Configuring Services Using Distributed Configuration

Outline

Configuration in a distributed system

Configuration with Spring Cloud Config

- Using config client and server
- Backend stores
- Updating Configuration & @RefreshScope
- Storing and retrieving sensitive configuration

What's so different about managing configuration in a cloud-native application?

Configuration: Non-distributed vs Distributed

From one or a handful of configuration files

To ...

Many, many configuration files

Configuration Management tooling to the rescue, **right**?

e.g. Chef/Puppet/Ansible

It'll work ... **but** it's not ideal in the cloud

Issues with Typical Configuration Management

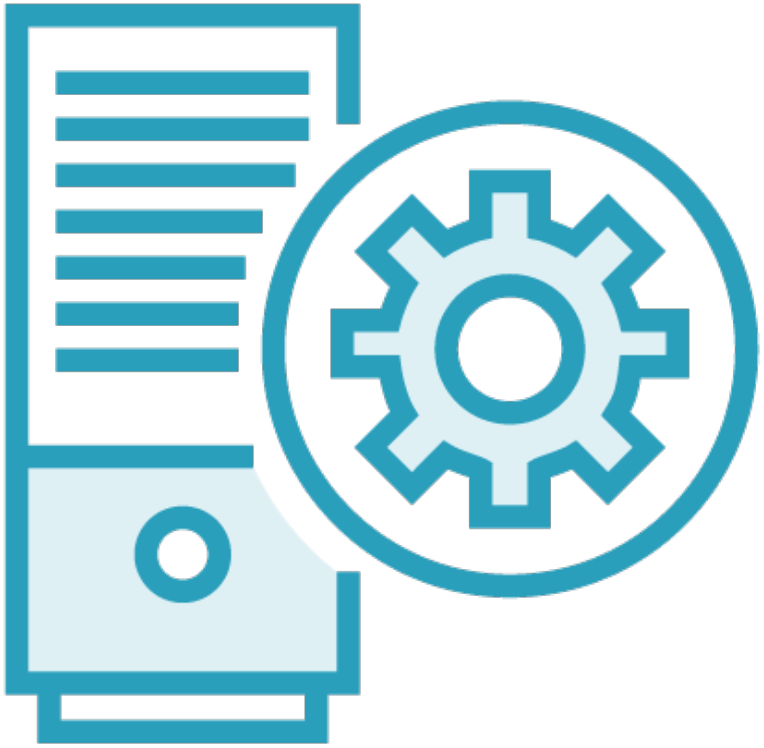
Deployment-oriented

Push-based is usually not dynamic enough

Pull-based adds latency with temporal polling

Q: If configuration management tooling doesn't solve our problem, what does?

A: Configuration Server



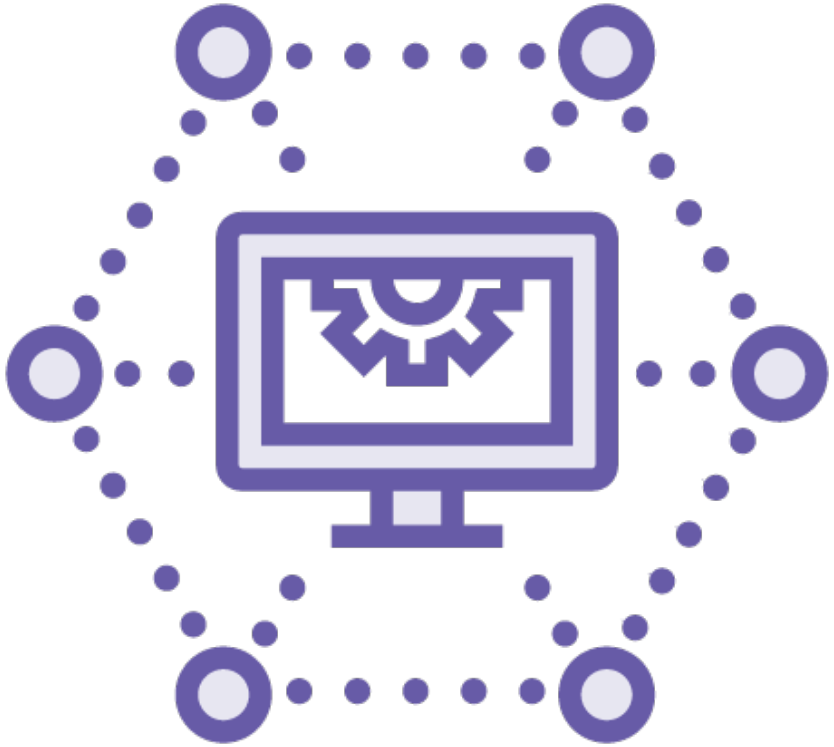
Application Configuration Server

- Dedicated, dynamic, centralized key/value store (may be distributed)
- Authoritative source
- Auditing
- Versioning
- Cryptography support

Managing Application Configuration with Spring Cloud

manage **config with:**

- Spring Cloud Consul
- Spring Cloud Zookeeper
- Spring Cloud Config



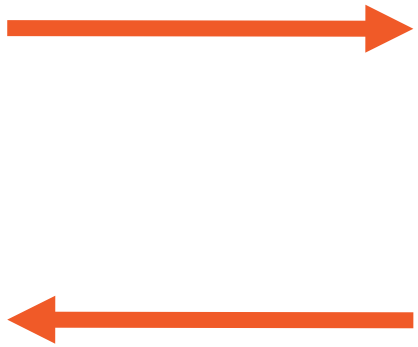
Spring Cloud Config

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system.

- Reference documentation

Integration with Spring Applications

Config Client



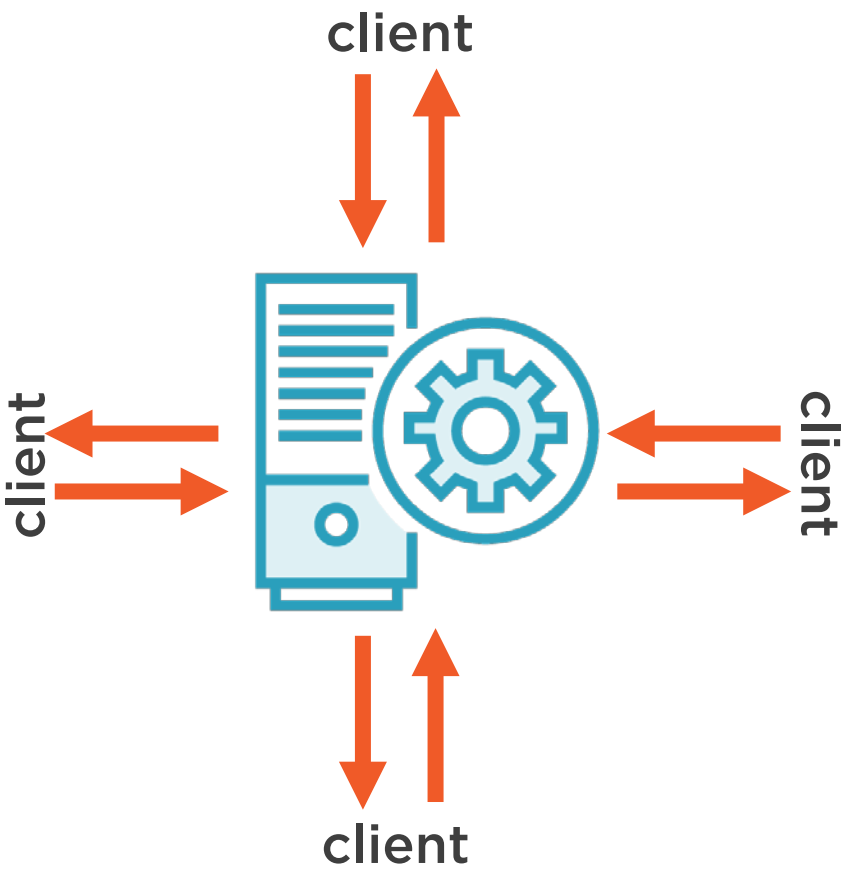
Config Server

- **Embedded in application**
- **Spring** `Environment` **abstraction**
  - `e.g. @Inject Environment`

- **Standalone (can be embedded)**
- **Spring** `PropertySource` **abstraction**
  - `e.g. classpath:file.properties`

Spring Cloud Config Server

Config Server



HTTP REST access

Output formats

- JSON (default)
- Properties
- YAML

Backend stores

- Git (default)
- SVN
- Filesystem

Configuration scopes

Using Spring Cloud Config Server

pom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Camden.SR2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Using Spring Cloud Config Server

pom.xml

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

Using Spring Cloud Config Server

Create a folder to store configuration	**(optional) Add a `properties` or `yml` file with a named `application`**	**Add `properties` or `yml` files named `{application}-{profile}`**
`git init`	`git add` `git commit`	**(optional) Setup remote git repository and** `git push`

** Note: git commands are abbreviated*

Using Spring Cloud Config Server

application.properties

```
server.port=8888
spring.cloud.config.server.git.uri=<uri_to_git_repo>
```

OR

application.yml

```
server:
  port: 8888
spring:
   cloud:
      config:
         server:
            git:
               uri: <uri_to_git_repo>
```

Using Spring Cloud Config Server

Application.java

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

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

** Tip: add eureka client dependencies, service-url configuration, and `@EnableDiscoveryClient` to make the config server discoverable!*

Don’t forget to secure your Config Server!

Easy to configure Spring Security

Spring Cloud Config Server: REST Endpoints

REST Endpoint Parameters

`{application}`

maps to
`spring.application.name`
on client

`{profile}`

maps to
`spring.profiles.active`
on client

`{label}`

server side feature to
refer to set of config
files by name

REST Endpoints

Endpoint

GET `/{application}/{profile}[/{label}]`

Example

- `/myapp/dev/master`
- `/myapp/prod/v2`
- `/myapp/default`

REST Endpoints

Endpoint

GET `/{application}-{profile}.(yml | properties)`

Example

- `/myapp-dev.yml`
- `/myapp-prod.properties`
- `/myapp-default.properties`

REST Endpoints

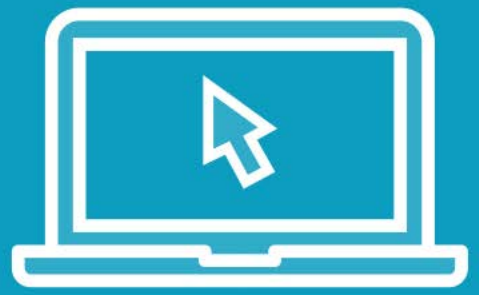
Endpoint

GET `/{label}/{application}-{profile}.(yml | properties)`

Example

- `/master/myapp-dev.yml`
- `/v2/myapp-prod.properties`
- `/master/myapp-default.properties`

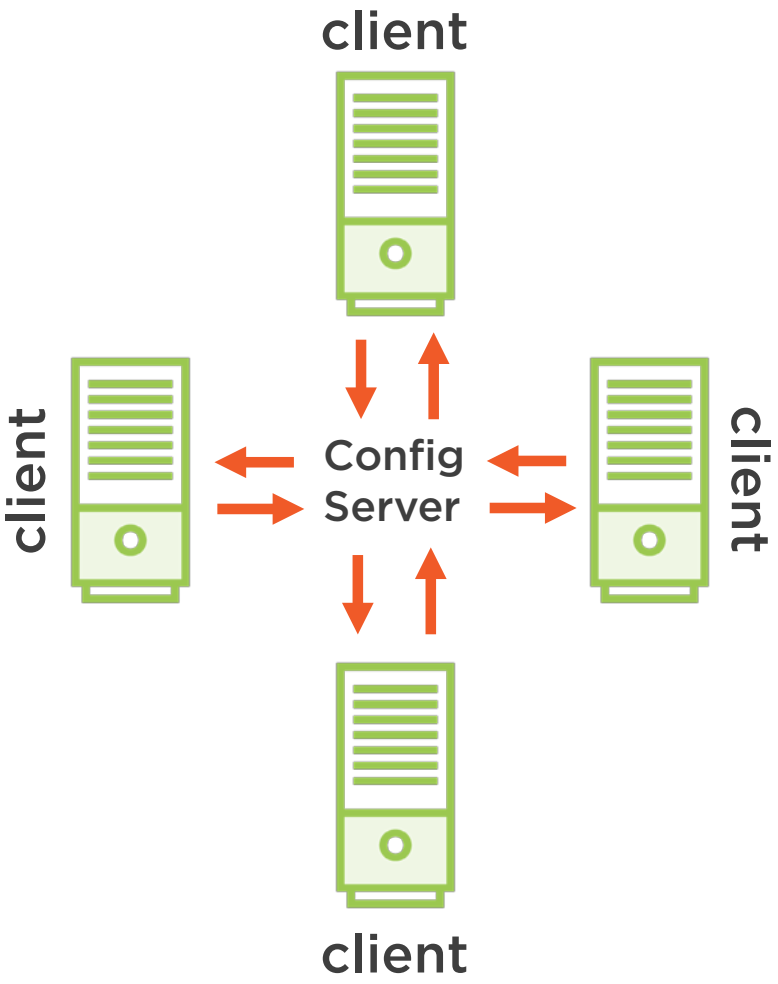
Demo



Creating and starting a config server

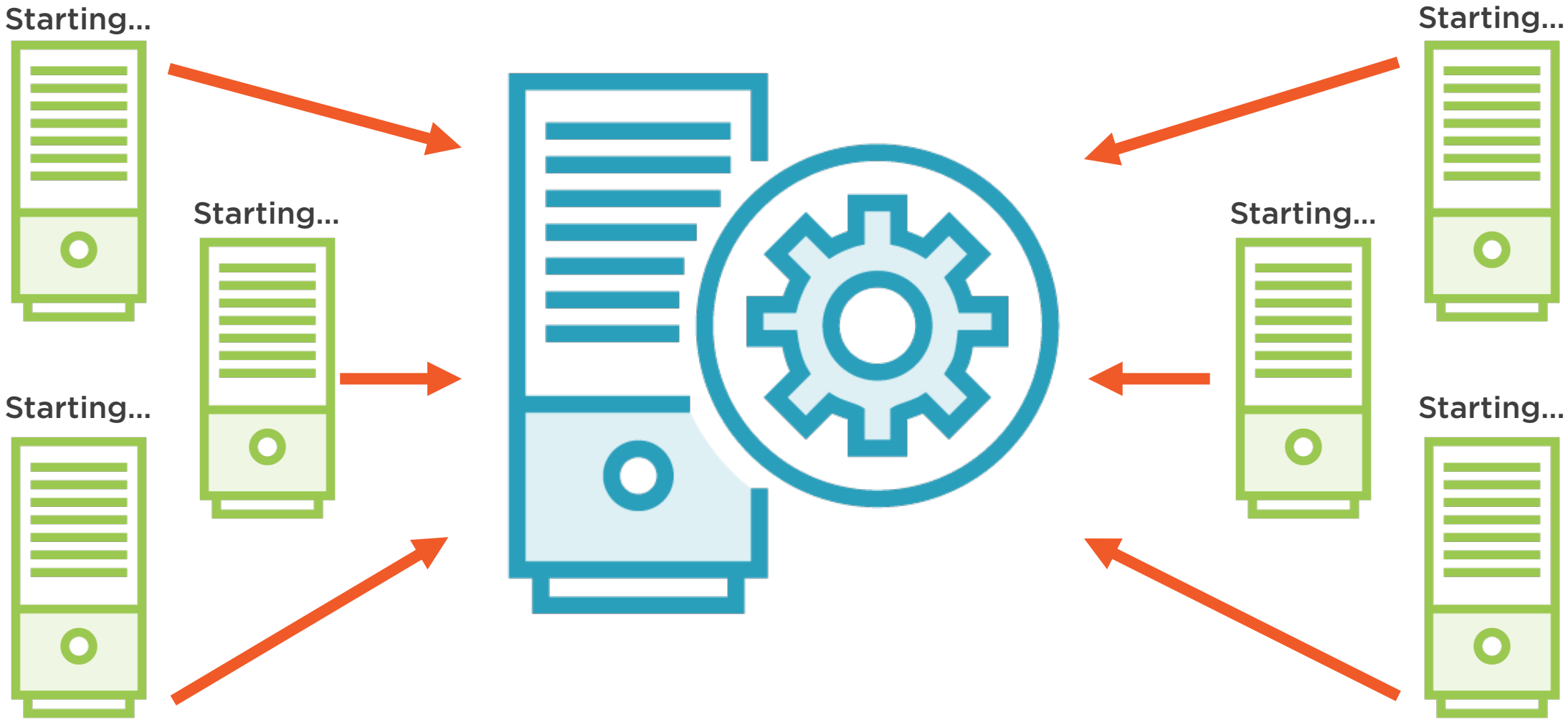
Spring Cloud Config Client

Config Client



Bootstrap, & fetch app configuration

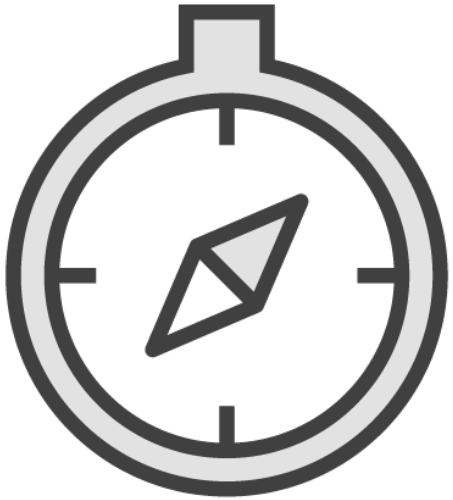
Fetching Configuration: Application Startup



Bootstrapping with `bootstrap.properties` or `bootstrap.yml`

Config first

Specify the location of the config server



Discovery first

Discover the location of the config server

Using Spring Cloud Config Client

pom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Camden.SR2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Using Spring Cloud Config Client

pom.xml

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

Using Spring Cloud Config Client: Config First

bootstrap.properties

```
spring.application.name=<your_app_name>
spring.cloud.config.uri=http://localhost:8888/
```

OR

bootstrap.yml

```
spring:
   application:
      name: <your_app_name>
   cloud:
      config:
         uri: http://localhost:8888/
```

Using Spring Cloud Config Client: Discovery First

bootstrap.properties

```
spring.application.name=<your_app_name>
spring.cloud.config.discovery.enabled=true
```

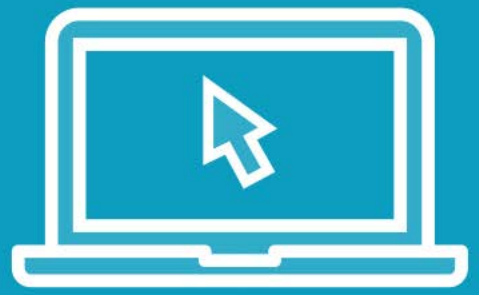
OR

bootstrap.yml

```
spring:
   application:
      name: <your_app_name>
   cloud:
       discovery:
          enabled: true
```

** Note: don't forget to add eureka client dependencies, service-url configuration, and `@EnableDiscoveryClient`*

Demo



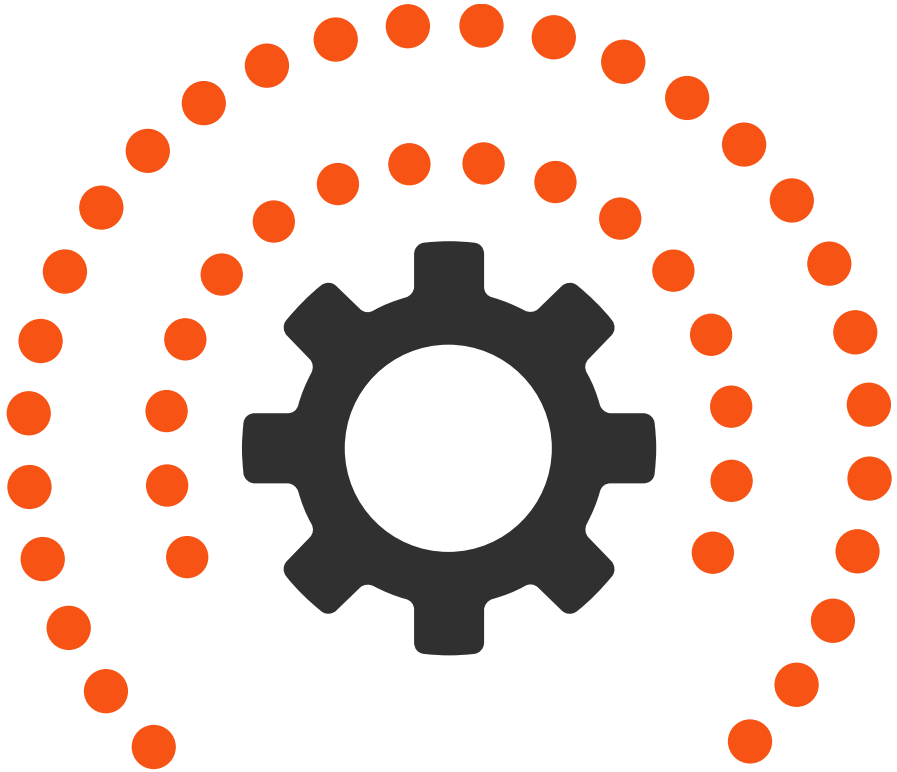
Bootstrap a service use the config client

Updating Configuration at Runtime

Refresh

@ConfigurationProperties

Update logging levels



```
git add .
git commit -m “made some configuration changes”
git push origin head
```

Step One: Configuration Changes

Edit & save configuration file(s)

Commit and/or push changes to a VCS

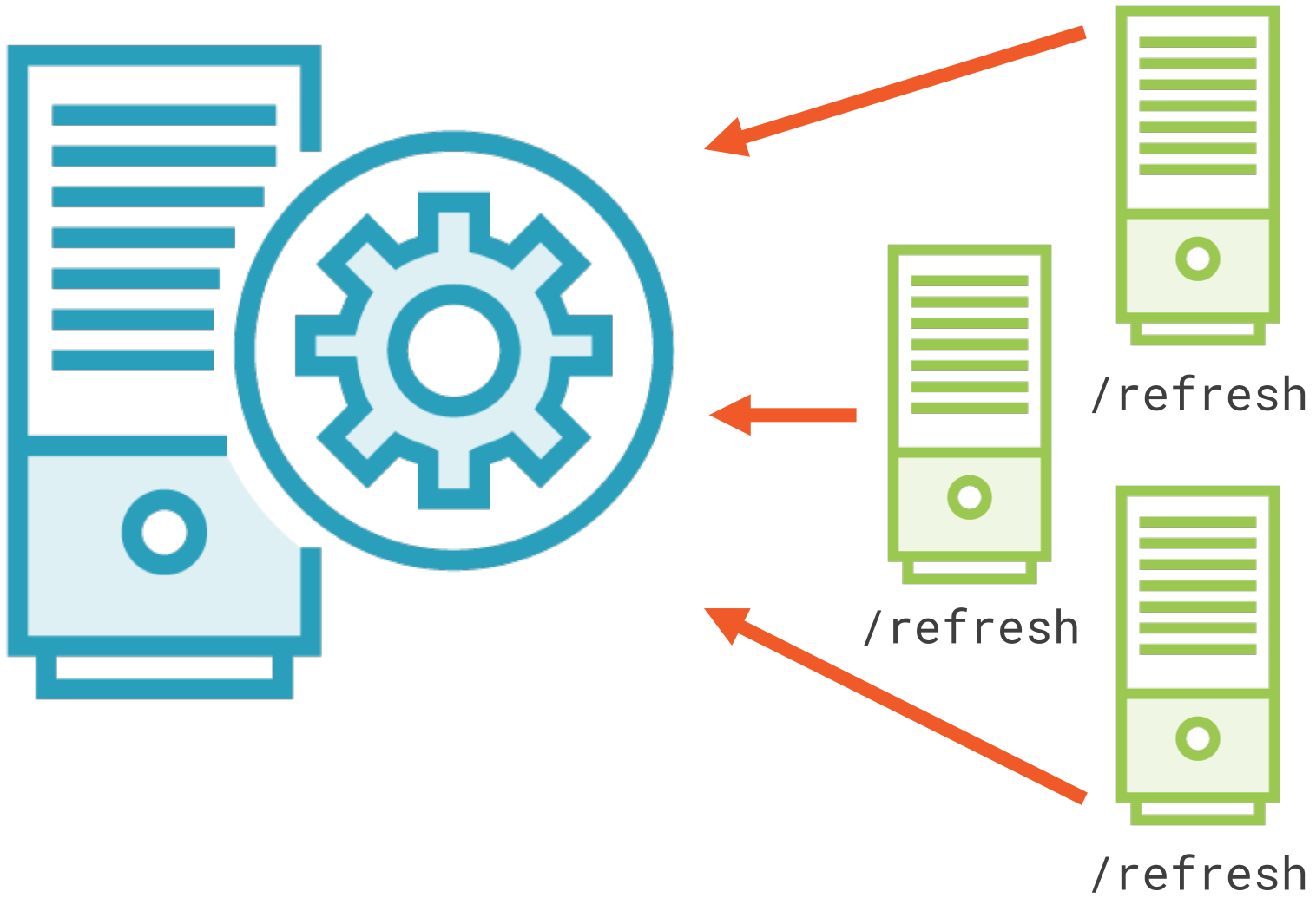
Step Two: Notify Application(s) to Refresh Configuration

`/refresh`
with
`spring-boot-actuator`

Fetching Configuration: Explicit Refresh

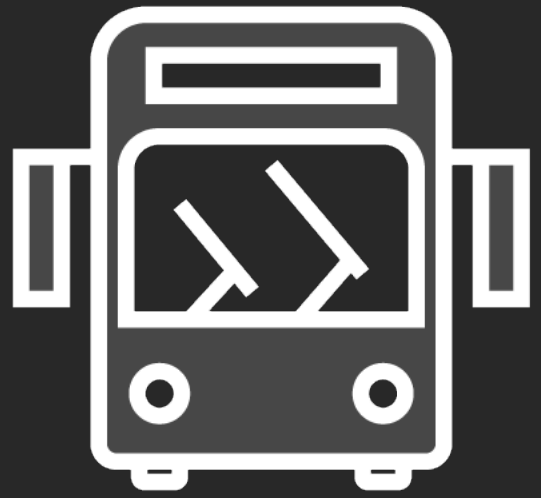
Manual

via each `/refresh` **endpoint**



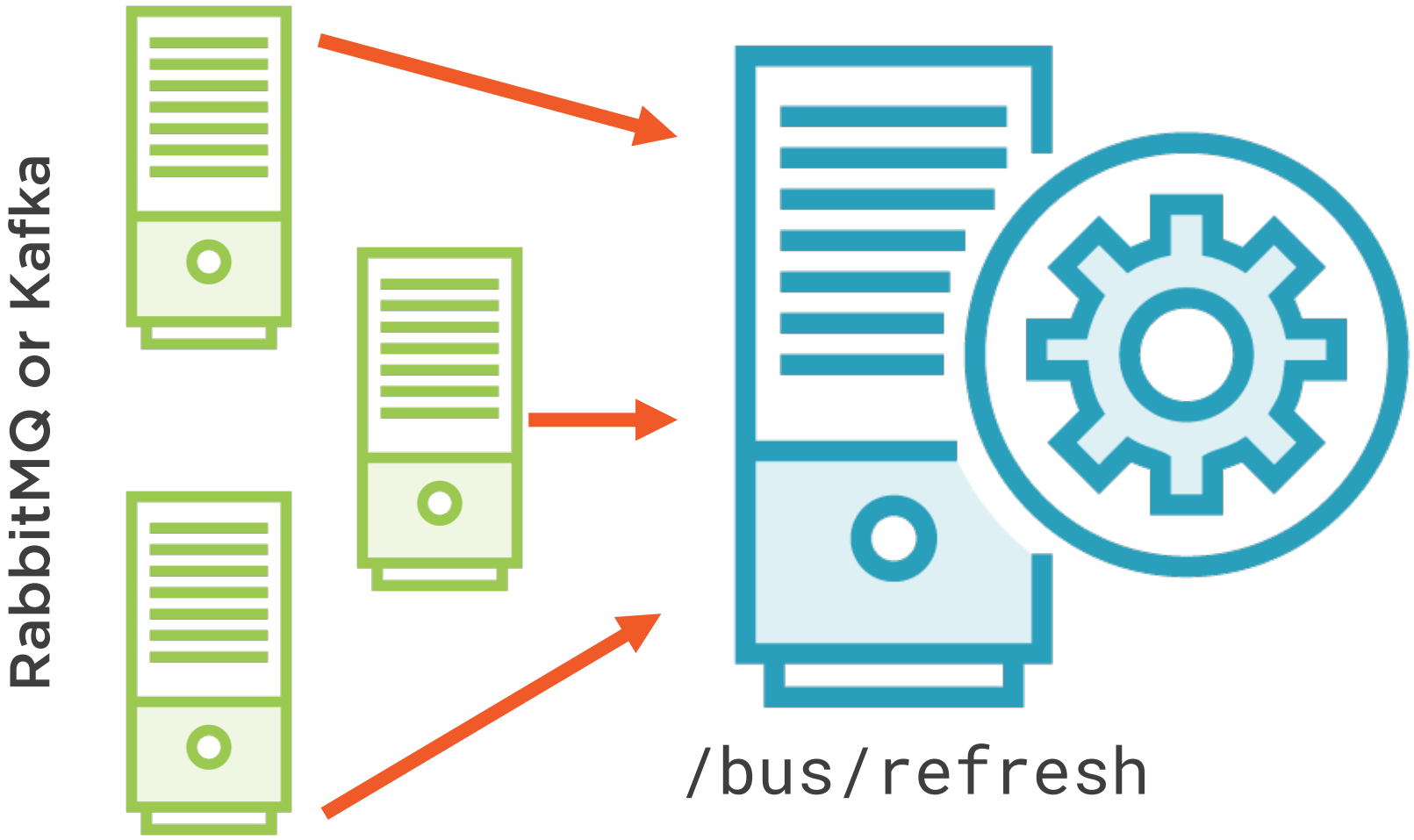
Step Two: Notify Application(s) to Refresh Configuration

`/refresh`
with
`spring-boot-actuator`



`/bus/refresh`
with
`spring-cloud-bus`

Fetching Configuration: Dynamic Push Refresh

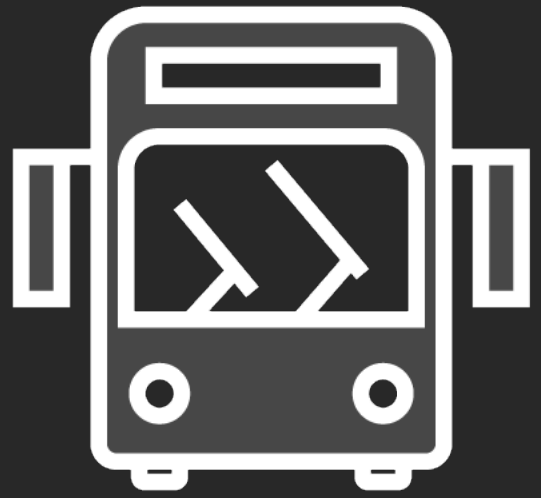


Automatic

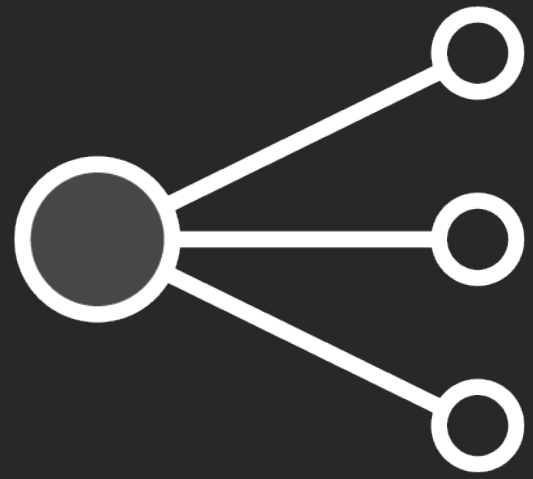
via Spring Cloud Bus broadcasting

Step Two: Notify Application(s) to Refresh Configuration

`/refresh`
with
`spring-boot-actuator`

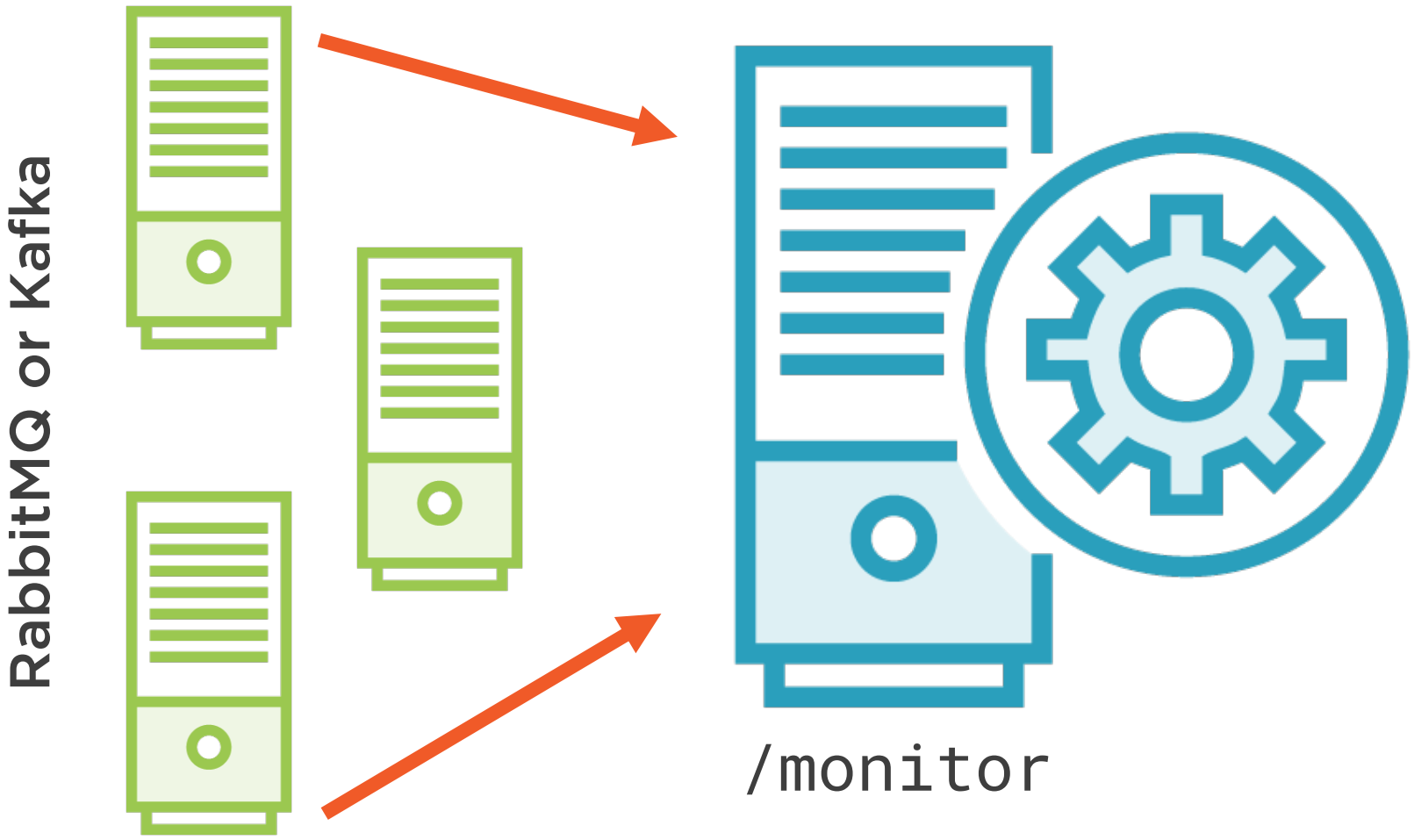


`/bus/refresh`
with
`spring-cloud-bus`



VCS + `/monitor`
with
`spring-cloud-config-monitor &`
`spring-cloud-bus`

Fetching Configuration: Smart Refresh



Automatic & Smart

via
post commit hooks
Spring Cloud Config Monitor
& Spring Cloud Bus
broadcasting

Step Three: Celebrate!

Brag to your colleagues about:

- **Making configuration updates on-the-fly without restarting!**
- **Updating all your apps at once or automatically!**
- **Audit log of all your changes**

Refreshing Configuration: What's Covered and What's Not?

`@ConfigurationProperties`

All logging levels defined by `logging.level.*` are updated

Any `@Bean` or `@Value` that *only* gets its configuration upon initialization

```java
@Configuration
public class SomeConfiguration
{
    @Bean
    public FooService fooService(FooProperties properties) {
        return new FooService(properties.getConfigValue());
    }

}
```

Example: @Bean Will Not See New Config Value After a Refresh

1. **Configuration updates are made**
    - Note that `FooProperties` **is a** `@ConfigurationProperties` class
2. **POST to /refresh**
3. **Result:** `FooService` **will still contain the OLD configuration value**
    - Only gets configuration during initialization

```java
@Configuration
public class SomeConfiguration
{
    @Value("${some.config.value}")
    String configValue;

    @Bean
    public FooService fooService() {
       return new FooService(configValue);
    }
}
```

Example: @Value Will Not See New Config Value After a Refresh

1. **Configuration updates are made**
2. **POST to /refresh**
3. **Result:** `FooService` **will still contain the OLD configuration value**
    - Only gets configuration during initialization

Q: How do I refresh a **`@Bean`** or **`@Value`** that only gets its configuration during initialization?

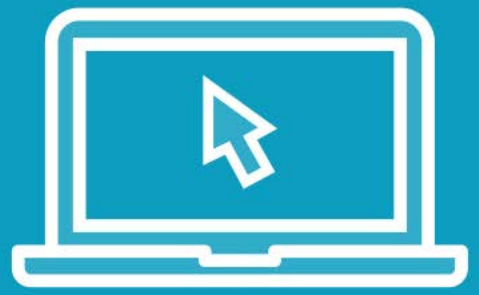
A: **`@RefreshScope`**

```java
@Configuration
public class SomeConfiguration
{
    @Bean
    @RefreshScope
    public FooService fooService(FooProperties properties) {
        return new FooService(properties.getConfigValue());
    }
}
```

Example: Utilizing @RefreshScope

1. **Add the** `@RefreshScope` **annotation to the** `@Bean`
2. **POST to** `/refresh`
3. **Result:** `FooService` **will now contain the NEW configuration value!**
    - `@RefreshScope` tells Spring to please reinitialize this `@Bean`

Demo

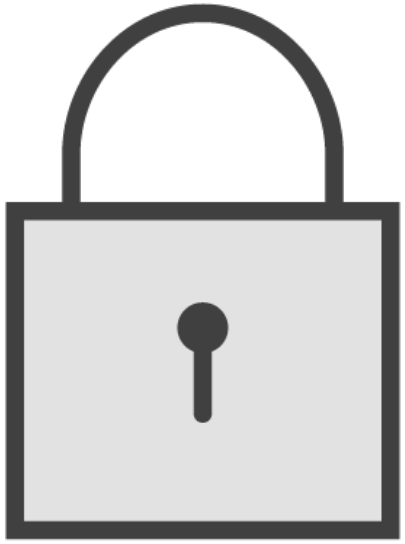


Updating Configuration on-the-fly

Encrypting & Decrypting Configuration

What Features Are Supported?

Encrypted configuration at rest and/or in-flight



An `/encrypt` endpoint to encrypt configuration

A `/decrypt` endpoint to decrypt configuration

Encrypting and decrypting with symmetric or asymmetric keys

What Does Encrypted Configuration Look Like?

application.properties

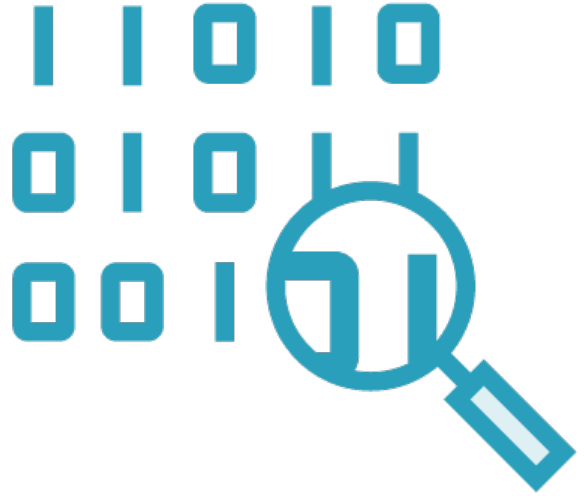
```
my.datasource.username=foobar
my.datasource.password={cipher}ASFIOWRODSKSDFIR32KJL
```

application.yml

```
my:
  datasource:
    username: foobar
    password: '{cipher}ASFIOWRODSKSDFIR32KJL'
```

At What Point Is Configuration Decrypted?

Upon request at the server (*default*)



Locally at the client on response

Configure the Config Server with

```
spring.cloud.config.server.encrypt.enabled=false
```

Before You Begin

You must have the Java Cryptography Extension (JCE) Unlimited Strength for Java 8 installed

Step One: Choose Your Key Type

Symmetric Key

Public Key

Private Key

Asymmetric Key

Step Two (Symmetric): Configure the Config Server

application.properties

```
encrypt.key=<your_super_secret_key>
```

application.yml

```
encrypt:
  key: <your_super_secret_key>
```

Step Two (Asymmetric): Configure the Config Server Option 1

application.properties

```
encrypt.key=<pem_encoded_key_as_text>
```

application.yml

```
encrypt:
  key: <pem_encoded_key_as_text>
```

Step Two (Asymmetric): Configure the Config Server Option 2

application.properties

```
encrypt.keyStore.location=<path_to_keystore>
encrypt.keyStore.password=<keystore_password>
encrypt.keyStore.alias=<key_name_in_keystore>
```

application.yml

```
encrypt:
 keyStore:
  location: <path_to_keystore>
  password: <key_name_in_keystore>
  alias: <key_name_in_keystore>
```

Using REST Endpoints to Encrypt and Decrypt Values

Utility REST Endpoints: Encrypt Values

Endpoint

POST `/encrypt`

Ensure this endpoint is secure!

Example

- **Request:** `/encrypt`
  Data: `<value_to_encrypt>`

Utility REST Endpoints: Decrypt Values

Endpoint

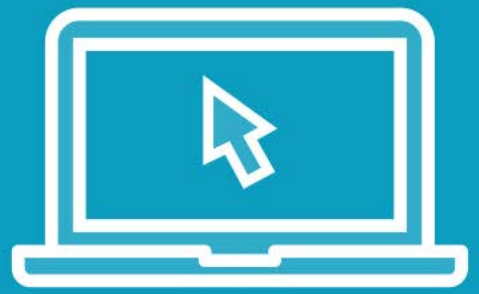
POST `/decrypt`

Ensure this endpoint is secure!

Example

- **Request:** `/decrypt`
  Data: `<value_to_decrypt>`

Demo



Encrypting and decryption configuration values

Summary

The explosion of configuration in the cloud and the need for a config server

Using the Spring Cloud Config Server & Client

Updating configuration at runtime without restarting

Encrypting and decrypting configuration