RÉPUBLIQUE DU BÉNIN
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ D'ABOMEY-CALAVI

INSTITUT DE FORMATION ET DE RECHERCHE EN INFORMATIQUE

BP 526 Cotonou Tel : +229 21 14 19 88
http://www.ifri-uac.net Courriel : contact@ifri.uac.bj

MÉMOIRE

pour l'obtention du

Diplôme de Licence en Informatique

Option : Sécurité Informatique

Présenté par :
David **SEIDOU**

Monitoring des applications web: Cas des applications conteneurisées

Sous la supervision :
Ing. Eric **ATTOU**

Membres du jury :

Carlos AGOSSOU	Maitre Assitant - CAMES	UCAO-UUC	Président
Victor OYETOLA	Ingénieur	IFRI-UAC	Examinateur

Année Académique : **2023-2024**

Sommaire

Dédicace ii

Remerciements iii

Résumé iv

Table des figures vi

Liste des acronymes viii

Introduction 1

1 Revue de littérature 4

2 Architecture et outils 17

3 Résultats et discussions 37

Conclusion 43

Bibliographie 44

Bibliographie 45

bibliographie 45

Webographie 46

Table des matières 47

Dédicace

À mon grand frère ***Samuel AGANI****,*
À ma mère ***Rose BIAOU****,*
pour tous leurs sacrifices, leur amour, leur soutien et leurs prières tout au long de mes études.
À mes deux grandes sœurs ***Véronique AGANI*** *et* ***Pierrette AGANI****.*

Remerciements

- Mes sincères remerciements à Dieu le Père, qui m'a aidé et m'a donné la patience et le courage durant ces trois années d'études. J'adresse mes sincères remerciements à toutes les personnes qui, de près ou de loin, m'ont apporté leur aide pour que ce travail aboutisse.

  Je tiens à remercier particulièrement :

- Le corps administratif de l'*IFRI* pour tous les efforts qu'il fournit afin d'assurer à ses étudiants une formation de qualité ;

- Mon encadreur, *Eric ATTOU*, pour ses multiples orientations et surtout sa patience, sans laquelle ce travail n'aurait pas pu être mené à terme ;

- *Hored Sossou*, *Success Manasseh*, *Aaron KAKPO* ;

- Mes frères et sœurs, oncles et tantes, qui ont toujours été à l'écoute de mes sollicitations diverses. Tous ceux qui m'ont accompagné, soutenu et encouragé pour que ce travail puisse aboutir. Je pense à *Narcisse BONI*, *Pierre AGANI*, *Grâce AGANI*, ainsi qu'à tous mes amis, cousins et cousines...

Résumé

L'évolution technologique au cours de ces dernières années a conduit à la possibilité de construire des systèmes informatiques de plus en plus sophistiqués et de moins en moins encombrants. Pour tout employé d'une entreprise, d'une administration ou d'un établissement d'enseignement, la détection des anomalies applicatives devient indispensable pour améliorer la productivité.
De plus, de nombreuses applications sont déployées sur internet. Les personnes communiquent en ligne depuis n'importe où, ce qui rend indispensable l'utilisation d'un outil de suivi pour les entreprises qui réalisent leur activité via des applications. La surveillance de ces applications est donc primordiale.
Notre travail au cours de ce projet consiste à mettre en place une architecture de surveillance composée d'un ensemble d'outils permettant de monitorer efficacement les applications web. Prometheus reste l'outil privilégié par de grandes entreprises et des fournisseurs de services cloud. La mise en place de notre architecture tire parti de la puissance de Prometheus, capable de faire la découverte automatique de services. Nous avons pu enregistrer et prendre le contrôle d'un conteneur Docker nouvellement injecté dans notre réseau local.

Mots-clés : Prometheus, Cloud, Docker, disponibilité, surveillance.

Abstract

Technological developments in recent years have led to the possibility of build more and more sophisticated computer systems and less and less bulky. For any employee of a company, an administration, and an establishment teaching, the detection of application anomalies becomes essential and effective in productivity.
More and more, many applications are deployed on the internet. People communicate in online from anywhere, a very essential tool for monitoring companies that do business through applications. Monitoring these applications is therefore essential.
Our work during this development consists of setting up a monitoring architecture composed of a set of tools allowing us to effectively monitor applications on the web. Prometheus remains the tool used by large companies and cloud service providers. The implementation of our architecture results from the power of Prometheus, capable of automatic service discovery. We were able to register and take control of a newly injected docker container into our local network.

Key words: Prometheus, Cloud, docker, Monitoring, availability.

Table des figures

1.1 Status du service nagios core . . . 5
1.2 Installation de la haute disponibilité de Zabbix . . . 10
1.3 *Confirguration web* . . . 13
1.4 Service Compatible d'Intégration . . . 14
1.5 Fonctionnement de l'Agent Datadog . . . 14
1.6 Activation/Désactivation de la collecte des logs applicatifs . . . 15

2.1 Surveillance d'un cluster Kubernetes avec Prometheus . . . 19
2.2 Dashboard Grafana . . . 22
2.3 Fonctionnement de cAdvisor . . . 23
2.4 Architecture proposée . . . 24
2.5 Initialisation de Prometheus . . . 25
2.6 Dossier de configuration . . . 25
2.7 Fichier de configuration . . . 27
2.8 Extrait du fichier d'alerte de Prometheus . . . 28
2.9 Règles d'alerte pour la détection automatique des instances injoignables . . . 28
2.10 Règle d'alerte pour la détection automatique des instances injoignables . . . 29
2.11 Installation et configuration . . . 30
2.12 Données recueillies par Cadvisor . . . 30
2.13 Données des conteneurs . . . 31
2.14 Chargement de cible Cadvisor . . . 31
2.15 Définition du service NodeExporter . . . 31
2.16 Métriques exportées par NodeExporter . . . 32
2.17 Service AlertManager . . . 33
2.18 Connexion entre Prometheus et AlertManager . . . 33
2.19 Application créée sur Slack . . . 34
2.20 Fichier de configuration AlertManager . . . 34
2.21 Démarrage du service Grafana . . . 35
2.23 Datasource . . . 35
2.24 Tableau de bord . . . 36

3.1 Absence du Conteneur nginx-test . . . 37
3.2 Lancement du Conteneur nginx-test . . . 38
3.3 Détection du Conteneur nginx-test . . . 38
3.4 Dernier message Slack avant l'arrêt de Mongo . . . 38
3.5 Notification Slack . . . 39
3.6 prometheus.yml . . . 39

3.7 Script pour application python 40
3.8 Résultat test python 40
3.9 Résultat serveur 41
3.10 Valeur métrique 41
3.11 Utilisateurs connectés 41
3.12 Utilisateur connecté 41
3.13 Alerte Promeutheus 42

Liste des acronymes

API :

Outil qui permet la transmission et l'accès à des données de façon simplifiée et standardisée. *Glossaire :* API

Cloud :

Ensemble distribué de serveurs qui hébergent des logiciels et des infrastructures, et auquel on accède par l'internet *Glossaire :*

CNCF :

Projet de la Linux Foundation qui a été fondé en 2015 pour aider à faire progresser la technologie des conteneurs1 et rassembler les industries technologiques autour de son évolution. *Glossaire :* CNCF

Docker :

Plate-forme logicielle qui vous permet de concevoir, tester et déployer rapidement des applications à l'aide de conteneurs. *Glossaire :*

HTTP :

Ensemble de règles régissant le transfert de fichiers (texte, images, son, vidéo, et autres fichiers multimédias) sur le Web. *Glossaire :* HTTP

Kubernetes :

Plateforme Open Source d'orchestration des conteneurs qui automatise de nombreux processus manuels associés au déploiement, à la gestion et à la mise à l'échelle des applications conteneurisées. *Glossaire :*

Microservice :

Méthode développement logiciel utilisée pour concevoir une application comme un ensemble de services modulaires. *Glossaire :*

Slack :

Application de messagerie pour les entreprises qui connecte les personnes aux informations dont elles ont besoin. *Glossaire :*

Introduction Générale

L'informatique est devenue l'épine dorsale de l'entreprise, quel que soit son secteur d'activité. Le système d'information est au cœur de l'activité des différentes entités métiers et doit fonctionner pleinement et en permanence pour garantir l'efficacité de l'entreprise. À tous les niveaux, les réseaux, les terminaux utilisateurs, les serveurs d'applications et les données constituent autant de maillons sensibles dont la disponibilité et la qualité de service conditionnent le bon fonctionnement de l'entreprise.
Les problèmes liés à l'informatique doivent donc être réduits au minimum, car une indisponibilité du système d'information a des impacts très préjudiciables sur l'activité et la notoriété d'une entreprise. Dès lors, ce sujet nous permet de tenter de prévenir les problèmes et, le cas échéant, de garantir une remontée rapide de l'information ainsi qu'une intervention minimale en termes de durée. Nous proposons une solution qui permet de superviser et d'assurer le bon fonctionnement des différentes applications en production.
C'est dans ce contexte que s'inscrit l'objectif de notre projet, qui consiste à surveiller l'état des actifs applicatifs.

0.1 Problématique

Pour toute entreprise, la stabilité de son système informatique est considérée comme un élément essentiel et indispensable à son développement. Cette stabilité impose à l'entreprise des investissements matériels et humains importants, ce qui n'est pas toujours disponible ou possible pour toutes les sociétés.
Ainsi, l'utilisation d'une solution logicielle permettant d'optimiser la réactivité des équipes techniques de maintenance devient très recherchée.
Cependant, certaines entreprises informatiques se tournent vers des solutions libres, faciles à mettre en place et à maintenir.

0.2 Contexte et Justification

Pour répondre à la problématique énoncée précédemment et malgré l'existence de logiciels libres et propriétaires, nous avons opté pour le déploiement d'une architecture constituée de plusieurs composants libres, fonctionnant avec prometheus. Prometheus est un système open source de collecte de métrique.
Les autres composants, à savoir **cadvisor**, **nodeExporter**, **pushgateway**, seront des fournisseurs de métriques.
Il est particulièrement adapté à la surveillance des applications web en microservices.

0.3 Objectifs

L'objectif principal de notre projet est de mettre en place un système de surveillance non seulement pour des applications web, mais également pour la surveillance des serveurs, des processus systèmes et des scripts bash. Pour cela, nous nous sommes fixés les objectifs spécifiques suivants :

- déployer les composants fournisseurs de métriques système ;
- collecter (scraper) les métriques avec prometheus en récupérant les données clés dont nous avons besoin ;
- déployer des tableaux de bord grafana pour visualiser les données collectées ;
- configurer un système d'alerte qui enverra des mails et/ou des notifications slack.

0.4 Organisation du mémoire

Le présent document est structuré en trois chapitres. Dans le premier chapitre, nous aborderons quelques applications du marché utilisées dans la surveillance des applications, leurs spécificités, et nous proposerons une comparaison entre ces différents systèmes. Le deuxième chapitre présentera notre architecture de surveillance et, enfin, le dernier chapitre montrera les résultats obtenus après l'implémentation de nos solutions, suivis d'une discussion à ce sujet.

Chapitre 1

Revue de littérature

Introduction

La disponibilité des actifs d'entreprise fait partie intégrante du principe de la sécurité informatique. Pour commencer, il convient de définir les concepts derrière le terme générique surveillance applicative, puis de comprendre le fonctionnement de notre architecture de surveillance que nous souhaitons mettre en place afin de la situer par rapport aux autres méthodes de surveillance d'application. Une fois que nous saurons exactement de quoi nous parlons et d'où cela vient, il sera temps de commencer à définir les outils et le fonctionnement de notre méthode de surveillance d'application web. Pour finir, un aperçu des mesures de sécurité offertes sera présenté.

1.1 Généralités sur la surveillance des applications

1.1.1 Définition

Le monitoring ou la surveillance des applications consiste à surveiller de manière proactive les différentes facettes d'un environnement applicatif afin de résoudre les problèmes identifiés avant qu'ils ne deviennent majeurs. Il existe deux types de surveillance des applications web : la **surveillance interne** et la **surveillance externe**.

- **La surveillance interne** est un type de surveillance qui permet de vérifier le fonctionnement des serveurs et services au sein de l'intranet de l'entreprise.
- **La surveillance externe** est le deuxième type de surveillance effectuée sur les applications. Elle a pour objectif de vérifier le fonctionnement de l'application du point de vue des utilisateurs (notamment en ce qui concerne les services de l'application) et au niveau des serveurs.

1.2 Présentation des outils de surveillance sur le web

Les applications disponibles que nous allons présenter possèdent chacune des particularités qui déterminent nos choix dans la recherche d'outils de surveillance adaptés à nos besoins. Parmi ces ap-

plications, certaines sont gratuites et d'autres payantes. Dans le cadre de notre développement, voici quelques-unes adaptées aux applications web :

1.2.0.1 Nagios

- **Concept**
  Nagios est un système de supervision de services, dont la version stable actuelle est la 1.214. Il a été développé pour fonctionner sur une plateforme Linux ou éventuellement **UNIX**, avec un concept assez simple : les services de surveillance sont lancés par l'intermédiaire de contrôles des services et des stations que nous définissons.
  Pour installer et utiliser cet outil de monitoring, il est nécessaire d'avoir un serveur Web avec un module **PHP**. Le logiciel **Apache** est recommandé. En plus du noyau du programme, il faut également télécharger le plugin officiel de Nagios.
  Ces modules sont responsables de l'exploitation et de l'analyse des différentes demandes de monitorage des réseaux, des hôtes et des services, et sont ainsi essentiels pour la fonctionnalité du logiciel.

- **Fonctionnement**
  Nagios récupère les informations fournies par les services de surveillance et les analyse. Si le résultat de cette analyse révèle un problème, les services de surveillance peuvent envoyer des avertissements ou des alertes à l'administrateur réseau par e-mail, messages instantanés ou **sms**.

```
seidou@seidou-Latitude-7480:~$ service nagios status
● nagios.service - Nagios Core 4.4.13
     Loaded: loaded (/lib/systemd/system/nagios.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2023-09-16 10:58:15 WAT; 4h 49min ago
       Docs: https://www.nagios.org/documentation
   Main PID: 1410 (nagios)
      Tasks: 8 (limit: 8953)
     Memory: 18.6M
     CGroup: /system.slice/nagios.service
             ├─1410 /usr/local/nagios/bin/nagios -d /usr/local/nagios/etc/nagios.cfg
             ├─1411 /usr/local/nagios/bin/nagios --worker /usr/local/nagios/var/rw/nagios.qh
             ├─1412 /usr/local/nagios/bin/nagios --worker /usr/local/nagios/var/rw/nagios.qh
             ├─1413 /usr/local/nagios/bin/nagios --worker /usr/local/nagios/var/rw/nagios.qh
             ├─1414 /usr/local/nagios/bin/nagios --worker /usr/local/nagios/var/rw/nagios.qh
             ├─1415 /usr/local/nagios/bin/nagios --worker /usr/local/nagios/var/rw/nagios.qh
             ├─1416 /usr/local/nagios/bin/nagios --worker /usr/local/nagios/var/rw/nagios.qh
             └─1551 /usr/local/nagios/bin/nagios -d /usr/local/nagios/etc/nagios.cfg
```

FIGURE 1.1 : Status du service nagios core

Cette capture d'écran nous montre l'état de Nagios Core.
Nagios fonctionne en mode maître/esclave. On fournit au core une configuration personnalisée à travers le fichier **nagios.cfg**. Les esclaves sont déclarés dans ce fichier.
L'utilisateur dispose d'une interface web pour créer des tableaux de reporting. Nagios Core se concentre sur la surveillance des services et des applications en les écoutant avec le **systemd** ou par les processus sur la machine hôte. Il permet aussi de surveiller des hôtes distants via leurs adresses IP.

Plusieurs processus et ports sont ouverts sur la machine pour faciliter la collecte des métriques par les workers.

- **Quelques limites à son utilisation :**

1. Nagios est une solution de monitoring avec des choix fonctionnels. Le code source libre signifie que le logiciel peut être personnalisé et étendu à volonté. Plus de 1 000 plugins ont déjà été développés, ainsi que différentes interfaces frontales alternatives qui sont généralement fournies gratuitement.
2. Le principal inconvénient de Nagios est la configuration complexe des différents types de serveurs et objets. La configuration de Nagios pour répondre à des besoins de surveillance spécifiques peut prendre plus de temps et d'efforts.
3. Manque de surveillance en temps réel : Nagios n'offre pas de surveillance en temps réel par défaut, ce qui signifie que certains problèmes critiques peuvent ne pas être détectés immédiatement. Cela peut être particulièrement problématique pour les organisations qui exigent des niveaux élevés de disponibilité.
4. Difficulté de mise à l'échelle : Nagios peut être difficile à mettre à l'échelle pour des systèmes volumineux ou complexes, en particulier si vous n'êtes pas familier avec le logiciel. Cela peut rendre la gestion de la surveillance des systèmes en croissance ou en évolution plus compliquée.

Notre choix s'oriente davantage vers des applications web basées sur la technologie Docker, qui devient la norme pour les ingénieurs de production.

1.2.0.2 Zabbix

- **Quelques définitions des termes**

1. **Hôte :**
   Un équipement sur le réseau que vous souhaitez surveiller, avec une adresse IP/DNS.
2. **Groupe d'hôtes :**
   Un regroupement logique d'hôtes ; il peut contenir des hôtes et des modèles. Les hôtes et les modèles d'un groupe d'hôtes ne sont pas liés entre eux. Les groupes d'hôtes sont utilisés pour l'attribution de droits d'accès aux hôtes pour différents groupes d'utilisateurs.
3. **Pré-traitement des valeurs :**
   Une transformation des valeurs des métriques reçues avant de les enregistrer dans la base de données.
4. **Élément :**
   Une donnée particulière que vous souhaitez recevoir d'un hôte, une métrique de données.
5. **Déclencheurs :**
   Une expression logique qui définit un seuil de problème et qui est utilisée pour évaluer les données reçues des éléments. Lorsque les données dépassent le seuil, les déclencheurs

passent de 'Ok' à 'Problème'. Si les données reviennent en dessous du seuil, les déclencheurs retournent à 'Ok'.

6. **Événement :**
   Une occurrence unique de quelque chose qui mérite l'attention, comme un changement d'état de déclenchement, un enregistrement automatique d'agent, ou une découverte d'agent. Un événement est un marqueur prédéfini utilisé pour la corrélation d'événements, la gestion des autorisations, etc.
7. **Corrélation d'événements :**
   Une méthode de corrélation des problèmes avec leur résolution de manière flexible et précise. Par exemple, un problème signalé par un déclencheur peut être résolu par un autre déclencheur qui pourrait utiliser une méthode différente de collecte de données.
8. **Problème :**
   Un déclencheur en état "Problème".
9. **Mise à jour d'un problème :**
   Options de gestion des problèmes dans Zabbix, telles que l'ajout de commentaires, l'acquittement, la modification de la gravité, ou la fermeture manuelle.
10. **Action :**
    Un moyen prédéfini de réagir à un événement.
11. **Escalade :**
    Un scénario personnalisé pour l'exécution d'opérations dans une action, comme une séquence d'envoi de notifications ou l'exécution de commandes distantes.
12. **Média :**
    Un canal utilisé pour la livraison des notifications.
13. **Notification :**
    Un message envoyé à un utilisateur via un canal choisi (média) pour signaler un événement.
14. **Commande à distance :**
    Une commande prédéfinie automatiquement exécutée sur un hôte surveillé sous certaines conditions.
15. **Modèle :**
    Un ensemble d'entités (éléments, déclencheurs, graphiques, etc.) prêtes à être appliquées à un ou plusieurs hôtes. Les modèles permettent un déploiement rapide des tâches de surveillance et facilitent les modifications collectives.
16. **Scénario Web :**
    Une ou plusieurs requêtes HTTP pour vérifier la disponibilité d'un site web.
17. **Interface Web :**
    L'interface web fournie avec Zabbix.
18. **Tableau de bord :**
    Une section personnalisable de l'interface web affichant des résumés et des visualisations d'informations importantes sous forme de widgets.
19. **Widget :**
    Une unité visuelle affichant des informations (résumés, graphiques, horloges, etc.) utilisées dans un tableau de bord.

20. **API Zabbix :**
    L'API Zabbix permet d'utiliser le protocole JSON RPC pour créer, mettre à jour, et récupérer des objets Zabbix (comme des hôtes, des éléments, etc.) ou effectuer d'autres tâches personnalisées.
21. **Serveur Zabbix :**
    Le processus central de Zabbix qui effectue la supervision, interagit avec les proxys et agents Zabbix, calcule les déclencheurs, et envoie des notifications. Il est considéré comme le référentiel central de données.
22. **Proxy Zabbix :**
    Un processus qui collecte des données pour le compte du serveur Zabbix, allégeant la charge de traitement du serveur.
23. **Agent Zabbix :**
    Un processus déployé sur les cibles de surveillance pour superviser activement les ressources et applications locales.
24. **Agent Zabbix 2 :**
    Une nouvelle génération de l'agent Zabbix permettant l'utilisation de plugins personnalisés pour la surveillance des ressources et applications locales.

- **Qu'est-ce que Zabbix ?**
  Zabbix est une solution de supervision professionnelle open source. Il permet de surveiller un grand nombre de paramètres d'un réseau, ainsi que la santé des serveurs, des machines virtuelles, des applications, des services, des bases de données, des sites web, du cloud, et bien plus encore.

  Zabbix dispose d'un mécanisme de notification flexible permettant de configurer des alertes par e-mail pour tout événement critique, facilitant une réponse rapide aux problèmes. Il offre des fonctionnalités de reporting et de visualisation de données, ce qui le rend idéal pour la gestion de la capacité.

  Zabbix supporte à la fois l'interrogation (polling) et la réception (trapping) des données. Tous les rapports, statistiques, et configurations de Zabbix sont accessibles via une interface web, permettant d'évaluer l'état du réseau et des serveurs depuis n'importe où.

  Correctement configuré, Zabbix peut jouer un rôle clé dans la supervision des infrastructures IT, qu'il s'agisse de petites organisations avec peu de serveurs ou de grandes entreprises avec des infrastructures complexes.

- **Processus et composantes**

  Serveur :
  Le serveur Zabbix est le processus central du logiciel Zabbix.

  Le serveur effectue l'interrogation et la récupération des données, il calcule les déclencheurs et envoie des notifications aux utilisateurs. C'est la composante centrale à laquelle les agents et les proxys Zabbix rapportent des données sur la disponibilité et l'intégrité des systèmes. Le

serveur peut également vérifier à distance les services réseaux (comme les serveurs Web et les serveurs de messagerie) à l'aide de simples vérifications de service.

Le serveur est le référentiel central où sont stockées toutes les données de configuration, statistiques et opérationnelles. C'est également l'entité de Zabbix qui alerte activement les administrateurs lorsque des problèmes surviennent dans l'un des systèmes surveillés.

Le fonctionnement d'un serveur Zabbix de base est divisé en trois composantes distinctes : le serveur Zabbix, l'interface Web et le stockage en base de données. Toutes les informations de configuration de Zabbix sont stockées dans la base de données, avec laquelle le serveur et l'interface Web interagissent.
Par exemple, lorsque vous créez un nouvel élément à l'aide de l'interface Web (ou de l'API), il est ajouté à la table des éléments dans la base de données.
Ensuite, environ une fois par minute, le serveur Zabbix interroge la table des éléments pour obtenir une liste des éléments actifs, qui sont ensuite stockés dans un cache sur le serveur Zabbix.
C'est pourquoi cela peut prendre jusqu'à deux minutes pour que toute modification apportée à l'interface Zabbix apparaisse dans la dernière section de données.

- **Processus et composants**

  Serveur :
  Le serveur Zabbix est le processus central du logiciel Zabbix. Le serveur effectue l'interrogation et la récupération des données, calcule les déclencheurs et envoie des notifications aux utilisateurs. C'est le composant central auquel les agents et les proxys Zabbix rapportent des données de disponibilité et d'intégrité des systèmes. Le serveur peut lui-même vérifier à distance les services réseaux (tels que les serveurs Web et les serveurs de messagerie) à l'aide de simples vérifications de service.

  Le serveur est le référentiel central dans lequel toutes les données de configuration, statistiques et opérationnelles sont stockées, et c'est l'entité de Zabbix qui alerte activement les administrateurs lorsque des problèmes surviennent dans l'un des systèmes surveillés.

  Le fonctionnement d'un serveur Zabbix de base est divisé en trois composants distincts : le serveur Zabbix, l'interface Web et le stockage en base de données. Toutes les informations de configuration de Zabbix sont stockées dans la base de données, avec laquelle le serveur et l'interface Web interagissent.
  Par exemple, lorsque vous créez un nouvel élément à l'aide de l'interface Web (ou de l'API), il est ajouté à la table des éléments de la base de données. Ensuite, environ une fois par minute, le serveur Zabbix interroge la table des éléments pour obtenir une liste des éléments actifs qui sont ensuite stockés dans un cache sur le serveur Zabbix. C'est pourquoi cela peut prendre jusqu'à deux minutes pour que toute modification apportée à l'interface Zabbix apparaisse dans la section des dernières données.

- **Configuration HA :**

  La haute disponibilité (HA) est généralement requise dans les infrastructures critiques qui ne peuvent se permettre pratiquement aucun temps d'arrêt. Ainsi, pour tout service susceptible d'être en erreur, une option de basculement doit être en place pour prendre le relais en cas d'échec du service actuel.

  Zabbix propose une solution haute disponibilité native facile à configurer et ne nécessitant aucune expertise préalable en HA. Le HA natif de Zabbix peut être utile pour une couche supplémentaire de protection contre les pannes logicielles/matérielles du serveur Zabbix ou pour réduire le temps d'arrêt lors de la maintenance.

  En mode haute disponibilité Zabbix, plusieurs serveurs Zabbix sont exécutés en tant que nœuds dans un cluster. Pendant qu'un serveur Zabbix du cluster est actif, les autres sont en veille, prêts à prendre le relais si nécessaire.



FIGURE 1.2 : Installation de la haute disponibilité de Zabbix

- **Proxy :**

  Le proxy Zabbix est un processus qui peut collecter des données de surveillance à partir d'un ou plusieurs équipements surveillés et envoyer les informations au serveur Zabbix, en travaillant essentiellement pour le compte du serveur. Toutes les données collectées sont mises en tampon localement puis transférées au serveur Zabbix auquel appartient le proxy.

  Le déploiement d'un proxy est facultatif, mais peut s'avérer très utile pour répartir la charge d'un seul serveur Zabbix. Si seuls les proxys collectent des données, le traitement sur le serveur devient moins gourmand en CPU et en E/S disque.

  Un proxy Zabbix est la solution idéale pour la surveillance centralisée des sites distants, des succursales et des réseaux sans administrateurs locaux. Le proxy Zabbix nécessite une base de données séparée.

- **Passerelle Java :**

  Configuration et exécution de la passerelle Java avec Ubuntu. Pour démarrer automatiquement la passerelle Java Zabbix au démarrage de l'OS :

```
$systemctl enable zabbix-java-gateway
```

- **Sender :**

  Le sender Zabbix est un utilitaire en ligne de commande qui peut être utilisé pour envoyer des données de performance au serveur Zabbix pour traitement.

  L'utilitaire est généralement utilisé dans les scripts utilisateur à exécution longue pour l'envoi périodique de données de disponibilité et de performance. Pour l'envoi des résultats directement au serveur ou proxy Zabbix, un élément trapper doit être configuré.

- **Get :**

  Le get Zabbix est un utilitaire en ligne de commande qui peut être utilisé pour communiquer avec l'agent Zabbix et récupérer les informations requises depuis l'agent. L'utilitaire est généralement utilisé pour le dépannage des agents Zabbix.

  Exécution du get Zabbix :
  Un exemple d'exécution du get Zabbix sous UNIX pour obtenir la valeur de la charge du processeur à partir de l'agent :

```
$shell> ./zabbix_get -s 127.0.0.1 -p 10050 -k system.cpu.load[all,avg1]
```

- **Web Service :**

  Le service Web Zabbix est un processus utilisé pour communiquer avec des services Web externes. Actuellement, le service Web Zabbix est utilisé pour générer et envoyer des rapports planifiés, avec des projets pour ajouter des fonctionnalités supplémentaires à l'avenir.

  Le serveur Zabbix se connecte au service Web via HTTP(S). Le service Web Zabbix nécessite que Google Chrome soit installé sur le même hôte ; sur certaines distributions, le service peut également fonctionner avec Chromium (voir problèmes connus).

- **Installation avec du Docker**

  1. **Créez un réseau dédié aux conteneurs de composantes Zabbix :**

```
$docker network create --subnet 172.20.0.0/16 --ip-range 172.20.240.0/20 zabbix-net
```

2. **Démarrez une instance de serveur MySQL vide**

```
$docker run --name mysql-server -t \
        -e MYSQL_DATABASE="zabbix" \
        -e MYSQL_USER="zabbix" \
        -e MYSQL_PASSWORD="zabbix_pwd" \
        -e MYSQL_ROOT_PASSWORD="root_pwd" \
        --network=zabbix-net \
        --restart unless-stopped \
        -d mysql:8.0 \
        --character-set-server=utf8 --collation-server=utf8_bin \
        --default-authentication-plugin=mysql_native_password
```

3. **Démarrez l'instance de la passerelle Java Zabbix**

```
$ docker run --name zabbix-java-gateway -t \
        --network=zabbix-net \
        --restart unless-stopped \
        -d zabbix/zabbix-java-gateway:alpine-6.0-latest
```

4. **Démarrez l'instance de serveur Zabbix et liez l'instance à l'instance de serveur MySQL créée**

```
$ docker run --name zabbix-server-mysql -t \
        -e DB_SERVER_HOST="mysql-server" \
        -e MYSQL_DATABASE="zabbix" \
        -e MYSQL_USER="zabbix" \
        -e MYSQL_PASSWORD="zabbix_pwd" \
        -e MYSQL_ROOT_PASSWORD="root_pwd" \
        -e ZBX_JAVAGATEWAY="zabbix-java-gateway" \
        --network=zabbix-net \
        -p 10051:10051 \
        --restart unless-stopped \
        -d zabbix/zabbix-server-mysql:alpine-6.0-latest
```

5. **Démarrez l'instance de la passerelle Java Zabbix**
6. **Démarrez l'interface Web Zabbix et liez l'instance avec le serveur MySQL créé et les instances de serveur Zabbix**

```
$ docker run --name zabbix-web-nginx-mysql -t \
        -e ZBX_SERVER_HOST="zabbix-server-mysql" \
        -e DB_SERVER_HOST="mysql-server" \
        -e MYSQL_DATABASE="zabbix" \
        -e MYSQL_USER="zabbix" \
        -e MYSQL_PASSWORD="zabbix_pwd" \
        -e MYSQL_ROOT_PASSWORD="root_pwd" \
        --network=zabbix-net \
        -p 80:8080 \
        --restart unless-stopped \
        -d zabbix/zabbix-w oheb-nginx-mysql:alpine-6.0-latest
```

- **Supervision d'application web :**

Pour activer la supervision Web, vous devez définir des scénarios Web. Un scénario Web consiste en une ou plusieurs requêtes HTTP ou "étapes". Les étapes sont exécutées périodiquement par le serveur Zabbix dans un ordre prédéfini. Si un hôte est supervisé par un proxy, les étapes sont exécutées par le proxy.

Les scénarios Web sont associés aux hôtes/modèles de la même manière que les éléments, les déclencheurs, etc. Cela signifie que les scénarios Web peuvent également être créés au niveau du modèle, puis appliqués à plusieurs hôtes en même temps.

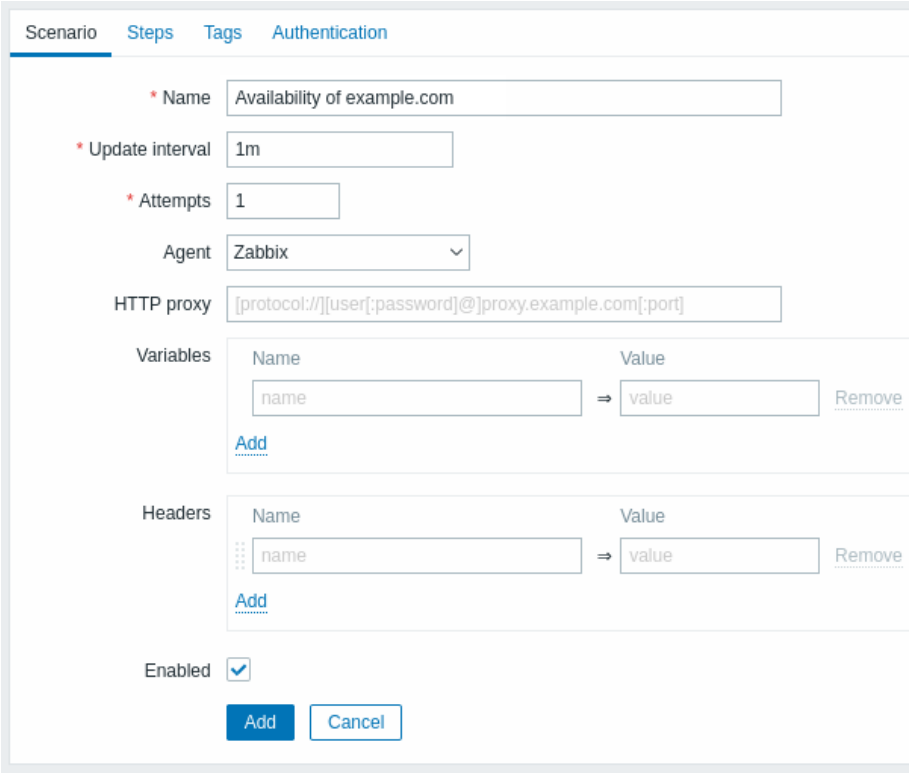


FIGURE 1.3 : *Confirguration web*

1.2.0.3 Datadog

- **Présentation :**
  Datadog permet de superviser les logs et les métriques tout en proposant des services d'Application Performance Management (APM) et de corrélation de métriques. De plus, Datadog s'adapte aux infrastructures traditionnelles ainsi qu'aux infrastructures serverless.
  Datadog est distribué en tant que service intégralement managé par l'entreprise éponyme. Ainsi, vous disposez d'une application hautement disponible, clé en main.

- **Intégration :**
  L'interconnexion de Datadog à vos environnements se fait via un système d'intégrations. Leur mise en place est simple et, grâce à leur polyvalence, les intégrations permettent de remonter les logs, les métriques, les traces, etc., avec la même solution.
  Datadog est fourni avec des intégrations pour de nombreuses applications et services très populaires (AWS, Azure, GCP, Nginx, Kubernetes, etc.).
  Comme son nom l'indique, c'est ce qui permet de relier Datadog à vos environnements, afin qu'il collecte l'ensemble des données (métriques, logs, traces) utiles. Les intégrations peuvent se faire à l'aide d'un agent ou bien d'un crawler.

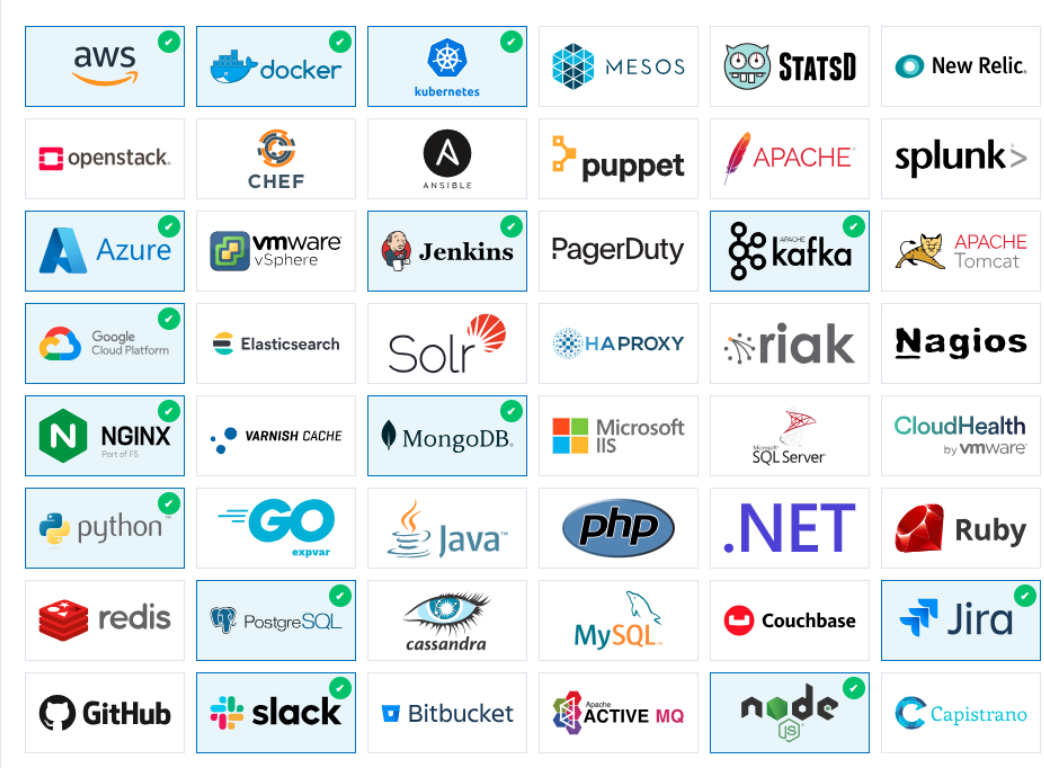


FIGURE 1.4 : Service Compatible d'Intégration

- **Agent Datadog :**

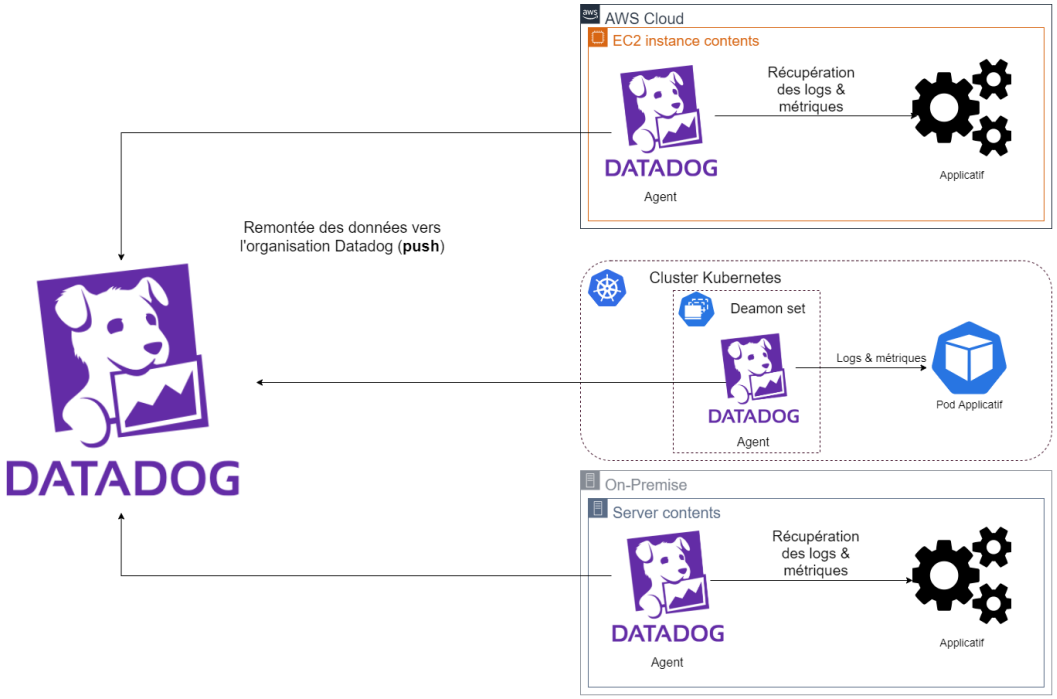


FIGURE 1.5 : Fonctionnement de l'Agent Datadog

L'Agent Datadog est un binaire à faire tourner sur les hôtes auxquels vous avez accès. Ce dernier est à utilité multiple, il permet à la fois de remonter les logs, les métriques, mais aussi les traces. Chacune de ces fonctionnalités peut s'activer dans la configuration YAML associée à l'agent (**/etc/datadog-agent/datadog.yaml**). L'intérêt de cet agent réside donc dans sa po-

lyvalence face aux besoins.

```
[...]
##################################

## Log collection Configuration ##

##################################

# @param logs_enabled - boolean - optional - default: false

# Enable Datadog Agent log collection by setting logs_enabled to true.

logs_enabled: false

# @param logs_config - custom object - optional

# Enter specific configurations for your Log collection.

# Uncomment this parameter and the one below to enable them.

# See https://docs.datadoghq.com/agent/logs/

logs_config:

  # @param container_collect_all - boolean - optional - default: false

  # Enable container log collection for all the containers (see ac_exclude to filter out containers)



  container_collect_all: false

[...]
```

FIGURE 1.6 : Activation/Désactivation de la collecte des logs applicatifs

- **Monitoring d'application web :**
  Pour le monitoring, Datadog propose plusieurs méthodes pour récupérer des métriques. Si l'on est dans un cloud public, il est possible d'ajouter une intégration directe, comme avec AWS, qui va récupérer vos métriques depuis CloudWatch ainsi que diverses autres sources de votre choix (Lambda, RDS, ElasticCache, etc.).

  Pour les instances, que ce soit on-premise ou dans le cloud, il est possible d'installer l'agent Datadog directement sur les machines. Cela permet de récupérer plus de métriques, de configurer l'agent pour s'intégrer avec vos outils, tels que Nginx, PHP-FPM, et d'avoir des health checks sur vos services.

  Beaucoup de librairies existent pour installer et configurer l'agent avec de l'IaC. Maintenant, nos conteneurs, on sait tous qu'ils nous tiennent à cœur ! Selon l'utilisation de vos conteneurs, l'installation va être différente.

  Il est possible d'avoir un conteneur sidecar qui va envoyer les métriques, par exemple pour Fargate ou ECS. Si vous avez des conteneurs Docker sur des instances, vous pouvez monitorer

cela grâce à l'agent et un peu de configuration.

Si vous utilisez un orchestrateur de conteneurs, des intégrations existent. Par exemple, pour Kubernetes, plusieurs possibilités s'offrent à vous : utiliser un DaemonSet pour mettre l'agent sur chaque nœud et renvoyer des métriques, ou bien utiliser un Cluster Agent. Je recommande grandement ce dernier car il offre de nouvelles fonctionnalités, telles que l'auto-scaling en utilisant des métriques provenant de Datadog.

1.3 Comparaison et choix d'implémentation

En résumé, cette partie, consacrée plus particulièrement à la veille technologique concernant les outils de surveillance utilisés sur le web, nous permet de déduire que, peu importe l'outil, ils ne sont pas tous adaptés à nos besoins. Nagios, tout comme Zabbix, sont plus orientés vers le service système. Avec une configuration complexe, ils présentent une insuffisance dans le contrôle de l'état de santé d'un conteneur Docker dynamiquement injecté. Toutefois, Datadog le fait, mais demeure un outil entièrement payant, donc il n'est pas accessible à tous.

Chapitre 2

Architecture et outils

Introduction

Dans ce chapitre de notre mémoire, nous essayerons de mettre en place l'architecture de surveillance composée de*prometheus* et **Cadvisor**. L'objectif ici est de presenter comment assurer la disponibilté des applications web conteneurisés.

2.1 Présentation des outils

2.1.1 Prometheus

- **Définition**

  Prometheus est un outil libre de surveillance et d'alerte des systèmes, initialement conçu par SoundCloud. Depuis sa création en 2012, de nombreuses entreprises et organisations ont adopté Prometheus, et le projet compte une communauté de développeurs et d'utilisateurs très active. Il s'agit désormais d'un projet open source autonome, maintenu indépendamment de toute entreprise. Pour souligner cela et clarifier la structure de gouvernance du projet, Prometheus a rejoint la **Cloud Native Computing Foundation** en 2016 en tant que deuxième projet hébergé, après **Kubernetes**.
  Prometheus collecte et stocke ses métriques sous forme de données de séries chronologiques, c'est-à-dire que les informations sur les métriques sont enregistrées avec l'horodatage auquel elles ont été collectées, accompagnées de paires clé-valeur facultatives appelées **étiquettes**.

- **Les Fonctions**

  Prometheus occupe les fonctions suivantes :

  - Un modèle de données multidimensionnel et de séries chronologiques identifiées par le nom de la métrique et des paires clé-valeur.
  - PromQL, un langage de requête flexible pour exploiter cette dimensionnalité.

  - Pas de dépendance au stockage distribué ; les nœuds de serveur unique sont autonomes.
  - La collecte de séries chronologiques via un modèle pull sur **HTTP**.
  - La diffusion de séries chronologiques est prise en charge via une passerelle intermédiaire.
  - Les cibles sont enregistrées grâce à la découverte de services ou à la configuration statique.
  - Plusieurs modes de prise en charge des graphiques et des tableaux de bord.

- **Les métriques**
  Les métriques sont des mesures numériques. Le terme série chronologique fait référence à l'enregistrement des changements au fil du temps. Ce que les utilisateurs souhaitent mesurer diffère d'une application à l'autre. Pour un serveur web, cela peut inclure le temps de requête pour une base de données, le nombre de connexions actives ou le nombre de requêtes en cours, etc.

  Les métriques jouent un rôle important dans la compréhension des raisons pour lesquelles votre application fonctionne d'une certaine manière. Supposons que vous exécutez une application web et que vous découvrez qu'elle est lente. Pour comprendre ce qui se passe avec votre application, vous aurez besoin de certaines informations. Par exemple, lorsque le nombre de requêtes est élevé, l'application peut devenir lente. Si vous disposez de la métrique relative au nombre de demandes, vous pouvez déterminer la cause du problème et augmenter le nombre de serveurs pour gérer la charge.

- **Composantes**
  L'écosystème Prometheus comprend plusieurs composantes, dont beaucoup sont facultatives :

  - Le serveur principal Prometheus, qui récupère et stocke les données de séries chronologiques.
  - Les bibliothèques clientes pour instrumenter le code de l'application.
  - Une passerelle push pour prendre en charge les emplois de courte durée.
  - Des exportateurs spécialisés pour des services comme **HAProxy**, **StatsD**, **Graphite**, etc.
  - Un gestionnaire d'alerte.

  La plupart des composantes de Prometheus sont écrites en Go, ce qui les rend faciles à créer et à déployer sous forme de binaires statiques.

- **Architecture**
  Illustration de l'écosystème de **Prometheus** :

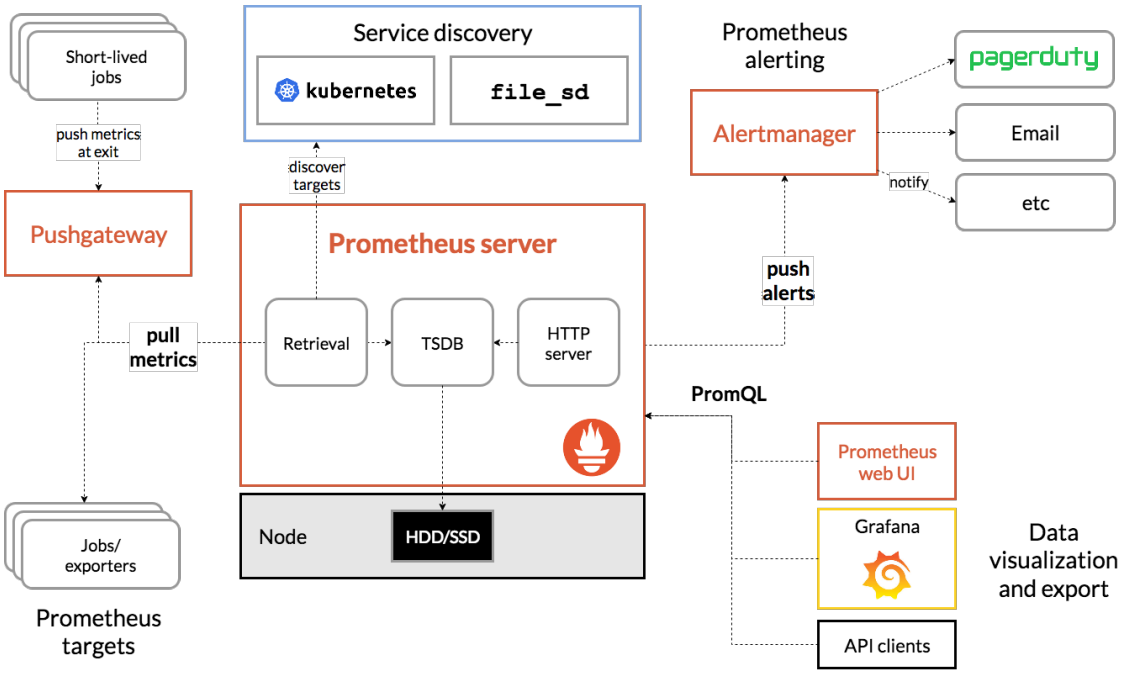


FIGURE 2.1 : Surveillance d'un cluster Kubernetes avec Prometheus

Prometheus récupère les métriques des tâches instrumentées directement via une passerelle push intermédiaire pour les tâches de courte durée. Il stocke localement tous les échantillons récupérés et exécute des règles sur ces données pour agréger et enregistrer de nouvelles séries chronologiques à partir de données existantes ou générer des alertes. Grafana ou d'autres consommateurs d'API peuvent être utilisés pour visualiser les données collectées.

- **Cas d'utilisation**
  Prometheus fonctionne bien pour enregistrer n'importe quelle série chronologique purement numérique. Il s'adapte à la fois à la surveillance centrée sur la machine et à la surveillance d'architectures hautement dynamiques orientées services. Dans un monde de microservices, sa prise en charge de la collecte et de l'interrogation de données multidimensionnelles est un atout majeur.

- **Alerte**
  L'alerte avec Prometheus est séparée en deux parties. Les règles d'alerte des serveurs Prometheus envoient des alertes à un Alertmanager. Alertmanager gère ensuite ces alertes, y compris la mise sous silence, l'inhibition, l'agrégation et l'envoi de notifications via des méthodes telles que le courrier électronique, les systèmes de notification d'astreinte et les plateformes de messagerie instantanée.

  Les principales étapes de configuration des alertes et des notifications sont :

  – Installer et configurer Alertmanager.
  – Configurer Prometheus pour communiquer avec Alertmanager.
  – Créer des règles d'alerte dans Prometheus.

  Prometheus est conçu pour la fiabilité, afin d'être le système auquel vous vous adressez en cas de panne, vous permettant de diagnostiquer rapidement les problèmes. Chaque serveur Prometheus est autonome et ne dépend pas du stockage réseau ou d'autres services distants.

Vous pouvez compter sur lui lorsque d'autres parties de votre infrastructure sont en panne, sans avoir besoin de configurer une infrastructure étendue pour l'utiliser.

2.1.2 PushGateway

Pushgateway est une fonctionnalité de Prometheus qui permet aux tâches éphémères et par lots d'exposer leurs métriques à l'application. Souvent, ces types de tâches n'ont pas un cycle de vie suffisamment long pour que Prometheus puisse les récupérer et extraire la grande quantité de mesures nécessaires à une surveillance efficace du réseau.

Au lieu de cela, vous pouvez transmettre vos métriques vers une passerelle Pushgateway qui, à son tour, expose ces métriques à Prometheus. Considérez Pushgateway comme un intermédiaire dans la collecte de métriques. Vous pouvez transférer vos tâches par lots et éphémères de courte durée sur Pushgateway, en l'utilisant comme cache de métriques qui permet ensuite à Prometheus de les supprimer, même lorsque les sujets surveillés n'existent plus.

Ce processus diffère de l'application de métriques en mode pull de Prometheus. Avec Pushgateway, votre série de métriques est continuellement exposée à Prometheus, à moins que quelqu'un ne la supprime manuellement à l'aide de l'API de Pushgateway. Lorsque Prometheus utilise les métriques du modèle pull, il n'a accès qu'aux instances qui existent encore. Cette application dispose de bibliothèques clientes pour Go, Java, Scala, Python et Ruby, ce qui la rend utile à un large public.

Bien que cet outil soit généralement associé à Prometheus et qu'il présente des avantages en termes de persistance des instances qui envoient des requêtes, il est à noter qu'il existe des pièges dans son utilisation :

- Lors de la surveillance de plusieurs instances via un seul Pushgateway, celui-ci devient à la fois un point de défaillance unique et un goulot d'étranglement potentiel.
- Vous perdez la surveillance automatique de l'état de l'instance de Prometheus via la métrique `up` (générée à chaque scrape).
- Le Pushgateway n'oublie jamais les séries qui lui sont transmises et les exposera pour toujours à Prometheus, à moins que ces séries ne soient supprimées manuellement via l'API du Pushgateway.

Ce dernier point est particulièrement pertinent lorsque plusieurs instances d'une tâche différencient leurs métriques dans Pushgateway via une étiquette d'instance ou similaire. Les métriques d'une instance resteront alors dans Pushgateway même si l'instance d'origine est renommée ou supprimée. En effet, le cycle de vie de Pushgateway en tant que cache de métriques est fondamentalement distinct du cycle de vie des processus qui y transmettent les métriques.

Comparez cela à la surveillance habituelle de style pull de Prometheus : lorsqu'une instance disparaît (intentionnellement ou non), ses métriques disparaissent automatiquement avec elle. Lorsque vous utilisez Pushgateway, ce n'est pas le cas, et vous devrez désormais supprimer manuellement toutes les métriques obsolètes ou automatiser vous-même cette synchronisation du cycle de vie.

2.1.3 AlertManager

Alertmanager gère les alertes envoyées par les applications clientes, telles que le serveur Prometheus. Il se charge de les dédupliquer, de les regrouper et de les acheminer vers l'intégration de récepteurs

appropriés, tels que la messagerie électronique, Slack ou d'autres outils de collaboration en entreprise. Il s'occupe également de la mise sous silence et de l'inhibition des alertes.

- **Regroupement :**
  Le regroupement catégorise les alertes de nature similaire en une seule notification. En tant qu'utilisateur, vous pouvez configurer Alertmanager pour regrouper les alertes par cluster et par nom d'alerte, afin d'envoyer une seule notification compacte. Cela vous permet de recevoir une seule page tout en voyant exactement quelles instances de service ont été affectées. Le regroupement des alertes, la synchronisation des notifications groupées et les récepteurs de ces notifications sont configurés par une arborescence de routage dans le fichier de configuration.

- **Inhibition :**
  Lorsqu'une alerte se déclenche, elle peut informer que l'ensemble d'un cluster n'est pas accessible. Alertmanager peut être configuré pour désactiver toutes les autres alertes concernant ce cluster si cette alerte particulière se déclenche. Cela empêche des notifications de centaines ou de milliers d'alertes de se déclencher, qui ne sont pas liées au problème réel.

- **Mode Silencieux :**
  Les silences sont un moyen de désactiver les alertes pendant une durée donnée. Un silence est configuré en fonction de matchers. Les alertes entrantes sont vérifiées pour savoir si elles correspondent à tous les comparateurs d'égalité ou d'expression régulière d'un silence actif. Si tel est le cas, aucune notification ne sera envoyée pour cette alerte. Les silences sont configurés dans l'interface web d'Alertmanager.

- **Haute disponibilité :**
  Alertmanager prend en charge la configuration d'un cluster pour une haute disponibilité. Cela peut être configuré à l'aide des indicateurs **–cluster-***. Cette configuration s'effectue dans le fichier de configuration `alertmanager.yml`.

2.1.4 Grafana

Grafana est un outil open source qui offre la possibilité d'interroger, de visualiser, d'alerter et d'explorer les métriques que nous mettons en place. Nous avons également la possibilité de naviguer sur nos journaux et traces, où qu'ils soient stockés.

Ce qui suit montre un exemple de tableau de bord Grafana qui interroge Prometheus pour obtenir des données :

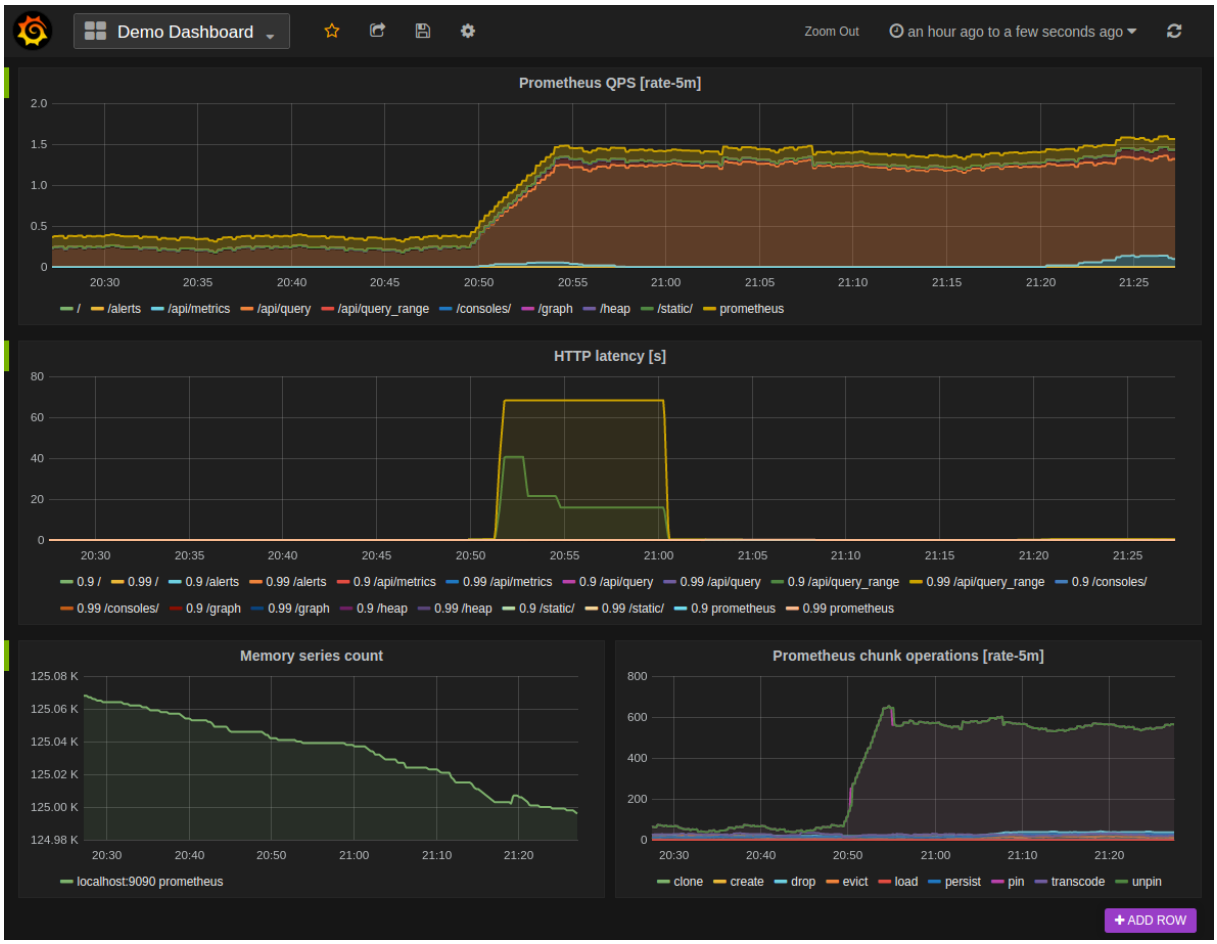


FIGURE 2.2 : Dashboard Grafana

Nous évoquerons dans la partie implémentation son installation et sa configuration.

2.1.5 Cadvisor

cAdvisor analyse et expose les données d'utilisation des ressources et de performances des conteneurs en cours d'exécution. cAdvisor expose les métriques Prometheus prêtes à l'emploi.

cAdvisor est livré sous forme de déploiement autonome. Il prend en charge nativement les conteneurs **Docker** et d'autres environnements de conteneurs, tels que **Kubernetes**, prêts à l'emploi. Il peut collecter des données analytiques sur n'importe quel conteneur en cours d'exécution, y compris l'utilisation du processeur, la mémoire, l'utilisation du système de fichiers, et les interfaces réseau physiques et virtuelles, tout en s'exécutant sur une seule image Docker.

Vous trouverez ci-dessous un exemple d'exécution de cAdvisor avec un conteneur Docker :

```
1 $docker run --volume=/:/rootfs:ro \ --volume=/var/run:/var/run:ro \ --volume=/sys:/sys:ro \ --volume=/va
```

- **cAdvisor et Prometheus**
  cAdvisor fournit des statistiques de conteneur à Prometheus à l'aide de points de terminaison HTTP standardisés. Ensuite, dans Prometheus, nous pouvons récupérer et capturer des données en temps réel en créant une ou plusieurs tâches pour traiter les données métriques de cAdvisor à partir des points de terminaison HTTP.

Prometheus est automatiquement configuré pour obtenir les données du démon cAdvisor en cours d'exécution. À partir de là, il est facile de surveiller les données des conteneurs dans Prometheus. Les utilisateurs peuvent voir un résumé de toutes les informations par hôte, ainsi qu'une vue globale de tous les hôtes. Vous pouvez surveiller la consommation de mémoire, l'utilisation du processeur et les limites des ressources, ainsi qu'afficher des interfaces spécifiques aux applications pour la visualisation des ressources.

- **Les fonctions :**

  1. Détecte automatiquement les conteneurs au sein de son nœud et commence à collecter des données.
  2. Plusieurs méthodes de mise en œuvre, notamment l'exécution en tant qu'application autonome au niveau du système d'exploitation, en tant que conteneur Docker ou un DaemonSet dans Kubernetes.
  3. Peut exporter des données vers des plugins de stockage tels que Prometheus, Elasticsearch et InfluxDB pour une analyse et un traitement plus approfondis.
  4. Possède une interface utilisateur Web intégrée qui expose les métriques en temps réel à partir des données collectées.
  5. API REST puissante pour interroger les métriques des conteneurs directement depuis cAdvisor.

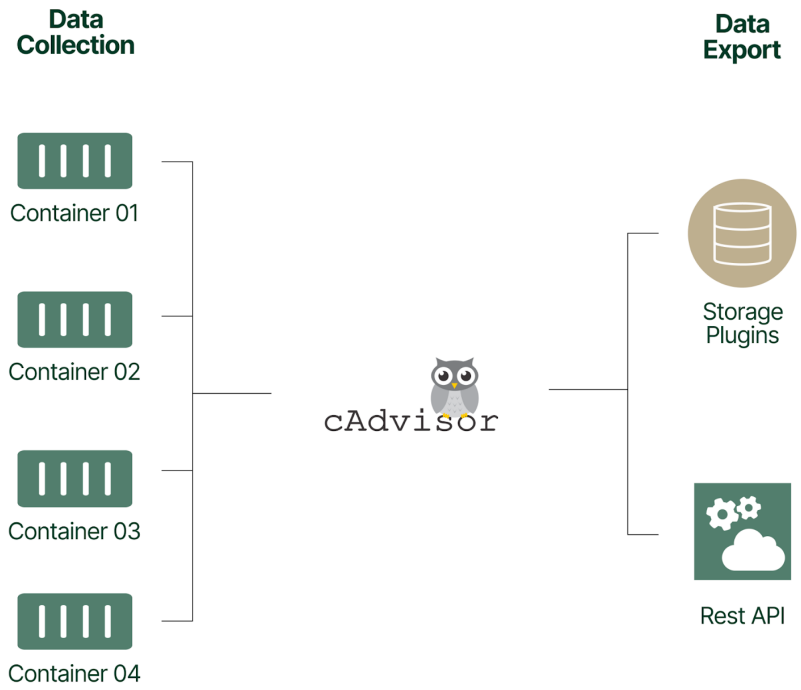


FIGURE 2.3 : Fonctionnement de cAdvisor

2.2 Présentation de l'architecture

Après avoir présenté les outils qui entreront en jeu dans cette partie, voici l'architecture que nous souhaitons mettre en place.

2.2.1 Architecture

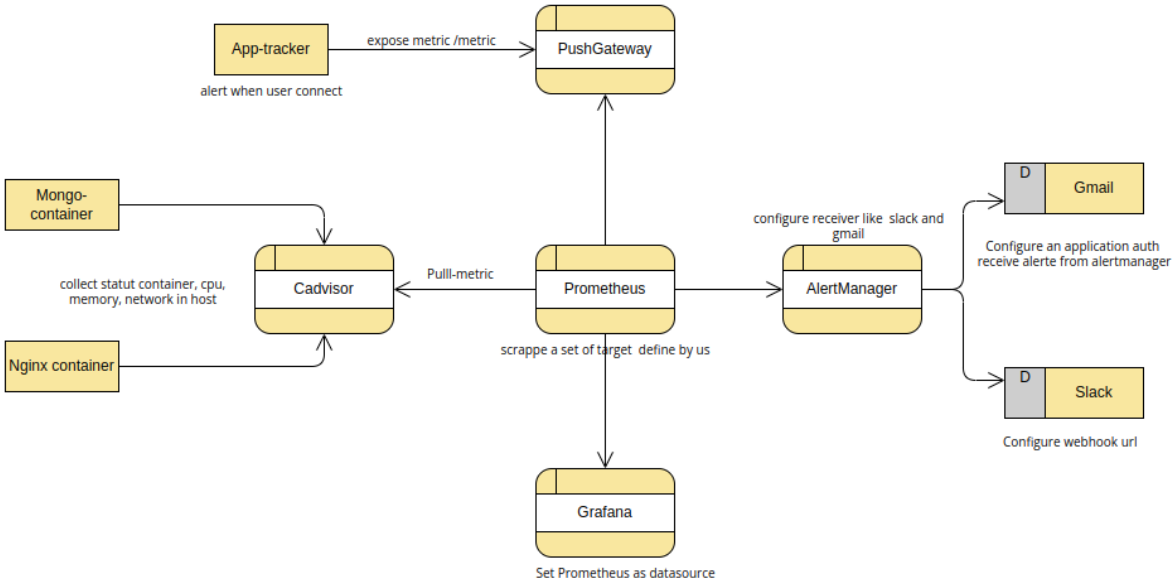


FIGURE 2.4 : Architecture proposée

L'architecture ci-dessus nous montre :

- **Cadvisor** collecte les données du serveur local, à savoir : les processus systèmes, la mémoire, le CPU, le flux réseau. Il dispose d'un accès en lecture au système de fichiers. Il est important, par mesure de sécurité, que le système de fichiers reste immuable au risque d'être exposé à des attaques. Il collecte notamment, en ce qui nous concerne, l'ensemble des données du démon **Docker**. Ainsi, il récupère les données des deux conteneurs ***(Mongo et Nginx)*** qui s'exécutent sur la machine hôte. L'ensemble de ces données sera exposé sur le point de terminaison /metrics via HTTP.

- Une fois les données exposées par Cadvisor, **Prometheus** scrappe l'endpoint via son processus de pulling. Nous utilisons le processus d'autodiscovery dans ce cas, puisque nous n'avons pas de contrainte particulière avec l'ensemble des conteneurs en termes de persistance ou de modèle de données exposé.

- Prometheus expose à son tour ces données, ce qui permet à **Grafana** de les interroger afin de construire des dashboards personnalisés selon nos besoins.

- **AlertManager** intercepte l'ensemble des alertes qui lui sont envoyées par Prometheus et les transfère vers des routes, qui sont des canaux de distribution de messages, tels que Slack et un

compte Gmail. Ce dernier disposera d'un mot de passe applicatif pour éviter d'être exposé à l'usurpation d'identité.

- **App-tracer** est une application Python que nous avons écrite. Elle utilise le client Prometheus pour envoyer les informations des utilisateurs connectés au serveur. Elle expose les données sur /metrics.

2.2.2 Mise en place de l'architecture

2.2.2.1 Installation de Prometheus et configuration

1. Nous disposons essentiellement de trois fichiers pour Prometheus : un fichier qui consiste en l'installation de Prometheus en lui passant des arguments, et deux autres pour la configuration.

```yaml
prometheus:
  image: prom/prometheus:v2.18.1
  container_name: prometheus
  volumes:
    - ./prometheus:/etc/prometheus
    - prometheus_data:/prometheus
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'
    - '--storage.tsdb.path=/prometheus'
    - '--web.console.libraries=/etc/prometheus/console_libraries'
    - '--web.console.templates=/etc/prometheus/consoles'
    - '--storage.tsdb.retention.time=200h'
    - '--web.enable-lifecycle'
  restart: unless-stopped
  ports:
    -   9090:9090
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

FIGURE 2.5 : Initialisation de Prometheus

Nous avons le service Prometheus qui est composé de :

- **image :** le nom de l'image Docker pour Prometheus que nous utilisons. Nous avons choisi la version v2.18.1.
- **volumes :** Nous disposons de deux volumes : un volume de type Docker, $prometheus_data$, qui permet de sauvegarder la base de données locale de Prometheus, et $./prometheus$, pour stocker l'ensemble des fichiers de configuration tels que prometheus.yml et alert.rules.

```
seidou@seidou-Latitude-7480:~/Memoire/foration_aws_monitoring/prometheus$ ls
alert.rules  prometheus.yml
```

FIGURE 2.6 : Dossier de configuration

- **command :** ici, nous passons des arguments aux conteneurs avant leur démarrage. Il s'agit de définir le stockage pour la base de données, le fichier de configuration de Prometheus, d'activer le cycle de vie et la rétention des données.

- **restart : unless-stopped** : nous spécifions à Docker de redémarrer Prometheus uniquement lorsqu'il s'est arrêté.
- **ports :** le port 9090 est celui sur lequel nous exposons Prometheus à l'extérieur de la machine hôte.
- **networks :** monitor-net est le réseau que nous avons créé pour stocker l'ensemble de la stack de monitoring. Pour créer le réseau, nous utilisons la commande : docker network create monitor-net.
- **labels :** il s'agit d'une balise pour identifier la stack de monitoring.

2. **Fichier de configuration des cibles :** Dans le fichier **.yml**, nous configurons les cibles ou les endpoints que Prometheus doit scrapper. Voici notre configuration avant la création de l'application App-tracker :

```yaml
global:
  scrape_interval:     15s
  evaluation_interval: 15s
  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
      monitor: 'docker-host-alpha'
# Load and evaluate rules in this file every 'evaluation_interval' seconds.
rule_files:
  - "alert.rules"
# A scrape configuration containing exactly one endpoint to scrape.
scrape_configs:
  - job_name: 'nodeexporter'
    scrape_interval: 5s
    static_configs:
      - targets: ['nodeexporter:9100']

  - job_name: 'cadvisor'
    scrape_interval: 5s
    static_configs:
      - targets: ['cadvisor:8080']

  - job_name: 'prometheus'
    scrape_interval: 10s
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'pushgateway'
    scrape_interval: 10s
    honor_labels: true
    static_configs:
      - targets: ['pushgateway:9091']

alerting:
  alertmanagers:
  - scheme: http
    static_configs:
    - targets:
      - 'alertmanager:9093'
```

FIGURE 2.7 : Fichier de configuration

Nous avons défini la fréquence de scrapping du serveur Prometheus à 15 secondes et ajouté un fichier pour définir les règles d'alerte. Ensuite, nous fournissons les endpoints à scrapper. L'AlertManager étant parfaitement intégré avec Prometheus, nous l'inscrivons comme canal de transmission des alertes.

3. **Fichier de définition des règles d'alerte :** Nous présentons ici la méthode manuelle de détection de l'état des conteneurs en écrivant, dans le fichier de règles, une expression spécifique au langage Prometheus, qui renverra une alerte lorsqu'une condition correspond aux données collectées par Prometheus.

   Ci-dessous, le fichier alert.rules :

```
- name: containers
  rules:
  - alert: DB Storage down on host
    expr: absent(container_memory_usage_bytes{name="mongo"})
    for: 20s
    labels:
      severity: critical
    annotations:
      summary: "DB mongo down"
      description: "Container {{ $labels.name }} is down for more than 30 seconds."


  - alert: Nginx down on host
    expr: absent(container_memory_usage_bytes{name="nginx"})
    for: 20s
    labels:
      severity: critical
    annotations:
      summary: "Nginx down"
      description: "Container {{ $labels.name }} is down for more than 30 seconds."
```

FIGURE 2.8 : Extrait du fichier d'alerte de Prometheus

Pushgateway est une problématique, car son utilisation nous empêche d'utiliser le fonctionnement par défaut de Prometheus, basé sur l'auto-découverte. Cela signifie que nous n'avons pas à préciser les conteneurs dans ce fichier. Dans la dernière partie du développement, nous ferons une démonstration des deux approches.

```
- name: AllInstances
  rules:
  - alert: An Instance of Stack monitoring down
    # Condition for alerting
    expr: up == 0
    for: 1m
    # Annotation - additional informational labels to store more information
    annotations:
      title: 'Instance {{ $labels.instance }} down'
      description: '{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 1 minute.'
    # Labels - additional labels to be attached to the alert
    labels:
      severity: 'critical'
```

FIGURE 2.9 : Règles d'alerte pour la détection automatique des instances injoignables

Ci-dessus, l'ensemble des instances actives ou inactives depuis le tableau de bord.

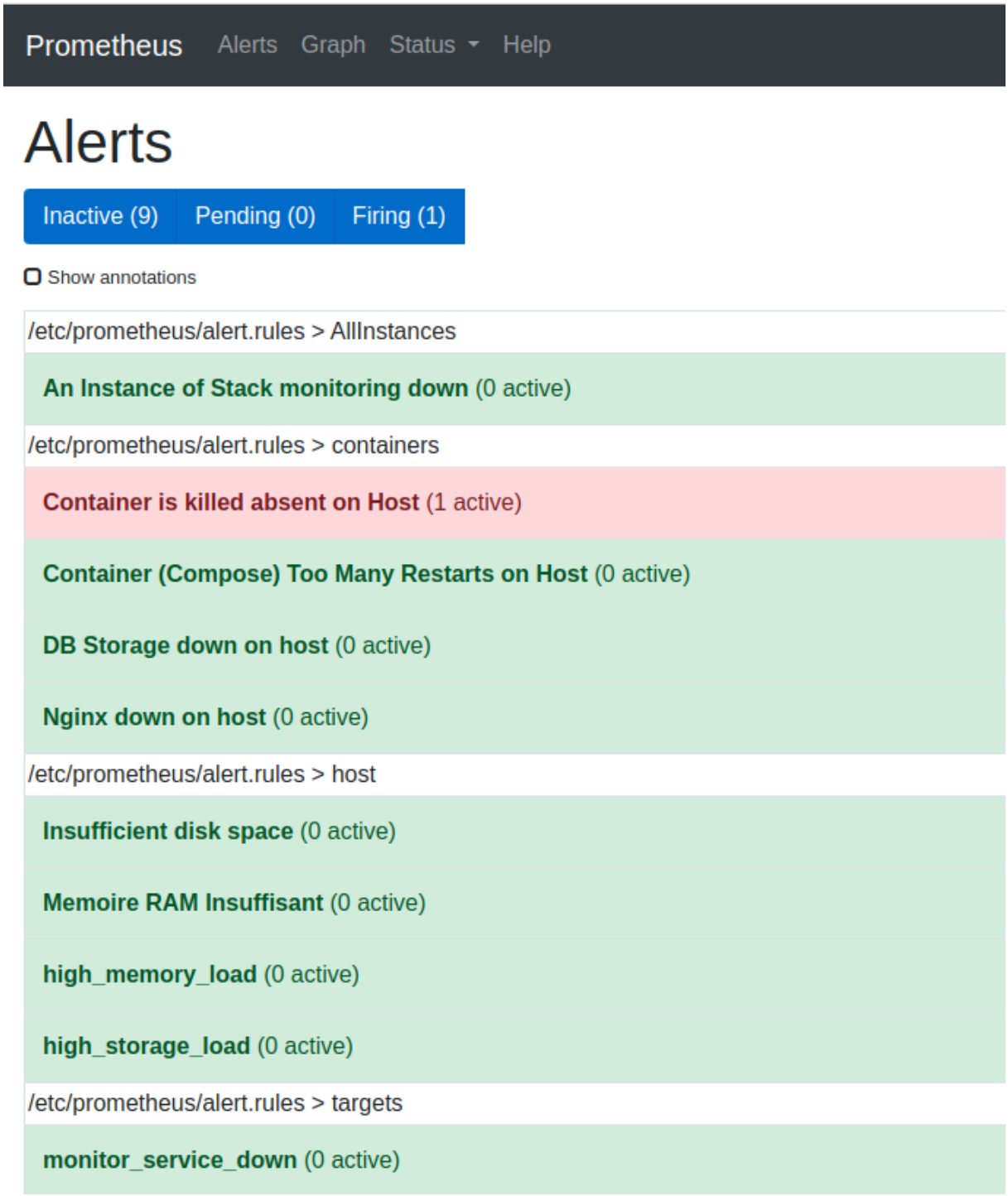


FIGURE 2.10 : Règle d'alerte pour la détection automatique des instances injoignables

0 active signifie que l'instance indexée est en direct, tandis que lorsqu'on a **1 active**, cela signifie que l'instance n'est pas joignable.

Passons à présent à l'installation et à la configuration des cibles fournissant les données.

2.2.2.2 Installation et Configuration de Cadvisor

L'installation consiste en une suite de déclarations d'instructions Docker comme suit :

```yaml
cadvisor:
  image: gcr.io/google-containers/cadvisor:v0.36.0
  container_name: cadvisor
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker:/var/lib/docker:ro
    #- /cgroup:/cgroup:ro #doesn't work on MacOS only for Linux
  restart: unless-stopped
  ports:
    - 8080:8080
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

FIGURE 2.11 : Installation et configuration

Dans cette capture, nous utilisons les instructions du fichier Compose pour définir le service **Cadvisor**.

Afin de recueillir les informations système, cAdvisor utilise le mode lecture seule *read-only* sur l'ensemble du système de fichiers. Nous redémarrons le service uniquement s'il s'arrête. Il dispose d'une interface exposée sur le port 8080 de la machine locale. Ci-dessous, les données des conteneurs recueillies sur la machine locale :

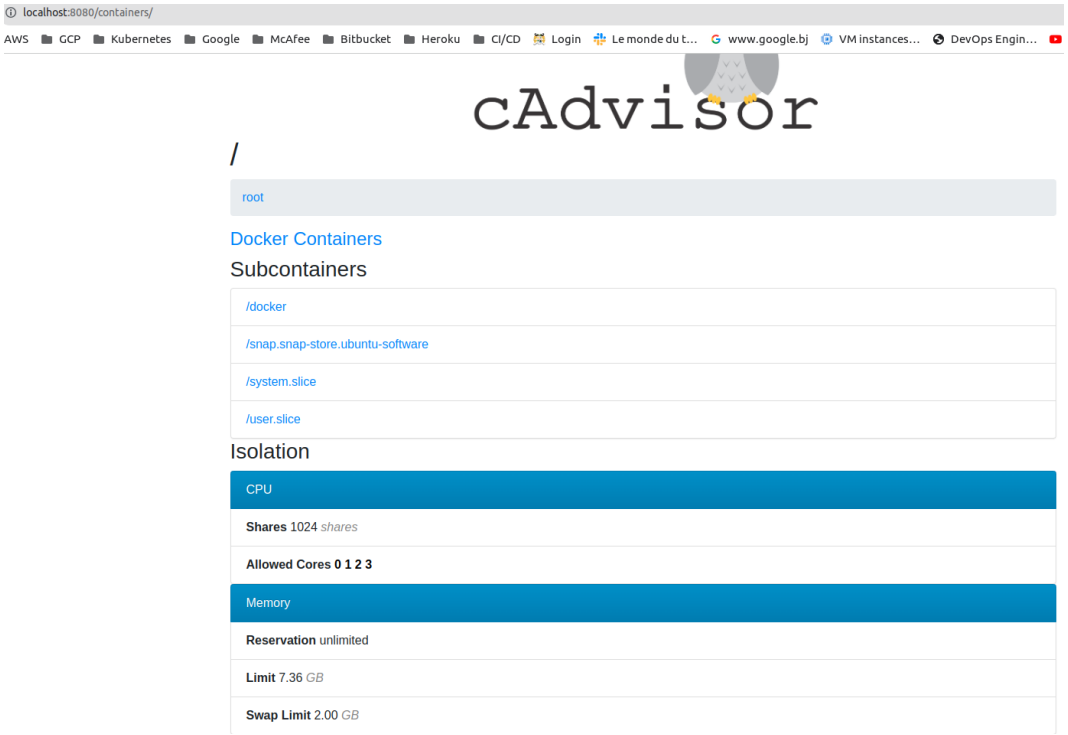


FIGURE 2.12 : Données recueillies par Cadvisor

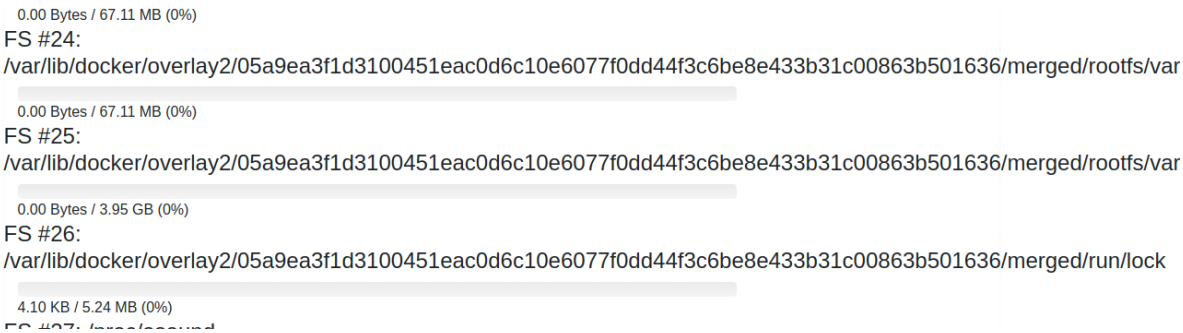


FIGURE 2.13 : Données des conteneurs

Prometheus envoie des requêtes chaque 5s pour receuillir les données exposées sur **cadvisor :8080/container**

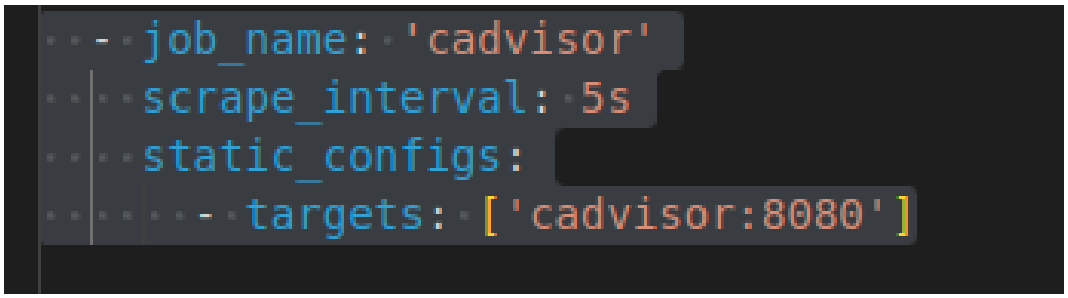


FIGURE 2.14 : Chargement de cible Cadvisor

2.2.2.3 Installation et Configuration de Nodeexporter

- **Installation :** Nous avons effectué son installation en utilisant le mode Docker, comme pour l'ensemble des composants de notre architecture.

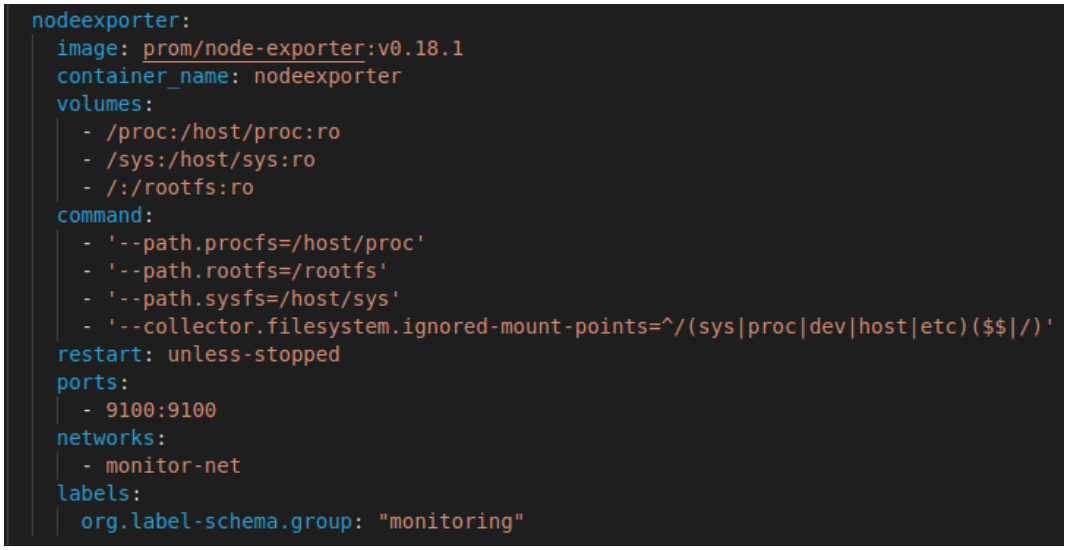


FIGURE 2.15 : Définition du service NodeExporter

1. **Image :** Nous utilisons la version **v0.18.1** de l'image officielle de Nodeexporter.
2. **Container name :** Le nom du conteneur.
3. **Volumes :** Tout comme Cadvisor décrit précédemment, Nodeexporter utilise également l'ensemble du système de fichiers en mode lecture afin de lire les données (mémoire,

disque, CPU, processus, fichiers) sur la machine cible.

4. **Command :** Ce sont des arguments spécifiques que nous passons avant le lancement du conteneur pour définir la source des données que le conteneur exposera.
5. **Restart : unless-stopped** Nous spécifions à Docker de ne redémarrer Prometheus que lorsqu'il s'est arrêté.
6. **Ports :** Le port 9100 est celui sur lequel nous exposons Prometheus à l'extérieur de la machine hôte.
7. **Networks :** 'monitor-net' est le réseau que nous avons créé pour stocker l'ensemble de la stack de monitoring. Pour créer le réseau, nous utilisons : "'bash docker network create monitor-net "'
8. **Labels :** Un label pour identifier la stack de monitoring.

- **Exposition des métriques :** L'ensemble des métriques est exposé sur la route conventionnelle */metrics*. Ci-dessous, la sortie de notre exporter :



FIGURE 2.16 : Métriques exportées par NodeExporter

2.2.2.4 Installation et configuration de l'AlertManager

L'alerte nous permet de recevoir les états des instances définies dans **alert.rules**.

- **Installation :**
  Avec les systèmes conventionnels de déploiement, il est possible d'installer le binaire Go, puis de l'exécuter comme suit : $./alertmanager --config.file = alertmanager.yml$

  Cependant, nous utilisons Docker pour notre stack, ce qui nous conduit à déployer l'AlertManager comme les services précédents :

```yaml
alertmanager:
  image: prom/alertmanager:v0.20.0
  container_name: alertmanager
  volumes:
    - ./alertmanager:/etc/alertmanager
  command:
    - '--config.file=/etc/alertmanager/config.yml'
    - '--storage.path=/alertmanager'
  restart: unless-stopped
  ports:
    - 9093:9093
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

FIGURE 2.17 : Service AlertManager

  Dans cette capture d'écran, le dossier de configuration local d'AlertManager est monté à l'intérieur du conteneur, lui permettant d'avoir accès au fichier de configuration **config.yml** au démarrage du conteneur.

- **Configuration :**
  La première étape après l'installation est de notifier Prometheus sur le système d'alerte à interroger. Nous avons ajouté ces lignes dans le fichier **prometheus.yml** comme suit :

```yaml
alerting:
  alertmanagers:
  - scheme: http
    static_configs:
    - targets:
      - 'alertmanager:9093'
```

FIGURE 2.18 : Connexion entre Prometheus et AlertManager

Avant de présenter le fichier de configuration d'AlertManager, créons le channel Slack.

- **Création du channel Slack :**
  AlertManager envoie les notifications à Slack via une URL de webhook. Le webhook est créé à partir d'une application Slack qui envoie les alertes dans un ou plusieurs canaux.

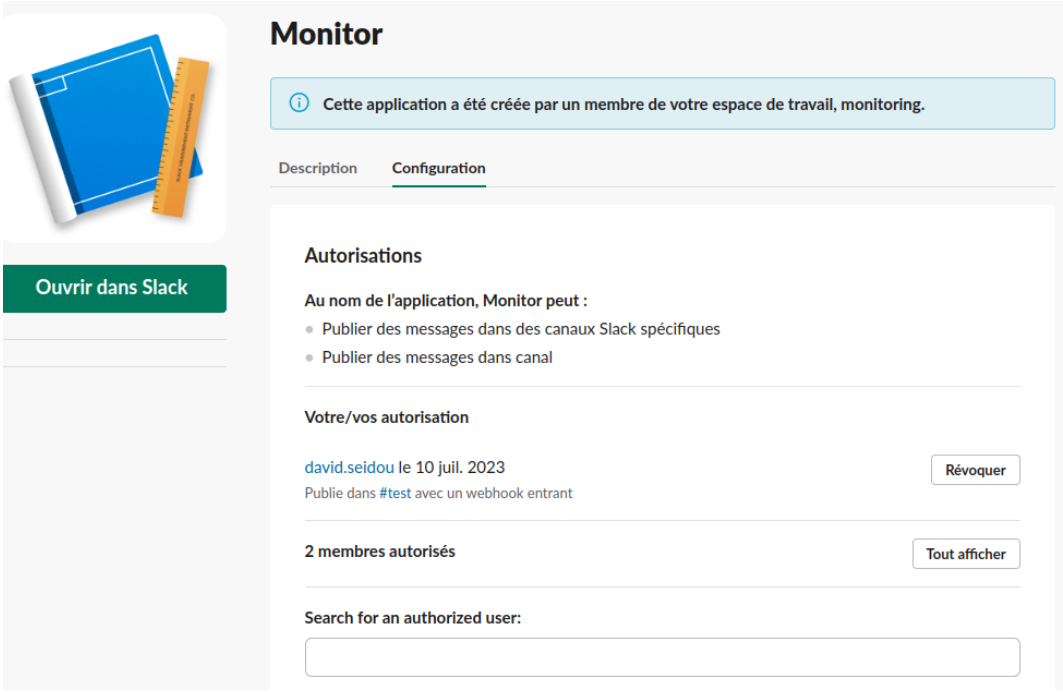


FIGURE 2.19 : Application créée sur Slack

Dans notre cas, l'application **Monitor** envoie les messages d'alerte vers le canal de test.

```
route:
    receiver: 'slack'

receivers:
  - name: 'slack'
    slack_configs:
    - send_resolved: true
      #text: "{{ .CommonAnnotations.description }}"
      username: 'david.seidou'
      channel: '#test'
      api_url: 'https://hooks.slack.com/services/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'
      icon_url: https://upload.wikimedia.org/wikipedia/commons/3/38/Prometheus_software_logo.svg
      title: |-
        [{{ .Status | toUpper }}{{ if eq .Status "firing" }}:{{ .Alerts.Firing | len }}{{ end }}] {{ .CommonLabels.alertname }} for {{ .CommonLabels.job }}
        {{- if gt (len .CommonLabels) (len .GroupLabels) -}}
          {{" "}}(
          {{- with .CommonLabels.Remove .GroupLabels.Names }}
            {{- range $index, $label := .SortedPairs -}}
              {{ if $index }}, {{ end }}
              {{- $label.Name }}="{{ $label.Value -}}"
            {{- end }}
          {{- end -}}
          )
        {{- end }}
      text: >-
        {{ range .Alerts -}}
        *Alert:* {{ .Annotations.title }}{{ if .Labels.severity }} - `{{ .Labels.severity }}`{{ end }}

        *Description:* {{ .Annotations.description }}

        *Details:*
          {{ range .Labels.SortedPairs }} • *{{ .Name }}:* `{{ .Value }}`
          {{ end }}
        {{ end }}
```

FIGURE 2.20 : Fichier de configuration AlertManager

2.2.2.5 Installation et configuration de Grafana

- **Installation :** Le service Grafana est déployé dans le même stack que l'ensemble de nos composants. Il est utilisé ici pour visualiser l'ensemble des données collectées sur les systèmes et les conteneurs. Ci-dessous, son déploiement :

```yaml
grafana:
  image: grafana/grafana:6.7.3
  container_name: grafana
  volumes:
    - grafana_data:/var/lib/grafana
    - ./grafana/provisioning:/etc/grafana/provisioning
    - ./grafana/config:/etc/grafana
  environment:
    - GF_SECURITY_ADMIN_USER=${ADMIN_USER:-admin}
    - GF_SECURITY_ADMIN_PASSWORD=${ADMIN_PASSWORD:-admin}
    - GF_USERS_ALLOW_SIGN_UP=false
  restart: unless-stopped
  ports:
    - 3000:3000
  networks:
    - monitor-net
  labels:
    org.label-schema.group: "monitoring"
```

FIGURE 2.21 : Démarrage du service Grafana

Le volume Docker **grafana**$_{d}atasertdebasededonnelocaleGrafana.Ilhbergelesinformationsdesutilisateurs$

Ajout de la datasource : Une fois le conteneur lancé, Grafana utilise par défaut la datasource que nous avons provisionnée.

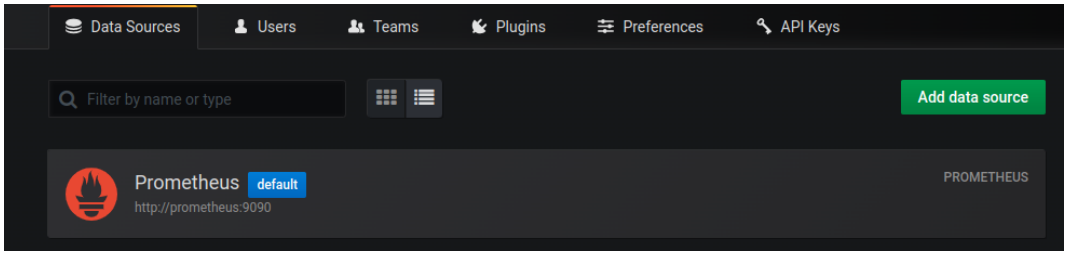


FIGURE 2.23 : Datasource

Ajout des tableaux de bord : Tout comme la datasource, nous avons récupéré trois tableaux de bord pour faciliter la visualisation. Ils sont montés via le hostPath sur **/etc/grafana/provisioning/dashboards**. L'exemple ci-dessous montre le tableau de bord qui présente l'ensemble des conteneurs, à savoir : App-tracker, Mongo, Nginx. Il détaille leur utilisation de la mémoire, du CPU et du disque, les temps de réponse, ainsi que d'autres paramètres liés à l'observabilité. Il montre également les spécifications du serveur.



FIGURE 2.24 : Tableau de bord

Chapitre 3

Résultats et discussions

Introduction

Dans ce chapitre, nous présentons les résultats des simulations effectuées pour tester le fonctionnement de notre stack de monitoring d'application. Dans un premier temps, nous démontrerons la notion d'autodécouverte de Prometheus en démarrant un conteneur Nginx via la ligne de commande, puis nous vérifierons si Prometheus récupère bien les informations et les transmet à Grafana. Ensuite, nous arrêterons manuellement les deux conteneurs Mongo et Nginx, et vérifierons si nous recevons des notifications sur le canal Slack **test**. Enfin, nous entamerons une discussion.

3.1 Test de découverte automatique d'application

La fonctionnalité essentielle est de s'assurer que lorsqu'un conteneur est démarré sur nos serveurs, nous pouvons voir son état et obtenir ses spécifications.

Au départ, nous n'observons pas de conteneur nommé **nginx-test**. Nous l'avons nommé ainsi puisqu'il existe déjà un conteneur nommé Nginx.

Assurons-nous qu'aucun conteneur nommé **nginx-test** n'existe.

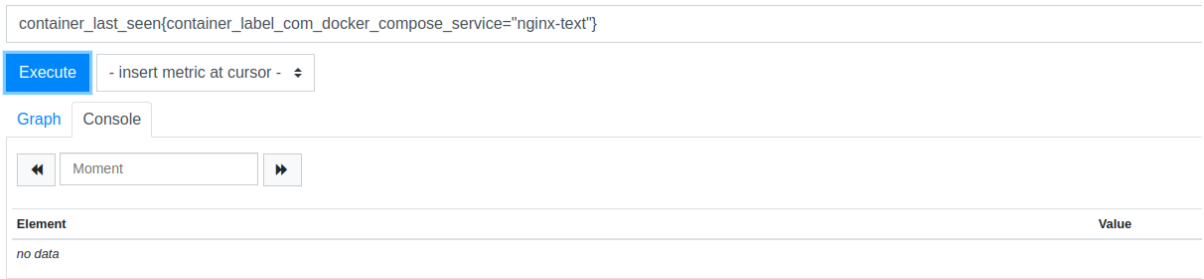


FIGURE 3.1 : Absence du Conteneur nginx-test

Comme vous pouvez le voir, nous n'avons aucune donnée collectée.

Démarrons maintenant notre conteneur :

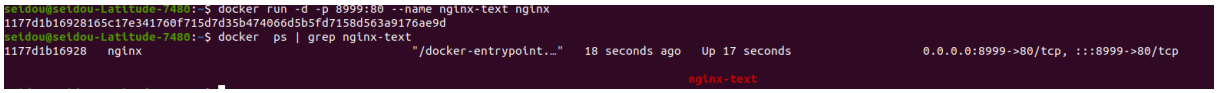


FIGURE 3.2 : Lancement du Conteneur nginx-test

Vérifions si le conteneur a été détecté par Prometheus via les données récupérées de Cadvisor.



FIGURE 3.3 : Détection du Conteneur nginx-test

Nous constatons que cette étape a bien été vérifiée. Passons à présent au second volet : recevoir une notification sur Slack en cas d'indisponibilité.

3.2 Arrêt instinctif des conteneurs

Nous allons vérifier si l'indisponibilité des applications, comme **mongo**, est bien notifiée.

Avant l'arrêt, nous avons :

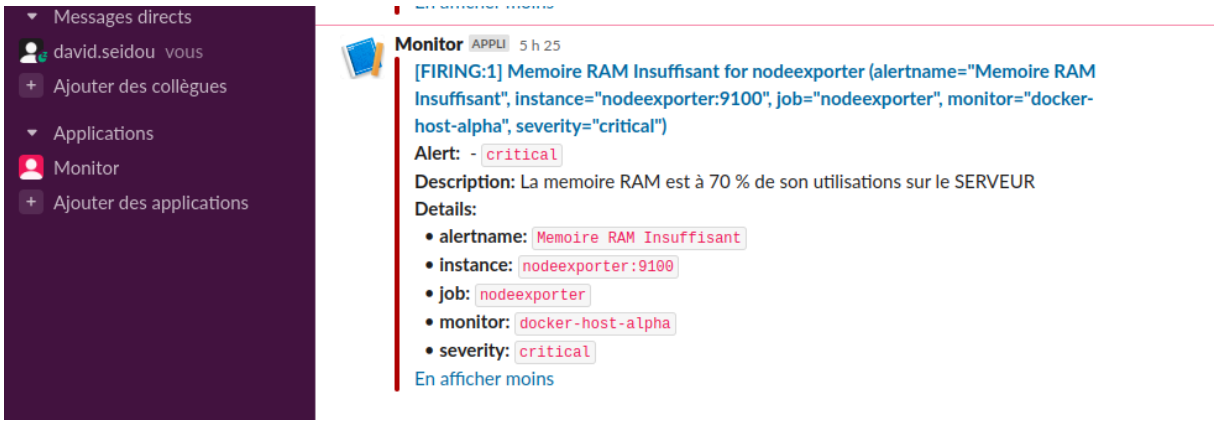


FIGURE 3.4 : Dernier message Slack avant l'arrêt de Mongo

Arrêtons le conteneur **mongo** puis vérifions si nous recevons une notification Slack :

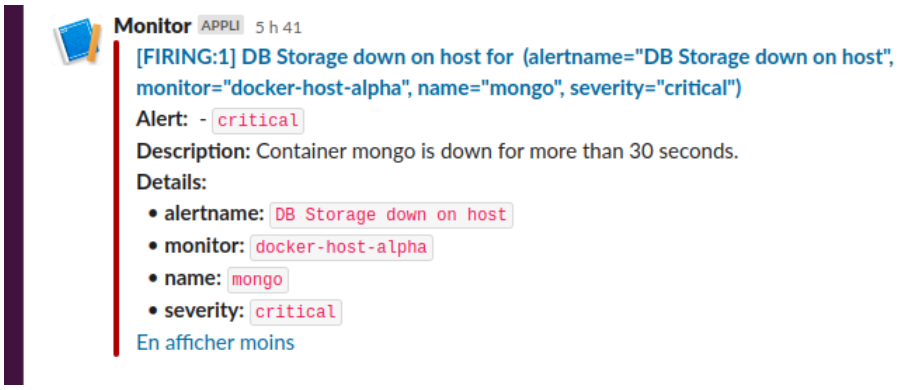


FIGURE 3.5 : Notification Slack

Nous recevons bien l'alerte indiquant que l'application Mongo est indisponible.

3.3 Surveillance d'accès à un serveur via Prometheus

Nous avons également écrit une application **Python** pour alerter en cas de connexion au shell d'un utilisateur quelconque sur le serveur.

Le client **Prometheus** permet d'exposer des métriques depuis une application afin d'être scrappées et visualisées dans Prometheus. Ainsi, nous l'avons intégré dans notre application dans le but d'exposer des métriques systèmes à Prometheus, cette fois-ci sur le chemin **/metrics**.

```
- job_name: 'app-tracker'
  scrape_interval: 5s
  metrics_path: /
  static_configs:
    - targets: ['192.168.66.25:8082']
```

FIGURE 3.6 : prometheus.yml

L'objectif ici est également de démontrer l'utilisation de Prometheus pour la surveillance des systèmes et de mettre en évidence l'utilisation des différents types de métriques.

Les bibliothèques proposent 4 types de métriques principaux : **Counter, Gauge, Histogram et Summary**.

Les **Gauges** sont utilisées pour récupérer des données numériques susceptibles d'être incrémentées ou décrémentées. Cela convient parfaitement à notre cas, puisque l'objectif est d'alerter en cas de connexion au serveur, ce qui implique que le nombre de connexions peut augmenter ou diminuer.

```python
import prometheus_client as prom
import random, time, os, subprocess
from dotenv import load_dotenv

load_dotenv(verbose=True)

req_summary = prom.Summary('control_user', 'Controler les utilisateur se connnectant au systeme')

@req_summary.time()
def process_request(t):
   time.sleep(t)
def getNumberUsers(cmd):
    returned_output = subprocess.check_output(cmd)
    returned_output = returned_output.decode("utf-8").split("\n")[0]
    returned_output = returned_output.split('u')
    number = returned_output[1][len(returned_output[1]) - 2]
    return number

def getListUsers(cmd):
    tabName = []
    tab = subprocess.check_output(cmd).decode("utf-8").split('\n')
    for i in range(2, len(tab) - 1):
        name = tab[i].split('  ')[0]
        tabName.append(name)
    return tabName
if __name__ == '__main__':
   counter = prom.Counter('up_time', 'temps de connexion de la machine')
   gauge = prom.Gauge('number_users_connect', 'Le nombre utilisateur connecte')

   histogram = prom.Histogram('python_my_histogram', 'This is my histogram')
   summary = prom.Summary('list_of_users_connect', 'La liste des utilisateurs connecte')

   prom.start_http_server(int(os.getenv("PORT_METRIC")))

   cmd = "w"

   liste = getListUsers(cmd)


   print("server start on 8082")

   while True:


      gauge.set(str(getNumberUsers(cmd))) #random.random() * 15 -
```

FIGURE 3.7 : Script pour application python

Vérifions les métriques qui ont été récupérées depuis l'URL suivante : **http ://192.168.66.25 :8082/**

```
# HELP number_users_connect Le nombre utilisateur connecte
# TYPE number_users_connect gauge
number_users_connect 2.0
# HELP python_my_histogram This is my histogram
# TYPE python_my_histogram histogram
python_my_histogram_bucket{le="0.005"} 0.0
python_my_histogram_bucket{le="0.01"} 1.0
python_my_histogram_bucket{le="0.025"} 1.0
python_my_histogram_bucket{le="0.05"} 1.0
python_my_histogram_bucket{le="0.075"} 4.0
python_my_histogram_bucket{le="0.1"} 5.0
python_my_histogram_bucket{le="0.25"} 28.0
```

FIGURE 3.8 : Résultat test python

Quand nous vérifions sur le serveur, on a ce résultat :

```
seidou@seidou-Latitude-7480:~$ w
 13:52:58 up 1 day,  1:24,  2 users,  load average: 1,93, 2,08, 2,20
UTIL.    TTY      DE               LOGIN@   IDLE   JCPU   PCPU QUOI
seidou   :0       :0               ven.12   ?xdm?   8:20m  0.02s /usr/li
seidou   pts/4    127.0.0.1        12:42    2.00s  0.14s  0.05s w
```

FIGURE 3.9 : Résultat serveur

Ces métriques sont progressivement récupérées par Prometheus, qui les expose ensuite. Recherchons la valeur de la métrique exposée par notre client Prometheus :

Element	Value
number_users_connect{instance="192.168.66.25:8082",job="app-tracker"}	2

FIGURE 3.10 : Valeur métrique

L'interface graphique personnalisée de Grafana nous montre en temps réel cette métrique.

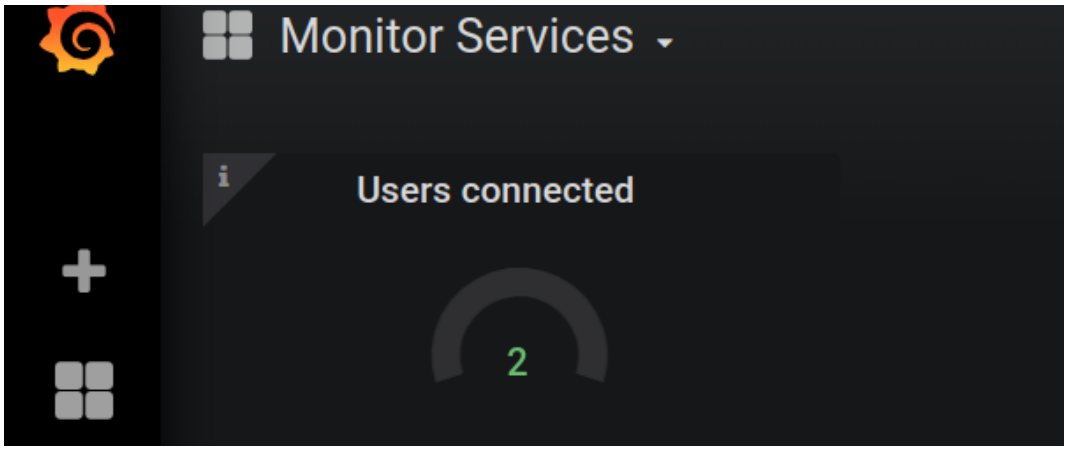


FIGURE 3.11 : Utilisateurs connectés

Nous remarquons ici deux utilisateurs connectés.

L'alerte a été récupérée par Prometheus, qui incrémente le nombre de ces actifs indisponibles conformément à la configuration d'Alertmanager. Pour rappel, c'est le système d'alertes intégré par défaut à Prometheus, permettant de définir les conditions d'alerte si nécessaire.

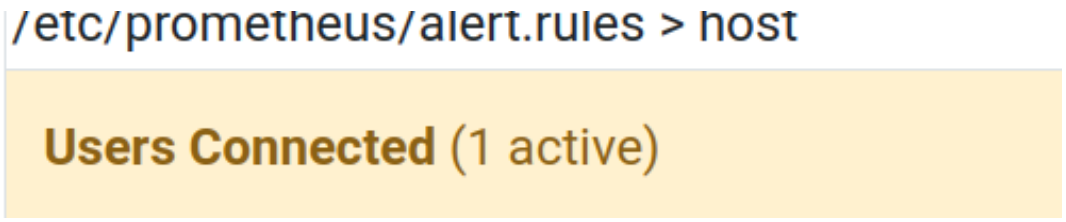


FIGURE 3.12 : Utilisateur connecté

Notre condition est que lorsque le nombre d'utilisateurs connectés est supérieur à 1, nous envoyons une alerte sur Slack pour notifier l'administrateur du serveur. Ci-dessous la condition dans alert.rules

```
1 - alert: Users Connected
2     expr: number_users_connect > 1
3     for: 20s
```

```
4     labels:
5       severity: critical
6     annotations:
7       summary: "Someone is connected to your server"
8       description: "Numbers: {{ humanize $value }}"
```

Par la suite, nous avons effectivement reçu l'alerte.

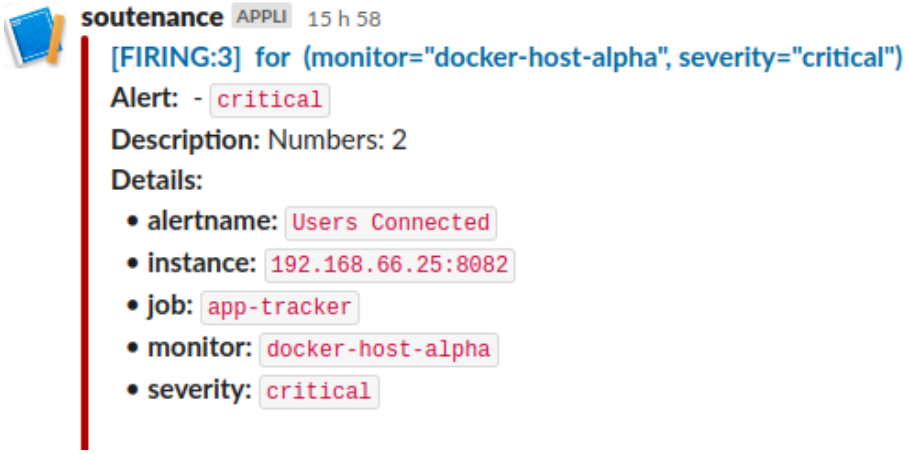


FIGURE 3.13 : Alerte Promeutheus

Nous avons donc pu non seulement surveiller les applications en conteneur, mais aussi surveiller l'activité système grâce à l'exploitation des clients Prometheus.

Conclusion

La disponibilité des applications restera toujours la principale de toutes les inquiétudes des ingénieurs de production et même des personnelles d'une entreprise. Car, lorsqu'une application est indisponible cela entrave l'image du propriétaire et à une lourde conséquence sur le bussiness. C'est pourquoi il est important de mettre en place des solutions d'alertes afin d'augmenter la productivité des équipes de contrôle.

Conclusion Générale et Perspectives

L'objectif de notre projet, après avoir mené des études sur la surveillance d'applications et présenté quelques outils existants, fut de mettre en place un ensemble d'outils favorisant la réactivité des équipes de développement.

Dans un premier temps, nous avons étudié les outils de surveillance, qu'ils soient libres ou payants, qui sont les plus utilisés sur le marché, tout en expliquant pourquoi nous ne les avons pas utilisés dans notre projet. Dans un second temps, nous avons présenté notre architecture de surveillance. Dans cette partie, nous avons décrit chacun des outils faisant partie de cette architecture. Enfin, nous avons installé, configuré ces outils et présenté les différentes interactions entre eux, avant de conclure par une démonstration illustrant les résultats de nos travaux.

Ce projet a été une expérience enrichissante qui nous a permis de mieux comprendre la surveillance des applications web. Nous savons désormais comment choisir la solution idéale en fonction des besoins spécifiques.

Le travail effectué dans ce mémoire peut encore être amélioré. Nous envisageons d'ajouter les points suivants :

- ajouter la notification via SMS ;
- ajouter des sondes Prometheus pour les clusters Kubernetes ;
- collecter et analyser les logs sur les serveurs pour la détection d'intrusion ;
- sécuriser la communication entre Prometheus et ses fournisseurs de métriques, en utilisant par exemple le protocole HTTPS pour les échanges.

Ces perspectives permettront de rendre le système utilisable dans des environnements en production et de renforcer la sécurité de notre solution. Elles contribueront également à accroître l'efficacité des administrateurs systèmes.

Bibliographie

[1]**James Turnbull** *The Docker Book*, *August 26, 2019*

Bibliographie

Webographie

[0] ***Prometheus,*** https://prometheus.io/

[2] ***Grafana,*** https://grafana.com/grafana

[3] ***Cadvisor,*** https://www.kubecost.com/kubernetes-devops-tools/cadvisor

[4] ***Node-Exporter,*** https://github.com/prometheus/node_exporter

[5] ***Pushgateway,*** https://www.metricfire.com/blog/what-is-pushgateway

[6] ***Visual-Paradigm,*** https://online.visual-paradigm.com

[7] ***Docker,*** https://hub.docker.com/

[8] ***Zabbix,*** https://www.zabbix.com/

[9] ***Datadog,*** https://docs.datadoghq.com/

[10] ***Introduction,*** https://www.uptrends.fr/qu-est-ce-que/monitoring-application-web#p%C3%A9nuries-dinventaire

[11] ***Révue de litterature,*** https://www.axiocode.com/monitoring-application/

[12] ***Inventaire,*** https://sokeo.fr/outils-monitoring-application-web/

Table des matières

Dédicace ii

Remerciements iii

Résumé iv

. iv

. v

Table des figures vi

Liste des acronymes viii

Introduction 1

0.1 Problématique 2

0.2 Contexte et Justification 2

0.3 Objectifs 2

0.4 Organisation du mémoire 3

1 Revue de littérature 4

Introduction 4

1.1 Généralités sur la surveillance des applications 4

1.1.1 Définition 4

1.2 Présentation des outils de surveillance sur le web 4

1.2.0.1 Nagios 5

. 5

1.2.0.2 Zabbix 6

. 6

1.2.0.3 Datadog 13

1.3 Comparaison et choix d'implémentation 16

2 Architecture et outils 17

Introduction 17

2.1 Présentation des outils 17

2.1.1 Prometheus 17

2.1.2 PushGateway 20

2.1.3 AlertManager 20

2.1.4 Grafana 21

2.1.5 Cadvisor 22

2.2 Présentation de l'architecture 24

2.2.1 Architecture . . . 24
2.2.2 Mise en place de l'architecture . . . 25
2.2.2.1 Installation de Prometheus et configuration . . . 25
2.2.2.2 Installation et Configuration de Cadvisor . . . 29
2.2.2.3 Installation et Configuration de Nodeexporter . . . 31
2.2.2.4 Installation et configuration de l'AlertManager . . . 33
2.2.2.5 Installation et configuration de Grafana . . . 35

3 Résultats et discussions **37**
Introduction . . . 37
3.1 Test de découverte automatique d'application . . . 37
3.2 Arrêt instinctif des conteneurs . . . 38
3.3 Surveillance d'accès à un serveur via Prometheus . . . 39
Conclusion . . . 42

Conclusion **43**

Bibliographie **44**

Bibliographie **45**

bibliographie **45**

Webographie **46**

Table des matières **47**