spring boot

微服务和云原生架构实践

MovieApp

kubernetes

一、课程大纲和项目架构

课程大纲上

1. 课程大纲和项目架构

2. JWT简介

3. 安全设计(Spring Security + JWT)

4. AuthSvc代码剖析

5. AuthSvc演示

6. MovieSvc代码剖析

7. 前端代码剖析一

8. 前端代码剖析二

课程大纲下

9. 微服务网关和代码剖析

10. 总体项目演示

11. Docker Compose部署文件剖析

12. 镜像构建和本地部署演示

13. 云原生和K8s

14. K8s核心概念

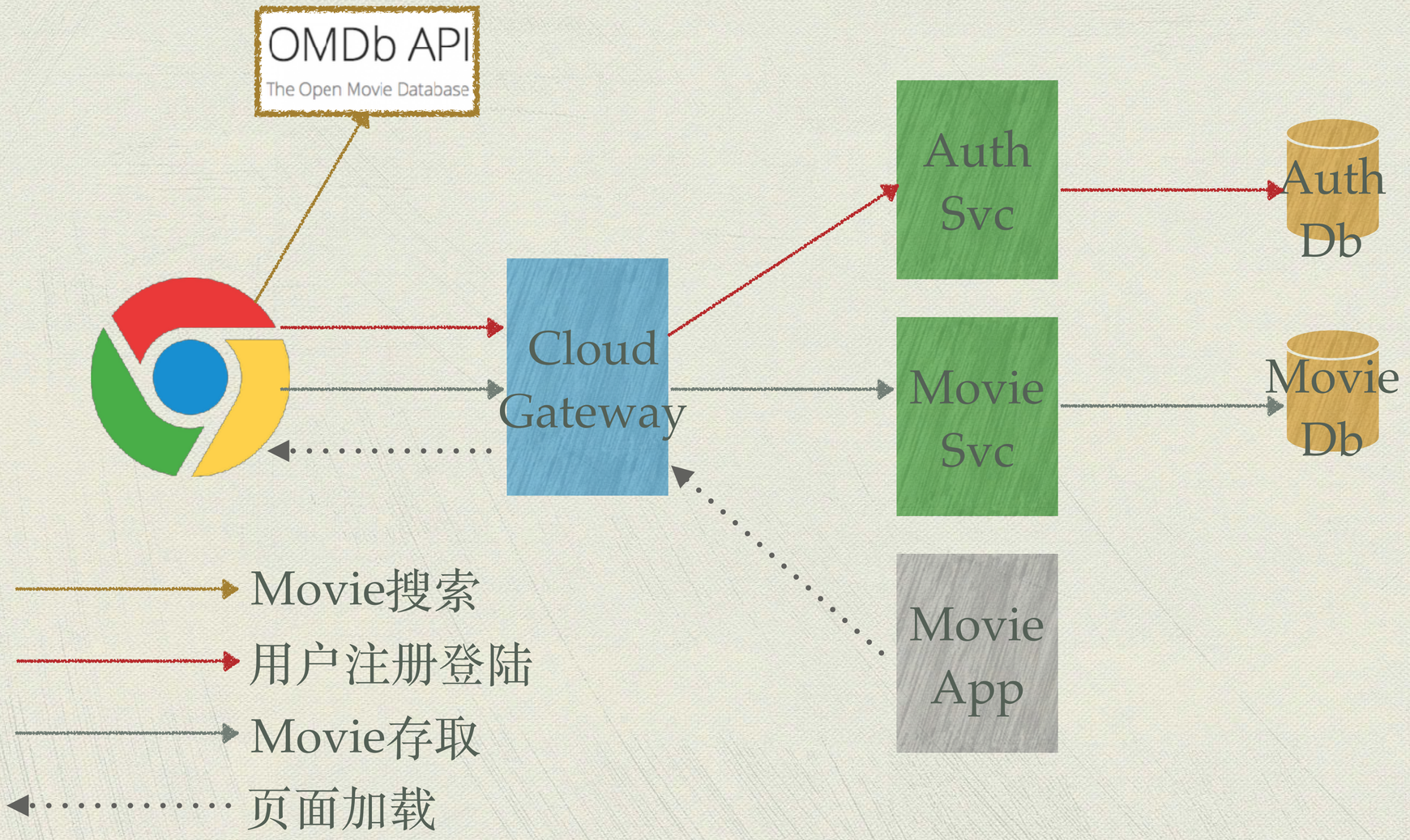
15. K8s部署文件剖析

16. K8s部署演示和项目小结

课程目标

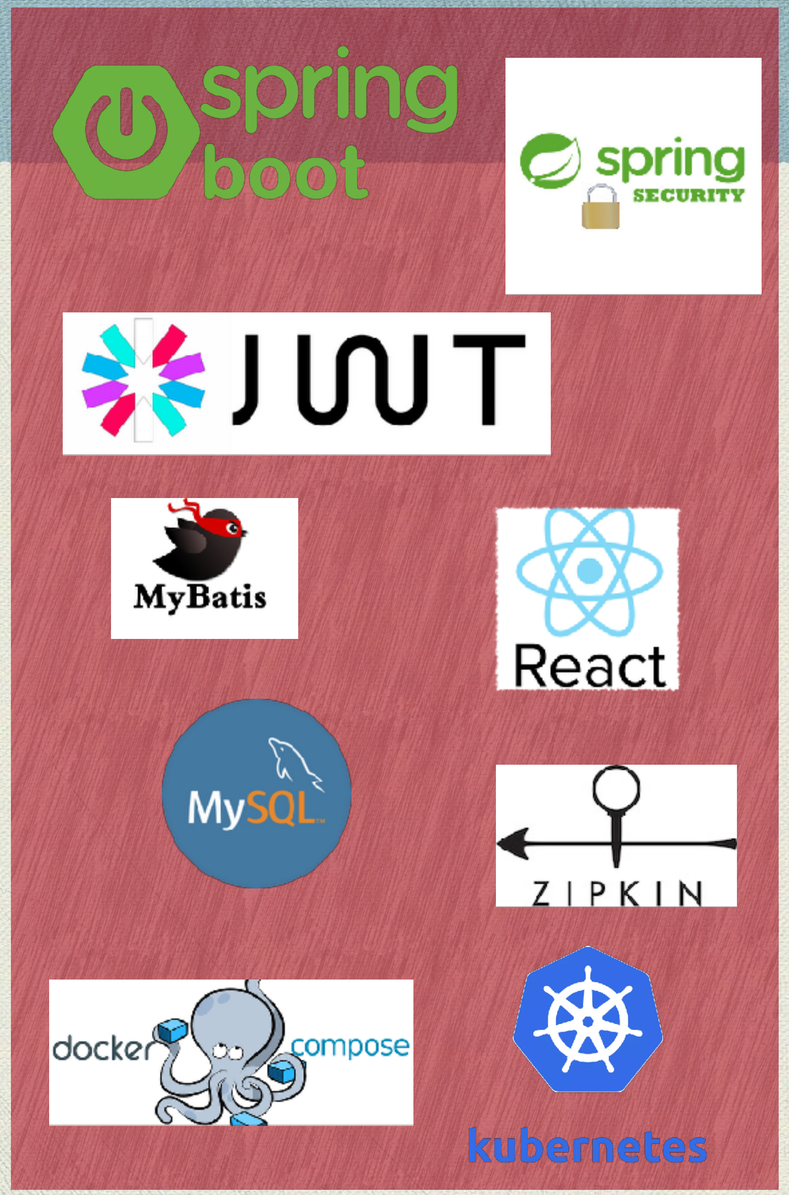
- 全栈架构和DevOps
  - 后端、前端、安全、微服务、测试、部署、监控
- 微服务架构
- 前后分离架构
- 云原生架构(Docker/Kubernetes)

项目架构


OMDb API
The Open Movie Database
Cloud Gateway
Auth Svc
Movie Svc
Movie App
Auth Db
Movie Db
Movie搜索
用户注册登陆
Movie存取
页面加载

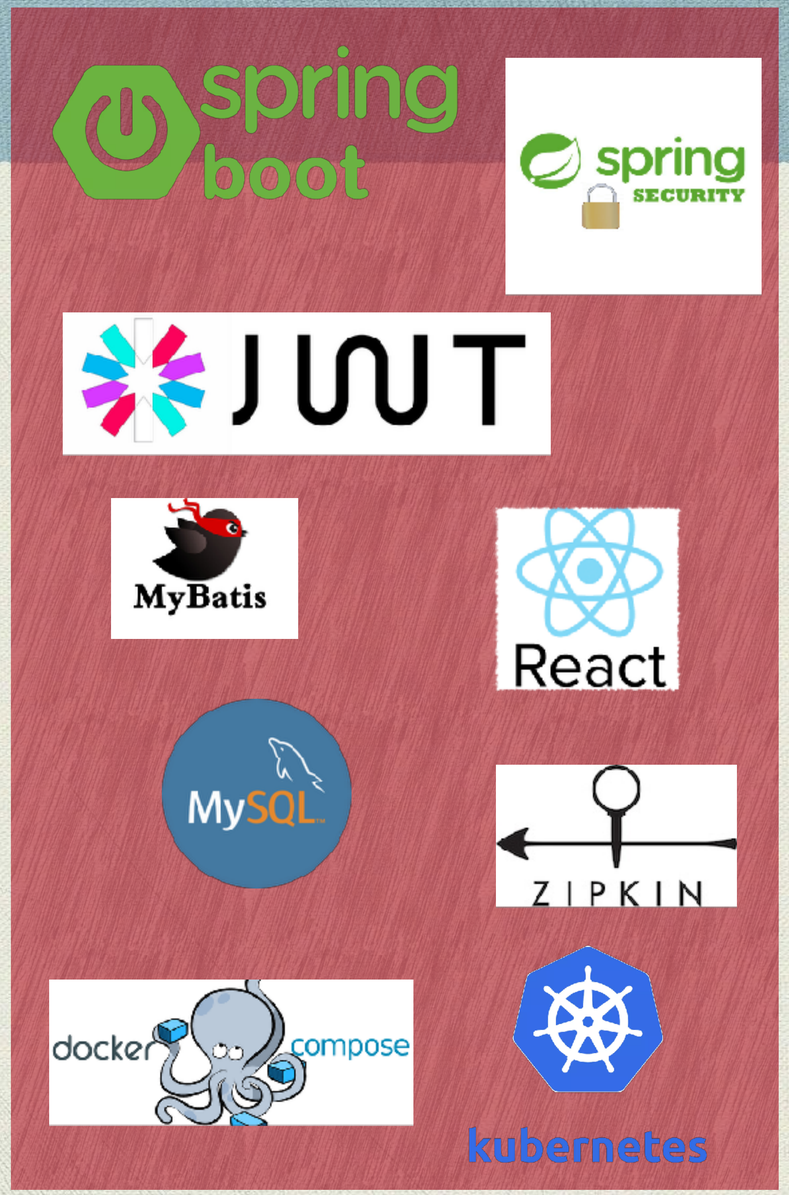
技术栈一

- Spring Boot ~ 后端开发框架
- Spring Security ~ 后端安全框架
- JWT ~ 令牌框架
- MyBatis ~ ORM框架
- React ~ 前端开发框架



技术栈二

- MySQL ~ 后端存储
- Spring Cloud Gateway ~ 微服务网关
- Zipkin ~ 调用链监控
- Docker Compose ~ 镜像构建和本地部署
- Kubernetes ~ 容器调度和部署



You successfully registered! Please Login!

Login

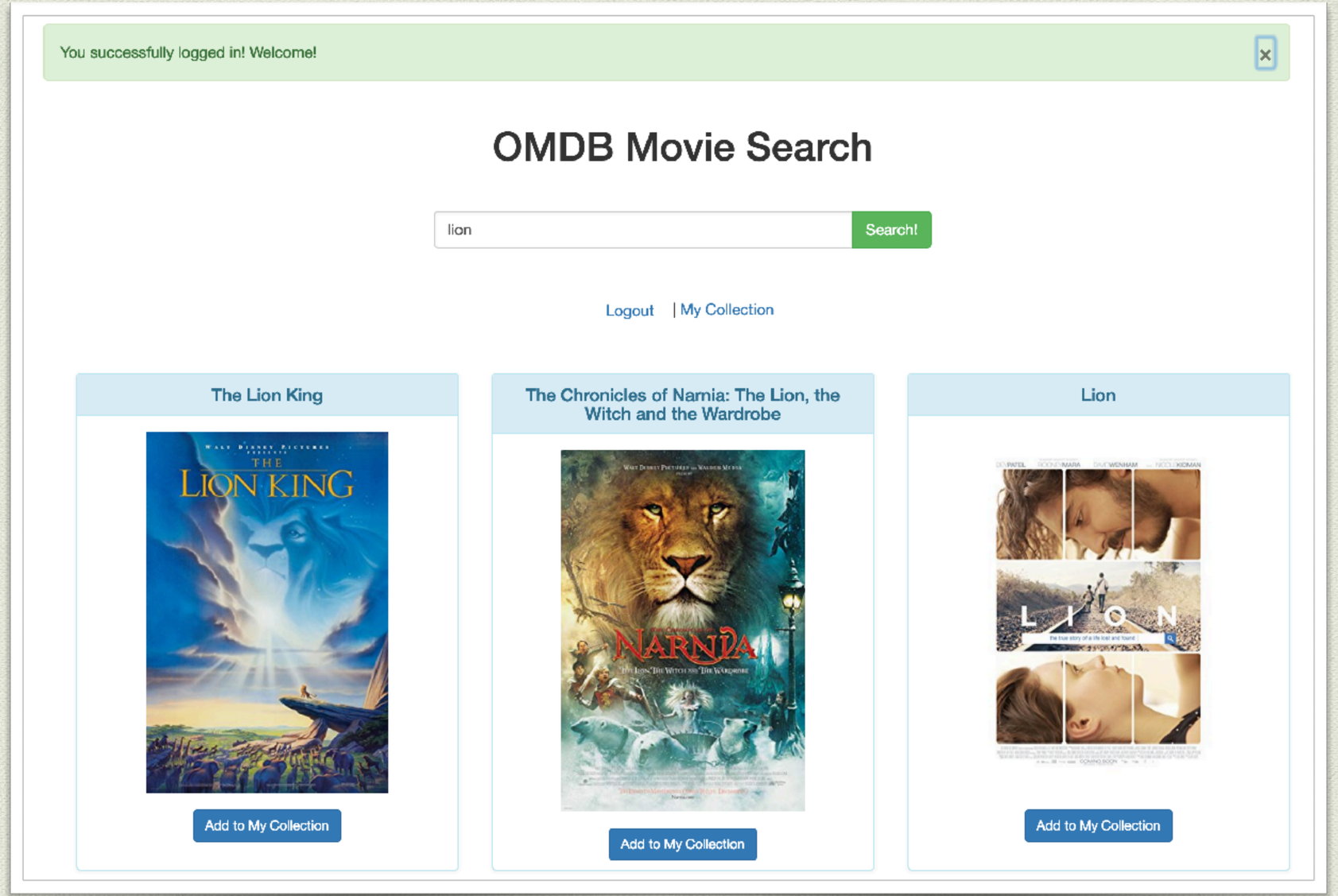
Username

Password

Log in Cancel

Need to register?

源码：github.com/jskillcloud/MovieApp



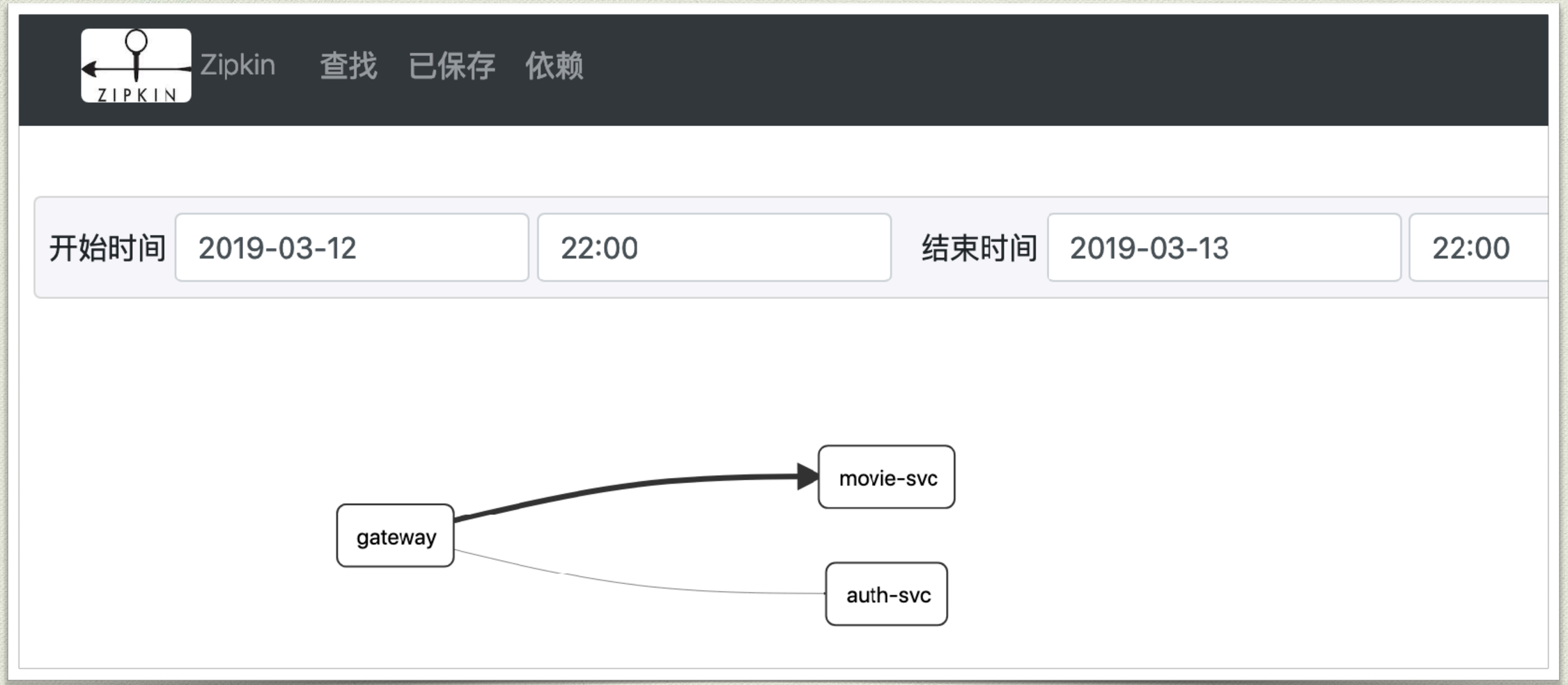
源码： github.com/jskillcloud/MovieApp

Collection

Home

ID	Title	Date Added
1	The Chronicles of Narnia: The Lion, the Witch and the Wardrobe	2019-03-13T13:59:24Z
2	The Lion King	2019-03-13T13:59:26Z
3	Lion	2019-03-13T13:59:30Z

源码：github.com/jskillcloud/MovieApp



源码：github.com/jskillcloud/MovieApp

二、JWT简介

什么是JWT

- JSON Web Token(JWT)是**RFC 7519**定义的一个开放标准，它定义了一种**紧凑**和**自包含**的JSON对象格式，通过它可以在多个实体间安全地传输信息。信息经过签名，**可信任**和**可校验**。主要用于**认证/SSO**和**信息交换**。
- 签名方式
  - HMAC~secret
  - RSA ~ public/private key

类比签名支票

Issuer

两种令牌类型

- 透明令牌
  - By Reference Token
  - 随机字符串，无法猜测，严格服务器集中校验
- 自包含令牌
  - By Value Token
  - 可以包含用户元数据或者声明(claims)，无状态校验
  - JSON Web Token(JWT)

JWT令牌结构

Header • Payload • Signature

JWT令牌结构

Header

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwczovL215LnNwcmluZzJnby5jb20iLCJpYXQiOjE0MzUxNzk2MDMsImV4cCI6MTQzNTE4MTQyMSwiYXVkIjoid3d3LnNwcmluZzJnby5jb20iLCJzdWIiOiJ3aWxsaWFtQGdtYWlsLmNvbSIsIlJvbGUiOlsiYXBwcm92ZXIiLCJ2aWV3ZXIiXX0.ZeyrPdt1unJDEFzcvugOK5NZ80_qCZfMdhyM-rOYjNY

Payload

Signature

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "iss": "https://my.jskillcloud.com",
  "iat": 1435179603,
  "exp": 1435181421,
  "aud": "www.jskillcloud.com",
  "sub": "bobo@jskillcloud.com",
  "Role": [
    "approver",
    "viewer"
  ]
}
```

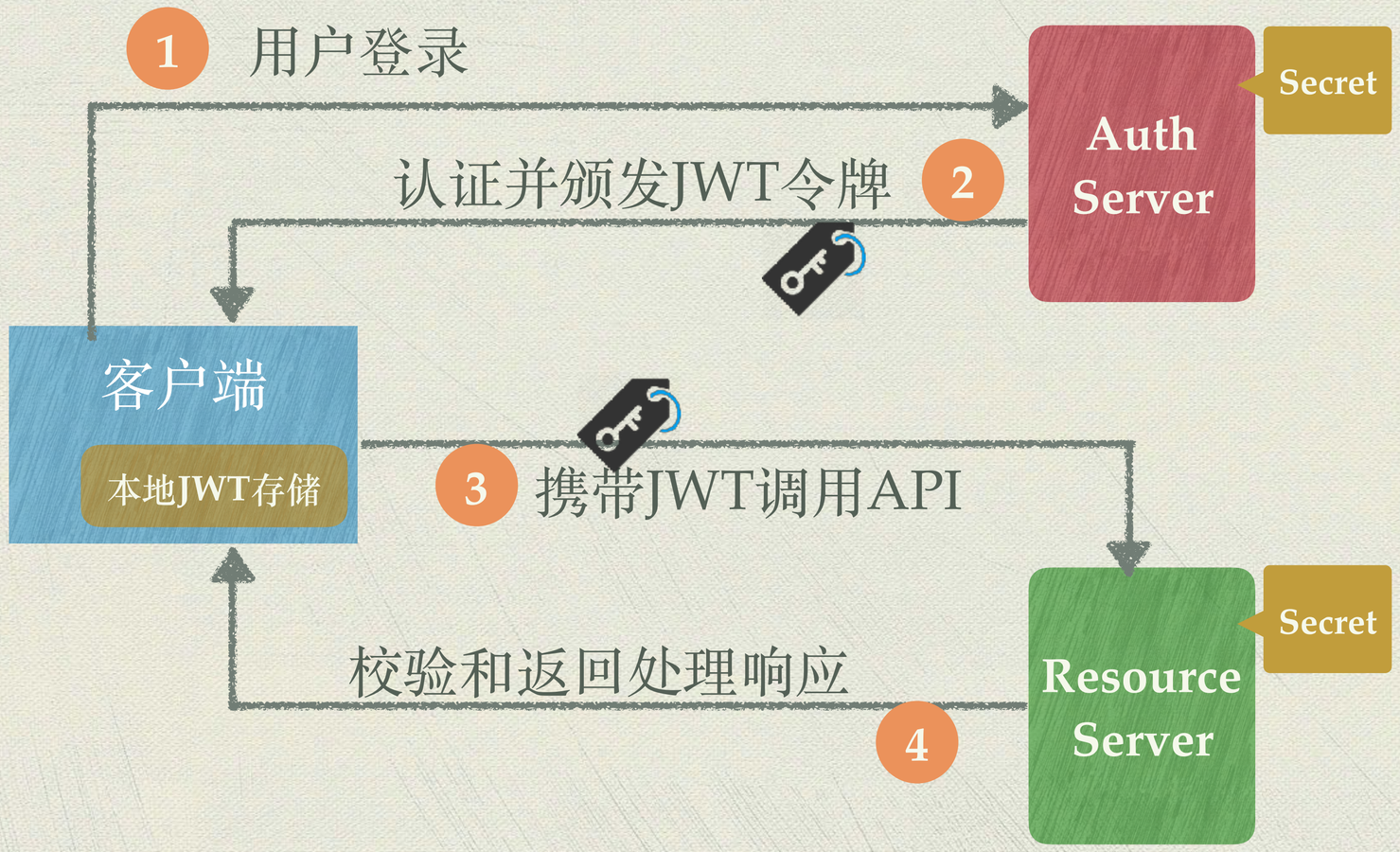
jwt.io

VERIFY SIGNATURE

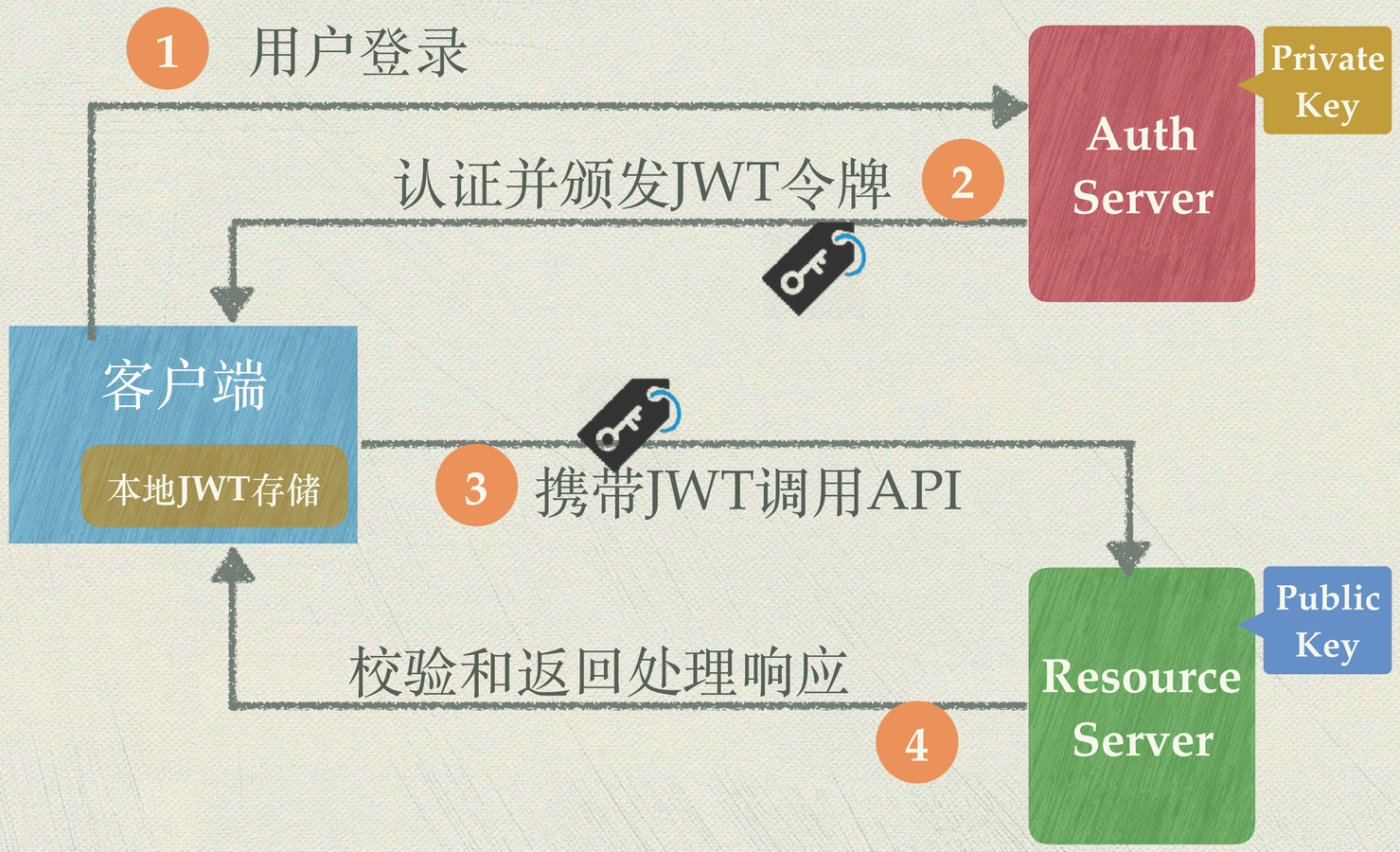
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

base64url(Header) + “.” + base64url(Claims) + “.” + base64url(Signature)

流程HMAC


1
用户登录
Auth Server
Secret
认证并颁发JWT令牌
2
客户端
本地JWT存储
3
携带JWT调用API
Resource Server
Secret
校验和返回处理响应
4

流程RSA


1
用户登录
Auth Server
Private Key
认证并颁发JWT令牌
2
客户端
本地JWT存储
3
携带JWT调用API
Resource Server
Public Key
校验和返回处理响应
4

JWT优劣

优点

- CORS
- 不需要CSRF保护
- 易于和Mobile集成
- **无状态校验**
  - 不需要分布式会话存储
  - 减少授权服务器压力

不足

- 信息公开可见
- 易于受XSS攻击
- 可能包含过期授权信息
- 令牌大小随信息量增长
- **无状态和吊销互斥**

三、安全设计(Spring Security + JWT)

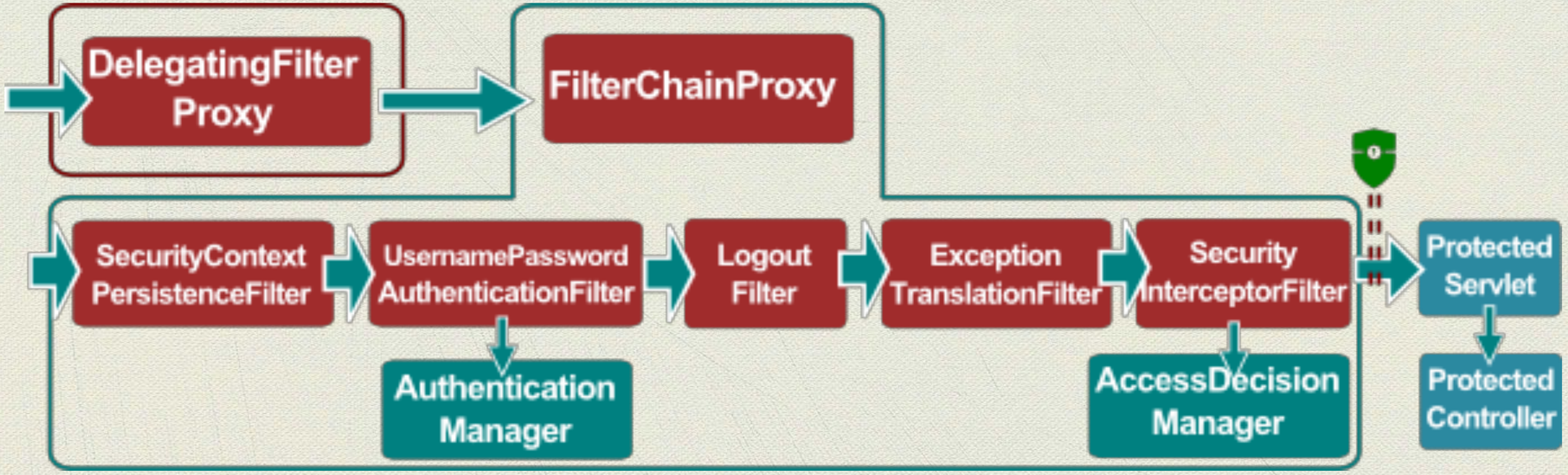
Spring Security基础概念一

- Principal
  - 主体，执行操作的用户
- Authentication
  - 认证，确认机密信息(如用户名/密码)的有效性
- Authorization
  - 授权，定义Principal的访问控制策略

Spring Security基础概念二

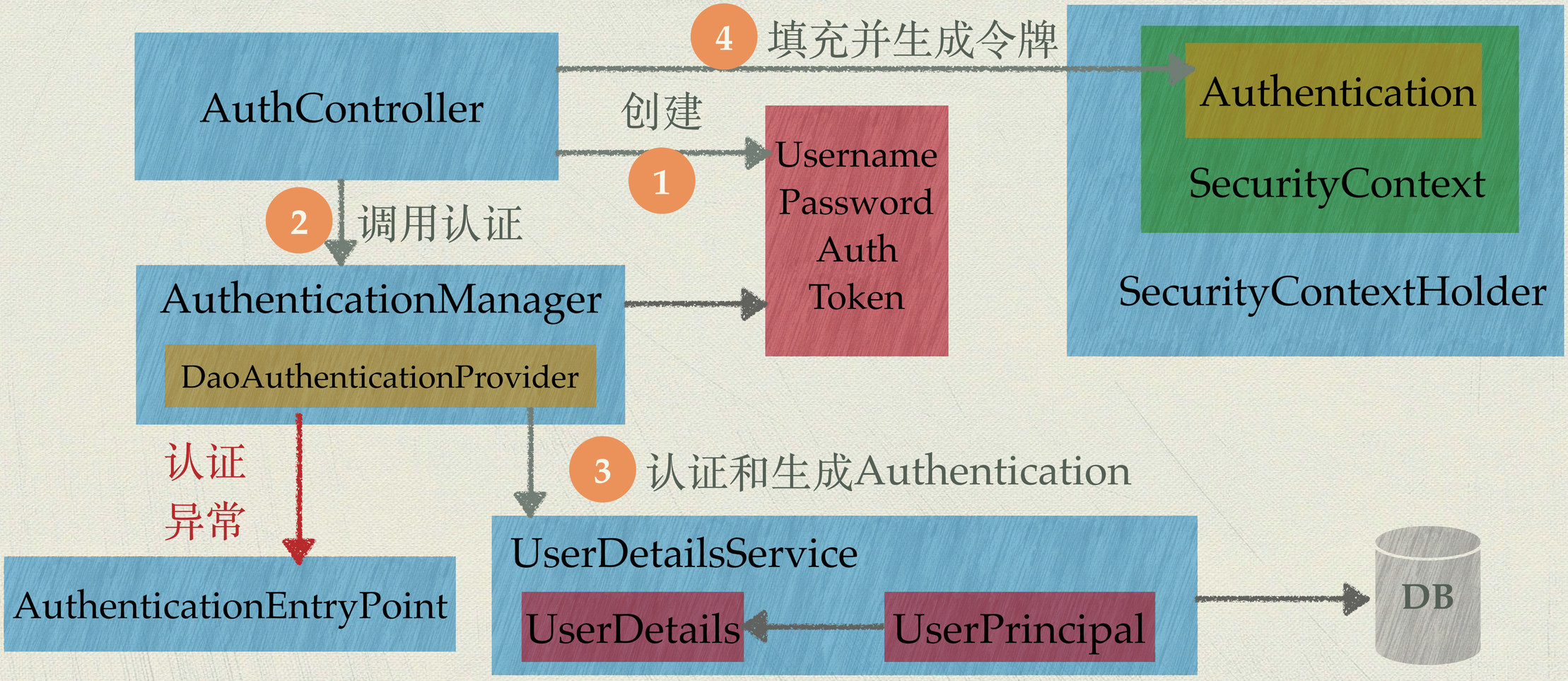
- GrantedAuthority
  - 授予Principal的应用级权限
- SecurityContext
  - 安全上下文，保存Authentication和其它安全信息
- SecurityContextHolder
  - 提供对SecurityContext的访问点

Spring Security过滤链

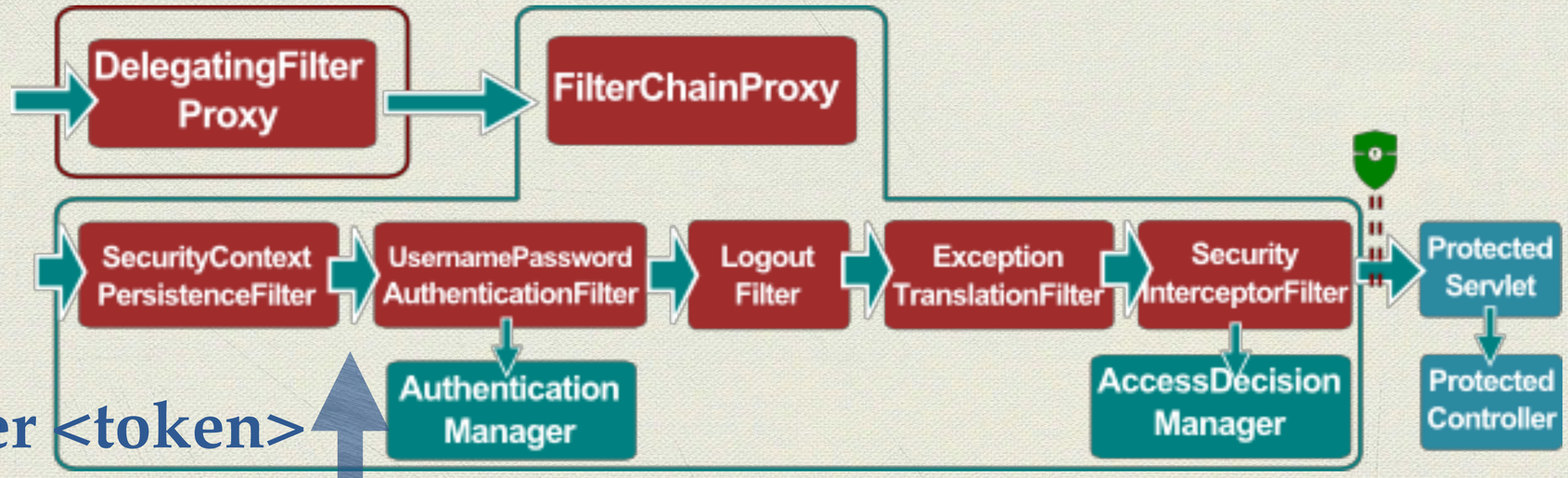


参考：http://www.einnovator.org/document/334/spring-security#web

AuthSvc用户名密码登录认证架构



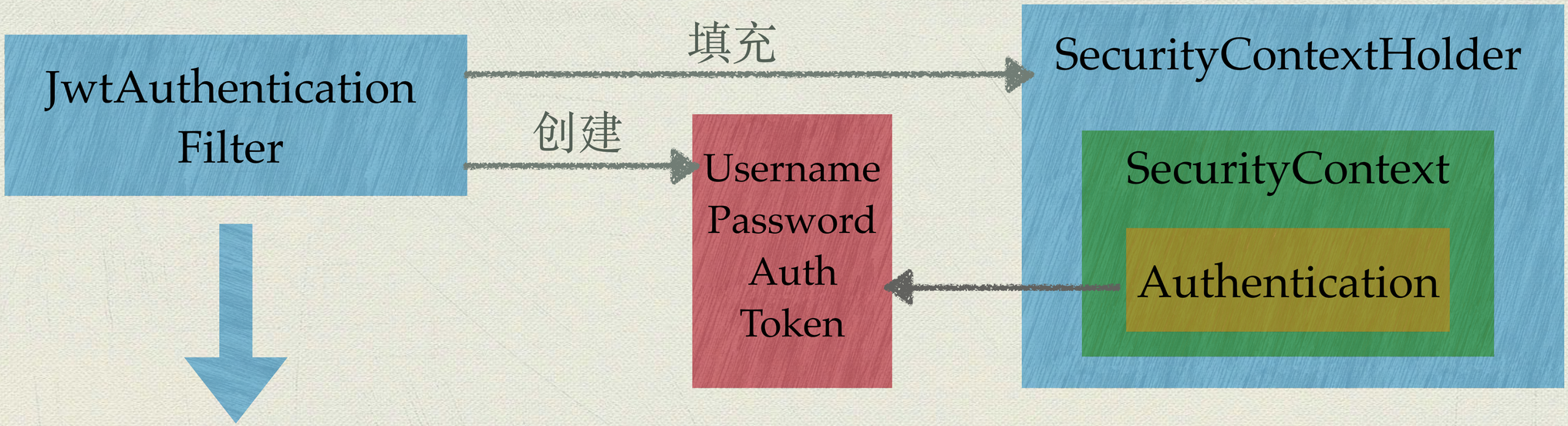
JWT认证



Authorization: Bearer <token>

JwtAuthentication Filter

JWT认证方法



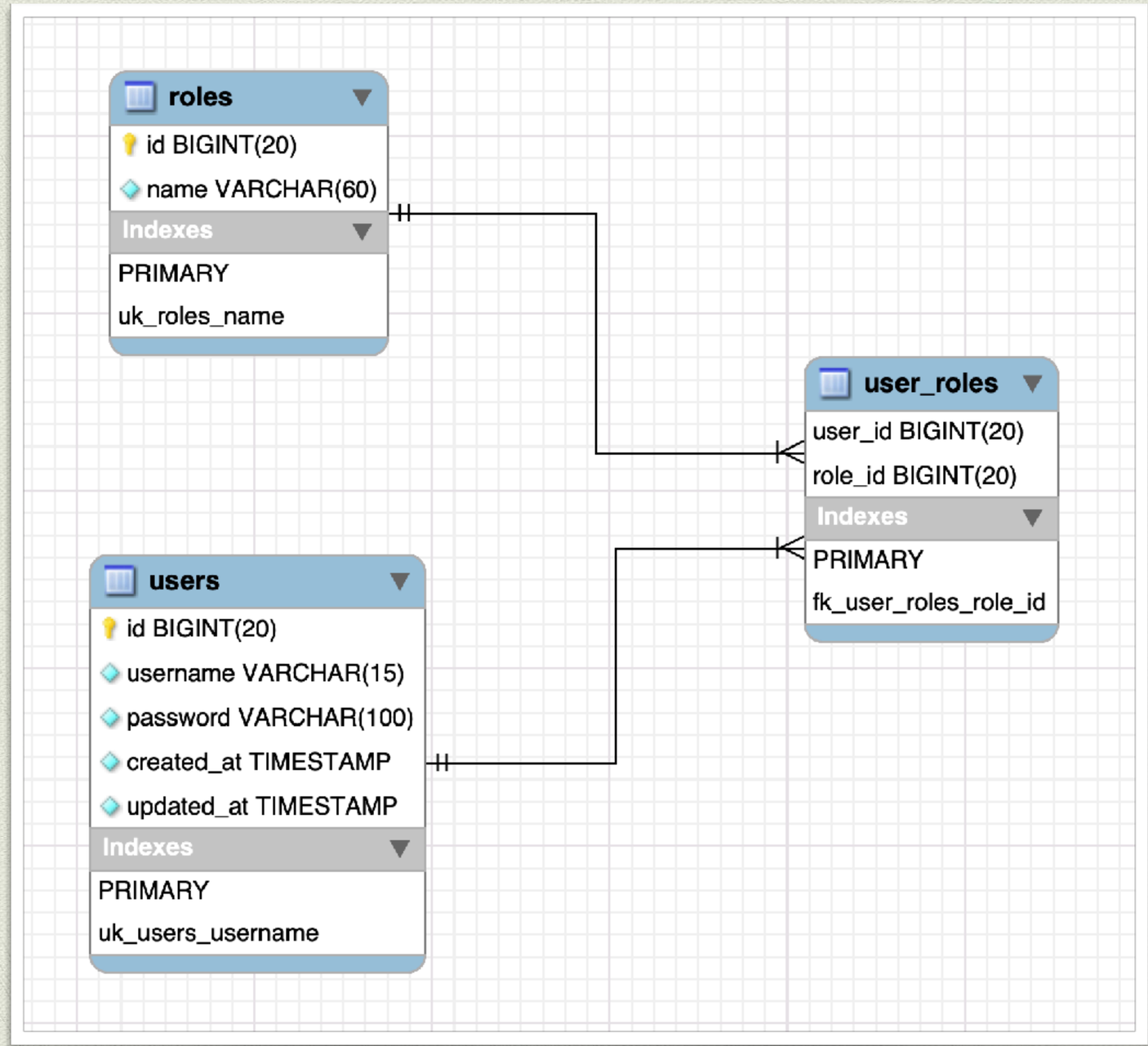
方法一： 通过令牌本身，无状态，参考MovieSvc

方法二： 通过UserDetailsService，参考AuthSvc

方法三： 通过远程AuthSvc用户信息接口

四、AuthSvc代码剖析

DB Schema


roles
id BIGINT(20)
name VARCHAR(60)
Indexes
PRIMARY
uk_roles_name
user_roles
user_id BIGINT(20)
role_id BIGINT(20)
Indexes
PRIMARY
fk_user_roles_role_id
users
id BIGINT(20)
username VARCHAR(15)
password VARCHAR(100)
created_at TIMESTAMP
updated_at TIMESTAMP
Indexes
PRIMARY
uk_users_username

五、AuthSvc演示

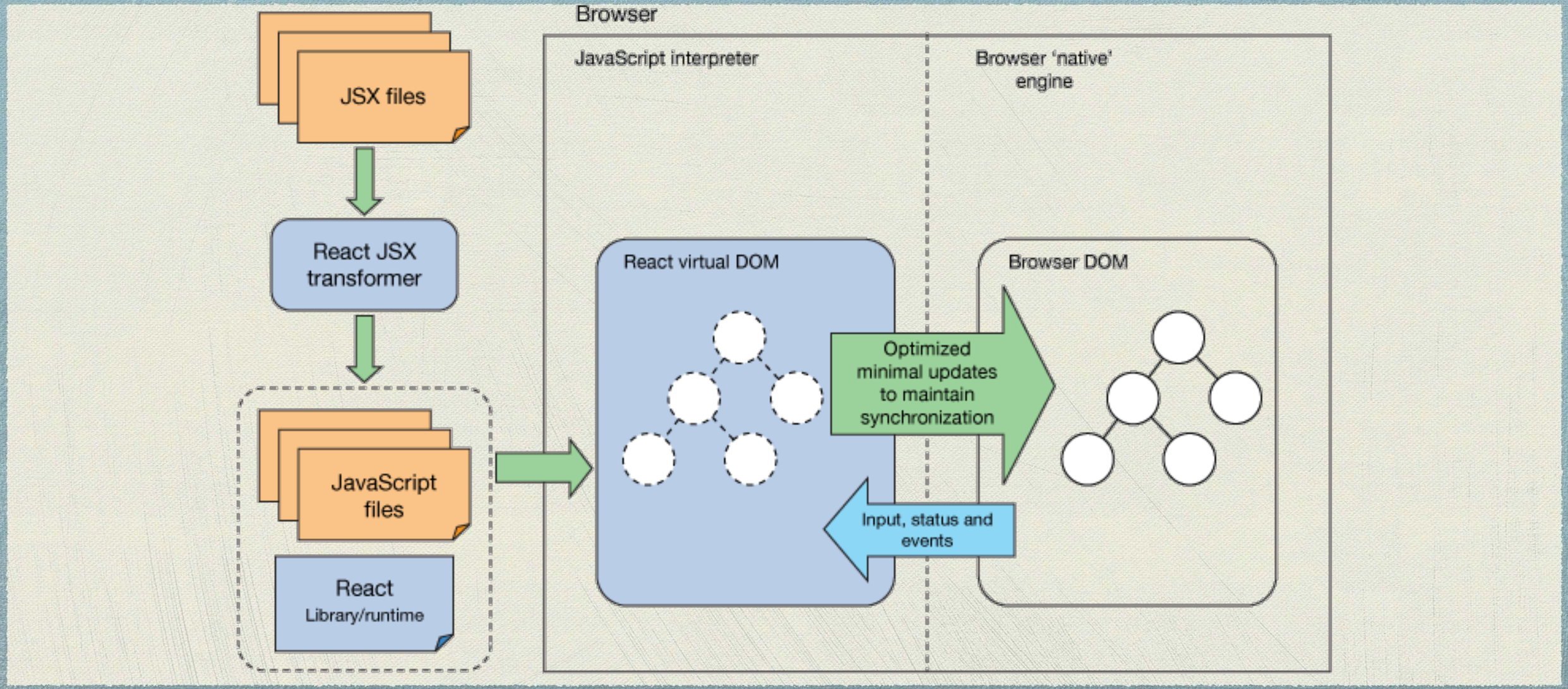
端点和权限

控制器	端点	认证和权限	用途
AuthController	signup	无	注册用户
	signin	无	用户登录和获取令牌
UserController	checkUsernameAvailability	无	检查用户名可用性
	me	令牌认证	获取用户详情
	testuser	令牌认证 USER/ADMIN	演示用户或管理员可访问
	testadmin	令牌认证 ADMIN	演示仅管理员可访问

六、MovieSvc代码剖析

七、前端代码剖析一

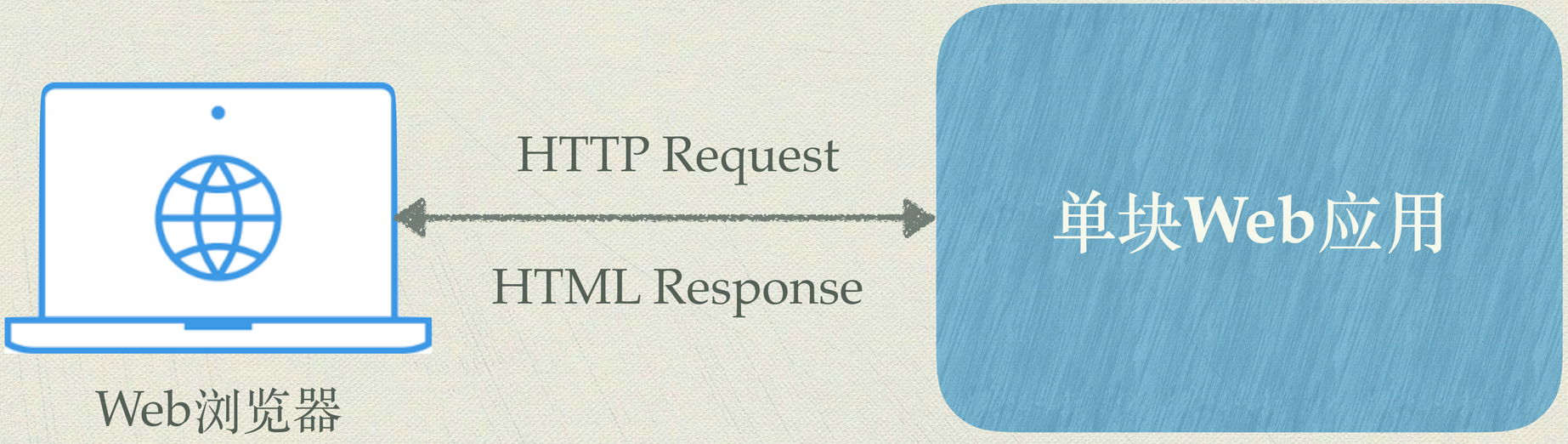
React JS Architecture



八、前端代码剖析二

九、微服务网关和代码剖析

单块应用时代



单块+API时代

Web浏览器+JS

HTTP Request

XML/JSON Response

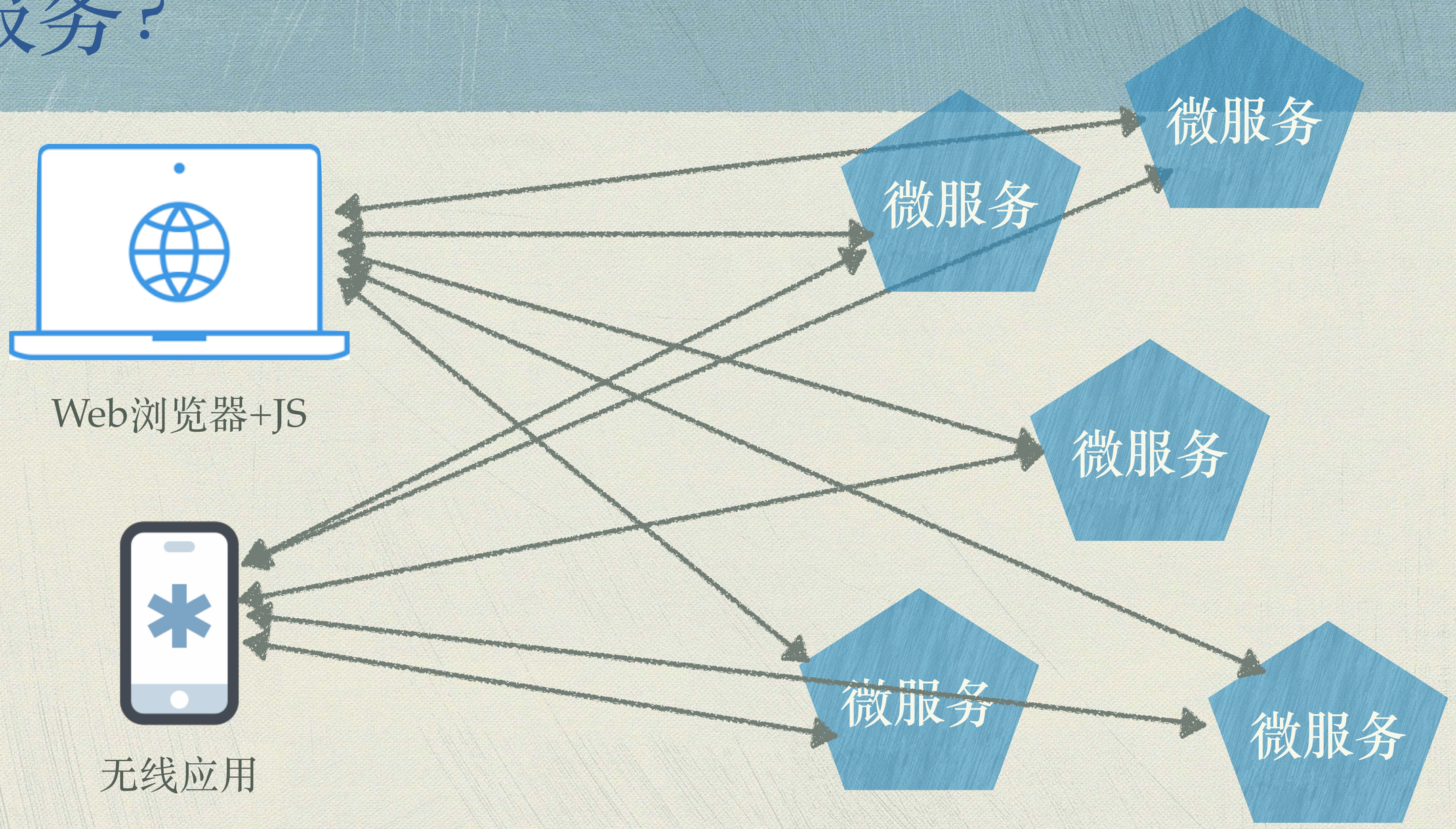
单块Web应用
+REST API

HTTP Request

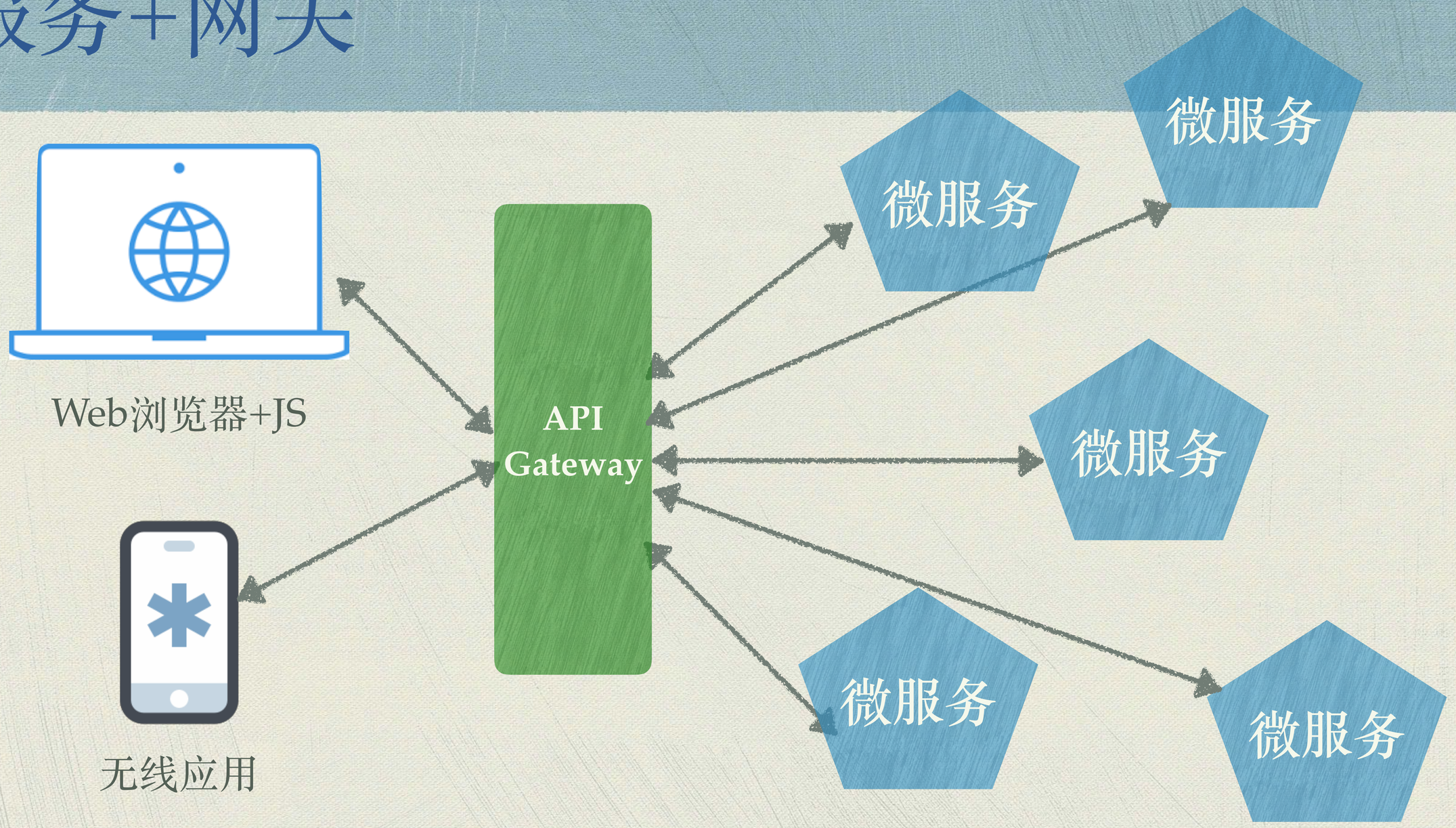
XML/JSON Response

无线应用

微服务？

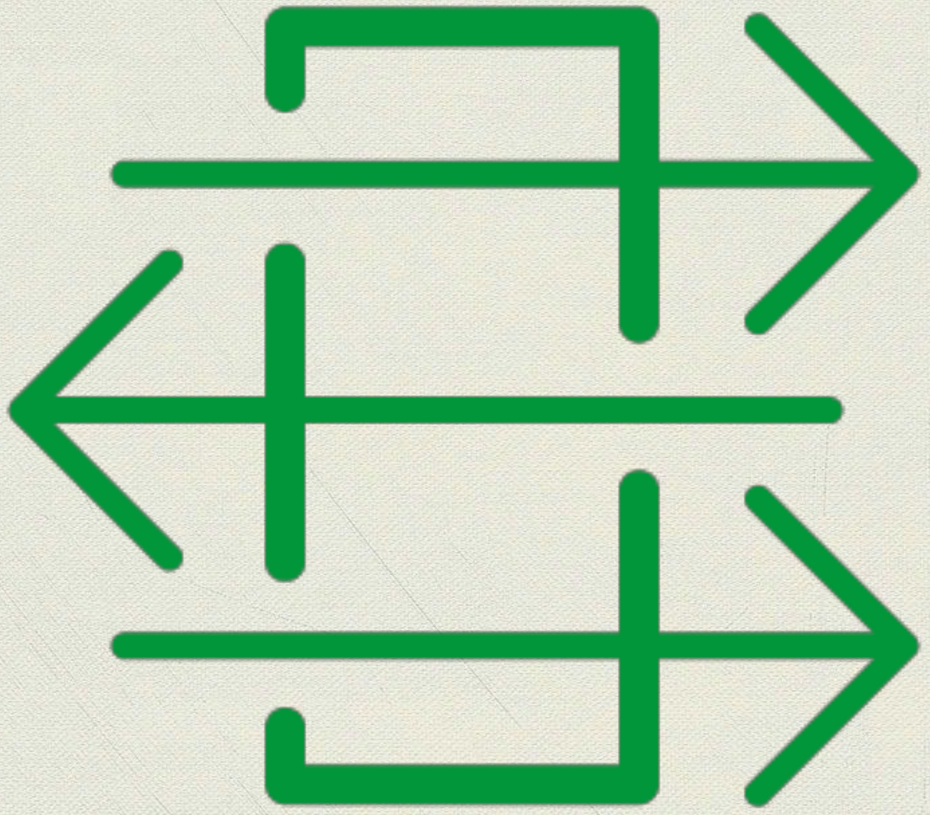


微服务+网关



网关职责

- 路由
- 安全
- 限流熔断
- 监控
- 金丝雀测试
- 单块迁移到微服务



关于Zuul1

- 基于blocking servlet API
- 不支持SSE或者WebSocket
- 路由灵活度一般
- Netflix专有功能

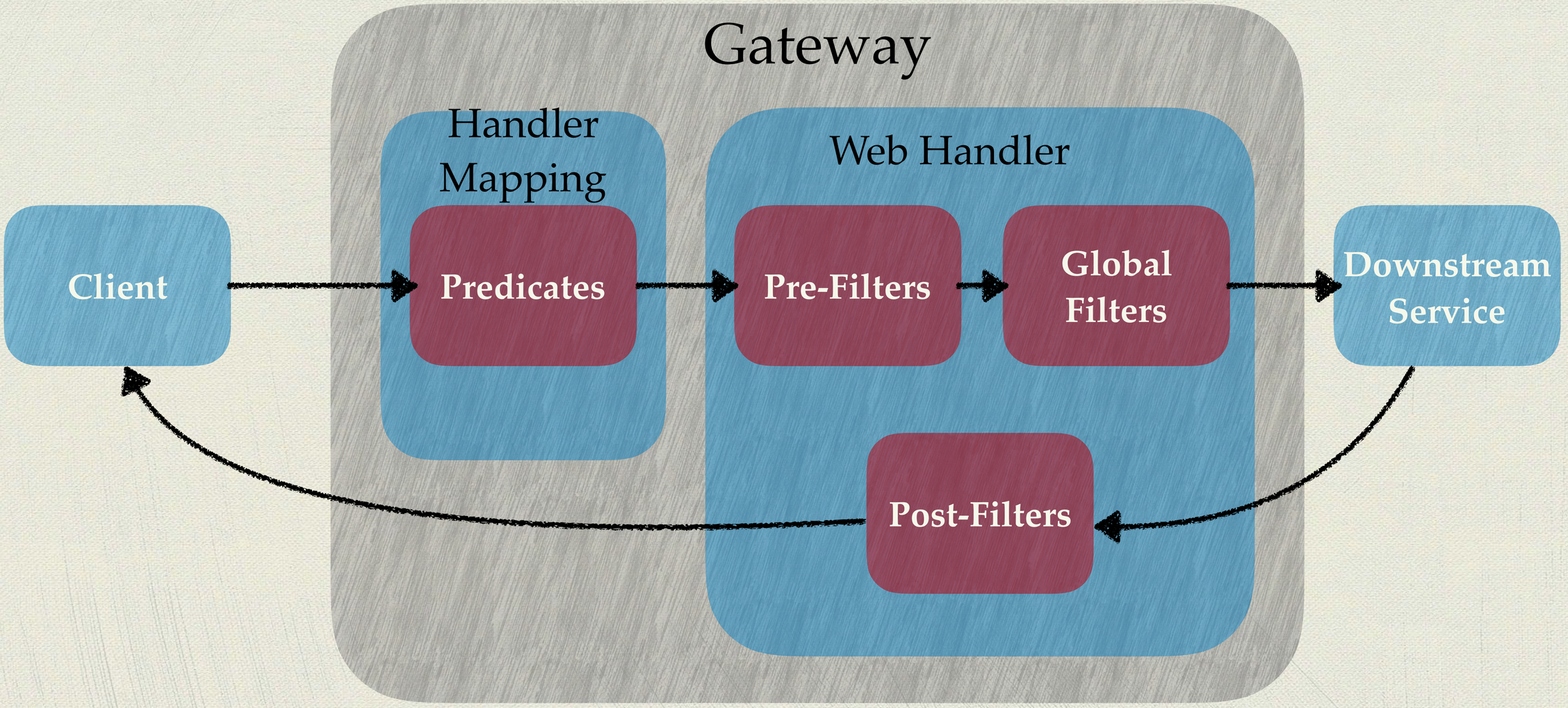
NETFLIX
ZUUL

Spring Cloud Gateway

- 基于Spring, Reactor, Boot 2
- 支持基于配置或Java代码方式的灵活路由
- 路由支持path/host/header/parameters等
- 支持过滤器

https://cloud.spring.io/spring-cloud-gateway/spring-cloud-gateway.html

架构


Gateway
Handler Mapping
Web Handler
Client
Predicates
Pre-Filters
Global Filters
Downstream Service
Post-Filters

简单过滤器

```
@Slf4j
public class SimpleFilter implements GatewayFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        log.info("BEFORE");
        return chain.filter(exchange)
                .then(Mono.fromRunnable(() -> {
                    log.info("AFTER");
                }));
    }

}
```

YAML配置

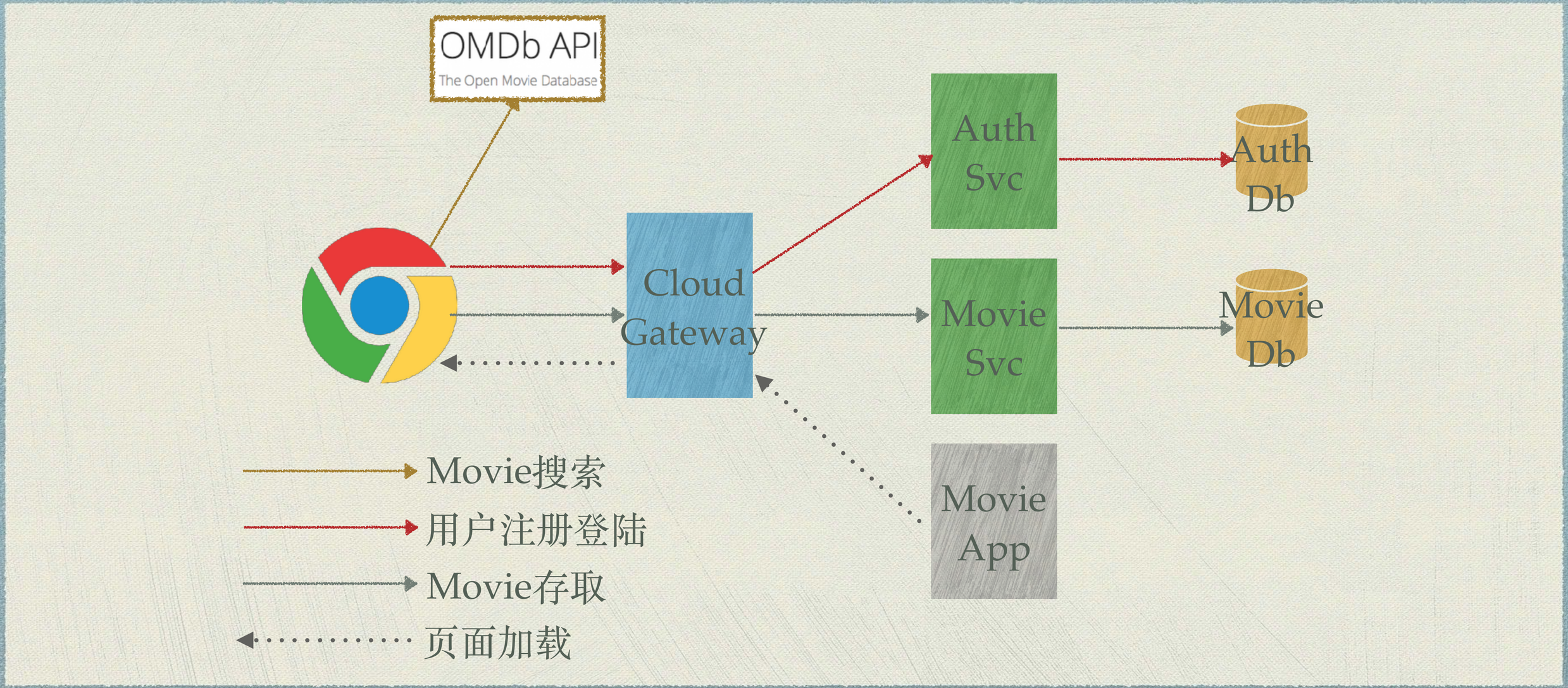
```yaml
spring:
  cloud:
    gateway:
      - id: foo_route
        uri: lb://foo
        predicates:
        - Host=**.foo.org
        - Path=/headers
        - Method=GET
        - Header=X-Request-Id, \d+
        - Query=foo, ba.
        - Query=baz
        - Cookie=chocolate, ch.p
        - After=1900-01-20T17:42:47.789-07:00[America/Denver]
        filters:
        - AddRequestHeader=X-Request-Foo, Bar
        - AddResponseHeader=X-Response-Foo, Bar
        - Hystrix=foo
        - SecureHeaders
        - RewritePath=/foo/(?<segment>.*), /$\{segment}
```

Java配置

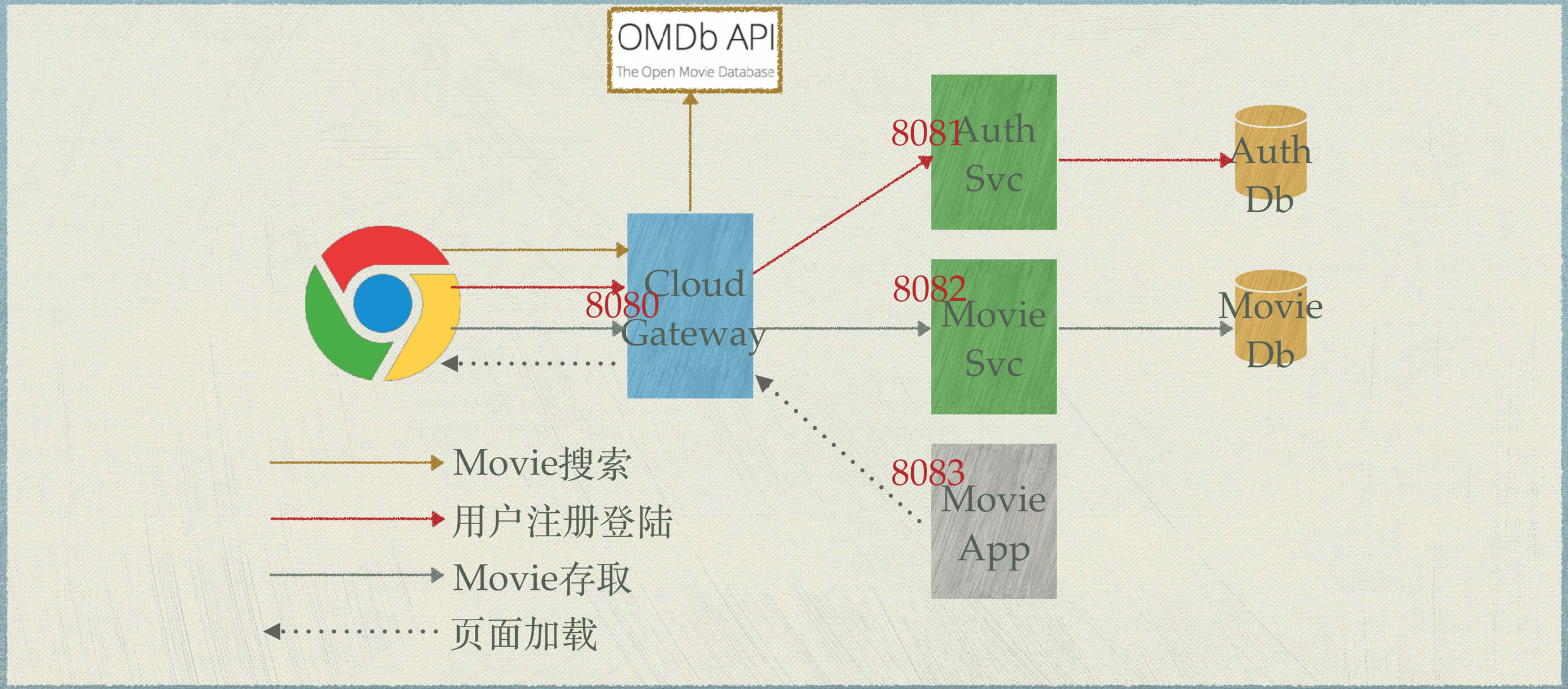
```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder,
        ThrottleGatewayFilterFactory throttle) {
    return builder.routes()
        .route(r -> r.host("**.abc.org").and().path("/image/png")
            .filters(f -> f.addResponseHeader("X-TestHeader", "foobar"))
            .uri("http://httpbin.org:80")
        )
        .route(r -> r.path("/image/webp")
            .filters(f -> f.addResponseHeader("X-AnotherHeader", "baz"))
            .uri("http://httpbin.org:80")
        )
        .route(r -> r.order(-1)
            .host("**.throttle.org").and().path("/get")
            .filters(f -> f.filter(throttle.apply(1, 1, 10,
                    TimeUnit.SECONDS)))
            .uri("http://httpbin.org:80")
        )
        .build();
}
```

十、总体项目演示

项目架构1.0


OMDb API
The Open Movie Database
Cloud Gateway
Auth Svc
Movie Svc
Movie App
Auth Db
Movie Db
Movie搜索
用户注册登陆
Movie存取
页面加载

项目架构1.1Update

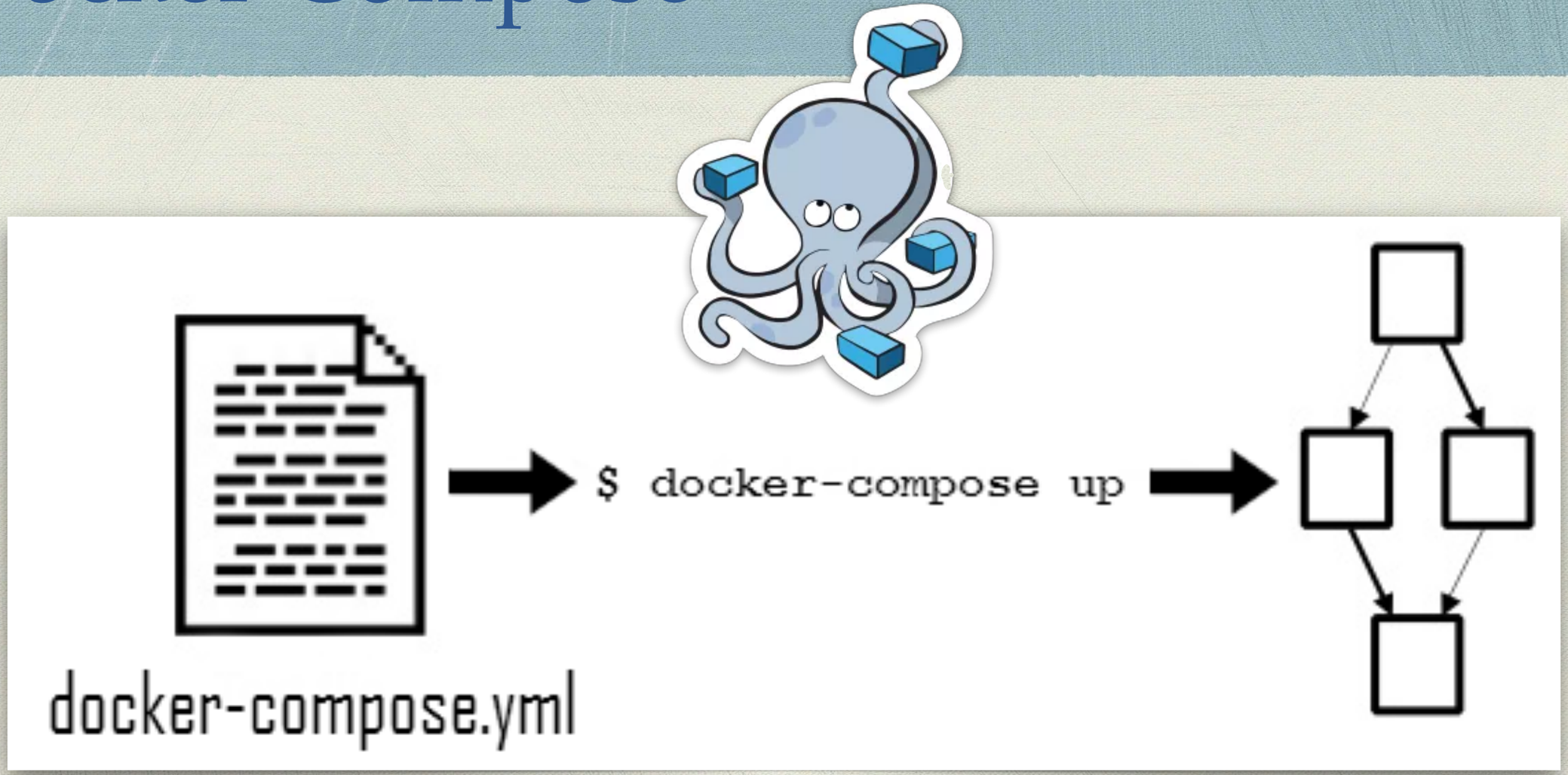


Check List

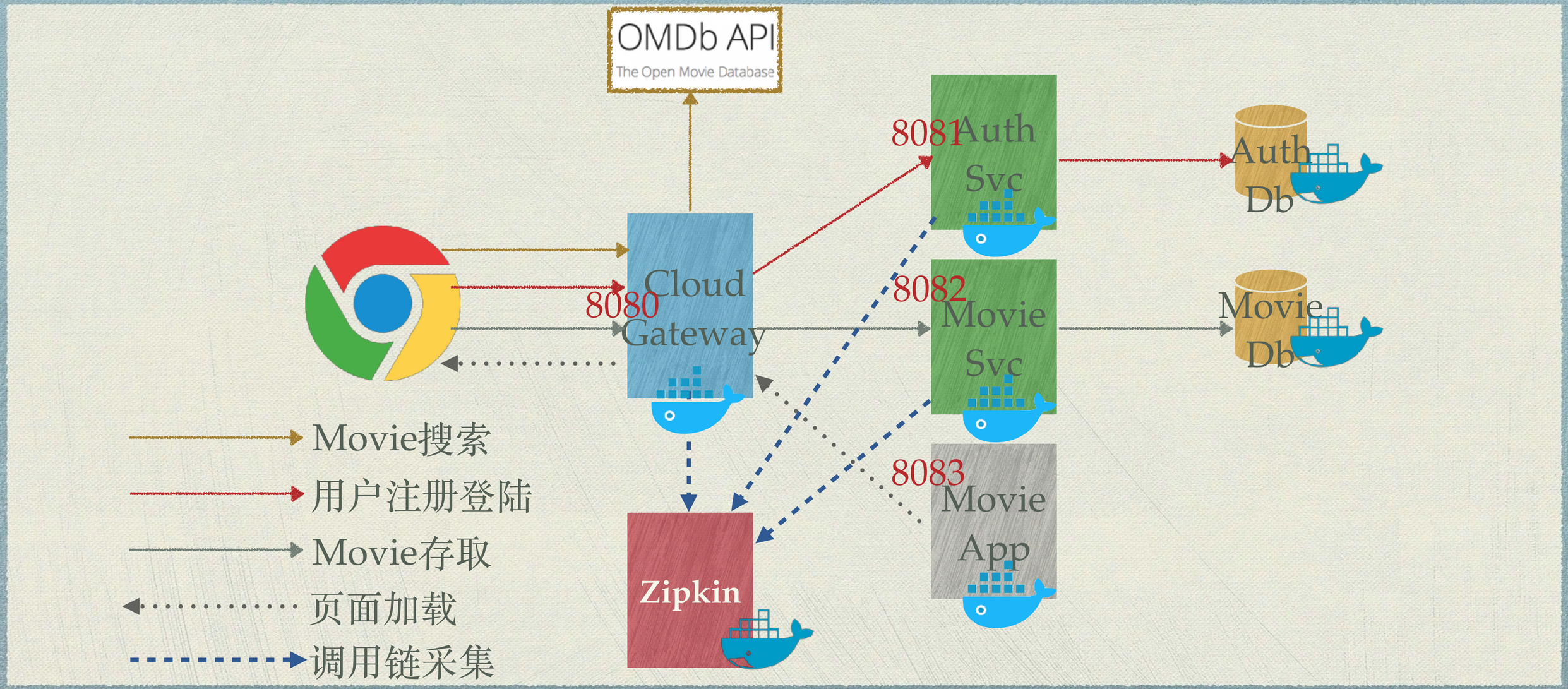
1. 创建auth_db MySQL数据库 ~ 3306
2. 创建movie_db MySQL数据库 ~ 3306
3. 启动auth_svc Spring Boot应用 ~ 8081
4. 启动movie_svc Spring Boot应用 ~ 8082
5. 启动gateway Spring Boot应用 ~ 8080
6. 启动movie_app React应用 ~ 8083
7. 校验UI ~ 8080

十一、Docker Compose部署文件剖析

Docker Compose



项目架构1.2Update



十二、镜像构建和本地部署演示

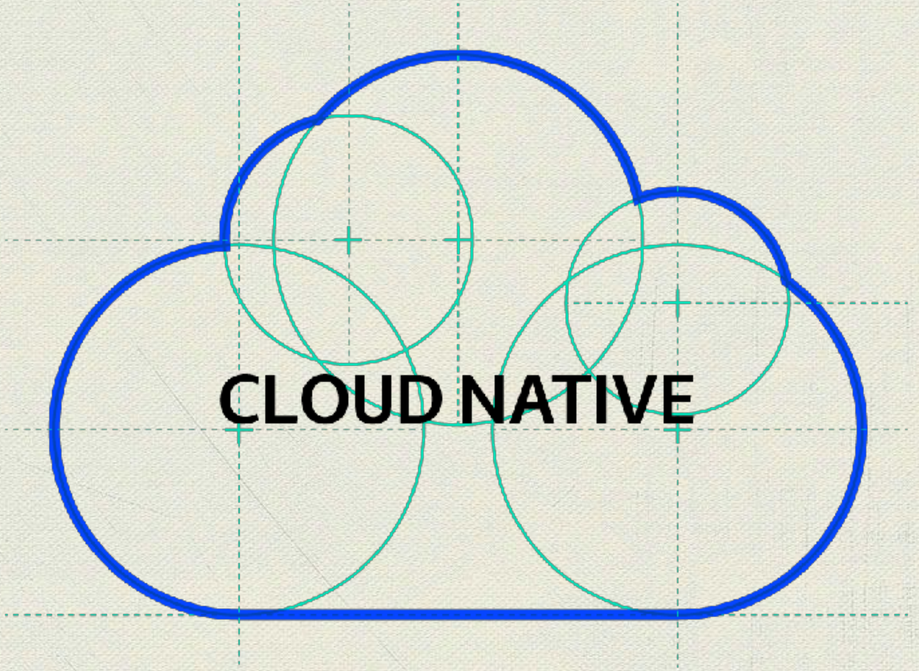
Check List

1. 构建Spring Boot应用(**mvn clean install -DskipTests**)
2. 构建镜像(**docker-compose build**)
3. 一键部署(**docker-compose up**)
4. 校验MovieApp UI ~ **8080**
5. 校验Zipkin UI ~ **9411**
6. 一键关闭(**docker-compose down**)

十三、云原生和K8s

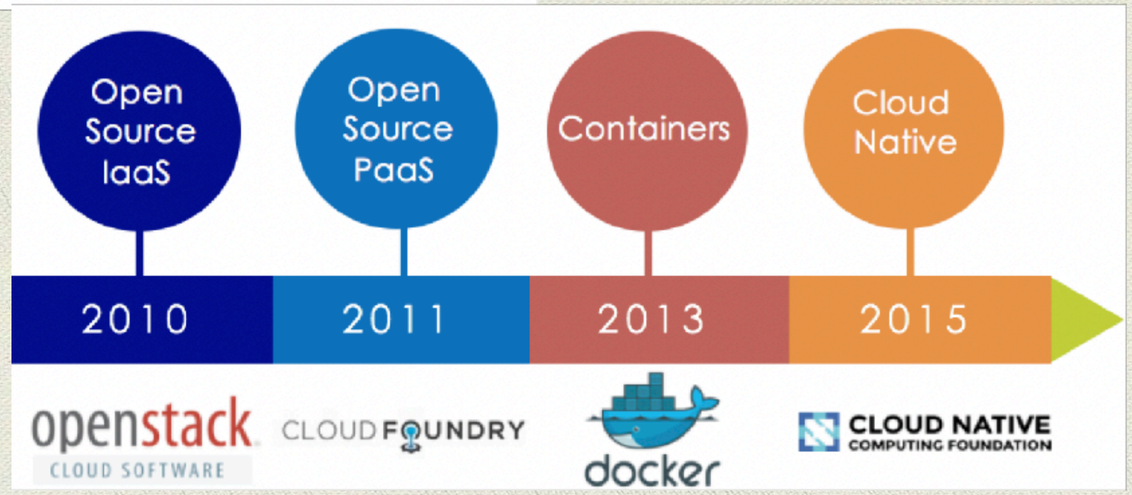
云原生(Cloud Native)应用定义

- **Applications** adopting the principles of **Microservices** packaged as **Containers** orchestrated by **Platforms** running on top of **Cloud infrastructure**, developed using practices such as **Continous Delivery** and **DevOps**

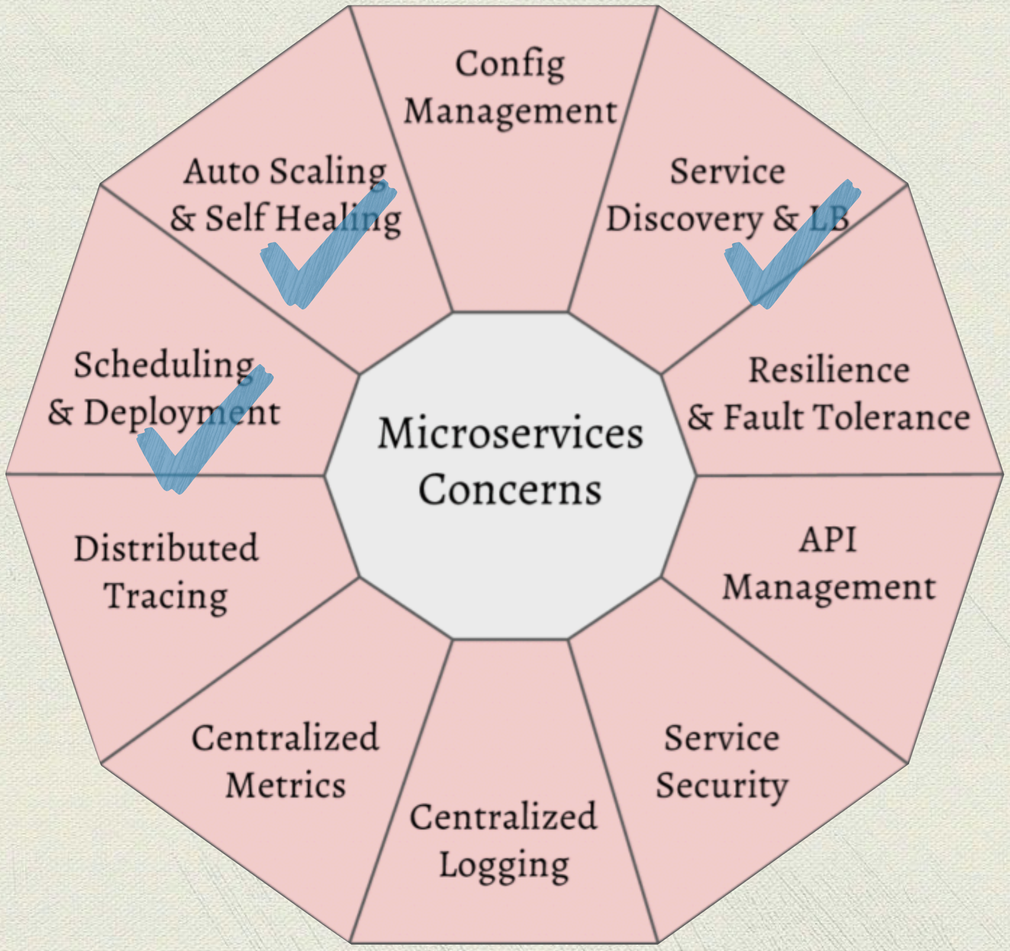


云原生演进史

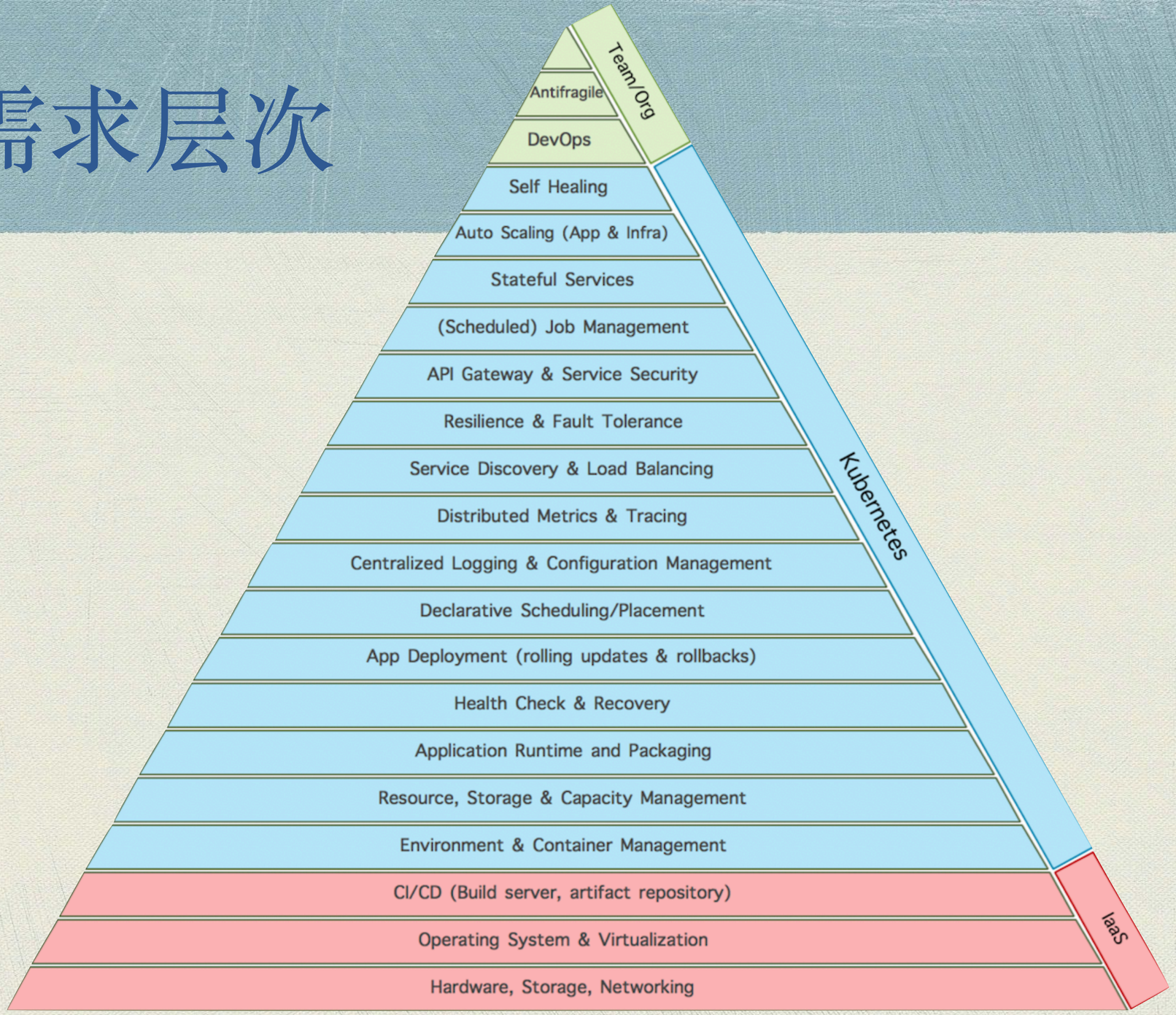
Non-Virtualized Hardware
Virtualiza-tion
IaaS
PaaS
2000
2001
2006
2009
Sun microsystems
vmware
amazon webservices
HEROKU


Open Source IaaS
Open Source PaaS
Containers
Cloud Native
2010
2011
2013
2015
openstack CLOUD SOFTWARE
CLOUD FOUNDRY
docker
CLOUD NATIVE COMPUTING FOUNDATION

微服务关注点

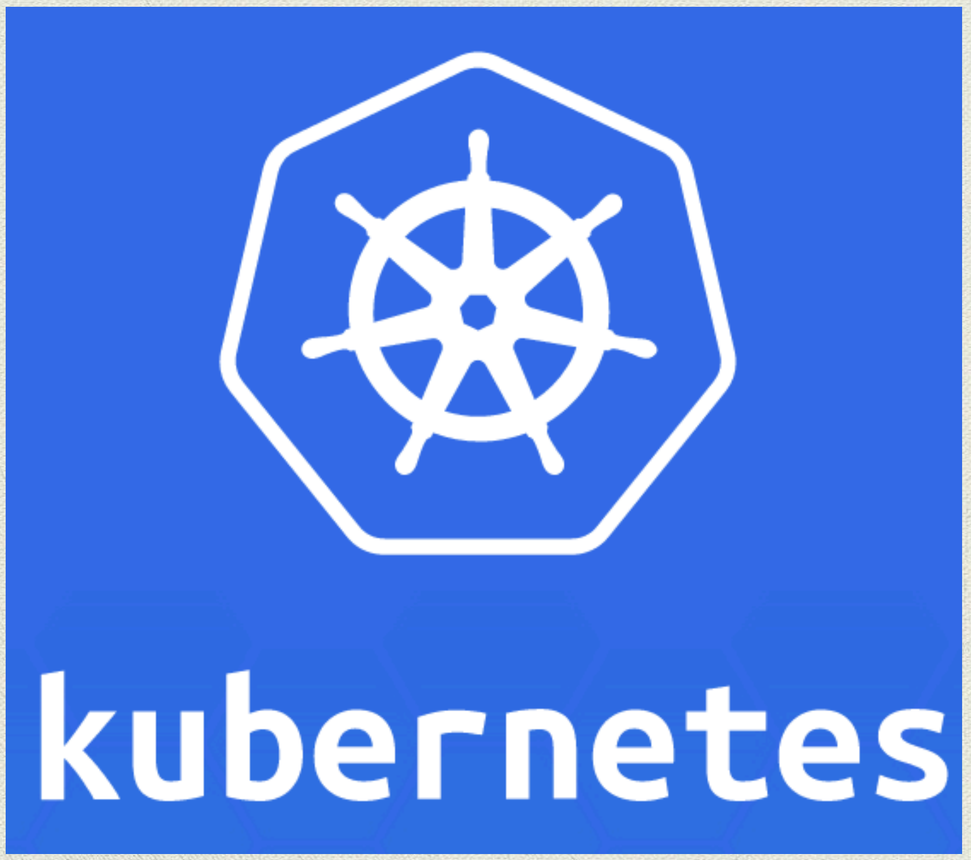

Microservices Concerns
Config Management
Service Discovery & LB
Resilience & Fault Tolerance
API Management
Service Security
Centralized Logging
Centralized Metrics
Distributed Tracing
Scheduling & Deployment
Auto Scaling & Self Healing

微服务需求层次

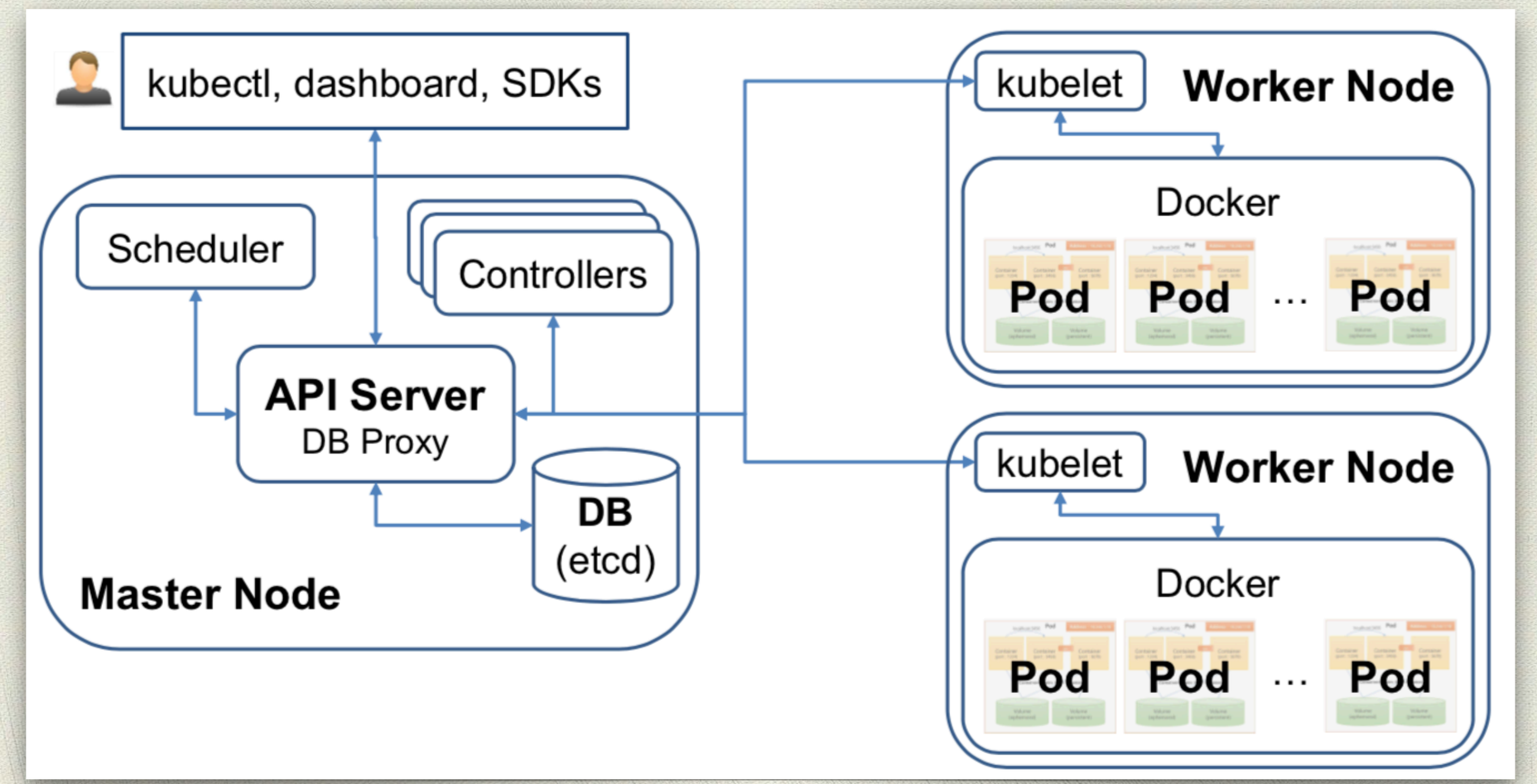

Team/Org
Antifragile
DevOps
Self Healing
Auto Scaling (App & Infra)
Stateful Services
(Scheduled) Job Management
API Gateway & Service Security
Resilience & Fault Tolerance
Service Discovery & Load Balancing
Distributed Metrics & Tracing
Centralized Logging & Configuration Management
Declarative Scheduling/Placement
App Deployment (rolling updates & rollbacks)
Health Check & Recovery
Application Runtime and Packaging
Resource, Storage & Capacity Management
Environment & Container Management
CI/CD (Build server, artifact repository)
Operating System & Virtualization
Hardware, Storage, Networking
Kubernetes
IaaS

K8s定义

- **Open source platform** for **deployment** and **management** of arbitrary **container-based**, **cloud native** applications on clusters of **servers**



K8s架构


kubectl, dashboard, SDKs
Scheduler
Controllers
API Server
DB Proxy
DB
(etcd)
Master Node
kubelet
Worker Node
Docker
Pod
Pod
…
Pod
kubelet
Worker Node
Docker
Pod
Pod
…
Pod

参考

- Designing Cloud Native Application with Kubernetes
  - https://www.slideshare.net/bibryam/designing-cloud-native-applications-with-kubernetes
- Cloud Native Application & Kubernetes
  - https://www.slideshare.net/AlexGlikson/cloudnative-application-and-kubernetes

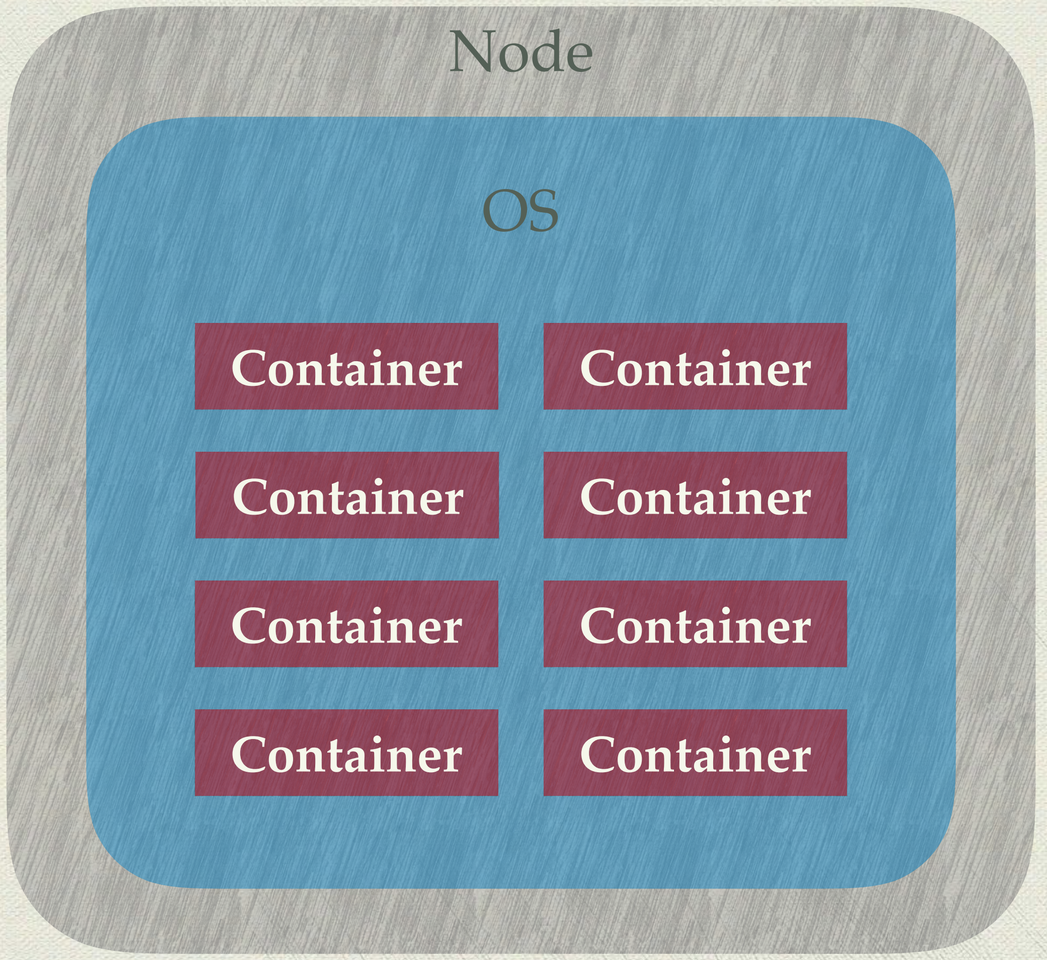
CLOUD NATIVE
COMPUTING FOUNDATION

cncf.io

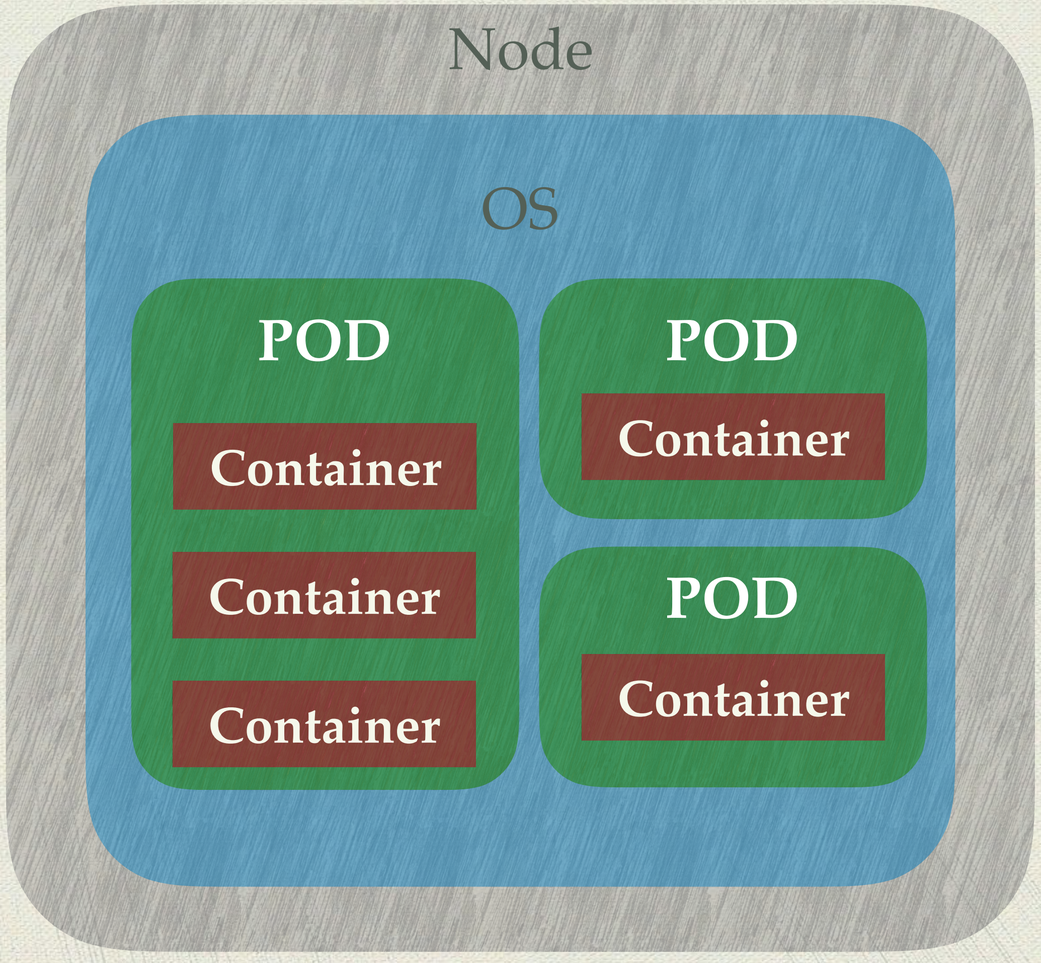
landscape.cncf.io

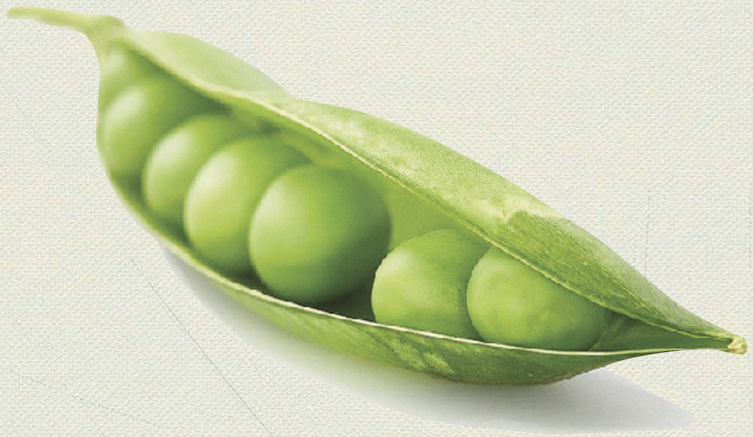
十四、K8s核心概念

容器

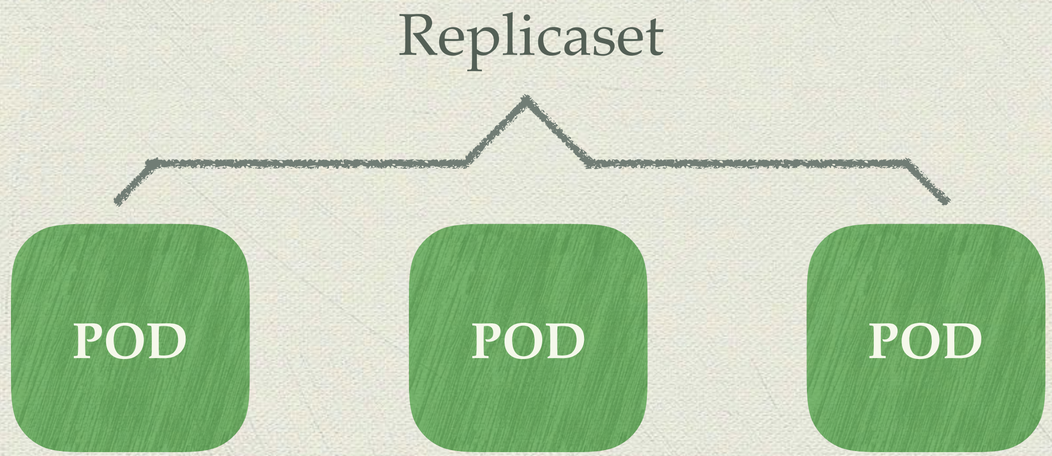

Node
OS
Container
Container
Container
Container
Container
Container
Container
Container

POD

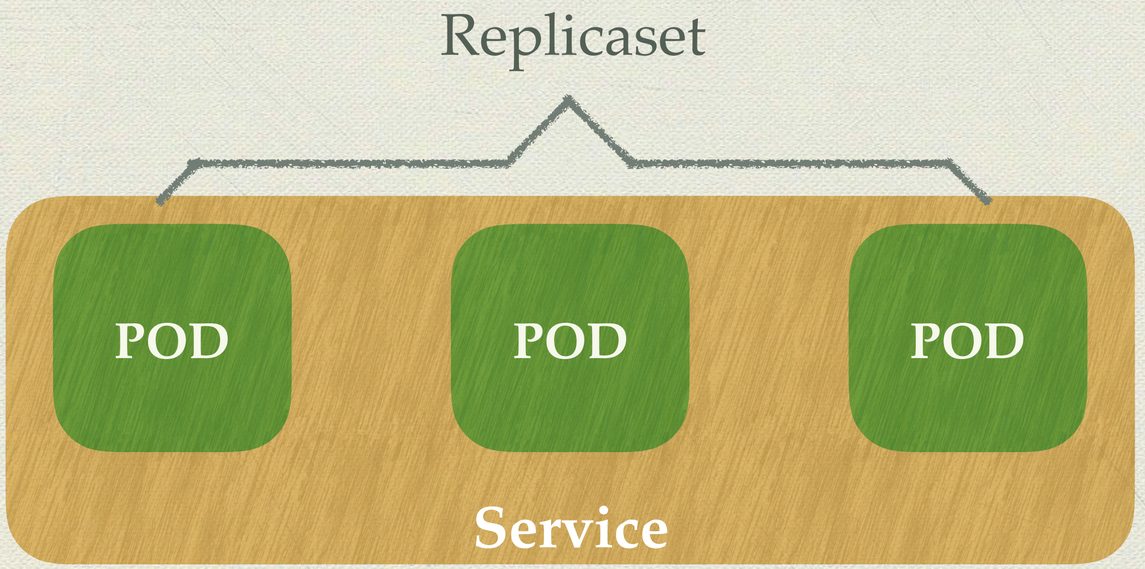

Node
OS
POD
Container
Container
Container
POD
Container
POD
Container



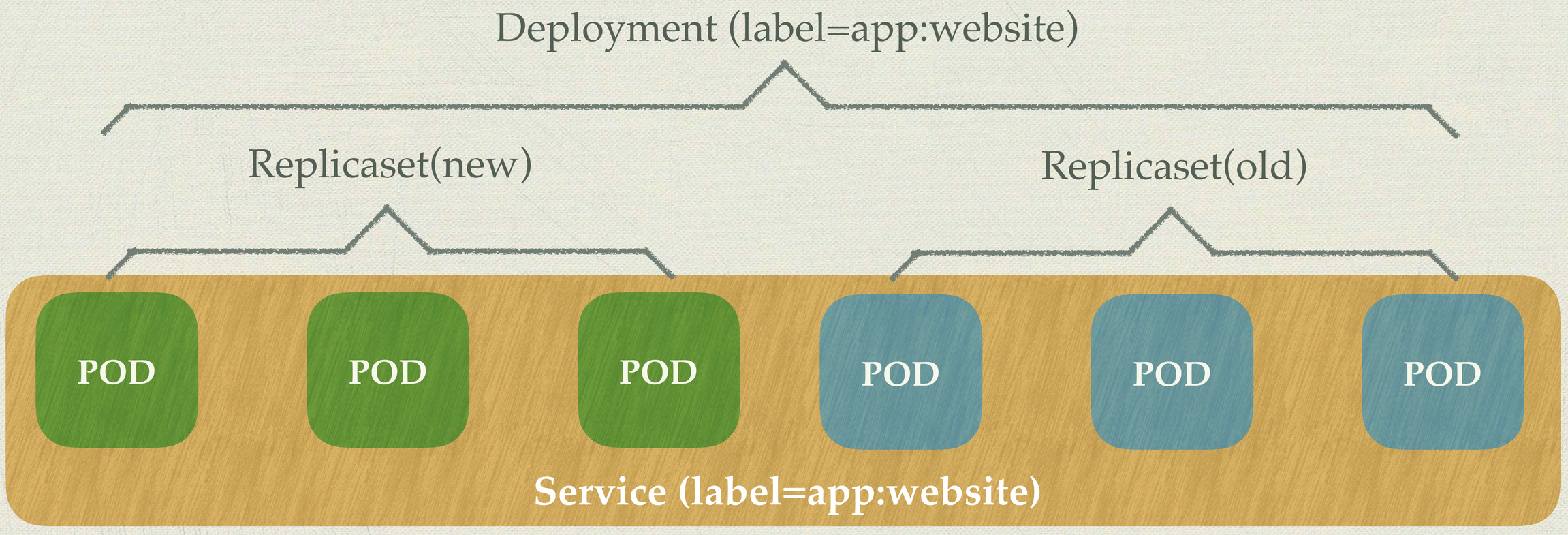
Replicaset



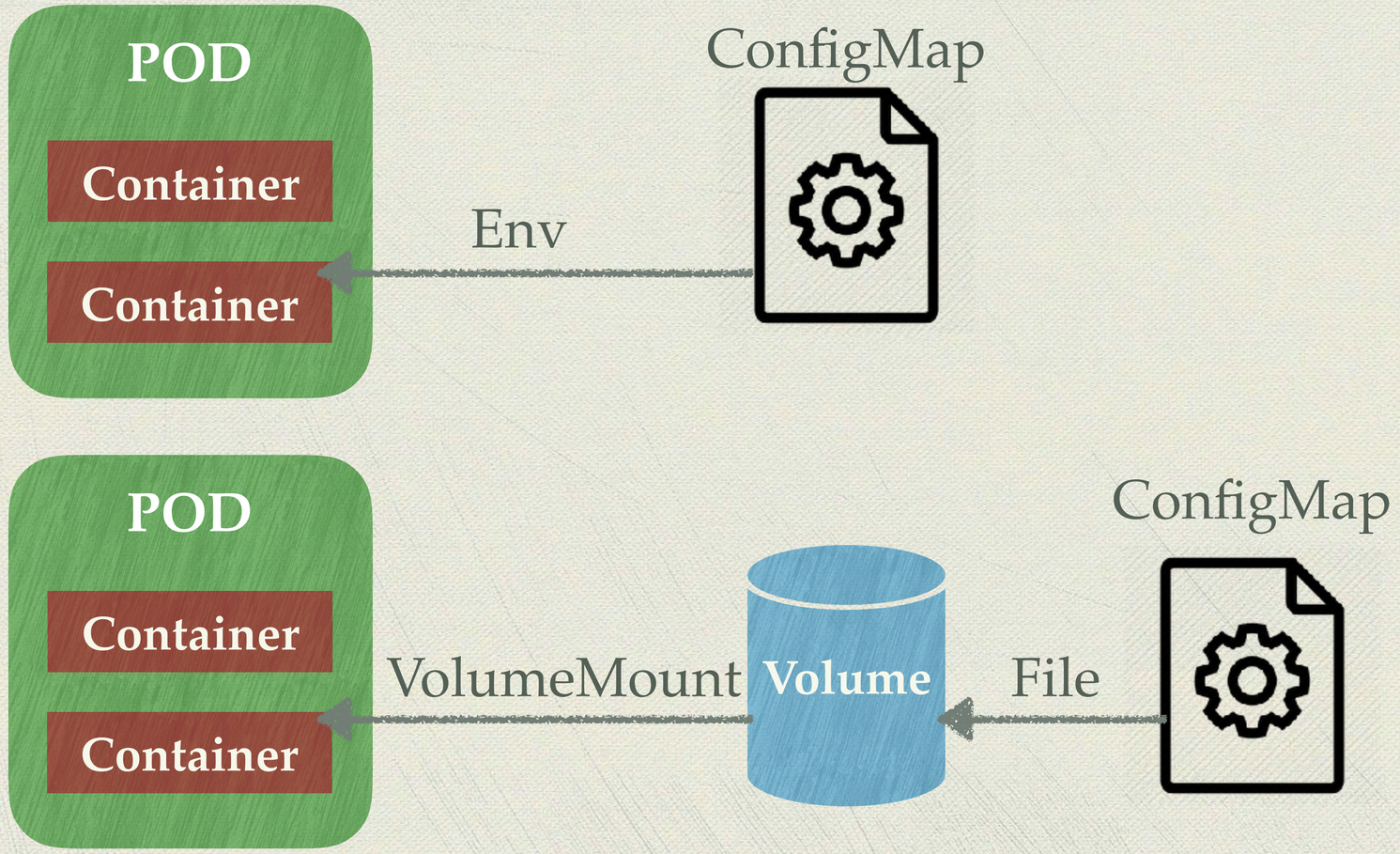
Service



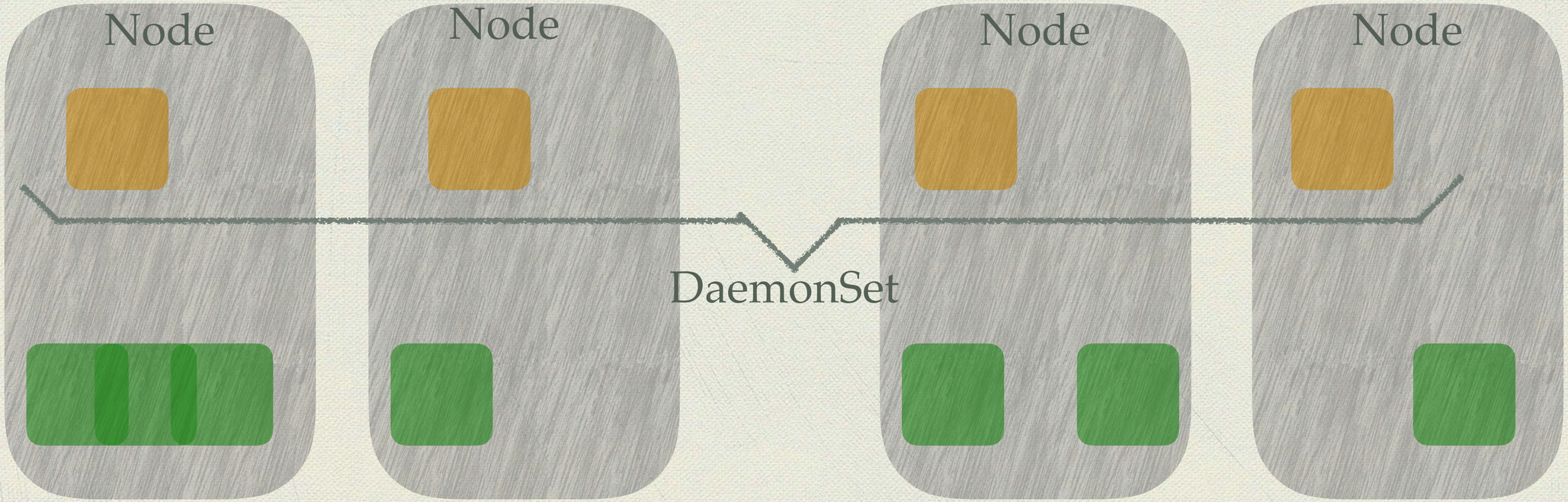
Deployment


Deployment (label=app:website)
Replicaset(new)
Replicaset(old)
POD
POD
POD
POD
POD
POD
Service (label=app:website)

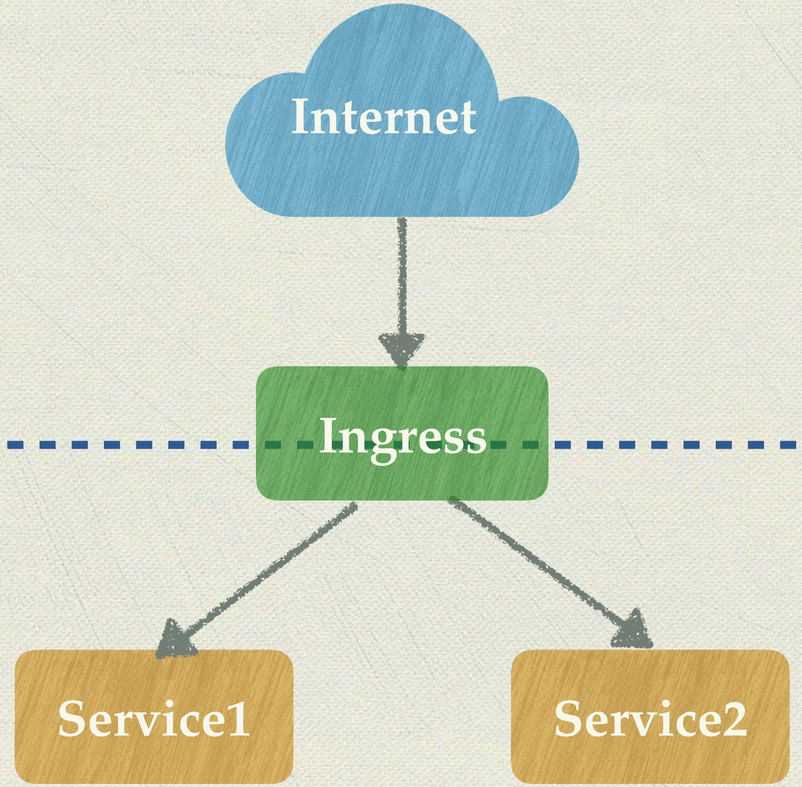
ConfigMap / Secret



DaemonSet

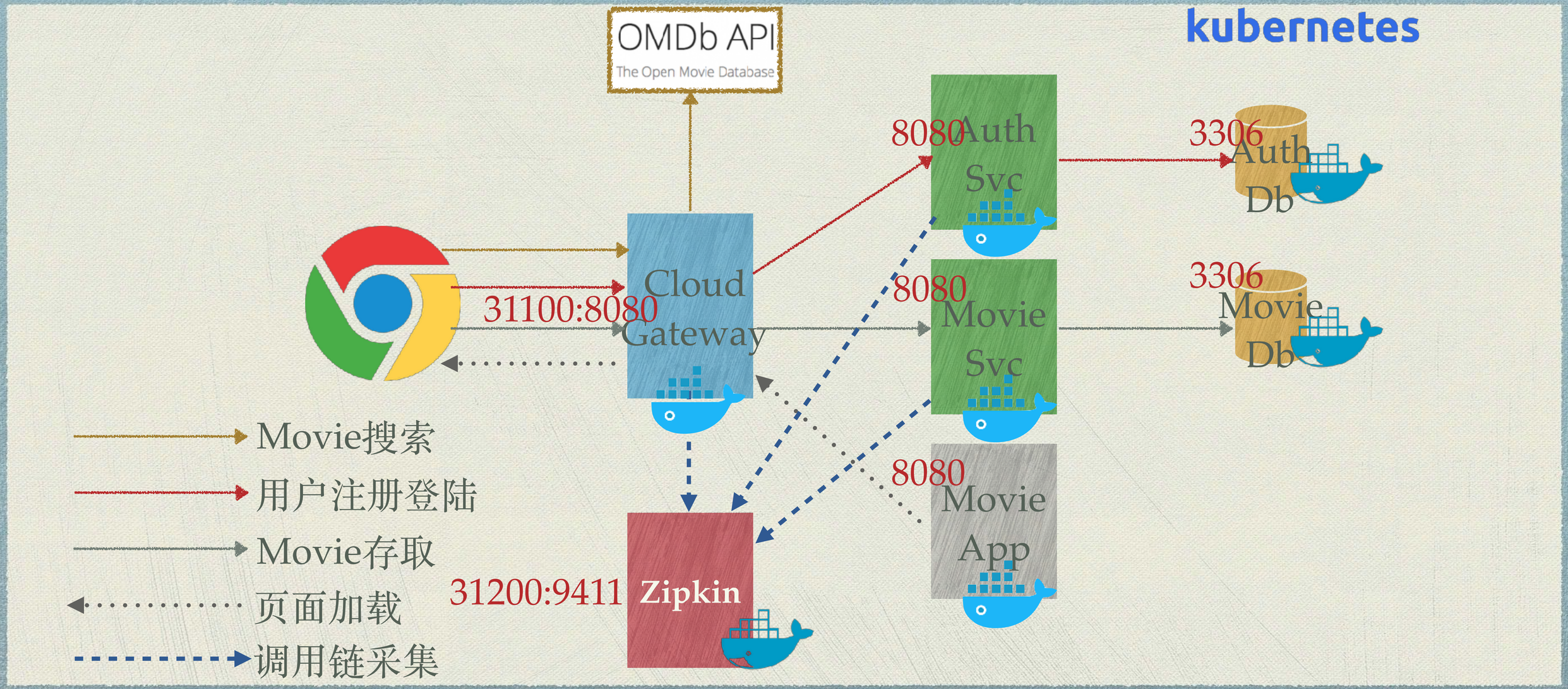


Ingress / LoadBalancer



十五、K8s部署文件剖析

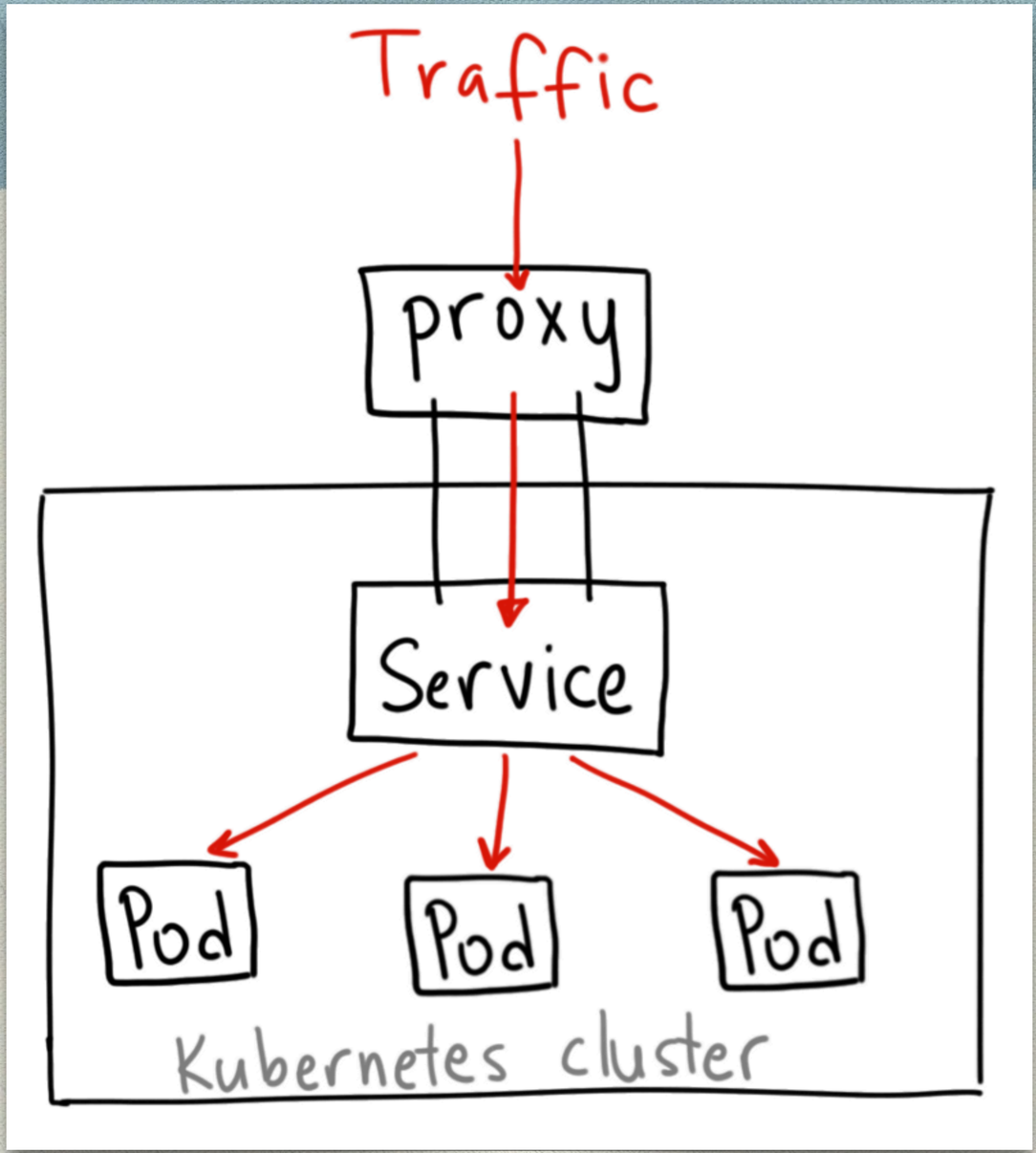
项目架构1.2.1 Update for K8s



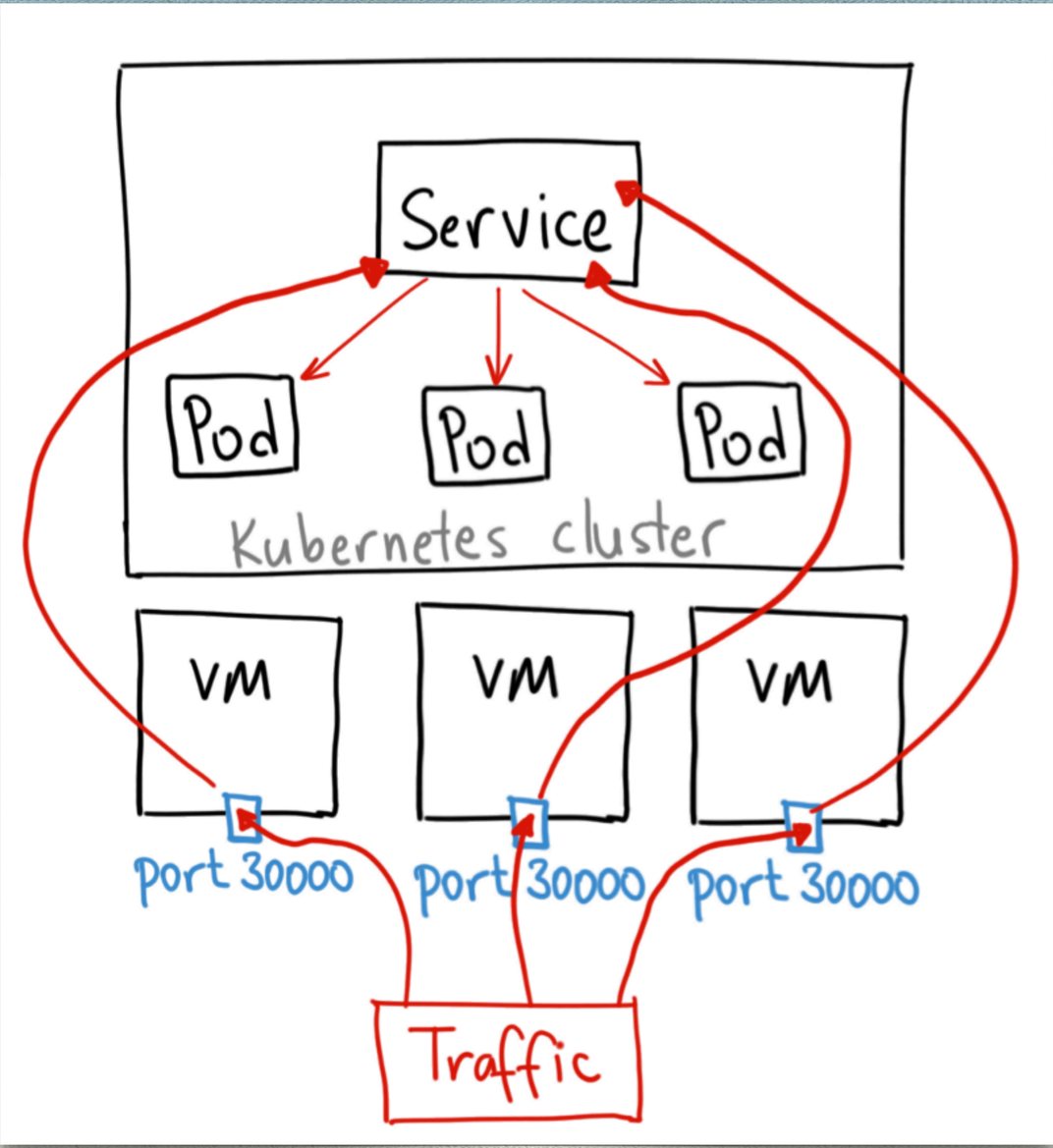
外网访问K8s内部服务方式

- Proxy ~ 调试
- Port Forwarding ~ 调试
- NodePort ~ 测试
- Load Balancer ~ 生产
- Ingress ~ 生产

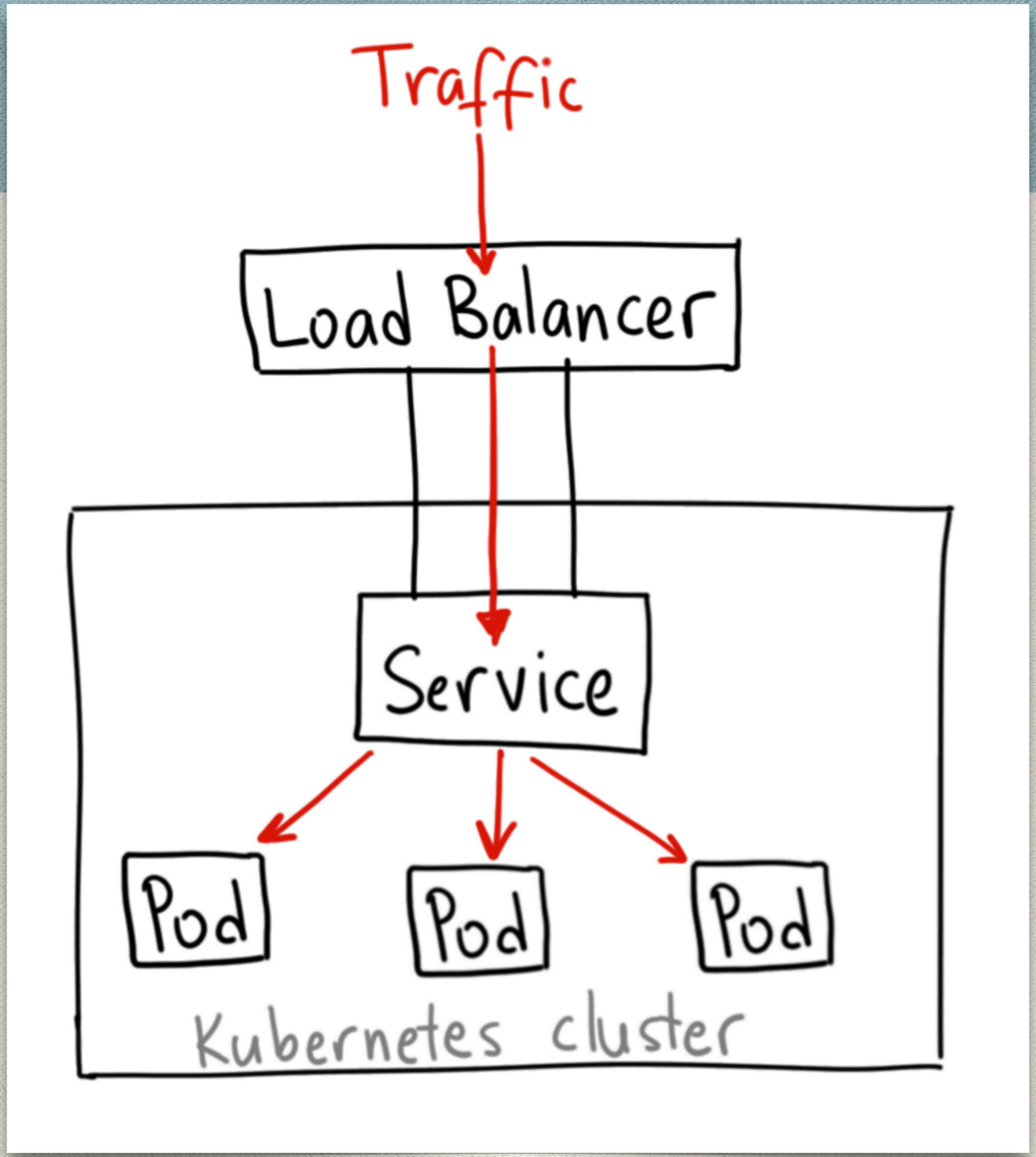
Proxy方式



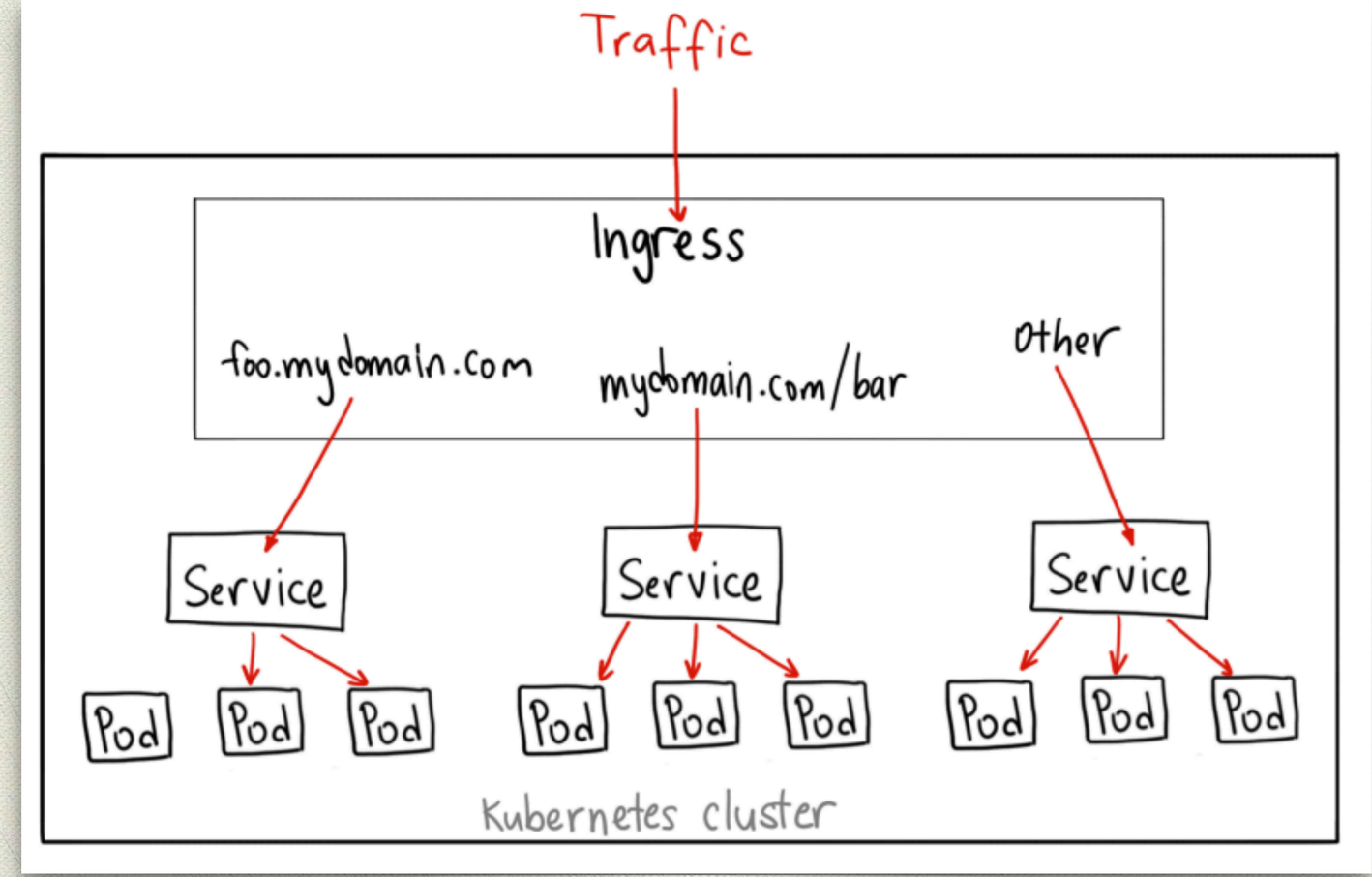
NodePort方式


Service
Pod
Pod
Pod
Kubernetes cluster
VM
VM
VM
port 30000
port 30000
port 30000
Traffic

Load Balancer方式



Ingress方式


Traffic
Ingress
foo.mydomain.com
mydomain.com/bar
other
Service
Service
Service
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Kubernetes cluster

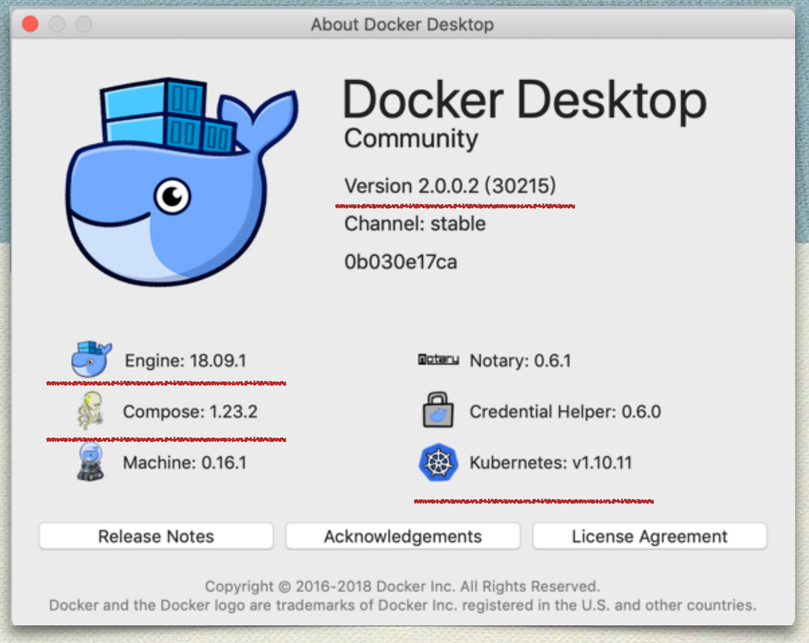
参考

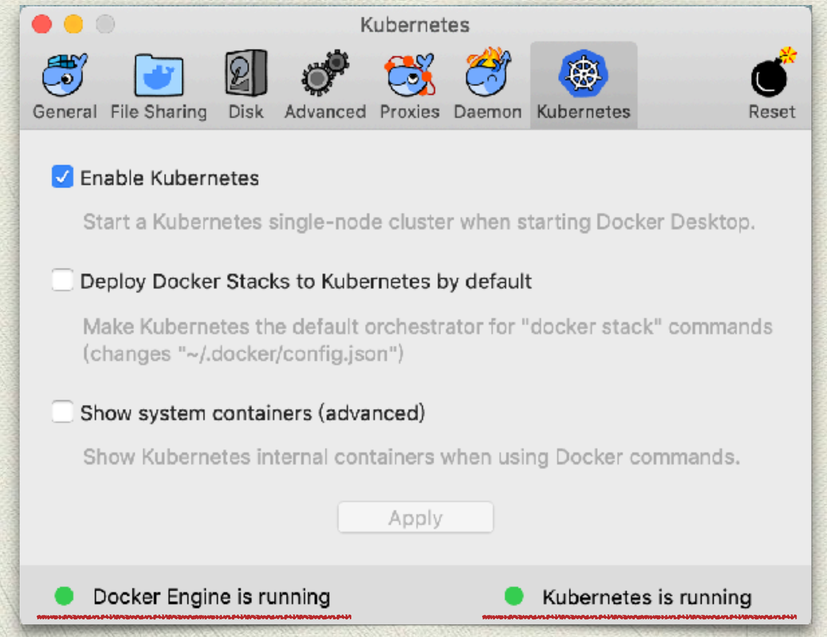
- Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use What?
  - https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0

十六、K8s部署演示

先决步骤

- 安装Docker
  - https://docs.docker.com/
- 启用Docker内置K8s
  - 单节点集群，不可配置，测试用





校验K8s安装

- kubectl version
- kubectl config current-context
- kubectl cluster-info
- kubectl get nodes

安装和访问K8s Dashboard ~ 代理方式

- 1. 安装K8s Dashboard
  - https://github.com/kubernetes/dashboard
- 2. 启动kube proxy
  - kubectl proxy
- 3. 生成访问令牌
  - kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep admin-user | awk '{print $1}')
  - ~/.kube/config
- 4. 访问Dashboard
  - http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/

访问K8s Dashboard ~ 端口转发方式

- 查看K8s名字空间
  - kubectl get namespaces
- 获取kube-system名字空间中的pods
  - kubectl get pods —namespace=kube-system
- 转发端口
  - kubectl port-forward kubernetes-dashboard-669f9bbd46-x7x4r 8443:8443 —namespace=kube-system
- 访问Dashboard
  - https://localhost:8443

运行MovieApp in K8s

- uber jar打包和镜像构建(之前docker compose做过可忽略)
  - mvn clean instal -DskipTests
  - docker compose build
- 发布
  - 进入MovieApp/k8s目录
  - kubectl create -R -f testing/

校验

- 命令行校验
  - kubectl get pods
  - kubectl get services
  - kubectl get deployments
- K8s Dashboard校验 ~ **8001(proxy) or 8443(port-forwarding)**
- MovieApp UI校验 ~ **31100**
- Zipkin校验 ~ **31200**

扩容和校验

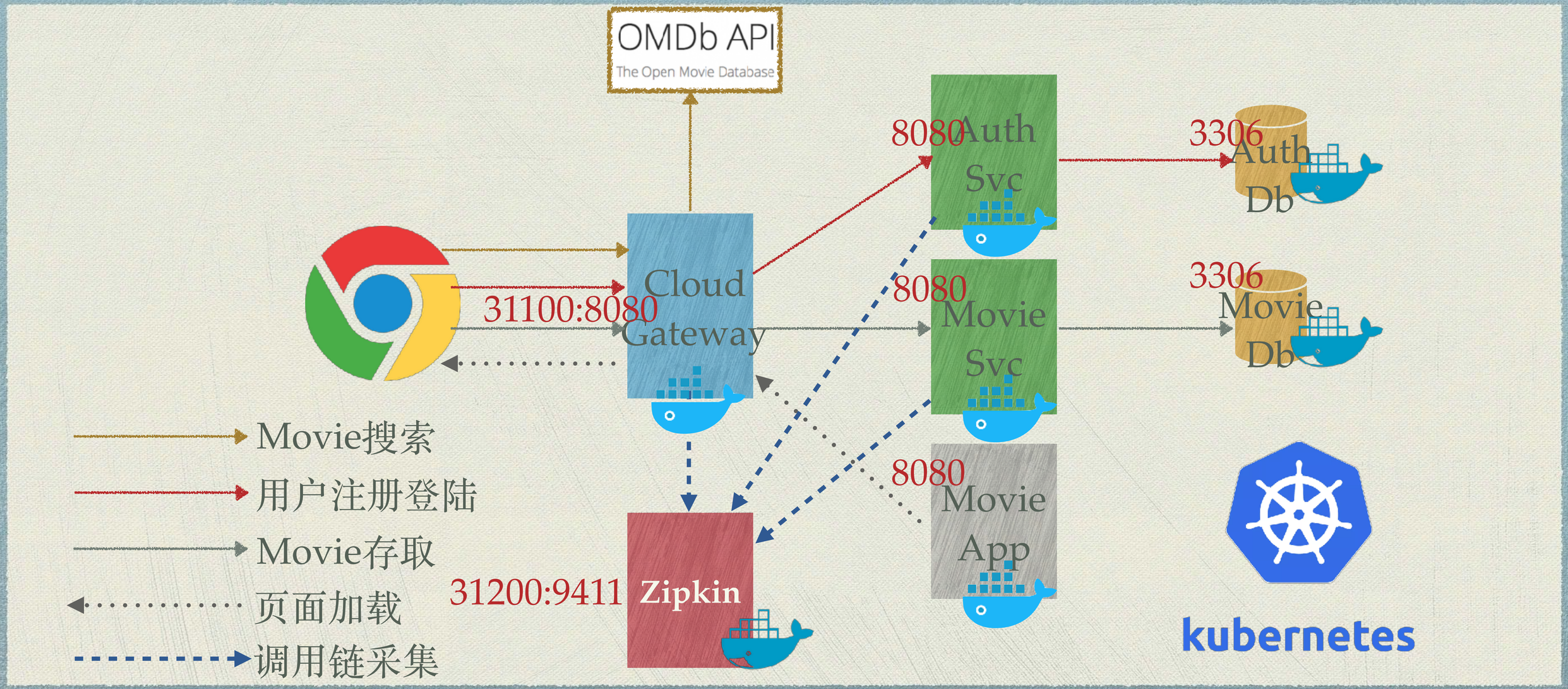
- kubectl get deployments
- kubectl scale —replicas=3 deployment/movie-svc-deployment
- kubectl get deployments

删除清理

- kubectl delete services —all
- kubectl delete deployments —all
- kubectl get pods
- kubectl get services
- kubectl get deployments

十七、项目小结

项目架构1.2.1 Update for K8s



课程目标复盘

架构

- 全栈架构
- 微服务应用架构
- 前后分离架构
- 云原生架构
- 基于Spring Security + JWT的微服务安全架构

技术

- Spring Boot微服务应用开发和测试
- 基于React的前端开发
- Spring Cloud Gateway在微服务架构中的应用
- Docker Compose镜像构建和本地应用部署
- Kubenetes核心概念和应用部署

项目和课程ppt下载

- https://github.com/jskillcloud/MovieApp

spring boot

微服务和云原生架构实践

MovieApp

kubernetes