Ai generated testing environment setup document

a comprehensive solution for your apartment accounting application test environment, including an enhanced docker-compose.yml, test container Dockerfiles, Terraform configuration, and recommendations.

Enhanced docker-compose.yml with Test Services
yaml

```yaml
version: '3.8'

services:
  # Production Services
  postgres:
    image: postgres:15-alpine
    container_name: apartment_accounting_db
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres_password
      POSTGRES_DB: apartment_accounting
      PGDATA: /var/lib/postgresql/data/pgdata
      TZ: UTC
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./database/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5
    networks:
      - apartment_network

  pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: admin_password
    ports:
      - "5050:80"
    depends_on:
      postgres:
        condition: service_healthy
```

```
    networks:
      - apartment_network

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: accounting_backend
    environment:
      FLASK_ENV: development
      DATABASE_URL: postgresql://postgres:postgres_password@postgres:5432/
apartment_accounting
      DB_SCHEMA: accounting
    expose:
      - "5000"
    networks:
      - apartment_network
    volumes:
      - ./backend:/app
    depends_on:
      postgres:
        condition: service_healthy

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: accounting_frontend
    environment:
      - NODE_ENV=development
      - REACT_APP_API_URL=/
    expose:
      - "8080"
    volumes:
      - ./frontend:/app
      - /app/node_modules
    networks:
      - apartment_network

  frontend-nginx:
    build:
      context: ./
      dockerfile: Dockerfile
    container_name: accounting_nginx
    ports:
      - "8000:80"
```

```
    depends_on:
      - backend
      - frontend
    networks:
      - apartment_network

  # Test Services
  postgres-test:
    image: postgres:15-alpine
    container_name: apartment_accounting_test_db
    environment:
      POSTGRES_USER: test_user
      POSTGRES_PASSWORD: test_password
      POSTGRES_DB: apartment_accounting_test
      PGDATA: /var/lib/postgresql/data/pgdata
      TZ: UTC
    volumes:
      - postgres_test_data:/var/lib/postgresql/data
      - ./database/test_init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5433:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U test_user"]
      interval: 5s
      timeout: 5s
      retries: 5
    networks:
      - apartment_test_network

  backend-test:
    build:
      context: ./backend
      dockerfile: Dockerfile.test
    container_name: accounting_backend_test
    environment:
      FLASK_ENV: test
      DATABASE_URL: postgresql://test_user:test_password@postgres-test:543
2/apartment_accounting_test
      DB_SCHEMA: accounting_test
    networks:
      - apartment_test_network
    depends_on:
      postgres-test:
        condition: service_healthy
    command: ["pytest", "-v"]
```

```
  frontend-test:
    build:
      context: ./frontend
      dockerfile: Dockerfile.test
    container_name: accounting_frontend_test
    environment:
      - NODE_ENV=test
      - REACT_APP_API_URL=http://backend-test:5000
    networks:
      - apartment_test_network
    depends_on:
      - backend-test
    command: ["npm", "run", "test"]

  playwright:
    image: mcr.microsoft.com/playwright:v1.40.0-jammy
    container_name: playwright_test
    networks:
      - apartment_test_network
    volumes:
      - ./frontend/e2e:/e2e
    working_dir: /e2e
    depends_on:
      - frontend-test
    command: ["npx", "playwright", "test"]

networks:
  apartment_network:
    driver: bridge
  apartment_test_network:
    driver: bridge

volumes:
  postgres_data:
  postgres_test_data:
```

Dockerfile Examples for Test Containers

backend/Dockerfile.test
dockerfile

```
FROM python:3.9-alpine

WORKDIR /app

# Install dependencies
RUN apk add --no-cache postgresql-dev gcc python3-dev musl-dev
```

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
RUN pip install pytest pytest-cov pytest-mock requests pg8000

# Copy application code
COPY . .

# Install pgTAP for database testing
RUN apk add --no-cache postgresql-client && \
    wget https://api.pgxn.org/dist/pgtap/1.3.0/pgtap-1.3.0.zip && \
    unzip pgtap-1.3.0.zip && \
    cd pgtap-1.3.0 && \
    make && \
    make install

CMD ["pytest", "-v", "--cov=.", "--cov-report=xml"]
```

frontend/Dockerfile.test
dockerfile
```
FROM node:18-alpine

WORKDIR /app

# Install dependencies
COPY package.json package-lock.json ./
RUN npm install
RUN npm install -g playwright
RUN npx playwright install-deps
RUN npx playwright install chromium

# Copy application code
COPY . .

# Set environment variables
ENV CI=true

CMD ["npm", "run", "test"]
```

Terraform Configuration for AWS EC2 Deployment

main.tf
terraform
```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
```

```
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "apartment_accounting" {
  ami                    = "ami-0c55b159cbfafe1f0" # Ubuntu 22.04 LTS
  instance_type          = "t3.medium"
  key_name               = "your-key-pair"
  vpc_security_group_ids = [aws_security_group.apartment_accounting_sg.id]
  user_data              = <<-EOF
              #!/bin/bash
              apt-get update
              apt-get install -y docker.io docker-compose
              systemctl enable docker
              systemctl start docker
              usermod -aG docker ubuntu
              EOF

  tags = {
    Name = "ApartmentAccountingTestEnv"
  }

  root_block_device {
    volume_size = 30
    volume_type = "gp3"
  }
}

resource "aws_security_group" "apartment_accounting_sg" {
  name        = "apartment_accounting_sg"
  description = "Allow inbound traffic for apartment accounting app"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 8000
```

```
    to_port     = 8000
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 5050
    to_port     = 5050
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

output "instance_public_ip" {
  value = aws_instance.apartment_accounting.public_ip
}
```

Recommendations and Best Practices

1. **Test Data Management**:
    - Use separate initialization scripts for production and test databases
    - Implement database snapshots for consistent test data
    - Consider using factories or fixtures for generating test data
    - Use transactional tests that roll back after each test case
2. **Security Considerations**:
    - Never hardcode credentials in Dockerfiles or compose files
    - Use AWS Secrets Manager or environment variables for sensitive data
    - Implement network isolation between test and production containers
    - Regularly update container images to patch vulnerabilities
    - Limit exposed ports to only what's necessary
3. **Monitoring Test Environments**:
    - Implement health checks for all services
    - Use Prometheus and Grafana for monitoring container metrics
    - Set up logging with ELK stack or AWS CloudWatch

    - Monitor resource usage to detect memory leaks or performance issues
4. **CI/CD Integration**:
    - Use the test containers in your CI pipeline
    - Implement parallel test execution where possible
    - Store test results as artifacts
    - Set up notifications for test failures
5. **Additional Recommendations**:
    - Use Docker build cache efficiently to speed up builds
    - Implement proper cleanup of test resources
    - Consider using test containers only during CI/CD, not in production
    - Version control your test data schemas along with your code
    - Implement automated database migrations for test environments

This configuration provides a complete, isolated test environment that can be easily reproduced and integrates well with CI/CD pipelines while maintaining security and performance best practices.