

Sistema Distribuído: Plataforma E-commerce "GlobalShop"

Arquitetura de Microserviços e REST

Membros do Grupo

- Arthur Renato Normando Vasconcelos
- Bruno Vaz Ferreira

1. Visão Geral do Sistema

Contexto Comercial

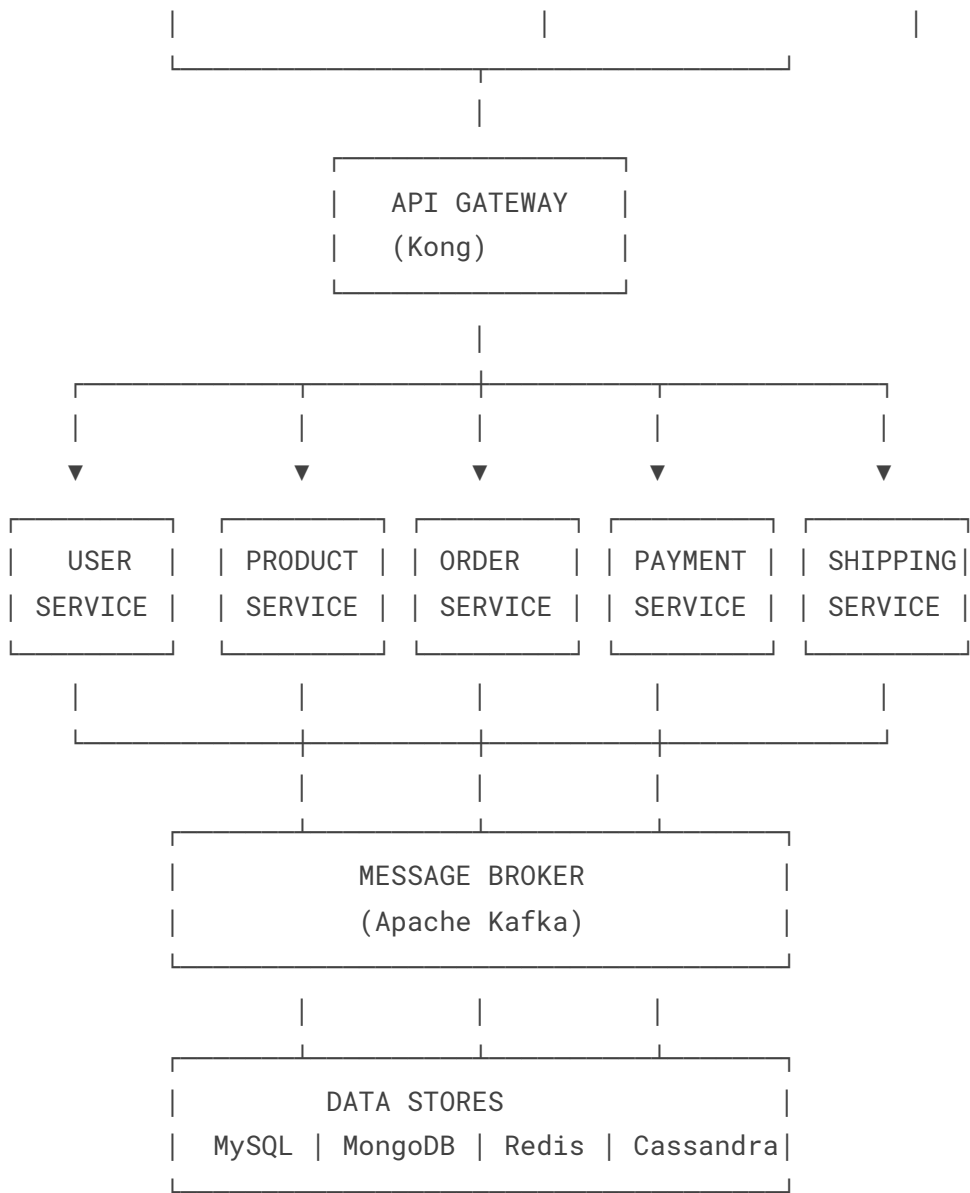
A **GlobalShop** é uma plataforma de e-commerce que opera em 15 países, processando mais de 50.000 pedidos diários. O sistema precisa ser:

- Altamente escalável** para suportar picos de tráfego
- Disponível** 24/7 com tolerância a falhas
- Manutenível** com deploy independente de componentes
- Rápido** com baixa latência global

2. Arquitetura de Microserviços

2.1 Diagrama da Arquitetura





3. Microservios Principais

3.1 User Service

Responsabilidade: Gerenciamento de usurios e autenticao

Endpoints REST:

- POST /api/users/register
- POST /api/users/login
- GET /api/users/profile
- PUT /api/users/profile
- POST /api/users/logout

Tecnologias:

- **Java 17** + Spring Boot
- **MySQL** - Dados persistentes
- **Redis** - Sessões e cache
- **JWT** - Tokens de autenticação

3.2 Product Service

Responsabilidade: Catálogo de produtos e estoque

Endpoints REST:

```
GET      /api/products
GET      /api/products/{id}
POST     /api/products
PUT      /api/products/{id}
DELETE   /api/products/{id}
GET      /api/products/search?q={query}
PUT      /api/products/{id}/stock
```

Tecnologias:

- **Node.js** + Express
- **MongoDB** - Dados semi-estruturados
- **Elasticsearch** - Busca
- **Redis** - Cache

3.3 Order Service

Responsabilidade: Processamento de pedidos

Endpoints REST:

```
POST     /api/orders
GET      /api/orders/{id}
GET      /api/orders/user/{userId}
PUT      /api/orders/{id}/status
```

Tecnologias:

- **Python** + FastAPI
- **PostgreSQL** - Transações ACID
- **Redis** - Filas

3.4 Payment Service

Responsabilidade: Processamento de pagamentos

Endpoints REST:

```
POST    /api/payments/process
GET     /api/payments/{id}
POST    /api/payments/refund
```

Tecnologias:

- **Java** + Spring Boot
- **Oracle DB** - Conformidade financeira
- **Apache Kafka** - Eventos de pagamento

3.5 Shipping Service

Responsabilidade: Logística e entrega

Endpoints REST:

```
POST    /api/shipping/calculate
POST    /api/shipping/{orderId}
GET     /api/shipping/track/{trackingNumber}
PUT     /api/shipping/{id}/status
```

Tecnologias:

- **Go** + Gin
- **Cassandra** - Dados de tracking
- **Redis** - Cache de cotações

4. Comunicação entre Serviços

4.1 Síncrona - REST API

```
// Exemplo: Order Service chamando Payment Service
@RestController
public class OrderController {

    @PostMapping("/api/orders")
    public ResponseEntity<Order> createOrder(@RequestBody OrderRequest request) {
        // 1. Validar produtos
```

```

Product[] products = restTemplate.getForObject(
    "http://product-service/api/products/validate",
    Product[].class,
    request.getProductIds()
);

// 2. Processar pagamento
PaymentResponse payment = restTemplate.postForObject(
    "http://payment-service/api/payments/process",
    request.getPayment(),
    PaymentResponse.class
);

// 3. Criar pedido
Order order = orderService.createOrder(request, products, payment);

return ResponseEntity.ok(order);
}
}

```

4.2 Assíncrona - Apache Kafka

```

// Product Service publicando evento de estoque
@Service
public class ProductService {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    public void updateStock(String productId, int quantity) {
        // Atualizar estoque no MongoDB
        productRepository.updateStock(productId, quantity);

        // Publicar evento
        StockEvent event = new StockEvent(productId, quantity, "STOCK_UPDATED");
        kafkaTemplate.send("stock-events", event);
    }
}

// Order Service consumindo evento
@Service
public class OrderService {

    @KafkaListener(topics = "stock-events")
    public void handleStockEvent(StockEvent event) {
        if ("STOCK_UPDATED".equals(event.getType())) {
            // Atualizar cache local
            cacheService.updateProductStock(

```

```
        event.getProductId(),
        event.getQuantity()
    );
}
}
```

5. Infraestrutura e Deployment

5.1 Containerização - Docker

```
# Dockerfile do User Service
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/user-service-1.0.0.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

5.2 Orquestração - Kubernetes

```
# deployment.yaml do User Service
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
        - name: user-service
          image: globalshop/user-service:1.0.0
          ports:
            - containerPort: 8080
          env:
            - name: DB_URL
              value: "jdbc:mysql://mysql-service:3306/users"
```

```
- name: REDIS_HOST
  value: "redis-service"

---

apiVersion: v1
kind: Service
metadata:
  name: user-service
spec:
  selector:
    app: user-service
  ports:
    - port: 80
      targetPort: 8080
```

5.3 Service Discovery

```
# Configuração Eureka
eureka:
  client:
    serviceUrl:
      defaultZone: http://eureka-server:8761/eureka/
  instance:
    preferIpAddress: true
```

6. Padrões REST Implementados

6.1 Design de Recursos

```
# Coleções
GET      /api/products      # Listar produtos
POST     /api/products  # Criar produto

# Itens individuais
GET      /api/products/123  # Obter produto específico
PUT      /api/products/123  # Atualizar produto
DELETE   /api/products/123  # Excluir produto

# Sub-recursos
GET      /api/products/123/reviews  # Reviews do produto
POST     /api/products/123/reviews  # Adicionar review
```

6.2 Códigos de Status HTTP

```

@RestController
public class ProductController {

    @GetMapping("/api/products/{id}")
    public ResponseEntity<Product> getProduct(@PathVariable String id) {
        return productService.findById(id)
            .map(product -> ResponseEntity.ok(product))
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping("/api/products")
    public ResponseEntity<Product> createProduct(@Valid @RequestBody Product product) {
        Product saved = productService.save(product);
        return ResponseEntity.status(HttpStatus.CREATED).body(saved);
    }
}

```

6.3 Versionamento de API

Versionamento por URL

GET /api/v1/products

GET /api/v2/products

Versionamento por Header

GET /api/products

Accept: application/vnd.globalshop.v1+json

7. Benefícios da Arquitetura

7.1 Escalabilidade

- **Escala independente:** Cada serviço escala conforme demanda
- **User Service:** 10 réplicas durante Black Friday
- **Product Service:** 5 réplicas com cache Redis
- **Payment Service:** 3 réplicas (menos variável)

7.2 Resiliência

// Circuit Breaker no Order Service

@RestController

```
public class OrderController {
```



```
@CircuitBreaker(name = "paymentService", fallbackMethod = "fallbackPayment")
@PostMapping("/api/orders")
public Order createOrder(@RequestBody OrderRequest request) {
    // Tenta processar pagamento
    return orderService.processOrder(request);
}

public Order fallbackPayment(OrderRequest request, Exception e) {
    // Salva pedido como "pending_payment"
    return orderService.savePendingOrder(request);
}
}
```

7.3 Desenvolvimento Ágil

- **Times independentes:** 5 squads (1 por microserviço principal)
 - **Deploy contínuo:** Cada serviço tem seu pipeline
 - **Tecnologias heterogêneas:** Java, Node.js, Python, Go
-

8. Métricas e Monitoramento

8.1 Dashboard Principal

- **Throughput:** 1.200 req/segundo (pico)
- **Latência média:** 180ms
- **Disponibilidade:** 99.95%
- **Error rate:** 0.02%

8.2 Ferramentas

- **Prometheus** - Coleta de métricas
 - **Grafana** - Dashboards
 - **Jaeger** - Distributed tracing
 - **ELK Stack** - Logs
-

9. Caso de Uso: Fluxo de Compra

9.1 Sequência de Chamadas

1. Cliente → API Gateway → User Service (autenticação)
2. Cliente → API Gateway → Product Service (carrinho)
3. Cliente → API Gateway → Order Service (criar pedido)
4. Order Service → Payment Service (processar pagamento)
5. Payment Service → Kafka → Shipping Service (iniciar entrega)
6. Shipping Service → Kafka → Order Service (atualizar status)
7. Order Service → Kafka → User Service (notificação)

9.2 Exemplo de Transação Completa

// Order Service orquestrando a compra

@Service

```
public class OrderService {
```

```
    public Order processOrder(OrderRequest request) {
```

```
        // 1. Validar usuário
```

```
        User user = userClient.getUser(request.getUserId());
```

```
        // 2. Validar produtos e estoque
```

```
        Product[] products = productClient.validateProducts(  
            request.getProductIds()  
        );
```

```
        // 3. Calcular frete
```

```
        ShippingQuote shipping = shippingClient.calculateShipping(  
            request.getAddress(), products  
        );
```

```
        // 4. Processar pagamento
```

```
        PaymentResponse payment = paymentClient.processPayment(  
            request.getPayment(), calculateTotal(products, shipping)  
        );
```

```
        // 5. Criar pedido
```

```
        Order order = createOrder(user, products, shipping, payment);
```

```
        // 6. Publicar eventos
```

```
        kafkaTemplate.send("order-events", new OrderCreatedEvent(order));
```






```
        return order;
```

```
    }
```

```
}
```

10. Conclusão

A arquitetura de microserviços com REST permitiu à **GlobalShop**:

-  **Escalar** de 1.000 para 50.000 pedidos/dia
-  **Reduzir** tempo de deploy de horas para minutos
-  **Manter** disponibilidade de 99.95%
-  **Desenvolver** novas features 60% mais rápido
-  **Isolar** falhas sem afetar todo o sistema

Tecnologias Chave: Java, Spring Boot, Docker, Kubernetes, Kafka, REST, MySQL, MongoDB, Redis

Documento elaborado por Arthur Renato Normando Vasconcelos e Bruno Vaz Ferreira - Arquitetura de Sistemas Distribuídos