Pulse - Uptime Monitoring Service

A robust and professional web-based application that continuously monitors the availability and performance of websites, APIs, and services. Pulse provides real-time dashboards, intelligent alerting, and comprehensive analytics for your critical infrastructure.

Table of Contents

Features

Core Monitoring

- **Real-Time Health Checks**: Monitor multiple targets with customizable check intervals (minimum 10 seconds)
- **HTTP Status Tracking**: Verify expected HTTP status codes and detect anomalies
- **Latency Measurement**: Track response times and identify performance degradation
- **Multi-Protocol Support**: Monitor HTTP, HTTPS, and custom HTTP methods (GET, POST, HEAD)

Alerting & Notifications

- **Multi-Channel Alerts**: Email, Slack, and Discord notifications
- **Configurable Rules**: Set thresholds for consecutive failures, uptime percentages, and response times
- **Alert Management**: Acknowledge, resolve, and track alert history
- **Smart Escalation**: Automatic severity levels based on alert type

Analytics & Reporting

- **Uptime Tracking**: Calculate and display uptime percentages (daily, weekly, monthly)
- **Historical Data**: Comprehensive check history with timestamps and response metrics
- **Performance Insights**: Average response times, success rates, and trends
- **Audit Logs**: Complete audit trail of all user actions for compliance

User Management

- **Role-Based Access Control**: Admin and user roles with appropriate permissions
- **OAuth2 Integration**: Secure authentication with Manus OAuth
- **User Settings**: Customize notification preferences and alert channels

- **Multi-User Support**: Manage multiple users with isolated data

Technology Stack

Frontend

- **React 19** with TypeScript for type-safe UI development
- **Next.js** for server-side rendering and static generation
- **Tailwind CSS 4** for responsive design
- **shadcn/ui** for accessible, customizable components
- **tRPC** for end-to-end type-safe API calls
- **Wouter** for lightweight client-side routing

Backend

- **Express.js 4** for HTTP server and middleware
- **tRPC 11** for type-safe RPC procedures
- **Node.js 22** runtime environment
- **Drizzle ORM** for database operations with type safety

Database

- **MySQL 8.0** for relational data storage
- **Drizzle Kit** for schema management and migrations

DevOps & Monitoring

- **Docker** for containerization
- **Docker Compose** for local development orchestration
- **Kubernetes** for production-grade orchestration
- **Prometheus** for metrics collection
- **Grafana** for visualization and dashboards

- **GitHub Actions** for CI/CD automation

Quick Start

Prerequisites

- Node.js 22.x or higher
- pnpm 9.x or higher
- Docker and Docker Compose (for containerized deployment)
- MySQL 8.0 or compatible database

Installation

1. **Clone the repository** `bash git clone https://github.com/yourusername/pulse.git cd pulse`

2. **Install dependencies** `bash pnpm install`

3. **Set up environment variables** `bash cp .env.example .env.local # Edit .env.local with your configuration`

4. **Initialize the database** `bash pnpm db:push`

5. **Start the development server** `bash pnpm dev`

The application will be available at `http://localhost:3000`.

Configuration

Environment Variables

Essential environment variables for running Pulse:

Variable	Description	Required
`DATABASE_URL`	MySQL connection string	Yes
`NODE_ENV`	Environment (development/production)	Yes
`JWT_SECRET`	Secret for session signing	Yes
`VITE_APP_ID`	OAuth application ID	Yes
`OAUTH_SERVER_URL`	OAuth server URL	Yes
`VITE_OAUTH_PORTAL_URL`	OAuth portal URL	Yes
`OWNER_OPEN_ID`	Owner's OAuth ID	Yes
`OWNER_NAME`	Owner's display name	Yes
`BUILT_IN_FORGE_API_URL`	Manus API URL	Yes
`BUILT_IN_FORGE_API_KEY`	Manus API key	Yes
`VITE_APP_TITLE`	Application title	No
`VITE_APP_LOGO`	Application logo URL	No

Monitoring Configuration

Configure monitoring behavior in `server/monitoring.ts`:

- **Check Interval**: Default 60 seconds (customizable per target)
- **Timeout**: Default 10 seconds per request
- **Retry Logic**: Configurable consecutive failure threshold
- **Alert Channels**: Email, Slack, Discord

Running Locally

Development Mode

```
# Install dependencies
pnpm install

# Run migrations
pnpm db:push

# Start development server
pnpm dev
```

Access the application at `http://localhost:3000`.

Production Build

```
# Build the application
pnpm build

# Start production server
pnpm start
```

Docker Deployment

Local Development with Docker Compose

1. **Build and start services** `bash cd infrastructure/docker docker-compose up -d`
2. **Access services**
3. Application: http://localhost:3000
4. Prometheus: http://localhost:9090
5. Grafana: http://localhost:3001
6. **Stop services** `bash docker-compose down`

Production Docker Build

```
# Build the Docker image
docker build -f infrastructure/docker/Dockerfile -t pulse:latest .

# Run the container
docker run -d \
  --name pulse \
  -p 3000:3000 \
  -e DATABASE_URL=mysql://user:password@db:3306/pulse \
  -e NODE_ENV=production \
  pulse:latest
```

Kubernetes Deployment

Prerequisites

- Kubernetes cluster (1.24+)
- kubectl configured
- Docker image pushed to registry

Deploy to Kubernetes

1. **Create namespace and secrets** `bash kubectl create namespace pulse kubectl create secret generic pulse-secrets \ --from-literal=database-url=<your-db-url> \ --from-literal=jwt-secret=<your-jwt-secret> \ --from-literal=vite-app-id=<your-app-id> \ --from-literal=oauth-server-url=<your-oauth-url> \ --from-literal=vite-oauth-portal-url=<your-portal-url> \ --from-literal=owner-open-id=<your-owner-id> \ --from-literal=forge-api-url=<your-api-url> \ --from-literal=forge-api-key=<your-api-key> \ -n pulse`

2. **Create ConfigMap** `bash kubectl create configmap pulse-config \ --from-literal=owner-name="Your Name" \ --from-literal=app-title="Pulse" \ --from-literal=app-logo="https://example.com/logo.png" \ -n pulse`

3. **Apply deployment** `bash kubectl apply -f infrastructure/kubernetes/deployment.yaml -n pulse`

4. **Verify deployment** `bash kubectl get pods -n pulse kubectl get svc -n pulse`

Scaling

The deployment includes HorizontalPodAutoscaler (HPA) that automatically scales based on CPU and memory usage:

```bash
# View HPA status
kubectl get hpa -n pulse

# Manual scaling
kubectl scale deployment pulse-app --replicas=5 -n pulse
```

API Reference

Authentication

All API endpoints require authentication via OAuth2. The application handles authentication automatically.

Monitoring Targets

List Targets

```typescript
const { data: targets } = trpc.targets.list.useQuery();
```

Create Target

```typescript
const createMutation = trpc.targets.create.useMutation();
await createMutation.mutateAsync({
  name: "My Website",
  url: "example.com",
  protocol: "https",
  method: "GET",
  checkInterval: 60,
  timeout: 10,
  expectedStatusCode: 200,
});
```

Test Target

```
const testMutation = trpc.targets.testCheck.useMutation();
const result = await testMutation.mutateAsync({ id: 1 });
```

Alert Rules

Create Alert Rule

```
const createRule = trpc.alertRules.create.useMutation();
await createRule.mutateAsync({
  targetId: 1,
  name: "High Failure Rate",
  ruleType: "consecutive_failures",
  threshold: 3,
  notificationChannels: ["email", "slack"],
});
```

Alerts

Get Active Alerts

```
const { data: alerts } = trpc.alerts.active.useQuery();
```

Update Alert Status

```
const updateStatus = trpc.alerts.updateStatus.useMutation();
await updateStatus.mutateAsync({
  id: 1,
  status: "resolved",
});
```

Monitoring & Logging

Prometheus Metrics

Pulse exposes metrics at `/metrics` endpoint for Prometheus scraping:

- Request count and latency
- Database connection pool status
- Health check success/failure rates

- Alert trigger counts

Grafana Dashboards

Pre-configured dashboards available in `infrastructure/docker/grafana/provisioning/`:

- **System Overview**: CPU, memory, disk usage
- **Application Metrics**: Request rates, latencies, errors
- **Monitoring Health**: Check success rates, alert trends
- **Database Performance**: Query times, connection pool status

Logging

Logs are written to stdout and can be collected by container orchestration platforms:

```
# View logs in Docker
docker logs pulse-app

# View logs in Kubernetes
kubectl logs -f deployment/pulse-app -n pulse
```

Security

Best Practices Implemented

1. **Authentication & Authorization**
2. OAuth2 integration for secure authentication
3. Role-based access control (RBAC)
4. Session-based authentication with JWT
5. **Data Protection**
6. HTTPS/TLS for all communications
7. Encrypted database connections
8. Secure password hashing

9. **Container Security**
10. Non-root user execution
11. Read-only root filesystem
12. Security scanning in CI/CD pipeline
13. Minimal base images
14. **Infrastructure Security**
15. Network policies in Kubernetes
16. Secrets management
17. Regular security updates
18. Audit logging

Secrets Management

Store sensitive data in environment variables or Kubernetes secrets:

```
# Kubernetes secrets
kubectl create secret generic pulse-secrets \
  --from-literal=database-url=... \
  --from-literal=jwt-secret=...
```

Troubleshooting

Common Issues

Database Connection Failed

```
# Check database connectivity
mysql -h localhost -u pulse -p -e "SELECT 1"

# Verify DATABASE_URL format
echo $DATABASE_URL
```

Port Already in Use

```
# Find process using port 3000
lsof -i :3000

# Kill process
kill -9 <PID>
```

Docker Build Fails

```
# Clear Docker cache
docker system prune -a

# Rebuild with verbose output
docker build --progress=plain -f infrastructure/docker/Dockerfile .
```

Kubernetes Pod Crashes

```
# Check pod logs
kubectl logs <pod-name> -n pulse

# Describe pod for events
kubectl describe pod <pod-name> -n pulse

# Check resource limits
kubectl top pod -n pulse
```

CI/CD Pipeline

The GitHub Actions workflow automatically:

1. **Tests**: Run linting, type checking, and unit tests
2. **Security**: Scan for vulnerabilities with Trivy
3. **Build**: Create Docker image and push to registry
4. **Deploy**: Deploy to staging on develop branch, production on main branch

Required Secrets for CI/CD

Configure these secrets in GitHub repository settings:

- `STAGING_DEPLOY_KEY`: SSH private key for staging server
- `STAGING_DEPLOY_HOST`: Staging server hostname

- `STAGING_DEPLOY_USER`: SSH user for staging
- `PROD_DEPLOY_KEY`: SSH private key for production
- `PROD_DEPLOY_HOST`: Production server hostname
- `PROD_DEPLOY_USER`: SSH user for production
- `SLACK_WEBHOOK`: Slack webhook for notifications

Project Structure

```
pulse/
├── client/                  # React frontend
│   ├── src/
│   │   ├── pages/          # Page components
│   │   ├── components/     # Reusable components
│   │   ├── lib/            # Utilities and helpers
│   │   └── App.tsx         # Main app component
│   └── public/             # Static assets
├── server/                 # Express backend
│   ├── db.ts               # Database queries
│   ├── routers.ts          # tRPC procedures
│   ├── monitoring.ts       # Monitoring engine
│   ├── notifications.ts    # Alert notifications
│   └── _core/              # Framework plumbing
├── drizzle/                # Database schema
│   ├── schema.ts           # Table definitions
│   └── migrations/         # Migration files
├── infrastructure/         # DevOps configurations
│   ├── docker/             # Docker files
│   ├── kubernetes/         # K8s manifests
│   ├── terraform/          # IaC (optional)
│   └── github-actions/     # CI/CD workflows
├── docs/                   # Documentation
└── README.md               # This file
```

Performance Optimization

Frontend

- Code splitting with dynamic imports
- Image optimization with next/image
- CSS-in-JS with Tailwind for minimal bundle
- Lazy loading of routes

Backend

- Database connection pooling
- Query optimization with Drizzle ORM
- Caching strategies for frequently accessed data
- Rate limiting for API endpoints

Infrastructure

- Horizontal pod autoscaling
- Load balancing across replicas
- Database replication for high availability
- CDN integration for static assets

Contributing

1. Fork the repository
2. Create a feature branch (`git checkout -b feature/amazing-feature`)
3. Commit your changes (`git commit -m 'Add amazing feature'`)
4. Push to the branch (`git push origin feature/amazing-feature`)
5. Open a Pull Request

License

This project is licensed under the MIT License - see the LICENSE file for details.

Support

For issues, questions, or suggestions, please open an issue on GitHub or contact the development team.

Author: KNR Rishik
Version: 1.0.0
Last Updated: 2024