AJLA Concierge API - Deployment Guide

Overview

This guide covers deploying the AJLA Concierge API to various cloud platforms. The application is a FastAPI backend with PostgreSQL database and WebSocket support.

Prerequisites

Before deploying, ensure you have:

- Git repository with latest code pushed
- PostgreSQL database (provided by platform or external service)
- Environment variables ready:
  - `DATABASE_URL` - PostgreSQL connection string
  - `JWT_SECRET_KEY` - Random secure string for token signing

Generate JWT Secret Key

```
python3 -c "import secrets; print(secrets.token_urlsafe(32))"
```

Option 4: AWS Deployment

AWS offers multiple deployment options. Here are two approaches:

Option 4A: AWS App Runner (Easiest)

App Runner is AWS's fully managed container service - similar to Railway/Render.

Step 1: Prerequisites

- AWS Account
- AWS CLI installed and configured
- Docker image pushed to ECR (or connect GitHub directly)

Step 2: Create RDS PostgreSQL Database

1. Go to **AWS Console → RDS**
2. Click **"Create database"**
3. Configure:
   - **Engine**: PostgreSQL 16
   - **Template**: Free tier
   - **DB instance identifier**: `ajla-db`
   - **Master username**: `ajla_user`
   - **Master password**: (save this!)
   - **Public access**: Yes (for initial setup)
   - **VPC security group**: Create new, allow inbound on port 5432
4. Click **"Create database"**
5. Wait for status to become "Available"
6. Copy the **Endpoint** (e.g., `ajla-db.xxxxx.us-east-1.rds.amazonaws.com`)

Your DATABASE_URL will be:

```
postgresql://ajla_user:YOUR_PASSWORD@ajla-db.xxxxx.us-east-1.rds.amazonaws.com:5432/postgres
```

Step 3: Create App Runner Service

1. Go to **AWS Console → App Runner**
2. Click **"Create service"**
3. **Source**:
   - Select **"Source code repository"**
   - Connect to GitHub and select your repo
   - Branch: `main`
4. **Build settings**:
   - **Runtime**: Python 3.9
   - **Build command**: `pip install -r requirements.txt`
   - **Start command**: `uvicorn main:app --host 0.0.0.0 --port 8080`
   - **Port**: 8080
5. **Service settings**:
   - **Service name**: `ajla-api`
   - **CPU**: 0.25 vCPU
   - **Memory**: 0.5 GB
6. **Environment variables**:

Variable	Value

Variable	Value
`DATABASE_URL`	`postgresql://ajla user:PASS@ajla-db.xxx.rds.amazonaws.com:5432/postgres`
`JWT_SECRET_KEY`	`your-generated-secret-key`
`JWT_ALGORITHM`	`HS256`
`DEBUG`	`false`
`PORT`	`8080`

7. Click **"Create & deploy"**
8. Your API will be at: `https://xxxxx.us-east-1.awsapprunner.com`

Option 4B: AWS EC2 + Docker (Full Control)

For more control and cost optimization on a VPS.

Step 1: Launch EC2 Instance

1. Go to **AWS Console → EC2 → Launch Instance**
2. Configure:
   - **Name**: `ajla-api-server`
   - **AMI**: Ubuntu 22.04 LTS
   - **Instance type**: `t3.micro` (free tier eligible)
   - **Key pair**: Create new or use existing
   - **Security group**: Allow ports 22 (SSH), 80 (HTTP), 443 (HTTPS), 8000 (API)
3. Click **"Launch instance"**

Step 2: Connect to EC2

```
ssh -i your-key.pem ubuntu@your-ec2-public-ip
```

Step 3: Install Docker

```
# Update system
sudo apt update && sudo apt upgrade -y

# Install Docker
curl -fsSL https://get.docker.com | sh
```

```
sudo usermod -aG docker ubuntu

# Install Docker Compose
sudo apt install docker-compose -y

# Logout and login again for group changes
exit
```

Step 4: Clone Repository

```
ssh -i your-key.pem ubuntu@your-ec2-public-ip

git clone https://github.com/thexahmed00/conceirgeKsa.git
cd conceirgeKsa
```

Step 5: Create Production Environment File

```
cat > .env.prod << EOF
DATABASE_URL=postgresql://ajla_user:YOUR_PASSWORD@your-rds-endpoint:5432/
JWT_SECRET_KEY=your-generated-secret-key
JWT_ALGORITHM=HS256
JWT_EXPIRATION_HOURS=24
DEBUG=false
LOG_LEVEL=INFO
PORT=8000
EOF
```

Step 6: Create Production Docker Compose

```
cat > docker-compose.prod.yml << 'EOF'
version: '3.8'

services:
  api:
    build: .
    ports:
      - "8000:8000"
    env_file:
      - .env.prod
    restart: unless-stopped
    healthcheck:
```

```
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
EOF
```

Step 7: Deploy

```
docker-compose -f docker-compose.prod.yml up -d --build
```

Step 8: Setup Nginx Reverse Proxy (Optional but Recommended)

```
sudo apt install nginx -y

sudo cat > /etc/nginx/sites-available/ajla << 'EOF'
server {
    listen 80;
    server_name your-domain.com;

    location / {
        proxy_pass http://localhost:8000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_read_timeout 86400;  # For WebSocket
    }
}
EOF

sudo ln -s /etc/nginx/sites-available/ajla /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

Step 9: Setup SSL with Let's Encrypt (Production)

```
sudo apt install certbot python3-certbot-nginx -y
sudo certbot --nginx -d your-domain.com
```

Option 4C: AWS ECS Fargate (Production-Grade)

For production workloads with auto-scaling.

Step 1: Push Docker Image to ECR

```
# Create ECR repository
aws ecr create-repository --repository-name ajla-api --region us-east-1

# Login to ECR
aws ecr get-login-password --region us-east-1 | docker login --username A

# Build and push
docker build -t ajla-api .
docker tag ajla-api:latest YOUR_ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.cc
docker push YOUR_ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/ajla-api:late
```

Step 2: Create ECS Cluster

1. Go to **AWS Console → ECS → Create cluster**
2. **Cluster name**: `ajla-cluster`
3. **Infrastructure**: AWS Fargate
4. Click **"Create"**

Step 3: Create Task Definition

1. Go to **Task definitions → Create new task definition**
2. Configure:
   - **Family**: `ajla-api-task`
   - **Launch type**: Fargate
   - **CPU**: 0.25 vCPU
   - **Memory**: 0.5 GB
3. **Container**:
   - **Name**: `ajla-api`
   - **Image URI**: `YOUR_ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/ajla-api:latest`
   - **Port mappings**: 8000
   - **Environment variables**: Add all required vars
4. Click **"Create"**

Step 4: Create Service

1. Go to your cluster → **Services** → **Create**
2. Configure:
   - **Launch type**: Fargate
   - **Task definition**: `ajla-api-task`
   - **Service name**: `ajla-api-service`
   - **Desired tasks**: 2
   - **Networking**: Select VPC, subnets, security group
3. **Load balancer**: Application Load Balancer
4. Click **"Create"**

Step 5: Configure ALB Health Check

- **Health check path**: `/health`
- **Healthy threshold**: 2
- **Interval**: 30 seconds

AWS Cost Estimates (Monthly)

Service	Free Tier	After Free Tier
App Runner	-	~$5-15/month
EC2 t3.micro	750 hrs/month (1 year)	~$8/month
RDS db.t3.micro	750 hrs/month (1 year)	~$15/month
ECS Fargate	-	~$10-30/month

AWS Security Best Practices

1. **Never commit credentials** - Use environment variables or AWS Secrets Manager
2. **Use IAM roles** - Don't use root account
3. **Enable RDS encryption** - Encrypt data at rest
4. **VPC security groups** - Restrict database access to only your app
5. **Enable CloudWatch** - Monitor logs and set up alerts
6. **Use Secrets Manager** for sensitive values:

```
# Store secret
aws secretsmanager create-secret --name ajla/jwt-secret --secret-string "
```

```
# Reference in ECS task definition
# Use "valueFrom": "arn:aws:secretsmanager:region:account:secret:ajla/jwt
```

Option 2: Docker (Self-Hosted / VPS)

For deployment on your own server (DigitalOcean, AWS EC2, etc.)

Step 1: Install Docker on Server

```
curl -fsSL https://get.docker.com | sh
```

Step 2: Create docker-compose.prod.yml

```
version: '3.8'

services:
  api:
    build: .
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://ajla_user:your_password@db:5432/ajla_db
      - JWT_SECRET_KEY=your-generated-secret-key
      - JWT_ALGORITHM=HS256
      - DEBUG=false
      - PORT=8000
    depends_on:
      - db
    restart: unless-stopped

  db:
    image: postgres:16-alpine
    environment:
      - POSTGRES_USER=ajla_user
      - POSTGRES_PASSWORD=your_password
      - POSTGRES_DB=ajla_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: unless-stopped
```

```
volumes:
  postgres_data:
```

Step 3: Deploy

```
docker-compose -f docker-compose.prod.yml up -d
```

Step 4: Setup Reverse Proxy (Nginx)

```
server {
    listen 80;
    server_name your-domain.com;

    location / {
        proxy_pass http://localhost:8000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Environment Variables Reference

Variable	Required	Default	Description
`DATABASE_URL`	✅ Yes	-	PostgreSQL connection string
`JWT_SECRET_KEY`	✅ Yes	-	Secret key for JWT signing
`JWT_ALGORITHM`	No	`HS256`	JWT signing algorithm
`JWT_EXPIRATION_HOURS`	No	`24`	Token expiration time
`DEBUG`	No	`true`	Enable debug mode
`LOG_LEVEL`	No	`INFO`	Logging level
`PORT`	No	`8000`	Server port (auto-set by Railway/Render)

Post-Deployment Checklist

- [ ] Health endpoint returns `{"status": "healthy"}`
- [ ] API docs accessible at `/docs`
- [ ] User registration works (`POST /api/v1/auth/register`)
- [ ] User login works (`POST /api/v1/auth/login`)
- [ ] WebSocket chat connects (`ws://your-domain/ws/chat/{id}?token=...`)
- [ ] Database migrations applied
- [ ] Admin user created (if needed)

Create Admin User

After deployment, create an admin user via SQL:

```
-- First register a user via API, then promote to admin:
UPDATE users SET is_admin = TRUE WHERE email = 'admin@example.com';
```

Troubleshooting

502 Bad Gateway

- Check if the app is binding to `$PORT` environment variable
- Verify health endpoint returns 200
- Check logs for startup errors

Database Connection Failed

- Verify `DATABASE_URL` format: `postgresql://user:pass@host:port/dbname`
- Check if database is accessible from the app
- For Railway: ensure PostgreSQL service is linked

JWT Errors

- Ensure `JWT_SECRET_KEY` is set
- Check token format in Authorization header: `Bearer <token>`

WebSocket Connection Failed

- WebSocket URLs use `wss://` for HTTPS domains
- Include token as query parameter: `?token=your-jwt-token`
- Check CORS settings if connecting from browser

API Endpoints Summary

Method	Endpoint	Description
GET	`/`	Welcome message
GET	`/health`	Health check
GET	`/docs`	Swagger UI
POST	`/api/v1/auth/register`	User registration
POST	`/api/v1/auth/login`	User login
GET	`/api/v1/auth/me`	Current user info
GET	`/api/v1/users/me`	User profile
PUT	`/api/v1/users/me`	Update profile
POST	`/api/v1/requests`	Create request
GET	`/api/v1/requests`	List requests
GET	`/api/v1/conversations`	List conversations
GET	`/api/v1/conversations/{id}`	Get conversation
POST	`/api/v1/conversations/{id}/messages`	Send message
WS	`/ws/chat/{conversation_id}`	WebSocket chat
GET	`/api/v1/admin/conversations`	Admin: List all
POST	`/api/v1/admin/conversations/{id}/messages`	Admin: Reply

Support

For issues, check:

1. Deploy logs in your platform's dashboard
2. Application logs for error messages
3. Database connectivity
4. Environment variables configuration