CRaSH guide

CRaSH

Julien Viet (eXo Platform)

Preface

CRaSH is a shell for Java Content Repository that comes bundled as a war file to deploy in eXo Portal 2.5 or GateIn. It provides a file system view of a repository.

Chapter 1. Interacting with the shell

1.1. Deployment

CRaSH is a war file that deploys in a servlet container and there are two packaging available

- A core war file found under *deploy/core/crash.war* provides the base CRaSH functionnalities.
- A GateIn war file found under *deploy/gatein/crash.war* provides the additional JCR features but deploys only in a GateIn server.

You have to copy the crash.war in the appropriate server, regardless of the packaging you use.

1.2. Shell usage

1.2.1. Connection

You need to connect using telnet or SSH to use the shell.

Telnet connection is done on port 5000:

```
(! 520)-> telnet localhost 5000
Trying ::1...
Connected to localhost.
Escape character is '^]'.
   ______
  .~      ~. |`````````,       .'.                    ..'''' |         |
 |           |''''|'''''      .''```.               .''       |_________|
 |           |    `.       .'       `.          ..'          |         |
  `.______.' |      `.   .'           `. ....''              |         | 1.0.0-beta17

Follow and support the project on http://crsh.googlecode.com
Welcome to julien.local + !
It is Fri Dec 03 16:20:40 CET 2010 now
```

SSH connection is done on port 2000 with the password **crash** :

```
juliens-macbook-pro:~ julien$ ssh -p 2000 -l root localhost
root@localhost's password:
CRaSH 1.0.0-beta17 (http://crsh.googlecode.com)
Welcome to juliens-macbook-pro.local!
It is Fri Jan 08 21:12:53 CET 2010 now.
%
```

The `bye` command disconnect from the shell.

1.2.2. Features

- Line edition: the current line can be edited via left and right arrow keys
- History: the key up and key down enable history browsing
- Quoting: simple quotes or double quotes allow to insert blanks in command options and arguments, for

instance *"old boy"* or *'old boy'*. One quote style can quote another, like *"ol' boy"*.

1.3. Command usage

1.3.1. Getting basic help

The `help` command will display the list of known commands by the shell.

```
[/]% help
Try one of these commands with the -h or --help switch [addmixin, cd, checkin, checkout, commit, connect, discon
```

1.3.2. Command line usage

The basic CRaSH usage is like any shell, you just type a command with its options and arguments. However it is possible to compose commands and create powerful combinations.

1.3.2.1. Basic command usage

Typing the command followed by options and arguments will do the job

```
% ls -d /
...
```

1.3.2.2. Command help display

Any command help can be displayed by using the -h argument:

```
% ls -h
List the content of a node
 VAL            : Path of the node content to list
 -d (--depth) N : Print depth
```

1.3.2.3. Advanced command usage

A CRaSH command is able to consume and produce a stream of object, allowing complex interactions between commands where they can exchange stream of compatible objets. Most of the time, JCR nodes are the objects exchanged by the commands but any command is free to produce or consume any type.

By default a command that does not support this feature does not consumer or produce anything. Such commands usually inherits from the `org.crsh.command.ClassCommand` class that does not care about it. If you look at this class you will see it extends the the `org.crsh.command.BaseCommand`.

More advanced commands inherits from `org.crsh.command.BaseCommand` class that specifies two generic types `<C>` and `<P>`:

- `<C>` is the type of the object that the command consumes
- `<P>` is the type of the object that the command produces

The command composition provides two operators:

- The pipe operator **|** allows to stream a command output stream to a command input stream
- The distribution operator **+** allows to distribute an input stream to several commands and to combine the output stream of several commands into a single stream.

1.3.2.4. Connecting a `<Void,Node>` command to a `<Node,Void>` command through a pipe

Example 1.1. Remove all nt:unstructed nodes

```
% select * from nt:unstructed | rm
```

1.3.2.5. Connecting a `<Void,Node>` command to two `<Node,Void>` commands through a pipe

Example 1.2. Update the security of all nt:unstructed nodes

```
% select * from nt:unstructured | setperm -i any -a read + setperm -i any -a write
```

1.3.2.6. Connecting two `<Void,Node>` command to a `<Node,Void>` commands through a pipe

Example 1.3. Add the mixin mix:referenceable to any node of type nt:file or nt:folder

```
% select * from nt:file + select * from nt:folder | addmixin mix:referenceable
```

1.3.2.7. Mixed cases

When a command does not consume a stream but is involved in a distribution it will not receive any stream but will be nevertheless invoked.

Likewise when a command does not produce a stream but is involved in a distribution, it will not produce anything but will be nevertheless invoked.

1.4. JCR

1.4.1. JCR commands

1.4.1.1. Connecting to a repository

You must first connect to a repository before any other JCR based operation. When you are connected the shell will maintain a JCR session and allows you to interact with the session in a shell oriented fashion. The `connect` command is used to perform the connection. The repository name must be specified and optionally you can specify a user name and password to have more privileges.

```
% connect -c portal portal-system
Connected to workspace portal-system
```

or

```
% connect -c portal -u root -p gtn portal-system
Connected to workspace portal-system
```

1.4.1.2. Listing the content of a node

The `ls` command shows the content of a node. By default it lists the content of the current node. It can accept a path argument that can be absolute or relative.

```
[/]% ls
/
+-properties
| +-jcr:primaryType: nt:unstructured
| +-jcr:mixinTypes: [exo:owneable,exo:privilegeable]
| +-exo:owner: '__system'
| +-exo:permissions: [any read,*:/platform/administrators read,*:/platform/administrators add_node,*:/platform/a
+-children
| +-/workspace
| +-/contents
| +-/Users
| +-/gadgets
| +-/folder
```

1.4.1.3. Changing the current node

The `cd` command allows to change the current path. The command used with no argument, change to the root directory but you can provide a path argument that can be absolute or relative.

```
[/]% cd /gadgets
/gadgets
```

1.4.1.4. Printing the current node

The `pwd` command shows the current node path.

```
[/gadgets]% pwd
/gadgets
```

1.4.1.5. Creating a node

The `addnode` command creates one of several nodes. The command takes at least one node as argument, but it can take more. Each path can be either absolute or relative. Relative path creates nodes relative to the current node. By default the node type is the default repository node type, but the option `-t` can be used to specify another one.

```
[/registry]% addnode foo
Node /foo created
```

```
[/registry]% addnode -t nt:file bar juu
Node /bar /juu created
```

Note

The `addnode` command is a `<Void,Node>` command that produces all the nodes that were created.

1.4.1.6. Copying a node

The `cp` command copies a node to a target location in the JCR tree.

```
[/registry]% cp Registry Registry2
```

1.4.1.7. Moving a node

The `mv` command can move a node to a target location in the JCR tree. It can be used also to rename a node.

```
[/registry]% mv Registry Registry2
```

Note

`mv` command is a `<Node,Node>` command consuming a stream of node to move them and producing nodes that were moved.

1.4.1.8. Removing a node or property

The `rm` command removes a node or property specified by its path either absolute or relative. This operation is executed against the JCR session, meaning that it will not be effective until it is commited to the JCR server.

```
[/]% rm foo
Node /foo removed
```

It is possible to specify several nodes.

```
[/]% rm foo bar
Node /foo /bar removed
```

Note

`rm` is a `<Node,Void>` command removing all the consumed nodes.

1.4.1.9. Updating a property

The `set` command updates the property of a node.

Create or destroy property foo with the value bar on the root node:

```
[/]% set foo bar
Property created
```

Update the existing foo property:

```
[/]% set foo juu
```

When a property is created and does not have a property descriptor that constraint its type, you can specify it with the *-t* option

```
[/]% set -t LONG long_property 3
```

Remove a property

```
[/]% set foo
```

Note

`set` is a `<Node,Void>` command updating the property of the consumed node stream.

1.4.1.10. Commiting or rolling back changes

The `commit` operation saves the current session. Conversely the `rollback` operation rollback session changes. For both operations It is possible to specify a path to commit a part of the tree.

1.4.1.11. Performing a SQL query

Queries in SQL format are possible via the `select` command. You can write a query with the same syntax defined by the specification and add options to control the number of results returned. By default the number of nodes is limited to 5 results:

```
[/]% select * from nt:base
The query matched 1114 nodes
+-/
| +-properties
| | +-jcr:primaryType: nt:unstructured
| | +-jcr:mixinTypes: [exo:owneable,exo:privilegeable]
| | +-exo:owner: '__system'
| | +-exo:permissions: [any read,*:/platform/administrators read,*:/platform/administrators add_node,*:/platform
+-/workspace
| +-properties
| | +-jcr:primaryType: mop:workspace
| | +-jcr:uuid: 'a69f226ec0a80002007ca83e5845cdac'
...
```

Display 20 nodes from the offset 10:

```
[/]% select * from nt:base -o 10 -l 20
The query matched 1114 nodes
...
```

It is possible also to remove the limit of displayed nodes with the -a option (you should use this option with care) :

```
[/]% select * from nt:base -a
The query matched 1114 nodes
...
```

Note

`select` is a `<Void,Node>` command producing all the matched nodes.

1.4.1.12. Performing an XPath query

todo

1.4.1.13. Exporting a node

It is possible to export a node as an nt file of the same workspace with the `exportnode` command. Then it is usually possible to access the nt file from webdav.

```
[/]% exportnode gadgets /gadgets.xml
The node has been exported
```

1.4.1.14. Importing a node

It is possible to import a node from an nt file located in the workspace with the `importnode` command.

```
[/]% importnode /gadgets.xml /
Node imported
```

1.4.1.15. Adding a mixin to a node

The `addmixin` command adds a mixin to an existing node.

```
[/gadgets]% addmixin . mix:versionable
```

Note

`addmixin` is a `<Node,Void>` command adding the specified mixin to the consumed nodes.

1.4.1.16. Checkin / checkout of versionable nodes

Use the commands `checkin` and `checkout`.

1.4.1.17. Configuring the node security

Warning

This is command is only available for eXo JCR

The `setperm` commands configures the security of a node based on eXo JCR access control. When a node is protected by access control, it owns a mixin called `exo:privilegeable` that contains a `exo:permissions` property, for instance:

```
[/production]% ls
/production
+-properties
| +-jcr:primaryType: nt:unstructured
| +-jcr:mixinTypes: [exo:privilegeable]
| +-exo:permissions: [*:/platform/administrators read,*:/platform/administrators add_node,*:/platform/administra
+-children
| +-/production/app:gadgets
| +-/production/app:applications
| +-/production/mop:workspace
```

You can alter the node permission list with the `setperm` command:

```
[/production]% setperm -i *:/platform/mygroup -a read -a add_node /
Node /production updated to [read,add_node]
```

You can also remove a permission by using the -r option.

```
[/production]% setperm -i *:/platform/mygroup -r add_node /
Node /production updated to [read]
```

Warning

The `setperm` command will add automatically the `exo:privilegeable` mixin on the node when it is missing.

Note

`setperm` is a `<Node,Void>` command altering the security of the consumed node stream.

1.4.1.18. Disconnecting

The `disconnect` command disconnect from the repository.

1.4.2. SCP usage

Secure copy can be used to import or export content. The username/password prompted by the SSH server will be used for authentication against the repository when the import or the export is performed.

1.4.2.1. Export a JCR node

The following command will export the node */gadgets* in the repository *portal-system* of the portal container *portal*:

```
scp -P 2000 root@localhost:portal:portal-system:/production/app:gadgets .
```

The node will be exported as *app_gadgets.xml*.

Note that the portal container name is used for GateIn. If you do omit it, then the root container will be used.

1.4.2.2. Import a JCR node

The following command will reimport the node:

```
scp -P 2000 gadgets.xml root@localhost:portal:portal-system/production/app:gadgets
```

The exported file format use the JCR system view. You can get more information about that in the JCR specification.

1.5. Logging

The logging commands provide useful interactions with the JVM logging system.

1.5.1. Listing all loggers

The `logls` command lists all the loggers available.

```
% logls
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/].[default]
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/eXoGadgetServer].[concat]
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/dashboard].[jsp]
...
```

The `-f` switch provides filtering with a regular expression.

```
% logls -f javax.*
javax.management.mbeanserver
javax.management.modelmbean
```

The `logls` command is a `<Void,Logger>` command and so any logger produced can be consumed.

1.5.2. Displaying information about a logger

The `loginfo` command displays information about one or several loggers.

```
% loginfo javax.management.modelmbean
javax.management.modelmbean<INFO>
```

The `loginfo` command is a <Logger,Void> command and it can consumed logger produced by the logls command.

```
% logls -f javax.* | loginfo
javax.management.mbeanserver<INFO>
javax.management.modelmbean<INFO>
```

1.5.3. Updating a logger level

The `logset` command sets the level of a logger. One or several logger names can be specified as arguments and the `-l` option specify the level among the *trace*, *debug*, *info*, *warn* and *error* levels.

```
% logset -l trace foo
```

When no level is specified, the level is cleared and the level will be inherited from its ancestors.

```
% logset foo
```

The logger name can be omitted and instead stream of logger can be consumed as it is a `<Logger,Void>` command. The following set the level *warn* on all the available loggers.

```
% logls | logset -l warn
```

Chapter 2. Configuring the shell

CRaSH can be configured by tweaking various files of the CRaSH web archive

- *WEB-INF/web.xml*
- *WEB-INF/groovy/login.groovy*

Note that to fully secure the server, you should remove the unauthenticated telnet access as describe below.

2.1. Change the SSH server key

The key can be changed by replacing the file *WEB-INF/sshd/hostkey.pem*. Alternatively you can configure the server to use an external file by using the *ssh.keypath* parameter. Uncomment the XML section and change the path to the key file.

```
<!--
<context-param>
  <param-name>ssh.keypath</param-name>
  <param-value>/path/to/the/key/file</param-value>
  <description>The path to the key file</description>
</context-param>
-->
```

2.2. Change the ports of the telnet or SSH server

The ports of the server are parameterized by the *ssh.port* and *telnet.port* parameters in the *web.xml* file

```
<context-param>
  <param-name>ssh.port</param-name>
  <param-value>2000</param-value>
  <description>The SSH port</description>
</context-param>
```

```
<context-param>
  <param-name>telnet.port</param-name>
  <param-value>5000</param-value>
  <description>The telnet port</description>
</context-param>
```

2.3. Remove the telnet or SSH access

To remove the telnet access, remove or comment the following XML from the *web.xml* file

```
<listener>
  <listener-class>org.crsh.term.spi.telnet.TelnetLifeCycle</listener-class>
</listener>
```

To remove the SSH access, remove or comment the following XML from the *web.xml* file

```
<listener>
  <listener-class>org.crsh.term.spi.sshd.SSHLifeCycle</listener-class>
</listener>
```

2.4. Configure the shell default message

The *login.groovy* file contains two closures that are evaluated each time a message is required

- The `prompt` closure returns the prompt message
- The `welcome` closure returns the welcome message

Those closure can be customized to return different messages.

Chapter 3. Extending the shell

3.1. Groovy command system

The shell command system is based on the Groovy language and can easily be extended.

3.1.1. Groovy file

Each command has a corresponding Groovy file that contains a command class that will be invoked by the shell. The files are located in the */WEB-INF/groovy/commands* directory and new files can be added here.

In addition of that there are two special files called *login.groovy* and *logout.groovy* in the */WEB-INF/groovy* directory that are executed upon login and logout of a user. Those files can be studied to understand better how the shell works.

3.1.2. Groovy execution

When the user types a command in the sell, the command line is parsed by the args4j framework and injected in the command class. A simple example, the command `connect -c portal -u root -p gtn portal-system` creates the connect command instance and args4j injects the options and arguments on the class:

```
@Description("Connect to a workspace")
class connect extends org.crsh.command.ClassCommand
{

  @Option(name="-u",aliases=["--username"],usage="user name")
  def String userName;

  @Option(name="-p",aliases=["--password"],usage="password")
  def String password;

  @Option(name="-c",aliases=["--container"],usage="portal container name (eXo portal specific)")
  def String containerName;

  @Argument(required=true,index=0,usage="workspace name")
  def String workspaceName;

  public Object execute() throws ScriptException {
    ...
  }
}
```

3.1.3. Shell context

A command is a Groovy object and it can access or use the contextual variables. A few variables are maintained by the shell and should be considered with caution. The shell also provides a few functions that can be used, those functions defined in *login.groovy*

3.1.3.1. Existing context variables

- The `session` variable is managed by the `connect` and `disconnect` commands. Commands should be able to use it for accessing JCR session but not update this variable.
- The `currentPath` variable contains the current path of the shell and it should not be used directly. Instead

one should use the function `getCurrentNode()` and `setCurrentNode(Node node)` to update the underlying path.

3.1.3.2. Existing functions

- The `assertConnected()` checks that the user is connected. It should be used at the beginning of a command that interacts with the session
- The `getCurrentNode()` returns the current node
- The `setCurrentNode(Node node)` updates the current node
- The `findNodeByPath()` functions returns a node based on the provided path. If the provided path is null then the "/" root path is considered. The path can be either relative or absolute. If the path is relative the current node will be used to find the node.
- The `formatValue(Value value)` formats a JCR value into a suitable text value.
- The `formatPropertyValue(Property property)` formats a JCR property value into a suitable text value. If the property is multiple then it will return a comma separated list surrounded by square brackets.

3.1.4. You want to contribute a command?

Drop me an email (see my @ on www.julienviet.com).