Microservices Architecture Project Structure

Distributed services architecture with Go. Multiple independent services with shared infrastructure.

#microservices #go #architecture #docker #grpc #distributed

PNG PDF Copy Prompt

Project Directory

```
platform/
  docker-compose.yml  Local dev stack
  docker-compose.prod.yml  Production over…
  Makefile  Build and run c…
  README.md
  .gitignore
  .env.example
  services/  Individual micr…
    user-service/  User management
      cmd/
        server/
          main.go  Service entry p…
      internal/
        handler/  HTTP/gRPC handl…
          user.go
          health.go
        service/  Business logic
          user.go
        repository/  Data access
          postgres.go
      Dockerfile
      go.mod
    order-service/  Order processing
      cmd/
        server/
          main.go
      internal/
        handler/
          order.go
        service/
          order.go
        repository/
          postgres.go
        client/  External servic…
          user_client.go
      Dockerfile
      go.mod
    api-gateway/  Public API entr…
      cmd/
        server/
          main.go
      internal/
        handler/
          routes.go
        middleware/
          auth.go
          ratelimit.go
      Dockerfile
      go.mod
  pkg/  Shared libraries
    go.mod
    logger/
      logger.go  Structured logg…
    middleware/
      tracing.go
      metrics.go
    errors/
      errors.go  Standard error …
  proto/  gRPC definitions
    user/
      user.proto
    order/
      order.proto
    buf.yaml  Buf configurati…
    buf.gen.yaml  Code generation
  deploy/  Deployment conf…
    k8s/  Kubernetes mani…
      base/
        namespace.yaml
      services/
        user-service.yaml
        order-service.yaml
        api-gateway.yaml
    terraform/  Infrastructure …
      main.tf
      variables.tf
  scripts/
    generate-proto.sh
    run-migrations.sh
```

Why This Structure?

Each service is independently deployable with its own `go.mod`, Dockerfile, and data store. The `pkg/` folder contains shared code imported as a Go module. Services communicate via gRPC internally and expose REST through the API gateway.

Key Directories

`services/` - Each microservice with its own cmd/, internal/, and Dockerfile

`pkg/` - Shared Go module for cross-service utilities

`proto/` - gRPC service definitions, generates Go code with Buf

`deploy/` - Kubernetes manifests and Terraform for infrastructure

Service Configuration

```yaml
# docker-compose.yml
services:
  user-service:
    build: ./services/user-service
    environment:
      - DATABASE_URL=postgres://...
      - GRPC_PORT=50051

  order-service:
    build: ./services/order-service
    depends_on: [user-service]
    environment:
      - USER_SERVICE_ADDR=user-service:50051
```

When To Use This

- Teams need to deploy services independently
- Different services have different scaling needs
- Polyglot persistence (different DBs per service)
- Multiple teams working on separate domains
- System requires high availability and fault isolation

Communication Patterns

`gRPC` - Service-to-service sync calls with strong typing

`REST` - External API via gateway, internal for simple cases

`Events` - Async via message queue (Kafka, NATS) for decoupling

Trade-offs

`Operational complexity` - More services = more things to monitor, deploy, debug

`Network overhead` - Remote calls slower than in-process, need retries

`Data consistency` - No transactions across services, eventual consistency

Testing Strategy

`Unit tests` - Per-service, mock external dependencies

`Integration` - Test with real DB via docker-compose

`Contract tests` - Verify gRPC/API contracts between services

`E2E` - Full stack tests through API gateway

Best Practices

- One database per service—never share databases
- Use circuit breakers for inter-service calls
- Implement health checks in every service
- Centralize logging and tracing (OpenTelemetry)
- Version your APIs and gRPC services

Naming Conventions

`Services` - `{domain}-service`: user-service, order-service

`Proto packages` - Match service name: `user.v1`, `order.v1`

`Docker images` - `{project}/{service}:{version}`