Implementacion de Plataforma Distribuida

Sistema de Gestion de Eventos en Kubernetes

Modelado de Plataformas de Aplicaciones Distribuidas
Universidad de Valladolid - Curso 2025-2026

28 de enero de 2026

Resumen

Este documento describe la implementacion de un prototipo funcional de plataforma distribuida para la gestion de eventos, desplegada sobre Kubernetes. Se detalla la arquitectura del sistema, los servicios utilizados, las configuraciones necesarias y las decisiones tecnicas tomadas durante el desarrollo. El objetivo es demostrar como los distintos componentes de una plataforma distribuida pueden integrarse y funcionar conjuntamente, resolviendo los requisitos de infraestructura que precisa un sistema de este tipo.

Índice

1. Vision General del Sistema

1.1. Descripcion del Problema

El sistema implementado es una **Plataforma de Gestion de Eventos** que permite a los usuarios crear, gestionar y participar en eventos. La arquitectura esta disenada siguiendo el patron de microservicios, donde cada dominio funcional esta encapsulado en un servicio independiente con su propia base de datos.

La distribucion es relevante en este sistema por varias razones:

- **Escalabilidad horizontal**: Cada microservicio puede escalar independientemente segun la demanda.
- **Aislamiento de fallos**: Un fallo en un servicio no afecta directamente a los demas.
- **Despliegue independiente**: Los equipos pueden desarrollar y desplegar servicios de forma autonoma.
- **Heterogeneidad tecnologica**: Cada servicio podria usar tecnologias diferentes si fuera necesario.

1.2. Arquitectura de Alto Nivel

El sistema sigue una arquitectura de capas claramente definidas:

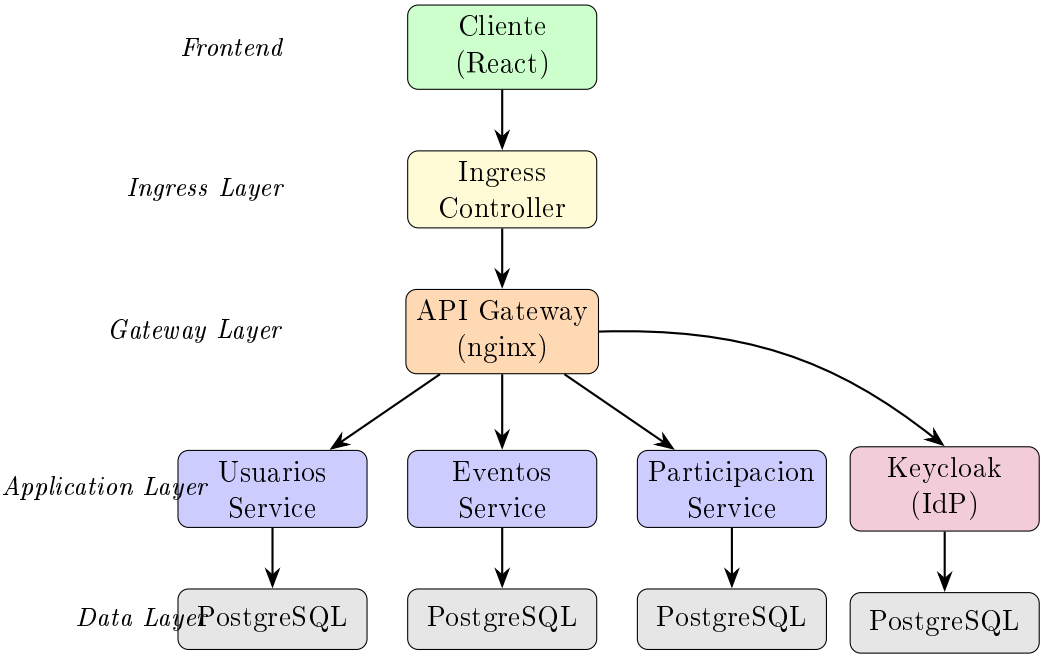

Figura 1: Arquitectura de capas del sistema

1.3. Flujo de una Peticion

El flujo tipico de una peticion HTTP en el sistema es:

1. El **Cliente React** realiza una peticion a `api.eventos.local`

2. El **Ingress Controller** recibe la peticion y la enruta segun el host
3. El **API Gateway** recibe la peticion, aplica CORS y la enruta al microservicio correspondiente
4. El **Microservicio** procesa la peticion, consulta su base de datos si es necesario
5. La respuesta recorre el camino inverso hasta el cliente

1.4. Componentes del Sistema

Componente	Tipo	Funcion
Client	Frontend	Interfaz de usuario React para visualizar estado de servicios
Ingress	Routing	Punto de entrada publico, enrutamiento por dominio
API Gateway	BFF	Proxy inverso, CORS, routing interno a microservicios
Usuarios Service	Microservicio	Gestion de perfiles de usuario
Eventos Service	Microservicio	CRUD de eventos, moderacion
Participacion Service	Microservicio	Inscripciones a eventos
Keycloak	Identity Provider	Autenticacion OAuth2/OIDC
PostgreSQL (x4)	Base de datos	Persistencia de datos por servicio

Cuadro 1: Componentes del sistema y sus funciones

1.5. Namespace y Aislamiento

Todos los recursos se despliegan en el namespace `eventos-system`, lo que proporciona:

- Aislamiento logico de otros workloads del cluster
- Posibilidad de aplicar ResourceQuotas y LimitRanges
- Control de acceso mediante RBAC a nivel de namespace
- Facilidad para limpiar el entorno eliminando el namespace

2. Servicios, Versiones y Alternativas

2.1. Stack Tecnologico Principal

2.2. Justificacion de Decisiones Tecnologicas

2.2.1. Kubernetes como Plataforma

Kubernetes fue elegido como la plataforma de referencia porque:

- Es el estandar de facto para orquestacion de contenedores

Tecnologia	Version	Proposito	Alternativas
Kubernetes	1.28+	Orquestacion de contenedores	Docker Swarm, Nomad
Docker Desktop	4.x	Cluster local de desarrollo	Minikube, Kind, K3s
Tilt	0.33+	Desarrollo local con hot-reload	Skaffold, DevSpace
nginx	1.25-alpine	API Gateway y servicios mock	Traefik, Kong, Envoy
PostgreSQL	15-alpine	Base de datos relacional	MySQL, MariaDB
Keycloak	23.0	Identity Provider OAuth2/OIDC	Auth0, Okta, Dex
React	18.2	Frontend SPA	Vue.js, Angular, Svelte
Vite	5.0	Build tool para frontend	Webpack, Parcel

Cuadro 2: Stack tecnologico y alternativas

- Proporciona primitivas declarativas (Deployments, Services, ConfigMaps)
- Ofrece auto-healing, rolling updates y service discovery integrados
- Permite escalar horizontalmente de forma sencilla

2.2.2. Tilt para Desarrollo Local

Tilt simplifica significativamente el ciclo de desarrollo:

- Detecta cambios y reconstruye/redespliega automaticamente
- Proporciona un dashboard web para monitorizacion
- Gestiona dependencias entre recursos
- Configura port-forwards automaticos

2.2.3. nginx como API Gateway

Se eligio nginx por su simplicidad para un prototipo:

- Configuracion declarativa y bien documentada
- Excelente rendimiento como proxy inverso
- Imagen Docker ligera (alpine)
- Facilidad para anadir CORS y headers personalizados

En produccion, se podria considerar **Kong** o **Envoy** para funcionalidades avanzadas como rate limiting, circuit breaker o validacion JWT integrada.

2.2.4. Keycloak como Identity Provider

Keycloak es una solucion completa de IAM:

- Soporte nativo para OAuth2 y OpenID Connect
- Consola de administracion web

- Federacion con proveedores externos (Google, GitHub, LDAP)
- Gestion de roles y permisos granular

2.3. Dependencias entre Componentes

Tilt gestiona las dependencias mediante `resource_deps`, asegurando que los componentes se inicien en el orden correcto: Bases de Datos → Keycloak → Microservicios → API Gateway → Cliente.

3. Scripts y Configuraciones del Sistema

A continuacion se presentan los scripts mas importantes del sistema, organizados por capa funcional.

3.1. Namespace

El namespace proporciona aislamiento logico para todos los recursos del sistema.

```
apiVersion: v1
kind: Namespace
metadata:
  name: eventos-system
  labels:
    app.kubernetes.io/name: eventos-system
    app.kubernetes.io/part-of: eventos-platform
    environment: development
```

Listing 1: namespace.yaml - Definicion del namespace

Comentario: Se utiliza el estandar de etiquetas de Kubernetes (`app.kubernetes.io/*`) para facilitar la identificacion y filtrado de recursos. La etiqueta `environment` permite distinguir entre entornos de desarrollo y produccion.

3.2. Secrets

Los secrets almacenan credenciales de forma segura.

```
apiVersion: v1
kind: Secret
metadata:
  name: db-usuarios-secret
  namespace: eventos-system
  labels:
    app.kubernetes.io/component: database
type: Opaque
stringData:
  POSTGRES_USER: "usuarios_user"
  POSTGRES_PASSWORD: "usuarios_password_dev"
  POSTGRES_DB: "usuarios_db"
```

Listing 2: secrets.yaml - Credenciales de base de datos (extracto)

Comentario: En produccion, se deberian usar soluciones como HashiCorp Vault o los gestores de secretos nativos de los proveedores cloud (AWS Secrets Manager, Azure Key Vault). Los valores aqui mostrados son unicamente para desarrollo.

3.3. Base de Datos PostgreSQL

Cada microservicio tiene su propia instancia de PostgreSQL, siguiendo el patron "Database per Service".

```
# PersistentVolumeClaim para persistencia
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-usuarios-pvc
  namespace: eventos-system
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-usuarios
  namespace: eventos-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db-usuarios
  strategy:
    type: Recreate  # Necesario para PVC ReadWriteOnce
  template:
    spec:
      containers:
        - name: postgres
          image: postgres:15-alpine
          ports:
            - containerPort: 5432
          envFrom:
            - secretRef:
                name: db-usuarios-secret
          resources:
            requests:
              memory: "128Mi"
              cpu: "100m"
            limits:
              memory: "256Mi"
              cpu: "500m"
          volumeMounts:
            - name: postgres-storage
              mountPath: /var/lib/postgresql/data
              subPath: postgres
          livenessProbe:
            exec:
              command: ["pg_isready", "-U", "usuarios_user"]
            initialDelaySeconds: 30
            periodSeconds: 10
      volumes:
        - name: postgres-storage
          persistentVolumeClaim:
```

```
            claimName: db-usuarios-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: db-usuarios
  namespace: eventos-system
spec:
  type: ClusterIP
  ports:
    - port: 5432
  selector:
    app: db-usuarios
```

Listing 3: postgres-usuarios.yaml - Base de datos del servicio de usuarios

Comentarios:

- `strategy: Recreate`: Necesario porque el PVC tiene modo `ReadWriteOnce`.
- `subPath: postgres`: Evita problemas con el directorio `lost+found`.
- `livenessProbe`: Utiliza `pg_isready` para verificar el estado.
- `ClusterIP`: El servicio solo es accesible internamente.

3.4. Keycloak - Identity Provider

Keycloak proporciona autenticacion OAuth2/OIDC para el sistema.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: keycloak-config
  namespace: eventos-system
data:
  KC_HOSTNAME_STRICT: "false"
  KC_HTTP_ENABLED: "true"
  KC_PROXY: "edge"
  KC_DB: "postgres"
  KC_DB_URL_HOST: "db-keycloak"
  KC_HEALTH_ENABLED: "true"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  namespace: eventos-system
spec:
  replicas: 1
  template:
    spec:
      initContainers:
        - name: wait-for-db
          image: busybox:1.36
          command: ['sh', '-c',
            'until nc -z db-keycloak 5432; do sleep 2; done']
      containers:
```

```
        - name: keycloak
          image: quay.io/keycloak/keycloak:23.0
          args: ["start-dev"]
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: keycloak-config
            - secretRef:
                name: keycloak-secret
          resources:
            requests:
              memory: "512Mi"
            limits:
              memory: "1Gi"
          readinessProbe:
            httpGet:
              path: /health/ready
              port: 8080
            initialDelaySeconds: 30
```

Listing 4: keycloak.yaml - Configuracion de Keycloak (extracto)

Comentarios:

- `initContainers`: Espera a que PostgreSQL este disponible.
- `start-dev`: Modo desarrollo sin HTTPS. En produccion usar `start`.
- `KC_PROXY: edge`: Indica que hay un proxy delante que termina TLS.

3.5. API Gateway

El API Gateway centraliza el acceso a los microservicios y gestiona CORS.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-gateway-nginx-config
  namespace: eventos-system
data:
  nginx.conf: |
    events { worker_connections 1024; }
    http {
        upstream usuarios_service {
            server usuarios-service:8080;
        }
        upstream eventos_service {
            server eventos-service:8080;
        }
        upstream participacion_service {
            server participacion-service:8080;
        }

        server {
            listen 8080;

            location /health {
```

```
                add_header Content-Type application/json;
                return 200 '{"status": "healthy"}';
            }

            location /api/users {
                if ($request_method = 'OPTIONS') {
                    add_header 'Access-Control-Allow-Origin' '*';
                    add_header 'Access-Control-Allow-Methods'
                               'GET, POST, PUT, DELETE, OPTIONS';
                    add_header 'Access-Control-Allow-Headers'
                               'Authorization, Content-Type';
                    return 204;
                }
                add_header 'Access-Control-Allow-Origin' '*' always;
                proxy_pass http://usuarios_service/api/users;
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
            }

            location /api/eventos {
                add_header 'Access-Control-Allow-Origin' '*' always;
                proxy_pass http://eventos_service/api/eventos;
            }

            location /api/participacion {
                add_header 'Access-Control-Allow-Origin' '*' always;
                proxy_pass http://participacion_service;
            }
        }
    }
```

Listing 5: api-gateway.yaml - Configuracion nginx del API Gateway

Comentarios:

- `upstream`: Define los backends para balanceo de carga.
- `CORS preflight`: Las peticiones OPTIONS se responden sin pasar al backend.
- En produccion, se anadira validacion JWT mediante `auth_request`.

3.6. Ingress

El Ingress expone los servicios al exterior mediante routing basado en host.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: main-ingress
  namespace: eventos-system
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  ingressClassName: nginx
  rules:
    - host: api.eventos.local
      http:
        paths:
```

```
          - path: /
            pathType: Prefix
            backend:
              service:
                name: api-gateway
                port:
                  number: 8080
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: client-ingress
  namespace: eventos-system
spec:
  ingressClassName: nginx
  rules:
    - host: eventos.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: client
                port:
                  number: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: keycloak-admin
  namespace: eventos-system
spec:
  ingressClassName: nginx
  rules:
    - host: keycloak.eventos.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: keycloak
                port:
                  number: 8080
```

Listing 6: ingress.yaml - Configuracion del Ingress

Comentarios:

- Tres Ingress separados para diferentes subdominios.
- Requiere configurar `/etc/hosts`: `127.0.0.1 eventos.local api.eventos.local keycloak.eventos.local`

3.7. Tiltfile

El Tiltfile orquesta el desarrollo local con Tilt.

```
# Configuracion para Docker Desktop
allow_k8s_contexts('docker-desktop')

# Desplegar manifiestos
k8s_yaml('namespace.yaml')
k8s_yaml('configmaps/secrets.yaml')
k8s_yaml('databases/postgres-keycloak.yaml')
k8s_yaml('databases/postgres-usuarios.yaml')
k8s_yaml('services/keycloak.yaml')
k8s_yaml('services/usuarios-service.yaml')
k8s_yaml('services/api-gateway.yaml')
k8s_yaml('services/client.yaml')
k8s_yaml('ingress/ingress.yaml')

# Build imagen del cliente
docker_build('eventos-client', '../client',
    dockerfile='../client/Dockerfile')

# Configurar recursos con dependencias
k8s_resource('keycloak',
    labels=['identity'],
    resource_deps=['db-keycloak'],
    port_forwards=['8080:8080'],
    links=['http://keycloak.eventos.local'])

k8s_resource('api-gateway',
    labels=['gateway'],
    resource_deps=['usuarios-service', 'eventos-service'],
    port_forwards=['9000:8080'],
    links=['http://api.eventos.local'])

k8s_resource('client',
    labels=['frontend'],
    resource_deps=['api-gateway'],
    port_forwards=['3000:80'],
    links=['http://eventos.local'])
```

Listing 7: Tiltfile - Orquestacion de desarrollo local

Comentarios:

- `allow_k8s_contexts`: Restringe a Docker Desktop por seguridad.
- `docker_build`: Construye la imagen del cliente automaticamente.
- `resource_deps`: Define el orden de inicio.
- `port_forwards`: Acceso directo sin Ingress.
- `links`: URLs clickeables en el dashboard de Tilt.

3.8. Cliente React

Aplicacion React minimalista para visualizar el estado de los servicios.

```
const API_URL = window.__API_URL__ || 'http://api.eventos.local'

const services = [
```

```
  { id: 'gateway', name: 'API Gateway', endpoint: '/' },
  { id: 'usuarios', name: 'Usuarios', endpoint: '/api/users' },
  { id: 'eventos', name: 'Eventos', endpoint: '/api/eventos' },
  { id: 'participacion', name: 'Participacion',
    endpoint: '/api/participacion' },
]

function App() {
  const [statuses, setStatuses] = useState({})

  const checkHealth = async (service) => {
    setStatuses(prev => ({ ...prev, [service.id]: 'loading' }))
    try {
      const res = await fetch(`${API_URL}${service.endpoint}`)
      await res.json()
      setStatuses(prev => ({ ...prev, [service.id]: 'healthy' }))
    } catch (error) {
      setStatuses(prev => ({ ...prev, [service.id]: 'error' }))
    }
  }

  useEffect(() => {
    services.forEach(service => checkHealth(service))
  }, [])

  return (
    <div className="container">
      <h1>Eventos Platform</h1>
      {services.map(service => (
        <ServiceCard key={service.id} service={service}
          status={statuses[service.id]} onTest={checkHealth} />
      ))}
    </div>
  )
}
```

Listing 8: App.jsx - Componente principal del cliente

Comentario: El cliente realiza peticiones a cada servicio a traves del API Gateway y muestra el estado de cada uno. La URL se inyecta en runtime mediante el script `docker-entrypoint.sh`.

4. Conclusiones y Posibles Mejoras

4.1. Estado Actual del Sistema

El prototipo implementado demuestra con exito:

- **Arquitectura de microservicios**: Separacion clara de responsabilidades con servicios independientes.
- **Orquestacion con Kubernetes**: Uso de primitivas estandar (Deployments, Services, ConfigMaps, Secrets, Ingress).
- **Patron API Gateway**: Punto de entrada centralizado que gestiona routing y CORS.

- **Identity Provider**: Keycloak configurado y listo para gestionar autenticacion.
- **Persistencia de datos**: Bases de datos PostgreSQL independientes por servicio.
- **Desarrollo local eficiente**: Tilt proporciona hot-reload y dashboard de monitorizacion.

4.2. Que se Puede Esperar del Sistema

4.2.1. Fortalezas

- **Reproducibilidad**: El sistema puede desplegarse de forma identica en cualquier cluster Kubernetes.
- **Escalabilidad**: Cada componente puede escalar horizontalmente modificando `replicas`.
- **Observabilidad basica**: Health checks en todos los servicios permiten deteccion de fallos.

4.2.2. Limitaciones

- **Servicios mock**: Los microservicios no tienen logica de negocio real.
- **Sin autenticacion activa**: El API Gateway no valida JWT (preparado pero no implementado).
- **Sin TLS**: Todo el trafico es HTTP en desarrollo.
- **Bases de datos no replicadas**: Instancias unicas sin alta disponibilidad.

4.3. Posibles Mejoras

4.3.1. Corto Plazo (Prototipo Avanzado)

1. **Implementar validacion JWT en el API Gateway**: Usar `auth_request` de nginx para validar tokens con Keycloak.
2. **Anadir logica a los microservicios**: Reemplazar nginx por aplicaciones reales (Node.js, Spring Boot, Go).
3. **Configurar Keycloak**: Crear el realm "eventos", clientes OAuth2 y roles.

4.3.2. Medio Plazo (Produccion)

1. **TLS/HTTPS**: Usar cert-manager con Let's Encrypt para certificados automaticos.
2. **Observabilidad completa**: Prometheus + Grafana para metricas, ELK para logs, Jaeger para tracing.
3. **CI/CD**: Pipeline con GitHub Actions para despliegue automatizado.
4. **GitOps**: Usar ArgoCD o Flux para gestion declarativa del cluster.

4.3.3. Largo Plazo (Produccion Escalable)

1. **Service Mesh**: Implementar Istio o Linkerd para mTLS y traffic management.
2. **Bases de datos gestionadas**: Migrar a servicios cloud (RDS, Cloud SQL).
3. **Event-Driven Architecture**: Anadir message broker (Kafka, RabbitMQ).
4. **Autoscaling**: Configurar HPA basado en metricas.

4.4. Conclusion Final

Este prototipo cumple su objetivo de demostrar como una plataforma de microservicios puede estructurarse y desplegarse sobre Kubernetes. Aunque los servicios son mocks sin logica real, la infraestructura esta completa y lista para evolucionar hacia un sistema de produccion.

La arquitectura implementada sigue principios solidos de diseno distribuido: separacion de responsabilidades, comunicacion a traves de APIs bien definidas, gestion centralizada de identidad, y orquestacion declarativa. Estas bases permiten que el sistema escale tanto en funcionalidad como en capacidad segun las necesidades futuras.

Documento generado para la asignatura de Modelado de Plataformas de Aplicaciones Distribuidas
Universidad de Valladolid - Curso 2025-2026