Université Bretagne Sud
IUT de Vannes
Département Informatique

BUT 3 – R5.A.04. Qualité algorithmique
TP 3 : tests de charge pour l'évaluation des performances et de la disponibilité

Nous allons utiliser dans ce TP l'outil Gatling[1] pour lancer des tests de charge. Il existe plusieurs outils de tests de charge. Le choix dans ce TP s'est porté sur Gatling parce qu'il comporte une bonne documentation, il génère du code en Java, un langage que vous maîtrisez bien et c'est développé à la base comme projet open source par une équipe française.
Dans cet outil, un test de charge est exécutée comme une « simulation », qui va créer un grand nombre d'utilisateurs virtuels qui vont chacun envoyer des requêtes HTTP à une application ayant un front-end ou une API Web. L'objectif est de mesurer les performances (temps de réponse moyen, débit, …) et la disponibilité (taux d'échec des requêtes, …) et s'assurer que son application (ou backend) répond bien à une forte charge d'utilisation ou à des fluctuations de trafic des requêtes.

Nous allons personnaliser un petit scénario d'envoi de requêtes et l'exécuter sur deux phases : 1) montée de charge (*ramp-up*) puis 2) charge soutenue (*sustained load*), avec un nombre d'utilisateurs virtuels précis N créés à la seconde. Lors de la phase 1, l'outil va créer 1 utilisateur par seconde jusqu'à N/sec pendant une certaine période. Puis pendant la seconde phase, il va créer N utilisateurs par seconde pendant une certaine période. Chaque utilisateur enverra toutes les requêtes HTTP décrites dans le scénario. Dans certains tests de charge, on précède la phase 1 par une phase d'échauffement (*warm-up*), qui consiste à envoyer des requêtes en faible nombre pour provoquer d'abord le chargement de code en mémoire et les allocations de ressources nécessaires à l'exécution de l'application à tester. Parfois, on met en place une dernière phase descendante (*ramp-down*) pour analyser le retour à une situation de faible charge. Certains tests de charge analysent le comportement d'une application (quels types de requêtes échouent pour quelles raisons). Dans ce TP, on s'intéresse à la mesure d'attributs de qualité : performance (efficacité) et disponibilité (fiabilité).

Télécharger et décompresser l'archive ZIP donnée dans la page Moodle du cours.

Il y a deux façons d'utiliser Gatling : 1) en se servant du *recorder*, pour enregistrer une session de navigation via un navigateur Web (une config particulière d'un proxy sur le navigateur doit être effectuée au préalable). Le *recorder* va ensuite générer le code (Java) pour reproduire la même session de navigation, 2) en éditant la classe de test de charge déjà générée et fournie (dans le dossier `user-files/simulations/` de l'archive), en mettant à jour les URL et les paramètres des requêtes HTTP.

Le test de charge doit être réalisée sur une API REST exposée par un backend déjà démarrée sur votre machine (ou sur une machine distante que vous contrôlez, qui ne risque pas de vous bannir, suite à une forte charge d'utilisation).

Lancer le backend développé dans le cours sur le style d'architecture à microservices : orchestration de services `users` et `locations`.

Adapter la classe Java du test puis lancer un test en exécutant le script `bin/gatling.sh`

1 https://gatling.io/

Choisir l'option d'exécution d'une simulation localement (option 1 : [1] *Run the Simulation locally*) puis choisir la simulation à exécuter (au départ, il n'y a qu'une seule, celle portant le nom de la classe Java). Vous pouvez donner une description à la simulation qui sera intégrée dans les résultats du test ou simplement cliquer sur la touche « Entrée » pour l'ignorer.

A la fin de l'exécution du test, consulter les résultats dans la page Web générée dans le dossier `results/`

Parcourez les différentes rubriques. Les résultats sont fortement dépendants de l'infrastructure où vous avez démarré votre application/backend. Dans une infra différente, les résultats ne seront pas les mêmes. Mais ces résultats vous donnent une première estimation de la qualité à l'exécution de votre application. Dans certains cas, cela nous permet d'identifier des goulots d'étranglement (*bottleneck*) : des traitements associées à des requêtes bien précises, avec des données bien précises. L'objectif derrière est de trouver des solutions pour que l'application/backend résiste mieux, soit en revoyant l'algorithmie de ces traitements ou bien certains choix d'implémentation (utilisation de services externes parfois trop lents à répondre). Dans certains cas, la réplication de certains services peut s'avérer une très bonne solution pour éliminer ces goulots d'étranglement.

Si les tests que vous avez lancés ne provoquent pas d'échec dans les requêtes HTTP, provoquer l'échec d'un certain nombre de requêtes, en rallongeant par exemple la durée de traitement de celles-ci, ce qui impliquera un dépassement de délai (*timeout* : un type d'échec possible).
Pour ce faire, vous pouvez mettre dans les méthodes de la couche service, qui sont appelées lors du traitement des requêtes HTTP, un `Thread.sleep(61000)` dans un `if(Math.random() < 0.5)` pour simuler des traitements lents aléatoires.

Trouver le point de rupture de votre application/backend (sans la provocation des *timeouts*) ; le point de rupture étant le nombre d'utilisateurs virtuels minimal (créés par seconde) qui provoque un taux d'échec des requêtes non-nul.

Si vous avez lancé votre application/backend comme un ensemble de services orchestrés, créer plusieurs répliques de ces services (varier le nombre 2, 4, ...) et relancer les tests pour voir si les résultats sont meilleurs.

Dans une orchestration Docker Compose, pour répliquer les services, il suffit simplement d'ajouter l'élément suivant dans votre configuration .yml

```
services:
  backend:
    image: ...
    ...
    deploy:
      replicas: 2
```

Reporter dans un document les différents résultats (les captures d'écran les plus pertinentes) et les modifications faites sur votre code/orchestration. Déposer ce document sur Moodle.