Documentação do Sistema de Monitoramento de Performance e Logs

1. Visão Geral

O sistema é uma solução avançada para **monitoramento de APIs externas** e **auto-monitoramento de aplicações**, permitindo a **detecção automatizada de anomalias de desempenho**. Ele analisa métricas como **uso de CPU, consumo de memória e tempo de resposta**, identificando comportamentos fora do padrão e **gerando alertas em tempo real** via e-mail.

Desenvolvido com **Spring Boot**, o sistema utiliza **PostgreSQL** para armazenar métricas de desempenho e **MongoDB** para logs de monitoramento, garantindo flexibilidade no processamento de dados. A comunicação assíncrona é gerenciada via **Kafka**, permitindo escalabilidade e alta disponibilidade.

Com uma arquitetura modular e distribuída, a aplicação é ideal para ambientes que exigem **monitoramento contínuo, resposta rápida a falhas e automação na detecção de problemas de performance**.

2. Arquitetura do Sistema

A aplicação é estruturada seguindo o princípio de **arquitetura em camadas**, garantindo **modularidade, escalabilidade e manutenção simplificada**. Seus componentes estão organizados de forma lógica para otimizar o processamento de dados, a comunicação entre serviços e a detecção de anomalias. A seguir, são detalhadas as principais camadas e seus respectivos papéis no sistema.

2.1 Camada de Modelos de Dados

A **camada de modelos** é responsável por representar as entidades centrais do sistema, garantindo a persistência e estruturação das informações coletadas durante o monitoramento. Cada modelo é armazenado em um banco de dados adequado à sua finalidade, otimizando a consulta e o processamento dos dados.

- **Log.java** (**MongoDB**) – Armazena registros detalhados dos eventos de **auto-monitoramento**, incluindo informações sobre **uso de recursos, tempos de resposta e status operacional** da aplicação.
- **Performance.java** (**PostgreSQL**) – Registra **métricas de desempenho das APIs monitoradas**, como **tempo de resposta, uso de CPU, consumo de memória e taxa de erro**, permitindo análises detalhadas sobre a saúde e eficiência dos serviços externos.

Essa estrutura garante um armazenamento eficiente e escalável, separando logs operacionais de métricas analíticas para um monitoramento mais preciso.

2.2 Camada de Repositórios

A **camada de repositórios** é responsável pela persistência dos dados da aplicação, abstraindo as operações de banco de dados e fornecendo métodos eficientes para **consulta, armazenamento e recuperação de informações**. Essa camada garante a integração fluida entre a lógica de negócio e a base de dados, otimizando a manipulação de registros.

- **LogRepository.java** (**MongoDB**) – Gerencia a persistência dos **logs de auto-monitoramento**, fornecendo métodos específicos para buscas por **componente, nível de log e intervalos de tempo**, permitindo análises detalhadas do comportamento da aplicação.

◇ **PerformanceRepository.java** (**PostgreSQL**) – Responsável pelo armazenamento e consulta das **métricas de desempenho das APIs monitoradas**, permitindo buscas por **identificador do sistema, períodos específicos e tempos de resposta**, auxiliando na análise de desempenho das aplicações monitoradas.

Essa abordagem proporciona uma organização eficiente dos dados, utilizando **MongoDB** para registros dinâmicos de logs e **PostgreSQL** para métricas estruturadas e análises aprofundadas.

2.3 Camada de Serviços

A **camada de serviços** é o núcleo das **regras de negócio** do sistema, sendo responsável pelo **processamento, análise e gerenciamento** das informações coletadas. Essa camada atua diretamente na persistência dos dados e na automação de processos críticos, como **monitoramento de APIs, detecção de anomalias e envio de alertas**.

◇ **LogService.java** – Gerencia os **logs de auto-monitoramento**, garantindo o registro e a recuperação eficiente de eventos do sistema. Permite buscas por **componente, nível de severidade e intervalos de tempo**.Garante a **rastreabilidade dos eventos** da aplicação e ajuda na **identificação de falhas operacionais**.

◇ **PerformanceService.java** – Responsável por **processar, validar e armazenar métricas de desempenho** das APIs monitoradas. Suporta consultas por **sistema monitorado, períodos específicos e tempos de resposta**. Permite a **análise histórica de desempenho**, ajudando na **otimização e escalabilidade das aplicações**.

◇ **AnomalyDetectionService.java** – Executa a análise de **dados de monitoramento**, verificando **uso excessivo de CPU, memória e tempos de resposta elevados**. Classifica eventos como **críticos, moderados ou normais**, acionando alertas quando necessário. Implementa uma lógica de **threshold configurável**, permitindo **ajustes dinâmicos** dos limites de anomalia.

◇ **EmailAlertService.java** – Gera notificações de anomalias críticas e as encaminha por e-mail. Possui suporte para **formatação automática das mensagens**, melhorando a clareza dos alertas. **Automação de respostas a falhas**, garantindo que equipes técnicas sejam informadas **imediatamente** sobre problemas críticos.

◇ **HealthCheckService.java** – Executa requisições de **health check** em serviços externos, verificando sua disponibilidade e tempo de resposta. Evita **interrupções inesperadas** ao identificar APIs fora do ar e fornecer **alertas preventivos**.

2.4 Camada de Comunicação Assíncrona

A **camada de comunicação assíncrona** é responsável pelo **envio e consumo de eventos** no sistema, garantindo a **troca eficiente de informações** entre os serviços sem bloqueios ou dependências diretas. Essa abordagem melhora a escalabilidade e a resiliência da aplicação, pois os serviços podem processar mensagens de forma desacoplada.

Nesta camada, **Kafka** é utilizado como **middleware de mensagens**, permitindo a **transmissão de eventos em tempo real** e facilitando a automação de processos, como **detecção de anomalias e envio de alertas**.

◇ **KafkaMessageProducer.java** – Responsável por **enviar eventos para os tópicos Kafka**, garantindo que as informações de monitoramento e alertas sejam transmitidas de forma assíncrona para outros serviços que precisam processá-las. Permite **distribuir eventos** entre múltiplos consumidores, garantindo **alta disponibilidade** e **tolerância a falhas**.

◇ **AlertEmailConsumer.java** – Monitora os tópicos Kafka em busca de **eventos de anomalias e alertas**, processando-os e acionando o **EmailAlertService** para **notificar os responsáveis** sobre problemas detectados. Automatiza **respostas rápidas** a falhas no sistema, permitindo que **alertas críticos sejam enviados imediatamente** para mitigar impactos.

Tópicos Kafka Criados (KafkaConfig.java)

1. alert-topic → Alertas de logs internos.
2. alert-performance-topic → Alertas de desempenho.
3. performance-topic → Registros de métricas.

Cada tópico Kafka funciona como um **canal de comunicação especializado**, permitindo que os serviços troquem mensagens de forma **desacoplada e eficiente**.

2.5 Camada de Monitoramento Agendado

A **camada de monitoramento agendado** é responsável por **automatizar a coleta de métricas e verificar o funcionamento dos sistemas monitorados**. Essa camada garante que a aplicação possa **identificar problemas proativamente**, sem a necessidade de intervenção manual, tornando o processo de monitoramento contínuo e eficiente.

Os serviços desta camada executam tarefas programadas para **analisar a saúde da aplicação e das APIs externas**, além de **gerar logs e métricas de desempenho**.

◇ **SelfMonitoringScheduler.java** – Executa verificações periódicas da própria aplicação a cada **60 segundos**, registrando informações sobre **uso de CPU, consumo de**

memória, tempo de resposta e disponibilidade do sistema. Caso sejam detectadas anomalias, ele aciona o **AnomalyDetectionService**, que avalia os dados e dispara alertas, se necessário. Garante a **autonomia do sistema**, permitindo que ele **detecte e registre falhas automaticamente**, sem depender de monitoramento externo.

- **MonitoringPerformance.java** – Verifica a saúde e o desempenho das APIs monitoradas, registrando **tempo de resposta, status da API, uso de recursos e possíveis falhas**. Caso um serviço esteja instável ou inoperante, ele aciona a detecção de anomalias e **gera alertas** via Kafka para notificar os administradores. Permite uma **resposta rápida a falhas externas**, garantindo maior **disponibilidade dos serviços monitorados**.

Como Funciona o Monitoramento?

1. O **SelfMonitoringScheduler** verifica **o próprio sistema** e gera **logs detalhados**.
2. O **MonitoringPerformance** monitora **APIs externas** e registra métricas de desempenho.
3. Ambos os serviços analisam os dados usando o **AnomalyDetectionService**.
4. Se forem detectadas **anomalias críticas**, os eventos são enviados para o **Kafka**, acionando alertas via e-mail.

2.6 Camada de Controle (Controllers)

A **camada de controle** é responsável por **expor a API da aplicação**, permitindo que clientes externos interajam com os dados de **logs e métricas de desempenho**. Os controladores mapeiam as requisições HTTP para os serviços correspondentes, garantindo o processamento correto das informações.

Os **endpoints** foram estruturados para fornecer **acesso eficiente aos dados**, permitindo **monitoramento e análise** dos sistemas monitorados.

- **PerformanceController.java** – Expõe **endpoints para monitoramento de APIs externas** e recuperação de métricas coletadas. Ele se comunica com o **MonitoringPerformance** para iniciar verificações de desempenho e com o **PerformanceService** para recuperar dados armazenados.

Endpoints disponíveis:

POST /api/performance/external → Inicia o **monitoramento de uma API externa (depende de configuração pela API externa)**.

GET /api/performance/system/{systemIdentifier} → Retorna **métricas de desempenho de um sistema específico**.

◇ **LogController.java** – Fornece **acesso aos logs gerados pelo sistema**, permitindo a consulta de eventos **filtrados por componente, nível de log e período de tempo**. Ele interage diretamente com o **LogService** para recuperar as informações necessárias.

Endpoints disponíveis:

GET /api/logs?component=SelfMonitor&level=INFO → Retorna **logs filtrados por componente e nível**.

GET /api/logs?start=2025-02-14T00:00:00&end=2025-02-14T23:59:59 → Recupera **logs dentro de um intervalo de tempo**.

2.7 Camada de Configuração

A **camada de configuração** centraliza os ajustes essenciais da aplicação, garantindo **segurança, comunicação assíncrona e integração com serviços externos**. Aqui são definidos os parâmetros para **controle de acesso, envio de e-mails e gerenciamento de eventos Kafka**, permitindo uma operação eficiente e segura do sistema.

Essa camada contém configurações para **autenticação, segurança, comunicação via Kafka e envio de alertas por e-mail**, tornando o sistema altamente flexível e configurável.

◇ **SecurityConfig.java** – Define as políticas de segurança da aplicação. No cenário atual, a configuração **desativa o CSRF (Cross-Site Request Forgery)** e permite **todas as requisições** sem necessidade de autenticação. Permite acesso irrestrito à API, podendo ser ajustado futuramente para **incluir autenticação via OAuth2 ou JWT** conforme necessário.

◇ **MailConfig.java** – Gerencia os **parâmetros de envio de e-mails**, utilizados para disparar **alertas de anomalias** para os responsáveis. A configuração inclui **autenticação SMTP, criptografia TLS** e definição de **timeouts** para garantir a entrega eficiente das mensagens. Suporta integração com **provedores de e-mail externos (ex.: Gmail, Outlook, SMTP corporativo)**, permitindo **notificações automáticas** sobre problemas detectados no sistema.

◇ **KafkaConfig.java** –Define os **tópicos Kafka** utilizados para comunicação assíncrona. Isso garante a **entrega confiável de eventos** entre os serviços do sistema.

1. alert-topic → Processamento de **alertas internos** de logs.
2. alert-performance-topic → Gerenciamento de **alertas de desempenho** das APIs monitoradas.
3. performance-topic → Armazenamento e processamento das **métricas de desempenho coletadas**.

3. Fluxo de Monitoramento

O **fluxo de monitoramento** da aplicação é projetado para garantir **detecção contínua de anomalias**, **registro de eventos** e **notificação imediata** de falhas. O sistema realiza **checagens automáticas** tanto da própria API quanto de **serviços externos**, analisando o desempenho e gerando alertas em caso de irregularidades.

Esse fluxo é baseado em uma **arquitetura assíncrona**, permitindo que cada componente execute sua função de forma **independente e eficiente**, garantindo **alta disponibilidade** e **resiliência** ao monitoramento.

3.1 Etapas do Monitoramento

SelfMonitoringScheduler

- Executa verificações periódicas sobre a **saúde da própria aplicação**.
- Coleta informações como **uso de CPU, memória e tempo de resposta**.
- **Armazena os logs no MongoDB** e envia os dados para análise.

MonitoringPerformance

- Faz **requisições periódicas** a serviços externos monitorados.
- Mede **tempo de resposta, status HTTP e disponibilidade da API**.
- **Armazena as métricas no PostgreSQL** para análise posterior.

AnomalyDetectionService

- Avalia os dados coletados (**logs internos e métricas de performance**).
- Verifica se há **anomalias críticas**, como **alto consumo de CPU, lentidão ou falhas**.
- Define **níveis de alerta (CRITICAL, MODERATE, NORMAL)** com base nos thresholds configurados.

KafkaMessageProducer

- Se forem detectadas **anomalias críticas ou moderadas**, os eventos são **publicados em tópicos Kafka**.
- Permite que os serviços responsáveis pelo envio de alertas sejam notificados **imediatamente**.
- Garante a **escalabilidade do sistema**, distribuindo as mensagens de forma eficiente.

AlertEmailConsumer

- Consome mensagens dos tópicos Kafka que indicam **anomalias críticas**.
- Aciona o **EmailAlertService** para **enviar notificações para os responsáveis**.

- Os e-mails contêm detalhes sobre a falha, permitindo **ação imediata**.

3.2 Benefícios da Arquitetura do Monitoramento

Monitoramento Automático, o sistema **detecta falhas** sem necessidade de intervenção manual. **Escalabilidade**, com a comunicação via **Kafka** permite processar **grandes volumes de dados** de forma assíncrona. **Alta Disponibilidade** com o monitoramento contínuo evita falhas inesperadas nos serviços monitorados. **Resposta Rápida a Falhas** por meio de alertas **imediatos por e-mail** garantem que a equipe técnica possa agir rapidamente.

4. Configuração da Aplicação

A aplicação foi projetada para ser altamente **configurável**, permitindo ajustes dinâmicos através de **variáveis de ambiente** definidas nos arquivos **`application.properties`** e **`application-dev.properties`**. Essa abordagem garante **flexibilidade** para diferentes ambientes, como **desenvolvimento, testes e produção**, sem a necessidade de alterações no código-fonte.

As configurações essenciais englobam **banco de dados, comunicação via Kafka, envio de e-mails e thresholds para detecção de anomalias**.

4.1 Configuração do Banco de Dados

A aplicação utiliza **dois bancos de dados distintos**, garantindo **eficiência no armazenamento** e **rápida recuperação de informações**:

PostgreSQL → Utilizado para armazenar **métricas de desempenho** coletadas das APIs monitoradas.

MongoDB → Responsável pelo armazenamento de **logs internos do sistema**, permitindo **consultas ágeis e flexíveis**.

As credenciais de acesso e URLs dos bancos são configuradas via **variáveis de ambiente**, garantindo maior segurança e flexibilidade:

```
# Configuração do PostgreSQL
spring.datasource.url=${SPRING_DATASOURCE_URL:jdbc:postgresql://localhost:5
432/monitor_db}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME:admin}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD:secret}
spring.datasource.driver-class-name=org.postgresql.Driver

# Configuração do MongoDB
```

```
spring.data.mongodb.uri=${SPRING_DATA_MONGODB_URI:mongodb://localhost:27017
/logs_db}
```

O uso de **variáveis de ambiente** permite que cada instância da aplicação utilize credenciais e configurações diferentes, garantindo maior segurança e adaptabilidade em diferentes contextos.

4.2 Configuração do Kafka

Para garantir a comunicação assíncrona entre os serviços da aplicação, utilizamos **Kafka**, um poderoso sistema de mensageria. As configurações incluem a definição dos **brokers Kafka e do grupo de consumidores**, possibilitando o processamento eficiente dos eventos.

```
spring.kafka.bootstrap-
servers=${SPRING_KAFKA_BOOTSTRAP_SERVERS:localhost:9092}
spring.kafka.consumer.group-id=${SPRING_KAFKA_CONSUMER_GROUP:monitor-group}
```

O uso de **brokers configuráveis** facilita a escalabilidade do sistema, permitindo que diferentes instâncias se comuniquem sem necessidade de reconfiguração manual.

4.3 Configuração do Serviço de E-mail

O envio de alertas críticos é um ponto essencial do sistema, e para isso utilizamos um serviço de e-mail SMTP configurável. Os parâmetros incluem **host, porta, credenciais e autenticação** para envio seguro das mensagens.

```
spring.mail.host=${SPRING_MAIL_HOST:smtp.gmail.com}
spring.mail.port=${SPRING_MAIL_PORT:587}
spring.mail.username=${SPRING_MAIL_USERNAME:meuemail@gmail.com}
spring.mail.password=${SPRING_MAIL_PASSWORD:minhasenha}
spring.mail.properties.mail.smtp.auth=${SPRING_MAIL_SMTP_AUTH:true}
spring.mail.properties.mail.smtp.starttls.enable=${SPRING_MAIL_SMTP_STARTTL
S_ENABLE:true}
```

O uso de variáveis de ambiente garante que credenciais sensíveis **não sejam expostas no código-fonte**, aumentando a segurança da aplicação.

4.4 Thresholds para Detecção de Anomalias

A aplicação possui **limiares configuráveis para análise de anomalias**, permitindo a detecção de **uso excessivo de CPU, memória e tempos de resposta elevados**. Esses parâmetros são ajustáveis conforme a necessidade do ambiente monitorado.

```
anomaly.cpu.critical.threshold=${ANOMALY_CPU_CRITICAL_THRESHOLD:90.0}
```

```
anomaly.memory.critical.threshold=${ANOMALY_MEMORY_CRITICAL_THRESHOLD:8000.0}
anomaly.responseTime.critical.threshold=${ANOMALY_RESPONSE_TIME_CRITICAL_THRESHOLD:3000}
```

Essa configuração permite que os limites sejam facilmente **ajustados em tempo de execução**, sem necessidade de recompilar ou reiniciar a aplicação.

5. Deploy com Docker

O sistema pode ser executado via docker-compose.yml.

```
services:
  postgres:
    image: postgres:latest
    environment:
      POSTGRES_DB: monitor_db
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
    ports:
      - "5432:5432"

  mongodb:
    image: mongo:latest
    ports:
      - "27017:27017"

  kafka:
    image: wurstmeister/kafka
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    ports:
      - "9092:9092"

  monitor-api:
    build: .
    ports:
```

```
      - "8080:8080"
    depends_on:
      - postgres
      - mongodb
      - kafka
```

5.1 Build e Execução

```
docker-compose up --build
```

---

6. Dependências no pom.xml

Principais dependências do projeto:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.kafka</groupId>
   <artifactId>spring-kafka</artifactId>
</dependency>
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
</dependency>
```

7. Conclusão

Este projeto foi desenvolvido com o objetivo de criar um **sistema robusto de monitoramento automatizado**, garantindo **alta disponibilidade e eficiência** na detecção de

anomalias em APIs externas e na própria aplicação. Através de uma arquitetura modular e distribuída, a aplicação permite a **coleta contínua de métricas**, **armazenamento estruturado de logs** e **notificação imediata de falhas** utilizando **Kafka e e-mail** para comunicação assíncrona.

A integração com **Docker** possibilita a escalabilidade do sistema, permitindo sua implantação em diferentes ambientes com facilidade. Além disso, durante o desenvolvimento, conceitos como **dockerização, mensageria com Kafka e RabbitMQ** foram explorados, proporcionando uma base sólida para futuras melhorias.

Para evoluir ainda mais o projeto, pretendo implementar um **sistema de autenticação** para validação das APIs externas monitoradas, garantindo maior **segurança e controle de acesso** aos dados armazenados. Além disso, planejo integrar o **Apache Spark**, que não apenas otimizará consultas em grandes volumes de dados, mas também poderá ser utilizado para **predição de tendências** com base em padrões identificados nas métricas coletadas.

Dessa forma, o projeto se consolida como uma solução **escalável, segura e preparada para futuras inovações**, permitindo análises preditivas e um monitoramento mais inteligente e automatizado.