Externalized Application Configuration Management

Description

This paper discusses the need for externalised configuration management and offers various design consideration and mechanisms to achieve a truly decoupled code and configuration setup for micro application deployment and management.

Business Need:

An application's configuration is anything that is most likely to vary between deployment environments or over a period of time. e.g: database or dependent service endpoints can change across environments. Number of threads in an application specific thread pool can also change over time depending on the outcome of performance tests and integration testing. It is very important to have a decoupled executable code and configuration setup so that we don't have to do a physical deployment to see effects of configuration changes.

Just like code has its own release ideology, configuration management also needs its own release and deployment methodology. Code and configuration most of the time go along together but there need to be provisions in place to move then independent of each other when needed. One of the most common use cases pertaining to release management is to release code from a lower environment to production after ratification of a battery of tests and fitment benchmarks. But like a fact of life, deployments fail in production and most of the time the issues are functional but cause of a faulty configuration. To fix the issue, a whole build is needed again on the code base, despite the fact that nothing was changed in the code at all, just a few configuration properties were tweaked. Doing this too often, leads to voluntary and temporary sidelining of the release process to promote the fixed build to production without having to make a stop at every environment in the midway. This is an enterprise micro services anti pattern.

Delays to deployment due to coupled code and configuration leads to higher TAT on build release management and this lowers the Time to market. This creates friction with other development ideologies like small feature builds pushed to production every 1 or 2 days. Because the fix for the current build needs to be promoted first, all subsequent planned builds / releases need to be halted to allow the fix to reach up and stabilise first. On the other hand some teams also choose the rollback path, which again leads to waste of LOE. Rollbacking a build because of a configuration issue is something you should never want to do.

One of the biggest problems with ratification of configuration management is that, when ever we test something in any environment, what we actually test is the behaviour of the code against the associated configuration in "**that"** environment. It is almost always, never possible to test the interaction of the same code in an environment against configuration of "**another"**environment. e.g: you can almost never fully gauge how your code will behave in prod by testing it in dev environment, by using the same code and prod's configuration. Since promotion of the release across environments almost never changes the executable code containers / deployment, only the configuration changes (and data in the data stores in that environment) so there is no proper way to test this fully, without making a mirror of an environment and testing there first.

The scope of this document is to provide a recommendation for a mechanism to:

- Adjust configurations in an environment quickly without redeploying the application build again.
- Facilitate configuration management, as separate to deployable and executable application code.
- Facilitate configuration rollback and revisions right at the configuration source without any additional tweaks.
- Facilitate security for sensitive configuration secrets and properties and how to deliver them to consumers of such configurations.
- Explore capabilities of the configuration delivery to stream arbitrary files that supplement the base application configurations e.g: RSA key pairs

Core Features

Considerations to keep in mind when choosing a mechanism for externalized configuration management:

1. **Configuration Store:**
    - Keep configuration in GIT as a separate folder / repository / branch
    - Keep it in a datastore e.g: DB, CMS, Cache store (Redis, Hazelcast)
    - Keep sensitive configurations and keys in vault and leverage it as an in memory config store at the application level.
2. **Configuration Topology:**
    - Flat configuration files or rows in GIT or DB or Key-value based data stores like Hazelcast?
    - Hierarchical breakdown:
        - Keep common / less frequently mutating properties in one place and env specifics in env specific / profile based files / branches.
        - Channel based segregation. e.g: Separate for mobile and web based on env specific / profile based files / branches.
        - Employ a mix of the above. Environment and channel specific files or tables or branches.
3. **Configuration Placement at Runtime:**
    - As an infrastructure piece / part
        - Kubernetes config maps
        - Kubernetes secrets
        - Environment properties
        - Sidecar / Init container
        - vault secrets store
    - As a centralized and externalized configuration delivery service app
        - a service endpoint that provides normalized configuration over HTTP/S in JSON or YAML transport format.
4. **Configuration Management:**
    - Owned By Dev, Managed by Ops / Infrastructure teams with Dev team oversight.
    - Owned By Dev, Managed by Dev as an Ops process pipeline with Ops / Infrastructure team's oversight.
5. **Configuration Delivery Mechanisms:**
    a. Push based: The configuration service or tool decide if the configuration changed and pushes the configuration to targets?
    b. Pull based: The targets pool or stream configuration and decide themselves if the change has happened or not?
6. **Consuming Configuration Changes in Targets:**
    a. Refresh a running application:

        i. Has limited capability to mutate aspects of application e.g: can't mutate bootstrapping constructs of an application like connection pool.
        ii. Configurations reflect fairly quickly as the application is largely not rebooted.
    b. Reload / restart the application context:
        i. Allows full control over all types of configuration driven changes to app.
        ii. Sequencing and flow control locks/mechanisms are needed to prevent all the containers of the cluster, from reloading / restarting at the same time.
7. **Polyglot development Support:**
    a. When using a central config server, the approach should be polyglot development friendly, when it comes to configuration delivery.
    b. All the config consumers should be able to access the configuration endpoints via their own well known language specific REST / HTTP / MQ frameworks.
    c. All the config consumers should be able to consume the configuration payload via their own well known frameworks for YAML, JSON or XML.
    d. The config server should allow the consumers to handle hierarchical configuration and configuration payload merges and overrides on their own. This is particularly easy to do with minimal JSON/XML/YAML code on most modern development languages that bode well with JSON/XML/YAML.

Non Functional requirements

Following are some of the most important NFRs when choosing externalized configuration delivery.

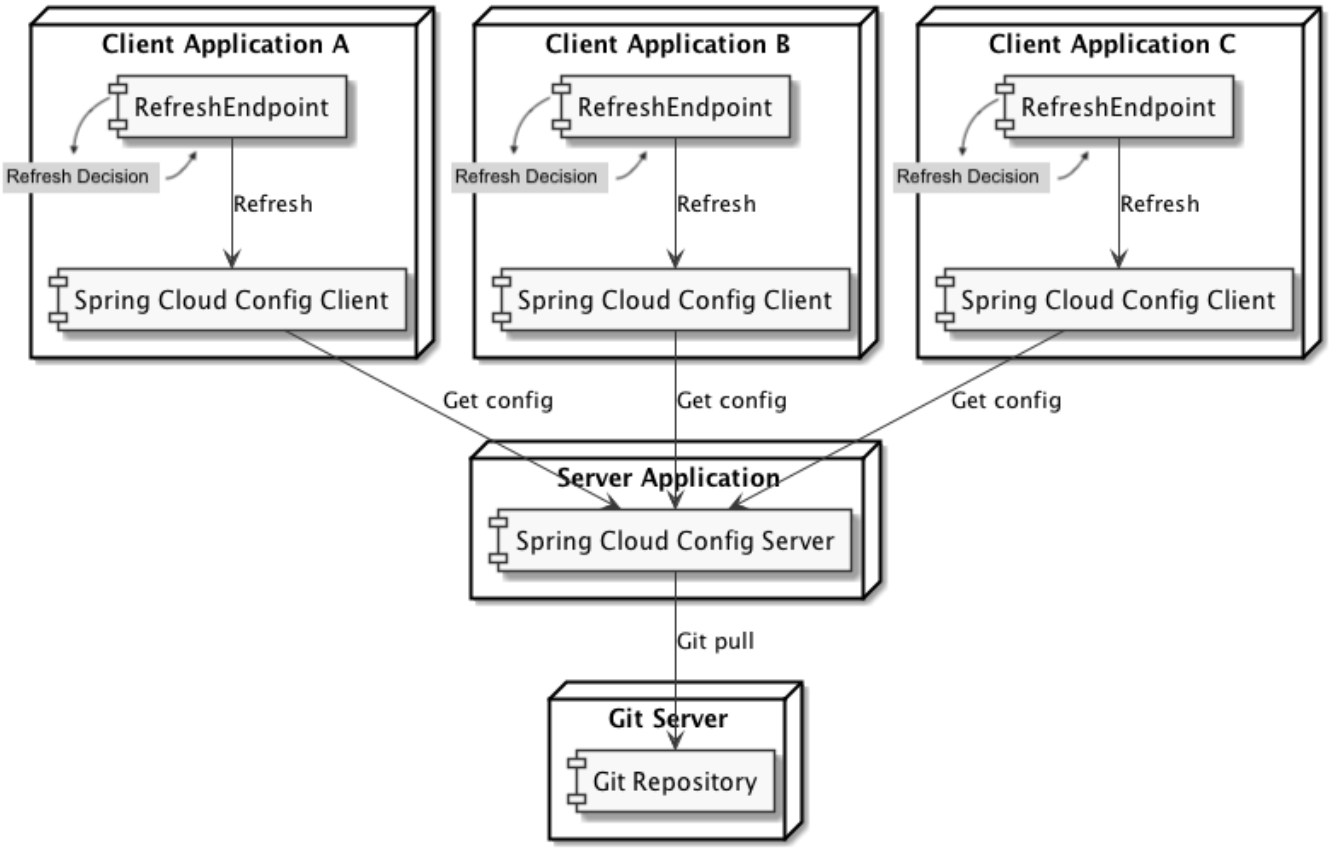
1. **Configuration delivery TAT:**
    - Sooner the better, but not too fast.
    - To add provisions for common human error, the configuration should not be tunnelled to targets so fast that a change done twice actually makes two reloads / refreshes. This is more problematic when synchronizing doing target reloads / restarts.
2. **Provisions for rollback:**
    - use subsequent configuration rollouts to actually revert the configuration changes.
    - employ health check tracking in the configuration delivery tool to automatically revert the config changes to last know good state.
3. **Manage Memento States of configurations changes:**
    - Use a config store that can keep revisions of changes e.g: GIT
    - Manually track, last 'x' number of configuration changes revisions.
4. **Security:**
    - Application configurations are always stored at rest, not just in the configuration store but also in application containers (when using env variables or kube config maps).
    - Sensitive properties should always be stored in a secrets store and never in plain text.
    - If possible encrypt the sensitive property values themselves before storing in the secrets store.
    - Using a siloed security approach, harden sensitive config security by using at-least 3 different keys to encrypt the property values. Even if we lose a key, the attacker cannot decrypt all the props with it.
5. **High availability (HA) provisions for configuration delivery:**
    a. When using infrastructure based configuration managed, HA provisions for configuration delivery are largely not required.
    b. When using application service based delivery mechanisms, a centralized configuration service needs to be clustered properly to avoid making it a point of failure for apps **that need configuration for their bootstrap**. It is important to understand, that target services that are already up and running are not impacted because they do not need to keep querying the config service to function properly. The impact on them, is that they might miss out on their targeted config changes because the config service is not available to tunnel then.
6. **Provisions for delivery of arbitrary text / files as a stream:**
    a. Should allow for provisions to deliver arbitrary text and binary formats as a stream to consumers, instead of JSON or YAML or XML.
    b. Some situational cases don't use property values as configuration input but need a whole file as a source.
        i. For example, env specific log4j configuration that can be reloaded automatically production. At normal times the production app runs with ERROR logging level, but sometimes for a brief instant, it needs to be run in INFO or DEBUG levels to diagnose some issues. Allowing logging configuration reload and dynamic delivery without restarting or redeploying the app is particularly useful in production.
        ii. Another common use case is Delivery of key chains or public key certificates or trust stores to the consumer apps. These are also non JSON/YAML based artefacts that need to be provided to the application. When used properly, having public keys at rest does not lead to security leaks (though its possible to generate private key hash fingerprints from public keys using modern back walking algorithms), but having them only in memory facilitates for a more hardened system.
    c. Kubernetes also offers mechanisms to store file contents as values in config maps or secrets, but these are always at rest. So with a little RBAC magic it is possible to see their contents and abuse them further.

Solution Approaches

Centralized Configuration delivery service app

Pull based approach

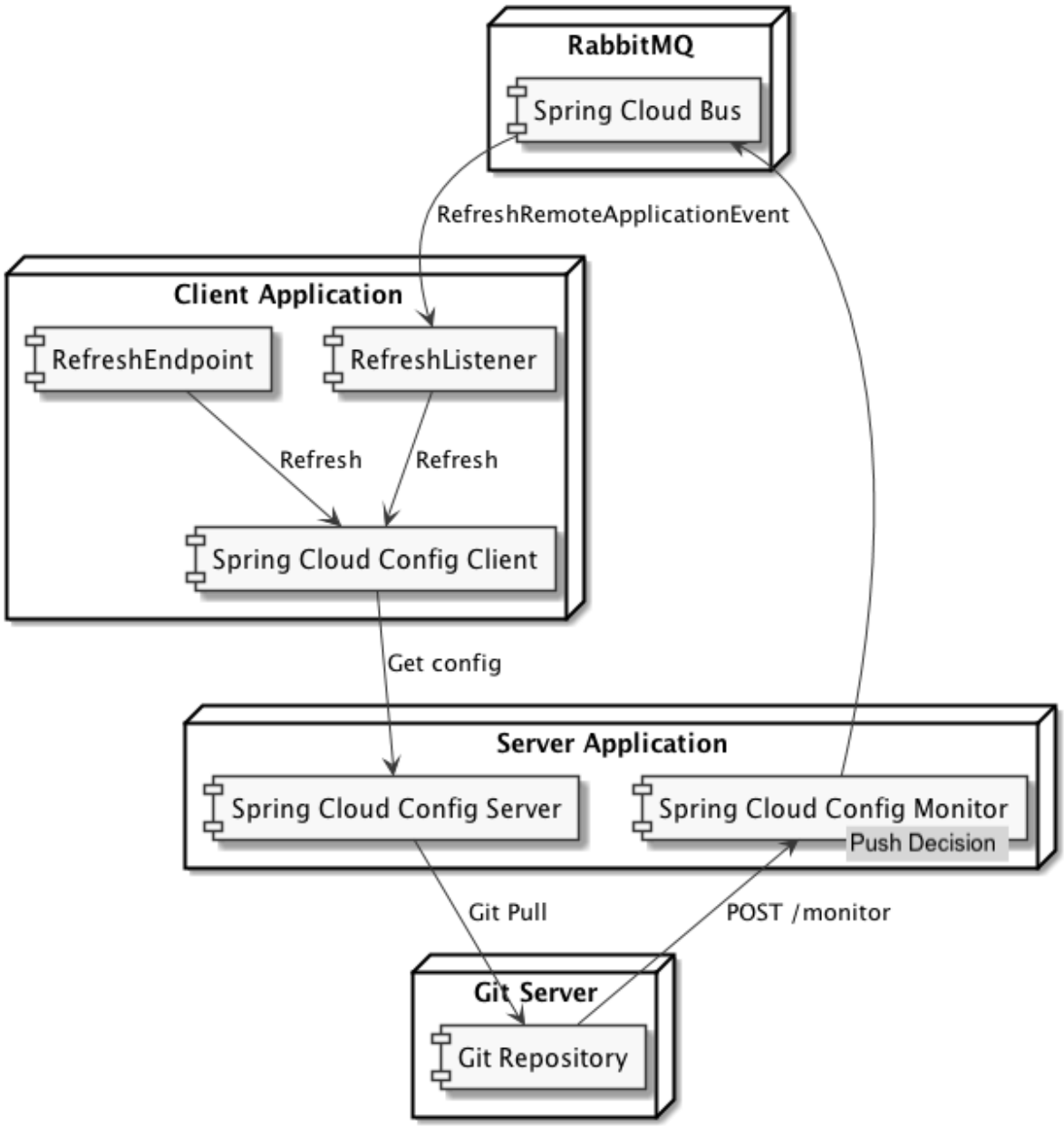
Following diagram shows a high level view of how spring cloud config server and client can be used to externalise and centralise configuration management. This approach is used for first time configuration pull during application bootstrap and can also be used for regular configuration state sync using a scheduled pool based mechanism.



- The configuration management architecture shows the pull based configuration delivery, using a GIT backed config store.
- Client applications poll the config service for their respective config endpoints and maintain a local decision log, wether they want to refresh / reload the app or not.
    - This is largely done using a scheduled job, that run for e.g. every 2 minutes.
    - This also provides a cushion for human error in configuration updates and or GIT configuration revision rollbacks if needed.
- For simpler use cases, local decision logs can maintain an SHA hash of the configuration properties or data and use that to ascertain wether configuration changed or not.
- For more complex cases, individual parts of the config data can be used to have multiple hashes that can provide a more fine grained control of the restart decision.
- One of the major concerns about this approach is that it needs continuous pull from the client app to keep track of the config changes.
    - While internal network bandwidth usage is largely free in many cloud providers, but it does incur CPU cycles across all running instances of client apps.

Push based approach

Another approach as show below, uses a push based delivery from config server using a suitable MOM. This approach is mostly used for regular configuration sync.



- The architecture diagram uses RabbitMQ, but any streaming topic based MOM solution can be leveraged in its place e.g: Kafka or Google PubSub.
- Major salient point of this architecture approach is that we move all the functional moving parts related to scheduled polling fo configuration store, configuration state change hashes and configuration normalisation etc. from the target client apps to config monitor piece inside the config server.
- It is also possible to use Spring cloud bus to notify us of changes in a repository through GIT webhooks.
    - We can configure the webhook through the provider's user interface as a URL and a set of events that can help trigger a refresh.
- Since the config server is no longer stateless because of the presence of Config Monitor, so some mechanism is needed to synchronise the "config change" decision state across running instances of config server.
- There is a drastic reduction of network bandwidth and CPU cycles incurred by client apps, since they don't need to query the config server all the time.
- Since this approach adds another moving piece, i.e. the MOM, clustering considerations are needed to prevent the MOM platform from become a single point of failure for configuration delivery.

Provisions for delivery of sensitive configuration values and collated alternate/plain formats

Spring cloud config provides mechanisms to leverage vault as a config store backend.

- We can provide multiple search locations to look for in a particular config store. e.g using the below setup, config server would query git and search properties from the "**default.env.instance**" (or '**live**' as default value if this property is not set) location add merge it hierarchically with the properties in "**default.env.instance"** (or 'live' as default value if this property is not set)**/<application name>** location.

- We can also configure multiple config backends in Spring cloud config server as shown below to pull configuration form both GIT and vault and normalise / merge them before delivery to target. Using the same approach we can use multiple git repositories, or multiple folders in same git repository or totally disparate config stores (e.g: GIT and DB) together to serve the collated and normalised configuration to the client application.
- Keeping the order precedence higher allows the target client app's Spring context to override these secret based property values during property name resolution. The following diagram shows the hierarchial list of `PropertySource` objects that make up the OMS application's `Environment` object at runtime.
- By default, Spring cloud config translates the properties to JSON format for the environment endpoints as this is the preferred approach for Spring applications, but we can consume the same data as YAML or Java properties by adding a suffix (".yml", ".yaml" or ".properties") to the resource path. This is ideal for non java based application e.g: a react based front end application.

Securing endpoints / access to the Configuration Store

When using Spring Cloud Config as Centralized Configuration Store

We can secure the Config Server in any way that makes sense to us (from physical network security to OAuth2 based bearer tokens). This is easy to do because Spring Cloud Config is pluggable with Spring boot and Spring security. To use the default Spring Boot based HTTP Basic security, we only need to include Spring Security on the classpath and `setting spring.security.user` & `password` properties in the Config Server.

When using Kubernetes as Centralized Configuration Store

If the inclination is to let infrastructure or Ops teams own configuration management, then configurations can be stored in K8S config maps and delivered to application containers via K8S Deployment's env properties. One important drawback of this approach is that application containers can no longer do light weight refresh anymore. All changes to application config maps data needs a redeployment of the K8S deployment pods.

- While Kubernetes deployment mechanisms allow us to prevent downtime by using "create then destroy" approach for application container pods, but it is important to understand that this is a hard reboot fo the containers and not a soft Spring container context reload.
    - Since the trigger for the restart/reload comes from outside the application (triggered manually by the Ops engineer or using a scheduled Kubernetes job) the application mayn't be Abel to shutdown gracefully one its own.
    - Provisions need to be added in place with for e.g. shutdown hooks and graceful thread pool shutdowns to prevent data loss (specifically for distributed batch based applications).
    - Nevertheless, the problem can only be minimised by extending the value of graceful termination period in Kubernetes deployment / statefulset manifests, working in conjunction with application level provisions, but it cannot be entirely eliminated.
    - The reason primarily is because it is not possible to ignore or influence a kubernetes deployment patch or termination event from inside the pod containers.
- With this approach, configuration management becomes a responsibility of the Operations team.
    - But they almost always lack the overlap and the knowledge needed to understand the intricacies of each property and its influence on client application behaviour.
    - While on one hand this process offers a gated provision to control what is changed and when and offers more traceability to process owners, the physical owners of those Kubernetes config maps are not developers anymore.
    - So it creates another layer of impractical delegation where the owner doesn't have any idea of the impact of the change.
    - This becomes more problematic for situations where finding the value for a property is not an excel science (for example thread pool size, connection pool size, order of fallback / failover targets for a dependent third part service) but instead needs a few iterations of "change monitor update" cycles by the developer to decide on an optimal value. Recollect that you cannot possibly test the change in a lower environment effectively unless you have an exact mirror of the environment in question. This makes the TAT, considerably more than what might have been possible if a centralized configuration service was used instead.
- For singular configurations, e.g DB endpoints, PubSub topic names etc, the TAT and process is very smooth and streamlined.
- Multiple config maps can be added to K8S deployment but managing their hierarchial merge is the responsibility of the user who is doing the changes.
- Spring Cloud Kubernetes provides ways to alleviate some of the pains. It provide Spring Cloud common interfaces implementations to consume Kubernetes native services, including Config maps. The main objective is to facilitate the integration of Spring Cloud/Spring Boot applications running inside Kubernetes directly with Kubernetes resources without the needs to add Operations boilerplate in between.
    - Spring Cloud Kubernetes Config submodule of this framework, allows the client application to read key/value properties directly from K8S config map and construct Environment PropertyPlaceholders from it.
    - There are utility modules to track config map changes and reload or refresh the application on changes. Low level refresh decision based on hashes or selective properties still rests within the application design domain.
    - Multiple config maps can be combined together to achieve a hierarchical merge of the configurations. Spring cloud kubernetes does this internally.
- There is no need for a secondary configuration server or service anymore. The client applications liaison directly or indirectly (via an in app Spring cloud kubernetes module dependency) with the infrastructure's kuberenet's side of things to work out their configurations. This eliminates additional single points of failure in configuration management and delivery.

Provisions for delivery of sensitive configuration values

There are primarily two approaches to secure configuration sensitives in infrastructure side of things:

- When using Kubernetes, it offers kube secrets store for storing config values that are sensitive.
    - K8S deployment manifests can map keys from kube secrets as environment variables and kubernetes injects the appropriate values when it creates and deployment pods.
    - By default kube secret values are base 64 encoded, but this can be extended to use custom cryptographic keys provided in the kubernetes controller plane.
    - Alternatively, custom encryption mechanisms can be used to encrypt the values before being pushed into kube secrets and then decrypted and stored "in memory" (e.g. using Spring's application listeners in Java based apps) in client apps when they bootstrap.
    - They security aspect of this mechanism is debatable and largely resolves to personal preferences.

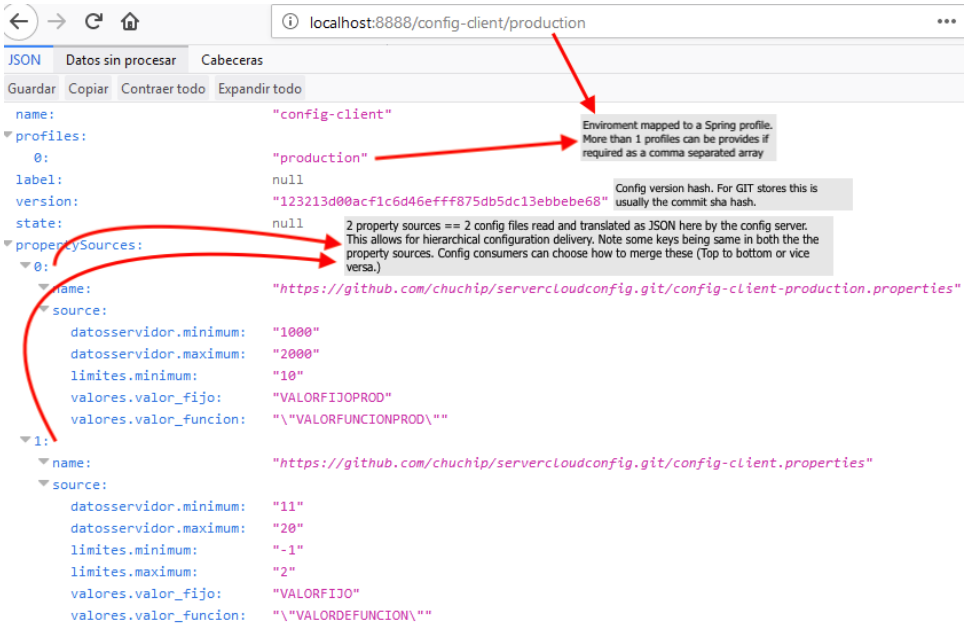
        - Since all secrets stay inside the kube cluster / namespace it can be argued that the principal of siloed security is largely violated if the kube cluster is compromised. This can be alleviated by using custom encryption of values and decrypting them when the application bootstraps.
        - Another view point asserts that its safe enough, because the kubernetes cluster is already hardened and secure using RBAC and bearer tokens on all major Kubernetes Engine provides in cloud platforms. So a compromize of secrets is actually an indicator of a much larger security problem rather than a simple application level secrets leak.
- When using a hybrid approach, Hashicorp vault can also be used as a secrets backend store for kubernetes or spring cloud config.
    - By leveraging "Vault K8S Agent" we can plug a secrets consumer into the deployments so that they can reach out to vault for configuration pulls.
    - Vault K8S agent runs as an agent daemon inside the pod / container. It can be configured to run in a sidecar or as an init container as the design need may require.
    - Vault K8S agent provides a number of different helper features:
        - Automatic authentication using Kube RBAC tokens
        - Secure delivery/storage of vault's auth tokens and secrets.
        - Lifecycle management of these tokens and their lease renewal.
    - This approach is more complex to implement as it needs a proper RBAC control on kube control plane first to make vault work seamlessly.
    - For a centralized app abed approach, Spring cloud Vault is available as an extension to transparently and declaratively talk to vault and pull the secrets at runtime. Unlike, kubernetes option variants, these secrets are never stores at rest and thus are not available for direct introspection unless the attacker has access to JVM app's memory heap dumps and knows exactly where to look for the values in the sea of byte arrays.

Polyglot development support

When using Spring Cloud Config as Centralized Configuration Store

With Spring cloud config we have primarily three options to facilitate this. In either option, reloading / refreshing configuration on the client app is the responsibility of the client application itself. Spring cloud config server cannot and know and dose not act as a mediator in this regard. All it does it deliver the configuration (Pull or Push based). How the apps consume this payload, is not its responsibility.

1. Pull based approach: This approach is used during first time boot up and regular config sync. Spring cloud config server offers the configuration payload as an array of JSON or YAML "propertySources" markup, each representing a Spring Environment object instance. Non-JVM apps don't need to bother with the theatrics and can just parse and consume the JSON as they feel like. The following images show an example of how the configuration payload for the config server looks like the following. Depending on how the refresh / reload is implemented in the client apps, they need to pool the config server to access config changes and act accordingly.



2. Push based approach: Second option is the push based one as outlined earlier. Applications bootstrap suing the pull abse approach and then sue the push based approach to consume the configuration mutations over time. Spring cloud config pushes the actual payload or an event trigger to an MQ message stream and the Non-JVM consumer clients can use this event as a trigger to pull or reload their configuration.
3. Third option is to leverage Spring cloud Netflix Sidecar. Spring cloud sidecar is not specific to Cloud config, but also offers capabilities to use other Netflix ecosystem projects like, Eureka Service Discover, Hystrix CB and Ribbon Client side LB etc in non-JVM applications, so its a complete package for apps looking to leverage other centralized capabilities that mixed match well together (e.g: zone or region aware client side load balancing using Ribbon and Eureka, while falling back based on Circuit breaker trips.). For apps looking for simpler approaches, approach 1 above is as simple as it gets.

When using Kubernetes as Configuration Store

Kubernetes doesn't offer any config delivery mechanisms of its own. Its really up to the clients to facilitate how to read these configMaps and secrets and how to consume them.

- Most common mechanism is to read these values and store them as env variables in kubernetes deployment YAMLs.
    - Kubernetes offers extension to mount secrets / config maps as a volume / env keys array.
    - Reloading of the application is not possible in most cases on dynamic property changes. The deployment / pods need to be restarted for property change to take effect.
- Most languages have kubernetes client libraries that can be leveraged to talk to kubernetes config maps and secrets and pull and consume the values.
    - The biggest drawback of this approach is that while the configuration store and delivery is centralized and standardised, the config consumption falters. Additional process management provisions maybe needed to standardised frameworks to avoid reinventing the wheel across product /project teams.

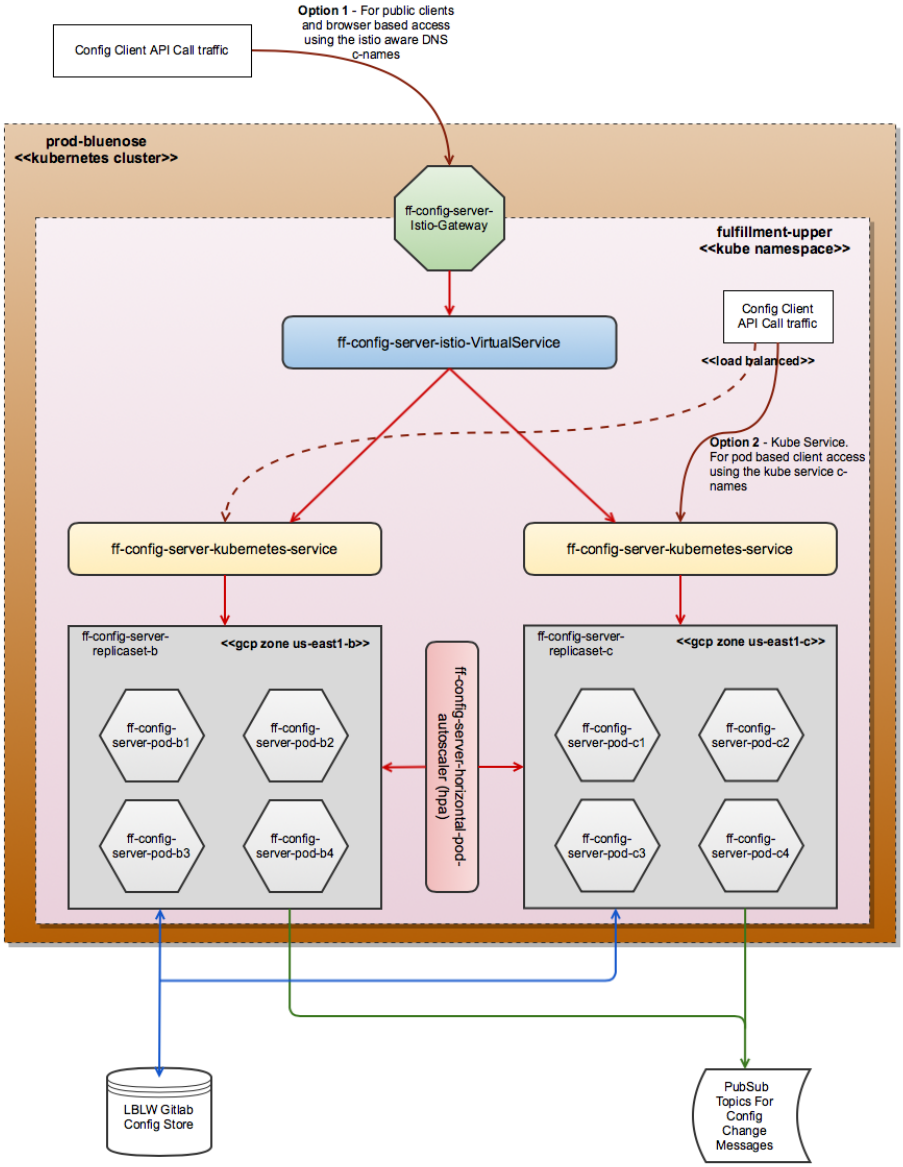
Ensuring High Availability of Spring Based Config Server - Deployment model

From a developer's perspective, a standalone Spring cloud config server is just a dependency in a spring boot application. The config server can be compiled, build and deployed as a standalone entity and uses its own configuration properties. OOTB Spring cloud config server doesn't require any coding to configure or run, just a few set of properties to point it to the respective GIT config stores and file / vault backends and let it know our configuration layout topology that we intend to follow.

For CI and CD, the considerations depend on which code management approach does the project leverage. It is recommended to host the config server in its own separate repository with its own CI and CD pipeline. Deployment can be done using standard deployment toolchains leveraging Helm or kubernetes.

Following diagram shows a HA setup for config server in a kubernetes cluster. At least 2 deployments of config server ensure redundancy and failover for infrastructure failure scenarios. The deployment model offers two options for configuration consumption for its clients (application based and normal browser / REST based).

- Option 1: allows provisions for public access to config server endpoints. This maybe is needed by developers to view configuration for debugging or validation, where the config server REST endpoints are accessed form outside the kubernetes cluster e.g: from a local browser.
- Option 2: allows provisions for internal access to config server endpoints e.g: from pod based clients that want to consume configuration for their bootstrap needs or refresh cycles. Since these clients would reside inside the same kubernetes cluster, even in the same namespace as of the config server, so they can directly leverage the config server specific kubernetes services instead of going to istio gateway.



Configuration Management Reference Matrix

Configuration Management Approach	Functional Capabilities					NFRs		
	Configuration Store Options	Configuration Topology Management	Configuration Ownership Process	Delivery Mechanisms	Consuming Configuration Changes in Targets	Configuration delivery TAT	Rollbacks	Revisions
Centralized Configuration Management - Pull based delivery	Any store as supported by Spring Cloud Config (GIT, SVN, DB, REDIS etc.)	User defined. Some e.g: • Env per GIT branch, Channel per file • Env per GIT branch, Channel per spring profile based file. • Env per channel as a GIT branch. • Env per application as a GIT branch	Dev Owned. Dev Managed. Ops oversight for infrastructural issues.	Pull based. Target Client apps use a scheduled pooling approach to pull from a central Config Server cluster.	Target Client apps decide how to infer if the configuration change is relevant for them. They also own the decision wether they want to do a restart / reload or just a light weight refresh of @Value based property injections.	Fairly fast considering network lags and keeping considerations for human errors in configuration changes.	Target Client app managed, unless the client shard reloaded (JVM is shutdown and rebooted). Client considerations needed for persistence of rollback memento state.	When rollbacks are done using a subsequent pull for a previous revision, GIT's revision tracking capabilities can be used to manage configuration file revisions.

		Hierarchical configuration delivery using common configuration files merged with Spring Profile based files.						
Centralized Configuration Management - Push based delivery	Any store as supported by Spring Cloud Config (GIT, SVN, DB, REDIS etc.)	User defined. Hierarchical configuration delivery using common configuration files merged with Spring Profile based files.	Dev Owned. Dev Managed. Ops oversight for infrastructural issues.	Push based. Config Server hosts a monitoring module that tracks the GIT store for potential changes and publishes a "update config" message to relevant target topics for target client apps to consume.	Config Server hosts a monitoring module that decides how to infer if the configuration change is relevant for a respective target client.	Fairly fast considering network lags and keeping considerations for human errors in configuration changes.	Config Server's monitoring module managed. Monitering module hosts mechanisms to persist rollback memento state.	When rollbacks are done using a subsequent pull for a previous revision, GIT's revision tracking capabilities can be used to manage configuration file revisions.
Kubernetes Config Maps based approach. NO config Server.	Key value pairs stored in kube config maps.	User defined flat topology. Hierarchical configuration delivery required manual decisions and more than one config map setup.	Ops Owned. Ops Managed. Dev oversight restricted to providing information about config value change.	• When managed by Ops, mapped as env variables to K8S deployments / workloads. Settable during container creation but immutable after that. • When using Spring Cloud Kubernetes, Spring managed workers track config map keys and state changes.	• When managed by Ops, values are consumed on first bootstrap of app and are immutable after the container starts. A change to config needs a hard app restart. • When using Spring Cloud Kubernetes, values are consumed on first bootstrap of app and reloads and refreshes can be managed	Fast. All configurations are available locally as env variables or read directly from config map by Spring Cloud Kubernetes.	No Provisions for rollbacks. A rollback == reverting the config to their old values in config maps.	Configuration needs to be managed externally e.g: in GIT to allow provisions for revisions. Config maps themselves are not revision capable without additional considerations e.g. keeping old copies of config maps in the cluster to allow for rollbacks.

				Allows for refresh / reload with some customizations.	within the app JVM using config map tracking.			

Recommendation:

Based on the comparative analysis of benefits and developer control of different approach as discussed above, following are the recommendations:

- **Configuration delivery approach:** Use Spring cloud config with GCP Cloud PubSub Push based approach.
- **Configuration Store:**
  - Use a separate Git repository as a central configuration store for all the microservcies and applications that need to externalise their configuration.
  - Rollback of configuration equates to a merge request revert in GIT.
  - Configuration revisions if required, can be mapped to GIT commit ids and tags.
- **Configuration Topology:**
  - For environments: demarcate each environment as a GIT branch in the config store repository, to allow them to be managed and maintained independently.
  - For multiple deployments of same application image in one enviroment (to facilitate application redundancy at an operational infrastructure level): use separate application name folders to maintain configuration in the GIT branch of that environment.
- **Configuration ownership:** Developers own the configuration as part of their DevRun responsibilities.
  - They own configuration release management
  - They own configuration rollback on functional failures.
  - They own configuration based feature release toggles.
  - They also own release note tagging for configuration changes, if any.
- **Consuming Configuration Changes in Targets:**
  - Use Spring Cloud config client with Hazelcast to provide true distributed and rolling configuration updates to running pod sets / containers of a specific application deployment.
  - Pod sets can choose to restart or refresh themselves based on their inference of the type of configuration changes.
- **Securing Sensitive configurations:** Use Spring cloud vault integration with Spring cloud config server, to deliver secret configurations as add ons to the base applications that is streamed to the clients.
- **Securing Config Server:** Use Basic or Digest authentication for a start and later on, if required, move to oAuth based authentication for config server.
- **Ensure HA of config Server:** Deploy at least 2 deployments (2 replica sets) of config server, each in a separate kubernetes zone, to allow maximum uptime for the config server (as depicted here). Let kube internal clients leverage kubernetes services for config endpoints and let public clients use the config server istio gateway.

Links & References:

- Spring Cloud Config v2.2.0.x
- Spring Cloud Kubernetes v2.1.0.x
- Spring Cloud Vault v2.2.0.x
- Kubernetes Secrets
- Kubernetes Config Maps
- Vault K8S Agent
- Managing Secrets in Kubernetes using Vault
- Kubernetes Client Libraries