Pattern: Externalized configuration

Context

An application typically uses one or more infrastructure and 3rd party services. Examples of infrastructure services include: a Service registry (service-registry.html), a message broker and a database server. Examples of 3rd party services include: payment processing, email and messaging, etc.

Problem

How to enable a service to run in multiple environments without modification?

Forces

- A service must be provided with configuration data that tells it how to connect to the external/3rd party services. For example, the database network location and credentials
- A service must run in multiple environments - dev, test, qa, staging, production - without modification and/or recompilation
- Different environments have different instances of the external/3rd party services, e.g. QA database vs. production database, test credit card processing account vs. production credit card processing account

Solution

Externalize all application configuration including the database credentials and network location. On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.

Examples

Spring Boot externalized configuration (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html) reads values from a variety of sources including operating system environment variables, property files and command line arguments. These values are available within the Spring application context.

`RegistrationServiceProxy` from the Microservices Example application (https://github.com/cer/microservices-examples) is an example of a component, which is written in Scala, is configured with the variable `user_registration_url` :

```

@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate) extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _
```

The `docker-compose.yml` file supplies its value as an operating system environment variable:

```
web:
  image: sb_web
  ports:
    - "8080:8080"
  links:
    - eureka
  environment:
    USER_REGISTRATION_URL: http://REGISTRATION-SERVICE/user (http://REGISTRATION-SERVICE/user)
```

`REGISTRATION-SERVICE` is the logical name of the service. It is resolved using Client-side discovery (client-side-discovery.html).

Resulting Context

This pattern has the following benefits:

- The application runs in multiple environments without modification and/or recompilation

There are the following issues with this pattern:

- How to ensure that when an application is deployed the supplied configuration matches what is expected?

Related patterns

- The service discovery patterns, Server-side service discovery (server-side-discovery.html) and Client-side service discovery (client-side-discovery.html), solve the related problem of how a service knows the network location of other application services

Tweet

Follow @MicroSvcArch