

CRaSH

CRaSH guide

Julien Viet

eXo Platform

Copyright © 2011 eXo Platform SAS

Table of Contents

Preface

- 1. Interacting with the shell
 - 1.1. Running CRaSH
 - 1.2. Shell usage
 - 1.3. Command usage
 - 1.4. Base commands
- 2. JCR extension
 - 2.1. JCR implementations
 - 2.2. JCR commands
 - 2.3. SCP usage
- 3. Configuring the shell
 - 3.1. Configuration properties
 - 3.2. Change the SSH server key
 - 3.3. Change the ports of the telnet or SSH server
 - 3.4. Remove the telnet or SSH access
 - 3.5. Configure the shell default message
 - 3.6. Configuration the authentication
- 4. Extending the shell
 - 4.1. Pluggable authentication
- 5. Developers
 - 5.1. Developping commands
 - 5.2. Inter command API
- 6. Hey, I want to contribute!

List of Examples

- 1.1. Remove all nt:unstructed nodes
- 1.2. Update the security of all nt:unstructed nodes
- 1.3. Add the mixin mix:referenceable to any node of type nt:file or nt:folder
- 5.1. dbscript.groovy

Preface

The Common Reusable SHell (CRaSH) deploys in a Java runtime and provides interactions with the JVM. Commands are written in Groovy and can be developped at runtime making the extension of the shell very easy with fast development cycle.

Interacting with the shell

1.1. Running CRaSH

CRaSH provides has various ways to be started, it can even be easily embedded.

1.1.1. Web archive deployment

CRaSH can use a standard web archive to be deployed in a web container. The war file is used for its packaging capabilities and triggering the CRaSH life cycle start/stop. In this mode CRaSH has two packaging available:

- A core war file found under *deploy/core/crash.war* provides the base CRaSH fonctionnalités.
- A gatein war file found under *deploy/gatein/crash.war* provides additional Java Content Repository (JCR) features but deploys only in a GateIn server (Tomcat or JBoss). It extends the core packaging and adds
 - JCR browsing and interactions
 - SCP support for JCR import and export

You have to copy the *crash.war* in the appropriate server, regardless of the packaging used.

1.1.2. Standalone mode

The standalone mode allows you to run CRaSH from the command line directly. It provides the same functionality as the war deployment but does not require a web container as it runs its own virtual machine. The directory crash directory in the application contains the standalone distribution.

The bin directory */crash/bin* can be added to the system path, it contains the *crash.sh* script that will start the standalone mode, for instance you can set it up this way:

Welcome to jerry + !

It is Thu Apr 12 21:19:35 CEST 2012 now

- The *bin* directory contains the *crash.sh* script and the standalone crash jar file
- The *conf* directory contains the JVM logging configuration *logging.properties*
- The *cmd* directory contains the commands that will be available in crash by default it contains a few example commands
- The *lib* directory contains the various libraries used by crash, you should place additional jar files there

The attach mode allows you to attach CRaSH to a JVM located on the same host with the attach API provided by the Hotspot JVM. It works thanks to the standalone mode, the main difference is when you run the command line you can specify a process id of a JVM and CRaSH will hook into the targetted JVM, let's see quickly an example of how to use it

```
> crash.sh 20650
```

~ x

1.0.0-c

Welcome to jerry + !

It is Thu Apr 12 22:09:23 CEST 2012 now

%

Page 5 of 47

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"*.
- Completion: an advanced completion system is available

1.3. Command usage

1.3.1. Getting basic help

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

```
[/]% help
% help
Try one of these commands with the -h or --help switch:

cd                changes the current node
commit           saves changes
consume          collects a set of nodes
cp               copy a node to another
env              display the term env
exportworkspace   Export a workspace on the file system (experimental)
fail             Fails
help             provides basic help
importworkspace   Import a workspace from the file system (experimental)
invoke           Invoke a static method
log              logging commands
ls               list the content of a node
man              format and display the on-line manual pages
mixin            mixin commands
mv               move a node
node             node commands
produce           produce a set of nodes
pwd              print the current node path
rm               remove one or several node or a property
rollback         rollback changes
select           execute a JCR sql query
setperm          modify the security permissions of a JCR node
sleep            sleep for some time
thread           vm thread commands
version           versioning commands
wait             Invoke a static method
ws               workspace commands
xpath            execute a JCR xpath query
```

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 /  
...
```

1.3.2.2. Command help display

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

```
% ls -h  
usage: ls [-h | --help] [-h | --help] [-d | --depth] path  
  
[-h | --help]  command usage  
[-h | --help]  command usage  
[-d | --depth] Print depth  
path           the path of the node content to list
```

In addition of that, commands can have a complete manual that can be displayed thanks to the `man` command:


```

% man ls
NAME
    ls - list the content of a node

SYNOPSIS
    ls [-h | --help] [-h | --help] [-d | --depth] [-d | --depth] path

DESCRIPTION
    The ls command displays the content of a node. By default it lists the content of the node. It
    accepts 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/
    +-children
    | +-/workspace
    | +-/contents
    | +-/Users
    | +-/gadgets
    | +-/folder

PARAMETERS
    [-h | --help]
        Provides command usage

    [-h | --help]
        Provides command usage

    [-d | --depth]
        Print depth

    path
        the path of the node content to list

```

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 objects. 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 `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

- $\langle P \rangle$ 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 $\langle \text{Void}, \text{Node} \rangle$ command to a $\langle \text{Node}, \text{Void} \rangle$ command through a pipe

Example 1.1. Remove all nt:unstructured nodes

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

1.3.2.5. Connecting a $\langle \text{Void}, \text{Node} \rangle$ command to two $\langle \text{Node}, \text{Void} \rangle$ commands through a pipe

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

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

1.3.2.6. Connecting two $\langle \text{Void}, \text{Node} \rangle$ command to a $\langle \text{Node}, \text{Void} \rangle$ 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. Base commands

1.4.1. *sleep* command

```
NAME
    sleep - sleep for some time

SYNOPSIS
    sleep [-h | --help] time

PARAMETERS
    [-h | --help]
        Provides command usage

    time
        sleep time in seconds
```

1.4.2. *man* command

```
NAME
    man - format and display the on-line manual pages

SYNOPSIS
    man [-h | --help] command

PARAMETERS
    [-h | --help]
        Provides command usage

    command
        the command
```

1.4.3. *log* command

```
NAME
    log add - create one or several loggers

SYNOPSIS
    log [-h | --help] add ... name

PARAMETERS
    [-h | --help]
        Provides command usage

    ... name
        The name of the logger
```

NAME

log set - configures the level of one of several loggers

SYNOPSIS

log [-h | --help] set [-l | --level] [-p | --plugin] ... name

DESCRIPTION

The set command sets the level of a logger. One or several logger names and the -l option specify the level among the trace, debug, info, warn and error. If no level is specified, the level is cleared and the level will be inherited from its parent.

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

The logger name can be omitted and instead stream of logger can be consumed. The following set the level warn on all the available loggers:

```
% log ls | log set -l warn
```

PARAMETERS

[-h | --help]

Provides command usage

[-l | --level]

The logger level to assign among {trace, debug, info, warn, error}

[-p | --plugin]

Force the plugin implementation to use

... name

The name of the logger

NAME

log send - send a message to a logger

SYNOPSIS

log [-h | --help] send [-m | --message] [-l | --level] name

DESCRIPTION

The send command log one or several loggers with a specified message. For example, you can use the javax.management.mbeanserver class and send a message on its own log.

```
## log send -m hello javax.management.mbeanserver
```

Send is a <Logger, Void> command, it can log messages to consumed log of

```
% log ls | log send -m hello -l warn
```

PARAMETERS

[-h | --help]

Provides command usage

[-m | --message]

The message to log

[-l | --level]

The logger level to assign among {trace, debug, info, warn, error}

name

The name of the logger

NAME

log info - display info about a logger

SYNOPSIS

log [-h | --help] info ... name

DESCRIPTION

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 loggers

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

PARAMETERS

[-h | --help]

Provides command usage

... name

The name of the logger

NAME

log ls - list the available loggers

SYNOPSIS

log [-h | --help] ls [-f | --filter]

DESCRIPTION

The logls command list all the available loggers., for instance:

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

The -f switch provides filtering with a Java regular expression

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

The logls command is a <Void,Logger> command, therefore any logger produced

PARAMETERS

[-h | --help]
Provides command usage

[-f | --filter]
A regular expressions used to filter the loggers

1.4.4. *thread* command

NAME

thread stop - stop vm threads

SYNOPSIS

thread [-h | --help] stop

DESCRIPTION

Stop a VM thread, this method cannot be called as is and should be used

PARAMETERS

[-h | --help]
Provides command usage

NAME

thread interrupt - interrupt vm threads

SYNOPSIS

thread [-h | --help] interrupt

DESCRIPTION

Interrupt a VM thread, this method cannot be called as is and should be used in conjunction with the thread ls command.

PARAMETERS

[-h | --help]
Provides command usage

NAME

thread ls - list the vm threads

SYNOPSIS

thread [-h | --help] ls [-n | --name] [-f | --filter] [-s | --state]

PARAMETERS

[-h | --help]
Provides command usage

[-n | --name]
Retain the thread with the specified name

[-f | --filter]
Filter the threads with a regular expression on their name

[-s | --state]
Filter the threads by their status (new,runnable,blocked,waiting,timed-out)

1.4.5. *system* command

NAME

system gc - call garbage collector

SYNOPSIS

system [-h | --help] gc

PARAMETERS

[-h | --help]
Provides command usage

NAME

system propls - list the vm system properties

SYNOPSIS

system [-h | --help] propls [-f | --filter]

PARAMETERS

[-h | --help]

Provides command usage

[-f | --filter]

filter the property with a regular expression on their name

NAME

system propset - set a system property

SYNOPSIS

system [-h | --help] propset name value

PARAMETERS

[-h | --help]

Provides command usage

name

The name of the property

value

The value of the property

NAME

system propget - get a system property

SYNOPSIS

system [-h | --help] propget name

PARAMETERS

[-h | --help]

Provides command usage

name

The name of the property

NAME

system proprm - remove a system property

SYNOPSIS

system [-h | --help] proprm name

PARAMETERS

[-h | --help]

Provides command usage

name

The name of the property

NAME

system freemem - show free memory

SYNOPSIS

system [-h | --help] freemem [-u | --unit] [-d | --decimal]

PARAMETERS

[-h | --help]

Provides command usage

[-u | --unit]

The unit of the memory space size {(B)yte, (O)ctet, (M)egaOctet, (G

[-d | --decimal]

The number of decimal (default 0)

NAME

system totalmem - show total memory

SYNOPSIS

system [-h | --help] totalmem [-u | --unit] [-d | --decimal]

PARAMETERS

[-h | --help]

Provides command usage

[-u | --unit]

The unit of the memory space size {(B)yte, (O)ctet, (M)egaOctet, (G

[-d | --decimal]

The number of decimal (default 0)

1.4.6. *jdbc* command

NAME

jdbc close - close the current connection

SYNOPSIS

jdbc [-h | --help] close

PARAMETERS

[-h | --help]
Provides command usage

NAME

jdbc open - open a connection from JNDI bound datasource

SYNOPSIS

jdbc [-h | --help] open globalName

PARAMETERS

[-h | --help]
Provides command usage

globalName
The datasource global JNDI name

NAME

jdbc connect - connect to database with a JDBC connection string

SYNOPSIS

jdbc [-h | --help] connect [-u | --username] [-p | --password] [--properties]

PARAMETERS

[-h | --help]
Provides command usage

[-u | --username]
The username

[-p | --password]
The password

[--properties]
The extra properties

connectionString
The connection string

NAME

jdbc info - describe the database

SYNOPSIS

jdbc [-h | --help] info ... tableNames

PARAMETERS

[-h | --help]

Provides command usage

... tableNames

the table names

NAME

jdbc execute - execute SQL statement

SYNOPSIS

jdbc [-h | --help] execute ... statement

PARAMETERS

[-h | --help]

Provides command usage

... statement

The statement

JCR extension

The CRaSH JCR extension allow to connect and interact with Java Content Repository implementations.

2.1. JCR implementations

2.1.1. eXo JCR

todo

2.1.2. Apache Jackrabbit

CRaSH has been tested with Jackrabbit in the following mode : deployment as a resource accessible via JNDI on JBoss 6.1.0.

2.2. JCR commands

2.2.1. *repo* command

NAME

`repo info - show info about the current repository`

SYNOPSIS

`repo [-h | --help] info`

DESCRIPTION

The `info` command print the descriptor of the current repository.

PARAMETERS

`[-h | --help]`
Provides command usage

NAME

`repo ls - list the available repository plugins`

SYNOPSIS

`repo [-h | --help] ls`

DESCRIPTION

The `ls` command print the available repository plugins.

PARAMETERS

`[-h | --help]`
Provides command usage

NAME

repo use - changes the current repository

SYNOPSIS

repo [-h | --help] use parameters

DESCRIPTION

The use command changes the current repository used by for JCR commands as main command argument that will be used to select a repository:

```
% repo use parameterName=parameterValue;nextParameterName=nextParameterName
```

The parameters is specific to JCR plugin implementations, more details c

PARAMETERS

[-h | --help]

Provides command usage

parameters

The parameters used to instantiate the repository to be used in this

2.2.2. ws command

NAME

`ws login` - login to a workspace

SYNOPSIS

`ws [-h | --help] login [-u | --username] [-p | --password] [-c | --container]`

DESCRIPTION

This command login to a JCR workspace and establish a session with the repository. When you are connected the shell maintain a JCR session and allows you to work in an interactive oriented fashion. The repository name must be specified and optionally you can specify a container to have more privileges.

Before performing a login operation, a repository must be first selected.

```
% repo use container=portal
```

Once a repository is obtained the login operation can be done:

```
% ws login portal-system
Connected to workspace portal-system
```

```
% ws login -u root -p gtn portal-system
Connected to workspace portal-system
```

PARAMETERS

`[-h | --help]`
Provides command usage

`[-u | --username]`
The user name

`[-p | --password]`
The user password

`[-c | --container]`
The portal container name (eXo JCR specific)

`workspaceName`
The name of the workspace to connect to

NAME

`ws logout` - logout from a workspace

SYNOPSIS

`ws [-h | --help] logout`

DESCRIPTION

This command logout from the currently connected JCR workspace

PARAMETERS

`[-h | --help]`
Provides command usage

2.2.3. *cd* command

NAME

`cd` - changes the current node

SYNOPSIS

`cd [-h | --help] path`

DESCRIPTION

The `cd` command changes the current node path. The command used with no arguments changes the current node to the root node. A relative or absolute path argument can be provided to specify a new path.

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

PARAMETERS

`[-h | --help]`
Provides command usage

`path`
The new path that will change the current node navigation

2.2.4. *pwd* command

NAME

`pwd` - print the current node path

SYNOPSIS

`pwd [-h | --help]`

DESCRIPTION

The `pwd` command prints the current node path, the current node is produced by the `cd` command.

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

PARAMETERS

`[-h | --help]`
Provides command usage

2.2.5. `ls` command

NAME

`ls` - list the content of a node

SYNOPSIS

`ls [-h | --help] [-d | --depth] path`

DESCRIPTION

The `ls` command displays the content of a node. By default it lists the content of the current node. It accepts 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/
+-children
| +-/workspace
| +-/contents
| +-/Users
| +-/gadgets
| +-/folder
```

PARAMETERS

`[-h | --help]`
Provides command usage

`[-d | --depth]`
The depth of the printed tree

`path`
The path of the node content to list

2.2.6. *cp* command

NAME

`cp` - copy a node to another

SYNOPSIS

`cp [-h | --help] source target`

DESCRIPTION

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

`[/registry]% cp foo bar`

PARAMETERS

`[-h | --help]`

Provides command usage

`source`

The path of the source node to copy

`target`

The path of the target node to be copied

2.2.7. *mv* command

NAME

`mv` - move a node

SYNOPSIS

`mv [-h | --help] source target`

DESCRIPTION

The `mv` command can move a node to a target location in the JCR tree. It command is a `<Node,Node>` command consuming a stream of node to move them

`[/registry]% mv Registry Registry2`

PARAMETERS

`[-h | --help]`

Provides command usage

`source`

The path of the source node to move, absolute or relative

`target`

The destination path absolute or relative

2.2.8. *rm* command

NAME

`rm` - remove one or several node or a property

SYNOPSIS

`rm [-h | --help] ... paths`

DESCRIPTION

The `rm` command removes a node or property specified by its path either a is executed against the JCR session, meaning that it will not be effective.

```
[/]% rm foo
```

Node /foo removed

It is possible to specify several nodes.

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

Node /foo /bar removed

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

PARAMETERS

`[-h | --help]`

Provides command usage

`... paths`

The paths of the node to remove

2.2.9. *node* command

NAME

node add - creates one or several nodes

SYNOPSIS

node [-h | --help] add [-t | --type] ... paths

DESCRIPTION

The addnode command creates one or several nodes. The command takes at least one path. Each path can be either absolute or relative, relative paths are relative to the current directory. By default the node type is the default repository node type, but the option -t or --type can be used to specify a different node type.

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

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

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

PARAMETERS

[-h | --help]
Provides command usage

[-t | --type]
The name of the primary node type to create.

... paths
The paths of the new node to be created, the paths can either be absolute or relative.

NAME

node set - set a property on the current node

SYNOPSIS

node [-h | --help] set [-t | --type] propertyName propertyValue

DESCRIPTION

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 with the -t option

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

Remove a property

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

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

PARAMETERS

[-h | --help]
Provides command usage

[-t | --type]
The property type to use when it cannot be inferred

propertyName
The name of the property to alter

propertyValue
The new value of the property

NAME

node import - imports a node from an nt file

SYNOPSIS

node [-h | --help] import source target

DESCRIPTION

Imports a node from an nt:file node located in the workspace:

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

PARAMETERS

[-h | --help]

Provides command usage

source

The path of the imported nt:file node

target

The path of the parent imported node

NAME

node export - export a node to an nt file

SYNOPSIS

node [-h | --help] export source target

DESCRIPTION

Exports a node as an nt file in the same workspace:

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

PARAMETERS

[-h | --help]

Provides command usage

source

The path of the exported node

target

The path of the exported nt:file node

2.2.10. *mixin* command

NAME

`mixin add` - add a mixin to one or several nodes

SYNOPSIS

`mixin [-h | --help] add mixin ... paths`

DESCRIPTION

The add command adds a mixin to one or several nodes, this command is used to add a mixin from an incoming node stream, for instance:

```
[/]% select * from mynode | mixin add mix:versionable
```

PARAMETERS

`[-h | --help]`
Provides command usage

`mixin`
the mixin name to add

`... paths`
the paths of the node receiving the mixin

NAME

`mixin remove` - removes a mixin from one or several nodes

SYNOPSIS

`mixin [-h | --help] remove mixin ... paths`

DESCRIPTION

The remove command removes a mixin from one or several nodes, this command is used to remove a mixin from an incoming node stream, for instance:

```
[/]% select * from mynode | mixin remove mix:versionable
```

PARAMETERS

`[-h | --help]`
Provides command usage

`mixin`
the mixin name to remove

`... paths`
the paths of the node receiving the mixin

2.2.11. *select* command

NAME

`select` - execute a JCR sql query

SYNOPSIS

`select [-h | --help] [-o | --offset] [-l | --limit] [-a | --all] ... query`

DESCRIPTION

Queries in SQL format are possible via the `##select##` command. You can use the specification and add options to control the number of results returned 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/workspace
+--/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:

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

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

PARAMETERS

`[-h | --help]`

Provides command usage

`[-o | --offset]`

The offset of the first node to display

`[-l | --limit]`

The number of nodes displayed, by default this value is equals to 5

`[-a | --all]`

Display all the results by ignoring the limit argument, this should be used with caution

`... query`

The query, as is

2.2.12. *xpath* command

NAME

`xpath` - execute a JCR xpath query

SYNOPSIS

`xpath [-h | --help] [-o | --offset] [-l | --limit] [-a | --all] query`

DESCRIPTION

Executes a JCR query with the xpath dialect, by default results are limited

PARAMETERS

`[-h | --help]`

Provides command usage

`[-o | --offset]`

The offset of the first node to display

`[-l | --limit]`

The number of nodes displayed, by default this value is equals to 5

`[-a | --all]`

Display all the results by ignoring the limit argument, this should

`query`

The query

2.2.13. *commit* command

NAME

`commit` - saves changes

SYNOPSIS

`commit [-h | --help] path`

DESCRIPTION

Saves the changes done to the current session. A node can be provided to commit this nodes and its descendants only.

PARAMETERS

`[-h | --help]`

Provides command usage

`path`

The path of the node to commit

2.2.14. *rollback* command

NAME
rollback - rollback changes

SYNOPSIS
rollback [-h | --help] path

DESCRIPTION
Rollbacks the changes of the current session. A node can be provided to this nodes and its descendants only.

PARAMETERS
[-h | --help]
Provides command usage

path
the path to rollback

2.2.15. *version* command

NAME
version checkin - checkin a node

SYNOPSIS
version [-h | --help] checkin path

DESCRIPTION
Perform a node checkin

PARAMETERS
[-h | --help]
Provides command usage

path
The node path to checkin

```
NAME
    version checkout - checkout a node

SYNOPSIS
    version [-h | --help] checkout path

DESCRIPTION
    Perform a node checkout

PARAMETERS
    [-h | --help]
        Provides command usage

    path
        The node path to checkout
```

2.3. 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.

2.3.1. 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 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.

2.3.2. Import a JCR node

The following command will reimport the node:

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

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

The SCP feature is experimental

3

Configuring the shell

3.1. Configuration properties

CRaSH is configured by a set of properties, these properties are defined in a configuration file. In the war file packaging, the configuration file can be found under */WEB-INF/crash/crash.properties* file of the archive. Configuration can be overridden by Java Virtual Machine system properties by using the same property name.

CRaSH properties are always prefixed by the *crash.* value

3.2. 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 *crash.ssh.keypath* parameter in the *crash.properties*. Uncomment the corresponding property and change the path to the key file.

```
#crash.ssh.keypath=/path/to/the/key/file
```

3.3. Change the ports of the telnet or SSH server

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

```
# SSH configuration
crash.ssh.port=2000
```

```
# Telnet configuration
crash.telnet.port=5000
```

3.4. Remove the telnet or SSH access

- to remove the telnet access, remove the jar file in the *WEB-INF/lib/crsh.shell.telnet-1.0.0.jar* .
- to remove the SSH access, remove the jar file in the *WEB-INF/lib/crsh.shell.ssh-1.0.0.jar* .

3.5. Configure the shell default message

The */WEB-INF/crash/commands/base/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.

3.6. Configuration the authentication

Authentication is used by the SSH server when a user authenticates. Authentication interface is pluggable and has default implementations. The [Section 4.1, “Pluggable authentication ”](#) explains how to write a custom authentication plugin, in this section we cover the configuration of the authentication.

The configuration of the authentication plugin is done via property, this is necessary because several plugins can be detected by CRaSH, and the plugin is selected via the property *crash.auth* that must match the authentication plugin name:

```
crash.auth=simple
```

CRaSH comes out of the box with two authentication plugins.

3.6.1. Simple authentication

Simple authentication provides a simple username/password authentication configured with the *crash.auth.simple.username* and *crash.auth.simple.password* properties:

```
# Authentication configuration
crash.auth=simple
crash.auth.simple.username=admin
crash.auth.simple.password=admin
```

3.6.2. Jaas authentication

Jaas authentication uses jaas to perform authentication configured with the *crash.auth.jaas.domain* property to define the jaas domain to use when performing authentication:

```
# Authentication configuration
crash.auth=jaas
crash.auth.jaas.domain=gatein-domain
```

Extending the shell

4.1. Pluggable authentication

Creating a custom is done by implementing a CRaSH plugin that provides an implementation of the `AuthenticationPlugin` interface, let's study the *simple* authentication plugin implementation.

The `AuthenticationPlugin` is the interface to implement to integrate CRaSH with an authentication mechanism:

```
public interface AuthenticationPlugin {

    /**
     * Returns the authentication plugin name.
     *
     * @return the plugin name
     */
    String getName();

    /**
     * Returns true if the user is authenticated by its username and password.
     *
     * @param username the username
     * @param password the password
     * @return true if authentication succeeded
     * @throws Exception any exception that would prevent authentication to happen
     */
    boolean authenticate(String username, String password) throws Exception;
}
```

The integration as a CRaSH plugin mandates to extend the class `CRaSHPlugin` with the generic type `AuthenticationPlugin`:

```

public class SimpleAuthenticationPlugin extends
    CRaSHPlugin<AuthenticationPlugin> implements
    AuthenticationPlugin {

    public String getName() {
        return "simple";
    }

    @Override
    public AuthenticationPlugin getImplementation() {
        return this;
    }

    ...
}

```

- The `getName()` method returns the *simple* value that matches the *crash.auth* configuration property
- The `getImplementation()` method returns the object that implements the `AuthenticationPlugin` class, this method is implemented from the `CRaSHPlugin` abstract class, in our case it simply returns `this` as the plugin and the implementation of `AuthenticationPlugin` are the same class

Now let's study how the plugin retrieves the configuration properties `crash.auth.simple.username` and `crash.auth.simple.password`:


```

public class SimpleAuthenticationPlugin extends
    CRaSHPlugin<AuthenticationPlugin> implements
    AuthenticationPlugin {

    public static final PropertyDescriptor<String> SIMPLE_USERNAME =
        PropertyDescriptor.create(
            "auth.simple.username",
            "admin",
            "The username");

    public static final PropertyDescriptor<String> SIMPLE_PASSWORD =
        PropertyDescriptor.create(
            "auth.simple.password",
            "admin",
            "The password");

    @Override
    protected Iterable<PropertyDescriptor<?>> createConfigurationCapabilities() {
        return Arrays.<PropertyDescriptor<?>>asList(
            SIMPLE_USERNAME,
            SIMPLE_PASSWORD);
    }

    private String username;

    private String password;

    @Override
    public void init() {
        PluginContext context = getContext();
        this.username = context.getProperty(SIMPLE_USERNAME);
        this.password = context.getProperty(SIMPLE_PASSWORD);
    }

    ...
}

```

- The `createConfigurationCapabilities()` method returns the constants `SIMPLE_USERNAME` and `SIMPLE_PASSWORD` that defines the configuration properties that the plugin uses
- The `init()` method is invoked by CRaSH before the plugin will be used, at this moment, the configuration properties are retrieved from the plugin context with the method `getContext()` available in the `CRaSHPlugin` base class

Finally the plugin needs to provide the `authenticate()` method that implement the authentication logic:

```
public boolean authenticate(String username, String password)
    throws Exception {
    return this.username != null &&
        this.password != null &&
        this.username.equals(username) &&
        this.password.equals(password);
}
```

The logic is straightforward with an equality check of the username and password.

Last but not least we must declare our plugin to make it recognized by CRaSH, this is achieved thanks to the `java.util.ServiceLoader` class. CRaSH uses the `ServiceLoader` for loading plugins and the loader needs a file to be present in the jar file containing the class under the name `META-INF/services/org.crsh.plugin.CRaSHPlugin` containing the class name of the plugin:

```
org.crsh.auth.SimpleAuthenticationPlugin
```

When all of this is done, the plugin and its service loader descriptor must be package in a jar file and available on the classpath of CRaSH.

You can learn more about the `java.util.ServiceLoader` by looking at the online [javadoc](#)

Developers

5.1. Developping commands

A CRaSH command is written in the Groovy language. The Groovy language provides several signifiant advantages:

- Commands can be bare scripts or can be a class
- Java developers can write Groovy commands without learning it
- Groovy is dynamic and expressive

Each command has a corresponding Groovy file that contains a command class that will be invoked by the shell. The files are located in

- In the standalone distribution the *cmd* directory
- In a web archive deployment the */WEB-INF/crash/commands* directory

New commands can directly be placed in the commands directory however they can also be placed in a sub directory of the command directory, which is useful to group commands of the same kind.

In addition of that there are two special files called *login.groovy* and *logout.groovy* that are executed upon login and logout of a user. They are useful to setup and cleanup things related to the current user session.

5.1.1. Commands as a script

The simplest command can be a simple script that returns a string

```
return "Hello World";
```

It is possible to use also the `out` implicit variable to send a message to the console:

```
out.println("Hello World");
```

5.1.2. Commands as a class

Class can also be used to defined a command, it provides significant advantages over scripts:

- Commands can declare options and arguments for the command
- Commands can use annotations to describe the command behavior and parameters

When the user types a command in the shell, the command line is parsed by the *cmdline* framework and injected in the command class. Previously the *args4j* framework was used but this framework does not support natively code completion and could not be extended to support it. The support of command line completion is the main motivation of the development of such a framework.

Let's study a simple class command example:

```
class date extends CRaSHCommand {
    @Usage("show the current time")
    @Command
    Object main(@Usage("the time format") @Option(names=["f","format"]) String format) {
        if (format == null)
            format = "EEE MMM d HH:mm:ss z yyyy";
        def date = new Date();
        return date.format(format);
    }
}
```

The command is pretty straightforward to understand:

- The `@Command` annotation declares the `main` method as a command
- The command takes one optional `format` option
- The `@Usage` annotation describes the usage of the command and its parameters

```
% date
Thu Apr 19 15:44:05 CEST 2012
```

The `@Usage` annotation is important because it will give a decent human description of the command

```
% date -h
usage: date [-h | --help] [-f | --format]

    [-h | --help]    command usage
    [-f | --format] the time format
```

5.1.3. Multi commands

A class can hold several commands allowing a single file to group several commands, let's study the JDBC command structure:

```

@Usage("JDBC connection")
class jdbc extends CRaSHCommand {

    @Usage("connect to database with a JDBC connection string")
    @Command
    public String connect(
        @Usage("The username") @Option(names=["u","username"]) String user,
        @Usage("The password") @Option(names=["p","password"]) String password,
        @Usage("The extra properties") @Option(names=["properties"]) Value.Properties,
        @Usage("The connection string") @Argument String connectionString) {
        ...
    }

    @Usage("close the current connection")
    @Command
    public String close() {
        ...
    }
}

```

We can see that the class declares two commands `connect` and `close`, they are invoked this way:

```

% jdbc connect jdbc:derby:memory:EmbeddedDB;create=true
Connected to data base : jdbc:derby:memory:EmbeddedDB;create=true
% jdbc close
Connection closed

```

5.2. Inter command API

In this section we study how a command can reuse existing commands, here is an example

Example 5.1. dbscript.groovy

```

jdbc.connect username:root, password:crash, "jdbc:derby:memory:EmbeddedDB;create=true"
jdbc.execute "create table derbyDB(num int, addr varchar(40))"
jdbc.execute "insert into derbyDB values (1956,'Webster St.')"
jdbc.execute "insert into derbyDB values (1910,'Union St.')"
jdbc.execute "select * from derbyDb"
jdbc.close

```

This script is written in Groovy and use Groovy DSL capabilities, let's study the first statement:

- the `jdbc.connect` statement can be decomposed into two steps
 - the `jdbc` is resolved as the command itself
 - the `connect` invokes the `connect` command
- the `username` and `password` are considered as command options

- the SQL statement "jdbc:derby:memory:EmbeddedDB;create=true" is the main argument of the command

It is equivalent to the shell command:

```
% jdbc connect --username root --password crash jdbc:derby:memory:EmbeddedDB;create=true
```

The rest of the script is fairly easy to understand, here is the output of the script execution:

```
% dbscript
Connected to data base : jdbc:derby:memory:EmbeddedDB;create=true
Query executed successfully
Query executed successfully
Query executed successfully
NUM          ADDR
1956         Webster St.
1910         Union St.
Connection closed
```

Hey, I want to contribute!

Drop me an email (see my @ on www.julienviet.com), any kind of help is welcome.