Introduction a CRaSH : application à la visualisation d'un cache EHcache

CRaSH : a shell to extend Java Platform

par Rieu Damien (Contributeur sur CRaSH)

Date de publication : 1 novembre 2012

Dernière mise à jour : 11 novembre 2012

TOUT PUBLIC

Durée : 15 mn

CRaSH est un projet open source récent créé par Julien **Viet** (coprésident du **Marseille JUG**). J'ai découvert CRaSH au Marseille JUG lors d'une session sur CRaSH et depuis peu j'y contribue.
Avec cet article, je souhaite vous faire découvrir CRaSH à travers un cas pratique et vous montrer comment on peut facilement agir directement au cœur de la JVM. Bon CRaSH !

I - Introduction

CRaSH permet de se connecter à une JVM en mode Shell puis d'exécuter des commandes directement sur cette JVM. Ainsi, nous allons accéder à un certain nombre de commandes prédéfinies (exemple la commande thread, jdbc, java…). Si vous souhaitez avoir un aperçu rapide sur les commandes disponibles et leurs utilisations, vous pouvez les tester directement sur le site de démo : **Démo en ligne**
Une des grandes forces de **CRaSH** est que l'on peut aussi définir ses propres commandes Shell par programmation. Il est alors possible de réaliser des commandes spécifiques à nos besoins !
Dans ce tutoriel, nous allons présenter **CRaSH** grâce à un cas pratique. Pour cela, nous allons réaliser une commande **CRaSH** d'affichage d'un cache puis nous intègrerons cette commande dans notre application.
Nous montrerons ensuite comment utiliser cette commande à l'aide de **CRaSH**.
L'application démo a été réalisée avec Spring. Dans cet article, nous montrerons à quel point il est facile d'intégrer CRaSH dans une application Spring.

II - Prérequis et installation

II-A - Prérequis

- Jboss 7.1.1.Final.
- Maven 3.
- Le code de l'appli se trouve sur GitHub : Source **https://github.com/crashub/spring-ehcache-demo**.

II-B - Installation

L'installation se résume à récupérer l'archive de **CRaSH** (**www.crashub.org)** et à la décompresser. Elle n'est pas obligatoire si vous utilisez le mode ssh ou Telnet.

II-C - Installation de l'application démo

- Téléchargez le code source de l'appli sur **GitHub :** Source **https://github.com/crashub/spring-ehcache-demo**.
- Construction du war :

```
1. cd sandbox
2. mvn clean package
```

- Déploiement dans JBoss

Une fois le war construit, il ne vous reste plus qu'à le copier dans le répertoire **deployments** de JBoss (${JBOSS_DIR}/standalone/deployments).

L'application démo n'a été testée que sur jboss-as-7.1.1.Final.

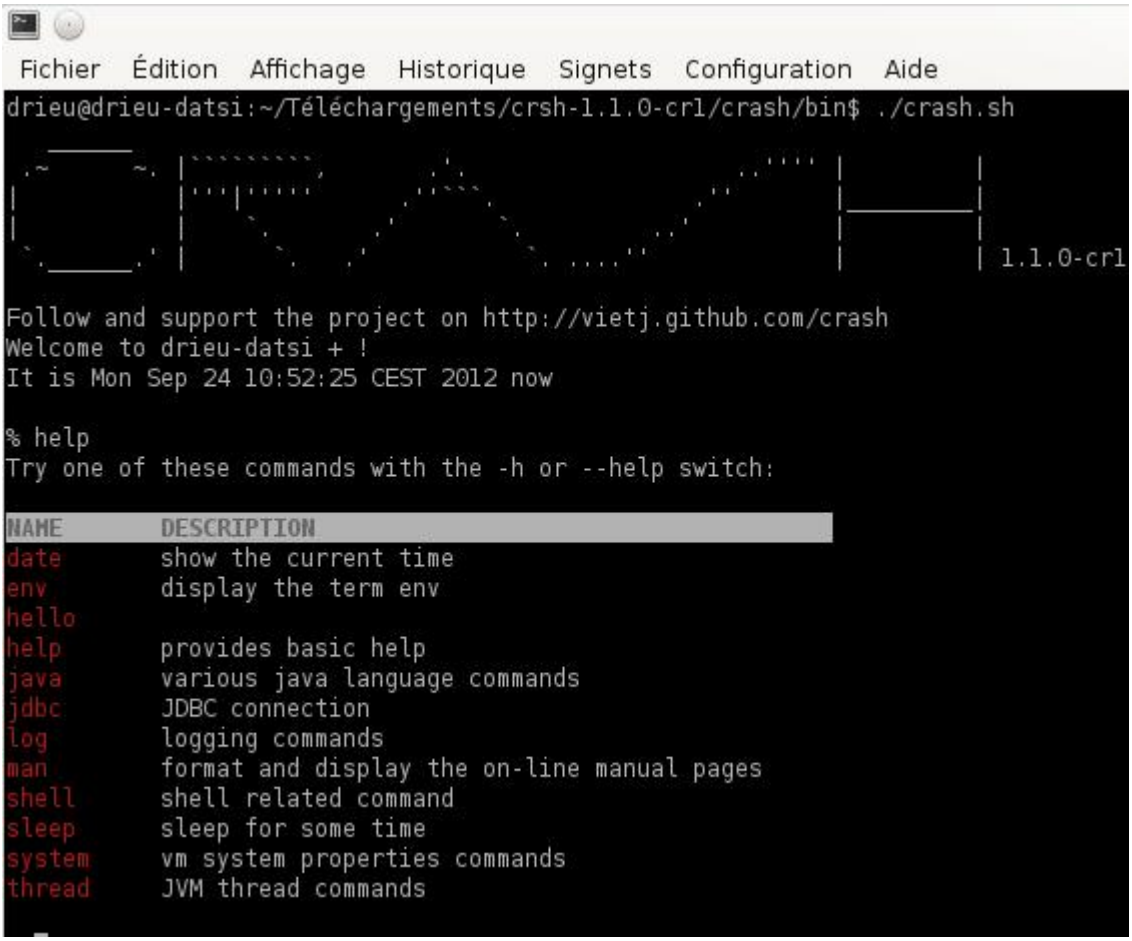
III - Présentation de CRaSH

Voici la définition officielle de **CRaSH** :

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.

En résumé, **CRaSH** déploie dans un runtime Java et permet d'interagir avec la JVM.
Avec **CRaSH**, nous serons capables d'accéder à la JVM au moyen d'un Shell. Nous pourrons donc exécuter des commandes pour voir les threads, les classes, se connecter aux sources de données disponibles et exécuter des requêtes comme si on était à la place de l'application !

Voici une liste de commandes disponibles :



Ce sont les commandes de base. Cependant, une des grandes forces de **CRaSH**, c'est que vous pouvez réaliser vos propres commandes Shell en **Groovy** (si vous ne connaissez pas **Groovy**, ce n'est pas grave car **Groovy** reconnait la syntaxe Java).

Dans notre exemple démo, nous allons réaliser un script **Groovy** qui va lire les informations contenues dans le cache.

IV - Application démo

IV-A - Objectif

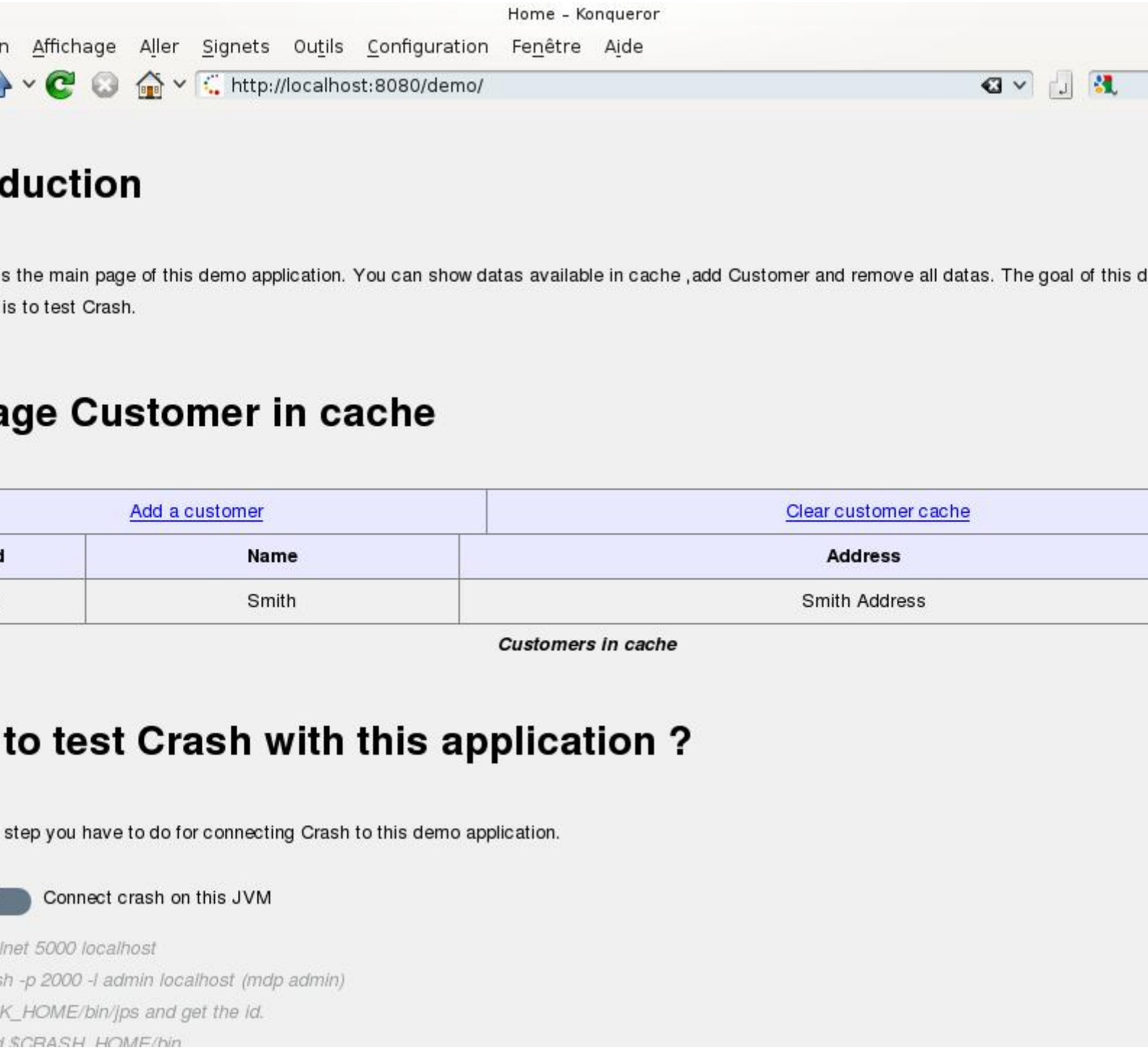
L'objectif est de contrôler le cache de l'application démo en se connectant à la JVM de cette application à l'aide de CRaSH et en exécutant un script d'affichage du cache.
Voici les étapes à réaliser :

- création d'un script d'affichage du cache : spring_cache.groovy ;

- intégrer CRaSH dans notre application démo (ajout des dépendances Maven, de spring_cache.groovy et des fichiers de configuration) ;
- packaging et déploiement de notre application démo sur JBoss ;
- Utilisation de CRaSH.

IV-B - Présentation de l'application démo

L'application démo est très simple. Elle donne la possibilité d'ajouter, de lister et d'effacer les utilisateurs. Pour accéder à l'application démo, il suffit de se rendre sur **http://localhost:8080/demo/** et vous arriverez sur la page suivante :



IV-C - Création d'un script d'affichage du cache : spring_cache.groovy

Pour afficher les données présentes dans notre cache, nous avons besoin d'un script. Ce script sera ensuite placé dans le répertoire **WEB-INF/crash/commands/** de notre application démo.

*Dans cette version (crsh.shell.embed.spring 1.1), il est obligatoire de mettre les scripts dans le répertoire **WEB-INF/crash/commands/** .*

IV-C-1 - Comment fonctionne ce script ?

Ce script va récupérer le bean **customerComponent** qui est présent dans le contexte.
Il est alors possible de récupérer la liste des Customers en appelant la méthode getCache() de ce bean : *List<Customer> lst = bean.getCache();*
Ensuite, le script boucle sur la liste des Customers et affiche leurs informations.

IV-C-2 - Script d'affichage du cache

Ce script va définir une commande CRaSH qui sera accessible en mode Shell :

- le nom de cette commande correspond au nom de la classe Groovy. Dans notre cas, c'est **spring_cache** ;
- cette commande peut avoir plusieurs sous-commandes. Dans notre cas, nous avons défini une sous-commande **showCache**.

Donc pour appeler showCache, il suffira de saisir dans le shell de CRaSH :

```
%spring_cache showCache
```

Voici le code de la commande spring_cache :

```
package crash.commands.base
import org.crsh.command.CRaSHCommand
import org.crsh.cmdline.annotations.Usage
import org.crsh.cmdline.annotations.Command
import org.crsh.cmdline.annotations.Argument

import org.crsh.cmdline.annotations.Required
import org.crsh.shell.ui.UIBuilder

import fr.dr.sandbox.controller.Customer;
import fr.dr.sandbox.controller.CustomerComponent;
import java.lang.reflect.Method;
import java.util.List;

/**
* Spring commands to list Customer in cache for the demo application.
* @author drieu
*
*/
@Usage("Spring cache commands")
@Command
class spring_cache extends CRaSHCommand {

 @Usage("Show values that are loaded in cache")
 @Command
 public void showCache() {
   def bean = context.attributes.beans["customerComponent"];
   if (null != bean) {
     out.println("Cache contents :");
     List<Customer> lst = bean.getCache();
     if (null != lst) {
     for(Customer c : lst) {
```

```
            out.println("Name:" + c.getName());
            out.println("Id:" + c.getId());
            c.show();
        }
    }
  }
  }
}
```

Voir Annexes V.E pour plus de détails sur la création d'une commande.

Cette commande est écrite en Groovy. Cependant, la syntaxe que l'on utilise ici est celle de Java.

IV-C-3 - Intégrer CRaSH dans notre application démo

L'intégration de **CRaSH** dans une application est très simple.
Elle consiste seulement à :

- ajouter la dépendance **Maven de CRaSH** ;
- ajouter un bean de configuration ;
- ajouter le script spring_cache.groovy créé précédemment ;
- packager le tout dans un war.

*Pour que ce script soit pris en compte, il devra être placé dans **WEB-INF/crash/commands/**. Vous pouvez bien entendu créer autant de scripts que vous le souhaitez et les mettre dans ce répertoire.*

Dans certains cas, il est possible de modifier, supprimer ou rajouter des commandes dynamiquement. La commande est alors rechargée automatiquement !

IV-C-4 - Ajout des dépendances Maven

Dans votre pom.xml :

pom.xml

```
<dependency>
    <groupId>org.crsh</groupId>
       <artifactId>crsh.shell.embed.spring</artifactId>
       <version>1.1.0</version>
</dependency>
<dependency>
   <groupId>org.crsh</groupId>
   <artifactId>crsh.shell.core</artifactId>
   <version>1.1.0</version>
</dependency>
<dependency>
   <groupId>org.crsh</groupId>
   <artifactId>crsh.shell.packaging</artifactId>
   <version>1.1.0</version>
   <type>war</type>
   <classifier>spring</classifier>
</dependency>
```

CRaSH utilise la version 3.1.1.RELEASE de Spring mais cela devrait fonctionner avec toutes les versions de Spring.

IV-C-5 - Configuration des modes d'accès à CRaSH

Cette étape est facultative car vous pouvez vous connecter directement sur la JVM(cf. Annexes pour les modes de connexions).
Par contre, si vous souhaitez utiliser le mode ssh ou Telnet, il suffit pour cela de modifier un bean (voir fichier **spring.xml** de l'application démo).
Dans l'exemple ci-dessous, nous configurons un accès Telnet par le port 5000 et nous ajoutons aussi une connexion ssh sur le port 2000 avec l'utilisateur admin.
Exemple :

```
<bean class="org.crsh.spring.SpringWebBootstrap">
   <property name="config">
   <props>
   <prop key="crash.telnet.port">5000</prop>
   <!-- VFS configuration -->
   <prop key="crash.vfs.refresh_period">1</prop>
   <!-- SSH configuration -->
   <prop key="crash.ssh.port">2000</prop>
   <!-- Authentication configuration -->
   <prop key="crash.auth">simple</prop>
   <prop key="crash.auth.simple.username">admin</prop>
   <prop key="crash.auth.simple.password">admin</prop>
   </props>
   </property>
   </bean>
   <bean class="org.crsh.telnet.TelnetPlugin"/>
```

Nous pourrons donc nous connecter à notre application de la façon suivante :

```
telnet 5000 localhost
ssh -p 2000 -l admin localhost (mdp admin)
```

IV-C-6 - Packaging et déploiement

Il suffit de construire le war en faisant :

```
mvn clean package
```

Nous déployons ensuite ce war dans JBoss. L'application est alors disponible à l'adresse **http://localhost:8080/demo/**. Dans la partie qui suit, nous allons voir comment utiliser CRaSH.

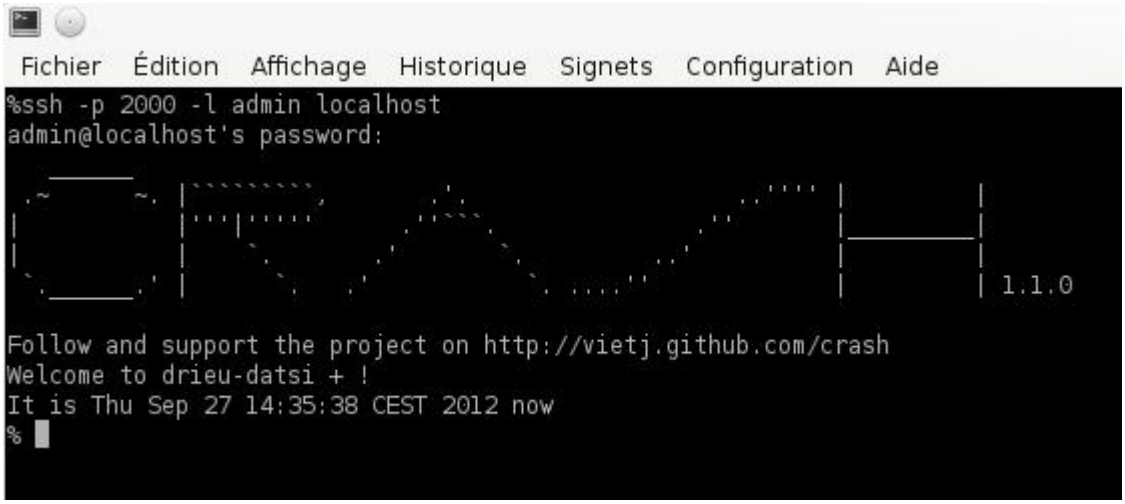
IV-D - Utilisation de CRaSH

L'utilisation de CRaSH est très simple. Pour se connecter à une JVM, il y a plusieurs solutions :

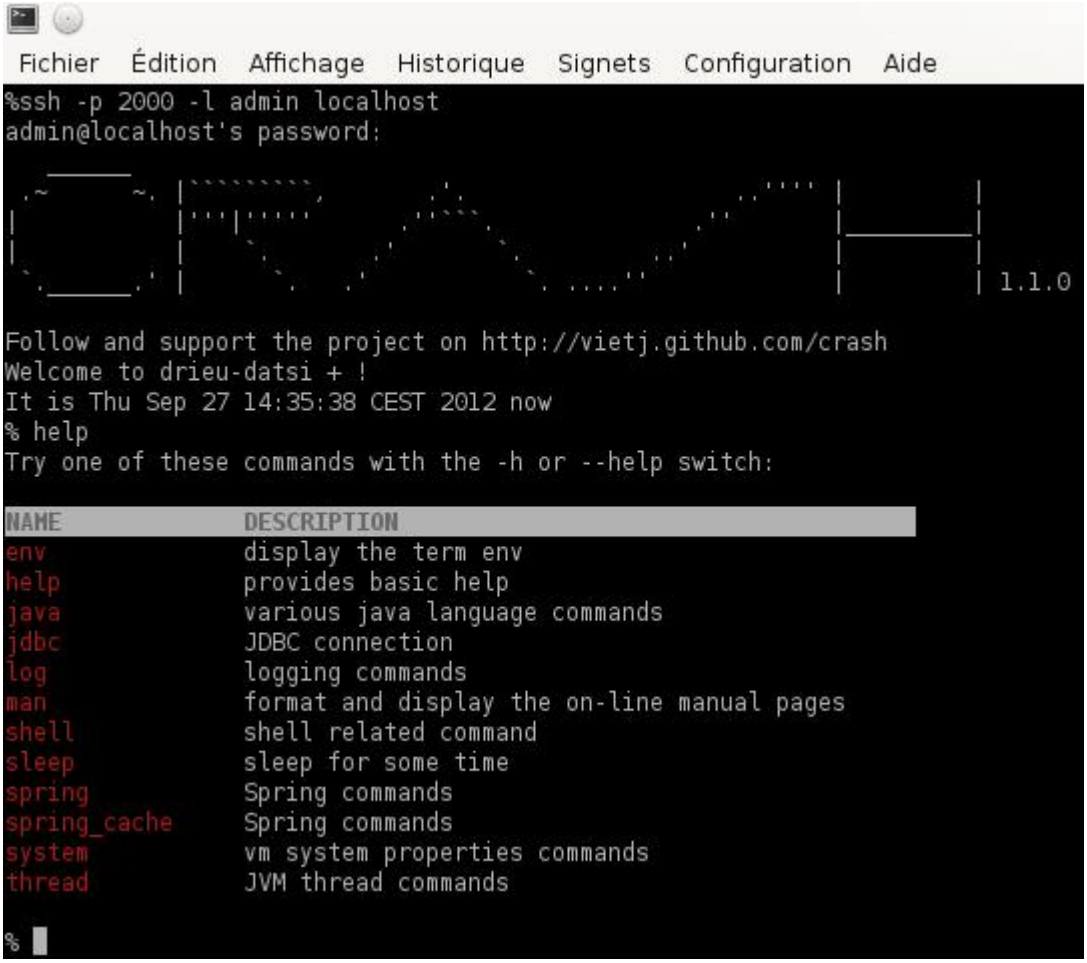
- mode ssh ;
- mode telnet ;
- mode Attach.

Dans notre exemple, nous allons utiliser le mode ssh. CRaSH étant embarqué dans l'application démo avec la configuration ssh (utilisateur admin et port 2000), il nous suffit d'avoir un client ssh pour se connecter :

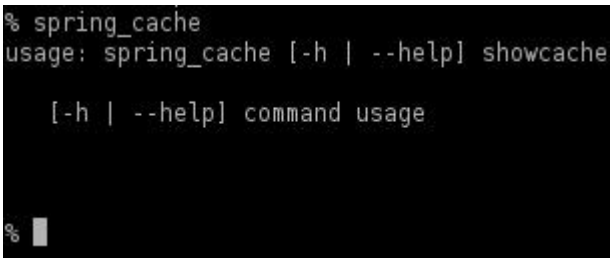
```
ssh -p 2000 -l admin localhost
```



Une fois connecté, vous pouvez saisir **help** pour avoir une liste des commandes disponibles :



Vous constatez la présence de notre commande **spring_cache**. Pour obtenir de l'aide sur cette commande, il faudra saisir : **spring_cache**.

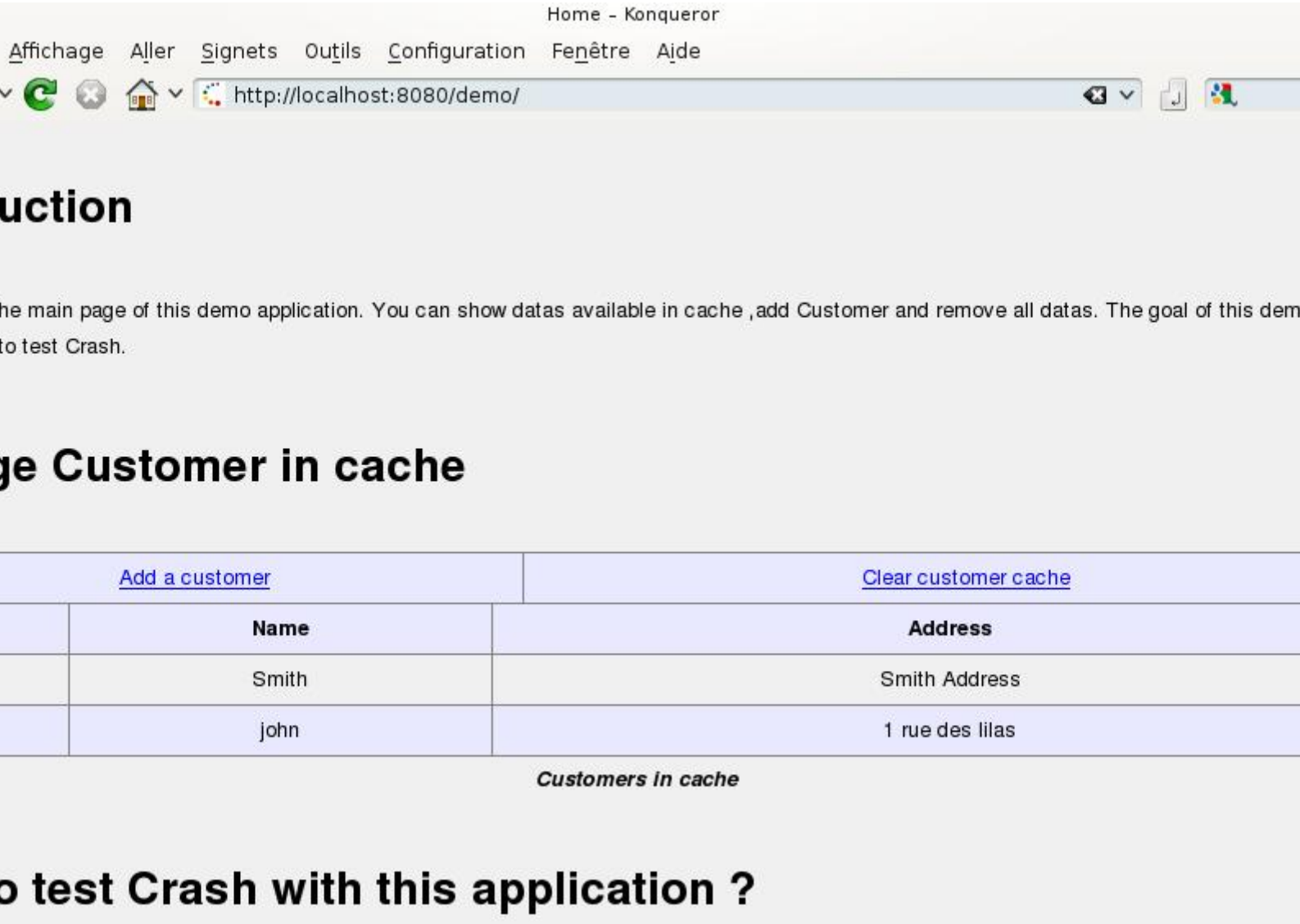


Pour utiliser notre commande, il nous suffit de faire : **spring_cache showcache**.
Cette commande va nous permettre d'afficher la liste des Customers présents dans notre cache.

Au démarrage, nous avons chargé un utilisateur dans notre cache (cf. Annexes) :

spring_cache_one
```
1. % spring_cache showcache
2. Cache contents :
3. Name:Smith
4. Id:1
```

Ensuite, nous pouvons ajouter un autre utilisateur dans notre cache. Pour cela, il suffit de cliquer sur **Add customer** sur la page **http://localhost:8080/demo/**.
Si nous retournons sur la page d'accueil après avoir ajouté le customer, nous verrons bien deux lignes dans notre tableau :



Ensuite, si on retourne dans notre Shell, on voit bien que le cache a été mis à jour :

```
1. % spring_cache showcache
2. Cache contents :
3. Name:Smith
4. Id:1
5. Name:john
6. Id:2
```

V - Annexes

V-A - Fonctionnement du cache de l'application démo

Cette partie décrit comment nous avons implémenté la gestion du cache dans l'application démo dans les deux cas suivants :

- au démarrage ;
- dans l'application.

Nous utilisons **ehcache** 1.6.1 avec **Spring** 3.1.1.RELEASE dans notre application démo.
Pour cela, nous définissons un cacheManager avec **Spring** dans le fichier sandbox.xml :

```
<ehcache:annotation-driven cache-manager="cacheManager" />
```

Nous utilisons ensuite ce cacheManager dans notre application pour effectuer des opérations de lecture, écriture, effacement. Nous accédons à ce cache de deux manières :

- au démarrage ;
- par l'interface web.

V-B - Chargement du cache au démarrage

Nous remplissons le cache au démarrage avec un **Customer**. Pour cela, nous avons défini un listener dans notre web.xml :

web
```
1. <listener-class>
2. fr.dr.sandbox.listener.CacheListener
3. </listener-class>
```

Ce listener contient une méthode contextInitialized dans laquelle nous avons placé notre code qui va charger un **Customer** en cache.

AddInCache
```
1. CacheManager cacheManager = (CacheManager)
 WebApplicationContextUtils.getWebApplicationContext(servletContext).getBean("cacheManager");
2. Cache cache = cacheManager.getCache("customer");
3. Customer c = new Customer();
4. c.setId("1");
5. c.setName("Smith");
6. c.setAddress("Smith Address");
7.
8. cache.put(new Element(c.getId(),c));
```

Au démarrage de l'application, on voit que ce chargement est bien effectué dans les logs :

log
```
1. 14:53:32,044 INFO  [fr.dr.sandbox.listener.CacheListener] (MSC service thread 1-2) ========>
 contextInitialized() : BEGIN.
2. 14:53:32,047 INFO  [fr.dr.sandbox.listener.CacheListener] (MSC service thread 1-2) ========>
 contextInitialized() : Customer Smith was added in cache.
3. 14:53:32,047 INFO  [fr.dr.sandbox.listener.CacheListener] (MSC service thread 1-2) ========>
 contextInitialized() : END
```

V-C - Accès au cache par l'interface web

Il est possible de réaliser des opérations sur le cache par l'interface web. Pour cela, nous définissons les opérations dans le Controller qui vont appeler les méthodes de gestion du cache qui se trouvent dans **CustomerComponent**.

Exemple :

L'accès à **http://localhost:8080/demo/clearCache** va appeler la méthode suivante du Controller :

clearCache
```
1. /**
2. * Clear all data in cache.
3. * @return String text message OK or KO.
```

clearCache
```
4. */
5. @RequestMapping(value = "/clearCache", method = { RequestMethod.GET })
6. @ResponseStatus(HttpStatus.OK)
7. public @ResponseBody
8. String clearCache() {
9.       customerComponent.clearCache();
10.       return "Cache successfully Cleaned";
11. }
```

V-D - Les modes de connexions

CRaSH fournit plusieurs solutions pour se connecter sur des JVM. Nous allons présenter dans cette partie les différents modes.

V-D-1 - Mode standalone

Ce mode permet de démarrer CRaSH directement à partir de la commande Shell :

```
%cd $CRASH_HOME/bin
%crash.sh
```

Il permet de démarrer CRaSH sur sa propre JVM.

V-D-2 - Mode Attach

Ce mode permet de connecter CRaSH à une JVM disponible en local.
Pour connaître la liste des JVM disponibles en local, il existe la commande jps disponible dans le JDK :

```
%$JDK_HOME/bin/jps
3165 RemoteMavenServer
20650 Test
20651 Jps
```

Pour se connecter, il suffira de passer le JVM PID à CRaSH :

```
%cd $CRASH_HOME/bin
%crash.sh 20650
```

V-D-3 - Mode embarqué

CRaSH peut être embarqué dans une application web déployée dans un serveur d'application. C'est le cas dans notre application démo.

V-D-4 - Autres modes

CRaSH est aussi livré sous la forme d'un war que l'on déploie sur un serveur d'application.
Ainsi, on aura accès à la JVM du serveur d'application et on pourra par exemple se connecter aux sources de données disponibles et exécuter des requêtes…
Pour se connecter au CRaSH déployé sur ce serveur d'application, il y a plusieurs méthodes :

- mode ssh ;
- mode telnet ;
- mode attach.

V-E - Créer sa commande CRaSH

Il existe deux possibilités pour créer une commande :

- en tant que Script.
  Dans ce cas, vous écrivez directement votre code dans votre script Groovy. Voici l'exemple de la commande clock :

```
package crash.commands.base
for (int i = 0;i < 10;i++) {
    out.cls();
    out << "Time is" + i + "\n";
    out.flush();
    Thread.sleep(1000);
}
```

- en tant que classe.

Cette méthode simplifie grandement la création de commande. Dans cette partie, nous présenterons seulement les annotations de base nécessaires à la création d'une commande.
Avec les commandes CRaSH, vous avez énormément de possibilités. Vous pouvez par exemple ajouter de la complétion, créer des tableaux de résultats…
Pour plus d'informations, allez voir la documentation sur **www.crashub.org**.

Voici la structure d'une commande :

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

  @Usage("connect to database with a JDBC connection string")
  @Command
  public String connect(
 ...
  }
```

@Command : permet de définir qu'une méthode ou une classe est une commande ;

@Usage : permet de définir l'aide qui sera affichée.

Si elle est placée au-dessus de la classe qui extends Command alors cette aide sera affichée avec la commande help.

Dans notre cas, lorsque l'on saisira la commande help, on aura la sortie suivante avec l'annotation *@Usage("JDBC connection")* :

```
%help
Try on of the following commands with -h or --help switch:

NAME    DESCRIPTION
env     dipslay the term env
help    provides basic help
java    various java language commands
jdbc    JDBC connection
```

Nous pouvons aussi placer cette annotation au-dessus d'une méthode identifiée comme Command. Par exemple, pour la méthode connect de JDBC, nous avons :@Usage("connect to database with a JDBC connection string"). Donc si on saisit :

```
%jdbc help
```

on retrouvera alors cette description :

```
The most commonly used jdbc commands are:
   props            show the database properties
   close            close the current connection
   table            describe the tables
   open             open a connection from JNDI bound datasource
   connect          connect to database with a JDBC connection string
   execute          execute SQL statement
   info             describe the database
   tables           describe the tables
```

Il existe aussi des annotations pour définir les arguments (exemple : @Argument) et les options(exemple :@Option(names=["p","password"]).

VI - Conclusion et remerciements

J'espère que cet article vous aura fait découvrir CRaSH et vous aura donné envie d'en savoir plus sur CRaSH. Si vous avez des questions, des remarques et des suggestions, n'hésitez pas à les poster sur le Google Groups de CRaSH :**crash-users@googlegroups.com**.
Je remercie **Mickael BARON** pour sa patience et la qualité de ses observations ainsi que **Claude Leloup** pour sa relecture attentive et assidue. Je remercie aussi Julien **Viet** et Romain **Linsolas** pour leur relecture.