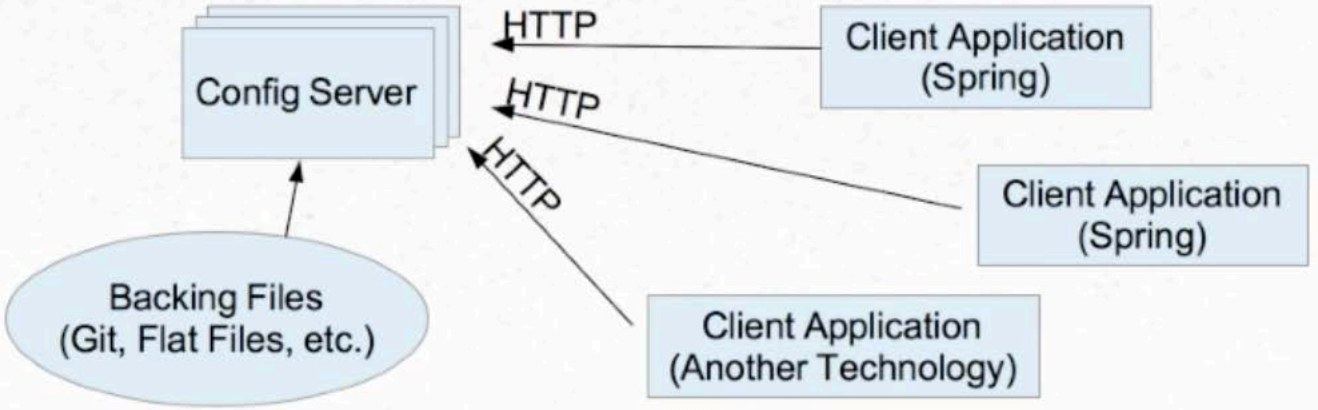
Module Outline

- The Problem: Dynamic Configuration Updates
- Spring Cloud Bus
- How Refresh Works

Recall Spring Cloud Config

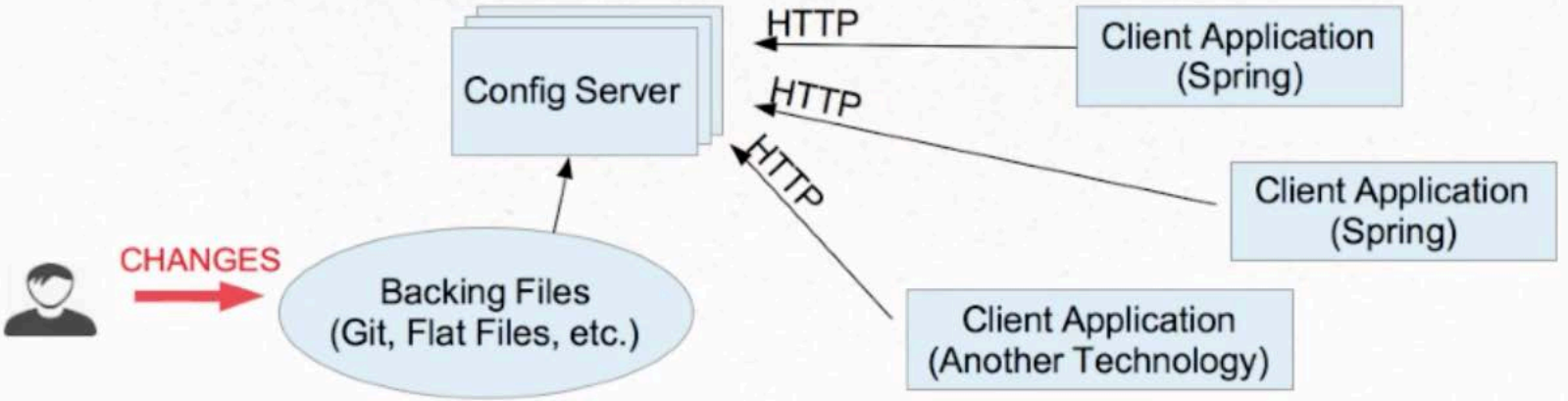
- Centralized server that serves-up configuration information
  - Configuration itself can be backed by source control

- Clients connect over HTTP and retrieve their configuration settings
  - Clients connect at startup time.



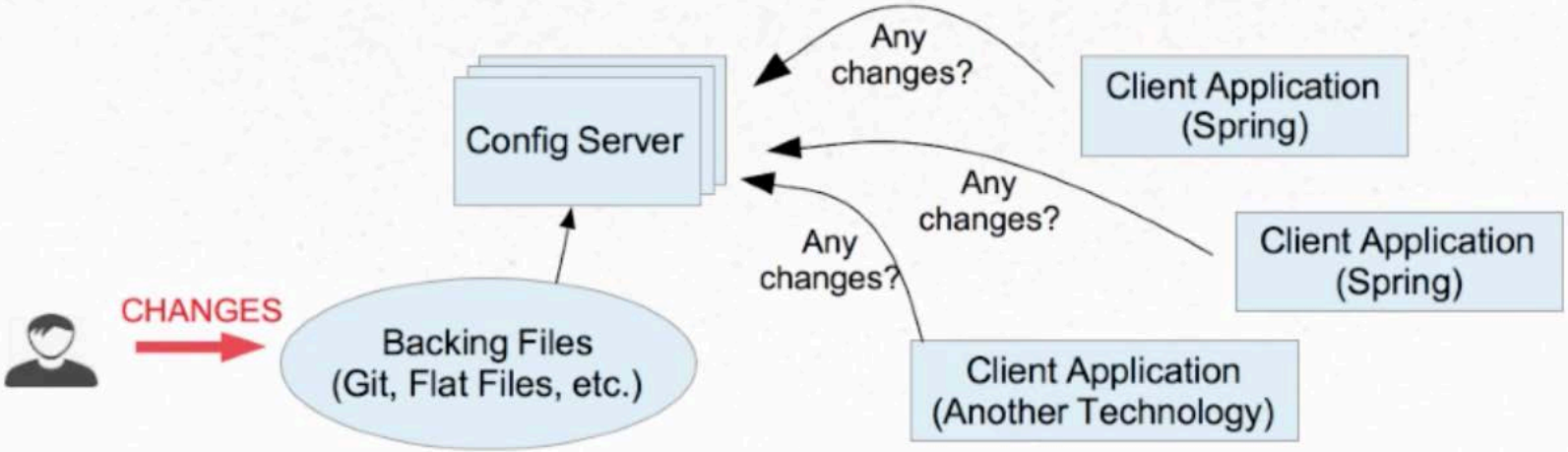
Dynamic Configuration Changes

- But what if we have configuration changes after the client applications are running?
- Traditional approach: “Bounce” all applications
  - Repeating the startup process.



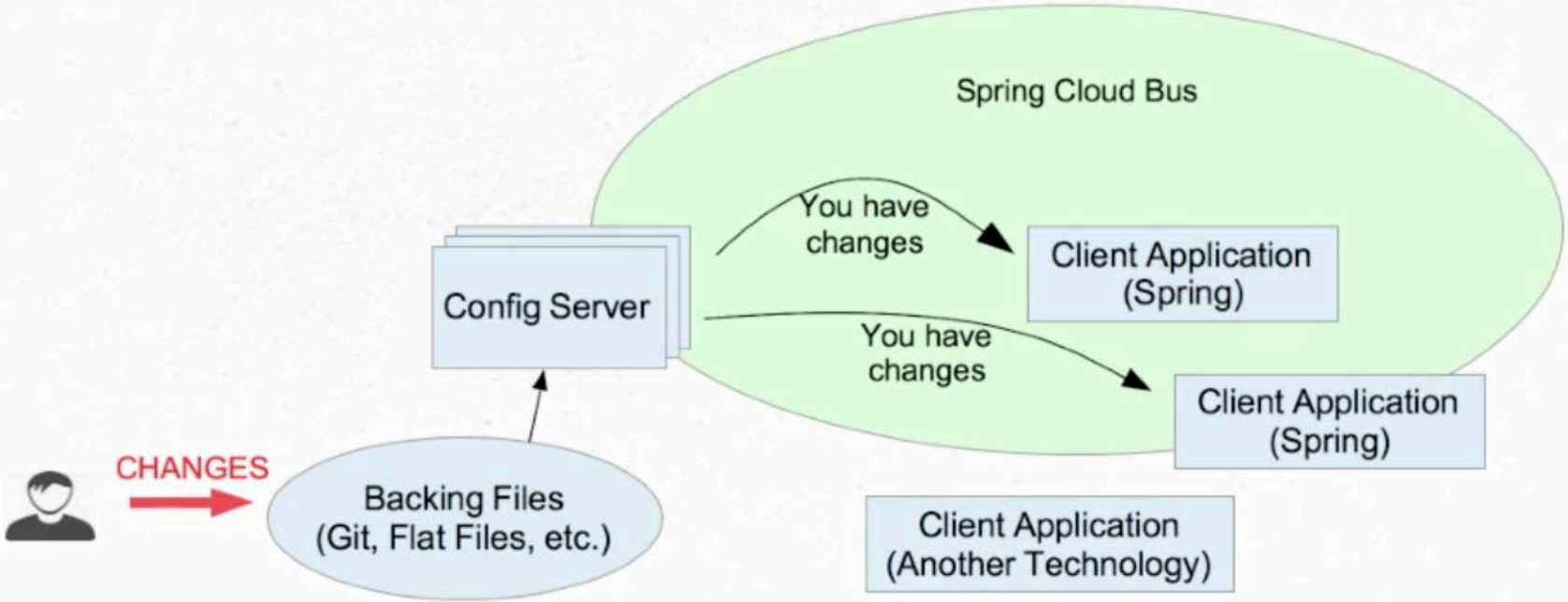
Potential Solution: Polling

- Applications could periodically poll the Config Server for changes
  - After all, they send Eureka heartbeats!
- Probably best to push the changes from server to client instead.
  - Config changes probably rare, no need to waste resources.



Spring Cloud Bus

- Push configuration changes to client applications via messaging technology, like AMQP.



Spring Cloud Bus

- Broadcasts configuration changes to clients.
  - Eliminates need for client polling
- Based on Messaging technology – Currently AMQP Only
  - Clients become subscribers to configuration changes.

Spring Cloud Bus Setup – Part 1

- Add dependency to the Spring Cloud Config Server:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
     <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

- Add the **same** dependency to each of your clients.
  - Code works automatically
  - Assumption: client code has spring cloud parent / dependency management section.

Spring Cloud Bus Setup – Part 2

- Run an AMQP server, such as Rabbit MQ

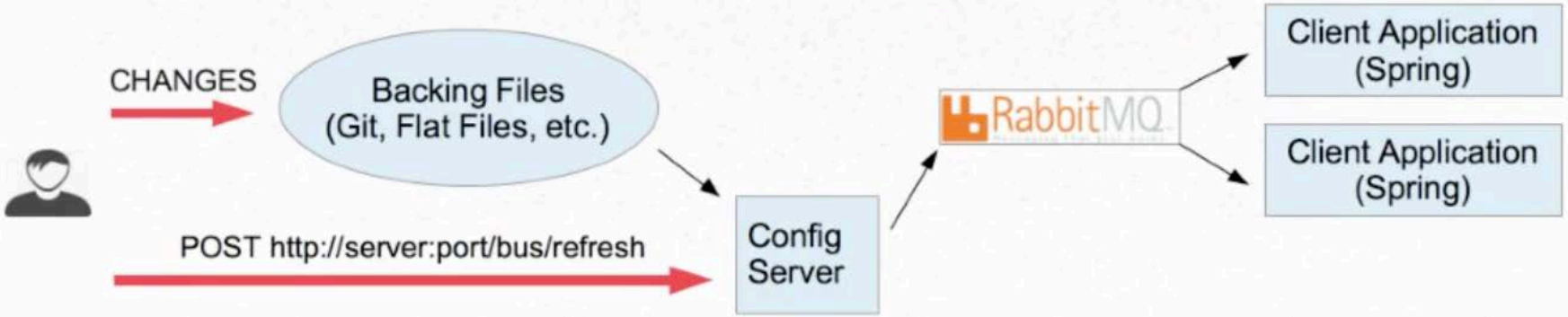
- Rabbit MQ:
  - Open Source
  - Easy to Install and Run
  - Pretty popular

- Spring Cloud Bus works automatically with Rabbit MQ on localhost.

AMQP - Advanced Message Queueing Protocol

Broadcasting Changes

1) Make changes to your config file(s)
   - Config Server does not poll for changes
2) POST /bus/refresh to your config server
3) Broker ensures message delivery to clients
4) Clients receive message and refresh themselves



How Refresh Works

- Spring Boot Applications can be Refreshed at Runtime
  - Actuator provides /refresh endpoint (POST)
    - org.springframework.boot / spring-boot-actuator dependency
  - ONLY affects the following:
    - Beans marked with @ConfigurationProperties
    - Beans marked with @RefreshScope
    - Logging level

@ConfigurationProperties

- Introduced in Spring Boot
- Easy alternative to multiple @Value annotations
- Properties rebound on POST /refresh

```
@RestController
@ConfigurationProperties(prefix="wordConfig")
public class LuckyWordController {

    String luckyWord;
    String preamble;

    @RequestMapping("/lucky-word")
    public String showLuckyWord() {
        return preamble + ": " + luckyWord;
    }

    // Getters and Setters
}
```

@RefreshScope

- Introduced in Spring Cloud
- Greater control, safe reloading of bean (not just property binding)
  - Side-effect: makes bean *lazy*
- Reloaded (not just rebound) on POST /refresh

```
@RestController
@RefreshScope
public class LuckyWordController {

    @Value("${wordConfig.lucky-word}") String luckyWord;
    @Value("${wordConfig.preamble}") String preamble;

    @RequestMapping("/lucky-word")
    public String showLuckyWord() {
        return preamble + ": " + luckyWord;
    }

    // Getters and Setters NOT required
}
```

How @RefreshScope Works

Spring creates a proxy for the actual bean

Proxy is dependency injected into other beans.

Proxy contains logic to call methods on the target bean.

On refresh, the “target” bean is changed to the newly created bean.

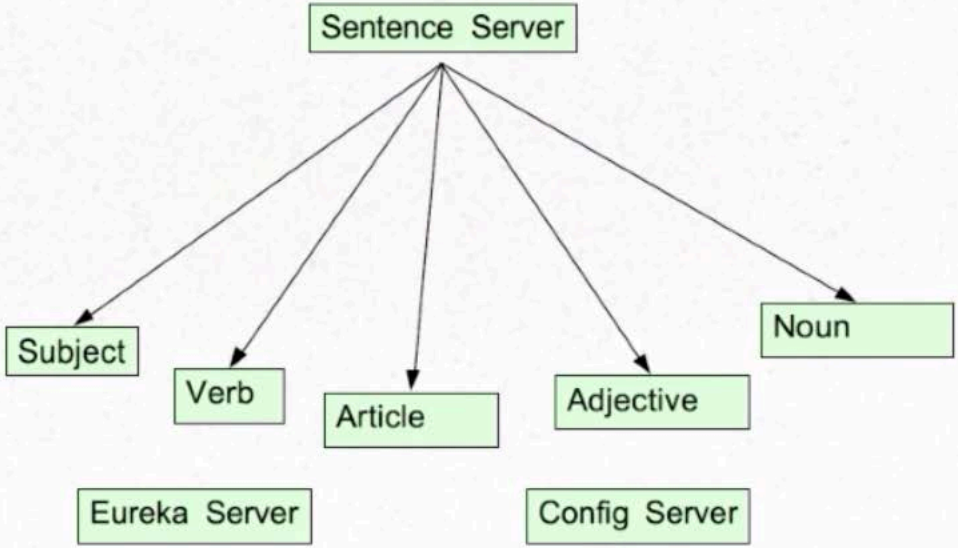
- Older bean is dereferenced

Result: users of original bean can safely finish their work.

The Current System

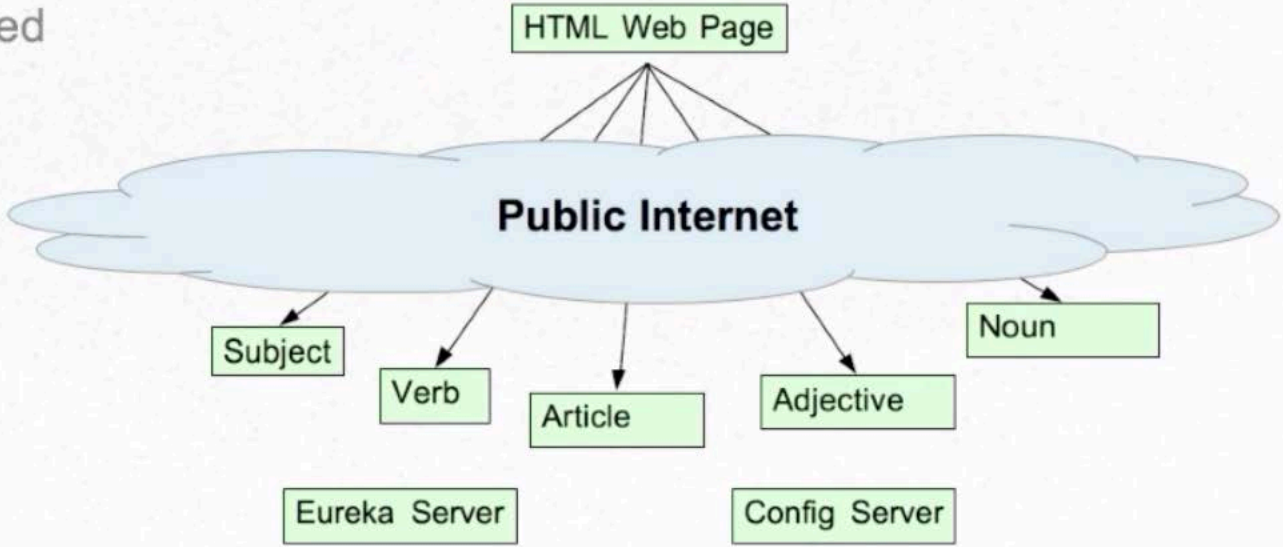
Existing Application

Runs fine, within a reliable, high-speed, secure network



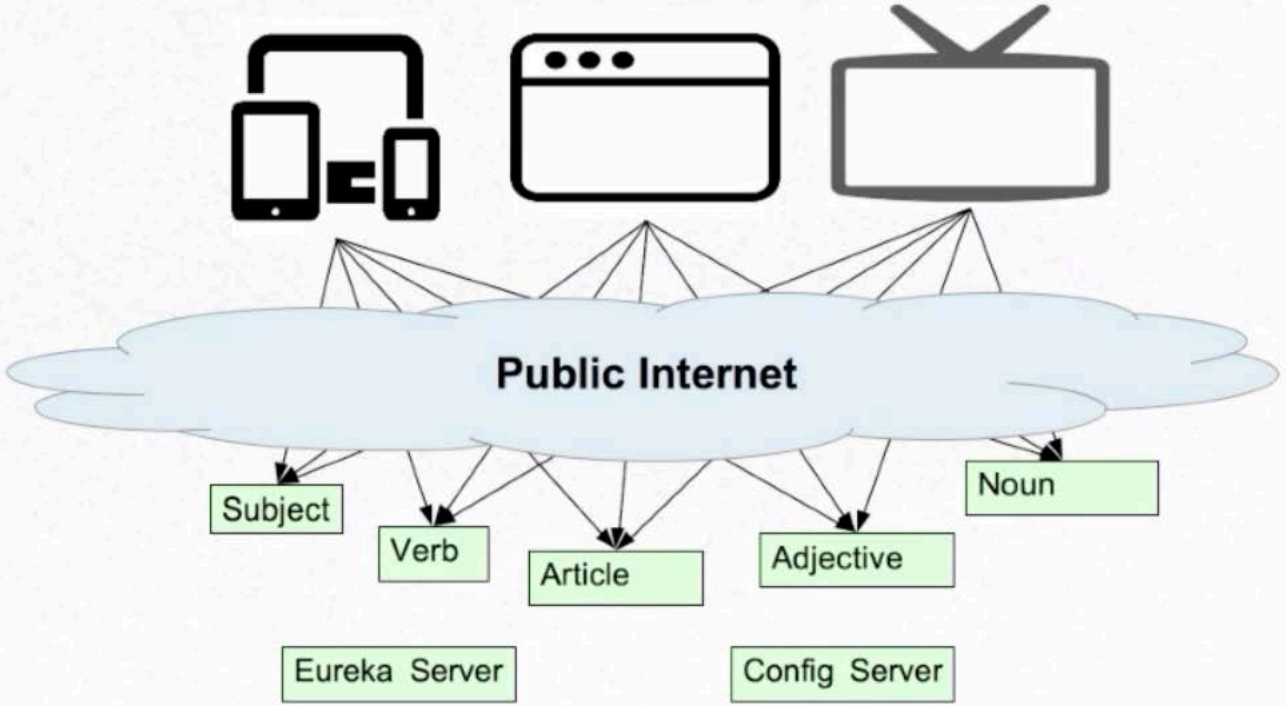
Accessing Microservices Via Web

- Accessing over public internet problematic
  - Internal API exposed
  - Security
  - CORS Required
  - Multiple Trips
  - Etc.



Accessing Microservices Via Web

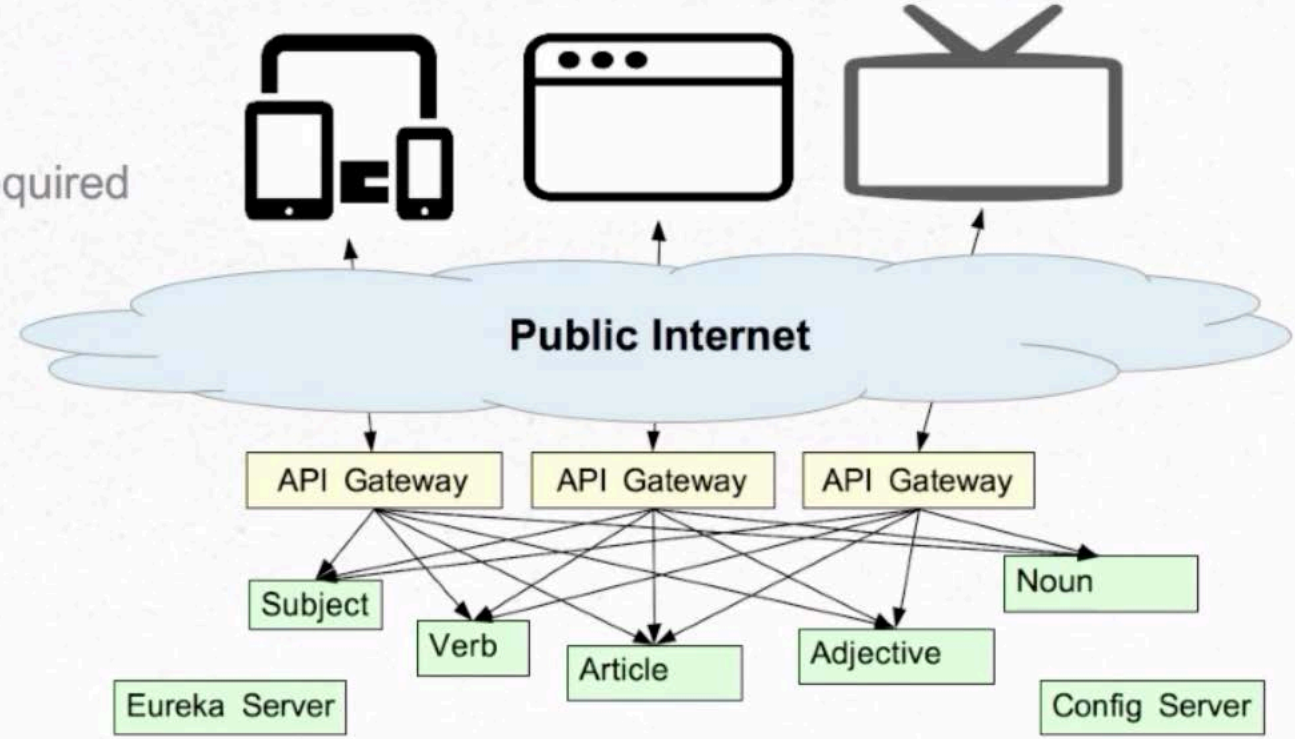
Different Clients Have Different Needs



API Gateway

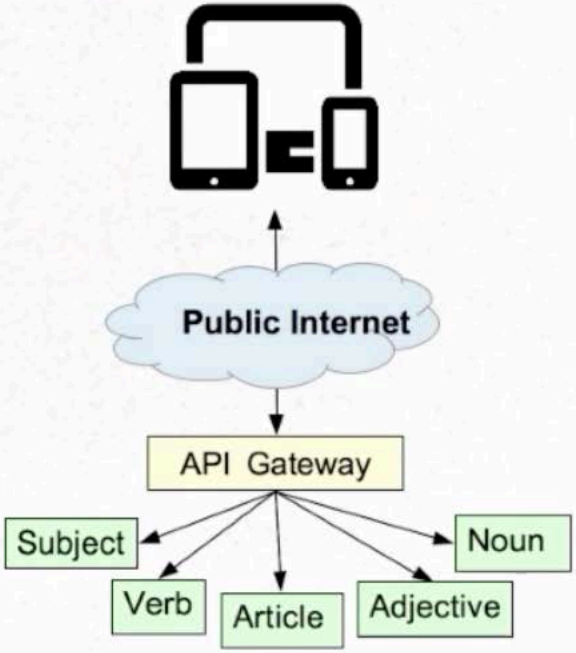
API Gateway provides simplified access for client

- Custom API
- Security
- No CORS Required
- Fewer Trips
- etc.



API Gateway

- Built for specific client needs (“facade”)
- Reduces # remote calls
- Routes calls to specific servers
- Handles Security / SSO
- Handles caching
- Protocol Translation
- Optimizes Calls / Link Expansion



Zuul – Routing and Filtering

- Zuul – JVM-based router and Load Balancer
  - Can be used for many API Gateway needs
  - Routing – Send request to real server
    - Reverse Proxy

Zuul Basic Usage

- Dependencies: (spring-cloud-starter-zuul)
  - Includes Ribbon and Hystrix
- Annotation: @EnableZuulProxy
- Default Behavior:
  - Eureka client ids become URIs
    - /subject routes to the “subject” service
    - /verb routes to the “verb” service
    - Etc.