Configuration

Table of Contents

As developers we usually try to make the software we create easy to configure so that it can be used in any environment.

However building Microservices on the cloud changes our approach to configuration since you no longer need to configure the following differently for each environment:

- file locations: since docker images have the same file system layout whatever the environment
- service locations: instead use service discovery
- secrets like usernames and passwords: instead use kubernetes secrets

So it turns out that many Microservices no longer need any configuration values which change for different environments thanks to the above!

This helps reduce possible errors; it means most configuration for your Microservice can be included inside the docker image and tested in every environment in your CI / CD Pipeline

However for times when you really need to configure things differently on a per environment/deployment basis then here are the main approaches:

Include configuration inside the docker image

The simplest thing to do with configuration is just include it inside your docker image. e.g. if you are building a Spring Boot Microservice then just include the `application.properties` or `application.yml` file inside your `src/main/resources` folder so that it gets included into your jar and the docker image. This then means that all your configuration gets tested as part of your CI / CD Pipeline

This assumes the configuration will be the same in each environment due to the above (using service discovery and kubernetes secrets).

For any environment specific values then you could specify environment variables in your kubernetes resource. e.g. your Kubernetes Deployments can include `env` values in the pod template's container section.

Though its good practice maximise the amount of immutable arfefacts you have (e.g. docker images and kubernetes resources) and to minimise the amount of artifacts you need to modify for each environment. So for per-environment configuration we recommend one of the following approaches:

ConfigMap

Kubernetes supports a resource called ConfigMap which is used to store **environment specific** configuration values - all other configuration values should be inside the docker image as described above.

You can then expose the `ConfigMap` resources either as volume mounts or as environment variables.

The spring-boot-webmvc quickstart uses a ConfigMap to store the application.properties file which is then mounted into each Pod via the deployment.yml. Then Spring Boot will load the `application.properties` entry from this ConfigMap on startup!

Another approach if you are using Spring is the ConfigMap based PropertySource which loads environment specific configurations from a ConfigMap.

Both these approaches have a similar effect; the former reuses kubernetes manifest to wire things up; the latter uses a new maven dependency and Java code to achieve a similar effect.

Git

One of the downsides of using `ConfigMap` as described above is that there's no history or change tracking; there's just the latest version of the ConfigMap. For complex configuration its very useful to have a changelog so you can see who changed what configuration values when so that when things start to go wrong you can easily revert changes or see the history.

So you can store your environment specific configuration in a git repository (maybe using a different branch or repo for each environment) then you can mount the git repository as a volume in your Microservice docker container via a `gitRepo` volume using a specific git revision.

You can then use the gitcontroller Microservice to watch the Kubernetes Deployments with one or more `gitRepo` volumes and it then watches for changes in the associated git repositories and branches.

When there are changes in a configuration git repository the `gitcontroller` will perform a rolling upgrade of the Kubernetes Deployments to use the new configuration git revision; or rollback. The rolling upgrade policy (e.g. speed and number of concurrent pods which update and so forth) is all specified by your rolling update configuration in the Deployment specification.

Here is an example of how to add a `gitRepo` volume to your application; in this case a spring boot application to load the `application.properties` file from a git repository.

You can either run `gitcontroller` as a Microservice in your namespace or you can use the `gitcontroller` binary at any time or as part of your CI / CD Pipeline process.

Choosing the right approach

We recommend you try to keep the environment specific configuration down to a minimum as that increases the amount of resuable immuable artifacts (docker images and kubernetes resources) that can be used without modification in any environment and promotes confidence and testing of

those artifacts across your CI / CD Pipeline.

So if you can avoid any configuration that is environment specific (thanks to service discovery and kubernetes secrets) then just including configuration inside your docker image makes the most sense.

When you absolutely must have some environment specific configuration which doesn't relate to service discovery or secrets then our recommendation is:

- if you want history and to be able to see who changes what and when so you can easily audit or revert changes then use `git` and a `gitRepo` volume to mount the configuration into your docker container. Otherwise use `ConfigMap`