Go + Flutter Course

Communication & Microservices

Timur Harin

Lecture 06: **Communication & Microservices**

Building scalable, distributed, real-time applications

Block 6: Real-time & Microservices

Lecture 06 Overview

- **Real-time Communication**: WebSockets, WebRTC, VoIP, and peer-to-peer systems
- **gRPC & Protocol Buffers**: High-performance RPC communication
- **Microservices Architecture**: Design patterns and communication strategies

What we'll learn

- **Why real-time communication matters in modern apps**
- **WebSocket implementation in Go and Flutter**
- **WebRTC for peer-to-peer audio/video communication**
- **VoIP systems with SIP protocol and audio processing**
- **gRPC service development and client integration**
- **Microservices patterns and anti-patterns**
- **Service discovery and inter-service communication**

Learning path

- **Foundation**: Real-time communication protocols and patterns
- **High-performance**: gRPC for efficient service-to-service communication
- **Architecture**: Microservices design and communication strategies
- **Integration**: Cross-platform mobile and server communication

Part I: Real-time Communication

> ***Real-time communication*** *enables instant data exchange between clients and servers, creating responsive and interactive user experiences.*

Why do we need different communication protocols?

- **HTTP Request/Response**: Good for traditional web pages, APIs
- **Real-time needs**: Chat, live updates, gaming require instant communication
- **Efficiency**: Different protocols optimize for different use cases
- **Network constraints**: Mobile networks, firewalls, NAT traversal

Common real-time scenarios

```
Traditional HTTP:
Client → Request → Server → Response → Client (high latency)

WebSocket:
Client ↔ Persistent Connection ↔ Server (instant bidirectional)

Server-Sent Events:
Client ← Stream of Updates ← Server (server-to-client only)
```

When to use WebSocket vs alternatives

WebSocket is perfect for:

- **Chat applications**: Instant messaging, group conversations
- **Live gaming**: Real-time multiplayer interactions
- **Collaborative editing**: Google Docs-style simultaneous editing
- **Trading platforms**: Real-time stock price updates
- **Live sports scores**: Instant score updates during matches

WebSocket vs other solutions:

Scenario	HTTP Polling	Server-Sent Events	WebSocket	Best Choice
Chat app	❌ Inefficient	❌ One-way only	✅ Perfect	**WebSocket**
Live news feed	❌ Delays	✅ Efficient	✅ Overkill	**SSE**
API calls	✅ Simple	❌ Wrong tool	❌ Overkill	**HTTP**
Gaming	❌ Too slow	❌ One-way	✅ Essential	**WebSocket**
Live charts	❌ Inefficient	✅ Good enough	✅ Better	**WebSocket**

HTTP vs WebSocket: Concrete example

HTTP Polling (inefficient)

```
# Request every 2 seconds
GET /api/messages?since=1642781234 HTTP/1.1
Host: example.com
Authorization: Bearer token123
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 45

{"messages": [], "has_new": false}
```

Problems:

- 🔴 Empty responses waste bandwidth
- 🔴 2-second delay for new messages
- 🔴 HTTP headers overhead (200+ bytes each)
- 🔴 Server resources wasted on empty polls

WebSocket (efficient)

```
# Initial handshake only
GET /chat HTTP/1.1
Upgrade: websocket
Connection: Upgrade

# Then instant messages:
{"type":"message","text":"Hello!","user":"John"}
{"type":"message","text":"Hi there!","user":"Jane"}
```

Benefits:

- ✅ Instant delivery (0ms delay)
- ✅ Only 50 bytes per message
- ✅ No HTTP header overhead
- ✅ Bidirectional communication
- ✅ Single persistent connection

Bandwidth comparison: 200+ bytes (HTTP) vs 50 bytes (WebSocket)

Simple WebSocket server example

Basic connection handling

```go
package main

import (
    "github.com/gorilla/websocket"
    "net/http"
)

var upgrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true // Allow all origins
    },
}

func handleWebSocket(w http.ResponseWriter, r *http.Request) {
    // Upgrade HTTP to WebSocket
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        return
    }
    defer conn.Close()

    // ... connection handling code ...
}
```

Message broadcasting

```go
// Simple message structure
type Message struct {
    Type    string `json:"type"`
    Content string `json:"content"`
    User    string `json:"user"`
}

// Broadcast to all connected clients
var clients = make(map[*websocket.Conn]bool)

func broadcastMessage(message Message) {
    for client := range clients {
        err := client.WriteJSON(message)
        if err != nil {
            client.Close()
            delete(clients, client)
        }
    }
}

func handleMessage(conn *websocket.Conn) {
    var msg Message
    err := conn.ReadJSON(&msg)
    if err == nil {
        broadcastMessage(msg) // Send to all clients
```

Flutter WebSocket client example

Simple connection setup

```dart
import 'dart:io';
import 'dart:convert';

class SimpleWebSocket {
  WebSocket? _socket;

  Future<void> connect(String url) async {
    _socket = await WebSocket.connect(url);

    // Listen for messages
    _socket!.listen((data) {
      final message = jsonDecode(data);
      print('Received: ${message['content']}');
    });
  }

  void sendMessage(String content) {
    final message = {
      'type': 'message',
      'content': content,
      'user': 'CurrentUser'
    };

    _socket?.add(jsonEncode(message));
```

Usage in Flutter widget

```dart
class ChatWidget extends StatefulWidget {
  @override
  _ChatWidgetState createState() => _ChatWidgetState();
}

class _ChatWidgetState extends State<ChatWidget> {
  final _webSocket = SimpleWebSocket();
  final _messages = <String>[];

  @override
  void initState() {
    super.initState();
    _webSocket.connect('ws://localhost:8080/chat');
  }

  void _sendMessage(String text) {
    _webSocket.sendMessage(text);
    setState(() {
      _messages.add('You: $text');
    });
  }
```

HTTP vs gRPC: Request size comparison

HTTP JSON Request (verbose)

```
POST /api/v1/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Accept: application/json
User-Agent: MyApp/1.0
Content-Length: 156

{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "age": 30,
  "city": "New York",
  "country": "USA",
  "phone": "+1-555-123-4567"
}
```

Total size: ~650 bytes

- Headers: ~480 bytes
- JSON body: ~170 bytes

gRPC Binary Request (efficient)

```
// user.proto
message CreateUserRequest {
  string name = 1;      // "John Doe"
  string email = 2;     // "john.doe@example.com"
  int32 age = 3;        // 30
  string city = 4;      // "New York"
  string country = 5;   // "USA"
  string phone = 6;     // "+1-555-123-4567"
}
```

Binary encoding:

```
0A 08 4A 6F 68 6E 20 44 6F 65 12 15 6A 6F 68 6E 2E 64 6F 65...
```

Total size: ~85 bytes

- HTTP/2 headers: ~20 bytes
- Protocol Buffer body: ~65 bytes

Protocol Buffers efficiency example

JSON representation

```
{
  "users": [
    {
      "id": 12345,
      "name": "John Doe",
      "email": "john@example.com",
      "active": true,
      "created_at": 1642781234
    },
    {
      "id": 12346,
      "name": "Jane Smith",
      "email": "jane@example.com",
      "active": true,
      "created_at": 1642781235
    }
  ],
  "total": 2,
  "page": 1
}
```

Size: 267 bytes

Protocol Buffers binary

```
message User {
  int32 id = 1;
  string name = 2;
  string email = 3;
  bool active = 4;
  int64 created_at = 5;
}

message ListUsersResponse {
  repeated User users = 1;
  int32 total = 2;
  int32 page = 3;
}
```

Binary encoding:

```
0A 2F 08 B9 60 12 08 4A 6F 68 6E 20 44 6F 65 1A 10 6A 6F 68 6E...
```

Size: 89 bytes

66% smaller!

JSON-RPC: Simple RPC over HTTP

> ***JSON-RPC*** *is a lightweight Remote Procedure Call protocol using JSON for data exchange. Simpler than gRPC but less efficient.*

JSON-RPC Request/Response

```
// Request
{
  "jsonrpc": "2.0",
  "method": "createUser",
  "params": {
    "name": "John Doe",
    "email": "john@example.com"
  },
  "id": 1
}

// Response
{
  "jsonrpc": "2.0",
  "result": {
    "id": 12345,
    "name": "John Doe",
    "email": "john@example.com",
    "created_at": 1642781234
  },
  "id": 1
}
```

Go JSON-RPC server

```
type UserService struct{}

type CreateUserParams struct {
    Name  string `json:"name"`
    Email string `json:"email"`
}

type User struct {
    ID        int    `json:"id"`
    Name      string `json:"name"`
    Email     string `json:"email"`
    CreatedAt int64  `json:"created_at"`
}

func (s *UserService) CreateUser(params CreateUserParams) (*User, error) {
    if params.Name == "" {
        return nil, fmt.Errorf("name is required")
    }

    user := &User{
        ID:        generateID(),
        Name:      params.Name,
```

Protocol comparison summary

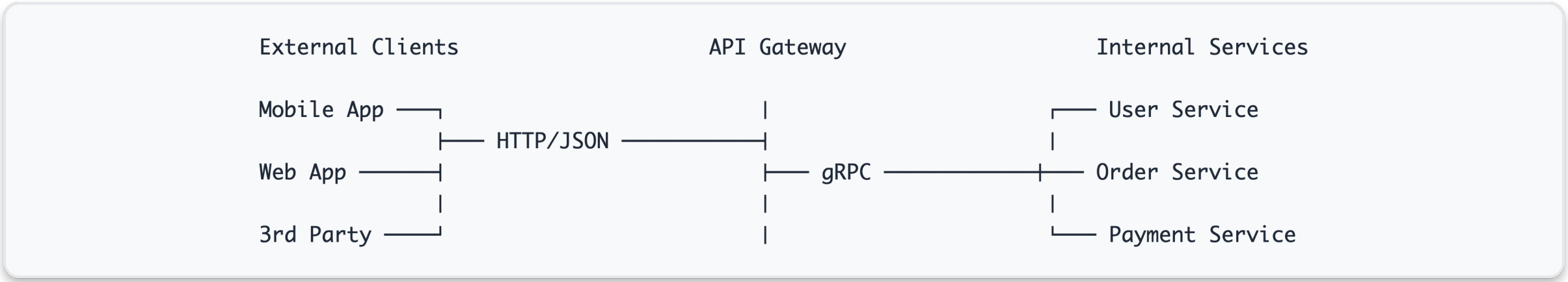
Choosing the right protocol for your use case:

Protocol	Best For	Pros	Cons	Size Efficiency
HTTP/JSON	REST APIs, Web apps	Simple, debuggable	Verbose, slower	★★
WebSocket	Real-time chat, gaming	Bidirectional, instant	More complex	★★★
gRPC	Microservices, high-performance	Fast, type-safe, streaming	Learning curve	★★★★★
JSON-RPC	Simple RPC calls	Easy to implement	Limited features	★★★
WebRTC	Video/audio calls	P2P, media streaming	Complex setup	★★★★

Real-world usage patterns:

Internal vs External API architecture

Why different protocols for different clients?



External API requirements (HTTP/REST):

- **Browser compatibility**: Native fetch API, no special libraries
- **Simplicity**: Easy to understand and debug with curl/Postman
- **Tooling**: Extensive ecosystem (Swagger, OpenAPI, etc.)
- **Caching**: HTTP caching with CDNs and proxies
- **Firewall friendly**: Port 80/443, well-known protocol

Internal API requirements (gRPC):

gRPC performance benefits for microservices

HTTP/JSON inter-service call

```
POST /api/orders HTTP/1.1
Host: order-service:8080
Content-Type: application/json
Accept: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiI...
User-Agent: UserService/1.0
Content-Length: 156

{
  "user_id": "user_123",
  "items": [
    {
      "product_id": "prod_456",
      "quantity": 2,
      "price": 29.99
    }
  ],
  "shipping_address": {
```

gRPC inter-service call

```
// order.proto
message CreateOrderRequest {
  string user_id = 1;
  repeated OrderItem items = 2;
  Address shipping_address = 3;
}

message OrderItem {
  string product_id = 1;
  int32 quantity = 2;
  double price = 3;
}
```

Binary representation:

```
0A 08 75 73 65 72 5F 31 32 33 12 0F 0A 08 70 72...
```

Size: ~95 bytes total

When to use gRPC vs HTTP APIs

Decision matrix for API protocol selection:

Use Case	gRPC	HTTP/REST	Reason
Mobile app → API Gateway	❌	✅	Browser/mobile compatibility
Web frontend → Backend	❌	✅	JavaScript fetch API, debugging
Service → Service	✅	❌	Performance, type safety
Real-time updates	✅	❌	Bidirectional streaming
High-frequency calls	✅	❌	Lower latency, smaller payload
3rd party integrations	❌	✅	Industry standard, tooling
Public API	❌	✅	Documentation, accessibility

Hybrid architecture pattern:

```
External Layer (HTTP/REST)
├── Mobile/Web clients use JSON APIs
├── 3rd party integrations use REST
└── Public documentation with OpenAPI
```

Simple gRPC service example

Protocol definition

```
// calculator.proto
syntax = "proto3";
package calculator;

service Calculator {
  rpc Add(Numbers) returns (Result);
  rpc Multiply(Numbers) returns (Result);
}

message Numbers {
  double a = 1;
  double b = 2;
}

message Result {
  double value = 1;
}
```

Go server implementation

```
type calculatorServer struct {
    pb.UnimplementedCalculatorServer
}

func (s *calculatorServer) Add(ctx context.Context, req *pb.Numbers) (*pb.Result, error) {
    result := req.A + req.B
    return &pb.Result{Value: result}, nil
}

func (s *calculatorServer) Multiply(ctx context.Context, req *pb.Numbers) (*pb.Result, error) {
    result := req.A * req.B
    return &pb.Result{Value: result}, nil
}

func main() {
    lis, _ := net.Listen("tcp", ":50051")
    server := grpc.NewServer()

    pb.RegisterCalculatorServer(server, &calculatorServer{})
    server.Serve(lis)
}
```

Generate code:

gRPC streaming example

Server streaming proto

```
service NumberService {
  // Server streaming: send multiple responses
  rpc GetPrimes(PrimeRequest) returns (stream Prime);

  // Client streaming: receive multiple requests
  rpc SumNumbers(stream Number) returns (Sum);

  // Bidirectional streaming
  rpc Chat(stream ChatMessage) returns (stream ChatMessage);
}

message PrimeRequest {
  int32 limit = 1;
}

message Prime {
  int32 value = 1;
}
```

Server streaming implementation

```
func (s *numberServer) GetPrimes(req *pb.PrimeRequest, stream pb.NumberService_GetPrimesServer) error {
    for i := 2; i <= int(req.Limit); i++ {
        if isPrime(i) {
            prime := &pb.Prime{Value: int32(i)}
            if err := stream.Send(prime); err != nil {
                return err
            }

            // Send one prime every 100ms
            time.Sleep(100 * time.Millisecond)
        }
    }
    return nil
}

// Client usage:
stream, _ := client.GetPrimes(ctx, &pb.PrimeRequest{Limit: 100})
for {
    prime, err := stream.Recv()
    if err == io.EOF {
        break
    }
    fmt.Printf("Prime: %d\n", prime.Value)
}
```

User Service with gRPC

User service proto definition

```proto
// user_service.proto
syntax = "proto3";
package user;

import "google/protobuf/timestamp.proto";
import "google/protobuf/empty.proto";

service UserService {
  rpc CreateUser(CreateUserRequest) returns (User);
  rpc GetUser(GetUserRequest) returns (User);
  rpc UpdateUser(UpdateUserRequest) returns (User);
  rpc DeleteUser(DeleteUserRequest) returns (google.protobuf.Empty);
  rpc ListUsers(ListUsersRequest) returns (stream User);
  rpc ValidateUser(ValidateUserRequest) returns (UserValidation);
}

message User {
  string id = 1;
  string name = 2;
  string email = 3;
  string phone = 4;
  bool active = 5;
  google.protobuf.Timestamp created_at = 6;
  google.protobuf.Timestamp updated_at = 7;
}

message CreateUserRequest {
  string name = 1;
  string email = 2;
  string phone = 3;
}

message ValidateUserRequest {
  string user_id = 1;
}
```

User service implementation

```go
package user

import (
    "context"
    "database/sql"
    "time"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "google.golang.org/protobuf/types/known/timestamppb"
    pb "github.com/example/user/proto"
)

type UserServer struct {
    pb.UnimplementedUserServiceServer
    db *sql.DB
}

func (s *UserServer) CreateUser(ctx context.Context, req *pb.CreateUserRequest) (*pb.User, error) {
    // Validate input
    if req.Name == "" {
        return nil, status.Errorf(codes.InvalidArgument, "name is required")
    }
    if req.Email == "" {
        return nil, status.Errorf(codes.InvalidArgument, "email is required")
    }

    // Check if user exists
    var exists bool
    err := s.db.QueryRow("SELECT EXISTS(SELECT 1 FROM users WHERE email = $1)", req.Email).Scan(&exists)
    if err != nil {
        return nil, status.Errorf(codes.Internal, "database error: %v", err)
    }
    if exists {
        return nil, status.Errorf(codes.AlreadyExists, "user with email %s already exists", req.Email)
    }

    // Create user
    user := &pb.User{
        Id:    generateUUID(),
        Name:  req.Name,
        Email: req.Email,
        Phone: req.Phone,
        Active: true,
        CreatedAt: timestamppb.Now(),
        UpdatedAt: timestamppb.Now(),
    }

    // Save to database
    _, err = s.db.Exec(`
        INSERT INTO users (id, name, email, phone, active, created_at, updated_at)
        VALUES ($1, $2, $3, $4, $5, $6, $7)`,
        user.Id, user.Name, user.Email, user.Phone, user.Active,
        user.CreatedAt.AsTime(), user.UpdatedAt.AsTime())
```

Order Service with inter-service communication

Order service proto

```
// order_service.proto
syntax = "proto3";
package order;

service OrderService {
  rpc CreateOrder(CreateOrderRequest) returns (Order);
  rpc GetOrder(GetOrderRequest) returns (Order);
  rpc UpdateOrderStatus(UpdateOrderStatusRequest) returns (Order);
  rpc ListUserOrders(ListUserOrdersRequest) returns (stream Order);
}

message Order {
  string id = 1;
  string user_id = 2;
  repeated OrderItem items = 3;
  OrderStatus status = 4;
  double total_amount = 5;
  google.protobuf.Timestamp created_at = 6;
}

message OrderItem {
  string product_id = 1;
  int32 quantity = 2;
  double price = 3;
  string name = 4;
}

enum OrderStatus {
  PENDING = 0;
  CONFIRMED = 1;
  SHIPPED = 2;
```

Order service with user validation

```
package order

import (
    "context"
    "fmt"
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    userpb "github.com/example/user/proto"
    orderpb "github.com/example/order/proto"
)

type OrderServer struct {
    orderpb.UnimplementedOrderServiceServer
    userClient userpb.UserServiceClient
    db         *sql.DB
}

func NewOrderServer(userServiceAddr string, db *sql.DB) (*OrderServer, error) {
    conn, err := grpc.Dial(userServiceAddr, grpc.WithInsecure())
    if err != nil {
        return nil, fmt.Errorf("failed to connect to user service: %v", err)
    }

    return &OrderServer{
        userClient: userpb.NewUserServiceClient(conn),
        db:         db,
    }, nil
}

func (s *OrderServer) CreateOrder(ctx context.Context, req *orderpb.CreateOrderRequest) (*orderpb.Order, error) {
    // Validate user exists via gRPC call
    userValidation, err := s.userClient.ValidateUser(ctx, &userpb.ValidateUserRequest{
        UserId: req.UserId,
    })
    if err != nil {
        return nil, status.Errorf(codes.InvalidArgument, "user validation failed: %v", err)
    }
    if !userValidation.IsValid {
        return nil, status.Errorf(codes.InvalidArgument, "invalid user: %s", userValidation.Reason)
    }

    // Calculate total
    var totalAmount float64
    for _, item := range req.Items {
        totalAmount += item.Price * float64(item.Quantity)
    }

    // Create order
    order := &orderpb.Order{
        Id:          generateOrderID(),
```

Notification Service with streaming

Notification service proto

```
// notification_service.proto
syntax = "proto3";
package notification;

service NotificationService {
  // Server streaming - send notifications to clients
  rpc Subscribe(SubscribeRequest) returns (stream Notification);

  // Send notification to specific user
  rpc SendNotification(SendNotificationRequest) returns (google.protobuf.Empty);

  // Broadcast to all users
  rpc Broadcast(BroadcastRequest) returns (google.protobuf.Empty);
}

message Notification {
  string id = 1;
  string user_id = 2;
  NotificationType type = 3;
  string title = 4;
  string message = 5;
  map<string, string> data = 6;
  google.protobuf.Timestamp created_at = 7;
}

enum NotificationType {
  INFO = 0;
  WARNING = 1;
  ERROR = 2;
  SUCCESS = 3;
  ORDER_UPDATE = 4;
  USER_MESSAGE = 5;
}

message SubscribeRequest {
  string user_id = 1;
  repeated NotificationType types = 2;
}

message SendNotificationRequest {
  string user_id = 1;
  NotificationType type = 2;
```

Real-time notification streaming

```
package notification

import (
    "context"
    "sync"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    pb "github.com/example/notification/proto"
)

type NotificationServer struct {
    pb.UnimplementedNotificationServiceServer
    subscribers map[string][]pb.NotificationService_SubscribeServer
    mutex       sync.RWMutex
}

func (s *NotificationServer) Subscribe(req *pb.SubscribeRequest, stream pb.NotificationService_SubscribeServer) error {
    // Add subscriber
    s.mutex.Lock()
    if s.subscribers[req.UserId] == nil {
        s.subscribers[req.UserId] = make([]pb.NotificationService_SubscribeServer, 0)
    }
    s.subscribers[req.UserId] = append(s.subscribers[req.UserId], stream)
    s.mutex.Unlock()

    // Send welcome notification
    welcomeNotification := &pb.Notification{
        Id:        generateNotificationID(),
        UserId:    req.UserId,
        Type:      pb.NotificationType_INFO,
        Title:     "Connected",
        Message:   "You are now connected to notifications",
        CreatedAt: timestamppb.Now(),
    }

    if err := stream.Send(welcomeNotification); err != nil {
        return err
    }

    // Keep connection alive until client disconnects
    <-stream.Context().Done()

    // Remove subscriber
    s.removeSubscriber(req.UserId, stream)
    return nil
}

func (s *NotificationServer) SendNotification(ctx context.Context, req *pb.SendNotificationRequest) (*emptypb.Empty, error) {
    notification := &pb.Notification{
        Id:        generateNotificationID(),
        UserId:    req.UserId,
        Type:      req.Type,
        Title:     req.Title,
        Message:   req.Message,
        Data:      req.Data,
        CreatedAt: timestamppb.Now(),
    }

    s.mutex.RLock()
    subscribers := s.subscribers[req.UserId]
    s.mutex.RUnlock()

    // Send to all user's active connections
    for _, stream := range subscribers {
        go func(stream pb.NotificationService_SubscribeServer) {
            if err := stream.Send(notification); err != nil {
                // Remove failed stream
```

gRPC middleware and interceptors

Authentication interceptor

```go
package middleware

import (
    "context"
    "strings"
    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/metadata"
    "google.golang.org/grpc/status"
)

func AuthInterceptor(authService AuthService) grpc.UnaryServerInterceptor {
    return func(ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
        // Skip auth for certain methods
        if isPublicMethod(info.FullMethod) {
            return handler(ctx, req)
        }

        // Extract metadata
        md, ok := metadata.FromIncomingContext(ctx)
        if !ok {
            return nil, status.Errorf(codes.Unauthenticated, "missing metadata")
        }

        // Get authorization header
        authHeaders := md.Get("authorization")
        if len(authHeaders) == 0 {
            return nil, status.Errorf(codes.Unauthenticated, "missing authorization token")
        }

        token := strings.TrimPrefix(authHeaders[0], "Bearer ")

        // Validate token
        userID, err := authService.ValidateToken(token)
        if err != nil {
            return nil, status.Errorf(codes.Unauthenticated, "invalid token: %v", err)
        }

        // Add user ID to context
        ctx = context.WithValue(ctx, "user_id", userID)

        return handler(ctx, req)
    }
}
```

Logging and metrics interceptor

```go
package middleware

import (
    "context"
    "log"
    "time"
    "google.golang.org/grpc"
    "google.golang.org/grpc/status"
    "github.com/prometheus/client_golang/prometheus"
)

var (
    grpcRequestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "grpc_requests_total",
            Help: "Total number of gRPC requests",
        },
        []string{"method", "status"},
    )

    grpcRequestDuration = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name: "grpc_request_duration_seconds",
            Help: "gRPC request duration in seconds",
        },
        []string{"method"},
    )
)

func LoggingInterceptor(ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
    start := time.Now()

    log.Printf("gRPC request started: %s", info.FullMethod)

    resp, err := handler(ctx, req)

    duration := time.Since(start)
    statusCode := status.Code(err)

    log.Printf("gRPC request completed: %s, duration: %v, status: %s",
        info.FullMethod, duration, statusCode)

    // Record metrics
    grpcRequestsTotal.WithLabelValues(info.FullMethod, statusCode.String()).Inc()
    grpcRequestDuration.WithLabelValues(info.FullMethod).Observe(duration.Seconds())

    return resp, err
}
```

Complete gRPC microservices setup

Server setup with all services

```go
package main

import (
    "log"
    "net"
    "google.golang.org/grpc"
    "google.golang.org/grpc/reflection"
    userpb "github.com/example/user/proto"
    orderpb "github.com/example/order/proto"
    notificationpb "github.com/example/notification/proto"
)

func main() {
    // Database connection
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/db")
    if err != nil {
        log.Fatalf("Failed to connect to database: %v", err)
    }
    defer db.Close()

    // Create gRPC server with interceptors
    server := grpc.NewServer(
        grpc.ChainUnaryInterceptor(
            middleware.LoggingInterceptor,
            middleware.AuthInterceptor(authService),
            middleware.MetricsInterceptor,
        ),
        grpc.ChainStreamInterceptor(
            middleware.StreamLoggingInterceptor,
            middleware.StreamAuthInterceptor(authService),
        ),
    )

    // Register services
    userServer := user.NewUserServer(db)
    orderServer, _ := order.NewOrderServer("localhost:50051", db)
    notificationServer := notification.NewNotificationServer()

    userpb.RegisterUserServiceServer(server, userServer)
    orderpb.RegisterOrderServiceServer(server, orderServer)
    notificationpb.RegisterNotificationServiceServer(server, notificationServer)

    // Enable reflection for development
    reflection.Register(server)

    // Start server
    listener, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatalf("Failed to listen: %v", err)
    }
```

Client connection pool

```go
package client

import (
    "context"
    "sync"
    "time"
    "google.golang.org/grpc"
    "google.golang.org/grpc/keepalive"
    userpb "github.com/example/user/proto"
    orderpb "github.com/example/order/proto"
)

type ServiceClients struct {
    User  userpb.UserServiceClient
    Order orderpb.OrderServiceClient
    conns []*grpc.ClientConn
    mutex sync.RWMutex
}

func NewServiceClients() (*ServiceClients, error) {
    // Connection options for production
    opts := []grpc.DialOption{
        grpc.WithInsecure(), // Use TLS in production
        grpc.WithKeepaliveParams(keepalive.ClientParameters{
            Time:                10 * time.Second,
            Timeout:             time.Second,
            PermitWithoutStream: true,
        }),
        grpc.WithDefaultCallOptions(
            grpc.MaxCallRecvMsgSize(4*1024*1024), // 4MB
            grpc.MaxCallSendMsgSize(4*1024*1024), // 4MB
        ),
    }

    // Connect to User Service
    userConn, err := grpc.Dial("user-service:50051", opts...)
    if err != nil {
        return nil, err
    }

    // Connect to Order Service
    orderConn, err := grpc.Dial("order-service:50052", opts...)
    if err != nil {
        userConn.Close()
        return nil, err
    }

    return &ServiceClients{
        User:  userpb.NewUserServiceClient(userConn),
        Order: orderpb.NewOrderServiceClient(orderConn),
        conns: []*grpc.ClientConn{userConn, orderConn},
    }, nil
}

func (sc *ServiceClients) Close() error {
    sc.mutex.Lock()
    defer sc.mutex.Unlock()
```

Microservices: Why split your application?

The monolith problem:

```
Single Large Application
├── User Management (Team A needs Python)
├── Payment Processing (Team B needs Java)
├── Email Service (Team C needs Node.js)
├── File Upload (Team D needs Go)
└── Analytics (Team E needs Scala)
```

Problems:

- 🔴 One technology stack for everything
- 🔴 Deploy entire app for small changes
- 🔴 One team's bug affects everyone
- 🔴 Difficult to scale individual features

The microservices solution:

Database design patterns for microservices

Database per Service pattern

```
User Service ↔ PostgreSQL (Users DB)
Order Service ↔ MongoDB (Orders DB)
Product Service ↔ Elasticsearch (Products DB)
Analytics Service ↔ ClickHouse (Analytics DB)
```

Benefits:

- ✅ Service autonomy and independence
- ✅ Technology diversity (choose best DB for use case)
- ✅ Data isolation and security
- ✅ Independent scaling

Challenges:

- 🔴 Cross-service queries become complex
- 🔴 Data consistency across services

Data consistency patterns

Saga Pattern (Distributed Transactions)

```
type OrderSaga struct {
    steps []SagaStep
}

type SagaStep struct {
    Execute     func() error
    Compensate  func() error
    Executed    bool
}

func (s *OrderSaga) Execute() error {
    for i, step := range s.steps {
        if err := step.Execute(); err != nil {
            // Compensate all executed steps
            s.rollback(i)
            return err
        }
        s.steps[i].Executed = true
    }
    return nil
}

func (s *OrderSaga) rollback(failedStep int) {
    for i := failedStep - 1; i >= 0; i-- {
        if s.steps[i].Executed {
            s.steps[i].Compensate()
```

Example: Order processing saga

```
func CreateOrderSaga(orderID, userID string, items []Item) *OrderSaga {
    return &OrderSaga{
        steps: []SagaStep{
            {
                Execute: func() error {
                    return reserveInventory(items)
                },
                Compensate: func() error {
                    return releaseInventory(items)
                },
            },
            {
                Execute: func() error {
                    return chargePayment(userID, total)
                },
                Compensate: func() error {
                    return refundPayment(userID, total)
                },
            },
            {
                Execute: func() error {
                    return createOrder(orderID, userID, items)
                },
                Compensate: func() error {
                    return cancelOrder(orderID)
                },
            },
```

Event Sourcing pattern

Traditional state storage

```
// Store current state only
type User struct {
    ID       string `json:"id"`
    Name     string `json:"name"`
    Email    string `json:"email"`
    Status   string `json:"status"`
    Balance  float64 `json:"balance"`
}

// Lost information:
// - When was user created?
// - How many times email changed?
// - Balance history?
// - Who made changes?
```

Event sourcing approach

```
type Event struct {
    ID        string      `json:"id"`
    Type      string      `json:"type"`
```

Rebuilding state from events

```
func (u *User) ApplyEvent(event Event) {
    switch event.Type {
    case "UserCreated":
        data := event.Data.(UserCreated)
        u.ID = data.UserID
        u.Name = data.Name
        u.Email = data.Email
        u.Status = "active"
        u.Balance = 0.0

    case "EmailChanged":
        data := event.Data.(EmailChanged)
        u.Email = data.NewEmail

    case "BalanceUpdated":
        data := event.Data.(BalanceUpdated)
        u.Balance = data.NewBalance
    }
}

func GetUserState(userID string) (*User, error) {
    events, err := getEventsByUserID(userID)
    if err != nil {
        return nil, err
    }

    user := &User{}
    for _, event := range events {
        user.ApplyEvent(event)
    }
```

CQRS (Command Query Responsibility Segregation)

Separation of reads and writes

```go
// Command side (writes)
type UserCommandHandler struct {
    eventStore EventStore
}

func (h *UserCommandHandler) CreateUser(cmd CreateUserCommand) error {
    // Validate command
    if cmd.Email == "" {
        return errors.New("email required")
    }

    // Create event
    event := Event{
        Type: "UserCreated",
        Data: UserCreated{
            UserID: cmd.UserID,
            Name:   cmd.Name,
            Email:  cmd.Email,
        },
        Timestamp: time.Now(),
    }

    // Store event
    return h.eventStore.SaveEvent(event)
}

func (h *UserCommandHandler) UpdateEmail(cmd UpdateEmailCommand) error {
    // Business logic validation
    if !isValidEmail(cmd.NewEmail) {
        return errors.New("invalid email")
    }

    event := Event{
        Type: "EmailChanged",
```

Query side (reads)

```go
// Query side (reads) - optimized for queries
type UserQueryHandler struct {
    readDB ReadDatabase // Could be MongoDB, Elasticsearch, etc.
}

type UserView struct {
    ID           string    `json:"id"`
    Name         string    `json:"name"`
    Email        string    `json:"email"`
    Status       string    `json:"status"`
    Balance      float64   `json:"balance"`
    LastActivity time.Time `json:"last_activity"`
    TotalOrders  int       `json:"total_orders"`
    // Denormalized data for fast queries
}

func (h *UserQueryHandler) GetUser(userID string) (*UserView, error) {
    return h.readDB.FindUserByID(userID)
}

func (h *UserQueryHandler) SearchUsers(query string) ([]*UserView, error) {
    return h.readDB.SearchUsers(query)
}

// Event handler updates read models
func (h *UserQueryHandler) HandleUserCreated(event Event) {
    data := event.Data.(UserCreated)
    userView := &UserView{
        ID:           data.UserID,
```

API versioning strategies

URL versioning

```go
// v1/users endpoint
func (h *UserHandler) GetUserV1(w http.ResponseWriter, r *http.Request) {
    user := &UserV1{
        ID:    "123",
        Name:  "John Doe",
        Email: "john@example.com",
    }
    json.NewEncoder(w).Encode(user)
}

// v2/users endpoint (added phone field)
func (h *UserHandler) GetUserV2(w http.ResponseWriter, r *http.Request) {
    user := &UserV2{
        ID:    "123",
        Name:  "John Doe",
        Email: "john@example.com",
        Phone: "+1-555-1234", // New field
    }
    json.NewEncoder(w).Encode(user)
}

// Routing
mux.HandleFunc("/api/v1/users/{id}", h.GetUserV1)
mux.HandleFunc("/api/v2/users/{id}", h.GetUserV2)
```

Header versioning

```go
func (h *UserHandler) GetUser(w http.ResponseWriter, r *http.Request) {
    version := r.Header.Get("API-Version")

    switch version {
    case "v1", "1.0":
        h.getUserV1(w, r)
    case "v2", "2.0":
        h.getUserV2(w, r)
    default:
        h.getUserLatest(w, r)
    }
}

// Client usage:
req.Header.Set("API-Version", "v2")
```

gRPC versioning

```protobuf
// user_v1.proto
syntax = "proto3";
package user.v1;

service UserService {
  rpc GetUser(GetUserRequest) returns (User);
}
```

Backward compatibility strategies

Additive changes (safe)

```go
// V1 - Original
type UserV1 struct {
    ID    string `json:"id"`
    Name  string `json:"name"`
    Email string `json:"email"`
}

// V2 - Add optional fields (backward compatible)
type UserV2 struct {
    ID      string  `json:"id"`
    Name    string  `json:"name"`
    Email   string  `json:"email"`
    Phone   *string `json:"phone,omitempty"`   // Optional
    Avatar  *string `json:"avatar,omitempty"`  // Optional
}

// V1 clients ignore new fields, still work
```

Breaking changes handling

```go
// Deprecation strategy
type DeprecatedUserResponse struct {
    ID         string `json:"id"`
    Name       string `json:"name"`
    Email      string `json:"email"`
    Deprecated bool   `json:"_deprecated"` // Warning flag
    NewAPIURL  string `json:"_new_api_url"`
}

func (h *UserHandler) GetUserDeprecated(w http.ResponseWriter, r *http.Request) {
    // Add deprecation warning header
    w.Header().Set("Deprecation", "version=\"v1\"")
    w.Header().Set("Sunset", "2024-12-31")
    w.Header().Set("Link", "</api/v2/users>; rel=\"successor-version\"")

    user := &DeprecatedUserResponse{
        ID:         "123",
        Name:       "John Doe",
        Email:      "john@example.com",
        Deprecated: true,
        NewAPIURL: "/api/v2/users/123",
    }

    json.NewEncoder(w).Encode(user)
}
```

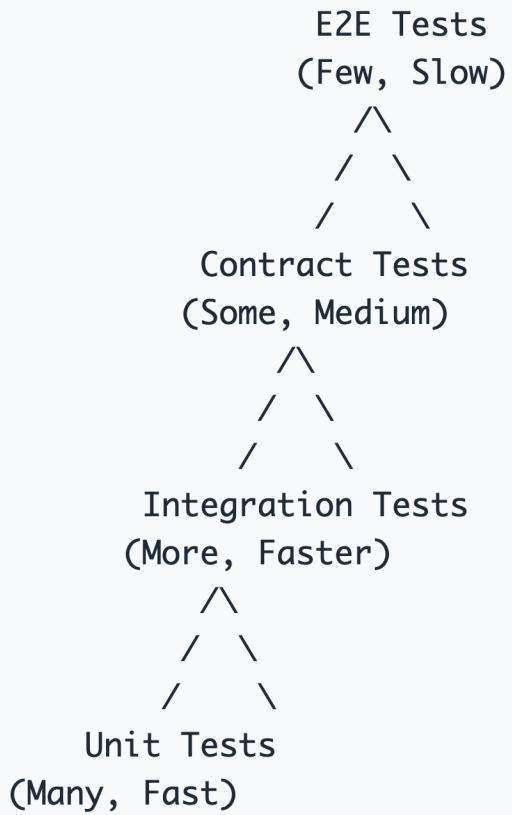
Non-breaking API evolution

```go
func (h *UserHandler) CreateUser(w http.ResponseWriter, r *http.Request) {
```

Version support matrix

Microservices testing strategies

Testing pyramid for microservices



Test types and responsibilities:

Unit testing example

Service under test

```go
package user

type UserService struct {
    repo UserRepository
    validator EmailValidator
}

type UserRepository interface {
    Save(user *User) error
    FindByID(id string) (*User, error)
    FindByEmail(email string) (*User, error)
}

type EmailValidator interface {
    IsValid(email string) bool
}

func (s *UserService) CreateUser(req CreateUserRequest) (*User, error) {
    // Validation
    if req.Name == "" {
        return nil, errors.New("name is required")
    }

    if !s.validator.IsValid(req.Email) {
        return nil, errors.New("invalid email")
    }

    // Check if user exists
    existing, _ := s.repo.FindByEmail(req.Email)
    if existing != nil {
        return nil, errors.New("user already exists")
    }

    // Create user
    user := &User{
        ID:    generateID(),
        Name:  req.Name,
        Email: req.Email,
    }

    if err := s.repo.Save(user); err != nil {
        return nil, err
```

Unit test with mocks

```go
package user

import (
    "testing"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/mock"
)

type MockUserRepository struct {
    mock.Mock
}

func (m *MockUserRepository) Save(user *User) error {
    args := m.Called(user)
    return args.Error(0)
}

func (m *MockUserRepository) FindByEmail(email string) (*User, error) {
    args := m.Called(email)
    return args.Get(0).(*User), args.Error(1)
}

type MockEmailValidator struct {
    mock.Mock
}

func (m *MockEmailValidator) IsValid(email string) bool {
    args := m.Called(email)
    return args.Bool(0)
}

func TestUserService_CreateUser_Success(t *testing.T) {
    // Arrange
    mockRepo := new(MockUserRepository)
    mockValidator := new(MockEmailValidator)

    service := &UserService{
        repo: mockRepo,
        validator: mockValidator,
    }

    req := CreateUserRequest{
        Name:  "John Doe",
        Email: "john@example.com",
    }

    mockValidator.On("IsValid", "john@example.com").Return(true)
    mockRepo.On("FindByEmail", "john@example.com").Return((*User)(nil), errors.New("not found"))
    mockRepo.On("Save", mock.AnythingOfType("*User")).Return(nil)

    // Act
    user, err := service.CreateUser(req)

    // Assert
```

Integration testing with test containers

Test with real database

```go
package user

import (
    "testing"
    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/modules/postgres"
)

func TestUserService_Integration(t *testing.T) {
    // Start PostgreSQL container
    ctx := context.Background()
    pgContainer, err := postgres.RunContainer(ctx,
        testcontainers.WithImage("postgres:15"),
        postgres.WithDatabase("testdb"),
        postgres.WithUsername("testuser"),
        postgres.WithPassword("testpass"),
    )
    require.NoError(t, err)
    defer pgContainer.Terminate(ctx)

    // Get connection string
    connStr, err := pgContainer.ConnectionString(ctx)
    require.NoError(t, err)

    // Setup database
    db, err := sql.Open("postgres", connStr)
    require.NoError(t, err)
    defer db.Close()

    // Run migrations
    err = runMigrations(db)
    require.NoError(t, err)

    // Create service with real repository
    repo := NewPostgresUserRepository(db)
    validator := NewEmailValidator()
    service := &UserService{
        repo: repo,
        validator: validator,
    }

    // Test
    user, err := service.CreateUser(CreateUserRequest{
        Name:  "John Doe",
        Email: "john@example.com",
```

Test with external API

```go
func TestUserService_WithExternalEmailValidator(t *testing.T) {
    // Start HTTP server mock
    server := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if r.URL.Path == "/validate" {
            email := r.URL.Query().Get("email")
            if email == "valid@example.com" {
                w.WriteHeader(http.StatusOK)
                json.NewEncoder(w).Encode(map[string]bool{"valid": true})
            } else {
                w.WriteHeader(http.StatusOK)
                json.NewEncoder(w).Encode(map[string]bool{"valid": false})
            }
        }
    }))
    defer server.Close()

    // Create service with HTTP email validator
    validator := NewHTTPEmailValidator(server.URL)
    service := &UserService{
        repo: NewInMemoryUserRepository(),
        validator: validator,
    }

    // Test valid email
    user, err := service.CreateUser(CreateUserRequest{
        Name:  "John Doe",
        Email: "valid@example.com",
    })
    assert.NoError(t, err)
    assert.NotNil(t, user)

    // Test invalid email
    _, err = service.CreateUser(CreateUserRequest{
        Name:  "Jane Doe",
        Email: "invalid@example.com",
    })
```

Contract testing with Pact

Consumer test (User Service)

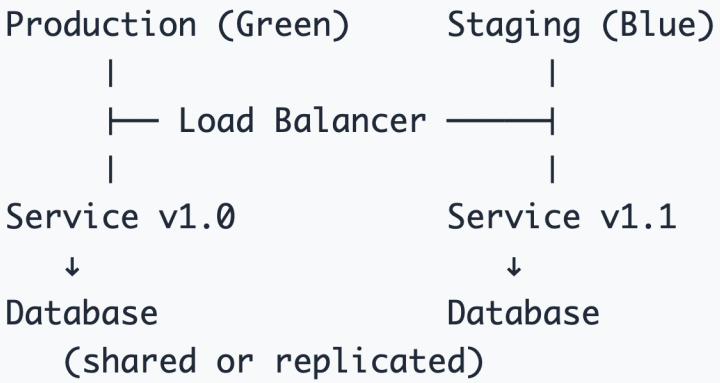
```go
package user

import (
    "testing"
    "github.com/pact-foundation/pact-go/dsl"
)

func TestUserService_GetUserFromOrderService(t *testing.T) {
    // Create Pact
    pact := &dsl.Pact{
        Consumer: "UserService",
        Provider: "OrderService",
    }
    defer pact.Teardown()

    // Define interaction
    pact.
        AddInteraction().
        Given("user 123 exists").
        UponReceiving("a request for user 123").
        WithRequest(dsl.Request{
            Method: "GET",
            Path:   dsl.String("/users/123"),
            Headers: dsl.MapMatcher{
                "Content-Type": dsl.String("application/json"),
            },
        }).
        WillRespondWith(dsl.Response{
            Status: 200,
            Headers: dsl.MapMatcher{
                "Content-Type": dsl.String("application/json"),
            },
            Body: dsl.Match(&User{
                ID:    "123",
                Name:  "John Doe",
                Email: "john@example.com",
            }),
        })

    // Test
    err := pact.Verify(func() error {
        client := NewOrderServiceClient(fmt.Sprintf("http://localhost:%d", pact.Server.Port))
        user, err := client.GetUser("123")

        assert.Equal(t, "123", user.ID)
        assert.Equal(t, "John Doe", user.Name)
```

Provider test (Order Service)

```go
package order

import (
    "testing"
    "github.com/pact-foundation/pact-go/dsl"
)

func TestOrderService_PactProvider(t *testing.T) {
    // Setup test server
    server := setupTestServer()
    defer server.Close()

    // Create provider verifier
    pact := &dsl.Pact{}

    // Verify against consumer contracts
    _, err := pact.VerifyProvider(t, dsl.VerifyRequest{
        ProviderBaseURL:        server.URL,
        PactURLs:               []string{"./pacts/userservice-orderservice.json"},
        ProviderStatesSetupURL: server.URL + "/setup",
        StateHandlers: dsl.StateHandlers{
            "user 123 exists": func() error {
                // Setup test data
                return createTestUser("123", "John Doe", "john@example.com")
            },
        },
    })

    assert.NoError(t, err)
}

func setupTestServer() *httptest.Server {
    mux := http.NewServeMux()

    mux.HandleFunc("/users/123", func(w http.ResponseWriter, r *http.Request) {
        user := User{
            ID:    "123",
            Name:  "John Doe",
            Email: "john@example.com",
        }
        json.NewEncoder(w).Encode(user)
```

Service deployment patterns

Blue-Green deployment

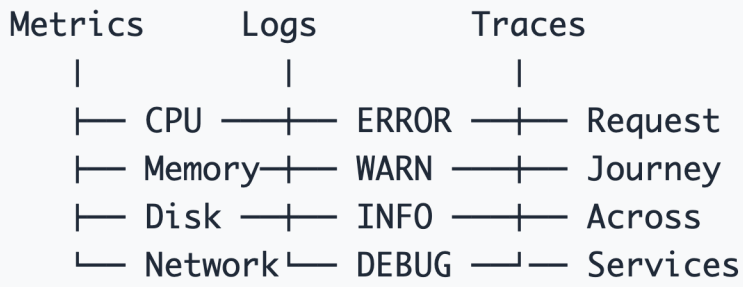


Rolling deployment

```
Instances: [v1.0] [v1.0] [v1.0] [v1.0]
Step 1:    [v1.1] [v1.0] [v1.0] [v1.0]  25% updated
Step 2:    [v1.1] [v1.1] [v1.0] [v1.0]  50% updated
Step 3:    [v1.1] [v1.1] [v1.1] [v1.0]  75% updated
Step 4:    [v1.1] [v1.1] [v1.1] [v1.1]  100% updated
```

Monitoring and observability

The three pillars of observability



Distributed tracing example

```go
import (
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/trace"
)

func (s *UserService) CreateUser(ctx context.Context, req CreateUserRequest) (*User, error) {
    // Start span
    tracer := otel.Tracer("user-service")
    ctx, span := tracer.Start(ctx, "CreateUser")
    defer span.End()

    // Add attributes
    span.SetAttributes(
        attribute.String("user.email", req.Email),
        attribute.String("user.name", req.Name),
    )

    // Validate (traced)
```

Custom metrics

```go
import (
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promauto"
)

var (
    userCreationsTotal = promauto.NewCounterVec(
        prometheus.CounterOpts{
            Name: "user_creations_total",
            Help: "Total number of user creations",
        },
        []string{"status"},
    )

    userCreationDuration = promauto.NewHistogramVec(
        prometheus.HistogramOpts{
            Name: "user_creation_duration_seconds",
            Help: "Duration of user creation operations",
            Buckets: prometheus.DefBuckets,
        },
        []string{"status"},
    )

    activeUsers = promauto.NewGauge(
        prometheus.GaugeOpts{
            Name: "active_users_total",
            Help: "Total number of active users",
        },
    )
)

func (s *UserService) CreateUser(ctx context.Context, req CreateUserRequest) (*User, error) {
    start := time.Now()

    user, err := s.createUser(ctx, req)

    // Record metrics
    duration := time.Since(start).Seconds()

    if err != nil {
        userCreationsTotal.WithLabelValues("error").Inc()
        userCreationDuration.WithLabelValues("error").Observe(duration)
    } else {
        userCreationsTotal.WithLabelValues("success").Inc()
        userCreationDuration.WithLabelValues("success").Observe(duration)
```

Health checks and circuit breakers

Health check implementation

```
type HealthChecker struct {
    db    *sql.DB
    redis *redis.Client
    deps  []Dependency
}

type HealthStatus struct {
    Status      string            `json:"status"`
    Version     string            `json:"version"`
    Uptime      string            `json:"uptime"`
    Dependencies map[string]string `json:"dependencies"`
}

func (h *HealthChecker) Check() HealthStatus {
    status := HealthStatus{
        Status:       "healthy",
        Version:      "1.0.0",
        Uptime:       time.Since(startTime).String(),
        Dependencies: make(map[string]string),
    }

    // Check database
    if err := h.db.Ping(); err != nil {
        status.Status = "unhealthy"
        status.Dependencies["database"] = "failed"
    } else {
        status.Dependencies["database"] = "healthy"
    }

    // Check Redis
    if err := h.redis.Ping().Err(); err != nil {
        status.Status = "degraded"
        status.Dependencies["cache"] = "failed"
    } else {
        status.Dependencies["cache"] = "healthy"
    }

    // Check external dependencies
    for _, dep := range h.deps {
        if err := dep.HealthCheck(); err != nil {
            status.Status = "degraded"
            status.Dependencies[dep.Name] = "failed"
        } else {
            status.Dependencies[dep.Name] = "healthy"
```

Circuit breaker pattern

```
type CircuitBreaker struct {
    maxFailures   int
    resetTimeout  time.Duration
    failures      int
    lastFailTime  time.Time
    state         CircuitState
    mutex         sync.RWMutex
}

type CircuitState int

const (
    Closed CircuitState = iota  // Normal operation
    Open                        // Failing, blocking calls
    HalfOpen                    // Testing if service recovered
)

func (cb *CircuitBreaker) Call(fn func() error) error {
    cb.mutex.RLock()
    state := cb.state
    failures := cb.failures
    cb.mutex.RUnlock()

    // Circuit is open, reject immediately
    if state == Open {
        if time.Since(cb.lastFailTime) < cb.resetTimeout {
            return errors.New("circuit breaker open")
        }
        // Try to half-open
        cb.mutex.Lock()
        cb.state = HalfOpen
        cb.mutex.Unlock()
    }

    // Execute function
    err := fn()

    cb.mutex.Lock()
    defer cb.mutex.Unlock()

    if err != nil {
        cb.failures++
        cb.lastFailTime = time.Now()

        if cb.failures >= cb.maxFailures {
            cb.state = Open
        }
        return err
    }
```

Security considerations

Service-to-service authentication

```
import (
    "github.com/golang-jwt/jwt/v4"
)

type ServiceClaims struct {
    ServiceName string `json:"service_name"`
    Permissions []string `json:"permissions"`
    jwt.StandardClaims
}

func (s *AuthMiddleware) ValidateServiceToken(token string) (*ServiceClaims, error) {
    claims := &ServiceClaims{}

    _, err := jwt.ParseWithClaims(token, claims, func(token *jwt.Token) (interface{}, error) {
        return s.jwtSecret, nil
    })

    if err != nil {
        return nil, err
    }

    // Validate service permissions
    if !s.hasPermission(claims.ServiceName, claims.Permissions) {
        return nil, errors.New("insufficient permissions")
    }

    return claims, nil
}

func (s *AuthMiddleware) ServiceAuthMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        token := r.Header.Get("Authorization")
        if token == "" {
            http.Error(w, "Missing authorization", http.StatusUnauthorized)
            return
        }

        claims, err := s.ValidateServiceToken(strings.TrimPrefix(token, "Bearer "))
        if err != nil {
            http.Error(w, "Invalid token", http.StatusUnauthorized)
            return
        }

        // Add service info to context
```

Input validation and sanitization

```
import (
    "github.com/go-playground/validator/v10"
    "html"
    "regexp"
)

type CreateUserRequest struct {
    Name  string `json:"name" validate:"required,min=2,max=50"`
    Email string `json:"email" validate:"required,email"`
    Age   int    `json:"age" validate:"min=0,max=120"`
}

type Validator struct {
    validator *validator.Validate
}

func NewValidator() *Validator {
    v := validator.New()

    // Custom validation for phone numbers
    v.RegisterValidation("phone", validatePhone)

    return &Validator{validator: v}
}

func (v *Validator) ValidateStruct(s interface{}) error {
    return v.validator.Struct(s)
}

func validatePhone(fl validator.FieldLevel) bool {
    phone := fl.Field().String()
    phoneRegex := regexp.MustCompile(`^\+?[1-9]\d{1,14}$`)
    return phoneRegex.MatchString(phone)
}

func (s *UserService) CreateUser(req CreateUserRequest) (*User, error) {
    // Validate input
    if err := s.validator.ValidateStruct(req); err != nil {
        return nil, fmt.Errorf("validation failed: %w", err)
    }

    // Sanitize input
    req.Name = html.EscapeString(strings.TrimSpace(req.Name))
    req.Email = strings.ToLower(strings.TrimSpace(req.Email))

    // Additional business validation
    if s.isEmailBlacklisted(req.Email) {
```

What we've learned about microservices

Database Design & Data Consistency

- **Database per Service**: Independent data stores for service autonomy
- **Saga Pattern**: Distributed transaction management with compensation
- **Event Sourcing**: Complete audit trail and time-travel capabilities
- **CQRS**: Separate read and write models for optimal performance

API Design & Versioning

- **Versioning strategies**: URL, header, and semantic versioning approaches
- **Backward compatibility**: Additive changes and deprecation strategies
- **Breaking changes**: Graceful migration and sunset policies
- **Contract management**: Clear API contracts and change procedures

Testing Strategies

What we've learned (continued)

Deployment & Operations

- **Deployment patterns**: Blue-green, rolling, and canary deployments
- **Health checks**: Service health monitoring and dependency checking
- **Circuit breakers**: Failure isolation and graceful degradation
- **Load balancing**: Traffic distribution and service discovery

Monitoring & Observability

- **Three pillars**: Metrics, logs, and distributed tracing
- **Custom metrics**: Business and technical metrics collection
- **Distributed tracing**: Request journey across service boundaries
- **Alerting**: Proactive monitoring and incident response

Security & Governance

- **Service authentication**: JWT tokens and service-to-service auth
- **Input validation**: Request sanitization and validation strategies

Thank You!

What's Next:

- Final Project: Build a complete microservices application

Resources:

- gRPC Documentation: https://grpc.io/docs/
- WebSocket RFC: https://tools.ietf.org/html/rfc6455
- Kubernetes Documentation: https://kubernetes.io/docs/
- Microservices Patterns: https://microservices.io/patterns/
- Course Repository: https://github.com/timur-harin/sum25-go-flutter-course

Contact:

- Email: timur.harin@mail.com
- Telegram: @timur_harin

Questions?