MGS plugin framework manual

Antoine Forgerou David Drevet Jérémy Bardon

Sommaire

Résumé

Le framework MGS permet de gérer de manière transparente la modularité des applications basées sur l'utilisation de plugins.

Plutot que d'obliger les utilisateurs à remplir de long fichiers de configurations, nous avons choisi le plus possible de nous concentrer sur une architecture de fichier la moins contraignante possible.

1 Organisation du framework

1.1 Configuration

Il existe un seul fichier de configuration qui permet d'ajuster le fonctionnement du framework. Ce fichier se nomme *settings.txt* et se trouve dans le répertoire *resources* au même niveau que les dossier *src* regroupant le coeur du framework.

pluginspath	Chemin vers le dossier qui contient tout les plugins.
startplugin	Nom du plugin [1] à charger au démarrage. Obligatoirement runnable.

TABLE 1 – Paramètres du fichier settings.txt

Ce répertoire contient également un dossier *plugins* qui rassemble les fichiers *jar* des plugins. La valeur par défaut de *pluginspath* est donc *resources/plugins*.

1.2 Plugins

Le dossier plugin renseigné par *pluginpath* est le répertoire qui contient tout les plugins que le framework pourra charger. On distingue deux types de plugins :

Runnable Plugin principal qui peut etre démarré par le framework s'il est renseigné comme *startplugin* dans le fichier de configuration du framework.

Classic Plugin annexe qui fourni des classes respectant les interfaces définies par un ou plusieurs plugin(s) runnable qui l'utiliseront.

1.2.1 Fichier de configuration

Chaque plugin – quelque soit son type – doit respecter une certaine architecture en termes d'organisation de son dossier. En effet, un plugin doit obligatoirement fournir un fichier *plugin.txt* qui le décrit et donne des informations sur comment il peut etre utilisé.

runnable	Indique si le plugin est un plugin runnable ou classic. Peut être égal à *0/1* ou *true/false*.
category	Classe ou interface que respecte le plugin. Pour un plugin runnable ce sera toujours *fr.univnantes.snake.framwork.MGSApplication* qui sera rempli automatiquement par le framework. Dans le cas d'un plugin classique ce sera une interface définie par un plugin runnable.
mainClass	Classe principale qui sera chargée par le framework. Elle doit implémenter ou hériter de celle donnée par la categorie.

TABLE 2 – Paramètres d'un fichier plugin.txt

Le dépot git du projet [2] regroupe les plugins développés par l'équipe (répertoire plugins), on peut y trouver le plugin *template* qui donne une base pour créer un plugin.

1.2.2 Détail sur les plugins runnable

Les plugins principaux sont capables de définir des interfaces afin de s'assurer que les plugins qu'ils utiliseront répondent à leurs besoins.

Plutôt que de donner la liste des interfaces dans le fichier de configuration – ce qui lourd – nous avons choisi d'obliger l'utilisateur à placer ces interfaces dans un sous-package *interfaces*. Ceci est moins contraignant en plus du fait que des utilisateurs organisées les auraient de toutes façons rassemblés dans un package à part.

2. Projet hébergé par GitHub : https ://github.com/masters-info-nantes/snake

2 Utilisation

2.1 Créer un plugin principal

Pour commencer, il est nécessaire de vérifier où le framework va chercher les plugins en regardant le fichier *settings.txt*. Dans notre exemple ce sera */home/user/plugins* mais il peut se trouver à n'importe quel endroit à condition d'avoir les droits de lecture et un chemin par trop exotique avec des espaces et des caractères accentués.

Créons le plugin *hello* qui aura pour but de dire bonjour dans différentes langues selon le plugin secondaire qui sera utilisé.

```
$ cd plugins
$ cp -R templates/runnable hello

$ cd hello/src/main/java/
$ mv runnableplugin hello
$ mv hello/RunnablePlugin.java Hello.java

$ vim Hello.java
$ cd hello
$ vim pom.xml
$ vim src/main/resources/plugin.txt
$ mvn package
$ cp target/hello-0.1.jar /platform/plugins

$ mvn install
```

Listing 1 – Création du plugin hello

Le fichier de configuration indique que le plugin *hello* est un plugin runnable et que classe à lancer (qui implémente MGSApplication) est *Hello*.

```
runnable=1
mainClass=hello.Hello
```

Listing 2 – Hello plugin configuration

La liaison entre le framework et le plugin principal se fait à travers la relation d'héritage avec la classe MGSApplication.

En fait, nous avons fait ce choix car l'utilisation d'un singleton aurait pu donner accès au framework à n'importe quel plugin (même secondaire). De plus, si nous avions choisi de passer l'accès en paramètre de la fonction surchargée – *run* – ceci aurait obligé l'utilisateur à ajouter un attribut pour stocker l'accès au framework.

```
package hello;

import fr.univnantes.snake.framework.MGSApplication;

public class Hello extends MGSApplication{

    @Override
    public void run() {
        System.out.println("Plugin hello");
    }
}
```

Listing 3 – Hello plugin main class

La dernière chose à faire est de dire au framework de démarrer notre plugin au démarage. Pour cela, il faut renseigner le nom du plugin dans le paramètre *startplugin* du fichier de configuration du framework.

2.2 Définir des patrons de plugins

Comme expliqué précédemment (voir 1.2.2), il est possible de définir des plugins secondaires qui seront utilisés par le plugin principal. Tout d'abord, il faut commencer par définir le périmètre fonctionnel des plugins secondaires que l'on veut à travers une interface.

```
package hello.interfaces;

public interface Speak {
    public void sayHello();
}
```

Listing 4 – Speak interface for Hello plugin

La prochaine étape consiste à créer un nouveau plugin – *frenchspeak* par exemple – qui aura pour catégorie *Speak*.

```
$ cd plugins/hello/src/main/java/hello/intefaces
$ mv SecondaryPlugin.java Speak.java

$ cd plugins
$ cp templates/classic frenchspeak
$ cd frenchspeak/src/main/java
$ mv secondarypluginone frenchspeak

$ cd frenchspeak
$ vim pom.xml
$ vim src/main/resources/plugin.txt
```

```
$ mvn package
$ cp target/frenchspeak-0.1.jar /platform/plugins
```

Listing 5 – Création du plugin secondaire frenchspeak

Ce nouveau plugin aura pour classe principale *French* qui implémentera l'interface *Speak* définie dans le plugin *Hello* plus tôt.

```
mainClass=frenchspeak.French
category=hello.interfaces.Speak
```

Listing 6 – FrenchSpeak plugin configuration

```
package frenchspeak;

import hello.interfaces.Speak;

public class SpeakFrench implements Speak{

    @Override
    public void sayHello() {
        System.out.println("Bonjour");
    }
}
```

Listing 7 – FrenchSpeak plugin main class

2.3 Donner les plugins au framework

La dernière chose à faire et donner accès au framework à nos plugins. Pour cela il faut fournir le projet *plugins* sour forme de *jar* et le placer dans le répertoire *platform/ressources*.

Tout les plugins étant rassemblés dans le projet maven *plugins*, voici la marche à suivre pour mettre à jour les mettre à jour.

```
$ cd plugins
$ mvn compile
$ mvn package
$ cp target/snake-0.1.jar ../platform/ressources
```

Listing 8 – Mise à jour des plugins

2.4 Appels au framework

Etant donné qu'un plugin principal peut utiliser des plugins secondaires gérés par le framework, il est possible d'instancier des plugins secondaires à

partir d'un plugin principal.

La classe *Hello* du plugin *hello* doit implémenter la méthode *run* qui prend en paramètre un objet de type *AppContext* et c'est ce lien qui va permettre de faire des appels au framework. Cette classe possède deux attributs accessibles par le plugin (via des getters) : *currentPlugin* et *pluginLoader*. Le premier regroupe tout simplement la configuration du plugin courant – hello – mais le second donne accès au gestionnaire de plugins.

`String getPluginsPath()`
Chemin vers le dossier où se trouvent les plugins
`Collection<Plugin> getClassicPlugins()`
Configuration de tout les plugins chargés par le framework (exepté les plugins principaux).
`List<Plugin> getClassicPluginsByCategory(String category)`
Tout les plugins secondaires ayant la catégorie donnée
`Set<String> getClassicPluginsList()`
Nom de tout les plugins chargés par le framework (exepté les plugins principaux).
`Object loadPlugin(Plugin plugin) throws IOException`
Charge le plugin donné en paramètre. Cette fonction assure que le plugin donné est bien une sous-classe de sa catégorie.

Table 3 – Méthodes du gestionnaire de plugins

3 Fonctionnement interne

Lors d'une première phase, le framework va lire son fichier de configuration – *settings.txt* – pour trouver le chemin vers le dossier des plugins. Ensuite, il

va scanner les jars, lire le fichier de configuration *plugins.txt* et stocker la liste des plugins dont la configuration est valide.

> **Attention !** Le framework indique au démarrage pour chaque plugin s'il a été chargé ou non.
>
> Certaines propriétés du fichier *plugin.txt* sont déduites par le framework : *runnable* est faux par défaut et ce n'est pas obligé de donner la *category* pour un plugin principal car c'est toujours la même (voir 1.2.1).

Une fois cette étape de chargement de la liste des plugins, le framework va démarrer le plugin par défaut qui a été indiqué avec *startplugin* dans son fichier de configuration.