POLITECHNIKA
GDAŃSKA

# Microservices - configuration

Internet Services Architectures

Michał Wójcik

## Producer

**Producer**:

- some beans cannot be automatically produced by Spring Context,
- some beans require complex creation,
- calling constructors inside beans constructors is against dependency injection pattern,
- using `@Bean` annotation beans can be registered in Spring Context,
- `@Bean` annotation can be used in classes annotated with `@SpringBootApplication` or `@Configuration`.

POLITECHNIKA GDAŃSKA

## Producer

Declare producer method:

```java
@SpringBootApplication
public class Application {

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

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

}
```

POLITECHNIKA GDAŃSKA

## Producer

From this moment `RestTemplate` can be injected as any other managed bean:

```java
@Repository
public class RestRepository {

    @Autowired
    private RestTemplate restTemplate;

}
```

## Producer

Different beans configurations (different creation parameters) can be distinguished with qualifiers.

```
@Bean @Qualifier("library")
public RestTemplate restTemplate() {
    return new RestTemplateBuilder()
            .rootUri("http://localhost:8080/library")
            .build();
}
```

```
@Autowired @Qualifier("library")
private RestTemplate restTemplate;
```

## Gateway

**Gateway**:

- clients (mobile, web) need to communicate with different services,
- services decomposition should be transparent for clients,
- clients should not need to known location of all distributed services,
- there should be single gateway endpoint routing requests to particular services.

## Gateway

Spring Cloud provides Gateway implementation:

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

## Gateway

Gateway routing configuration is done by providing `RouteLocator` to Spring Context:

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder
            .routes()
            .route("library", r -> r
                    .host("localhost:8080")
                    .and()
                    .path("/api/books", "/api/books/**")
                    .uri("http://localhost:8081"))
            .build();
}
```

## Discovery

**Discovery**:

- services can be deployed on different addresses,
- services should not need to known exactly where other services are,
- there should be single (or distributed) catalog service,
- all services need to known only address of the catalog service.

## Discovery

Discovery services well integrated with Spring Cloud (and not only):

- Consul by HashiCorp,
- Eureka by Netflix.

## Consul

Consul can be started with Docker command:

```
docker run --rm --name consul -p 8500:8500 consul:1.10.3
```

## Consul

In client module appropriate dependency is required:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-all</artifactId>
</dependency>
```

and configuration in `application.properties`:

```
spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instance-id=library-1
spring.cloud.consul.discovery.service-name=library
```

## Discovery client

Spring discovery client is independent from implementation (Consul, Eureka).

Enable client on main class (not required in newer versions):

```java
@SpringBootApplication
@EnableDiscoveryClient
public class Application {

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

}
```

POLITECHNIKA GDAŃSKA

## Discovery client

Inject discovery client into component:

```
@Repository
public class RestRepository {

    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    public void delete() {
            URI uri = discoveryClient.getInstances("library")
                .stream()
                .findAny()
                .orElseThrow()
                .getUri();
            restTemplate.delete(uri + "/" + id);
    }
}
```

## Load balancer

**Load balancer**:

- there can be multiple instances of the same service,
- client can choose which service will be called,
- in order to balance the load different instances should be used,
- *hits* can be counted,
- round robin can be used,
- load balancing based on data ranges.

POLITECHNIKA GDAŃSKA

## Load balancer

Local (client side) load balancer is integrated with Spring Cloud discovery service:

```
@Repository
public class RestRepository {

    @Autowired
    private LoadBalancerClient loadBalancerClient;

    @Autowired
    private RestTemplate restTemplate;

    public void delete() {
            URI uri = loadBalancerClient.choose("library")
                .getUri();
            restTemplate.delete(uri + "/" + id);

}
```

POLITECHNIKA GDAŃSKA

## Load balancer

Load balancer can be also used automatically in Gateway:

```java
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder
            .routes()
            .route("library", r -> r
                    .host("localhost:8080")
                    .and()
                    .path("/api/books", "/api/books/**")
                    .uri("lb://library"))
            .build();
}
```

## Health check

**Health check**:

- there can be multiple instances of the same service,
- load balancer selects appropriate one to call,
- some of them can be down because of some errors,
- discovery service should be aware which are down,
- monitor tool for administrators would be helpful,
- can be called by service registry, load balancer or monitoring tool.

## Health check

Most of health check expose `/health` endpoint with JSON response:

```
{
    "status": "UP"
}
```

and `200` HTTP response code.

POLITECHNIKA GDAŃSKA

## Health check

It is enough to add Spring dependency to enable health check endpoint:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

POLITECHNIKA GDAŃSKA

## Database migration

**Database migration**:

- database schema needs to be created before application is stared,
- automatic generation mechanisms (eg. JPA) should not be used on production,
- schema can change with application development,
- on production with existing data appropriate migrations must be performed.

## Database migration

Some used Java libraries for migrations:

- Flyway (popular with Java EE applications),
- Liquibase (popular with Spring applications).

## Database migration

Dependency for Liquibase:

```
 <dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
</dependency>
```

Changelog (migrations) can be provided in XML and need to be configured in `application.properties`:

```
spring.liquibase.change-log=classpath:/db/changelog.xml
```

## Database migration

Main changelog can include number of changelogs:

```
<databaseChangeLog
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <include file="changelog/01-create-scheme.xml"
             relativeToChangelogFile="true"/>

</databaseChangeLog>
```

POLITECHNIKA GDAŃSKA

## Database migration

Migration described in XML format:

```xml
<databaseChangeLog
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
        xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

    <changeSet id="01" author="psysiu">

        <createTable tableName="books">
            <column name="isbn" type="varchar(255)">
                <constraints primaryKey="true"/>
            </column>
            <column name="title" type="varchar(255)"/>
        </createTable>
    </changeSet>

</databaseChangeLog>
```

## Centralized configuration

**Centralized configuration**:

- default configuration stored in `application.properties`,
- default configuration can be overwritten with environment variables,
- changing shared configuration requires modification in every module,
- configuration could be stored in centralized application.

## Centralized configuration

Server dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Client dependency:

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

POLITECHNIKA GDAŃSKA

## Centralized configuration

Server configuration:

```
@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {


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


}
```

## Centralized configuration

Defining configuration localization:

```
spring.profiles.active=native
spring.cloud.config.server.native.search-locations=classpath:/configuration
```

Defining shared configuration in `/configuration/application.properties` or defining particular service configuration in `/configuration/library.properties`.

POLITECHNIKA GDAŃSKA

## Centralized configuration

Configuring client in `bootstrap.properties` (in newer versions in `application.properties`):

```
spring.cloud.config.uri=http://localhost:8084
spring.cloud.config.fail-fast=true
```

Requires service (application) name in `application.properties:

```
spring.application.name=library
```