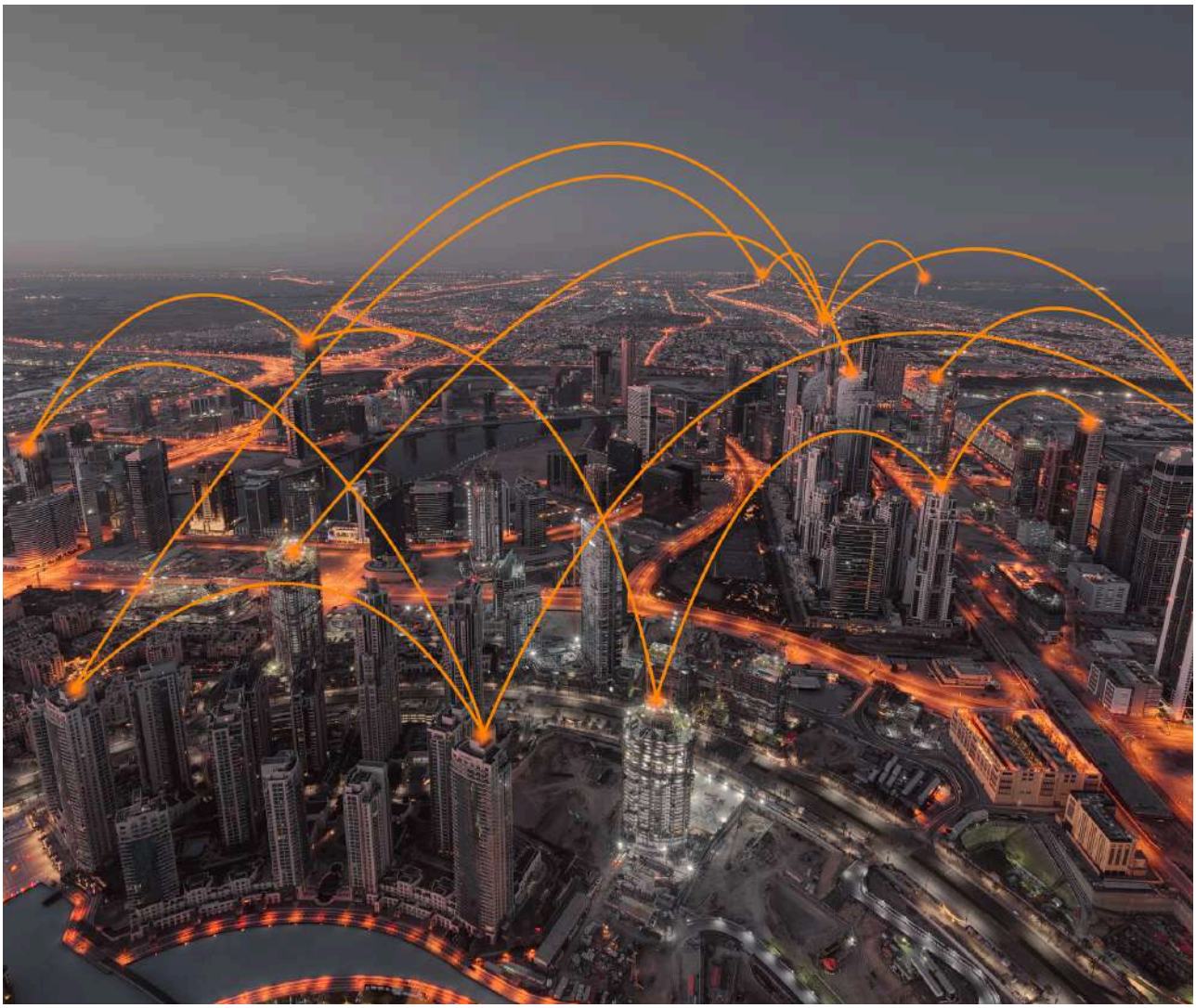
NUMÉRO 1 : FÉVRIER 2025

QUALITÉ & TESTS

DOSSIER : TESTS DE CHARGE	LOCUST, K6, JMETER, OCTOPERF, ...	ENTRETIEN AVEC NICOLAS PIETRARU, UN EXPERT

LE MAG DU TESTEUR



LETTRE DE L'EDITEUR

Chers lecteurs,

Bienvenue dans ce premier numéro de Le Mag du Testeur, un espace pensé pour vous, passionnés et professionnels de la qualité logicielle, qui cherchez à approfondir vos connaissances et découvrir les dernières pratiques en matière de test.

Pour cette première édition, j'ai choisi de plonger dans un sujet nécessaire pour les équipes IT modernes : les tests de charge.

Dans un monde où la performance d'une application peut faire toute la différence pour les personnes et l'image de marque d'une entreprise, il est plus que jamais vital de maîtriser l'art des tests de charge. Que ce soit lors d'une campagne de lancement, d'un événement marketing, ou au quotidien, nous sommes confrontés à la nécessité de garantir une expérience fluide, peu importe la charge ou les circonstances.

Dans ce dossier spécial, vous découvrirez des outils comme JMeter, Locust, K6, Neoload, LoadRunner ou Octoperf. J'aborde leurs spécificités, leurs forces et leurs limites, pour vous permettre de choisir celui qui conviendra le mieux à vos besoins. Vous trouverez aussi des conseils pour éviter les pièges courants, des études de cas, et des exemples concrets d'utilisation.

FAVORITE

Le petit logo favorite désigne les outils que je préfère, c'est une opinion personnelle. Chacun des outils présentés a ses avantages et ses inconvénients. Vous pouvez donc choisir votre outil selon votre contexte et les tests que vous envisagez.

Mon souhait est de faire de ce magazine, Le Mag du Testeur, une ressource précieuse pour tous ceux qui œuvrent à l'amélioration de la qualité logicielle, dans un secteur où la rigueur et la performance sont plus importantes que jamais.

J'espère que ce numéro vous inspirera et vous donnera les clés pour garantir des systèmes toujours plus robustes et performants.

Bonne lecture, et n'hésitez pas à me faire part de vos retours et de vos expériences – car c'est ensemble que nous continuerons à élever les standards de qualité.

Avec enthousiasme,

Fanny Velsin

SOMMAIRE

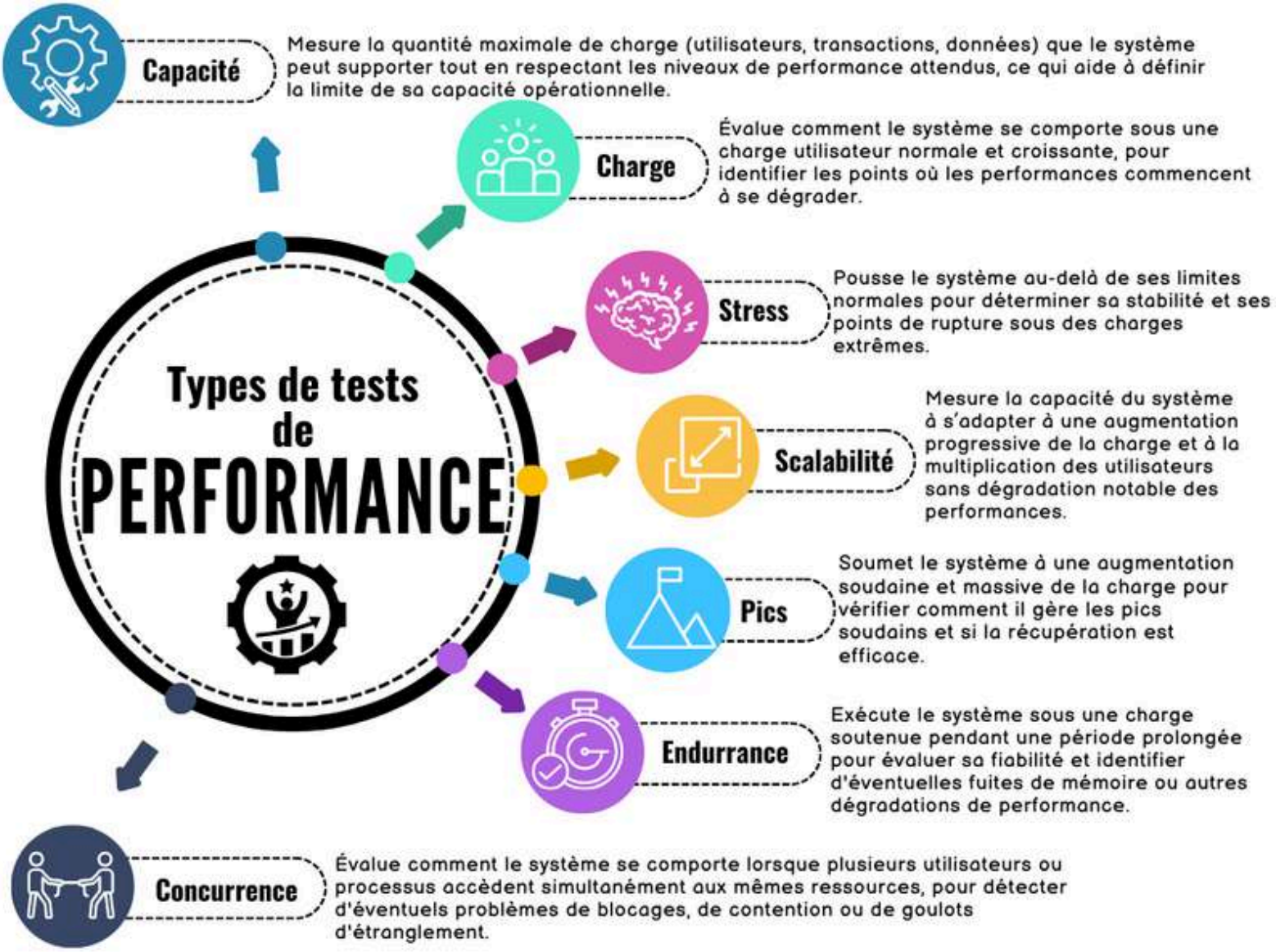
Nous serions ravis de lire vos impressions et suggestions pour les prochains numéros :

feedback@lemagdutesteur.fr

INTRODUCTION AUX TESTS DE PERFORMANCE

Une application lente ou instable peut non seulement frustrer les utilisateurs, mais aussi nuire gravement à la réputation d'une entreprise. Les tests de performance sont donc importants pour évaluer, comprendre et améliorer la réactivité, la stabilité, et la robustesse des systèmes sous différentes conditions d'utilisation.

Les tests de performance regroupent plusieurs catégories, adaptées à des questions spécifiques sur le comportement de l'application. Voici les principaux types de tests de performance :



QU'EST-CE QU'UN MODE DEGRADE ?

Un système fonctionne en mode dégradé lorsqu'il continue à fournir un service, mais avec des fonctionnalités réduites, une qualité diminuée ou des performances limitées. Ce mode est souvent conçu pour éviter une panne complète et offrir un niveau minimal de service.

Exemples de mode dégradé :

- Bases de données : une base secondaire prend le relais en lecture seule si la base principale est indisponible.

- Caches : si le cache est saturé ou non disponible, le système interroge directement la base de données, avec un impact sur les performances.

- Ressources réseau : si un serveur principal ne répond pas, un serveur de secours est activé.

Objectifs des tests de charge en mode dégradé

- Évaluer la résilience : Tester la capacité du système à maintenir un service minimal en cas de dégradation.

- Observer les performances : Mesurer les temps de réponse, la consommation des ressources et les capacités en mode dégradé.

- Identifier les points de rupture : Déterminer à quel niveau de charge le système passe d'un mode dégradé à une panne totale.

- Valider les mécanismes de fallback : S'assurer que les systèmes de secours (redirection, basculement) fonctionnent comme prévu.

- Améliorer l'expérience d'utilisation : Vérifier que l'application reste utilisable même en cas de dégradation.

Mise en œuvre des tests de charge en mode dégradé

Définissez les scénarios de dégradation

Il est important d'identifier les composants critiques et les situations où le mode dégradé est activé.

Configurez les outils de test

Les outils de test de charge comme K6 ou JMeter peuvent être configurés pour simuler des scénarios de dégradation. Par exemple :

- En injectant des erreurs 500 dans certaines réponses HTTP.
- En limitant les ressources disponibles (CPU, mémoire).

Simulez la charge

- Progressive : augmenter graduellement le nombre de profil en mode dégradé.

- Pointe : soumettre immédiatement une forte charge au système dégradé pour tester sa limite.

Analysez les résultats en collectant les métriques

- Temps de réponse.
- Taux d'erreurs.
- Consommation des ressources.
- Nombre de personnes simultanés supportés avant panne totale.

Exemple pratique

Cas d'une application web :

- Contexte : Une application web repose sur un cache Redis et une base de données MySQL.
- Scénario : Redis devient inaccessible, et le système passe en mode dégradé en interrogeant directement MySQL.
- Test :
  a. Déconnecter Redis.
  b. Lancer un test de charge avec 1 000 personnes simultanés.
  c. Observer si le système maintient une réponse correcte (même ralentie) ou s'effondre.

Cas d'une application :

- Contexte : Une application mobile est utilisée par des randonneurs dans des zones reculées avec une couverture réseau limitée (3G, latences élevées, déconnexions fréquentes). Cette application offre des fonctionnalités comme :
  - Cartes téléchargeables.
  - Signalement d'urgence en cas de danger.
  - Statistiques de randonnée synchronisées dans le cloud.
- Test :
  - Lorsque la connexion internet devient instable ou très lente, l'application doit fonctionner en mode déconnecté ou avec des fonctionnalités réduites, tout en stockant les données localement pour une synchronisation ultérieure.

Bénéfices des tests de charge en mode dégradé

- **Anticiper les faiblesses en identifiant les goulots d'étranglement.**
- **Renforcer la stratégie de reprise en optimisant les mécanismes de basculement.**
- **Réduire les risques en préparant le système à des conditions imprévues.**

COMMENT GARANTIR UNE STABILITE OPTIMALE POUR VOTRE SYSTEME ?

Dans le domaine de la gestion de projets IT, la qualité du système que vous mettez en production est indispensable pour assurer une expérience d'utilisation sans faille et une performance optimale de votre infrastructure.

Les tests de charge sont un élément clé de cette assurance qualité, notamment pour les applications critiques, utilisées par un grand nombre de personnes simultanément ou qui manipulent des volumes de données importants.
Dans cet article, nous allons explorer ce que sont les tests de charge, pourquoi ils sont nécessaires et comment les mettre en œuvre efficacement pour garantir la stabilité de vos systèmes.

Qu'est-ce que le test de charge ?

Un test de charge est un type de test non fonctionnel qui vise à évaluer la capacité d'un système ou d'une application à gérer un nombre important de personnes ou une grande quantité de données. Ce test pousse le système jusqu'à ses limites pour mesurer sa performance, ses temps de réponse, et sa stabilité. Il est particulièrement utile dans les situations où le nombre de personnes peut varier fortement, comme lors d'une campagne marketing ou d'un lancement de produit.
Ces tests permettent de simuler des scénarios réels où la charge du système peut être élevée et de vérifier comment il réagit. L'objectif est d'identifier les potentiels points de défaillance avant qu'ils n'affectent les personnes finales.

Pourquoi ces tests de charge sont-ils incontournables ?

Pour un chef de projet IT axé sur la qualité, les tests de charge revêtent plusieurs intérêts :

- Assurer la confiance des parties prenantes : les équipes de développement et de production sont souvent sous pression pour déployer rapidement de nouvelles fonctionnalités ou pour supporter des événements ponctuels. Un système échouant aux tests de charge peut se bloquer ou ralentir sous une forte utilisation, affectant l'image de marque et entraînant des pertes financières.

- Prévenir les pannes : En testant un système sous des charges élevées, on peut identifier les goulots d'étranglement ou les composants qui nécessitent une optimisation avant la mise en production. Cela minimise significativement les risques d'interruption de service.

- Optimiser les coûts de l'infrastructure : Les tests de charge permettent aussi de vérifier que votre infrastructure est bien calibrée pour répondre aux besoins sans surcoût. En identifiant les limites de votre système, vous pouvez ajuster les ressources de manière optimale.

Comment réaliser des tests de charge efficaces ?

- Planifier et définir des objectifs : Avant de commencer les tests, il faut de définir les indicateurs de performance clés (KPI), comme les temps de réponse maximum, le nombre de personnes simultanées attendues, ou les taux d'erreurs acceptables. Assurez-vous que ces objectifs sont réalistes et basés sur des scénarios d'utilisation concrets.

- Choisir ses outils : Il existe une multitude d'outils de tests de charge comme JMeter, K6 et Locust. Chacun de ces outils possède des fonctionnalités spécifiques qui peuvent répondre à différents besoins, que ce soit pour simuler un grand nombre de personnes, collecter des données de performance ou analyser les comportements du système.

- Créer des scénarios de test : Pour un test de charge réaliste, créez des scénarios qui reproduisent les usages réels de votre clientèle. Par exemple, si votre application est un site e-commerce, vous pourriez simuler une augmentation des visiteurs en période de soldes, avec des personnes naviguant, ajoutant des articles au panier, et finalisant leurs achats.

- Analyser et ajuster : Après le test, une analyse détaillée des résultats est incontournable. C'est la raison de l'exécution de ce test. Cherchez les points faibles identifiés : les réponses lentes, les erreurs fréquentes, ou les surcharges sur certains serveurs. En fonction des données recueillies, vous pourrez apporter des optimisations, que ce soit au niveau du code, de l'infrastructure, ou des configurations de réseau

.

- Automatiser et répéter : Les tests de charge ne devraient pas être faits une seule fois. Avec des cycles de développement agiles et des mises en production fréquentes, les tests doivent être automatisés et exécutés régulièrement pour identifier les problèmes avant qu'ils ne deviennent critiques.

Quels sont les pièges à éviter?

Pour maximiser l'efficacité de vos tests de charge, quelques erreurs courantes sont à éviter :

- Ignorer les charges variables : Les charges de test doivent refléter les variations réelles de trafic (heures de pointe, périodes de faible activité) pour éviter de biaiser les résultats.
- Négliger la surveillance en temps réel : Lors des tests, surveillez les logs et les métriques en temps réel. Cela vous permettra d'identifier rapidement tout comportement anormal sans attendre la fin des tests.
- Trop dépendre d'un seul outil : Les outils peuvent varier en précision et en scope. Combinez différents outils si nécessaire pour couvrir l'ensemble des cas d'usage de votre système.

QU'EST-CE QU'UN KPI ?

Un KPI, indicateur de performance, est une métrique mesurable utilisée pour évaluer la performance d'un système ou d'une application. En test de charge, les KPI permettent de répondre à des questions telles que :

- Le système peut-il gérer le nombre d'utilisateurs attendu ?
- Les temps de réponse respectent-ils les attentes ?
- Y a-t-il des risques de dégradation sous forte charge ?

Exemples de KPI courants dans les tests de charge

- Temps de réponse (Response Time) : Temps écoulé entre une requête et la réponse du système.
  - Objectif : Identifier les latences pour chaque type de requête.
  - Exemple : Une API critique doit répondre en moins de 200 ms dans 95 % des cas.

- Débit (Throughput) : Nombre de requêtes ou transactions traitées par seconde.
  - Objectif : Mesurer la capacité du système à gérer des charges importantes.
  - Exemple : Supporter 500 requêtes/secondes sans dégradation de la performance.

- Taux d'échec (Error Rate) : Pourcentage de requêtes échouées.
  - Objectif : Identifier des problèmes de fiabilité sous charge.
  - Exemple : Le taux d'échec ne doit pas dépasser 1 %.

- Utilisation des ressources (Resource Utilization) : Consommation de CPU, RAM, disque et réseau.
  - Objectif : Vérifier que le système utilise ses ressources efficacement.
  - Exemple : La consommation CPU ne doit pas excéder 80 %.

- Temps de montée en charge (Ramp-Up Time) : Durée nécessaire pour atteindre une charge cible.
  - Objectif : Mesurer la capacité à absorber une augmentation rapide de trafic.

LES GOLDEN SIGNALS

Les "golden signals" sont une méthode reconnue pour surveiller les systèmes de production et détecter les problèmes de performance. Ces quatre métriques peuvent également être appliquées aux tests de charge.

- Latence
  - Définition : Temps nécessaire pour répondre à une requête.
  - Indicateur clé : Latence moyenne, médiane et aux percentiles (95e, 99e).
  - Pourquoi c'est important ?
    - Une latence élevée peut indiquer des goulots d'étranglement.
    - Mesure la perception utilisateur (expérience fluide ou frustration).

- Erreurs (Errors)
  - Définition : Proportion de requêtes échouées ou incorrectes.
  - Indicateur clé : Taux d'erreur global, erreurs spécifiques par endpoint.
  - Pourquoi c'est important ?
    - Les erreurs peuvent révéler des défaillances système, des problèmes de code ou des limites de configuration.

- Trafic (Traffic)
  - Définition : Volume de demandes ou de transactions gérées par le système.
  - Indicateur clé : Requêtes par seconde, bande passante utilisée.
  - Pourquoi c'est important ?
    - Aide à déterminer la charge maximale supportée.
    - Évalue si les ressources allouées sont suffisantes.

- Saturation (Saturation)
  - Définition : Niveau de consommation des ressources système.
  - Indicateur clé : CPU, RAM, IOPS disque, utilisation réseau.
  - Pourquoi c'est important ?
    - Identifie les points où le système commence à s'effondrer.
    - Prédit la capacité maximale avant panne.

Golden Signal	KPI associé	Exemple
Latence	Temps de réponse	95 % des requêtes < 200 ms
Trafic	Débit	500 requêtes/s pendant les soldes
Erreurs	Taux d'échec	Moins de 1 % d'erreurs HTTP 500
Saturation	Utilisation CPU	CPU < 80 % avec 1 000 utilisateurs

QU'EST-CE QU'UN SLA EN TEST DE CHARGE ?

Un SLA est un engagement formel ou un accord entre un fournisseur de service (par exemple, une équipe technique ou un prestataire) et un client (interne ou externe). Il définit les niveaux de service attendus et établit des objectifs mesurables.

Contrairement aux KPI, qui sont des outils d'analyse, les SLA servent de référence pour évaluer si le système respecte les attentes.
Les SLA s'appuient souvent sur des KPI pour mesurer leur respect, mais ils définissent un seuil minimal à atteindre

- **Par exemple** :
  - Temps de réponse : 95 % des requêtes doivent avoir un temps de réponse inférieur à 200 ms.
  - Disponibilité : Le service doit être opérationnel 99,9 % du temps.
  - Capacité : Le site doit gérer jusqu'à 1000 utilisateurs simultanés sans dégradation significative.

	KPI	SLA
Définition	Indicateur utilisé pour mesurer la performance du système	Engagement formel sur des niveaux de service à atteindre
Objectif	Identifier et analyser les performances.	Garantir un niveau minimal de performance.
Exemple	Temps de réponse moyen Taux d'utilisation CPU Taux d'échec des requêtes	temps de réponses maximum autorisé Taux de disponibilité garanti Capacité minimale à supporter
Focus	Performance globale et analyse détaillée	Minimum garanti

LES OUTILS DE TESTS DE CHARGE

Les outils comme Locust, JMeter, et K6 sont principalement connus pour leurs capacités en matière de tests de charge, mais ils sont également capables d'exécuter certains des autres tests de performance décrits ci-dessus. Voici un aperçu de leurs capacités :

- **Apache JMeter** : Cet outil polyvalent est utilisé pour une large gamme de tests de performance, y compris les tests de charge, de stress, et d'endurance. Sa flexibilité et sa capacité à gérer des protocoles variés (HTTP, FTP, WebSocket, etc.) le rendent idéal pour des tests de charge sur des applications web, des APIs et même des bases de données. JMeter est également très utilisé pour les tests de capacité et de scalabilité.

- **Locust** : Locust est un outil de test de charge basé sur Python, axé sur la simplicité et l'évolutivité. Il est particulièrement efficace pour simuler des charges importantes avec des scénarios personnalisés. Locust peut être utilisé pour des tests de charge, de stress, et d'endurance, en particulier sur des APIs, des applications web, et des microservices. Cependant, il est moins adapté pour les tests de protocoles non-HTTP.

- **K6** : K6 est un outil moderne et facile à prendre en main, idéal pour les tests de charge, de stress, et d'endurance, notamment pour les applications web et les APIs REST. Il est basé sur JavaScript, ce qui le rend accessible aux développeurs. Bien que principalement utilisé pour les tests de charge, il permet aussi de faire des tests de scalabilité et de pics.

- **OctoPerf** : OctoPerf est une solution SaaS simple et rapide à prendre en main pour les tests de performance. Avec son interface conviviale, il permet de créer des scénarios sans programmation et de tester les performances des applications web et APIs. Il s'intègre facilement aux outils DevOps pour une automatisation fluide.

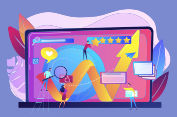
- **LoadRunner** : LoadRunner est un outil puissant pour les tests à grande échelle, capable de simuler des millions d'utilisateurs. Il prend en charge de nombreux protocoles (HTTP, FTP, SAP, etc.) et offre des diagnostics avancés pour identifier les goulots d'étranglement. Il est adapté aux environnements cloud et complexes.

- **NeoLoad** : NeoLoad est conçu pour tester les applications modernes (APIs, microservices, cloud). Avec des scripts dynamiques et une intégration DevOps, il simplifie les tests continus. Il supporte divers protocoles et propose une analyse détaillée pour optimiser les performances.

Les tests de performance ne se limitent pas aux tests de charge : ils couvrent une gamme de scénarios destinés à évaluer le comportement d'un système sous différentes conditions de stress.
Utiliser ces outils permet aux équipes de tester efficacement leurs applications en simulant des situations variées. Ces outils apportent chacun leurs spécificités et avantages, et en les utilisant judicieusement, les équipes peuvent garantir que leur application sera rapide, stable, et fiable, quelles que soient les conditions de charge.

Mise en place d’un test de charge

1 KPIs



Identifiez pourquoi vous faites le test de charge. Est-ce pour évaluer la capacité de l’application à gérer des pics de trafic ? Pour tester la performance après une mise à jour majeure ?

2 Comportements

Identifiez les comportements types des personnes utilisant votre application. Par exemple, sur un site e-commerce, les actions courantes pourraient être la navigation, l’ajout au panier, et la finalisation de l'achat.

3 Choix

Choisissez un outil qui correspond aux spécificités de votre projet.
Installez l’outil et assurez-vous qu’il est bien configuré pour accéder aux ressources de votre application (authentification, connexions aux bases de données, etc.).

4 Environnement

Pour éviter que le test de charge n’impacte les personnes réelles, effectuez-le dans un environnement qui reproduit les configurations de production sans interférer avec celle-ci.
Intégrez des outils de monitoring (comme Grafana, Prometheus ou même des dashboards AWS pour le cloud) afin de suivre en temps réel les ressources consommées (CPU, RAM, utilisation de la bande passante).

5 Scénario

Écrivez les scripts qui simulent le comportement des personnes en suivant les scénarios définis.
Configurez les paramètres, tels que le nombre de personnes simulées, le taux d’augmentation de la charge, et la durée de chaque étape du test. Certains outils permettent une montée progressive de la charge pour observer le comportement sous des charges croissantes.

6 Lancer le test et monitorez

Démarrez le test en surveillant les indicateurs de performance clés en direct. Assurez-vous que les métriques de l’application (comme les taux d’erreur ou les pics de latence) sont visibles pendant l’exécution pour identifier rapidement des problèmes.

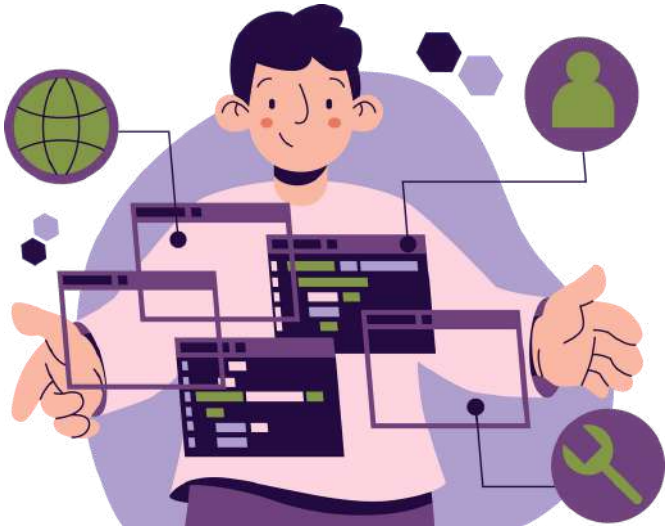
7 Analysez les résultats

Évaluez les résultats par rapport aux objectifs initiaux : Étudiez les temps de réponse, le débit, et les taux d’échec.
Identifiez les goulots d'étranglement et notez les sections où l’application a montré des faiblesses.
Par exemple, si une requête API est lente sous forte charge, envisagez d’optimiser le code ou la configuration du serveur.

8 Répétez les tests

Automatisez les tests de charge pour les exécuter régulièrement, notamment après chaque mise à jour significative. Intégrer les tests de charge dans un pipeline CI/CD permet de détecter les régressions de performance en continu.
Des tests périodiques vous assurent que les performances restent conformes aux attentes dans le temps.

API D'EXEMPLES



Nous utiliserons JSONPlaceHolder pour illustrer nos comparaisons entre les différents outils.

JSONPlaceholder est une API REST gratuite qui permet de tester et simuler des requêtes sans configurer de serveur. Elle est idéale pour les essais, les démos ou les formations sur les appels API. Cette API expose plusieurs endpoints standards, qui simulent des données réalistes pour des ressources typiques comme des comptes, des posts, des commentaires, des photos et des tâches (todos).
Accessible à l'adresse https://jsonplaceholder.typicode.com,
JSONPlaceholder supporte les méthodes HTTP classiques (GET, POST, PUT, DELETE) sur ses ressources, permettant aux développeurs de réaliser des tests complets sans affecter des données réelles. Elle est largement utilisée dans les exemples de code, les tests d'intégration et les outils de charge, car elle offre une base stable et gratuite pour expérimenter les concepts d'API.

L'outil offre des endpoints simulant une API REST réelle. Voici quelques endpoints que vous pouvez tester :

- /posts : Liste de faux articles (POST).
- /comments : Liste de commentaires (GET).
- /users : Informations de comptes fictifs (GET).

JSONPlaceholder est une API publique gratuite, mais elle a des limites de charge. Pour éviter d'éventuels blocages et par respect pour cet outil, réalisez vos tests avec un faible nombre de personnes et une montée en charge progressive.

Sinon, vous pouvez également créer vos propres données de test avec Mockend. Mockend est un service en ligne qui permet aux développeurs de créer des API REST et GraphQL simulées rapidement, en utilisant un simple fichier de configuration JSON. L'idée de Mockend est de fournir une API fictive facilement configurable, basée sur des données de test, pour le développement et les tests sans avoir besoin de mettre en place un backend réel. C'est un outil pratique pour le prototypage, le développement frontend, les tests d'intégration, et les démonstrations de produit.

BUILD VS BUY : QUEL CHOIX POUR VOS TESTS DE CHARGE ?

Lorsqu'une organisation décide de mettre en place des tests de charge, elle doit choisir entre deux grandes approches :

- "Build" : Construire une solution en s'appuyant sur des outils open source ou internes.
- "Buy" : Acheter une solution commerciale prête à l'emploi.

Chacune de ces approches a ses avantages, ses inconvénients et ses implications, qui doivent être soigneusement évalués.

Approche Build : flexibilité, mais effort conséquent

L'approche "Build" consiste à assembler ou développer une solution en utilisant des outils open source comme Locust, JMeter, ou K6, ou en créant une solution personnalisée adaptée aux besoins spécifiques.

Avantages :

- Coût initial faible : Les outils open source sont souvent gratuits, éliminant les frais de licence.
- Flexibilité totale : Vous avez un contrôle total sur la configuration et pouvez personnaliser l'outil pour répondre à vos besoins exacts.
- Communauté active : Les outils open source bénéficient souvent d'une communauté active qui propose des plugins, des mises à jour et un support technique informel.
- Apprentissage et expertise interne : Développer une solution open source permet à vos équipes d'acquérir des compétences techniques profondes.

Inconvénients :

- Temps de développement élevé : La mise en place et la personnalisation d'une solution peuvent nécessiter des semaines, voire des mois.
- Coût de maintenance non négligeable : Bien que les outils soient gratuits, leur maintenance (mises à jour, bugs, documentation) peut nécessiter des ressources importantes.
- Support limité : En cas de problème critique, vous dépendez de la communauté ou de vos propres compétences internes.
- Manque de standardisation : Chaque organisation peut développer des workflows différents, ce qui complique les transferts de connaissances.

Pour qui ?

- Organisations ayant des équipes techniques compétentes et le temps nécessaire pour personnaliser et maintenir des outils.
- Projets avec des besoins très spécifiques que les outils commerciaux ne couvrent pas.

Approche Buy : efficacité, mais coût élevé

L'approche "Buy" implique l'acquisition d'une solution commerciale, comme LoadRunner, Neoload ou Octoperf.

Avantages :

- Prêt à l'emploi : Les solutions commerciales sont immédiatement fonctionnelles avec des fonctionnalités intégrées comme le monitoring, les rapports avancés, et les scénarios prédéfinis.

- Support technique professionnel : Vous bénéficiez d'un support 24/7 en cas de problème, garantissant une continuité des opérations.

- Scalabilité éprouvée : Les solutions commerciales sont souvent optimisées pour des tests à grande échelle, avec une prise en charge étendue des protocoles (SAP, Citrix, bases de données, etc.).

- Rapidité de mise en œuvre : Idéal pour les entreprises avec des contraintes de temps, car ces outils nécessitent peu de configuration.

Inconvénients :

- Coût élevé : Les licences annuelles peuvent coûter plusieurs dizaines de milliers d'euros, notamment pour des tests à grande échelle.

- Moins de flexibilité : Bien que les outils commerciaux soient robustes, ils peuvent ne pas offrir autant de possibilités de personnalisation qu'une solution open source.

- Dépendance au fournisseur : Les évolutions de l'outil dépendent du fournisseur, et un changement de solution peut engendrer des coûts importants.

Pour qui ?

- Entreprises ayant des besoins standards et des budgets élevés.

- Projets critiques nécessitant un support garanti et une scalabilité immédiate.

	Build	Buy
Coût initial	Faible (gratuité des outils open source)	Élevé (licences commerciales)
Coût total (TCO)	Variable : coût de développement et maintenance.	Élevé : licences, mais maintenance incluse.
Scalabilité	Dépend des compétences internes.	Généralement très bonne.
Temps de mise en œuvre	Long.	Rapide.
Personnalisation	Totale.	Limitée.
Support technique	Communauté (non garanti).	Professionnel, réactif.
Couverture des protocoles	Dépend de l'outil choisi.	Très large (SAP, Citrix, etc.).

COMPARAISON DES OUTILS GRATUITS

	Locust	K6	JMeter 
Langage	Python	Javascript	Java
Type de test	Charge, stress	Charge, stress, endurance	Charge, stress, endurance
Principaux avantages	Simplicité, extensible pour de gros volumes, interfaces web en temps réel	Facilité d'utilisation, moderne, API REST	très flexible, prise en charge de multiples protocoles (HTTP, FTP, etc,..)
Inconvénients	Moins adapté aux protocoles non-HTTP, rapports de résultats simplifiés	Support limité pour certains protocoles, moins d'options de visualisation par défaut	Interface graphique Peut nécessiter plus de mémoire pour des tests intensifs
Adapté pour	API, Applications web, Microservices	Application web, API REST	Application web, API, bases de données

Voici une analyse détaillée des outils présentés

LOCUST

Introduction à Locust

Locust est un outil open-source de test de charge conçu pour être utilisé de manière très flexible et extensible. Contrairement à de nombreux outils classiques de test de charge, Locust permet d'écrire des scénarios de test en Python, offrant ainsi aux équipes une souplesse dans la création de comportements réalistes et complexes.

Pourquoi choisir Locust ?

La force principale de Locust réside dans sa capacité à simuler des personnes réalistes. Plutôt que de simplement envoyer des requêtes, Locust permet de reproduire des comportements détaillés, ce qui est particulièrement utile pour tester des systèmes ayant des flux spécifiques ou des applications web avec des interactions multiples. De plus, son architecture distribuée permet d'évoluer pour tester des charges très importantes.

Fonctionnalités Clés de Locust

- Locust utilise Python pour définir les scénarios, rendant les tests intuitifs et permettant aux testeurs de tirer parti des bibliothèques Python. La syntaxe est facile à apprendre et permet une grande flexibilité dans les comportements simulés.
- Locust dispose d'une interface web intégrée permettant de surveiller les tests en temps réel, d'ajuster la charge ou d'arrêter le test à tout moment.
- En configurant plusieurs instances de Locust, il est possible de simuler des centaines de milliers de personnes simultanées, ce qui est essentiel pour tester des applications très sollicitées.
- Locust permet d'imiter des comportements humains, avec des pauses entre les actions, des enchaînements logiques et des variables de temps d'attente.

Cas d'utilisation de Locust en entreprise

- Tests de charge pour des applications à flux complexes : Locust est particulièrement adapté aux applications où les personnes suivent des parcours spécifiques, comme des achats ou des formulaires en ligne.
- Simulation de grandes charges distribuées : Avec son architecture distribuée, Locust est idéal pour des environnements où des pics de charge significatifs sont attendus, par exemple lors de campagnes marketing.
- Locust est couramment employé pour les API REST grâce à la flexibilité de son scripting Python

Limites de Locust

Bien que Locust offre une grande flexibilité, il requiert des connaissances en Python, ce qui peut être une barrière pour les équipes sans développeurs Python. En outre, Locust peut être gourmand en ressources lorsque les tests nécessitent de nombreuses personnes simultanées, ce qui peut demander une infrastructure importante pour les gros tests.

RESUME DES POINTS CLEFS

Les avantages de Locust

1. Simplicité : Écrire des tests de charge avec Python permet une grande flexibilité. Locust est particulièrement adapté pour simuler des comportements complexes, tout en restant accessible aux développeurs et testeurs.
2. Extensibilité : Locust peut être déployé sur plusieurs machines pour simuler des milliers, voire des millions de personnes simultanément, rendant possible des tests massifs.
3. Interface web conviviale : Locust fournit une interface web simple pour lancer, surveiller et ajuster vos tests en temps réel.
4. Comportement personnalisé : Grâce à la programmation des tests en Python, vous pouvez simuler des scénarios réalistes et variés.

Les inconvénients de Locust

- Dépendance au langage Python : Bien que Python soit largement utilisé, cela peut être un obstacle pour les équipes qui ne sont pas familières avec ce langage. Si votre équipe n'a pas de compétences en Python, cela peut rendre l'écriture des scénarios de test plus difficile.

- Complexité de mise en place des tests distribués : Bien que Locust supporte les tests distribués (lorsque vous voulez simuler des milliers de personnes à partir de plusieurs machines), la mise en place peut parfois être complexe. Il faut gérer manuellement les workers (machines) et assurer la bonne synchronisation entre elles, ce qui peut devenir un défi pour les tests à très grande échelle.

- Rapports de résultats simplifiés : Par défaut, Locust fournit des rapports assez basiques avec le nombre de requêtes, le taux d'échec, et le temps de réponse moyen. Contrairement à des outils comme JMeter, il manque certaines métriques avancées comme les percentiles des temps de réponse (bien qu'ils puissent être calculés manuellement) ou des graphiques détaillés intégrés. Pour des rapports plus complexes, il faut souvent exporter les données et les traiter avec des outils externes.

- Pas d'interface graphique pour la création de scénarios : Locust repose entièrement sur l'écriture de code Python pour la configuration des tests. Contrairement à JMeter, qui propose une interface graphique (GUI) pour configurer les scénarios de test, Locust nécessite des compétences en programmation pour définir les interactions, ce qui peut être un frein pour certains testeurs moins techniques.

- Support restreint pour les protocoles non-HTTP : Locust est principalement orienté vers les tests d'applications web et d'API REST (protocoles HTTP/HTTPS). Il n'est pas adapté à d'autres types de protocoles comme FTP, SMTP, ou encore des tests d'applications utilisant des protocoles plus spécifiques, contrairement à des outils comme JMeter ou LoadRunner.

INSTALLATION DE LOCUST

L'installation de Locust est rapide. Après avoir installé Python, il suffit de taper :

```
pip install locust
```

CREATION D'UN TEST DE BASE

Ensuite, créez un script de test Python qui décrit les actions que votre clientèle effectuera. Voici un exemple simple, nommé le fichier *mon_fichier.py* :

```
from locust import HttpUser, task, between

class JSONPlaceholderUser(HttpUser):
  wait_time = between(1, 3)

  @task(3)
  def get_posts(self):
    self.client.get("/posts")

  @task(2)
  def get_comments(self):
    self.client.get("/comments")

  @task(1)
  def get_users(self):
    self.client.get("/users")
```

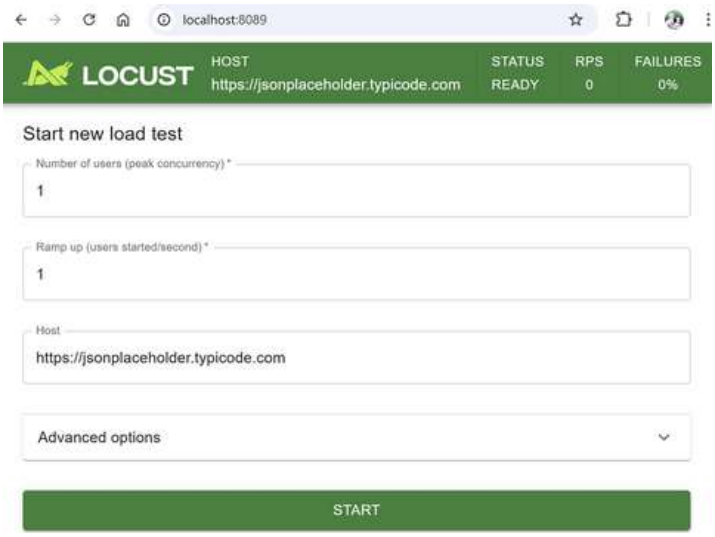
Ce script simule des personnes qui visitent deux pages (/ et /about). wait_time est utilisé pour spécifier un temps d'attente aléatoire entre deux actions afin de mieux simuler un comportement humain.

LANCER LE TEST

Pour démarrer le test, il suffit de lancer la commande suivante :

```
locust -f mon_script.py
```

Cela ouvrira une interface web à l'adresse http://localhost:8089. À partir de là, vous pouvez configurer le nombre de personne et leur fréquence d'arrivée pour commencer le test.



Ci-dessus est l'interface graphique de Locust. Vous y trouverez les options :

- **Nombre de personnes simulés** (Number of users) : vous permet de spécifier combien de personnes simultanées seront simulés pendant le test.

- **Taux de montée en charge** (Spawn rate) : Ici, vous définissez le nombre de personne par seconde qui seront ajoutés jusqu'à atteindre le nombre total de personnes. Cela permet de moduler la montée en charge plutôt que de lancer toutes les personnes en même temps.

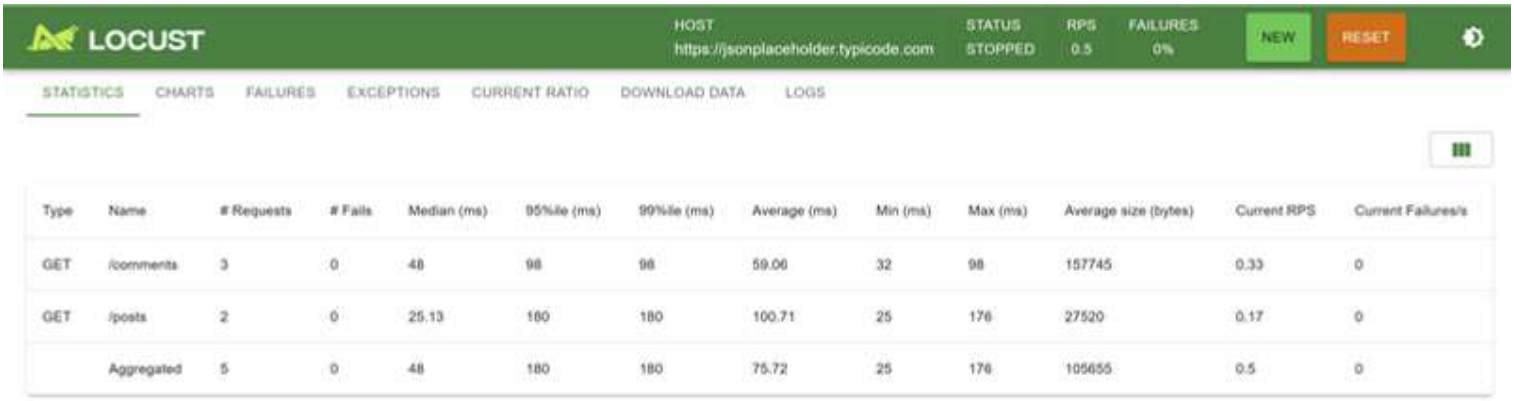
- **Hôte** (Host) : Cette option vous permet de définir l'URL de l'application cible.

- **Durée du test** (Run time): Sous sous le bouton advanced options, vous pouvez définir une durée spécifique pendant laquelle Locust exécutera le test avant de s'arrêter automatiquement. C'est particulièrement pratique lorsque vous voulez simuler une charge sur une période précise sans avoir à surveiller manuellement l'interface pour arrêter le test.

- **Bouton de démarrage** (Start) : Une fois les paramètres définis, cliquez sur "Start" pour lancer le test.

ANALYSE DE RESULTATS

Après avoir mis 10s en run time et cliquez sur start, vous aurez un résultat similaire à :



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/comments	3	0	48	98	98	59.06	32	98	157745	0.33	0
GET	/posts	2	0	25.13	180	180	100.71	25	176	27520	0.17	0
	Aggregated	5	0	48	180	180	75.72	25	176	105655	0.5	0

Le tableau montre deux endpoints testés :

1. /comments : 3 requêtes
2. /posts : 2 requêtes

Aucun échec n'a été enregistré (# Fails = 0), ce qui montre que les endpoints testés ont répondu correctement sous la charge appliquée. La stabilité du système est satisfaisante pour ce scénario.

Temps de réponse :

Temps médian :

- /comments : 48 ms
- /posts : 25,13 ms

Ces valeurs indiquent que 50 % des requêtes ont un temps de réponse égal ou inférieur à ces chiffres, montrant une bonne rapidité globale.

Temps moyen (Average) :

- /comments : 59,06 ms
- /posts : 100,71 ms

Ces valeurs légèrement supérieures aux médianes montrent une petite variabilité, en particulier pour /posts.

95e et 99e percentiles :

- /comments : 98 ms pour les deux percentiles, montrant une certaine constance.
- /posts : 180 ms pour les deux percentiles, indiquant une dispersion plus importante, avec des requêtes plus lentes pour ce endpoint.

Taille moyenne des réponses :

Les tailles des réponses diffèrent considérablement :

- /comments : 157 745 octets (réponses volumineuses).
- /posts : 27 520 octets (taille beaucoup plus réduite).

Les réponses volumineuses pour /comments peuvent avoir un impact plus important sur les temps de réponse et la bande passante.

Débit:

Le débit global est de 0,5 requêtes par seconde (RPS). Cela correspond à une charge légère, adaptée pour un test exploratoire ou initial.

Les endpoints ont un débit faible :

- /comments : 0,33 requêtes/s
- /posts : 0,17 requêtes/s

Max et min temps de réponse :

/comments :

- Temps minimum : 32 ms
- Temps maximum : 98 ms

Ces valeurs sont proches de la médiane et des percentiles, montrant une bonne stabilité pour ce endpoint.

/posts :

- Temps minimum : 25 ms
- Temps maximum : 176 ms

L'écart plus important pour /posts indique une légère variabilité, possiblement liée à des opérations backend plus complexes ou à des ressources réseau.

GRAPHIQUES

Dans l'interface graphique, vous avez 3 graphiques, dont voici les deux plus intéressants pour notre test :

Total Requests per Second (RPS) - Requêtes par seconde :

- Le graphique du haut montre l'évolution du nombre de requêtes par seconde (RPS) au cours du test. Le graphique montre une montée progressive du RPS, qui atteint 1 requête par seconde à 15:23:24. Ce débit reste stable à son maximum avant de redescendre à 0,5 vers 15:23:28. Ce résultat montre que la charge de test appliquée est faible, avec une seule requête toutes les deux secondes en moyenne, ce qui est normal si le but est de tester des capacités basiques de réponse et de stabilité.
- L'absence de failures/s (échecs par seconde), indiquée par la ligne rouge plate à zéro, est un signe positif : le système n'a rencontré aucune défaillance lors de l'exécution de ces requêtes. C'est un bon indicateur de stabilité sous cette charge spécifique.

Response Times - Temps de réponse :

- Le graphique du bas détaille les temps de réponse en millisecondes. Deux percentiles sont affichés :
  - 50e percentile (jaune) : Indique le temps de réponse médian, soit 180 ms à 15:23:24, ce qui signifie que 50 % des requêtes ont un temps de réponse inférieur ou égal à cette valeur.
  - 95e percentile (violet) : Le temps de réponse au 95e percentile est également de 180 ms, ce qui signifie que 95 % des requêtes ont été traitées en 180 ms ou moins.

- Les 50e et 95e percentiles étant identiques, cela indique une faible variabilité dans les temps de réponse pour ce test. Cela suggère une stabilité dans le traitement des requêtes, avec peu ou pas de déviations majeures.

Users :

Le graphique User permet évaluer la montée en charge, la stabilité et la résilience de l'application testée. En combinant l'analyse de ce graphique avec les métriques de performance (temps de réponse, taux d'erreur), vous pouvez déterminer les limites actuelles de son système et prioriser les optimisations à apporter pour améliorer la capacité de charge.

Interprétation :

Bien que les temps de réponse soient stables, le temps médian de 180 ms peut être considéré comme élevé pour certains types de requêtes GET. Ce résultat pourrait indiquer :

1. Une latence légèrement accrue due à des opérations backend.
2. Une saturation mineure des ressources pour ces requêtes spécifiques.

UTILISATION AVANCEE DE LOCUST

Pour des scénarios plus complexes, Locust offre de nombreuses possibilités :

- Tests distribués : Vous pouvez lancer plusieurs travailleurs (workers) sur différentes machines pour distribuer la charge.

- Simuler différents types d'utilisateurs : Locust permet de créer plusieurs classes d'utilisateurs avec des comportements variés, afin de tester différents flux d'interaction dans votre application.

- Intégration continue : Vous pouvez intégrer Locust dans vos pipelines de CI/CD pour effectuer régulièrement des tests de charge, garantissant que les performances restent optimales au fur et à mesure des modifications.

Test distribué avec Locust

Locust permet de réaliser des tests distribués grâce à son architecture évolutive. L'outil fonctionne en mode maître/esclave, où le maître coordonne les tests et les esclaves génèrent la charge.

Pour configurer un test distribué avec Locust, vous devez définir plusieurs instances de Locust (esclaves) qui communiqueront avec l'instance principale (maître). Chaque esclave peut être sur une machine distincte, ce qui permet de simuler un grand nombre de personne sans saturer une seule machine.
Par exemple, pour démarrer un test distribué :

```
locust -f locustfile.py --master
```

Et sur chaque machine esclave :

```
locust -f locustfile.py --worker --
master-host=<IP_du_master>
```

Lors de la configuration de plusieurs machines, vérifiez que la montée en charge est bien répartie entre les personnes simulées par seconde. Une montée en charge trop rapide sur une seule machine peut entraîner des goulots d'étranglement.

K6

Introduction à K6

K6 est un outil de test de charge open-source conçu pour être simple à utiliser, mais suffisamment puissant pour réaliser des tests de performance avancés. Construit en Go, K6 permet d'exécuter des tests de charge fiables avec un minimum de configuration et un maximum de flexibilité. Son approche en ligne de commande et sa compatibilité avec les pipelines d'intégration continue en font un choix attractif pour les équipes de développement et de DevOps qui cherchent à intégrer des tests de performance dans leurs processus.

Pourquoi choisir K6 ?

L'un des grands atouts de K6 est sa facilité d'utilisation couplée à une grande efficacité. Contrairement aux outils traditionnels de test de charge, souvent lourds à configurer, K6 permet d'écrire des scripts en JavaScript, ce qui le rend immédiatement accessible aux développeurs familiers avec ce langage. Cette approche simplifie aussi le partage des scripts entre les équipes, facilitant la collaboration. K6 offre une interface claire et concise, permettant aux testeurs et développeurs de comprendre facilement les résultats.

Fonctionnalités Clés de K6

- K6 utilise JavaScript pour ses scripts, avec une API accessible pour définir les charges de travail, les seuils et les métriques. Un développeur peut facilement créer un script de test qui simule des personnes et des actions réalistes.
- K6 s'intègre parfaitement dans les environnements CI/CD via son interface en ligne de commande. Cela permet de déclencher des tests de charge lors de chaque déploiement pour identifier des régressions de performance avant que le code n'atteigne la production.
- Étant conçu en Go, K6 est capable de simuler un grand nombre de personnes tout en maintenant une consommation de ressources faible.
- K6 propose des visualisations en temps réel et des métriques détaillées, tant pour les ressources du système que pour les statistiques réseau (latence, temps de réponse, débit, etc.). En plus des rapports locaux, il peut également être intégré à des services externes comme Grafana pour une visualisation plus poussée.
- Une fonctionnalité intéressante de K6 est la possibilité de définir des seuils sur les métriques. Par exemple, si le taux de succès d'une requête passe en dessous d'un certain pourcentage, K6 peut automatiquement déclencher une alerte et stopper le test. Cela permet de s'assurer que les performances respectent toujours les critères définis en amont.

Cas d'utilisation de K6 dans un contexte d'entreprise

K6 est utilisé par des entreprises de toutes tailles pour réaliser des tests de charge dans des environnements variés, allant des sites web aux microservices complexes. Voici quelques scénarios typiques d'utilisation :

- Validation des performances applicatives avant mise en production : K6 permet de simuler des charges importantes pour vérifier que les applications peuvent gérer le volume de personnes prévu.

- Tests de charge pour les APIs : Avec K6, les entreprises peuvent tester leurs APIs pour s'assurer que les réponses sont rapides et fiables, même sous une forte charge.

- Intégration avec des outils d'observabilité : Lorsqu'il est combiné avec des solutions comme Grafana, K6 fournit une vue complète sur les performances et les goulots d'étranglement.

Limites de K6

Malgré ses nombreux avantages, K6 présente également quelques limites. Par exemple, il est uniquement en ligne de commande, ce qui peut être un obstacle pour les équipes habituées aux interfaces graphiques. De plus, bien que K6 soit open-source, les fonctionnalités avancées (comme les tests sur le cloud) sont accessibles via la version payante, ce qui peut limiter l'utilisation pour certaines entreprises au budget restreint.

INSTALLATION DE K6

Assurez-vous que K6 est installé sur votre machine. Vous pouvez l'installer en suivant les instructions sur k6.io.

CREATION D'UN TEST DE BASE

Voici un exemple de script en JavaScript pour K6 :

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '10s', target: 1 },
    { duration: '5s', target: 1 },
    { duration: '5s', target: 0 },
  ],
};

export default function () {
  let posts = http.get('https://jsonplaceholder.typicode.com/posts');
  check(posts, {
    'status is 200': (r) => r.status === 200,
    'response time < 200ms': (r) => r.timings.duration < 200,
  });

  let comments =
http.get('https://jsonplaceholder.typicode.com/comments');
  check(comments, {
    'status is 200': (r) => r.status === 200,
    'response time < 200ms': (r) => r.timings.duration < 200,
  });

  let users = http.get('https://jsonplaceholder.typicode.com/users');
  check(users, {
    'status is 200': (r) => r.status === 200,
    'response time < 200ms': (r) => r.timings.duration < 200,
  });

  sleep(1);
}
```

Explication du Script

- Configuration des Stages : La section options définit trois étapes de montée en charge :
  - Une montée en charge progressive pendant 1 minute pour atteindre 10 personnes.
  - Une charge stable pendant 3 minutes avec 10 personnes.
  - Une descente progressive à personne en 1 minute.

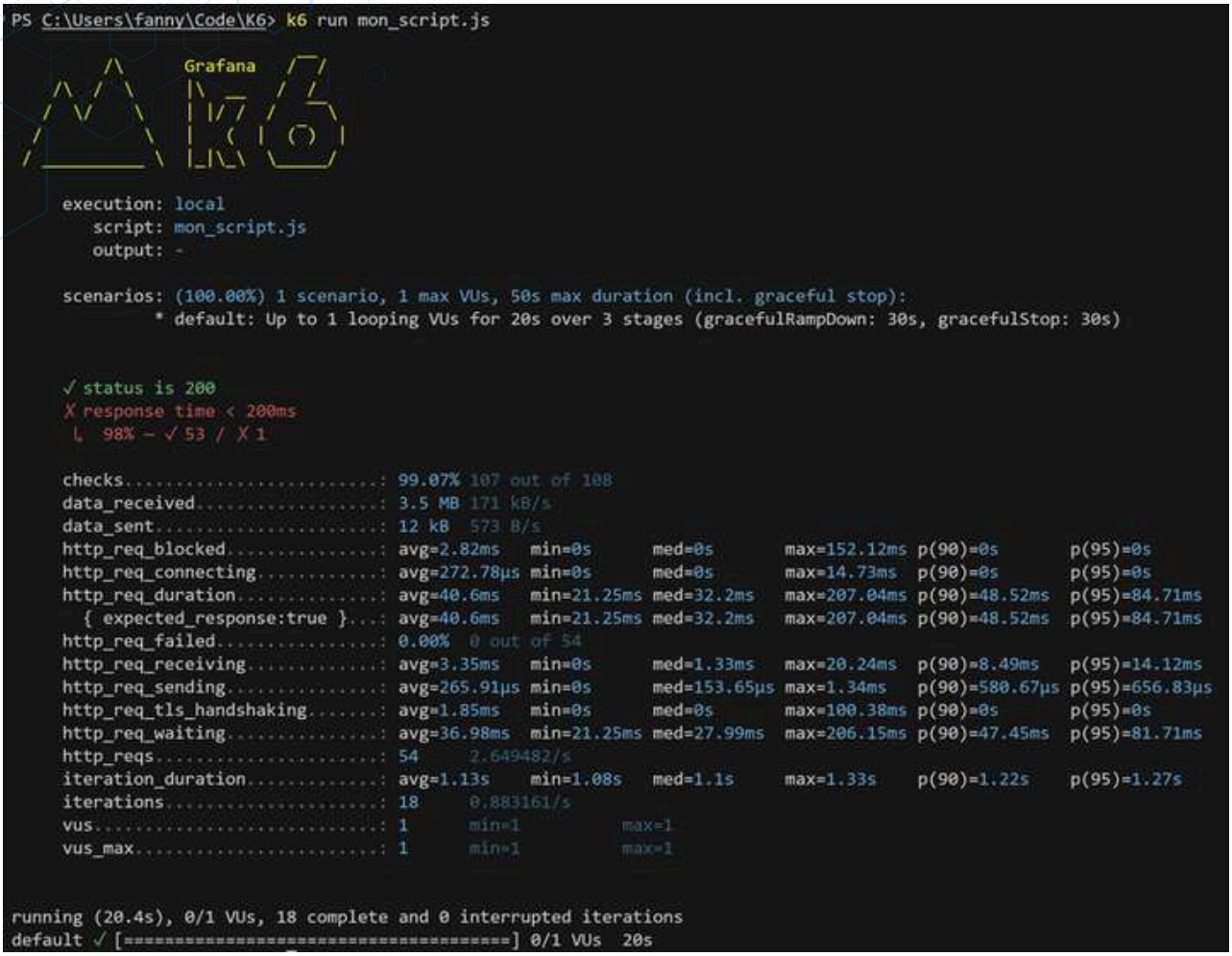
- Requêtes et Vérifications :
  - Pour chaque endpoint (/posts, /comments, /users), une requête GET est envoyée, et des vérifications sont effectuées pour s'assurer que le code de réponse est 200 et que le temps de réponse est inférieur à 200 ms.

- Simuler une Pause : sleep(1); introduit une pause d'une seconde pour simuler un comportement humain réaliste entre les requêtes.

Lancer le test

Pour exécuter ce script, placez-le dans un fichier nommé mon_script.js et lancez-le avec la commande suivante dans votre terminal :

```
k6 run mon_script.js
```



Statut HTTP :

Le test montre que la majorité des requêtes ont obtenu un code de statut 200 (✓ status is 200), ce qui indique que les requêtes ont réussi.

Cependant, on voit également que la condition "temps de réponse inférieur à 200ms" n'est pas toujours respectée (X response time < 200ms), avec 53 échecs pour cette condition sur un total de 54 itérations. Cela suggère que le temps de réponse est parfois trop élevé.

Métriques générales :

Data reçue : 3.5 MB de données ont été reçues avec une moyenne de 171 kB/s.

Data envoyée : 12 kB ont été envoyés à un débit moyen de 573 B/s.

Ces valeurs montrent que le test a simulé un certain volume de trafic, ce qui est utile pour évaluer la bande passante consommée.

Détails sur les requêtes HTTP :

- http_req_duration : C'est la durée totale des requêtes, de la connexion initiale à la réponse finale. La moyenne est de 40.6ms, mais on observe une valeur maximale élevée de 207.04ms, avec les percentiles 90 (p(90)) et 95 (p(95)) proches de 48ms et 85ms respectivement.

- http_req_blocked : Il s'agit du temps pendant lequel la requête a été bloquée, avec une moyenne de 2.82ms. Cela peut inclure le temps d'attente pour établir une connexion. La valeur maximum est à 8.52ms, ce qui est assez raisonnable.

- http_req_connecting : Ce champ montre le temps pris pour établir la connexion TCP, ici 272.78µs en moyenne, ce qui est négligeable par rapport au reste.

Autres métriques HTTP :

- http_req_waiting : C'est le temps d'attente de la réponse du serveur après l'envoi de la requête. La moyenne est de 36.98ms, ce qui est globalement bon, mais le maximum (206.15ms) montre que certains appels ont pris plus de temps que d'autres.

- http_req_receiving : Temps pour recevoir la réponse, avec une moyenne de 3.35ms, maximum à 20.24ms. C'est rapide, indiquant que le téléchargement des réponses n'est pas un point de lenteur ici.

Iterations et Scénario :

- Iterations : 18 itérations ont été réalisées dans ce test, avec une durée moyenne par itération de 1.13s.

- vus (virtual users) : Le test a utilisé 1 utilisateur virtuel (VU), avec un maximum de 1 VU, ce qui indique que c'était un test léger.

Analyse des performances

- Temps de réponse : Bien que les temps moyens de requête soient relativement bas, la contrainte de moins de 200ms pour toutes les requêtes n'a pas été respectée, ce qui indique que, dans des conditions de charge plus élevées ou dans des cas spécifiques, l'application pourrait avoir des goulots d'étranglement.

- Stabilité : La stabilité du test semble bonne, avec très peu d'échecs (0%) et des valeurs de percentiles raisonnables, mais il faudrait probablement augmenter la charge (plus de VUs) pour voir comment l'application se comporte sous un stress plus intense.

FAVORITE

JMETER

Introduction à JMeter

JMeter est un outil de test de charge mature et populaire, développé par la Fondation Apache. Initialement conçu pour tester des applications web, JMeter prend désormais en charge une large gamme de protocoles, y compris HTTP, FTP, SMTP, et plus encore. JMeter est souvent le choix privilégié pour les tests de charge et de stress en raison de sa robustesse et de ses fonctionnalités avancées.

Pourquoi choisir JMeter ?

JMeter se distingue par sa capacité à tester différents types d'applications grâce à sa prise en charge de multiples protocoles, en plus d'être extensible avec de nombreux plugins. C'est un outil bien établi et éprouvé, ce qui signifie qu'il est bien documenté et dispose d'une grande communauté. Il propose également une interface graphique pour la création de tests, ce qui peut être un avantage pour les équipes qui préfèrent une approche visuelle.

Fonctionnalités Clés de JMeter

- Support multi-protocole : JMeter permet de tester non seulement des applications web, mais aussi des applications FTP, des serveurs de messagerie, des bases de données, etc., ce qui le rend extrêmement polyvalent.

- Interface graphique complète : JMeter offre une interface graphique qui simplifie la création de tests pour les personnes non-développeuse. Elle permet de visualiser chaque étape d'un test, de configurer facilement les éléments de test et de surveiller les résultats.

- Fonctionnalités de Reporting : JMeter génère des rapports détaillés et permet de visualiser les résultats sous forme de graphiques, ce qui aide à analyser les performances et identifier les goulots d'étranglement.

- Extensions via des plugins : La communauté de JMeter propose une large gamme de plugins pour étendre ses capacités et l'adapter aux besoins spécifiques des tests.

Cas d'utilisation de JMeter en entreprise

- Tests de charge d'applications web et d'APIs : JMeter est idéal pour tester les performances d'applications web et des APIs REST ou SOAP.
- Validation de services diversifiés : Avec sa capacité multi-protocole, JMeter est particulièrement adapté aux environnements complexes qui nécessitent des tests pour différents services comme les bases de données ou les serveurs de fichiers.
- Tests pour des environnements d'intégration continue : Grâce à ses fonctionnalités avancées de reporting et de ligne de commande, JMeter s'intègre bien dans les pipelines CI/CD.

Limites de JMeter

La nature basée sur Java de JMeter le rend parfois gourmand en ressources, en particulier pour des charges importantes. Bien que son interface graphique soit un avantage pour certains, elle peut être lente lors des gros tests et moins flexible que les solutions basées sur le code.

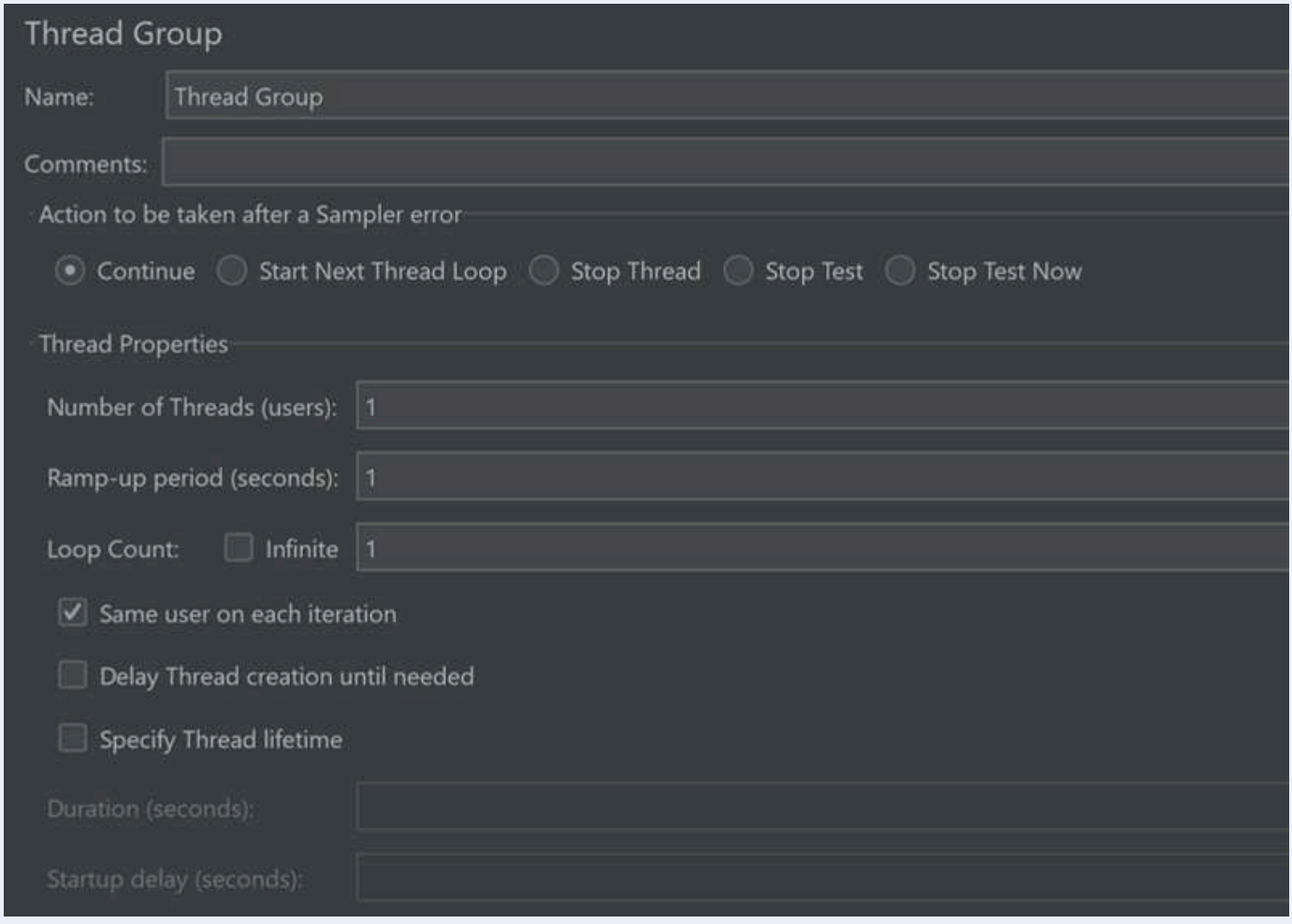
INSTALLATION DE JMETER

Assurez-vous que Jmeter est installé sur votre machine. Vous pouvez l'installer en téléchargeant le binaire sur :
https://jmeter.apache.org/download_jmeter.cgi
Puis exécuter le fichier jmeter.bat qui se trouve dans le dossier bin. L'interface GUI apparaitra quelques secondes après.

CREATION D'UN TEST DE BASE

1. Créez un nouveau plan de test :
   - Cliquez sur « File > New ».
   - Renommez le plan de test en cliquant avec le bouton droit sur « Test Plan » et en sélectionnant « Rename ».
2. Ajoutez un groupe d'utilisateurs virtuels :
   - Cliquez avec le bouton droit sur le plan de test, puis « Add > Threads (Users) > Thread Group ».
   - Configurez le nombre de personne, la durée du test et le ramp-up time selon vos besoins.

Le mode GUI ne doit être utilisé que pour créer le script de test, le mode CLI (NON GUI) doit être utilisé pour les tests de charge



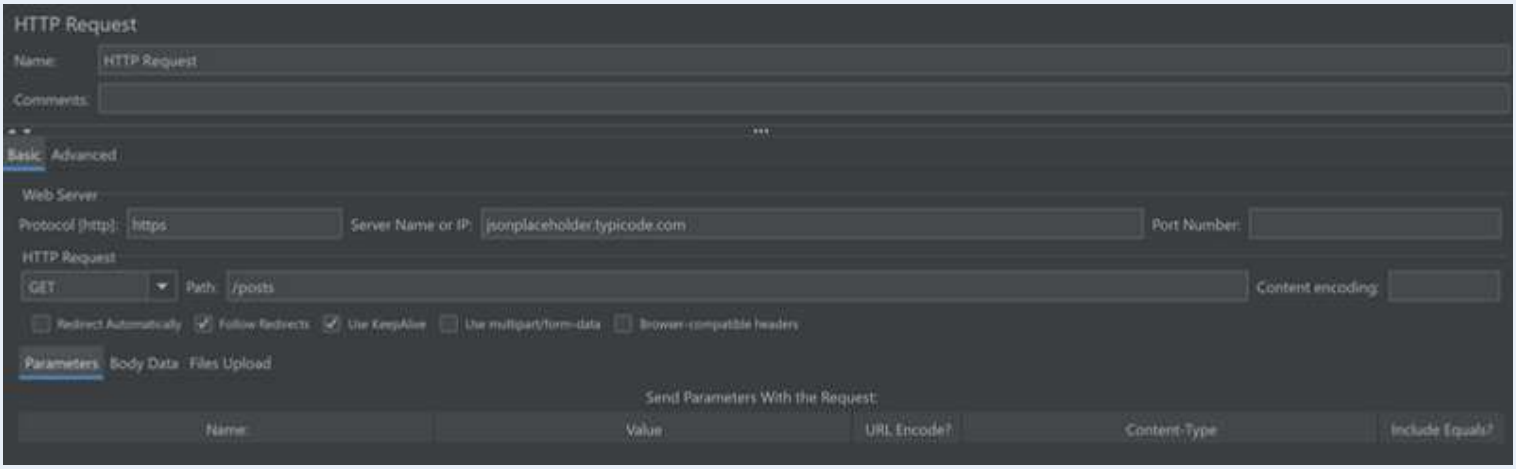
Configurer la requête HTTP

1. Ajoutez un requêteur HTTP :
   - Cliquez avec le bouton droit sur le groupe d'utilisateurs, puis « Add > Sampler > HTTP Request ».
2. Configurez les détails de la requête :
   - Serveur : jsonplaceholder.typicode.com
   - Méthode : GET (ou POST si vous voulez tester des écritures).
   - Chemin : /posts (ou tout autre endpoint de l'API).
3. Ajoutez des paramètres si nécessaire (par exemple, pour filtrer les posts via userId).

Ajout d'un listener pour les résultats

Ajoutez un écouteur pour voir les résultats des tests :

- Cliquez avec le bouton droit sur le groupe d'utilisateurs, puis « Add > Listener > View Results Tree ».
- Vous pouvez également ajouter d'autres écouteurs, comme « Summary Report » ou « Aggregate Report ».



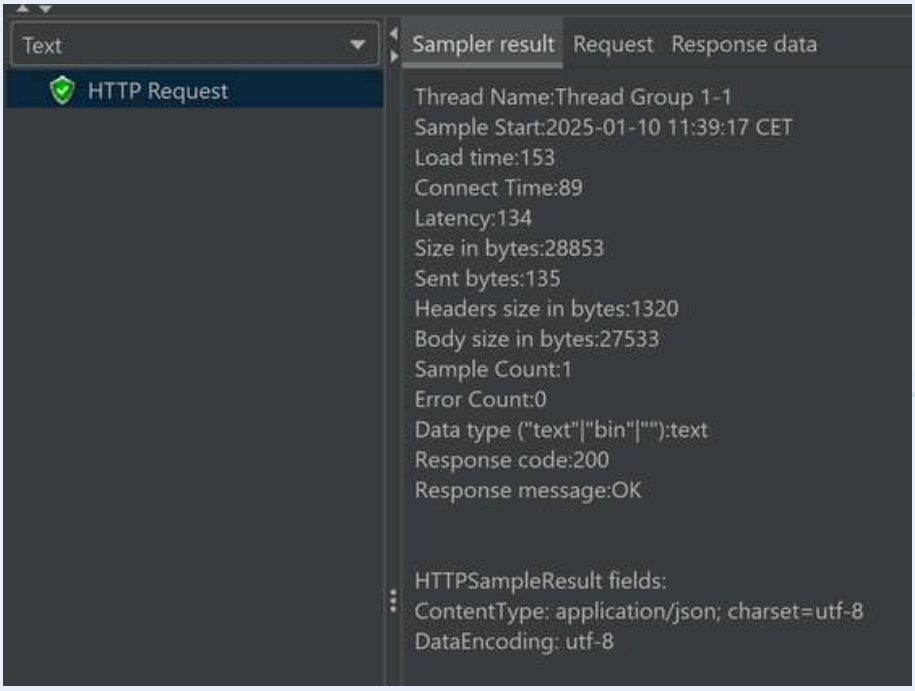
Lancer le test

1. Lancez le test en cliquant sur le bouton Run (flèche verte dans la barre d'outils).
2. Sauvegardez votre test
3. Observez les résultats dans les écouteurs configurés.

Analyse des résultats

- View Results Tree : Permet de vérifier les détails de chaque requête et réponse.
- Summary Report : Donne un aperçu des performances globales, notamment le temps moyen de réponse, le nombre d'échecs, etc.

ANALYSE DES RESULTATS



Détails techniques observés :

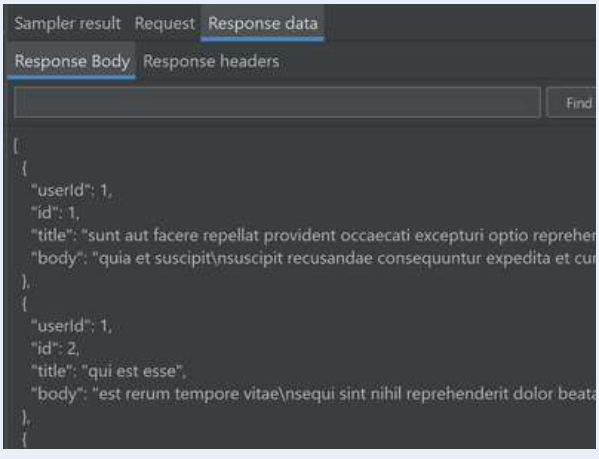
- Load time (153 ms) : C'est le temps total pris pour exécuter la requête et recevoir une réponse complète. Ce temps semble raisonnable pour une requête simple sur une API REST.
- Connect Time (89 ms) : Le temps pris pour établir la connexion avec le serveur. Un temps inférieur à 100 ms est généralement bon, surtout si le serveur est distant.
- Latency (134 ms) : Le temps entre l'envoi de la requête et le début de la réception des données. Ce temps est légèrement plus élevé que le connect time, ce qui est logique.
- Body size in bytes (27533) : La taille du corps de la réponse en octets. Cela indique que vous avez reçu une réponse conséquente, probablement un tableau JSON contenant plusieurs objets.
- Response Code (200) : Le code HTTP 200 signifie que la requête a été exécutée avec succès.
- Content-Type (application/json; charset=utf-8) : La réponse est au format JSON avec un encodage UTF-8. Cela confirme que la réponse est bien formatée pour des données structurées.

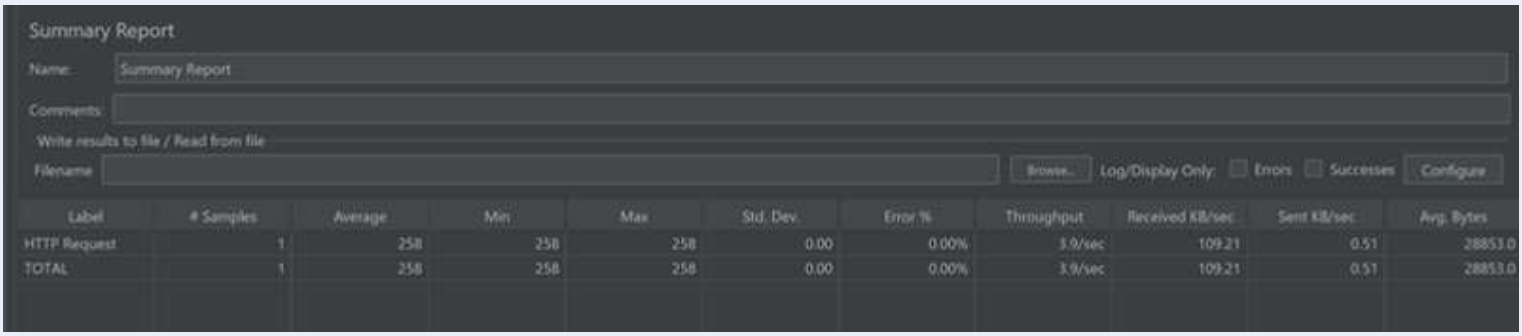
Analyse des performances :

- Performance acceptable : Les temps (load time, connect time et latency) sont tous dans des limites normales pour un test d'une API publique comme JSONPlaceholder.

- Réponse correcte : Le code HTTP 200 et le format JSON confirment que l'API fonctionne correctement et renvoie les données attendues.

- Taille de la réponse (Body size) : La taille de 27 533 octets montre que la réponse contient probablement de nombreuses données (un tableau de posts en JSON, typique pour /posts). Vous pouvez voir les données dans l'onglet response Data :





Résumé des résultats :

- # Samples : 1
  - Cela signifie que vous avez exécuté une seule requête HTTP (un échantillon unique).
  - Cela est cohérent pour un test de base ou un test initial.

- Average, Min, Max : 258 ms
  - Ces trois valeurs étant identiques (258 ms) indiquent que vous n'avez qu'une seule requête dans ce test, donc pas de variation dans les temps de réponse.
  - Temps de réponse moyen (258 ms) : Cela est un temps raisonnable pour une API publique.

- Error % : 0.00%
  - Aucune erreur n'a été rencontrée lors de l'exécution du test. C'est une bonne indication que la configuration est correcte et que le serveur a bien répondu.

- Throughput : 3.9/sec
  - Cette métrique indique que votre système pourrait gérer environ 3,9 requêtes par seconde dans les conditions actuelles.
  - Avec une seule requête exécutée, ce chiffre reflète le temps entre la requête et la réponse.

- Received KB/sec : 109.21 KB/sec
  - La vitesse à laquelle les données de réponse ont été reçues.
  - Cela dépend de la taille de la réponse (27 533 bytes pour une requête), et c'est cohérent pour une réponse JSON volumineuse.

- Sent KB/sec : 0.51 KB/sec
  - La vitesse des données envoyées. Cela est faible car la requête GET n'inclut que des headers et très peu de données.
- Avg. Bytes : 28 853.0
  - Taille moyenne de la réponse (en bytes) pour chaque requête. Cela correspond bien à la taille de la réponse JSON de l'API /posts.

Vous pouvez également ajouter d'autres listeners : aggregate report ou graph results pour visualiser les performances globales et l'évolution du temps de réponse

ET LES AUTRES ?

Outil	Simplicité	Scénarios complexes	Échelle	Protocoles multiples	Idéal pour
Apache Bench	★★★	✕	★★	✕	Tests de base rapide HTTP
Artillery.io	★★	★★★	★★★	★	Scénarios complexes APIs
Drill	★★	★★	★★	✕	Tests simples et basiques
Hey	★★★	✕	★★	✕	Tests rapides HTTP
Siege	★★	★★	★★	★	Tests de charge web
Tsung	★	★★★	★★★	★★★	Tests avancés multi-protocoles
Wrk	★★★	★	★★★	✕	Tests rapides et massifs HTTP

COMPARAISON DES OUTILS PAYANTS

	NeoLoad	LoadRunner	OctoPerf 
Protocoles supportés	Prise en charge de nombreux protocoles : HTTP, WebSocket, SOAP, REST, etc.	Large éventail de protocoles, notamment HTTP, SAP, Oracle, Citrix, etc.	Principalement orienté web (HTTP/HTTPS), WebSocket, REST.
Facilité d'utilisation	Interface intuitive avec une courbe d'apprentissage rapide pour les nouveaux utilisateurs.	Interface riche mais complexe, nécessitant une courbe d'apprentissage plus longue.	Plateforme SaaS avec une interface simple et intuitive, facile pour les débutants.
Flexibilité	Grande capacité de personnalisation des scénarios avec des scripts Dynatrace ou API.	Très flexible avec des langages de script comme C, JavaScript.	Utilisation basée sur JMeter avec des options prêtes à l'emploi pour des scénarios complexes.
Intégration CI/CD	Compatible avec Jenkins, Bamboo, GitLab CI/CD, et autres outils DevOps.	Bien intégré avec des outils DevOps comme Jenkins, GitLab, Azure DevOps.	Bien intégré avec Jenkins, GitLab et API REST.
Scalabilité	Supporte des tests massifs avec des milliers d'utilisateurs virtuels.	Excellente scalabilité, notamment pour des tests complexes avec des centaines de milliers d'utilisateurs.	Très scalable avec la possibilité d'exécuter des tests à partir de plusieurs régions géographiques.
Rapports	Rapports détaillés et interactifs avec des graphiques personnalisables.	Rapports approfondis avec des capacités avancées d'analyse.	Rapports clairs avec des options pour partager directement avec les parties prenantes.
Coût	Licence coûteuse mais adaptable au besoin (modèle basé sur les utilisateurs virtuels).	Coût élevé, souvent justifié pour les grandes entreprises ayant des besoins complexes.	Modèle de prix compétitif et transparent, basé sur l'utilisation et les tests.

Voici une analyse détaillée des outils présentés.

FAVORITE

OCTOPERF

OctoPerf est un outil de test de performance conçu pour permettre aux équipes de réaliser des tests de charge et de performance de manière intuitive, rapide et rentable. Il se distingue par son interface conviviale et son intégration avec des technologies modernes pour aider à évaluer la robustesse des applications et des systèmes.

De plus, leur objectif initial était de rendre les tests de charge accessibles à toutes les équipes, même celles qui n'ont pas d'expertise technique avancée, tout en gardant des performances comparables aux solutions plus coûteuses et complexes comme LoadRunner.

Elle a été fondée en 2014 par une équipe d'ingénieurs spécialisés dans les tests de performance. Le siège social de l'entreprise est situé à Aubagne, dans le sud de la France.

Fonctionnalités principales

- OctoPerf est souvent perçu comme un équivalent technique et performant à NeoLoad, tout en offrant une interface moderne et intuitive

- Création de scénarios de test sans codage :
  - Interface graphique permettant de concevoir facilement des scénarios.
  - Éditeur de scénarios avec enregistrement des actions utilisateur

- Bien que l'outil évite de recourir au Test as Code, OctoPerf offre également la possibilité d'enrichir ses scripts via du code pour répondre aux besoins des équipes techniques.

- Simulation réaliste des charges :
  - Génération de trafic avec des utilisateurs virtuels depuis le cloud ou en local. OctoPerf a démontré sa capacité à gérer jusqu'à 1 million d'utilisateurs virtuels (VU) simultanés dans un environnement SaaS.
  - Prise en charge de différents protocoles (HTTP, HTTPS, WebSocket, etc.).

- Analyse et reporting :
  - Tableaux de bord interactifs pour suivre les performances en temps réel.
  - Export des résultats sous divers formats (PDF, Excel, etc.).
  - Identification rapide des goulots d'étranglement.

- Intégration continue et API REST :
  - Connexions aux pipelines CI/CD (Jenkins, GitLab, Azure DevOps, etc.).
  - Utilisation de l'API REST pour automatiser les tests.

- Mode SaaS et On-Premise : déploiement flexible pour les tests cloud ou internes (self-hosted). Cette option permet aux entreprises ayant des exigences de sécurité élevées, comme les banques, les assurances ou les administrations, de travailler dans un environnement entièrement cloisonné et sécurisé

- Support des tests sur des applications modernes :
  - Prise en charge de SPA (Single Page Applications) et applications avec AJAX.
  - Paramétrage avancé pour la gestion des données dynamiques (correlation, variables).

- OctoPerf propose une fonctionnalité pratique de capture de chemin (Path) et d'importation de fichiers .har. Cela permet aux équipes de gagner du temps dans la préparation des scénarios en utilisant directement les données capturées lors de sessions utilisateur réelles.

Avantages

- Facilité d'utilisation :
  - Moins technique que certains outils comme JMeter.
  - Idéal pour les équipes souhaitant une solution rapide à mettre en œuvre.

- Coût compétitif :
  - Prix ajustable en fonction des besoins (paiement à la demande ou forfaits).
  - Moins coûteux que d'autres outils comparables pour des tests cloud.

- Flexibilité :
  - Possibilité de combiner des charges locales et cloud.
  - Tests adaptables pour divers protocoles et environnements.

- Rapidité de mise en œuvre :
  - Mise en place rapide grâce à des configurations prédéfinies.
  - L'enregistrement des scénarios réduit le temps de préparation.

- Support technique de qualité :
  - Assistance réactive avec une documentation bien fournie.

- Version On-Premise, identique d'un point de vue UI à la SaaS, permet à aux clients ayant des contraintes de sécurité élevées (Banques / Assurances / Administrations etc...) de travailler dans un environnement parfaitement clos et sécurisé

Le fait qu'OctoPerf soit une solution française est également un atout pour les entreprises soucieuses de collaborer avec des fournisseurs européens, en particulier pour des questions de conformité RGPD et de support local.

Limitations

- Capacités avancées limitées : bien qu'il soit possible d'enrichir les scripts avec du code, OctoPerf n'est pas entièrement orienté vers le Test as Code, ce qui peut ne pas convenir aux équipes privilégiant cette approche pour des scénarios très complexes. D'autant plus que les scripts peuvent augmenter le poids des utilisateurs virtuels. Mais si vous devez écrire des scripts, privilégiez le Groovy car c'est le seul langage qui peut être compilé dans JMeter.

Cas d'utilisation

- Tests de performance web : vérifier la capacité de charge des sites web e-commerce, plateformes SaaS ou portails publics.

- Test des SAP (Single Page Applications) et API REST.

- Simulation de scénarios réalistes en évaluant des performances dans des conditions proches de la réalité avec des utilisateurs globaux.

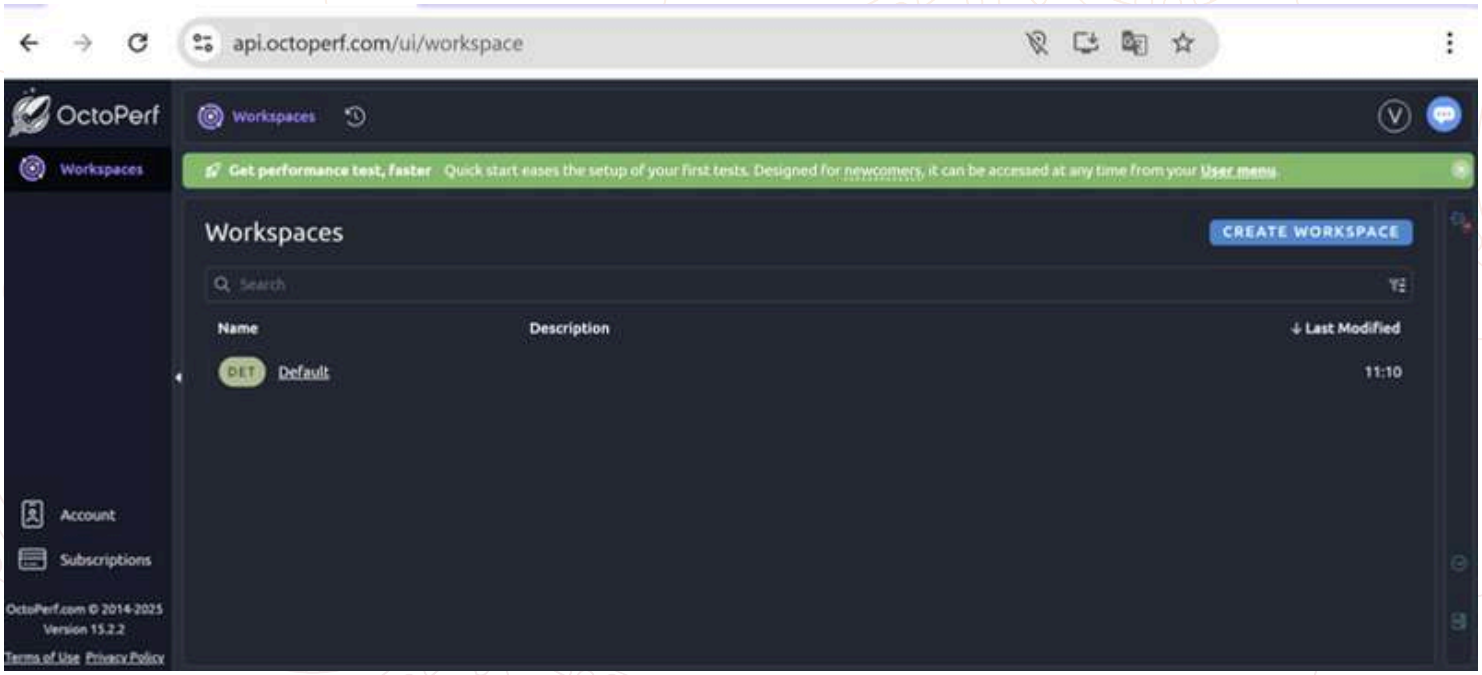
- Approches DevOps et CI/CD en automatisant des tests de charge dans les pipelines de déploiement continu.

- Tests de charge ponctuels : idéal pour les entreprises qui ne réalisent pas de tests de charge régulièrement, grâce à la facturation à l'usage.

La création de compte sur OctoPerf est simple et rapide, sans nécessiter de carte bancaire. L'équipe commerciale se distingue par sa disponibilité et son professionnalisme, avec des canaux de contact efficaces (email et LinkedIn). Ils sont également prêts à organiser une démonstration personnalisée pour comprendre les besoins spécifiques de votre équipe et vous accompagner dans votre essai.

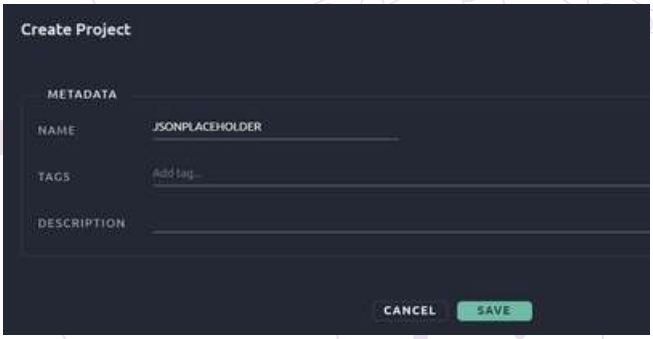
CONNEXION

- Rendez-vous sur le site: https://octoperf.com.
- Inscrivez-vous ou connectez-vous avec vos identifiants.
- Une fois connecté, vous arrivez sur le tableau de bord.



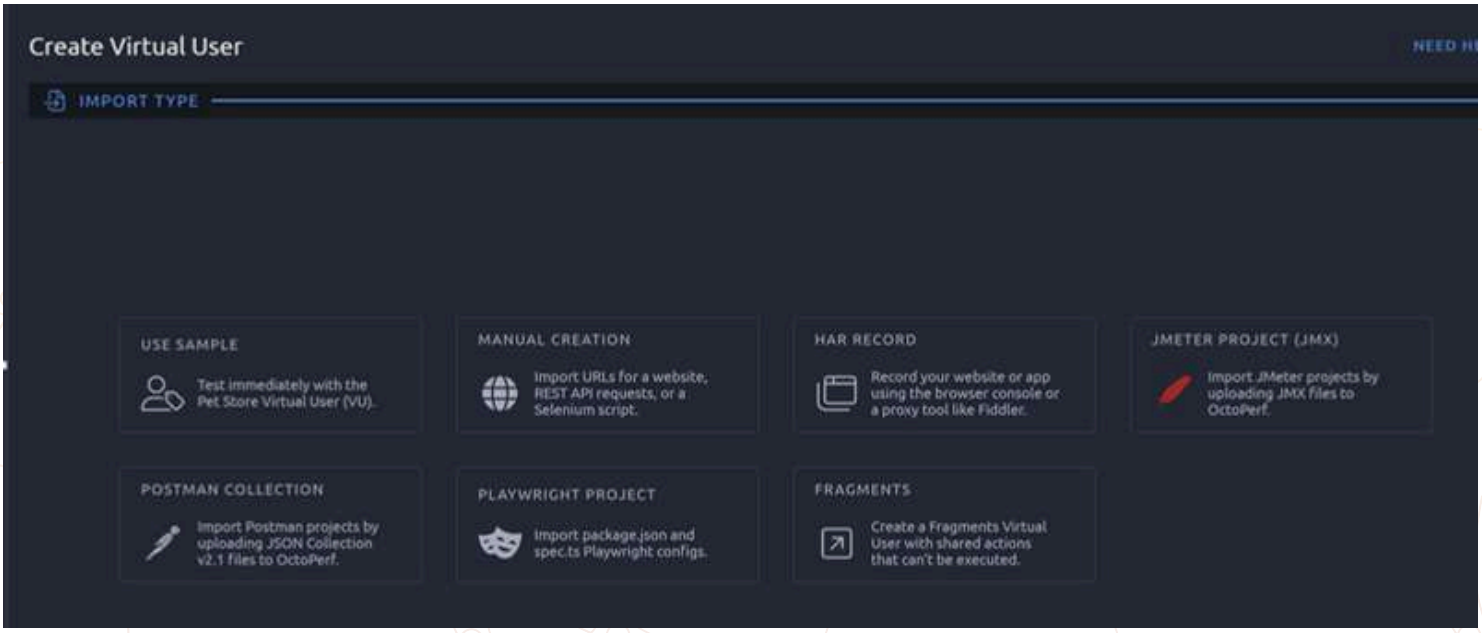
Créez un nouveau workspace:

1. Cliquez sur le bouton "Create workspace".
2. Donnez un nom à votre workspace (par exemple : "Test de performance API").
3. Ajoutez une description (facultatif) et validez.



Créez un nouveau projet:

1. Cliquez sur le bouton "Create project".
2. Donnez un nom à votre projet (par exemple : "Test de JSONPLACEHOLDER").
3. Ajoutez une description (facultatif) et validez.



Choix de la création d'utilisateur virtuel :

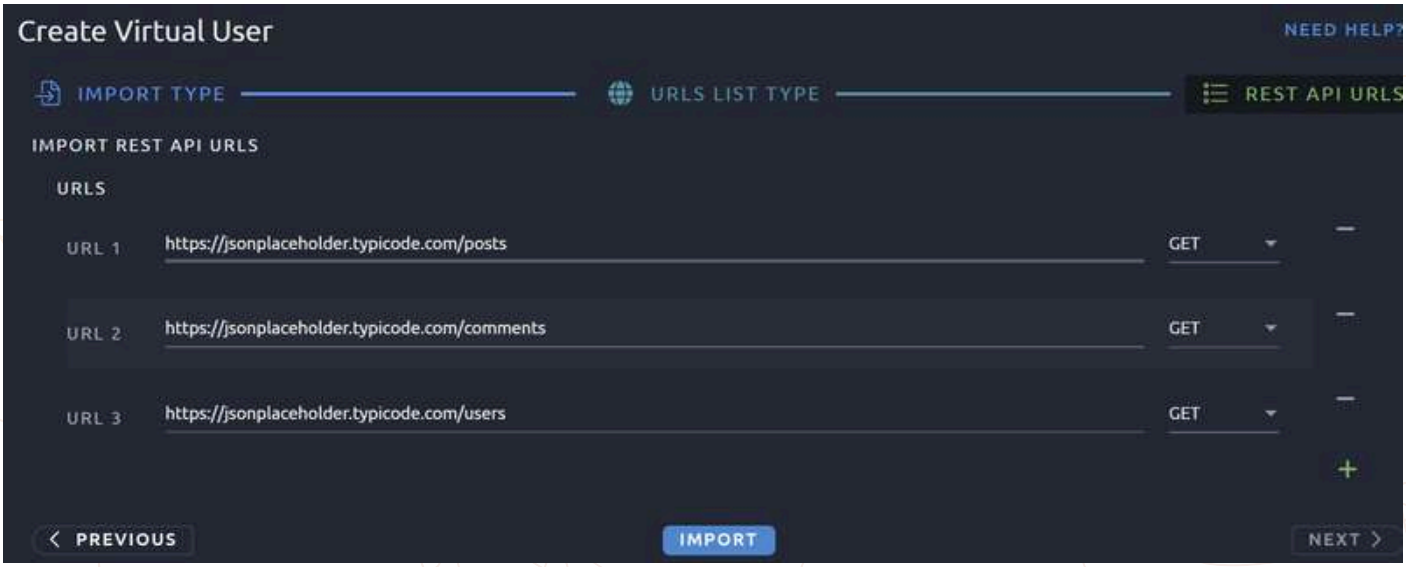
- Use sample : si vous débutez, cette option permet d'utiliser un exemple prédéfini pour vous familiariser avec l'outil.
- Manual creation : idéal si vous souhaitez créer un scénario en ajoutant manuellement des requêtes HTTP, REST API ou des scripts Selenium.
- HAR Record : si vous avez déjà enregistré vos interactions avec l'application (par exemple via Fiddler ou les outils réseau du navigateur), vous pouvez importer un fichier HAR pour générer automatiquement un scénario.
- JMeter project (JMX) : si vous avez un projet JMeter existant, importez le fichier .jmx.
- Postman collection : si vous utilisez Postman, vous pouvez importer une collection JSON pour tester vos API.
- Playwright project : si vous avez un projet Playwright, cette option vous permet d'importer vos configurations.
- Fragments : permet de créer des actions partagées ou réutilisables pour d'autres scénarios.

CREATION D'UN TEST DE BASE

Pour notre test, prenez "manual creation" pour définir les requêtes API manuellement.

Puis, choisissez "Rest API", cette option est spécifiquement conçue pour tester les requêtes HTTP/HTTPS avec les méthodes GET, POST, PUT ou DELETE;

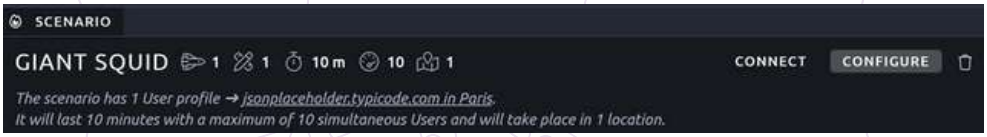
Ajoutez les requêtes HTTP :



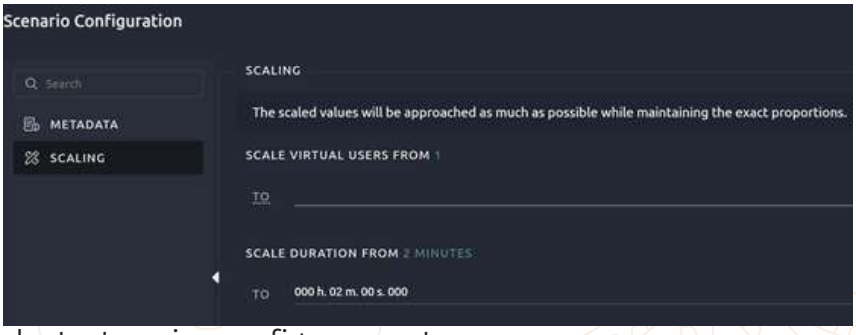
Puis cliquez sur import, puis selectionnez votre virtual user.
Choisissez votre emplacement :

Cliquez sur run validation, un bouton launch test apparait. Cliquez dessus

Par défaut, le test va se lancer sur 10min pour un utilisateur. Vous pouvez changer cette configuration en cliquant sur le bouton “Configure” puis scaling

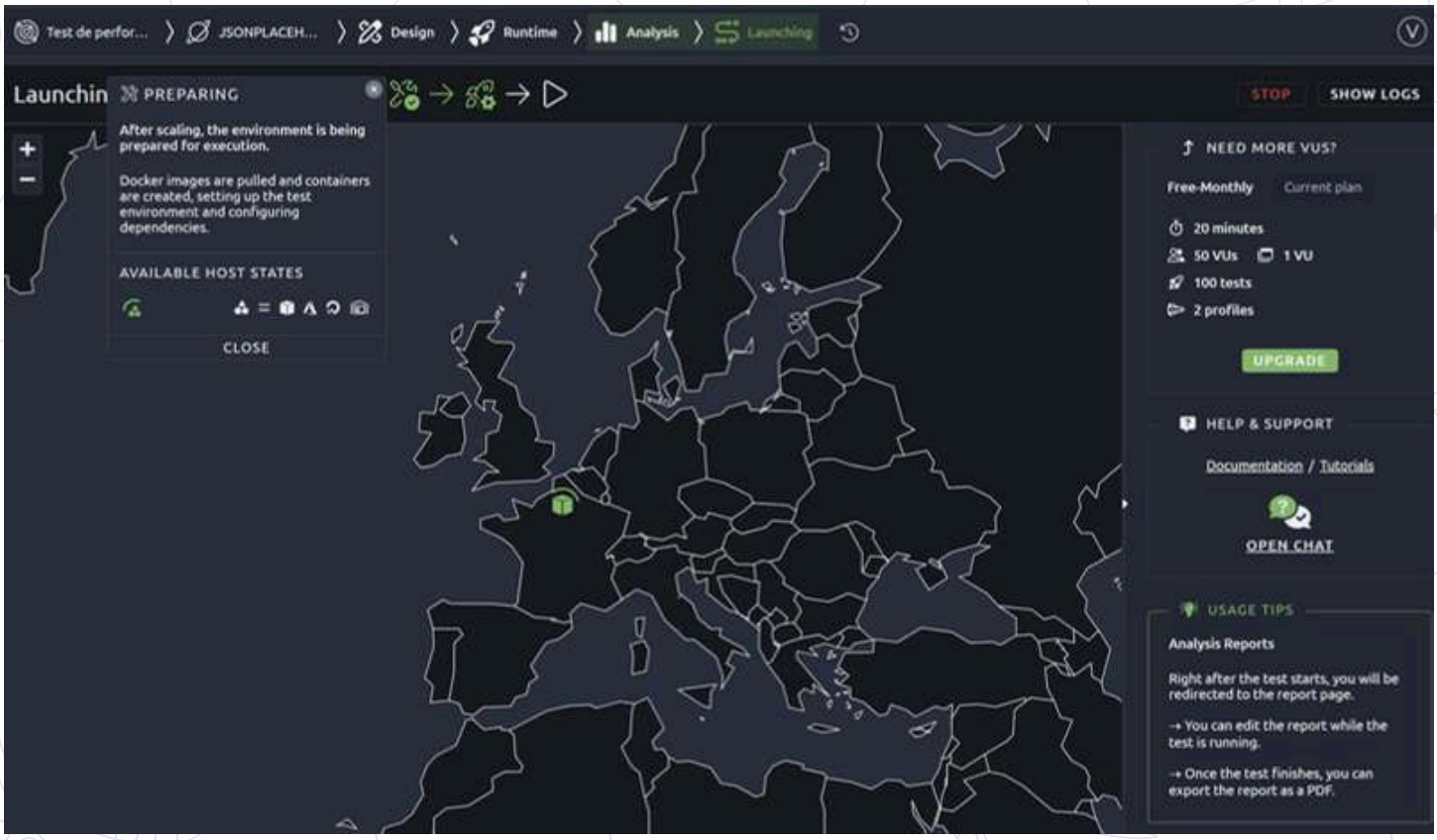


Pour le test, nous allons utiliser un utilisateur sur 2 min.

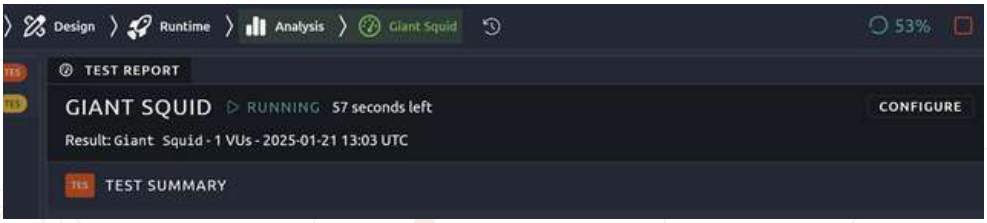


Cliquez sur launch test puis configurer votre rapport :

Lancer le test tout de suite. Vous pouvez également le planifier si vous avez besoin.



Vous pouvez voir la progression de l'exécution :



Les graphiques sont bien plus complets

Hit count (Nombre d'exécutions) : 117

- Cela indique que 117 requêtes ont été effectuées pendant le test.

Error percentage (Pourcentage d'erreurs) : 0%

- Ce pourcentage confirme qu'il n'y a eu aucune requête échouée.

Latency standard deviation (Écart-type de la latence) : 0.008 secondes

- L'écart-type de la latence est très faible (8 ms), ce qui indique une grande stabilité des temps de réponse.
- Cela signifie que les temps de réponse sont homogènes et qu'il n'y a pas eu de pics ou de lenteurs significatives.

Error count (Nombre d'erreurs) : 0

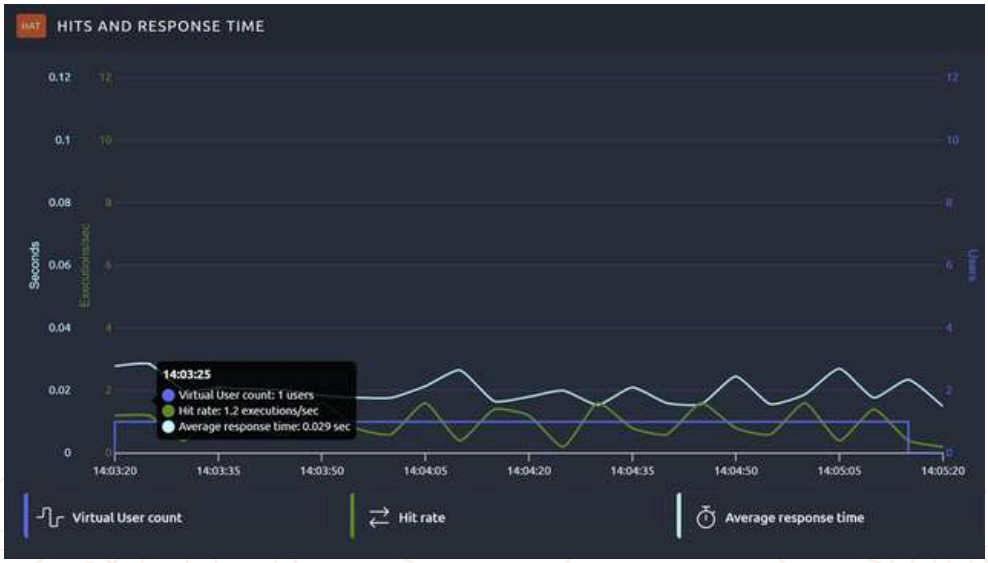
- Aucun échec n'a été détecté parmi les 117 requêtes.
- Cela montre que toutes les requêtes ont reçu une réponse correcte du serveur
- Cela confirme une bonne stabilité du système ou de l'API testée.

Average response time (Temps de réponse moyen) : 0.02 secondes

- Le temps moyen de réponse est 20 ms, ce qui est extrêmement rapide.

Received data rate (Taux de données reçues) : 61.9 KB/sec

- Cela correspond au débit moyen des données téléchargées depuis le serveur.
- Une valeur relativement faible (ce qui est attendu pour une API REST qui ne renvoie probablement que des données JSON).



Virtual user count (Nombre d'utilisateurs virtuels, violet) :

- Il n'y a bien qu'un seul utilisateur

Hit Rate (Taux d'exécutions, vert) :

- En moyenne, environ 1,2 exécutions par seconde ont été réalisées.
- Cette métrique est stable avec de légères variations.
- Cela montre que le serveur peut traiter les requêtes en continu à un rythme constant.

Average response time (Temps de réponse moyen, cyan) :

- Le temps de réponse est très stable, oscillant autour de 0,02 à 0,03 secondes.
- Cela reflète une bonne performance du serveur sous cette charge faible.
- Aucune hausse significative du temps de réponse n'est visible, même pendant les pics d'exécutions.

Resp time avg (Average Response Time), temps de réponse moyen pour chaque endpoint (en secondes).

- Les valeurs sont extrêmement basses, ce qui indique une très bonne performance de l'API.
  - /posts : 0.028 secondes (28 ms)
  - /comments : 0.018 secondes (18 ms)
  - /users : 0.014 secondes (14 ms)
  -

Resp time perc 90 (90e percentile), le temps de réponse maximum pour 90 % des requêtes.

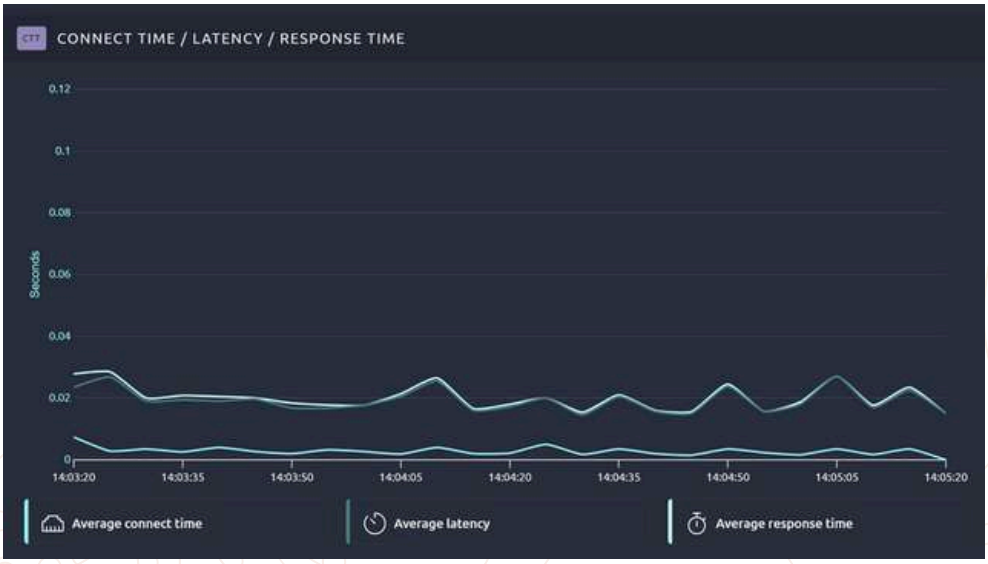
- Ces valeurs sont également très basses, reflétant une excellente stabilité.
  - **/posts : 0.037 secondes (37 ms)**
  - **/comments : 0.024 secondes (24 ms)**
  - **/users : 0.019 secondes (19 ms)**

Hit count (Nombre d'exécutions) :

- Chaque endpoint a été appelé 39 fois.
- Les résultats sont bien répartis, ce qui montre que toutes les requêtes ont été envoyées correctement.
-

Error % (Pourcentage d'erreurs) :

- Aucun échec enregistré (0 % d'erreurs pour tous les endpoints).



Average connect time : temps nécessaire pour établir la connexion au serveur, indique la réactivité initiale du serveur à établir une connexion réseau avec le client.

- Très faible dans votre cas (proche de zéro), ce qui signifie que le serveur est rapidement accessible.

Average latency : temps écoulé entre l'envoi de la requête et la réception de la première réponse (avant que le contenu complet ne soit téléchargé).

- La latence est stable, proche de zéro également (environ 0,01 à 0,02 secondes).

Average response time : temps total nécessaire pour que la réponse soit reçue intégralement par le client, correspond à la somme de la latence et du temps nécessaire pour télécharger les données.

- Les temps de réponse sont légèrement plus élevés (0,02 à 0,03 secondes), mais restent extrêmement rapides.

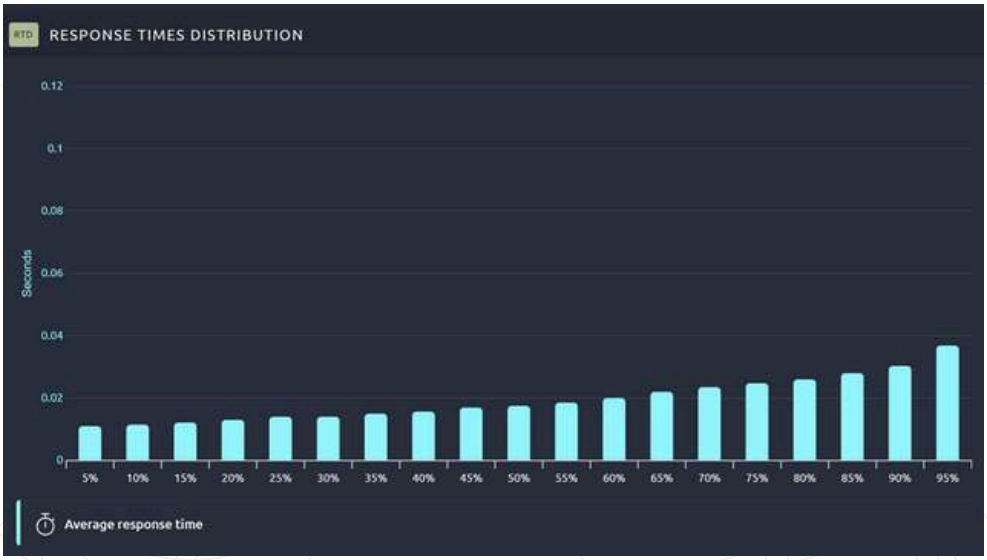
Le graphique en ligne (haut) représente les temps de réponse pour chaque endpoint (GET /posts, GET /comments, GET /users) en fonction du temps.

- Chaque ligne colorée correspond à un endpoint spécifique :
  - Violet : /posts
  - Bleu clair : /comments
  - Jaune : /users
- Le graphique en barre (bas) montre le temps de réponse maximal atteint par chaque endpoint pendant le test. Ces barres permettent de comparer les endpoints en un coup d'œil.

Axe X : percentiles (%), qui représentent la répartition des temps de réponse en fonction des requêtes.

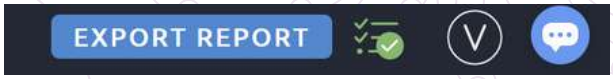
- Par exemple, le 50e percentile (ou médiane) montre que 50 % des requêtes ont un temps de réponse inférieur à cette valeur.

Axe Y : Temps de réponse (en secondes).



Débit global (en kilobits par seconde, Kb/s ou Mb/s), représentant la quantité de données transférées sur le réseau

- GET /comments (violet) a le débit le plus élevé, ce qui indique que cette requête transfère le plus de données par rapport aux autres endpoints.
- GET /posts (bleu) a un débit modéré.
- GET /users (jaune) a un débit très faible, suggérant que la réponse associée est beaucoup plus légère.



Vous pouvez exporter le rapport en PDF et vous pouvez personnaliser les métriques visibles dans les rapports.

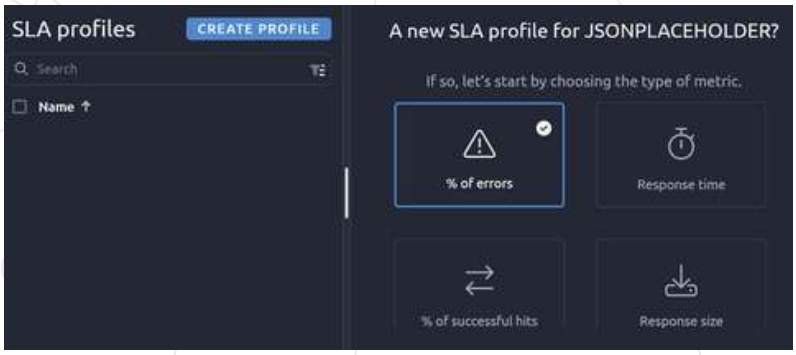
SLA PROFILES

Les SLA profiles (Service Level Agreement Profiles) permettent de définir des seuils ou objectifs de performance pour vos tests. Ces seuils sont utilisés pour mesurer si votre application ou système respecte les attentes en termes de qualité de service.

Objectif des SLA profiles :

- Évaluer si votre système respecte les niveaux de performance prédéfinis.
- Identifier rapidement les problèmes en cas de dépassement des seuils.
- Générer des rapports détaillés pour les équipes métier ou techniques, avec une évaluation claire de la conformité.

Comment configurer les SLA ?



Options disponibles pour les SLA Profiles :

- % of errors (Pourcentage d'erreurs) :
  - Permet de surveiller le pourcentage de requêtes ayant échoué.
  - Exemple d'utilisation :
    - Déclencher un avertissement si plus de 5 % des requêtes échouent.
    - Définir un seuil critique si plus de 10 % des requêtes échouent.
- Response time (Temps de réponse) :
  - Suivi du temps de réponse des requêtes pour garantir qu'il reste sous un certain seuil.
  - Exemple d'utilisation :
    - Avertissement si 95 % des requêtes dépassent 200 ms.
    - Seuil critique si 95 % des requêtes dépassent 500 ms.

% of successful hits (Pourcentage de succès des requêtes) :

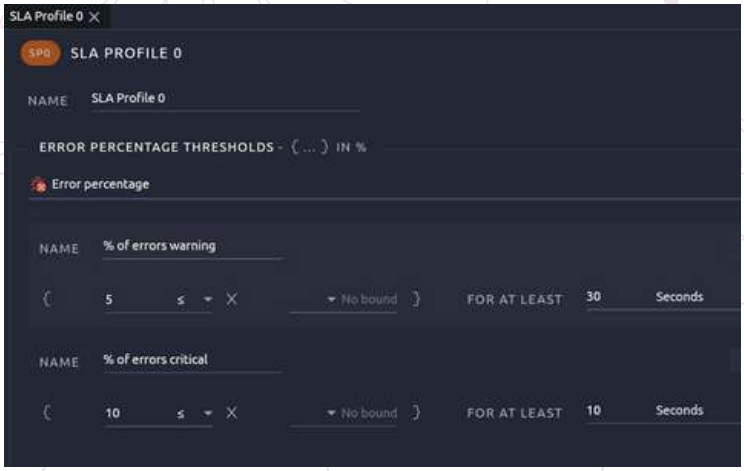
- Inverse du pourcentage d'erreurs, cette métrique mesure le ratio de requêtes ayant abouti avec succès.
- Exemple d'utilisation :
  - Définir un objectif pour que 99 % des requêtes soient réussies.

Response size (Taille de la réponse) :

- Permet de contrôler la taille des réponses HTTP (en KB ou MB) et de détecter les anomalies.
- Exemple d'utilisation :
  - Déclencher une alerte si la taille des réponses dépasse une limite définie.

Mise en place

1. Sélectionnez une des métriques selon vos besoins (par exemple, % of errors si vous souhaitez surveiller les erreurs).
2. Une fois la métrique choisie, vous pourrez configurer des seuils personnalisés :
   - Warning : Avertissement si une limite est atteinte (par exemple, 5 % d'erreurs).
   - Critical : Situation critique si une autre limite plus stricte est dépassée (par exemple, 10 % d'erreurs).



IMPORTEZ VOS SCENARIOS JMETER

OctoPerf permet d'importer directement des fichiers JMeter au format .jmx. Cette fonctionnalité peut vous aider si vous souhaitez migrer ou enrichir vos scénarios de test existants en exploitant la puissance et la simplicité d'OctoPerf.

Les avantages sont :

- **Réutilisation des scripts existants** : Si vous avez déjà des scénarios de test créés avec JMeter, vous n'avez pas besoin de repartir de zéro avec OctoPerf. Cette compatibilité vous fait gagner du temps.

- **Flexibilité** : Vous pouvez travailler dans JMeter pour des cas spécifiques, puis profiter des fonctionnalités avancées d'OctoPerf (visualisation, collaboration, intégration cloud, etc.).

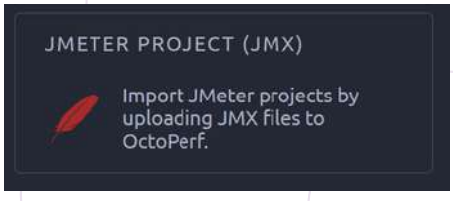
- **Standardisation** : Cela permet aux équipes habituées à JMeter de collaborer avec d'autres testeurs, moins techniques, grâce à l'interface OctoPerf.

Comment importer un script JMeter ?

- Rendez-vous dans l'onglet projets et sélectionnez votre projet ou créez en un nouveau.

- Cliquez sur le bouton Importer.

- Sélectionnez votre fichier JMX (format de script JMeter).



- Validation : OctoPerf analysera automatiquement votre fichier et convertira les éléments compatibles pour une utilisation directe.

EXPORTEZ VOS SCENARIOS OCTOPERF VERS JSON

OctoPerf offre la possibilité d'exporter vos scénarios de test au format JSON. Ce qui permet de :

- Sauvegarder vos scénarios : Vous pouvez exporter vos projets pour les archiver ou les réutiliser plus tard sur la plateforme OctoPerf.

- Partager avec votre équipe : Le fichier JSON peut être importé par d'autres utilisateurs d'OctoPerf pour collaborer ou effectuer des ajustements.

LE PLUGIN MAVEN D'OCTOPERF

Le plugin vous permet de configurer vos tests directement dans un projet Maven.

```
<plugin>
 <groupId>org.octoperf</groupId>
 <artifactId>octoperf-maven-plugin</artifactId>
 <version>1.0.0</version>
 <configuration>
  <apiKey>VOTRE_API_KEY</apiKey>
  <projectId>VOTRE_PROJECT_ID</projectId>
 <scenarioId>VOTRE_SCENARIO_ID</scenarioId>
 </configuration>
</plugin>
```

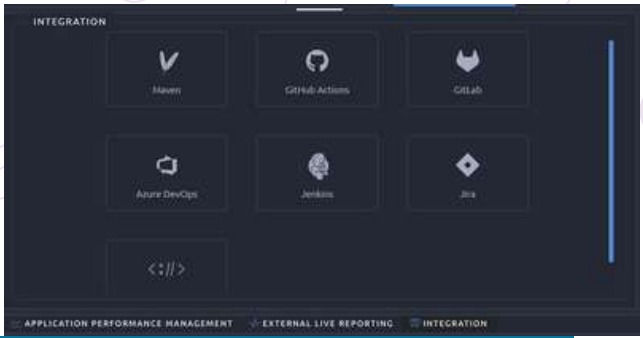
Une fois configuré, le plugin permet d'exécuter vos scénarios avec une simple commande Maven :

```
mvn octoperf:run
```

Les résultats sont directement accessibles sur la plateforme OctoPerf, avec des graphiques et des rapports détaillés.

- Le plugin s'intègre parfaitement dans des outils CI/CD comme Jenkins, GitLab CI ou GitHub Actions.
- Vous pouvez automatiser vos tests de charge pour chaque déploiement ou mise à jour de votre application.

Vous pouvez également le configurer avec l'aide de l'interface et la documentation :



MONITORING VS APM

QUELS OUTILS POUR VOUS ACCOMPAGNER DANS VOS TESTS ?

Pourquoi le monitoring est utile après un test de charge ?

Les tests de charge permettent de pousser un système à ses limites pour évaluer ses performances et sa stabilité sous contrainte. Cependant, une fois ces tests réalisés, la collecte de données seule n'est pas suffisante. C'est ici que le monitoring entre en jeu, offrant une analyse approfondie des résultats et permettant de transformer des chiffres bruts en informations exploitables. Le monitoring post-test peut répondre à ces questions :

- Quels sont les goulots d'étranglement identifiés ?
- Quelles ressources étaient sous-utilisées ou saturées ?
- Quels éléments ont contribué à des temps de réponse élevés ou à des erreurs systèmes ?

Le monitoring ne se limite pas aux tests de charge ponctuels. Il s'inscrit dans une stratégie de surveillance continue, garantissant que les performances restent conformes aux attentes, même en production. Deux grandes approches se distinguent pour cette tâche : les solutions de monitoring basées sur Grafana et Prometheus, et les outils d'APM (Application Performance Monitoring) comme Dynatrace, New Relic ou Datadog. Ces solutions, bien que complémentaires, répondent à des besoins différents.

Grafana + Prometheus : La puissance du monitoring open source

Prometheus collecte les métriques des systèmes (CPU, mémoire, latence, etc.) à intervalles réguliers via des endpoints exposés. Grafana, quant à lui, transforme ces métriques en tableaux de bord visuels et interactifs. Ensemble, ces outils open source permettent de créer une solution de monitoring flexible et personnalisable.

Avantages :

- Solution économique, sans frais de licence.
- Grande flexibilité dans la configuration des tableaux de bord et des alertes.
- Adapté pour monitorer les tests de charge, notamment avec des outils comme K6, Locust ou JMeter.

Limites :

- Configuration technique plus complexe, exigeant des compétences en administration système.
- Absence de certaines fonctionnalités avancées comme la traçabilité distribuée.
- Analyse limitée aux métriques systèmes, sans analyse approfondie des transactions utilisateur.

Quand les utiliser ?

- Si vous avez une architecture relativement simple (serveurs ou APIs isolés).
- Pour monitorer des tests de charge ponctuels ou réguliers.
- Si vous recherchez une solution économique et open source.

APM : Une analyse avancée pour les systèmes complexes

Les outils d'APM vont au-delà des métriques systèmes. Ils permettent une surveillance approfondie des performances des applications, en mettant l'accent sur les transactions utilisateur, les dépendances entre services, et les anomalies potentielles.

Avantages :

- Traçabilité distribuée : Identification des goulots d'étranglement sur tout le parcours d'une requête.
- Surveillance proactive : Détection d'anomalies grâce à l'IA et machine learning.
- Intégration facile avec des écosystèmes complexes (cloud, microservices, conteneurs).

Limites :

- Coût élevé, souvent facturé par hôte ou par transaction.
- Moins adapté pour les environnements simples ou à petit budget.

Quand les utiliser ?

- Si votre architecture comprend des microservices ou des conteneurs.
- Pour les applications critiques où une défaillance pourrait impacter directement les utilisateurs.
- Lorsque vous avez besoin d'une solution « clé en main » avec peu de configuration manuelle.

	Grafana + Prometheus	**Dynatrace, New Relic, Datadog (APM)**
Type d'outil	Monitoring et visualisation de métriques	Monitoring avancé et gestion des performances applicatives
Cas d'usage principal	Collecte et affichage de métriques système (CPU, RAM, erreurs, etc.)	Analyse des transactions utilisateur, traces distribuées, dépendances
Traçabilité distribuée	Non pris en charge nativement (nécessite Jaeger ou Zipkin)	Intégré, avec suivi des requêtes de bout en bout
Alertes et anomalies	Seuils manuels dans Prometheus	Alertes automatiques basées sur l'IA
Scalabilité	Dépend de l'infrastructure déployée	Haute scalabilité pour des systèmes complexes
Coût	Open source (coûts d'infrastructure et maintenance)	Payant (coûts par hôte ou transaction monitorée)

INTEGRATION AVEC PROMETHEUS

reprenons l'exemple de Locust et intégrons les outils de monitoring Grafana et Prometheus pour visualiser et analyser les données de performance en temps réel, de manière plus détaillée et centralisée. Cette approche est très utile pour les équipes souhaitant un suivi approfondi et historique des tests de charge, souvent intégré dans un système de surveillance global.

Pour cela, il faut installer la bibliothèque :

```
pip install prometheus-client
```

Puis, ajoutez un serveur de métriques basé sur prometheus-client dans votre script Locust :

Plus de code sur le repo Github :

```
from locust import HttpUser, task, between, events
from prometheus_client import start_http_server, Summary, Counter

REQUEST_LATENCY = Summary('http_request_latency_seconds', 'Time spent processing HTTP requests')
REQUEST_COUNT = Counter('http_request_count', 'Number of HTTP requests processed', ['endpoint'])

@events.init.add_listener
def start_metrics_server(environment, **kwargs):
    start_http_server(9100)

class JSONPlaceholderUser(HttpUser):
    host = "https://jsonplaceholder.typicode.com"
    wait_time = between(1, 3)

    @task(3)
    @REQUEST_LATENCY.time()
    def get_posts(self):
        response = self.client.get("/posts")
        REQUEST_COUNT.labels(endpoint="/posts").inc()
        if response.status_code != 200:
            print(f"Failed request to /posts with status {response.status_code}")

    @task(2)
    @REQUEST_LATENCY.time()
    def get_comments(self):
        response = self.client.get("/comments")
        REQUEST_COUNT.labels(endpoint="/comments").inc()
        if response.status_code != 200:
            print(f"Failed request to /comments with status {response.status_code}")

    @task(1)
    @REQUEST_LATENCY.time()
    def get_users(self):
        response = self.client.get("/users")
        REQUEST_COUNT.labels(endpoint="/users").inc()
        if response.status_code != 200:
            print(f"Failed request to /users with status {response.status_code}")
```

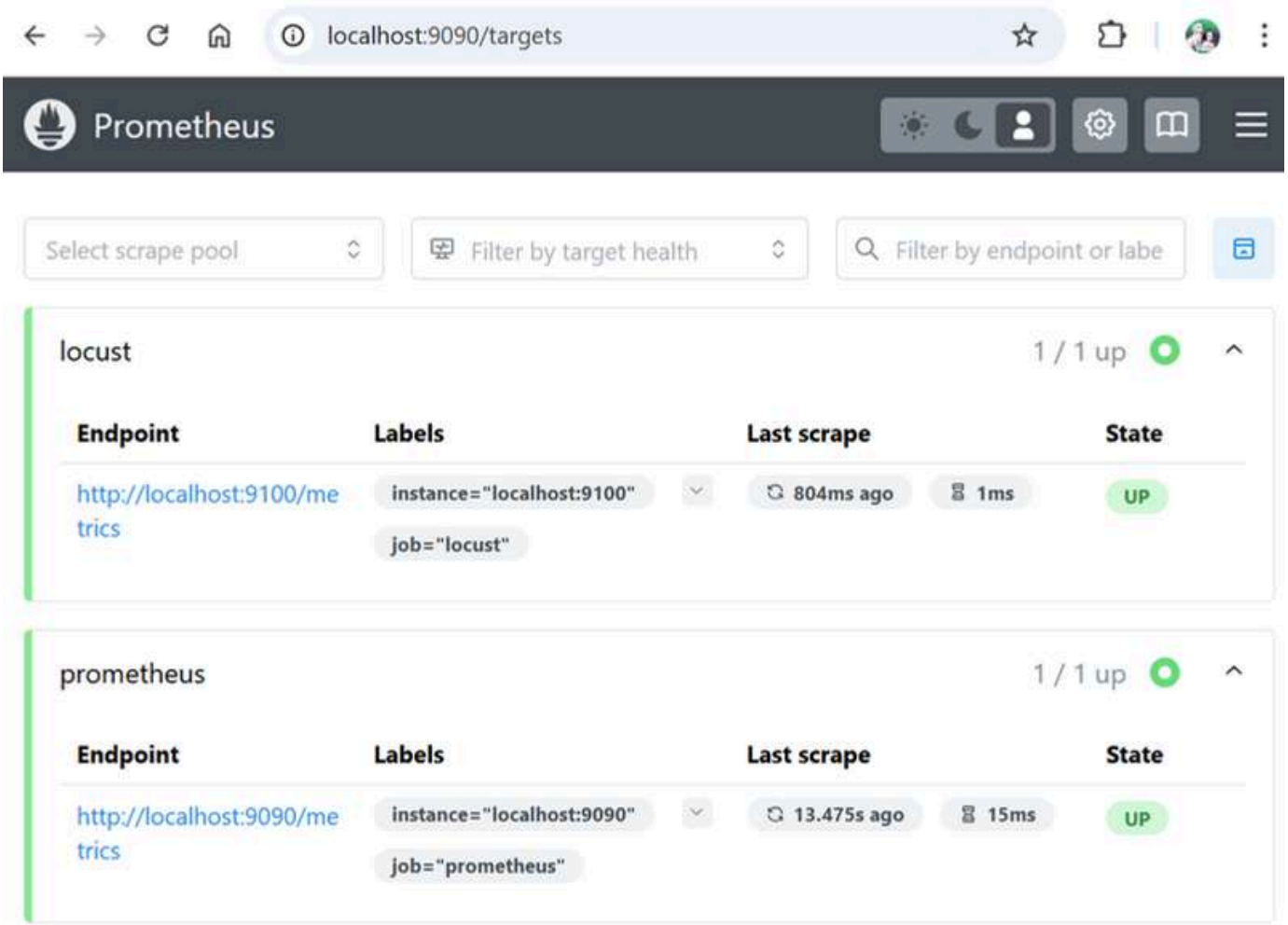
Configurer Prometheus pour collecter les données Locust :

- Dans le fichier de configuration de Prometheus (prometheus.yml), ajoutez une section pour surveiller l'endpoint Locust.

```
scrape_configs:
 - job_name: "locust"
   metrics_path: "/metrics"
   static_configs:
     - targets: ["localhost:9100"]
```

Cette configuration dit à Prometheus de scruter l'endpoint /metrics de Locust à intervalles réguliers. Les données collectées incluent des informations sur les temps de réponse, les taux d'échec, les requêtes par seconde, et d'autres métriques de performance.

Maintenant, dans localhost:9090/targets, vous devez voir le endpoint de locust et celui de prometheus :



Puis exécutez le script avec :

```
locust -f mon_script.py --host https://jsonplaceholder.typicode.com --headless  -u 1 -r 10 --run-time 10s
```

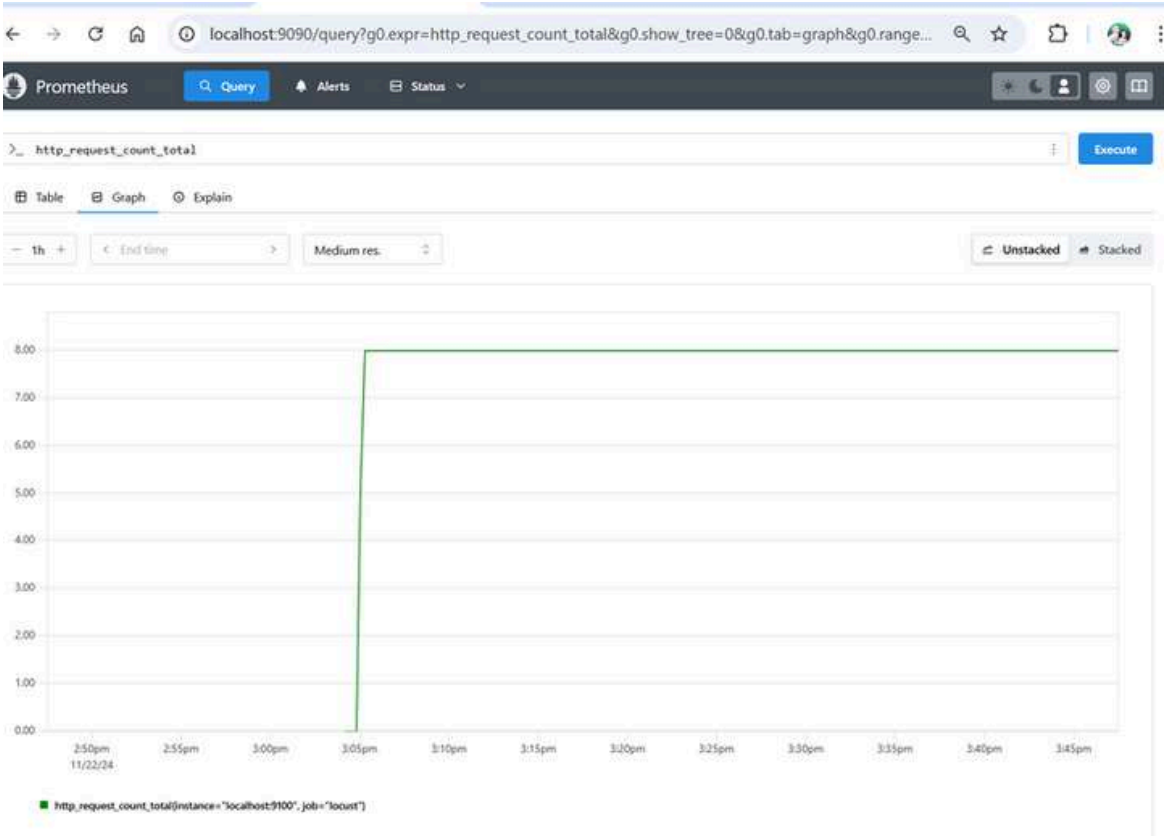
- --headless : Cette option exécute Locust en mode sans interface graphique. Cela signifie que vous n'avez pas accès à l'interface web de Locust, et que toutes les informations de suivi seront affichées directement dans le terminal. Ce mode est souvent utilisé pour des tests automatisés, comme dans un pipeline CI/CD.

- -u 100 : L'option -u permet de définir le nombre total d'utilisateurs simulés (ou "virtual users") qui participeront au test. Dans cet exemple, Locust simulera 100 utilisateurs simultanés au maximum.

- -r 10 : Cette option (-r) définit le taux de montée en charge (ou "spawn rate"), soit le nombre d'utilisateurs qui seront ajoutés chaque seconde jusqu'à atteindre le total défini avec -u. Ici, 1 utilisateur est ajouté chaque seconde, donc il faudra environ 10 secondes pour atteindre les 10 utilisateurs spécifiés.

- --run-time 10s : Cette option fixe la durée totale du test. Ici, le test est configuré pour s'exécuter pendant 10 secondes. Vous pouvez spécifier la durée en secondes (s), minutes (m), ou heures (h). Par exemple, --run-time 30m pour 30 minites ou --run-time 1h pour une heure.

Puis dans Graph, tapez http_request_count_total, vous devriez avoir un graphique :



Vous êtes prêts pour créer un dashboard Grafana !

INTEGRATION AVEC GRAFANA

Pour installer Grafana en local, en premier, il faut ajouter le dépôt officiel Grafana et de la clé GPG manquante :

```
wget -q -O - https://packages.grafana.com/gpg.key | sudo gpg --dearmor -o /usr/share/keyrings/grafana-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/grafana-archive-keyring.gpg] https://packages.grafana.com/oss/deb stable main" | sudo tee /etc/apt/sources.list.d/grafana.list > /dev/null
```

Puis, mettez à jour les listes de paquets et installez Grafana :

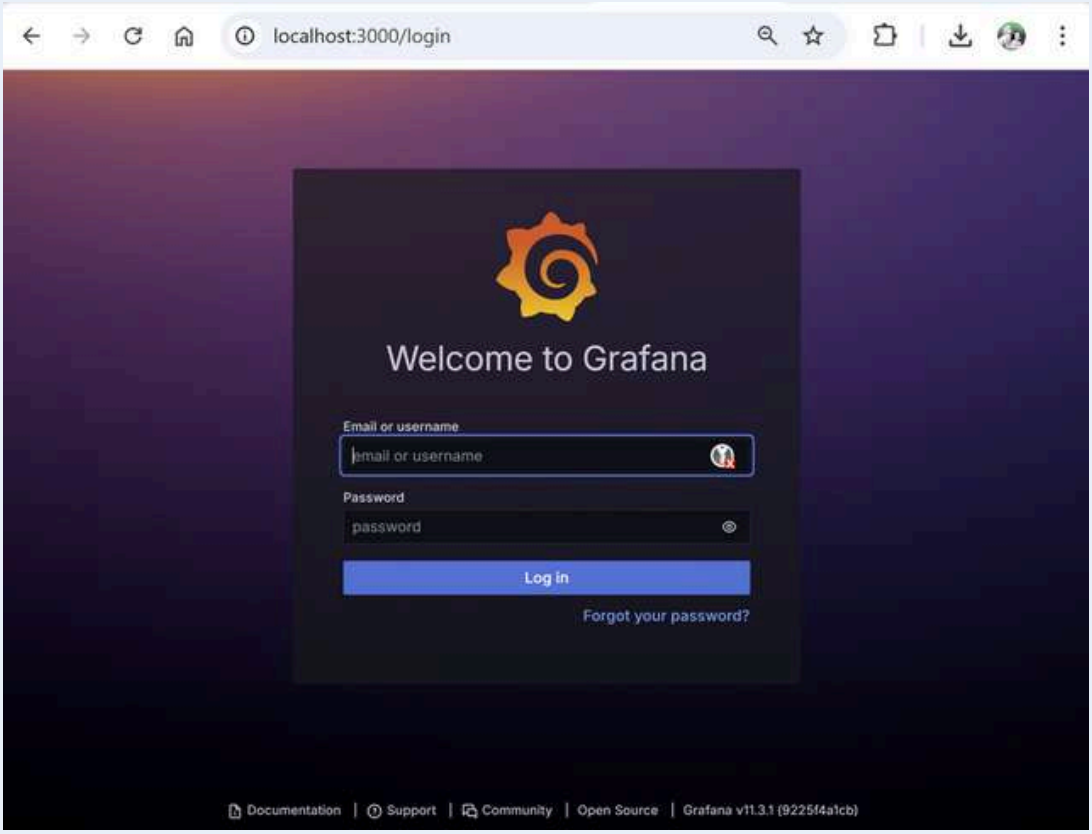
```
sudo apt-get update
sudo apt-get install grafana
```

Et enfin, démarrez Grafana avec des permissions root (si nécessaire) :

```
sudo grafana-server --homepath=/usr/share/grafana
```

Vous pouvez ainsi ouvrir Grafana :

```
http://localhost:3000
```



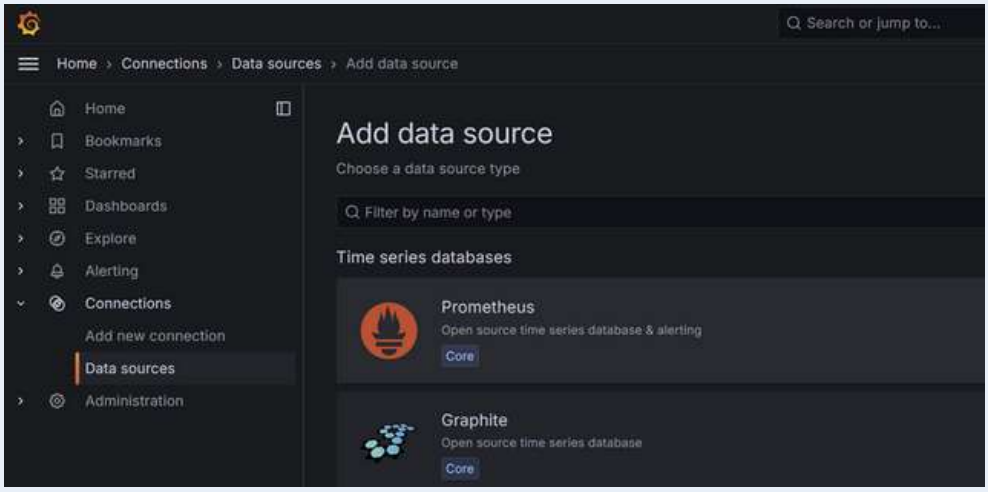
Le mot de passe et nom d'utilisateur est admin

INTEGRATION AVEC GRAFANA

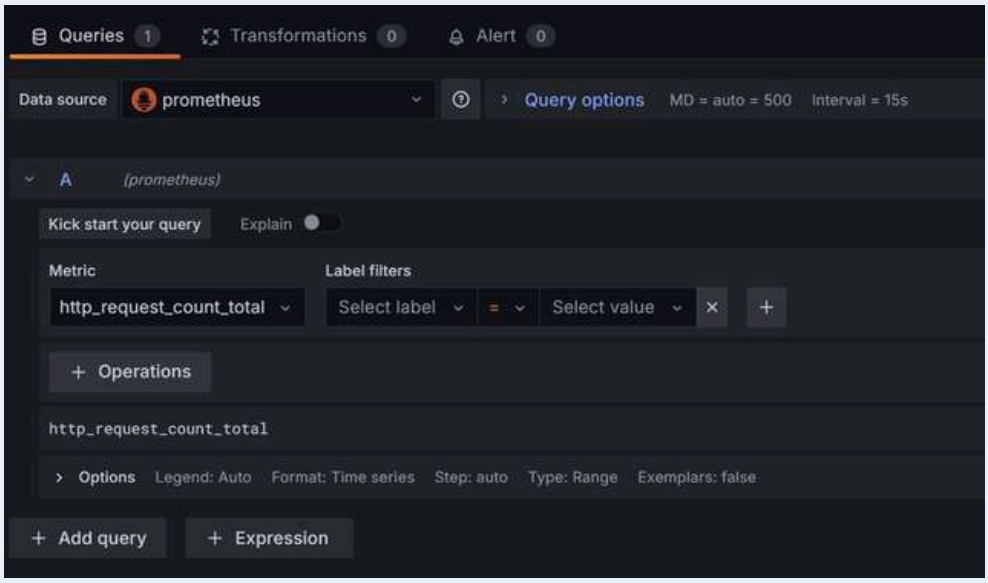
Plusieurs solutions s'offrent à vous.

Après avoir configurer Prometheus, vous pouvez utiliser Grafana pour les visualiser.
Pour cela, ajoutez Prometheus comme source de données dans Grafana :

- Dans Grafana, allez dans Configuration > Data Sources, puis sélectionnez Prometheus.
- Entrez l'URL de votre serveur Prometheus (ici, http://localhost:9090) et enregistrez la source de données.
- Ensuite, vous pouvez créer des tableaux de bord personnalisés dans Grafana pour visualiser les métriques spécifiques de Locust. Grafana propose de nombreux types de graphiques (courbes, jauges, histogrammes) pour représenter les données de manière claire.



Créer un dashboard et ajouter un panel :



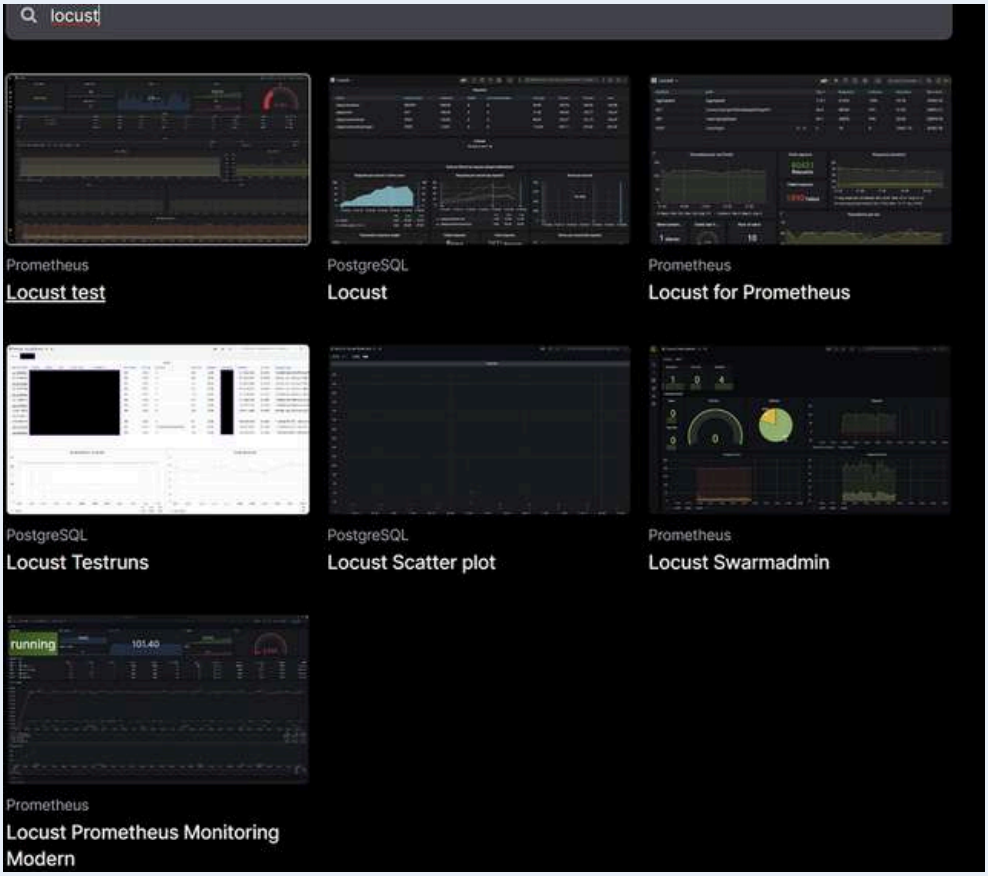
Ajouter Prometheus en data source, Puis ajouter une métrique, par exemple, http_request_count_total.
Sauvegardez, vous aurez un graphique ressemblant à :

Vous pouvez ainsi créer votre dashboard personnalisé.

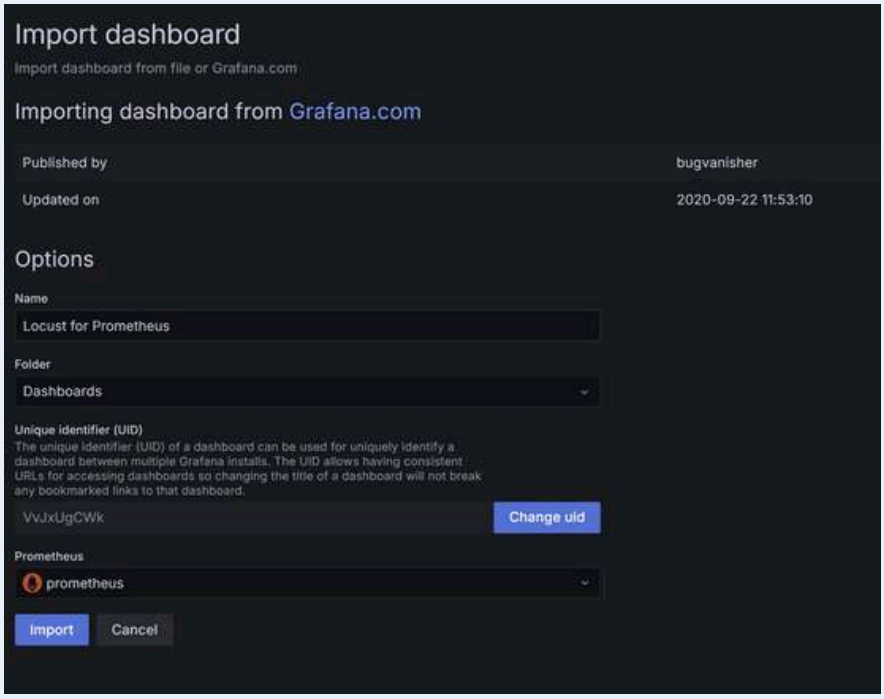
La seconde solution est d'importer un dashboard existant :

- Pour simplifier, vous pouvez trouver des dashboards préconçus dans la Galerie de Dashboards Grafana. Par exemple, certains dashboards sont déjà configurés pour les tests de charge Locust et les métriques exposées par Prometheus. Vous pouvez surveiller :
  - Le nombre d'utilisateurs actifs et leur montée en charge.
  - Les temps de réponse par percentile (médiane, 90e, 95e, etc.).
  - Le nombre de requêtes par seconde (RPS) et les taux d'échec.
  - L'utilisation des ressources système si Prometheus collecte également des données de monitoring de votre infrastructure.
  - Vous pouvez également configurer des alertes dans Grafana, par exemple, pour être informé si le taux d'échec dépasse un certain seuil ou si le temps de réponse devient trop élevé.

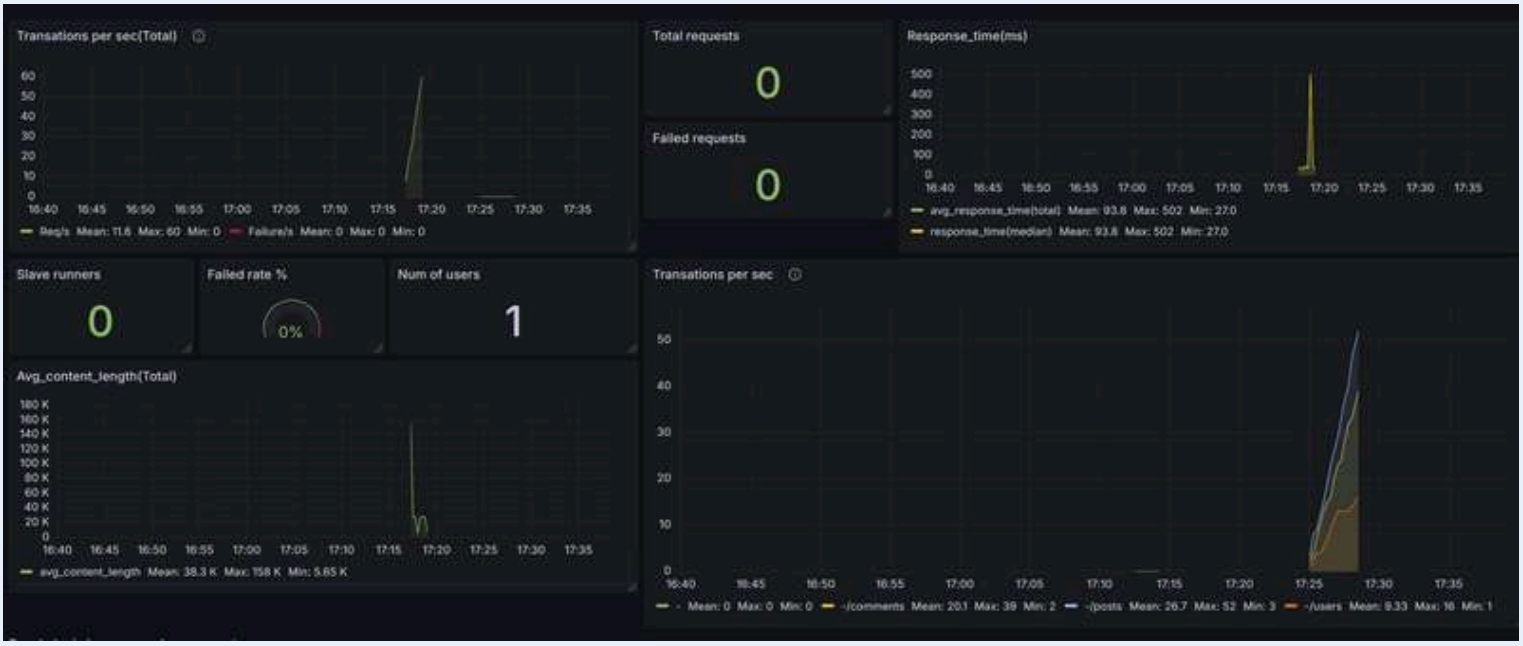
Vous pouvez trouver la liste des dashboards à cette URL : https://grafana.com/grafana/dashboards/?search=locust



Nous prenons pour exemple Locust for Prometheus. Cliquez sur Copy to clipboard. Puis importer un nouveau dashboard :



Voici le résultat de deux campagnes de test :



Nous pouvons voir que :

- Stabilité : Le système semble stable avec 1 utilisateur, sans échecs, et des temps de réponse moyens rapides.
- Temps de réponse :
  - La majorité des requêtes sont traitées rapidement (93 ms en moyenne).
  - Un pic de 502 ms peut nécessiter une investigation plus approfondie (endpoint concerné ou moment précis).
- Charge utilisateur : Ce test montre une faible charge, adaptée pour vérifier la base. Augmenter progressivement les utilisateurs permettra de valider la capacité de montée en charge.
- Données cohérentes :
  - Les tailles de réponse sont alignées avec les endpoints JSON.
  - Les proportions entre les endpoints sont respectées dans les RPS.

Sinon, vous pouvez utiliser Grafana sans Prometheus. Grafana est compatible avec plusieurs sources de données, et il est tout à fait possible de l'utiliser pour visualiser des données à partir de fichiers CSV ou d'autres sources, bien que cela demande quelques étapes de configuration supplémentaires.

Grafana ne prend pas en charge les fichiers CSV nativement, mais il existe des plugins qui permettent de les utiliser comme source de données.
Un plugin populaire pour cela est "CSV Plugin for Grafana" (ou parfois appelé Infinity Plugin). Vous pouvez l'installer via la commande suivante (nécessite les droits d'administrateur sur le serveur où Grafana est installé) :

```
grafana-cli plugins install grafana-infinity-datasource
```

Puis redémarrer avec :

```
sudo systemctl restart grafana-server
```

Ajouter une source de données CSV dans Grafana

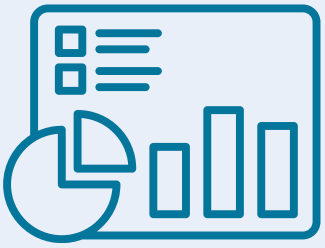
- Une fois le plugin installé et Grafana redémarré, allez dans Configuration > Data Sources dans l'interface de Grafana, puis cliquez sur Add data source.

- Recherchez et sélectionnez Infinity (ou CSV plugin selon le plugin utilisé).

- Configurez le plugin pour qu'il puisse lire des fichiers CSV. Selon le plugin, vous aurez la possibilité soit d'indiquer un chemin d'accès à un fichier local, soit de spécifier une URL pour un fichier CSV hébergé (par exemple, sur un serveur HTTP).

Charger les données CSV :

- Configurez le chemin du fichier CSV ou l'URL directe vers le fichier CSV. Assurez-vous que le fichier est accessible par le serveur où Grafana est installé si vous utilisez un chemin local.
- Le plugin vous permet de spécifier des paramètres comme le séparateur de colonnes (, pour CSV, ; pour certains fichiers), le format de date, et d'autres options pour parser le contenu correctement.



Créer des dashboards et graphiques :

- Dans le panneau de configuration des graphiques, sélectionnez la source de données CSV et configurez les visualisations en fonction des colonnes de votre fichier CSV.
- Par exemple, si votre fichier CSV contient des colonnes "timestamp", "response_time", et "RPS", vous pouvez utiliser ces champs pour créer des graphiques de série temporelle, des jauges, ou des histogrammes.

APM : DYNATRACE

Dynatrace est une plateforme avancée de monitoring et d'observabilité, souvent reconnue pour ses capacités d'analyse de performance en temps réel et son intelligence artificielle pour détecter automatiquement les anomalies. Bien que son cœur d'activité soit le monitoring, Dynatrace offre des fonctionnalités puissantes pour les tests de charge, en particulier lorsqu'elles sont combinées à d'autres outils de test de performance.

Dynatrace ne remplace pas directement les outils de test de charge comme LoadRunner ou NeoLoad. Cependant, il complète ces solutions en offrant une visibilité approfondie sur les performances systèmes et les comportements applicatifs pendant les tests. Voici comment il fonctionne :

- Surveillance des métriques clés : Dynatrace collecte et analyse en temps réel les métriques de performance (temps de réponse, utilisation des ressources, transactions, etc.) pendant un test de charge.

- Corrélation des données : La plateforme associe les données de performance aux logs, aux traces distribuées et aux événements systèmes pour fournir une vue unifiée de la santé des systèmes.

- Anomalies automatisées : Grâce à son IA, Dynatrace identifie automatiquement les goulets d'étranglement, les anomalies de performance et les dépendances critiques qui peuvent limiter la scalabilité.

- Rapports détaillés : Les rapports Dynatrace mettent en évidence les problèmes rencontrés, tels que les problèmes de base de données, les limitations de CPU/mémoire, ou les délais liés aux appels API.

Avantages

- Observabilité approfondie : Contrairement aux outils traditionnels, Dynatrace fournit une vue détaillée sur les microservices, les containers, et les environnements cloud.

- Détection prédictive des anomalies : L'IA de Dynatrace anticipe les problèmes potentiels avant qu'ils ne deviennent critiques.

- Automatisation : Les tests peuvent être intégrés aux workflows CI/CD pour des tests de performance continus.

- Corrélation avec l'expérience utilisateur : Dynatrace relie les performances systèmes aux impacts sur les utilisateurs finaux, aidant ainsi à prioriser les résolutions.

Limitations

- Coût : Dynatrace est une solution premium, ce qui peut représenter un investissement important pour les petites entreprises.

- Dépendance à d'autres outils : Il ne remplace pas les outils de test de charge comme JMeter ou LoadRunner ; il les complète.

- Configuration initiale : Bien que la plateforme soit intuitive, une configuration initiale minutieuse est nécessaire pour des résultats optimaux.

Cas d'utilisation

- Tests pré-déploiement : S'assurer que l'application peut gérer les charges prévues dans un environnement de production.

- Optimisation cloud : Identifier les goulots d'étranglement dans les architectures cloud-natives.

- Validation des microservices : Tester les microservices individuellement pour vérifier leur performance et leur scalabilité.

- Surveillance continue : Effectuer des tests réguliers dans le cadre des pipelines CI/CD pour garantir la stabilité.

MISE EN OEUVRE AVEC K6 : VISUALISATION DES RESULTATS DANS DYNATRACE

Dynatrace peut être intégré avec des outils comme K6 pour surveiller en temps réel les métriques de performance.

Voici les étapes pour configurer cette intégration et visualiser les résultats dans un tableau de bord Dynatrace.

Préparation de l'environnement

Utilisez l'outil xk6 pour ajouter l'extension Dynatrace à K6 :

```
xk6 build --with github.com/Dynatrace/xk6-output-dynatrace@latest
```

Cela génère un fichier k6.exe capable d'envoyer des métriques directement à Dynatrace.

Avant, vous aurez peut-être besoin d'installer Go :

- Rendez-vous sur le site officiel : https://go.dev/dl/
- Téléchargez la version compatible avec votre système d'exploitation
- Suivez les instructions de l'installateur pour terminer l'installation

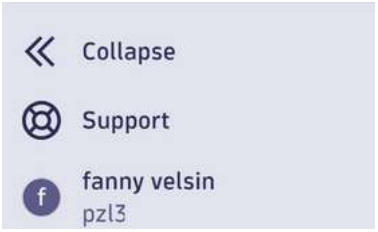
Puis créez votre compte Dynatrace sur : https://www.dynatrace.com/

Ensuite, créez votre token, Rendez-vous dans Settings > Access Tokens et générez un token avec au minimum la permission "Ingest metrics"

Définissez les variables d'environnement

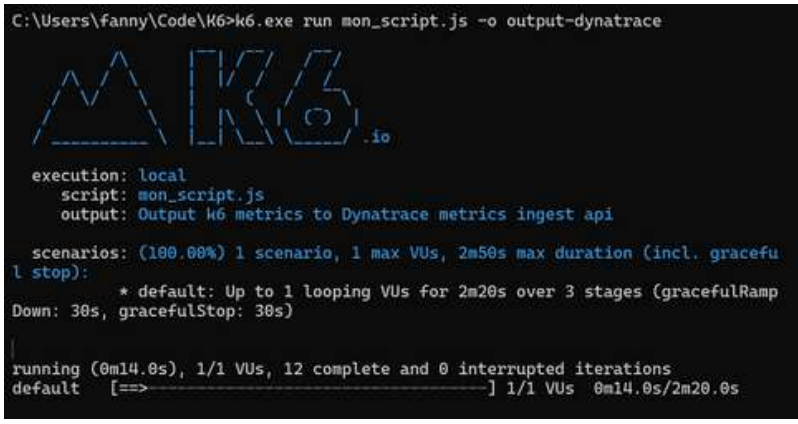
K6_DYNATRACE_URL="https://<environmentid>.live.dynatrace.com"
K6_DYNATRACE_APITOKEN="<api-token>"

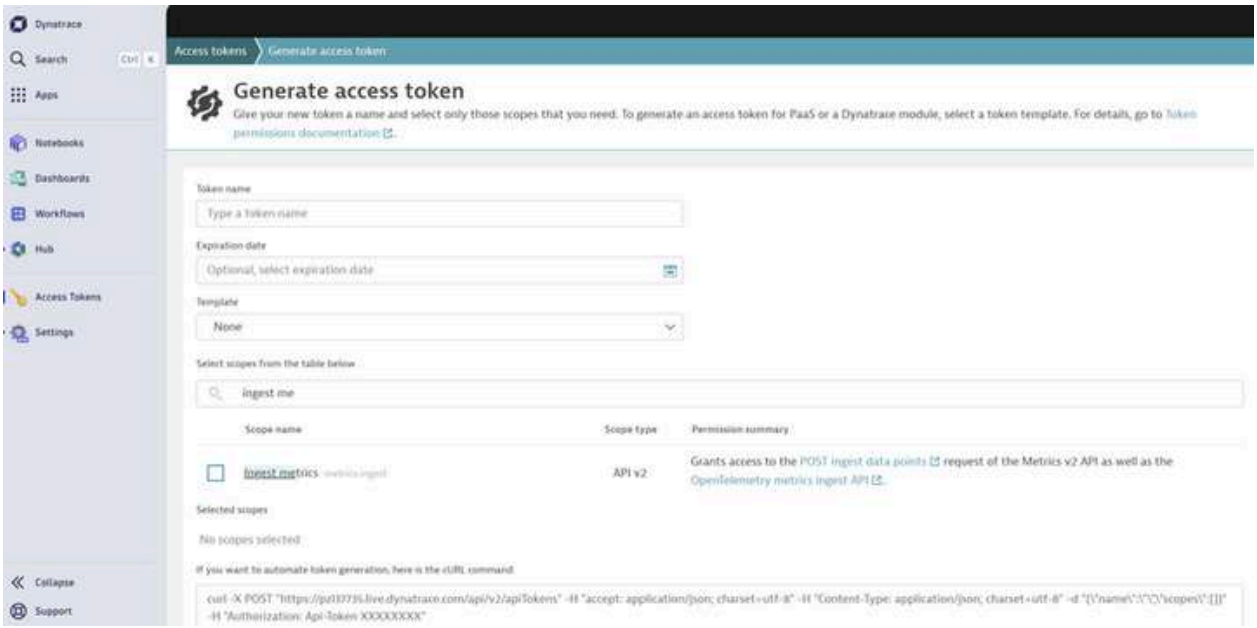
L'api token est le token que vous avez créer précédemment. Vous trouverez l'environnementid en bas sous votre nom :



Lancer le test K6 avec l'extension Dynatrace

```
k6.exe run mon_script.js -o output-dynatrace
```





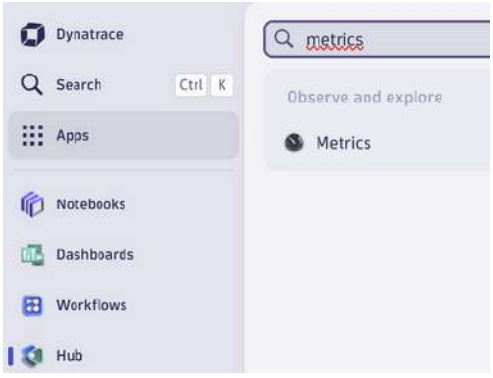
Reprenons l'exemple page 25 du script K6, et adaptez les durations pour l'exécuter plus de 20 min :

```
stages: [
    { duration: '5m', target: 20 },
    { duration: '5m', target: 40 },
    { duration: '1m', target: 0 },
],
```

Après 20 min d'exécutions :

```
C:\Users\fanny\Code\K6>k6.exe run mon_script.js -o output-dynatrace

  execution: local
     script: mon_script.js
     output: Output k6 metrics to Dynatrace metrics ingest api

  scenarios: (100.00%) 1 scenario, 1 max VUs, 21m30s max duration (incl. graceful stop):
           * default: Up to 1 looping VUs for 21m0s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)


     ✓ status is 200
     ✗ response time < 200ms
       ↳  99% — ✓ 3446 / ✗ 1

     checks.........................: 99.98% ✓ 6893      ✗ 1
     data_received..................: 222 MB 176 kB/s
     data_sent......................: 720 kB 572 B/s
     http_req_blocked...............: avg=34.03µs min=0s      med=0s      max=116.06ms p(90)=0s       p(95)=0s
     http_req_connecting............: avg=4.33µs  min=0s      med=0s      max=14.93ms  p(90)=0s       p(95)=0s
   ✓ http_req_duration..............: avg=31.78ms min=20.53ms med=29.19ms max=210.86ms p(90)=43.51ms  p(95)=47.75ms
       { expected_response:true }...: avg=31.78ms min=20.53ms med=29.19ms max=210.86ms p(90)=43.51ms  p(95)=47.75ms
   ✓ http_req_failed................: 0.00%  ✓ 0         ✗ 3447
     http_req_receiving.............: avg=4.39ms  min=0s      med=1.54ms  max=107.3ms  p(90)=16.46ms  p(95)=18.22ms
     http_req_sending...............: avg=119.8µs min=0s      med=0s      max=2.02ms   p(90)=581.91µs p(95)=667.73µs
     http_req_tls_handshaking.......: avg=19.19µs min=0s      med=0s      max=66.16ms  p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=27.26ms min=19.34ms med=25.52ms max=109.27ms p(90)=33.81ms  p(95)=38.34ms
     http_reqs......................: 3447   2.735687/s
     iteration_duration.............: avg=1.09s   min=1.07s   med=1.09s   max=1.39s    p(90)=1.11s    p(95)=1.12s
     iterations.....................: 1149   0.911896/s
     test_counter...................: 1149   0.911896/s
     vus............................: 1      min=1       max=1
     vus_max........................: 1      min=1       max=1


running (21m00.0s), 0/1 VUs, 1149 complete and 0 interrupted iterations
default ✓ [======================================] 0/1 VUs  21m0s
```

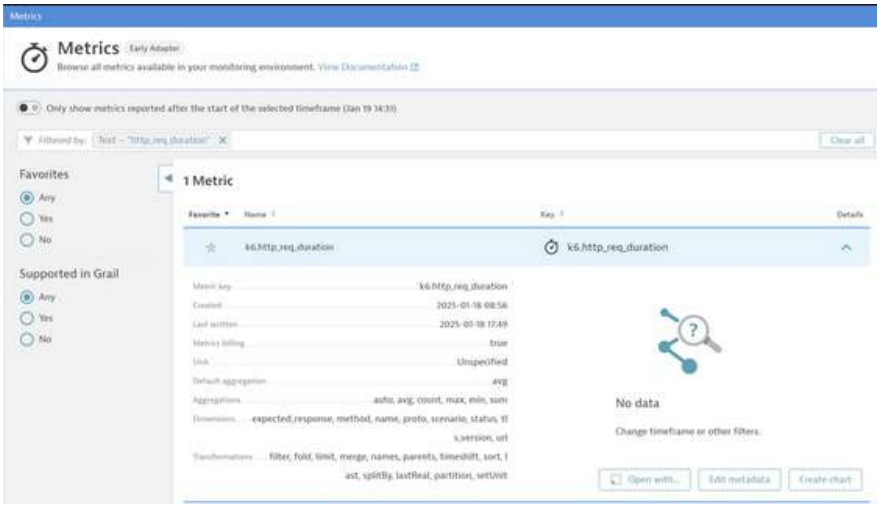
Vous pouvez donc vérifiez vos métriques dans Dynatrace dans la section metrics :



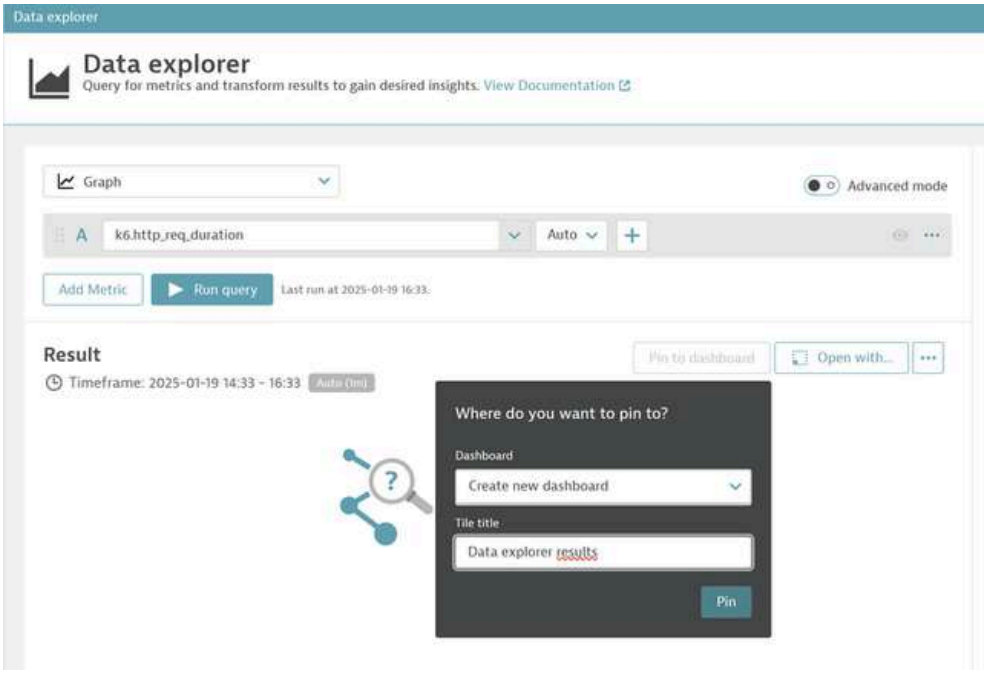
Si vous n'avez pas de metrics, vous pouvez tester votre connexion :
curl -X POST "https://<environmentid>.live.dynatrace.com/api/v2/metrics/ingest" -H "Authorization: Api-Token <votre-token-api>" -H "Content-Type: text/plain" --data "custom.metric.test,environment=test 1"

Si vous avez 401 comme reponse, revoyez vos variables d'environnements

vous pouvez donc vérifiez vos métriques dans Dynatrace dans la section metrics :



Vous pouvez créer un dashboard personnalisé en cliquant sur “Create Chart” puis sur Pin to dashboard en spécifiant le nom du dashboard que vous voulez créer :



Vous pouvez utiliser les différentes informations :

Pour les requetes HTTP :

http_reqs : Nombre total de requêtes HTTP.
http_req_duration : Durée totale des requêtes HTTP (y compris le temps de réception et d'attente).
http_req_waiting : Temps d'attente pour la réponse du serveur.
http_req_connecting : Temps pour établir une connexion TCP.
http_req_tls_handshaking : Temps pour établir une connexion TLS/SSL.
http_req_sending : Temps pour envoyer la requête HTTP.
http_req_receiving : Temps pour recevoir la réponse HTTP.
http_req_failed : Taux d'échec des requêtes HTTP (en pourcentage).

Utilisation des ressources :

vus : Nombre d'utilisateurs virtuels actifs (Virtual Users).
vus_max : Nombre maximal d'utilisateurs virtuels pendant le test.
iterations : Nombre total d'itérations complétées (chaque utilisateur exécute une itération du script).

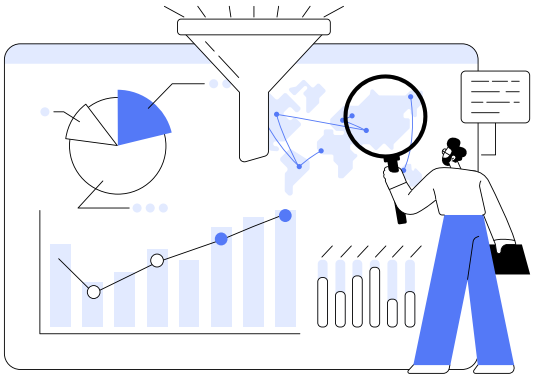
Performance et latence :

iteration_duration : Temps moyen pour terminer une itération complète (par utilisateur).
checks : Pourcentage de contrôles réussis ou échoués dans le test.

Données transmises :

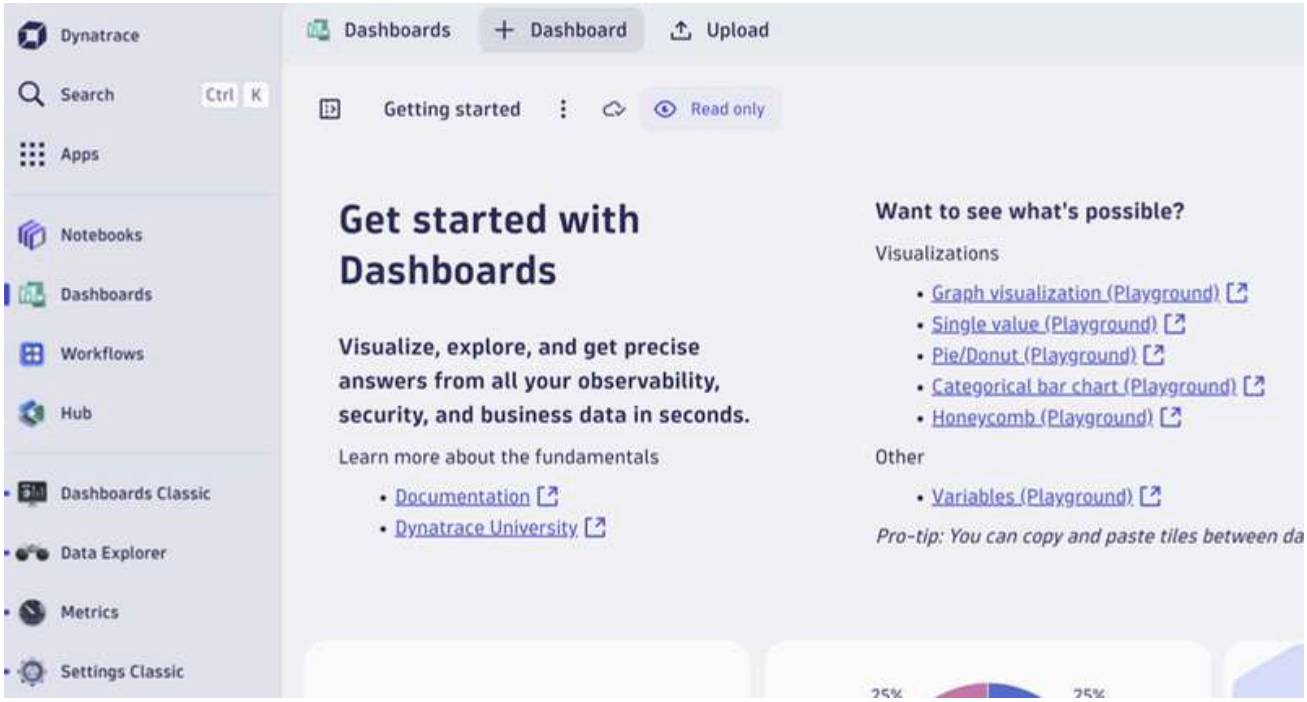
data_sent : Volume total de données envoyées (en octets).
data_received : Volume total de données reçues (en octets).

Import de dashboard K6



Sinon, vous pouvez importer des dashboards existants. Gardons l'exemple de K6 et rendez vous sur le github : https://github.com/dynatrace-perfclinics/dynatrace-getting-started/blob/main/dashboards/k6/Grafana%20k6%20Dashboard.json

Téléchargez le dashboard et importez le dans dynatrace en cliquant sur upload :



Le résultat du test que nous avons fait est alors :

Section "Quick Glance" (Aperçu rapide)

- Average VUs (Virtual Users) :
  - Moyenne des utilisateurs virtuels (VUs) actifs pendant le test.
  - Dans notre cas, 10,59 utilisateurs virtuels étaient actifs en moyenne.
  - La flèche indique une augmentation de 1318,6 %, ce qui signifie une forte montée en charge par rapport à la session précédente qui ne chargeait que d'une minute.

- Max VUs (Virtual Users) :
  - Nombre maximum d'utilisateurs virtuels simultanés au cours du test.
  - Ici, 40,00 utilisateurs actifs ont été atteints.

- Failed request count :
  - Nombre total de requêtes HTTP ayant échoué.
  - Dans notre test, aucune requête n'a échoué (la barre reste vide).

- Min Request Duration :
  - Temps de réponse minimum (en millisecondes) parmi toutes les requêtes HTTP.
  - Ici, la requête la plus rapide a pris 18,69 ms.
  - La flèche rouge (-13,3 %) indique une amélioration (réduction du temps minimum par rapport à la session précédente).

- Avg Request Duration :
  - Temps moyen de réponse des requêtes HTTP.
  - Les requêtes ont pris en moyenne 32,05 ms pour obtenir une réponse.
  - Une augmentation de 4,81 % indique une légère dégradation de la performance moyenne par rapport au test précédent.

- Max Request Duration :
  - Temps de réponse maximum pour les requêtes HTTP.
  - La requête la plus lente a pris 355,32 ms.
  - Une augmentation de 305,67 % indique qu'une ou plusieurs requêtes ont pris beaucoup plus de temps qu'avant, ce qui peut signaler un problème ponctuel ou un goulot d'étranglement.

Average VUs (Utilisateurs Virtuels Moyens)

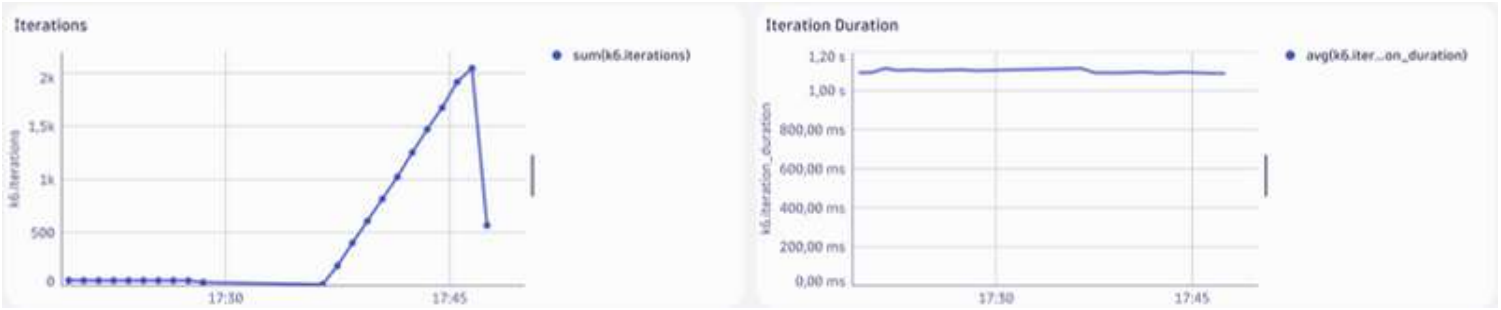
- Le graphique montre la moyenne des utilisateurs virtuels actifs (VUs) à différents moments du test.

- Interprétation :
  - Une montée progressive est visible jusqu'à atteindre environ 40 utilisateurs actifs, avant de diminuer à la fin du test.
  - Cela correspond aux étapes de montée en charge définies dans notre script

Max VUs (Utilisateurs Virtuels Maximum)

- Le graphique montre le nombre maximal d'utilisateurs simultanés atteints à chaque instant.

- Interprétation :
  - Vous atteignez un maximum de 40 utilisateurs simultanés à la fin de la montée en charge, ce qui semble stable avant la phase de descente.



Itérations

- Le graphique affiche le nombre total d'itérations complétées au fil du temps.

- Interprétation :
  - Une augmentation régulière est observée, correspondant à l'accélération de la montée en charge (plus d'utilisateurs exécutant simultanément des itérations).
  - Le nombre d'itérations décroît lorsque les utilisateurs virtuels diminuent à la fin du test.
  - Ce total est directement lié aux utilisateurs actifs et à la durée du test.

- Utilisation :

Confirmez si le nombre d'itérations correspond aux attentes.

Iteration Duration (Durée des Itérations)

- Le graphique montre la durée moyenne d'une itération (en secondes ou millisecondes).

- Interprétation :
  - Les itérations prennent en moyenne 1,2 secondes, ce qui reste constant tout au long du test.
  - Cela indique que le système semble capable de gérer la charge imposée par les utilisateurs, sans augmentation significative des temps d'exécution.

- Utilisation :
  - Une augmentation soudaine ou progressive de la durée des itérations pourrait signaler un goulot d'étranglement ou une saturation des ressources.

Data Sent (Données envoyées)

- Le graphique montre le volume total de données envoyées par les utilisateurs virtuels (VUs) tout au long du test.

- Interprétation :
  - Une augmentation progressive est observée au fur et à mesure que le nombre d'utilisateurs actifs augmente.
  - À son pic, environ 1,2 Mo de données ont été envoyées.
  - La diminution à la fin correspond à la phase de réduction des utilisateurs virtuels.

- Utilisation :

Permet de vérifier si le volume des données envoyées reste conforme à ce qui est attendu pour les tests de charge (par exemple, tailles des requêtes HTTP).

Data Received (Données reçues)

- Le graphique montre le volume total de données reçues en réponse aux requêtes HTTP.

- Interprétation :
  - Le volume de données reçues augmente proportionnellement au nombre d'utilisateurs virtuels actifs et atteint environ 400 Mo au pic.
  - Une forte diminution est visible à la fin, reflétant la réduction des utilisateurs actifs.

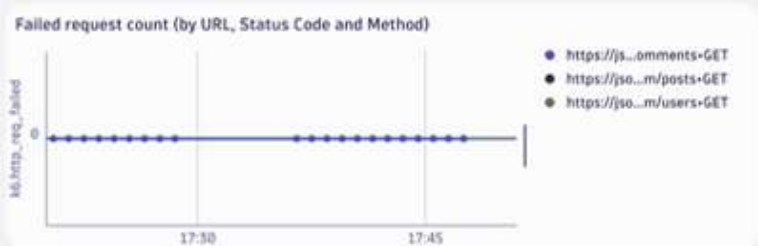
- Utilisation :
  - Confirmez si le volume de données reçues correspond à la taille attendue des réponses serveur.
  - Une diminution ou des anomalies pourraient indiquer des problèmes côté serveur (comme des erreurs HTTP).

Request Count (Nombre de requêtes)

- Le graphique montre le nombre total de requêtes HTTP exécutées par les utilisateurs virtuels au fil du temps.

- Interprétation :
  - Les trois endpoints testés sont affichés individuellement :
    - /comments
    - /posts
    - /users
  - Les requêtes augmentent avec la montée en charge et atteignent un pic lors du maximum d'utilisateurs actifs.
  - Une diminution est observée lorsque les utilisateurs virtuels sont réduits.

- Utilisation :
  - Permet d'évaluer si chaque endpoint reçoit la charge prévue.
    - Une chute soudaine ou une absence de requêtes pourrait indiquer un problème (par exemple, des erreurs ou un endpoint non accessible).



Failed Request Count (Nombre de requêtes échouées)

- Le graphique montre le nombre de requêtes HTTP ayant échoué (par URL, statut HTTP et méthode) tout au long du test.

Interprétation :

- Ligne plate à zéro :
  - Le fait que cette ligne reste à zéro indique qu'aucune requête HTTP n'a échoué pendant le test.
  - Tous les endpoints testés (/comments, /posts, /users) ont retourné des réponses valides (statut HTTP 200).

Utilisation :

- Cela montre la fiabilité des endpoints sous la charge imposée par K6.
- Si des échecs apparaissaient, cela pourrait signaler :
  - Des problèmes côté serveur (surcharge, erreurs applicatives).
  - Des erreurs réseau ou de configuration (par exemple, des URL incorrectes).

Blocked Request Count (Nombre de requêtes bloquées)

- Le graphique montre la durée moyenne pendant laquelle les requêtes HTTP ont été bloquées avant d'être envoyées.

Interprétation :

- Augmentation au début de la montée en charge :
  - Une augmentation est visible vers 17:30, ce qui correspond à la montée en charge des utilisateurs virtuels.
  - Cela peut indiquer que le client K6 a temporairement mis en attente certaines requêtes en raison de :
    - Temps d'établissement de connexions (TCP ou TLS).
    - Latences dues à la gestion des threads.
- Diminution progressive :
  - Une fois la montée en charge stabilisée, les requêtes bloquées sont quasiment inexistantes.
  - Cela indique que le système testé et l'infrastructure réseau peuvent gérer la charge imposée sans retards significatifs.

Utilisation :

- Des requêtes bloquées prolongées peuvent signaler :
  - Des problèmes côté client (temps de configuration des connexions).
  - Des limitations réseau ou serveur qui ralentissent l'envoi des requêtes.

Request Sending Time (Temps d'envoi des requêtes)

- Le graphique montre le temps moyen nécessaire pour envoyer une requête HTTP au serveur.

- Interprétation :
  - La valeur moyenne se situe autour de 100 à 200 µs (microsecondes).
  - Une légère augmentation est visible lors de la montée en charge, mais le temps reste très faible.
  - Vers la fin du test, le temps d'envoi diminue progressivement lorsque la charge baisse.

- Utilisation :
  - Un temps d'envoi constant et faible indique que le client (K6) n'a pas rencontré de difficultés à transmettre les requêtes au serveur.
  - Une augmentation marquée du temps d'envoi pourrait signaler des problèmes au niveau de la connectivité réseau ou de l'infrastructure de test.

Request Waiting Time (Temps d'attente des requêtes)

- Le graphique montre le temps moyen que K6 a attendu pour recevoir une réponse après avoir envoyé une requête.

- Interprétation :
  - La valeur moyenne se situe autour de 30 ms, avec une stabilité notable tout au long du test.
  - Une légère hausse est observée pendant la montée en charge (entre 17:30 et 17:45), probablement due à l'augmentation du nombre d'utilisateurs simultanés.
  - La stabilité générale indique que le serveur a pu gérer la charge sans dégradation significative des performances.

- Utilisation :
  - Un temps d'attente faible et constant montre que le serveur a répondu rapidement aux requêtes.
  - Une augmentation progressive ou des pics soudains pourraient indiquer une surcharge du serveur ou des goulots d'étranglement (par exemple, une base de données lente).

Request Receiving Time (Temps de réception des requêtes)

- Le graphique montre le temps moyen nécessaire pour recevoir la réponse complète d'une requête HTTP après qu'elle a été envoyée.

Interprétation :

- Pic initial (10 ms) :
  - Un pic est visible au début du test (aux alentours de 17:30), ce qui est typique lorsque les premiers utilisateurs virtuels commencent leurs interactions.
  - Cela peut être lié à des latences initiales dues à la mise en cache ou à la montée en charge des serveurs.

- Stabilisation (5 ms) :
  - Après ce pic, le temps de réception diminue et se stabilise autour de 5 ms, indiquant des réponses rapides et efficaces du serveur.

Utilisation :

- Un temps de réception stable et bas indique que le serveur peut répondre rapidement, même sous charge.
- Une augmentation progressive ou des pics constants pourraient signaler un problème côté serveur (latences élevées dues à des processus longs, surcharge réseau, ou tailles de réponses importantes).

Request TLS Handshaking (Établissement de la connexion TLS)

- Le graphique montre le temps moyen pris pour établir une connexion sécurisée TLS (Transport Layer Security).

Interprétation :

- Pic pendant la montée en charge (1,5 s) :
  - Une augmentation soudaine est visible lors de la montée en charge (17:30), liée à l'augmentation rapide du nombre de connexions simultanées nécessitant une négociation TLS.
- Diminution rapide après stabilisation :
  - Une fois la montée en charge terminée, le temps de handshake diminue et reste proche de 0 s, indiquant que le serveur peut gérer efficacement les connexions TLS après leur initialisation.

Utilisation :

- Un pic lors de la montée en charge est normal, car chaque utilisateur virtuel initialise une nouvelle connexion TLS.
- Si les temps de handshake restent élevés ou augmentent avec le temps, cela pourrait signaler des problèmes liés :
  - À la capacité du serveur à gérer les connexions sécurisées.
  - À des configurations TLS inefficaces (par exemple, certificats lourds ou configurations obsolètes).

Request Connecting (Temps de connexion)

- Le graphique montre le temps moyen pris pour établir une connexion TCP avant d'envoyer une requête HTTP.

Interprétation :

- Pic initial (400 µs) :
  - Un pic de temps de connexion est visible au début du test, vers 17:30. Cela correspond à la montée en charge où de nombreuses connexions TCP sont initialisées simultanément.

- Stabilisation à 0 µs :
  - Après la montée en charge, le temps de connexion diminue rapidement à 0 µs, ce qui est attendu lorsque les connexions TCP sont déjà établies et réutilisées pour des requêtes ultérieures.

TESTS DE CHARGE DANS LES ENVIRONNEMENTS MODERNES : CLOUD, DEVOPS, CI/CD

Intégration avec le cloud

- Scalabilité à la demande : Le cloud permet de simuler des charges massives sans investir dans une infrastructure physique. Des outils comme K6 Cloud proposent des environnements prêts à l'emploi pour exécuter des tests de charge depuis plusieurs régions simultanément.

- Tests multi-régions : Les environnements cloud permettent d'évaluer les performances d'une application en fonction de la localisation géographique des utilisateurs, en simulant des scénarios de latence réels.

- Optimisation des coûts : Facturation à l'usage pour les tests, ce qui réduit les dépenses pour des tests ponctuels ou réguliers.

Automatisation avec DevOps et CI/CD

- Pipelines CI/CD : Les outils modernes s'intègrent directement dans les pipelines de déploiement continu, permettant de déclencher automatiquement des tests de charge à chaque modification de code. Par exemple JMeter ou K6 peuvent être configurés pour des exécutions régulières dans Jenkins ou GitLab CI.

- Surveillance des régressions : En exécutant des tests à chaque déploiement, les équipes peuvent identifier précocement les régressions de performance avant qu'elles n'atteignent la production.

- Itération rapide : Les environnements CI/CD permettent d'adapter rapidement les scénarios de tests de charge à mesure que l'application évolue.

Monitoring et analyse en temps réel

- Métriques avancées : Les tests cloud et CI/CD s'accompagnent souvent de solutions de monitoring comme Grafana et Prometheus, permettant de suivre en temps réel les indicateurs clés (latence, erreurs, saturation des ressources).

- Traçabilité distribuée : Dans des architectures complexes comme les microservices, les outils modernes peuvent isoler les goulots d'étranglement sur des services individuels.

Environnements modernes

- Microservices et containers : Les tests de charge doivent prendre en compte la nature décentralisée des architectures modernes. Des outils comme Locust permettent de tester chaque service individuellement ou l'ensemble de l'architecture. Ces environnements dynamiques présentent des défis spécifiques pour les tests de charge :
  1. Elasticité et scalabilité : Les applications doivent être testées pour s'adapter à des pics de charge imprévisibles tout en optimisant les coûts cloud.
  2. Infrastructure as Code (IaC) : Les outils de test doivent être capables d'évaluer la performance d'une infrastructure dynamique qui peut être modifiée à la volée.

- Edge computing : Avec l'émergence de l'edge, les tests doivent évaluer non seulement la performance des serveurs centraux mais aussi des nœuds de calcul décentralisés.

Edge computing désigne une approche informatique où le traitement des données est effectué à proximité de leur source, plutôt que dans un serveur centralisé ou un cloud distant. Cela signifie que des petits centres de données ou des appareils locaux (appelés "nœuds de calcul") traitent les informations directement sur le terrain.

Checklist

Avant le test

Définir les objectifs des tests :

- [ ] Identifier les KPIs (temps de réponse, débit, taux d'erreurs).
- [ ] Établir des conditions de succès claires.
- [ ] Définir les leviers d'atténuation en cas de problème.
- [ ] Identifier les scénarios clés d'utilisation à simuler (exemple : navigation sur un site, achat en ligne).
- [ ] Estimer la charge attendue (nombre d'utilisateurs, pics de trafic).

Créer des modèles réalistes :

- [ ] Collecter les données réelles d'utilisation (logs, analyses web).
- [ ] Prioriser les scénarios critiques en fonction de leur impact.
- [ ] Décomposer les flux en étapes et valider leur représentativité.

Préparation de l'environnement

- [ ] Mettre en place un environnement de test proche de la production.
- [ ] Configurer les outils de monitoring pour suivre les KPIs en temps réel.
- [ ] Vérifier que les données simulées reflètent la réalité.
- [ ] Configurer les outils nécessaires (JMeter, Locust, K6, etc.).
- [ ] Vérifier l'intégration avec les outils de monitoring (Grafana, Prometheus, etc.).
- [ ] Préparer une API ou des données fictives si nécessaire (ex. : JSONPlaceholder).

Checklist

Rédaction des scripts :

- [ ] Écrire des scripts représentant des comportements réalistes.
- [ ] Tester les scripts avec une faible charge pour s'assurer qu'ils fonctionnent correctement.
- [ ] Inclure des pauses réalistes (wait times) pour simuler le comportement humain.

Pendant le test

Lancement des tests :

- [ ] Configurer la montée en charge progressive (ramp-up)
- [ ] Surveiller en temps réel les métriques clés : latence, taux d'erreurs, saturation.
- [ ] Ajuster la configuration si des anomalies bloquantes apparaissent.

Surveillance des ressources :

- [ ] Vérifier les consommations CPU, mémoire, et bande passante sur les serveurs.
- [ ] Identifier les éventuels goulots d'étranglement.

Après le test

Analyse des résultats

- [ ] Comparer les métriques collectées avec les objectifs initiaux.
- [ ] Identifier les endpoints ou sections du système présentant des problèmes (temps de réponse élevés, erreurs).
- [ ] Analyser les graphiques et rapports générés (ex. : graphiques Grafana, rapports JMeter).

Optimisation

- [ ] Prioriser les corrections à apporter (code, configuration, infrastructure).
- [ ] Ajuster les ressources si nécessaire (ajout de CPU, RAM, équilibrage de charge).

Checklist

Automatisation et suivi :

- [ ] Intégrer les tests dans un pipeline CI/CD pour les exécuter régulièrement.
- [ ] Planifier des tests périodiques pour vérifier la stabilité sur le long terme.

Points à ne pas oublier

- Utiliser un environnement isolé pour éviter d'impacter les utilisateurs réels.
- Documenter les résultats et les ajustements effectués pour les itérations futures.
- Communiquer les résultats aux parties prenantes avec des visuels clairs (exemple : tableau de bord Grafana).

ENTRETIEN AVEC UN EXPERT

NICOLAS PIETRARU

INGÉNIEUR QA À NOUMÉA

Q : Peux-tu me parler de ton parcours dans les tests de performance ?

Mon parcours dans les tests de performance peut être considéré comme relativement classique, bien que marqué par des opportunités variées. Après avoir intégré un master en alternance à l'université, j'ai eu la chance de réaliser mon alternance au sein de Sogeti/Capgemini. Dès mon arrivée, j'ai été introduit à l'univers du test logiciel, avec la possibilité de découvrir plusieurs spécialités. J'ai exprimé mon souhait de toucher à différents aspects du métier, et cette approche m'a permis d'explorer trois grands domaines : le test manuel, l'automatisation et les tests de performance.

Cette polyvalence a débuté pendant mon alternance et s'est poursuivie après mon intégration en CDI chez Capgemini. En fonction des opportunités et des missions, j'ai pu travailler dans ces trois domaines de manière approfondie. J'ai notamment dirigé des équipes d'automaticiens, tout en gardant un lien avec le test manuel – un aspect moins captivant pour moi, mais que je considère indispensable, car il nourrit indirectement l'automatisation et les tests de performance.

L'un des moments forts de mon parcours a été la possibilité de partir à l'étranger, en Allemagne, pour travailler sur un projet de tests de performance chez Volkswagen. Cette expérience m'a particulièrement marqué, car elle touchait à des enjeux technologiques passionnants. Nous testions des applications mobiles liées à la conception des voitures de demain. Cette mission m'a permis de comprendre l'impact concret de notre travail sur des innovations majeures.

Par ailleurs, le fait d'évoluer dans une grande ESN comme Capgemini m'a offert une multitude d'opportunités. J'ai travaillé sur différents sujets, technologies, et méthodes, tout en collaborant avec des équipes internationales, que ce soit en Europe, au Maroc, ou en Inde. Ce contexte m'a enrichi sur le plan professionnel et personnel.

En résumé, intégrer une structure de cette envergure au début de ma carrière a été un formidable tremplin. Cela m'a permis d'acquérir des compétences variées, de m'adapter à des environnements diversifiés et de me préparer à relever de nouveaux défis par moi-même à l'avenir.

Q : Quelles sont les spécificités de ton travail en Nouvelle-Calédonie ?

Travailler en Nouvelle-Calédonie offre un contexte unique, marqué par des structures souvent à échelle réduite, ce qui demande une grande capacité d'adaptation. Les besoins varient en fonction des missions, des acteurs locaux et des projets en cours, et il est important de pouvoir jongler entre différentes approches.

Je me concentre principalement sur l'automatisation et les tests de performance, avec des missions qui peuvent être très fluctuantes. J'ai commencé par des missions assez courtes avant de m'engager sur des projets plus longs, comme celui sur lequel je travaille actuellement, d'une durée d'un an et demi à deux ans.

Comme en métropole, les principales missions concernent des secteurs tels que la banque et le secteur public. J'ai déjà eu des expériences similaires en métropole, notamment avec le groupe BPCE dans le secteur bancaire. Par exemple, lors d'une mission à la BNP, où j'ai travaillé pendant un an, une cellule dédiée aux tests de performance était en place. Tous les projets, qu'il s'agisse d'évolutions majeures ou de nouvelles fonctionnalités, ne partaient pas en production sans le tampon de validation des tests de performance. Ces services permettent aux grandes structures de se prémunir contre les problématiques de performance.

Cependant, ici en Nouvelle-Calédonie, la liberté dans le choix des outils est un véritable atout. On m'a demandé de définir les solutions adaptées, et je privilégie souvent des outils open source comme JMeter, que je maîtrise bien. Cet outil est idéal pour démarrer rapidement et adapter les scénarios aux besoins spécifiques de chaque mission.

Travailler dans ce contexte implique également de s'ajuster aux moyens parfois plus limités qu'ailleurs. Cela se reflète dans la manière dont les tests de performance sont conduits. En général, les campagnes de tests de charge sont ponctuelles :

- On commence par préparer l'environnement,
- Ensuite, on exécute les tests,
- Puis on analyse les résultats,
- On met en place des correctifs si nécessaire,
- Et enfin, on valide avant le déploiement.

Travailler ici, c'est aussi une opportunité unique : on a la chance de faire ce que l'on aime dans un environnement ensoleillé. Cela ajoute une dimension très agréable au quotidien. Bien sûr, les défis restent présents, mais ils sont compensés par cette liberté de choisir les outils et par la satisfaction de contribuer à des projets variés dans un cadre aussi agréable.

Q: As-tu une préférence pour certains outils de test de charge ?

Oui, j'ai travaillé avec plusieurs outils, notamment JMeter, LoadRunner et NeoLoad. Chaque outil a ses forces et ses faiblesses, donc mon choix dépend du projet.

J'apprécie particulièrement JMeter pour sa flexibilité, notamment lorsque je dois partir de zéro et configurer rapidement des scénarios. Parfois, il est nécessaire de coupler JMeter avec des outils comme Postman ou Wireshark, surtout pour récupérer ou analyser des requêtes spécifiques.

Il m'arrive aussi de basculer entre différents outils sur un même projet. Par exemple, si je rencontre un problème de corrélation ou une difficulté à gérer une requête complexe, je peux utiliser NeoLoad pour trouver une solution, puis revenir à JMeter pour finaliser le scénario.

Q: Et concernant Postman ? Est-ce que tu l'utilises aussi pour trouver des requêtes ou tester des codes de retour ?

J'ai utilisé Postman dans un contexte spécifique. L'objectif était de vérifier que les échanges par API fonctionnaient correctement. Dans ce cas précis, on m'a fourni les requêtes nécessaires, et j'ai commencé par implémenter et tester les appels avec Postman, principalement pour m'assurer que les requêtes étaient correctes et conformes.
Ensuite, j'ai transféré ces scénarios vers JMeter pour réaliser les tests de charge.

Globalement, pour des applications web classiques comme des interfaces e-commerce, je préfère enregistrer les échanges entre le client et le serveur directement avec un outil comme Fiddler.

Cela me permet de capturer toutes les requêtes pertinentes, de nettoyer les appels inutiles (par exemple, ceux liés à Google Analytics ou autres trackers tiers) et de m'assurer que seules les actions ayant un impact réel sur l'application sont conservées. Ces actions incluent, par exemple, la consultation d'un panier, d'un solde, ou la réalisation d'une opération spécifique.

Cette méthodologie garantit que les requêtes sélectionnées pour les tests reflètent réellement les performances de l'application et évitent tout biais lié à des appels superflus ou mal filtrés.

Q : Comment fais-tu pour estimer correctement la charge afin qu'elle soit représentative ?

L'estimation de la charge repose principalement sur la volumétrie et peut être abordée de deux manières, en fonction de la maturité de l'application :

1. Pour une application déjà en production :

- Je commence par collecter des données existantes. Cela inclut :
  - Les pics d'activité observés : quels moments de la journée ou de l'année enregistrent le plus grand nombre d'utilisateurs ?
  - Le type d'application : par exemple, une application interne à une entreprise sera généralement utilisée pendant les heures de bureau (8 heures par jour), tandis qu'une application grand public peut avoir des schémas d'utilisation saisonniers.
  - Le contexte métier : pour une plateforme e-commerce, les périodes comme le Black Friday ou Noël sont des moments critiques. Pour des événements spécifiques, comme une plateforme de réservation de concerts, la volumétrie dépendra du nombre de places disponibles et des tendances observées lors de la réservation d'événements similaires très prisés.

2. Pour une application en développement ou sans historique :
Dans ce cas, je me base sur des hypothèses élaborées avec les parties prenantes, en utilisant :

- Les études marketing réalisées par l'entreprise. Ces études permettent d'estimer les volumes attendus, comme le nombre d'utilisateurs potentiels ou le trafic anticipé.
- Les objectifs stratégiques du projet : pourquoi l'application est-elle développée et quelle attractivité est attendue ?
- Une marge de sécurité : je prends systématiquement une estimation légèrement supérieure à celle prévue, pour simuler des situations de forte charge ou des pics imprévus.

Enfin, j'adapte ces estimations en fonction du type de tests souhaités. Par exemple :

- Si l'objectif est de simuler une montée en charge progressive, j'ajuste les volumes progressivement pour observer le comportement du système.
- Si des anomalies ou des comportements inattendus apparaissent, j'affine l'analyse en augmentant ou en modifiant les scénarios de charge.

Cette approche combinée, basée à la fois sur les données existantes et les estimations projetées, garantit que les tests sont alignés avec les réalités du système et les attentes des parties prenantes.

Q: Es-tu capable de déterminer la cause d'un goulot d'étranglement ?

Oui, la détermination des goulots d'étranglement repose sur les outils et méthodes à ma disposition, ainsi que sur une analyse approfondie des données collectées. Voici comment je procède :

1. Analyse initiale avec l'outil d'injection :

L'outil de test de charge (comme JMeter, NeoLoad, ou autre) me fournit des données telles que :

- Les temps de réponse des requêtes, ce qui permet d'identifier si certaines actions ou endpoints dépassent les SLA définis (par exemple, 300 ms pour une API, ou 3 secondes pour une page web).
- Les erreurs détectées dans les requêtes, comme des codes 4xx ou 5xx.
- Des informations globales sur le réseau, pour vérifier si les délais sont liés à des problèmes de connectivité ou de latence.

2. Investigation approfondie avec des outils de monitoring :
Si les temps de réponse sont en deçà des SLA ou si des erreurs apparaissent, je poursuis l'analyse en utilisant des outils d'APM comme Dynatrace, Datadog, ou d'autres systèmes similaires. Ces outils permettent de :

- Identifier les points de saturation au sein de l'architecture (serveurs front-end, back-end, bases de données, caches, etc.).
- Suivre les performances des processus, par exemple sur des environnements JVM où les paramètres comme l'utilisation de la mémoire, la gestion des threads, ou le CPU sont critiques.
- Déployer des sondes sur les serveurs ou directement dans l'application pour capturer des informations précises sur les goulots.

Exemple : Lors d'un test de charge sur une application, j'ai constaté que la JVM d'un serveur était configurée pour utiliser un maximum de 10 Go de RAM, alors que la machine disposait de 50 Go de mémoire.

Cette mauvaise configuration empêchait le serveur de tirer parti de ses ressources complètes, provoquant des ralentissements et une saturation rapide. Après ajustement de la configuration (augmentation de la heap memory), les performances se sont nettement améliorées et la charge a pu être correctement absorbée.

Q: Quelle action recommandes-tu après la détection d'un problème lors d'un test de charge ? Cela implique-t-il d'investiguer ?

Lorsqu'un problème est détecté lors d'un test de charge, la priorité est d'entamer une investigation approfondie.

Cette analyse repose sur les outils à disposition et sur les informations qu'ils permettent de collecter. Les outils de test, comme ceux d'injection, fournissent des données sur les temps de réponse des requêtes, les éventuelles erreurs, et parfois des informations globales sur le réseau.

Ces données permettent de repérer les zones où l'activité est particulièrement élevée ou les anomalies comme des décalages dans la charge.

Par exemple, si l'architecture comporte un load balancer entre plusieurs serveurs back-end, une analyse peut révéler qu'un des serveurs est saturé à 100 % pendant que l'autre reste inutilisé, indiquant une mauvaise répartition de la charge. Ces constats doivent être partagés avec les parties prenantes pour ajuster les configurations.

Ce type d'analyse est particulièrement enrichissant dans le métier de testeur de performance, mais il comporte aussi des défis. Souvent, les problèmes identifiés nécessitent l'intervention de plusieurs équipes, comme celles du réseau, des infrastructures, ou des développeurs. Cela implique de collaborer avec tact pour expliquer les problèmes, comme une configuration incorrecte ou une erreur de développement, sans que cela soit perçu comme une critique personnelle. L'objectif n'est pas de désigner des responsables, mais de résoudre le problème de manière constructive.

Une fois l'investigation terminée, il est nécessaire de présenter une analyse claire et structurée, car c'est dans cette analyse que réside la véritable valeur ajoutée du travail réalisé. Elle doit être adaptée au besoin initial : s'agit-il de valider que l'application peut absorber une charge donnée, ou bien d'optimiser son comportement en fonction des besoins réels ?

Dans certains cas, l'objectif est simplement de vérifier que l'infrastructure peut supporter une charge définie, sans chercher une optimisation approfondie. Dans d'autres, il s'agit de dimensionner l'infrastructure pour qu'elle corresponde précisément aux besoins en charge, tout en minimisant les coûts.

Ces choix stratégiques dépendent aussi de la vision du client et de ses priorités budgétaires.

Par exemple, dans un environnement cloud, où les ressources sont allouées dynamiquement, une mauvaise configuration peut entraîner des coûts excessifs. Les tests doivent donc intégrer ces paramètres et permettre d'identifier des problèmes spécifiques, comme des accumulations de mémoire non gérées correctement, qui peuvent impacter les performances sur le long terme.

L'objectif final est d'apporter une compréhension claire des limites et des comportements de l'application pour permettre au client de prendre des décisions éclairées, qu'il s'agisse d'optimiser l'infrastructure, d'adapter les configurations, ou simplement de valider que le système répond à ses attentes.

Un exemple classique concerne des applications qui supportent très bien un grand nombre d'utilisateurs pendant une courte période, mais qui rencontrent des problèmes lorsqu'un petit nombre d'utilisateurs travaille de manière prolongée, à cause de fuites de mémoire ou d'une mauvaise gestion du garbage collector en Java. Dans ces cas, les performances se dégradent progressivement, jusqu'à l'épuisement des ressources mémoire.

Q: Quels conseils donnerais-tu pour former toute une équipe de test sur les bonnes pratiques des tests de charge, avec l'expérience que tu as ?

Former une équipe aux bonnes pratiques des tests de charge est un processus exigeant, car cela requiert de maîtriser à la fois des aspects techniques, métiers, et organisationnels. Voici mon approche, basée sur mon expérience.

La formation doit être structurée pour permettre à chaque membre de comprendre et de participer à l'ensemble des étapes d'un projet de tests de performance. L'objectif est que chaque testeur puisse gérer une campagne de bout en bout : recueillir les besoins, comprendre les problématiques métiers, concevoir et exécuter les tests, analyser les résultats et produire des rapports clairs. Cette approche globale est importante, car les tests de charge racontent une "histoire" : il faut comprendre pourquoi l'on teste, ce que l'on cherche à vérifier, et comment interpréter les résultats dans leur contexte.

Pour rendre un testeur autonome, il est nécessaire de prévoir une période de deux à trois mois à temps plein, durant laquelle il sera accompagné par un expert. Pendant cette période, la personne apprendra à observer des points spécifiques dans les analyses, à affiner ses scripts, et à poser les bonnes questions. Cela nécessite idéalement un bagage en développement ou en tests logiciels, car certains outils comme K6 demandent des compétences en Java, tandis que des outils plus graphiques comme NeoLoad, LoadRunner ou OctoPerf sont plus accessibles mais nécessitent toujours une appétence pour la technique.

Le rôle d'un testeur de performance exige de jongler entre plusieurs domaines : infrastructure, métier, et test logiciel.

Ce mélange de compétences rend cette spécialité très enrichissante. Un testeur performant acquiert avec le temps une compréhension approfondie des architectures, ce qui peut le mener à évoluer vers des rôles d'architecte logiciel ou de spécialiste en optimisation. Cependant, pour assurer une formation complète, il est également important d'aborder la dimension sécurité. Lors des tests, les configurations de sécurité doivent souvent être adaptées, comme par exemple gérer les tokens d'API qui se renouvellent automatiquement ou isoler les modules de sécurité pour tester leur comportement. Ces aspects nécessitent une rigueur et une technicité accrues.

La formation doit aussi inclure des cas pratiques sur des scénarios complexes, comme :

- Gérer des données partagées entre plusieurs injecteurs : Lors de tests à grande échelle, il peut être nécessaire d'utiliser plusieurs machines pour simuler une charge importante. Cela implique de répartir efficacement les données entre les nœuds et de gérer les échanges dans les deux sens.
- Concevoir des scripts sophistiqués : Par exemple, ajuster la fréquence des requêtes pour des serveurs ayant des capacités différentes, ou gérer des tokens API sans surcharger le serveur d'autorisation.
-

Enfin, il est nécessaire d'utiliser des outils adaptés aux besoins de l'équipe et des projets. Les outils payants comme NeoLoad ou LoadRunner offrent souvent des fonctionnalités avancées qui simplifient la gestion des cas complexes, en particulier lorsqu'il s'agit de synchronisation ou de répartition des charges. Cependant, il est tout aussi important d'apprendre à exploiter au mieux les outils open-source comme JMeter ou Gatling, qui nécessitent plus de configuration, mais permettent une grande flexibilité.

Former une équipe sur les tests de charge ne se limite pas à transmettre des compétences techniques. Cela implique de développer une culture de la collaboration et du pragmatisme, car les tests de performance touchent à des domaines transverses et nécessitent une interaction fréquente avec d'autres équipes (développeurs, réseau, infrastructure, etc.). C'est en combinant une expertise technique solide, une compréhension métier claire et une communication efficace que l'on peut bâtir une équipe capable d'exceller dans les tests de charge.

Q: Comment vois-tu l'avenir du métier de testeur de performance avec les nouvelles technologies qui émergent ?

Le métier de testeur de performance évolue dans un environnement en constante mutation, à la fois en raison des nouvelles technologies et des attentes croissantes des utilisateurs.

D'un côté, la performance doit toujours répondre à une exigence de timing précis : ni trop rapide, ni trop lente. Par exemple, une page web qui s'affiche instantanément peut créer une confusion chez l'utilisateur, donnant l'impression que rien ne s'est passé, alors qu'un délai trop long risque de frustrer les utilisateurs et de nuire à l'expérience client. Ce paradoxe illustre bien les enjeux actuels : fournir une réponse adaptée au contexte et à l'objectif du service. Dans des cas spécifiques, comme les services publics, l'utilisateur peut tolérer un délai plus long, car le besoin est impératif, mais pour des services commerciaux, le temps de réponse peut avoir un impact direct sur le chiffre d'affaires.

Ce qui est fascinant, c'est que l'avenir du métier ne se limite pas à perfectionner les outils ou à accélérer les tests. L'un des enjeux majeurs sera de s'adapter à l'évolution des interfaces d'accès à l'information.

Aujourd'hui, nous utilisons principalement des sites web, mais demain, ces interfaces pourraient être remplacées par d'autres formats, comme des agents conversationnels ou des systèmes intégrés à des objets connectés.

Cette évolution impliquera de nouveaux types de tests et de méthodologies.

En parallèle, les tests de performance deviendront de plus en plus fréquents et réguliers, en partie grâce à l'automatisation et aux capacités des outils modernes. L'intelligence artificielle jouera un rôle, en particulier dans l'analyse des résultats. Déjà, avec des approches de machine learning, il est possible de former des modèles pour analyser des courbes de temps de réponse et détecter des anomalies. Ces modèles, qui s'appuient sur des exemples préalablement classés comme acceptables ou non, permettent de gagner du temps dans l'analyse initiale et d'identifier plus rapidement les goulots d'étranglement.

Q: Quels outils recommandes-tu d'explorer, pour des débutants comme pour des profils plus expérimentés ?

Pour les débutants comme pour les experts, il existe une variété d'outils et de méthodologies qui permettent d'évoluer dans les tests de performance. Le choix des outils dépend souvent du niveau d'expérience, mais également des objectifs de montée en compétence et des besoins des projets.

Pour ceux qui débutent, il est souvent judicieux de commencer par des outils graphiques et conviviaux comme NeoLoad ou LoadRunner. Ces outils permettent de rapidement construire des scripts, d'exécuter des tests, et d'obtenir des résultats sans avoir à entrer immédiatement dans des détails techniques complexes. Ils offrent une courbe d'apprentissage relativement douce, ce qui permet aux nouveaux de se concentrer d'abord sur les fondamentaux : comprendre les concepts de performance, les indicateurs clés, et l'importance des scénarios réalistes.

Ces outils sont excellents pour apprendre les bases, mais ils peuvent aussi masquer certains aspects techniques, ce qui limite l'apprentissage en profondeur.

Une fois que cette première étape est franchie, je recommande vivement de se tourner vers des outils plus techniques, comme JMeter ou K6. Ces outils open-source nécessitent plus d'efforts pour être maîtrisés, mais ils offrent une opportunité d'apprendre à un niveau beaucoup plus profond. En travaillant avec eux, on est obligé de comprendre les protocoles, les architectures, et les mécanismes sous-jacents qui impactent les performances. Ce processus, parfois exigeant, est incroyablement formateur. Il développe non seulement une meilleure compréhension des outils eux-mêmes, mais aussi des systèmes testés, ce qui est essentiel pour progresser en tant que testeur de performance.

Pour les testeurs plus expérimentés, il est important de se concentrer sur l'intégration des outils de tests de performance avec des systèmes de supervision et de visualisation en temps réel, comme Grafana couplé à Prometheus ou InfluxDB. Ces outils permettent de surveiller les métriques serveur en parallèle des exécutions, de générer des tableaux de bord personnalisés, et de mieux comprendre comment les performances de l'infrastructure réagissent à la charge. Être capable de visualiser en temps réel ce qui se passe sur les serveurs pendant une campagne de tests est non seulement satisfaisant, mais aussi essentiel pour mener des analyses précises et convaincantes.

De manière générale, il est important d'adopter une mentalité exploratoire. Les outils plus avancés et techniques demandent souvent plus de temps et d'efforts, mais c'est ce processus qui fait la différence entre quelqu'un qui exécute des tests et quelqu'un qui comprend réellement ce qu'il fait.

Par exemple, avec NeoLoad, il est possible de créer rapidement des scénarios performants grâce à son interface intuitive. Cependant, si on ne sait pas pourquoi ou comment cela fonctionne en arrière-plan, cela limite la capacité à résoudre les problèmes ou à ajuster les scénarios pour des cas spécifiques.

En revanche, des outils comme JMeter ou K6, qui demandent une configuration plus manuelle, obligent à s'immerger dans la technique, ce qui enrichit considérablement les compétences à long terme.

Pour les seniors, je recommande de se pencher sur des méthodologies plus avancées et émergentes, comme l'intégration des tests de performance dans des pipelines DevOps ou l'exploration des capacités des intelligences artificielles. Les modèles de machine learning, par exemple, peuvent être utilisés pour analyser des courbes de temps de réponse et identifier automatiquement des anomalies ou des tendances. Ces approches, bien que plus complexes, permettent de passer moins de temps sur les analyses répétitives et de se concentrer sur les tâches à plus forte valeur ajoutée, comme l'optimisation des architectures ou la collaboration stratégique avec d'autres équipes.

En résumé, il faut progresser par étapes : commencer par des outils accessibles, puis explorer des solutions plus techniques pour approfondir ses compétences, et enfin se tourner vers des méthodologies avancées et des outils d'intégration pour rester à la pointe. Le chemin peut être exigeant, mais chaque difficulté surmontée contribue à bâtir une expertise durable et une véritable compréhension du métier.

TEASING

Partagez-nous vos retours !

Ce magazine a été conçu pour vous accompagner dans l'amélioration de vos tests. Vos impressions, idées et questions sont précieuses pour que Le Mag du Testeur continue de grandir et de s'adapter à vos besoins.

N'hésitez pas à m'écrire à **feedback@lemagdutesteur.fr** pour échanger autour de vos expériences et de vos suggestions !

Un avant-goût du prochain numéro...
Cypress contre Selenium : Playwright les surpasse-t-il vraiment ? Et où se cache Robot Framework dans tout ça ? Le verdict... dans notre prochain numéro.

Merci pour votre fidélité et à très bientôt dans les pages de Le Mag du Testeur !