

# Plateforme SAF

*Météorologie des données*

*Deulyne Destin*

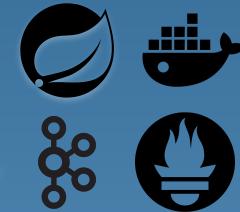
*Axel Cottrant*

*Maxime Crayssac*

*Théo De Morais*

*Florian Chapin*

*ING2-GIA1-Groupe2*



# Sommaire

## Introduction

- Concepts avancés de Spring Boot utilisés
- Architecture SAF (Core / Runtime / Control)
- Micro-services (Client / Ville / Capteurs)

## Architecture Système et Routage

- Création d'acteurs (ActorSystem / ActorFactory)
- Routage des messages

## Architecture du Core

- Modèle d'acteur, message et cycle de vie
- Transport des messages (Mailbox / Stockage)

## Runtime

- Sécurité & Isolation (ActorContext / Logging)
- Scalabilité (Virtual Threads)

## Architecture Distribuée et Résilience

- Passage à l'échelle via Kafka
- Supervision (Gestion des Pannes)

## Conclusion & Démonstration Rapide

- Conclusion
- Démonstration d'un scénario

# Concepts avancés de Spring Boot utilisés

# Planification automatique

Planification : Surveillance automatique périodique avec @Scheduled

**@Scheduled** déclenche automatiquement une méthode à intervalle régulier, sans intervention manuelle.

## Cas d'usage dans SAF :

- Vérification santé des agents toutes les **30 secondes**
- Health check des services toutes les **10 secondes**
- Heartbeat système toutes les **15 secondes**

```
@Scheduled(fixedRate = 10_000, initialDelay = 15_000)
public void checkServicesHealth() {
    log.debug("Health check pour {} services",
              serviceRegistry.getAllServices().size());

    for (ServiceInfo service : serviceRegistry.getAllServices()) {
        checkServiceHealth(service);
    }
}
```

ServiceHealthMonitor.java (ligne 67)

```
@Scheduled(fixedRate = 30000) // 30 secondes
public void checkAgentHealth() {
    Instant now = Instant.now();

    controlService.list().forEach(agent -> {

        // Ignorer agents en quarantaine
        if (quarantine.contains(agent.id())) {
            return;
        }

        // Détection agent sans heartbeat
        if (isAgentStale(agent.id(), now)) {
            System.out.println("Agent " + agent.id() + " INACTIF");
            supervisionService.handle(agent);
            quarantine.add(agent.id());
        }
    });
}
```

AgentMonitoringService.java (ligne 142)

```
@Configuration
@EnableScheduling
public class SchedulingConfig {
}
```

SchedulingConfig.java

- Bénéfice : Détection automatique des pannes sans intervention manuelle

# Traitements asynchrones

Asynchrone : Health checks non-bloquants

Synchrone (bloquant) :	Asynchrone (non-bloquant)
Request → <b>Attend</b> → Response	Request → <b>Continue</b> → Callback
1 service à la fois	Tous en parallèle
10 services = 10 sec	10 services = 1 sec

```
private Mono<Boolean> performActiveHealthCheck(ServiceInfo service) {
    String healthUrl = service.getUrl() + "/actuator/health";

    return webClient.get()
        .uri(healthUrl)
        .retrieve()
        .bodyToMono(String.class)
        .map(response -> true)
        .timeout(HEALTH_CHECK_TIMEOUT) // 5 secondes max
        .onErrorResume(error -> {
            log.debug("Health check échoué: {}", error.getMessage());
            return Mono.just(false);
        });
}
```

ServiceHealthMonitor.java (ligne 87)

```
private void checkServiceHealth(ServiceInfo service) {
    performActiveHealthCheck(service)
        .subscribe(
            healthy -> handleServiceHealthy(service),
            error -> handleServiceDown(service, error)
        );
}
```

ServiceHealthMonitor.java (ligne 76)

Bénéfice : Parallélisation maximale, API toujours réactive, timeout configuré

# Threads Virtuels (Java 21)

Threads Virtuels : Concurrence massive et légère

Problème des threads classiques :

Limitation	Impact
2 MB par thread	Mémoire saturée rapidement
Création coûteuse	Overhead CPU
Max ~1000 threads	Scalabilité limitée

→ Impossible de gérer des milliers d'acteurs simultanément

Solution : Threads Virtuels (Java 21) ✓

Avantage	Résultat
1 KB par thread	x2000 plus léger
Création quasi-gratuite	Overhead minimal
Millions possibles	Scalabilité illimitée

→ 1 acteur = 1 thread virtuel = SAF ultra-scalable

```
@Component
public class VirtualThreadDispatcher implements Dispatcher {

    private final ExecutorService executor =
        Executors.newVirtualThreadPerTaskExecutor(); // Java 21

    @Override
    public void dispatch(ActorRef actorRef, Mailbox mailbox,
                         Actor actor, SupervisionStrategy supervisionStrategy) {

        // Un thread virtuel par acteur
        executor.submit(() ->
            processMailbox(actorRef, mailbox, actor, supervisionStrategy)
        );
    }
}
```

### Mémoire optimisée :

100 000 acteurs = ~100 MB (vs ~200 GB en threads classiques)

```
private void processMailbox(...) {
    int processedCount = 0;

    // Traiter jusqu'à 50 messages
    while (processedCount < THROUGHPUT_LIMIT) {
        Message message = mailbox.dequeue();
        if (message == null) return;

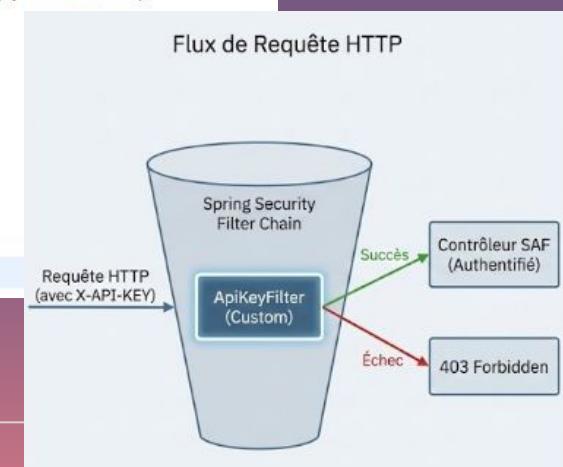
        actor.receive(message);
        processedCount++;
    }

    // Replanifier si messages restants
    if (!mailbox.isEmpty()) {
        dispatch(actorRef, mailbox, actor, supervisionStrategy);
    }
}
```

# Sécurisation du contrôle d'accès (Spring Security)

```
@Configuration
public class SecurityConfig {

    @Bean
    SecurityFilterChain filter(HttpSecurity http) throws Exception {
        http.csrf(AbstractHttpConfigurer::disable)
            .cors(cors -> cors.configure(http)) // Enable CORS
            .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .addFilterBefore(apiKeyFilter, AbstractPreAuthenticatedProcessingFilter.class)
            .authorizeHttpRequests(auth -> auth
                .dispatcherTypeMatchers(DispatcherType.ASYNC, DispatcherType.ERROR)
                .permitAll()
                //Les endpoints publics
                .requestMatchers(PUBLIC_PATHS).permitAll()
                //Toutes les autres requêtes nécessitent le filtre
                .anyRequest().authenticated()
            );
        return http.build();
    }
}
```



# Spring Lifecycle

Intégration des Virtual Threads dans le cycle de vie du conteneur Spring

1) Instanciation & injection

```
@Component // -> Singleton géré par Spring
public class VirtualThreadDispatcher implements Dispatcher {
    // Ressources critique : le pool Java 21
    private final ExecutorService executor =
        Executors.newVirtualThreadPerTaskExecutor();

    // Méthodes de dispatch ...

    // Hook d'arrêt pour éviter les fuites
    @Override
    public void shutdown() {
        executor.shutdown();
    }
}
```

2) Initialisation

3) Exécution (Runtime)

4) Arrêt

- Garantie qu'aucun thread ne reste "zombie" à l'arrêt de l'application
- Le pattern Singleton (@Component) assure une instance unique du moteur d'exécution

# Gestion de la messagerie avec Kafka

- Utilisation d'un producer qui se charge d'envoyer des messages sur un canal (topic kafka)
- Les messages sont stockés sur des brokers de messages (serveurs kafka)
- Utilisation d'un consumer qui va s'abonner à un canal (topic kafka) afin de “consommer” les messages (lire les messages sur ce topic)
- Communication asynchrone : les messages peuvent être lus à tout moment
- Les composants ne communiquent plus directement entre eux -> Découplage
- Exemple minimal d'utilisation de kafka avec SpringBoot (producer + consumer + configuration)  
=> **Slide suivante**

## Producer

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class Producer {
    private final KafkaTemplate<String, String> kafkaTemplate;

    public Producer(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String message) {
        kafkaTemplate.send("mon-topic", message);
    }
}
```

## Consumer ->

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class Consumer {

    @KafkaListener(topics = "mon-topic")
    public void listen(String message) {
        System.out.println("Message reçu : " + message);
    }
}
```

## Configuration Kafka (application.yml)

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: mon-groupe
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
```

# Observabilité & Actionneurs - Spring Boot Actuator

## Spring Boot Actuator fournit :

- Health Checks :  
Vérification de l'état des services
- Endpoints de monitoring :  
Santé, métriques, info
- Métriques :  
Collecte de données de performance

## Dans notre projet SAF Platform :

### Configuration (`application.yml`)

```
management:  
  endpoints:  
    web.exposure.include: health,info,metrics,prometheus  
  endpoint:  
    health.probes.enabled: true
```

### Dépendances Maven (`pom.xml`)

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<dependency>  
  <groupId>io.micrometer</groupId>  
  <artifactId>micrometer-registry-prometheus</artifactId>  
</dependency>
```

### Endpoints exposés

Endpoint	URL	Description
Health	/actuator/health	État de santé du service
Metrics	/actuator/metrics	Métriques JVM et applicatives
Prometheus	/actuator/prometheus	Format Prometheus pour scraping
Info	/actuator/info	Informations sur l'application

# Observabilité & Actionneurs - Métriques avec Micrometer

## L'architecture des métriques dans SAF Platform :

### RuntimeMetricsStore - Stockage des données

```
@Component
public class RuntimeMetricsStore {
    private final AtomicInteger actorCount = new AtomicInteger(0);
    private final AtomicReference<Double> averageMailboxSize = new
AtomicReference<>(0.0);

    public int actorCount() { return actorCount.get(); }
    public void setActorCount(int count) { actorCount.set(count); }

    public double averageMailboxSize() { return averageMailboxSize.get(); }
    public void setAverageMailboxSize(double value) {
        averageMailboxSize.set(value);
    }
}
```

### RuntimeMetrics - Enregistrement des Gauges

```
@Component
public class RuntimeMetrics {
    public RuntimeMetrics(MeterRegistry registry, RuntimeMetricsStore store) {
        // Métrique personnalisée : nombre d'acteurs
        Gauge.builder("saf.runtime.actors.count", store,
RuntimeMetricsStore::actorCount)
            .description("Number of actors running in the runtime")
            .register(registry);

        // Métrique personnalisée : taille moyenne des mailbox
        Gauge.builder("saf.runtime.mailbox.avg.size", store,
RuntimeMetricsStore::averageMailboxSize)
            .description("Average mailbox size across runtime actors")
            .register(registry);
    }
}
```

Soit deux métriques personnalisées exposées :

- |  |  |
|--|--|
| <ol style="list-style-type: none"> <li>1. <i>saf.runtime.actors.count</i></li> <li>2. <i>saf.runtime.mailbox.avg.size</i></li> </ol> | <ul style="list-style-type: none"> <li>→ Nombre d'acteurs en cours d'exécution dans le micro service</li> <li>→ Taille moyenne des boîtes aux lettres</li> </ul> |
|--|--|

# Observabilité & Actionneurs - Intégration Prometheus

L'architecture de Prometheus dans SAF Platform :

Configuration Docker (`docker-compose.yml`)

```
prometheus:
  image: prom/prometheus:latest
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
```

Configuration Prometheus (`prometheus.yml`)

```
global:
  scrape_interval: 15s

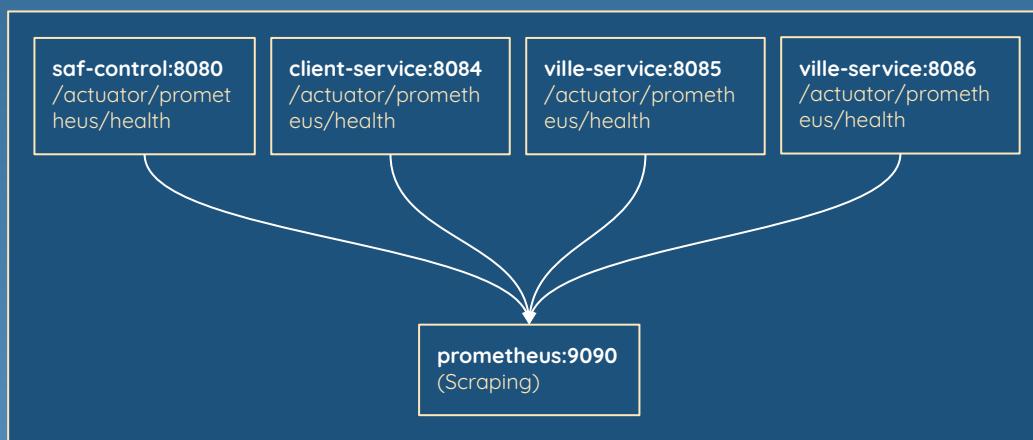
scrape_configs:
  - job_name: saf-control
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["saf-control:8080"]

  - job_name: client-service
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["client-service:8084"]

  - job_name: ville-service
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["ville-service:8085"]

  - job_name: capteur-service
    metrics_path: /actuator/prometheus
    static_configs:
      - targets: ["capteur-service:8086"]
```

Résumé de l'architecture Observabilité :



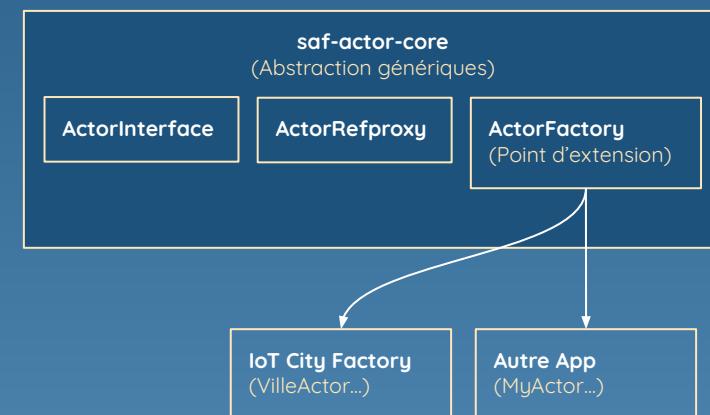
# Architecture SAF (Core / Runtime / Control) - Core

**Son rôle est d'être :** Librairie Java 100% générique qui définit les abstractions d'acteurs sans aucune dépendance métier.

**Ses composants clés sont :**

Actor	Interface définissant le comportement (state + receive(message))
ActorRef	Référence vers un acteur (proxy de communication)
ActorSystem	Orchestrator du système d'acteurs
Message	Abstraction des messages échangés
Mailbox	File d'attente des messages d'un acteur
SupervisionStrategy	Stratégies de gestion des erreurs (restart/resume/stop)
ActorFactory	Point d'extension pour créer des acteurs métier

**Philosophie “Plugin” :**



# Architecture SAF (Core / Runtime / Control) - Runtime

**Son rôle est d'être :** Bibliothèque **Spring Boot** fournissant les **classes de base** pour exécuter les acteurs dans un microservice.

**Ses composants clés sont :**

DefaultActorSystem	Implémentation complète de ActorSystem
InMemoryMailbox	File de messages en mémoire
BaseActorRuntimeController	Contrôleur REST de base pour /runtime/actors
BaseServiceRegistrationInitializer	Auto-enregistrement auprès de SAF-Control
KafkaMessageTransport	Transport des messages via Apache Kafka
RuntimeMetrics	Métriques Micrometer (acteurs, mailbox)

**Pattern d'utilisation :**



# Architecture SAF (Core / Runtime / Control) - Control

**Son rôle est d'être :** Service centralisé (API Gateway) qui orchestre la plateforme entière sans connaître les acteurs métier.

**Ses composants clés sont :**

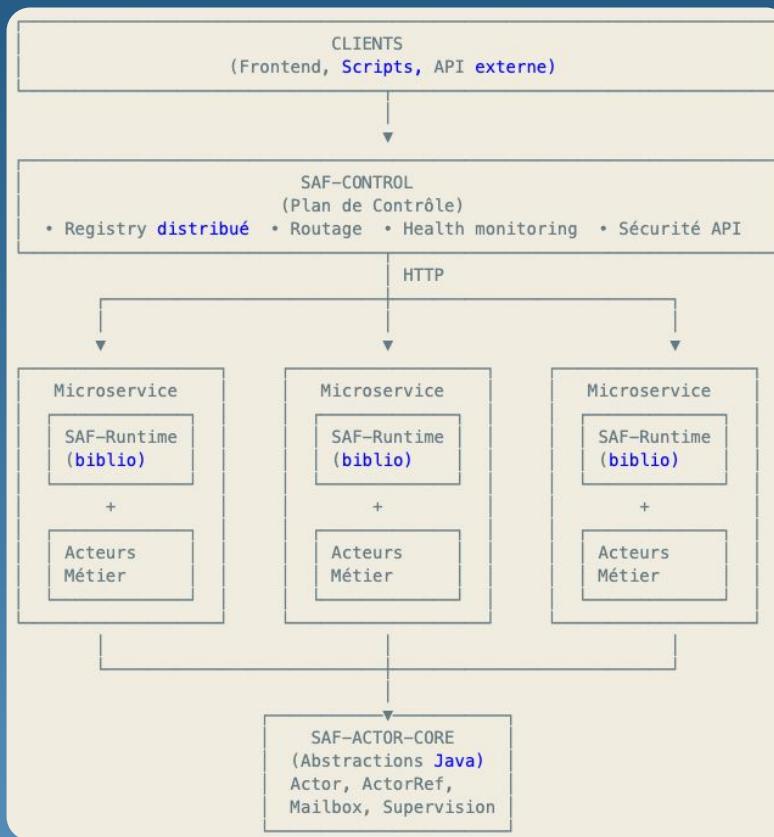
Registry distribué	Annuaire des acteurs et leur localisation
Découverte de services	Les microservices s'enregistrent au démarrage
Routage des requêtes	Distribution des demandes vers les bons microservices
Cycle de vie	CRUD des acteurs (spawn, stop, configure)
Health monitoring	Surveillance active des microservices
Sécurité	Authentification par clé API
Temps réel	WebSocket pour l'UI (logs, événements)

**Création d'un acteur :**

1. Client → SAF-Control  
POST /api/v1/actors  
{ serviceId: "ville-service", type: "VilleActor", params: {...} }
2. Control consulte ServiceRegistry  
→ Trouve l'URL du microservice cible
3. Control → Microservice  
POST /runtime/actors (routage HTTP)
4. Microservice utilise ActorFactory  
→ Spawn de l'acteur
5. Control enregistre dans DistributedActorRegistry  
actorId → (serviceId, microservice URL)
6. Réponse au client avec l'actorId

# Architecture SAF (Core / Runtime / Control)

Schéma tri-couches :



# Micro-services (Client / Ville / Capteurs) - Client

**Son rôle est d'être :** Un **utilisateur (client)** dans le système IoT City. Chaque instance du frontend crée automatiquement un ClientActor pour recevoir les données climatiques.

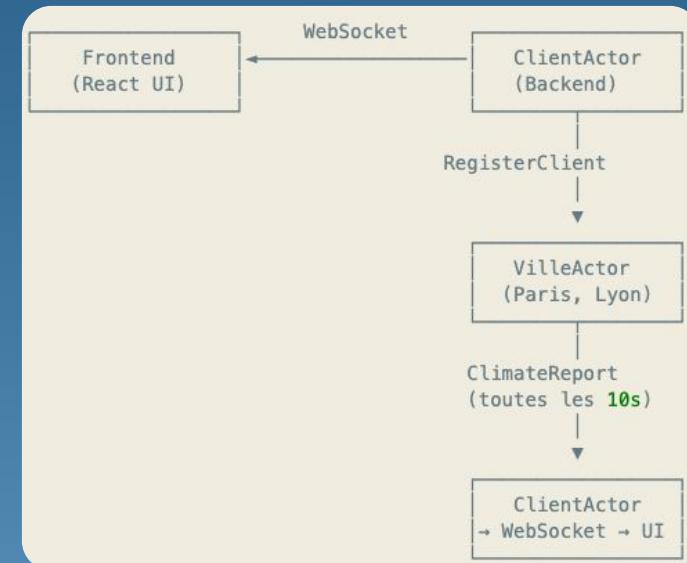
**Ses caractéristiques sont :**

sessionId	Identifiant unique de la session utilisateur
currentVilleId	Ville à laquelle le client est actuellement inscrit
WebSocket	Connexion temps réel vers le frontend

**Ses messages traités sont :**

ClimateReport	Rapport climatique reçu d'une ville → forward vers WebSocket
VilleInfoResponse	Informations d'une ville → forward vers WebSocket
ENTER:villeId	S'inscrire à une ville pour recevoir ses rapports
LEAVE	Se désinscrire de la ville actuelle
GET_VILLE_INFO:villeId	Demander les informations d'une ville

**Avec un flux de communication :**



# Micro-services (Client / Ville / Capteurs) - Ville

**Son rôle est d'être :** Une ville dans le système IoT. Agrège les données des capteurs et diffuse des rapports climatiques aux clients inscrits.

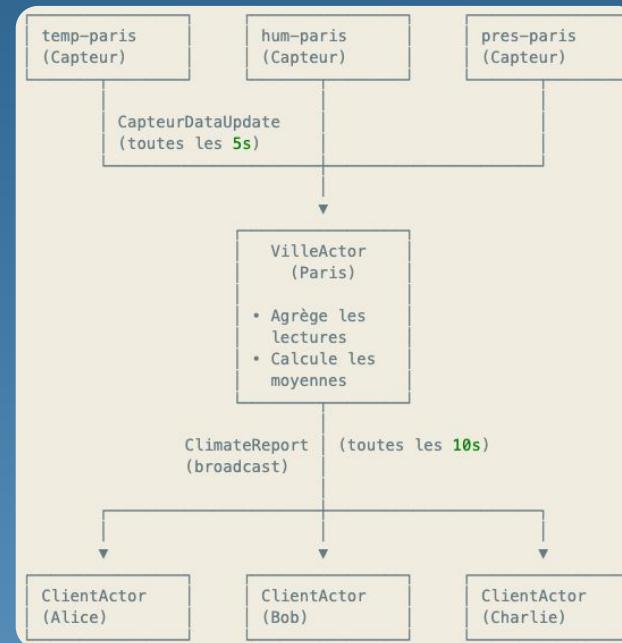
**Ses caractéristiques sont :**

<b>name</b>	Nom de la ville (Paris, Lyon, Marseille)
<b>climateConfig</b>	Configuration climatique (moyennes, variance)
<b>registeredClients</b>	Liste des clients inscrits pour recevoir les rapports
<b>registeredCapteurs</b>	Liste des capteurs associés à cette ville
<b>latestReadings</b>	Dernières lectures de chaque capteur

**Ses messages traités sont :**

<b>RegisterClient</b>	Un client s'inscrit pour recevoir les rapports
<b>UnregisterClient</b>	Un client se désinscrit
<b>RegisterCapteur</b>	Enregistrement d'un nouveau capteur
<b>CapteurDataUpdate</b>	Réception d'une lecture de capteur
<b>RequestVilleInfo</b>	Demande d'informations sur la ville

**Avec un flux de communication :**



# Micro-services (Client / Ville / Capteurs) - Capteur

**Son rôle est d'être :** Un capteur IoT physique. Génère des mesures périodiques basées sur la configuration climatique de sa ville.

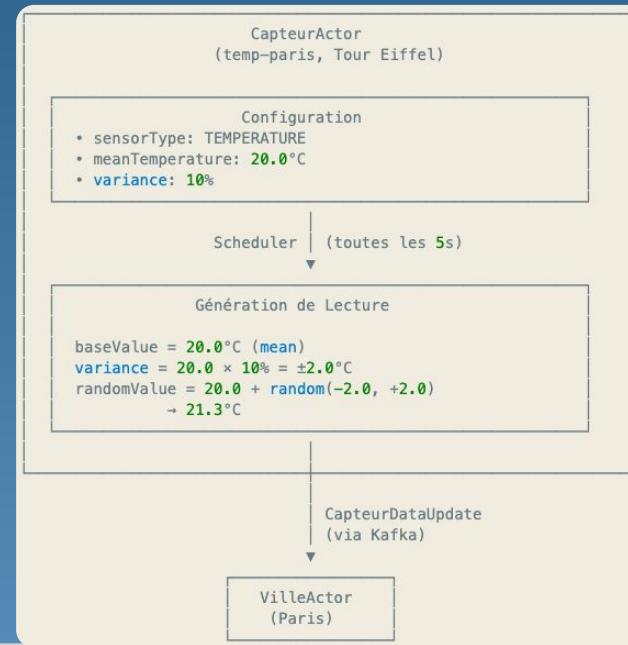
**Ses caractéristiques sont :**

sensorType	Type de capteur : TEMPERATURE, HUMIDITY, PRESSURE
villeId	Identifiant de la ville associée
location	Emplacement physique (Tour Eiffel, Louvre, etc.)
villeConfig	Configuration climatique reçue de la ville

**Ses messages traités sont :**

AssociateCapteurToVille	Association du capteur à une ville
ClimateConfigUpdate	Mise à jour de la configuration climatique

**Avec un flux de production de données :**



# ActorSystem & ActorFactory : créer des acteurs

## ActorFactory

- ActorFactory : transforme (type + params) en instance d'Actor concrète.

## ActorSystem

- spawn(type, params) : génère un **id**, appelle la factory, crée un **ActorRef**, enregistre l'instance
- Registry interne : associe actorId → ActorInstance.

```
@Override  
public ActorRef spawn(String type, Map<String, Object> params) {  
    String id = UUID.randomUUID().toString();  
    return spawn(type, id, params);  
}
```

## Cycle de vie

stop(id) / restartActor(id) / health : contrôle d'état (CREATED/RUNNING/STOPPED/FAILED).

```
// Transition to STARTING state  
instance.setState(ActorLifecycleState.STARTING);  
  
try {  
    actor.preStart();  
    // Transition to RUNNING state after successful start  
    instance.setState(ActorLifecycleState.RUNNING);  
    System.out.println("Acteur créé et démarré: " + id + " (type: " + type + " - State: RUNNING");  
} catch (Exception e) {  
    // Transition to FAILED state on startup failure  
    instance.setState(ActorLifecycleState.FAILED);  
    System.err.println("Erreur lors du démarrage de l'acteur " + id + ": " + e.getMessage());  
    e.printStackTrace();  
}
```

# Routage des messages vers le Runtime

## Flux end-to-end (chaîne de traitement)

- Entrée Runtime (API / endpoint) : réception de la requête
- Résolution de cible : on identifie l'acteur via actorId
- Encapsulation & dispatch : mise au format interne + envoi au dispatcher
- Mailbox / queue : mise en file d'attente (ordre + asynchrone)
- Livraison à l'acteur : dépilement + appel du handler (onReceive / handle)

```
ActorRef target = actorSystem.getActor(command.getTargetActorId());
if (target == null) {
    Log.error("[{}] Actor not found: {}", serviceName, command.getTargetActorId());
    return ResponseEntity.NotFound().build();
}

ActorRef sender = command.getSenderActorId() != null
    ? actorSystem.getActor(command.getSenderActorId())
    : null;

target.tell(command.getMessage(), sender);
```

```
@Override
public void tell(Message message, ActorRef sender) {
    system.processMessage(id, message, sender, null);
}

@Override
public CompletableFuture<Object> ask(Message message, long timeout, TimeUnit unit) {
    CompletableFuture<Object> future = new CompletableFuture<>();
    future.orTimeout(timeout, unit);
    system.processMessage(id, message, this, future);
    return future;
}
```

## Composants clés

- ActorRef : point d'appel tell/ask côté client interne.
- ActorSystem : valide, route, et délègue au mécanisme de queue/dispatch.
- Mailbox / Dispatcher : buffer + exécution contrôlée (threading, backpressure).

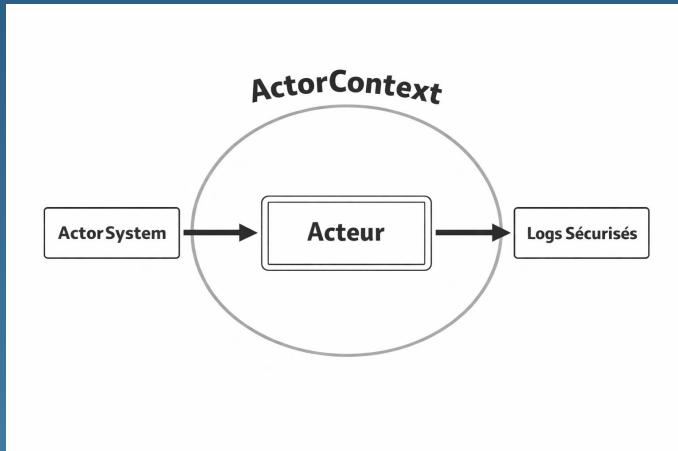
# Transport des messages : Mailbox & stockage in memory

- Interface Mailbox : Chaque acteur possède une boîte aux lettres stockant les messages en mémoire dans l'ordre d'arrivée.
- Thread-Safe : Les messages sont traités séquentiellement sans perte tant que la RAM le permet
- Ordonnancement garanti : Les messages d'un même acteur sont traités dans l'ordre FIFO

Interface Mailbox ->

```
package com.acme.saf.actor.core;
public interface Mailbox{
    void enqueue(Message message);
    Message dequeue();
    boolean isEmpty();
    int size();
    void clear();
}
```

# L'isolation



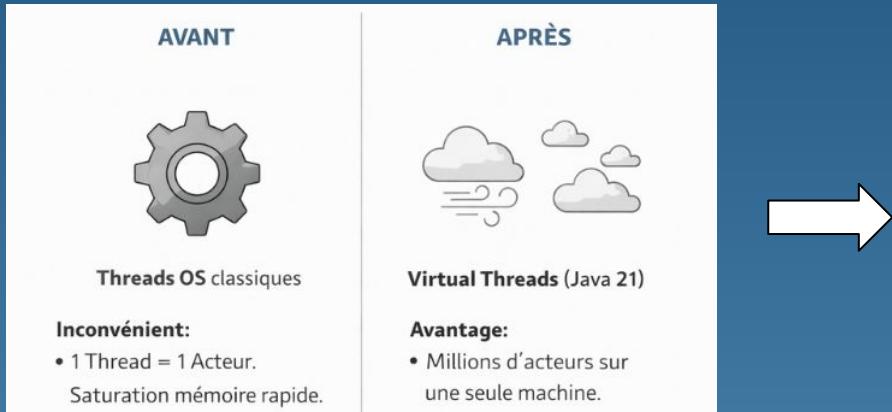
- L'acteur n'a aucun accès direct aux ressources système (fichiers, threads, etc) - **Sécurité**
- L'ActorContext est la seule passerelle pour intéragir avec l'extérieur - **Point d'entrée unique**
- Gestion thread-safe des logs et propagation des ID de corrélation - **Logging et tracabilité**

```
// Extrait de l'interface ActorContext
public interface ActorContext {
    // Identification
    ActorRef self();
    ActorRef sender();

    // Logging structuré et tracing
    ActorLogger getLogger();
    String getCorrelationId();

    // Communication externe
    void sendToWebSocket(Object message);
    ActorRef actorFor(String actorId);
}
```

# La scalabilité



- Remplacement des threads OS coûteux par des threads gérés par la JVM - Thread virtuels légers
- Empêche la famine CPU en limitant le traitement par acteur (boucle while du code) - Équité

```

@Component
public class VirtualThreadDispatcher implements Dispatcher {

    private final ExecutorService executor =
        Executors.newVirtualThreadPerTaskExecutor();

    // Nombre de messages traités par chaque acteur par équité
    private static final int THROUGHPUT_LIMIT = 50;

    private void processMailbox(...) {

        // Boucle de consommation limitée pour l'équité
        while (processedCount < THROUGHPUT_LIMIT) {
            // Récupérer le prochain message
            Message message = mailbox.dequeue();
            // Traitement du message par l'acteur...
            processedCount++;
        }
        // Si encore des messages on se replanifie
        if (!mailbox.isEmpty()) {
            dispatch(...);
        }
    }
}

```

# Passage à l'échelle via Kafka

- Inter-Pod Messaging System : Acteurs (CapteurActor, VilleActor et ClientActor) communiquent entre pods via un broker Kafka à travers les classes “RemoteMessageTransport” et “InterPodMessaging”
- Communication Hybride : Local (même pod via tell()) + Distribué (cross-pod via Kafka Topic)
- Producer/Consumer : CapteurActor publie ses mesures, VilleActor les lit de manière asynchrone. ClientActor s'abonne à VilleActor afin de récupérer les données météo de VilleActor de manière automatique
- Sérialisation Type-Safe : Une classe convertit les objets métiers (CapteurDataUpdate) en JSON pour le transport via Kafka
- Scalabilité Horizontale : Ajouter des pods = ajouter des producteurs, tous publient automatiquement sur les Topics partagés sans reconfiguration

Les classes citées sur cette slide sont visibles sur les slides suivantes

# Classe “RemoteMessageTransport”

```
public interface RemoteMessageTransport {  
  
    void sendMessage(String actorUrl, Message message, ActorRef sender) throws Exception;  
  
    CompletableFuture<Object> askMessage(String actorUrl, Message message, long timeout, TimeUnit un  
  
    boolean checkActorExists(String actorUrl) throws Exception;  
  
    void stopActor(String actorUrl) throws Exception;  
  
    ActorLifecycleState getActorState(String actorUrl) throws Exception;  
}
```

# Classe “InterPodMessaging”

```
// Source code is decompiled from a .class file using FernFlower decompiler (from IntelliJ IDEA).
package com.acme.saf.saf_runtime.messaging;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class InterPodMessaging {
    private static final Logger logger = LoggerFactory.getLogger(InterPodMessaging.class);
    private static volatile InterPodMessaging instance;
    private final MessageProducer producer;
    private final MessageConsumer consumer;

    private InterPodMessaging(MessageProducer producer, MessageConsumer consumer) {
        this.producer = producer;
        this.consumer = consumer;
    }

    public static InterPodMessaging initialize(MessageProducer producer, MessageConsumer consumer) {
        if (instance != null) {
            logger.warn("InterPodMessaging already initialized, reinitializing");

            try {
                instance.shutdown();
            } catch (Exception var3) {
                logger.error("Error shutting down previous instance", var3);
            }
        }

        instance = new InterPodMessaging(producer, consumer);
        logger.info("InterPodMessaging initialized");
        return instance;
    }

    public static InterPodMessaging getInstance() {
        if (instance == null) {
            throw new IllegalStateException("InterPodMessaging not initialized. Call initialize() first");
        } else {
    
```

```
        } else {
            return instance;
        }
    }

    public MessageProducer getProducer() {
        return this.producer;
    }

    public MessageConsumer getConsumer() {
        return this.consumer;
    }

    public boolean isConnected() {
        return this.producer.isConnected() && this.consumer.isConnected();
    }

    public void shutdown() throws Exception {
        logger.info("Shutting down InterPodMessaging");

        try {
            this.producer.close();
            this.consumer.close();
            instance = null;
        } catch (Exception var2) {
            logger.error("Error during shutdown", var2);
            throw var2;
        }
    }
}
```

# Intégration ActorSystem / ActorFactory

Intégration : spawn() dans DefaultActorSystem

```
@Override
public ActorRef spawn(String type, String id, Map<String, Object> params) {
    // 1. Factory crée l'instance concrète
    Actor actor = factory.create(type, params);
    if (actor == null) {
        throw new IllegalArgumentException("Type non supporté: " + type);
    }

    // 2. Création de la référence et du conteneur
    ActorRefImpl ref = new ActorRefImpl(id, type, this);
    ActorInstance instance = new ActorInstance(actor, ref);
    actors.put(id, instance);

    // 3. Injection du contexte d'exécution
    Logger logger = new SimpleLogger(id);
    DefaultActorContext context = new DefaultActorContext(
        ref, logger, mailbox, actor
    );
    injectContext(actor, context);
}
```

```
// 4. Lifecycle : STARTING → preStart() → RUNNING
instance.setState(ActorLifecycleState.STARTING);
actor.preStart();
instance.setState(ActorLifecycleState.RUNNING);

return ref;
}
```

## Étapes clés :

- **Factory Pattern** : Délégation de la création
- **ActorRef** : Référence immuable pour l'envoi de messages
- **ActorContext** : Injection des capacités (logging, sender, mailbox)
- **Lifecycle** : Transition d'état avec hook `preStart()`

# ActorContext : Injection des capacités

ActorContext : Outilage de l'acteur

Injection par réflexion :

```
private void injectContext(Actor actor, ActorContext context) {
    try {
        Method setContextMethod = actor.getClass()
            .getMethod("setContext", ActorContext.class);
        setContextMethod.invoke(actor, context);
    } catch (NoSuchMethodException e) {
        // Optionnel : acteur sans contexte
    }
}
```

DefaultActorSystem.java (lignes 79-89)

```
public class ClientActor implements Actor {
    private ActorContext context;

    public void setContext(ActorContext ctx) { this.context = ctx; }

    @Override
    public void receive(Message message) {
        context.logInfo("Traitement: " + message.getPayload());
        ActorRef sender = context.sender();
        // ... logique métier
    }
}
```

Interface ActorContext.java

```
public interface ActorContext {
    ActorRef self();           // Référence à soi-même
    ActorRef sender();         // Émetteur du message courant
    void logInfo(String msg);  // Logging intégré
    void logError(String msg, Throwable t);
    Mailbox getMailbox();      // Accès à la boîte aux lettres
    ActorRef actorFor(String id); // Résolution d'autres acteurs
    void sendToWebSocket(Object message); // Communication client
}
```

# Supervision (Gestion des pannes)

Supervision : Résilience face aux pannes

Superviser → Décision automatique de résilience

Les 3 politiques de supervision dans SAF :

Politique	Action	Cas d'usage
RESTART	Détruire et recréer (état perdu)	État corrompu, erreur récupérable
RESUME	Ignorer l'erreur, continuer	Erreur transitoire, non critique
STOP	Arrêt définitif	Erreur fatale, intervention manuelle

**Configuration :** Chaque agent a sa politique définie à la création via AgentCreateRequest

```
public record AgentCreateRequest(
    String type,
    String host,
    int port,
    SupervisionPolicy policy // RESTART, RESUME, ou STOP
) {}
```

# SupervisionService : Application des politiques

Centralisation de la supervision

```
@Service
public class SupervisionService {

    private final ControlService controlService;

    public void handle(AgentView agent) {
        System.out.println("Supervision: " + agent.id()
                           + " avec politique " + agent.policy());

        switch (agent.policy()) {
            case RESTART -> restart(agent);
            case RESUME   -> resume(agent);
            case STOP      -> stop(agent);
        }
    }
}
```

```
public void resume(AgentView agent) {
    System.out.println("Reprise sans intervention: " + agent.id());
    // Aucune action : l'agent continue son exécution
}

public void stop(AgentView agent) {
    System.out.println("Arrêt définitif: " + agent.id());
    controlService.destroy(agent.id());
}
```

```
public void restart(AgentView agent) {
    System.out.println("Redémarrage de " + agent.id());

    // 1. Destruction de l'agent défaillant
    controlService.destroy(agent.id());

    // 2. Recréation avec les mêmes paramètres
    controlService.spawn(new AgentCreateRequest(
        agent.type(),
        agent.host(),
        agent.port(),
        agent.policy() // Conservation de la politique
    ));

    System.out.println("Nouvel agent créé");
```

Centralisation : Toute la logique de supervision dans un seul service

# Détection et traitement des pannes

Intégration Monitoring → Supervision → Quarantaine

```
@Scheduled(fixedRate = 30000) // Toutes les 30 secondes
public void checkAgentHealth() {
    Instant now = Instant.now();

    controlService.list().forEach(agent -> {

        // Ignorer les agents déjà en quarantaine
        if (quarantine.contains(agent.id())) {
            return;
        }

        // Détection : agent sans heartbeat récent
        if (isAgentStale(agent.id(), now)) {
            System.out.println("Agent " + agent.id() + " INACTIF");

            // 1. Supervision : RESTART/RESUME/STOP
            supervisionService.handle(agent);

            // 2. Quarantaine : éviter boucle infinie
            quarantine.add(agent.id());
        }
    });
}
```

```
@Scheduled(fixedRate = 10_000, initialDelay = 15_000)
public void checkServicesHealth() {
    for (ServiceInfo service : serviceRegistry.getAllServices()) {
        checkServiceHealth(service);
    }
}

private Mono<Boolean> performActiveHealthCheck(ServiceInfo service) {
    String healthUrl = service.getUrl() + "/actuator/health";

    return webClient.get()
        .uri(healthUrl)
        .retrieve()
        .bodyToMono(String.class)
        .timeout(HEALTH_CHECK_TIMEOUT)
        .map(response -> true)
        .onErrorResume(error -> Mono.just(false));
}
```

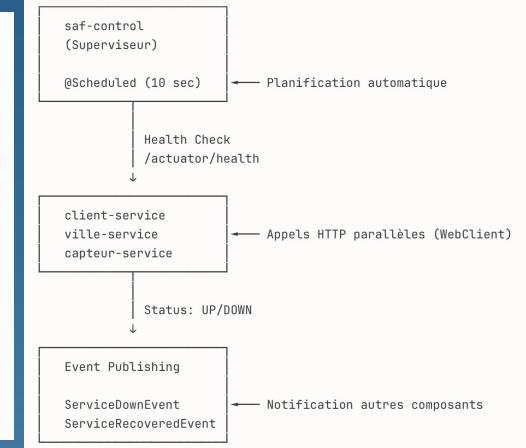
```
public void processEvent(RuntimeEvent event) {
    if (event.type().equals("ActorFailed")) {

        // 1. Récupération de l'agent
        AgentView agent = controlService.get(event.agentId())
            .orElseThrow(() -> new IllegalStateException("Agent introuvable"));

        // 2. Application politique de supervision
        supervisionService.handle(agent);

        // 3. Mise en quarantaine
        quarantine.add(event.agentId());

        System.out.println("Agent supervisé et mis en quarantaine");
    }
}
```



Bénéfices combinés (@Scheduled + Asynchrone) :

- **Automatique** : vérification toutes les 10 secondes
- **Parallèle** : tous les services vérifiés simultanément
- **Non-bloquant** : API SAF-Control reste réactive
- **Résilient** : timeout configuré + gestion erreurs
- **Observable** : events publiés pour orchestration

## Conclusion - Forces

### Framework solide et réutilisable :

- **Architecture 3 couches** claire : Actor-Core → Runtime → Control
- **Philosophie plugin** : le framework est agnostique métier
- **Inspiré d'Akka** : patterns éprouvés (mailbox, supervision, tell/ask)

### Observabilité intégrée :

- **Spring Boot Actuator + Prometheus** pour le monitoring
- **Health checks** automatiques et auto-récupération des services
- Métriques personnalisées (nombre d'acteurs, taille des mailbox)

### Application IoT City Fonctionnelle :

- Démonstration concrète du framework avec 3 types d'acteurs
- Communication temps réel via WebSocket
- Flux de données réaliste (Capteurs → Villes → Clients)

### Bonne cohésion d'équipe :

- Travail collaboratif efficace
- Planification des tâches via Jira
- Orienté sur le développement des compétences de chacun

## Conclusion - Perspectives et Améliorations

### Scalabilité horizontale avec Kubernetes

- Auto-scaling des microservices selon la charge
- Service Mesh (Istio) pour la gestion du trafic
- Résilience renforcée avec pods répliqués

### Amélioration du Frontend :

- Dashboard plus riche avec visualisations avancées
- Cartes interactives des villes et capteurs
- Meilleure UX/UI et responsive design

### Fonctionnalités framework :

- Persistence des acteurs (Event Sourcing / CQRS)
- Circuit Breaker intégré (Resilience4j)
- Tests de charge et benchmarks

### Enrichissement applicatif :

- Plus de types de capteurs (pollution, bruit, trafic)
- Alertes et notifications (seuils dépassés)
- Ajout de plus d'actions complexe dans chaque micro-service

**Note :** Focus sur le framework réutilisable plutôt que sur l'application métier, créant ainsi un socle solide pour de futures évolutions, nous permettant d'engranger plus de compétences sur la partie réellement technique de Spring Boot.

# Capture de démo

## IoT City Monitor

Real-time climate monitoring across cities

Session: 2a4b2df9... | Actor: 9e80faf6...



### Available Cities

Select a city to monitor

Marseille

50 reports

ACTIVE

Paris

2 reports

ACTIVE

Lyon

ACTIVE

Select a city to view real-time climate data

### Connection Status

Connected

# Capture de démo

## IoT City Monitor

Real-time climate monitoring across cities

Session: 2a4b2df9... | Actor: 9e80faf6...



### Available Cities

Select a city to monitor

Marseille

50 reports

ACTIVE

Paris

2 reports

ACTIVE

Lyon

ACTIVE

Leave Current City

### Connection Status

Connected



Waiting for climate data...

# Capture de démo

## IoT City Monitor

Real-time climate monitoring across cities

Session: 2a4b2df9... | Actor: 9e80faf6...



### Available Cities

Select a city to monitor

Marseille

50 reports

ACTIVE

Paris

2 reports

ACTIVE

Lyon

1 reports

ACTIVE

Leave Current City

### Connection Status

Connected

Lyon

Last updated: 6:08:32 PM

3 sensors active

Pressure

**929.68 hPa**

Humidity

**58.58%**

Temperature

**20.89°C**

● Live updates - Receiving climate reports from Lyon

# Capture de démo

## IoT City Monitor

Real-time climate monitoring across cities  
Session: 2a4b2df9... | Actor: 9e80faf6...

### Available Cities

Select a city to monitor

Marseille	50 reports	ACTIVE
Paris	2 reports	ACTIVE
Lyon	4 reports	ACTIVE

[Leave Current City](#)

### Connection Status

Connected

### Lyon

Last updated: 6:09:02 PM

3 sensors active

Pressure: **998.88 hPa**

Humidity: **61.25%**

Temperature: **19.14°C**

Live updates - Receiving climate reports from Lyon

### Climate Time Series

4 data points collected during this session

The chart displays three data series over four time points: 18:08:32, 18:08:42, 18:08:52, and 18:09:02. The Y-axis has two scales: Temp (C) and Pressure (hPa) on the left, and Humidity (%) on the right.

Time	Temp (C)	Humidity (%)	Pressure (hPa)
18:08:32	15	20	105
18:08:42	15	35	105
18:08:52	15	65	105
18:09:02	15	45	105

Legend: Temperature (C) (Red line with dots), Humidity (%) (Green line with dots), Pressure (hPa) (Blue line with dots)

### Report History

Last 20 reports received

# Capture de démo

Marseille 50 reports ACTIVE

Paris 2 reports ACTIVE

Lyon 5 reports ACTIVE

[Leave Current City](#)

**Pressure**  
**1055.88 hPa**

**Humidity**  
**56.59%**

**Temperature**  
**18.19°C**

● Live updates - Receiving climate reports from Lyon

**Connection Status**  
Connected

**Climate Time Series**  
5 data points collected during this session

The chart displays three data series: Temperature (red line with circles), Humidity (green line with circles), and Pressure (blue line with circles). The x-axis shows time points: 18:08:32, 18:08:42, 18:08:52, 18:09:02, and 18:09:12. The y-axis has two scales: Temperature (C) from 15 to 75 and Pressure (hPa) from 90 to 110. A callout highlights a data point at 18:08:42 with values: Temperature: 18.8°C, Humidity: 60.6%, and Pressure: 974 hPa.

Temp (C) / Humidity (%) Pressure (hPa)

18:08:32 18:08:42 18:08:52 18:09:02 18:09:12

15 30 45 60 75 90 95 100 105 110

18:08:42  
Temperature : 18.8°C  
Humidity : 60.6%  
Pressure : 974 hPa

→ Temperature (C) → Humidity (%) → Pressure (hPa)

**Report History**  
Last 20 reports received

Time	Temperature (C)	Humidity (%)	Pressure (hPa)
6:09:12 PM	T: 18.2 C	H: 56.6%	P: 1056 hPa
6:09:02 PM	T: 19.1 C	H: 61.3%	P: 999 hPa
6:08:52 PM	T: 20.1 C	H: 56.7%	P: 1081 hPa

Latest

# Capture de démo

## IoT City Monitor

Real-time climate monitoring across cities  
Session: 2a4b2df9... | Actor: 9e80faf6...

**Available Cities**  
Select a city to monitor

Marseille      50 reports      ACTIVE

Paris      2 reports      ACTIVE

Lyon      5 reports      ACTIVE

[Leave Current City](#)

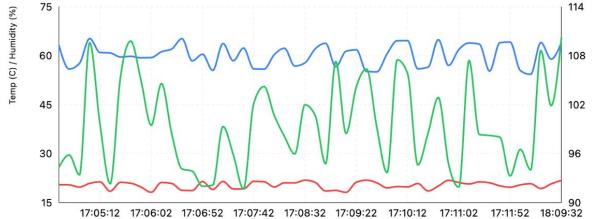
**Marseille**  
Last updated: 6:09:32 PM      3 sensors active

Pressure: **1111.63 hPa**      Humidity: **64.61%**      Temperature: **21.88°C**

● Live updates - Receiving climate reports from Marseille

**Connection Status**  
Connected

**Climate Time Series**  
50 data points collected during this session



The chart displays three data series over time (from 17:05:12 to 18:09:32). The Y-axis has two scales: Temperature (C) from 15 to 75 and Pressure (hPa) from 90 to 114. The X-axis shows dates and times. The Temperature series (red line) fluctuates between 15°C and 25°C. The Humidity series (green line) fluctuates between 30% and 70%. The Pressure series (blue line) fluctuates between 98 hPa and 108 hPa.

Legend: Temperature (C)   Humidity (%)   Pressure (hPa)

**Report History**  
Last 20 reports received

# Capture de démo

Marseille      50 reports      ACTIVE

Paris      2 reports      ACTIVE

Lyon      5 reports      ACTIVE

[Leave Current City](#)

**Connection Status**  
Connected

Pressure: **1075.56 hPa**      Humidity: **61.98%**      Temperature: **21.26°C**

● Live updates - Receiving climate reports from Marseille

**Climate Time Series**  
50 data points collected during this session

Temp (C) / Humidity (%)      Pressure (hPa)

17:05:22 17:06:12 17:07:02 17:07:52 17:08:42 17:09:32 17:10:22 17:11:12 17:12:02 18:09:42

Temperature (C)      Humidity (%)      Pressure (hPa)

**Report History**  
Last 20 reports received

Time	Temperature (T)	Humidity (H)	Pressure (P)	Action
6:09:42 PM	T: 21.3 C	H: 62.0%	P: 1076 hPa	<a href="#">Latest</a>
6:09:32 PM	T: 21.9 C	H: 64.6%	P: 1112 hPa	
6:09:22 PM	T: 20.9 C	H: 58.3%	P: 1011 hPa	

# Bibliographie

- <https://docs.spring.io/spring-kafka/reference/kafka.html>
- <https://docs.spring.io/spring-boot/reference/features/task-execution-and-scheduling.html>
- <https://docs.spring.io/spring-batch/reference/index.html>
- <https://docs.spring.io/spring-boot/reference/actuator/index.html>
- <https://docs.spring.io/spring-boot/reference/using/auto-configuration.html>
- <https://www.baeldung.com/spring-boot-custom-starter>
- <https://dzone.com/articles/spring-boot-3-creating-a-custom-starter>
- <https://mkyong.com/spring-boot/spring-boot-conditionalonproperty-example/>
- <https://docs.spring.io/spring-security/reference/servlet/oauth2/login/index.html>
- <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/index.html>
- <https://spring.io/projects/spring-authorization-server>
- <https://docs.spring.io/spring-authorization-server/reference/index.html>