



Ecosystème Microservices - Spring Framework - Spring Boot - Spring -Cloud



Marwén Saidi

mail : marwen.saidi@orange.com

LinkedIn : <https://www.linkedin.com/in/marwensaidi/>

Github : <https://github.com/marwensaidi>

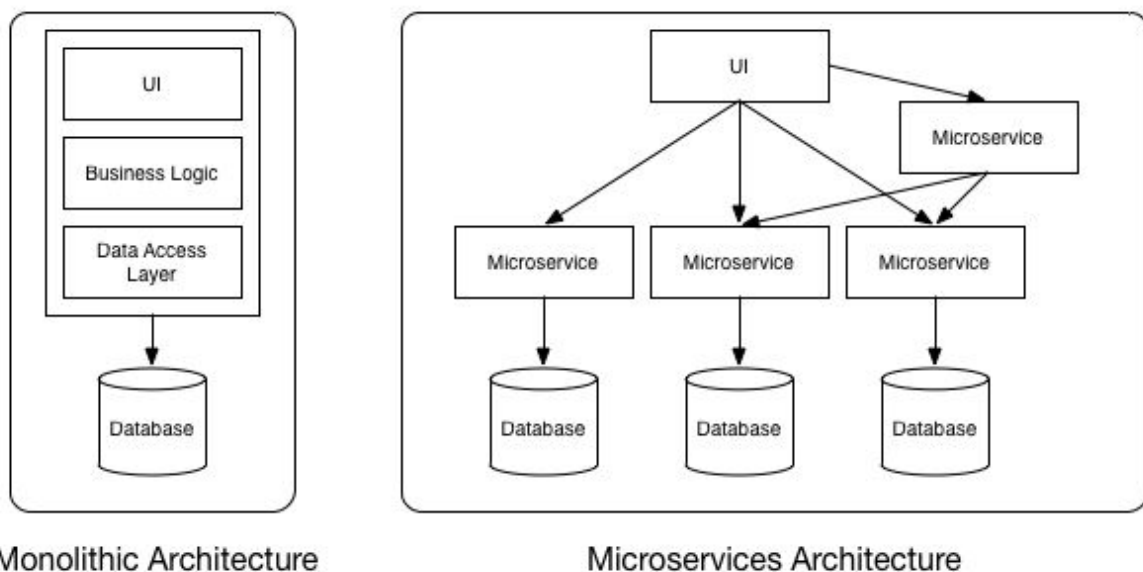
Microservices - Introduction

Les microservices sont le dernier mot à la mode de l'industrie et tout le monde semble en parler, d'une manière ou d'une autre. Voyons ce que sont les microservices? Dans ce tutoriel, nous allons essayer de comprendre la définition, les concepts et les principes des microservices.

1. Définition des microservices

Aujourd'hui, les microservices sont l'un des modèles d'architecture de plus en plus populaires à côté de SOA (Services Oriented Architecture). Si vous suivez les tendances de l'industrie, alors vous vous rendez compte qu'aujourd'hui les entreprises ne sont plus intéressées par le développement de grandes applications pour gérer leurs fonctions commerciales de bout en bout comme elles le faisaient il y a quelques années, elles optent plutôt pour des applications rapides et agiles qui coûtent cher eux aussi moins d'argent.

Les microservices aident à briser les limites des grandes applications et à créer des systèmes plus petits logiquement indépendants à l'intérieur du système. Par exemple, en utilisant Amazon AWS, vous pouvez créer une application cloud avec un minimum d'effort. C'est un bon exemple de ce que les microservices peuvent faire.



Architecture monolithique vs MicroServices

Comme vous pouvez le voir dans le diagramme ci-dessus, chaque microservice possède sa propre couche métier et sa propre base de données. Ce faisant, les modifications apportées à un microservice n'ont aucun impact sur les autres.

En général, les microservices communiquent entre eux à l'aide de protocoles légers largement adoptés, tels que HTTP et REST, ou de protocoles de messagerie, tels que JMS ou AMQP. Dans des scénarios spécifiques, ils peuvent également opter pour des protocoles plus spécialisés.

2. Principes des microservices

Examinons maintenant les principes "indispensables" d'un microservice.

Principe de responsabilité unique

Le principe de responsabilité unique est l'un des principes définis dans le cadre du modèle de conception SOLID. Cela implique qu'une unité, que ce soit une classe, une fonction ou un microservice, devrait avoir une et une seule responsabilité.

À aucun moment, un microservice ne doit avoir plus d'une responsabilité.

Construit autour des capacités métier

Les microservices doivent se concentrer sur certaines fonctions de l'entreprise et s'assurer que cela aide à faire avancer les choses. Un microservice ne doit jamais s'empêcher d'adopter une pile technologique appropriée ou un stockage de base de données backend qui est le plus approprié pour résoudre l'objectif commercial.

C'est souvent la contrainte lorsque nous concevons des applications monolithiques où nous essayons de résoudre plusieurs solutions métier avec quelques compromis dans certains domaines. Les microservices vous permettent de choisir ce qui convient le mieux au problème à résoudre.

Vous le construisez, vous le possédez!

Un autre aspect important d'une telle conception est lié aux responsabilités avant et après le développement. Dans une grande organisation, une équipe développe généralement l'emplacement de l'application et, après quelques sessions de transfert de connaissances, transmet le projet à l'équipe de maintenance. Dans les microservices, l'équipe qui construit le service en est propriétaire et est responsable de sa maintenance à l'avenir.

Vous le construisez, vous le possédez !!

Cette propriété met les développeurs en contact avec le fonctionnement quotidien de leurs logiciels et ils comprennent mieux comment leur produit construit est utilisé par les clients dans le monde réel.

Automatisation de l'infrastructure

La préparation et la construction d'une infrastructure pour les microservices est un autre besoin très important. Un service doit être déployable de manière indépendante et doit regrouper toutes les dépendances, y compris les dépendances de bibliothèque, et même les environnements d'exécution tels que les serveurs Web et les conteneurs ou les machines virtuelles qui font abstraction des ressources physiques.

L'une des principales différences entre les microservices et SOA réside dans leur niveau d'autonomie. Alors que la plupart des implémentations SOA fournissent une abstraction au niveau du service, les microservices vont plus loin et font abstraction de l'environnement de réalisation et d'exécution.

Dans les développements d'applications traditionnels, nous construisons un WAR ou un EAR, puis nous le déployons dans un serveur d'applications JEE, comme avec JBoss, WebLogic, WebSphere, etc. Nous pouvons déployer plusieurs applications dans le même conteneur JEE. Dans un scénario idéal, dans l'approche microservices, chaque microservice sera construit comme un gros Jar, intégrant toutes les dépendances et exécuté en tant que processus Java autonome.

Conception pour l'échec

Un microservice doit être conçu en tenant compte des cas de défaillance. Que faire si le service échoue ou tombe en panne pendant un certain temps. Ce sont des questions très importantes et doivent être résolues avant le début du codage réel - pour estimer clairement comment les pannes de service affecteront l'expérience utilisateur.

Fail fast est un autre concept utilisé pour construire des systèmes résilients et tolérants aux pannes. Cette philosophie préconise des systèmes qui s'attendent à des défaillances par rapport à des systèmes de construction qui ne tombent jamais en panne. Étant donné que les services peuvent échouer à tout moment, il est important de pouvoir détecter rapidement les pannes et, si possible, de restaurer automatiquement le service.

Les applications de microservices mettent beaucoup l'accent sur la surveillance en temps réel de l'application, en vérifiant à la fois les éléments architecturaux (combien de demandes par seconde la base de données reçoit-elle) et les mesures pertinentes pour l'entreprise (comme le nombre de commandes par minute reçues). La surveillance sémantique peut fournir un système d'alerte précoce en cas de problème qui incite les équipes de développement à effectuer un suivi et à enquêter.

3. Avantages des microservices

- Les microservices offrent un certain nombre d'avantages par rapport aux architectures monolithiques à plusieurs niveaux traditionnelles. Énumérons-les:
- Avec les microservices, les architectes et les développeurs peuvent choisir des architectures et des technologies adaptées à leur objectif pour chaque microservice (architecture polyglotte). Cela donne la flexibilité nécessaire pour concevoir des solutions mieux adaptées de manière plus rentable.
- Les services étant assez simples et de plus petite taille, les entreprises peuvent se permettre d'expérimenter de nouveaux processus, algorithmes, logique métier, etc. Il permet aux entreprises de faire de l'innovation de rupture en offrant la possibilité d'expérimenter et d'échouer rapidement.
- Les microservices permettent de mettre en œuvre une évolutivité sélective, c'est-à-dire que chaque service peut être mis à l'échelle indépendamment à la hausse ou à la baisse et le coût de la mise à l'échelle est comparativement inférieur à l'approche monolithique.
- Les microservices sont des modules de déploiement autonomes et indépendants permettant la substitution d'un microservice par un autre microservice similaire, lorsque le second ne fonctionne pas selon nos besoins. Cela aide à prendre les bonnes décisions d'achat contre construction, qui sont souvent un défi pour de nombreuses entreprises.
- Les microservices nous aident à construire des systèmes de nature organique (les systèmes organiques sont des systèmes qui se développent latéralement sur une période de temps en y ajoutant de plus en plus de fonctions). Parce que les microservices sont tous des services gérables indépendamment, cela permet d'ajouter de plus en plus de services au fur et à mesure que le besoin se fait sentir avec un impact minimal sur les services existants.
- Les changements technologiques sont l'un des obstacles au développement de logiciels. Avec les microservices, il est possible de modifier ou de mettre à niveau la technologie pour chaque service individuellement plutôt que de mettre à niveau une application entière.
- À mesure que les microservices empaquetent l'environnement d'exécution du service avec le service lui-même, cela permet de faire coexister plusieurs versions du service dans le même environnement.
- Enfin, les microservices permettent également le développement d'équipes agiles plus petites et plus ciblées. Les équipes seront organisées en fonction des limites des microservices.

4. Conclusion

Souvent, les vraies conséquences de vos décisions architecturales ne sont évidentes que plusieurs années après que vous les ayez prises.

Dans cet article, je n'ai répertorié que quelques points positifs sur les microservices qui ont été vus dans de nombreuses organisations à ma connaissance limitée. Une application monolithique, soutenue par une conception solide et des codeurs brillants, peut également s'avérer une bonne décision et le produit peut rester assez longtemps pour soutenir la décision.

De même avec les microservices, de mauvaises décisions de conception s'avéreront coûteuses. Ils peuvent sembler simplifier les composants, mais ils peuvent compliquer la communication entre eux, ce qui est plus difficile à contrôler et à gérer.

Enfin, c'est une bonne conception et une équipe qualifiée qui vous apporteront la victoire. Une équipe moins qualifiée créera toujours un système médiocre et il est très difficile de dire si les microservices réduisent le désordre dans ce cas ou l'aggravent.

Je suggérerai de commencer par une conception d'application monolithique, et lorsque vous sentez que cela rend le système complexe, essayez les microservices pour vérifier s'ils rendent l'application moins complexe. De cette façon, vous aurez une décision plus éclairée et meilleure.

Spring Annotation

Découvrez les annotations de spring les plus utilisées. Dans ce didacticiel, nous couvrirons brièvement les annotations importantes fournies par spring core pour définir des beans et créer des configurations de contexte d'application complexes.

1. Annotations Bean

Une liste d'annotations donnée est utilisée pour définir les beans au spring et pour contrôler leurs préférences d'enregistrement dans le contexte de l'application.

Annotation	Description
@Bean	Une annotation au niveau de la méthode utilisée pour déclarer un spring bean. Lorsque la configuration exécute une méthode annotée, elle enregistre la valeur de retour en tant que bean dans une BeanFactory.

	<p>Par défaut, le nom du bean sera le même que le nom de la méthode. Pour personnaliser le nom du bean, utilisez son attribut «nom» ou «valeur».</p> <pre>@Bean EmployeeService employeeService() { return new EmployeeServiceImpl(); }</pre>
@Component	<p>Indique qu'une classe annotée est un «composant» et sera détectée automatiquement lors de l'utilisation de la configuration basée sur les annotations et de l'analyse du chemin de classe.</p> <p>Pour utiliser cette annotation, appliquez-la sur la classe comme ci-dessous:</p> <pre>@Component public class EmployeeDAOImpl implements EmployeeDAO { ... }</pre>
@Repository	<p>Une spécialisation de l'annotation @Component. En plus d'importer les classes DAO annotées dans le conteneur DI, cela rend également les exceptions non vérifiées (levées à partir des méthodes DAO) éligibles pour la traduction dans Spring DataAccessException.</p> <pre>@Repository public class EmployeeDAOImpl implements EmployeeDAO { //... }</pre>
@Service	<p>Une spécialisation de l'annotation @Component. Cela indique qu'une classe est une «façade de services» ou quelque chose de similaire.</p> <pre>@Service ("employeeManager") public class EmployeeManagerImpl implements EmployeeManager</pre>

	<pre> { @Autowired EmployeeDAO dao; ... } </pre>
@Controller	<p>Une spécialisation du @Component pour annoter les contrôleurs (par exemple un contrôleur Web). Il est utilisé en combinaison avec des méthodes de gestion annotées basées sur l'annotation RequestMapping.</p> <pre> @Controller ("employeeController") public class EmployeeController { ... } </pre>
@Qualifier	<p>Lors du câblage automatique, si plusieurs bean du même type sont disponibles dans le conteneur, le conteneur lèvera une exception d'exécution. Pour résoudre ce problème, nous devons indiquer spécifiquement à spring quel bean doit être injecté en utilisant cette annotation.</p> <p>Dans l'exemple donné, s'il y a deux beans de type Repository alors au runtime, le bean avec le nom «fsRepository» sera injecté.</p> <pre> public class EmployeeService { @Autowired @Qualifier("fsRepository") private Repository repository; } </pre>
@Autowired	<p>Marque un constructeur, un champ, une méthode de définition ou une méthode de configuration comme étant câblé automatiquement par l'injection de dépendances. Nous pouvons indiquer si la dépendance annotée est requise (obligatoire à remplir) ou non en utilisant l'attribut «requis». Par défaut, sa valeur est "true".</p>

	<pre>public class EmployeeService { @Autowired private EmployeeDao dao; }</pre>
@Required	<p>L'auto-câblage du bean par défaut vérifie uniquement que la dépendance a été définie. Il ne vérifie pas si la valeur attribuée est nulle ou non. En utilisant @Required, nous pouvons vérifier si les valeurs définies ne sont pas nulles. Il est désormais obsolète.</p>
@Value	<p>Applicable au niveau du champ ou du paramètre de méthode / constructeur, et indique une expression de valeur par défaut pour l'argument concerné.</p> <pre>public class SomeService { @Value("\${ENV}") private String environment; }</pre>
@Lazy	<p>Indique si un bean doit être initialisé paresseusement. Par défaut, dans Spring DI, une initialisation hâtive se produira.</p> <p>Lorsqu'elle est appliquée sur n'importe quel bean, l'initialisation de ce bean n'aura pas lieu tant qu'elle ne sera pas référencée par un autre bean ou explicitement récupérée depuis le BeanFactory englobant.</p> <pre>public class SomeService { @Autowired @Lazy private RemoteService remoting; }</pre>
@DependsOn	<p>Lors de l'analyse des composants, il est utilisé pour spécifier les beans dont dépend le bean actuel. Il est garanti que les beans spécifiés seront créés par le conteneur avant ce bean.</p> <pre>public class SomeService { @Autowired</pre>

	<pre> @DependsOn ("pingService") private RemoteService remoting; } </pre>
@Lookup	<p>Indique une méthode comme méthode de «recherche». Il est préférable de l'utiliser pour injecter un bean à portée prototype dans un bean singleton.</p> <pre> @Component @Scope("prototype") public class AppNotification { //prototype-scoped bean } @Component public class NotificationService { @Lookup public AppNotification getNotification() { //return new AppNotification(); } } </pre>
@Primary	<p>Indique qu'un bean doit avoir la préférence lorsque plusieurs candidats sont qualifiés pour le câblage automatique d'une dépendance à valeur unique.</p> <p>Lorsque vous n'utilisez pas @Primary, nous pouvons avoir besoin de fournir l'annotation @Qualifier pour injecter correctement les beans.</p> <p>Dans l'exemple donné, lorsque FooRepository sera câblé automatiquement, l'instance de HibernateFooRepository sera injectée - jusqu'à ce que l'annotation @Qualifier soit utilisée.</p> <pre> @Component public class JdbcFooRepository extends FooRepository { } @Primary @Component public class HibernateFooRepository </pre>

	<pre>extends FooRepository { }</pre>
@Scope	indique le nom d'une étendue à utiliser pour les instances du type annoté. Au spring 5, les beans peuvent être dans l'une des six portées, à savoir singleton, prototype, requête, session, application et websocket.

2. Context configuration annotations

Ces annotations aident à lier les différents beans ensemble pour former le contexte d'application d'exécution.

Annotation	Description
@ComponentScan	<p>@ComponentScan avec @Configuration est utilisé pour activer et configurer l'analyse des composants. Par défaut, si nous ne spécifions pas le chemin, il scanne le package actuel et tous ses sous-packages pour les composants.</p> <p>À l'aide de l'analyse des composants, spring peut analyser automatiquement toutes les classes annotées avec les annotations stéréotypées @Component, @Controller, @Service et @Repository et les configurer avec BeanFactory.</p> <pre>@Configuration @ComponentScan(basePackages = {com.howtodoinjava.data.jpa}) public class JpaConfig { }</pre>
@Configuration	<p>Indique qu'une classe déclare une ou plusieurs méthodes @Bean et peut être traitée par le conteneur pour générer des définitions de bean lorsqu'elle est utilisée avec @ComponentScan.</p> <pre>@Configuration @ComponentScan(basePackages = {com.howtodoinjava.data.jpa}) public class JpaConfig {</pre>

	<pre>}</pre>
@Profile	<p>Indique qu'un composant est éligible pour l'enregistrement de bean lorsqu'un ou plusieurs profils spécifiés sont actifs. Un profil est un groupement logique nommé de beans, par ex. dev, prod etc.</p> <pre>@Configuration @ComponentScan(basePackages = {com.howtodoinjava.data.jpa}) public class JpaConfig { }</pre>
@Import	<p>Indique une ou plusieurs classes de composants à importer - généralement des classes @Configuration. Les définitions @Bean déclarées dans les classes @Configuration importées doivent être accessibles à l'aide de l'injection @Autowired.</p> <pre>@Configuration @ComponentScan(basePackages = {com.howtodoinjava.data.jpa}) public class JpaConfig { }</pre>
@ImportResource	<p>Indique une ou plusieurs ressources contenant des définitions de bean à importer. Il est utilisé pour les définitions de bean XML, tout comme @Import est utilisé pour la configuration java à l'aide de @Bean.</p> <pre>@Configuration @ImportResource({ "spring-context.xml" }) public class ConfigClass { }</pre>

Spring Boot Annotation

Les annotations de spring boot sont principalement placées dans les packages `org.springframework.boot.autoconfigure` et `org.springframework.boot.autoconfigure.condition`. Découvrons quelques annotations de démarrage à ressort fréquemment utilisées ainsi que celles qui fonctionnent en coulisse.

1. @SpringBootApplication

Spring Boot concerne principalement la configuration automatique. Cette auto-configuration est effectuée par analyse des composants, c'est-à-dire en trouvant toutes les classes dans classpath pour l'annotation `@Component`. Cela implique également l'analyse de l'annotation `@Configuration` et l'initialisation de quelques beans supplémentaires.

L'annotation `@SpringBootApplication` permet toutes les choses en une seule étape. Il active les trois fonctionnalités:

`@EnableAutoConfiguration`: activer le mécanisme de configuration automatique

`@ComponentScan`: activer l'analyse `@Component`

`@SpringBootConfiguration`: enregistrez des beans supplémentaires dans le contexte

La classe java annotée avec `@SpringBootApplication` est la classe principale d'une application Spring Boot et l'application démarre à partir d'ici.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

2. @EnableAutoConfiguration

Cette annotation permet la configuration automatique du contexte d'application Spring, en essayant de deviner et de configurer les beans dont nous aurons probablement besoin en fonction de la présence de classes prédéfinies dans le chemin de classe.

Par exemple, si nous avons tomcat-embedded.jar sur le chemin de classe, nous voudrions probablement une TomcatServletWebServerFactory.

Comme cette annotation est déjà incluse via `@SpringBootApplication`, l'ajouter à nouveau sur la classe principale n'a aucun impact. Il est également conseillé de n'inclure cette annotation qu'une seule fois via `@SpringBootApplication`.

Les classes de configuration automatique sont des beans de configuration Spring standard. Ils sont localisés à l'aide du mécanisme `SpringFactoriesLoader` (associé à cette classe). En général, les beans de configuration automatique sont des beans `@Conditional` (utilisant le plus souvent les annotations `@ConditionalOnClass` et `@ConditionalOnMissingBean`).

3. `@SpringBootConfiguration`

Il indique qu'une classe fournit la configuration de l'application Spring Boot. Il peut être utilisé comme une alternative à l'annotation `@Configuration` standard de Spring afin que la configuration puisse être trouvée automatiquement.

L'application ne doit jamais inclure qu'une seule `@SpringBootConfiguration` et la plupart des applications Spring Boot idiomatiques l'hériteront de `@SpringBootApplication`.

La principale différence réside dans les deux annotations: `@SpringBootConfiguration` permet de localiser automatiquement la configuration. Cela peut être particulièrement utile pour les tests unitaires ou d'intégration.

4. `@ImportAutoConfiguration`

Il importe et applique uniquement les classes de configuration automatique spécifiées. La différence entre `@ImportAutoConfiguration` et `@EnableAutoConfiguration` est que les tentatives ultérieures de configurer les beans qui se trouvent dans le chemin de classe lors de l'analyse, alors que `@ImportAutoConfiguration` n'exécute que les classes de configuration que nous fournissons dans l'annotation.

Nous devrions utiliser `@ImportAutoConfiguration` lorsque nous ne voulons pas activer la configuration automatique par défaut.

```
@ComponentScan("path.to.your.controllers")
@ImportAutoConfiguration({WebMvcAutoConfiguration.class
    ,DispatcherServletAutoConfiguration.class
    ,EmbeddedServletContainerAutoConfiguration.class
    ,ServerPropertiesAutoConfiguration.class
    ,HttpMessageConvertersAutoConfiguration.class})
public class App
{
```

```

    public static void main(String[] args)
    {
        SpringApplication.run(App.class, args);
    }
}

```

5. @AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder

Nous pouvons utiliser les annotations `@AutoConfigureAfter` ou `@AutoConfigureBefore` si notre configuration doit être appliquée dans un ordre spécifique (avant ou après).

Si nous voulons commander certaines auto-configurations qui ne devraient pas avoir de connaissance directe les unes des autres, nous pouvons également utiliser `@AutoConfigureOrder`. Cette annotation a la même sémantique que l'annotation `@Order` normale, mais fournit un ordre dédié pour les classes de configuration automatique.

```

@Configuration
@AutoConfigureAfter(CacheAutoConfiguration.class)
@ConditionalOnBean(CacheManager.class)
@ConditionalOnClass(CacheStatisticsProvider.class)
public class RedissonCacheStatisticsAutoConfiguration
{
    @Bean
    public RedissonCacheStatisticsProvider
redissonCacheStatisticsProvider(){
        return new RedissonCacheStatisticsProvider();
    }
}

```

6. Condition Annotations

Toutes les classes de configuration automatique ont généralement une ou plusieurs annotations `@Conditionnelles`. Il permet d'enregistrer un bean uniquement lorsque la condition est remplie. Voici quelques annotations conditionnelles utiles à utiliser.

5.1. @ConditionalOnBean and @ConditionalOnMissingBean

Ces annotations permettent d'inclure un bean en fonction de la présence ou de l'absence de beans spécifiques.

Son attribut value est utilisé pour spécifier les beans par type ou par nom. L'attribut de recherche nous permet également de limiter la hiérarchie ApplicationContext à prendre en compte lors de la recherche de beans.

L'utilisation de ces annotations au niveau de la classe empêche l'enregistrement de la classe @Configuration en tant que bean si la condition ne correspond pas.

Dans l'exemple ci-dessous, le bean JpaTransactionManager ne sera chargé que si un bean de type JpaTransactionManager n'est pas déjà défini dans le contexte de l'application.

```
@Bean
@ConditionalOnMissingBean(type = "JpaTransactionManager")
JpaTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory)
{
    JpaTransactionManager transactionManager = new
JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory);
    return transactionManager;
}
```

5.2. @ConditionalOnClass and @ConditionalOnMissingClass

Ces annotations permettent d'inclure des classes de configuration en fonction de la présence ou de l'absence de classes spécifiques. Notez que les métadonnées d'annotation sont analysées à l'aide du module Spring ASM, et même si une classe peut ne pas être présente au moment de l'exécution, vous pouvez toujours faire référence à la classe dans l'annotation.

Nous pouvons également utiliser l'attribut value pour faire référence à la classe réelle ou l'attribut name pour spécifier le nom de la classe en utilisant une valeur String.

La configuration ci-dessous ne créera EmbeddedAcmeService que si cette classe est disponible à l'exécution et qu'aucun autre bean du même nom n'est présent dans le contexte de l'application.

```
@Configuration
@ConditionalOnClass(EmbeddedAcmeService.class)
static class EmbeddedConfiguration
{

    @Bean
    @ConditionalOnMissingBean
```



```
public EmbeddedAcmeService embeddedAcmeService() { ... }  
  
}
```

5.3. @ConditionalOnNotWebApplication and @ConditionalOnWebApplication

Ces annotations permettent d'inclure la configuration selon que l'application est une «application Web» ou non. Au spring, une application Web répond à au moins l'une des trois exigences suivantes:

- utilise un Spring WebApplicationContext
- définit une portée de session
- a un environnement StandardServletEnvironment

5.4. @ConditionalOnProperty

Cette annotation permet d'inclure la configuration en fonction de la présence et de la valeur d'une propriété Spring Environment.

Par exemple, si nous avons différentes définitions de sources de données pour différents environnements, nous pouvons utiliser cette annotation.

```
@Bean  
@ConditionalOnProperty(name = "env", havingValue = "local")  
DataSource dataSource()  
{  
    // ...  
}  
  
@Bean  
@ConditionalOnProperty(name = "env", havingValue = "prod")  
DataSource dataSource()  
{  
    // ...  
}
```

5.5. @ConditionalOnResource

Cette annotation permet d'inclure la configuration uniquement lorsqu'une ressource spécifique est présente dans le chemin de classe. Les ressources peuvent être spécifiées en utilisant les conventions Spring habituelles.

```

@ConditionalOnResource(resources = "classpath:vendor.properties")
Properties additionalProperties()
{
    // ...
}

```

5.6. @ConditionalOnExpression

Cette annotation permet d'inclure la configuration en fonction du résultat d'une expression SpEL. Utilisez cette annotation lorsque la condition à évaluer est complexe et doit être évaluée comme une seule condition.

```

@Bean
@ConditionalOnExpression("${env} && ${havingValue == 'local'}")
DataSource dataSource()
{
    // ...
}

```

5.7. @ConditionalOnCloudPlatform

Cette annotation permet d'inclure la configuration lorsque la plate-forme cloud spécifiée est active.

```

@Configuration
@ConditionalOnCloudPlatform(CloudPlatform.CLOUD_FOUNDRY)
public class CloudConfigurationExample
{
    @Bean
    public MyBean myBean(MyProperties properties)
    {
        return new MyBean(properties.getParam());
    }
}

```

Microservices - Monitoring

Spring Boot et Spring Cloud sont largement utilisés pour fournir des applications basées sur des microservices. Finalement, il est devenu une nécessité de surveiller les microservices basés sur les applications Spring Boot s'exécutant sur différents hôtes. Il existe de nombreux outils disponibles pour surveiller diverses statistiques de santé de ces microservices.

Dans ce tutoriel Spring Cloud, nous allons apprendre à utiliser trois de ces outils de surveillance, à savoir le tableau de bord Hystrix, le tableau de bord d'administration Eureka et le tableau de bord d'administration Spring Boot.

1. Vue d'ensemble

Dans cette démo, nous allons créer trois applications.

Employee Service - Cette application de microservice est chargée de récupérer les données des employés.

Api-Gateway - Cette application est de fournir une passerelle commune tout en accédant à différents microservices. Dans l'exemple suivant, il agira comme une passerelle vers le service aux employés ci-dessus.

Serveur Eureka - Cette application de microservice fournira la découverte de services et l'enregistrement des microservices ci-dessus.

Cette démo a été créée autour de Netflix Eureka pour gérer et surveiller de manière centralisée les applications enregistrées. Comme vous le savez peut-être déjà, le serveur Netflix Eureka est destiné à la création d'un serveur de registre de service et des clients Eureka associés, qui s'enregistreront pour rechercher d'autres services et communiquer via des apis REST.

2. Technology stack

- Java 1.8
- Spring tool suite
- Spring cloud
- Spring boot
- Spring Rest

- Maven

3. Employee Service

Créer une app Spring boot avec ces dépendance : **Eureka Discovery, Actuator, Web, Rest repositories.**

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.10

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

Eureka Discovery Actuator Rest Repositories Web

Generate Project

Don't know what to look for? Want more options? [Switch to the full version.](#)

Cloner le répo :

4. API-Gateway with Hystrix

Create a Spring boot project from [Spring boot initializer](#) with dependencies Eureka Discovery, Actuator, Web, Hystrix, Hystrix Dashboard, Rest repositories.

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.10

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

Eureka Discovery Actuator Rest Repositories Web
Hystrix Hystrix Dashboard

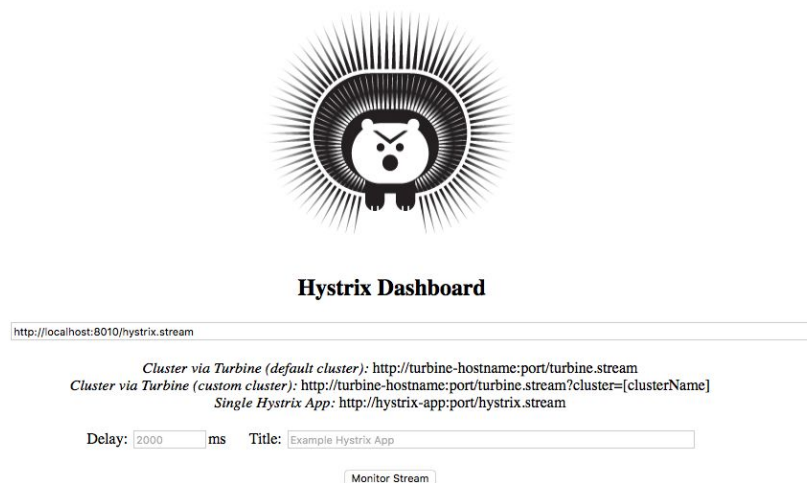
Generate Project

Don't know what to look for? Want more options? [Switch to the full version.](#)

Cloner le répo :

5. Vue du tableau de bord Hystrix

Pour surveiller via le tableau de bord Hystrix, ouvrez le tableau de bord Hystrix à l'adresse <http://localhost:8010/hystrix>



Il s'agit de la page d'accueil où l'URL du flux d'événements doit être placée pour la surveillance.

Maintenant, affichez le flux hystrix dans le tableau de bord - <http://localhost:8010/hystrix.stream>



Cela fournit des informations en temps réel sur toutes les commandes Hystrix et les pools de threads.

6. Vue du tableau de bord d'administration d'Eureka

Apprenons maintenant à utiliser la vue du tableau de bord d'administration d'Eureka.

Créez un projet de démarrage Spring à partir de l'initialiseur de démarrage Spring avec ces dépendances Eureka Server, Actuator, Web, Spring Boot Admin Server.

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.10

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies


Actuator Web Eureka Server Spring Boot Admin (Server)

Generate Project

Don't know what to look for? Want more options? [Switch to the full version.](#)

Cloner le repo :

Démarrez l'application. Mais avant cela, assurez-vous que le reste des applications clientes mentionnées ci-dessus sont démarrées avant afin de voir toutes les applications enregistrées. Cette application est accessible à l'adresse <http://localhost:8761>

HOMELAST 1000 SINCE STARTUP

System Status

Environment	test
Data center	default

Current time	2018-02-18T22:56:41 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	5
Renews (last min)	0

DS Replicas

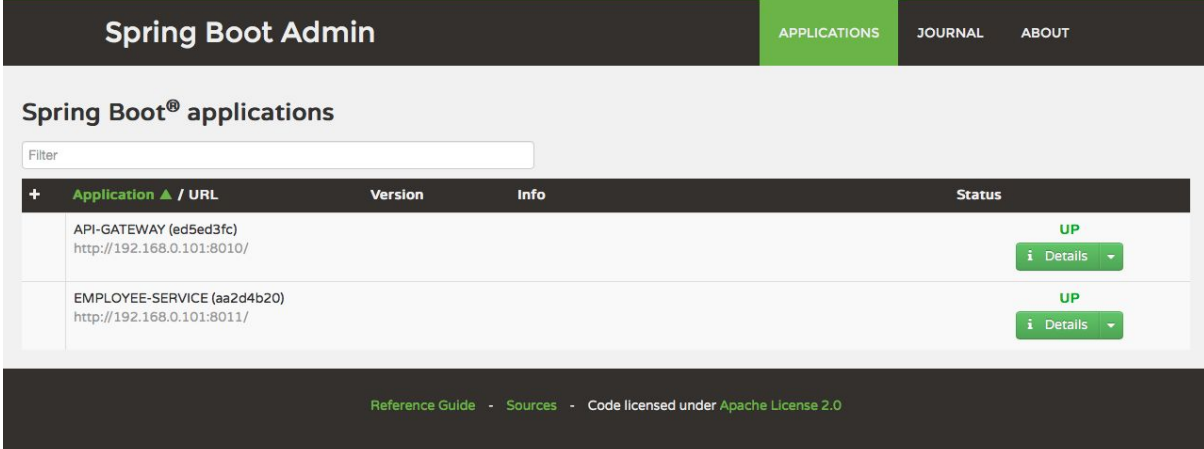
localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.0.101:api-gateway:8010
EMPLOYEE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.101:employee-service:8011

7. Vue du tableau de bord d'administration de Spring Boot

Pour surveiller via le serveur Spring Boot Admin, invoquez cette URL exécutée à un chemin de contexte différent - <http://localhost:8761/admin>

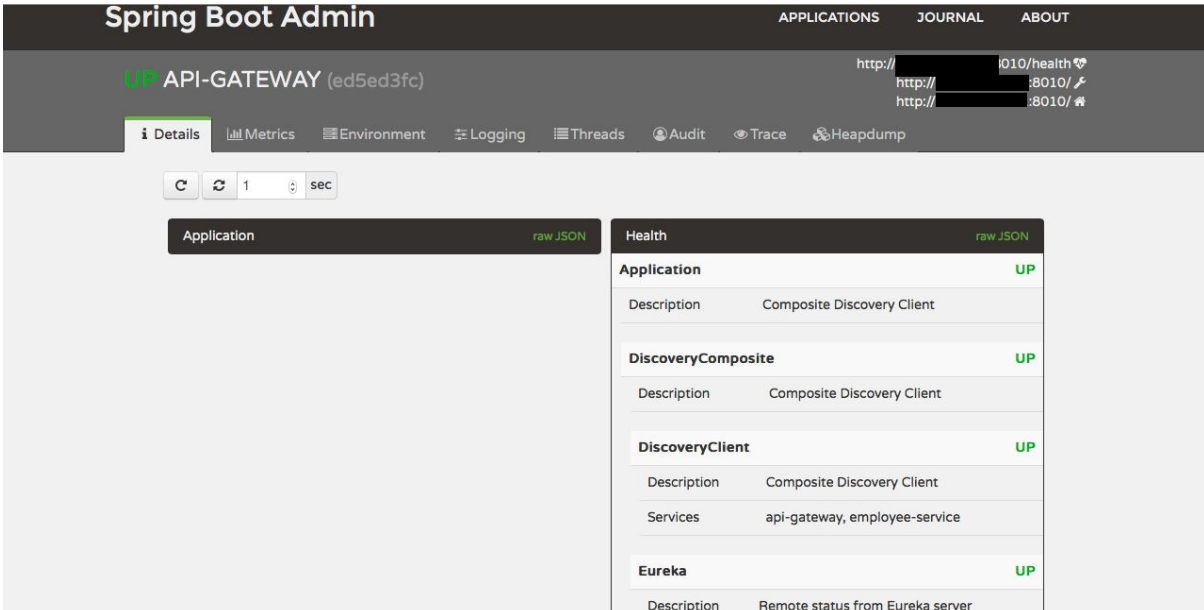


The screenshot shows the Spring Boot Admin dashboard. At the top, there's a navigation bar with 'Spring Boot Admin' and tabs for 'APPLICATIONS', 'JOURNAL', and 'ABOUT'. Below the navigation bar, the title 'Spring Boot® applications' is followed by a 'Filter' input field. A table lists the applications:

Application / URL	Version	Info	Status
API-GATEWAY (ed5ed3fc) http://192.168.0.101:8010/			UP Details
EMPLOYEE-SERVICE (aa2d4b20) http://192.168.0.101:8011/			UP Details

At the bottom, there are links for 'Reference Guide', 'Sources', and 'Code licensed under Apache License 2.0'.

Cette interface d'administration fournit une vue d'ensemble de l'application, des notifications sur le bureau, une vérification de l'état de l'application, la navigation dans les fichiers logs, JMX Beans, un vidage de tas de threads, etc. Pour voir l'état de santé de chaque application et surveiller ses métriques, cliquez sur le bouton de détail. Il vous mènera au tableau de bord d'administration de l'application individuelle.



The screenshot shows the detailed view of the 'API-GATEWAY (ed5ed3fc)' application. The top navigation bar is the same. Below it, the application name and version are displayed, along with a 'UP' status indicator. A list of health endpoints is shown: 'http://192.168.0.101:8010/health', 'http://192.168.0.101:8010/...', and 'http://192.168.0.101:8010/...'. Below this, there are tabs for 'Details', 'Metrics', 'Environment', 'Logging', 'Threads', 'Audit', 'Trace', and 'Heapdump'. The 'Details' tab is selected. Below the tabs, there are buttons for 'Application' and 'raw JSON'. The 'Health' section is expanded, showing a list of components and their status:

Component	Status
Application	UP
Description	Composite Discovery Client
DiscoveryComposite	UP
Description	Composite Discovery Client
DiscoveryClient	UP
Description	Composite Discovery Client
Services	api-gateway, employee-service
Eureka	UP
Description	Remote status from Eureka server

Utilisation du tableau de bord pour gérer les niveaux de log :

The screenshot shows the UP API-GATEWAY dashboard with the 'Logging' tab selected. The dashboard header includes the title 'UP API-GATEWAY (ed5ed3fc)' and a navigation bar with tabs: Details, Metrics, Environment, Logging, Threads, Audit, Trace, and Heapdump. The 'Logging' tab is active, showing a list of components and their log levels. A filter bar at the top of the list shows 'Filter by name ...' and a count of '434/694'. The list of components includes 'ROOT' and several classes from the 'com.netflix' package. Each component has a row of buttons for selecting the log level: TRACE, DEBUG, INFO, WARN, ERROR, and OFF. The 'INFO' button is highlighted for all components. A 'show more' link is visible at the bottom of the list.

Component	TRACE	DEBUG	INFO	WARN	ERROR	OFF
ROOT						
com.netflix.appinfo.ApplicationInfoManager						
com.netflix.appinfo.InstanceInfo						
com.netflix.appinfo.providers.Archaius1VipAddressResolver						
com.netflix.client.ClientFactory						
com.netflix.client.config.DefaultClientConfigImpl						
com.netflix.config.ChainedDynamicProperty						
com.netflix.config.ConcurrentCompositeConfiguration						
com.netflix.config.ConcurrentMapConfiguration						
com.netflix.config.ConfigurationManager						

Utilisation du tableau de bord pour gérer les propriétés de l'environnement d'exécution :

Navigation bar: Details | Metrics | **Environment** | Logging | Threads | Audit | Trace | Heapdump

Active profiles raw JSON

-

Environment manager

property name property value

Refresh context Update environment Reset environment

Filter C

PropertySource server.ports

local.server.port	8010
-------------------	------

PropertySource commandLineArgs

spring.output.ansi.enabled	always
----------------------------	--------

PropertySource servletContextinitParams

-

Vous pouvez également l'utiliser pour afficher les traces HTTP :

Navigation bar: Details | Metrics | Environment | Logging | Threads | Audit | **Trace** | Heapdump

C ↺ 3 sec

13:48:46.214 200 - GET /health
04.03.2018

13:48:26.217 200 - GET /health
04.03.2018

13:48:06.219 200 - GET /health
04.03.2018

13:47:46.215 200 - GET /health
04.03.2018

13:47:26.221 200 - GET /health
04.03.2018

13:47:06.234 200 - GET /health
04.03.2018

13:46:46.219 200 - GET /health
04.03.2018

13:46:26.217 200 - GET /health
04.03.2018

13:46:06.223 200 - GET /health
04.03.2018

13:45:46.217 200 - GET /health
04.03.2018

Microservices - Virtualization

La virtualisation de microservices est une technique permettant de simuler le comportement de composants spécifiques dans des applications hétérogènes basées sur des composants telles que des applications basées sur des API, des applications basées sur le cloud ou des architectures orientées services.

Découvrez en détail les concepts de [virtualisation des services](#) et examinez également un outil de virtualisation de services populaire et utile - Hoverfly. Découvrez également comment Hoverfly peut être utilisé pour capturer des requêtes / réponses en mode proxy et utiliser ces réponses capturées pendant qu'[Hoverfly](#) fonctionnera en mode simulation.

Qu'est-ce que la virtualisation des microservices?

De nos jours, les applications cloud utilisent de nombreux microservices qui interagissent les uns avec les autres pour accomplir certaines capacités commerciales. En développant ce type d'écosystème, nous sommes parfois confrontés à des problèmes communs qui ont généralement un impact sur la productivité de toute l'équipe, par exemple.

- Tous les services de l'écosystème peuvent ne pas être disponibles actuellement. Peut-être que ceux-ci sont également développés par une autre équipe.
- Certains des services hérités ne sont pas libres d'avoir un environnement de développement ou cela peut être coûteux et pour des raisons évidentes, nous ne pouvons pas utiliser la version de production pour les tests.
- Certains services peuvent être interrompus pour une raison quelconque.
- Problème de gestion des données de test. Souvent, pour écrire des tests appropriés, vous avez besoin d'un contrôle précis sur les données de vos simulacres ou stubs. La gestion des données de test sur de grands projets avec plusieurs équipes introduit des goulots d'étranglement qui ont un impact sur les délais de livraison.

Ainsi, nous pouvons facilement comprendre que les problèmes ci-dessus auront un impact sur le développement du produit actuel et peuvent également avoir un impact sur le calendrier de livraison, qui est directement proportionnel au coût de développement de ce produit. Alors, quelles sont les solutions pour cela?

- Nous pouvons penser à nous mocker de ces services en utilisant certains frameworks Mock populaires. Il présente également certains inconvénients, par ex. Les simulations sont généralement spécifiques à un scénario et il faut beaucoup

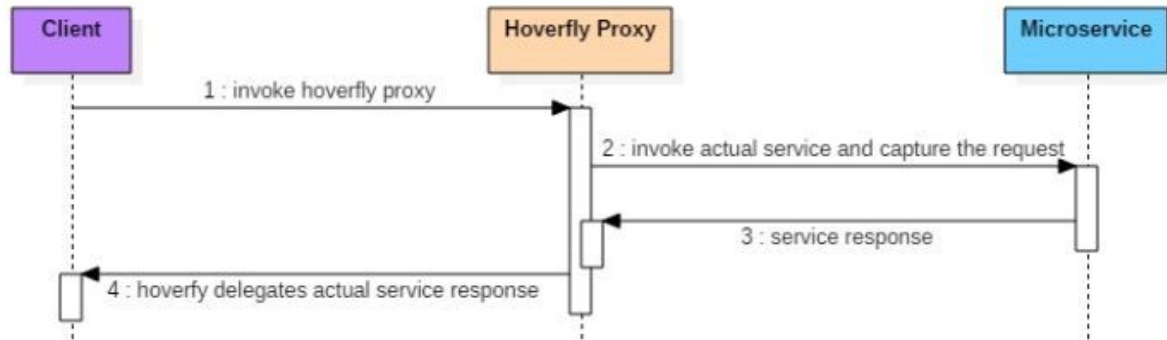
d'efforts pour créer une réponse simulée pour ces services et la simulation est également mieux adaptée à la phase de test unitaire uniquement (JUnit).

- Nous pouvons utiliser des services stubbed, où nous développerons des faux services avec des réponses codées en dur - encore une fois, cela n'a pas de sens car nous devons développer quelque chose pour que cela fonctionne.
- De plus, maintenant, nous devons faire une intégration continue pendant le développement et dans ces cas, les services Mocking et Stubbed ne sont pas de très bons candidats pour cela.

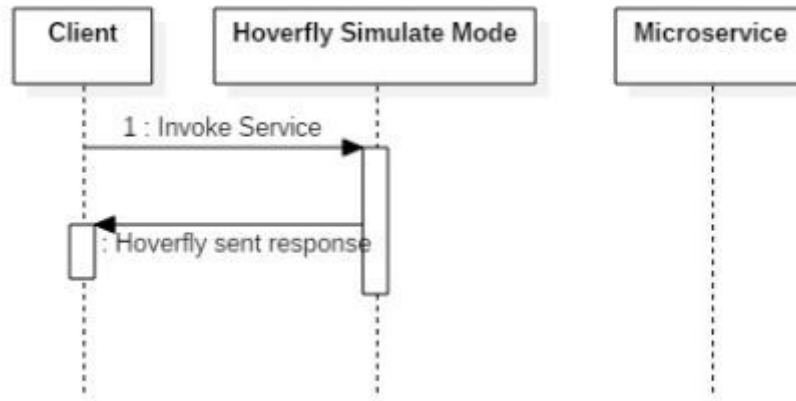
Nous avons un très bon [article infoq](#) sur ces types de techniques similaires basé sur le problème commun dont nous avons discuté.

Étant donné que les services Mock et stubbed ont peu de problèmes à utiliser efficacement, donc pour résoudre le problème ci-dessus, nous avons une technique appelée virtualisation de service où nous pouvons capturer / simuler des services réels. Hoverfly est l'un de ces outils nouvellement développé avec un nouveau langage JVM GO et il propose des étapes très simples et réalistes pour résoudre le problème.

Vous trouverez ci-dessous le diagramme de séquence lorsque nous avons un service virtualisé en cours d'exécution pour une meilleure compréhension.



Hoverfly en mode Capture - Agissant comme un service proxy pour le service réel



Hoverfly en mode simulation - Répond directement sans passer au service réel

Démo

Nous suivrons les étapes données pour démontrer l'utilisation de Hoverfly comme outil de virtualisation de service.

- Nous allons créer un petit écosystème de microservices qui interagiront les uns avec les autres.
- Nous utiliserons Hoverfly pour intercepter la requête / réponse réelle lorsqu'elle est en mode capture.
- Enfin, nous verrons comment Hoverfly peut agir en tant que serveur de virtualisation de services pour renvoyer des requêtes / réponses déjà capturées alors qu'il est en mode simulation.
- Nous vérifierons également que le temps d'arrêt du service sous-jacent n'aura pas beaucoup d'effet sur notre développement.
- Nous verrons également avec quelle facilité nous pouvons basculer Hoverfly pour revenir en mode capture et transmettre la demande au service réel.

Conditions préalables

Avant de commencer la démonstration de la fonctionnalité Hoverfly, assurez-vous que les prérequis ci-dessous sont installés:

- Hoverfly [[Guide d'installation](#)]
- JDK 8
- Éclipse
- Maven

Créer un downstream service

Créer un projet Spring Boot

Créez un projet Spring boot avec des dépendances, c'est-à-dire des référentiels Web et Rest. Donnez d'autres coordonnées GAV maven comme ci-dessous et téléchargez le projet.

Décompressez et importez le projet dans un IDE en tant que projet maven existant. Dans cette étape, effectuez une nouvelle compilation de maven à l'aide de la commande mvn clean install afin que toutes les dépendances de maven soient téléchargées correctement.

Ajouter un Rest Endpoint

Ajoutez une classe RestController qui exposera un simple endpoint `/service/ hoverfly`. Ce endpoint sera consommé par un deuxième service que nous développerons par la suite. Par souci de simplicité, nous renvoyons simplement des valeurs codées en dur avec un temps de réponse ajouté dans la réponse.

cloner le code source :

Vérifier le service

Changez le port de l'application en 9080 avec la propriété server.port = 9080. Démarrez ce projet en tant qu'application de démarrage de Spring en exécutant la commande java -jar target \ hoverfly-actual-service-0.0.1-SNAPSHOT.jar.

Une fois le serveur démarré, accédez au navigateur et testez si ces points de terminaison fonctionnent : <http://localhost:9080/service/hoverfly>



The screenshot shows a web browser window with the address bar displaying `localhost:9080/service/hoverfly`. The browser shows a JSON response from the API:

```
{
  message: "returned value from HoverflyActualServiceApplication",
  responseTime: "Wed Aug 09 13:09:24 IST 2017",
  transactionid: "a2191945-f9a8-4029-86be-b243bccc7f9c"
}
```

Downstream Service Response in browser

Notre premier downstream microservice est donc opérationnel. Nous allons maintenant créer le deuxième microservice qui invoquera ce service.

Créer un service client

Suivez à nouveau les étapes ci-dessus pour créer ce service. Après avoir importé le projet dans eclipse, ajoutez le code du contrôleur.

Ajouter un Rest Endpoint

Ajoutez une RestControllerclass qui exposera un simple endpoint `/invoke`. Ce endpoint appellera en interne le downstream service que nous venons de développer (hoverfly-actual-service).

De plus, nous avons ajouté une logique lors de la création du bean RestTemplate en prenant une propriété système appelée mode.

Si nous passons mode=proxy lors du démarrage du service, toutes les demandes à ce sujet seront d'abord acheminées via le proxy Hoverfly.

Si nous passons mode=production, toutes les demandes à ce sujet seront directement adressées au service réel.

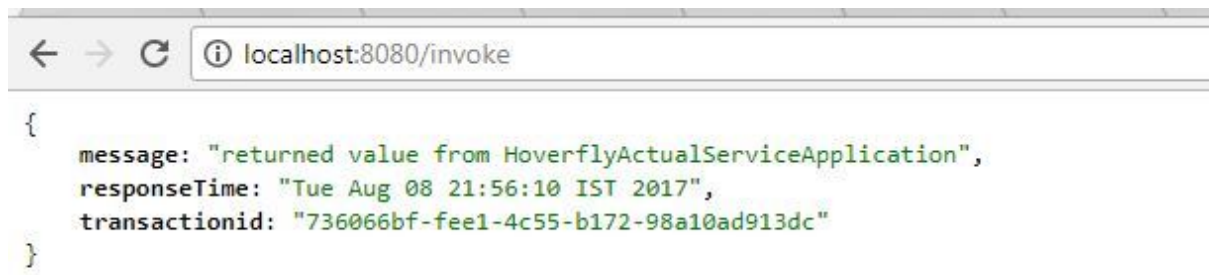
Veuillez regarder attentivement la méthode restTemplate() pour comprendre le mode proxy. Remarque Le serveur proxy Hoverfly fonctionnera à l'adresse <http://localhost:8500> .

Vérifier le service

Ce service s'exécute sur le port par défaut 8080 en local. Faites une construction maven en utilisant la commande mvn clean install et démarrez ce projet en tant qu'application de démarrage de Spring en exécutant la commande java -jar -Dmode=proxy target\hoverfly-actual-service-client-0.0.1-SNAPSHOT.jar .

Veuillez noter que -Dmode=proxy est passé lors du démarrage du service en mode proxy. Dans des environnements réels comme la production, nous ne passerons pas ce paramètre. Appelez l'API dans le navigateur.

<http://localhost:8080/invoke>

A screenshot of a web browser window. The address bar shows 'localhost:8080/invoke'. The page content displays a JSON object with three fields: 'message', 'responseTime', and 'transactionid'.

```
{
  message: "returned value from HoverflyActualServiceApplication",
  responseTime: "Tue Aug 08 21:56:10 IST 2017",
  transactionid: "736066bf-fee1-4c55-b172-98a10ad913dc"
}
```

Client Service running in proxy mode

Nous avons donc également développé l'application client et nous pouvons également tester le service client et obtenir une réponse du service down stream. Nous avons également configuré correctement le serveur proxy Hoverfly dans notre code client afin que nous puissions intégrer facilement hoverfly ensuite.

Démo Hoverfly

Nous allons maintenant démarrer le vol stationnaire dans notre local et testerons différents modes et verrons comment cela aide réellement en cas d'indisponibilité du service. Hoverfly propose 4 modes différents, capture, simulate, modify et synthesize. Nous recherchons uniquement le mode capture et simulate dans cette démo.

Démarrez Hoverfly en mode capture

Ouvrez une fenêtre de commande dans le répertoire Hoverfly (répertoire décompressé) et tapez la commande hoverctl start. Il démarrera l'hoverfly sur le poste de travail local en proxy mode et démarrera l'interface utilisateur d'administration sur le port 8888 et le serveur proxy sur le port 8500.

Tapez maintenant capture en mode hoverctl dans la même invite de commande pour changer le mode hoverfly à capturer. Après ces deux commandes, la fenêtre d'invite de commande ressemblera à :

```
C:\Windows\system32\cmd.exe

F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl start
Hoverfly is now running

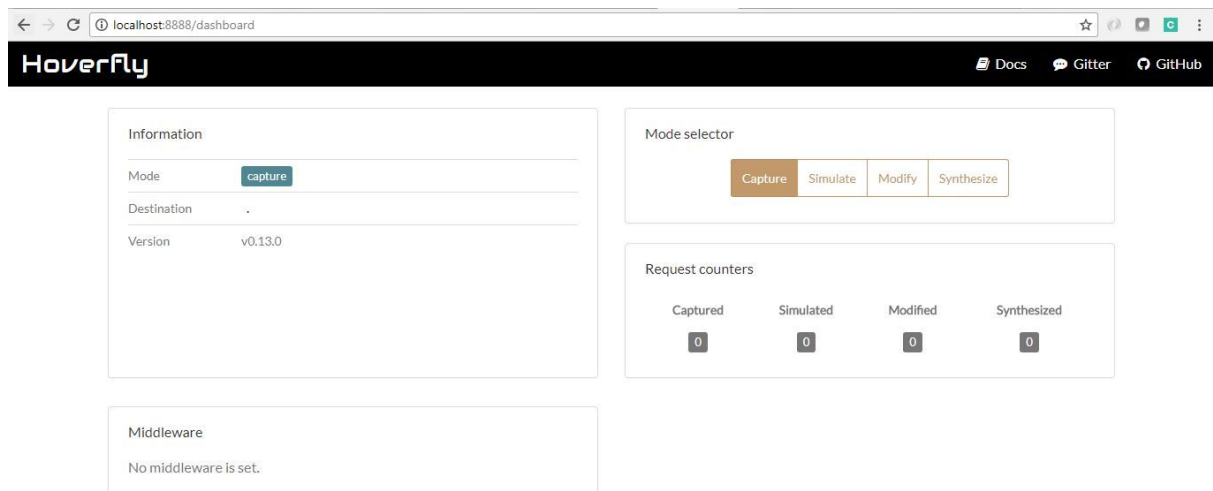
+-----+-----+
| admin-port | 8888 |
| proxy-port | 8500 |
+-----+-----+

F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl mode capture
Hoverfly has been set to capture mode

F:\Study\installations\hoverfly_bundle_windows_amd64>
```

Start Hoverfly in Capture mode

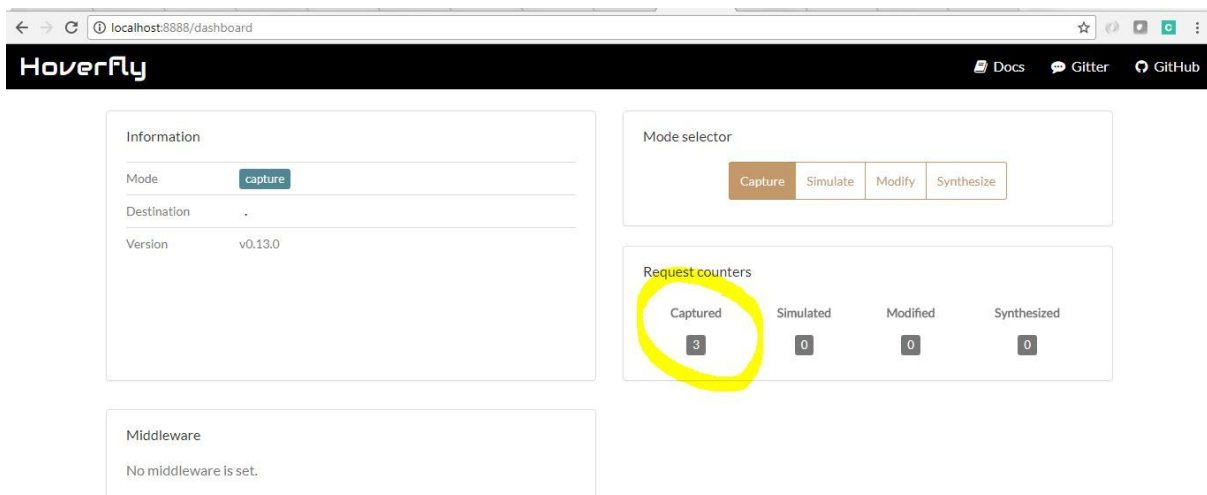
Maintenant, allez dans le navigateur <http://localhost:8888/dashboard> et il affichera l'interface utilisateur d'administration où nous pouvons également changer le mode et voir combien de demandes ont été capturées ou simulées.



Hoverfly admin UI

Capturez les Requests/Responses

Maintenant, avec ces paramètres, dans la fenêtre du navigateur, exécutez le [service client](#) plusieurs fois pendant qu'Hoverfly est en mode capture. Maintenant, allez à nouveau dans l'interface utilisateur d'administration et notez que le compteur de capture a été augmenté au nombre de fois où vous avez accédé à l'application de service client dans le navigateur.



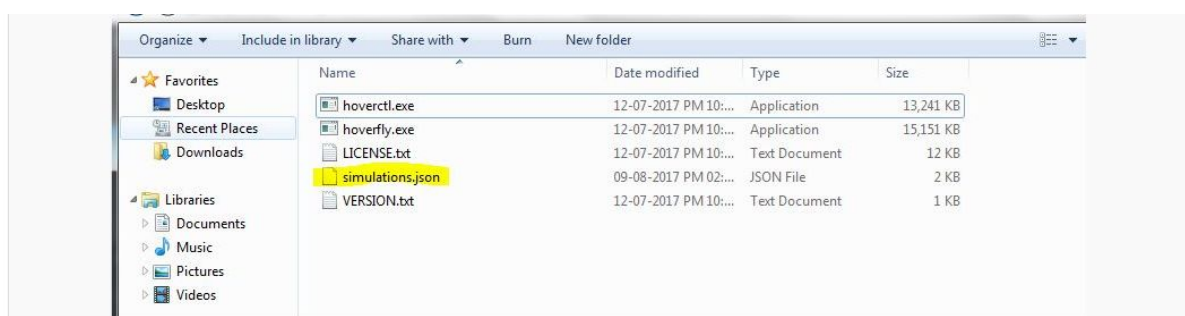
Hoverfly has captured transactions

Export/Import les demandes capturées

Il est judicieux de stocker les demandes et les réponses simulées à un autre endroit, afin que nous n'ayons pas besoin d'exécuter Hoverfly en permanence. Chaque fois que nous en aurons besoin, nous réimporterons les demandes / réponses enregistrées et commencerons à simuler les services.

Nous allons maintenant exporter ces demandes capturées dans un fichier JSON, puis importer ce fichier dans Hoverfly et démarrer Hoverfly en mode simulation.

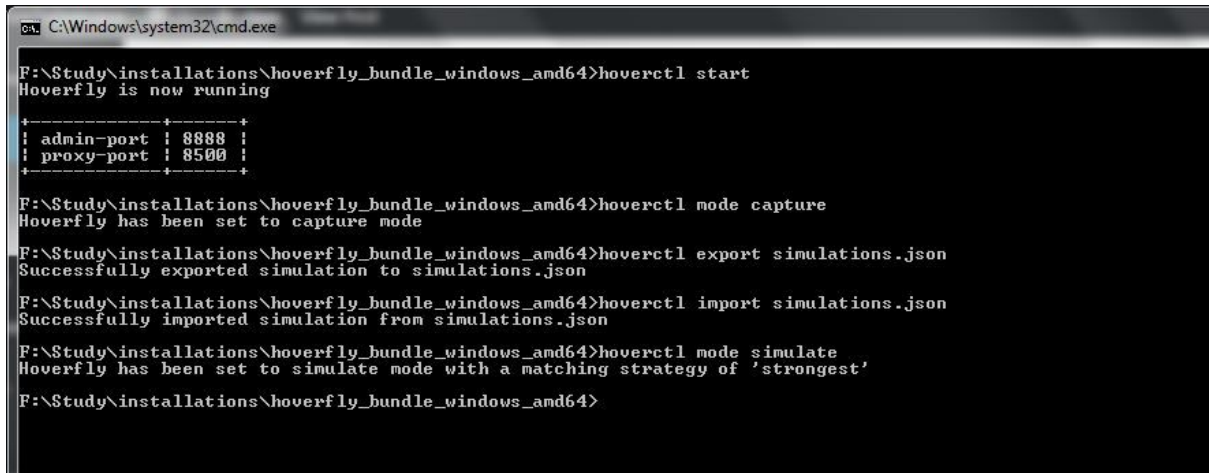
Pour exporter, ouvrez la fenêtre de commande Hoverfly et tapez la commande `hoverctl export simulations.json`, cela exportera le fichier déjà capturé (dans notre cas 3 transactions) dans un fichier json `simulations.json` avec toutes les URLs, requêtes, etc. Une fois le fichier exporté sera créé dans le répertoire personnel de hoverfly.



Export to a JSON file

Pour importer le fichier `simulations.json`, vous pouvez taper la commande `hoverctl import simulations.json` pour importer les définitions capturées.

Une fois importé, nous changerons le mode Hoverfly pour `simulate` par la commande de simulation `hoverctl mode simulate`. Nous pouvons également le faire à partir de la page de l'interface utilisateur d'administration de Hoverfly.



```
C:\Windows\system32\cmd.exe
F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl start
Hoverfly is now running

+-----+
| admin-port | 8888 |
| proxy-port | 8500 |
+-----+

F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl mode capture
Hoverfly has been set to capture mode

F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl export simulations.json
Successfully exported simulation to simulations.json

F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl import simulations.json
Successfully imported simulation from simulations.json

F:\Study\installations\hoverfly_bundle_windows_amd64>hoverctl mode simulate
Hoverfly has been set to simulate mode with a matching strategy of 'strongest'

F:\Study\installations\hoverfly_bundle_windows_amd64>
```

Import/Export and simulate commands

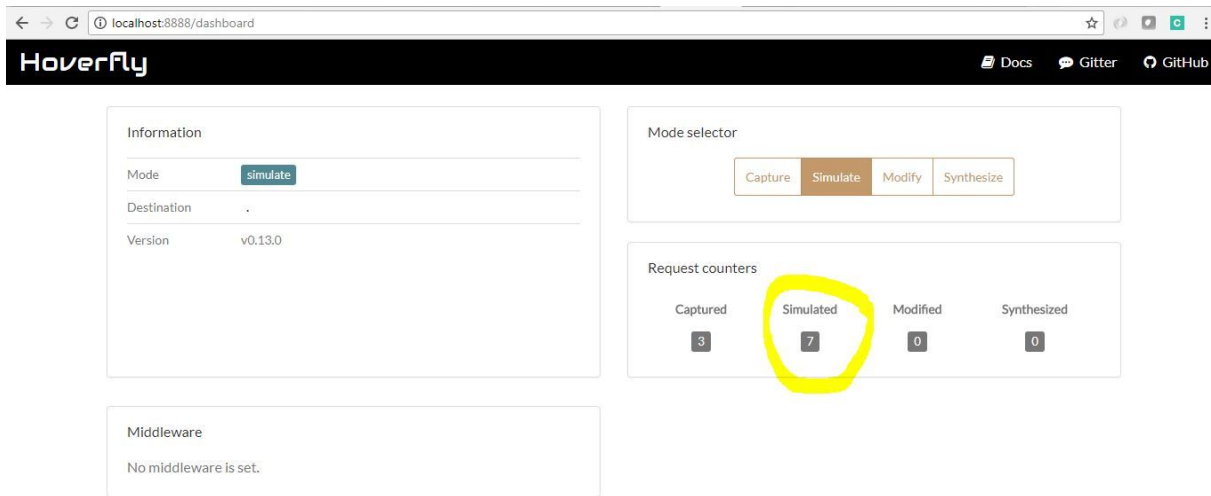
Nous sommes maintenant prêts à changer le mode Hoverfly à `simulate` et tester.

Test en mode simulation

Passez en mode simuler avec cette commande - `simuler le mode hoverctl`. Suivez les étapes simples suivantes:

- Ouvrez l'application cliente dans le navigateur et appuyez sur le bouton d'actualisation pour voir que la réponse ne change pas dans le navigateur (temps de réponse et champ d'identifiant de transaction), cela signifie qu'Hoverfly est actif et envoie des réponses pour tous les modèles d'URL qui correspondent au fichier importé.
- Non, accédez à l'interface utilisateur Hoverfly Admin et voyez que le compteur `Simuler` a été augmenté au nombre de fois où l'application cliente a été accédée en mode simulation.
- Arrêtez maintenant le service `down stream` et appuyez sur l'application cliente, vous pouvez facilement voir que le Hoverfly répond au nom du service `down stream` simulé. Ce qui est génial et vraiment utile dans un scénario réel où nous voulons tester les fonctionnalités lorsque le service réel est arrêté.
- Redémarrez le service `down stream` et changez le mode de Hoverfly pour `capturer` à partir de l'invite de commande ou de l'interface utilisateur d'administration, et

voyez que toute demande adressée au service client est déclenchée jusqu'à ce que le service down stream et le compteur de capture du Hoverfly augmentent. Il est donc très facile de basculer entre le mode capture et simuler, ce qui est nécessaire si nous voulons capturer un nouveau type de demandes des downstream services.



Hoverfly simulated responses

Microservices - ELK Stack

Avec l'utilisation de microservices, nous avons pu surmonter de nombreux problèmes hérités et cela nous permet de créer des applications distribuées stables avec le contrôle souhaité sur le code, la taille de l'équipe, la maintenance, le cycle de publication, l'ennoblissement du cloud, etc. Mais cela a également introduit peu de défis dans d'autres domaines, par exemple gestion des logs distribués et possibilité d'afficher les logs de transaction complète répartis entre de nombreux services et le débogage distribué en général.

En fait, le défi est que les microservices sont isolés entre eux et qu'ils ne partagent pas la base de données et les fichiers logs communs. Au fur et à mesure que le nombre de microservices augmente et que nous permettons le déploiement dans le cloud avec des outils d'intégration continue automatisés, il est absolument nécessaire de prévoir un débogage des composants lorsque nous avons un problème.

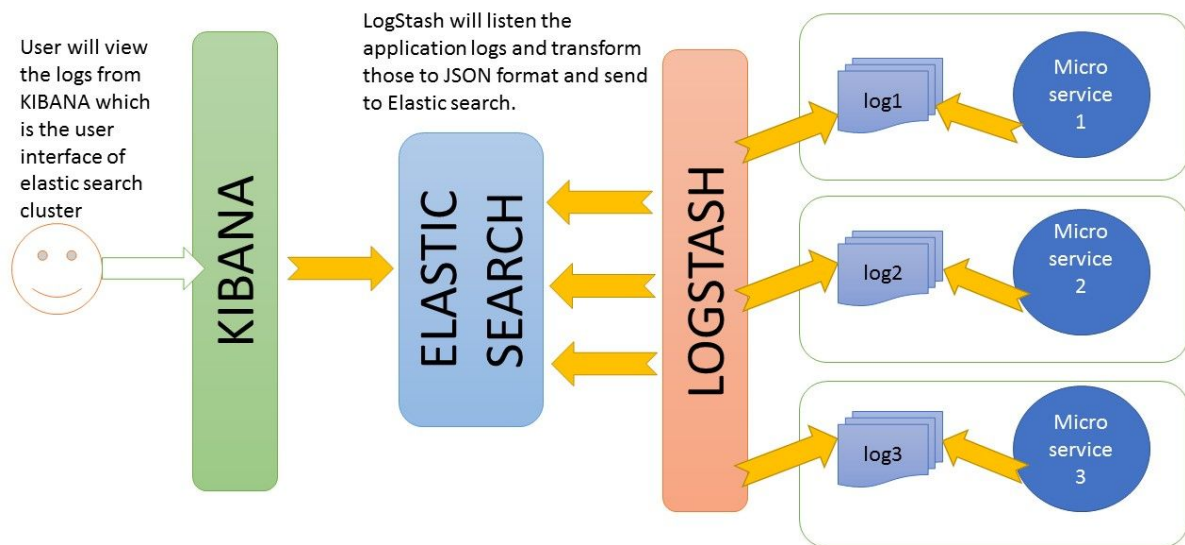
Merci au mouvement open source. Nous avons déjà un ensemble d'outils qui peuvent faire la magie s'ils sont utilisés correctement ensemble. [Elasticsearch](#), [Logstash](#) et [Kibana](#) sont l'un de ces ensembles d'outils populaires, appelés ensemble ELK stack. Ils sont utilisés pour rechercher, analyser et visualiser les données du journal en temps réel.

1. Qu'est-ce que ELK Stack

- Elasticsearch est un moteur de recherche et d'analyse distribué basé sur JSON, conçu pour une évolutivité horizontale, une fiabilité maximale et une gestion facile.
- Logstash est un pipeline de collecte de données dynamique avec un écosystème de plugins extensible et une forte synergie Elasticsearch.
- Kibana donne la visualisation des données via une interface utilisateur.

1.1. Architecture de pile ELK

Logstash traite les fichiers logs de l'application en fonction des critères de filtre que nous avons définis et envoie ces logs à Elasticsearch. Grâce à Kibana, nous visualisons et analysons ces logs si nécessaire.



ELK stack interaction with different applications based on Log file

2. Configuration de la pile ELK

Tous ces trois outils sont basés sur JVM et avant de commencer à les installer, veuillez vérifier que JDK a été correctement configuré. Vérifiez que l'installation standard du JDK 1.8, la configuration de JAVA_HOME et de PATH sont déjà effectuées.

2.1. Elasticsearch

- Téléchargez la dernière version d'Elasticsearch à partir de cette [page](#) de téléchargement et décompressez-la dans n'importe quel dossier.
- Exécutez `bin\elasticsearch.bat` à partir de l'invite de commande.
- Par défaut, il commencerait à <http://localhost:9200>

2.2. Kibana

- Téléchargez la dernière distribution à partir de la page de téléchargement et décompressez-la dans n'importe quel dossier.
- Ouvrez `config / kibana.yml` dans un éditeur et définissez `elasticsearch.url` pour qu'il pointe vers votre instance Elasticsearch. Dans notre cas, car nous utiliserons l'instance locale, décommentez simplement `elasticsearch.url`:
`"http://localhost:9200"`
- Exécutez `bin\kibana.bat` à partir de l'invite de commande.
- Une fois démarré avec succès, Kibana démarrera sur le port par défaut 5601 et l'interface utilisateur de Kibana sera disponible sur <http://localhost:5601>

2.3. Logstash

Téléchargez la dernière distribution à partir de la page de téléchargement et décompressez-la dans n'importe quel dossier.

Créez un fichier `logstash.conf` selon [les instructions de configuration](#). Nous reviendrons sur ce point lors de la démonstration réelle pour la configuration exacte.

Exécutez maintenant `bin/logstash -f logstash.conf` pour démarrer logstash

La pile ELK n'est pas opérationnelle. Nous devons maintenant créer quelques microservices et pointer logstash vers le chemin du journal de l'API.

3. Exemple de pile ELK - Créer un microservice

3.1. Créer un projet Spring Boot

Créons une application Spring Boot

3.2. Ajouter des points de terminaison REST

Ajoutez une classe RestController qui exposera quelques points de terminaison tels que /elk, /elkdemo, /exception. En fait, nous allons tester seulement quelques instructions de journal, alors n'hésitez pas à ajouter / modifier les logs selon votre choix.

Cloner le répo :

3.3. Configurer le logging Spring Boot

Ouvrez application.properties sous le dossier resources et ajoutez les entrées de configuration ci-dessous.

```
logging.file = elk-example.log  
spring.application.name = exemple elk
```

3.4. Vérifier les logs générés par microservice

Effectuez une compilation finale de maven à l'aide de mvn clean install et démarrez l'application à l'aide de la commande java -jar target\elk-example-spring-boot-0.0.1-SNAPSHOT.jar et testez en parcourant <http://localhost:8080/elk> .

N'ayez pas peur en voyant la grande trace de pile à l'écran, car cela a été fait intentionnellement pour voir comment ELK gère le message d'exception.

Accédez au répertoire racine de l'application et vérifiez que le fichier journal, c'est-à-dire elk-example.log, a été créé et effectuez quelques visites sur les points de terminaison et vérifiez que les logs sont ajoutés dans le fichier journal.

4. Configuration de Logstash

Nous devons créer un fichier de configuration logstash afin qu'il écoute le fichier journal et envoie les messages du journal vers la recherche élastique. Voici la [configuration](#) de logstash utilisée dans l'exemple, veuillez modifier le chemin du journal selon votre configuration.

```

input {
  file {
    type => "java"
    path => "F:/Study/eclipse_workspace_mars/elk-example-spring-boot/elk-example.log"
    codec => multiline {
      pattern => "%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}.*"
      negate => "true"
      what => "previous"
    }
  }
}

filter {
  #If log line contains tab character followed by 'at' then we will tag that entry as
  stacktrace
  if [message] =~ "\tat" {
    grok {
      match => ["message", "^(\\tat)"]
      add_tag => ["stacktrace"]
    }
  }

  grok {
    match => [ "message",
      "(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME})
%{LOGLEVEL:level} %{NUMBER:pid} --- \\[(?<thread>[A-Za-z0-9-]+)\\]
[A-Za-z0-9-]*\\. (?<class>[A-Za-z0-9#_]+)\\s*:\\s+(?<logmessage>.*)",
      "message",
      "(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME})
%{LOGLEVEL:level} %{NUMBER:pid} --- .+? :\\s+(?<logmessage>.*)"
    ]
  }

  date {
    match => [ "timestamp" , "yyyy-MM-dd HH:mm:ss.SSS" ]
  }
}

output {

  stdout {
    codec => rubydebug
  }

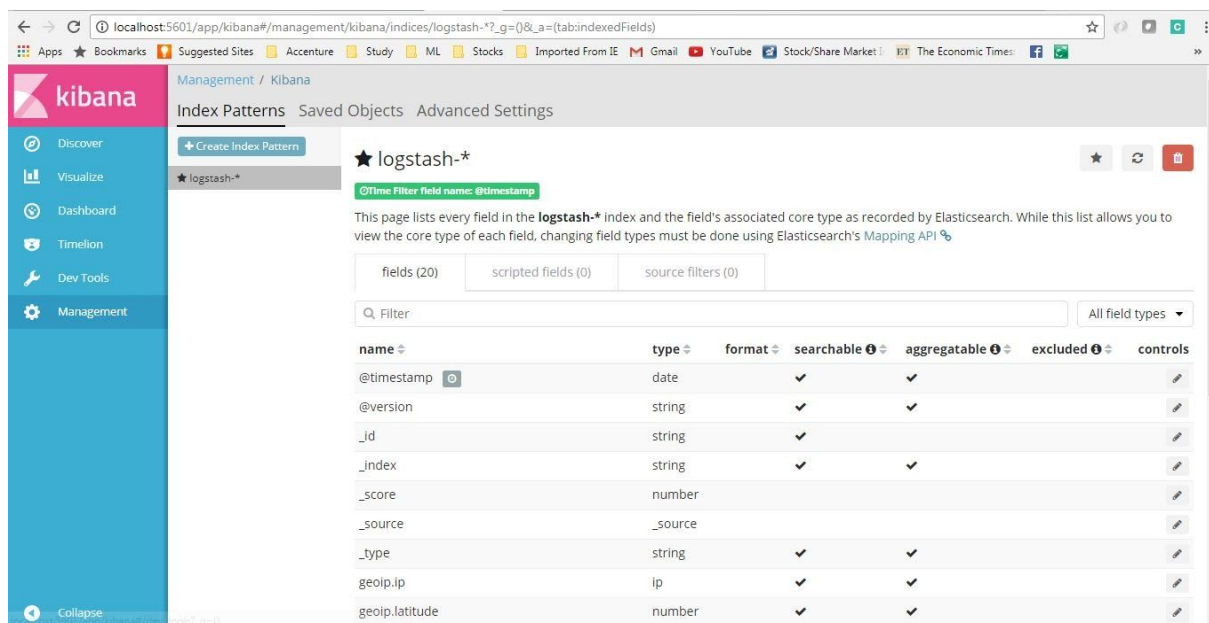
  # Sending properly parsed log events to elasticsearch
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}

```

5. Kibana Configuration

Avant d'afficher les logs dans Kibana, nous devons configurer les modèles d'index. Nous pouvons configurer `logstash-*` comme configuration par défaut. Nous pouvons toujours changer ce modèle d'index côté logstash et le configurer dans Kibana. Pour plus de simplicité, nous travaillerons avec la configuration par défaut.

La page de gestion des modèles d'index ressemblera à ci-dessous. Avec cette configuration, nous pointons Kibana vers les index Elasticsearch de votre choix. Logstash crée des index avec le modèle de nom `logstash-YYYY.MM.DD`. Nous pouvons faire toutes ces configurations dans la console Kibana <http://localhost:5601/app/kibana> et en accédant au lien de gestion dans le panneau de gauche.



Logstash configuration in Kibana

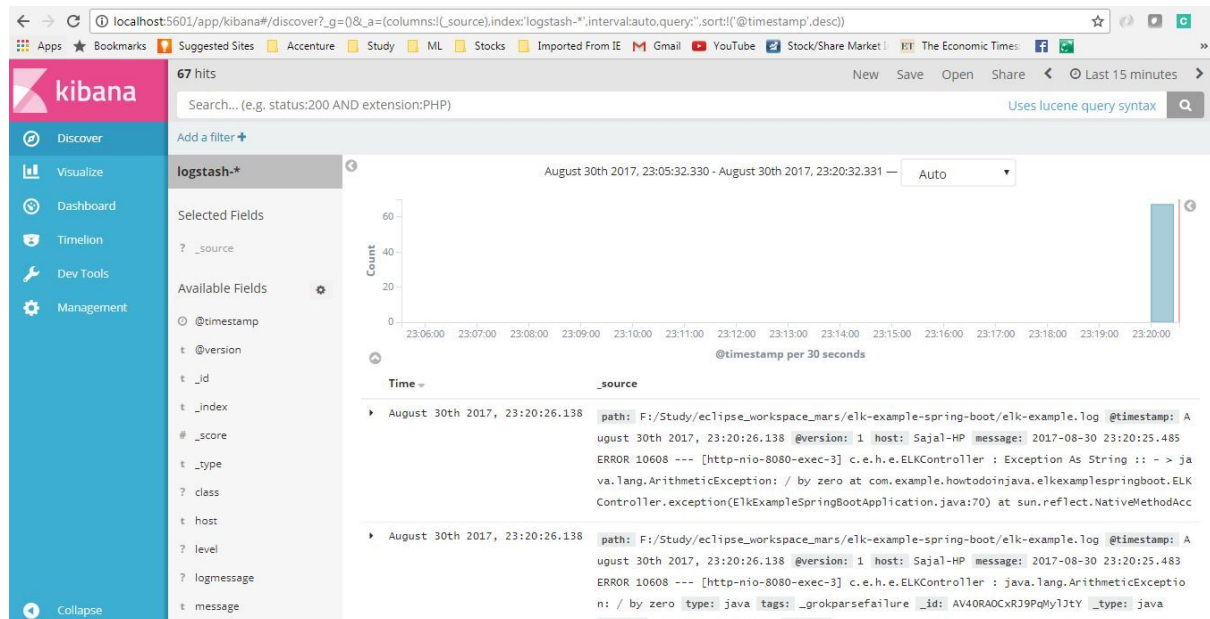
6. Vérifiez la pile ELK

Maintenant que tous les composants sont opérationnels, vérifions l'ensemble de l'écosystème.

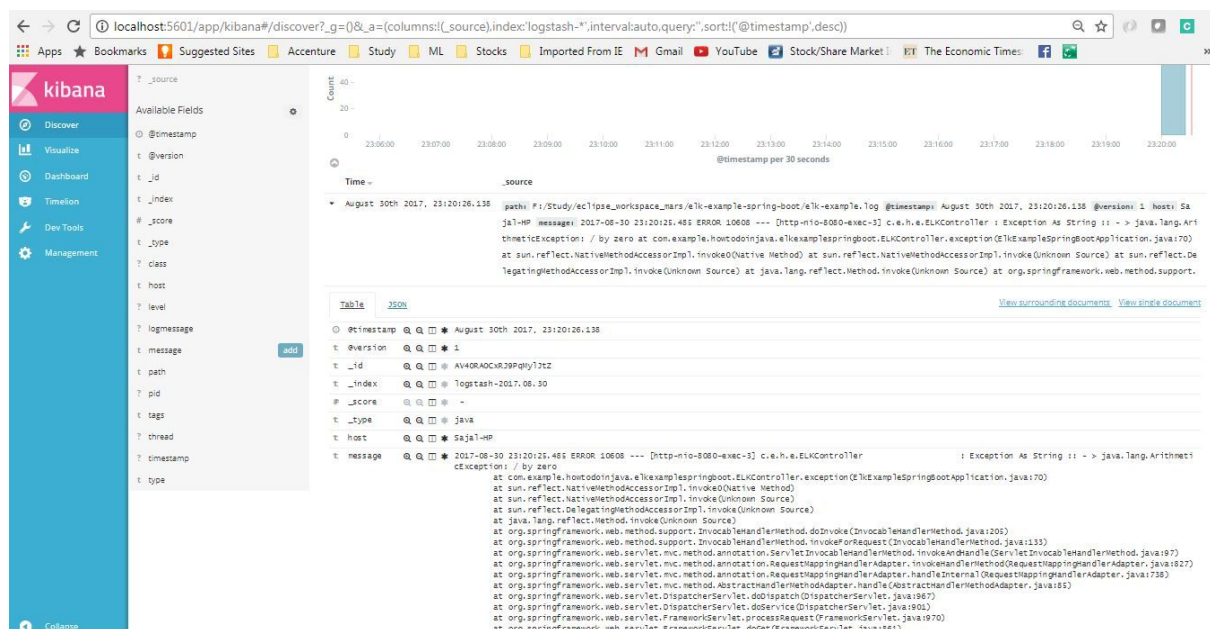
Accédez à l'application et testez les points de terminaison plusieurs fois afin que les logs soient générés, puis accédez à la console Kibana et voyez que les logs sont correctement

empilés dans Kibana avec beaucoup de fonctionnalités supplémentaires comme nous pouvons filtrer, voir différents graphiques, etc.

Voici la vue des logs générés dans Kibana.



Présentation de Kibana Logs



Écran des détails de Kibana Logs

7. Tutoriel ELK Stack - Résumé

Dans cet exemple ELK, nous avons appris à configurer la pile ELK et nous avons vu comment nous pouvons pointer nos fichiers logs d'application vers ELK et afficher et

analyser les logs dans Kibana. Je vais vous proposer de jouer avec les configurations et de partager vos apprentissages avec nous. par exemple.

- Au lieu de logstash pour écouter nos logs, nous pouvons utiliser la configuration de journalisation pour utiliser l'append TCP pour envoyer des logs à une instance Logstash distante via le protocole TCP.
- Nous pouvons pointer plusieurs fichiers logs avec Logstash.
- Nous pouvons utiliser des filtres plus sophistiqués dans le fichier de configuration de logstash pour faire plus selon nos besoins.
- Nous pouvons utiliser un cluster ELK distant pour pointer vers nos fichiers logs, ou pousser les logs vers, cela est fondamentalement nécessaire lorsque toutes les applications seront déployées dans le cloud.
- Créez différents modèles d'index dans logstash.

Spring Cloud - Config Server

L'approche des microservices est maintenant devenue une norme de l'industrie pour tout nouveau développement d'API, et presque toutes les organisations en font la promotion. Spring cloud fournit d'excellents outils pour créer ces microservices en plus du framework Spring Boot.

Dans ce didacticiel de configuration du cloud de Spring, nous aborderons une fonctionnalité spécifique de Microservice appelée Config Server. Le serveur de configuration est l'endroit où tous les paramètres configurables de tous les microservices sont stockés et maintenus.

Cela ressemble plus à l'externalisation du fichier de propriétés / ressources de la base de code du projet vers un service externe afin que toute modification apportée à une propriété donnée ne nécessite pas le redéploiement du service qui utilise cette propriété. Toutes ces modifications de propriété seront reflétées sans redéployer le microservice.

1. Pourquoi utiliser Spring Cloud Config Server

L'idée de serveur de configuration est venue du [manifeste d'application à 12 facteurs](#) lié aux directives des meilleures pratiques de développement d'applications modernes natives dans le cloud. Il suggère d'externaliser les propriétés ou les fichiers de ressources hors du serveur où les valeurs de ces ressources varient pendant l'exécution - généralement des configurations différentes qui diffèrent dans chaque environnement.

À titre d'exemple, disons qu'un service dépend d'un autre service (appelé pour des scénarios commerciaux spécifiques) et si cette URL de service dépendante a été remplacée par autre chose. Ensuite, nous devons généralement créer et déployer notre service avec l'URL mise à jour. Maintenant, si nous suivons l'approche de l'application à 12 facteurs et si nous lisons ces propriétés de configuration à partir d'un service externe, il nous suffit de mettre à jour l'URL dans le serveur de configuration et d'actualiser cette configuration de service client pour utiliser l'URL mise à jour.

Donc, l'idée est évidente et efficace. Voyons maintenant comment créer un serveur de configuration Spring Cloud.

2. Pile technique

Nous utiliserons une API Spring-Cloud basée sur Spring-Boot, facilement disponible et très populaire. Il s'appelle Config Server dans la nomenclature Spring Framework. De plus, nous utiliserons la configuration git pour héberger le fichier de propriétés.

Donc, enfin, notre pile technologique pour cette démo sera:

- Java 1.8
- IDE Eclipse
- Spring Cloud
- Spring
- Repos de Spring
- GitHub comme référentiel de ressources
- Maven
- Client REST

Pour commencer, nous allons développer deux microservices en utilisant spring boot.

- l'un est le service du serveur de configuration, fournissant la configuration au moment de l'exécution
- l'autre est le service client de configuration, utilisant la configuration exposée comme serveur de configuration.

3. Configuration du serveur de configuration

Commençons par créer la partie serveur de configuration avec les étapes données:

Générer la structure du projet

Générer un projet serveur avec Config Server Starter POM

Cloner le repo :
Importer le projet dans votre IDE

Build

L'étape suivante consistera à exécuter `mvn clean install` à partir de l'invite de commande ou de eclipse tout ce que vous êtes à l'aise. Dans cette étape, toutes les dépendances nécessaires seront téléchargées à partir du repo maven. Assurez-vous de l'essayer depuis n'importe quel réseau sur lequel aucune restriction de téléchargement n'est présente. Une intégration réussie à cette étape est indispensable pour passer aux étapes suivantes.

Ajouter l'annotation `@EnableConfigServer`

Ouvrez maintenant la classe Spring Application que Spring a déjà fournie et ajoutez l'annotation `@EnableConfigServer` avant la classe et générez à nouveau le projet. Avec cette annotation, cet artefact agira comme un serveur de configuration de Spring.

Après avoir ajouté cette annotation, la classe ressemblera à ci-dessous - le nom de la classe peut être différent en fonction du nom du projet que vous avez donné lors de la génération. Vous pouvez également changer manuellement le nom de la classe en un nom que vous aimez.

Propriétés du client dans le git repository

La prochaine étape essentielle consiste à créer un référentiel git local. Il peut facilement être converti en référentiel distant ultérieurement en configurant son URL dans le fichier de propriétés. Nous placerons le fichier de propriétés externe [configuration], qui sera utilisé par le microservice du serveur Config pour fournir la configuration externe des propriétés. Nous devons suivre les étapes ci-dessous pour créer un référentiel git local et des exemples de fichiers de propriétés d'archivage.

- Assurez-vous que git shell est installé sur votre machine et que vous pouvez exécuter git bash à partir de l'invite de commande. Pour le vérifier, ouvrez l'invite de commande et tapez git, s'il reconnaît la commande, vous avez probablement installé git, sinon veuillez suivre le site Web de git, téléchargez et installez selon les instructions.
- Créez maintenant un répertoire **config-server-repo** sur votre bureau.
- Ensuite, créez un fichier `config-server-client.properties` dans le répertoire config-server-repo et ajoutez-y le message msg = Hello world - this is from config server.
- Ensuite, créez un autre fichier `config-server-client-development.properties` dans le répertoire config-server-repo et ajoutez le message msg = Hello world - this is from config server – Development environment.

- Ensuite, créez un autre fichier `config-server-client-production.properties` dans le répertoire `config-server-repo` et ajoutez-y le message `msg = Hello world - this is from config server – Production environment`.
- Ici, nous conservons le même nom de propriété pour différents environnements, car nous conservons généralement des propriétés pour différents environnements comme les URL, les informations d'identification, les détails de la base de données, etc. Ici, le point le plus important est que nous devons ajouter un trait d'union (-) avec le nom de l'environnement dans chaque propriété afin que le serveur de configuration le comprenne. En outre, nous devons nommer le fichier de propriétés avec le nom du service client de configuration que nous créerons après cela.
- Ouvrez maintenant l'invite de commande à partir du répertoire **config-server-repo** et exécutez la commande `git init` pour faire de ce répertoire un référentiel git.
- Maintenant, lancez `git add .` pour tout ajouter à ce repo.
- Enfin, nous devons valider le fichier de propriétés en exécutant la commande `git commit -m "initial checkin"`. Cela devrait archiver tous les fichiers du référentiel git. Voici la capture d'écran de l'invite de commande pour le même.

```

CA Command Prompt
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\sajal.chakraborty>cd Desktop\config-server-repo

C:\Users\sajal.chakraborty\Desktop\config-server-repo>dir
Volume in drive C has no label.
Volume Serial Number is AE3E-35AF

Directory of C:\Users\sajal.chakraborty\Desktop\config-server-repo

05-07-2017  10:32 AM    <DIR>          .
05-07-2017  10:32 AM    <DIR>          ..
05-07-2017  10:46 AM                72 config-server-client-development.properties
05-07-2017  10:46 AM                71 config-server-client-production.properties
05-07-2017  10:45 AM                46 config-server-client.properties
                3 File(s)                189 bytes
                2 Dir(s)  449,198,411,776 bytes free

C:\Users\sajal.chakraborty\Desktop\config-server-repo>git init
Initialized empty Git repository in C:/Users/sajal.chakraborty/Desktop/config-server-repo/.git/

C:\Users\sajal.chakraborty\Desktop\config-server-repo>git add .

C:\Users\sajal.chakraborty\Desktop\config-server-repo>git commit -m "Initial Checkin"
[master (root-commit) d40bfb1b] Initial Checkin
Committer: Chakraborty <sajal.chakraborty@accenture.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

3 files changed, 3 insertions(+)
create mode 100644 config-server-client-development.properties
create mode 100644 config-server-client-production.properties
create mode 100644 config-server-client.properties

C:\Users\sajal.chakraborty\Desktop\config-server-repo>

```

Property Check-in in Git

client-config.properties :

msg = Hello world - this is from config server - default profile

client-config-development.properties :

msg = Hello world - this is from config server - Development Environment

client-config-production.properties :

msg = Hello world - this is from config server - Prodcution Environment

Git commands to execute in properties folder

\$ git init

\$ git add .

```
$ git commit -m "initial commit"
```

Pointez le dépôt Git depuis Config Server

Créez un fichier appelé `bootstrap.properties` dans le répertoire `src/main/resources` du projet `spring-config-sever` et ajoutez les lignes ci-dessous.

```
#Server port
server.port = 8888

#Git repo location
spring.cloud.config.server.git.uri=E:\\devsetup\\gitworkspace\\spring-cloud\\config-git-repo

#Verify any repository issue in service startup
spring.cloud.config.server.git.cloneOnStart=true

#Disable security of the Management endpoint
management.security.enabled=false
```

Voyons maintenant ces propriétés.

- `server.port` définit le port sur lequel le serveur intégré démarrera.
- `spring.cloud.config.server.git.uri` liera l'emplacement git pour rechercher la configuration. Ici, nous utilisons le repo git local, mais nous pouvons être basculés vers l'emplacement obtenu distant en modifiant simplement cet emplacement.
- `management.security.enabled=false` désactivera la sécurité de Spring sur les endpoints de gestion comme `/env`, `/refresh` etc. Ceci est pour les paramètres de développement, en production, la sécurité doit être activée.

Donc, cette étape pointera vers un emplacement git et un port de serveur.

Toutes les étapes ci-dessus sont très importantes du côté du serveur de configuration, exécutez maintenant une commande finale `mvn clean install` sur ce projet afin que tout soit compilé correctement et également emballé dans le dossier cible ainsi que dans le référentiel maven local. Nous allons démarrer le service du serveur de configuration une fois que la partie client est prête et nous allons enfin tester la fonctionnalité.

Vérifier la configuration

La commande pour exécuter le service en mode intégré est `java -jar target\spring-config-server-0.0.1-SNAPSHOT.jar`, mais nous reviendrons sur cela dans la partie test.

Pour vérifier si le serveur de configuration peut reconnaître les propriétés, exécutez d'abord le microservice du serveur de configuration à partir de l'invite de commande en utilisant la commande donnée à partir de l'invite de commande de l'emplacement de base du code du projet.

```
java -jar target\spring-config-server-0.0.1-SNAPSHOT.jar
```

Maintenant, ouvrez le navigateur et vérifiez ci-dessous les URL, il retournera la sortie JSON et dans la section `propertySources`, nous pouvons voir toutes les propriétés que nous avons ajoutées dans les propriétés. Cela garantit que le serveur de configuration fonctionne correctement, qu'il a reconnu l'emplacement git et qu'il sert la configuration pour différents environnements.

- `http://localhost: 8888/client-config/development`
- `http://localhost:8888/client-config/production`

Pour vérifier également si un changement dans le fichier de propriétés est reflété par le serveur sans redémarrage, modifiez la valeur de la propriété de l'environnement et du fichier de propriété d'archivage. Exécutez ensuite le endpoint de cet environnement spécifique et vérifiez que la valeur de propriété modifiée doit être reflétée immédiatement sans redémarrer le serveur.

Pour faire l'archivage git, après avoir effectué la modification et enregistré le fichier dans n'importe quel éditeur de texte, exécutez la commande `git add .` et `git commit -m "test"`

4. Configuration du client de configuration

Nous allons maintenant passer à l'implémentation côté client où nous utiliserons ces propriétés à partir d'un microservice séparé, ce qui est notre objectif final - pour externaliser la configuration vers différents services.

Créer un projet Maven

Accédez au portail Web <https://start.spring.io/> et générez le projet client avec les artefacts sélectionnés ci-dessous:

1. Actuator
2. Config Client
3. Web
4. Rest Repositories

L'écran ressemblera à ci-dessous avant la génération; une fois que nous cliquons sur générer, nous obtiendrons l'option de téléchargement de fichier .zip. Comme Spring-Config-Server, décompressez le fichier dans un répertoire et importez-le dans l'IDE.

Cloner le code :

Créer une ressource REST

Ajoutez un `RestController` pour afficher les valeurs de propriété côté serveur dans la réponse. Pour ce faire, ouvrez le fichier de classe `@SpringBootApplication` qui a été généré et ajoutez la petite classe ci-dessous à la fin de ce fichier. C'est très simple et direct, nous exposons juste une méthode à `/message` URL où nous retournerons simplement la valeur de propriété de `msg` qui sera fournie par le microservice du serveur de configuration, qui est configuré sur un référentiel git local (qui sera migré vers un référentiel git distant en production!).

Lier avec le serveur de configuration

Créez un fichier appelé `bootstrap.properties` dans le répertoire `src/main/resources` et ajoutez les propriétés ci-dessous pour vous connecter au serveur de configuration avec une configuration requise.

Voyons maintenant les propriétés.

- `spring.application.name` est simplement le nom d'application du microservice qui serait déployé.
- `spring.cloud.config.uri` est la propriété de mentionner l'url du serveur de configuration. Faites remarquer que notre serveur de configuration fonctionne sur le port 8888; vérifiez-le en ouvrant le fichier `application.properties` de la base de code du serveur Spring Config et vérifiez le `server.port=8888`.
- `management.security.enabled=false` désactivera la sécurité de Spring sur les points de terminaison de gestion tels que `/env`, `/refresh`, etc. C'est pour les paramètres de développement, en production, la sécurité doit être activée.

Vérifier la configuration du client

C'est beaucoup que nous devons faire du côté du client de configuration, pas faire une commande finale `mvn clean install` sur ce projet afin que tout soit compilé correctement et emballé également dans le dossier cible ainsi que dans le référentiel maven local. Nous allons démarrer le service client de configuration avec le côté serveur et nous allons enfin tester la fonctionnalité.

5. Démo

Testons l'application du serveur de configuration.

Créer et exécuter un projet de serveur de configuration

Ouvrez l'invite de commande à partir du dossier spring-config-server et exécutez la commande `mvn clean install`. Une fois la construction terminée, exécutez l'application à partir de cette invite de commande elle-même par la commande `java -jar` telle que `java -jar target\spring-config-server-0.0.1-SNAPSHOT.jar`.

Cela démarrera le service du serveur de configuration sur le port 8888 de l'hôte local.

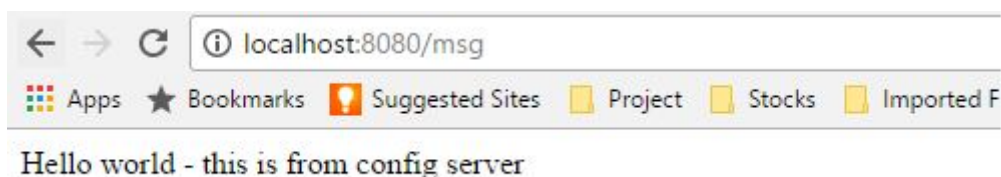
Créer et exécuter un projet client de configuration

De même, ouvrez l'invite de commande à partir du dossier spring-config-client et exécutez la commande `mvn clean install`. Une fois la construction terminée, exécutez l'application à partir de cette invite de commande elle-même par la commande `java -jar` telle que `java -jar target\spring-config-client-0.0.1-SNAPSHOT.jar`.

Cela démarrera le service Config Client dans localhost port 8080.

Tester REST Endpoint

Maintenant, dans le navigateur, ouvrez le endpoint `/msg` rest en parcourant l'url <http://localhost:8080/msg>. Il devrait renvoyer Hello world - this is from config server mentionné dans le fichier `config-server-client-development.properties`.



Test REST End Point

Tester le changement de propriété

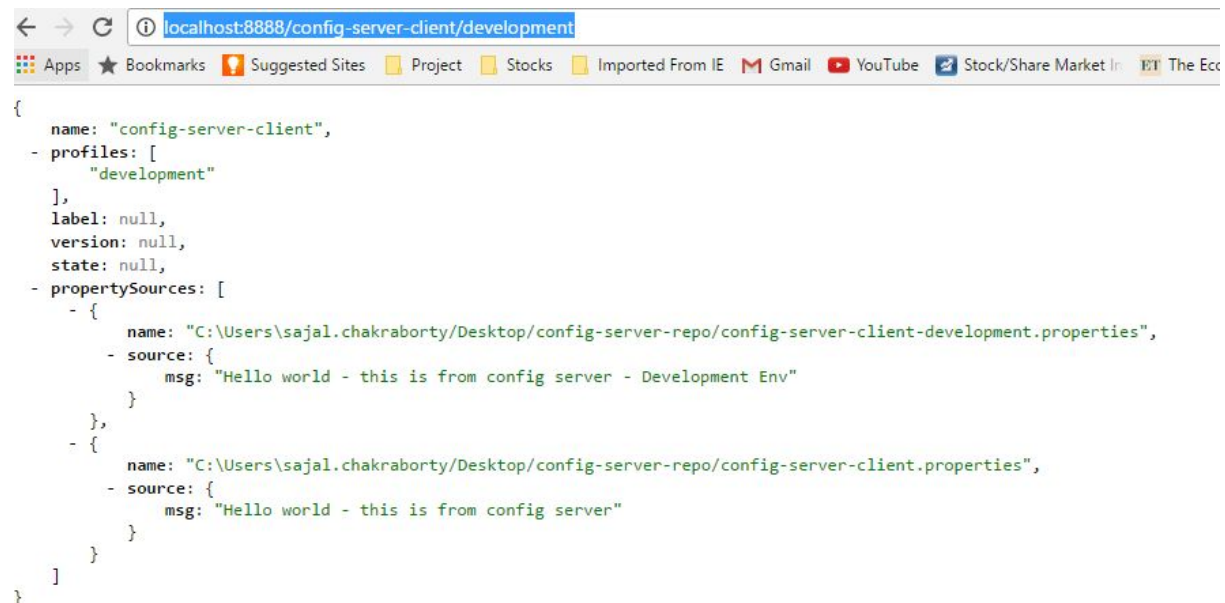
Nous allons maintenant faire un changement de propriété et tester si cela peut être reflété dans le service client de configuration sans redémarrer aucun des microservices. Faites quelques changements, dans la valeur de la propriété msg dans le `config-server-client-development.properties` et enregistrez-vous dans le git local, puis appuyez à nouveau sur <http://localhost:8080/msg> dans le navigateur, vous allez l'ancienne valeur uniquement.

Pour refléter la nouvelle valeur, nous devons actualiser la configuration en appuyant sur <http://localhost:8080/refresh> endpoint à l'aide de la méthode POST à partir de l'un des clients REST.

Une fois que vous avez correctement actualisé le service client de configuration, la nouvelle valeur doit être reflétée dans la réponse du service. C'est parce que `@RefreshScope` annotation le contrôleur de repos que nous avons exposé.

6. Choses à vérifier en cas d'erreur

Le nom du fichier de propriétés et le nom du service du module client `spring.application.name=config-server-client` doivent être exactement les mêmes, sinon les propriétés ne seront pas détectées. En fait, Config Server expose les propriétés dans un point final du nom du fichier de propriétés, si vous parcourez l'URL <http://localhost:8888/config-server-client/development>, il retournera toutes les valeurs d'environnement de développement.



```
{
  name: "config-server-client",
  - profiles: [
    "development"
  ],
  label: null,
  version: null,
  state: null,
  - propertySources: [
    - {
      name: "C:\\Users\\sajal.chakraborty\\Desktop\\config-server-repo\\config-server-client-development.properties",
      - source: {
        msg: "Hello world - this is from config server - Development Env"
      }
    },
    - {
      name: "C:\\Users\\sajal.chakraborty\\Desktop\\config-server-repo\\config-server-client.properties",
      - source: {
        msg: "Hello world - this is from config server"
      }
    }
  ]
}
```

All Dev Properties View

- Assurez-vous d'avoir archivé les fichiers de propriétés dans le référentiel git en utilisant les commandes `git init / add / commit` comme décrit ci-dessus.
- Assurez-vous que vous avez actualisé l'environnement du service client en appelant la méthode POST de <http://localhost:8080/refresh> par n'importe quel client REST. Sinon, les valeurs modifiées ne seront pas reflétées dans le service client.
- Assurez-vous qu'au moment du démarrage du service client de configuration, le service de serveur de configuration est déjà en cours d'exécution. Sinon, l'enregistrement peut prendre un certain temps, ce qui peut créer de la confusion lors du test.

Il s'agit de créer un serveur de configuration pour les microservices. Veuillez ajouter des commentaires si vous rencontrez des difficultés pour configurer tous les points mentionnés dans cet article, nous nous ferons un plaisir de nous pencher sur le problème.

Spring Cloud - Netflix Service Discovery

Apprenez à créer un microservice, basé sur [Spring cloud](#), sur le serveur de registre [Netflix Eureka](#) et comment d'autres microservices (clients Eureka) l'utilisent pour enregistrer et découvrir des services pour appeler leurs API.

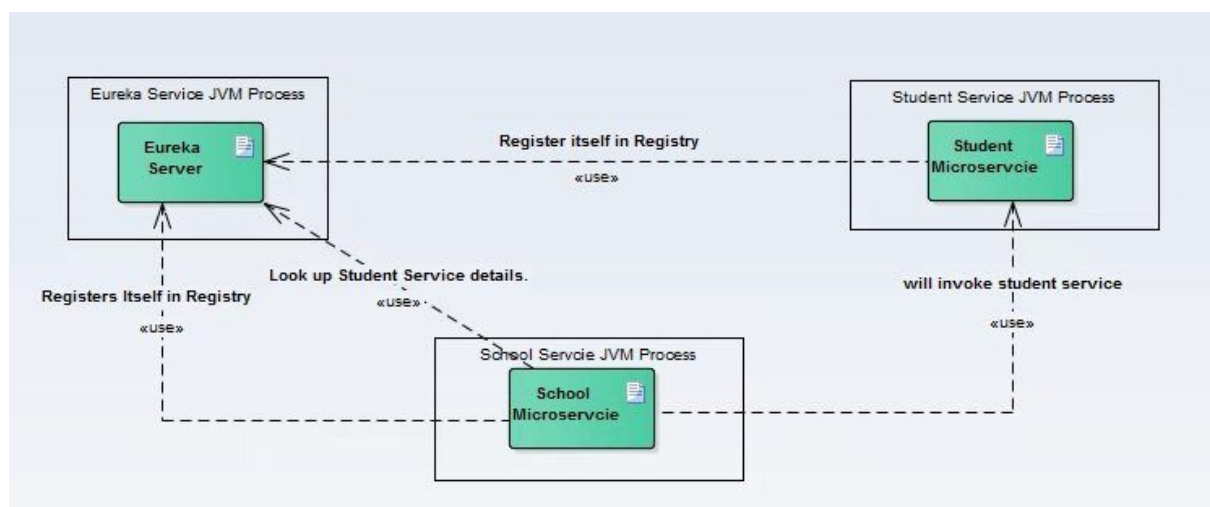
Nous utiliserons l'API Spring Cloud basée sur Spring Boot. Nous utiliserons le serveur Netflix Eureka pour créer le serveur de registre de service et les clients Eureka qui s'enregistrent et découvriront d'autres services pour appeler les API REST.

Overview

Nous allons créer trois microservices pour cet exemple Netflix Eureka.

- **Eureka Service Registry Server**- Ce microservice fournira le registre de service et le serveur de découverte.
- **Student Microservice** - Ce qui donnera des fonctionnalités basées sur l'entité Student. Ce sera un service reposant et surtout un service client eureka, qui discutera avec le service eureka pour s'enregistrer dans le registre des services.
- **School Microservice**- Même type que le service student - la seule fonctionnalité ajoutée est qu'il appellera le service student avec un mécanisme de recherche de service. Nous n'utiliserons pas l'URL absolue du service student pour interagir avec ce service.

Voici le diagramme d'interaction entre les trois services énumérés ci-dessus.



Component Interaction with each other

Qu'est-ce que le serveur et les clients Netflix Eureka?

Comme nous le savons ces jours-ci, il y a beaucoup d'élan autour des microservices. La transition d'une architecture monolithique à une architecture basée sur Microservice offre de nombreux avantages pour l'avenir en termes de maintenabilité, d'évolutivité, de haute disponibilité, etc. Cependant, en même temps, il existe de nombreux défis lors de cette migration. L'un d'eux consiste à gérer les adresses de microservices individuelles. Cette tâche peut être extrêmement complexe - en fonction du nombre de services et de leur nature dynamique. Si toute l'infrastructure est distribuée et qu'il y a également une certaine réplication, la maintenance de ces adresses de service devient plus difficile.

Pour résoudre ce problème, dans l'informatique distribuée, il existe un concept appelé «enregistrement et découverte de service» où un serveur dédié est chargé de maintenir le registre de tous les microservices qui ont été déployés et supprimés. Cela agira comme un annuaire téléphonique de toutes les autres applications / microservices.

Considérez-le comme un service de recherche où les microservices (clients) peuvent s'enregistrer et découvrir d'autres microservices enregistrés. Lorsqu'un microservice client s'enregistre auprès d'Eureka, il fournit des métadonnées telles que l'hôte, le port et l'indicateur d'intégrité permettant ainsi à d'autres microservices de le découvrir. Le serveur de découverte attend un message de pulsation régulier de chaque instance de microservice. Si une instance commence à échouer systématiquement à envoyer une pulsation, le serveur de découverte supprimera l'instance de son registre. De cette façon, nous aurons un écosystème très stable de microservices collaborant entre eux, et en plus de cela, nous n'avons pas à gérer manuellement l'adresse d'un autre microservice, ce qui est une tâche presque impossible si l'échelle haut / bas est très fréquente, à la demande et nous utilisons un hôte virtuel pour héberger les services spécialement dans l'environnement cloud.

Serveur de registre de service Eureka

Suivez ces étapes pour créer et exécuter le serveur Eureka.

Créer un serveur Eureka

Créer une application Spring Boot avec les dépendances `Eureka server` et `Actuator`

Cloner le code :

Maintenant, ouvrez la classe `SpringEurekaServerApplication` que Spring a déjà générée dans le projet téléchargé et ajoutez la `@EnableEurekaServer` sur la classe.

Générez à nouveau le projet. Avec cette annotation, cet artefact agira comme un registre de microservice et un serveur de découverte.

Configuration du serveur

Créez un fichier appelé `application.yml` dans le répertoire `src/main/resources`. Ajoutez ces propriétés -

```
server:
  port: ${PORT:8761} # Indicate the default PORT where this service will be started

eureka:
  client:
    registerWithEureka: false #telling the server not to register himself in the
    service registry
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0 #wait time for subsequent sync
```

Create another file called `bootstrap.yml` in the `src/main/resources` directory. Add these properties –

```
spring:
  application:
    name: eureka
  cloud:
    config:
      uri: ${CONFIG_SERVER_URL:http://localhost:8888}
```

Tester le serveur Eureka

Démarrez l'application en tant qu'application de démarrage de Spring. Ouvrez le navigateur et allez sur <http://localhost:8761/>, vous devriez voir la page d'accueil du serveur eureka qui ressemble à ci-dessous.

The screenshot shows the Spring Eureka console interface. At the top, there's a navigation bar with the 'spring Eureka' logo and links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below this, the 'System Status' section contains two tables. The first table lists 'Environment' as 'test' and 'Data center' as 'default'. The second table shows 'Current time' as '2017-07-07T22:50:44 +0530', 'Uptime' as '00:00', 'Lease expiration enabled' as 'false', 'Renews threshold' as '1', and 'Renews (last min)' as '0'. A red warning message states: 'EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.' Below the warning is the 'DS Replicas' section with a search bar containing 'localhost'. The 'Instances currently registered with Eureka' section shows a table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status', with the message 'No instances available'. Finally, the 'General Info' section displays a table with 'Name' and 'Value' columns, showing 'total-avail-memory' as '222mb', 'environment' as 'test', and 'num-of-cpus' as '4'.

System Status	
Environment	test
Data center	default

System Status	
Current time	2017-07-07T22:50:44 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	222mb
environment	test
num-of-cpus	4

Eureka Console Without Any Client

Veillez noter qu'à ce stade, aucun service n'est enregistré ici, ce qui est attendu et une fois que nous mettrons en marche les services client, ce serveur sera automatiquement mis à jour avec les détails des services client.

Eureka Client – Student Service

Suivez ces étapes pour créer et exécuter le client Eureka exécutant le service student.

Créer un projet Spring boot avec les dépendances : Actuator, Web, Rest Repositories, Eureka Discovery.

Cloner le code depuis :

Ajoutez maintenant l'annotation `@EnableEurekaClient` sur la classe d'application Spring Boot présente dans le dossier `src`. Avec cette annotation, cet artefact agira comme un client Spring Discovery et s'enregistrer lui-même dans le serveur eureka attaché à ce service.

Configuration du client

Créez un fichier appelé `application.yml` dans le répertoire `src/main/resources` et ajoutez les lignes ci-dessous.

```
server:
  port: 8098    #default port where the service will be started
```

```
eureka:      #tells about the Eureka server details and its refresh time
instance:
  leaseRenewalIntervalInSeconds: 1
  leaseExpirationDurationInSeconds: 2
client:
  serviceUrl:
    defaultZone: http://127.0.0.1:8761/eureka/
  healthcheck:
    enabled: true
  lease:
    duration: 5

spring:
  application:
    name: student-service #current service name to be used by the eureka server

management:
  security:
    enabled: false #disable the spring security on the management endpoints like /env,
    /refresh etc.

logging:
  level:
    com.example.howtodoinjava: DEBUG
```

Ajouter l'API REST

Ajoutez maintenant un RestController et exposez un endpoint de repos pour obtenir tous les détails de student pour une school particulière. Ici, nous exposons le Endpoint `/getStudentDetailsForSchool/{schoolname}` pour servir l'objectif commercial. Pour plus de simplicité, nous codons en dur les détails du student.

La class Student est un simple POJO

Tester le client Eureka

Démarrez ce projet en tant qu'application de démarrage de Spring. Vérifiez maintenant que ce service a été enregistré automatiquement sur le serveur Eureka. Accédez à la console de service Eureka et actualisez la page. Maintenant, si tout se passe bien, nous verrons une entrée pour le student-service dans la console de service eureka. Cela indique que le serveur et le client Eureka se reconnaissent mutuellement.

The screenshot shows the Spring Eureka console interface. At the top, there's a navigation bar with the Spring Eureka logo and links for HOME and LAST 1000 SINCE STARTUP. Below this, the 'System Status' section displays two tables. The left table shows 'Environment' as 'test' and 'Data center' as 'default'. The right table shows 'Current time' as '2017-07-07T23:59:39 +0530', 'Uptime' as '01:09', 'Lease expiration enabled' as 'true', 'Renews threshold' as '3', and 'Renews (last min)' as '232'. Below the system status is the 'DS Replicas' section showing 'localhost'. The 'Instances currently registered with Eureka' section shows a table with one instance: 'STUDENT-SERVICE' with 'n/a (1)' AMIs, '(1)' Availability Zones, and status 'UP (1) - Sajal-HP-student-service:8098'. Finally, the 'General Info' section shows a table with system metrics: 'total-avail-memory' (238mb), 'environment' (test), 'num-of-cpus' (4), 'current-memory-usage' (79mb (33%)), and 'server-up-time' (01:09).

Environment	test
Data center	default

Current time	2017-07-07T23:59:39 +0530
Uptime	01:09
Lease expiration enabled	true
Renews threshold	3
Renews (last min)	232

DS Replicas

localhost

Application	AMIs	Availability Zones	Status
STUDENT-SERVICE	n/a (1)	(1)	UP (1) - Sajal-HP-student-service:8098

Name	Value
total-avail-memory	238mb
environment	test
num-of-cpus	4
current-memory-usage	79mb (33%)
server-up-time	01:09

Eureka console with Student service registered

Nous allons maintenant vérifier que le endpoint `/getStudentDetailsForSchool/{schoolname}` est opérationnel. Allez dans le navigateur et allez à <http://localhost:8098/getStudentDetailsForSchool/abcschool> , il donnera à student les détails d'une school particulière abcschool

The screenshot shows a web browser with the URL `sajal-hp:8098/getStudentDetailsForSchool/abcschool` in the address bar. The response is a JSON array containing two student objects. The first object has a name of 'Sajal' and a className of 'Class IV'. The second object has a name of 'Lokesh' and a className of 'Class V'.

```
[
  - {
    name: "Sajal",
    className: "Class IV"
  },
  - {
    name: "Lokesh",
    className: "Class V"
  }
]
```

Student Service response

Client Eureka - Service School

Maintenant, nous allons créer un service school qui s'enregistrera avec le serveur eureka - et il découvrira et invoquera le service student sans chemin d'URL codé en dur.

Suivez les étapes exactes pour créer un service student, pour créer et exécuter également le service school client Eureka.

Créer un projet client Eureka (spring boot)

Ajoutez maintenant l'annotation `@EnableEurekaClient` sur la classe d'application Spring Boot présente dans le dossier `src`. Avec cette annotation, cet artefact agira comme un client Spring Discovery et s'enregistrera lui-même dans le serveur eureka attaché à ce service.

Cloner le code :

Configuration du client

Créez un fichier appelé `application.yml` dans le répertoire `src/main/resources` et ajoutez les lignes ci-dessous. Ces configurations sont très similaires au service student, à l'exception du numéro de port et du nom du service.

```
server:
  port: 9098    #port number

eureka:
  instance:
    leaseRenewalIntervalInSeconds: 1
    leaseExpirationDurationInSeconds: 2
  client:
    serviceUrl:
      defaultZone: http://127.0.0.1:8761/eureka/
    healthcheck:
      enabled: true
    lease:
      duration: 5

spring:
  application:
    name: school-service    #service name

logging:
```

```
level:  
com.example.howtodoinjava: DEBUG
```

Ajouter l'API REST qui utilise l'API REST du service student

Ajoutez maintenant un `RestController` et exposez un Endpoint de repos pour obtenir les détails de school. Ce endpoint utilisera l'URL de style de découverte de service en utilisant le nom de l'application, au lieu de l'URL complète avec host:port.

De cette façon, nous pouvons nous débarrasser de la configuration de service spécifique et donner la responsabilité de la recherche de service au serveur eureka et au modèle de repos fourni ici. Nous pouvons également appliquer l'équilibrage de charge (voir l'annotation `@LoadBalanced`) ici si les multiples instances sont en cours d'exécution pour le même service.

L'URL que nous avons utilisée est

<http://student-service/getStudentDetailsForSchool/{schoolname}>. Il est clair que nous n'utilisons que le nom de service `student-service` à la place de `host:port`. Cela sera géré en interne par le framework Spring, le serveur eureka et le modèle de repos ensemble.

Démo Du Service Discovery and Calling

Maintenant, démarrez également le service school. Les trois services sont démarrés. Vérifiez la console du serveur eureka. Les services students et schools doivent y être enregistrés.

The screenshot shows the Spring Eureka console interface. At the top, there's a navigation bar with the Spring Eureka logo and links for HOME and LAST 1000 SINCE STARTUP. Below this, the 'System Status' section displays environment details (test, default) and system metrics (current time, uptime, lease expiration, renewals threshold, and last renewal). The 'DS Replicas' section shows the local instance 'localhost'. The 'Instances currently registered with Eureka' section lists two services: 'SCHOOL-SERVICE' and 'STUDENT-SERVICE', both with one instance each. The 'General Info' section at the bottom provides system-level details like total available memory, environment, number of CPUs, and current memory usage.

System Status	
Environment	test
Data center	default
Current time	2017-07-08T00:21:42 +0530
Uptime	01:31
Lease expiration enabled	true
Renews threshold	5
Renews (last min)	117

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SCHOOL-SERVICE	n/a (1)	(1)	UP (1) - Sajal-HP:school-service:9098
STUDENT-SERVICE	n/a (1)	(1)	UP (1) - Sajal-HP:student-service:8098

General Info	
Name	Value
total-avail-memory	245mb
environment	test
num-of-cpus	4
current-memory-usage	67mb (27%)

Eureka console with both services registered

Allez dans le navigateur et allez à <http://localhost:9098/getSchoolDetails/abcschool> , il donnera les détails de school pour une school particulière `abcschool`. Nous avons invoqué le service student en interne. La réponse ressemblera à celle du navigateur:



School Service Response

Choses à vérifier en cas d'erreur

- Les annotations `@EnableEurekaServer` et `@EnableEurekaClient` sont au cœur de l'écosystème applicatif. Sans ces deux choses, cela ne fonctionnera pas du tout.
- Assurez-vous qu'au moment du démarrage du service client de configuration, le service serveur eureka est déjà en cours d'exécution, sinon l'enregistrement pourrait prendre un certain temps, ce qui pourrait créer une confusion lors des tests.

Spring Cloud - Consul Service Discovery

Apprenez à créer des microservices, basés sur Spring cloud, en vous enregistrant sur le serveur de registre [HashiCorp Consul](#) et comment d'autres microservices (clients de découverte) l'utilisent pour enregistrer et découvrir des services pour appeler leurs API.

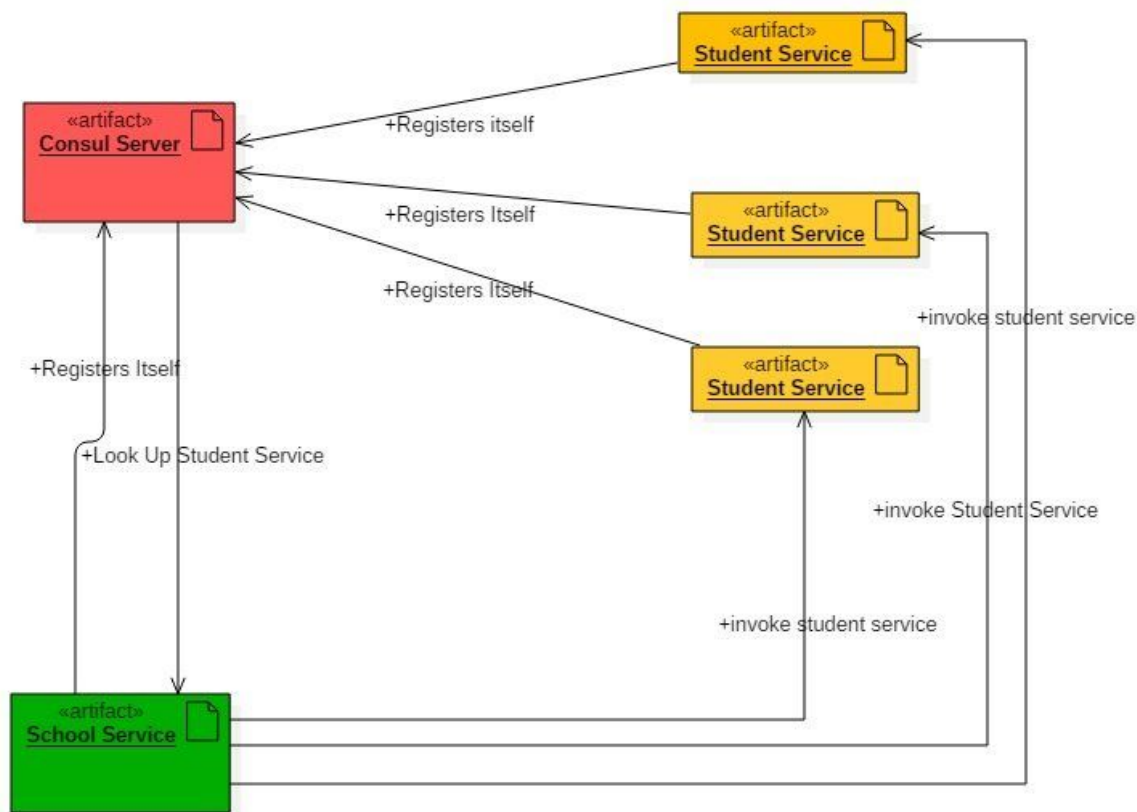
Nous utiliserons l'API Spring Cloud basée sur Spring Boot. Nous utiliserons le serveur de registre Consul pour créer le serveur de registre de service et les clients de découverte génériques qui s'enregistreront et découvriront d'autres services pour appeler les API REST.

Overview

Consul fournit de multiples fonctionnalités telles que la découverte de services, la gestion de la configuration, la vérification de l'état de santé et le stockage de clés-valeurs, etc. Nous développerons les composants ci-dessous pour construire un système Eco distribué où chaque composant dépend en quelque sorte les uns des autres, mais ils sont très faiblement couplés et bien sûr tolérants aux pannes.

- **Agent Consul** - fonctionnant sur l'hôte local agissant en tant que fonctionnalité de serveur de découverte / registre.
- **Student Microservice** - qui donnera des fonctionnalités basées sur l'entité Student. Ce sera un service basé sur le repos et, surtout, un client de service de découverte, qui parlera avec le serveur / agent Consul pour s'enregistrer dans le registre de service.
- **School Microservice** - Même type que le service student - la seule fonctionnalité ajoutée est qu'il appellera le service student avec un mécanisme de recherche de service. Nous n'utiliserons pas l'URL absolue du service student pour interagir avec ce service. Nous utiliserons la fonction de découverte Consul et l'utiliserons pour rechercher une instance de service student avant de l'appeler.

Voici le diagramme global d'interaction des composants pour le même.



Component Diagram

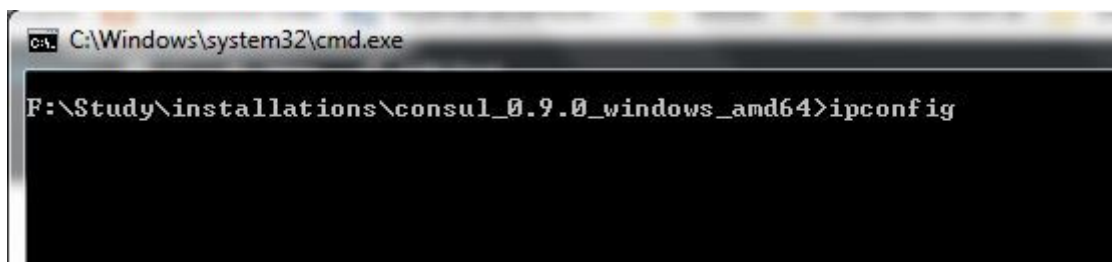
Configuration de Consul dans le poste de travail local

Avant de commencer l'exercice, nous devons d'abord télécharger, configurer et exécuter l'agent consul dans localhost.

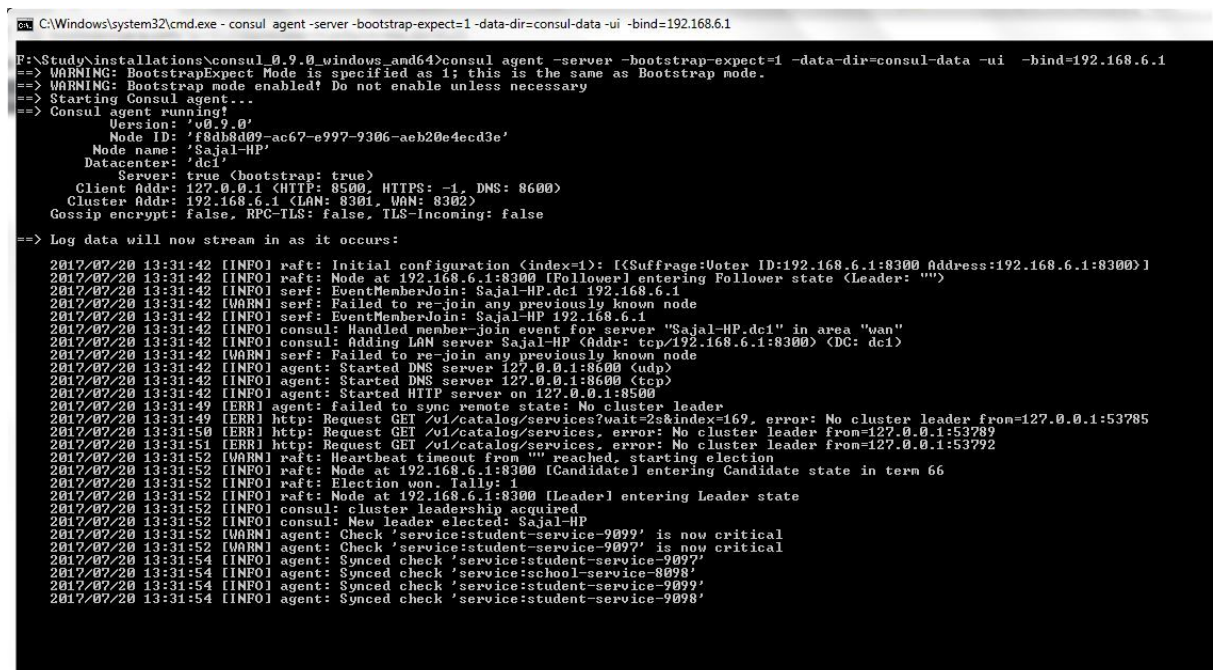
- Téléchargement depuis le [portail Consul](#). Choisissez un package particulier en fonction du système d'exploitation. Une fois téléchargé le zip, nous devons le décompresser à l'endroit souhaité.
- Démarrez l'Agent Consul sur le poste de travail local - Le fichier Zip que nous avons décompressé ne contient qu'un seul fichier exe appelé `consul.exe`. Nous allons démarrer une invite de commande ici et utiliser la commande ci-dessous pour démarrer l'agent.

```
consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui
-bind=192.168.6.1
```

Assurez-vous de saisir l'adresse de liaison correcte, elle serait différente en fonction des paramètres LAN. Faites un `ipconfig` dans l'invite de commande pour connaître votre adresse IPv4 et l'utiliser ici.

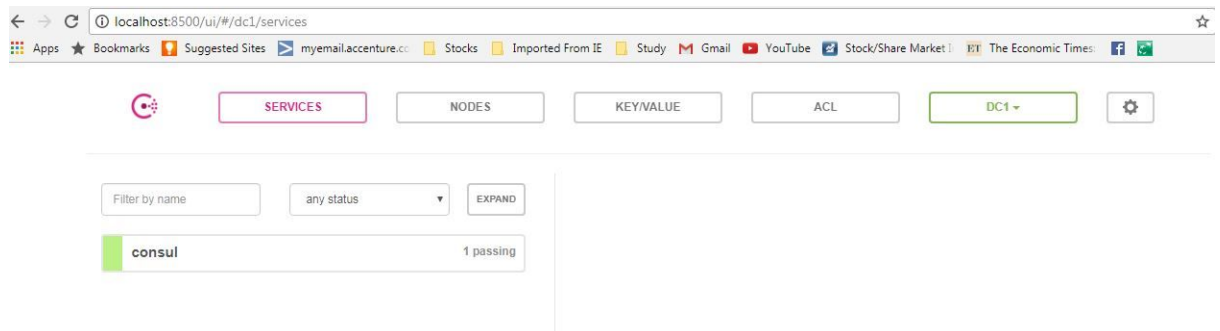


ipconfig command



Agent Start Command Log

Testez si Consul Server est en cours d'exécution - Consul s'exécute sur le port par défaut et une fois que l'agent a démarré avec succès, parcourez <http://localhost:8500/ui> et vous devriez voir un écran de console comme



Consul console – No service registered

Nous avons donc configuré consul dans notre machine locale et l'agent consul fonctionne avec succès. Nous devons maintenant créer des clients et tester le registre de service et la partie découverte.

Student Service

Créer un projet Spring Boot avec les dépendances : Actuator, Web, Consul Discovery et Rest Repositories.

Ajoutez maintenant l'annotation

`@org.springframework.cloud.client.discovery.EnableDiscoveryClient` sur la classe d'application de démarrage Spring présente dans le dossier `src`. Avec cette annotation, cet artefact agira comme un client de Spring Discovery et s'enregistrera dans le serveur consul attaché à ce service.

Configuration du service

Ouvrez `application.properties` et ajoutez ci-dessous les propriétés

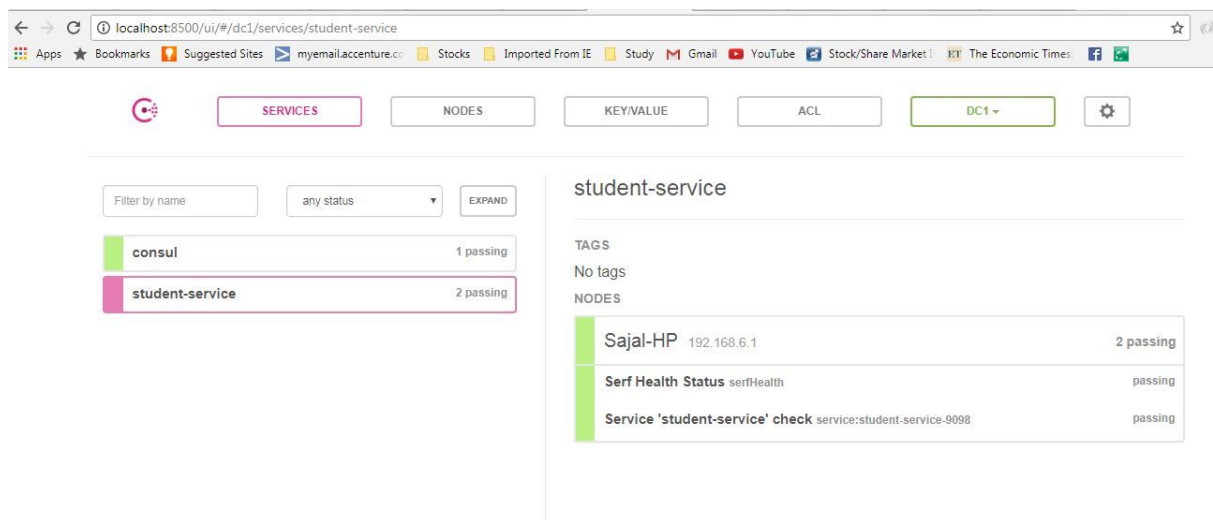
- `server.port = 9098` - démarrera le service dans le port par défaut 9098.
- `spring.application.name: student-service` - s'enregistrera dans le serveur consul à l'aide de la balise `student-service` et d'autres services rechercheront ce service avec ce nom lui-même.
- `management.security.enabled = false` - n'est pas réellement requis pour cet exercice, mais cela désactivera la sécurité de Spring dans les points de terminaison de gestion fournis par le module d'actionneur.

Ajouter des API REST

Ajoutez maintenant un `RestController` et exposez un endpoint de repos pour obtenir tous les détails de student pour une school particulière. Ici, nous exposons le endpoint `/getStudentDetailsForSchool/{schoolname}` pour servir l'objectif commercial. Pour plus de simplicité, nous codons en dur les détails du student.

Vérifier le service student

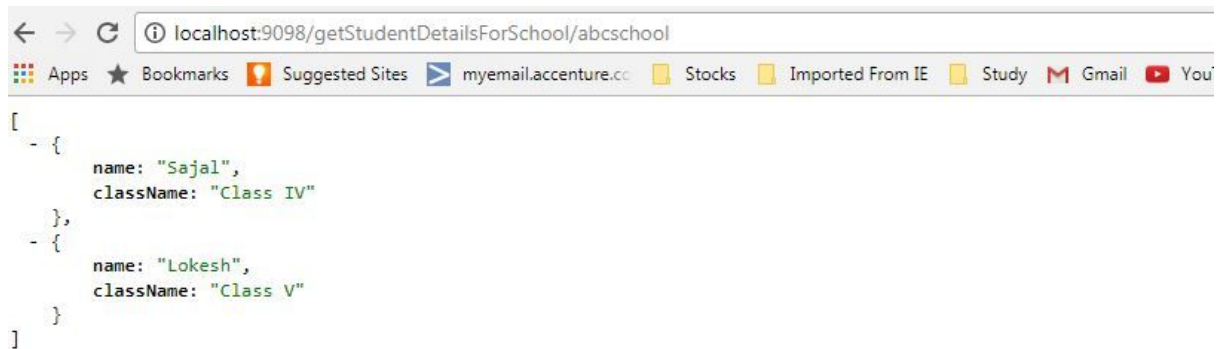
Démarrez ce projet en tant que spring boot project. Vérifiez maintenant que ce service a été enregistré automatiquement sur le serveur Consul. Accédez à la console Consul Agent et actualisez la page. Maintenant, si tout se passe bien, nous verrons une entrée pour le `student-service` dans la console Consul Agent.



Consul console Student Service registered

Cela indique que le serveur et le client Consul sont au courant l'un de l'autre et qu'il s'agit d'une sorte d'enregistrement automatique et de découverte entre le serveur consul et le service student.

Nous allons maintenant vérifier que le endpoint `/getStudentDetailsForSchool/{schoolname}` est opérationnel. Allez dans le navigateur et allez à <http://localhost:9098/getStudentDetailsForSchool/abcschool> , il donnera les détails de student pour une school **abcschool** particulière.



```
[
  {
    name: "Sajal",
    className: "Class IV"
  },
  {
    name: "Lokesh",
    className: "Class V"
  }
]
```

Student Service Response

School Service – Discovery Client

Maintenant, nous allons créer un service school qui s'enregistrera avec le serveur consul - et il découvrira et invoquera le service student sans chemin URL codé en dur.

Suivez les mêmes étapes pour créer et exécuter le service school. Ce sera un client de découverte s'enregistrant sur le service de consultation qui est déjà en cours d'exécution dans notre machine en ce moment.

Il appellera en interne le service student déjà développé et utilisera la fonction de découverte du service consul pour découvrir l'instance student.

Create School Project

Créer un projet SB avec les dépendances : Actuator, Web, Rest Repository, Consul Discovery

Cloner le code :

Ajoutez maintenant l'annotation

`@org.springframework.cloud.client.discovery.EnableDiscoveryClient` sur la classe d'application de démarrage Spring présente dans le dossier src. Avec cette annotation, cet artefact agira comme un client de Spring Discovery et s'enregistrera dans le serveur consul attaché à ce service.

Configuration du service

Ouvrez application.properties et ajoutez ci-dessous les propriétés

```
server.port=8098
spring.application.name: school-service
```

```
management.security.enabled=false
```

Ajouter l'API REST qui utilise l'API REST du service Student

Ajoutez maintenant un RestController et exposez un endpoint pour obtenir les détails de school. Cet Endpoint utilisera l'URL de style de Discovery Service en utilisant le nom de l'application, à la place de la convention d'URL complète avec host: port.
(SchoolServiceController.java)

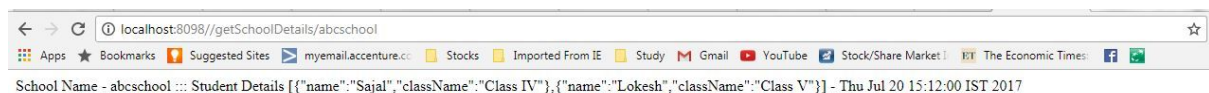
Regardez le code ci-dessus. Dans `StudentServiceDelegate`, nous avons utilisé `RestTemplate` pour appeler le service student et utilisé l'URL du service student comme `http://student-service/getStudentDetailsForSchool/{schoolname}`.

De cette façon, nous pouvons nous débarrasser de la configuration de service spécifique et nous pouvons confier la responsabilité de la recherche de service au serveur `consul` et au modèle de repos fourni ici. Nous pouvons également appliquer l'équilibrage de charge (voir l'annotation `@LoadBalanced`) ici si les multiples instances sont en cours d'exécution pour le même service.

Demo

Suivez les étapes suivantes une par une pour tout comprendre -

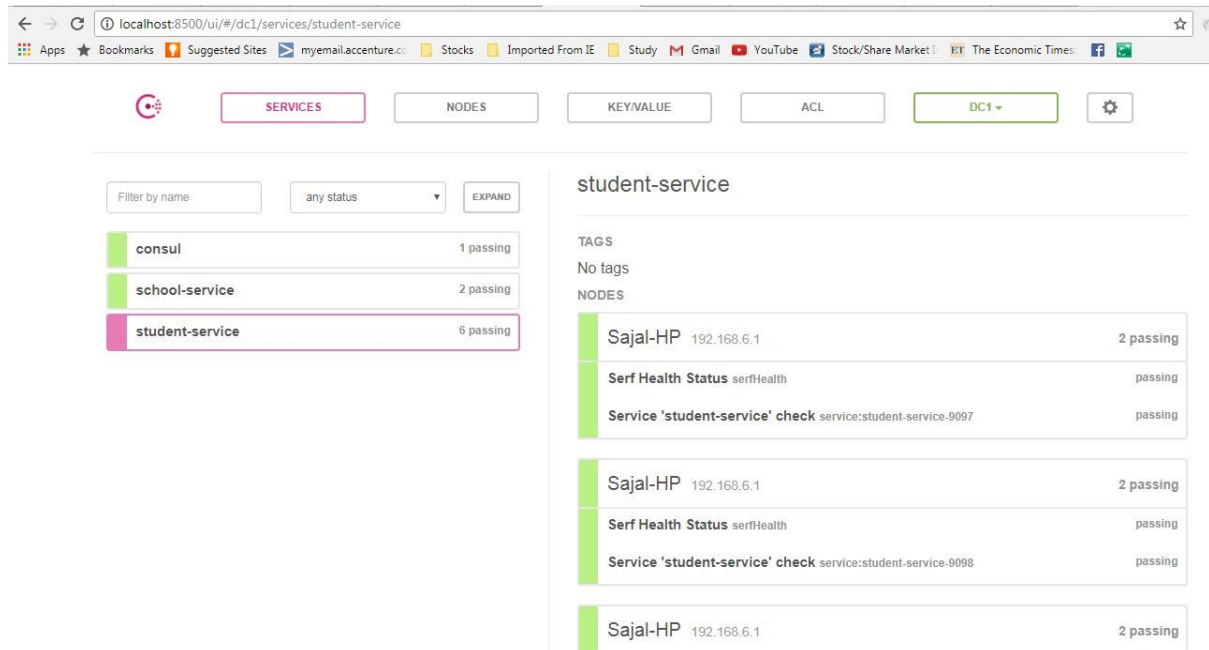
- **Vérifiez que l'agent Consul est toujours en cours d'exécution** - Ouvrez le navigateur et parcourez <http://localhost:8500/ui>. Il doit afficher la console consul comme décrit ci-dessus.
- **Vérifier que le service student est déjà en cours d'exécution** - Vérifiez à partir de la page d'administration du consul et du navigateur que le service student est opérationnel. Sinon, démarrez ce service et vérifiez qu'il a été enregistré sur le serveur consul.
- **Démarrer et vérifier le service school** - Démarrez le service school à partir de l'invite de commande et vérifiez qu'il a été enregistré sur le serveur consul.
- Ouvrez le navigateur et testez le service REST de school en utilisant l'URL <http://localhost:8098//getSchoolDetails/abcschool>. Il donnera la réponse ci-dessous et il invoquera en interne le service student en utilisant le service consul.



School Service Response

Essayez également de démarrer plusieurs instances Student Service en changeant le port par `java -jar "-Dserver.port = 9099 target\spring-cloud-consul-student-0.0.1-SNAPSHOT.jar`. Ceux-ci seront également enregistrés dans consul et comme nous le sommes en utilisant l'annotation `@LoadBalanced` dans le `RestTemplate`, l'équilibrage de charge sera également effectué en interne. Vérifiez la console de service de student respective pour vérifier quelle instance a été appelée dans le scénario multi-instance.

Voici à quoi ressemblera le serveur consul une fois que nous aurons enregistré plusieurs services et plusieurs instances.



Consul All Services Running

Choses à vérifier en cas d'erreur

- Annotation `@EnableDiscoveryClient` et l'agent Consul en cours d'exécution sont au cœur de l'écosystème applicatif. Sans ces deux choses, cela ne fonctionnera pas du tout.
- Assurez-vous qu'au moment du démarrage du service school, du service student, de l'agent du serveur consul est déjà en cours d'exécution. Sinon, l'enregistrement peut prendre un certain temps et entraîner une confusion lors du test.

Spring Cloud - Hystrix Circuit Breaker

Apprenez à tirer parti du composant de pile `Spring Cloud Netflix` appelé `Hystrix` pour implémenter un Circuit Breaker tout en appelant le microservice sous-jacent. Il est

généralement nécessaire d'activer la tolérance aux pannes dans l'application où un service sous-jacent est en panne / génère une erreur de manière permanente, nous devons revenir automatiquement à un chemin différent d'exécution du programme. Ceci est lié au style de calcul distribué du système Eco utilisant de nombreux microservices sous-jacents. C'est là que le modèle de Circuit Breaker aide et **Hystrix** est un outil pour construire ce circuit breaker.

Exemple Hystrix : cas d'usage

La configuration Hystrix se fait en quatre étapes principales.

1. Ajoutez des dépendances de démarrage et de tableau de bord Hystrix.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

2. Ajouter l'annotation **@EnableCircuitBreaker**
3. Ajouter une annotation **@EnableHystrixDashboard**
4. Ajouter une annotation **@HystrixCommand** (fallbackMethod = "myFallbackMethod")

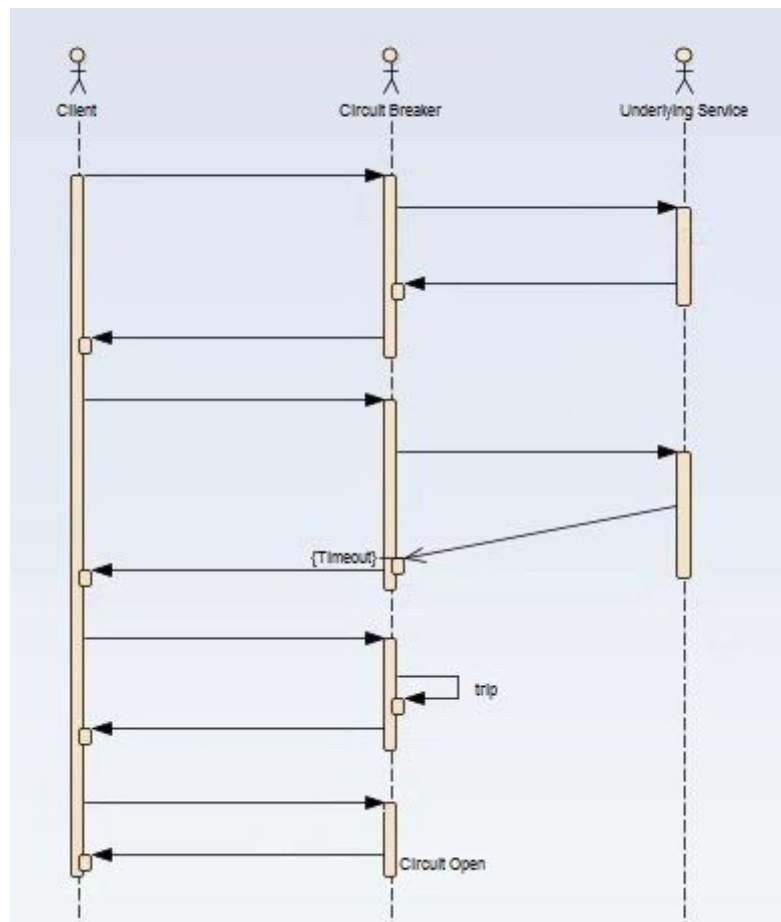
Qu'est-ce que le modèle de Circuit Breaker

Si nous concevons nos systèmes sur une architecture basée sur des microservices, nous développerons généralement de nombreux microservices et ceux-ci interagiront fortement les uns avec les autres pour atteindre certains objectifs commerciaux. Maintenant, nous pouvons tous supposer que cela donnera le résultat attendu si tous les services sont opérationnels et que le temps de réponse de chaque service est satisfaisant.

Maintenant, que se passera-t-il si un service, du système Eco actuel, rencontre un problème et cesse de répondre aux demandes. Cela entraînera des délais timeout/exception et l'ensemble du système Eco deviendra instable en raison de ce point de défaillance unique.

Ici, le modèle de Circuit Breaker est pratique et il redirige le trafic vers un chemin de repli une fois qu'il voit un tel scénario. En outre, il surveille de près le service défectueux et rétablit le trafic une fois que le service est revenu à la normale.

Ainsi, le Circuit Breaker est une sorte d'enveloppe de la méthode qui effectue l'appel de service et il surveille l'état du service et une fois qu'il a un problème, le Circuit Breaker se déclenche et tous les autres appels vont au Circuit Breaker retombent et enfin se rétablit automatiquement une fois le service est revenu !! C'est cool, non?



Circuit Breaker Sequence of Invocation

Exemple de Circuit Breaker Hystrix

Pour démo Circuit Breaker, nous allons créer les deux microservices suivants où le premier dépend d'un autre.

- Microservice Student- Ce qui donnera des fonctionnalités de base sur l'entité Student. Ce sera un service basé sur REST. Nous appellerons ce service du service School pour comprendre Circuit Breaker. Il fonctionnera sur le port 8098 dans localhost.
- Microservice School- Encore une fois, un microservice simple basé sur REST où nous allons implémenter un Circuit Breaker en utilisant Hystrix. Le service Student sera appelé à partir d'ici et nous testerons le chemin de secours une fois que le service Student sera indisponible. Il fonctionnera sur le port 9098 dans localhost.

Tech Stack

Java 1.8

IDE

Maven

Spring Cloud Hystrix

Spring boot

Spring Rest

Créer un service Student

Suivez ces étapes pour créer et exécuter le service Student- un service REST simple fournissant certaines fonctionnalités de base de l'entité Student.

Créer un projet Spring Boot

Paramètres du port du serveur

Ouvrez `application.properties` et ajoutez des informations sur le port.

```
server.port = 8098
```

Cela permettra à cette application de s'exécuter sur le port par défaut 8098. Nous pouvons facilement remplacer cela en fournissant l'argument `-Dserver.port = XXXX` au moment du démarrage du serveur.

Créer des API REST

Ajoutez maintenant une classe de contrôleur REST appelée `StudentServiceController` et exposez un endpoint de repos pour obtenir tous les détails de student pour un school particulière. Ici, nous exposons le endpoint `/getStudentDetailsForSchool/{schoolname}` pour servir l'objectif commercial. Pour plus de simplicité, nous codons en dur les détails de Student. (`StudentServiceController.java`)

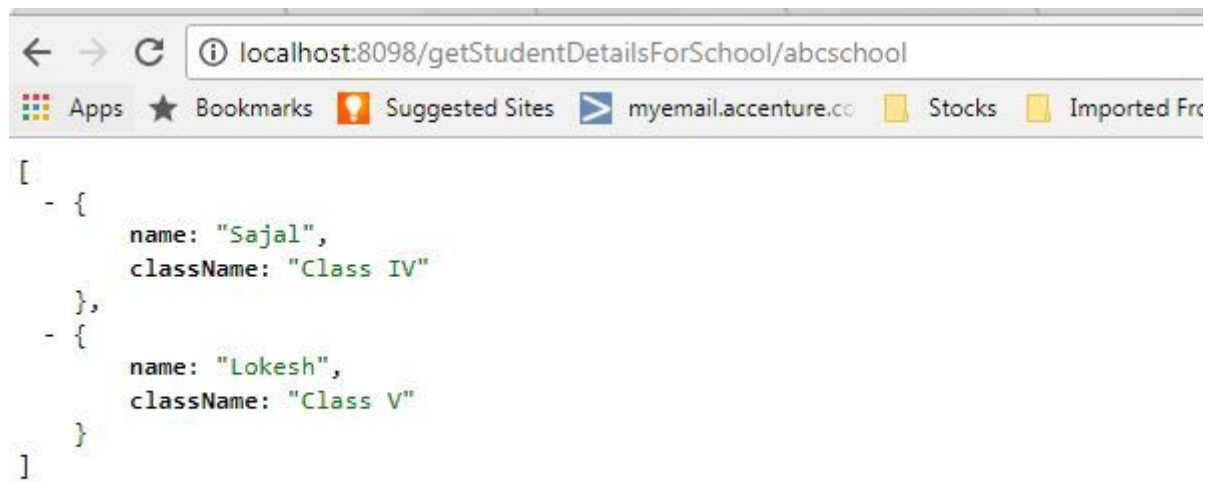
Créer et tester le service Student

Maintenant, faites une version finale à l'aide de `mvn clean install` et exécutez le serveur à l'aide de la commande `java -jar target\spring-hystrix-student-service-0.0.1-SNAPSHOT.jar`. Cela démarrera le service student dans le port par défaut 8098.

Ouvrez le navigateur et tapez

<http://localhost:8098/getStudentDetailsForSchool/abcschool> .

Il devrait afficher la sortie ci-dessous dans le navigateur -



```
[
  - {
    name: "Sajal",
    className: "Class IV"
  },
  - {
    name: "Lokesh",
    className: "Class V"
  }
]
```

Student Service Response

Créer un service school - Hystrix activé

Similaire au service student, créez un autre microservice pour school. Il invoquera en interne le service student déjà développé.

Générer un projet Spring-boot

Créez un projet de démarrage Spring à partir du portail d'initialisation Spring Boot avec ces dépendances principalement.

- **Web** – REST Endpoints
- **Actuator** – providing basic management URL
- **Hystrix** – Enable Circuit Breaker
- **Hystrix Dashboard** – Enable one Dashboard screen related to the Circuit Breaker monitoring

Paramètres du port du serveur

Ouvrez application.properties et ajoutez des informations sur le port.

```
server.port = 9098
```

Cela permettra à cette application de s'exécuter sur le port par défaut 9098. Nous pouvons facilement remplacer cela en fournissant l'argument -Dserver.port = XXXX au moment du démarrage du serveur.

Activer les paramètres Hystrix

Ouvrez `SpringHystrixSchoolServiceApplication`, c'est-à-dire la classe générée avec `@SpringBootApplication` et ajoutez les annotations `@EnableHystrixDashboard` et `@EnableCircuitBreaker`.

Cela activera le Circuit Breaker Hystrix dans l'application et ajoutera également un tableau de bord utile fonctionnant sur l'hôte local fourni par Hystrix.

Ajouter un contrôleur REST

Ajoutez `SchoolServiceController` Rest Controller où nous exposerons le endpoint `/getSchoolDetails/{schoolname}` qui renverra simplement les détails de school ainsi que les détails de student. Pour les détails de student, il appellera le endpoint du service student déjà développé. Nous allons créer une couche de délégué `StudentServiceDelegate.java` pour appeler le service Student. (`SchoolServiceController.java`)

StudentServiceDelegate

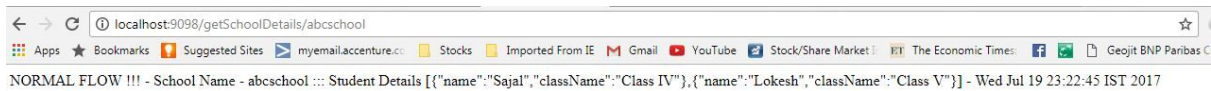
Nous allons faire les choses suivantes ici pour activer le circuit breaker Hystrix.

- Appelez le service student via le framework Spring fourni par `RestTemplate`
- Ajoutez la commande Hystrix pour activer la méthode de secours - `@HystrixCommand (fallbackMethod = "callStudentServiceAndGetData_Fallback")` - cela signifie que nous devons ajouter une autre méthode `callStudentServiceAndGetData_Fallback` avec la même signature, qui sera appelée lorsque le service Student réel sera arrêté.
- Ajoutez une méthode de secours - `callStudentServiceAndGetData_Fallback` qui renverra simplement une valeur par défaut.

Construction et test du service School

Maintenant, faites une version finale en utilisant `mvn clean install` et exécutez le serveur en utilisant la commande `java -jar target\spring-hystrix-school-service-0.0.1-SNAPSHOT.jar`. Cela démarrera le service School dans le port par défaut `9098`.

Démarrez le service Student comme décrit ci-dessus, puis testez le service School en ouvrant le navigateur et tapez <http://localhost:9098/getSchoolDetails/abcschool> . Il devrait afficher la sortie ci-dessous dans le navigateur:



School Service Response

Test du Circuit Breaker Hystrix - D mo

Ouvrez le navigateur et tapez <http://localhost:9098/getSchoolDetails/abcschool> .

Il devrait afficher la sortie ci-dessous dans le navigateur -



School Service Response

Maintenant, nous savons d j  que le service school appelle le service student en interne et qu'il obtient les d tails des Student de ce service. Donc, si les deux services sont en cours d'ex cution, le service School affiche les donn es renvoy es par le service Student, comme nous l'avons vu dans la sortie du navigateur du service School ci-dessus. Il s'agit de l' tat **CIRCUIT CLOSED**.

Arr tons maintenant le service student en appuyant simplement sur **CTRL + C** dans la console du serveur de service student (arr tons le serveur) et testons   nouveau le service School   partir du navigateur. Cette fois, il renverra la r ponse de la m thode de secours. Ici, Hystrix entre en image, il surveille le service Student   intervalles fr quents et, comme il est en panne, le composant Hystrix a ouvert le circuit et le chemin de secours activ .

Voici la sortie de secours dans le navigateur.



School Service Response Fallback path

Red marrez le service Student, attendez quelques instants et retournez au service School et il recommencera   r pondre dans un flux normal.

Tableau de bord Hystrix

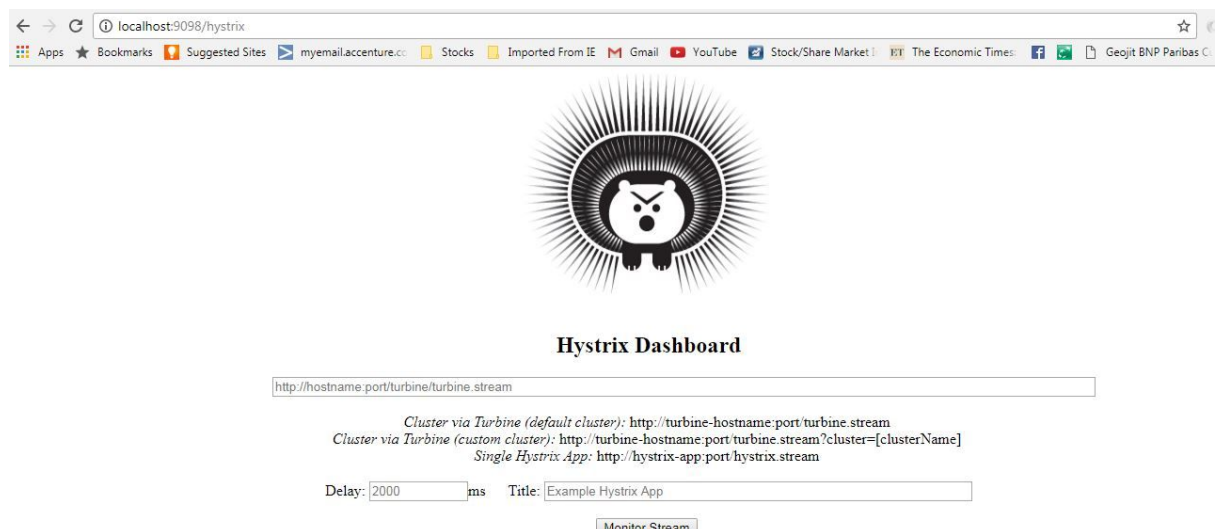
Comme nous avons ajouté la dépendance au tableau de bord hystrix, hystrix a fourni un joli tableau de bord et un flux Hystrix dans les URL ci-dessous:

<http://localhost:9098/hystrix.stream> - C'est un flux continu généré par Hystrix. Il s'agit simplement d'un résultat de vérification de l'état ainsi que de tous les appels de service surveillés par Hystrix. L'exemple de sortie ressemblera à celui du navigateur -

```
ping:
data:
{"type":"HystrixCommand","name":"callStudentServiceAndGetData","group":"StudentServiceDelegate","currentTime":1500489058032,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":4,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmit":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":4,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":1026,"latencyExecute":{"0":1002,"25":1002,"50":1002,"75":1051,"90":1051,"95":1051,"99":1051,"99.5":1051,"100":1051},"latencyTotal_mean":1029,"latencyTotal":{"0":1003,"25":1003,"50":1003,"75":1055,"90":1055,"95":1055,"99":1055,"99.5":1055,"100":1055},"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"StudentServiceDelegate"}
data:
{"type":"HystrixThreadPool","name":"StudentServiceDelegate","currentTime":1500489058032,"currentActiveCount":0,"currentCompletedTaskCount":0,"currentCorePoolSize":10,"currentLargestPoolSize":6,"currentMaximumPoolSize":10,"currentPoolSize":6,"currentQueueSize":0,"currentTaskCount":6,"rollingCountThreadsExecuted":4,"rollingMaxActiveThreads":1,"rollingCountCommandRejectionSize":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}
ping:
data:
{"type":"HystrixCommand","name":"callStudentServiceAndGetData","group":"StudentServiceDelegate","currentTime":1500489058532,"isCircuitBreakerOpen":false,"errorPercentage":0,"errorCount":0,"requestCount":4,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCountEmit":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rollingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"rollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":4,"rollingCountThreadPoolRejected":0,"rollingCountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":1026,"latencyExecute":{"0":1002,"25":1002,"50":1002,"75":1051,"90":1051,"95":1051,"99":1051,"99.5":1051,"100":1051},"latencyTotal_mean":1029,"latencyTotal":{"0":1003,"25":1003,"50":1003,"75":1055,"90":1055,"95":1055,"99":1055,"99.5":1055,"100":1055},"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"propertyValue_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"StudentServiceDelegate"}
data:
{"type":"HystrixThreadPool","name":"StudentServiceDelegate","currentTime":1500489058532,"currentActiveCount":0,"currentCompletedTaskCount":6,"currentCorePoolSize":10,"currentLargestPoolSize":6,"currentMaximumPoolSize":10,"currentPoolSize":6,"currentQueueSize":0,"currentTaskCount":6,"rollingCountThreadsExecuted":4,"rollingMaxActiveThreads":1,"rollingCountCommandRejectionSize":0,"propertyValue_queueSizeRejectionThreshold":5,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"reportingHosts":1}
```

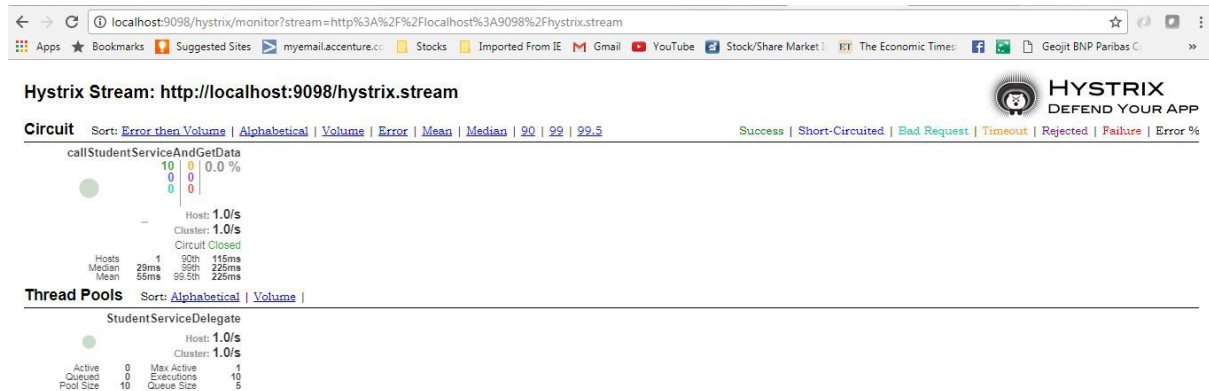
Hystrix Stream output

<http://localhost:9098/hystrix> - Il s'agit de l'état initial du tableau de bord visuel.



Hystrix Initial Dashboard

Ajoutez maintenant <http://localhost:9098/hystrix.stream> dans le tableau de bord pour obtenir une représentation visuelle dynamique significative du circuit surveillé par le composant Hystrix. Tableau de bord visuel après avoir fourni l'entrée Stream dans la page d'accueil -



Hystrix visual Dashboard

Spring Cloud - Zipkin and Sleuth

[Zipkin](#) est un outil très efficace pour le traçage distribué dans l'écosystème de microservices. Le traçage distribué, en général, est la mesure de la latence de chaque composant dans une transaction distribuée où plusieurs microservices sont appelés pour servir un seul cas d'utilisation métier. Disons que depuis notre application, nous devons appeler 4 services / composants différents pour une transaction. Ici, avec le traçage distribué activé, nous pouvons mesurer quel composant a pris combien de temps.

Ceci est utile lors du débogage lorsque de nombreux systèmes sous-jacents sont impliqués et que l'application devient lente dans une situation particulière. Dans ce cas, nous devons d'abord identifier le service sous-jacent qui est réellement lent. Une fois le service lent identifié, nous pouvons travailler pour résoudre ce problème. Le traçage distribué aide à identifier ce composant lent dans l'écosystème.

Zipkin

Zipkin a été développé à l'origine sur Twitter, sur la base du concept d'un article de Google décrivant le débogueur d'applications distribuées de Google, [dapper](#). Il gère à la fois la collecte et la recherche de ces données. Pour utiliser Zipkin, les applications sont instrumentées pour lui rapporter des données de synchronisation.

Si vous résolvez des problèmes de latence ou des erreurs dans l'écosystème, vous pouvez filtrer ou trier toutes les traces en fonction de l'application, de la longueur de la trace, de

l'annotation ou de l'horodatage. En analysant ces traces, vous pouvez décider quels composants ne fonctionnent pas conformément aux attentes et vous pouvez les corriger.

En interne, il dispose de 4 modules -

Collector- Une fois qu'un composant envoie les données de trace arrivent au démon collecteur Zipkin, il est validé, stocké et indexé pour les recherches par le collecteur Zipkin.

Storage- Ce module stocke et indexe les données de recherche dans le backend. Cassandra, Elasticsearch et MySQL sont pris en charge.

Search- Ce module fournit une API JSON simple pour rechercher et récupérer les traces stockées dans le backend. Le principal consommateur de cette API est l'interface utilisateur Web.

Web UI - Une très belle interface utilisateur pour visualiser les traces.

Comment installer Zipkin

Les étapes d'installation détaillées peuvent être trouvées pour différents systèmes d'exploitation, y compris l'image Docker sur [la page de démarrage rapide](#). Pour l'installation de Windows, téléchargez simplement le dernier serveur Zipkin à partir du [référentiel maven](#) et exécutez le fichier jar exécutable en utilisant la commande ci-dessous.

```
java -jar zipkin-server-1.30.3-exec.jar
```

Une fois Zipkin démarré, nous pouvons voir l'interface utilisateur Web à l'adresse <http://localhost:9411/zipkin/> .

La commande ci-dessus démarrera le serveur Zipkin avec la configuration par défaut. Pour une configuration avancée, nous pouvons configurer de nombreuses autres choses comme le stockage, les écouteurs de collecteur, etc.

Pour installer Zipkin dans l'application Spring Boot, nous devons ajouter la dépendance Zipkin Starter dans le projet Spring Boot.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

Sleuth

Sleuth est un outil de la famille Spring Cloud. Il est utilisé pour générer l'ID de trace, l'ID de portée et ajouter ces informations aux appels de service dans les en-têtes et MDC, afin

qu'il puisse être utilisé par des outils tels que Zipkin et ELK, etc. pour stocker, indexer et traiter les fichiers logs. Comme il appartient à la famille Spring Cloud, une fois ajouté au CLASSPATH, il s'intègre automatiquement aux canaux de communication courants tels que

-

- demandes effectuées avec le RestTemplate, etc.
- demandes qui passent par un microproxy Netflix Zuul
- En-têtes HTTP reçus sur les contrôleurs Spring MVC
- demandes sur des technologies de messagerie comme Apache Kafka ou RabbitMQ, etc.

Utiliser Sleuth est très simple. Nous devons juste ajouter le pom démarré dans le projet Spring Boot. Il ajoutera le Sleuth au projet et donc dans son exécution.

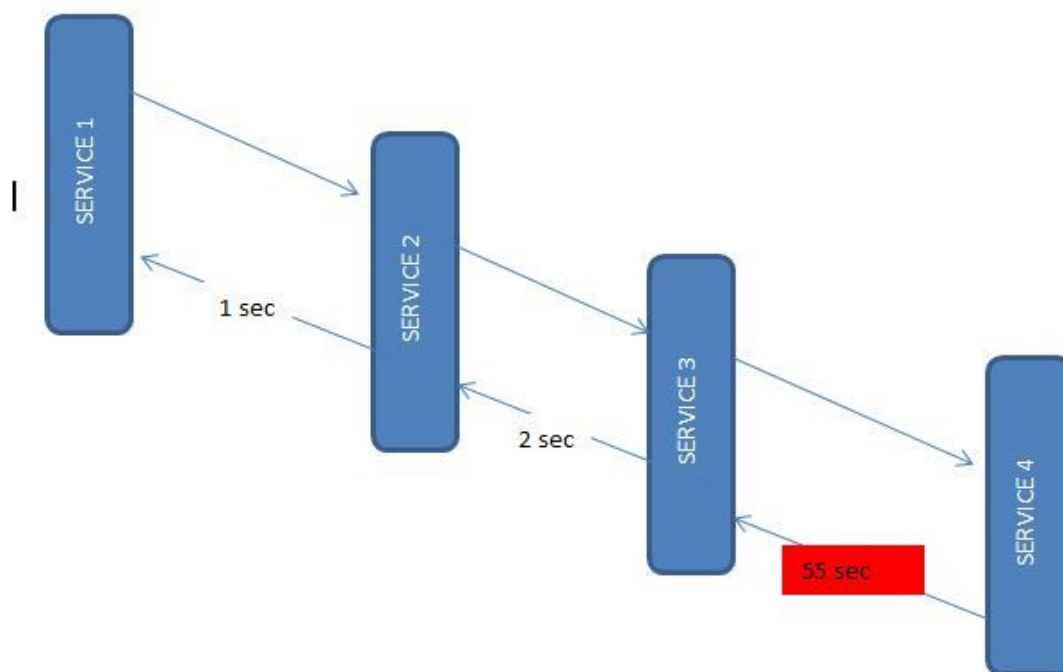
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Jusqu'à présent, nous avons intégré Zipkin et Sleuth aux microservices et exécuté le serveur Zipkin. Voyons comment utiliser cette configuration.

Exemple d'intégration Zipkin et Sleuth

Pour cette démo, créons 4 microservices basés sur Spring Boot. Ils auront tous à la fois des dépendances de démarrage Zipkin et Sleuth. Dans chaque microservice, nous exposerons un endpoint et à partir du premier service, nous appellerons le deuxième service, et à partir du deuxième service, nous invoquerons le troisième et ainsi de suite en utilisant le modèle de repos.

Et comme nous l'avons déjà mentionné, Sleuth fonctionne automatiquement avec le modèle de repos afin d'envoyer ces informations d'appel de service instrumentées au serveur Zipkin attaché. Zipkin commencera alors la comptabilité du calcul de la latence avec quelques autres statistiques comme les détails des appels de service.



Microservices Interactions

Créer un microservice

Les quatre services auront la même configuration, la seule différence réside dans les détails d'appel du service là où le endpoint change. Créons des applications de démarrage Spring avec les dépendances Web, Rest Repository, Zipkin et Sleuth.

J'ai regroupé ces services dans un projet parent afin que ces quatre services puissent être construits ensemble pour gagner du temps. Vous pouvez procéder à une configuration individuelle si vous le souhaitez. J'ai également ajouté des scripts Windows utiles pour démarrer / arrêter tous les services avec une seule commande.

Il s'agit d'un exemple de contrôleur de repos qui expose un endpoint et appelle également un service down stream à l'aide d'un modèle de repos. (ZipkinServiceApplication) (Cloner le code)

Configurations d'application

Comme tous les services fonctionneront sur une seule machine, nous devons les exécuter dans différents ports. Aussi pour nous identifier dans Zipkin, nous devons donner des noms propres. Configurez donc le nom de l'application et les informations de port dans le fichier `application.properties` sous le dossier resources.

```
server.port = 8081
spring.application.name = zipkin-server1
```

De même pour les 3 autres services, nous utiliserons les ports 8082, 8083, 8084 et le nom sera également comme zipkin-server2, zipkin-server3 et zipkin-server4.

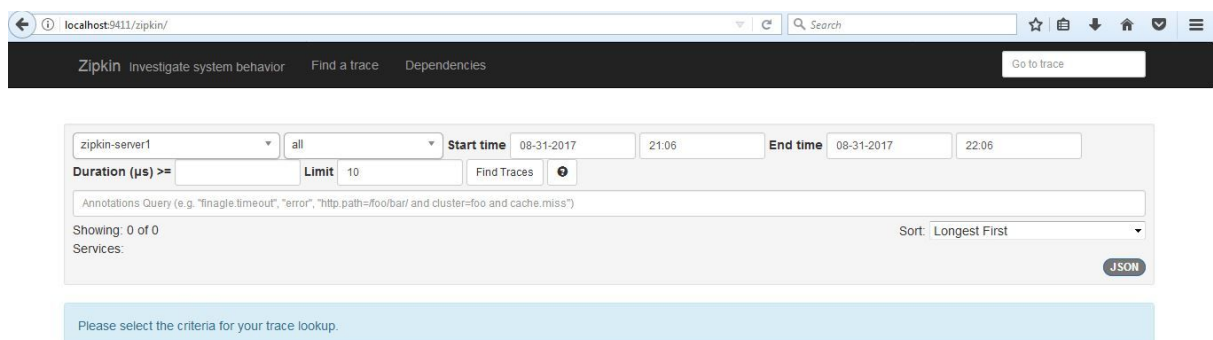
Nous avons également intentionnellement introduit un retard dans le dernier service afin que nous puissions le voir dans Zipkin.

Demo

Faites une version finale de maven à l'aide de la commande `mvn clean install` dans les microservices, démarrez les 4 applications avec le serveur zipkin.

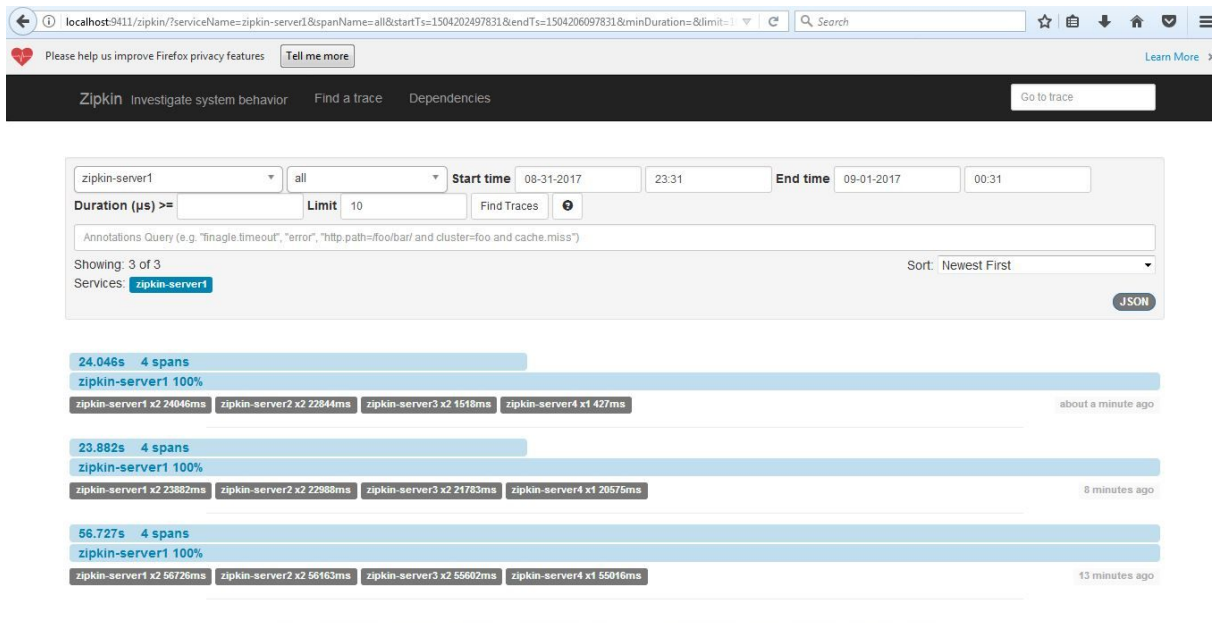
Maintenant, testez le premier endpoint de service deux fois à partir du navigateur - <http://localhost:8081/zipkin> . Veuillez noter qu'il y a un retard intentionnel dans l'un des 4 services ci-dessus. Il y aura donc un retard dans la réponse finale attendue, n'abandonnez pas.

Une fois l'appel d'API réussi, nous pouvons voir les statistiques de latence sur l'interface utilisateur zipkin <http://localhost:9411/zipkin/> . Choisissez le premier service dans la première liste déroulante et cliquez une fois sur le bouton Rechercher des traces.

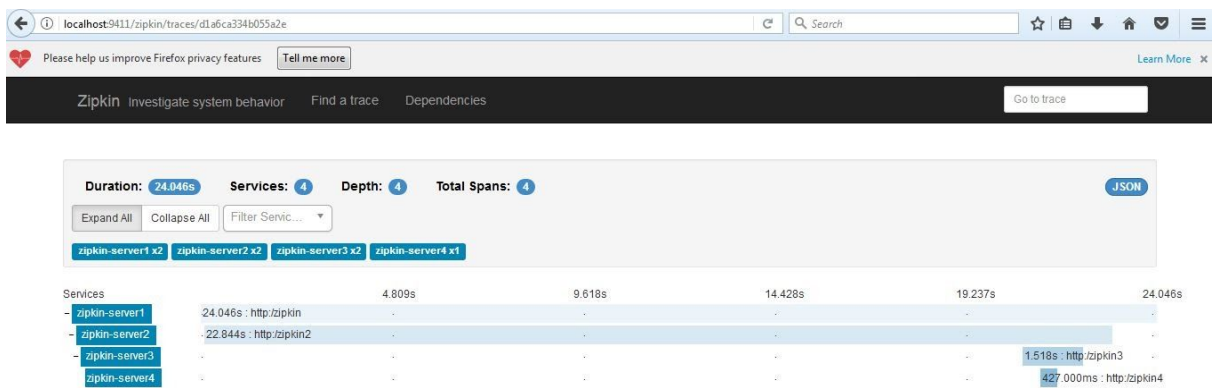


Zipkin Home screen

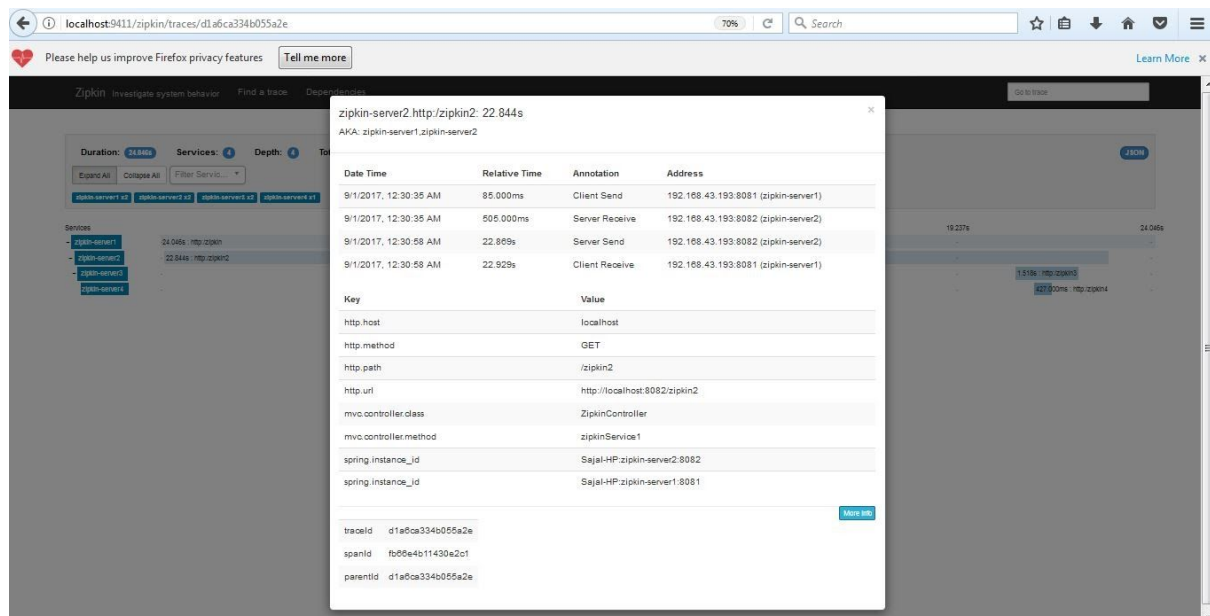
Vous devriez voir ce type d'interface utilisateur où vous pouvez effectuer une analyse des performances en examinant les données de suivi.



Find Traces UI



One particular transaction overview



Details of a particular service call statistics

Spring Security

[Spring Security](#) est un cadre qui se concentre sur la fourniture à la fois d'authentification et d'autorisation aux applications logicielles d'entreprise basées sur Java EE.

Maven Dependency

Pour inclure Spring security dans votre projet basé sur maven, incluez la dépendance ci-dessous:

```
<repositories>
  <repository>
    <id>spring-snapshot</id>
    <name>Spring Snapshot Repository</name>
    <url>http://repo.spring.io/snapshot</url>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>4.1.1.RELEASE</version>
  </dependency>
</dependencies>
```

```
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-config</artifactId>
<version>4.1.1.RELEASE</version>
</dependency>
</dependencies>
```

Si vous utilisez des fonctionnalités supplémentaires telles que LDAP, OpenID, etc., vous devrez également inclure les modules appropriés.

Si vous rencontrez un problème de dépendance transitive provoquant des problèmes de chemin de classe au moment de l'exécution, vous pouvez envisager d'ajouter un fichier de BOM de Spring Security.

```
<dependencies>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-framework-bom</artifactId>
  <version>4.3.1.RELEASE</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
</dependencies>
```

Gradle Dependency

Pour inclure Spring security dans votre projet basé sur Gradle, incluez la dépendance ci-dessous:

```
repositories {
  mavenCentral()
}
dependencies {
  compile 'org.springframework.security:spring-security-web:4.1.1.RELEASE'
  compile 'org.springframework.security:spring-security-config:4.1.1.RELEASE'
}
```

Référence(s):

[Spring Security Reference](#)

Spring Security Auth UT

Apprenez à tester l'authentification de sécurité Spring à l'aide du scénario de test JUnit à l'aide de In Memory DaoImpl. Apprenez également à créer un objet d'authentification entièrement rempli par programme, puis à l'utiliser dans l'application.

SecurityContextHolder

La sécurité Spring est basée sur le contexte de sécurité, qui est de nature statique. Cela signifie essentiellement que vous n'avez pas besoin d'injecter sa référence dans vos beans ou classes dans spring container. Vous pouvez accéder au contexte de spring à tout moment simplement en utilisant la méthode `SecurityContextHolder.getContext()`.

Ce contexte a la référence du principal ou de l'utilisateur réel que nous devons valider pour ses autorisations d'accès.

Unit test Spring Security

Je crée un projet maven très simple et j'écrirai un code minimal afin que je puisse me concentrer sur le test uniquement de ce qui est dans la portée de ce post, à savoir l'authentification. Ensuite, j'écrirai une classe de service de démonstration avec une seule méthode qui nécessitait «ROLE_USER» pour y accéder. Si vous essayez d'accéder à cette méthode et que vous n'avez pas «ROLE_USER», vous obtiendrez l'exception `AccessDeniedException` attendue. Assez simple, non?

Step 1) Project setup

clone kata-spring-security project from my github
explore the pom.xml

Step 2) Create security configuration file

J'ai créé le fichier `application-security.xml` et mis la configuration de sécurité à l'intérieur.

Step 3) Create secured method

```
package com.howtodoinjava;

import org.springframework.security.access.annotation.Secured;

public class DemoService
{
    @Secured("ROLE_USER")
    public void method()
    {
        System.out.println("Method called");
    }
}
```

Step 4) Test the authentication with JUnit test

Dans les tests junit, nous allons configurer le contexte de spring par programme, puis accéder aux utilisateurs par nom d'utilisateur à partir du service de détails de l'utilisateur par défaut. Dans tous les cas, il s'agit d'une implémentation en mémoire qui, dans votre cas, peut différer d'un service de détails utilisateur basé sur jdbc ou d'un autre service de détails utilisateur personnalisé également. Veuillez donc modifier la recherche en conséquence.

Nous testerons divers scénarios comme un utilisateur valide, un utilisateur invalide, un rôle invalide, etc. Vous pouvez ajouter / supprimer des scénarios en fonction de votre choix.

JUnit 5 Tutorial

Ce didacticiel JUnit 5 explique comment il a adapté le style de codage Java 8 et plusieurs autres fonctionnalités. Découvrez en quoi il est différent de JUnit 3 ou 4.

JUnit 5 est le framework de test le plus largement utilisé pour les applications Java. Depuis très longtemps, JUnit fait parfaitement son travail. Entre les deux, JDK 8 a apporté des fonctionnalités très intéressantes dans java et plus particulièrement les expressions lambda. JUnit 5 vise à adapter le style de codage java 8 et plusieurs autres fonctionnalités, c'est pourquoi java 8 est nécessaire pour créer et exécuter des tests dans JUnit 5 (bien qu'il soit possible d'exécuter des tests écrits avec JUnit 3 ou JUnit 4 pour la compatibilité descendante) .

JUnit 5 Architecture

Par rapport à JUnit 4, JUnit 5 est composé de plusieurs modules différents issus de trois sous-projets différents:

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

1. JUnit Platform

Pour pouvoir lancer des tests junit, les IDE, les outils de construction ou les plugins doivent inclure et étendre les API de la plateforme. Il définit l'API TestEngine pour développer de nouveaux frameworks de test qui s'exécutent sur la plate-forme. Il fournit également un lanceur de console pour lancer la plate-forme à partir de la ligne de commande et créer des plugins pour Gradle et Maven.

2. JUnit Jupiter

Il inclut de nouveaux modèles de programmation et d'extension pour l'écriture de tests. Il a toutes les nouvelles annotations junit et l'implémentation TestEngine pour exécuter des tests écrits avec ces annotations.

3. JUnit Vintage

Son objectif principal est de prendre en charge l'exécution des tests écrits JUnit 3 et JUnit 4 sur la plate-forme JUnit 5. C'est une rétrocompatibilité.

Installation

Vous pouvez utiliser JUnit 5 dans votre projet maven ou gradle en incluant au minimum deux dépendances, à savoir la dépendance de Jupiter Engine et la dépendance de Platform Runner.

```
//pom.xml

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>

  <maven.compiler.target>${maven.compiler.source}</maven.compiler.target>
  <junit.jupiter.version>5.5.2</junit.jupiter.version>
  <junit.platform.version>1.5.2</junit.platform.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>${junit.platform.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>

//build.gradle

testRuntime("org.junit.jupiter:junit-jupiter-engine:5.5.2")
testRuntime("org.junit.platform:junit-platform-runner:1.5.2")
```

JUnit 5 Annotations

JUnit 5 propose les annotations suivantes pour écrire des tests.

Annotation	Description
@BeforeEach	La méthode annotée sera exécutée avant chaque méthode de test de la classe de test.
@AfterEach	La méthode annotée sera exécutée après chaque méthode de test dans la classe de test.
@BeforeAll	La méthode annotée sera exécutée avant toutes les méthodes de test de la classe de test. Cette méthode doit être statique.
@AfterAll	La méthode annotée sera exécutée après toutes les méthodes de test de la classe de test. Cette méthode doit être statique.
@Test	Il est utilisé pour marquer une méthode comme test junit
@DisplayName	Utilisé pour fournir un nom d'affichage personnalisé pour une classe de test ou une méthode de test
@Disable	Il est utilisé pour désactiver ou ignorer une classe ou une méthode de test de la suite de tests.
@Nested	Utilisé pour créer des classes de test imbriquées
@Tag	Mark teste des méthodes ou des classes de test avec des balises pour tester la découverte et le filtrage
@TestFactory	Mark a method est une usine de test pour les tests dynamiques

Writing Tests in JUnit 5

Il n'y a pas beaucoup de changement entre JUnit 4 et JUnit 5 dans les styles d'écriture de test. Voici des exemples de tests avec leurs méthodes de cycle de vie.

```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

import com.howtodoinjava.junit5.examples.Calculator;

public class AppTest {
```

```

@BeforeAll
static void setup(){
    System.out.println("@BeforeAll executed");
}

@BeforeEach
void setupThis(){
    System.out.println("@BeforeEach executed");
}

@Tag("DEV")
@Test
void testCalcOne()
{
    System.out.println("=====TEST ONE EXECUTED=====");
    Assertions.assertEquals( 4 , Calculator.add(2, 2));
}

@Tag("PROD")
@Disabled
@Test
void testCalcTwo()
{
    System.out.println("=====TEST TWO EXECUTED=====");
    Assertions.assertEquals( 6 , Calculator.add(2, 4));
}

@AfterEach
void tearThis(){
    System.out.println("@AfterEach executed");
}

@AfterAll
static void tear(){
    System.out.println("@AfterAll executed");
}
}

```

Test Suites

En utilisant les suites de tests JUnit 5, vous pouvez exécuter des tests répartis dans plusieurs classes de test et différents packages. JUnit 5 fournit deux annotations: `@SelectPackages` et `@SelectClasses` pour créer des suites de tests.

Pour exécuter la suite, vous utiliserez `@RunWith (JUnitPlatform.class)`.

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.howtodoinjava.junit5.examples")
public class JUnit5TestSuiteExample
{
}
```

De plus, vous pouvez utiliser les annotations suivantes pour filtrer les packages de test, les classes ou même les méthodes de test.

- `@IncludePackages` et `@ExcludePackages` pour filtrer les packages
- `@IncludeClassNamePatterns` et `@ExcludeClassNamePatterns` pour filtrer les classes de test
- `@IncludeTags` et `@ExcludeTags` pour filtrer les méthodes de test

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.howtodoinjava.junit5.examples")
@IncludePackages("com.howtodoinjava.junit5.examples.packageC")
@ExcludeTags("PROD")
public class JUnit5TestSuiteExample
{
}
```

Assertions

Les assertions aident à valider la sortie attendue avec la sortie réelle d'un cas de test. Pour simplifier les choses, toutes les assertions JUnit Jupiter sont des méthodes statiques de la classe `org.junit.jupiter.Assertions`, par exemple `assertEquals()`, `assertNotEquals()`.

```
void testCase()
{
    //Test will pass
    Assertions.assertNotEquals(3, Calculator.add(2, 2));

    //Test will fail
    Assertions.assertNotEquals(4, Calculator.add(2, 2),
    "Calculator.add(2, 2) test failed");

    //Test will fail
    Supplier<String> messageSupplier = ()-> "Calculator.add(2, 2) test failed";
}
```



```
Assertions.assertNotEquals(4, Calculator.add(2, 2),  
messageSupplier);  
}
```

Assumptions

La classe Assumptions fournit des méthodes statiques pour prendre en charge l'exécution de tests conditionnels basés sur des hypothèses. Une hypothèse échouée entraîne l'arrêt d'un test. Les hypothèses sont généralement utilisées chaque fois que la poursuite de l'exécution d'une méthode de test donnée n'a pas de sens. Dans le rapport de test, ces tests seront marqués comme réussis.

La classe JUnit Jupiter Assumptions a deux méthodes de ce type: `assumeFalse()`, `assumeTrue()`.

```
public class AppTest {  
    @Test  
    void testOnDev()  
    {  
        System.setProperty("ENV", "DEV");  
        Assumptions.assumeTrue("DEV".equals(System.getProperty("ENV")),  
AppTest::message);  
    }  
  
    @Test  
    void testOnProd()  
    {  
        System.setProperty("ENV", "PROD");  
  
        Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));  
    }  
  
    private static String message () {  
        return "TEST Execution Failed :: ";  
    }  
}
```

Backward Compatibility for JUnit 3 or JUnit 4

JUnit 4 existe depuis assez longtemps et il existe de nombreux tests écrits en jUnit 4. JUnit Jupiter doit également prendre en charge ces tests. Pour cela, un sous-projet JUnit Vintage est développé.

JUnit Vintage fournit une implémentation TestEngine pour exécuter des tests basés sur JUnit 3 et JUnit 4 sur la plate-forme JUnit 5.