# Koronet - DevOps Engineer Practical Test

Repository scaffold containing all requested artifacts.

## Files included (top-level)

* server/ — Node.js web server code + tests
* Dockerfile — multi-stage Dockerfile for small images
* docker-compose.yml — runs web, redis, postgres
* .github/workflows/ci.yml — GitHub Actions pipeline: build, test, push to Docker Hub
* monitoring/diagram.md — high-level Prometheus/Grafana diagram (Mermaid)
* terraform/ — Terraform configuration to provision VPC, ECS cluster, task definition with sidecar, and necessary networking
* README.md — quick run / deploy instructions and environment variables

# README.md

# Koronet DevOps Practical - Sample Repository  
  
This repository contains a small Node.js web server that:  
- Responds to `/` with `Hi Koronet Team.` (newline included)  
- Connects to PostgreSQL (relational DB)  
- Connects to Redis  
- Exposes `/metrics` for Prometheus (via prom-client)  
  
Included artifacts:  
- Dockerfile (multi-stage, optimized)  
- docker-compose.yml (web + redis + postgres)  
- GitHub Actions CI pipeline  
- Terraform configs to provision AWS ECS cluster, networking, and an ECS Task with a metrics sidecar  
- Monitoring diagram (Mermaid)  
  
## Quick local run  
  
1. Copy `.env.example` to `.env` and adjust values.  
2. Start services:  
  
```bash  
docker-compose up --build

1. Visit http://localhost:3000/ -> should show Hi Koronet Team.
2. Metrics available at http://localhost:3000/metrics

## CI / CD

The included GitHub Actions workflow builds the Docker image, runs tests, and pushes to Docker Hub (requires secrets)

Secrets required in GitHub: - DOCKERHUB\_USERNAME - DOCKERHUB\_TOKEN

## AWS Deployment (Terraform)

See terraform/ for full infra. To deploy (high-level):

# configure AWS credentials in environment  
cd terraform  
terraform init  
terraform plan -out plan.tfplan  
# inspect plan, then  
terraform apply "plan.tfplan"

The Terraform will create: - VPC, subnets, internet gateway - Security Groups - ECR repository (optional) and ECS cluster - ECS Task Definition referencing the Docker Hub image and a metrics sidecar container

Note: you will need to replace image names with the image pushed by CI (docker hub) or push to ECR and update task definition.

---  
  
# server/package.json  
  
```json  
{  
 "name": "koronet-web",  
 "version": "1.0.0",  
 "main": "index.js",  
 "scripts": {  
 "start": "node index.js",  
 "test": "jest --runInBand"  
 },  
 "dependencies": {  
 "express": "^4.18.2",  
 "ioredis": "^5.3.2",  
 "pg": "^8.11.1",  
 "prom-client": "^14.0.1"  
 },  
 "devDependencies": {  
 "jest": "^29.6.1",  
 "supertest": "^6.4.4"  
 }  
}

# server/index.js

const express = require('express');  
const { Pool } = require('pg');  
const Redis = require('ioredis');  
const client = require('prom-client');  
  
const app = express();  
const collectDefaultMetrics = client.collectDefaultMetrics;  
collectDefaultMetrics();  
const requestCounter = new client.Counter({ name: 'http\_requests\_total', help: 'Total HTTP requests', labelNames: ['method','path','status'] });  
  
// Postgres pool (reads connection info from env vars)  
const pool = new Pool({  
 connectionString: process.env.DATABASE\_URL || 'postgresql://postgres:postgres@postgres:5432/postgres'  
});  
  
// Redis client  
const redis = new Redis({  
 host: process.env.REDIS\_HOST || 'redis',  
 port: process.env.REDIS\_PORT ? Number(process.env.REDIS\_PORT) : 6379  
});  
  
app.get('/', async (req, res) => {  
 // increment counter  
 res.setHeader('Content-Type', 'text/plain');  
 try {  
 requestCounter.inc({ method: req.method, path: '/', status: 200 }, 1);  
 // simple DB ping (non-blocking)  
 pool.query('SELECT 1').catch(() => {});  
 // Redis ping  
 redis.ping().catch(() => {});  
 } catch (e) {}  
 res.send('Hi Koronet Team.\n');  
});  
  
app.get('/health', async (req, res) => {  
 try {  
 // Check Postgres and Redis quickly  
 await pool.query('SELECT 1');  
 await redis.ping();  
 requestCounter.inc({ method: req.method, path: '/health', status: 200 }, 1);  
 res.json({ status: 'ok' });  
 } catch (e) {  
 requestCounter.inc({ method: req.method, path: '/health', status: 500 }, 1);  
 res.status(500).json({ status: 'error', error: e.message });  
 }  
});  
  
app.get('/metrics', async (req, res) => {  
 res.set('Content-Type', client.register.contentType);  
 res.end(await client.register.metrics());  
});  
  
const PORT = process.env.PORT || 3000;  
app.listen(PORT, () => {  
 console.log(`Server listening on ${PORT}`);  
});  
  
module.exports = app; // for tests

# server/**tests**/app.test.js

const request = require('supertest');  
const app = require('../index');  
  
describe('GET /', () => {  
 it('responds with Hi Koronet Team.', async () => {  
 const res = await request(app).get('/');  
 expect(res.statusCode).toBe(200);  
 expect(res.text).toBe('Hi Koronet Team.\n');  
 });  
});

# Dockerfile

# Builder stage  
FROM node:18-alpine AS builder  
WORKDIR /app  
COPY server/package.json server/package-lock.json\* ./  
RUN npm ci --silent  
COPY server/ ./  
RUN npm run build --if-present  
  
# Production stage (still small - use alpine)  
FROM node:18-alpine  
WORKDIR /app  
ENV NODE\_ENV=production  
COPY --from=builder /app/node\_modules ./node\_modules  
COPY --from=builder /app .  
EXPOSE 3000  
CMD ["node","index.js"]

Notes: using node:18-alpine keeps the image small; multi-stage ensures dev deps are not copied. For even smaller image, consider node:18-slim + distroless in future.

# docker-compose.yml

version: '3.8'  
services:  
 web:  
 build: .  
 ports:  
 - "3000:3000"  
 environment:  
 - PORT=3000  
 - DATABASE\_URL=postgresql://postgres:postgres@postgres:5432/postgres  
 - REDIS\_HOST=redis  
 depends\_on:  
 - redis  
 - postgres  
 networks:  
 - koronet-net  
  
 redis:  
 image: redis:7-alpine  
 ports:  
 - "6379:6379"  
 networks:  
 - koronet-net  
  
 postgres:  
 image: postgres:15-alpine  
 environment:  
 - POSTGRES\_PASSWORD=postgres  
 ports:  
 - "5432:5432"  
 networks:  
 - koronet-net  
  
networks:  
 koronet-net:  
 driver: bridge

# .github/workflows/ci.yml

name: CI  
on:  
 push:  
 branches: [ main ]  
 pull\_request:  
 branches: [ main ]  
  
jobs:  
 build-and-push:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
  
 - name: Set up Docker Buildx  
 uses: docker/setup-buildx-action@v3  
  
 - name: Cache Docker layers  
 uses: actions/cache@v4  
 with:  
 path: /tmp/.buildx-cache  
 key: ${{ runner.os }}-buildx-${{ github.sha }}  
 restore-keys: |  
 ${{ runner.os }}-buildx-  
  
 - name: Build image  
 run: |  
 docker build -t ${{ secrets.DOCKERHUB\_USERNAME }}/koronet-web:${{ github.sha }} .  
  
 - name: Run tests  
 run: |  
 cd server  
 npm ci  
 npm test  
  
 - name: Log in to DockerHub  
 uses: docker/login-action@v2  
 with:  
 username: ${{ secrets.DOCKERHUB\_USERNAME }}  
 password: ${{ secrets.DOCKERHUB\_TOKEN }}  
  
 - name: Push image to Docker Hub  
 run: |  
 docker tag ${{ secrets.DOCKERHUB\_USERNAME }}/koronet-web:${{ github.sha }} ${{ secrets.DOCKERHUB\_USERNAME }}/koronet-web:latest  
 docker push ${{ secrets.DOCKERHUB\_USERNAME }}/koronet-web:${{ github.sha }}  
 docker push ${{ secrets.DOCKERHUB\_USERNAME }}/koronet-web:latest  
  
 # Optionally: add a deploy job that calls Terraform or prints the commands needed for ECS deployment.

# monitoring/diagram.md

# Monitoring Diagram (Mermaid)  
  
```mermaid  
flowchart LR  
 subgraph Docker/ECS  
 web[Web App (Node)\nexposes /metrics]  
 redis[Redis]  
 end  
  
 prometheus[Prometheus Server]  
 grafana[Grafana]  
 node\_exporter[Metrics Sidecar or node\_exporter]  
  
 web -->|scrape /metrics| prometheus  
 node\_exporter -->|export host metrics| prometheus  
 redis -->|redis exporter| prometheus  
 prometheus -->|data| grafana  
  
 style web fill:#f9f,stroke:#333,stroke-width:1px  
 style prometheus fill:#fffbcc,stroke:#333  
 style grafana fill:#ccf,stroke:#333

**Notes**: - The Web app exposes /metrics via prom-client. - For Redis, run oliver006/redis\_exporter as a sidecar or separate task and have Prometheus scrape it. - In ECS, prefer running Prometheus externally (e.g., in monitoring account) or use a scraping/sidecar approach. - Grafana reads Prometheus as datasouce and shows dashboards for http\_requests\_total, node metrics, Redis metrics.

---  
  
# terraform/variables.tf  
  
```hcl  
variable "aws\_region" { default = "us-east-1" }  
variable "cluster\_name" { default = "koronet-cluster" }  
variable "vpc\_cidr" { default = "10.0.0.0/16" }  
variable "public\_subnets" { default = ["10.0.1.0/24","10.0.2.0/24"] }  
variable "private\_subnets" { default = ["10.0.11.0/24","10.0.12.0/24"] }  
variable "allowed\_cidr" { default = "0.0.0.0/0" }

# terraform/providers.tf

terraform {  
 required\_version = ">= 1.3.0"  
}  
provider "aws" {  
 region = var.aws\_region  
}

# terraform/vpc.tf

resource "aws\_vpc" "this" {  
 cidr\_block = var.vpc\_cidr  
 tags = { Name = "koronet-vpc" }  
}  
  
resource "aws\_subnet" "public" {  
 for\_each = toset(var.public\_subnets)  
 vpc\_id = aws\_vpc.this.id  
 cidr\_block = each.value  
 availability\_zone = data.aws\_availability\_zones.available.names[0]  
 map\_public\_ip\_on\_launch = true  
 tags = { Name = "koronet-public-${each.key}" }  
}  
  
resource "aws\_subnet" "private" {  
 for\_each = toset(var.private\_subnets)  
 vpc\_id = aws\_vpc.this.id  
 cidr\_block = each.value  
 availability\_zone = data.aws\_availability\_zones.available.names[0]  
 tags = { Name = "koronet-private-${each.key}" }  
}  
  
resource "aws\_internet\_gateway" "igw" {  
 vpc\_id = aws\_vpc.this.id  
}  
  
resource "aws\_route\_table" "public" {  
 vpc\_id = aws\_vpc.this.id  
 route {  
 cidr\_block = "0.0.0.0/0"  
 gateway\_id = aws\_internet\_gateway.igw.id  
 }  
}  
  
resource "aws\_route\_table\_association" "public\_assoc" {  
 for\_each = aws\_subnet.public  
 subnet\_id = each.value.id  
 route\_table\_id = aws\_route\_table.public.id  
}  
  
data "aws\_availability\_zones" "available" {}

# terraform/ecs.tf

resource "aws\_ecs\_cluster" "this" {  
 name = var.cluster\_name  
}  
  
# Task role and execution role  
resource "aws\_iam\_role" "ecs\_task\_execution\_role" {  
 name = "ecsTaskExecutionRole-koronet"  
 assume\_role\_policy = data.aws\_iam\_policy\_document.ecs\_task\_assume\_role.json  
}  
  
data "aws\_iam\_policy\_document" "ecs\_task\_assume\_role" {  
 statement { actions = ["sts:AssumeRole"]  
 principals { type = "Service"; identifiers = ["ecs-tasks.amazonaws.com"] }  
 }  
}  
  
# Attach managed policies for ECS tasks (pull images, write logs)  
resource "aws\_iam\_role\_policy\_attachment" "ecs\_task\_execution\_policy" {  
 role = aws\_iam\_role.ecs\_task\_execution\_role.name  
 policy\_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"  
}  
  
resource "aws\_security\_group" "ecs\_sg" {  
 name = "koronet-ecs-sg"  
 vpc\_id = aws\_vpc.this.id  
 description = "Allow HTTP outbound and ephemeral inbound"  
 ingress {  
 description = "HTTP"  
 from\_port = 80  
 to\_port = 80  
 protocol = "tcp"  
 cidr\_blocks = [var.allowed\_cidr]  
 }  
 egress { from\_port = 0; to\_port = 0; protocol = "-1"; cidr\_blocks = ["0.0.0.0/0"] }  
}  
  
# Example task definition (FARGATE)  
resource "aws\_ecs\_task\_definition" "web\_task" {  
 family = "koronet-web"  
 network\_mode = "awsvpc"  
 requires\_compatibilities = ["FARGATE"]  
 cpu = "256"  
 memory = "512"  
 execution\_role\_arn = aws\_iam\_role.ecs\_task\_execution\_role.arn  
 container\_definitions = jsonencode([  
 {  
 name = "web",  
 image = "${var.docker\_image}",  
 essential = true,  
 portMappings = [{ containerPort = 3000, hostPort = 3000, protocol = "tcp" }],  
 environment = [  
 { name = "DATABASE\_URL", value = "${var.database\_url}" }  
 ]  
 },  
 {  
 name = "prometheus-sidecar",  
 image = "prom/prometheus:latest",  
 essential = false,  
 portMappings = [{ containerPort = 9090, hostPort = 9090, protocol = "tcp" }]  
 }  
 ])  
}  
  
# Variables used by ecs.tf

# terraform/outputs.tf

output "ecs\_cluster\_arn" {  
 value = aws\_ecs\_cluster.this.arn  
}  
  
output "task\_def\_arn" {  
 value = aws\_ecs\_task\_definition.web\_task.arn  
}

# terraform/extra.tfvars (example)

aws\_region = "us-east-1"  
cluster\_name = "koronet-cluster"  
docker\_image = "dockerhubuser/koronet-web:latest"  
database\_url = "postgresql://postgres:postgres@postgres:5432/postgres"

# Notes and assumptions

* The Terraform here is intentionally opinionated but minimal; it’s a starting point that creates VPC, subnets, ECS cluster, IAM roles, SGs and a Fargate task definition which contains the main app container and a prometheus-sidecar container. In production you’d separate Prometheus and run it in a monitoring account or managed solution.
* The GitHub Actions pipeline pushes images to Docker Hub. The ECS task definition references the Docker Hub image (set via var.docker\_image) — either push there or push to ECR and update the task definition.
* Redis exporter can be added as a separate container to the task definition or as a sidecar in a different task.

# How to test locally

1. Copy .env.example values into .env (not included here; set defaults shown in docker-compose)
2. docker-compose up --build
3. curl http://localhost:3000/ should return Hi Koronet Team.
4. curl http://localhost:3000/metrics shows Prometheus format metrics

# End of repository scaffold