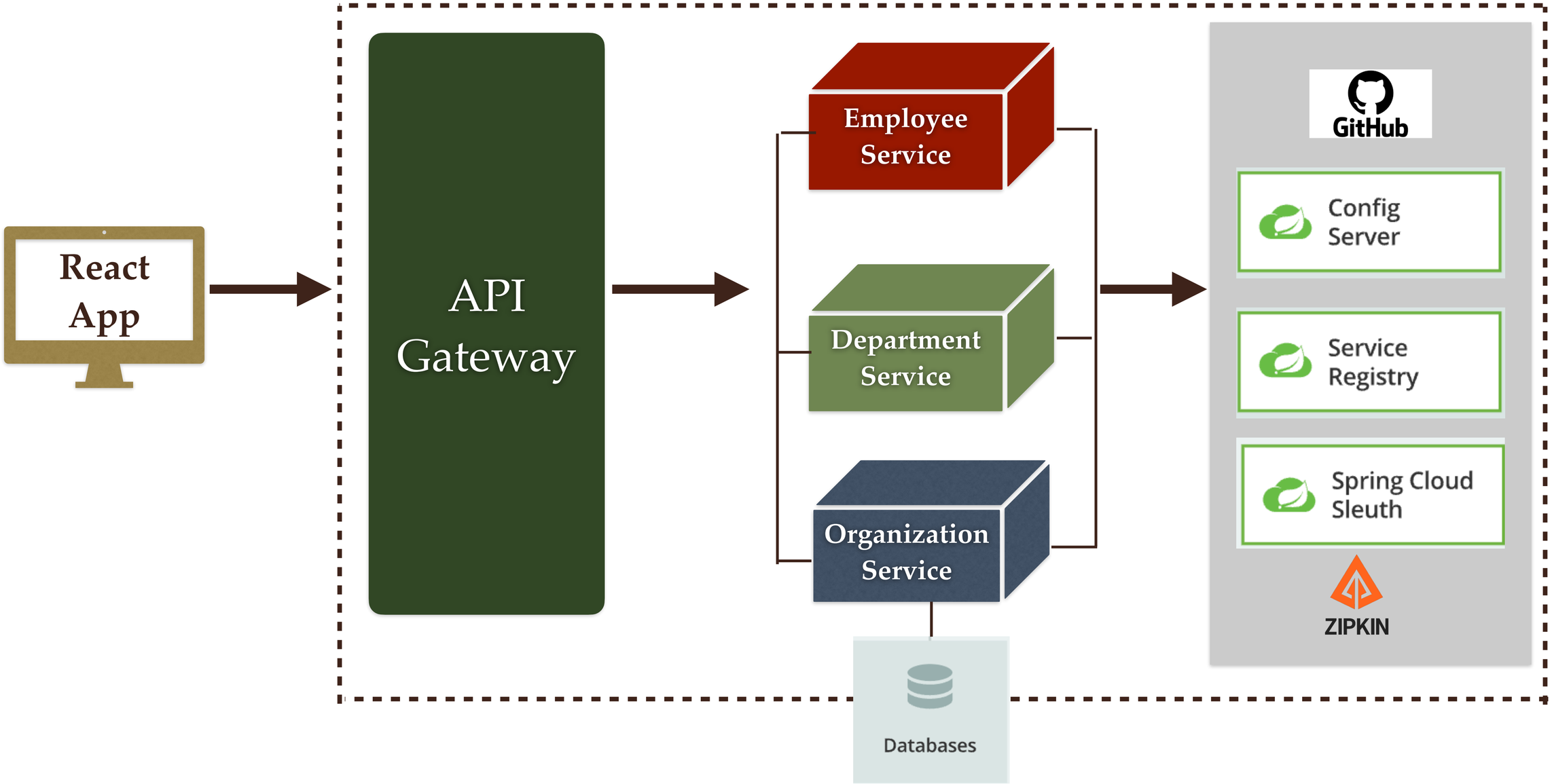
Building Microservices using Spring Boot and Spring Cloud

Spring Boot REST API's

1. Create Spring Boot REST API Basics (Learn Important Annotations)
2. Learn Creating CRUD REST API's using Spring Boot

Microservices Architecture using Spring boot and Spring Cloud



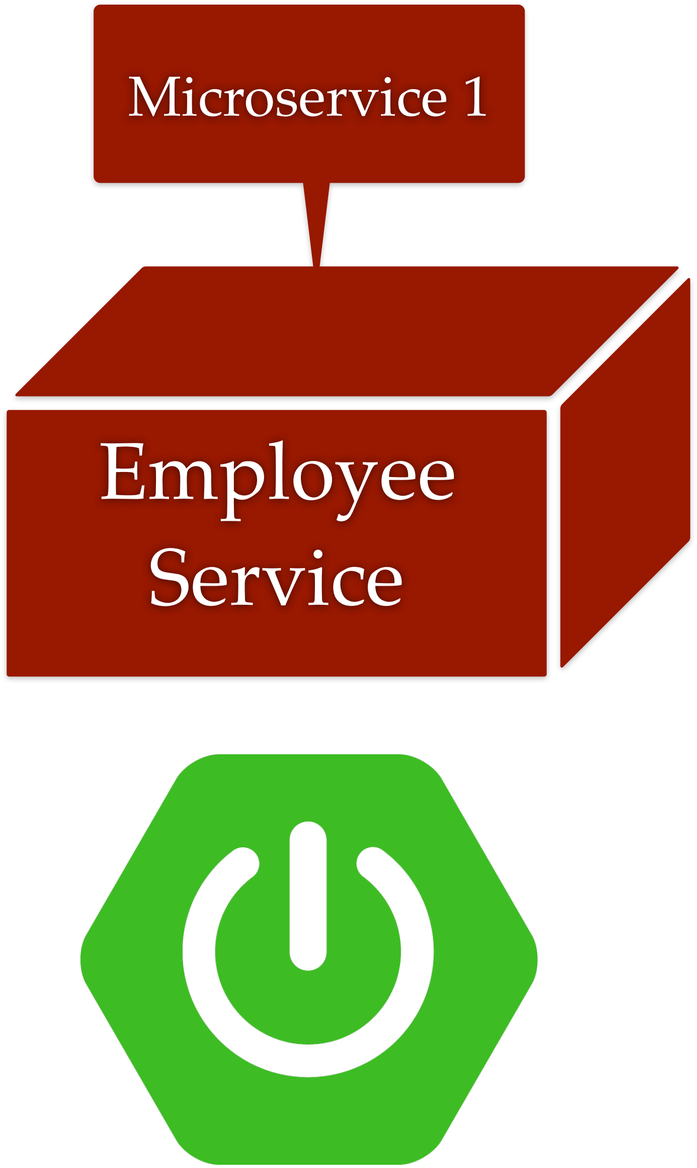
Choose the compatible version of Spring boot and Spring cloud

Why Spring Boot and Spring Cloud are a good choice for Microservices?

What is Spring Cloud

1. Spring Cloud is essentially an implementation of various design patterns to be followed while building Cloud Native applications. Instead of reinventing the wheel, we can simply take advantage of various Spring Cloud modules and focus on our main business problem than worrying about infrastructural concerns.

Create Two Microservices



Import and Setup Two Microservices in IntelliJ

By Ramesh Fadatare (Java Guides)

Configure MySQL Database in DepartmentService

By Ramesh Fadatare (Java Guides)

Create Department JPA Entity and Spring Data JPA Repository in DepartmentService

By Ramesh Fadatare (Java Guides)

Create Spring Data JPA Repository In DepartmentService

By Ramesh Fadatare (Java Guides)

Build Save Department REST API in DepartmentService

By Ramesh Fadatare (Java Guides)

Build Get Department REST API in DepartmentService

By Ramesh Fadatare (Java Guides)

Configure MySQL Database in EmployeeService

By Ramesh Fadatare (Java Guides)

Create Department JPA Entity In EmployeeService

By Ramesh Fadatare (Java Guides)

Create Spring Data JPA Repository In EmployeeService

By Ramesh Fadatare (Java Guides)

Build Save Employee REST API in EmployeeService

By Ramesh Fadatare (Java Guides)

Development Steps

1. Create EmployeeDto
2. Create Service Layer
3. Create Controller Layer

Build Get Employee REST API in EmployeeService

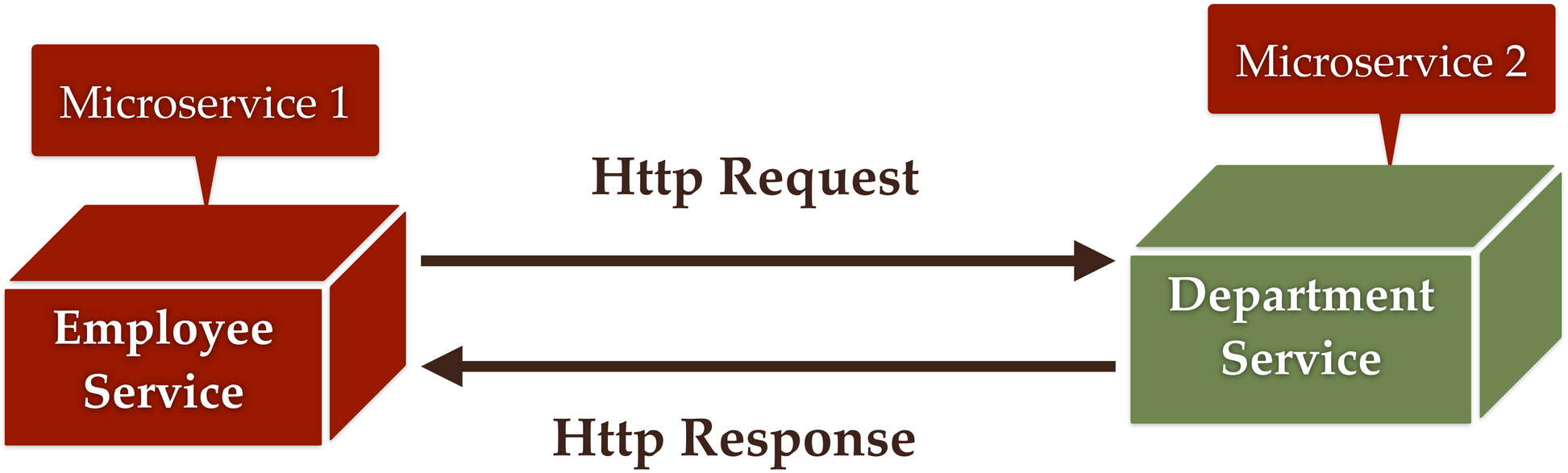
By Ramesh Fadatare (Java Guides)

Synchronous Communication

1. The client sends a request and waits for a response from the service.

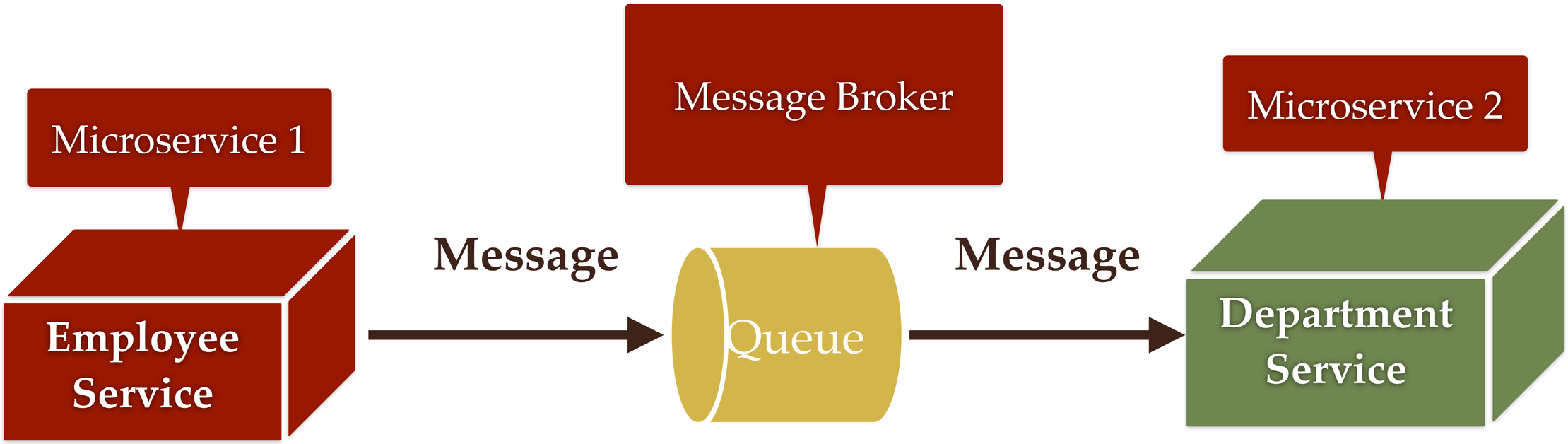
2. The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.

3. RestTemplate, WebClient and Spring Cloud Open Feign library



Asynchronous Communication

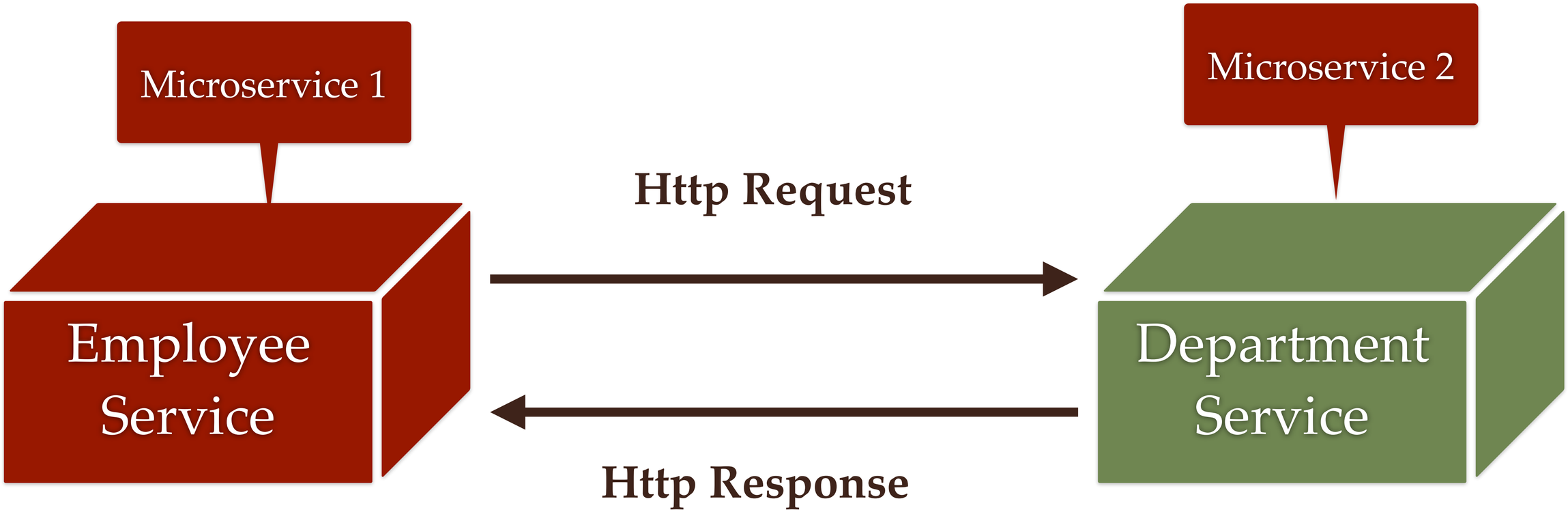
1. The client sends a request and does not wait for a response from the service.
2. The client will continue executing it's task - It don't wait for the response from the service.
3. RabbitMQ or Apache Kafka



Microservices Communication 3 Different Ways

Microservices Communication using RestTemplate

Make a REST API call from Employee-Service to Department-Service



Requirements

1. Consider Employee belongs to department and employee has a unique department code.
2. Change Get Employee REST API to return Employee along with it’s department in response.

Development Steps

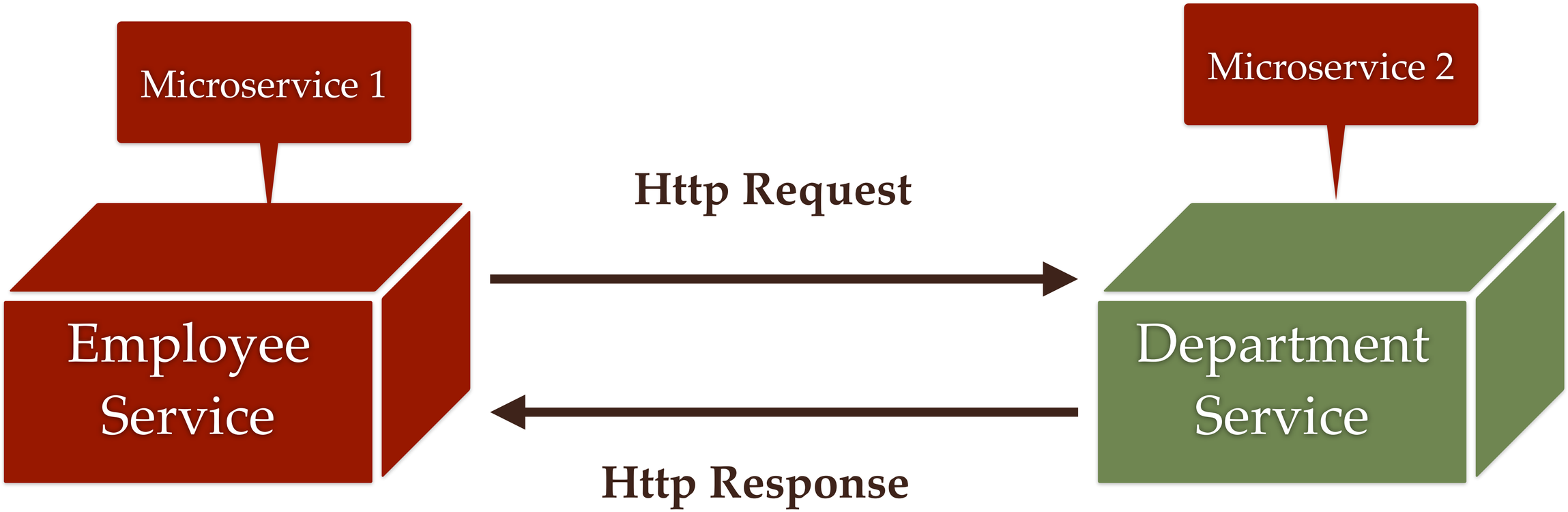
1. Add departmentCode field in Employee JPA Entity
2. Create DepartmentDto class
3. Configure RestTemplate as Spring Bean
4. Inject and use RestTemplate to make REST API call in EmployeeServiceImpl class

RestTemplate class is in maintenance mode

As of 5.0, the RestTemplate class is in maintenance mode and soon will be deprecated. So the Spring team recommended using org.springframework.web.reactive.client.WebClient that has a modern API and supports sync, async, and streaming scenarios.

Microservices Communication using WebClient

Make a REST API call from Employee-Service to Department-Service

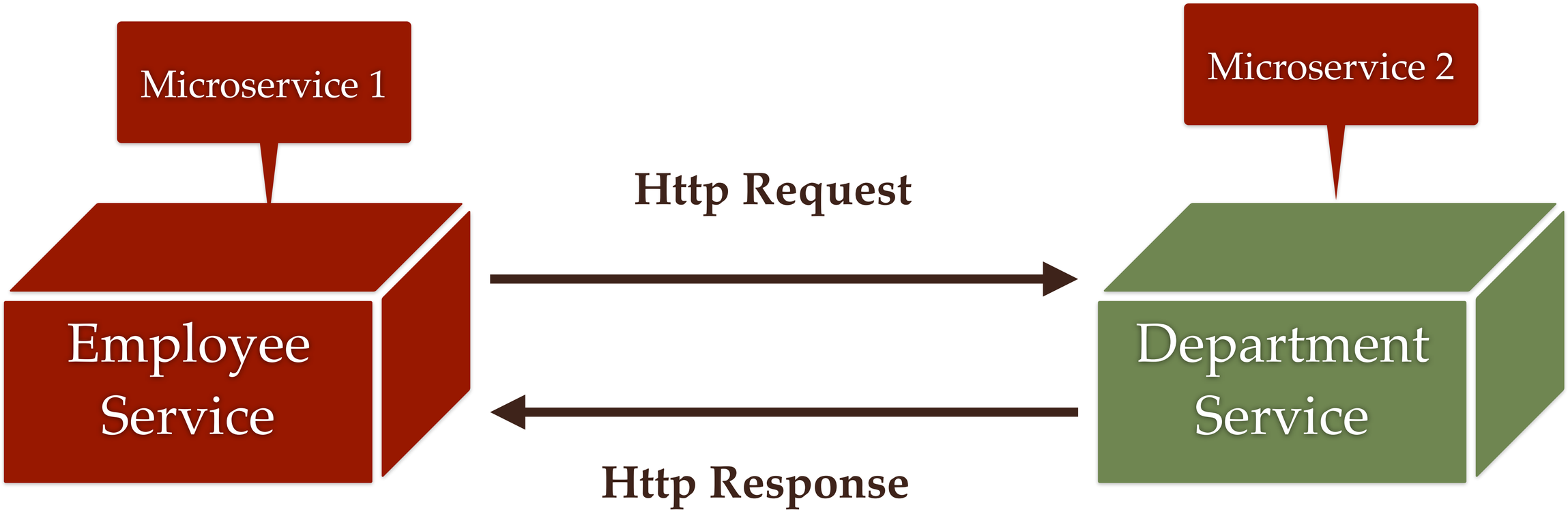


Development Steps

1. Add Spring WebFlux Dependency
2. Configure WebClient as Spring Bean
3. Inject and Use WebClient to Call the REST API
4. Test using Postman Client

Microservices Communication using Spring Cloud OpenFeign

Make a REST API call from Employee-Service to Department-Service



Development Steps

1. Add Spring cloud open feign Maven dependency to Employee-Service
2. Enable Feign Client using @EnableFeignClients
3. Create Feign API Client
4. Change the getEmployeeById method to use APIClient
5. Test using Postman Client

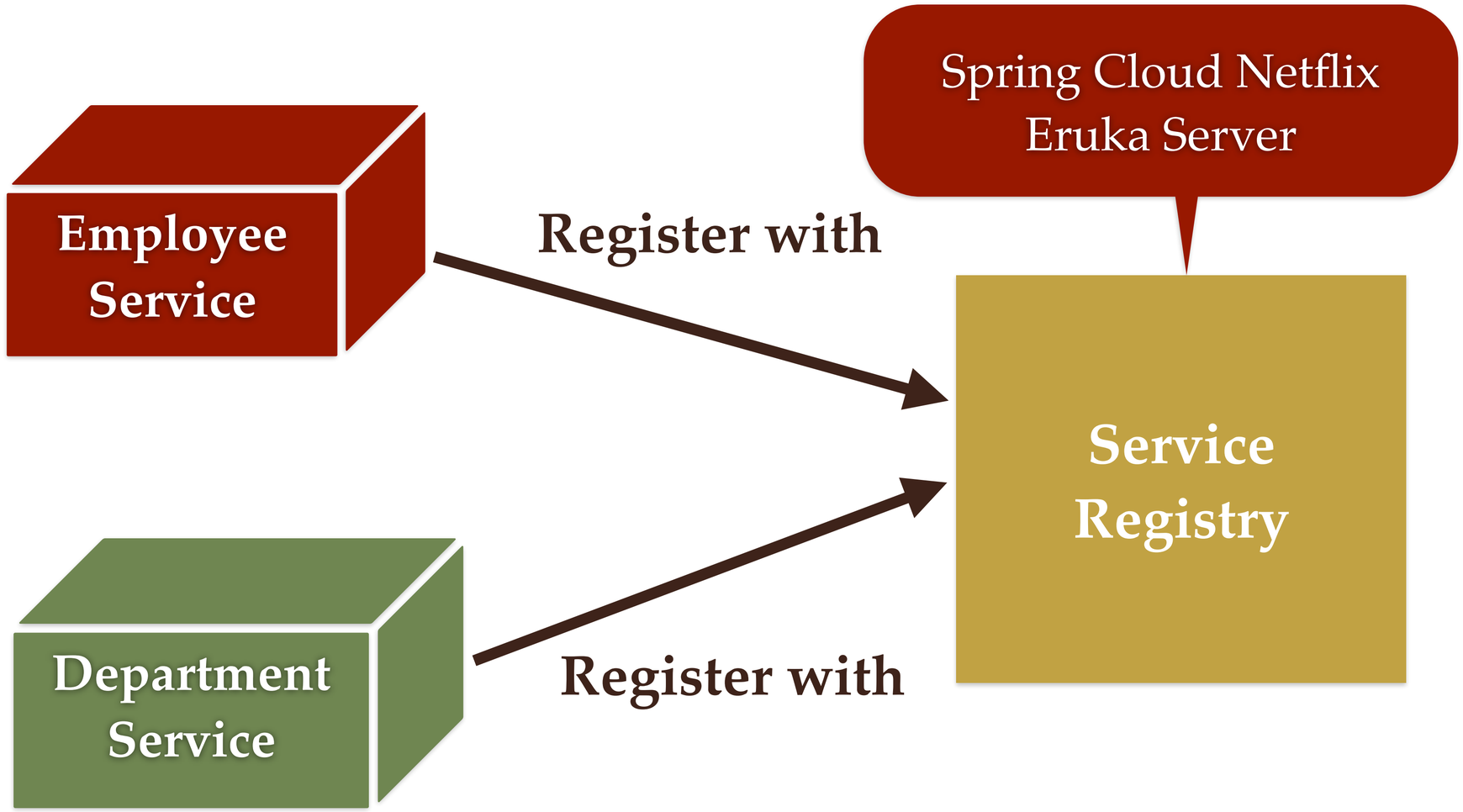
Service Registry and Discovery

1. In the microservices projects, **Service Registry and Discovery** play an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers.

2. In addition to that, in Cloud environments service instances may come up and go down anytime. So we need some automatic service registration and discovery mechanism.

3. Spring Cloud addresses this problem by providing **Spring Cloud Netflix Eureka** project to create Service Registry and Discovery.

Spring Cloud Netflix Eureka Server



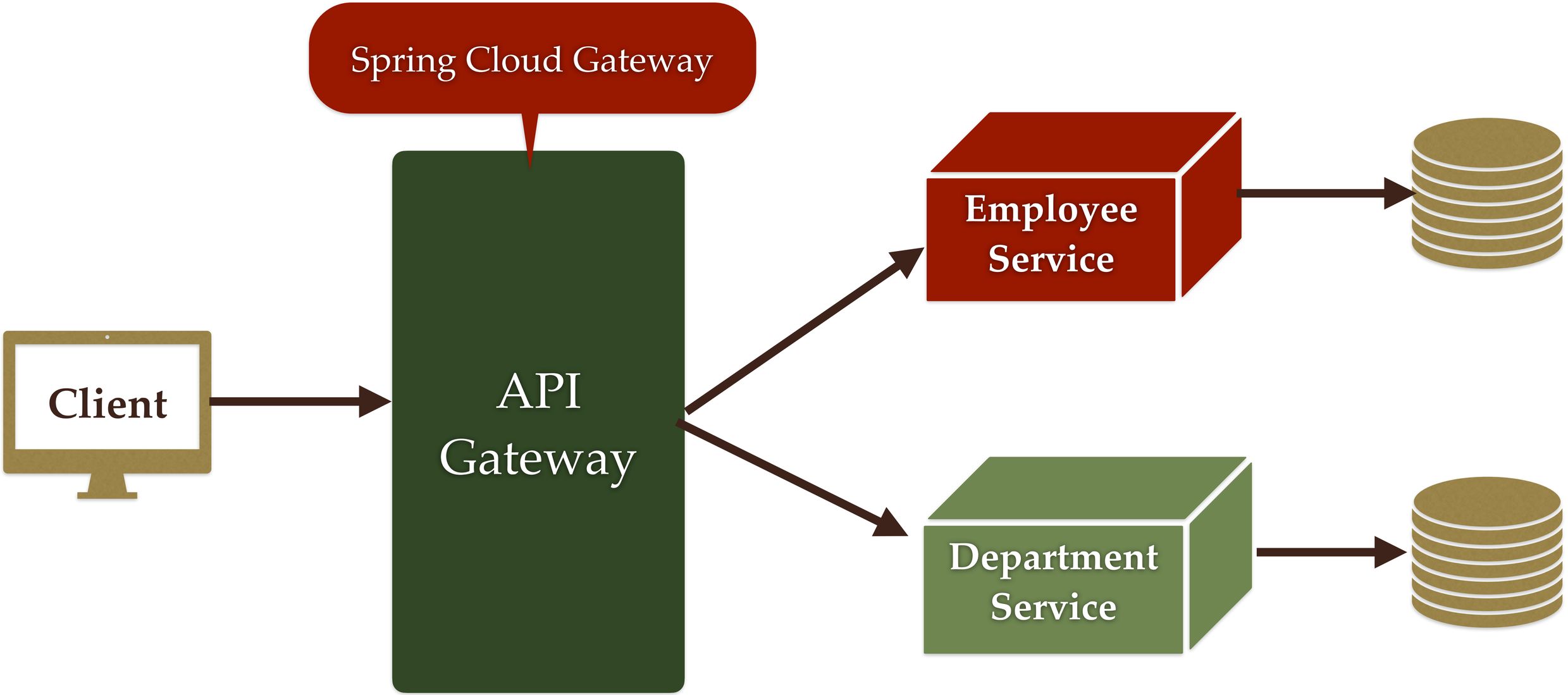
Development Steps

1. Create Spring boot project as Microservice (service-registry)
2. Add @EnableEurekaServer annotation
3. Disable Eureka Server as Eureka Client
4. Launch Eureka Server (Demo)
5. Registering Department-Service Microservice as Eureka Client
6. Run department-service Eureka Client (Demo)
7. Registering Employee-Service Microservice as Eureka Client
8. Run employee-service Eureka Client (Demo)
9. Multiple Instances of Department-Service

API Gateway

1. API Gateway provides a unified interface for a set of microservices so that clients no need to know about all the details of microservices internals.
2. API Gateway centralize cross-cutting concerns like security, monitoring, rate limiting etc
3. Spring Cloud provides **Spring Cloud Gateway** to create API Gateway

API Gateway


Spring Cloud Gateway
Client
API Gateway
Employee Service
Department Service

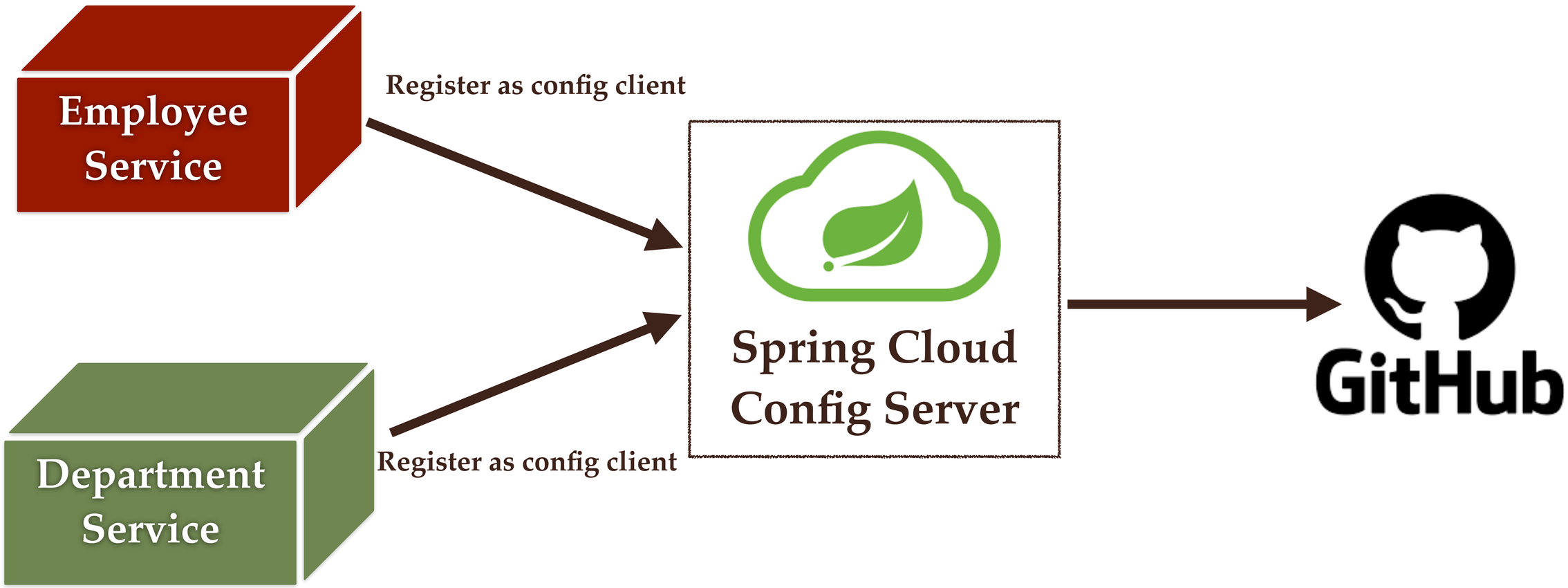
Development Steps

1. Create Spring boot project as Microservice (api-gateway)
2. Register API-Gateway as Eureka Client to Eureka Server (Service Registry)
3. Configuring API Gateway Routes and Test using Postman Client

What we will build?

1. We can create a Spring Cloud Config Server which provides the configuration values for all of our microservices. We use **git** as a backend to store the configuration parameters.
2. Next, we configure the location of Spring Cloud Config server in our microservice so that it will load all the properties when we start the application.
3. In addition to that, whenever we update the properties we can invoke **/refresh** REST endpoint in our microservice so that it will reload the configuration changes without requiring to restart the application.

Spring Cloud Config Server



Development Steps

1. Create Spring boot project as Microservice (config-server)
2. Register Config-Server as Eureka Client
3. Set up Git Location for Config Server
4. Refactor Department-Service to use Config Server
5. Refactor Employee-Service to use Config Server
6. Refresh Use case

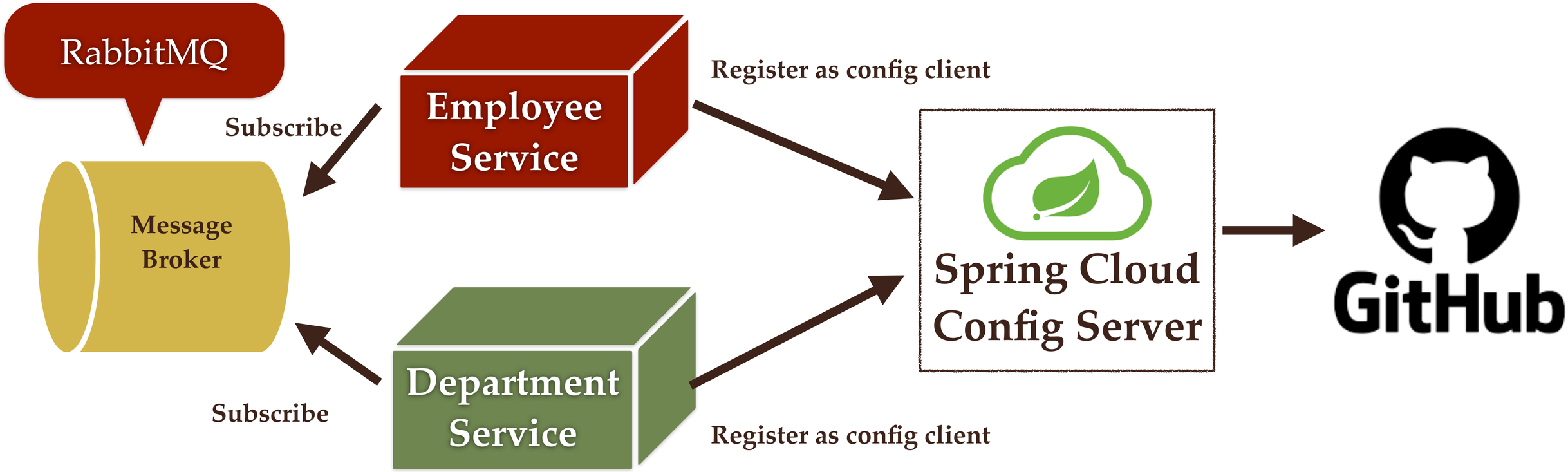
Refresh Use case

1. Whenever we change configuration file then we don’t have to restart the microservice and it’s instances
2. We need to call spring boot actuator **/refresh** API to reload the updated values from config server

Problem using Spring Cloud Config Server

1. In order to reload the config changes in Config Client applications (department-service and employee-service), we need to trigger **/refresh** endpoint manually. This is not practical and viable if you have large number of applications.
2. Spring Cloud Bus module provides a solution.
3. Spring Cloud Bus module can be used to link multiple applications with a message broker and we can broadcast configuration changes.

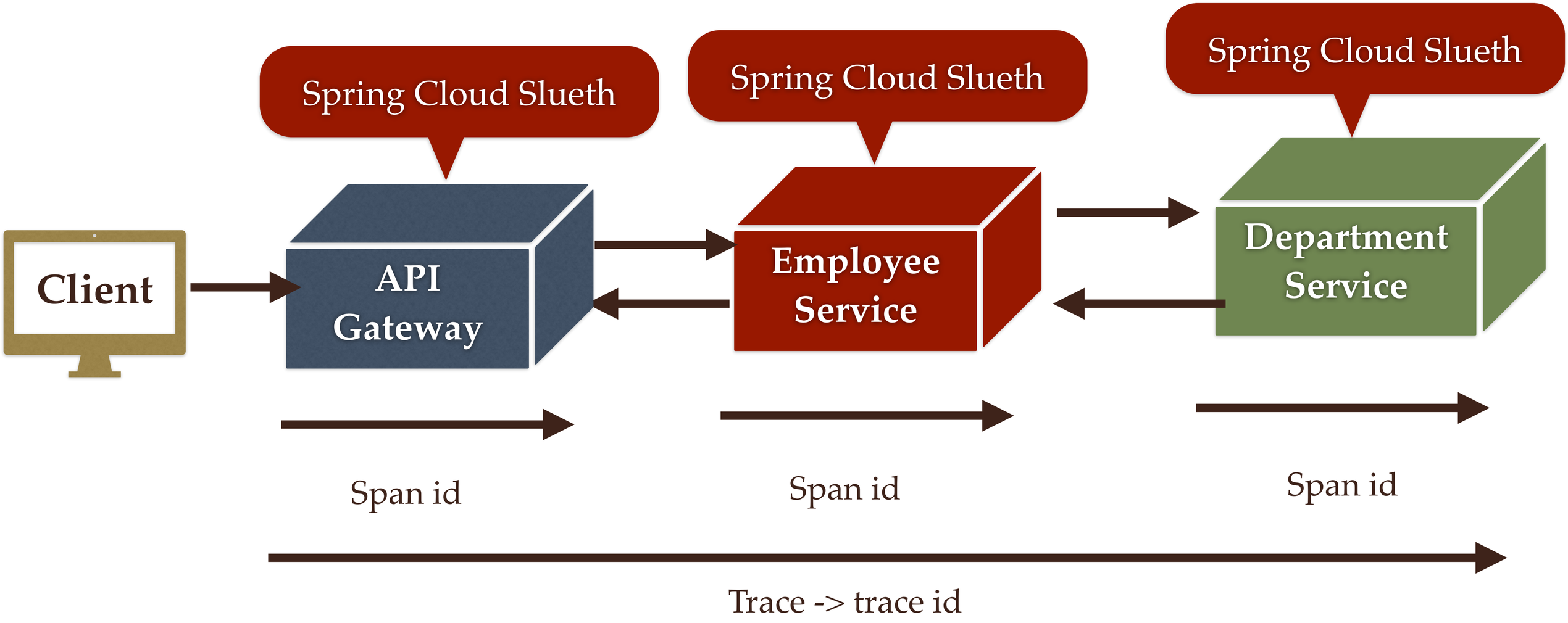
Spring Cloud Bus



Development Steps

1. Add **spring-cloud-starter-bus-amqp** dependency to department-service and employee-service
2. Install RabbitMQ using Docker
3. RabbitMQ configuration in **application.properties** of department-service and employee-service
4. Create Simple REST API in employee-service
5. Change department-service and employee-service properties file and call **/busrefresh**
6. Demo

Distributed Tracing



Distributed Tracing with Spring Cloud Sleuth and Zipkin

1. We use Spring Cloud Sleuth for distributed tracing

2. We use Zipkin to visualize trace information through UI

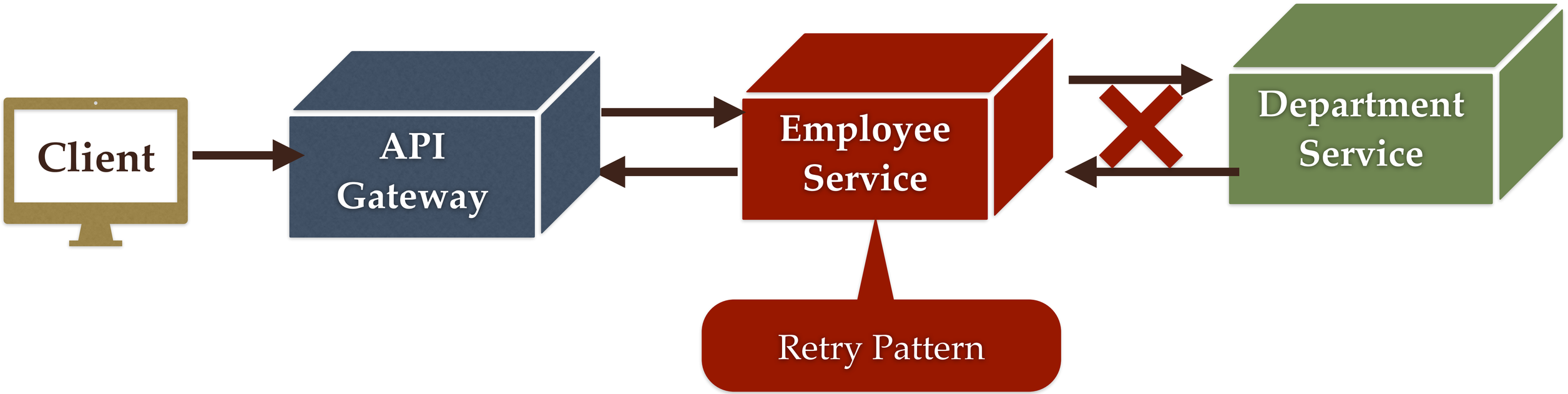
Development Steps

1. Implementing Distributed Tracing using Spring Cloud Sleuth Library
2. Using Zipkin to Visualize Trace Information through UI

Development Steps

1. Add dependencies
2. Using @CircuitBreaker annotation to a method (it is calling to external service)
3. Fallback method implementation
4. Add Circuit Breaker configuration in application.properties file
5. Restart employee-service and demo

Retry Pattern Implementation with Resilience4j



Development Steps

1. Using @Retry annotation to a method (it is calling to external service)
2. Fallback method implementation
3. Add Retry configuration in application.properties file
4. Restart employee-service and demo

Ports

App name: API-GATEWAY - **Port: 9191**

App name: DEPARTMENT-SERVICE - **Ports: 8080, 8082**

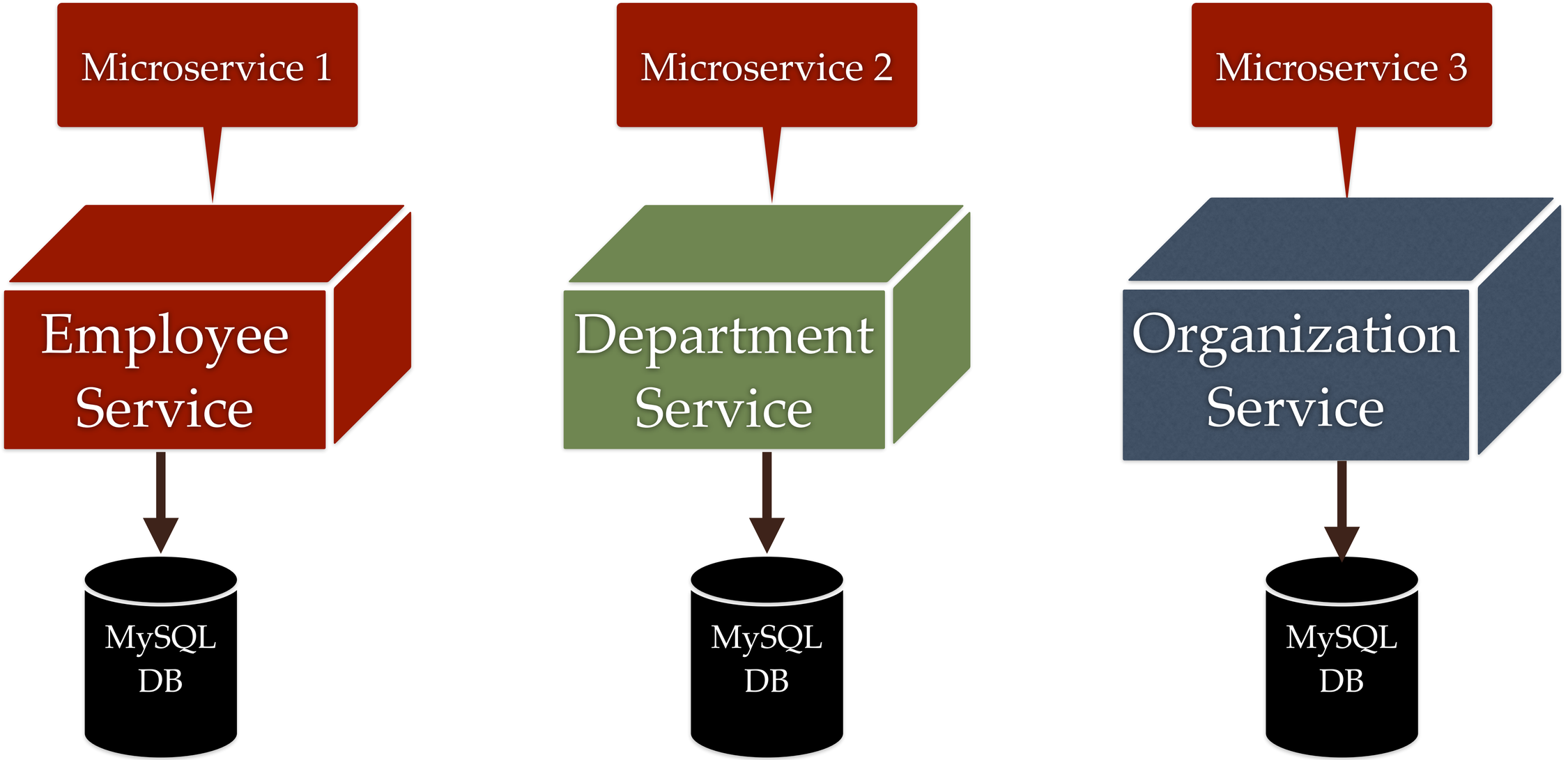
App name: EMPLOYEE-SERVICE - **Port: 8081**

App name: CONFIG-SERVER - **Port: 8888**

App name: SERVICE-REGISTRY - **Port: 8761**

Zipkin Server: **9411**

Microservices



Steps to Create Organization Service

1. Create Organization-Service using Spring Boot
2. Configure MySQL Database
3. Create Organization JPA Entity and Spring Data JPA Repository
4. Create OrganizationDto and OrganizationMapper
5. Build Save Organization REST API
6. Build Get Organization By Code REST API
7. Make REST API Call from Employee-Service to Organization-Service
8. Register Organization-Service as Eureka Client
9. Refactor Organization-Service to use Config Server
10. Configure Spring Cloud Bus and Routes for Organization-Service in API-Gateway
11. Implement distributed tracing in Organization-Service

Requirements

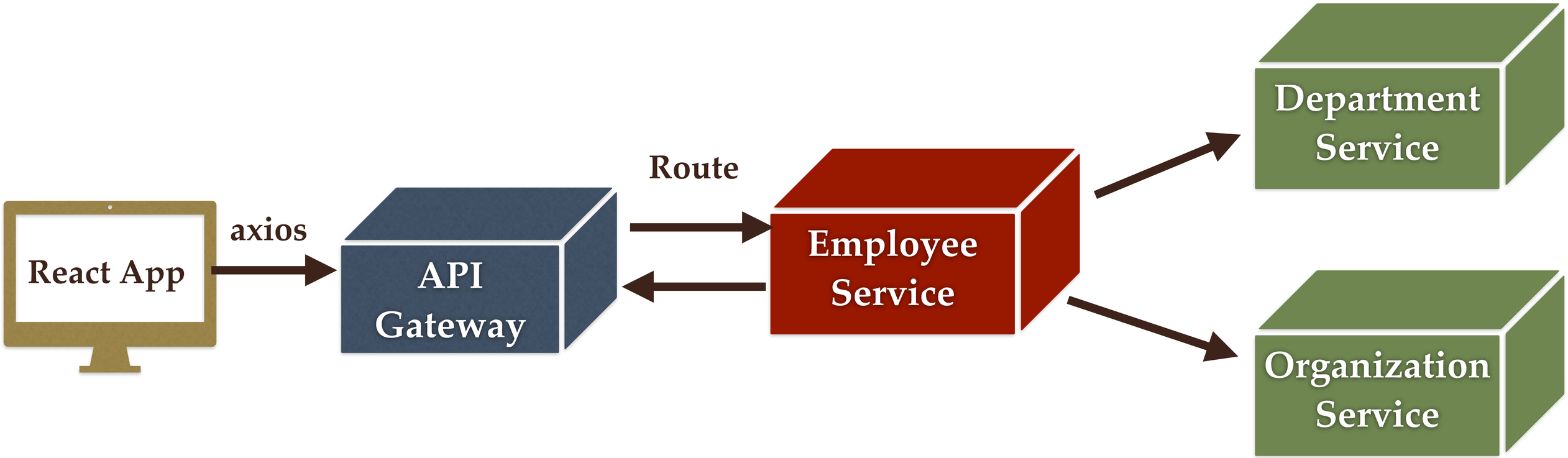
Client want's employee, department and organization details in a response.

Understanding the requirement:

Consider Employee belongs to organization and employee has a unique organization code.

Change Get Employee REST API to return Employee along with it's organization in response.

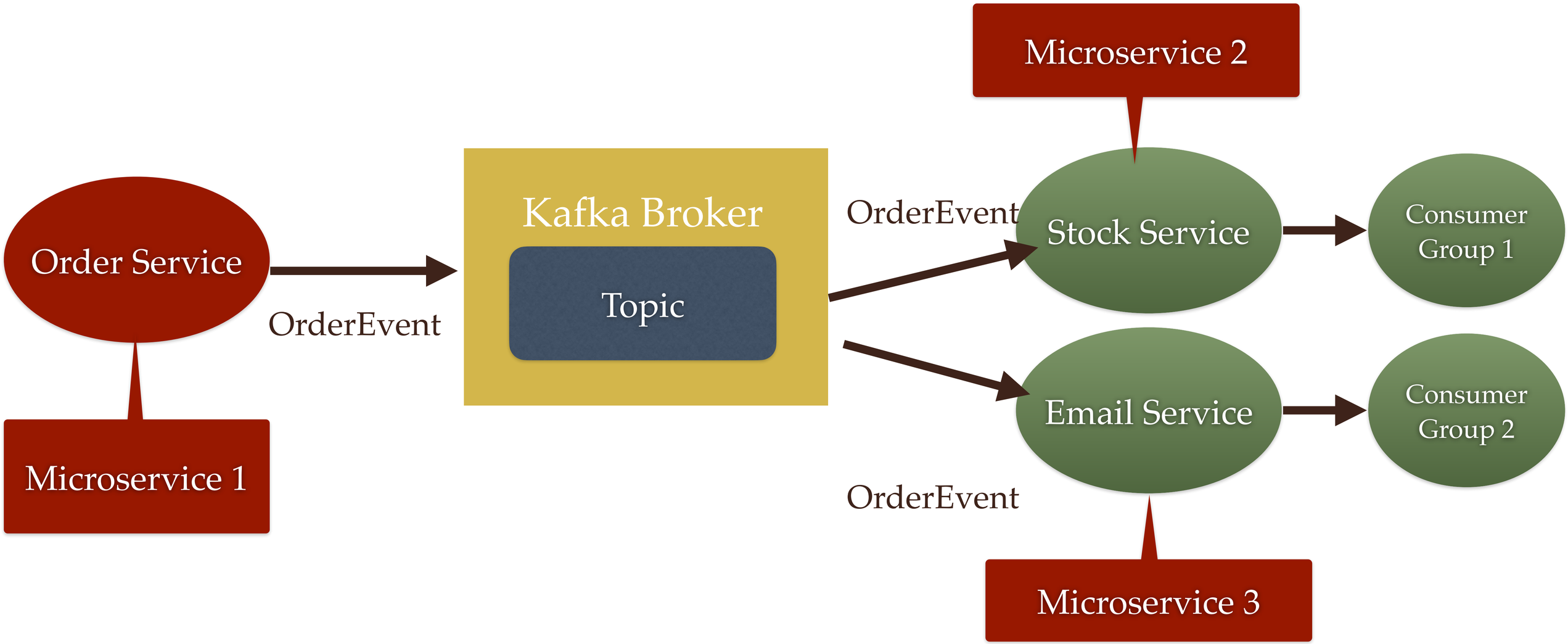
Frontend React App



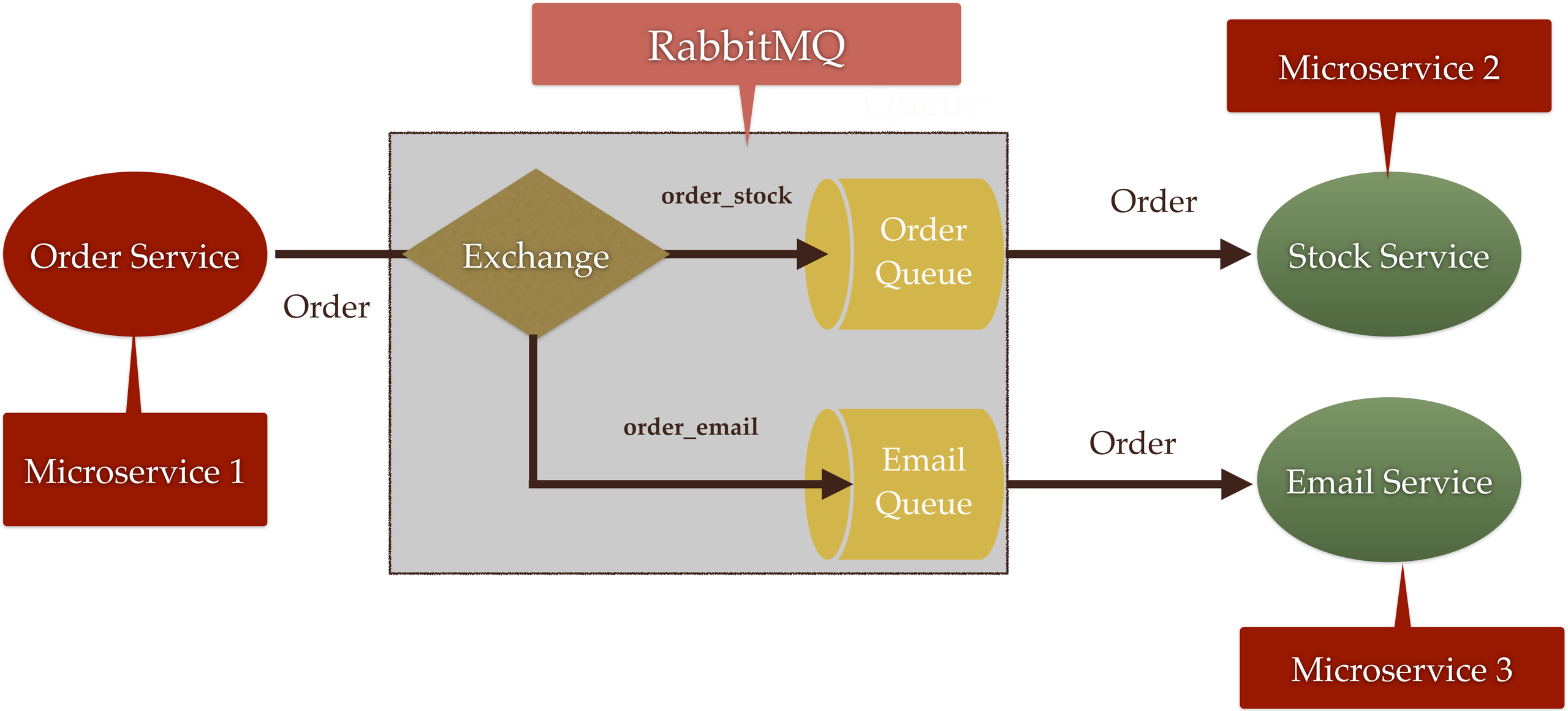
Development Steps

1. Create React App using Create React App Tool
2. Adding Bootstrap in React Using NPM
3. Connecting React App with API-Gateway (REST API Call)
4. Develop a React Component to Display User, Department and Organization Details
5. Run React App and Demo

Spring Boot Kafka Event-Driven Microservices Architecture with Multiple Consumers



Spring Boot RabbitMQ Event-Driven Microservices Architecture with Multiple Queues



Reference/Credit : Microservices Architecture from Spring official website