Luca Cabibbo

Architettura dei Sistemi Software

Spring Cloud

dispensa asw850

marzo 2021

There are no rules of architecture for a castle in the clouds.

Gilbert Keith Chesterton

- Fonti

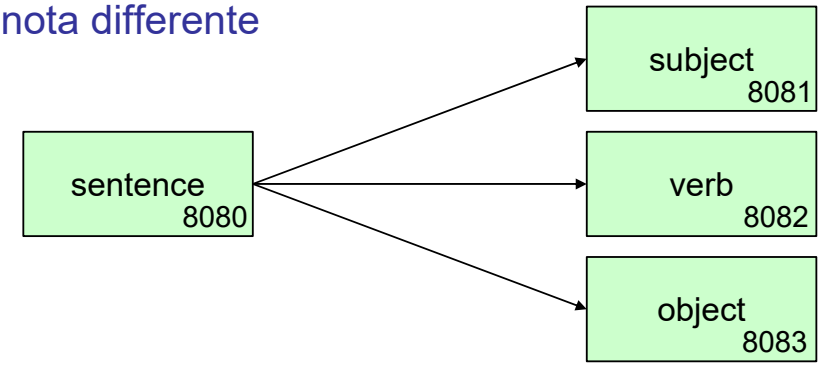
- Spring Cloud
  - https://spring.io/projects/spring-cloud
- **Spring Cloud Documentation**, 2020.0.2, 2021.
  - https://spring.io/projects/spring-cloud#learn
- Progetti Spring Cloud di interesse
  - Spring Cloud Config
  - Spring Cloud Bus
  - Spring Cloud Consul Discovery
  - Spring Cloud LoadBalancer
  - Spring Cloud OpenFeign
  - Spring Cloud Circuit Breaker
  - Spring Cloud Gateway
- Netflix Open Source Software
  - https://netflix.github.io/

- Obiettivi e argomenti

- Obiettivi
  - fornire un'introduzione a Spring Cloud
  - presentare alcuni sottoprogetti di Spring Cloud
- Argomenti
  - un'applicazione composta da più servizi
  - introduzione a Spring Cloud
  - gestione delle configurazioni con Spring Cloud Config
  - service discovery con Spring Cloud Consul Discovery
  - client-side load balancing con Spring Cloud LoadBalancer
  - client REST dichiarativi con Spring Cloud OpenFeign
  - circuit breaker con Spring Cloud Circuit Breaker
  - API gateway con Spring Cloud Gateway
  - invocazioni remote asincrone
  - discussione

* Un'applicazione composta da più servizi

- Come applicazione di esempio, consideriamo ora un'applicazione composta da più servizi (componenti applicativi) distribuiti (ovvero, da eseguire in processi o nodi distinti)
  - usiamo un servizio principale (sentence) per generare frasi in modo casuale – ogni frase è composta da soggetto, verbo e complemento oggetto – usiamo anche degli ulteriori servizi per generare parole di tipo diverso (subject, verb e object)
  - inizialmente, questi servizi vengono mandati in esecuzione in processi distinti su uno stesso nodo, e ciascuno collegato a una porta nota differente



Un'applicazione composta da più servizi

- L'applicazione è composta da un servizio principale (sentence) per le frasi, insieme a degli ulteriori servizi per generare le parole che compongono le frasi (subject, verb e object)
  - il servizio principale per le frasi (sentence) è un'applicazione Spring Boot – in esecuzione sulla porta 8080
  - per la generazione dei diversi tipi di parole, usiamo un altro servizio (word) che restituisce delle parole casuali – un'altra applicazione Spring Boot, con tre profili, uno per ciascun tipo di parole (subject, verb e object)
    - usiamo tre istanze di questo servizio (una per profilo), in esecuzione rispettivamente sulle porte 8081, 8082 e 8083
  - il servizio sentence dipende dai servizi subject, verb e object
    - questi servizi comunicano tra di loro tramite REST

Il servizio per le parole

- Il servizio per le parole

```
package asw.sentence.wordservice.domain;

import ...

@Service
public class WordService {

   @Value("${asw.sentence.wordservice.words}")
   private String words;

   private Random random = new Random();

   public String getWord() {
      String[] wordArray = words.split(",");
      int randomIndex = random.nextInt(wordArray.length);
      return wordArray[randomIndex];
   }

}
```

```
# application.yml
asw.sentence:
  wordservice:
    words: Ann,Mary,John,My dog
```

L'adattatore REST per le parole

- L'adattatore REST per il servizio per le parole

```
package asw.sentence.wordservice.rest;

import asw.sentence.wordservice.domain.WordService;

import ...

@RestController
public class WordController {

    @Autowired
    private WordService wordService;

    @GetMapping("/")
    public String getWord() {
        return wordService.getWord();
    }

}
```


Il file application.yml per le parole

- Questo è il file application.yml per il servizio per le parole, che definisce tre profili per i diversi tipi di parole
  - ogni profilo specifica un insieme di parole e una porta specifica per il servizio

```
# application.yml
---
spring:
  config.activate.on-profile: subject
  application:
    name: subject

asw.sentence:
  wordservice:
    words: Ann,Mary,John,My dog

server:
  port: 8081
```

```
---
spring:
  config.activate.on-profile: verb
  application:
    name: verb

asw.sentence:
  wordservice:
    words: likes,loves,dreams,does not like

server:
  port: 8082

---
spring:
  config.activate.on-profile: object
  application:
    name: object

asw.sentence:
  wordservice:
    words: cats,pizza,spring,my cat

server:
  port: 8083
```

Il servizio per le frasi

- Il servizio per la generazione delle frasi

```
package asw.sentence.sentenceservice.domain;

import ...

@Service
public class SentenceService {

    @Autowired
    private WordService subjectService;
    @Autowired
    private WordService verbService;
    @Autowired
    private WordService objectService;

    public String getSentence() {

        return subjectService.getWord() + " " +
               verbService.getWord() + " " +
               objectService.getWord() + ".";

    }

}
```


L'interfaccia per accedere al servizio delle parole

- La porta (interfaccia) per accedere ai servizi per le parole

```
package asw.sentence.sentenceservice.domain;

public interface WordService {

    public String getWord();

}
```

L'adattatore REST per accedere al servizio delle parole

- L'adattatore REST per accedere al servizio subject delle parole – quelli per verb e object sono simili

```
package asw.sentence.sentenceservice.wordservice;

import asw.sentenceservice.sentence.domain.WordService;

import ...

@Service
public class SubjectService implements WordService {

   @Value("${asw.sentence.sentenceservice.subject.uri}")
   private String subjectUri;

   @Autowired
   private WordClient wordClient;

   public String getWord() {
      return wordClient.getWord(subjectUri);
   }

}
```

```
# application.yml
asw.sentence:
  sentenceservice:
    subject.uri=http://localhost:8081
```


L'adattatore REST per accedere al servizio delle parole

- L'interfaccia WordClient per effettuare le chiamate REST

```
package asw.sentence.sentenceservice.wordservice;

public interface WordClient {

   public String getWord(String wordUri);

}
```

L'adattatore REST per accedere al servizio delle parole con RestTemplate

- Il componente per effettuare le chiamate REST
  - usando RestTemplate

```
package asw.sentenceservice.sentence.wordservice;

import ...

@Service
public class WordClientRestTemplate implements WordClient {

   @Autowired private RestTemplate restTemplate;

   public String getWord(String wordUri) {

      String word = null;
      try {
           word = restTemplate.getForObject(wordUri, String.class);
      } catch (RestClientException e) { ... eccezione remota ... }
      return word;

   }

}
```


L'adattatore REST per accedere al servizio delle parole con RestTemplate

- Il componente per effettuare le chiamate REST
  - configurazione per RestTemplate

```
package asw.sentence.sentenceservice.wordservice.config;

import ...

@Configuration
public class RestTemplateConfig {

   @Bean
   public RestTemplate restTemplate() {
      return new RestTemplate();
   }

}
```

L'adattatore REST per accedere al servizio delle parole con WebClient

- Il componente per effettuare le chiamate REST
  - usando WebClient

```
package asw.sentenceservice.sentence.wordservice;

import ...

@Service
public class WordClientWebClient implements WordClient {

    @Autowired private WebClient webClient;

    public String getWord(String wordUri) {

        String word = null;
        Mono<String> response = webClient
                  .get().uri(wordUri).retrieve().bodyToMono(String.class);
        try {
             word = response.block();
        } catch (WebClientException e) { ... eccezione remota ... }
        return word;

    }
}
```


L'adattatore REST per accedere al servizio delle parole con WebClient

- Il componente per effettuare le chiamate REST
  - configurazione per WebClient

```
package asw.sentence.sentenceservice.wordservice.config;

import ...

@Configuration
public class WebClientConfig {

    @Bean
    public WebClient webClient() {
        return WebClient.builder().build();
    }

}
```

L'adattatore REST per le frasi

- L'adattatore REST per il servizio per le frasi

```
package asw.sentence.sentenceservice.rest;

import asw.sentence.sentenceservice.domain.SentenceService;

import ...

@RestController
public class SentenceController {

    @Autowired
    private SentenceService sentenceService;

    @GetMapping("/")
    public String getSentence() {

        return sentenceService.getSentence();

    }

}
```


Il file application.yml per le frasi

- Il file application.yml per il servizio per le frasi
  - in esecuzione sulla porta 8080
  - specifica anche la locazione dei servizi per le parole (definita staticamente)

```
# application.yml
---
spring:
  application:
    name: sentence

server:
  port: 8080

asw.sentence:
  sentenceservice:
    subject.uri: http://localhost:8081
    verb.uri: http://localhost:8082
    object.uri: http://localhost:8083
```

Discussione

- Abbiamo mostrato una semplice applicazione distribuita composta da più servizi
  - questa applicazione può essere mandata in esecuzione
    - avviando tre istanze del servizio delle parole, usando per ciascuna un profilo diverso, per i tre tipi di parole
    - avviando un'istanza del servizio per le frasi
  - il servizio per le frasi può essere acceduto con l'operazione GET http://localhost:8080
    - che potrebbe restituire, ad es., frasi come Mary loves cats. oppure My dog does not like pizza.
  - l'applicazione può essere anche distribuita su più nodi
    - in questo caso è necessario modificare il file di configurazione del servizio delle frasi, per specificare la locazione dei servizi per i diversi tipi di parole

Discussione

- Questa applicazione ha anche numerose limitazioni – eccone alcune
  - come fare se voglio cambiare dinamicamente la configurazione dell'applicazione?
    - ad es., come fare a cambiare le parole usate dai generatori di parole casuali?
  - come fare se voglio avviare molte istanze per ciascuno dei servizi?
    - a che porte vanno collegati i servizi?
    - come fanno a comunicare i servizi fra di loro?
    - come fare bilanciamento del carico tra le diverse istanze dei servizi?
  - come fare se qualche istanza va in crash?
    - come tollerare guasti di questo tipo, o comunque sostenere un isolamento dei guasti tra servizi?

* Introduzione a Spring Cloud

- *Spring Cloud* è un progetto Spring, basato su Spring Boot, che fornisce agli sviluppatori degli strumenti per affrontare alcune problematiche comuni nello sviluppo delle applicazioni distribuite (in particolare, applicazioni a microservizi), sulla base di un insieme di pattern comuni
  - i servizi e le applicazioni Spring Cloud possono essere rilasciati in ogni ambiente distribuito – ambienti fisici e virtuali, sul proprio personal computer, ma anche e soprattutto nel cloud
  - così come Spring è un progetto "ombrello" composto da altri progetti – tra cui Spring Framework, Spring Boot e Spring Cloud – anche Spring Cloud è a sua volta composto da altri progetti – tra cui
    - Spring Cloud Commons, Spring Cloud Config, Spring Cloud Consul, Spring Cloud LoadBalancer, Spring Cloud OpenFeign, Spring Cloud Circuit Breaker, Spring Cloud Gateway, Spring Cloud Netflix, …

Spring Cloud

- In pratica, Spring Cloud fornisce delle librerie per applicare alcuni pattern comuni utili nello sviluppo di applicazioni distribuite e a microservizi – tra cui
  - gestione delle configurazioni (centralizzata, con controllo delle versioni)
  - registrazione e discovery di servizi
  - invocazione di servizi mediante client dichiarativi
  - bilanciamento del carico
  - circuit breaker
  - API gateway e routing
  - …

Spring Cloud Netflix

- Alcuni sottoprogetti di Spring Cloud hanno lo scopo di integrare in Spring alcune librerie, tecnologie e piattaforme per il cloud di terze parti – per consentirne un accesso unificato e semplificato
  - tra questi, il progetto *Spring Cloud Netflix* si occupa dell'integrazione di Spring Cloud con Netflix OSS
  - *Netflix OSS* (*Netflix Open Source Software*) è un insieme di librerie per il cloud che Netflix ha sviluppato (e usato in produzione) e poi ha reso pubbliche come progetti open source
    - Eureka, Hystrix, Ribbon, Zuul, …
  - questo progetto ha svolto un ruolo importante in Spring Cloud
    - tuttavia, Netflix ha smesso di sviluppare attivamente alcune di queste librerie – anche per questo, Spring Cloud Netflix è attualmente in "maintenance mode" (ovvero, non vi verranno aggiunte nuove funzionalità)
    - inoltre, per le stesse funzionalità, Spring Cloud fornisce dei progetti propri (o si integra con altre librerie)

Osservazioni preliminari

- Alcune osservazioni preliminari su Spring Cloud
  - i progetti Spring Cloud sono tutti basati su Spring Boot – che semplifica lo sviluppo e la gestione delle applicazioni Spring
    - Spring Cloud è però basato su un processo di startup (per la definizione dell'application context) che è modificato rispetto a quello di Spring Boot
  - attenzione all'uso dei termini "client" e "server"
    - il loro significato è relativo, e viene usato per descrivere il ruolo svolto da due elementi nel contesto di una specifica relazione/interazione
    - ad es., in generale un microservizio è un server (dal punto di vista applicativo), ma potrebbe anche comportarsi da client nei confronti di altri servizi applicativi e di utilità (ad es., il servizio di service discovery)

Osservazioni preliminari

- Alcune osservazioni preliminari su Spring Cloud
  - nel file di build build.gradle per Gradle (e in modo analogo per Maven) vanno aggiunte le dipendenze starter per Spring Cloud, nonché alcune sezioni preliminari per la gestione delle dipendenze per Spring Cloud

```
buildscript { ... }
plugins { ... }
dependencyManagement { ... }

dependencies {
   implementation 'org.springframework.cloud:spring-cloud-starter'
   ...
}
```


Osservazioni preliminari

- Alcune ulteriori osservazioni preliminari
  - in questa dispensa, se un'applicazione è composta da più servizi (applicativi o di supporto), allora, per semplicità questi servizi sono pensati per essere eseguiti tutti su un singolo nodo (localhost), con il servizio principale esposto/pubblicato sulla porta 8080
    - questa semplificazione verrà rimossa in una successiva dispensa (sulla composizione e l'orchestrazione di container), in cui verrà discussa l'esecuzione di questi servizi in un ambiente veramente distribuito
  - inoltre, i servizi di supporto (come quello di configurazione o di service discovery) verranno esposti su porte note
  - invece, eventuali servizi applicativi secondari verranno collegati a porte casuali – e dunque non note a priori

* Gestione delle configurazioni con Spring Cloud Config

- Abbiamo già discusso il fatto che molte applicazioni e servizi richiedono delle configurazioni complesse – ad es., relative ai parametri di un'applicazione, all'accesso alle risorse (come l'accesso a una base di dati) e alla locazione di servizi e applicazioni
  - la *gestione delle configurazioni* è ancora più complessa nel caso di applicazioni e servizi distribuiti o nel cloud
  - in questo contesto, infatti, le soluzioni di base offerte dal framework Spring (basate soprattutto su file di configurazione locali) sono spesso inadeguate
  - piuttosto, è in genere opportuna una gestione delle configurazioni accessibile remotamente, ma centralizzata

Configurazione di applicazioni distribuite

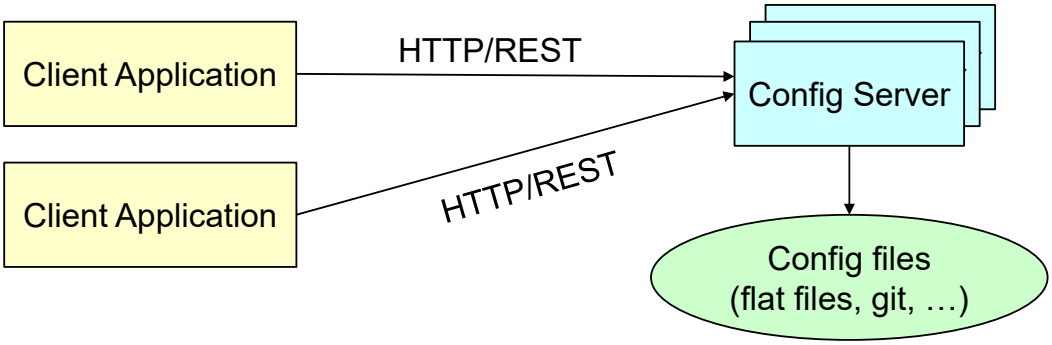
- Sono possibili diverse soluzioni per la configurazione di applicazioni e servizi distribuiti
  - configurazioni cablate nel codice o in file assemblati con le applicazioni e i servizi – è una soluzione rigida
  - file di configurazione in un file system comune – non sempre nel cloud è disponibile un file system
  - uso di variabili d'ambiente – talvolta difficili da gestire
  - uso di soluzioni specifiche per uno specifico ambiente cloud – l'accoppiamento con una soluzione proprietaria può essere indesiderato
  - ulteriori sfide nella gestione delle configurazioni
    - nelle applicazioni a microservizi ci possono essere dozzine o centinaia di servizi da configurare
    - potrebbero essere richiesti aggiornamenti dinamici delle configurazioni – nonché il loro controllo delle versioni

Spring Cloud e configurazioni

- Spring Cloud offre diverse soluzioni per la configurazione di applicazioni e servizi distribuiti – con queste caratteristiche
  - in modo indipendente dalla piattaforma cloud e dal linguaggio di programmazione
  - gestione centralizzata delle configurazioni – di solito è l'approccio più semplice ed efficace
  - configurazioni dinamiche – in cui è possibile aggiornare la configurazione di applicazioni e servizi in esecuzione
  - controllo delle configurazioni – consente di usare gli stessi strumenti di controllo delle versioni usati per il codice sorgente
  - scoperta passiva – le applicazioni e i servizi si registrano automaticamente, poi possono essere scoperte da altre applicazioni e servizi
  - alcune soluzioni sono implementate da Spring Cloud Config e Spring Cloud Bus

- Spring Cloud Config

- *Spring Cloud Config* fornisce un servizio di gestione centralizzata delle configurazioni di applicazioni e servizi distribuiti, di tipo client-server
  - con un server centralizzato per la gestione delle configurazioni – basato sulla dipendenza starter spring-cloud-config-server
  - i client – in aggiunta alle proprie informazioni di configurazione locali – possono ritrovare le proprie configurazioni accedendo al server delle configurazioni – questo accesso è basato sulla dipendenza starter spring-cloud-starter-config



- Spring Cloud Config Server

- Iniziamo mostrando l'implementazione di un server di configurazione con *Sprint Cloud Config Server*
  - un configuration server può essere implementato come un'applicazione Spring Boot minimale, usando la dipendenza spring-cloud-config-server e l'annotazione **@EnableConfigServer**
  - il file di configurazione application.properties o application.yml va usato per specificare la locazione delle informazioni di configurazione per i client (in un file locale al server oppure in un repository Git)
  - in questo esempio, il configuration server viene eseguito sulla porta 8888 – specificata mediante la proprietà server.port – che è quella usata convenzionalmente per questo servizio
    - i client del servizio di configurazione devono conoscere la locazione e la porta del configuration server

Spring Cloud Config Server

- Questa è la classe principale per il server di configurazione

```
package asw.sentence.configserver;

import ...

@SpringBootApplication
@EnableConfigServer
public class CommonConfigServer {

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

}
```

Spring Cloud Config Server

- In questo esempio, le informazioni di configurazione delle applicazioni sono su un repository Git – come specificato nel file di configurazione del server di configurazione

```
# application.yml
---
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/aswroma3/asw
          searchPaths: projects/config-data

server:
  port: 8888
```

  - tuttavia, avviando l'applicazione con il profilo native, il configuration server usa i file di configurazione tra le proprie risorse locali

Spring Cloud Config Server

- Possibili file di configurazione per la nostra applicazione
  - i file subject.yml, verb.yml, object.yml sul repository Git – per i servizi per le parole

```
# subject.yml
---
spring.config.activate.on-profile: subject
asw.sentence.wordservice.words: Spring Cloud Config,Spring Cloud Config
```

```
# verb.yml
---
spring.config.activate.on-profile: verb
asw.sentence.wordservice.words: works,works
```

```
# object.yml
---
spring.config.activate.on-profile: object
asw.sentence.wordservice.words: from remote files,from remote files
```

Spring Cloud Config Server

- Possibile file di configurazione per la nostra applicazione
  - i file subject.yml, verb.yml, object.yml sul repository Git – per i servizi per le parole

```
# subject.yml
---
spring.config.activate.on-profile: subject
asw.sentence.wordservice.words: Spring Cloud Config,Spring Cloud Config
```

  - in generale, con Spring Boot, i file di configurazione seguono delle convenzioni basate
    - sul nome del profilo – quello selezionato all'avvio dell'applicazione è spring.config.activate.on-profile
    - sul nome dell'applicazione client (spring.application.name) – per il servizio delle parole, useremo il nome del profilo

Spring Cloud Config Server

- Questo è quanto basta per realizzare un configuration server con Spring Cloud Config
  - il configuration server va mandato in esecuzione
  - se ne possono mandare in esecuzione anche più istanze (per disponibilità), usando anche un servizio di bilanciamento del carico (descritto più avanti)

- Spring Cloud Config Client

- Mostriamo ora l'implementazione di un client per il servizio di configurazione – basato su *Sprint Cloud Config Client*
  - supponiamo di partire da un'applicazione Spring Boot, pensata inizialmente per trovare le proprie informazioni di configurazione nel file di configurazione locale application.properties o application.yml
  - l'applicazione va modificata per trovare queste informazioni tramite il servizio di configurazione remoto
    - va usata la dipendenza starter spring-cloud-starter-config – non è necessaria nessuna annotazione per abilitare il client
    - le informazioni per l'accesso al server di configurazione remoto (l'URI del servizio) vanno specificate nel file application.properties o application.yml
    - il codice dell'applicazione non deve essere modificato

Il file application.yml

- Consideriamo l'applicazione word per il servizio delle parole
  - il file application.yml – tra le risorse locali del servizio word

```
# application.yml
---
spring:
  config.import: optional:configserver:http://localhost:8888
  application.name: ${spring.config.activate.on-profile}

# parole di default (per tutti i profili)
asw.sentence.wordservice.words: default,default,default
```

  - la proprietà spring.config.import specifica la locazione del configuration server
  - settiamo lo spring.application.name in corrispondenza al profilo selezionato all'avvio – subject, verb o object
  - in virtù dello spring.application.name, sul configuration server verrà acceduto il file di configurazione opportuno (ad es., subject.yml)

Il file application.yml

- Consideriamo l'applicazione word per il servizio delle parole
  - il file application.yml – segue

```
---
spring:
  config.activate.on-profile: subject

server:
  port: 8081

---
spring:
  config.activate.on-profile: verb

server:
  port: 8082

---
spring:
  config.activate.on-profile: object

server:
  port: 8083
```


Processo di configurazione

- La configurazione di un'applicazione avviene in questo modo
  - quando viene avviata l'applicazione, bisogna specificare la proprietà spring.profiles.active per selezionare un profilo
  - all'avvio, l'applicazione legge dal file application.properties o application.yml la locazione del server di configurazione (nell'esempio, localhost:8888) e il nome dell'applicazione
  - sulla base di queste proprietà, l'applicazione richiede e ottiene automaticamente dal server di configurazione la propria configurazione (per il profilo specificato)
  - se però il server di configurazione non è raggiungibile, allora l'applicazione utilizza i dati di configurazione specificati localmente nel file application.properties o application.yml
    - questi valori locali sono utili per specificare dei valori di default per le proprietà – ma hanno precedenza minore di quelli sul server di configurazione (non li sovrascrivono)

Discussione

- Spring Cloud Config fornisce una soluzione (anche se parziale) al problema della gestione delle configurazioni di applicazioni e servizi distribuiti
  - è una soluzione centralizzata, compatibile con la configurazione basata su proprietà tipica del framework Spring
  - l'esempio ha mostrato una soluzione sostanzialmente statica – in pratica, la configurazione viene caricata all'avvio di un'applicazione, e normalmente non viene più aggiornata
  - con Spring Cloud Config e Spring Cloud Bus è anche possibile la gestione di configurazioni dinamiche – per aggiornare la configurazione delle applicazioni in esecuzione (discusso più avanti)
  - altri servizi di Spring Cloud supportano invece la scoperta dei servizi, che è complementare alla gestione delle configurazioni

- Configurazioni dinamiche con Spring Cloud Config

- Spring Cloud Config supporta anche la gestione di configurazioni dinamiche – per aggiornare la configurazione di un'applicazione in esecuzione
  - i componenti le cui configurazioni devono poter essere aggiornate vanno annotati con **@RefreshScope**
    - nel nostro esempio, la sola classe WordService
  - inoltre, bisogna utilizzare Spring Boot Actuator ed esporre l'endpoint *refresh* (/actuator/refresh)

Configurazioni dinamiche con Spring Cloud Config

- Come fare per aggiornare la configurazione di un'applicazione in esecuzione?
  - di per sé, la modifica dei file di configurazione non implica il refresh della configurazione dell'applicazione in esecuzione
  - piuttosto, per avviare il refresh della configurazione di un'applicazione in esecuzione bisogna effettuare una richiesta HTTP/POST al suo endpoint /actuator/refresh
  - nel nostro caso, se si vogliono aggiornare le parole per tutti e tre i servizi per le parole, bisogna fare richieste POST agli URI localhost:8081/actuator/refresh, localhost:8082/actuator/refresh e localhost:8083/actuator/refresh
  - questa soluzione è però inadeguata qualora ci siano molte istanze di servizi di cui va aggiornata la configurazione (o, peggio, quando non si sa quante siano e quale sia la loro locazione)

- Configurazioni dinamiche con Spring Cloud Bus

- Si supponga di avere in esecuzione molte repliche di un'applicazione e che si voglia effettuare un aggiornamento dinamico della configurazione di tutte queste repliche
  - usando solo Spring Cloud Config, bisogna richiedere il refresh individualmente a ciascuna delle repliche dell'applicazione
- Nel contesto della gestione delle configurazioni, *Spring Cloud Bus* semplifica il refresh di tutte le repliche di un'applicazione
  - nell'applicazione, oltre a **@RefreshScope**, bisogna utilizzare la dipendenza spring-cloud-starter-bus-amqp ed esporre mediante Actuator l'endpoint *bus-refresh* (/actuator/busrefresh)
  - bisogna anche avere un server per lo scambio di messaggi (ad es., RabbitMQ) e configurarne l'accesso nell'applicazione
  - una richiesta HTTP/POST all'endpoint /actuator/busrefresh di una singola replica dell'applicazione in esecuzione avvia il refresh della configurazione di tutte le repliche dell'applicazione

- Discussione

- Nei sistemi distribuiti, la gestione delle configurazioni è importante
  - in ogni caso, per semplicità nei prossimi esempi non viene più utilizzato un server delle configurazioni

* Service discovery con Spring Cloud Consul Discovery

- In un'applicazione distribuita, composta da più servizi (applicativi), è in genere necessario un servizio (infrastrutturale) di *service discovery* che consenta a ciascun servizio (applicativo) di conoscere gli altri servizi dell'applicazione e la loro locazione
  - un servizio di service discovery (che nell'architettura a servizi è l'analogo di un "broker") è utile, ad es., affinché un servizio possa interagire con altri servizi dell'applicazione
  - alcune soluzioni comuni sono Consul, Etcd, Zookeper e Netflix Eureka
  - queste soluzioni sono basate su un approccio *passivo* al service discovery
    - ogni nuovo servizio (applicativo) deve registrare la propria presenza al servizio di service discovery (che in primo luogo gestisce un registry di servizi)
    - dopo di che, questo servizio (applicativo) può essere scoperto (trovato) dagli altri servizi dell'applicazione

Consul

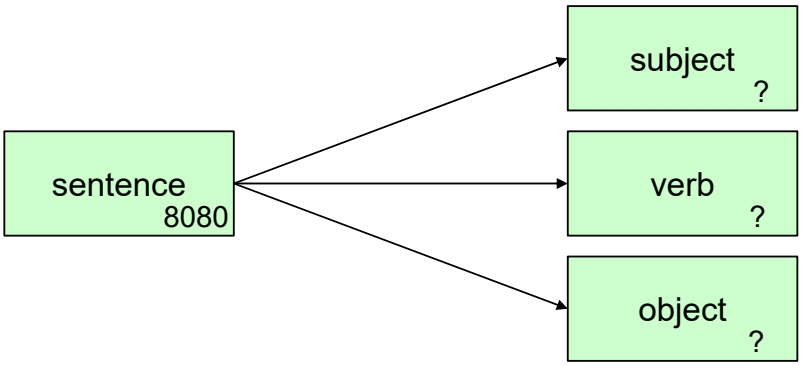
- *Consul* (www.consul.io) è uno strumento per connettere servizi distribuiti in qualunque piattaforma
  - fornisce un servizio di service discovery, di tipo client-server – nonché un servizio di storage key-value distribuito
  - al loro avvio, i servizi applicativi si registrano presso il server Consul – comunicando la propria locazione (host, port, ecc.)
    - nota: i servizi applicativi agiscono da "client" nei confronti del servizio di service discovery
  - al loro arresto, normalmente i servizi si deregistrano da Consul
    - che fare però per gestire l'eventuale crash di un servizio applicativo?
    - ogni servizio "client" deve inviare periodicamente degli heartbeat a Consul – che altrimenti, dopo un po', lo rimuove
  - inoltre, Consul offre un servizio di lookup dei servizi "client" registrati

Consul e Spring Cloud Consul Discovery

- *Consul* (www.consul.io) è uno strumento per connettere servizi distribuiti in qualunque piattaforma
  - *Spring Cloud Consul Discovery* fornisce una libreria per interagire con Consul (lato client) in modo semplice

- Esempio

- Consideriamo ancora l'applicazione per generare frasi in modo casuale
  - vogliamo modificare l'applicazione affinché sia possibile rilasciare i servizi per le parole (subject, verb e object) in locazioni non note a priori – in questo esempio, su porte casuali



  - il servizio principale sentence potrà accedere ai diversi servizi per le parole tramite il servizio di service discovery

- Server Consul

- L'utilizzo di Consul richiede, in genere, la definizione di un cluster, con uno o più nodi
  - un modo semplice per utilizzare Consul, soprattutto durante lo sviluppo, è di mandarlo in esecuzione con Docker
    - docker run -d --hostname localhost --name asw-consul --publish 8500:8500 consul
    - Consul sarà in ascolto sulla porta 8500 (di localhost)
  - Consul può essere poi arrestato con i seguenti comandi
    - docker stop asw-consul
    - docker rm asw-consul

- Client del servizio di service discovery

- Nella nostra applicazione, sia il servizio per le frasi che i servizi per le parole devono operare come client nei confronti del servizio di service discovery Consul – pertanto, in questi servizi
  - va utilizzata la dipendenza starter spring-cloud-starter-consul-discovery
  - la classe principale dell'applicazione va annotata con **@EnableDiscoveryClient**
    - come servizio di service discovery verrà usato Consul, perché è presente nel classpath
  - va configurato l'accesso al server Consul nel file application.properties o application.yml
  - si noti, nel seguito, che il codice sorgente del client non dipende in alcun modo da Consul – infatti, il **DiscoveryClient** di Spring Cloud Commons fornisce un'astrazione dalle possibili implementazioni concrete del servizio di service discovery

Il file application.yml per le parole

- Per quanto riguarda il servizio per le parole, nel file application.yml
  - la proprietà server.port va configurata in modo da assegnare al servizio una porta casuale – va usata l'espressione `${PORT:${SERVER_PORT:0}}`
    - ciascuna istanza in esecuzione di questo servizio sarà collegata a una porta casuale diversa
    - dunque, la locazione dei servizi non è nota a priori

```
# application.yml (word)
---
server:
  port: ${PORT:${SERVER_PORT:0}}
```

  - ora è anche possibile mandare in esecuzione più istanze di questo servizio per ciascun tipo di parole

Il file application.yml per le parole

- Per quanto riguarda il servizio per le parole, nel file application.yml
  - inoltre, per ciascuno dei profili (per i tre tipi di parole) bisogna specificare (oltre a un insieme di parole) il nome specifico per l'applicazione (spring.application.name) con cui registrare l'applicazione nel servizio di discovery – utilizziamo il nome del profilo

```
# application.yml
---
spring:
  application.name: ${spring.config.activate.on-profile}

---
spring:
  config.activate.on-profile : subject
words: Ann,Mary,John,My dog
```

  - gli altri profili sono analoghi

Il file application.yml per le parole

- Per quanto riguarda il servizio per le parole, nel file application.yml
  - nella sezione comune a tutti i profili, va anche configurato l'accesso al servizio di service discovery Consul

```
spring:
  application.name: ${spring.config.activate.on-profile}
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        instanceId: ${spring.application.name}-${random.value}
        healthCheckPath: /actuator/health
        healthCheckInterval: 10s
```

  - l'instanceId va configurato per garantire che ogni istanza del servizio delle parole sia registrata nel servizio di service discovery con un identificatore univoco
  - la ricerca del servizio viene effettuata usando solo lo spring.application.name

- Client per il servizio delle parole

- Nel servizio principale per le frasi, è utile modificare un pochino gli adattatori per accedere ai servizi per le parole – per accedere a un servizio tramite il suo nome "logico" anziché il suo URI "fisico"
  - ecco l'adattatore REST per accedere al servizio subject – quelli per verb e object sono simili

```
package asw.sentence.sentenceservice.wordservice;

import asw.sentence.sentenceservice.domain.WordService;

import ...

@Service
public class SubjectService implements WordService {

    @Autowired
    private WordClient wordClient;

    public String getWord() {
        return wordClient.getWord("subject");
    }

}
```


Client per il servizio delle parole

- Va modificato anche il componente WordClient usato per effettuare le chiamate REST per accedere ai servizi per le parole
  - il componente può operare in questo modo
    - prima deve cercare il servizio per il tipo di parola richiesto utilizzando il **DiscoveryClient** di Spring Cloud Commons
      - la ricerca di un servizio avviene mediante il suo spring.application.name e restituisce il suo URI – o l'URI di ciascuna istanza del servizio in esecuzione
      - nel nostro esempio questo passaggio è necessario, perché non conosciamo la locazione (le porte) per i servizi per i diversi tipi di parole
    - poi può accedere al servizio, a partire dal suo URI, mediante RestTemplate oppure WebClient

Accesso alle parole mediante DiscoveryClient

- L'interfaccia **WordClient** modificata
  - ora il parametro dell'operazione getWord è il nome "logico" del servizio da accedere, non il suo URI "fisico"

```
package asw.sentence.sentenceservice.wordservice;

public interface WordClient {

   public String getWord(String service);

}
```

  - nel seguito, la sua implementazione basata sul **DiscoveryClient**, con **RestTemplate** e **WebClient**

Accesso alle parole mediante DiscoveryClient e RestTemplate

```
package asw.sentenceservice.sentence.wordservice;

import ...

@Service
public class WordClientDiscoveryClientRestTemplate implements WordClient {

   @Autowired
   private DiscoveryClient discoveryClient;
   @Autowired
   private RestTemplate restTemplate;

   public String getWord(String service) {
      URI uri = getWordUri(service);
      return restTemplateGet(uri);
   }

   private URI getWordUri(String service) { … }

   private String restTemplateGet(URI uri) { … }

}
```

Accesso alle parole mediante DiscoveryClient e RestTemplate

- Metodi di supporto

```
private URI getWordUri(String service) {
   URI uri = null;
   List<ServiceInstance> list = discoveryClient.getInstances(service);
   if (list!=null && list.size()>0) {
        /* scegli il primo, oppure un elemento casuale della lista */
        uri = list.get(0).getUri();
   }
   return uri;
}

private String restTemplateGet(URI uri) {
   String word = null;
   try {
        word = restTemplate.getForObject(uri, String.class);
   } catch (RestClientException e) { ... eccezione remota ... }
   return word;
}
```


Accesso alle parole mediante DiscoveryClient e WebClient

```
package asw.sentenceservice.sentence.wordservice;

import ...

@Service
public class WordClientDiscoveryClientWebClient implements WordClient {

   @Autowired
   private DiscoveryClient discoveryClient;

   @Autowired
   private WebClient webClient;

   public String getWord(String service) {
      URI uri = getWordUri(service);
      return webClientGet(uri);
   }

   private URI getWordUri(String service) { ... come prima ...  }

   private String webClientGet(URI uri) { … }

}
```

Accesso alle parole mediante DiscoveryClient e WebClient

- Metodi di supporto

```
private String webClientGet(URI uri) {
   String word = null;
   Mono<String> response = webClient
            .get()
            .uri(uri)
            .retrieve()
            .bodyToMono(String.class);
   try {
        word = response.block();
   } catch (WebClientException e) { ... eccezione remota ... }
   return word;
}
```


Il file application.yml per le frasi

- Bisogna modificare il file application.yml per il servizio per le frasi
  - il servizio viene ancora mandato in esecuzione sulla porta 8080

```
# application.yml (sentence)
---
spring.application.name: sentence
spring.cloud:
  consul:
    host: localhost
    port: 8500
    discovery:
      instanceId: ${spring.application.name}-${random.value}
      healthCheckPath: /actuator/health
      healthCheckInterval: 10s

server.port: 8080
```

  - si noti che sono stati rimossi gli indirizzi statici per i servizi per le parole

Discussione

- Il server Consul è pensato per essere mandato in esecuzione con più istanze
  - ogni server Consul aggiuntivo va avviato specificando la locazione del primo server Consul – in modo che possano sincronizzarsi e coordinarsi
- I server Consul gestiscono le informazioni sui servizi registrati (solo) in memoria principale
  - Consul (come altri servizi di service discovery) si affida alla registrazione dei servizi da parte dei client – e ai loro successivi heartbeat
  - i dati sui servizi sono sempre in memoria (non vengono mai registrati su disco) e sono sempre aggiornati

Discussione

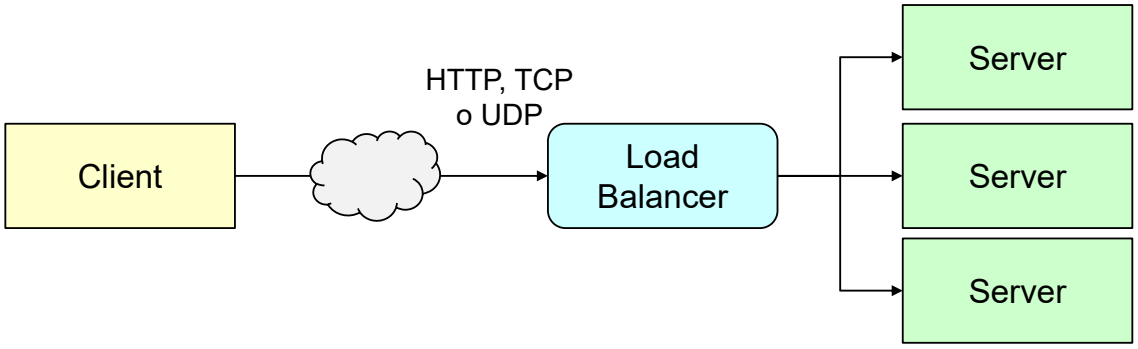
- È anche possibile un utilizzo congiunto di un server di configurazione e di un servizio di service discovery
  - una possibilità è usare il server delle configurazioni per specificare la locazione del servizio di service discovery
  - un'altra possibilità è usare il servizio di service discovery per gestire l'accesso al server delle configurazioni
  - in entrambi i casi, i servizi che compongono l'applicazione devono conoscere la locazione di uno solo tra questi server
  - peraltro, Consul (che è un servizio di storage key-value distribuito) è in grado di memorizzare configurazioni e altri metadati
    - *Spring Cloud Consul Config* è un'alternativa al Config Server descritto in precedenza

Discussione

- Va osservato che molte piattaforme cloud e per l'orchestrazione di container offrono un proprio servizio di service discovery
  - di volta in volta bisognerà decidere se usare il servizio di service discovery della piattaforma oppure un'implementazione separata di un tale servizio
  - ad es., il progetto Spring Cloud Kubernetes fornisce un'implementazione di DiscoveryClient per accedere al servizio di service discovery fornito da Kubernetes

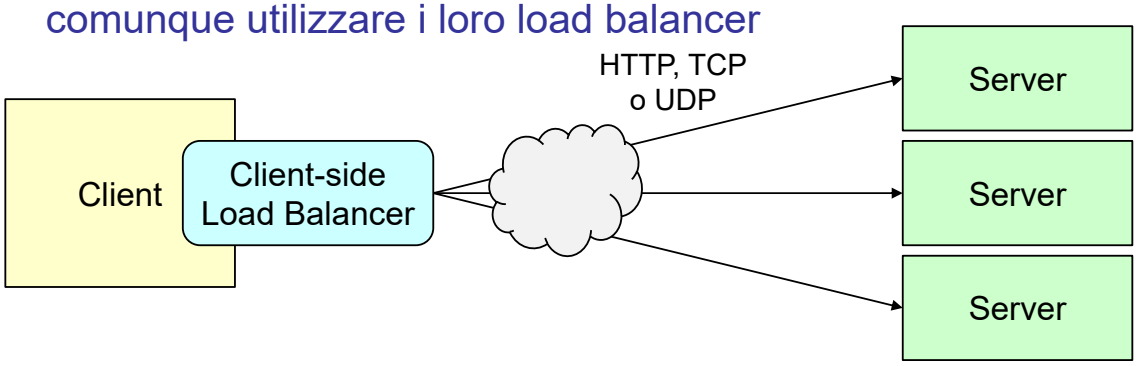
* Client-side load balancing con Spring Cloud LoadBalancer

- Nelle applicazioni distribuite, i servizi sono spesso replicati (ovvero, ne esistono più istanze), per motivi di disponibilità e di scalabilità – in questo caso, i client possono accedere ai servizi mediante un servizio intermedio di *load balancing* (bilanciamento del carico)
  - spesso il *load balancer* è un componente *server-side* (lato server) – può essere un componente software (come Apache, NGINX o HA Proxy) oppure hardware



Client-side load balancing

- Talvolta è anche utile un load balancer di tipo *client-side* (lato client)
  - in questo caso, il load balancer è parte dell'applicazione client, e ha la responsabilità di selezionare il server da contattare, sulla base di qualche criterio – anche i server possono comunque utilizzare i loro load balancer



  - la soluzione client-side può essere vantaggiosa nel cloud, quando i server sono dislocati in zone o regioni geografiche diverse – un load balancer client-side può essere configurato per preferire server più vicini, con minor latenza o più affidabili

Spring Cloud LoadBalancer

- *Spring Cloud LoadBalancer* (parte del progetto *Spring Cloud Commons*) fornisce un servizio di *load balancing* di tipo *client-side*
  - inoltre, si integra automaticamente con il servizio di service discovery utilizzato e con altri strumenti, come quelli per la definizione dichiarativa di client REST (OpenFeign) e la resilienza (Circuit Breaker), discussi più avanti
  - nel servizio client va usata la dipendenza spring-cloud-starter-loadbalancer

Spring Cloud LoadBalancer

- Alcune nozioni di Spring Cloud LoadBalancer
  - per ogni servizio c'è una lista di server
    - in genere viene gestita dinamicamente tramite il servizio di service discovery
  - è possibile definire criteri per configurare il comportamento del load balancer – ad esempio
    - per selezionare l'implementazione del load balancer da utilizzare (ad es., Ribbon di Netflix)
    - per scegliere l'algoritmo di load balancing
    - per limitare l'accesso solo ad alcuni server (in genere quelli nella stessa zona)
    - per effettuare un health check dei server tramite dei ping – ma, per default, se ne occupa il servizio di service discovery

Accesso alle parole mediante Load Balancer

- Nella nostra applicazione, il servizio delle frasi potrebbe utilizzare un load balancer nell'accesso ai servizi delle parole
  - per utilizzare un load balancer, va utilizzata una diversa implementazione dell'interfaccia WordClient per accedere al servizio delle parole
  - la soluzione è simile alla precedente, ma utilizza il **LoadBalancerClient** di Spring Cloud LoadBalancer anziché il **DiscoveryClient** di Spring Cloud Commons

Accesso alle parole mediante Load Balancer e RestTemplate

```
package asw.sentence.sentenceservice.wordservice;

import ...

@Service
public class WordClientLoadBalancerClientRestTemplate implements WordClient {

   @Autowired
   private LoadBalancerClient loadBalancer;
   @Autowired
   private RestTemplate restTemplate;

   public String getWord(String service) {
      URI uri = getWordUri(service);
      return restTemplateGet(uri);
   }

   private URI getWordUri(String service) { … }

   private String restTemplateGet(URI uri) { ... come prima ... }

}
```


Accesso alle parole mediante Load Balancer e RestTemplate

- Metodi di supporto

```
private URI getWordUri(String service) {
   URI uri = null;
   ServiceInstance instance = loadBalancer.choose(service);
   if (instance!=null) {
        uri = instance.getUri();
   }
   return uri;
}
```

Accesso alle parole mediante Load Balancer e WebClient

```
package asw.sentence.sentenceservice.wordservice;

import ...

@Service
public class WordClientLoadBalancerClientWebClient implements WordClient {

    @Autowired
    private LoadBalancerClient loadBalancer;

    @Autowired
    private WebClient webClient;

    public String getWord(String service) {
        URI uri = getWordUri(service);
        return webClientGet(uri);
    }

    private URI getWordUri(String service) { ... come prima ...  }

    private String webClientGet(URI wordUri) { ... come prima ... }

}
```


Accesso alle parole in modo bilanciato

- In alternativa
  - è possibile configurare il **RestTemplate** o il **WebClient** in modo che operino automaticamente in modo bilanciato
  - dopo di che, l'accesso al servizio delle parole di interesse deve essere effettuato con riferimento al nome "logico" del servizio – mediante un URI "virtuale" e non "fisico"
    - ad es., http://subject

Accesso alle parole mediante WebClient load balanced

```
package asw.sentence.sentenceservice.wordservice.config;

import ...

@Configuration
public class LoadBalancedWebClientConfig {

   @Bean("loadBalancedWebClientBuilder")
   @LoadBalanced
   public WebClient.Builder loadBalancedWebClientBuilder() {
      return WebClient.builder();
   }

   @Bean("loadBalancedWebClient")
   public WebClient loadBalancedWebClient(
            WebClient.Builder loadBalancedWebClientBuilder) {
      return loadBalancedWebClientBuilder.build();
   }

}
```


Accesso alle parole mediante WebClient load balanced

```
package asw.sentence.sentenceservice.wordservice;

import ...

@Service
public class WordClientLoadBalancedWebClient implements WordClient {

   @Autowired @Qualifier("loadBalancedWebClient")
   private WebClient webClient;

   public String getWord(String service) {
      String serviceUri = "http://" + service;
      String word = null;
      Mono<String> response = webClient
            .get()
            .uri(serviceUri)
            .retrieve()
            .bodyToMono(String.class);
      try {
         word = response.block();
      } catch (WebClientException e) { ... eccezione remota ... }
      return word;
   }
}
```

Discussione

- L'utilizzo di un load balancer lato client può consentire una comunicazione più efficiente tra i servizi di un'applicazione distribuita – nel caso comune in cui ci siano servizi replicati
  - il modo più semplice ed efficace di utilizzare un load balancer lato client è in modo integrato con un servizio di service discovery

* Client REST dichiarativi con Spring Cloud OpenFeign

- *OpenFeign* (github.com/OpenFeign/feign) è una libreria per semplificare la realizzazione di client HTTP/REST in Java
  - consente di effettuare chiamate REST in modo semplificato
  - costituisce un'alternativa (più semplice) a `RestTemplate` di Spring
    - purtroppo, ad oggi non supporta ancora i client reactive basati su `WebClient`
  - inoltre si integra automaticamente con i servizi di service discovery e di load balancing lato client, nonché con i circuit breaker (discussi dopo)
  - *Spring Cloud OpenFeign* fornisce un wrapper per utilizzare OpenFeign in modo semplice nelle applicazioni Spring Cloud
  - nel servizio client va usata la dipendenza starter `spring-cloud-starter-openfeign`

OpenFeign

- OpenFeign è basato sulla definizione di interfacce (non classi) per il codice client REST
  - queste interfacce vanno etichettate con l'annotazione **@FeignClient** – vanno inoltre utilizzate delle annotazioni che sono analoghe a quelle usate con Spring Web MVC per annotare un servizio REST
  - non c'è però bisogno di implementare queste interfacce – piuttosto, OpenFeign implementa dinamicamente e automaticamente il codice client a runtime (in modo analogo ai repository dinamici di Spring Data), definendo il codice per chiamare il servizio REST ed elaborare la risposta
  - l'applicazione principale deve essere annotata anche con **@EnableFeignClients** – che abilita la generazione dinamica dei client OpenFeign

Un'interfaccia client OpenFeign

- Come esempio, nel servizio delle frasi è possibile usare questa interfaccia OpenFeign per accedere tramite REST al servizio subject

```
package asw.sentence.sentenceservice.wordservice;

import ...

@FeignClient("subject")
public interface SubjectClient {

   @GetMapping("/")
   public String getWord();

}
```

  - le interfacce per verb e object sono analoghe
  - OpenFeign supporta anche altre annotazioni di Spring Web MVC – ad es., richieste POST e PUT, nonché operazioni con parametri e tipi di ritorno

Un'interfaccia client OpenFeign

- Si noti che nell'annotazione **@FeignClient** è stato usato il nome "logico" di un servizio (ovvero, lo spring.application.name del servizio)
  - è lo stesso nome usato per registrare il servizio presso il servizio di service discovery

```
@FeignClient("subject")
public interface SubjectClient { ... }
```

  - in alternativa, si può usare OpenFeign anche per accedere a un servizio di cui è noto l'URI "fisico" – scrivendo, ad esempio

```
@FeignClient(name="lucky-word", url="localhost:8080/lucky-word")
public interface LuckyWordClient { ... }
```


L'adattatore REST per accedere al servizio delle parole

- L'adattatore REST per accedere al servizio subject va modificato come segue – e in modo analogo gli adattatori per verb e object

```
package asw.sentence.sentenceservice.wordservice;

import asw.sentence.sentenceservice.domain.WordService;

import ...

@Service
public class SubjectService implements WordService {

    @Autowired
    private SubjectClient feignClient;

    public String getWord() {
        String word = null;
        try {
            word = feignClient.getWord();
        } catch (FeignException e) { ... eccezione remota ... }
        return word;
    }

}
```

Discussione

- OpenFeign consente la specifica dichiarativa e semplificata di client REST
  - OpenFeign gestisce l'implementazione dinamica del codice
  - inoltre, se nel classpath sono presenti un servizio di service discovery e un load balancer lato client, allora
    - automaticamente viene utilizzato il servizio di service discovery per accedere a tutti i servizi applicativi registrati con l'identificatore specificato nel client OpenFeign
    - automaticamente viene utilizzato il load balancer specificato
  - è necessario scrivere (e verificare) molto meno codice (rispetto a quello mostrato in precedenza)
  - in ogni caso, per semplicità nei prossimi esempi torneremo a considerare client web basati su RestTemplate o WebClient

* Circuit breaker con Spring Cloud Circuit Breaker

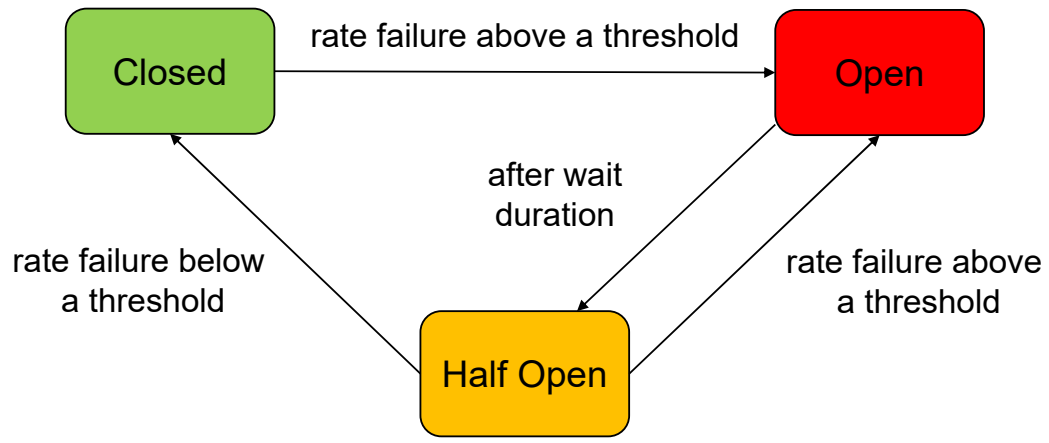
- *Circuit Breaker* è un pattern per la resilienza (per la tolleranza ai guasti e per sostenere l'isolamento dei guasti) nell'accesso a un servizio remoto
  - questo termine significa "interruttore automatico" o "salvavita" – è un dispositivo (presente in tutte le case) che in caso di problemi apre un circuito elettrico, per evitare problemi peggiori
  - come pattern software, è possibile usare un circuit breaker per incapsulare le chiamate a un servizio remoto che però potrebbe essere non disponibile (oppure essere disponibile in modo intermittente) – ovvero, questo servizio remoto potrebbe spesso restituire degli errori oppure avere una latenza alta
    - vogliamo però proteggere (isolare) i client di questo servizio remoto da questi problemi

Circuit breaker

- Funzionamento di un circuit breaker
  - quando il circuito è “chiuso”, il servizio remoto viene effettivamente chiamato
  - se però si verificano dei problemi nelle chiamate (superiori a certe soglie), allora il circuito si “apre” automaticamente
    - quando il circuito è “aperto”, le chiamate successive non provano nemmeno a raggiungere il servizio remoto
  - dopo un po’, il circuito prova a richiudersi automaticamente – passando in uno stato “semi-aperto”
    - il servizio remoto viene nuovamente chiamato
    - se i problemi cessano (scendono sotto certe soglie), il circuito torna nello stato “chiuso”, altrimenti torna in quello “aperto”

Circuit breaker

- Funzionamento di un circuit breaker



Circuit breaker

- Il pattern Circuit Breaker affronta dunque alcuni problemi nell'accesso a un servizio remoto
  - ha soprattutto lo scopo di evitare fallimenti a cascata – per evitare che l'indisponibilità di un servizio faccia fallire tutti i servizi che ne dipendono
  - un po' di matematica – supponiamo che ci siano dei servizi con una propria disponibilità del 99,95%
    - un singolo servizio ha un downtime di 22 minuti al mese (potrebbe andare bene)
    - un gruppo di 30 servizi interdipendenti ha un downtime di 11 ore al mese (male)
    - un gruppo di 100 servizi interdipendenti ha un downtime di 36 ore al mese (molto male)
  - è complementare a una soluzione basata sulla ripetizione delle richieste – che è efficace per guasti transienti

Resilience4J

- *Resilience4J* (resilience4j.readme.io/) è una libreria per la tolleranza ai guasti, ispirata a Netflix Hystrix
  - implementa diversi pattern per la resilienza – Circuit Breaker, Rate Limiter, Retry e Bulkhead
  - il comportamento di un circuit breaker è personalizzabile – le impostazioni di default sono le seguenti
    - un circuito si apre se in una finestra di 100 chiamate si verificano il 50% di fallimenti
    - un circuito passa automaticamente allo stato semi-aperto dopo 60 secondi – in cui rimane per 20 chiamate
  - *Spring Cloud Circuit Breaker* fornisce un wrapper per utilizzare Resilience4J in modo abbastanza semplice
  - nel servizio client va usata la dipendenza spring-cloud-starter-circuitbreaker-resilience4j

Circuit breaker con Resilience4J

- Con Resilience4J, una chiamata REST può essere protetta con un circuit breaker delegando la chiamata al circuit breaker
  - in Java, va utilizzata una funzione di ordine superiore come decoratore
  - inoltre, è possibile definire un metodo di "fallback" ("ripiego") – da usare in caso di fallimento del servizio remoto
  - l'adattatore REST per accedere al servizio subject va modificato come mostrato nel seguito – in modo analogo gli adattatori per verb e object

L'adattatore REST per accedere al servizio delle parole

```
package asw.sentence.sentenceservice.wordservice;

import asw.sentence.sentenceservice.domain.WordService;

import ...

@Service
public class WordClientCircuitBreakerRestTemplate implements WordClient {

   @Autowired @Qualifier("loadBalancedRestTemplate")
   private RestTemplate restTemplate;

   @Autowired private CircuitBreakerFactory cbFactory;

   public String getWord(String service) {
      String serviceUri = "http://" + service;
      return cbFactory.create(service).run(
               () -> restTemplate.getForObject(serviceUri, String.class),
               throwable -> this.getFallbackWord()
          );
   }

   private String getFallbackWord() { return "* fallback *"; }

}
```

L'adattatore REST per accedere al servizio delle parole

- È possibile anche configurare il circuit breaker – va definito un "customizer"

```
package asw.sentence.sentenceservice.wordservice.config;

import ...

@Configuration
public class Resilience4JConfig {

   @Bean
   public Customizer<Resilience4JCircuitBreakerFactory>
                                                defaultCustomizer() {

      return factory -> factory.configureDefault( ... );

   }

}
```

  - le configurazioni da usare con RestTemplate e WebClient sono leggermente diverse

Discussione

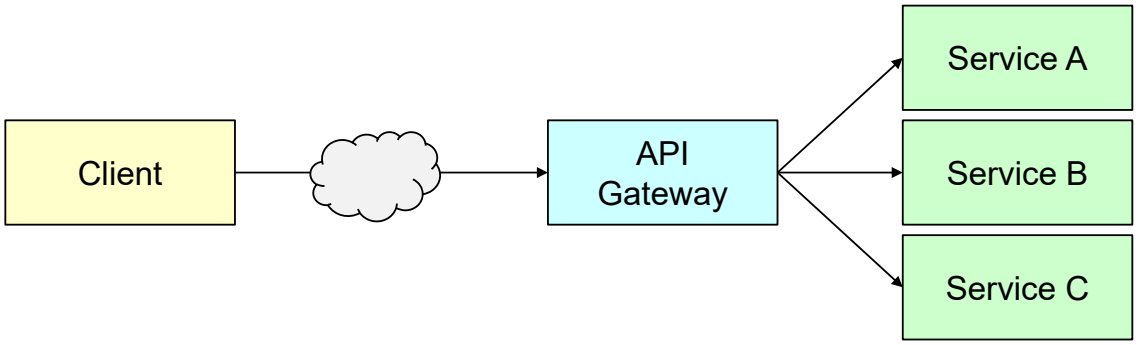
- Il pattern Circuit Breaker offre una protezione nell'accesso a un servizio remoto
  - Spring Cloud Circuit Breaker è compatibile con diverse implementazioni – tra cui Resilience4J e Hystrix di Netflix
  - è possibile la definizione di metodi di fallback
  - è in genere possibile una configurazione flessibile del circuit breaker
  - è possibile anche il monitoraggio dei circuiti e la raccolta di metriche
  - il pattern Circuit Breaker non è però una soluzione universale per tutti i problemi di affidabilità – è solo una possibile soluzione, che in genere è adeguata solo in casi specifici

* API Gateway con Spring Cloud Gateway

- Nelle applicazioni a microservizi, è utile fornire ai client (finali) un punto di accesso integrato all'applicazione e ai suoi microservizi mediante un'API unificata
  - talvolta ci sono diversi tipi di client (ad es., web, mobile, smart tv) e, nei diversi casi, si desidera un'integrazione differente dei microservizi, basata su un'API specializzata
  - *API Gateway* è un pattern per gestire questo problema
    - è un caso specifico del pattern *Gateway*, che è un intermediario che incapsula l'accesso a risorse o servizi esterni – che a sua volta è una variante dei pattern *Facade* e *Adapter*

API Gateway

- Un *API gateway* è un servizio (lato server) che fornisce un punto di accesso unificato a un'applicazione composta da più (micro) servizi, con un'API personalizzata per un tipo specifico di client



  - effettua il routing delle chiamate del client ai servizi specifici
  - consente di ridurre il numero di chiamate remote
  - come pattern, può avere anche altre responsabilità, come la gestione della sicurezza (Single-Sign On), la composizione di API e la conversione di protocolli

Spring Cloud Gateway

- *Spring Cloud Gateway* è una libreria per costruire un API gateway sopra Spring Web MVC
  - l'obiettivo principale è fornire funzionalità semplici ma efficaci per il routing dinamico delle richieste alle API di altri servizi
    - è possibile definire le rotte di interesse in modo dichiarativo e flessibile mediante predicati e filtri (discussi dopo)
  - inoltre supporta la sicurezza, il monitoraggio e la resilienza (mediante circuit breaker) – sulla base di un'integrazione semplificata con altre librerie di Spring Cloud
  - agisce come un reverse proxy – ovvero, un proxy, lato server, che, per conto dei suoi client, effettua delle richieste e recupera dei contenuti da uno o più servizi (in modo trasparente ai suoi client)

API gateway con Spring Cloud Gateway

- Implementazione di un API gateway con Spring Cloud Gateway
  - va implementata un'applicazione Spring Boot, con la dipendenza starter spring-cloud-starter-gateway
  - ora è questa applicazione che va esposta sulla porta 8080
    - il servizio sentence va invece esposto su una porta casuale – e potrà essere raggiunto tramite l'API gateway
  - questa applicazione va anche configurata in modo opportuno – ad esempio
    - per agire da client del servizio di service discovery, va usata la dipendenza starter spring-cloud-starter-consul-discovery con l'annotazione **@EnableDiscoveryClient**
    - vanno configurati load balancer lato client e circuit breaker
  - inoltre, l'API gateway può definire dei propri controller REST
    - ad es., per definire la composizione di API in modo programmatico o per definire operazioni di fallback

Il servizio per le frasi

- Il servizio sentence per le frasi va modificato come segue
  - il servizio va esposto su una porta casuale – anziché sulla porta 8080 (che ora è usata dall'API gateway)
  - ora è anche possibile avere più istanze del servizio sentence in esecuzione – oltre che più istanze dei servizi per i diversi tipi di parole

Routing delle richieste

- Vogliamo configurare l'API gateway per offrire ai suoi client questa API REST
  - l'operazione GET / deve restituire una frase casuale
    - ovvero, complessivamente quest'operazione deve essere accessibile con GET http://localhost:8080
    - questa operazione va mappata sull'operazione GET / del servizio sentence per le frasi
    - si ricordi che la vera locazione (di un'istanza) del servizio sentence sarà al path / su una porta casuale di localhost – ad es., http://localhost:58564 – è l'API gateway che deve effettuare il routing da http://localhost:8080 a http://localhost:58564
  - l'operazione GET /subject deve restituire un soggetto casuale
    - va mappata sull'operazione GET / del servizio subject
  - in modo analogo, le operazioni GET /verb e GET /object

Configurazione dell'API gateway

- La configurazione dell'API gateway è definita in modo dichiarativo mediante rotte, predicati e filtri
  - una *rotta* è una destinazione a cui vogliamo inoltrare delle specifiche richieste
    - ad es., l'operazione GET / del servizio sentence per le frasi
  - un *predicato* è una condizione relativa alla richiesta (può essere relativa a qualunque elemento della richiesta, ad es., al suo path)
    - ad es., l'operazione GET / dell'API gateway va inoltrata a questa rotta
  - un *filtro* consente di modificare richieste e risposte
    - ad es., per riscrivere il path da usare nella richiesta o per usare un circuit breaker nella gestione della richiesta

Configurazione dell'API gateway

- Questo è una porzione del file application.yml dell'API gateway, relativa al routing delle richieste

```
spring.cloud.gateway:
  routes:
  - id: sentence_route
    uri: lb://sentence
    predicates:
    - Path=/
    filters:
    - SetPath=/
    - name: CircuitBreaker
      args:
        name: defaultCircuitBreaker
        fallbackUri: forward:/fallback/sentence
  - id: subject_route
    ...
  - id: verb_route
    ...
  - id: object_route
    ...
```

il servizio a cui inoltrare le richiesta

il path su cui viene ricevuta la richiesta

riscrittura del path

Configurazione dell'API gateway

- Questo è una porzione del file application.yml dell'API gateway, relativa al routing delle richieste

```
spring.cloud.gateway:
  routes:
  - id: sentence_route
    ...
  - id: subject_route
    uri: lb://subject
    predicates:
    - Path=/subject
    filters:
    - SetPath=/
    - name: CircuitBreaker
      args:
        name: defaultCircuitBreaker
        fallbackUri: forward:/fallback/subject
  - id: verb_route
    ...
  - id: object_route
    ...
```

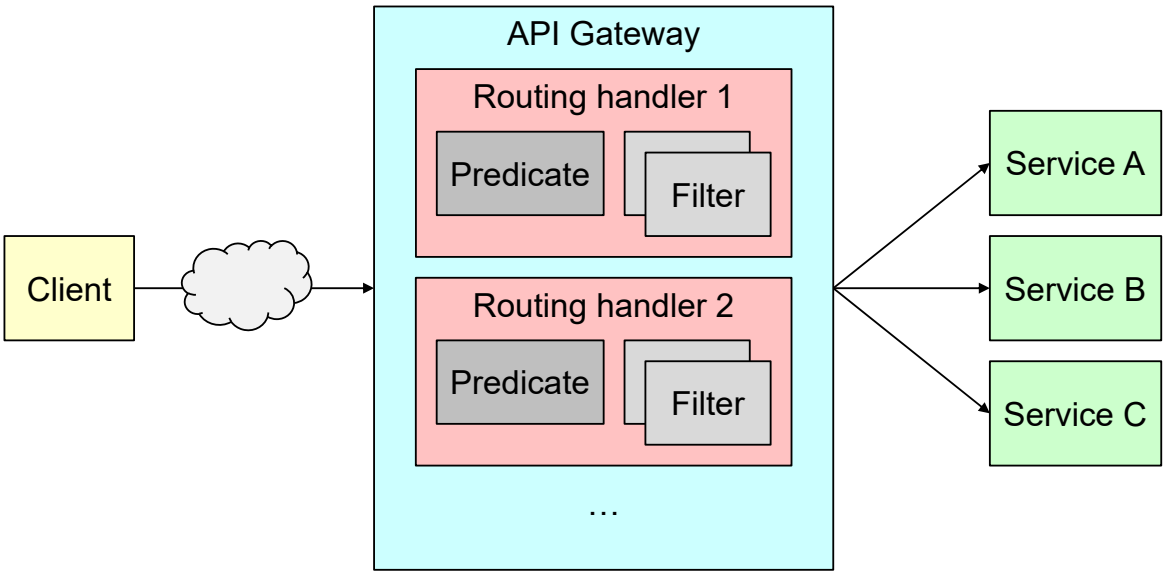
il servizio a cui inoltrare le richiesta

il path su cui viene ricevuta la richiesta

riscrittura del path

Configurazione dell'API gateway

- La configurazione dell'API gateway è definita in modo dichiarativo mediante rotte, predicati e filtri



Discussione

- Spring Cloud Gateway è una libreria di supporto all'implementazione del pattern API Gateway
  - si occupa del routing delle richieste dei client
  - opera in modo integrato con il servizio di service discovery, load balancer lato client e circuit breaker
    - ad es., è possibile la definizione implicita di rotte a partire dai servizi registrati sul servizio di service discovery

Discussione

- Va osservato che molte piattaforme cloud e per l'orchestrazione di container offrono un proprio servizio di routing delle richieste verso i servizi interni
  - di volta in volta bisognerà decidere se usare il servizio di routing delle richieste della piattaforma oppure una propria implementazione di un tale servizio
  - in ogni caso, un API gateway è spesso necessario – ad es., per la gestione della sicurezza e la composizione di API

* Invocazioni remote asincrone

- Uno dei limiti dell'implementazione dell'applicazione mostrata finora è che il servizio per le frasi (sentence) invoca i servizi per le parole casuali (subject, verb e object) in modo strettamente sequenziale
  - le eventuali latenze nelle invocazioni dei servizi per le parole vengono pertanto sommate
  - tuttavia, per evitare questo problema, è anche possibile effettuare le invocazioni remote in modo concorrente e in modo asincrono
  - esistono diverse possibilità, ne esaminiamo una

@Async e CompletableFuture<V>

- Ecco un modello di programmazione per operazioni asincrone
  - l'annotazione **@Async** indica un'operazione asincrona
    - può essere utilizzata in un'operazione del client che racchiude un'invocazione remota, per effettuare quell'invocazione remota in modo asincrono
      - intuitivamente, un'operazione asincrona viene eseguita in un thread separato rispetto a quello del suo chiamante
    - per abilitare le operazioni asincrone, la classe per l'applicazione Spring Boot va annotata con **@EnableAsync**
    - alcune osservazioni su **@Async**
      - va usata solo con metodi pubblici
      - non funziona se l'operazione asincrona viene invocata da un altro metodo della stessa classe

@Async e CompletableFuture<V>

- Ecco un modello di programmazione per operazioni asincrone
  - la classe **CompletableFuture<V>** può essere usata in un'operazione asincrona che deve restituire un valore di tipo **V**
    - l'invocazione remota *remoteCall()* può essere racchiusa in CompletableFuture.completedFuture(*remoteCall()*)
    - al momento dell'invocazione di un'operazione asincrona, al client viene immediatamente restituito un oggetto di tipo **CompletableFuture<V>** – che offre queste operazioni
      - **V get()** (bloccante) – per ottenere il valore restituito
      - **boolean isDone()** – per sapere se l'esecuzione dell'operazione asincrona è terminata normalmente
      - **CompletableFuture.allOf(*cf1*, *cf2*, …).join()** – attende la terminazione delle invocazioni ***cf1***, ***cf2***, …
      - **cancel(true)** – per chiedere la cancellazione dell'operazione asincrona

Cambiamenti al servizio delle frasi

- L'interfaccia asincrona degli adattatori per accedere ai servizi per le parole

```
package asw.sentence.sentenceservice.domain;

import java.util.concurrent.CompletableFuture;

public interface AsyncWordService {

    public CompletableFuture<String> getWord();

}
```

Cambiamenti al servizio delle frasi

- Il servizio di dominio per la generazione delle frasi

```
package asw.sentence.sentenceservice.domain;

import ...

@Service
public class SentenceService {

    @Autowired
    private AsyncWordService subjectService;
    @Autowired
    private AsyncWordService verbService;
    @Autowired
    private AsyncWordService objectService;

    public String getSentence() { ... segue ... }

}
```


Cambiamenti al servizio delle frasi

- Il servizio per la generazione delle frasi

```
    public String getSentence() {

        CompletableFuture<String> subject = subjectService.getWord();
        CompletableFuture<String> verb = verbService.getWord();
        CompletableFuture<String> object = objectService.getWord();
        CompletableFuture.allOf(subject, verb, object).join();
        String sentence = null;
        try {
             sentence =
                  subject.get() + " " +
                  verb.get() + " " +
                  object.get() + ".";

        } catch (Exception e) { ... eccezione remota ... }
        return sentence;

    }
```

Cambiamenti al servizio delle frasi

- L'adattatore REST per accedere al servizio subject delle parole – quelli per verb e object sono simili

```
package asw.sentence.sentenceservice.wordservice;

import asw.sentence.sentenceservice.domain.AsyncWordService;

import ...

@Service
public class SubjectService implements AsyncWordService {

    @Autowired
    private AsyncWordClient wordClient;

    public CompletableFuture<String> getWord() {
        return wordClient.getWord("subject");
    }

}
```


Cambiamenti al servizio delle frasi

- L'interfaccia AsyncWordClient per effettuare le chiamate REST in modo asincrono

```
package asw.sentence.sentenceservice.wordservice;

import java.util.concurrent.CompletableFuture;

public interface AsyncWordClient {

    public CompletableFuture<String> getWord(String service);

}
```

Cambiamenti al servizio delle frasi

- Un'implementazione di questa interfaccia

```
package asw.sentence.sentenceservice.wordservice;

import ...

@Service
public class AsyncWordClientRestTemplate implements AsyncWordClient {

    @Autowired @Qualifier("loadBalancedRestTemplate")
    private RestTemplate restTemplate;

    @Async
    public CompletableFuture<String> getWord(String service) {
        return
             CompletableFuture.completedFuture( getWordWrapper(service) );
    }

    private String getWordWrapper(String service) {
        String serviceUri = "http://" + service;
        return restTemplate.getForObject(serviceUri, String.class);
    }

}
```


Discussione

- Le operazioni asincrone e le invocazioni asincrone sono utili in caso di operazioni di lunga durata – oppure per implementare richieste di tipo *send-and-forget*
  - esistono diversi modi per implementare operazioni asincrone e gestire invocazioni asincrone
    - ne abbiamo mostrato uno, in cui, intuitivamente, l'invocazione di un'operazione asincrona avviene in un thread separato
    - può essere utilizzato anche per effettuare invocazioni remote asincrone
  - le operazioni asincrone abilitano anche l'esecuzione concorrente di più operazioni asincrone
    - questo è utile per evitare di sommare le latenze di più invocazioni remote, se queste possono effettivamente avvenire in modo concorrente anziché in modo strettamente sequenziale

* Discussione

- Spring Cloud, basato su Spring Boot, fornisce delle librerie che implementano alcuni pattern comuni, utili per lo sviluppo di servizi e applicazioni distribuite (in particolare, applicazioni a microservizi)
  - Spring Cloud integra alcune librerie, tecnologie e piattaforme per il cloud – consentendo un accesso unificato e semplificato
  - i servizi supportati da Spring Cloud comprendono
    - gestione centralizzata delle configurazioni (anche dinamiche)
    - registrazione e discovery di servizi
    - invocazione di servizi mediante client REST dichiarativi
    - bilanciamento del carico
    - circuit breaker
    - API gateway e routing
  - queste applicazioni possono poi essere rilasciate in ambienti distribuiti, fisici o virtuali, e anche e soprattutto nel cloud