

SAF : Simple Actor Framework

Développement d'un framework d'acteurs distribués avec Spring Boot

Ayman OUGUARD Nahel SAIT Hadjer NEDJAR

Louaye SAGHIR Matias VINKOVIC

3e année d'école d'Ingénieur, Génie Informatique en Apprentissage

Module Web Avancé JEE

25 décembre 2025



Présentation du Projet SAF

Qu'est-ce que SAF ?

Le framework **SAF** (Simple Actor Framework) est un outil de développement d'applications distribuées conçu avec **Spring Boot**[cite : 7].

- **Inspiration Akka** : Il transpose le modèle des acteurs (boîte noire, état isolé) dans l'univers Java/Spring[cite : 4, 60].
- **Communication par Messages** : Les acteurs ne s'appellent pas entre eux ; ils s'envoient des signaux via une Mailbox[cite : 62, 88].
- **Cœur Microservices** : Chaque acteur peut vivre localement ou être joint sur un autre service via le réseau[cite : 61].

Philosophie

L'objectif est de simplifier la gestion de la concurrence et du réseau : le développeur se concentre sur *quel message envoyer*, et SAF s'occupe de *comment le transmettre*[cite : 8].

Pourquoi ce framework ?

Le développement distribué moderne nécessite des systèmes capables de réagir aux imprévus. SAF répond à trois enjeux majeurs :

- **La Résilience** : Capacité à détecter le "crash" d'un composant et à le redémarrer sans stopper l'application entière[cite : 14, 68].
- **L'Élasticité** : Possibilité de créer ou supprimer des instances d'acteurs à la volée selon la charge de travail[cite : 15, 71].
- **La Traçabilité** : Un système de logs intégré qui permet d'auditer chaque action utilisateur et chaque événement système[cite : 22, 84].

Ce document présente notre framework dans son aspect technique, en expliquant de manière vulgarisé et rapide le code de chaque Classifier Java que nous avons implanté.

- **Inspiration Akka** : Utilisation du modèle d'acteurs (simplifié) où chaque entité possède son propre état et communique uniquement par messages.
- **Objectif Microservices** : Créer des briques autonomes capables de collaborer via le réseau avec Spring Boot.
- **Isolation Totale** : Contrairement aux objets classiques, on n'appelle jamais une méthode sur un acteur, on lui envoie un message dans sa Mailbox. Il est du rôle du framework d'analyser si ce Message peut être transmis, si oui comment, si une erreur alors la logger etc...

Le cahier des charges impose deux piliers fondamentaux que SAF implémente :

- ❶ **La Résilience** : Le système ne doit pas s'effondrer si un acteur rencontre une erreur. Le framework doit être capable de "soigner" l'acteur, en le redemarrant par exemple.
- ❷ **L'Élasticité** : Capacité à créer des instances d'acteurs dynamiquement (via la commande `spawn` du Shell) pour s'adapter à la charge de travail.

Le module saf-core : Architecture logicielle

Ce module constitue le socle du framework, sans dépendance externe.

- **Actor** : Interface définissant ce qu'est un acteur, ses méthodes à redéfinir.
- **ActorRef** : Abstraction de l'adresse d'un acteur pour l'envoi de messages.
- **LocalActorRef** : Implémentation gérant l'accès direct à la mailbox locale. (du même microservice)
- **ActorSystem** : Moteur d'exécution des acteurs, superviseur local.
- **ActorContext** : Environnement d'exécution fourni à l'acteur.
- **Mailbox** : LinkedList sous forme de Queue pour faire passer les messages par ordre d'arrivé.
- **Message & MessageEnvelope** : Structures de données pour le transport des messages.
- **LoggerService** : Système d'audit technique et métier.

Chaque acteur possède sa mailbox, caractérisée par la classe MailBox. Cette classe, possède une `Queue<MessageEnvelope>` `queue = new LinkedList<>()`; ainsi que les méthodes associées pour `dequeue`, `enqueue`

- Elle peut `enqueue` un `MessageEnvelope`, ce qui nous permet d'encapsuler un `Message` dans une boîte, contenant le `Message` et le `sender`
- `Message` est une simple interface et les instances donc les messages que les acteurs peuvent envoyer sont libre de leur implémentation.

Focus : Le moteur ActorSystem

ActorSystem est le chef d'orchestre du framework.

- **createActor(Class, actorName)** : Utilise la réflexion Java (`newInstance()`) pour créer l'acteur et lui associer une Mailbox unique.
- **processOneCycle()** : C'est le cœur du système. Cette méthode parcourt tous les acteurs. Pour chacun, elle vide la mailbox en boucle tant qu'elle contient des messages.

Supervision Hiérarchique

Si `onReceive()` lève une exception, `processOneCycle` capture l'erreur, loggue le crash via `LoggerService`, et appelle `ref.restart()` pour réinstancier l'acteur proprement.

Actor : la racine du projet

Actor, est l'interface que chaque type d'acteurs que l'utilisateur final va créer, devra implémenter. Chaque classe d'acteur spécifique devra donc redéfinir la méthode `onReceive(Message, context)`, ce qui correspond à : "quand je reçois un message, alors je réalise tel action".

- La méthode `onReceive` peut filter sur le type de message qu'elle reçoit et adapter le comportement de l'acteur en fonction, via `if(msg instanceof DemandeReservation` par exemple => cela permet à un acteur de traiter de manière différente chaque type de message.
- Elle lance une exception en cas d'erreur de l'acteur, erreur qui sera catch par `ActorSystem` qu'on verra après

ActorSystem : rôle de système d'acteurs

ActorSystem, joue le rôle de boîte à acteurs, il définit la méthode `createActor` qui va simplement ajouter l'acteur au système, et donc le stocker dans une unité de donnée, une Hashmap private final `Map<String, LocalActorRef> actors = new HashMap<>();`

- Elle prend en paramètre une classe d'acteur.
- Elle créer de toute pièce l'instance de l'acteur : sa mailbox, son instance, sa référence locale, et ajoute simplement via `actors.put(actorName, ref)` l'acteur au système !
- Elle catch une Runtime également pour préserver la résilience du système.

Puisque ActorSystem est le système d'acteurs, c'est ici que nous avons mis la méthode de supervision `processOneCycle`. Cette méthode ne prenant aucun paramètres, est chargée de parcourir le HashMap d'acteurs locaux, et de regarder si leur mailbox est remplie, si oui et que l'acteur n'est pas bloqué (`ref.isBlocked()==false`), alors on lui fait savoir, et l'acteur exécute sa méthode `onReceive()`

- Elle catch l'Exception générée par le `onReceive()` de l'acteur (`catch (Exception e)`) et tente de redémarrer automatiquement l'acteur dans ce cas via la méthode `restart()` qui crée une nouvelle instance de l'acteur.
- Elle est exécutée dans le module `saf-spring`, lors du lancement du système, toutes les x secondes (nous le verrons après).

Focus : Communication Asynchrone (Tell)

Dans `LocalActorRef`, la méthode `tell` définit la communication asynchrone (sans attente de réponse).

- Elle prend en paramètre un **Message**.
- Elle se charge simplement d'enqueue la mailbox de la référence de l'acteur `mailbox.enqueue(new MessageEnvelope(msg, null));`, avec le **Message** qui va bien.

Focus : Communication Synchrone (Ask)

Dans `LocalActorRef`, la méthode `ask` simule un blocage synchrone.

- Elle crée un **CompletableFuture**.
- Elle génère un **Callback ActorRef** temporaire.
- Ce callback, lorsqu'il reçoit la réponse via `tell`, complète le futur, débloquent ainsi l'appelant.

Le module saf-spring : Distribution & REST

Ce module étend le cœur pour supporter le réseau et Spring Boot.

- **SAF** : Façade statique pour démarrer le système et le moteur.
- **RestRemoteActorRef** : Proxy réseau utilisant Eureka et HTTP.
- **SafMessageController** : Point d'entrée HTTP pour les messages distants.
- **ActorSupervisor** : Orchestrateur temporel via Spring Scheduling.
- **SAFShell** : Console de commande interactive par réflexion.
- **IncomingMessageDTO** : Format d'échange JSON universel.

Focus : Communication InterMicroservice via RestRemoteActorRef 1

Cette classe permet d'envoyer un message à un acteur distant du microservice actuel, mais de la même manière qu'on l'aurait fait pour un acteur local via tell.

- Au lieu de déposer un message dans la mailbox d'un acteur directement, elle transforme l'appel en une requête HTTP REST vers le microservice distant approprié.
- **resolveTargetUrl()** : Interroge le `DiscoveryClient` (Eureka) pour obtenir le port du microservice visé, c'est le principe de découverte de microservice.
- **tell(Message)** (asynchrone) : Sérialise le message en JSON via `ObjectMapper`, l'encapsule dans un DTO et l'envoie par POST vers `/actors/messages`.

Focus : Communication InterMicroservice via RestRemoteActorRef 2

- **tell(Message)** : c'est la même méthode que celle présente dans ActorRef du package saf-core, mais on l'Override pour l'adapter au contexte HTTP REST. Et au lieu de faire un enqueue, on `http.postForEntity(baseUrl + "/actors/messages", entity, Void.class)`, ce qui envoie au endpoint `/actors/messages` un message sérialisé par Jackson.
- Evidemment, on catch les erreurs dans le cas où il y en a, afin de rendre le code résilient et qu'il n'y ait aucun crash en cas de crash d'un acteur.

Focus : Le Chef d'orchestre SAF.java

La classe SAF est la porte d'entrée unique pour l'utilisateur. Elle cache toute la complexité du démarrage, nous avons utilisé **Le Design Pattern Façade** vu en cours.

- **Initialisation Hybride** : Elle ne se contente pas de lancer Spring Boot ; elle récupère le bean `ActorSystem` pour lier les deux mondes.
- **SAF-Engine-Thread** : C'est le cœur battant du microservice. On crée un thread dédié (Daemon) qui force l'exécution de `processOneCycle()` toutes les 100ms.
- **Pourquoi un Thread ?** : Cela garantit que même si l'application Spring est "au repos", les mailboxes des acteurs continuent d'être vidées de manière asynchrone.
- **Synchronisation Réseau** : (*`waitForNetworkDiscovery`*) est basiquement une méthode qui met en pause le microservice durant 4s afin de laisser le temps à Eureka de découvrir les MicroServices. Cela ayant été fait suite à des bugs de synchronisation.

Focus : Le Contrôleur de Messages (SafMessageController)

C'est un peu comme la tour de contrôle qui reçoit les avions (messages) venant de l'extérieur. Autrement dit les requêtes Http pour ses endpoints /actors et sous endpoint /actors/messages.

- **Réception Asynchrone (/messages)** : Quand un message arrive sur ce endpoint, le contrôleur le déserialise et cherche l'acteur local correspondant avec `findLocal(actorName)`, méthode du système ActorSystem (tout est lié!) . S'il existe, il dépose le message dans sa mailbox et libère la connexion immédiatement.
- **Réception Synchrone (/messages/ask)** : Pour les messages synchrones le contrôleur "invente" un acteur temporaire (`temporarySender`).
- Cet acteur éphémère n'a qu'un but : intercepter la réponse de l'acteur local et la réinjecter dans le thread HTTP suspendu pour répondre au client.

Focus : Le Pilotage par Console (SAFShell)

Le Shell est un interpréteur de commandes simple qui permet de piloter le framework en temps réel sans toucher au code Java.

- **Structure en Switch** : Le cœur du Shell est une boucle `while` contenant un grand `switch/case`. Chaque mot tapé (`spawn`, `tell`, `ref`) déclenche une action spécifique.
- **Spawn = Injection** : Quand on tape `spawn`, le Shell ne fait qu'injecter une commande `system.createActor()` dans le système. Cela permet de créer un acteur "à la volée" pendant que le microservice tourne.
- **Stockage local** : Pour faciliter l'utilisation, le Shell garde en mémoire (via des `HashMap`) les objets que l'on crée, comme les messages ou les références distantes, pour pouvoir les réutiliser plus tard avec une simple étiquette.

Pour rendre ce framework distribué et automatique, nous avons utilisé les briques avancées de Spring :

- **Service Discovery (Eureka)** : Utilisation du `DiscoveryClient` pour que les microservices se trouvent par leur nom, sans connaître leurs adresses IP.
- **Ordonnancement (@Scheduled)** : Une tâche planifiée tourne en arrière-plan pour déclencher le moteur d'acteurs toutes les xms (ex : `processOneCycle()`)
- **Inversion de Contrôle (@Bean)** : L' `ActorSystem` est géré comme un composant Spring réutilisable partout dans l'application.
- **Communication REST** : Utilisation de `RestTemplate` et `@RestController` pour transformer des messages Java en flux JSON circulant sur le réseau.

Conception du framework / architecture

Diagrammes et schémas clairs

Diagramme 1 : Architecture Globale (Vue Multi-Niveaux)

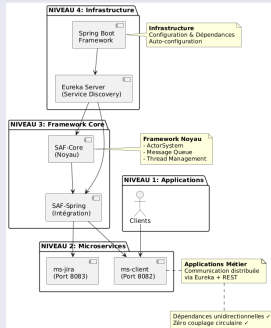


Figure – Architecture Globale du Framework SAF

Diagrammes et schémas clairs

Diagramme 2 : Composants du Noyau (SAF-Core)

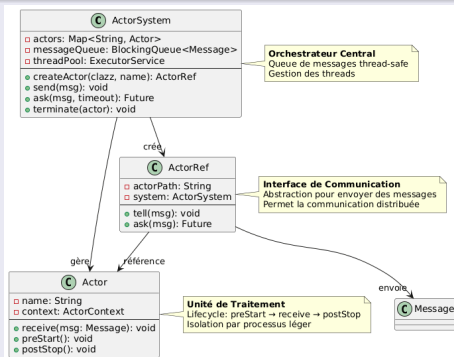


Figure – Classes Principales du Framework SAF-Core

Diagrammes et schémas clairs

Diagramme 3 : SAF-Spring (Intégration Spring)

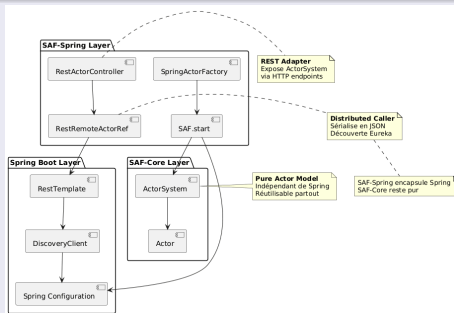


Figure – SAF-Spring : Bridge entre SAF-Core et Spring

Diagrammes et schémas clairs

Diagramme 4 : Flux d'Exécution Complet

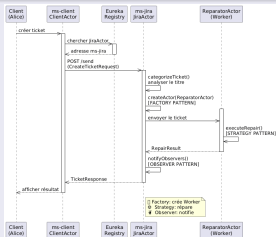


Figure – Diagramme de sequence : De la Création au Résultat

Diagrammes et schémas clairs

Diagramme 5 : Design Patterns Combinés

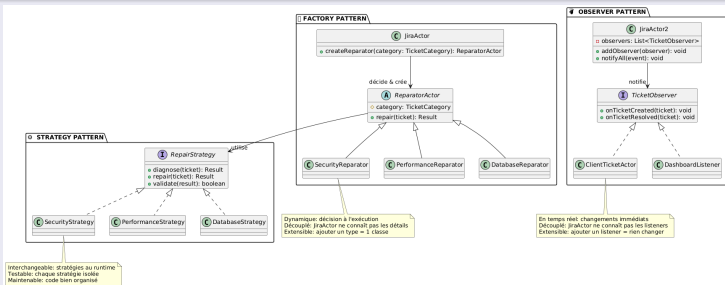


Figure – Les 3 Design Patterns du Framework (Factory + Strategy + Observer)

Pour valider le framework, nous avons déployé un scénario distribué :

- **Microservice Jira** : Héberge un `JiraActor` (nommé "tom") prêt à recevoir des commandes[cite : 27].
- **Microservice Client** : Utilise une `RestRemoteActorRef` pour envoyer des commandes à "tom" via le réseau[cite : 30, 31].
- **Découverte** : Le client ne connaît pas l'adresse IP du jira, il la découvre dynamiquement grâce à **Eureka**.

Stratégie de Test : Cycle de Vie et Distribution

Scénario de Validation :

- **Découverte** : Le Client trouve "tom" via **Eureka** sans connaître son IP.
- **Élasticité** : Test de création et suppression dynamique (Log BORN/DIED).
- **Résilience** : Vérification de l'arrêt total des messages post-mortem.

```
=== TEST : NAISSANCE ===
[2025-12-26 19:13:52] [INFO] [alice] BORN -> Actor created successfully
[2025-12-26 19:13:52] [INFO] [martin] BORN -> Actor created successfully

=== TEST : TRAITEMENT (AVANT MORT) ===
[Test] Attente du cycle de traitement (3s)...
[2025-12-26 19:13:54] [INFO] [alice] RECEIVE -> Processing message: BasicMessage
Message reçu: Coucou Alice (vivante)

=== TEST : MORT ===
[2025-12-26 19:13:55] [WARN] [alice] DIED -> Acteur supprimé et ne recevra plus de messages.

=== TEST : POST-MORTEM (APRÈS MORT) ===
[Test] Attente du cycle de traitement (3s) pour vérifier l'absence de réponse...
```

Figure – Preuve du cycle de vie dans la console

Extrait du Test Unitaire :

Test de suppression

```
system.killActor("alice");
assertNull(system.findLocal("alice"));
```

L'élasticité est un pilier de notre framework. Les logs ci-dessous démontrent que le système n'alloue des ressources que lorsque cela est nécessaire. Lorsqu'un ticket est créé, le `jira-manager` instancie dynamiquement un acteur de réparation spécifique.

Extrait des logs - MS-JIRA (19 :22 :59)

```
[2025-12-26 19 :22 :59] [INFO] [REPARATEUR-6] BORN -> Actor created successfully  
RÉPARATEUR ASSIGNÉ → Ticket : JIRA-1006
```

Une fois la tâche accomplie, l'acteur est supprimé pour libérer la mémoire.

Extrait des logs - MS-JIRA (19 :23 :04)

```
[2025-12-26 19 :23 :04] [WARN] [REPARATEUR-6] DIED -> Acteur supprimé et ne recevra plus de messages.  
RÉPARATEUR TERMINÉ ET NETTOYÉ : REPARATEUR-6
```

Conclusion : Le framework gère le cycle de vie complet sans intervention manuelle, évitant ainsi les fuites de mémoire.

La robustesse du framework SAF est prouvée par sa capacité à intercepter

les exceptions critiques et à restaurer l'état du système.

À 19 :23 :13, nous avons simulé l'envoi d'un ticket nul (`null`) pour provoquer une `NullPointerException` dans le thread de l'acteur.

Log du Crash - Supervision (19 :23 :13)

```
[19 :23 :13] [ERROR] [jira-manager] CRASH -> Exception during  
CreateTicketRequest : Cannot invoke ... getTicket() is null  
:23 :13
```

```
[WARN] [jira-manager] SUPERVISION -> Restarting actor instance...  
:23 :13
```

```
[INFO] [jira-manager] RESTART -> Actor successfully healed and ready.
```

Contrairement à une architecture standard où une telle exception pourrait figer le service ou corrompre la boucle de traitement, SAF applique le pattern "**Let it crash**" :

- **Isolation** : L'erreur est confinée à l'instance de l'acteur.
- **Restauration** : Le superviseur recrée une instance propre.

- **Continuité** : Le service reste disponible pour les messages suivants.

Le client a récupéré la liste des tickets via le réseau sans connaître l'IP du serveur, en utilisant le nom logique MS-JIRA.

- **Localisation transparente** : Résolution via Eureka.

- **Fiabilité** : *"Ask réussi après 1 tentative(s)"*.

Nous avons testé la capacité à mettre un acteur "en pause" sans perdre de messages.

Scénario de blocage - MS-CLIENT (19 :23 :16)

[19 :23 :16] [WARN] [supervisor-local] LIFECYCLE -> Actor has been
BLOCKED

:23 :17

[INFO] [supervisor-local] BLOCKED -> Acteur bloqué, messages en queue

...

:23 :20

[INFO] [supervisor-local] LIFECYCLE -> Actor has been UNBLOCKED

:23 :21

[INFO] [supervisor-local] RECEIVE -> Processing message : TestMessage
(1, 2, 3)

Conclusion globale : Les tests démontrent que SAF est un framework distribué complet, capable de gérer l'asynchronisme, la distribution réseau

et l'auto-guérison.

Le framework SAF propose une fonctionnalité de **Backpressure** simplifiée via le blocage d'acteurs. Ce test démontre que les messages ne sont pas perdus lorsque le traitement est suspendu.

À 19 :28 :01, l'acteur `supervisor-local` a été marqué comme **BLOQUÉ**. Trois messages ont été envoyés durant cette période d'indisponibilité simulée.

Logs de mise en file d'attente (19 :28 :02 - 19 :28 :04)

```
[19 :28 :01] [WARN] [supervisor-local] LIFECYCLE -> Actor has been  
BLOCKED
```

```
:28 :02
```

```
[INFO] [supervisor-local] BLOCKED -> Acteur bloqué, impossible de  
communiquer
```

```
... (Messages 1, 2 et 3 mis en attente dans la Mailbox)
```

Dès le passage à l'état UNBLOCKED, le moteur SAF a dépilé l'intégralité de la Mailbox, garantissant l'intégrité du flux métier.

Logs de reprise (19 :28 :06)

```
[19 :28 :05] [INFO] [supervisor-local] LIFECYCLE -> Actor has been  
UNBLOCKED  
:28 :06
```

```
[INFO] [supervisor-local] RECEIVE -> Processing message : TestMessage  
(1)  
:28 :06
```

```
[INFO] [supervisor-local] RECEIVE -> Processing message : TestMessage  
(2)...
```

Conclusion : Ce mécanisme permet d'isoler un composant défaillant ou en maintenance sans impacter la disponibilité globale du système distribué. Un système distribué doit être capable de fonctionner en mode dégradé. Nous avons testé le comportement du micro-service MS-CLIENT lorsque l'infrastructure (**Eureka**) et le service cible (**MS-JIRA**) sont indisponibles. Le framework SAF intercepte les exceptions de connexion pour éviter le

crash du micro-service initiateur.

Logs de résilience réseau (19 :27 :41)

```
[19 :27 :41] [WARN] [jira-manager] REMOTE_ASK_RETRY -> Tentative  
1 échouée : Service ms-jira introuvable  
:27 :43
```

```
[WARN] [jira-manager] REMOTE_ASK_RETRY -> Tentative 3 échouée  
...  
Erreur : java.lang.RuntimeException : Échec de la communication  
synchrone (ask)
```

Bien que la communication distante soit rompue, les acteurs locaux continuent d'échanger des messages normalement (Phase 9 : Communication Alice ↔ Bob).

Analyse : L'architecture par acteurs offre un **découplage temporel et spatial** total. Une panne réseau est traitée comme un simple état de l'acteur distant, ne compromettant jamais la stabilité du système local.

Comme exigé, chaque action est tracée pour permettre un audit complet de l'activité[cite : 22, 37] :

Exemple de trace générée par LoggerService

```
[2025-09-15 14:02:10] [INFO] [tom] RECEIVE -> Processing  
message: OrderRequest  
[2025-09-15 14:02:10] [ERROR] [tom] CRASH -> Exception  
during OrderRequest : Cuisine en feu !  
[2025-09-15 14:02:10] [WARN] [tom] RESTART -> Actor  
successfully healed.
```

- **Akka Documentation** : Concepts fondamentaux du modèle d'acteurs et de la supervision.
- **Spring Cloud Netflix** : Guide d'utilisation d'Eureka pour le Service Discovery.
- **Spring Boot Reference** : Utilisation des annotations `@Scheduled`, `@Bean` et du `RestTemplate`.