# Documentation d'utilisation de CLEF

Le framework CLEF permet de créer des applications de messagerie instantanée

## 1) Description et installation de la plateforme

### a) Installation du projet (spécifique aux ordinateurs de la fac)

- `git clone https://github.com/ErrOrnAmE/CLEF.git`
- lancer eclipse : " `cd /media/Enseignant/eclipe && ./eclipse`"
- dans eclipse :
    - window -> preferences -> installedJREs -> add -> Standard VM
        - -> jre Home: directory -> " /usr/lib/jvm/java-8-oracle " -> finish
    - file -> import… -> Existing projects into Workspace
    - select root directory du projet
    - finish
    - clic droit sur le projet "CLEF"
        - maven -> update project -> CLEF -> ok
    - clic droit sur le projet "CLEF"
        - properties -> java compiler -> décocher "Use compliance [ … ]"
        - Compiler compliance level -> 1.8
        - apply -> ok


- ouvrir src.framework.Framework.java -> run

### b) Utilisation de l'application existante

- Host :
    - laisser l'adresse en localhost
    - définir le port voulu (ne pas oublier d'ouvrir le port en question sur son routeur)
    - définir son pseudo
    - Appuyer sur "Connect"
    - Voilà, vous êtes connectés en temps qu'hôte
- Client :

    - Se définir en temps que guest
    - Définir son pseudo
    - Définir l'adresse du serveur auquel se connecter
    - Définir le port auquel se connecter
    - Appuyer sur "Connect"
- Pour les deux
    - Pour envoyer un message, écrivez puis appuyez sur "entrer" ou sur le bouton "send"
    - Pour se déconnecter, appuyez sur le bouton "Disconnect". Si vous êtes l'host, tous les clients seront déconnectés.

- Problèmes existants :
    - En cas de déconnexion du serveur ou du client, il faut relancer le programme.

### c) Utilisation du moniteur

Afin de visualiser l'état des plugin (à savoir, chargé ou non chargé) un monitor est disponible. Il est possible de charger une extension manuellement via le bouton *load* et d'en décharger une avec le bouton *kill* (Cette action est susceptible d'empêcher le bon fonctionnement de l'application).
Le moniteur ne se rafraichit pas automatiquement, il faut passer par le bouton *refresh* pour mettre à jour l'état des plugins.

# Documentation technique de CLEF

## 2) Fonctionnement du framework

### a) La classe Framework

La classe Framework est la pièce maîtresse de l'application: elle s'occupe de charger les fichiers de configuration, de trouver toutes les extensions dont l'application a besoin et de créer les proxies pour chacune des extensions.

Le Framework effectue les étapes suivantes:

1- Charge la configuration ("config.json")

2- Charge les dépendances; les extensions qui seront nécessaires

3- Exécute les extensions qui sont déclarés en "autorun" dans leur configuration

### b) Une extension

Une extension est une implémentation d'une des interfaces disponibles dans le framework. Les interfaces disponible sont les suivantes:

- IMonitoring : Interface du monitoring

- INetworkClient : Interface pour fonctionnement de client
- INetworkServer : Interface pour fonctionnement de serveur
- IGUI: Interface de l'affichage de l'application
- IMonitorGUI: Interface de l'affichage du monitoring

Une extension est représenté par un package dans l'application (ex: "extension.networkServer"). Dans ce package se trouve un fichier de configuration ("config.json") de la forme suivante:

```
{
      "name": "NetworkClient",
      "type": "INetworkClient",
      "description": "Cette extension permet de se connecter à un serveur
et d'envoyer des messages",
      "autorun": false,
      "killable": false
}
```

Les paramètres sont les suivants:
- name: Le nom de l'extension, ainsi que le nom de la classe principale de l'extension
- type: Le nom de l'interface qui est implémenté
- description: Description de l'extension
- autorun: Indique si l'extension doit être lancé au lancement de l'application
- killable: Indique si l'extension peut être "killé" durant l'exécution de l'application

### c) La proxification des extensions

Chaque extension est placée derrière un proxy par le framework. Cela permet de gérer le chargement dynamique et le chargement paresseux de l'extension.
De plus, ce proxy (ExtensionContainer), permet de gérer les méthodes "kill" et "load" utilisé notamment par l'interface de monitoring.

### d) Configuration de l'application

L'application correspond à un fichier de configuration situé dans le dossier racine du projet. Il permet d'indiquer toutes les extensions qui pourront être utilisé par l'application.
La configuration est la suivante ("config.json"):

```
{
      "name": "CLEF",                              // nom Application
      "description": "Truc",                       // description
application
      "extensions": [                              // liste des extensions
nécessaires
            "extensions.monitoring.Monitoring",    //
            "extensions.app.App",                  //
            "extensions.test.Test"                 //
      ]
```

```
}
```

### e) Gestion des event

Le framework propose un système d'événement pour gérer les échanges entre différentes extensions.

Pour cela, l'extension qui veut souscrire à un type événement (exemple: "message.received"), doit ajouter la ligne suivante (dans la méthode run() par exemple):

```
Framework.subscribeEvent("message.received", this);
```

Il doit ensuite redéfinir la méthode "handleEvent" (exemple avec "message.received"):

```
@Override
public void handleEvent(String name, Object event) {
    if(name.equals("message.received")){
        System.out.println("Le message suivant a été
reçu:"+(String)event);
    }
}
```

Par la suite, quand une autre extension veut créer un événement, elle utilisera le code suivant:

```
Framework.event("message.received",”Bonjour, comment vas-tu?”);
```

## 3) Création d'une extension

Comme expliqué précédemment, une extension implémente une interface.

Vous pouvez soit réutiliser une interface déjà disponible (Elles sont situées dans framework.extensions), soit en créer une nouvelle. Si vous décidez d'en créer une nouvelle, n'oubliez pas d'étendre l'interface IExtension, la base de chaque interface.

Une fois que votre interface existe (Ex: IMonInterface), vous allez maintenant créer le package correspondant à l'interface (Ex: "extensions.monExtension").

Puis, vous allez créer votre classe principale:

```
public class MonExtension implements IMonInterface {
```

```
	// Méthodes ...

	public void run() {
		//Do stuff
	}

	public void handleEvent(String name, Object event) {
		//Do stuff
	}
}
```

Enfin, créez le fichier de configuration “config.json” à placer dans la package précédemment créé:

```
{
	“name”: “MonExtension”,
	“type”: “IMonInterface”,
	“description”: “Cette extension fait des trucs”,
	“autorun”: false,
	“killable”: false
}
```

## 4) Utilisation du Framework

# get(Class<?>)

`public static List<ExtensionContainer> get(Class<?> inter)`

Retourne la liste des implémentations d'une interface donnée en paramètre.

```
List<ExtensionContainer> affichages = Framework.get(IAffichage.class);
IAffichage affichageConsole = (IAffichage) affichages.get(0).getExtension();
```

# get(Class, Class)

`public static IExtension get(Class<?> inter, Class<?> impl)`

Retourne l'implémentation en fonction de son interface et de sa classe donnée en paramètre.

```
IAffichage affichageConsole = (IAffichage) Framework.get(IAffichage.class,
AffichageConsole.class);
```

## getExtension(Class<?>)

`public static IExtension getExtension(Class<?> inter)`

Retourne la première implémentation d'une interface donnée en paramètre.

```
IAffichage affichage = (IAffichage) Framework.getExtension(IAffichage);
```