This repository Search Pull requests Issues Gist

qaware / hitchhikers-guide-cloudnative Unwatch 4 Star 0 Fork 1

Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

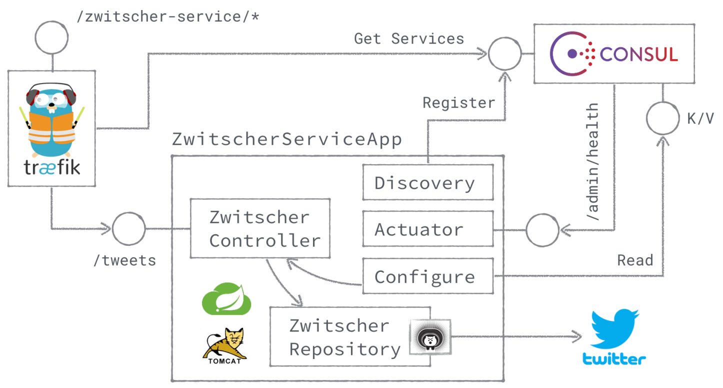
Branch: master hitchhikers-guide-cloudnative / labs / 03-composition / README.md Find file Copy path

lreimer Merge branch 'lab04-step02'. c8465c4 an hour ago

1 contributor

241 lines (177 sloc) 8.58 KB Raw Blame History

Exercise 3: Composition



Step 01: Enhance the Twitter service with Consul (Service Discovery)

In this step we will add basic Service Discovery capabilities using Consul so that our Zwitscher microservices instances can be found and addressed. The following steps are required to get things up and running:

- Start Consul in development mode as Docker container using the CLI or Gradle
- Add the Consul Discovery Starter dependency
- Enable the discovery client mechanism and configure the relevant properties
- Configure the management endpoints

Starting a Consul service via Docker should be straight forward after the last exercise. Pull and run the Docker container in the background, make sure to expose port 8500.

```
$ docker pull consul
$ docker run -d --name zwitscher-consul -p 8500:8500 consul

$ docker stop zwitscher-consul
$ docker start zwitscher-consul
```

Alternatively, you can wrap these commands in a Gradle `exec` task, e.g. like

```
task createConsul(type: Exec, group: 'application', description: 'Create Consul container') {
    commandLine 'docker', 'run', '-d', '--name', 'zwitscher-consul', '-p', '8500:8500', 'consul'
}
```

Next we will add the required Consul Discovery Starter dependency to the Gradle build file. Once done, we need to add the `@EnableDiscoveryClient` annotation to our main application.

```
dependencies {
    // ... the other dependencies ...
    compile 'org.springframework.cloud:spring-cloud-starter-consul-discovery'
}
```

Per default, this mechanism expects the Consul server to run on `localhost:8500`. Since we run the server with Docker we need to change the Consul host to the external Docker IP address, e.g. `192.168.99.100`. We also need to configure a unique instance ID used when registering with Consul. We will also register a health check URL to be run at regular intervals with Consul. For this, we will use the `/health` endpoint from the Actuator dependency. The final application properties might look something like the following.

```
spring.application.name=zwitscher-service
info.name=Zwitscher Service (W-JAX 2016 Hitchhiker's Guide)

# all actuator endpoints are below /admin
management.context-path=/admin

# specify Consul host and port
spring.cloud.consul.host=192.168.99.100
spring.cloud.consul.port=8500

# assign a unique instance ID
spring.cloud.consul.discovery.instance-id=${spring.application.name}:${spring.application.instance_id:$

# register IP address and heartbeats
spring.cloud.consul.discovery.prefer-ip-address=true
spring.cloud.consul.discovery.heartbeat.enabled=true

# specify the health check path and interval
spring.cloud.consul.discovery.health-check-path=${management.context-path}/health
spring.cloud.consul.discovery.health-check-interval=5s
```

Please make sure you assign something unique to the `info.name` property, so include your local username in here. Also replace the `hitchhikersguide` in the instance ID property with your username. This will make sure all instances are unique across all the tutorial participants.

Step 02: Enhance the Twitter service with Consul (Configuration)

In this step we want to enhance the service so that it uses Consul as configuration server to obtain the values for the Tweets query. The following steps are required to get things up and running:

- Add the Consul Configuration Starter dependency
- Enable the discovery client mechanism and configure the relevant properties
- Change our implementation to use properties instead of hard-coded values

First we will add the required Consul Config Starter dependency to the Gradle build file. Once done, we need to add a `src/main/resources/bootstrap.properties` file to configure the mechanism.

```
dependencies {
    // ... the other dependencies ...
    compile 'org.springframework.cloud:spring-cloud-starter-consul-config'
}
```

First, move the consul server configuration properties from `application.properties` to the newly created `bootstrap.properties` file. Add further properties to configure the desired behaviour of the configuration mechanism:

- Enable the configuration mechanism and configure prefix and default context.
- Disable fail-fast behaviour, so that the application will still start even if the Consul server is not running.
- Store properties in Consul as blob using a property syntax, opposed to having a key-value pair per property.

The `bootstrap.properties` file should now look something like this:

```
spring.application.name=zwitscher-service

# specify Consul host and port
spring.cloud.consul.host=${consul.host:192.168.99.100}
spring.cloud.consul.port=${consul.port:8500}

spring.cloud.consul.config.enabled=true
spring.cloud.consul.config.prefix=configuration
spring.cloud.consul.config.default-context=application

# do not fail at startup if Consul is not there
spring.cloud.consul.config.fail-fast=false

# store properties as blob in property syntax
# e.g. configuration/zwitscher-service/data
spring.cloud.consul.config.format=properties
spring.cloud.consul.config.data-key=data
```

Use the Consul UI and create a new key/value entry for the key `configuration/zwitscher-service/data` that contains the following data:

```
tweet.query=cloudnativenerd
tweet.pageSize=42
```

Now modify the `ZwitscherController` and inject and use these properties instead of the statically configured values. You may also annotate the class using `@RefreshScope`. Change the configuration in Consul while running the application, query the `/tweets` endpoints and see what happens.

```java
@RestController
@RequestMapping("/tweets")
@RefreshScope
public class ZwitscherController {

    private String query;
    private int pageSize;

    @Value("${tweet.query:cloudnativenerd}")
    public void setQuery(String query) {
        this.query = query;
    }

    @Value("${tweet.pageSize:42}")
    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }
}
```

Step 03: Add a Traefik edge server

In this step we will add a Traefik edge server as API gateway for our Zwitscher service. The following steps are required to get things up and running:

- Start Traefik as Docker container using the CLI
- Configure Traefik to access Consul for frontends and backends
- Register Traefik specific tags in Consul

Starting a Traefik instance via Docker is as usual pretty straight forward. Pull and run the offical Docker image in the background, make sure to expose port 8888 as well as port 80.

```
$ docker pull traefik
$ docker run -d --name zwitscher-traefik -p 8888:8888 -p 80:80 -v $PWD/traefik.toml:/etc/traefik/traefik

$ docker stop zwitscher-traefik
$ docker start zwitscher-traefik
```

Next, we need to customize the Traefik default behaviour. We want to serve the Web UI on port 8888 and we want to obtain the frontend and backend configuration dynamically from Consul. Create a file called `traefik.toml` with the following content and pass it as Docker volume to the container:

```
[web]
address = ":8888"

[consulCatalog]
endpoint = "192.168.99.100:8500"
domain = "consul.localhost"
prefix = "traefik"
```

Furthermore, we can customize how Traefik will route the incoming requests to the registered services by specifying additional tags during service registration. This is done via the `application.properties` .

```
spring.cloud.consul.discovery.tags=traefik.enable=true,traefik.frontend.rule=PathPrefixStrip:/zwitscher-
```

At this point you should now be able to get tweets via the edge server using a command like `curl 192.168.99.100/zwitscher-service/tweets` .

Step 04: Write a Docker Compose file

As the final step we will be using Docker Compose to start and wire all three Docker containers (Consul, Traefik, Zwitscher Service). For this create a file called `docker-compose.yml` and add the definitions for the required building blocks.

The solution provides a Docker compose file for v1 and v2. Choose either, even though v2 probably is the recommended format now.

```
$ docker-compose -p zwitscher -f docker-compose_v1.yml up -d

$ docker-compose -p zwitscher -f docker-compose_v2.yml up -d

$ docker-compose -p zwitscher -f docker-compose_v2.yml ps
$ docker-compose -p zwitscher -f docker-compose_v2.yml logs -f

$ docker-compose -p zwitscher -f docker-compose_v2.yml down
```

References

- https://hub.docker.com/_/consul/
- https://cloud.spring.io/spring-cloud-consul/
- https://cloud.spring.io/spring-cloud-config/spring-cloud-config.html

- https://hub.docker.com/r/_/traefik/
- https://docs.traefik.io/toml/#consul-catalog-backend
- https://docs.docker.com/compose/compose-file/

© 2016 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About