Development Environment](#)

[Developing Components with the CloudView Eclipse Plugin](#)

[Generate WAR Files for Heavy Customization](#)

[Upgrading to a Newer Version](#)

About Developing Custom Features

While the most common Mashup UI customization use cases can be implemented using the Mashup Builder, it is sometimes necessary to develop your own mashup components.

If your Mashup UI customizations are:

- **minor**, we recommend using the Exalead CloudView Eclipse Plugin, which allows you to package customized items in zip files. For example, if you want to implement custom widgets, feeds, triggers etc.
- more **complex**, you need to generate and deploy a **complete WAR file**. For example, if you want to implement business logic using controllers on the server, or to enrich the JSP tag libraries to delegate complex java tasks.

To address these requirements, the Mashup UI SDK is delivered in the Exalead CloudView kit in the `sdk/java-mashupui` directory. It is **delivered as** a fully **standalone Eclipse project**, along with the javadocs of the different APIs.

Installing a Development Environment

Preparing your Development environment consists in:

- installing Eclipse,
- initializing the Mashup UI SDK,
- and importing the Mashup UI configuration file as a project in Eclipse.

You must have the following software on your machine:

- Java JDK 1.8 or higher (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>)
- Eclipse IDE for JAVA EE Developers Edition 3.6 or higher (<http://www.eclipse.org/downloads/>)

- Apache Tomcat server 7 or higher (<http://tomcat.apache.org/>)

Important: To let the Tomcat server know where the license is stored, export the `CVLICENSE` environment variable with the command: `export CVLICENSE=<path_to_license.dat>`

1. Go to your `<INSTALLDIR>/sdk/java-mashupui/project` directory
2. Initialize the SDK with the following:
 - a. run `init-sdk.<bat/sh>` depending on your operating system,
 - b. define the `hostname` of the Exalead CloudView instance,
 - c. define the `port` number of the Exalead CloudView instance,
 - d. define the Application id (for example, "default"),
 - e. enter the admin login and password.

A `development.properties` file is generated in: `<INSTALLDIR>/sdk/java-mashupui/project/war/WEB-INF/config`. The SDK can now communicate with Exalead CloudView to read the configuration and work with the Mashup API.

3. Import the Exalead CloudView mashup project into Eclipse:
 - a. Select **File > Import** and browse to your `<INSTALLDIR>/sdk/java-mashupui` directory.
 - b. Select the project and click **Finish**.

The mashup project is added to the eclipse **Project Explorer**.

4. Add the Apache Tomcat server into Eclipse:
 - a. Select **File > New > Server**
 - b. Define your Apache Tomcat server as a new server.
5. Run the Exalead CloudView project as "Tomcat Server", and go to: `http://localhost:8080/mashup-ui`

The Mashup UI opens.

Developing Components with the CloudView Eclipse Plugin

The Exalead CloudView Eclipse plugin is provided to help you develop and deploy plugins in Eclipse Indigo 3.7 or later. The documentation is packaged with this plugin and is available at <http://eclipse.exalead.com>

Which Mashup components can you develop?

Use the Exalead CloudView Eclipse plugin to develop and deploy as zip files your own Mashup components:

- Widgets
- Feeds
- Feed Triggers
- Mashup Triggers
- Pre-Request Triggers
- Security Providers

You can also develop custom components for Exalead CloudView core. For example, connectors, document processors, prefix handlers, security sources, etc.

Why use it?

- Easy deployment – If you use your Mashup UI application within your Exalead CloudView environment, you can package your plugin with customized components to be exported and deployed automatically on the selected Exalead CloudView instance.

If you want to deploy the Mashup UI outside of the Exalead CloudView environment, this avoids rebuilding and redeploying the `360-standalone-mashup-ui.war` package for each customized item.

- Quick export – You can package your plugin with classes that you want to export and then export it as a zip file on a selected path.
- Manage installs – You can list the deployed plugins on a selected instance of Exalead CloudView and then select the plugins to remove.

Generate WAR Files for Heavy Customization

If you need to make heavy customization on your Mashup UI applications that cannot be covered by adding a few custom plugins as zip files, you will have to compile the sources and regenerate a `.war` file.

Our framework allows you to generate:

- A standard `.war` file if you intend to use your Mashup UI application within your Exalead CloudView environment.
- A standalone `.war` file (embedding the 360 configuration) if you want to deploy the Mashup UI outside of the Exalead CloudView environment.

Requirements

Generate a standard WAR file

Generate a standalone WAR File

Requirements

Tools

You need to have the following tools properly installed and configured to recompile the sources and generate a `.war` file:

- `javac`, the java compiler found in the `jdk`
- `ant` (<http://ant.apache.org>)

Supported application servers

This section lists the supported platforms when deploying the Mashup UI outside of the Exalead CloudView environment.

Platform	Level of support
Apache Tomcat 6	Validated
Apache Tomcat 7	Compatible
Jetty 8	Validated

See <http://www.3ds.com/fileadmin/Support/Documents/Platform-support-policies.pdf> for Dassault Systèmes support policy.

Generate a standard WAR file

Configuration files for MashupUI/360/Exalead CloudView are fetched from the file system. Therefore, we recommend using a standard WAR file when the deployment server has file system access to Exalead CloudView configuration folders. Changes made to the Mashup UI configuration are immediately reflected in the application, no redeployment is needed.

1. Go to your `<INSTALLDIR>/sdk/java-mashupui/project` directory.

Note: If you want to move your project to another instance, you have to edit the `/WEB-INF/config/development.properties` config file which is automatically filled when you run `init-sdk.sh`. For more details, see step 2 in [Installing a Development Environment](#).

2. Run `cloudview-war.<bat/sh>` depending on your operating system.
3. Copy the `360-cloudview-mashup-ui.war` file into the `webapps` directory of your Exalead CloudView instance (`<DATADIR>/webapps`).
4. Go to the `<DATADIR>/config` directory and open the `Deployment.xml` file.

5. Find the `<Role>` named "MashupUI" and add:
6. In your `<DATADIR>/bin` directory, run the following script to apply the configuration files:
`cvcmd.sh applyConfig`
7. Restart Exalead CloudView.

Generate a standalone WAR File

It is possible to generate a standalone WAR file embedding the 360 configuration. This allows you to deploy the Mashup UI outside of the Exalead CloudView environment, for example, on another server.

To use the war with Tomcat, edit the `<installdir>/sdk/java-mashupui/project/war/WEB-INF/web.xml` file, and:

- Delete (or comment) the Gzip filter specific to Jetty.

```
<!--
<filter>
  <filter-name>GzipFilter</filter-name>
  <filter-class>org.eclipse.jetty.servlets.GzipFilter</filter-class>
  <init-param>
    <param-name>mimeTypes</param-name>
    <param-value>text/html,text/plain,text/xml,application/xhtml+xml,text/css,
      application/javascript,application/x-javascript,image/svg+xml</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>GzipFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
-->
```

- Delete (or comment) the default servlet.

```
<!--
<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>org.eclipse.jetty.servlet.DefaultServlet</servlet-class>
  -->
<!--
  servlet parameter to solve locked files on windows
  More information https://www.eclipse.org/jetty/documentation/9.4.x/troubleshooting-locked-files-on-windows.html
  -->
<!--
  <init-param>
    <param-name>useFileMappedBuffer</param-name>
```

```

    <param-value>false</param-value>
  </init-param>
  <load-on-startup>0</load-on-startup>
</servlet>
-->

```

1. Go to your `<INSTALLDIR>/sdk/java-mashupui/project` directory.
2. Run `standalone-war.<bat/sh>` depending on your operating system.

This prepares a `360-standalone-mashup-ui.war` file with an embedded 360 configuration.

3. Copy the `360-standalone-mashup-ui.war` into the serving path of your remote javax-compatible Web Server.
4. Start the Mashup UI.

Important: Redeploy the application every time you change the configuration through the Mashup Builder. The configuration is read-only in standalone mode.

Upgrading to a Newer Version

The Exalead CloudView migration option launched during the installation process, for example, `./install.<bat/sh> -migrate`, automatically includes all the components of the Exalead CloudView solution as well as configuration updates.

The migration option cannot migrate the custom code that you may have developed and deployed in `<DATADIR>/webapps/360-mashup-ui`.

During the installation process:

1. The system detects that you have an existing Mashup UI application and moves it to `webapps/360-mashup-ui.old`.
2. A new instance of the application is then deployed to the `webapps/360-mashup-ui` directory.

To migrate your custom code on the new application you have to copy back:

- Resources (stylesheets, images, javascript)
- Custom widgets
- Custom jar files
- Custom JSP files and tags

To ease the migration process, a good practice is to always make your custom code “visible”. For example, instead of:

- modifying the `style.css` file, create a new file that just contains your rules.

- modifying a deployed widget, copy it first to create your very own widget.

These good practices can significantly speed up the migration process.

Customizing the Mashup UI

This chapter describes how to create or customize all Mashup UI components using the Mashup UI SDK

[Understanding the SDK Architecture](#)

[Using Developer Tools](#)

[Customizing Style Sheets \(CSS\)](#)

[Customizing the Mashup UI Language](#)

[Enforcing the Application Language](#)

[Creating Widgets](#)

[Creating Collaborative Widgets Using Storage Service](#)

[Creating Feeds](#)

[Creating Triggers](#)

[Creating Controllers](#)

[Managing URL Rewriting](#)

[Implementing custom layout or templates as plugins](#)

Understanding the SDK Architecture

The Mashup UI SDK architecture (in `<INSTALLDIR>/sdk/java-mashupui/`) is based on the following conventions. Only key components are described here.

Directory	/sub-dir	Contains...
docs		The javadocs of the different APIs including the Widget Reference documentation.
project		Project root
	lib/	Exalead CloudView related libraries, already available in the global Exalead CloudView classpath.
	src-core/	Sources root. <ul style="list-style-type: none"><code>com/exalead/cv360/searchui/mvc/controller</code>: Application controllers

Directory	/sub-dir	Contains...
		<ul style="list-style-type: none"> ◦ <code>MashupController.java</code>: Main controller (establishes connection with Mashup API and render pages) ◦ <code>ConnectController.java</code>: Security / Authentication controller • <code>com/exalead/cv360/searchui/view/wi</code>: JSP Template tags • <code>widgets/tagcom/exalead/cv360/searchui/helper</code>: Simple helper classes • <code>com/exalead/cv360/searchui/services</code>: Mashup services for trigger and resource management
	<code>war/</code>	<p>Public content root.</p> <ul style="list-style-type: none"> • <code>WEB-INF/</code>: Web application code and configuration <ul style="list-style-type: none"> ◦ <code>web.xml</code>: Web application deployment file which contains information about enabled servlets and filters ◦ <code>urlrewrite.xml</code>: URL rewriting rules (advanced). Enable the URL Rewriting filter in the <code>web.xml</code> before use. ◦ <code>lib/</code>: Java libraries needed ONLY by this web application ◦ <code>jsp/</code>: JSP pages. <code>widgets/</code>: Contains all the widgets. For each widget, we have a <code>widget.xml</code> file (Widget definition file) and a <code>widget.jsp</code> file (Widget JSP code) ◦ <code>i18N/</code>: Contains language definition files for internationalization. • <code>resources/</code>: Application assets <ul style="list-style-type: none"> ◦ <code>css/</code>: Global application stylesheet ◦ <code>javascript/</code>: Global application javascript libraries ◦ <code>images/</code>: Global application images

Using Developer Tools

Mashup Builder includes several tools that can be useful to develop and debug your own search applications.

Monitor the Developer Area status

1. In **Mashup Builder**, click **Application** and select **Developer area**.
2. Look at the **Overview** section.

The tab displays a list flagging the various indicators status.

Note: You can click **Refresh** to get up-to-date status views.

3. From the **Actions** section, click **Check global configuration** to display the elements that are not configured as expected for production.

Develop and work on a non-packaged Mashup UI

To test your custom components (widgets, feeds, etc.) in a development environment like Eclipse, it is very useful to define a workshop Mashup UI. It allows you to test your components before deploying them on your Exalead CloudView instance.

1. In **Mashup Builder**, click **Application** and select **Developer area**.
2. In **Mashup UI URL**, enter the URL of your workshop Mashup UI.

Switch to Debug mode

1. In **Mashup Builder**, click **Application** and select **Developer area**.
2. In the **Mashup UI** section, select **Mashup UI debug mode**.

Results:

- The Messages panel displays that **The Mashup UI is currently in debug mode**.
- On the Mashup UI, a **Debug** bar appears at the bottom of your search application pages.

Use the Mashup UI Debug tools

When the Development mode is on in the Mashup Builder, the Mashup UI displays a **Debug** bar at the bottom of your search application pages.



Click	to...
I18N	Highlight the internationalization elements. 1. Select the language for which you want to edit the internationalization.

Click	to...
	<p>2. Hover above highlighted elements to see and edit the text that can be translated.</p> <p>Editing the text through the interface updates the <code>MashupI18N.xml</code> file, then, when I18N is reloaded it:</p> <ul style="list-style-type: none"> • (For edits on a specific language only) creates an <code>application_{lang}.properties</code> file. • (For edits on ALL languages) edits the <code>application.properties</code> file. <p>See also Manage I18N for multiple applications.</p> <p>Important: Do not modify the <code>application.properties</code> file directly!</p>
Widgets	<p>Highlight the various widgets used in the page. When you hover over one of them, you can see:</p> <ul style="list-style-type: none"> • its name (as defined in the <code>widget.xml</code> files) and WUID • the path of its JSP file template • the feeds it uses
Templates	<p>Highlight the various JSP components used in the page.</p> <p>When you hover over one of them, you can see its full path. This saves time to find the JSP that must be edited for a given component.</p>
Timeline	<p>Open a reporting screen if Reporting was enabled in the Mashup Builder in Application > Application Properties menu, and/or in Application > API Properties:</p> <ul style="list-style-type: none"> • The <code>mashup-ui-reporting</code> collects data relative to task execution and to CPU activity on the Mashup UI. For example, when a user queries a page, the reporter retrieves data such as the execution and CPU time of pages, widgets and triggers. • The <code>mashup-api-reporting</code> reporter collects data relative to feeds, subfeeds and triggers execution. This reporter allows you to understand explicitly the feed execution process, with subfeeds and triggers and to identify possible problematic issues.

Customizing Style Sheets (CSS)

One of the most common cases of user interface customization is to make the application look and feel the way you want. Even if some of these simple style issues can be fixed by changing the Image widget or adding a few CSS rules with a Custom CSS widget, sometimes you need to go further and make changes in the style of the application.

All the Mashup UI stylesheets are under the same global resource directory, located at:

`<DATADIR>/webapps/360-mashup-ui/resources/css`

Note: Each widget may declare additional stylesheets that are usually located under their `css/` directory: `<DATADIR>/webapps/360-mashup-ui/WEB-INF/jsp/widgets/<WIDGET NAME>/css`

Customizing the Mashup UI Language

This section describes how to customize the UI language using I18n.

[About Internationalization features](#)

[Set default language](#)

[Add a new language](#)

[Manage I18N for multiple applications](#)

[Develop with I18N](#)

About Internationalization features

Text labels and headings can be externalized into resource bundles, allowing the Mashup UI internationalization features to automatically display web pages in the appropriate language.

Three resource bundle files are used:

- `messages_en.properties` for English,
- `messages_fr.properties` for French,
- and `messages.properties`, the default used if no other file is found.

The following files contain key-value pairs representing text for the web pages in different languages, for example:

- `messages_en.properties` with `application.name=hello world`
- `messages_fr.properties` with `application.name=bonjour monde`

- `messages.properties` with `application.name=hello world`

Important: All properties defined in the widget's `messages*.properties` (`/WEB-INF/jsp/widgets/*/messages*.properties`) will override global properties defined in `/WEB-INF/i18n/messages*.properties`. The override mechanism is case-sensitive.

Set default language

You can change the Mashup UI default language in the product configuration. For example, to change the default language to French, set the value to `fr`.

1. Go to `<INSTALLDIR>/sdk/java-mashupui/project/war/WEB-INF/` directory.
2. Edit the `360-search-ui.xml` file.
3. To set a default language:
 - a. Uncomment the `defaultLocale` property
 - b. Set its value to the ISO language of your choice.

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLoc
<!--<property name="defaultLocale" value="en"/>-->
...
</bean>
```

Note: If necessary, you can also specify the country code of the language. For example, `en_US` represents U.S. English.

4. Save and close the file.
5. Go to **Administration Console > Home page** and restart the search server in the list of **Processes**.

Add a new language

We recommend maintaining all UI labels in a centralized `messages_<ISO NAME>.properties` file. Changing labels using the Debug mode I18n tool, as described in [Using the Mashup Builder Developer Tools](#), is not recommended, as it saves changes in another file.

- Do not configure your internationalization resources in the `application.properties` file.
- Editing the `messages*.properties` located in `<INSTALLDIR>/sdk/java-mashupui/project/war/WEB-INF/i18n/`, will impact the whole Mashup UI.
- To make a change for a specific widget only (for example, change a specific meta label), edit the `messages*.properties` file located in the widget's folder. The widget-specific file will override the global one.

Create a new *.properties file

1. Go to `<INSTALLDIR>/sdk/java-mashupui/project/war/WEB-INF/i18N`
2. Copy the `message.properties` file to create a new `messages_<ISO NAME>.properties` file.

Configure the UI labels

1. Edit the `messages_<ISO NAME>.properties` file
2. Edit the existing meta or facet label value you want to display in the UI.
3. To add a new meta, prefix it by `meta_` and use the following format: `meta_<new_meta>=<UI label>` For example, `meta_publicurl = Public URL`
4. To add a new facet, prefix it by `facet_` and use the following format: `facet_<facet_Root>=<UI label>` For example, `facet_Top/classproperties/file_folder=File folder`
5. Save and close the file.
6. Go to **Administration Console > Home page** and restart the search server in the list of **Processes**.

Manage I18N for multiple applications

If you have multiple Mashup UI applications made with a single Exalead CloudView instance, use the `MashupI18n.xml` file of each application to control its localization.

1. Go to the `<DATADIR>/config/360/applications/<app-name>/*`
2. Edit the `MashupI18n.xml` file of the application as needed. The following sample shows an application with two languages.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<MashupI18N xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  fallbackOnNormalizedForm="true" xsi:noNamespaceSchemaLocation="">
  <MessageList lang="fr">
    <Message message="Suggestions activées" code="noresults.suggestionEnabled"/>
  </MessageList>
  <MessageList lang="en">
    <Message message="Suggestions enabled" code="noresults.suggestionEnabled"/>
  </MessageList>
</MashupI18N>
```

Develop with I18N

This procedure gives the main steps to develop your own I18N features.

1. You must first add the `<SupportI18N>` tag in the `widget.xml`, as shown below.

```
<Widget name="Widget name">
    ...
    <SupportI18N supported="true" />
    ...
</Widget>
```

This `<SupportI18N>` xml tag allows all `messages*.properties` files located in the widget folder to be loaded by the `I18NLoader` automatically.

2. Convert JSP pages using the `<i18n:message>` tag from the `i18n` tag library.

```
<%@ taglib prefix="i18n" uri="http://www.exalead.com/jspapi/i18n"%>
```

The `<i18n:message>` tag is a subset of the Spring `<spring:message>` tag (see <http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/servlet/tags/MessageTag.html>)

```
<i18n:message code="application.name" text="hola mundo" />
```

If the `application.name` key is undefined in all properties files, the `text=""` attribute is used as backup.

3. By convention, all metas are prefixed by `meta_` and facets by `facet_`. To get consistency when displaying meta and facets between all widgets, you must set the following:

```
<i18n:message code="meta_${metaName}" text="${metaName}" />
<i18n:message code="facet_${fn:replace(category.path, ' ', '_')}" text="${category
```

4. I18N can be used server-side like in a custom controller. To achieve this, you need the `ServletContext` and the `HttpServletRequest`, for example:

```
import java.util.Locale;
import org.springframework.context.MessageSource;
import org.springframework.web.context.support.WebApplicationContextUtils;
MessageSource messageSource = (MessageSource) WebApplicationContextUtils.getWebApp
(this.pageContext.getServletContext()).getBean("messageSource");
Locale locale = RequestContextUtils.getLocale((HttpServletRequest) this.pageCont
String str = I18NLoader.getMessage(messageSource, "application.name", "hola mund
System.out.println(str);
```

Enforcing the Application Language

The Mashup UI uses by default the **I18N Trigger** at the application level, that is to say that it is executed for all pages. This trigger forwards a `lang` parameter to the Mashup API with the current locale of the Mashup UI.

Otherwise, the default configuration of the Mashup UI is to use a `LocaleResolver` called `CookieLocaleResolver`. This is a Spring interface that will:

- either send the locale stored in a cookie (`/mashup-ui/lang/{locale}`), which is created when the user selects a language, for example, when using the language selector widget.
- or, if there is no cookie, send either:
 - the default language (none set by default)
 - or the request's accept-header locale, which is the browser's language.

This resolver is configured in `<DATADIR>/webapps/360-mashup-ui/WEB-INF/360-search-ui.xml`:

```
<bean id="localeResolver" class="com.exalead.cv360.searchui.handler.CookieLocaleResolver">
  <!--<property name="defaultLocale" value="en"/>-->
  <property name="cookieName" value="clientlanguage"/>
  <property name="cookieMaxAge" value="100000"/>
</bean>
```

You may want to modify the `LocaleResolver` to enforce the language so that it will not be determined by the browser or be editable by users. Note that several implementations are available out-of-the-box:

- `AcceptHeaderLocaleResolver` – uses the primary locale specified in the "accept-language" header of the HTTP request.
- `FixedLocaleResolver` – always returns a fixed default locale.
- `SessionLocaleResolver` – uses a locale attribute in the user's session in case of a custom setting, with a fallback to the specified default locale or the request's accept-header locale.
- `CookieLocaleResolver` – uses a cookie sent back to the user in case of a custom setting, with a fallback to the specified default locale or the request's accept-header locale.

Creating Widgets

This section describes how to create, implement and update widgets.

[Widget architecture](#)

[Widget manifest](#)

[Create a widget](#)

[Create a widget template](#)

[Implement how to display subwidgets](#)

[Update widgets with Mashup Ajax Client](#)

[Troubleshooting](#)

Widget architecture

Widget Manifest and Logic

For each widget, there is a `<widgetName>` directory located in: `<DATADIR>/webapps/360-mashup-ui/WEB-INF/jsp/widgets`

Each widget has at least two files:

- `widget.xml`

The manifest, describing the widget and its options (used by the Administration Console).

- `widget.jsp`

The actual widget logic, using JSP.

The Tag Library Descriptors (TLD) documentation is available in the following directory:

`<INSTALLDIR>/sdk/java-mashupui/project/doc/tld`

Tip: If you have installed the Eclipse Plugin, you can open these TLDs from the **Project Explorer** panel. Right-click **MashupUI** and select **CloudView Mashup > Open tag libraries documentation**.

The Widget Reference documentation, which lists all available predefined widgets, is available in:

`<INSTALLDIR>/sdk/java-mashupui/project/doc/widget-reference/index.html`

Tip: If you have installed the Eclipse Plugin, you can open the widget reference documentation from the **Project Explorer** panel. Right-click **MashupUI** and select **CloudView Mashup > Open widgets documentation**.

Resources

A Widget can also contain resources such as images, JavaScript, CSS and so on. These resources must be located in sub-folders of the widget directory, for example:

- `images/`
 - `test-image.png`
- `css/`
 - `style.css`
- `js/`
 - `script.js`

Theses resources can be accessed through a conventional URL:

```
http://<HOSTNAME>:<BASEPORT>/<CONTEXT PATH>/resources/widgets/
<widgetName>/<sub-directory>/<resourcename.ext>
```

Example:

```
http://localhost:10000/mashup-ui/resources/widgets/myWidget/images/test-
image.png
```

You can refer to them in your widget JSP code with a relative path, using the tag `<wh:resource path="path/to/resource" />`

Example:

```
<wh:resource path="images/test-image.png" />
```

Libraries

The `<widgetName>/lib/` directory can contain a set of packaged JAR files. For example, you can add the jar of a controller under:

- myWidget/
 - lib/
 - myController.jar

Note: This is just an example. You do not necessarily need to create a widget to embed a controller. For more information about controllers, see [Create and package a controller](#).

Widget manifest

Let's take a look at a simple widget in , Hello World, to understand how a widget is structured:

widget.xml

The Widget Manifest file is required to create widgets in the Mashup Builder. For example, the Hello World Widget Manifest file is as follows:

```
<xml version="1.0" encoding='UTF-8'>
<Widget1 name="Hello World" group="Style Tools">
  <Description2> Widget description</Description>
  <Preview3>
    <![CDATA[
      <h2>Hello World</h2>
    ]]>
  </Preview>
  <Includes4>
    <Include type="css" path="css/style.css" />
    <Include type="css" path="/resources/static/css/style.css" />
  </Includes4>
</Widget1>
```

```

</Includes>
<SupportWidgetsId5 arity="ZERO_OR_MANY" />
<SupportFeedsId6 arity="MANY" />
<SupportI18N7 supported="true" />
<OptionsGroup8 name="Configuration">
  <Option9 id="person" name="Person" arity="ONE">
    <Description10>Person you want to say Hello to</Description>
    <Values11>
      <Value>World</Value>
      <Value>Tintin</Value>
    </Values>
    <Functions12>
      <ContextMenu13>Pages ()</ContextMenu>
      <Display14>SetHeight (3)</Display>
      <Check15>isPageName</Check>
    </Functions>
  </Option>
</OptionsGroup>
<DefaultValues16>
  <DefaultValue name="person">World</DefaultValue>
</DefaultValues>
</Widget>

```

This file contains the following information:

#	Element/tag	Description
1	Widget Declaration	<p>name: The widget's user friendly name.</p> <p>group: The widget's group name (can be hierarchical using slashes); used to group similar widgets in the Mashup Builder.</p>
2	Description	The widget's description.
3	Preview	The preview can contain any text or HTML embedded using CDATA tags.
4	Includes	<p>List of widget assets to inject in the page.</p> <ul style="list-style-type: none"> type: The type of the asset (css or js). path: Path of the resource. The path can be relative to the widget (for example, css/style.css) or absolute (for example, /resources/static/css.style.css).
5	SupportWidgetsId	<p>Defines how this widget supports sub widgets.</p> <p>arity: ZERO, ZERO_OR_MANY, ONE, MANY</p> <p>If you want to restrict the possibilities of sub widgets types, you can nest WidgetId tags to describe them: <WidgetId>tabContent</WidgetId></p>

#	Element/tag	Description
6	SupportFeedsId	<p>Defines how this widget supports feeds.</p> <p>arity: ZERO, ZERO_OR_MANY, ONE, MANY</p> <p>If you want to restrict the possibilities of feed types, you can nest <code>FeedId</code> tags to describe them: <code><FeedId>myFeed</FeedId></code></p>
7	SupportI18N	<p>Widget internationalization support.</p> <p>supported: Boolean value to enable internationalization</p>
8	OptionsGroup	<p>Declares an Option Group (displayed as a tab in the widget configuration).</p> <p>name: Option group name</p>
9	Option	<p>Declares a widget Option.</p> <ul style="list-style-type: none"> • <code>id</code>: Internal ID of the option, to be used to retrieve the option value • <code>name</code>: Displayable name of the option • <code>arity</code>: ZERO, ZERO_OR_MANY, ONE, MANY
10	Description	User-friendly option description.
11	Values	Declares the option's value(s). Declaring several values will create a selection box.
12	Functions	A container to describe this Option's expected display, behavior, error checking, etc.
13	ContextMenu	<p>Use this tag to specify the types of functions that will be available in the Value tab of the dynamic list displayed on the left of the widget properties panel:</p> <ul style="list-style-type: none"> • addContext(name, values): Appends a custom context to the dynamic list. The context is defined with names and values. For example, list <code>["a", "b"]</code> or <code>[{display:"A",value:"a_"}, {display:"B",value:"b_"}]</code> • Aggregations(facetOptionId): Gets all facet aggregations, for example, count, score etc. Optionally you can use <code>facetOptionId</code> to specify the ID of the option that contains the facet aggregation you want to retrieve. • ApiCommand(): Gets all the search API commands of all Search API configurations (see > Search API). • ApiConfig(): Gets the names of all the Search API configurations (see Administration Console > Search API). For example, <code>sapi0</code>, <code>sapi1</code>, <code>sapi2</code>, etc.

#	Element/tag	Description
		<ul style="list-style-type: none"> • appendOnChange(str): A click of the user on the dynamic list, will append the specified string to the active input. • Cookies(): Displays how to access the attributes of the dynamic list > Cookies parameters node in Mashup Expression Language. For example, the name attribute is accessed via the <code>\${cookies['__cookieName__'].name}</code> expression. • DataModelClass(): Gets a list of all data model classes (see Administration Console > Classes). • DateFacets(): Gets all Date facets from the feeds used by the widget. • emptyOnChange(): A click of the user on the dynamic list, will remove the current value of the option before setting the clicked value. • EntryVariables(): Displays how to access the variables of an entry in Mashup Expression Language. • Eval(): Gets all possibilities evaluated by the widget. • EvalCategory(): Gets all possibilities evaluated by the category. • EvalFacet(): Gets all possibilities evaluated by the facet. • EvalMeta(): Gets all possibilities evaluated by the meta. • Facets(facetType, refinementPolicy): Gets all facets from the feeds used by the widget. <code>facetType</code> can be a string <code>'DATETIME'</code> or an array <code>['DATETIME', 'NUMERICAL']</code> • Feeds(): Gets the feeds called by the widget. • FeedVariables(): Displays how to access the variables of a feed in Mashup Expression Language. • Fields(): Gets all the virtual fields, numerical fields and RAM-based fields from the feeds used by the widget. • GeoFacets(refinementPolicy): Gets only Geographical facets from the feeds used by the widget. • Hierarchical2DFacets(refinementPolicy): Gets only Hierarchical2D facets from the feeds used by the widget. • I18N(): Displays how to access the attributes of the dynamic list > Internationalization and localization parameters node in Mashup Expression Language. For example, the code attribute is accessed via the <code>\${i18n['__code__']}</code> expression.

#	Element/tag	Description
		<ul style="list-style-type: none"> • JsKeys(): Gets a list of all the I18N jskeys available. • Metas(): Gets all metas from the feeds used by the widget. • MultiDimension2DFacets(refinementPolicy): Gets Multi-dimension 2D facets from the feeds used by the widget. • MultiDimensionFacets(refinementPolicy): Gets Multi-dimension facets from the feeds used by the widget. • NormalFacets(refinementPolicy): Gets only Category facets from the feeds used by the widget. • NumericalFacets(refinementPolicy): Gets Numerical facets from the feeds used by the widget. • PageParameters(parameterId): Displays how to access the attributes of the dynamic list > Page parameters node in Mashup Expression Language. For example, the lang attribute is accessed via the <code>\${page.params['lang']}</code> expression. • Pages(): Gets the names of available pages. • PageVariables(): Displays how to access the variables of a page in MEL. • QueryPrefixes(): Gets all the query prefix handlers of the search logics (see Administration Console > Search logics > Query Language). • Reporters(): Gets all the reporters defined in Administration Console > Reporting). • Request(): Displays how to access the attributes of the dynamic list > Request parameters node in Mashup Expression Language. For example, the authPath attribute is accessed via the <code>\${request.authType}</code> expression. • SearchLogics(): Gets a list of all Search Logics (see Administration Console > Search logics). • SearchTargets(): Gets a list of all Search Targets (see Administration Console > Build Groups > Search targets). • Security(): When the user is logged in, it gets all user-related information (the token, the user name and the display name). • SecuritySources(): Gets a list of all Security Sources (see Administration Console > Security Sources). • SelfMetas(): Gets the metas of the current feed only.

#	Element/tag	Description
		<ul style="list-style-type: none"> • Session(): Displays how to access the attributes of the dynamic list > Session parameters node in Mashup Expression Language. For example, the creationTime attribute is accessed via the <code>\${session.creationTime}</code> expression. • Sorts(): Gets all the feed elements that can be sorted. • SubMetas(): Gets the current subfeed metas. • SuggestNames(): Gets the names of all the suggests and suggest dispatchers (see Administration Console > Suggest). • SuggestServices(): Gets the complete URLs of all the suggests and suggest dispatchers (see Administration Console > Suggest). • WUIDS(): Gets a list of this page Widget's unique IDs.
14	Display	<p>Use this tag to specify how the property will be displayed. For example, you can force the property to act as a select box, force it to act as an input (default is Text area), force it to act as a password input, etc.</p> <ul style="list-style-type: none"> • Code Editor: Transforms the property's input field into a code editor • Number: Transforms the property's input field into a number input field. • Password: Transforms the property's input field into a password input field (encrypted). • Radio: Transforms the property's input field into a radio button. • SetHeight: Sets the minimum height (in terms of lines) of the option's input field. • TextEditor: Transforms the property's input field into a rich text editor. • TextArea: Transforms the property's input field into a text area field. • ToggleDisplay: Shows/hides properties conditionally depending on the selected property value. For example, value1 of PropertyA will display PropertyB and PropertyD, value2 will display PropertyB and PropertyC. You need to set: * the value to match (using the <code>valueToMatch</code> and <code>ifEquals</code> attributes) * the options to hide (in <code>hideOptions</code>) * the options to show (in <code>showOptions</code>).
15	Check	<p>Error checking functions to validate user input.</p> <ul style="list-style-type: none"> • isInteger: Checks that the value is an integer and displays an error message if not.

#	Element/tag	Description
		<ul style="list-style-type: none"> • isAlphanum: Checks that the value is a chain of alphanumerical characters and displays an error message if not. • isPageName: Checks that the value is a page name and displays an error message if not. • isEmpty: Checks that the value is NOT empty and displays an error message if it is the case.
16	Default Values	Specifies the default value of the option (one of the values specified for the "Value" element).

widget.jsp

The actual code of the widget is pretty simple too, for example:

```
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>1
<%@ taglib prefix="wh" uri="http://www.exalead.com/widgets-helpers"%>2
<tiles:importAttribute name="widget" />3
<h2>${wh:getOption(widget, 'person') 4}</h2>
```

The file contains the following functions:

- Import of the tiles functions (required to retrieve our widget configuration).
- Import of the widget helpers functions (useful to manipulate widgets).
- Import of the "widget" variable (our widget configuration) into the current scope.
- Retrieval of the "person" option value.

Create a widget

This section explains how to create a new widget. Read the previous sections to get information on the two main widget components, the `widget.xml` file and the `widget.jsp` file.

Create a widget manually

1. Start by creating a new directory for your widget.
2. Describe it in the `widget.xml` file.
3. Add the logic in the `widget.jsp` file.

Your widget should appear automatically after reloading the widgets in the Mashup Builder, and should be ready to be used.

Note: JSP files are compiled at the application startup. Changing a .jsp file requires restarting the searchserver.

Create a widget using the Eclipse plugin

Using the Eclipse plugin is more convenient as the widget is created with all required files at once.

1. In the Eclipse **Project Explorer** panel, right-click a Mashup UI project.
2. Select **New > Other**. A wizard dialog box opens.
3. Select **CloudView Mashup Components > Mashup-UI widget** and click **Next**.
4. Define your widget general configuration:
 - a. Specify a source folder. Widgets are usually stored under `<project>/war/WEB-INF/jsp/widgets`
 - b. Give it a name.
 - c. Specify in which widget group this widget must be filed.
5. Click **Finish**.

The new widget is added to the selected project in the **Project Explorer** panel.

6. Edit the widget file as needed.

Create a widget template

You can customize the look-and-feel of several standard widgets supporting templates (for example, the **Result List** and **HTML** widgets), by creating and referencing your own widget templates.

These templates must be JSP files, that can either include specific HTML codes or reference CSS files.

Reference a JSP file in the Result List widget

1. In **Mashup Builder**, go to the **/search** page and select the **Design** view.
2. Click the header of the **Result List** widget.
3. On the widget properties panel, go to the **Hit templates** tab.
4. In the **Hit JSP template** field, enter the absolute path of your custom JSP file.

For example: `/WEB-INF/jsp/custom.jsp`

Reference a JSP file in the HTML widget

1. In **Mashup Builder**, open a page and select the **Design** view.
2. Drag the **HTML** widget to the **Design** view.

3. On the widget properties panel, go to the **Advanced** tab.
4. In the **JSP file path** field, enter the absolute path of your custom JSP file.

For example: `/WEB-INF/jsp/custom.jsp`

Implement how to display subwidgets

You may want to include widgets within other widgets, this is what we call subwidgets.

You can choose between two modes to display subwidgets in widget containers:

- **List** mode (default behavior), to display subwidgets in a list, one after the other.
- **Layout** mode, to display subwidgets in rows and columns. For example, see the **Tab** and **Table** widgets.

Once the widget is packaged with the required code in the `widget.xml` and `widget.jsp` files, a new **Edit Widget Layout** entry is available in the widget menu, to let you customize the layout, exactly like a page.

Display subwidgets in List mode

`widget.xml`

```
<SupportWidgetsId arity="ZERO_OR_MANY" displayType="LIST" />
<-- or -->
<SupportWidgetsId arity="ZERO_OR_MANY" />
```

`widget.jsp`

```
<%@ taglib prefix="render" uri="http://www.exalead.com/jspapi/render" %>
<%@ taglib prefix="widget" uri="http://www.exalead.com/jspapi/widget" %>
<!-- Iterate and render all subwidgets --%>
<render:subWidgets />
<!-- Iterate and render all subwidgets for a specific entry --%>
<widget:forEachSubWidget widgetContainer="${widget}" feed="${feed}" entry="${entry}" %>
    <render:widget />
</widget:forEachSubWidget>
```

Display subwidgets in Layout mode

`widget.xml`

```
<SupportWidgetsId arity="ZERO_OR_MANY" displayType="LAYOUT" />
```

`widget.jsp`

```
<%@ taglib prefix="render" uri="http://www.exalead.com/jspapi/render" %>
<%@ taglib prefix="widget" uri="http://www.exalead.com/jspapi/widget" %>
<render:import varWidget="widget" varParentEntry="parentEntry" varParentFeed="parentFeed" %>
```

```
<render:definition name="tableLayout">
  <render:parameter name="prefix" value="\${widget.wuid}" />
  <render:parameter name="layout" value="\${widget.layout}" />
  <render:parameter name="parentFeed" value="\${parentFeed}" />
  <render:parameter name="parentEntry" value="\${parentEntry}" />
</render:definition>
```

Update widgets with Mashup Ajax Client

Below are examples on how to:

Let's consider the following page:

```
<div class="container">
  <div class="wuid djH4dg djH4dg_0_page">
    <div class="wuid jkg934f jkg934f_0_cloudview">
    </div>
    <div class="wuid jkg934f jkg934f_1_cloudview">
    </div>
  </div>
</div>
```

The `$('.container')` parameter shown in the following code snippets, is the starting point for all DOM lookup by the client. For performance reasons, it should be as close as possible to the widgets that will be updated. If omitted, the whole body of the page will be used.

Update all subwidgets

The widget container has the wuid `jkg934f`

```
var client = new MashupAjaxClient($('.container'));
client.addWidget('jkg934f');
client.update();
```

Update the top widget

Here the widget at the top of the widget container has the wuid `djH4dg`

```
var client = new MashupAjaxClient($('.container'));
client.addWidget('djH4dg');
client.update();
```

Update a specific subwidget with extra parameters

Here the subwidget has the wuid `jkg934f_1_cloudview`

```
var client = new MashupAjaxClient($('.container'));
client.addWidget('jkg934f_1_cloudview');
client.addParameter('paramName', 'paramValue');
client.addParameters('paramName2', ['value1', 'value2']);
```



```
client.update();
```

Update a specific subwidget every 5 seconds

```
var client = new MashupAjaxClient($('.container'));
client.addWidget('jkg934f_1_cloudview');
client.updateInterval(5000);
```

Troubleshooting

You may encounter the following issues:

Changes in .jsp files are not taken in account

While changes to Javascript, CSS and images files are taken into account directly, JSP files are compiled by Jetty for performance reasons.

However, it is possible to switch the Jetty JSP servlet in development mode in the Mashup UI `web.xml` file located in `<DATADIR>/webapps/360-mashup-ui/WEB-INF/web.xml`. To do so, edit the following section:

```
<!-- Enables development mode on JSP servlet -->
<servlet>
  <servlet-name>jsp</servlet-name>
  <init-param>
    <param-name>development</param-name>
    <!-- SWITCH THIS TO TRUE TO GET YOUR JSP FILES RELOADED ON EACH REQUESTS -->
    <param-value>true</param-value>
  </init-param>
</servlet>
```

URL/XML encoding issues

Sometimes, the default behavior of the Mashup UI does not fit your needs:

- A meta value is URL encoded where it should not be, leading to invalid links (or the opposite).
- A meta value is not XML escaped, leading to invalid HTML (or the opposite).

Having complex rules to try to determine whether we should escape the content or not would be very hard and will obviously fail at some point. Instead, a default behavior per situation that matches 90% of the cases is used. For example, when building links, as in the **Hit title link** field, meta values will be URL encoded by default.

Nevertheless, a few flags have been added to let the user invert the default behavior which allows you to control everything by yourself:

- `!u`: Invert URL encoding, for example: `@{bi.publicurl!u}`

- `!x`: Invert XML escaping, for example: `@{bi.description!x}`
- `!h`: Remove the highlight on metas
- You can use `+` or `-` signs after the exclamation mark to force or remove the value, for example: `@{bi.publicurl!+u}` or `@{bi.description!-x}`

You can also combine flags, for example: `@{bi.name!ux}`.

I can't see my new code after deploying a custom plugin

For MS Windows deployments, it may happen that you can't see your newly coded element after deploying a custom plugin (e.g. a custom widget).

The problem comes from the new version of Jetty (9.4.) which locks the files it accesses in mmap mode. For more information, see

The problem was fixed by adding the following servlet definition in the Mashup UI `<DATADIR>/webapps/360-mashup-ui/WEB-INF/web.xml` file:

```
<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>org.eclipse.jetty.servlet.DefaultServlet</servlet-class>
  <init-param>
    <param-name>useFileMappedBuffer</param-name>
    <param-value>false</param-value>
  </init-param>
  <load-on-startup>0</load-on-startup>
</servlet>
```

Important: If you meet this problem in your Mashup UI instance, restarting the search server should solve the issue.

Creating Collaborative Widgets Using Storage Service

This section describes how to create and manage collaborative widgets.

Creating Collaborative Widgets Using Storage Service

Storage type scopes

Common operations

How clients communicate with the Storage Service

Creating Collaborative Widgets Using Storage Service

When using a collaborative widget, you actually enrich your document with data corresponding to the user's inputs. This data can be stored using the storage service.

The storage service is an untyped key/value storage that is accessed through a HTTP REST interface using simple commands such as `get` or `put`.

The storage service main intent is to provide persistence for Mashup UI applications that stretches beyond HTTP cookies and flat file storage. It is not a general-purpose database, and does not support SQL or any other query language.

Two client wrapper implementations exist for Javascript client use (asynchronous) and Java client use (synchronous).

Using the storage, you can:

- Get arbitrary chunks of data (JSON strings, images, Java objects, etc.) attached to your Mashup UI application.
- Delete, replace and append metas and categories to Exalead CloudView documents.
- Store private data per user.

Note: Keep in mind that data consistency/operation atomicity is not supported (not suitable for sensitive data).

Limitations

The storage does not:

- Support any kind of typing - All data that is put in the storage is treated as binary blobs, which means that they are not treated at all.
- Support transactions or ACID.
- Guarantee data consistency.

By default the storage works with an SQLite engine and is therefore limited in terms of:

- concurrent accesses, as each write action blocks the whole database,
- scalability, when the number of entries is really big.

However, you can configure its JDBC connection to work with compatible databases (SQL Server, MySQL or Oracle) if you need the storage to be more scalable.

For more information, see [Configure a compatible database for better performance](#).

Key/value pairs

Each key can be associated with one or [1:M] values. If a key name ends with '[]' it points to [1:M] values.

Examples:

- single value pair: shoppingListLength -> 3
- multi-valued pair: shoppingList[] -> [eggs, milk, toast]

The same key can exist in several scopes. Each pair is scoped using:

- The Mashup UI application ID. If you are running multiple Mashup UI applications in your Exalead CloudView instance they have different ids.
- The storage type scope which can be DOCUMENT, SHARED or USER.

Configure a compatible database for better performance

Tweaking the settings of the StorageService.xml file allows you to:

- Switch from SQLite to another RDBMS for storage backend.
- Change connection pooling parameters.
- Disable the internal locking mechanism for SQLite.

The Storage does not expose its settings in any GUI, instead you have to manipulate the

<DATADIR>/config/360/StorageService.xml file directly.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<StorageService xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="">
<StorageEngine className="com.exalead.cv360.storage.engine.jdbc.JdbcStorageEngine">
<!-- The JDBC driver's class name -->
<Parameter value="org.sqlite.JDBC" name="driver"/>
<!-- JDBC connection string, vendor-specific -->
<Parameter value="jdbc:sqlite:{dataDir}/storageService/storage.db.sqlite"
name="connection"/>
<!-- Tables are prefixed, useful if you are using the same database for many
different applications -->
<Parameter value="cv360" name="tablePrefix"/>
<!-- Internal read/write synchronization, disable for performance gain. Tries to
avoid potential file-system-level locking problems that can occur if any
of the following applies:
1. You are using SQLite over a networked file-system
2. You are using SQLite under Mac OS/X
3. You are using an old version of SQLite
If none of the above applies you can safely disable this option. See http://
www.sqlite.org/lockingv3.html for more information on this issue -->
```

```

<Parameter value="true" name="synchronizedWrites"/>
<!-- DB username / password -->
<Parameter value="" name="username"/>
<Parameter value="" name="password"/>
<!-- The database connections are pooled, to avoid reconnecting every time. When
using a non-SQLite RDBMS, You can tweak (increase) the settings below for a
performance gain.poolMaxActive: Max. num opened connections - Keep low for -->
<Parameter value="1" name="poolInitialSize"/>
<Parameter value="3" name="poolMaxActive"/>
<Parameter value="2" name="poolMaxIdle"/>
</StorageEngine>
<RepushDocuments>false</RepushDocuments>
</StorageService>

```

Important: After each configuration change, you need to run the `<DATADIR>/bin/cvcmd.sh applyConfig` script and restart your Exalead CloudView instance.

Storage type scopes

Document scope

The `DOCUMENT` scope is used for attaching pairs (metas) to Exalead CloudView documents. Pairs stored on documents can optionally be indexed and pushed back into the index.

The `DOCUMENT` scope saves the doc ID, the build group and the source of the document, to link the data to the document. This is how it works:

- When a document X is pushed in the storage, a pair A is added to the storage and linked to document X.
- A `repushFromCache` request is triggered on document X.
- The document travels through the analysis pipeline, the **StorageServiceDocumentProcessor** queries the storage to retrieve all the pairs linked to document X.
- Each document pair is added as a meta within document X. For example, the processor attaches meta A to document X (if a meta A already exists it is replaced, otherwise it is created). Multi-valued pairs are pushed as multi-valued metas.

User scope

The `USER` scope allows a user to store private data. If user X stores the key A in his/her user scope, it is not accessible to any other user. This is good for keeping per-user state in applications (shopping carts, preferences, etc.).

The `USER` scope data is not meant to be indexed by Exalead CloudView as it contains private user data.

Shared scope

The `SHARED` scope is a general-purpose scope which is shared across Mashup UI applications. It allows you to store data that you do not want to link to a document (see `DOCUMENT` scope).

The `SHARED` scope data is not meant to be indexed by Exalead CloudView.

Common operations

The common operations supported by the REST protocol are described in the following table.

Note: Both the Javascript and the Java clients have been designed to help you execute these operations.

Operation	Description
AGGREGATE	Returns aggregated values applied to the key. The possible operations are <code>COUNT</code> , <code>MIN</code> , <code>MAX</code> , <code>AVG</code> , <code>SUM</code> . The value must be a number except for <code>COUNT</code> which works with everything (numbers, alphanumerical strings, etc.)
GET	Gets all values associated with a single key.
GET_MANY	Gets many keys and all their corresponding values simultaneously.
ENUMERATE	Prefixed <code>get</code> . Provides a prefix and gets all pairs matching that prefix. For example, an enumerate query <code>'france_paris_'</code> would get all values associated with <code>'france_paris_chatelet'</code> , <code>'france_paris_montmartre'</code> , etc.'
SET	Adds a key/value pair to the storage. If the key exists, it is replaced or appended (whether it is single or multi-valued).
SET_MANY	Sets many values for a given key and replaces existing ones by the new ones. See also <code>PUT_MANY</code> .
PUT	Adds a key/value pair to the storage. If the key exists, an error is returned and the value is left untouched.
PUT_MANY	Adds several values to a given key and keeps existing ones.
DELETE	Deletes a single key with its associated values

How clients communicate with the Storage Service

There are two ways to communicate with the storage service:

The Java client communicates directly with the service at: `http://<HOSTNAME>:<BASEPORT+10>/storage-service`

The Javascript client communicates with the service through the storage proxy at: `http://<HOSTNAME>:<BASEPORT>/mashup-ui/storage`

The Javascript client is meant to communicate through the proxy for the 3 following reasons:

- The XHR requests issued by the Javascript client is subject to the cross-domain restrictions that apply to all XHR requests. Therefore, a proxy on the same port/domain is necessary.
- The proxy has rudimentary XSRF protection (X-Requested-With header checking) for the Javascript client's calls, preventing a user X to make changes to the state of user Y's data using XSRF.
- The USER scope. The proxy automatically appends the login name of the user who is currently logged in to the outgoing requests. When communicating directly with the storage service, the user needs to supply the user token manually.

Javascript client use

By default, the Javascript client is included in the Mashup UI. Your custom widget must reference it in its `widget.xml` file as shown below.

```
<Includes>
<Include type="js" path="/resources/javascript/storage/storage-client-0.2.js" />
</Includes>
```

You must use the `StorageService` class.

```
// Instantiates a new JS storage client
var storage = new StorageClient(storageType, url, options)
```

The parameters of the `StorageClient` class are described in the following table.

Parameter	Description
<code>storageType</code>	The value can be <code>user</code> , <code>shared</code> or <code>document</code> depending on the selected scope.
<code>url</code>	<p>The URL of the storage proxy:</p> <p><code>http://<HOSTNAME>:<BASEPORT>/mashup-ui/storage</code></p> <p>Note that the JS client is only capable of communicating with the proxy, never directly with the storage-service (because of the cross-domain restriction of XHR requests).</p>

Parameter	Description
	If you are in the context of a Mashup UI page, the <code>url</code> parameter is optional. It will be discovered by the storage client automatically.
<code>options</code>	Optional object that accepts any/all of the following properties: <ul style="list-style-type: none"> <code>timeout</code>: Timeout of the XHR requests in milliseconds (ms) before the error callback is invoked. <code>defaultErrorCallback</code>: <code>function(httpStatusCode, XmlHttpRequestObj, storageErrorEnum, storageErrorEnumDesc):</code> Overrides the default error handler that is invoked on every failed request. <code>defaultSuccessCallback</code>: <code>function(items, XmlHttpRequestObj):</code> Overrides the default success handler that is invoked on every successful request.

All requests are asynchronous. You always need a callback function to read the output from the client's calls:

```
// This will not work
// The alert is executed before the response is ever received
var storage = new StorageClient('shared');
var value = storage.get('myStorageKey');
alert(value); // undefined
// This will work better: A callback function is invoked
// after the client has received the response
storage.get('myStorageKey', function(items) {
    alert(items[0].value); // prints the first item in the collection
});
```

If the storage service client is instantiated with `USER` or `SHARED`, all non-destructive calls (`get`, `getMany`, `enumerate`) take a key parameter, a success and an error callback.

```
storage.get(keyString, successCallback, errorCallback);
storage.enumerate(successCallback, errorCallback);
storage.getMany([key1, key2, ...], successCallback, errorCallback);
storage.aggregate([key1, key2, ...], [aggr1, aggr2, ...], successCallback, errorCallback);
```

The `DOCUMENT` scope calls take extra parameters: `docBuildGroup`, `docSource`, `docUrl`

```
storage.get(docBuildGroup, docSource, docUrl, keyString, successCallback, errorCallback);
storage.enumerate(docBuildGroup, docSource, docUrl, KeyString, successCallback, errorCallback);
storage.enumerate(docBuildGroup, docSource, docUrl, successCallback, errorCallback);
// Gets all the possible permutations for the docObject and key parameters
var docObject1 = {
    docBuildGroup : "docBuildGroup",
    docSource : "docSource",
    docUrl : "docUri"
```



```
};
storage.getMany([docObject1, docObject2, ...], [key1, key2, ...], successCallback, errorCallback);
storage.aggregate(docBuildGroup, docSource, docUrl, [key1, key2, ...], [aggr1, aggr2, ...], successCallback, errorCallback);
```

For the **USER** and **SHARED** scopes, destructive calls look like this:

```
// singleKey is a key targeting a single element (Ex: 'mySingleKey')
// tries to put singleKey -> value in the storage, call errorCallback if
// key already exists for this scope
storage.put(singleKey, value, successCallback, errorCallback);
// puts singleKey -> value in the storage. If the key already exists,
// its value is overwritten.
storage.set(singleKey, value, successCallback, errorCallback);
// removes the pair with key singleKey from the storage
storage.del(singleKey, successCallback, errorCallback);
// multiKey is targeting [1:M] values
// appends another value to the bundle pointed to by multiKey
storage.put(multiKey, value, successCallback, errorCallback);
storage.set(multiKey, value, successCallback, errorCallback);
// adds several values to the key and keeps existing ones
storage.putMany(multiKey, [value1, value2, ...], successCallback, errorCallback);
// sets several values to the key and replaces existing ones
storage.setMany(multiKey, [value1, value2, ...], successCallback, errorCallback);
// deletes all values stored for multiKey
storage.del(multiKey, successCallback, errorCallback);
// multiKeyElement is targeting 1 value in a multi valued context
// The only way to get a 'multiKeyElement' is to get all the pairs for a multiKey,
// and then to use one of the keys in the response
storage.set(multiKeyElement, value, successCallback, errorCallback);
// updates the existing value
storage.del(multiKeyElement, successCallback, errorCallback); // deletes one value
```

For the **DOCUMENT** scope, destructive calls require:

- a **documentBuildGroup** (build group name),
- a **documentSource** (source connector name),
- and a **documentId** argument.

```
storage.put(documentBuildGroup, documentSource, documentId, singleKey, value, successCallback, errorCallback);
storage.set(documentBuildGroup, documentSource, documentId, singleKey, value, successCallback, errorCallback);
storage.del(documentBuildGroup, documentSource, documentId, singleKey, successCallback, errorCallback);
// adds several values to the key and keeps existing ones
storage.putMany(documentBuildGroup, documentSource, documentId, multiKey, [value1, value2, ...], successCallback, errorCallback);
// sets several values to the key and replaces existing ones
storage.setMany(documentBuildGroup, documentSource, documentId, multiKey, [value1, value2, ...], successCallback, errorCallback);
```

```

    successCallback, errorCallback);
// deletes all pairs stored for a given document:
storage.del(documentBuildGroup, documentSource, documentId, successCallback, errorCallback);
// ... the multiKey and multiKeyElement examples are like above, but with
// documentBuildGroup, documentSource, documentId prepended the other parameters

```

For real examples, go to your <DATADIR>/webapps/mashup-ui/WEB-INF/jsp/widgets/ directory, and look at the source code of the following widgets:

- `savedQueries`: which uses multi-valued keys with `USER` or `SHARED` storage
- `todoList`: which uses single-valued keys with `USER` or `SHARED` storage
- `starRating`: which uses single-valued keys with `DOCUMENT` storage.

These widgets make extensive use of the storage service.

Java client use

To use the Java client, make sure that the `360-storage-client.jar` is included in your Eclipse project.

All the Java client's calls are synchronous, meaning that they are blocked until a response is received from the server.

The Java client is made to communicate directly with the storage service at:

`http://<HOSTNAME>:<BASEPORT+10>/storage-service`

The Java client provides very destructive methods (clear operations) therefore it should be used with caution.

```

StorageClient client = new StorageClient(Constants.STORAGE_SERVICE_URL, Constants.MAS
client.scratch(); // Scratches everything in the Storage
client.scratchForApplication("default"); // Scratches everything for application 'def
Other applications' states are preserved
client.scratchForWidget("default", "starRating"); // Scratches all states for the 's
of application 'default'
DocumentClient dclient = client.getDocumentStorage();
//For Cloudview Documents: use a document descriptor(buildgroup,source,docurl)
DocumentDescriptor descriptor = new DocumentDescriptor("bg0", "docSource", "doc1");
dclient.put(descriptor , "helloWorldKey" "Hello World DocumentStorage!".getBytes("UTF
// puts value if key does not already exist
List<byte[]> myValues = new ArrayList<byte[]>();
myValues.add("Hello World DocumentStorage!".getBytes("UTF-8"));
myValues.add("Good bye!".getBytes("UTF-8"));
dclient.putMany(descriptor , "helloWorldKey[]", myValues);
//puts multiple values to the key and keeps existing ones
dclient.setMany(descriptor , "helloWorldKey[]", myValues);
//sets multiple values to the key and replaces existing values
Entry entry = dclient.get(descriptor , "helloWorldKey");

```

```
dclient.set(descriptor , "helloWorldKeyDuplicate", entry.getValue());
// replaces whatever value (if any) that was previously set in helloWorldKeyDuplicate
DocumentDescriptor descriptor = new DocumentDescriptor("buildGroup", "source", "document");
String[] bagKey = new String[] {"testbagkey[]"};
StorageAggregationType[] aggrs = new StorageAggregationType[] {
    StorageAggregationType.COUNT,
    StorageAggregationType.AVG,
    StorageAggregationType.MAX,
    StorageAggregationType.MIN,
    StorageAggregationType.SUM
};
List<AggregationsResult> resultList = this.doc.aggregate(descriptor, bagKey, aggrs);
```

Example: Simple 'Badge Manager'

On some social networks, a user's profile can be awarded with one or several predefined badges. Badges can be represented as a string of text, and each badge can be either 'on' or 'off'. In the example below, a badge is 'on' if the text string is stored for a particular user, and 'off' if it does not exist.

The class attaches one or several predefined text strings to a particular Exalead CloudView document with a multi-valued key ending with '[]'. No other values than the predefined ones are allowed to exist on the key.

```
public class BadgeManager {
    private final String badgeDatabaseKey;
    private final ImmutableSet<String> availableBadges;
    private final DocumentStorage db;
    // availableBadges are a list of the predefined text-strings that you want to use,
    // for example ['cool', 'humid', 'warm', 'really-warm']
    // badgeTag is the multi-valued key to store on in the storage, for example "perceive
    public BadgeManager(String[] availableBadges, String badgeTag) {
        this.availableBadges = ImmutableSet.of(availableBadges);
        this.badgeDatabaseKey = badgeTag;
    }
    // Instantiate the Java Storage Client given the URL to the Storage Service
    // (Normally http://CVHOST:[BASEPORT+10]/storage-service) and the Mashup Application
    // Instead of .getDocumentStorage() we could have used .getUserStorage() or .getShare
    this.db = new StorageClient(Constants.STORAGE_SERVICE_URL,
        Constants.MASHUPUI_APPID).getDocumentStorage();
    }
    public Set<String> getAvailableBadges() {
        return this.availableBadges;
    }
    public void addBadge(DocumentDescriptor document, String badge) throws Except
        if (!hasBadgeEntry(componentId, badge)) {
    }
    // To put a pair on a document we supply the Document Source and the document id
    // The document source is the source connector's name, the document id is the URI of
    // All values are stored as byte blobs, so the string needs to be converted into a by
```

```

        db.put(document, badgeDatabaseKey, badge.getBytes("UTF-8"));
    }
}

public void removeBadge(DocumentDescriptor document, String badge) throws Exception {
    if (hasBadgeEntry(document, badge)) {
        Entry toRemove = getBadgeEntry(document, badge);
// To delete a single entry from a multi-valued meta an argument "unique" is needed.
// Unique is an extra key that identifies a single value in a multi-valued pair
        deleteByUniqueKey(document, toRemove.getKey().getKey(), toRemove.getValue());
    }
}

public Set<String> getAllBadgesFor(DocumentDescriptor document) throws Exception {
    Set<String> badges = Sets.newHashSet();
    for (Entry e : getBadgesFromStorage(document)) {
        badges.add(new String(e.getValue(), "UTF-8"));
    }
    return badges;
}

public void removeAllFor(DocumentDescriptor document) throws StorageClientException {
    // If no unique argument is provided, all of the pair's values are deleted
    db.delete(document, badgeDatabaseKey);
}

private boolean hasBadgeEntry(String documentId, String badge) throws Exception {
    return getBadgeEntry(document, badge) != null;
}

private Entry getBadgeEntry(DocumentDescriptor document, String badge) throws Exception {
    for (Entry b : getBadgesFromStorage(document)) {
        if (new String(b.getValue(), "UTF-8").equals(badge)) {
            return b;
        }
    }
    return null;
}

private List<Entry> getBadgesFromStorage(DocumentDescriptor document) throws Exception {
    // Returns all the values associated with the document 'docId' and the badgeDatabaseKey
    return db.get(document, badgeDatabaseKey);
}
}

```

Creating Feeds

Creating feeds can be useful when the standard feeds library is not sufficient. For example, if you want to connect and retrieve data from an unsupported database.

Using the Eclipse plugin

The Eclipse plugin allows you to create the feed with all required files at once.

1. In Eclipse, select **File > New > Other**.
2. Select **CloudView Mashup Components > Mashup-API feed** and click **Next**.
3. Define your feed general configuration:
 - a. Specify a source folder.
 - b. Specify a package.
 - c. Give it a name.
4. Click Finish.

The new feed is added to the specified source folder in the **Project Explorer** panel. Edit the feed files as needed.

Abstract class Feed

```
public abstract class Feed {
    /*
     * API to implement
     */
    /**
     * Return a human friendly name for this Feed, to be used by the Administration C
     */
    abstract public String getDisplayName();
    /**
     * Execute routine. Called when receiving a request without any specific ID.
     * @param context
     * @return Return the ResultFeed for the given QueryContext
     * @throws AccessException
     */
    abstract public ResultFeed execute(QueryContext context) throws AccessException;
    /**
     * Get routine. Called when receiving a request with a specific ID. Must return o
     * @param context
     * @param id
     * @throws AccessException
     */
    abstract public ResultFeed get(QueryContext context, String id) throws AccessExce
    /**
     * Get the list of available metas for the given feed configuration.
     * @param feedConf
     * @throws AccessException
     */
    @Override
```

```

    public AvailableMetas getAvailableMetas(Map<String, String[]> feedConf) throws AccessException {
        AvailableMetas m = new AvailableMetas();
        String[] mv = { "metaName", "feedOption1" };
        m.addType(new AvailableMetas.Type("all", mv, null));
        return m;
    }
    /**
     * Get the list of supported parameters by this Feed.
     * @return An array of supported Parameters
     */
    abstract protected Parameter[] getSupportedParameters();
}

```

Sample Feed

```

import java.util.HashMap;
import java.util.Map;
import com.exalead.access.feedapi.AccessException;
import com.exalead.access.feedapi.Entry;
import com.exalead.access.feedapi.Feed;
import com.exalead.access.feedapi.Meta;
import com.exalead.access.feedapi.QueryContext;
import com.exalead.access.feedapi.ResultFeed;
import com.exalead.access.feedapi.utils.FeedHelper;
import com.exalead.access.feedapi.v10.AvailableMetas;
import com.exalead.access.feedapi.v10.Parameter;
import com.exalead.cv360.customcomponents.CustomComponent;
@CustomComponent(displayName = "Sample Feed")
public class SampleFeed extends Feed {
    @Override
    public String getDisplayName() {
        return "Sample Feed";
    }
    @Override
    public ResultFeed execute(QueryContext context) throws AccessException {
        String feedOption1 = this.getEvaluatedParameter(context, "feedOption1");
        ResultFeed r = new ResultFeed(this);
        Entry e = new Entry("42");
        e.addMeta(new Meta("metaName", "metaValue"));
        e.addMeta(new Meta("feedOption1", feedOption1));
        r.addEntry(e);
        return r;
    }
    @Override
    public ResultFeed get(QueryContext context, String id) throws AccessException {
        return null;
    }
    @Override
    public AvailableMetas getAvailableMetas(Map<String, String[]> feedConf) throws AccessException {

```

```

    AvailableMetas m = new AvailableMetas();
    String[] mv = { "metaName", "feedOption1" };
    m.addType(new AvailableMetas.Type("all", mv, null));
    return m;
}

@Override
protected Parameter[] getSupportedParameters() {
    return new Parameter[] { new Parameter("feedOption1") };
}

public static void main(String[] args) throws Exception {
    com.exalead.access.configuration.v10.Feed config = new com.exalead.access.configuration.v10.Feed();
    config.addParameter(new com.exalead.access.configuration.v10.Feed.Parameter("feedOption1", "feedOption1Value"));
    Feed pageSearch = FeedHelper.getTestPageFeed("search");
    pageSearch.getSubFeeds().add(FeedHelper.getTestFeed(SampleFeed.class, "test", config));
    // FeedTrigger trigger = new testFeedTrigger();
    // testFeed.getTriggers().add(trigger);
    Map<String, String> params = new HashMap<String, String>();
    params.put("test.feedOption1", "feedOption1Value");
    ResultFeed result = FeedHelper.runFirstFeed(pageSearch, params);
    for (Entry e : result.getEntries()) {
        for (Object s : e.getMetas().keySet().toArray()) {
            System.out.println(s + " : " + e.getMeta((String) s).getFirstValue());
        }
    }
}
}

```

Creating Triggers

This section describes how to create triggers. Using the Exalead CloudView Eclipse plugin, select **File > New > Other > CloudView Mashup Components > <Type of> Trigger**

[About Feed and Design Triggers](#)

[Mashup UI interface](#)

[Mashup API interface](#)

About Feed and Design Triggers

Feed Triggers

A Feed Trigger is an entry point to alter the behavior of a Feed. It is called by the Mashup API before and after the feed execution, allowing for query manipulation, context modification and results manipulation. Therefore, Feed Triggers can do the following:

- Decide to override the query to be issued to the actual 'execute' method based on query expansion (`beforeQuery` method)
- Decide to replay the feed execution because the result obtained is not satisfying, for example, if there are no results (`afterQuery` method that returns `Result.EVAL_AGAIN`)

Example:

You can use a Feed Trigger on any Feed in your configuration to customize query processing or feeds behavior for different purposes such as:

- Query rewriting
- Query computing from previously retrieved results
- Enabling / Disabling feeds

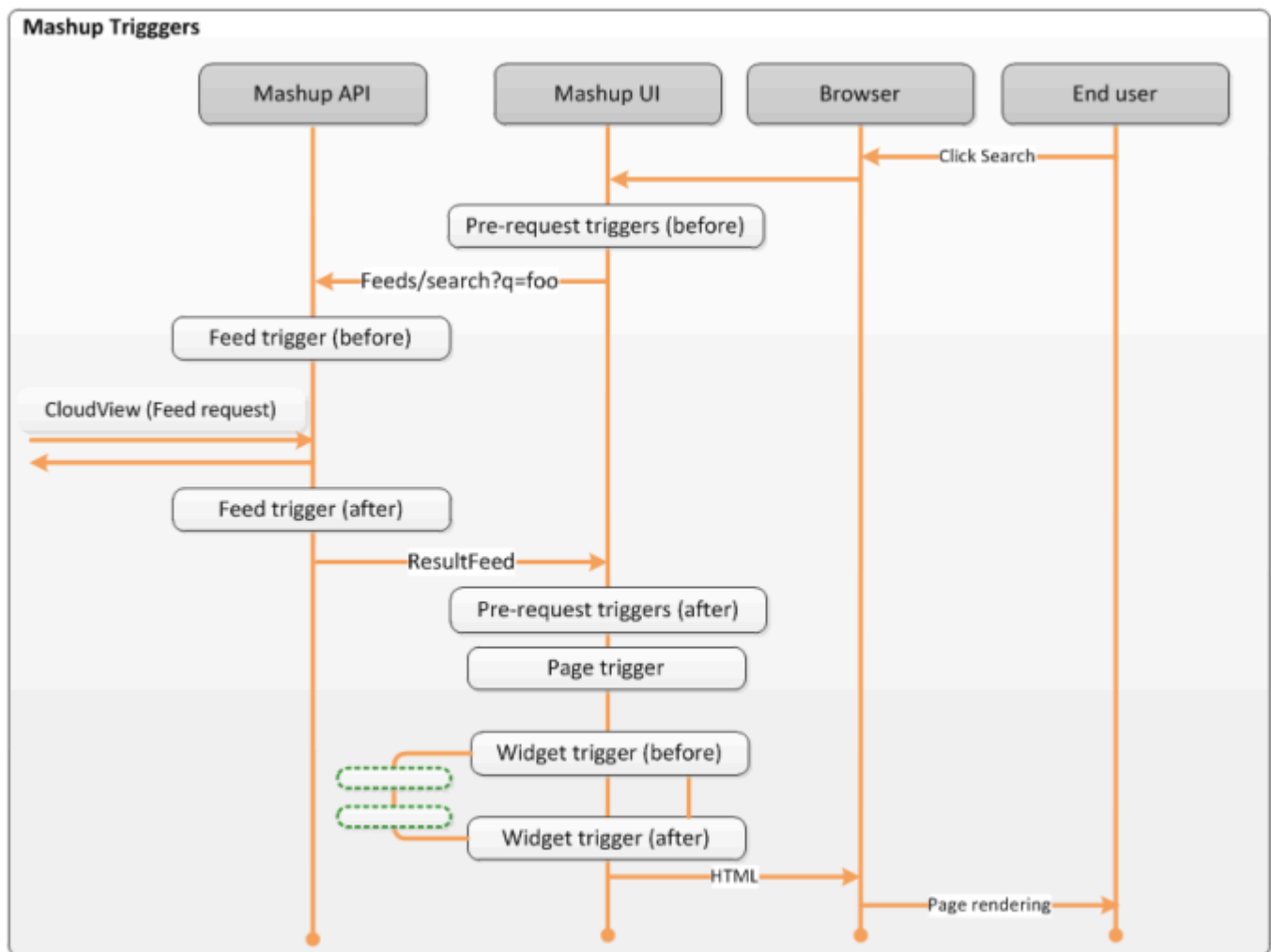
Design Triggers

Design triggers are called by the Mashup UI and include:

- Pre-request triggers which can be used to decide whether the user should be redirected to another page or not. The back end is not yet called, so no special load is triggered on the system. For example, redirect the user to the page called `/imagesearch` if the query starts with `image(s)`.
- Page and widget triggers which have the possibility to alter the behavior of the display by making decisions to draw things or even change the configuration (each user request gets a fresh copy of the original configuration, so any changes at query time are safe). For example: decide whether a widget should be displayed or not.
- Application triggers which are executed on all application pages.

Execution Flow

The Mashup trigger sequence is as follows:



Mashup UI interface

Pre-request triggers

```

@CustomComponent(displayName = "Custom redirect (PreRequestTrigger)")
public class CustomRedirectNew implements PreRequestTrigger {
    @Override
    public boolean beforeQuery(Map<String, Object> modelMap, AccessRequest accessRequest,
        HttpServletRequest request, HttpServletResponse response, HttpSession session) throws ServletException {
        response.sendRedirect("http://exalead.com");
        return false;
    }
    @Override
    public boolean afterQuery(Map<String, Object> modelMap, Map<String, ResultFeed> resultFeeds,
        HttpServletRequest request, HttpServletResponse response, HttpSession session) throws ServletException {
        return true;
    }
}

```

Page & widget triggers

```
@CustomComponent(displayName = "Removes entry (Test)")
public class RemoveEntry implements MashupWidgetTrigger {
    @Override
    public boolean beforeRendering(DataWidgetWrapper dww, TriggerContext triggerContext) {
        for (String feedName : dww.getResultFeeds().keySet()) {
            java.util.Iterator<Entry> it = dww.getResultFeeds().get(feedName).iterator();

            while (it.hasNext()) {
                Entry entry = it.next();
                if (entry.getMeta("source").equals("hotel")) {
                    it.remove();
                }
            }
        }
        return true; // return false to hide the widget
    }
    @Override
    public void afterRendering(DataWidgetWrapper dww, TriggerContext triggerContext) {
    }
}
```

Application triggers

The code is exactly the same as the one shown above. The only difference is the way you call them in Mashup Builder, that is to say, either on a page or on an application.

Implementing a page trigger

```
public class MyPageTrigger implements MashupTrigger<MashupPage> {
    /* Implementation */
}
```

Implementing a widget trigger

```
public class MyWidgetTrigger implements MashupTrigger<Widget> {
    /* Implementation */
}
```

Mashup API interface

Feed triggers

```
/**
 * Feed Trigger interface: A Feed Trigger is an entry point to alter the
 * behavior of a Feed. It is called before and after the feed execution,
```

```

* allowing for query manipulation, context modification and result
* manipulation. IMPORTANT: Feed Triggers are shared across the request and thus
* MUST be both stateless and threadsafe. If you need to keep states
* between the beforeQuery call and the afterQuery call, you can store the
* variables in the QueryContext:
* beforeQuery: context.setVariable("myComputedScore", 42);
* afterQuery: context.getVariable("myComputedScore");
*/
public interface FeedTrigger {
    /**
     * Before Feed Execution call. Return Result.STOP to skip the evaluation of
     * the feed, Result.CONTINUE for the normal behavior.
     *
     * @param feed
     * @param context
     * @throws AccessException
     */
    public Result beforeQuery(Feed feed, QueryContext context) throws AccessException;
    /**
     * After Feed Execution call. Return Result.CONTINUE for the normal behavior,
     * or Result.EVAL_AGAIN to re-execute the feed after overriding a few
     * parameters.
     *
     * @param feed
     * @param context
     * @throws AccessException
     */
    public Result afterQuery(Feed feed, QueryContext context, ResultFeed resultFeed) throws
    AccessException;
    public enum Result {
        CONTINUE,
        STOP,
        EVAL_AGAIN
    }
    /**
     * A Trigger that implements BeforeSubfeedAware will receive an additional event after
    method,
     * but before the sub feeds processing.
     */
    public static interface BeforeSubfeedAware {
        public Result beforeSubfeeds(Feed feed, QueryContext context, ResultFeed resultFeed)
    AccessException;
    }
}

```

Feed trigger example

The following Java snippet is an example of basic query rewriting, replacing all occurrences of the word "rain" by the word "sun".

```
package my.test;
import com.exalead.access.feedapi.AccessException;
import com.exalead.access.feedapi.Feed;
import com.exalead.access.feedapi.FeedTrigger;
import com.exalead.access.feedapi.QueryContext;
import com.exalead.access.feedapi.ResultFeed;
public class SampleTrigger implements FeedTrigger {
    public Result beforeQuery(Feed feed, QueryContext context) throws AccessException {
        // Fetches the original q parameter
        String originalQuery = feed.getEvaluatedParameter(context, "q");
        System.out.println("Original query: " + originalQuery);
        // Computes the new query
        String newQuery = originalQuery.replace("rain", "sun");
        System.out.println("New query:" + newQuery);
        // Forces the "q" parameter to the new query
        feed.overrideParameter(context, "q", newQuery);
        return Result.CONTINUE;
    }
    public Result afterQuery(Feed feed, QueryContext context, ResultFeed resultFeed) throws AccessException {
        return Result.CONTINUE;
    }
}
// To get your custom code running into your CloudView 360 instance, compile it into
// a .jar file and drop it in the javabin directory of your CloudView 360 kit.
// Finally, to plug your Trigger in the Access.xml configuration file,
// just add a <Trigger> tag to the targeted feed:
<Feed id="bi" enable="true" embed="true" className="com.exalead.access.basefeeds.BusinessFeed">
    <Trigger className="my.test.SampleTrigger" />
    <Parameters>
        <Parameter name="searchAPIVersion">5.0</Parameter>
        <Parameter name="searchapi">{access-api.searchapi.url}/search</Parameter>
        <Parameter name="q">${page.params["q"]}</Parameter>
        <Parameter name="type">all</Parameter>
        <Parameter name="page">1</Parameter>
        <Parameter name="defaultQuery">all</Parameter>
        <Parameter name="per_page">10</Parameter>
    </Parameters>
</Feed>
```

Creating Controllers

You can create custom Spring Controllers to add server side logic to the application.

[Create and package a controller](#)

[Reference JSP in a controller](#)

Create and package a controller

Create a controller

You must respect the following requirements:

- Your class must be in a subpackage of package:
`com.exalead.cv360.searchui.view.widgets.controller`. For example:
`com.exalead.cv360.searchui.view.widgets.controller.hello`
- There must be an `@CustomComponent` annotation.
- The Spring Controller is a class annotated with the `@Controller` annotation.

Example of an "Hello World" controller:

```
package com.exalead.cv360.searchui.view.widgets.controller;
@CustomComponent(displayName="Hello World")
@Controller
public class HelloWorldController {
    @RequestMapping(value = "/helloWorld", method = { RequestMethod.GET })
    public void helloWorld(HttpServletRequest request, HttpServletResponse response) throws IOException {
        response.getWriter().print("Hello World!!");
    }
}
```

This Controller can then be reached at: `http://<HOSTNAME>:<BASEPORT>/<context-name>/helloWorld`

Package the controller in a custom Plugin

The Exalead CloudView Eclipse plugin must be installed.

1. From the **Package Explorer** panel, right-click on the Controller class and select **CloudView Mashup > Export to file**.
2. In the **Mashup > Export plugin** window, specify the **name** and **destination** of the plugin and click **Finish**.
3. The plugin is packaged as .zip file and can be uploaded in Mashup Builder, in **Application > Manage components > Plugins**.

4. Restart the search server.
 - a. Go to **Application > Developer area**.
 - b. Click **Reload components**
 - c. Select the **Restart search server processes** option.
5. Go to **Application > Manage components > Controllers**.

The new controller should be added to the list of controllers.

Package your controller manually in a jar

You can also package your controller manually in a jar but we strongly recommend to package your controller as a plugin.

1. Right-click the controller class and select **Export...**
2. In the **Select** window, expand the **Java** node and select **JAR file**.
3. Click **Next** and in the **Jar File Specification** window:
 - a. From **Select the resources to export**, select the class file to export.
 - b. From **Select the export destination**, select where you want to export the JAR file.
 - c. Select the **Add directory entries** option.
 - d. Click **Finish**.
4. Copy the exported jar in your `<DATADIR>/webapps/360-mashup-ui/WEB-INF/lib/` folder.
5. Open Mashup Builder and restart the search server.
 - a. Go to **Application > Developer area**.
 - b. Click **Reload components**.
 - c. Select the **Restart search server processes** option.
6. In your browser, open the mashup-ui page, for example, `http://myhost:10000/mashup-ui/helloWorld`.

The page should display `Hello World!!`

Reference JSP in a controller

Follow the procedure below to reference JSP in a controller

1. Copy your JSP somewhere within the following directory: `<DATADIR>/webapps/360-mashup-ui/WEB-INF/jsp/`
2. Edit the following file: `<DATADIR>/webapps/360-mashup-ui/WEB-INF/tiles-def.xml`
3. Add a line within `<tiles-definitions>`, for example: `<definition name="mytemplate" template="/WEB-INF/jsp/path/of/the/jsp/page.jsp" />`

4. Create and package a custom controller and deploy it as a plugin, for example:

```
package com.exalead.cv360.searchui.view.widgets.controller;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
@Controller
public class TestController {
    @RequestMapping(value = "/test", method = { RequestMethod.GET, RequestMethod.P
    public String test(HttpServletRequest request, HttpServletResponse response) {
        return "mytemplate";
    }
}
```

5. Restart the search server.

Your page should be accessible at the following URL: `http://<HOSTNAME>:<PORT>/mashup-ui/test` Where:

- `mashup-ui` is the context path of the application deployed by jetty.
- `/test` is the path specified by the `@RequestMapping` annotation.

Note: The default controller renders pages as `/page/pageName`, for example, `http://<HOSTNAME>:<PORT>/mashup-ui/page/search` and is written as follows:

```
@RequestMapping(value = "/page/{pageName}", method = RequestMethod.GET)
public String renderPage(HttpServletRequest request, HttpServletResponse respo
("pageName") String pageName) throws Exception {
}
```

Managing URL Rewriting

Defining pretty URL for accessing and configuring a Mashup UI page is a very common use case for business-focused Apps and eCommerce.

For example, we want to rewrite the URL: `http://www.mysite.com/mashup-ui/page/search?type=clothes&products=shirts` to map it to a specific product refinement on the /products page and get a cleaner URL like: `http://www.mysite.com/products/clothes/shirt`

To handle URL rewriting, we use Url Rewrite Filter version 3.2. For more information, see the documentation at: <http://urlrewritefilter.googlecode.com/svn/trunk/src/doc/manual/3.2/index.html#filterparams>

Important: The rewriting rules you define take precedence on the default Mashup UI behavior and may lead to errors. For example, some widgets allow exporting data through the service `http://www.mysite.com/export`. If your rule rewrites all pages to `http://www.mysite.com/<PAGENAME>`, requests for `/export` will redirect to a page called `export` instead of the export service. To prevent this, you should set exception rules to redirect requests to the correct service.

Enable URL rewriting

1. Go to `<DATADIR>/webapps/360-mashup-ui/WEB-INF/`
2. Edit the `WEB-INF/web.xml` file.
3. Uncomment the `UrlRewriteFilter` filter node.

```
<filter>
  <filter-name>UrlRewriteFilter</filter-name>
  <filter-class>org.tuckey.web.filters.urlrewrite.UrlRewriteFilter</filter-class>
  <init-param>
    <param-name>logLevel</param-name>
    <param-value>log4j</param-value>
  </init-param>
  <init-param>
    <param-name>confReloadCheckInterval</param-name>
    <param-value>5</param-value>
  </init-param>
  <init-param>
    <param-name>statusEnabled</param-name>
    <param-value>>false</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>UrlRewriteFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

4. Save and close the file.

Configure URL rewriting

1. In the `<DATADIR>/webapps/360-mashup-ui/WEB-INF/` directory, edit the `WEB-INF/urlrewrite.xml` file.
2. Uncomment the content of the `urlrewrite` node, and add/edit the rewriting rules as needed. For our example of `/products/clothes/shirt` we would have the following configuration.

```
<urlrewrite>
  <!-- Services have the highest priority -->
  <rule enabled="true">
    <name>Services</name>
```



```

    <from>^(resources|fetch|alerting|export|utils|login|errors|lang|storage|logou
|staging-builder|testProduction)(.*)?</from>
    <to last="true">/$1$2</to>
</rule>
<!-- redirects /page/index to / -->
<rule enabled="true">
    <name>RedirectIndex</name>
    <from>^/page/index$</from>
    <to last="true" type="redirect">{%context-path}/?{%query-string}</to>
</rule>
<!-- redirects /page/pageName to /pageName -->
<rule enabled="true">
    <name>RedirectPages</name>
    <from>^/page/(\w+)$</from>
    <to last="true" type="redirect">{%context-path}/$1?{%query-string}</to>
</rule>
<!-- / -->
<rule enabled="true">
    <name>Index</name>
    <from>^/$</from>
    <to last="true">/page/index</to>
</rule>
<!-- /pageName -->
<rule enabled="true">
    <name>Pages</name>
    <from>^/(\w+)$</from>
    <to last="true">/page/$1</to>
</rule>
<!-- /products/clothes/shirt -->
<rule enabled="true">
    <name>Search</name>
    <from>^/(\w+)/(\w+)/(\w+)</from>
    <to>/page/search?type=$1&$2=$3</to>
</rule>
</urlrewrite>

```

Note that:

- First, all the services that can be called by the Mashup UI (export, alerting, storage, etc.) are processed. We make sure they will not be impacted by URL rewriting to avoid errors. We can then add filter rules for the URLs of the application pages.
- /page/ is hidden from the URL
- The / redirects to the index page

3. Save and close the file.

Implementing custom layout or templates as plugins

You may want to implement custom layouts or templates as plugins and use them in the Mashup UI.

1. Create a plugin with a `cvplugin.properties` file looking like

```
cloudview.version=V6R2016x.R2.80158
plugin.author=you
plugin.components.layouts=NAME_OF_YOUR_FOLDER
plugin.copyright=Dassault Systèmes
plugin.description=blabla
plugin.name=pouet
plugin.type=mashup
plugin.version=420.0.0-SNAPSHOT
```

Note: For more information about CVPlugin packaging, see "Packaging Custom Components as Plugins" in the Exalead CloudView Programmer's Guide.

2. In your folder, add `layout.jsp` file containing your custom layout.
3. In Mashup Builder:
 - a. Click the **Edit page settings** at the right of the screen icon at the right of the top bar.
 - b. In **General > Custom layout** enter the name of the plugin containing your custom layout.
 - c. Select the **Preview** to check your configuration changes.
 - d. Click **Apply**.

Using the Mashup API

The Mashup API, also known as the Access API, provides a public HTTP interface to access the commands defined in CloudView's Search API configuration.

[About the Mashup API](#)

[Choosing between the Mashup API and the Search API](#)

[Using the Mashup API Java client](#)

[Using the HTTP Mashup API](#)

[Using the Atom Output Format](#)

[Creating Parallel Requests](#)

[Configuring Hits Enrichment](#)

About the Mashup API

The Mashup API, also known as the Access API, provides a public HTTP interface to access the commands defined in Exalead CloudView Search API configuration.

The Mashup API provides a standardized way to search and retrieve hits from heterogeneous sources, called feeds. Feeds can be queried independently to retrieve different types of information, or nested to enrich hits from a previous feed.

By default the Mashup API retrieves hits using parallel requests. If you depend on the results of a given feed to alter another feed, you can set this feed as **Synchronized** in the **Feed settings**. For more information, see "Synchronizing feeds on a page" in the Exalead CloudView Mashup Builder User's Guide.

Choosing between the Mashup API and the Search API

You can use either the Mashup API or the Search API for search in your applications.

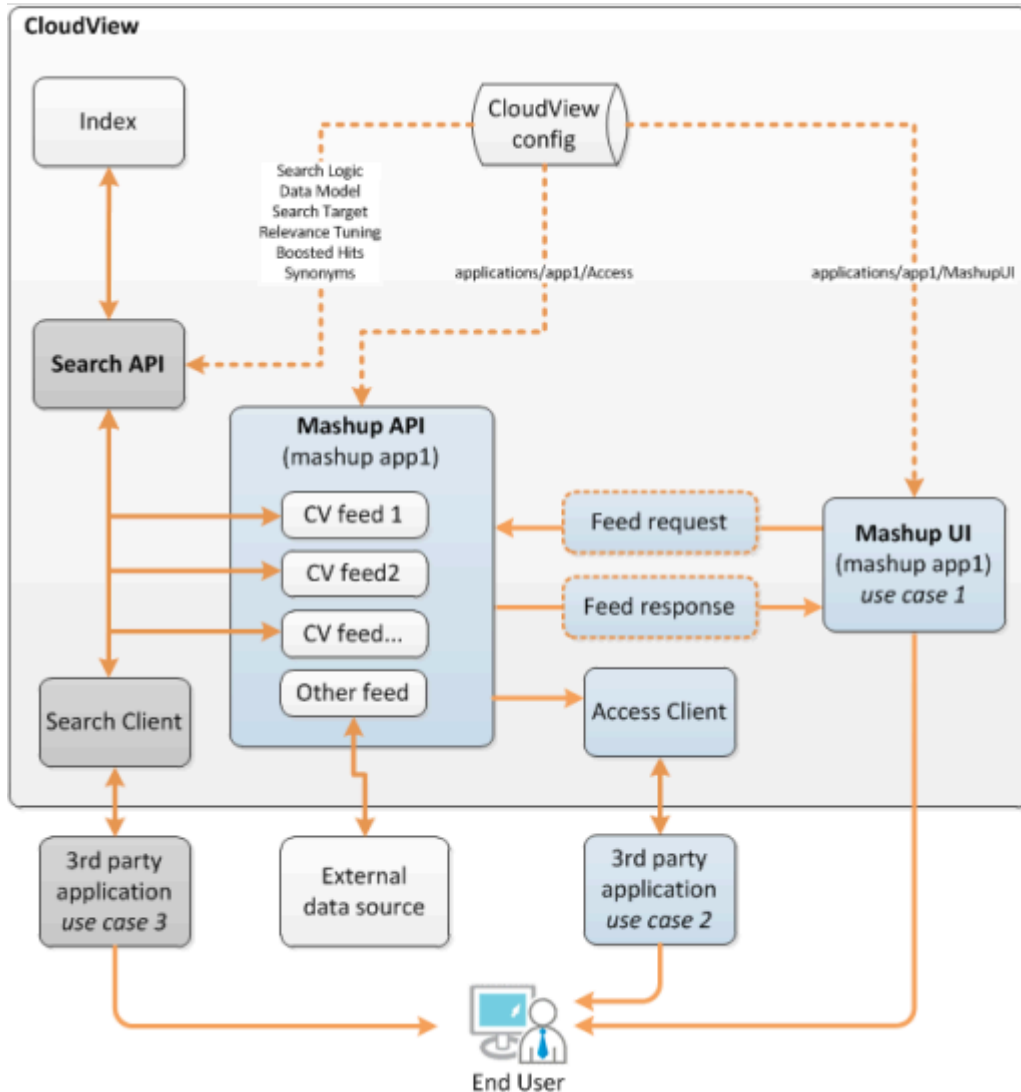
What you can do

Exalead CloudView gives you the following search possibilities:

- Use case 1: You can use the Mashup API with a Mashup UI application, for example, `app1`.
- Use case 2: You can use the Mashup API with a third-party application, if you don't want to use a Mashup UI application for the front-end.

- Use case 3: You can use the Search API with a third-party application, if you don't want to use the Mashup API and address the Search API directly.

These search possibilities are illustrated in the following diagram.



How to choose between the two APIs

Use the Mashup API if you want to:

- Federate several Exalead CloudView searches.
- Federate with external data sources (supported by Mashup feeds).
- Enable security on your application pages using the Mashup Builder Security Providers.
- Build advanced front-end applications easily with the Mashup Builder.
- Use the Mashup Expression Language (MEL) which allows you to:
 - construct text that contains dynamic content from your feeds,

- perform common operations that would usually require editing JSP files.

Use the Search API if you want to:

- Fetch a large number of results (>1000).
- Fetch a large number of facets or facet values.
- Have specific tools to formulate complex queries (SearchQuery accessors, helpers, etc.).
- Handle security entirely on your own.

Using the Mashup API Java client

You can access the Mashup API Java client in `<INSTALLDIR>/sdk/java-clients`

You interact with the Mashup API using the `AccessClient` java library (`access-core.jar`).

The `AccessClient` library is a simple wrapper around the HTTP/Atom protocol. It delivers results in two different formats:

- **ResultFeed Object** - using the `getResultFeed(AccessRequest request)` method
- **Atom XML as an InputStream** - using the `getResultStream(AccessRequest request)` method

Note: The `AccessClient` object is thread safe. You should use only one instance of the `AccessClient` for your whole program. It will ensure that the HTTP connection is alive, to maintain request queueing without establishing useless connections to the service.

AccessClient example

```
ServerInfos serverInfos = new ServerInfos("http://host:10010/access");
AccessClient client = AccessClientFactory.createAccessClient(new ServerInfos[] {
    serverInfos });
AccessRequest request = new AccessRequest();
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

Note: `SERVICE_URI` is the Mashup API endpoint where the `AccessClient` should make the requests.

Where is the Mashup API endpoint?

For the default application: `http://<HOSTNAME>:<BASEPORT+10>/access`

For every other applications: `http://<HOSTNAME>:<BASEPORT+10>/access.<applicationId>`

How to configure a proxy

```
ServerInfos serverInfos = new ServerInfos("http://HOST:PORT/access");
serverInfos.setProxyHost("host");
serverInfos.setProxyPort(8080);
serverInfos.setProxyLogin("login");
serverInfos.setProxyPassword("password");
AccessClient accessClient = AccessClientFactory.createAccessClient(new ServerInfos[]
{serverInfos });
AccessRequest request = new AccessRequest();
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

How to send security tokens to a secured Search API

Requirement: To send security tokens, you must first enable security on your Mashup UI pages. For more information, see "Adding Security to Your Application" in the Exalead CloudView Mashup Builder User's Guide.

You can then list security tokens as follows:

```
ServerInfos serverInfos = new ServerInfos("http://HOST:PORT/access");
AccessClient client = AccessClientFactory.createAccessClient(new ServerInfos[]
{serverInfos });
List<String> tokens = new ArrayList<String>(); tokens.add("Everybody");
AccessRequest request = new AccessRequest();
request.addParameters(AccessParameter.SECURITY, tokens);
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

How to configure failover

To enable failover, you must specify the `isAlive` path. Note that:

- The higher the power the higher the request priority.
- If they have the same power, requests will be uniformly distributed between the hosts.

```
ServerInfos host1 = new ServerInfos("http://HOST1:PORT/access");
host1.setIsAlivePath("/admin/isAlive");
host1.setPower(1);
ServerInfos host2 = new ServerInfos("http://HOST2:PORT/access");
host2.setIsAlivePath("/admin/isAlive");
host2.setPower(10);
AccessClient accessClient = AccessClientFactory.createAccessClient(new ServerInfos[]
{host1, host2 });
```

```
AccessRequest request = new AccessRequest();
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

How to configure the max number of concurrent connections to the distant host

```
ServerInfos serverInfos = new ServerInfos("http://HOST:PORT/access");
Properties options = new Properties();
options.put("http.max_number_of_connections_per_server", 10);
AccessClient accessClient = AccessClientFactory.createAccessClient(new ServerInfos[] {
serverInfos }, options);
AccessRequest request = new AccessRequest();
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

How to configure the stale connection check

See Apache documentation: http://hc.apache.org/httpclient-3.x/performance.html#Stale_connection_check

```
ServerInfos serverInfos = new ServerInfos("http://HOST:PORT/access");
Properties options = new Properties();
options.put("http.commons-httpclient.stale_checking_enabled", "true");
AccessClient accessClient = AccessClientFactory.createAccessClient(new ServerInfos[] {
serverInfos }, options);
AccessRequest request = new AccessRequest();
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

How to configure a socket read timeout

The following code snippet shows how to configure a socket read timeout to 5000 milliseconds (5 seconds). If set to 0, timeout is disabled.

```
ServerInfos serverInfos = new ServerInfos("http://HOST:PORT/access");
Properties options = new Properties();
options.put("http.socket.timeout", 5000); // 5 seconds timeout
AccessClient accessClient = AccessClientFactory.createAccessClient(new ServerInfos[] {
serverInfos }, options);
AccessRequest request = new AccessRequest();
request.setPage("search");
request.addParameter("q", "disney");
ResultFeed result = client.getResultFeed(request);
```

Using the HTTP Mashup API

The Mashup API can also be queried directly through HTTP GET at the following URL:

```
http://<HOSTNAME>:<BASEPORT+10>/access/feeds/<page name>
```

Page parameters can be given just as normal HTTP parameters. Subfeed parameters have to be prefixed by the subfeed id.

The following pages and subfeeds are available:

- page: search (parameters: q)
 - subfeed: bi (id: bi, parameters: page, per_page, etc.)
- page: details (parameters: id)
 - subfeed: bi (id: bi, parameters: page, per_page, etc.)
 - subfeed: flickr (id: flickr, parameters: tag, latitude, longitude, etc.)

Note: To output a JSON string instead of the default XML output format, add `&outputFormat=json` at the end of you query.

A few sample requests are:

- `http://<HOSTNAME>:<BASEPORT+10>/access/feeds/<page name>/?q=disney`
- Page parameter and pagination:


```
http://<HOSTNAME>:<BASEPORT+10>/access/feeds/<page name>/?
q=disney&bi.page=2 http://<HOSTNAME>:<BASEPORT+10>/access/feeds/<page
name>/?detailsid=42
```
- Page parameter and image tags:


```
http://<HOSTNAME>:<BASEPORT+10>/access/feeds/<page name>/
detailsid=42&flickr.tag=disney
```

Note: Adding a `<class name>` parameter after `<page name>` allows you to target more specific information, and thus avoid adding query restriction parameters.

For details of the output, see [Using the Atom Output Format](#).

Using the Atom Output Format

This section describes the `MashupFeed` response elements. Collectively these formats may be referred to as `MashupFeed 1.0` or simply `MashupFeed`. The `MashupFeed` Schema makes use of the `OpenSearch v1.1` extension to the Atom format.

The following elements are described below:

namespace

The XML Namespace URI for the XML data formats described in this specification is:

```
http://schemas.exalead.com/access/1.0
```

Note that this namespace is subject to change in the `MashupFeed 1.1` version to:

```
http://schemas.exalead.com/mashupfeed/1.1
```

All the examples below will assume that the namespace is used as the "exa" prefix.

link element

In the `MashupFeed` Schema, nested data feeds described with the `<link>` element with an "application/access+xml" type may be embedded right into the `<link>` element content.

```
<link type="application/access+xml" rel="bi" href="http://host:9510/access/feeds/
search/cT0lMjNhbGw/bi">
<feed>
<id>bi</id>
<generator>com.exalead.access.basefeeds.BusinessItemFeed</generator>
<entry>
...
</entry>
</feed>
</link>
```

entry element

In the `MashupFeed` schema, an entry has the following properties:

- one `id` which is optional (required by the standard specification)
- a `title` which is optional (required by the standard specification)
- many `exa:meta` which are optional

```
<entry>
  <id>MDA1QTAwMDAwMDBRakRwSUFL</id>
  <link rel="self" type="application/access+xml" href="http://localhost:9510/acces
```

```
cT0lMjNhbGw/bi/MDA1QTAwMDAwMDBRakRwSUFL"/>
<exa:meta name="id" displayName="id">005A0000000QjDpIAK</exa:meta>
<exa:meta name="username" displayName="username">doe@mycompany.com</exa:meta>
<exa:meta name="email" displayName="email">john.doer@mycompany.com</exa:meta>
<exa:meta name="name" displayName="name">John Doe</exa:meta>
</entry>
```

meta element

The `MashupFeed` meta element describes a single information element as represented in the EXALEAD index. Its location is Standard `<entry>` element. The meta element syntax is defined by:

- a required `name` attribute describing the meta name (string)
- a value in the content part of the tag (string, may be empty)
- an optional `displayName` attribute for internationalized / nicely formatted names

```
<exa:meta name="firstname">John</exa:meta>
<exa:meta name="lastname" displayName="family name">Doe</exa:meta>
```

feed element

The `feed` element may contain many facet elements that describe the different axes available for faceted search. Its location is Standard `<feed>` element.

facet element

The `facet` element describes the root of a faceted search axe and can contain several category elements.

Its location is Standard `<facets>` element. The facet element syntax is defined by:

- a required `name` attribute for the facet full name (string)
- a required `description` attribute for the facet display name (string)
- many category elements which are optional

```
<exa:facet name="Top/user/name" description="user name">
  <exa:category path="Top/user/name/john doe" description="John Doe" count="1" sta
</exa:facet>
```

category element

The `category` element is a leaf of the facet tree. It may contain many category elements that are sub categories of the current category.

Its location is Standard `<facet>` element. The category element syntax is defined by:

- a required `path` attribute: category full path in the facet tree (string)
- a required `description` attribute: category display name (string)
- a required `count` attribute: number of hits of this category for the given query (long)
- a required `state` attribute: state of the category (enum: `DISPLAYED|EXCLUDED|REFINED`)
- optionally many `category` sub-elements

```
<exa:category path="Top/case/type/electrical" description="Electrical" count="9" s
```

Creating Parallel Requests

Creating parallel requests is useful if you have different indexes or if you want to search for specific content in the same index using different queries.

The following use case describes how to proceed if you want your application to fetch hits from two sources using the same query basis:

- One of these source is a filesystem crawl retrieving data through a **Files** connector.

In our example, the set of documents located under the `<INSTALLDIR>/docs/` directory.

- The other one is a web crawl retrieving data through an **HTTP** connector.

In our example, the Exalead website (<http://www.exalead.com/software/>).

Our application is configured in the Mashup Builder to use two **CloudView Search** feeds that will select the 2 most relevant hits for the request on the search results page `${page.params["q"]}` to focus on the two specific data sources.

By making a query on the Mashup UI, we see below that the search page displays results coming from the two data sources:

- one from an Exalead website URL
- another from the filesystem

Example of parallel requests on the Mashup UI

<http://www.exalead.com/software/products/cloudview/360/>

CloudView 360 Package v4

[Download](#)
[Preview](#)

These modules include the **Mashup Builder**, the Semantic Factory, the Business Console and Trusted Queries ... The **Mashup Builder** is a Drag'n'Drop interface for rapidly prototyping and deploying SBAs ... Using the **Mashup Builder**, you can prototype a robust, full-featured [\[Read more\]](#)



File name	Exalead-Cloudview-360-for-SBA-in-the-Enterprise-EN.pdf	File size	1843267
publicurl	http://www.exalead.com/software/common/pdfs/products/cloudview/Exalead-Cloudview-360-for-SBA-in-the-Enterprise-EN.pdf		

<http://www.exalead.com/software/common/pdfs/products/cloudview/Exalead-Cloudview-360-for-SBA-in-the-Enterprise-EN.pdf>

Mashup Builder User's Guide

[Download](#)
[Preview](#)

Mashup Builder allows you to build more than a simple search front end ... **Mashup Builder** allows you to create and customize your own search applications through a drag and drop interface ... **Mashup Builder** gives you the possibility of customizing the layout of your search applications very precisely



File path	/data/ngproduct/vquesnia/cloudview-dev_trunk.dev.37878-linux-x64/docs/MashupBuilder_UserGuide_EN_V6.pdf	File name	MashupBuilder_UserGuide_EN_V6.pdf
File size	1296640		

id: /%2Fdata%2Fngproduct%2Fvquesnia%2Fcloudview-dev_trunk.dev.37878-linux-x64%2Fdocs/MashupBuilder_UserGuide_EN_V6.pdf

Understanding the **Mashup Builder** Interface

[Download](#)
[Preview](#)

XML configuration example

The following XML Configuration represents how the parallel requests presented above are coded.

```
<Feed id="page" enable="true" embed="true" className="com.exalead.access.basefeeds.Pa
<Parameters>
<Parameter name="q">${page.params["q"]}</Parameter>
```

```

</Parameters>
<SubFeeds>
<Feed id="crawls" enable="true" embed="true" className="com.exalead.access.basefeeds.C
<Parameters>
<Parameter name="searchAPIVersion">V6R2015x</Parameter>
<Parameter name="searchapi">{access-api.searchapi.url}/search</Parameter>
<Parameter name="q">source:web ${page.params["q"]}</Parameter>
<Parameter name="defaultQuery">#all</Parameter>
<Parameter name="page">1</Parameter>
<Parameter name="per_page">3</Parameter>
</Parameters>
<Properties>
<Property kind="TITLE" name="TITLE">${entry.metas["title"]}</Property>
</Properties>
</Feed>
<Feed id="files" enable="true" embed="true" className="com.exalead.access.basefeeds.C
<Parameters>
<Parameter name="searchAPIVersion">V6R2015x</Parameter>
<Parameter name="searchapi">{access-api.searchapi.url}/search</Parameter>
<Parameter name="q">source:fs ${page.params["q"]}</Parameter>
<Parameter name="defaultQuery">#all</Parameter>
<Parameter name="page">1</Parameter>
<Parameter name="per_page">3</Parameter>
</Parameters>
<Properties>
<Property kind="TITLE" name="TITLE">${entry.metas["filename"]}</Property>
</Properties>
</Feed>
</SubFeeds>
</Feed>

```

Configuring Hits Enrichment


Nesting feeds in one another gives you the possibility of enriching the hits of a query with other sources to bring related content to the user. This can be achieved because nested feeds can use parent metas.


In the following use case, our application starts by retrieving information about TV series using a CloudView **Search** feed. It then tries to enrich each hit with images retrieved by a **Google Search** feed using the TV series title.

Example of hit enrichment in Mashup UI

Le Trône de fer : **Game of Thrones** > Saison 2 - AlloCiné
Download
Preview

Accueil > Séries TV > Le Trône de fer : **Game of Thrones** > Saison 2 ... Le Trône de fer : **Game of Thrones** > Saison 2 ... Vous avez vu la saison 2 de Le Trône de fer : **Game of Thrones**



File size	17960	publicurl	http://www.allocine.fr/series/ficheserie-7157/saison-19981/
Source	series	Data model class	document
Language	 French	Last modification	2012 > 01 > 09

<http://www.allocine.fr/series/ficheserie-7157/saison-19981/>



Game of Thrones saison 2: une première bande-annonce | TVQC

La deuxième **saison** de **Game of**

imageId: ANd9GcRE4Gf-aAK6Sg7jR20KYlj93nTuWVnJzQAp6hIk4N_7ZZZYcXz_icbOmd

titleNoFormatting: Game of Thrones saison 2: une première bande-annonce | TVQC

unescapeUrl: <http://www.tvqc.com/wp-content/uploads/2011/12/peter-dinklage-game-of-thrones.jpg>

tbUrl: http://t1.gstatic.com/images?q=tbn:ANd9GcRE4Gf-aAK6Sg7jR20KYlj93nTuWVnJzQAp6hIk4N_7ZZZYcXz_icbOmd

tbWidth: 131

tbHeight: 84

width: 520

height: 333

content: La deuxième **saison** de **Game of**

originalContextUrl: <http://www.tvqc.com/2011/12/game-of-thrones-saison-2-une-premiere-bande-annonce/>

id: ANd9GcRE4Gf-aAK6Sg7jR20KYlj93nTuWVnJzQAp6hIk4N_7ZZZYcXz_icbOmd

The following XML Configuration represents how the hit enrichment presented above is coded.

```
<Feed id="test" enable="true" embed="true" className="com.exalead.access.basefeeds.Pa
<Parameters>
<Parameter name="q">${page.params["q"]}</Parameter>
</Parameters>
<SubFeeds>
<Feed id="movies" enable="true" embed="true" className="com.exalead.access.basefeeds.
<Parameters>
<Parameter name="searchAPIVersion">V6R2015x</Parameter>
<Parameter name="searchapi">{access-api.searchapi.url}/search</Parameter>
<Parameter name="q">${page.params["q"]}</Parameter>
```

```
<Parameter name="defaultQuery">#all</Parameter>
<Parameter name="page">1</Parameter>
<Parameter name="per_page">10</Parameter>
</Parameters>
<Properties>
<Property kind="TITLE" name="TITLE">${entry.metas["title"]}</Property>
</Properties>
<SubFeeds>
<Feed id="pictures" enable="true" embed="true" className="com.exalead.access.basefeed
<Parameters>
<Parameter name="searchapi">http://api.exalead.com/search/</Parameter>
<Parameter name="q">${entry.metas["title"]}</Parameter>
<Parameter name="defaultQuery">#all</Parameter>
<Parameter name="type">image</Parameter>
<Parameter name="apiKey">appkey</Parameter>
<Parameter name="page">1</Parameter>
<Parameter name="per_page">10</Parameter>
</Parameters>
</Feed>
</SubFeeds>
</Feed>
</SubFeeds>
</Feed>
```

About the Administration API

The Mashup Builder has its own administration APIs accessible through HTTP REST. You can use them very easily with the `360-admin-client.jar` Java library that abstracts everything to let you work with high level objects.

Administration Methods

Here is a list of the administration methods available:

Method	Description
<code>void saveConfiguration(Object configuration)</code>	Saves a configuration object to the staging area. You only need to provide a configuration object as configuration classes are naturally tied to a filename.
<code>Object getConfiguration(Class configurationClass)</code>	Retrieves the last version of a given configuration class, from the staging area. If there are no staged configurations available, the latest applied configuration is returned.
<code>void resetConfiguration(Class configurationClass)</code>	Removes the staged instance of the given configuration class to cancel changes.
<code>boolean hasStagedConfiguration(Class configurationClass)</code>	Returns <code>true</code> if an instance of this configuration class is stored in the staging area.
<code>ApplyStagingConfigurationAnswer applyStagingConfiguration()</code>	Applies all the staged configuration files.
<code>ApplyStagingConfigurationAnswer applyStagingConfiguration(List<Class> confClasses)</code>	Applies only the given staged configuration files.
<code>void checkStagingConfiguration()</code>	Runs a full configuration consistency check towards the staged configuration.
<code>void generateCloudViewConfiguration()</code>	Enforces the evaluation of the Data Ontology to generate and apply the underlying Mashup Builder configuration.

Default Services Locations

The following table lists the various service types and their locations. If you have multiple instances of the Mashup Builder, add the application name to the path.

Service Type	Deployment Role	Instance name	Location	Path
mashup-builder	Mashup Builder	360-administration-console	BASEPORT + 1 (gateway)	/mashup-builder
360-administration-service	Gateway	360-administration-service	BASEPORT + 11 (gateway)	/360-admin-service
360-mashup-ui	Mashup UI	360-mashup-ui-mu0	BASEPORT + 0 (searchservss0)	/mashup-ui/page/<PAGENAME>
access-api	Mashup API	access-api-ac0	BASEPORT + 10 (searchservss0)	/access/feeds/<PAGENAME>
trustedQueriesService	Trusted Queries API	trustedQueriesService tqs0	BASEPORT + 10 (searchservss0)	/360-trustedqueries
businessConsole	Business Console	businessConsole	BASEPORT + 1 (gateway)	/business-console

Mashup Builder services all answer to `/admin/isAlive` if they are alive. For example:

```
http://<HOSTNAME>:<BASEPORT+10>/access/admin/isAlive
```

Configuration System

Exalead CloudView modules are managed by a distinct configuration system, introducing a new staging area and a per module configuration management. The configuration works as follows:

- On **save** (through Mashup Builder / Administration API), files are staged in `<DATADIR>/gct/staging`

- On **apply** (through Mashup Builder / Administration API), all or specified files are copied in `<DATADIR>/config/360`. Configuration files in this directory are considered as applied. Then, the `applyConfiguration` call of the underlying Exalead CloudView instance is invoked.
- At runtime, services read their configuration from `<DATADIR>/gct/<last version>/master/360`
- When a new configuration is applied, services are notified and reload their configuration without having to restart.

If you want to modify the Mashup Builder configuration manually in the `<DATADIR>/config/360` directory, you need to invoke an apply configuration order on Exalead CloudView by running the command: `<DATADIR>/bin/cvcmd.sh applyConfig`