Spring Cloud Config

Spring Cloud

- **Spring Cloud Contract**: supports and facilitates contract testing
- **Spring Cloud Netflix**: umbrella project offering a number of Netflix services and libraries adapted for Spring:
  - Hystrix, Eureka, Ribbon, Feign, Zuul

- **Spring Cloud Config Server**: configuration as a service
- **Spring Cloud Sleuth**: distributed tracing

About

- Developed by the Spring team

Initially

- Traditionally with Spring applications, configuration is stored with the application, and fetched from the class path
- Projects place configuration in a .properties file under `src/main/resources`

Evolution

Spring also supports reading configuration from:

- Yaml files (.yml)
- Java system properties
- Environment variables

Config Server: Main Concepts

- Externalization of configuration (outside the application)
- Centralization of configuration information for multiple services and environments
- Configuration *as a service*
  i.e. HTTP/REST endpoints for reading application configuration

Design

- Notion of a "backend" where configuration files are stored
- Config server reads configurations from the back-end and exposes HTTP REST endpoints for applications to consume
- Conventions: how files and REST endpoints are named make it easy to expose configuration for different applications in different environments

Backends

- Supports multiple types of backends
- Options include: Git, Subversion, HashiCorp Vault, File System, JDBC
- Also supports a composite of backends

Conventions

- An application's `spring.application.name` is used to identify an application
- The names of files stored in the backend
- The paths for the endpoints exposed by the config server have a specific pattern

Backend file naming

- Default Pattern:

  `{application}-{profile}.[properties|yml]`

- Example:

  ```
  spring.application.name: greeting
  spring.profiles.active: qa
  ```

  Configuration stored in a file: `greeting-qa.yml` (or `greeting-qa.properties`)

NOTE: The pattern can be customized via a configuration property named `searchPaths`

HTTP Service Endpoints

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

- *With the Git backend, {label} maps to a branch name.*
- *{label} is optional, and if not specified, defaults to master*

Configuration hierarchy

- Generic configuration that applies to *all applications* in *all environments* can be placed in a file specially-named `application.yml`

- Generic configuration that applies to *all applications* in a *specific environment* can be placed in a file specially-named `application-{profile}.yml`

Example

Given:

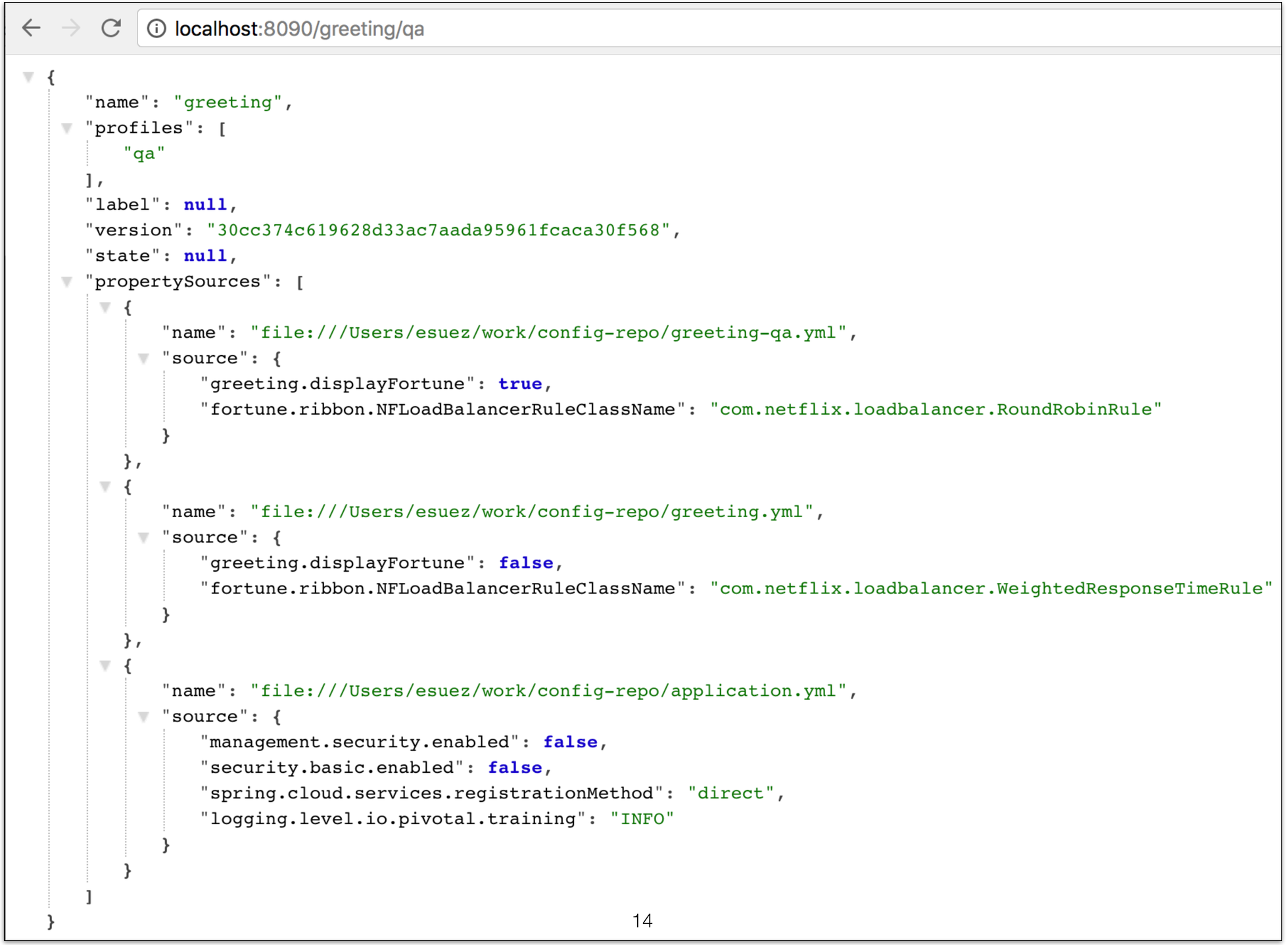
```
spring.application.name: greeting
spring.profiles.active: qa
```

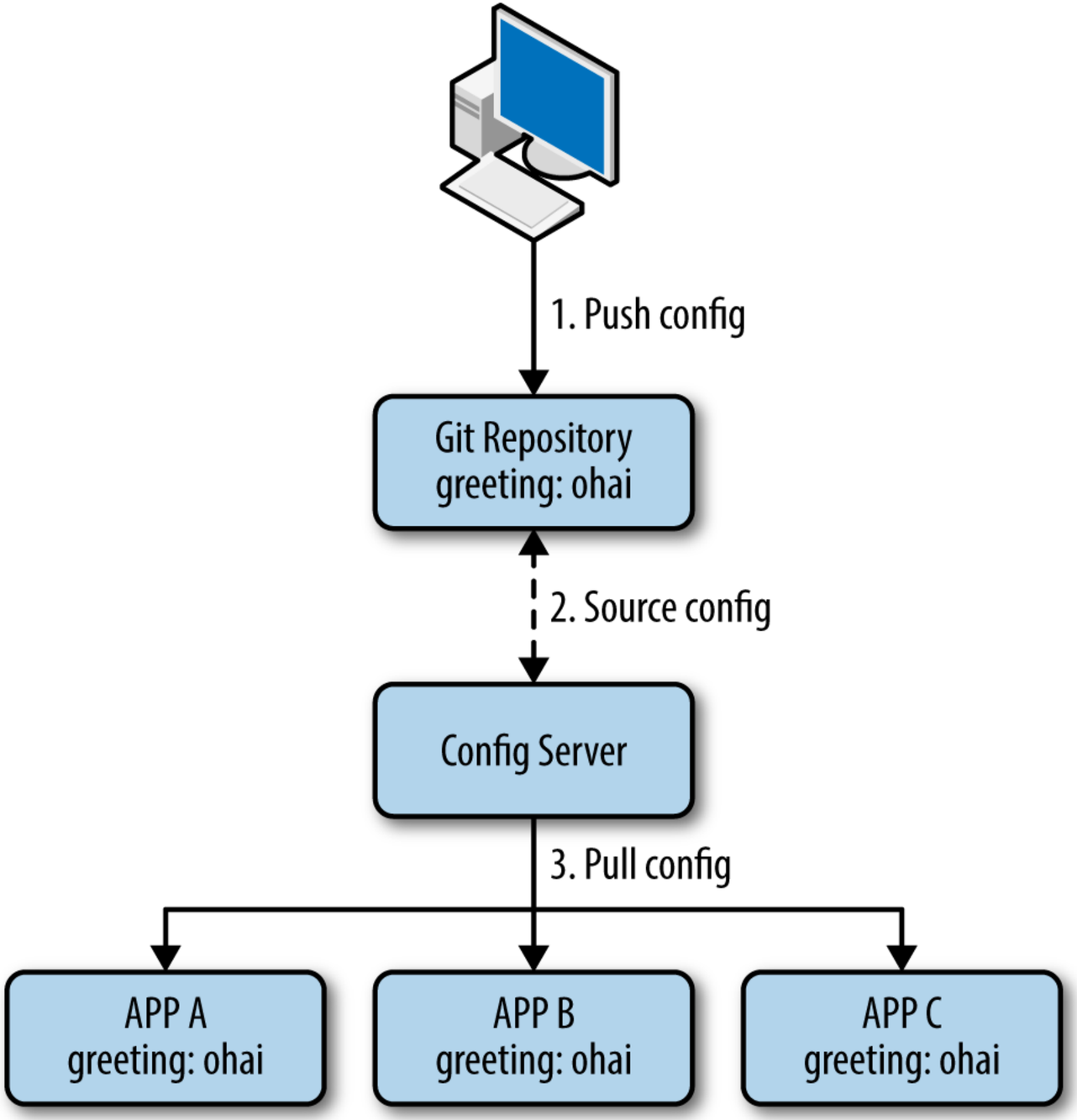
Config server endpoint becomes: `http://{host}:{port}/greeting/qa`

If any of the below files exist, their configuration would be returned as a single json response:

```
application.yml (or .properties)
application-qa.yml
greeting.yml
greeting-qa.yml
```

Configuration properties in the more specifically-named files override a setting in the more general file.


localhost:8090/greeting/qa
{
"name": "greeting",
"profiles": [
"qa"
],
"label": null,
"version": "30cc374c619628d33ac7aada95961fcaca30f568",
"state": null,
"propertySources": [
{
"name": "file:///Users/esuez/work/config-repo/greeting-qa.yml",
"source": {
"greeting.displayFortune": true,
"fortune.ribbon.NFLoadBalancerRuleClassName": "com.netflix.loadbalancer.RoundRobinRule"
}
},
{
"name": "file:///Users/esuez/work/config-repo/greeting.yml",
"source": {
"greeting.displayFortune": false,
"fortune.ribbon.NFLoadBalancerRuleClassName": "com.netflix.loadbalancer.WeightedResponseTimeRule"
}
},
{
"name": "file:///Users/esuez/work/config-repo/application.yml",
"source": {
"management.security.enabled": false,
"security.basic.enabled": false,
"spring.cloud.services.registrationMethod": "direct",
"logging.level.io.pivotal.training": "INFO"
}
}
]
}


1. Push config
Git Repository
greeting: ohai
2. Source config
Config Server
3. Pull config
APP A
greeting: ohai
APP B
greeting: ohai
APP C
greeting: ohai

Setting up the Server

1. Add dependency: `spring-cloud-config-server`

2. Annotate Spring Boot application class with `@EnableConfigServer`

3. Example configuration of git backend to a public repository:

   application.properties:

   ```
   spring.cloud.config.server.git.uri=https://github.com/{username}/config-repo.git
   ```

Many more configuration options exist for the server (consult project reference manual).

Configuring Clients

1. Add dependency: `spring-cloud-starter-config`

2. Set `spring.cloud.config.uri` for the location of the configuration server

3. `spring.application.name` must be set in the config file *bootstrap.yml*, not *application.yml*

4. Alternatively, if config server is registered with Eureka, can lookup config server via eureka by setting `cloud.config.discovery.enabled`

Refreshing the Configuration

- On initialization, the client fetches its configuration from the config server (as a function of the spring application name and the active profiles)

- Spring Boot Actuator dependency provides a /refresh endpoint over HTTP POST, used to instruct the client to re-fetch updates to the configuration from the config server

- Procedure:
  1. Update the configuration in the backend (commit/push the change)
  2. Send an HTTP POST to the application's /refresh endpoint
- The Spring application context will be reloaded with the updated configuration

Which Properties Can be Refreshed?

- logging levels
- Spring Beans of type @ConfigurationProperties
- Any Spring Beans annotated with Spring’s @RefreshScope

Example

```
@SpringBootApplication
@RestController
@RefreshScope // <-- Add RefreshScope annotation
public class ClientApplication {

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

    @Value("${greeting}")
    private String greeting;

    @RequestMapping("/greeting")
    String greeting() {
        return String.format("%s World", greeting);
    }

}
```

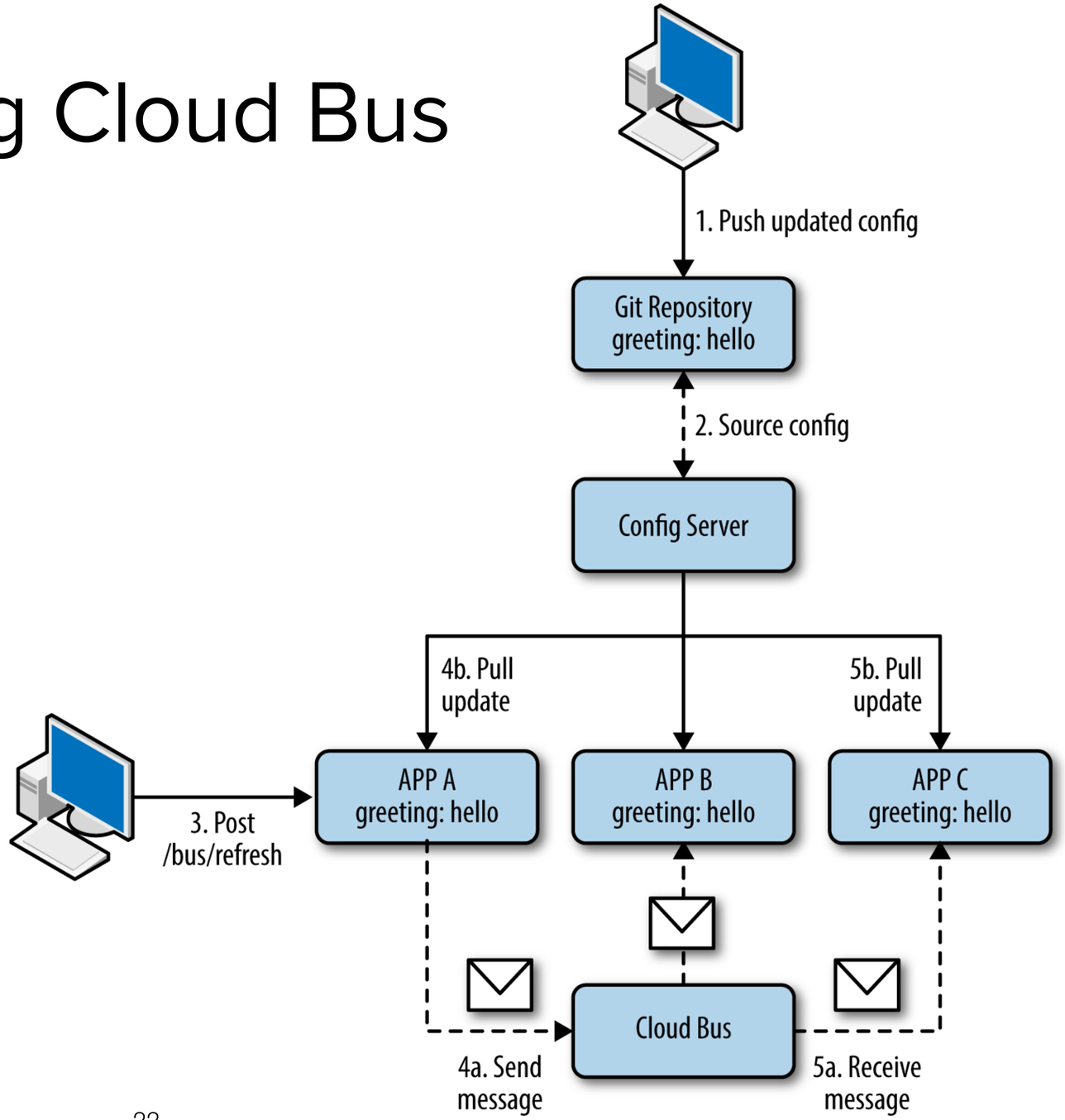
Spring Cloud Bus

- A solution to the problem of having to refresh multiple instances of multiple applications, without having to send each application instance a /refresh
- Leverages a message bus, supports AMQP protocol (e.g. RabbitMQ), and Kafka
- Exposes its own `/bus/refresh` endpoint, where receiving application instance broadcasts refresh message to all other instances

See: *https://cloud.spring.io/spring-cloud-bus/*

Config Server + Spring Cloud Bus

- The `/bus/refresh` endpoint provides a `destination` parameter to support specifying which applications and application instances to target



Benefits of Config Server

- All configuration is available in one place (one place to review and modify configuration)
- Separation of application development lifecycle from configuration lifecycle
- Ability to re-configure aspects of a running application without downtime (e.g. log level, feature toggles)
- Supports encryption of sensitive configuration properties using a number of mechanisms (symmetric encryption, asymmetric key pair)
- Choice of git backend provides complete configuration history, auditability (who made what configuration change, and when)