1. FishEye and Crucible Developer Documentation . . . 2
1.1 Development Overview . . . 4
1.2 Frequently Asked Questions . . . 5
1.3 Plugin Development . . . 5
1.4 REST API Guide . . . 6
1.4.1 Authenticating REST Requests . . . 7
1.4.2 FishEye Legacy Remote API . . . 8
1.5 Gadget Development . . . 15
1.6 Integration Tutorials . . . 15
1.6.1 Writing a REST Client in Perl . . . 15
1.6.2 Writing a REST Client in Python . . . 17
1.7 Plugin Tutorials . . . 19
1.7.1 Crucible SCM Plugin Tutorial . . . 20
1.7.1.1 Crucible ClearCase plugin . . . 32
1.7.2 Event Listener Plugin Module Tutorial . . . 34
1.7.3 FishEye Twitter Integration Plugin Tutorial . . . 35
1.7.4 Gadget Tutorial . . . 40
1.7.5 Gutter Renderer Plugin Tutorial . . . 41
1.7.6 Rendering a Velocity Template from Your Servlet . . . 42
1.7.7 REST Service Plugin Module Tutorial . . . 43
1.7.8 Storing Plugin Settings . . . 45
1.7.9 Using Logging From Your Plugin . . . 46
1.8 Authentication Plugins . . . 47
1.9 Plugin Module Types . . . 47
1.9.1 Downloadable Plugin Resources . . . 48
1.9.2 Event Listener Module Type . . . 49
1.9.3 Gadget Module Type . . . 49
1.9.4 Gutter Renderer Module Type . . . 51
1.9.5 REST Module Type . . . 51
1.9.6 SCM Module Type . . . 54
1.9.7 Servlet Module Type . . . 55
1.9.8 Spring Component Module Type . . . 56
1.9.9 Web Item Module Type . . . 58
1.9.9.1 Crucible Web Item Locations . . . 61
1.9.9.2 Discovering Web Items - Enable the FishEye & Crucible Development Mode Plugin . . . 64
1.9.9.3 FishEye Web Item Locations . . . 65
1.9.9.3.1 Web Item Helpers . . . 71
1.9.9.4 Page Decorators . . . 71
1.9.9.5 Web Item Conditions . . . 72
1.9.10 Web Resources . . . 74
1.10 Java API . . . 75
1.11 API Javadoc . . . 77

FishEye and Crucible Developer Documentation

Introduction

Welcome to the FishEye and Crucible developer documentation, a resource for plugin authors and programmers. This documentation will help you do the following:

- Develop software such as plugins and gadgets that extend or enhance FishEye and Crucible.
- Integrate FishEye and Crucible software with other systems.
- Learn about the architecture of FishEye and Crucible.

Go directly to the Advanced Developer Index if you're a seasoned developer looking for a quick technical reference.

Quick Links
FishEye User Documentation Crucible User Documentation FAQ API Javadoc REST API Guide

---

Tutorials

"Is there an easy-to-follow tutorial?"
Yes. Click here.

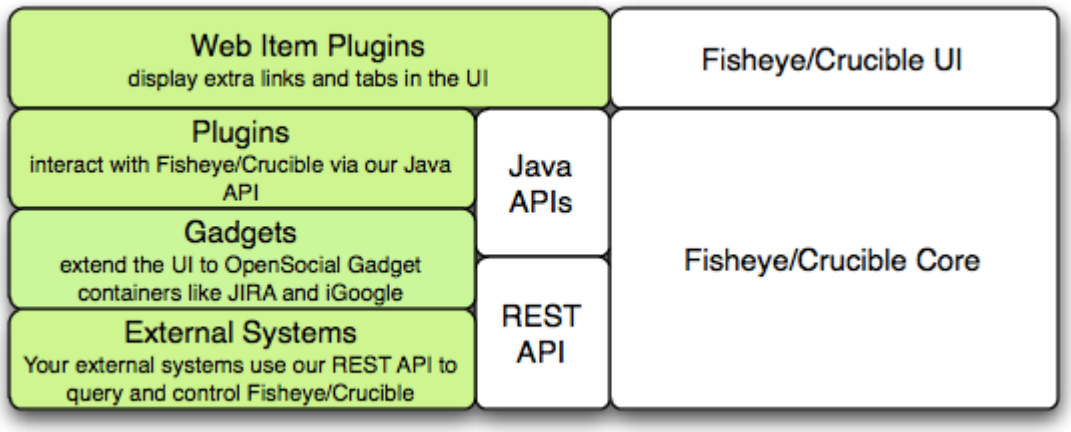
"How can I customize and extend Fisheye and Crucible?"
See the Overview to get started.

"Can I write Gadgets for FishEye or Crucible?
OpenSocial gadgets allow Crucible and Fisheye to be integrated into gadget containers such as iGoogle and the JIRA dashboard. Learn about Gadgets.

Tutorial Links
Plugins Integration

A map of FishEye and Crucible technology

This image shows how the various components of FishEye and Crucible interact with each other:



Key: In this diagram, Green items are your code. White items are FishEye/Crucible components.

FishEye & Crucible Map Links
Web Item Plugins Plugins Gadgets External Systems Java APIs REST APIs

Using Remote APIs

Remote APIs allow your applications to query and control Fisheye/Crucible using REST over HTTP. The documentation details the interfaces which are available to remote applications and gives examples of client code in some common languages.

See The Fisheye and Crucible Remote API Guide to get started.

Tutorials

- Writing a REST Client in Perl
- Writing a REST Client in Python

Remote API Links

Fisheye Services		Crucible Services
Repository		Projects
Changeset		Repository
Revisions		Review
Search		Search
		User

Developing Plugins

Plugins customise Fisheye/Crucible's user interface and behaviour.

Teach me how to write plugins for Fisheye/Crucible
These documents are specifically about plugins for Fisheye/Crucible. They describe:

- How to set up a plugin development environment.
- The plugin module types that Fisheye/Crucible supports. A plugin is a set of modules.
- The Java API which plugins use to access Fisheye/Crucible.
- Tutorials which illustrate the uses of these module types and APIs

Tutorials

- Crucible SCM Plugin Tutorial — Crucible SCM modules are plugins that make version control systems accessible to Crucible.
- Event Listener Plugin Module Tutorial — This is a brief tutorial which teaches you how to write a trivial event listener plugin.
- FishEye Twitter Integration Plugin Tutorial — The plugin created in this tutorial sends each of your commit messages to your Twitter account.
- Gadget Tutorial — This tutorial will teach you how to write a simple gadget which can display information from Crucible on the JIRA dashboard.
- Gutter Renderer Plugin Tutorial — This tutorial teaches you how to add extra information to annotated views of files, and to diffs
- Rendering a Velocity Template from Your Servlet — Velocity allows your Servlet Plugins to render HTML pages from simple templates.
- REST Service Plugin Module Tutorial — provide your own REST API
- Storing Plugin Settings — This tutorial demonstrates how to use SAL (Shared Access Layer) to let your plugin store its configuration settings.
- Using Logging From Your Plugin — This tutorial describes how to log messages from your plugin.

Plugin Links
Module Types • Event Listener • REST • Source Control System (SCM) • Servlet • Web Item • Gadget • Gutter Renderer • Spring Component • Authentication **Java APIs** • Fisheye and Crucible • Crucible SCM Utilities • SAL Overview, SAL Javadoc

Development Overview

Who Should Read This Page?

If you have an idea for making FishEye/Crucible more useful to you, then this page helps you decide how to do it.

You can translate your requirements into a set of software components you will need to write.

Look through the list of use cases below and find those which intersect with what you want to do.

What do you want to do?

Report on data in FishEye/Crucible

If you want to display the report in FishEye/Crucible, you can write a Servlet Plugin Module. If you are using an external reporting tool you can write a REST Client to extract the data. You can't access data in FishEye/Crucible via SQL – if you want to use an SQL based tool you'll need to extract data via a REST Client and insert it into your reporting database.

Useful Tutorials

- Writing a REST Client in Python

Pull data from FishEye/Crucible into my application

You should write a REST Client to do this. If you needed to use another protocol, you could write a Servlet Plugin Module and implement the protocol yourself. If you need to process the data before your application asks for (perhaps you just want to transfer a summary) you can write a REST module to provide a different interface to that provided by the standard FishEye/Crucible REST interface.

Useful Tutorials

- Writing a REST Client in Python
- REST Service Plugin Module Tutorial

Have FishEye/Crucible push data to my application

You can write an Event Listener Module which will be called when something happens (e.g. FishEye indexes a new change set, or a user completes a review). These can make an HTTP request to your application, update your database or open a socket to your application.

Useful Tutorials

- FishEye Twitter Integration Plugin Tutorial
- Event Listener Plugin Module Tutorial

Create reviews of files in a type of Source Control System which FishEye/Crucible doesn't support

You can write an SCM Module to allow Crucible to see the directory structure and history of another revision control system. This doesn't provide all the features FishEye has, but it does allow you to perform code reviews. You can also create reviews from other Source Control Systems **without** writing a plugin by creating patch file reviews, manually via the UI or via the REST interface.

Useful Tutorials

- Crucible SCM Plugin Tutorial

Tell my application when a Crucible review changes state.

You can write an Event Listener Module and have it contact your application when it sees a state change event.

Useful Tutorials

- Event Listener Plugin Module Tutorial

Use FishEye/Crucible from an OpenSocial container

You can write a Gadget Module which gets information from Fisheye/Crucible via a REST Module (or just uses the existing REST API and displays it in an OpenSocial container.

Show information from my application in the FishEye/Crucible UI

You can add your own pages to FishEye/Crucible by writing a Servlet Plugin which renders a Velocity template. Your page will be decorated by FishEye/Crucible so that it has the correct headers and footers. You use Web Item Module Type to place links to your page on existing FishEye/Crucible pages. These links can pass parameters to your page, so, for instance, you can have a link on the file history page which passes the path of the file being viewed to your page.

Useful Tutorials

- FishEye Twitter Integration Plugin Tutorial
- Rendering a Velocity Template from Your Servlet

Add more annotations to the FishEye/Crucible Source View

You can write a Gutter Renderer Module to provide extra information for each line of source code displayed on the annotated file page.

Useful Tutorials

- Gutter Renderer Plugin Tutorial

Frequently Asked Questions

This page contains answers to frequently asked questions posed by FishEye/Crucible developers.

Feel free to comment, make submissions, or pose your own question on FishEye/Crucible Development here.

- **Q**: *I'm getting the error "API access is disabled" as a response from `http://fisheye/api/rest/repositories` on my installation. How do I enable the API as a Fisheye administrator?*
  Click here to expand...

  A: There is a toggle to enable the API under "Server Settings" in the web admin interface. See Configuring the FishEye Web Server for more details.

- **Q**: *Is there any way to return unique results from an EyeQL query?*
  Click here to expand...

  A: It is not currently possible to return unique results.
  An improvement request exists: FE-1136. Your vote and comments on that issue are appreciated.

- **Q**: *How do I use AUI on a page generated by a servlet plugin module?*
  Click here to expand...

  A: FishEye 2.4 and earlier don't include AUI by default, so your servlet will need to explicitly include it. There are two ways to do this:

  1. In FishEye/Crucible 2.4 and later, specify that the context of its page decorator requires the `com.atlassian.auiplugin:ajs` resource. i.e. your `atlassian-plugin.xml` file should include:

     ```
     <web-resource key="aui">
         <dependency>com.atlassian.auiplugin:ajs</dependency>
         <context>atl.general</context>
     </web-resource>
     ```

     assuming your generated page will request the `atl.general` decorator.

  2. If you are rendering a velocity template, include `$webResourceManager.requireResource('com.atlassian.auiplugin:ajs')` in your template.

Plugin Development

FishEye and Crucible use the standard Atlassian Plugins framework, so your first step should be to understand how to use the Atlassian Plugin SDK.

How do I begin developing Atlassian plugins?

The best way to develop plugins for FishEye and Crucible is to read the Atlassian Plugin SDK Tutorial. That tutorial shows you how to set up your development environment, create an empty plugin template, and the basic principles of building, debugging, and testing a plugin. It will take you through the prerequisites and introduce you to some of the resources that Atlassian provides for plugin developers.

Note that where the documentation says `APPLICATION` you need to substitute `fecru`.

Quick Start for the Impatient

For more detail on the initial setup of the SDK, and the first steps below, see Developing your Plugin using the Atlassian Plugin SDK.

First, create your plugin skeleton:

```
$ atlas-create-fecru-plugin
...
Define value for groupId: : com.example.ampstutorial
Define value for artifactId: : fecrutwitter
Define value for version:  1.0-SNAPSHOT: :               # just accept the default
Define value for package:  com.example.ampstutorial: :   # again, just press enter for the default
```

Now run FishEye/Crucible with the skeleton plugin:

```
$ cd fecrutwitter
$ atlas-run
```

You'll need to wait a while for files to download.

Then point your browser to `http://localhost:3990/fecru/admin/viewplugins.do`, giving the administrator password 'password'.

You should see a list of plugins, including one named `fecrutwitter` which should be in the state `Enabled`. If you expand that plugin, you should see that it contains one module, described as 'A Sample Servlet Module'.

Now open the project in your IDE, according to these instructions

We are ready to start implementing our plugin.

How do I learn to develop FishEye/Crucible plugins?

Once you understand the process of developing with the Atlassian Plugin SDK, you need to understand the specifics of developing plugins for FishEye and Crucible.

Read our tutorials.

Understand the plugin module types that FishEye/Crucible supports and the Java API which plugins use to access Fisheye/Crucible.

Where can I find more information?

Other resources for developers:

- Atlassian Developer Network

News and forums:

- Atlassian Answers
- Atlassian Developer blog

REST API Guide

The FishEye/Crucible REST interface provides a simple way for external application to talk to FishEye and Crucible by making HTTP requests.

Introduction to FishEye/Crucible's REST APIs

FishEye/Crucible's REST APIs provide access to resources (data entities) via URI paths. To use a REST API, your application will make an HTTP request and parse the response. By default, the response format is XML. If you wish, you can request JSON instead of XML.

Because the REST API is based on open standards, you can use any web development language to access the API.

An example use case would be a gadget that provides information about build recent changes to a source repository, or lists your open reviews.

FishEye/Crucible's REST APIs provide the following capabilities:

- Browse changes to source repositories.
- Retrieve a list of FishEye or Crucible projects.
- Retrieve user or committer information.
- Search reviews based on custom criteria.
- Create or manipulate code reviews.

Getting Started

If you would like to know more about REST in general, start with the RESTwiki's guide to REST In Plain English.

Read the tutorials for examples of using the FishEye/Crucible REST interface.

How to Authenticate your REST Requests.

FishEye/Crucible API References

- FishEye REST API reference
- Crucible REST API reference

You may find the WADL (Web Application Description Language – can be used to generate REST client stubs) files generated for our REST API useful when writing REST clients. These are not human readable. The WADL files include their respective XML Schema documents that describe the structure of all possible request and response documents.

- Crucible REST API WADL file
    - Crucible REST XML Schema file
- FishEye REST API WADL file
    - FishEye REST XML Schema file

Accessing FishEye/Crucible REST Resources

URIs for a FishEye/Crucible REST API resource have the following structure:

Application	URL
FishEye	http://host:port/webcontext/rest-service-fe/resource-name-v1
Crucible	http://host:port/webcontext/rest-service/resource-name-v1

Example:

```
http://myhost.com:8085/crucible/rest-service/reviews-v1
```

Here is an explanation for each part of the URI:

- `host` and `port` define the host and port where the FishEye/Crucible application lives.
- `webcontext` is the webcontext under which FishEye/Crucible is hosted (as configured in the `<web-server context=""/>` element in the application's `config.xml`
- `rest-service` denotes the Crucible REST API (`rest-service-fe` for FishEye resources).
- `resource-name-v1` identifies the versioned resource such as `/reviews-v1`. In some cases, this may be a generic resource name such as `/foo`. In other cases, this may include a generic resource name and key. For example, `/foo` returns a list of the `foo` items and `/foo/{key}` returns the full content of the `foo` identified by the given `key`.

Legacy REST/XML-RPC API

Fisheye also provides a REST and XML-RPC API which is now deprecated. It is documented here to assist in the maintenance of existing applications, but it will not be developed further and should not be used for new applications.

Authenticating REST Requests

Introduction

By default, requests to the FishEye/Crucible REST interface are executd as the 'anonymous' user – i.e. as though no login has occurred. You'll often need to perform requests as a particular user. There are a number of ways of supplying REST requests with authentication credentials.

Basic Authentication

Basic authentication adds a header to each request which contains a [Base64 encoded] username/password pair. See the Perl REST Client Tutorial for an example of using basic authentication from a REST client. Basic authentication has the disadvantage that every request must contain the username and password in unencrypted text.

Trusted Applications

If you are making a request from a plugin inside another Atlassian application, you can add Trusted Applications headers to the request, by using the SAL RequestFactory service to create a request and calling `addTrustedTokenAuthentication()` to add the Trusted Application headers. Then if FishEye/Crucible has been configured to trust the application your client plugin is installed in, the call will be made as a user with the same name as the remote user, assuming a user with the same name exists.

Token Login

You can use the REST Authentication Service to get a login token which you can then use in other requests. The token should be passed as a parameter named `FEAUTH`, e.g. `http://host:port/context/rest-service/reviews-v1?FEAUTH=admin:1:ac577aa07753052c09c25b4f88fb2c15`.

The advantages of token login over basic authentication are:

- Only the login request contains your password. Further requests just contain the token, which does not contain your password.
- A token can be revoked by logging out of FishEye/Crucible.
- You don't need to add headers to requests.

Note that there is a potential problem with using authentication tokens in REST. Although session tokens are not set to expire, it is possible for users to explicitly purge all their authenticated sessions on the FishEye/Crucible logout page. Doing that will also delete the sessions of any REST client that runs under that user's username. This will cause FishEye/Crucible to treat further invocations with the deleted session token as anonymous (and will not result in an error). This in turn will restrict access to anonymous content only.

FishEye Legacy Remote API

> This API is deprecated. This documentation is provided to assist with the maintenance of existing applications which use this API. New applications should use the FishEye/Crucible REST API.

For developers who are interested in accessing the FishEye functionality remotely, this page describes the methods, data types and structures for accessing the FishEye Remote API.

Additional documentation is available as part of your FishEye installation, under `FISHEYE_HOST/api/`, such as in this example:

```
http://localhost:8060/api/
```

This loads a local HTML page, where you will be able to see whether API Access is currently enabled or disabled on your FishEye instance. You will also be able to link to local code examples for REST and XML-RPC from the FishEye folders.

API mechanisms are REST-ful and XML-RPC.

Before you begin using the remote API, you will need to enable it through the FishEye Admin interface. For instructions, see Configuring the FishEye Web Server.

XML-RPC API

The XML-RPC API can be accessed from `FISHEYE_HOST/api/xmlrpc`, such as in this example:

```
http://localhost:8060/api/xmlrpc
```

REST API

The REST API can be accessed from `FISHEYE_HOST/api/rest/`.

More information on the data types and services can be seen on the specific FishEye REST API pages.

REST Example:

```
http://localhost:8060/api/rest/changeset?rep=cvs&csid=BRANCH_2_2%3Amatt%3A20050517064053
```

This returns the details of a single changeset. Note that parameter values must be URL encoded.

REST return values are always enclosed in a `<response>` root element.

Dates are IS0-8601, in the general form

```
YYYY-MM-DDTHH:MM:SS(Z|[+-]HH:MM)
```

The timezone is optional (GMT is used if omitted). The time component is also optional. The seconds component can contain a fractional part.

For XMLRPC, FishEye returns all dates in GMT using

```
YYYYMMDDTHH:MM:SS
```

Note that no timezone is used.

Authentication

FishEye may be configured to require authentication before accessing a repository. Most methods accept an authentication token parameter. To call a method anonymously, use the empty-string for this parameter.

An authentication token can be acquired (and released) using the `login()` and `logout()` methods.

Examples

The following code example files can be found in the API folder under your FishEye instance:

```
FISHEYE_HOME\content\api\
```

Browse to that folder and you will be able to access the files below:

- Python XML-RPC example: `xmlrpc_example.py`
- Python REST example: `rest_example.py`
- Java REST example: `RestClient.java`
- The open source FishEye Plugin for JIRA provides an example of querying using the API.

Methods

> Each of the REST URLs shown below must be supplied with the same set of parameters as the XML-RPC method (although auth is optional).
> Thus the URL to use for login is `api/rest/login?username=jim&password=rover`.

Log in

```
String login(String username, String password)
```

Description
Log in and create an authentication token. Returns the token if log in was successful, or returns an error otherwise.

REST

```
api/rest/login
```

XML-RPC

```
String login(String username, String password)
```

Log out

```
boolean logout(String auth)
```

Description
Disables the given auth token. Returns true in all cases.

REST

```
api/rest/logout
```

XML-RPC

```
boolean logout(String auth)
```

FishEye Version

```
String fisheyeVersion()
```

Description
Returns the version number of this FishEye instance.

REST

```
/api/rest/fisheyeVersion
```

XML-RPC

```
String fisheyeVersion()
```

Example Return Values
"1.3.8", "1.4"

Since
FishEye 1.4 / Crucible 1.2

Crucible Version

```
String crucibleVersion()
```

Description
Returns the Crucible version number if Crucible is installed. This API method will return an empty String if this isn't a Crucible instance.

REST

```
/api/rest/crucibleVersion
```

XML-RPC

```
String crucibleVersion()
```

Example Return Values
"1.2", "1.2.1", "" (if not a Crucible instance)

Since
FishEye 1.4 / Crucible 1.2

List Repositories

```
String[] listRepositories(String auth)
```

Description
Returns a list of repository names in this FishEye instance.

REST

```
api/rest/repositories
```

XML-RPC

```
String[] listRepositories(String auth)
```

List Paths

```
PathInfo[] listPaths(String auth, String rep, String path)
```

Description
Returns a list of paths immediately under the given path. A path represents either a file or a directory.

REST

```
api/rest/paths
```

XML-RPC

```
PathInfo[] getPaths(String auth, String rep, String path)
```

Get Revision

```
Revision getRevision(String auth, String rep, String path, String rev)
```

Description
Returns the details of a particular revision.

REST

```
api/rest/revision
```

XML-RPC

```
Revision getRevision(String auth, String rep, String path, String rev)
```

List Tags for Revision

```
String[] listTagsForRevision(String auth, String rep, String path, String rev)
```

Description
Returns the tags associated with particular revision as an array of strings.

REST

```
api/rest/tags
```

XML-RPC

```
RevisionTags listTagsForRevision(String auth, String rep, String path, String rev)
```

Path History

```
PathHistory listPathHistory(String auth, String rep, String path)
```

Description
Returns history of a particular path.

REST

```
api/rest/pathHistory
```

XML-RPC

```
PathHistory listPathHistory(String auth, String rep, String path)
```

Get Changeset

```
Changeset getChangeset(String auth, String rep, String csid)
```

Description
Gets the details of a particular changeset.

REST

```
api/rest/changeset
```

XML-RPC

```
Changeset getChangeset(String auth, String csid)
```

List Changesets

```
Changesets listChangesets(String auth, String rep, String path, Date start=null, Date end=null,
Integer maxReturn=null)
```

Description
Lists changes under a given path, optionally between two dates. Returned structure contains a list of changeset ids, from most-recent to least-recent.

REST

```
api/rest/changesets
```

XML-RPC

```
Changesets listChangesets(String auth, String rep, String path)
Changesets listChangesets(String auth, String rep, String path, Date start)
Changesets listChangesets(String auth, String rep, String path, Date start, Date end)
Changesets listChangesets(String auth, String rep, String path, Date start, Date end, Integer
maxReturn)
```

To get changes for the whole repository, use a path of "/"
If the start date is not specified, there is no lower bound.
If the end date is not specified, "now" is used.

The maxReturn clause limits the number of changesets returned by this method. If no limit is specified, FishEye will use its own internal limit (a few thousand). If this limit is exceeded, the return value will be truncated so that it contains the most-recent changesets. The value of this limit is contained in the returned data structure.

EyeQL Query

```
query(String auth, String rep, String query)
```

Description
Execute an EyeQL query. For a "normal" query, returns a list of revision keys that matched to query. If the query contains a "return" clause, then returns a custom Row for each match. The contents of the Row will depend upon the "return" clause.

REST

```
api/rest/query
```

XML-RPC

```
RevisionKey[] query(String auth, String rep, String query)
```

or

```
Row[] query(String auth, String rep, String query)
```

Changeset Bounds

```
ChangesetBounds getChangesetBounds(String auth, String rep, String path=null, Date start=null,
Date end=null)
```

Description
NOT IMPLEMENTED YET. Gets the details of a particular changeset.

REST

```
api/rest/changesetBounds
```

XML-RPC

```
ChangesetBounds getChangesetBounds(String auth, String rep)
ChangesetBounds getChangesetBounds(String auth, String rep, Date start)
ChangesetBounds getChangesetBounds(String auth, String rep, Date start, Date end)
ChangesetBounds getChangesetBounds(String auth, String rep, String path)
ChangesetBounds getChangesetBounds(String auth, String rep, String path, Date start)
ChangesetBounds getChangesetBounds(String auth, String rep, String path, Date start, Date end)
```

Data Types and Structures

Data types used are the same as defined in XML-RPC.

Some methods return data structures. These map into XML-RPC as expected.

For REST calls, structs are encoded as XML elements of the same name (but all lowercase). Members are encoded as sub-elements, or as attributes as indicated below.

RevisionKey

```
struct RevisionKey {
String path; // (REST: attribute)
String rev; // (REST: attribute)
}
```

PathInfo

```
struct PathInfo {
String name; // (REST: attribute)
boolean isFile; // (REST: attribute)
boolean isDir; // (REST: attribute)
boolean isHeadDeleted; // (REST: attribute)
}
```

Revision

```
struct Revision {
String path; // (REST: attribute)
String rev; // (REST: attribute)
String author; // (REST: attribute)
Date date; // (REST: attribute)
String state; // one of "changed" "added" or "deleted" (REST: attribute)
int totalLines; // (REST: attribute)
int linesAdded; // (REST: attribute)
int linesRemoved; // (REST: attribute)
String log;
String csid; // optional (REST: attribute)
String ancestor; // optional (REST: attribute)
}
```

Changeset

```
struct Changeset {
String csid; // (REST: attribute)
Date date; // (REST: attribute)
String author; // (REST: attribute)
String branch; // (REST: attribute)
boolean sealed; // (REST: attribute)
String log;
RevisionKey[] revisions;
}
```

Changesets

```
struct Changesets {
int maxReturn; // (REST: attribute)
String[] csids;
}
```

Description

A list of Changeset ids, most-recent changeset first. `maxReturn` indicates the maximum number of changesets FishEye is configured to return from this method.

ChangesetBounds

```
struct ChangesetBounds {
Changeset first;
Changeset last;
}
```

Row

```
struct Row {
...
}
```

Description
A custom structure, depending on the given EyeQL statement. Each member of Row is typed.

Gadget Development

FishEye/Crucible and Gadgets

FishEye/Crucible can publish gadgets, but it is not a gadget container.

That means that the gadgets published by FishEye/Crucible plugins can be displayed in other gadget containers, such as iGoogle and the JIRA dashboard, but gadgets can't be used as part of the FishEye/Crucible UI itself.

Writing Atlassian Gadgets

This documentation explains how to develop an Atlassian Gadget. All this information applies to developing gadgets to be published by FishEye and Crucible.

This tutorial teaches you how to write a Gadget plugin for FishEye/Crucible.

Integration Tutorials

These tutorials show how to interact with FishEye/Crucible via REST from various client languages.

- Writing a REST Client in Perl
- Writing a REST Client in Python

You may also be interested in the plugin tutorials.

Writing a REST Client in Perl

This tutorial will write a Perl script which removes a user from all open reviews – this might be useful if a reviewer is no longer available, and is holding up the completed status of all their reviews.

First install the `REST::Client` and `JSON` packages from CPAN – the details of doing this will depend on your platform.

> `Data::Dumper` is very useful when developing Perl REST clients:
>
> ```
> print Dumper(from_json($client->responseContent()));
> ```

Our script will retrieve a list of all open reviews, get the uncompleted reviewers for each review, and if one of these matches the user passed as a command line parameter we will complete the reviewer.

- Get all open reviews: `GET /reviews-v1/filter/{filter}`, setting `{filter}` to `allOpenReviews`.
- Get the incomplete reviewers for each of these reviews: `GET /reviews-v1/{id}/reviewers/uncompleted`
- Remove a reviewer: `POST /reviews-v1/{id}/reviewers/{username}`

> When JSON produces lists of objects, the structures produced depend on the number of items in the list.
> If the `/reviews-v1/filter` URL returns a single review, the JSON will look like this:
>
> but if several reviews are returned the JSON will be:
>
> and if there are no reviews it will simply be:

The `toList` function in the code below handles the three cases above.

```
use REST::Client;
use JSON;
# Data::Dumper makes it easy to see what the JSON returned actually looks like
# when converted into Perl data structures.
use Data::Dumper;
use MIME::Base64;

sub toList {
   my $data = shift;
   my $key = shift;
   if (ref($data->{$key}) eq 'ARRAY') {
       $data->{$key};
   } elsif (ref($data->{$key}) eq 'HASH') {
       [$data->{$key}];
   } else {
       [];
   }
}
if ($#ARGV ne 0) {
    print "usage: $0 <username>\n";
    exit 1;
}
my $reviewerToRemove = $ARGV[0];
my $username = 'admin';
my $password = 'admin';
my $headers = {Accept => 'application/json', Authorization => 'Basic ' . encode_base64($username .
':' . $password)};
my $client = REST::Client->new();
$client->setHost('http://localhost:3990');
$client->GET(
    '/fecru/rest-service/reviews-v1/filter/allOpenReviews',
    $headers
);
my $response = from_json($client->responseContent());
my $reviews = toList($response->{'reviews'},'reviewData');
foreach $review (@$reviews) {
 my $id = $review->{'permaId'}->{'id'};
 $client->GET(
     '/fecru/rest-service/reviews-v1/' . $id . '/reviewers/uncompleted',
     $headers
 );
 my $response = from_json($client->responseContent());
 my $incompleteReviewers = toList($response->{'reviewers'},'reviewer');
 foreach $reviewer (@$incompleteReviewers) {
  $myreviewerUserName = $reviewer->{'userName'};
  if ($reviewerToRemove eq $myreviewerUserName) {
   print "Removing " . $reviewer->{'displayName'} . " from review " . $id . "\n";
      $client->DELETE(
          '/fecru/rest-service/reviews-v1/' . $id . '/reviewers/' . $reviewer->{'userName'},
          $headers
         );
      print Dumper($client->responseContent());
     }
 }
}
```

Writing a REST Client in Python

This tutorial teaches you how to interact with FishEye/Crucible's REST interface from a Python program.

We'll write a Python script which lists the users who are uncompleted reviewers of at least one open review.

This tutorial assumes that you have Python 2.6.3 installed. Note that the default version on Mac OS X 10.6 is 2.5.

Create a new directory to write your client in and `cd` into it.

First you'll need to install the python-rest-client package. Download the 0.2 distribution and unpack it in your current directory.

Set your `PYTHONPATH` environment variable to include the python-rest-client:

```
export PYTHONPATH=./python-rest-client
```

To get all open reviews we use the URL /reviews-v1/filter/{filter}, setting `{filter}` to `allOpenReviews`, and to get the incomplete reviewers for each of these reviews we use the URL /reviews-v1/{id}/reviewers/uncompleted

When JSON produces lists of objects, the structures produced depend on the number of items in the list.
If the `/reviews-v1/filter` URL returns a single review, the JSON will look like this:

but if several reviews are returned the JSON will be:

and if there are no reviews it will simply be:

The `toList` function in the code below normalises each of the responses shown above to a list:

```
reviews = ...one of the dictionaries above...
reviewList = toList(reviews["reviews"], "reviewData")
```

```
# import the standard JSON parser
import json
# import the REST library
from restful_lib import Connection

base_url = "http://localhost:6060/foo/rest-service/reviews-v1"

conn = Connection(base_url, username="admin", password="admin")

# the rest library can't distinguish between a property and a list of properties with one element.
# this function converts a json object into a list with many, one, or no elements
# o is the dictionary containing the list
# key is the key containing the list (if any)
def toList(o, key):
    if isinstance(o,dict):
        elements = o[key]
        if not isinstance(elements,list):
            return [elements]
        else:
            return elements
    else:
        return []

# a function to get the uncompleted reviwers for a single review
def uncompletedReviewers(review):
    id = review[u'permaId'][u'id']
    resp = conn.request_get("/" + id + "/reviewers/uncompleted", args={},
headers={'content-type':'application/json', 'accept':'application/json'})
    status = resp[u'headers']['status']
    if status == '200' or status == '304':
        reviewers = toList(json.loads(resp[u'body'])[u'reviewers'],u'reviewer')
        return map(lambda r: r[u'displayName'], reviewers)
    else:
        return []

# get a dictionary containing the response to the GET request
# we specify JSON as the format as that is easy to parse in Python
resp = conn.request_get("/filter/allOpenReviews", args={},
headers={'content-type':'application/json', 'accept':'application/json'})

status = resp[u'headers']['status']
# check that we either got a successful response (200) or a previously retrieved, but still valid
response (304)
if status == '200' or status == '304':
    reviews = toList(json.loads(resp[u'body'])[u'reviews'],u'reviewData')
    reviewerLists = map(uncompletedReviewers,reviews)
    reviewers = reduce(lambda a, b: set(a).union(set(b)), reviewerLists, set())
    print 'Incomplete Reviewers: '
    for r in reviewers:
        print '    ',r
else:
    print 'Error status code: ', status
```

Plugin Tutorials

These tutorials will help you understand how each of the plugin module types supported by FishEye/Crucible work.

- Crucible SCM Plugin Tutorial — Crucible SCM modules are plugins that make version control systems accessible to Crucible.
- Event Listener Plugin Module Tutorial — This is a brief tutorial which teaches you how to write a trivial event listener plugin.
- FishEye Twitter Integration Plugin Tutorial — The plugin created in this tutorial sends each of your commit messages to your Twitter account.
- Gadget Tutorial — This tutorial will teach you how to write a simple gadget which can display information from Crucible on the JIRA dashboard.
- Gutter Renderer Plugin Tutorial — This tutorial teaches you how to add extra information to annotated views of files, and to diffs
- Rendering a Velocity Template from Your Servlet — Velocity allows your Servlet Plugins to render HTML pages from simple templates.
- REST Service Plugin Module Tutorial — provide your own REST API
- Storing Plugin Settings — This tutorial demonstrates how to use SAL (Shared Access Layer) to let your plugin store its configuration settings.
- Using Logging From Your Plugin — This tutorial describes how to log messages from your plugin.

You may also be interested in the integration tutorials.

Crucible SCM Plugin Tutorial

On this page:

- Crucible SCM Plugins
  - Creating a Project
  - Crucible SCM Plugin API
  - Servlet Based Administration Pane
  - Packaging, Deploying and Running

Crucible SCM Plugins

Crucible SCM modules are plugins that make version control systems accessible to Crucible. An SCM plugin can be used to give Crucible the ability to work with a custom version control system that is not supported out of the box. SCM plugins are independent from FishEye's version control integrations and allow Crucible to run standalone. Crucible ships with a number of built-in SCM plugins, including Subversion and Perforce.

In this section we will implement a new Crucible SCM Plugin and explore Crucible's public SCM API. The example builds a module that exposes the underlying file system as the "repository", so that users can perform reviews of files on the server file system.

Creating a Project

To start, we use the Atlassian Plugins SDK to create a new plugin project. If you haven't done so already, download and install the SDK first.

```
$ atlas-create-fecru-plugin
Executing: /Users/ervzijst/opt/atlassian-plugin-sdk-3.0-beta7/apache-maven/bin/mvn
com.atlassian.maven.plugins:maven-fecru-plugin:3.0-beta7:create
[INFO] Scanning for projects...
...
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [com.atlassian.maven.plugins:maven-fecru-plugin:3.0-beta7:create]
(aggregator-style)
[INFO] ------------------------------------------------------------------------
...
[INFO] [fecru:create]
...
[INFO] Setting property: classpath.resource.loader.class =>
'org.codehaus.plexus.velocity.ContextClassLoaderResourceLoader'.
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
...
Define value for groupId: : com.atlassian.crucible.example.scm
Define value for artifactId: : example-scm-plugin
Define value for version:  1.0-SNAPSHOT: :
Define value for package:  com.atlassian.crucible.example.scm: :
Confirm properties configuration:
groupId: com.atlassian.crucible.example.scm
artifactId: example-scm-plugin
version: 1.0-SNAPSHOT
package: com.atlassian.crucible.example.scm
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating OldArchetype: fecru-plugin-archetype:3.0-beta7
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.atlassian.crucible.example.scm
[INFO] Parameter: packageName, Value: com.atlassian.crucible.example.scm
[INFO] Parameter: basedir, Value: /Users/ervzijst/workspace
[INFO] Parameter: package, Value: com.atlassian.crucible.example.scm
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: example-scm-plugin
[INFO] ********************* End of debug info from resources from generated POM
***********************
[INFO] OldArchetype created in dir: /Users/ervzijst/workspace/example-scm-plugin
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 minute 57 seconds
[INFO] Finished at: Fri Oct 02 10:12:05 EST 2009
[INFO] Final Memory: 28M/50M
[INFO] ------------------------------------------------------------------------
```

Note that this step interactively asks you to supply the groupId, artifactId, package and version number you want to use for your new plugin.

This creates a new project that has a dependency on `atlassian-fisheye-api`. This library contains the basic API components required by plugins. It also comes with dependencies on `atlassian-crucible-scmutils` (which provides a collection of utility class that helps you spawn processes outside JVM – which can be useful for SCM plugins that fork command line binaries to talk to their repositories) as well as `atlassian-plugins-core`.

The `pom.xml` looks something like:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.atlassian.crucible.example.scm</groupId>
    <artifactId>example-scm-plugin</artifactId>
```

```
    <version>1.0-SNAPSHOT</version>

    <organization>
        <name>Example Company</name>
        <url>http://www.example.com/</url>
    </organization>

    <name>example-scm-plugin</name>
    <description>This is the com.atlassian.crucible.example.scm:example-scm-plugin plugin for
Atlassian FishEye/Crucible.</description>
    <packaging>atlassian-plugin</packaging>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.atlassian.fisheye</groupId>
            <artifactId>atlassian-fisheye-api</artifactId>
            <version>${fecru.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.atlassian.crucible</groupId>
            <artifactId>atlassian-crucible-scmutils</artifactId>
            <version>${fecru.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.atlassian.plugins</groupId>
            <artifactId>atlassian-plugins-core</artifactId>
            <version>2.3.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>com.atlassian.maven.plugins</groupId>
                <artifactId>maven-fecru-plugin</artifactId>
                <version>3.0-beta7</version>
                <extensions>true</extensions>
                <configuration>
                    <productVersion>${fecru.version}</productVersion>
                    <productDataVersion>${fecru.data.version}</productDataVersion>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <properties>
        <fecru.version>2.0.5-429</fecru.version>
```

```
        <fecru.data.version>2.0.4.1-SNAPSHOT</fecru.data.version>
    </properties>
</project>
```

> **IDEA Users**
> If you are using IntelliJ for development, then depending on the version of IDEA, you might need to run `atlas-mvn idea:idea` to generate the project files. Opening the pom file directly is known to miss the parent dependencies.

Crucible SCM Plugin API

Crucible's public API can be browsed online and contains the functionality needed to develop a custom SCM plugin in the package `com.atlassian.crucible.scm`. It consists of a set of interfaces, some of which are optional, for browsing a repository, accessing its directories, retrieving file contents and exploring changes between revisions.

At the very least, your SCM plugin should implement the `com.atlassian.crucible.scm.SCMModule` interface that defines the new plugin. The module is then used to create one or more repository instances:

```
package com.atlassian.scm;

import com.atlassian.crucible.scm.SCMModule;
import com.atlassian.crucible.scm.SCMRepository;
import com.atlassian.plugin.ModuleDescriptor;

import java.util.Collection;
import java.util.Collections;

public class ExampleSCMModule implements SCMModule {

    private ModuleDescriptor moduleDescriptor;
    private List<SCMRepository> repos = Collections.emptyList();

    public String getName() {
        return "Example File System SCM.";
    }

    public Collection<? extends SCMRepository> getRepositories() {
        return repos;
    }

    public void setModuleDescriptor(ModuleDescriptor moduleDescriptor) {
        this.moduleDescriptor = moduleDescriptor;
    }

    public ModuleDescriptor getModuleDescriptor() {
        return moduleDescriptor;
    }
}
```

When your module is instantiated, Crucible passes a `ModuleDescriptor` instance to it containing information about the plugin. The `getRepositories()` method returns the repositories offered by this plugin. Currently we're returning an empty collection.

To be able to use the Crucible administration console to configure our plugin and specifiy the locations of the repositories we want to use, we will also implement the `Configurable` interface that allows for the injection of a custom configuration bean (by implementing `SimpleConfiguration`) whose properties can be manipulated through the administration interface for which we will write a small servlet. In our custom configuration bean we'll add a property for the base path or root directory of the file system based repositories we want to offer.

The plugin configuration is written to disk and fed to our `SCMModule` when Crucible starts up. Our plugin is responsible for generating and parsing that data, so we're free to choose the format. The `ModuleConfigurationStore` provides persistent storage and will automatically be injected into our plugin if we create a constructor that takes it as an argument. For the serialization, let's use simple XML serialization through XStream (using XStream is convenient as it is one of the dependencies for `atlassian-crucible-scmutils`):

```
package com.atlassian.scm;

import com.atlassian.fisheye.plugins.scm.utils.SimpleConfiguration;

public class ExampleConfiguration implements SimpleConfiguration {

    private String name;
    private String basePath;

    public String getName() {
        return name;
    }

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

    public String getBasePath() {
        return basePath;
    }

    public void setBasePath(String basePath) {
        this.basePath = basePath;
    }
}
```

Now we make the required changes to our SCMModule to read and write the configuration:

```
public class ExampleSCMModule implements SCMModule, Configurable<List<ExampleConfiguration>> {

    private ModuleDescriptor moduleDescriptor;
    private ModuleConfigurationStore store;

    public ExampleSCMModule(ModuleConfigurationStore store) {
        this.store = store;
    }

    [...]

    public List<ExampleConfiguration> getConfiguration() {
        byte[] configData = store.getConfiguration(moduleDescriptor);
        if (configData != null) {
            try {
                return (List<ExampleConfiguration>)getXStream().fromXML(new String(configData,
"UTF8"));
            } catch (Exception e) {
                throw new RuntimeException("Error reading configuration:" + configData, e);
            }
        }
        return new ArrayList<ExampleConfiguration>();
    }

    public void setConfiguration(List<ExampleConfiguration> config) {
        try {
            store.putConfiguration(moduleDescriptor, getXStream().toXML(config).getBytes("UTF8"));
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException("UTF8 encoding not supported", e);
        }
    }

    private XStream getXStream() {
        XStream xstream = new XStream();
        xstream.setClassLoader(moduleDescriptor.getPlugin().getClassLoader());
        return xstream;
    }
    [...]
```

Now that we have access to the configuration data, which describes the repositories, we can go ahead and implement our file system based repository class.

The `SCMRepository` interface offers basic functionality for retrieving file contents of specific file revisions. It is queried by Crucible when a user adds files to a review. Depending on the optional interfaces you implement in addition to `SCMRepository`, your implementation could

also have the ability to browse the repository and to explore different versions of each file. Because a standard file system does not store version information, we'll only offer directory browsing in this example. As a revision key or version number we shall simply use the last modification date that is stored by the file system.

```
package com.atlassian.scm;

import com.atlassian.crucible.scm.SCMRepository;
import com.atlassian.crucible.scm.RevisionData;
import com.atlassian.crucible.scm.RevisionKey;
import com.atlassian.crucible.scm.DetailConstants;
import com.cenqua.crucible.model.Principal;

import java.io.OutputStream;
import java.io.IOException;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Date;
import java.net.MalformedURLException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

import org.apache.commons.io.IOUtils;

public class ExampleSCMRepository implements SCMRepository {

    private final ExampleConfiguration config;

    public ExampleSCMRepository(ExampleConfiguration config) {
        this.config = config;
    }

    public boolean isAvailable(Principal principal) {
        return true;
    }

    public String getName() {
        return config.getName();
    }

    public String getDescription() {
        return getName() + " file system repo at: " + config.getBasePath();
    }

    public String getStateDescription() {
        return "Available";
    }

    public RevisionData getRevisionData(Principal principal,
        RevisionKey revisionKey) {
        if (revisionKey.equals(currentKey(revisionKey.getPath()))) {
            File f = getFile(revisionKey.getPath());

            RevisionData data = new RevisionData();
            data.setDetail(DetailConstants.COMMIT_DATE, new Date(f.lastModified()));
            data.setDetail(DetailConstants.FILE_TYPE, f.isDirectory() ? "dir" : "file");
            data.setDetail(DetailConstants.ADDED, true);
            data.setDetail(DetailConstants.DELETED, false);
            try {
                data.setDetail(DetailConstants.REVISION_LINK, f.toURL().toString());
            } catch (MalformedURLException e) {
            }
            return data;
        } else {
            throw new RuntimeException("Revision " + revisionKey.getRevision() + " of file " +
revisionKey.getPath() + " is no longer available.");
        }
    }

    public void streamContents(Principal principal, RevisionKey revisionKey,
        OutputStream outputStream) throws IOException {
        if (revisionKey.equals(currentKey(revisionKey.getPath()))) {
            InputStream is = new FileInputStream(getFile(revisionKey.getPath()));
```

```
            try {
                IOUtils.copy(is, outputStream);
            } finally {
                IOUtils.closeQuietly(is);
            }
        } else {
            throw new RuntimeException("Revision " + revisionKey.getRevision() + " of file " +
revisionKey.getPath() + " is no longer available.");
        }
    }

    public RevisionKey getDiffRevisionKey(Principal principal,
        RevisionKey revisionKey) {
        // diffs are not supported in this example
        return null;
    }

    /**
     * Returns a {@link RevisionKey} instance for the specified file. Because we
     * do not support versioning, the revision string will be set to the file's
     * last modification date.
     *
     * @param path
     * @return
     */
    private RevisionKey currentKey(String path) {
        File f = getFile(path);
        return new RevisionKey(path, createDateFormat().format(new Date(f.lastModified())));
    }

    /**
     * Takes the name of a file in the repository and returns a file handle to the
     * file on disk.
     *
     * @param path
     * @return
     */
    private File getFile(String path) {
        return new File(config.getBasePath() + File.separator + path);
    }

    private DateFormat createDateFormat() {
```

```
        return new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
    }
}
```

In the above code, the `getRevisionData()` method is used by Crucible to retrieve versioning properties for a specific revision of a file in the repository. Although the file system does not keep track of older versions, we can provide some of the properties. Most important are the predefined constants `DetailConstants.FILE_TYPE`, `DetailConstants.ADDED`, `DetailConstants.DELETED` (the last two indicate whether the file was newly created (ADDED), or has been removed from the repository (DELETED) as part of the revision) and `DetailConstants.REVISION_LINK`. In addition to the predefined constants, a repository implementation is free to add custom properties.

We are not able to implement `getDiffRevisionKey()` due to the lack of version information on the file system.

Before we continue to extend the functionality of the `ExampleSCMRepository`, we should go back to `ExampleSCMModule` and implement `getRepositories()`:

```
[...]

    // initialize at null to trigger loading from the configuration
    private List<SCMRepository> repos = null;

    public synchronized Collection<SCMRepository> getRepositories() {
        if (repos == null) {
            repos = new ArrayList<SCMRepository>();
            for (ExampleConfiguration config : getConfiguration()) {
                repos.add(new ExampleSCMRepository(config));
            }
        }
        return repos;
    }

    public void setConfiguration(List<ExampleConfiguration> config) {
        try {
            store.putConfiguration(moduleDescriptor, xstream.toXML(config).getBytes("UTF8"));
            // we're given a new configuration, so reset our repositories:
            repos = null;
        } catch (UnsupportedEncodingException e) {
            throw new RuntimeException("UTF8 encoding not supported", e);
        }
    }

    [...]
```

Our SCMModule now properly creates the repository instances according to the configuration.

The above code gives us a very simple Crucible SCM plugin. However you would normally also want to implement the `com.atlassian.crucible.scm.DirectoryBrowser` and `com.atlassian.crucible.scm.HasDirectoryBrowser` interfaces. The `DirectoryBrowser` gives Crucible the ability to let the user interactively browse the repository and select files to review. If you do not provide a `DirectoryBrowser`, the only way to create a review for files in your repository is when the required files and file revisions are known up front.

In this example, we'll implement `DirectoryBrowser`:

```
public class FileSystemSCMRepository implements HasDirectoryBrowser, DirectoryBrowser {

    [...]

    public DirectoryBrowser getDirectoryBrowser() {
        return this;
    }

    public List<FileSummary> listFiles(Principal principal, String path) {
        List<FileSummary> files = new ArrayList<FileSummary>();
        for (String p : list(path, true)) {
            files.add(new FileSummary(currentKey(p)));
        }
        return files;
    }

    public List<DirectorySummary> listDirectories(Principal principal, String path) {
        List<DirectorySummary> files = new ArrayList<DirectorySummary>();
        for (String p : list(path, false)) {
            files.add(new DirectorySummary(p));
        }
        return files;
    }

    public FileHistory getFileHistory(Principal principal, String path, String pegRevision) {
        return new FileHistory(Collections.singletonList(currentKey(path)));
    }

    private List<String> list(String path, boolean returnFiles) {
        File parent = getFile(path);
        List<String> files = new ArrayList<String>();
        if (parent.isDirectory()) {
            File[] children = parent.listFiles();
            // this may be null if we can't read the directory, for instance.
            if (children != null) {
                for (File f : children) {
                    if (f.isFile() && returnFiles || f.isDirectory() && !returnFiles) {
                        files.add(getPath(f));
                    }
                }
            }
        }
        return files;
    }

    /**
     * @return the path for a given File relative to the base configured for this
     *         repository -- the path doesn't include the base component.
     */
    private String getPath(File file) {
        String s = file.getAbsolutePath();
        if (!s.startsWith(config.getBasePath())) {
            throw new RuntimeException("Invalid file with path " + s + " is not under base " +
config.getBasePath());
        }
        return s.substring(config.getBasePath().length() + 1);
    }

    [...]
```

This is as far as we can go with the file system. In most cases you will be integrating version control systems that keep track of all previous revisions of the resources in the repository and you would expose this to Crucible by also implemening `HasChangelogBrowser` and `ChangelogBrowser`.

Servlet Based Administration Pane

With the code for the module and the repository in place, we can focus on our servlet that provide plugin administration in Crucible's administration section. The easiest way to do this is to subclass `com.atlassian.fisheye.plugins.scm.utils.SimpleConfigurationServlet` and implement the three abstract methods:

```
package com.atlassian.crucible.example.scm;

import com.atlassian.fisheye.plugins.scm.utils.SimpleConfigurationServlet;
import com.atlassian.plugin.PluginAccessor;
import com.atlassian.crucible.spi.FisheyePluginUtilities;

public class ExampleSCMConfigServlet extends SimpleConfigurationServlet<ExampleConfiguration> {

    public ExampleSCMConfigServlet(PluginAccessor pluginAccessor,
        FisheyePluginUtilities fisheyePluginUtilities) {
        super(pluginAccessor, fisheyePluginUtilities);
    }

    protected ExampleConfiguration defaultConfig() {
        return new ExampleConfiguration();
    }

    protected String getProviderPluginModuleKey() {
        return "com.atlassian.crucible.example.scm.example-scm-plugin:scmprovider";
    }

    protected String getTemplatePackage() {
        return "/examplescm-templates";
    }
}
```

The `getTemplatePackage()` method returns the name of the resource directory that contains the velocity templates that determine how the configuration pane will be rendered. The template directory must be in `src/main/resources` so Crucible can find them. We'll create three different pages: one that lists the current configuration `list.vm`, one to edit a repository's configuration `edit.vm` and one that is displayed when the user tries to manipulate a non-existing repository instance (`nosuchrepo.vm`):

src/main/resource/examplescm-templates/list.vm

```
<html>
<head>
    <link rel="stylesheet" href="$request.contextPath/$STATICDIR/main.css" type="text/css" />
</head>
<body class="plugin">
<div class="box formPane">
<table class="adminTable">
#if ($configs.empty)
        <tr><td>No File System repositories are configured.</td></tr>
#else
    <tr>
        <th>Name</th>
        <th>Base Path</th>
        <th><!-- for edit link --></th>
        <th><!-- for delete link --></th>
    </tr>
    #foreach ($config in $configs)
    <tr>
        <td>$config.name</td>
        <td>$config.basePath</td>
        <td><a href="./examplescm?name=$config.name">Edit</a></td>
        <td><a href="./examplescm?name=$config.name&amp;delete=true">Delete</a></td>
    </tr>
    #end
#end
    <tr>
        <td class="verb"><a href="./examplescm?name=_new">Add a repository.</a></td>
    </tr>
</table>
</div>
</body>
</html>
```

src/main/resource/examplescm-templates/edit.vm

```
<html>
<head>
    <link rel="stylesheet" href="$request.contextPath/$STATICDIR/main.css" type="text/css" />
</head>
<body class="plugin">
<div class="box formPane">
<form action="./examplescm" method="POST">
    #if ($config.name)
    <input type="hidden" name="name" value="$!config.name"/>
    #end
    <table class="adminTable">
         #if ($errorMessage)
        <tr><td colspan="2"><span class="errorMessage">$errorMessage</span></td></tr>
        #end
        <tr>
            <td class="tdLabel"><label class="label">Name:</label></td> <td><input
            #if ($config.name)
                disabled="true"
            #else
                name="name"
            #end
            type="text"  value="$!config.name"/> </td>
        </tr>
        <tr>
            <td class="tdLabel"><label class="label">Base Path:</label></td> <td><input
type="text" name="basePath" value="$!config.basePath"/> </td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save"/>
            </td>
        </tr>
    </table>
</form>
</div>
</body>
</html>
```

src/main/resource/examplescm-templates/nosuchrepo.vm

```
<html>
<head>
    <link rel="stylesheet" href="$request.contextPath/$STATICDIR/main.css" type="text/css" />
</head>
<body class="plugin">
<p>
There is no repository named '$name'.
</p>
</body>
</html>
```

Finally we tie everything together in the mandatory `atlassian-plugin.xml` file that describes the new plugin, contains its name, location of the servlet and the classnames Crucible uses to instantiate the components. Because this is an SCM plugin, we must add the `<scm/>` element:

src/main/resources/atlassian-plugin.xml

```
<atlassian-plugin key="${atlassian.plugin.key}" name="example-scm-plugin" plugins-version="2">
    <plugin-info>
        <description>An example SCM provider for the local file system</description>
        <vendor name="Atlassian" url="http://www.atlassian.com"/>
        <version>1.0-SNAPSHOT</version>
        <param name="configure.url">/plugins/servlet/examplescm</param>
    </plugin-info>

    <scm name="Example File System SCM" key="scmprovider"
class="com.atlassian.crucible.example.scm.ExampleSCMModule">
        <description>Example SCM implementation for local file system</description>
    </scm>

    <servlet name="Example File System SCM Configuration Servlet" key="configservlet"
class="com.atlassian.crucible.example.scm.ExampleSCMConfigServlet" adminLevel="system">
        <description>Allows Configuration of File System example SCM Plugin</description>
        <url-pattern>/examplescm</url-pattern>
    </servlet>
</atlassian-plugin>
```

Packaging, Deploying and Running

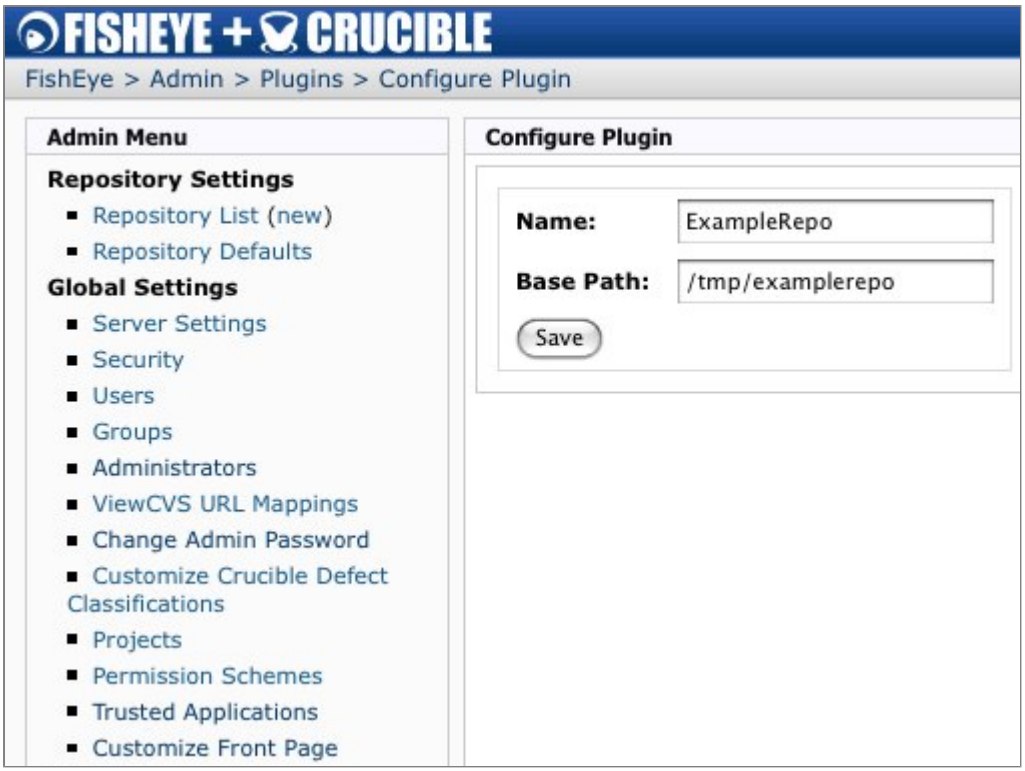
Now we can test the plugin by deploying it into a Crucible instance. With the Atlassian Plugin SDK this is conveniently done with the `atlas-run` command. This will start the bundled, pre-confgured FishEye/Crucible instance and automatically compile, package and deploy your new plugin:

```
$ atlas-run Executing: /Users/ervzijst/opt/atlassian-plugin-sdk-3.0-beta7/apache-maven/bin/mvn
com.atlassian.maven.plugins:maven-amps-dispatcher-plugin:run
[INFO] Scanning for projects...
[INFO] ----------------------------------------------------------------------
[INFO] Building example-scm-plugin
...
[INFO] Building jar:
/Users/ervzijst/workspace/example-scm-plugin/target/example-scm-plugin-1.0-SNAPSHOT.jar
...
INFO  - FishEye/Crucible 2.0.5 (build-429), Built on 2009-09-28
INFO  - FishEye: Developer License registered to Atlassian. ()
INFO  - Periodic polling for software updates is disabled.
INFO  - Starting plugin system...
INFO  - Starting database...
INFO  - Server started on :3990 (http) (control port on 127.0.0.1:39901)
[INFO] fecru started successfully and available at http://localhost:3990/fecru
[INFO] Type CTRL-C to exit
```

Now visit `http://localhost:3990/fecru` and use the admin password "`password`" to go to the admin section and configure a new instance of our new file system SCM plugin.

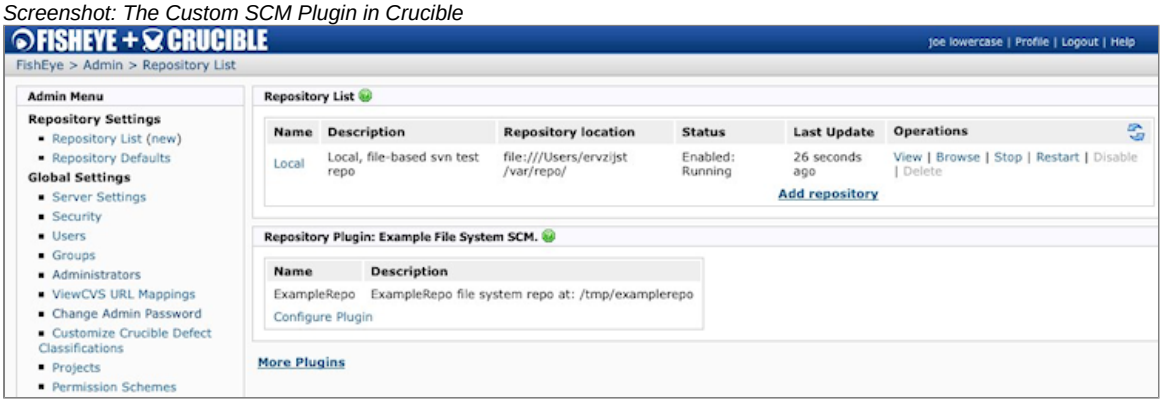
Click "Configure" to create a file system based repository:

Screenshot: Creating a File-System Based Repository



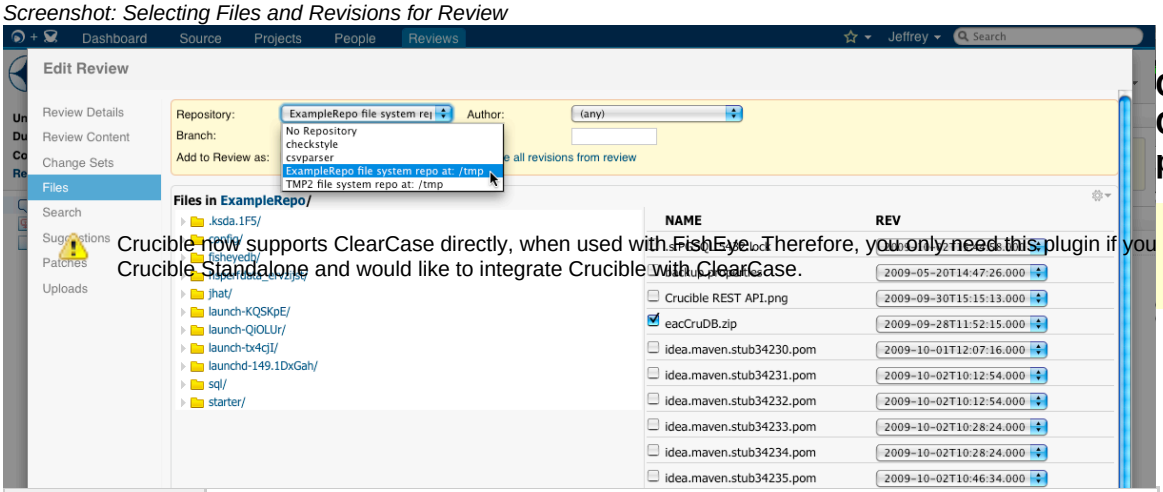
When the repository is created, navigate to "Repository List". Our custom Crucible SCM Plugin will now show up in the list and is ready to use:

Screenshot: The Custom SCM Plugin in Crucible



When reviewing files from the plugin repository, click on the "Manage Files" tab in a new or existing review and then select the repository from the pull down list and select the files and revisions you want to review:

Screenshot: Selecting Files and Revisions for Review



Crucible ClearCase plugin

Crucible now supports ClearCase directly, when used with FishEye. Therefore, you only need this plugin if you are using Crucible Standalone and would like to integrate Crucible with ClearCase.

Name	Crucible ClearCase plugin
Version	0.2.1

Product Versions	1.6.1 to 2.0
Author(s)	Ross Rowe
Homepage	http://confluence.atlassian.com/display/CRUCIBLE/Crucible+ClearCase+plugin
Price	Free
License	BSD
JavaDocs	TBA
IssueTracking	Jira
Subversion URL	https://svn.atlassian.com/svn/public/contrib/crucible/crucible-clearcase-plugin or browse via fisheye
Download JAR	crucible-clearcase-0.0.1.jar (supports 1.6.1 and 1.6.2) crucible-clearcase-0.0.3.jar (supports 1.6.3 to 1.6.5) crucible-clearcase-0.0.5.jar (supports 1.6.6) crucible-clearcase-0.2.1.jar (supports 2.0)
Download Source	TBA

Description/Features

A plugin for Crucible that facilitates the usage of ClearCase UCM source code repositories.

Usage

> This plugin requires Crucible 1.6.1 or higher. In addition, at the moment this plugin only supports the LiteSCM module provided by Crucible and ClearCase UCM.

Installation

The plugin can be installed by copying the crucible-clearcase-0.2.0.jar file into the CRUCIBLE_HOME/var/plugins/user directory and restarting Crucible. Detailed instructions on the plugin installation steps can be found at the Managing Plugins page.

Configuring the plugin

Once the plugin has been installed, under the 'Administration' - 'Repository List' option, there should be a 'Plugin Repository List: ClearCase' entry. Select 'Configure Plugin', then 'Add a repository'. The fields required are:

Field	Description
Name	The name for the repository eg. *Project*
Main Component	The component containing the source files eg. *project_main@\pvob*
Integration Stream	The integration stream for your project eg. *stream:Project_Integration@\pvob*
View Location	The location of the main component for the view (snapshot or dynamic) for the project eg. *c:/projects/project/project_main*

Reviews can be driven from change sets, which are populated from the list baselines that exist for the ClearCase project, or from the file contents of the view location.

Version History

Version	Date	Description
0.2.1	29 Sep 2009	Included fix for CCCP-5
0.2.0	18 Sep 2009	Included several performance improvements to the change log and file browsers
0.1.0	7 Jul 2009	CCCP-3 Updated plugin to support Crucible 2.0
0.0.5	4 Mar 2009	Updated fix for CCCP-1
0.0.4	10 Feb 2009	Recompiled plugin to support Crucible 1.6.6
0.0.3	4 Feb 2009	Included fix for CCCP-1
0.0.2	11 Nov 2008	Updated plugin to support Crucible 1.6.3

0.0.1	1 Oct 2008	Initial version of plugin

Screenshots

Screenshots

There are no images attached to this page.

Event Listener Plugin Module Tutorial

This is a brief tutorial which teaches you how to write a trivial event listener plugin. The FishEye Twitter Integration Plugin Tutorial contains an event listener plugin module as part of a more complex plugin.

An Event Listener must implement com.atlassian.event.EventListener.

Your implementation is added to `atlassian-plugin.xml`:

```
<listener key="trivial-listener" class="com.example.eventlistener.MyTrivialListener"/>
```

Your implementation's `getHandledEventClasses` method must return an array of the event classes you wish to be notified of, in our example, comment creation and update events. (here are other event types)

```
public class MyTrivialListener implements EventListener {
...
    public Class[] getHandledEventClasses() {
        return new Class[] {CommentCreatedEvent.class, CommentUpdatedEvent.class};
    }
}
```

When a commit occurs our `handleEvent` method will be called:

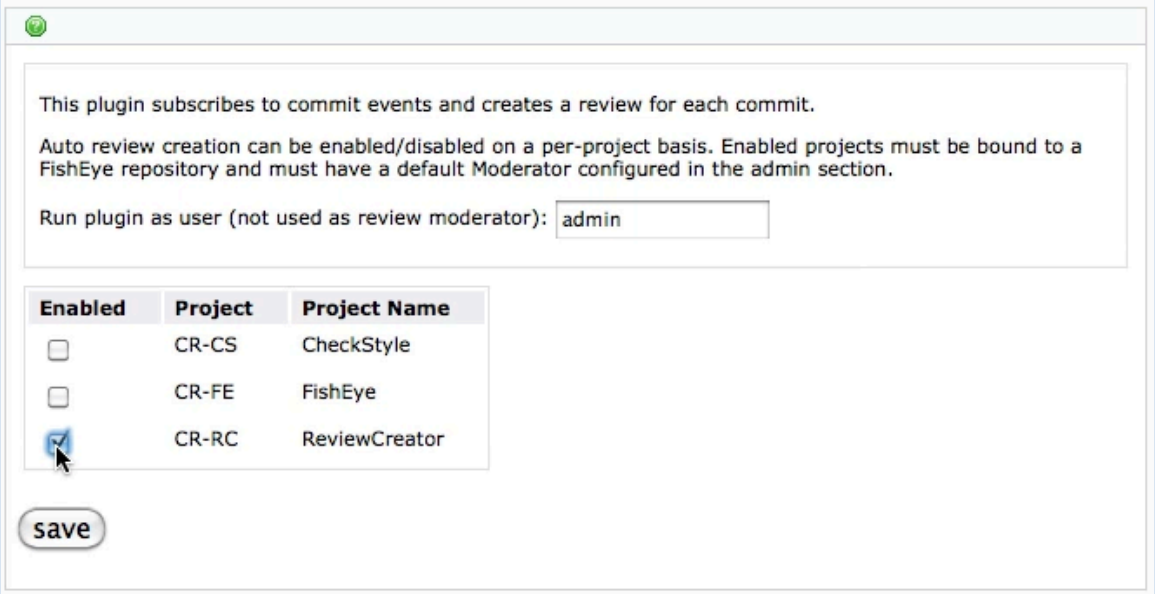
```
public class MyTrivialListener implements EventListener {
...
    public void handleEvent(Event event) {
        ReviewCommentEvent commentEvent = (ReviewCommentEvent)event;
        System.out.println("Changed comment " + commentEvent.getCommentId());
    }
...
}
```

Learn More
For more event listener example code that can help you get started, browse the source code of the **Automatic Review Creator Plugin** at:

http://svn.atlassian.com/fisheye/browse/public/atlassian/crucible/plugins/review-creator/trunk

or check it out with:

```
svn co http://svn.atlassian.com/svn/public/atlassian/crucible/plugins/review-creator/trunk
review-creator
```



FishEye Twitter Integration Plugin Tutorial

The plugin created in this tutorial sends each of your commit messages to your Twitter account.

It teaches you how to:

- Get the output of your servlet 'decorated' so that it has the correct headers and footers.
- Use a servlet to add a new pane to the user settings dialog.
- Render your servlet output using velocity.
- Write an event listener to listen for commit events.
- Make a REST call to an external application.
- Use a Spring Component module to provide a service to several of your plugin modules.
- Depend on a 3rd party library.

The source code for this tutorial is available at
`http://svn.atlassian.com/svn/public/atlassian/fisheye/plugins/fecrutwitter-tutorial/trunk`

For more detail on the initial setup of the SDK, and the first steps below, see Developing your Plugin using the Atlassian Plugin SDK.

First, create your plugin skeleton:

```
$ atlas-create-fecru-plugin
...
Define value for groupId: : com.example.ampstutorial
Define value for artifactId: : fecrutwitter
Define value for version:  1.0-SNAPSHOT: :               # just accept the default
Define value for package:  com.example.ampstutorial: :   # again, just press enter for the default
```

Now run FishEye/Crucible with the skeleton plugin:

```
$ cd fecrutwitter
$ atlas-run
```

You'll need to wait a while for files to download.

Then point your browser to `http://localhost:3990/fecru/admin/viewplugins.do`, giving the administrator password 'password'.

You should see a list of plugins, including one named `fecrutwitter` which should be in the state `Enabled`. If you expand that plugin, you should see that it contains one module, described as 'A Sample Servlet Module'.

Now open the project in your IDE, according to these instructions

We are ready to start implementing our plugin.

First let's see what the example servlet included in the skeleton plugin does. Go to `http://localhost:3990/fecru/plugins/servlet/example-servlet` and you'll see the message `Hello from a servlet plugin`.

The Atlassian Plugin SDK allows you to reload you plugin without restarting FishEye/Crucible, which will speed up our work:

1. Open a new terminal window and `cd` to `fecrutwitter`
2. Run `atlas-cli`
3. Change the message in `ExampleServlet.java` and save the file
4. Type `pi` at the `atlas-cli` `maven2>` prompt
5. Refresh your browser and you should see the new message.

The page we are seeing doesn't look much like a user profile tab, so first let's tell FishEye/Crucible to decorate it as a user profile page tab. Add the following lines at the beginning of the servlet's `doGet` method:

```
request.setAttribute("decorator", "fisheye.userprofile.tab");
response.setContentType("text/html");
```

Now refresh the page. This page would normally be loaded in a dialog via Ajax, so it doesn't look quite right, but you can see the various profile tab links down the left hand side.
There is not yet a link for our tab, so you can't tell what tab we are on, and there's no link in the application to get to the tab. Now we will add a Web Item to fix those problems.

```
<web-item key="config-link" section="system.userprofile.tab">
   <link>/plugins/servlet/example-servlet</link>
   <label key="Twitter Configuration"/>
</web-item>
```

Install the plugin again with `pi`, log in as `admin/admin` and go to the profile page. You'll see the 'Twitter Configuration' link. Click on it, and you'll see the output of your servlet. The link isn't highlighted – that's because the decorator can't identify the link belonging to the active tab. We need to set the `profile.tab.key` attribute to the key of the Web Item:

```
response.getWriter().print(
  "<html><head>" +
  "<meta name='profile.tab.key' content='com.example.ampstutorial.fecrutwitter:config-link'/>" +
  "</head><body><div>Hello from a decorated servlet plugin " + new Date() +
  "</div></body></html>");
```

`com.example.ampstutorial.fecrutwitter:config-link` is the plugin key (from the `<atlassian-plugin>` element) and the key of the Web Item, separated by a colon.

Now the correct text should be highlighted when you click on the 'Twitter Configuration' link.

Your servlet can use a templating library to produce HTML. FishEye/Crucible provide some utility classes to help you use Velocity, but you could use other libraries too.

We'll rewrite our servlet to use Velocity:

```
public class ExampleServlet extends HttpServlet {
    private final TemplateRenderer templateRenderer;

    @Autowired
    public ExampleServlet(TemplateRenderer templateRenderer) {
        this.templateRenderer = templateRenderer;
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{
        request.setAttribute("decorator", "fisheye.userprofile.tab");
        response.setContentType("text/html");
        templateRenderer.render("config.vm", new HashMap<String,Object>(), response.getWriter());
    }
}
```

The template is created in our `src/resources` directory.

The last piece of the puzzle is getting the user's new preferences and persisting them.

We'll record the user's Twitter user name and password.

Our Velocity template looks like this:

```
#* @vtlvariable name="loginRecord" type="com.example.ampstutorial.TwitterLoginRecord" *#
<html>
<head>
    <meta name='profile.tab.key' content='com.example.ampstutorial.fecrutwitter:config-link'/>
</head>
<body>
<div>
    <form action="./example-servlet" method="post">
        <table class="dialog-prefs" cellspacing="0">
            <thead>
            <tr>
                <th colspan="2"><h3>Twitter Settings</h3></th>
            </tr>
            </thead>
            <tbody>
            <tr>
                <td class="tdLabel"><label for="userName" class="label">User Name:</label></td>
                <td><input type="text" name="userName" value="${loginRecord.userName}"
id="userName"/></td>
            </tr>
            <tr>
                <td class="tdLabel"><label for="password" class="label">Password:</label></td>
                <td><input type="password" name="password" value="${loginRecord.password}"
id="password"/></td>
            </tr>
            <tr>
                <td></td>
                <td class="action"><input type="submit" value="Save"/></td>
            </tr>
            </tbody>
        </table>
    </form>
</div>
</body>
</html>
```

and we add a `doPost` method to handle the form. Note that `doPost` redirects back to the servlet after processing the POST request:

```
public class ExampleServlet extends HttpServlet implements PluginIdAware {
    private final TemplateRenderer templateRenderer;
    private final ImpersonationService impersonationService;
    private final TwitterLoginRecordStore twitterLoginRecordStore;
    private PluginId pluginId;

    @Autowired
    public ExampleServlet(TemplateRenderer templateRenderer, ImpersonationService
impersonationService,TwitterLoginRecordStore twitterLoginRecordStore) {
        this.templateRenderer = templateRenderer;
        this.impersonationService = impersonationService;
        this.twitterLoginRecordStore = twitterLoginRecordStore;
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{
        request.setAttribute("decorator", "fisheye.userprofile.tab");
        response.setContentType("text/html");
        Map<String,Object> params = new HashMap<String,Object>();
        TwitterLoginRecord loginRecord = getLoginRecord();
        if (loginRecord == null) {
            loginRecord = new TwitterLoginRecord("","");
        }
        params.put("loginRecord", loginRecord);
        templateRenderer.render("config.vm", params, response.getWriter());
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        String newUserName = request.getParameter("userName");
        String newPassword = request.getParameter("password");
        if (newUserName != null && newPassword != null) {
            putLoginRecord(newUserName, newPassword);
        }
        response.sendRedirect("./example-servlet");
    }

    private void putLoginRecord(String newUserName, String newPassword) {
        twitterLoginRecordStore.storeByUsername(getCurrentUser().getUserName(), new
TwitterLoginRecord(newUserName, newPassword));
    }

    public void setPluginId(PluginId pluginId) {
        this.pluginId = pluginId;
    }

    private UserData getCurrentUser() {
        try {
            return impersonationService.getCurrentUser(pluginId);
        } catch (ServerException e) {
            throw new RuntimeException(e);
        }
    }

    private TwitterLoginRecord getLoginRecord() {
        return twitterLoginRecordStore.getByUsername(getCurrentUser().getUserName());
    }
}
```

For the purposes of this tutorial, the username and password are just stored in a `Map` in a field of the servlet – this is not persistent. To learn how to save configuration data for your plugins see Storing Plugin Settings.

To send a Twitter message when a commit is made, we need to run some of our plugin code when something internal happens in FishEye/Crucible. This is what the Event Listener Module Type is for.

An Event Listener must implement com.atlassian.event.EventListener.

Your implementation is added to `atlassian-plugin.xml`:

```
<listener key="commit-listener" class="com.example.ampstutorial.CommitListener"/>
```

Your implementation's `getHandledEventClasses` method must return an array of the event classes you wish to be notified of, in our example, just commit events. (here are other event types)

```
public class CommitListener implements EventListener {
...
    public Class[] getHandledEventClasses() {
        return new Class[] {CommitEvent.class};
    }
}
```

When a commit occurs our `handleEvent` method will be called:

```
public class CommitListener implements EventListener {
...
    public void handleEvent(Event event) {
        CommitEvent ce = (CommitEvent) event;
        ChangesetDataFE csData = revisionDataService.getChangeset(ce.getRepositoryName(),
ce.getChangeSetId());
        if (csData != null) {
            TwitterLoginRecord twitterLogin =
                        twitterLoginRecordStore.getByCommitterAndRepo(csData.getAuthor(),
ce.getRepositoryName());
            if (twitterLogin != null) {
                Twitter twitter = new
Twitter(twitterLogin.getUserName(),twitterLogin.getPassword());
                try {
                    twitter.updateStatus(csData.getComment());
                } catch (TwitterException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
...
}
```

The code above uses two services:

```
public class CommitListener implements EventListener {
    private final RevisionDataService revisionDataService;
    private final TwitterLoginRecordStore twitterLoginRecordStore;
    ...
}
```

These dependencies are both injected into our event listener module when it is created:

```
public class CommitListener implements EventListener {
    ...
    public CommitListener(RevisionDataService revisionDataService, TwitterLoginRecordStore
twitterLoginRecordStore) {
        this.revisionDataService = revisionDataService;
        this.twitterLoginRecordStore = twitterLoginRecordStore;
    }
    ...
}
```

The `RevisionDataService` is one of FishEye's standard services, but the `TwitterLoginRecordStore` is a component that we have created to abstract the means by which our servlet module stores login data and the means by which our event listener retrieves it.

When you create your own component plugin module you should create an interface and a class which implements it. The declaration of your component in `atlassian-plugin.xml` indicates the class to create an instance of, and the interface which other components should specify in order to be injected with the singleton:

```
<component key="twitterLoginRecordStore"
class="com.example.ampstutorial.TwitterLoginRecordStoreImpl">
        <interface>com.example.ampstutorial.TwitterLoginRecordStore</interface>
    </component>
```

Our Event Listener implementation also uses a third-party library, Twitter4J. This library is 'mavenised', that is, it appears in the default public maven repository, so we can simply declare a dependency on it in out plugin's `pom.xml`:

```
<dependency>
            <groupId>net.homeip.yusuke</groupId>
            <artifactId>twitter4j</artifactId>
            <version>2.0.9</version>
        </dependency>
```

If the library you wish to use is not available via Maven, and you don't run your own local maven repository, you can install the library jar into your local maven repository.

Gadget Tutorial

This tutorial will teach you how to write a simple gadget which can display information from Crucible on the JIRA dashboard. It uses the standard FishEye/Crucible REST interface to retrieve information. Other gadgets you write may need to use a REST module to retrieve the specific information they require in the most efficient way.

> The source for this tutorial is available from `https://svn.atlassian.com/svn/public/atlassian/crucible/plugins/gadget-tutorial/trunk`

To do this tutorial you will need an installation of JIRA 4.0 to act as the Gadget container.

First we'll write a trivial gadget which doesn't request any information from the gadget provider.

Declare the gadget in your `atlassian-plugin.xml` file:

```
<gadget key="trivial-gadget" location="gadgets/helloworld.xml"/>
```

Then create the gadget file at `src/main/resources/gadgets/helloworld.xml`. You can use any directory structure under `src/main/resources` for your gadgets.

The gadget specification, `helloworld.xml` contains this code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
    <ModulePrefs title="Crucible Tutorial Hello World Gadget" author_email="adent@example.com"
                 directory_title="Crucible Tutorial Hello World Gadget">
        <Require feature="minimessage"/>
        <Optional feature="dynamic-height"/>
    </ModulePrefs>
    <Content type="html">
<![CDATA[
        Hello,world!
]]>
    </Content>
</Module>
```

Install your plugin into FishEye/Crucible, then add it to JIRA thus:

1. Log in to JIRA as an administrator.
2. Select the Dashboards, Manage Dashboards menu option.
3. Click 'create new dashboard'
4. Type a name in the 'Name' field and click add. (defaults are fine for everything else)
5. Click on the name of your new dashboard and you'll be taken to an empty dashboard which will allow you to add gadgets.
6. Click on 'add a new gadget' and click the 'Add Gadget To Directory' button – you will only see this if you are a JIRA administrator.
7. Paste the URL `http://localhost:3990/fecru/rest/gadgets/1.0/g/com.example.crucible.gadget-tutorial:trivial-gadget` into the text field and click 'Add Gadget'.
8. Your gadget will appear, highlighted in yellow, in the gadget directory. Click 'Add it Now' and then 'Finished'.

The URL your gadget is served from depends on the plugin module key of the gadget module, which is composed of the artifact group and id you specified when you created your plugin project (you can see these in your `pom.xml`), and the key you gave your gadget module (which you can see in `atlassian-plugin.xml`).

So in the URL above, the section `com.example.crucible.gadget-tutorial:trivial-gadget` is composed of:

group id	com.example.crucible
artifact id	gadget-tutorial
module key	trivial-gadget

You should now see your trivial gadget on your JIRA dashboard:

To develop more complicated gadgets, see Gadget Development.

Gutter Renderer Plugin Tutorial

This tutorial teaches you how to add extra information to annotated views of files, and to diffs

The source code for this tutorial is available at `http://svn.atlassian.com/svn/public/atlassian/fisheye/plugins/gutter-tutorial/trunk`

Introduction To Gutter Renderers

A gutter renderer should not be performing time-consuming analysis of code at render time – if (for instance) you wished to perform style checking of code and report issues with a gutter renderer, the analysis would be best done either in an external system (e.g. your continuous integration server) or with an Event Listener. The stored results from the analysis would be used by the gutter renderer.

A gutter renderer plugin adds an extra column in the form of a `<td>` tag at the left hand side of the text of annotated file views and diff views.

The plugin controls the HTML `class` of the `td`, can give it a `style` (to avoid the need to provide a custom CSS file) and of course provides the HTML contained by the `td`.

Structure of a Gutter Renderer

Writing a gutter renderer requires that you create implementations of three interfaces:

- `GutterRenderer` – This interface is called once per file to be decorated.
  - It has two methods, both named `getAnnotationDecorators`.
  - One is called for annotated files, and returns a `List` of `LineDecorator`s.
  - The other is called when a diff is being displayed, and returns a `List` of `DiffLineDecorator`s.
  - If your implementation is called for a file/diff it doesn't wish to decorate then it must return an empty list.
  - Of course you can return different concrete implementations of the decorator classes for different files/diffs.

- `LineDecorator` – The implementation returned by `getAnnotationDecorators` has its `decorateLine` method called once for each line in the file. It must return a `GutterRenderResult` instance.
- `DiffLineDecorator` – The implementation returned by `getAnnotationDecorators` has one of its `decorateLine/Added/Changed/Common/Removed` methods called for each line of the diff. The method called depends on the type of changes which occurred in that line. These methods all return a `GutterRenderResult` instance.

A Sample Gutter Renderer

The gutter renderer we develop in this tutorial will simply display some text on the *n*th line of each file whose name contains the letter 'a', where *n* is the number of characters in the name. It will not provide any decoration for diffs.

The `GutterRenderer` implementation looks like this:

```
public class TutorialGutterRenderer implements GutterRenderer {

    public List<LineDecorator> getAnnotationDecorators(String repository, String path, String rev)
{
        String fileName = path.substring(path.lastIndexOf('/')+1);
        if (fileName.matches(".*[aA].*")) {
            return Collections.<LineDecorator>singletonList(new
TutorialLineDecorator(fileName.length()));
        } else {
            return Collections.emptyList();
        }
    }

    public List<DiffLineDecorator> getAnnotationDecorators(DiffType diffType, String repository,
String fromPath, String fromRev, String toPath, String toRev) {
        return Collections.emptyList();
    }
}
```

and the `LineDecorator` implementation looks like this:

```
public class TutorialLineDecorator implements LineDecorator {
    private final int lineToDecorate;

    public TutorialLineDecorator(int lineToDecorate) {
        this.lineToDecorate = lineToDecorate;
    }

    public int getWidth() {
        return 10; // the width required for this gutter column, in ems
    }

    public GutterRenderResult decorateLine(int i) {
        GutterRenderResult r = new GutterRenderResult();
        if (i == lineToDecorate) {
            r.setInnerHtml("Decoration!");
        }
        return r;
    }
}
```

Rendering a Velocity Template from Your Servlet

Velocity allows your Servlet Plugins to render HTML pages from simple templates. This tutorial briefly shows the basics of using velocity. The FishEye Twitter Integration Plugin Tutorial shows velocity being used to generate a form for maintaining user preferences as part of a larger plugin.

Velocity templates are placed in the resources part of your plugin project, i.e. `myplugin/src/main/resources/mytemplate.vm`.

Your servlet uses Velocity like this:

```
public class MyServlet extends HttpServlet {
    private final VelocityHelper velocityHelper;

    @Autowired
    public MyServlet(VelocityHelper velocityHelper) {
        this.velocityHelper = velocityHelper;
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
{
        Map<String,Object> params = new HashMap<String,Object>();

        params.put("first", "Hello");
        params.put("second", "World");
        velocityHelper.renderVelocityTemplate("mytemplate.vm", params, response.getWriter());
    }
}
```

where `mytemplate.vm` contains:

```
<html>
<head>
</head>
<body>
  ${first}, ${second}.
</body>
</html>
```

In order to render the text "Hello, World.".

REST Service Plugin Module Tutorial

This tutorial teaches you how to provide your own REST API. This is useful when your external application's use cases don't fit well with the FishEye/Crucible REST API, resulting in too many round trips.

Here's a more detailed tutorial – it uses JIRA as the application which will host the plugin, but the tutorial also applies to FishEye/Crucible.

Suppose we want to find the set of users who have not completed all their open reviews. Using the REST API we need to make one request to get the list of all open reviews, then for each review we must as for a list of uncompleted reviewers. This results in many HTTP requests, and much data transfer. If we calculate the set on the server using the `ReviewService`, we can retrieve it in a single call.

> The source code for this tutorial is available at `http://svn.atlassian.com/svn/public/atlassian/crucible/plugins/rest-service-tutorial/trunk`

Our REST interface is a Java class with annotations which define which methods service which REST URLs.

Plugins which include REST Service modules need to add a dependency on `jsr311` to their pom, to provide the REST annotations:

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>jsr311-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
```

We'll start with a very simple class which just returns the string "Hello World" when we make a GET request to the URL `http://localhost:3990/fecru/rest/completion-status/1.0/users`.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public class RestCompletionStatusService {
    @GET
    @Path("users")
    public Response getUncompletedUsers() {
        return Response.ok("Hello World").build();
    }
}
```

Add this module to `atlassian-plugin.xml`:

```
<rest key="completion-status-rest" path="/completion-status" version="1.0">
    <description>Review completion information</description>
</rest>
```

Go to the URL above in your browser, and you should (as long as you are logged in) get the XML:

```
<pair><first>Hello</first><second>World</second></pair>
```

Now we'll make the implementation less trivial.

```
@Path("/")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public class RestCompletionStatusService {
    private final ReviewService reviewService;

    public RestCompletionStatusService(ReviewService reviewService) {
        this.reviewService = reviewService;
    }

    // We can't return a naked Collection, it must be wrapped in a container
    @XmlRootElement
    public static class Reviewers {
        public Reviewers(Collection<ReviewerData> reviewer) {
            this.reviewer = reviewer;
        }

        // Needed by JAXB
        private Reviewers() {
        }

        // Must be public. By naming this field 'reviewer' each element in the Collection is
        // placed inside a '<reviewer>' element in the XML
        public Collection<ReviewerData> reviewer;
    }

    @GET
    @Path("users")
    public Response getUncompletedUsers() {
        Set<ReviewerData> users = new HashSet<ReviewerData>();
        for (ReviewData reviewData : reviewService.getFilteredReviews("allOpenReviews",false)) {
            System.out.println(reviewData.getPermaId());
            users.addAll(reviewService.getUncompletedReviewers(reviewData.getPermaId()));
        }
        return Response.ok(new Reviewers(users)).build();
    }
}
```

This version uses the `ReviewService` to retrieve all open reviews, and then accumulate their incomplete reviewers in a `Set`.

The returned XML looks like:

```
<reviewers>
  <reviewer>
    <displayName>Jeffrey</displayName>
    <userName>jeffrey</userName>
    <completed>false</completed>
  </reviewer>
  <reviewer>
    <displayName>Jim Jones</displayName>
    <userName>jim</userName>
    <completed>false</completed>
  </reviewer>
</reviewers>
```

Storing Plugin Settings

Overview

This tutorial demonstrates how to use SAL (Shared Access Layer) to let your plugin store its configuration settings. SAL offers a product independent way to store and retrieve settings.

Configuring your Plugin

When you are developing against FishEye/Crucible 2.1.x, the SAL library is already configured as a maven dependency, which means you won't need to configure anything when you use an IDE with maven integration.

Prior to FishEye/Crucibe 2.1, the SAL library did not come as a pre-configured maven dependency and you will need to manually add it to your `pom.xml` file:

pom.xml

```
<dependency>
    <!-- This dependency is only required for FishEye/Crucible 2.0.x, NOT 2.1 or later. -->
    <groupId>com.atlassian.sal</groupId>
    <artifactId>sal-api</artifactId>
    <version>2.0.6</version>
    <scope>provided</scope>
</dependency>
```

Next thing to do is to import the SAL service we are going to use by declaring it in our `atlassian-plugin.xml` file. SAL services that are not explicitly declared will not be available:

atlassian-plugin.xml

```
<component-import key="pluginSettingsFactory"
interface="com.atlassian.sal.api.pluginsettings.PluginSettingsFactory" />
```

Data Storage and Retrieval

With SAL configured, you can now get FishEye/Crucible to inject the `com.atlassian.sal.api.pluginsettings.PluginSettingsFactory` into your plugin modules. This factory gives access to plugin settings.

In FishEye/Crucible, plugin settings are linked to repositories. This allows plugins to use different settings per repository. Per-repository settings are accessed through the `PluginSettingsFactory.createSettingsForKey(String repoKey)` method.

In addition to the per-repository settings there is a global settings pool where your plugin can store settings that are not repository dependent. This is used by the example below:

```
package com.atlassian.contrib;

import com.atlassian.sal.api.pluginsettings.PluginSettingsFactory;
import com.atlassian.sal.api.pluginsettings.PluginSettings;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.util.List;
import java.util.ArrayList;

public class ExampleServlet extends HttpServlet {

    private final PluginSettingsFactory settingsFactory;

    public ExampleServlet(PluginSettingsFactory settingsFactory) {
        this.settingsFactory = settingsFactory;
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        PluginSettings settings = settingsFactory.createGlobalSettings();
        String value1 = (String)settings.get("myPlugin.value1");
        List<String> value2 = (List<String>) settings.get("myPlugin.value2");

        ...
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        PluginSettings settings = settingsFactory.createGlobalSettings();
        settings.put("myPlugin.value1", "foo");

        List<String> value2 = new ArrayList<String>();
        value2.add("foo");
        value2.add("bar");
        settings.put("myPlugin.value2", value2);

        ...
    }
}
```

SAL's `PluginSettings` service supports the following value types:

- `java.lang.String` – stored as-is
- `java.util.List` – calls `toString()` on the list items
- `java.util.Map` – calls `toString()` on both the keys and values in the Map
- `java.util.Properties` – uses `Properties.store()` to let the `Properties` object serialize itself

Limitations

SAL's `PluginSettings` mechanism is suitable for storing simple string-based key/value pairs, but should generally not be used to store large amounts of data. FishEye's SAL implementation stores your plugin's settings as strings in the application's `config.xml` file. While it would be technically possible to store a large chunk of binary data through this service (for instance by base-64 encoding the data in a large string), this can easily cause memory problems at runtime, as the entire config.xml file is kept in memory by FishEye/Crucible.

Plugin configuration properties are not isolated from other plugins, but all share the same storage area. As a result, your plugin settings could be retrieved by other plugins if they knew which keys you are using.
Consequently, you must be careful not to accidentally overwrite other plugins' settings by using the same property keys. Instead, always prefix your configuration keys with a unique string. Your plugin's unique key string makes a good prefix.

Using Logging From Your Plugin

This tutorial describes how to log messages from your plugin. Logging allows you to write information to the FishEye/Crucible log file when your plugin encounters an error, which will help you diagnose problems with your plugin.

Messages may be written at a `DEBUG` level – these message only appear when Debug Logging is turned on on the Admin, Server Settings page. Messages written at `INFO`, `WARNING` and `ERROR` levels always appear in the log.

You can log anywhere in your plugin code. Just add a snippet of code like this:

```
...
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
private static final Logger LOG = LoggerFactory.getLogger("atlassian.plugin");
...
LOG.error("Something bad happened");
LOG.debug("Information useful during development");
LOG.info("Information you wish your plugin administrators to view");
```

The logged output will appear in `target/fecru/home/var/log/fisheye-debug.log.YYYY.MM.DD`.

Authentication Plugins

> Authentication Plugins do not use the Atlassian Plugins framework – they are simply a `jar` added to FishEye's classpath.

To implement an arbitrary form of authentication and authorisation for FishEye you need to provide a class which extends `com.cenqua.fisheye.user.plugin.AbstractFishEyeAuthenticator`. You can find more information about custom FishEye authorisation in the online javadocs and the library jar.

For FishEye to use the authenticator, it must be compiled, placed in a jar archive and then put in the `$FISHEYE_INST/lib` directory. If other third-party libraries are required by your authenticator, they must also be in the `$FISHEYE_INST/lib` directory.

Global Configuration

After implementing a custom authenticator, the next step is to configure FishEye to use it.

Click the '**Setup Custom authentication**' link on the FishEye '**Admin**' -> '**Users/Security**' page.

You will be presented with a form containing the following fields to be set:

Classname	The fully qualified class name of your AbstractFishEyeAuthenticator, e.g. `com.cenqua.fisheye.user.plugin.ExampleFishEyeAuthenticator`.
Cache TTL (positive)	How long FishEye should cache permission checks. Example values are: `0 secs`, `5 mins`.
Auto-add	FishEye can automatically create a user it has not previously encountered if the user can successfully authenticate against your authenticator.

Properties	Any properties your authenticator requires. These will be passed to its `init()` method. This field should comply with the java.util.Properties format. Example:
	`# comments`
	`name1=value1`
	`name2=value2`

Per-Repository Constraint Configuration

You may also require a per-repository constraint to restrict access to specific repositories using your custom authenticator. If a custom authenticator is set, then the Permissions Summary table will display the constraint per repository and a link to enable you to edit it.

> The '**Authentication Test**' page allows you to enter a user's credentials and to test the user's authentication. It will also test which repositories the user is authorised to access.

Plugin Module Types

What is a Module Type?

Atlassian products each define a set of **plugin points**, where plugins can add functionality. Each **module type** provides functionality to a different **plugin point**. For example, the event system will send events to all the modules of type 'Event Listener' in all the plugins loaded by FishEye/Crucible.

Module Types Supported in FishEye/Crucible

- Downloadable Plugin Resources — If your plugin requires FishEye/Crucible to include additional static files such as images, you will need to use downloadable plugin resources to make them available.
- Event Listener Module Type — An event listener plugin module is an object which is notified when certain internal Crucible or FishEye events occur.
- Gadget Module Type — A Gadget module allows FishEye/Crucible to serve your `gadget.xml` file. See Gadget Development for more details on developing gadgets.
- Gutter Renderer Module Type — The Gutter Renderer plugin point allows you to display custom content in the margin of your source code view.
- REST Module Type — You can use the REST module type to expose services and data entities as REST APIs
- SCM Module Type — An SCM plugin module gives Crucible the ability to work with a custom version control system that is not supported out of the box.
- Servlet Module Type — Servlet plugin modules enable you to deploy Java servlets as a part of your plugins.
- Spring Component Module Type — Component plugin modules enable you to share Java components between other modules in your plugin and optionally with other plugins in the application.
- Web Item Module Type — Web UI plugin modules allow you to add links, tabs and sections of links to the Fisheye user interface. By adding a link to a servlet plugin, you can add your own pages to the UI.
- Web Resources — If your plugin requires FishEye/Crucible to include additional static Javascript or CSS files, you will need to use downloadable web resources to make them available.

Plugin Resources

If your plugins uses static content like images, Javascript or CSS, you make these available as resources.

- Downloadable Plugin Resources – resources you refer to directly in HTML your plugin serves, usually images.
- Web Resources – Javascript and CSS files which are included in your page's `<HEAD>`.

Authentication Plugins

You can also write a plugin to allow FishEye/Crucible to use other authentication methods. These do **not** use the Atlassian Plugins framework.
Writing Authentication Plugins.

Downloadable Plugin Resources

FishEye/Crucible plugins may define downloadable resources. If your plugin requires FishEye/Crucible to include additional static files such as images, you will need to use downloadable plugin resources to make them available. You can also serve Javascript and CSS files as downloadable resources, but we recommend that you use Web Resources instead. Web Resources are included in the header of a page and can take advantage of caching (i.e. only download a resource if it has changed) and batching (i.e. serve multiple files in one request).

Please see the Web Resources documentation, if you want to include Javascript or CSS files for download from the header of a page.

Defining a Single Downloadable Resource

Downloadable resources are configured to map a name of some downloadable file to its location within the plugin jar-file.

```
<resource type="download" name="icn_auto_fav.gif" location="icn_auto_fav.gif"/>
```

- Resources can be downloaded either within a plugin module, or as a resource of the entire plugin.
    - Resources are always looked up relative to a plugin module (see below). If a resource can not be found in the plugin module, it will then be looked for in the plugin itself.
- Each resource must be of `type="download"`
- The name of the resource is how it will be referenced from within the application
- The location of the resource is where it appears within the plugin itself
- An optional `content-type` parameter can be used to supply the file's MIME type
    - In the absence of a `content-type`, the application will attempt to guess the file's type from its file extension. For common file extensions, an explicit content-type is not necessary.

Defining a Directory of Downloadable Resources

If your plugin requires a lot of resources, you may wish to expose a directory of files as resources, rather than writing definitions for each individual file.

```
<resource type="download" name="icons/" location="templates/extra/autofavourite/icons/"/>
```

- The name and location must both have trailing slashes
- Subdirectories are also exposed, so in the example above, `icons/small/icn_auto_fav.gif` will be mapped to the resource `templates/extra/autofavourite/icons/small/icn_auto_fav.gif`

Referring to Downloadable Resources

Once declared, the URL for a downloadable resource is as follows:
`{server root}/download/resources/{plugin key}:{module key}/{resource name}`

`{module key}` is optional.

For example:

`http://fisheye.example.com/download/resources/com.atlassian.fisheye.plugin.autofavourite:autofavourite-r`

Event Listener Module Type

An event listener plugin module is an object which is notified when certain internal Crucible or FishEye events occur.

To include an event listener module add a `listener` element to your `atlassian-plugins.xml` file:

```
<listener key="example-listener"
class="com.atlassian.crucible.example.plugin.spring.ExampleListener"/>
```

and create a class which implements `com.atlassian.event.EventListener`. See the FishEye event and Crucible event javadoc for specific event types. See the javadoc for EventListener to understand the general details regarding events.

For example, if we want to listen for all events, and print a message to standard output we would write:

```
public class ExampleListener implements EventListener {
    public void handleEvent(Event event) {
        System.out.println("Got event: " + event);
    }

    public Class[] getHandledEventClasses() {
        return new Class[0];
    }
}
```

Event listeners may implement `StateAware` if they need to be notified when the module is enabled or disabled.

Gadget Module Type

A Gadget module allows FishEye/Crucible to serve your `gadget.xml` file. See Gadget Development for more details on developing gadgets.

This page tells you how to add your gadget to an Atlassian application (JIRA, Confluence, etc) as a plugin.

In short: Add a `<gadget>` module type to your `atlassian-plugin.xml` file.

On this page:

- Standalone Gadget or Plugin?
- Embedding your Gadget into a Plugin
  - Prerequisites
  - Purpose of the Gadget Module Type
  - Configuration
  - Example
  - URL for Published Gadgets

Standalone Gadget or Plugin?

How feasible is it to create an Atlassian gadget that consists purely of HTML, CSS and Javascript? Assuming that the Confluence or JIRA (or whatever) REST API could do everything that the gadget needs, can you bypass the plugin side of things altogether?

You can choose to write a **standalone** gadget or a gadget embedded in a **plugin**.

- A standalone gadget consists entirely of HTML, CSS and Javascript, all contained within the gadget XML file. To retrieve data from an application, you can use a REST API, for example.
- Alternatively you can write a gadget and a plugin, and embed the gadget into the plugin XML using the gadget plugin module. The plugin needs to be installed into the Atlassian application, such as JIRA or Confluence.

Limitations if you do not embed your gadget in a plugin:

- You will not be able to use #-directives. This means that, if your gadget is requesting data from an Atlassian application, the gadget can only access a single, specific instance of that application because you will need to hard-code the base URL.
- It is significantly more difficult to use the Atlassian Gadget JavaScript Framework, because you will need to code everything that is normally automatically generated by the `#supportedLocales` and `#includeResources` directives.

That said, there are two main reasons why you may be interested in writing standalone gadgets:

- It is a much simpler way to write very basic gadgets. This provides an easier way to learn how to write gadgets.
- A non-plugin gadget may be sufficient if your gadget is for your own company's internal use with a single Atlassian application/site.
- You may want to write gadgets that request data from some non-Atlassian system or web service, in order to integrate that data into an Atlassian application such as JIRA or Confluence.

Embedding your Gadget into a Plugin

The rest of this page describes the gadget module type that you will use to embed your gadget within an Atlassian plugin.

Prerequisites

- Your version of the Atlassian application must support gadgets. See the gadget version compatibility matrix.
- Your version of the Atlassian application must support the Atlassian Plugin Framework version 2.2 or later. See the plugin framework version compatibility matrix.
- Your plugin must be an OSGi-based plugin, as supported by the Atlassian Plugin Framework.
- You will need to install the REST plugin module type, if not already bundled with the application.

Purpose of the Gadget Module Type

Gadget plugin modules enable you to add your gadget to an Atlassian application (JIRA, Confluence, etc) as a plugin. Your gadget can then make use of the application's remote API to fetch data and interact with the application.

Configuration

The element for the Gadget plugin module is **`gadget`**. It allows the following attributes for configuration:

Attributes

Name	Required	Description	Default
key	✔	The `key` attribute is a standard module key, so it is required and must be unique within the plugin across **all** module types. Atlassian Gadgets does not use this key for anything special, so you can choose any key you like.	
location	✔	The `location` attribute can be either the relative path to a resource within the plugin, or the absolute URL of an externally-hosted gadget.	

Example

The syntax of the module type is:

URL for Published Gadgets

Gadgets published by an Atlassian container (such as JIRA or Confluence) are provided by the REST plugin module built into the Atlassian Gadgets framework. The URL of published gadgets has the following format — with context:

Or without context:

Example:

RELATED TOPICS

Creating your Gadget XML Specification
Writing an Atlassian Gadget
Overview of REST Implementation using the REST Plugin Module

Gutter Renderer Module Type

On this page:

- Purpose of this Module Type
- Configuration
  - Attributes
  - Elements
- Example
- Notes

Purpose of this Module Type

The Gutter Renderer plugin point allows you to display custom content in the margin of your source code view. The content is generated per line.

Configuration

The root element for the Gutter Renderer plugin module is **`gutter-renderer`**. It allows the following attributes and child elements for configuration:

Attributes

Name	Required	Description	Default
class	✓	The Gutter Renderer module Java class. Must implement `com.atlassian.fisheye.ui.filedecoration.GutterRenderer`.	N/A
key	✓	The identifier of the Gutter Renderer module.	N/A

Elements

Name	Required	Description	Default
description		The description of the Gutter Renderer module.	

Example

Here is a sample atlassian-plugin.xml fragment for a web section:

```
<gutter-renderer key="my-gutter-renderer" class="com.atlassian.example.plugin.MyGutterRenderer"/>
```

Notes

See the Gutter Renderer Plugin Tutorial to learn how to develop an Gutter Renderer module plugin.

REST Module Type

You can use the REST plugin module to create plugin points easily in Atlassian applications, by exposing services and data entities as REST APIs. The Atlassian REST plugin module is bundled with our applications.

REST APIs provide access to resources via URI paths. To use a REST API, your plugin or script will make an HTTP request and parse the response. You can choose JSON or XML for the response format. Your methods will be the standard HTTP methods like GET, PUT, POST and DELETE. Because the REST API is based on open standards, you can use any web development language to access the API.

> **Plugin Framework 2 Only**
> The REST plugin module described below is available only for OSGi-based plugins using version 2.2 or later of the Atlassian Plugin Framework.

On this page:

- Purpose of the REST Plugin Module
- Configuration
  - Attributes
  - Elements

- Example
- Notes on REST API and Plugin Development
  - JAX-RS and Jersey
  - Including the REST Plugin Module into your Project
  - Developing your REST API as an Atlassian Plugin
  - Accessing your REST Resources
  - How the REST Plugin Module Finds your Providers and Resources
  - Easy Way to Request JSON or XML Response Format
  - Working with JSON in Firefox
  - Differences in JSON and XML rendering of REST responses
- Help, I'm still stuck!
- References

Purpose of the REST Plugin Module

REST plugin modules enable you to expose services and data as REST resources.

Configuration

The root element for the REST plugin module is **`rest`**. It allows the following attributes and child elements for configuration.

Attributes

Name	Required	Description	Default
key	✓	The identifier of the plugin module, i.e. the identifier of the REST module. This key must be unique within the plugin where it is defined. Sometimes you will need to uniquely identify a module. Do this with the **complete module key**. A module with key `fred` in a plugin with key `com.example.modules` will have a complete key of `com.example.modules:fred`.	N/A
path	✓	The path to the REST API exposed by this module. For example, if set to `/foo`, the REST API will be available at `http://localhost:8080/context/rest/foo/1.0`, where `1.0` is the version of the REST API.	N/A
version	✓	This is the version of the REST API. This is not the same thing as the plugin version. Different versions of the same API can be provided by different plugins. Version numbers follow the same pattern as OSGi versions, i.e. `major.minor.micro.qualifier` where `major`, `minor` and `micro` are integers.	N/A

Elements

Name	Required	Description	Default
description		The description of the plugin module, i.e. the description of the REST module. The 'key' attribute can be specified to declare a localisation key for the value instead of text in the element body.	N/A
package		The package from which to start scanning for resources and providers. Can be specified multiple times. Defaults to scanning the whole plugin. **Since 2.0**	N/A
dispatcher		Determines when the filter is triggered. You can include multiple `dispatcher` elements. If this element is present, its content must be one of the following: `REQUEST`, `INCLUDE`, `FORWARD`, `ERROR`. Note: This element is only available in Plugin Framework 2.5 and later. If this element is not present, the filter will be fired on all conditions. (This is also the behaviour for Plugin Framework releases earlier than 2.5.)	Filter will be triggered on all conditions

Example

Here is an example `atlassian-plugin.xml` file containing a single public component:

Notes on REST API and Plugin Development

JAX-RS and Jersey

Here is some information to be aware of when developing a REST plugin module:

- The REST module is based on JAX-RS. Specifically, it uses Jersey, which is the JAX-RS reference implementation.
- The REST module is based on **version 1.0.2 of Jersey**.
- Jersey is configured to use all its default providers, but you should be able to change the configuration by adding your own providers in your plugin. For example, you may want to enhance your JSON.

See Overview of REST Implementation using the REST Plugin Module.

Including the REST Plugin Module into your Project

You can include the REST plugin module as a Maven dependency from our Maven repository.

Developing your REST API as an Atlassian Plugin

To develop a REST API and deploy it into an Atlassian application, you will follow the same process as for any other Jersey application:

1. Develop your JAX-RS resources, using the annotations `@Path`, `@Provider`, etc.
2. Bundle your resource classes in your plugin JAR file, along with the plugin descriptor `atlassian-plugin.xml`.
3. Deploy your plugin to the application.

See Developing your Plugin using the Atlassian Plugin SDK and Creating your Plugin Descriptor.

Accessing your REST Resources

Your REST resources will be available at this URL:

- `host` and `port` define the host and port where the application lives.
- `context` is the servlet context of the application. For example, for Confluence this would typically be `confluence`.
- `helloworld` is the path declared in the REST module type in the plugin descriptor.
- `1.0` is the API version.

If `1.0` is the latest version of the *helloworld* API installed, this version will also be available at this URL:

How the REST Plugin Module Finds your Providers and Resources

The REST plugin module scans your plugin for classes annotated with the `@Provider` and `@Path` annotation. The `@Path` annotations can be simply declared on a method within a class and do not need to be present at the entity level.

For those not familiar with the JAX-RS specification, the @Path annotation can be declared at a package, class, or method level. Furthermore, their effects are cumulative. For example, if you define this at the package level:

then this at the class level:

then this at the method level:

The final URL would be for the helloWorld plugin above:

The REST plugin module will not scan a library bundled with your plugin, typically bundled in the `META-INF/lib` directory of the JAR. So make sure you put all providers and resources at the top level in your plugin JAR.

Only one resource method can be bound to the root "/" path.

Easy Way to Request JSON or XML Response Format

When using JAX-RS (and Jersey), the standard way to specify the content type of the response is to use the HTTP Accept header. While this is a good solution, it not always convenient.

The REST plugin module allows you to use an extension to the resource name in the URL when requesting JSON or XML.

For example, let's assume I want to request JSON data for the resource at this address:

I have two options:

- **Option 1:** Use the HTTP Accept header set to `application/json`.
- **Option 2:** Simply request the resource via this URL:

If I want content of type `application/xml`, I will simply change the extension to `.xml`. Currently the REST plugin module supports only `application/json` and `application/xml`.

Working with JSON in Firefox

A handy tool: The JSONView add-on for Firefox allows you to view JSON documents in the browser, with syntax highlighting. Here is a link to the Firefox add-on page.

Differences in JSON and XML rendering of REST responses

By default, JSON responses include **any** object fields which are explicitly annotated with a JAXB annotation, while XML responses include public fields and fields with public getters.

To get the same behaviour for JSON you need to either annotate each field with `@XmlElement` or `@XmlAttribute`, or annotate the class or package with `@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)`.

Help, I'm still stuck!

Start with [DEVNET:General Development Tips] and then proceed to [DEVNET:Plugin Development Tips, FAQ and Troubleshooting]

References

- JAX-RS (JSR311) home page
  - 1.0 Spec (pdf)
  - 1.0 API
- Jersey home page
  - Wiki
  - 1.0.2 API (implements JAX-RS 1.0)
  - Configuring JSON for RESTful Web Services in Jersey 1.0

RELATED TOPICS

Plugin Tutorial - Writing REST Services
Developing your Plugin using the Atlassian Plugin SDK
Overview of REST Implementation using the REST Plugin Module
Atlassian REST API Design Guidelines version 1
Guidelines for Atlassian REST API Design
Atlassian Plugin Framework Documentation

SCM Module Type

On this page:

- Purpose of this Module Type
- Configuration
  - Attributes
  - Elements
- Example
- Notes

Purpose of this Module Type

An SCM plugin module gives Crucible the ability to work with a custom version control system that is not supported out of the box.

Configuration

The root element for the SCM plugin module is **`scm`**. It allows the following attributes and child elements for configuration:

Attributes

Name	Required	Description	Default
class	✔	The SCM module Java class. Must implement `com.atlassian.crucible.scm.SCMModule`.	N/A
key	✔	The identifier of the SCM module.	N/A
name		The human-readable name of the SCM module.	The plugin key.

Elements

Name	Required	Description	Default
description		The description of the SCM module.	

Example

Here's a complete `atlassian-plugin.xml` containing an SCM module.

Note that an SCM module is very likely to specify a `configure.url` and to contain a configuration servlet.

```
<atlassian-plugin key="${atlassian.plugin.key}" name="crucible-git-scm-plugin" pluginsVersion="2">
    <plugin-info>
        <description>An SCM provider for Git</description>
        <vendor name="Atlassian Software Systems" url="http://www.atlassian.com"/>
        <version>1.0-SNAPSHOT</version>
        <param name="configure.url">/plugins/servlet/gitscm</param>
    </plugin-info>

    <scm name="Git Light SCM" key="scmprovider"
class="com.atlassian.crucible.plugins.scm.git.GitSCMModule">
        <description>SCM implementation for Git</description>
    </scm>

    <servlet name="Git Configuration Servlet" key="configservlet"
             class="com.atlassian.crucible.plugins.scm.git.GitConfigServlet" adminLevel="system">
        <description>Allows Configuration of Git SCM Plugin</description>
        <url-pattern>/gitscm</url-pattern>
    </servlet>
</atlassian-plugin>
```

Notes

See the Crucible SCM Plugin Tutorial to learn how to develop an SCM module plugin.

Servlet Module Type

In FishEye/Crucible, servlets which should only be accessible to users with administration privileges must be given an `adminLevel="system"` attribute on the `<servlet>` tag. i.e.:

```
<servlet name="My Servlet" key="foo" class="com.example.AdminServlet"
adminLevel="system">
...
</servlet>
```

On this page:

- Purpose of this Module Type
- Configuration
  - Attributes
  - Elements
- Example
- Accessing your Servlet
- Notes

Purpose of this Module Type

Servlet plugin modules enable you to deploy Java servlets as a part of your plugins.

Configuration

The root element for the Servlet plugin module is **`servlet`**. It allows the following attributes and child elements for configuration:

Attributes

Name	Required	Description	Default
class	✔	The servlet Java class. Must be a subclass of `javax.servlet.http.HttpServlet`. See the plugin framework guide to creating plugin module instances.	
disabled		Indicate whether the plugin module should be disabled by default (value='true') or enabled by default (value='false').	false
i18n-name-key		The localisation key for the human-readable name of the plugin module.	

key	✓	The identifier of the plugin module. This key must be unique within the plugin where it is defined. Sometimes, in other contexts, you may need to uniquely identify a module. Do this with the **complete module key**. A module with key `fred` in a plugin with key `com.example.modules` will have a complete key of `com.example.modules:fred`. I.e. the identifier of the servlet.	N/A
name		The human-readable name of the plugin module. I.e. the human-readable name of the servlet.	The plugin key.
system		Indicates whether this plugin module is a system plugin module (value='true') or not (value='false'). Only available for non-OSGi plugins.	false

Elements

Name	Required	Description	Default
description		The description of the plugin module. The 'key' attribute can be specified to declare a localisation key for the value instead of text in the element body. I.e. the description of the servlet.	
init-param		Initialisation parameters for the servlet, specified using `param-name` and `param-value` sub-elements, just as in `web.xml`. This element and its child elements may be repeated.	N/A
resource		A resource for this plugin module. This element may be repeated. A 'resource' is a non-Java file that a plugin may need in order to operate. Refer to Adding Plugin and Module Resources for details on defining a resource.	N/A
url-pattern	✓	The pattern of the URL to match. This element may be repeated. The URL pattern format is used in Atlassian plugin types to map them to URLs. On the whole, the pattern rules are consistent with those defined in the Servlet 2.3 API. The following wildcards are supported: • * matches zero or many characters, including directory slashes • ? matches zero or one character ***Examples*** • `/mydir/*` matches `/mydir/myfile.xml` • `/*/admin/*.??ml` matches `/mydir/otherdir/admin/myfile.html`	N/A

Example

Here is an example `atlassian-plugin.xml` file containing a single servlet:

Accessing your Servlet

Your servlet will be accessible within the Atlassian web application via each `url-pattern` you specify, beneath the `/plugins/servlet` parent path.

For example, if you specify a `url-pattern` of `/helloworld` as above, and your Atlassian application was deployed at http://yourserver/jira — then your servlet would be accessed at http://yourserver/jira/plugins/servlet/helloworld .

Notes

Some information to be aware of when developing or configuring a Servlet plugin module:

- Your servlet's `init()` method will not be called on web application startup, as for a normal servlet. Instead, this method will be called the first time your servlet is accessed after each time it is enabled. This means that if you disable a plugin containing a servlet, or a single servlet module, and re-enable it again, the servlet is re-instantiated and its `init()` method will be called again.
- Because all servlet modules are deployed beneath a common `/plugins/servlet` root, be careful when choosing each `url-pattern` under which your servlet is deployed. It is recommended to use a value that will always be unique to the world!

RELATED TOPICS

Information sourced from Plugin Framework documentation

Spring Component Module Type

On this page:

- Purpose of this Module Type

- Configuration
  - Attributes
  - Elements
- Example
- Notes

Purpose of this Module Type

Component plugin modules enable you to share Java components between other modules in your plugin and optionally with other plugins in the application.

Configuration

The root element for the Component plugin module is **`component`**. It allows the following attributes and child elements for configuration:

Attributes

Name	Required	Description	Default
alias		The alias to use for the component when registering it in the internal bean factory.	The plugin key
class		The class which implements this plugin module. The class you need to provide depends on the module type. For example, Confluence theme, layout and colour-scheme modules can use classes already provided in Confluence. So you can write a theme-plugin without any Java code. But for macro and listener modules you need to write your own implementing class and include it in your plugin. See the plugin framework guide to creating plugin module instances. The Java class of the component. This does not need to extend or implement any class or interface.	
key	✓	The identifier of the plugin module. This key must be unique within the plugin where it is defined. Sometimes, in other contexts, you may need to uniquely identify a module. Do this with the **complete module key**. A module with key `fred` in a plugin with key `com.example.modules` will have a complete key of `com.example.modules:fred`. I.e. the identifier of the component.	N/A
i18n-name-key		The localisation key for the human-readable name of the plugin module.	
name		The human-readable name of the plugin module. I.e. the human-readable name of the component.	The plugin key.
public		Indicates whether this component should be made available to other plugins via the Component Import Plugin Module or not.	false
system		Indicates whether this plugin module is a system plugin module (value='true') or not (value='false'). Only available for non-OSGi plugins.	false

Elements

Name	Required	Description	Default
interface		The Java interface under which this component should be registered. This element can appear zero or more times.	N/A
description		The description of the plugin module. The 'key' attribute can be specified to declare a localisation key for the value instead of text in the element body.	
service-properties		Map of simple properties to associate with a public component (Plugin Framework 2.3 and later). Child elements are named `entry` and have `key` and `value` attributes.	

Example

Here is an example `atlassian-plugin.xml` file containing a single public component:

Here is an example public component with several service properties:

Notes

Some information to be aware of when developing or configuring a Component plugin module:

- Components, at installation time, are used to generate the `atlassian-plugins-spring.xml` Spring Framework configuration file, transforming Component plugin modules into Spring bean definitions. The generated file is stored in a temporary plugin jar and installed into the framework. The plugin author should very rarely need to override this file.

- The injection model for components first looks at the constructor with the largest number of arguments and tries to call it, looking up parameters by type in the plugin's bean factory. If only a no-arg constructor is found, it is called then Spring tries to autowire the bean by looking at the types used by setter methods. If you wish to have more control over how your components are created and configured, you can create your own Spring configuration file, stored in `META-INF/spring` in your plugin jar.
- If the `public` attribute is set to 'true', the component will be turned into an OSGi service under the covers, using Spring Dynamic Modules to manage its lifecycle.
- This module type in non-OSGi (version 1) plugins supported the StateAware interface in some products to allow a component to react to when it is enabled or disabled. To achieve the same effect, you can use the two Spring lifecycle interfaces: InitializingBean and DisposableBean. The init() and destroy() methods on these interfaces will be called when the module is enabled or disabled, exactly like StateAware. Making this change to a component in an existing plugin will be backwards compatible in all but JIRA. That is, a component module in a legacy plugin which implements InitializingBean will have its init() method called when it is enabled, exactly the same as such a component in an OSGi plugin.
- Components for non-OSGi (version 1) plugins behave very differently to components for OSGi plugins. For version 1 plugins, components are loaded into the application's object container, be it PicoContainer for JIRA or Spring for all other products that support components. For OSGi plugins, components are turned into beans for the Spring bean factory for that specific plugin. This provides more separation for plugins, but means you cannot do things like override JIRA components in OSGi plugins, as you can for static plugins.

RELATED TOPICS

Information sourced from Plugin Framework documentation

Web Item Module Type

Web UI plugin modules allow you to add links, tabs and sections of links to the Fisheye user interface. By adding a link to a servlet plugin, you can add your own pages to the UI. Your pages will need to ask FishEye/Crucible to decorate them.

This page describes Web Items in general terms – read about Crucible Web Item Locations, FishEye Web Item Locations and Web Item Conditions for details of using Web Items in FishEye and Crucible.

On this page:

- Purpose of this Module Type
- Configuration
  - Attributes
  - Elements
  - Label Elements
  - Tooltip Elements
  - Link Elements
  - Icon Elements
  - Param Elements
  - Context-provider Element
  - Condition and Conditions Elements
- Example

Purpose of this Module Type

Web Item plugin modules allow plugins to define new links in application menus.

Configuration

The root element for the Web Item plugin module is **`web-item`**. It allows the following attributes and child elements for configuration:

Attributes

Name	Required	Description	Default
class		The class which implements this plugin module. The class you need to provide depends on the module type. For example, Confluence theme, layout and colour-scheme modules can use classes already provided in Confluence. So you can write a theme-plugin without any Java code. But for macro and listener modules you need to write your own implementing class and include it in your plugin. See the plugin framework guide to creating plugin module instances.	
disabled		Indicate whether the plugin module should be disabled by default (value='true') or enabled by default (value='false').	false
i18n-name-key		The localisation key for the human-readable name of the plugin module.	
key	✓	The identifier of the plugin module. This key must be unique within the plugin where it is defined. Sometimes, in other contexts, you may need to uniquely identify a module. Do this with the **complete module key**. A module with key `fred` in a plugin with key `com.example.modules` will have a complete key of `com.example.modules:fred`.	N/A
name		The human-readable name of the plugin module. Used only in the plugin's administrative user interface.	

section	✓	Location into which this web item should be placed. For non-sectioned locations, this is just the location key. For sectioned locations it is the location key, followed by a slash ('/'), and the name of the web section in which it should appear.	N/A
system		Indicates whether this plugin module is a system plugin module (value='true') or not (value='false'). Only available for non-OSGi plugins.	false
weight		Determines the order in which web items appear. Items are displayed top to bottom or left to right in order of ascending weight. The 'lightest' weight is displayed first, the 'heaviest' weights sink to the bottom. The weights for most applications' system sections start from 100, and the weights for the links generally start from 10. The weight is incremented by 10 for each in sequence so that there is ample space to insert your own sections and links.	1000

Elements

The table summarises the elements. The sections below contain further information.

Name	Required	Description	Default
condition		Defines a condition that must be satisfied for the web item to be displayed. If you want to 'invert' a condition, add an attribute 'invert="true"' to it. The web item will then be displayed if the condition returns false (not true).	N/A
conditions		Defines the logical operator type to evaluate its condition elements. By default 'AND' will be used.	AND
context-provider		Allows dynamic addition to the velocity context available for various web item elements (in XML descriptors only). Currently only one context-provider can be specified per web item and section.	
description		The description of the plugin module. The 'key' attribute can be specified to declare a localisation key for the value instead of text in the element body. I.e. the description of the web item.	
icon		Defines an icon to display with or as the link. **Note:** In some cases the icon element is required. Try adding it if your web section is not displaying properly.	N/A
label	✓	Is the i18n key that will be used to look up the textual representation of the link.	N/A
link	✓	Defines where the web item should link to. The contents of the link element will be rendered using Velocity, allowing you to put dynamic content in links. For more complex examples of links, see below.	N/A
param		Parameters for the plugin module. Use the 'key' attribute to declare the parameter key, then specify the value in either the 'value' attribute or the element body. This element may be repeated. An example is the configuration link described in Adding a Configuration UI for your Plugin. This is handy if you want to use additional custom values from the UI.	N/A
resource		A resource for this plugin module. This element may be repeated. A 'resource' is a non-Java file that a plugin may need in order to operate. Refer to Adding Plugin and Module Resources for details on defining a resource.	N/A
tooltip		Is the i18n key that will be used to look up the textual mouse-over text of the link.	N/A

Label Elements

Label elements may contain optional parameters, as shown below:

- The parameters allow you to insert values into the label using Java's MessageFormat syntax.
- Parameter names must start with `param` and will be mapped in *alphabetical order* to the substitutions in the format string. I.e. param0 is {0}, param1 is {1}, param2 is {2}, etc.
- Parameter values are rendered using Velocity, allowing you to include dynamic content.

Tooltip Elements

Tooltip elements have the same attributes and parameters as the label elements. See above.

Link Elements

Link elements may contain additional information:

- The `linkId` is **optional**, and provides an XML id for the link being generated.
- The `absolute` is **optional** and defaults to false unless the link starts with `http://` or `https://`

The body of the link element is its URL. The URL is rendered with Velocity, so you can include dynamic information in the link. For example,

in Confluence, the following link would include the page ID:

Icon Elements

Icon elements have a `height` and a `width` attribute. The location of the icon is specified within a link element:

Param Elements

Param elements represent a map of key/value pairs, where each entry corresponds to the param elements attribute: `name` and `value` respectively.

The value can be retrieved from within the Velocity view with the following code, where $item is a `WebItemModuleDescriptor`:

If the `value` attribute is not specified, the value will be set to the body of the element. I.e. the following two param elements are equivalent:

Context-provider Element

Available:	Atlassian Plugins 2.5, Confluence 2.5, Bamboo 3.0, JIRA 4.2 and later

The context-provider element adds to the Velocity context available to the web section and web item modules. You can add what you need to the context, to build more flexible section and item elements. Currently only one context-provider can be specified per module. Additional context-providers are ignored.

The `context-provider` element must contain a class attribute with the fully-qualified name of a Java class. The referenced class:

- must implement `com.atlassian.plugin.web.ContextProvider`, and
- will be auto-wired by Spring before any additions to the Velocity context.

For example, the following context-provider will add `historyWindowHeight` and `filtersWindowHeight` to the context.

In the following example, `HeightContextProvider` extends `AbstractJiraContextProvider`, which is only available in JIRA and happens to implement `ContextProvider`. The `AbstractJiraContextProvider` conveniently extracts the `User` and `JiraHelper` from the context map, which you would otherwise have to do manually.

The above `HeightContextProvider` can be used by nesting the following element in a web item module.

The newly added context entries `historyWindowHeight` and `filtersWindowHeight` can be used in the XML module descriptors just like normal velocity context variables, by prefixing them with the dollar symbol ($):

Condition and Conditions Elements

Conditions can be added to the web section, web item and web panel modules, to display them only when **all** the given conditions are true.

Condition elements must contain a class attribute with the fully-qualified name of a Java class. The referenced class:

- must implement `com.atlassian.plugin.web.Condition`, and
- will be auto-wired by Spring before any condition checks are performed.

Condition elements can take optional parameters. These parameters will be passed in to the condition's `init()` method as a map of string key/value pairs after autowiring, but before any condition checks are performed. For example:

To invert a condition, add the attribute 'invert="true"' to the condition element. This is useful where you want to show the section if a certain condition is *not* satisfied.
Conditions elements are composed of a collection of condition/conditions elements and a type attribute. The type attribute defines what logical operator is used to evaluate its collection of condition elements. The type can be one of **AND** or **OR**.

For example: The following condition is true if the current user is a system administrator OR a project administrator:

Example

Here is an example `atlassian-plugin.xml` file containing a single web item:

RELATED TOPICS

Information sourced from Plugin Framework documentation

Crucible Web Item Locations

This page describes the places in the Crucible UI where you can add a Web Item. You control wheteher a Web Item is shown using a condition.

There are also FishEye Web Items.

On this page:

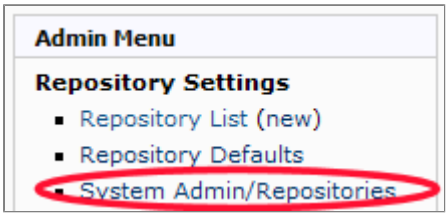
- Web Items Listing and Reference
- Visual Locations of Crucible Web Items

Web Items Listing and Reference

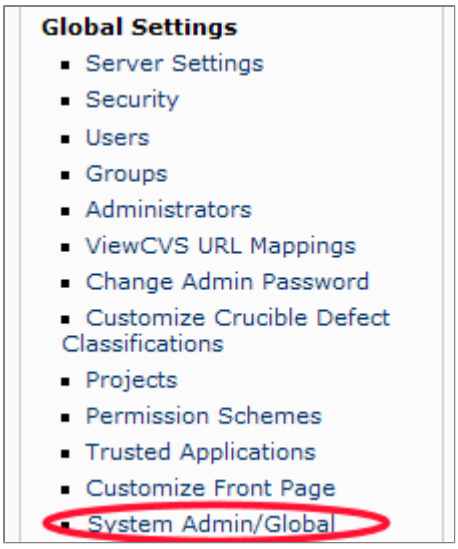
Each location has a number of Helper Objects available, which can be used to generate parts of the links and titles.

Key	Description	Helpers available
system.admin	Links on the admin menu. Sections: repositories, global, system.	global
system.crucible.review	Actions which can be performed on a review. These appear in the tools menu of the pop-up shown for a review in a review list, and the main Tools menu of the Review page.	global, project, review
system.crucible.review.comment	Actions which can be performed on a review comment. These appear as buttons in the comment header bar, left of the reply, edit and delete buttons.	global, project, review, reviewItem, repository, comment
system.crucible.review.fileitem	Actions which can be performed on a revision in a review. Displayed next to the 'Remove', 'Change Diff' buttons.	global, project, review, reviewItem, repository
system.header.item	Links in the header, separated by a pipe.	global
system.userprofile.tab	The user profile tabs.	global

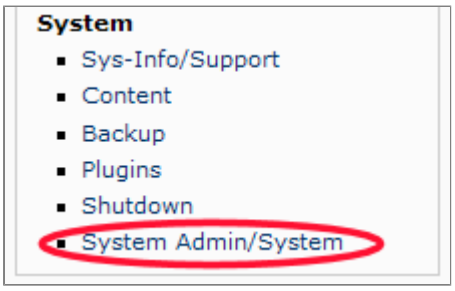
Visual Locations of Crucible Web Items



Screenshot: Crucible's system.admin/global Web Item



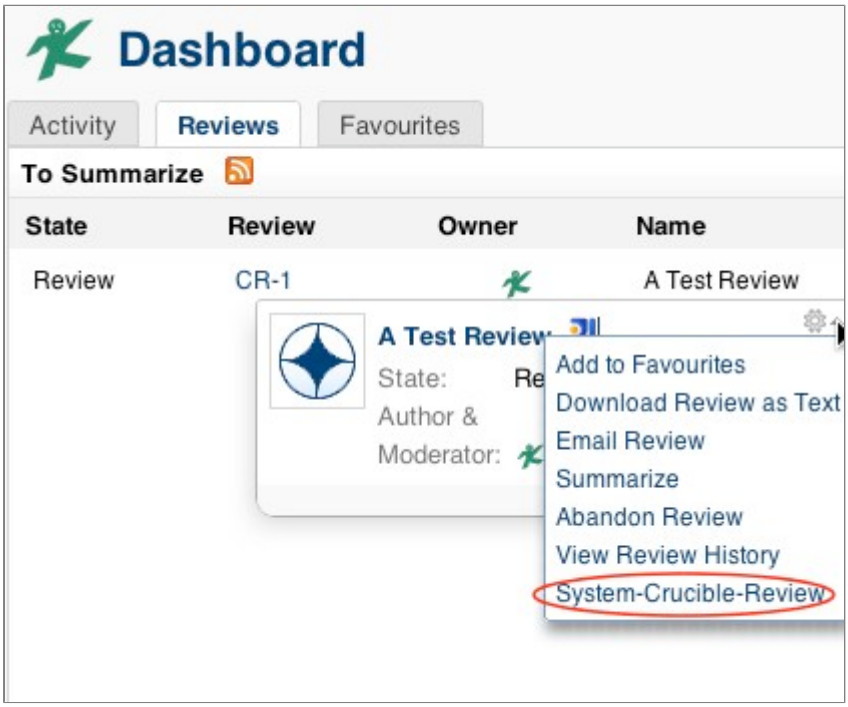
Screenshot: Crucible's system.admin/system Web Item

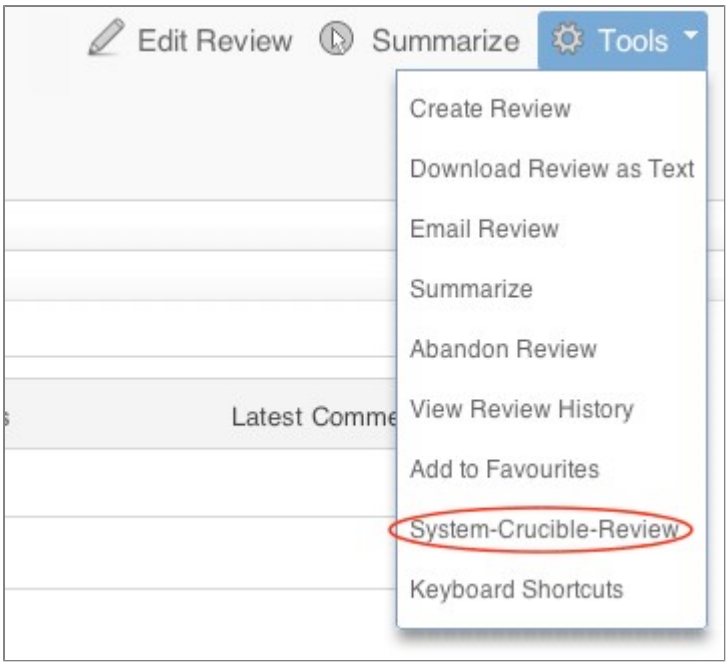


system.crucible.review

This item relates to actions that can be performed on a review, appearing in various places inside the Crucible UI.

Screenshot: Crucible's system.crucible.review Web Item





system.crucible.review.comment

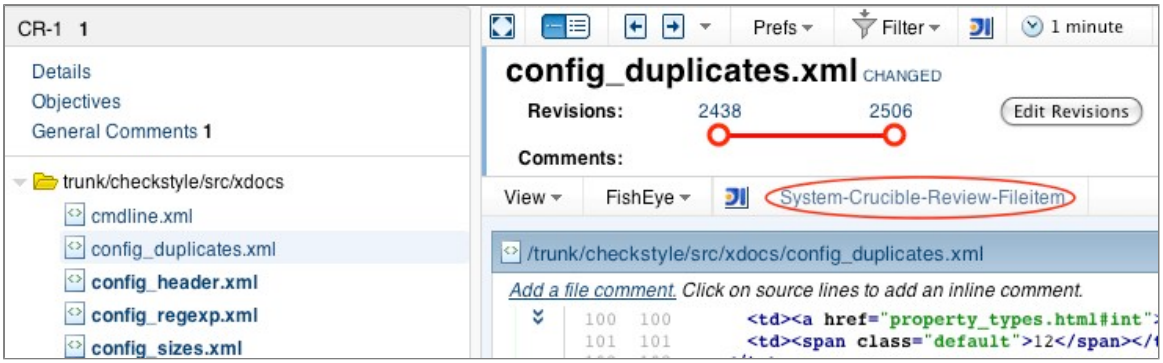
This item relates to actions that can be performed on a review, appearing in various places inside the Crucible UI.

Screenshot: Crucible's system.crucible.review.comment Web Item

system.crucible.review.fileitem

This item relates to actions which can be performed on a revision in a review, in the Crucible UI.

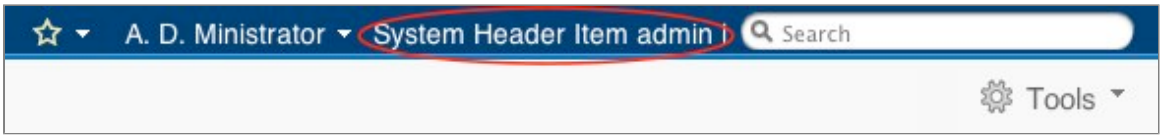
Screenshot: Crucible's system.crucible.review.fileitem Web Item



system.header.item

This item creates a link in the Crucible header, at the top right of the Crucible screen.

Screenshot: Crucible's system.header.item Web Item



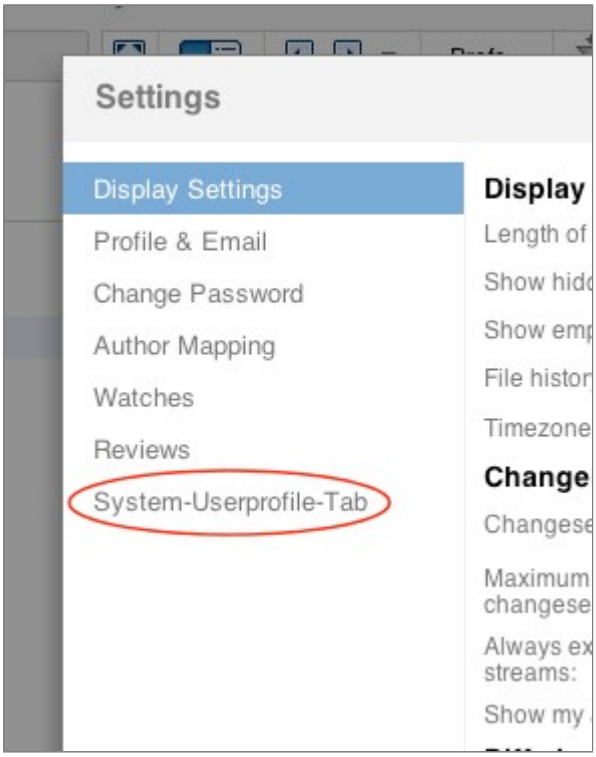
This example includes the user's login name in the link and the label.

```
<web-item key="test3" section="system.header.item">
  <link>/plugins/servlet/thing-servlet?name=${helper.global.user.userName}</link>
  <label key="System Header Item {0}">
     <param name="param0">${helper.global.user.userName}</param>
  </label>
</web-item>
```

system.userprofile.tab

This item relates to user profile tabs in the Crucible UI.

Screenshot: Crucible's system.userprofile.tab Web Item



Looking for the FishEye web items? Click here.

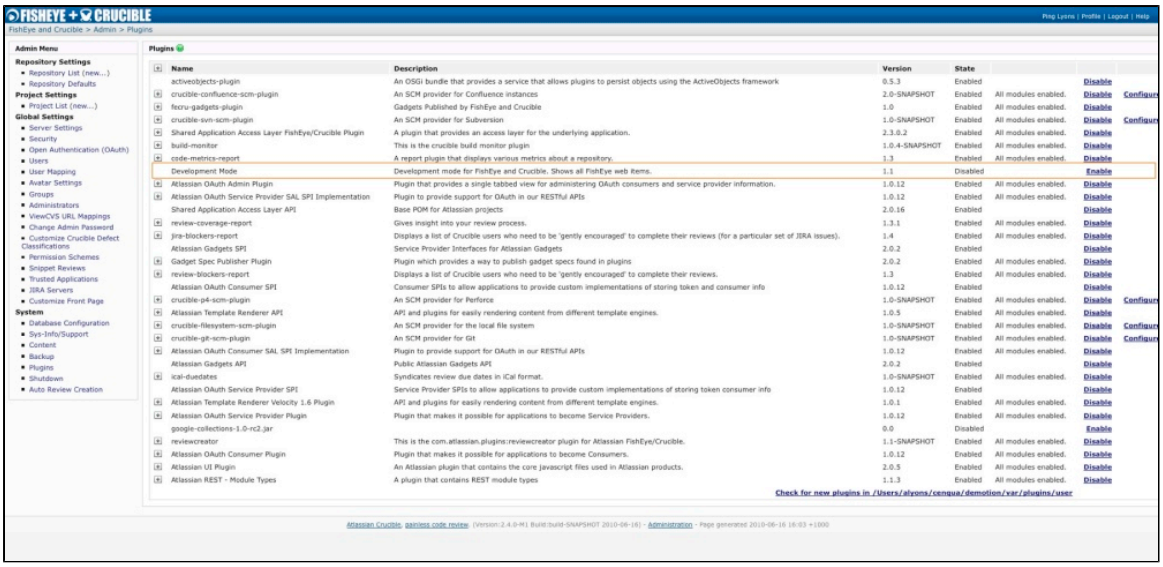
Discovering Web Items - Enable the FishEye & Crucible Development Mode Plugin

This page is a draft and will not be available to the public until the release of 2.4

As of FishEye & Crucible 4.0 finding where you can put your own links is much easier. Simply activitate the development mode plugin, and web items will be added in each web item location.

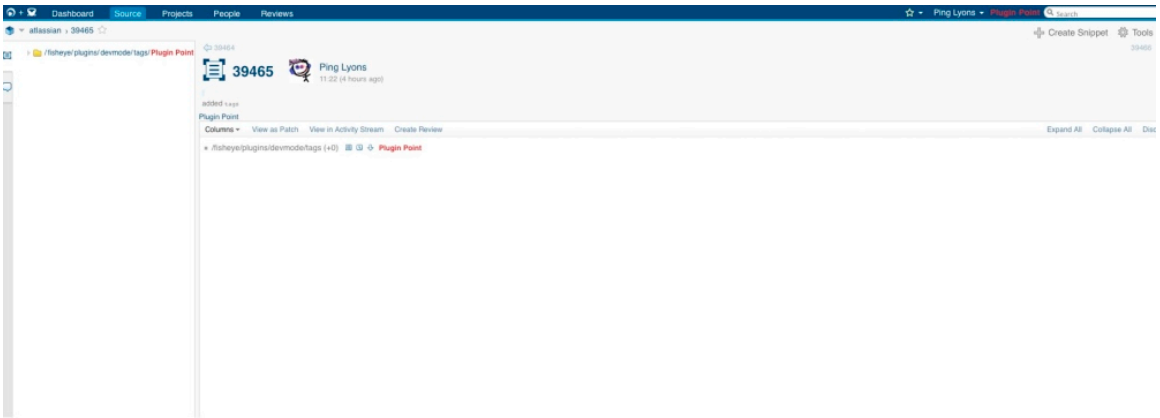
Enabling the Plugin

Go to Administration > Plugins and enable the Development mode plugin.



Using the Plugin

All web item locations will now show up as you browse FishEye and Crucible. Each web item location will now have one placeholder web item with the text 'Plugin Point', in bold and red. Hover over the link to find out the key to use for that web item in your atlassian-plugin.xml. The placeholders link to the web item documentation.



FishEye Web Item Locations

This page details the places in the FishEye UI where Web Items can be added.
There are also Crucible Web Items. You can control whether a Web Item is displayed using a condition.

On this page:

- Web Section Definition
- Web Items Listing and Reference
- Visual Locations of FishEye Web Items

Web Section Definition

You may choose to create your own web sections or add to FishEye's predefined ones.

Here is a sample `atlassian-plugin.xml` fragment for a web section:

```
<web-section key="testsection" name="example.section" location="system.fisheye.file">
      <label key="System Fisheye File Section"/>
</web-section>

<web-item key="eg7" section="system.fisheye.file/example.section">
      <label key="System Fisheye File/Example Section"/>
</web-item>
```

The above example will creates a new **section** in `system.fisheye.file`. You can then add a **web item** in the section. The location of the web item in the second code block is the same as the first, but it uses the web item `example.section` which is defined in the first code block.

Web Items Listing and Reference

Each location has a number of Helper Objects available, which can be used to generate parts of the links and titles.

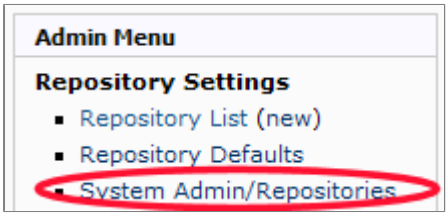
Key	Description	Helpers available
system.admin	Links on the admin menu. Sections: repositories, global, system.	global
system.admin.repo.operation *(since 2.4)*	Links in the **Operations** section of the admin **Repository List**	global, repository
system.header.item	Links in the header, separated by a pipe.	global
system.fisheye.changeset	This item relates to actions performed on a changeset, displayed in the tools menu next to each changeset activity item, and on the change set page.	global, changeset, repository
system.fisheye.repo	This item adds columns to the table of repositories displayed on the Repositories page. The plugin must supply a section which is used as the column header in the table.	global, repository
system.fisheye.directory	Per path (file or directory) links. Displayed in the directory tree in the left-hand navigation bar, under the Fisheye Source tab.	global, directory, repository
system.fisheye.file	Per file links. Displayed in the 'Files' table on the Fisheye Source tab. The plugin must supply a section which is used as the column header in the table.	global, file, repository
system.fisheye.revision	Actions performed for a revision. Displayed in the header of the revision details pane, and next to each revision in the changeset view.	global, changeset, revision, repository
repository.report.tab	This item adds a link to the reports tab of a repository. This would usually be a link to a servlet which generates a page specifying the `fisheye.report.tab` decorator.	global, repository

Visual Locations of FishEye Web Items

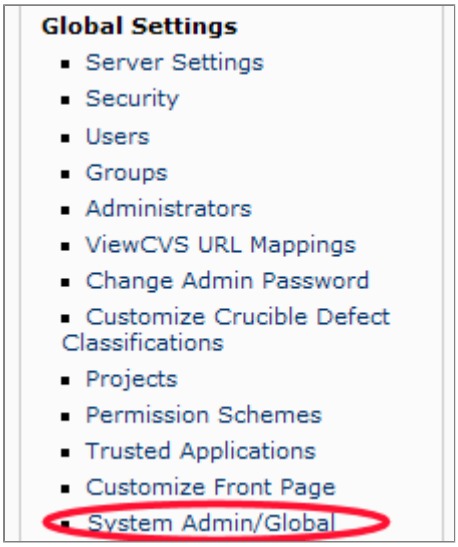
system.admin

This location creates links in the left navigation bar in the FishEye admin menu. The section controls which panel the link appears in.

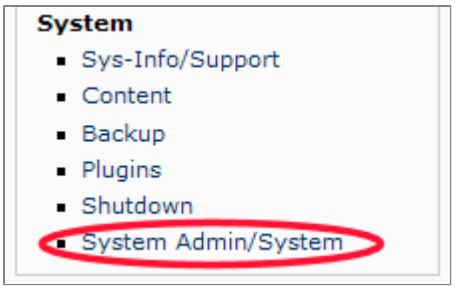
Screenshot: FishEye's system.admin/repositories Web Item



Screenshot: FishEye's system.admin/global Web Item



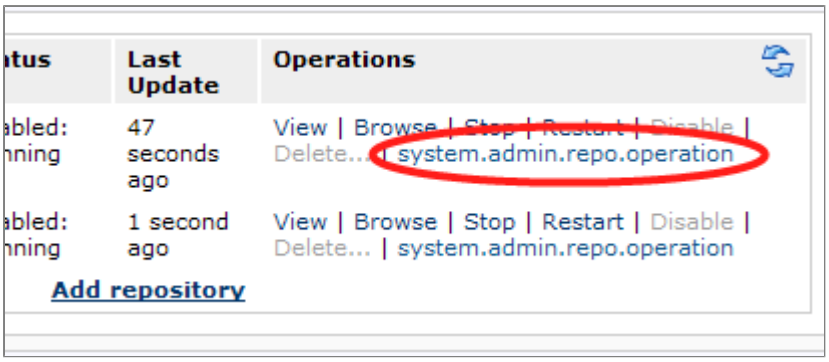
Screenshot: FishEye's system.admin/system Web Item



system.admin.repo.operation *(since FishEye 2.4)*

This location creates links in the **Operations** column of the **Repository List** table, located in the administration console.

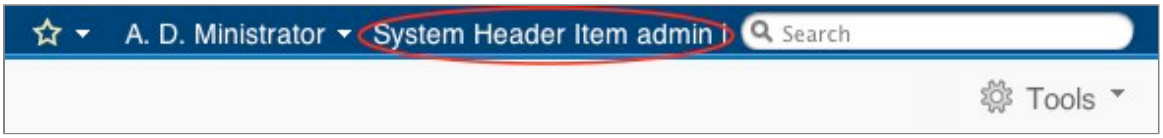
Screenshot: FishEye's system.admin.repo.operation Web Item



system.header.item

This item creates a link in the Crucible header, at the top right of the Crucible screen.

Screenshot: Crucible's system.header.item Web Item



This example includes the user's login name in the link and the label.

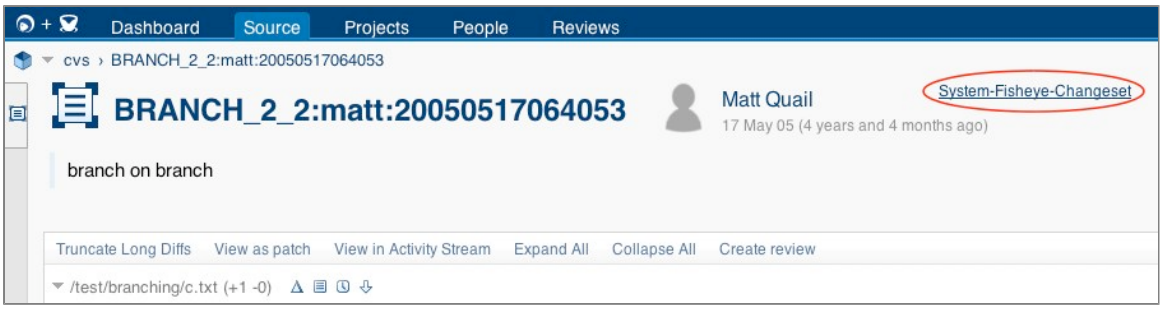
```
<web-item key="test3" section="system.header.item">
  <link>/plugins/servlet/thing-servlet?name=${helper.global.user.userName}</link>
  <label key="System Header Item {0}">
     <param name="param0">${helper.global.user.userName}</param>
  </label>
</web-item>
```

system.fisheye.changeset

This item relates to actions performed on a changeset, displayed next to the review creation link in the FishEye UI.

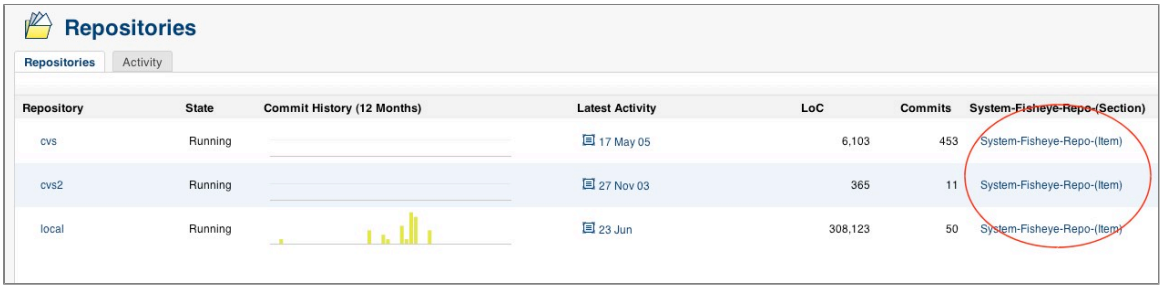
Screenshot: FishEye's system.fisheye.changeset Web Item

and on the changeset page:



system.fisheye.repo

This item adds columns to the table of repositories displayed on the Repositories page. The plugin must supply a section which is used as the column header in the table.



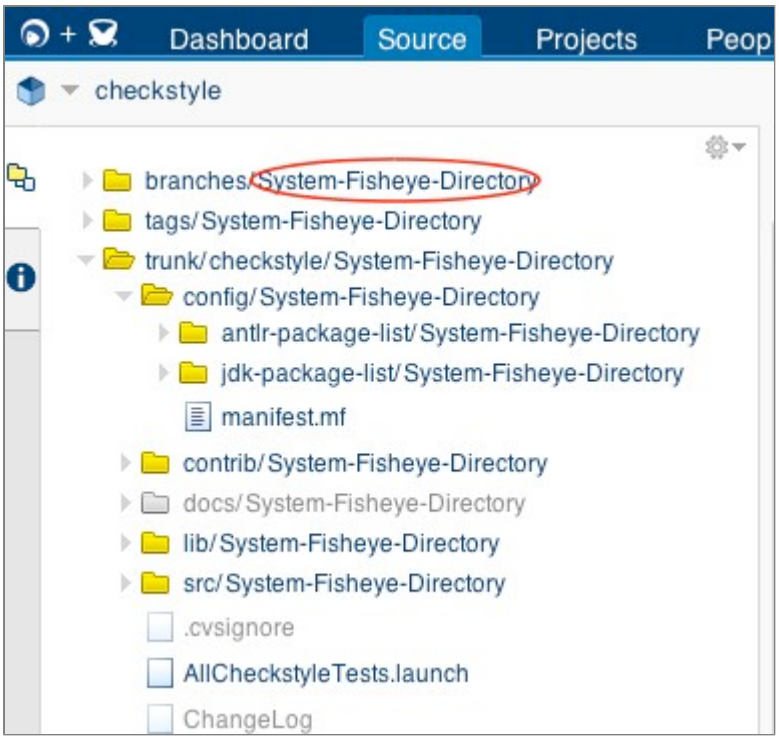
The web item above was declared with this XML:

```
<web-section key="repo-section" name="repoSection" location="system.fisheye.repo">-->
    <label key="System-Fisheye-Repo-(Section)" />
</web-section>
<web-item key="repo-link" section="system.fisheye.repo/repoSection">
    <link>/plugins/servlet/my-servlet?path=${helper.repository.path}</link>
    <label key="System-Fisheye-Repo-(Item)"/>
</web-item>
```

system.fisheye.directory

This item relates to links in the directory tree shown in the left hand navigation bar, under the FishEye Source Tab.

Screenshot: FishEye's system.fisheye.directory Web Item



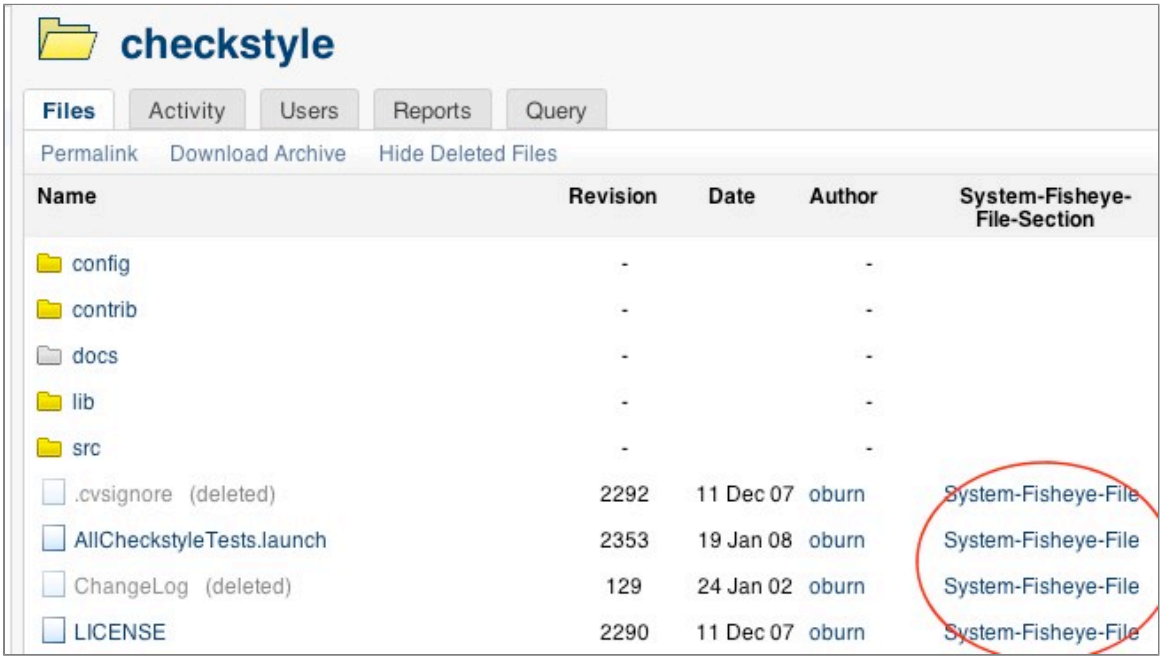
Note that while `system.fisheye.directory` web items appear on both files **and** directories in the tree, we have suppressed the file items by using a condition:

```
<web-item key="directory-link" section="system.fisheye.directory">
        <link>/plugins/servlet/my-servlet?path=${helper.repository.path}</link>
        <label key="System-Fisheye-Directory"/>
        <condition class="com.atlassian.fisheye.plugin.web.conditions.IsRootOrDirectory"/>
</web-item>
```

system.fisheye.file

This item relates to per-file links on the FishEye Browse Tab.

Screenshot: FishEye's system.fisheye.file Web Item



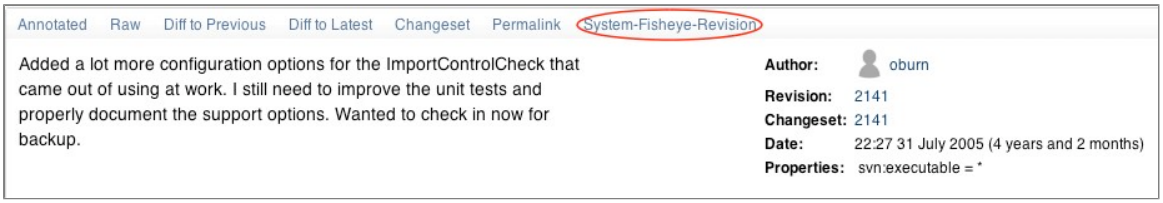
The `system.fisheye.file` location needs a section to be defined, to provide the new column in the file table:

```
<web-section key="file-link-section" name="first-section" location="system.fisheye.file">
        <label key="System-Fisheye-File-Section" />
</web-section>
<web-item key="file-link" section="system.fisheye.file/first-section">
        <link>/plugins/servlet/my-servlet?path=${helper.repository.path}</link>
        <label key="System-Fisheye-File"/>
</web-item>
```

system.fisheye.revision

This item relates to actions performed for a revision in the FishEye UI. It appears in the revision summary box at the bottom of the file history page, and next to each change set item on the change set page and activity lists.

Screenshot: FishEye's system.fisheye.revision Web Item on File History Page



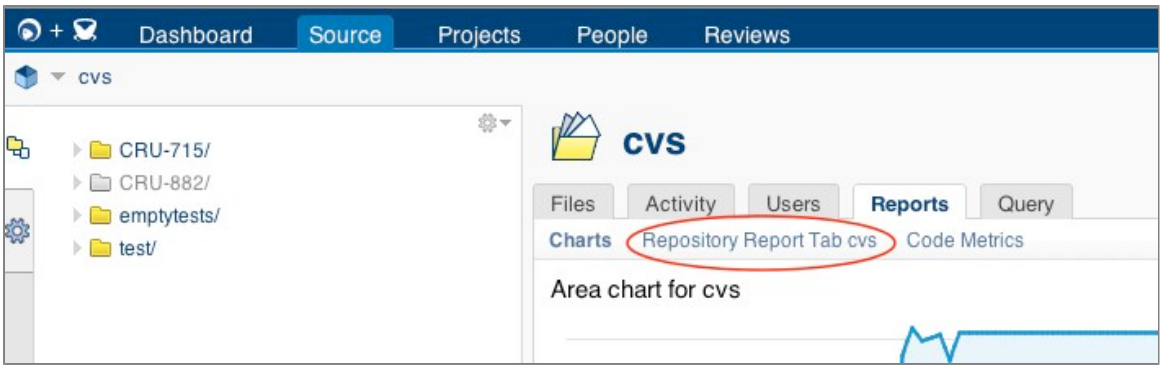
Screenshot: FishEye's system.fisheye.revision Web Item on Change Set Page

repository.report.tab

This item adds a link to the reports tab of a repository. This would usually be a link to a servlet which generates a page specifying the `fisheye.report.tab` decorator.

```
<web-item key="test" section="repository.report.tab">
     <link>/plugins/servlet/report-servlet?name=${helper.repository.path}</link>
     <label key="Repository Report Tab {0}">
         <param name="param0">${helper.repository.repositoryData.name}</param>
     </label>
</web-item>
```

Screenshot: FishEye's repository.report.tab Web Item on Reports tab



Looking for the Crucible web items? Click here.

Web Item Helpers

Name	Class	Example
global	GlobalHelper	`${helper.global.user.userName}`
project	ProjectData	`${helper.project.defaultModerator}`
review	ReviewData	`${helper.review.creator.userName}`
reviewItem	ReviewItemData	`${helper.reviewItem.permId.id}`
repository	RepositoryHelper	`${helper.repository.path}`
comment	CommentData	`${helper.comment.message}`
changeset	ChangesetDataFE	`${helper.changeset.author}`
revision	FileRevisionData	`${helper.revision.path}`

Page Decorators

Decorators allow content generated by one of your servlet modules to appear in the FishEye/Crucible UI without you needing to generate headers, footers, sidebars and so on.

On this page:

- What is a Decorator?
- Decorator Code Example
- Table of Decorator Parameters

What is a Decorator?

A decorator creates standard parts of the page and inserts the content created by a plugin servlet in the appropriate place. For example, the `atl.general` decorator provides the standard header and footer, while the `atl.admin` decorator also provides the left hand column of administration links, with the plugin generated content in a section to its right.

So, a servlet which is rendering a URL from a web item in the `system.admin` location would request the `atl.admin` decorator.

All servlet plugins whose response is rendered as a Fisheye/Crucible page should specify a decorator in the response. Meta tags in the `head` of the HTML page are used to choose a decorator and provide it with any parameters it needs.

Decorator Code Example

The decorator name and parameters are given thus:

```
<head>
...
<meta name='decorator' content='fisheye.userprofile.tab'/>
<meta name='profile.tab.key'
content='com.atlassian.crucible.example.plugin.event.crucible-example-event-plugin:test-profile-tab'/
```

The value given by the `content` parameter of the `profile.tab.key` is the plugin module key of the web-item which creates the tab we want, shown as selected when this page is rendered.

Alternatively you can request a decorator by setting an attribute on the request in your servlet:

```
request.setAttribute("decorator", "fisheye.userprofile.tab");
// request.setAttribute("meta.profile.tab.key", "twitter"); -- should work, but doesn't
response.setContentType("text/html");
```

Note that:

- Specifying `meta` parameters as attributes doesn't seem to work – put them in your page instead.
- Only responses with a content type of `text/html` are decorated.

Table of Decorator Parameters

Decorator	Description	Parameters
`atl.general`	Provides standard header and footer.	None
`atl.admin`	Provides left hand column of administration links.	None
`fisheye.userprofile.tab`	Content rendered as a single tab on the user profile page.	`profile.tab.key` The plugin module key of the web item which created the tab.
`fisheye.report.tab`	Content rendered as a single tab on the Source, Reports page	`report.tab.key` The plugin module key of the web item which created the tab.
`crucible.report.tab`	Content rendered as a single tab on the Reviews, Reports page	`report.tab.key` The plugin module key of the web item which created the tab. `repository.path` this needs to be provided if the report relates to a particular repository and path. It has the form `<repo name>/<path within repo>`.

Web Item Conditions

Web Item Conditions

Conditions control whether a given web item will be displayed.

com.atlassian.fisheye.plugin.web.conditions.HasCrucible
This condition measures whether the product runs with a Crucible license.

com.atlassian.fisheye.plugin.web.conditions.HasFishEye
This condition measures whether the product runs with a FishEye license. This is useful to prevent a Crucible plugin from rendering in an instance that only has a FishEye license.

com.atlassian.fisheye.plugin.web.conditions.HasProjectPermission
This condition measures whether the user has project permission. Takes parameters.

com.atlassian.fisheye.plugin.web.conditions.HasReviewPermission
This condition measures whether the user has review permission (i.e. is able to take part in the review). Takes parameters.

com.atlassian.fisheye.plugin.web.conditions.IsFile
This condition passes if there is a context repository and path, and that path references a repository file.

com.atlassian.fisheye.plugin.web.conditions.IsReviewInState
This condition measures whether the review is in a given state. Takes parameters.

com.atlassian.fisheye.plugin.web.conditions.IsRootOrDirectory
This condition passes if there is either: a context repository and no repository context path; or a repository path is present, and that path references a directory.

com.atlassian.fisheye.plugin.web.conditions.IsSystemAdministrator
This condition measures whether the user has system administrator permissions.

com.atlassian.fisheye.plugin.web.conditions.UserCanAccessCrucible
This condition measures whether the user can access Crucible.

com.atlassian.fisheye.plugin.web.conditions.UserLoggedInCondition
This condition measures whether the user is logged in.

Condition Parameters

The following conditions take parameters:

- HasProjectPermission
- HasReviewPermission
- IsReviewInState

The usage and conditions that these parameters apply to are tabled below.

Parameter Value	Parameter Name	Description	Applies to
action:abandonReview	actionName	Causes the current review to be abandoned.	HasProjectPermission, HasReviewPermission
action:approveReview	actionName	Causes the current review to be approved.	HasProjectPermission, HasReviewPermission
action:closeReview	actionName	Causes the current review to be closed.	HasProjectPermission, HasReviewPermission
action:recoverReview	actionName	Causes the current review to be recovered.	HasProjectPermission, HasReviewPermission
action:reopenReview	actionName	Causes the current review to be re-opened.	HasProjectPermission, HasReviewPermission
action:rejectReview	actionName	Causes the current review to be rejected.	HasProjectPermission, HasReviewPermission
action:submitReview	actionName	Causes the current review to be submitted.	HasProjectPermission, HasReviewPermission
action:summarizeReview	actionName	Causes the current review to be summarised.	HasProjectPermission, HasReviewPermission
Approval	stateName	Measures whether the current review is in the approval state.	IsReviewInState
Closed	stateName	Measures whether the current review is in the closed state.	IsReviewInState
Dead	stateName	Measures whether the current review is in the dead state.	IsReviewInState
Draft	stateName	Measures whether the current review is in the draft state.	IsReviewInState
Review	stateName	Measures whether the current review is in the review state.	IsReviewInState
Rejected	stateName	Measures whether the current review is in the rejected state.	IsReviewInState
Summarize	stateName	Measures whether the current review is in the summarize state.	IsReviewInState
Unknown	stateName	Measures whether the current review is in the unknown state.	IsReviewInState

Applying these values will cause the action to be enacted on the currently logged-in user.

Example of Condition Parameters in Use

```
<condition class="com.atlassian.fisheye.plugin.web.conditions.HasReviewPermission">
            <param name="actionName" value="action:approveReview"/>
        </condition>
```

Example of a Web Item Condition in Use

```
<web-item key="hello-file2" section="system.crucible.review.fileitem">
       <link>/plugins/servlet/my-servlet?name=${helper.global.user.displayName}</link>
       <label key="Id: {0}">
           <param name="param0">${helper.review.permaId.id}</param>
       </label>
       <condition class="com.atlassian.fisheye.plugin.web.conditions.IsReviewInState">
           <param name="stateName" value="Draft"/>
       </condition>
</web-item>
```

Web Resources

FishEye/Crucible plugins may define downloadable resources.

If your plugin requires FishEye/Crucible to include additional static Javascript or CSS files, you will need to use downloadable web resources to make them available. Web resources are included for download, at the top of the page in the HTML's header.

Web resources can also take advantage of caching (i.e. only download a resource if it has changed) and batching (i.e. serve multiple files in one request). If you would like to include other static files for download, such as images, please see Downloadable Plugin Resources.

Defining a Single Web Resource

Downloadable resources are configured to map a `name` of some downloadable file to its `location` within the plugin jar-file.

```
<web-resource key="autofavourite-resources" name="Auto-Favourite Resources">
    <resource type="download" name="autofavourite.css" location="autofavourite.css"/>
</web-resource>
```

- Resources must be contained within a `webresource` tag.
- The `key` of the webresource is how it will be referenced from within the application
- Each resource must be of `type="download"`
- The `name` of the resource will be suffixed to the URL
- The `location` of the resource is where it appears within the plugin itself

Referring to Web Resources

In your plugin's Velocity template, you need to refer to a **`WebResourceManager`**, and call the `requireResource()` method:

```
$webResourceManager.requireResource("com.atlassian.fisheye.plugin.autofavourite:autofavourite-resourc
```

Because FishEye/Crucible does not transparently support template engines like Velocity of Freemarker, you will need to register the `WebResourceManager` in the page context yourself, before passing control to your template engine of choice.

For instance, to use a web-resource in a velocity template, let FishEye/Crucible inject both the `VelocityHelper` and the `WebResourceManager` into your servlet and then register the `WebResourceManager` in the velocity context before invoking the template engine:

```
public class MyServlet extends HttpServlet {

    private final WebResourceManager webResourceManager;
    private VelocityHelper velocityHelper;

    @Autowired
    public MyServlet(WebResourceManager webResourceManager, VelocityHelper velocityHelper) {
        this.webResourceManager = webResourceManager;
        this.velocityHelper = velocityHelper;
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        final Map<String, Object> context = new HashMap<String, Object>();
        context.put("webResourceManager", webResourceManager);

        response.setContentType("text/html");
        velocityHelper.renderVelocityTemplate(
                getClass().getResourceAsStream("/templates/myPageTemplate.vm"),
                context, response.getWriter());
    }
}
```

You can then use the web-resource in the template `/templates/myPageTemplate.vm`

```
<html>
<head>
    <title>My Plugin</title>
    $webResourceManager.requireResource("com.atlassian.fisheye.plugin.myplugin:my-resources")
</head>

<body>
  ...
</body>
</html>
```

To be able to use the `WebResourceManager`, make sure you include `com.atlassian.plugins:atlassian-plugins-webresource` to your project's dependencies:

pom.xml

```
<dependency>
    <groupId>com.atlassian.plugins</groupId>
    <artifactId>atlassian-plugins-webresource</artifactId>
    <version>...</version>
</dependency>
```

Java API

This page covers the FishEye/Crucible API and how you can use service interfaces that are exposed to plugins.

FishEye/Crucible is built around Spring, an open-source component framework for for Java.

If you are familiar with Spring, then you may only wish to know that FishEye/Crucible plugin modules (and their implementing classes) are autowired by name. Thus, if you want to access a FishEye/Crucible component from your plugin, just include the appropriate setter method in your implementing class.

If you want to write FishEye/Crucible plugins but are unfamiliar with Spring, the rest of this page should give you more than enough information on how to have your plugin interact with FishEye/Crucible.

Interacting with FishEye/Crucible

When you are writing anything but the simplest FishEye/Crucible plugin, you will need to interact with the FishEye/Crucible application itself in order to retrieve, change or store information. This document describes how this can be done.

Service Objects

At the core of FishEye/Crucible is a group of "Service" objects. For example, the `ReviewService` facilitates the retrieval and manipulation of Crucible code reviews.

Dependency Injection

Traditionally, in a component-based system, components are retrieved from some kind of central repository. For example, in an EJB-based system, you would retrieve the bean from the application server's JNDI repository.

FishEye/Crucible works the other way round. When a plugin module is instantiated, FishEye/Crucible determines which components the module needs, and delivers them to it.

FishEye/Crucible determines which components a module needs by reflecting on the module's methods. Any method with a signature that matches a standard JavaBeans-style setter of the same name as a FishEye/Crucible component will have that component passed to it when the module is initialised.

So, if your plugin module needs to access the `ReviewService`, all you need to do is declare the class to your autowired constructor. For instance:

```
public class MyServlet extends HttpServlet {

    private final ReviewService reviewService;

    @Autowired
    public MyServlet(ReviewService reviewService) {
        this.reviewService = reviewService;
    }
}
```

You can also use setter injection on your plugin class:

```
public void setReviewService(ReviewService reviewService) {
    this.reviewService = reviewService;
}
```

Note that you **cannot** mix constructor and setter injection in the same class – if you mark a constructor with `@Autowired`, no setters will be used for injection.

All plugin module classes which are created by the plugin system are injected by Spring. That is, the `HttpServlet` subclass of a servlet plugin module, the `SCMModule` implementation of a Light SCM plugin module and the `EventListener` implementation of an event listener plugin module.

SAL - Shared Access Library

The Atlassian plugins framework comes with a set of generic modules and services that are common to all Atlassian products. These services are offered through the SAL API. Using these services is the same on all products. Consequently, a plugin that exclusively uses SAL services and nothing application-specific, will be able to run on any Atlassian product.

When developing a plugin for FishEye/Crucible, your plugins automatically has a transitive maven dependency on a version of SAL appropriate to the version of FishEye/Crucible that you are programming for.

Examples of services offered by SAL include:

- Persistent storage for plugin configuration settings
- Access to the application's license
- Support for making network requests (including Trusted Apps)

For more information on which SAL services are supported by each Atlassian product, refer to the supported platform matrix.

FishEye/Crucible JavaDoc References

- FishEye Services Classes – used to browse source repositories and committer activity
    - Data Types (POJO's) – the data objects used as parameters by the FishEye Services
- Crucible Services Classes – used to search and create reviews
    - Data Types (POJO's) – the data objects used as parameters by the Crucible Services
- SAL – Atlassian's Shared Access Library that offers functionality common to all Atlassian products

The complete javadoc reference can be found at: http://docs.atlassian.com/fisheye-crucible/2.4.2/javadoc.

You can modify the version shown in the URL to get to the documentation for the version you are using.

API Javadoc

FishEye/Crucible JavaDoc References

- FishEye Services Classes – used to browse source repositories and committer activity
    - Data Types (POJO's) – the data objects used as parameters by the FishEye Services
- Crucible Services Classes – used to search and create reviews
    - Data Types (POJO's) – the data objects used as parameters by the Crucible Services
- SAL – Atlassian's Shared Access Library that offers functionality common to all Atlassian products

The complete javadoc reference can be found at: http://docs.atlassian.com/fisheye-crucible/2.4.2/javadoc.

You can modify the version shown in the URL to get to the documentation for the version you are using.