Enterprise Kitchen Management System

A production-grade, enterprise-ready kitchen management system built with Rust, featuring comprehensive restaurant operations, real-time order management, inventory tracking, and staff coordination with advanced security and observability patterns.

🚀 Features

Kitchen Management Core

- **Menu Management** - Digital menu creation, pricing, ingredient tracking, and seasonal updates
- **Order Processing** - Real-time order management from receipt to completion
- **Inventory Control** - Automated stock tracking, low-stock alerts, and supplier integration
- **Staff Coordination** - Role-based access, shift management, and task assignment
- **Table Management** - Reservation system, table status tracking, and seating optimization

Authentication & Security

- **JWT Authentication** - Secure token-based authentication with refresh tokens
- **User Management** - Registration, login, profile management with role-based access
- **PostgreSQL Integration** - Type-safe database operations with connection pooling
- **Password Security** - Argon2 hashing with configurable parameters

Real-time Operations

- **WebSocket Integration** - Live order updates and kitchen display systems
- **gRPC Services** - High-performance inter-service communication
- **Mobile API** - Native mobile app support for staff and management
- **Kitchen Display** - Real-time order status and preparation tracking

Enterprise Features

- **Distributed Tracing** - OpenTelemetry integration with Jaeger/Zipkin
- **Metrics & Monitoring** - Prometheus metrics with Grafana dashboards
- **Health Checks** - Kubernetes-ready liveness and readiness probes
- **Rate Limiting** - Token bucket algorithm with Redis backend
- **Audit Logging** - Comprehensive security event logging
- **Configuration Management** - Environment-aware configuration with validation

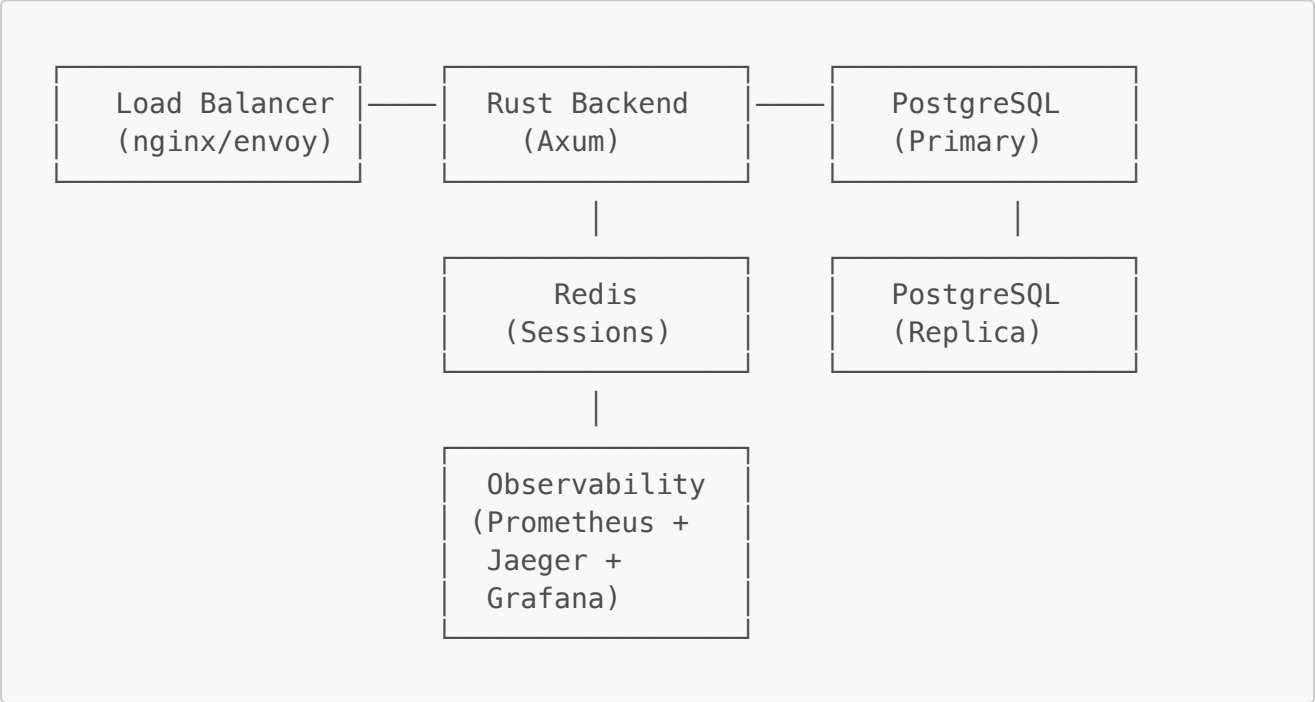
Security & Compliance

- **OWASP Compliance** - Protection against top 10 web vulnerabilities
- **PII Encryption** - Field-level encryption for sensitive data
- **Account Lockout** - Brute force protection with configurable policies
- **Session Management** - Secure session handling with automatic cleanup
- **CORS Policy** - Configurable cross-origin resource sharing
- **Input Validation** - Comprehensive request validation and sanitization

Operational Excellence

- **Docker Support** - Multi-stage builds with distroless images
- **Kubernetes Ready** - Helm charts and deployment manifests
- **CI/CD Pipeline** - GitHub Actions with security scanning
- **Load Testing** - K6 performance test suite
- **Database Migrations** - Version-controlled schema management
- **Backup Strategy** - Automated PostgreSQL backup procedures

🏗️ Architecture



🛠️ Tech Stack

Core Technologies

- **Runtime**: Tokio (async runtime)
- **Web Framework**: Axum (type-safe, performant)
- **Database**: PostgreSQL 15+ with SQLx
- **Authentication**: JWT with RS256 signing
- **Caching**: Redis for sessions and rate limiting

Observability

- **Logging**: tracing + tracing-subscriber

- **Metrics**: Prometheus with custom business metrics
- **Tracing**: OpenTelemetry with Jaeger backend
- **Health Checks**: Custom health check framework

Security

- **Password Hashing**: Argon2id (OWASP recommended)
- **Encryption**: AES-256-GCM for PII data
- **Rate Limiting**: Token bucket with Redis
- **Input Validation**: Custom validation framework

📋 Prerequisites

- **Rust**: 1.75+ (MSRV policy: latest stable - 2 versions)
- **PostgreSQL**: 15+
- **Redis**: 7+
- **Docker**: 24+ (for development)
- **Kubernetes**: 1.28+ (for production deployment)

🚀 Quick Start

Development Environment

```
# Clone the repository
git clone https://github.com/company/rust-jwt-backend.git
cd rust-jwt-backend

# Start dependencies
docker-compose up -d postgres redis

# Install dependencies and run migrations
cargo install sqlx-cli
sqlx migrate run

# Copy environment configuration
cp .env.example .env.local

# Run the application
cargo run
```

Production Deployment

```
# Build optimized container
docker build -t rust-jwt-backend:latest .

# Deploy to Kubernetes
helm upgrade --install jwt-backend ./helm/jwt-backend \
```

```
  --namespace production \
  --values ./helm/jwt-backend/values.prod.yaml
```

📁 Project Structure

```
.
├── src/
│   ├── api/                    # HTTP handlers and routing
│   │   ├── auth/               # Authentication endpoints
│   │   ├── users/              # User management endpoints
│   │   └── health/             # Health check endpoints
│   ├── core/                   # Business logic layer
│   │   ├── auth/               # Authentication services
│   │   ├── users/              # User services
│   │   └── security/           # Security utilities
│   ├── infrastructure/         # External integrations
│   │   ├── database/           # Database repositories
│   │   ├── cache/              # Redis integration
│   │   └── observability/      # Metrics and tracing
│   ├── config/                 # Configuration management
│   ├── middleware/             # HTTP middleware
│   └── main.rs                 # Application entry point
├── migrations/                 # Database migrations
├── tests/                      # Integration tests
│   ├── api/                    # API endpoint tests
│   ├── performance/            # Load testing scripts
│   └── security/               # Security testing
├── helm/                       # Kubernetes deployment
├── .github/                    # CI/CD workflows
├── docker/                     # Docker configurations
└── docs/                       # Documentation
    ├── api/                    # OpenAPI specifications
    ├── deployment/             # Deployment guides
    └── security/               # Security documentation
```

🔧 Configuration

Environment Variables

Variable	Description	Default	Required
`APP_SERVER__HOST`	Server bind address	`0.0.0.0`	No
`APP_SERVER__PORT`	Server port	`3000`	No
`APP_DATABASE__URL`	PostgreSQL connection string	-	Yes
`APP_AUTH__JWT_SECRET`	JWT signing secret (min 32 chars)	-	Yes

Variable	Description	Default	Required
`APP_REDIS__URL`	Redis connection string	-	Yes
`APP_LOGGING__LEVEL`	Log level (trace, debug, info, warn, error)	`info`	No

Configuration Files

```yaml
# config/production.yaml
server:
  host: "0.0.0.0"
  port: 3000
  request_timeout_secs: 30
  max_request_size: 2097152

database:
  max_connections: 50
  min_connections: 5
  acquire_timeout_secs: 10

auth:
  jwt_expiration_hours: 24
  password_hash_cost: 12
  max_login_attempts: 5
  lockout_duration_minutes: 15

observability:
  tracing:
    jaeger_endpoint: "http://jaeger:14268/api/traces"
  metrics:
    prometheus_endpoint: "0.0.0.0:9090"
```

📊 API Documentation

Authentication Endpoints

Register User

```http
POST /api/v1/auth/register
Content-Type: application/json

{
  "email": "user@example.com",
  "password": "SecurePassword123!",
  "full_name": "John Doe"
}
```

Login

```
POST /api/v1/auth/login
Content-Type: application/json

{
  "email": "user@example.com",
  "password": "SecurePassword123!"
}
```

Refresh Token

```
POST /api/v1/auth/refresh
Authorization: Bearer <refresh_token>
```

User Management

Get Current User

```
GET /api/v1/users/me
Authorization: Bearer <access_token>
```

Update Profile

```
PUT /api/v1/users/me
Authorization: Bearer <access_token>
Content-Type: application/json

{
  "full_name": "Jane Doe",
  "preferences": {
    "theme": "dark",
    "notifications": true
  }
}
```

Health Checks

Liveness Probe

```
GET /health/live
```

Readiness Probe

```
GET /health/ready
```

Detailed Health

```
GET /health
Authorization: Bearer <admin_token>
```

🧪 Testing

Unit Tests

```
cargo test --lib
```

Integration Tests

```
cargo test --test integration
```

Load Testing

```
# Install k6
brew install k6  # macOS
# or
sudo apt install k6  # Ubuntu

# Run performance tests
k6 run tests/performance/load_test.js
```

Security Testing

```
# Run security audit
cargo audit

# Check for vulnerabilities
cargo deny check
```

```
# Static analysis
cargo clippy -- -D warnings
```

📈 Monitoring & Observability

Metrics

The application exposes the following Prometheus metrics:

- `http_requests_total` - Total HTTP requests by method and status
- `http_request_duration_seconds` - HTTP request duration histogram
- `auth_attempts_total` - Authentication attempts by outcome
- `database_connections_active` - Active database connections
- `jwt_tokens_issued_total` - Total JWT tokens issued
- `rate_limit_exceeded_total` - Rate limit violations

Tracing

Distributed tracing is implemented using OpenTelemetry:

- Request correlation IDs for end-to-end tracing
- Database query tracing with performance metrics
- External service call instrumentation
- Custom business logic spans

Dashboards

Grafana dashboards are provided for:

- Application performance metrics
- Database performance and health
- Authentication and security events
- Infrastructure metrics
- Business KPIs

🔒 Security

Security Measures

- **Password Policy**: Enforced complexity requirements
- **Rate Limiting**: Per-IP and per-user rate limits
- **Account Lockout**: Temporary lockout after failed attempts
- **JWT Security**: Short-lived access tokens + refresh tokens
- **Input Validation**: Comprehensive request validation
- **SQL Injection Protection**: Parameterized queries only
- **XSS Protection**: Content Security Policy headers
- **CSRF Protection**: Double-submit cookie pattern

Compliance

- **GDPR**: Personal data encryption and deletion capabilities
- **SOX**: Comprehensive audit logging
- **PCI DSS**: Secure handling of sensitive data
- **OWASP**: Protection against top 10 vulnerabilities

Security Headers

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
Content-Security-Policy: default-src 'self'
```

🚀 Deployment

Docker

```
# Multi-stage build for minimal production image
FROM rust:1.75-alpine AS builder
# ... build steps ...

FROM gcr.io/distroless/cc
COPY --from=builder /app/target/release/rust-jwt-backend /
EXPOSE 3000
ENTRYPOINT ["/rust-jwt-backend"]
```

Kubernetes

```
# Deploy using Helm
helm upgrade --install jwt-backend ./helm/jwt-backend \
  --namespace production \
  --set image.tag=v1.2.3 \
  --set replicaCount=3 \
  --set resources.requests.memory=256Mi \
  --set resources.limits.memory=512Mi
```

Production Checklist

- [ ] TLS certificates configured
- [ ] Database backups scheduled
- [ ] Monitoring alerts configured
- [ ] Log aggregation setup

- [ ] Secrets management implemented
- [ ] Network policies applied
- [ ] Resource limits set
- [ ] Auto-scaling configured

🤝 Contributing

Development Workflow

1. **Fork** the repository
2. **Create** a feature branch (`git checkout -b feature/amazing-feature`)
3. **Implement** your changes with tests
4. **Run** the test suite (`make test`)
5. **Commit** your changes (`git commit -m 'Add amazing feature'`)
6. **Push** to the branch (`git push origin feature/amazing-feature`)
7. **Open** a Pull Request

Code Standards

- **Formatting**: `cargo fmt` (enforced in CI)
- **Linting**: `cargo clippy` (no warnings allowed)
- **Testing**: Minimum 80% code coverage
- **Documentation**: All public APIs must be documented
- **Commits**: Conventional commit format

Review Process

- All PRs require 2 approvals
- Automated security scanning must pass
- Performance benchmarks must not regress
- Integration tests must pass in staging environment

📄 License

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

🆘 Support

Getting Help

- **Documentation**: docs.company.com/rust-jwt-backend
- **Issues**: GitHub Issues
- **Discussions**: GitHub Discussions
- **Slack**: #rust-backend channel

Reporting Security Issues

Please report security vulnerabilities to security@company.com. Do not use public issue trackers for security-related problems.

🗺️ Development Roadmap - Kitchen Management Enhancement

📋 24 Prioritized Enhancement Tasks

This roadmap transforms our JWT authentication backend into a comprehensive kitchen management system. Each task is designed to build upon existing infrastructure while adding restaurant-specific functionality.

🧪 Testing (3 tasks)

T1: Add unit tests for core business logic (user authentication, menu management)

- **Priority**: High
- **Estimated effort**: 3-5 days
- **Dependencies**: None
- **Description**: Implement comprehensive unit tests for authentication services, menu CRUD operations, and user management functions
- **Acceptance criteria**:
  - 90%+ code coverage for core modules
  - Tests for authentication flows, menu validation, and user role management
  - CI integration with automated test execution

T2: Implement integration tests for API endpoints

- **Priority**: High
- **Estimated effort**: 4-6 days
- **Dependencies**: T1
- **Description**: Create end-to-end integration tests for all REST API endpoints including authentication, menu management, and user operations
- **Acceptance criteria**:
  - Full API endpoint coverage
  - Database integration testing
  - Mock external service dependencies

T3: Set up end-to-end test scenarios for critical user flows

- **Priority**: Medium
- **Estimated effort**: 5-7 days
- **Dependencies**: T1, T2
- **Description**: Implement E2E tests for complete user journeys: staff login → order creation → kitchen workflow → completion
- **Acceptance criteria**:
  - Automated browser testing for frontend
  - Complete workflow validation
  - Performance benchmarking integration

📚 Documentation (3 tasks)

D1: Add comprehensive inline documentation for all public APIs

- **Priority**: High
- **Estimated effort**: 2-3 days
- **Dependencies**: None
- **Description**: Document all public functions, structs, and API endpoints with comprehensive rustdoc comments
- **Acceptance criteria**:
  - All public APIs documented with examples
  - OpenAPI/Swagger documentation generation
  - Code examples for common use cases

D2: Create Architecture Decision Records (ADRs) for key technical decisions

- **Priority**: Medium
- **Estimated effort**: 3-4 days
- **Dependencies**: None
- **Description**: Document architectural decisions, technology choices, and design patterns used in the kitchen management system
- **Acceptance criteria**:
  - ADRs for database design, authentication strategy, and microservices architecture
  - Decision rationale and alternatives considered
  - Template for future ADRs

D3: Develop API usage examples and update README with setup instructions

- **Priority**: Medium
- **Estimated effort**: 2-3 days
- **Dependencies**: D1
- **Description**: Create comprehensive examples for API usage, including kitchen-specific workflows and integration patterns
- **Acceptance criteria**:
  - Complete setup guide for development environment
  - API usage examples with curl and code samples
  - Deployment documentation for production

⚡ **Performance (3 tasks)**

P1: Implement connection pooling for gRPC services

- **Priority**: High
- **Estimated effort**: 3-4 days
- **Dependencies**: None
- **Description**: Optimize gRPC communication with connection pooling for better performance under load
- **Acceptance criteria**:
  - Configurable connection pool size
  - Connection health monitoring

    - Performance benchmarks showing improvement

P2: Integrate Redis for caching frequently accessed data

- **Priority**: High
- **Estimated effort**: 4-5 days
- **Dependencies**: None
- **Description**: Implement Redis caching for menu items, user sessions, and frequently accessed restaurant data
- **Acceptance criteria**:
    - Cache invalidation strategies
    - TTL configuration for different data types
    - Cache hit ratio monitoring

P3: Add request batching for bulk operations

- **Priority**: Medium
- **Estimated effort**: 3-4 days
- **Dependencies**: P2
- **Description**: Implement batching for bulk menu updates, order processing, and inventory operations
- **Acceptance criteria**:
    - Batch processing for menu updates
    - Bulk order status updates
    - Performance improvements for large datasets

🔒 **Security (3 tasks)**

S1: Implement request/response validation middleware

- **Priority**: High
- **Estimated effort**: 3-4 days
- **Dependencies**: None
- **Description**: Add comprehensive input validation and sanitization for all API endpoints
- **Acceptance criteria**:
    - Schema validation for all request payloads
    - Input sanitization to prevent injection attacks
    - Detailed validation error responses

S2: Add rate limiting per endpoint with Redis

- **Priority**: High
- **Estimated effort**: 2-3 days
- **Dependencies**: P2
- **Description**: Implement granular rate limiting per endpoint and user role using Redis
- **Acceptance criteria**:
    - Configurable rate limits per endpoint
    - Different limits for different user roles

- Rate limit monitoring and alerting

S3: Configure security headers and CORS policies

- **Priority**: Medium
- **Estimated effort**: 1-2 days
- **Dependencies**: None
- **Description**: Implement comprehensive security headers and CORS policies for web security
- **Acceptance criteria**:
  - Security headers (CSP, HSTS, etc.)
  - Configurable CORS policies
  - Security header testing and validation

🍽️ **Core Features (5 tasks)**

CF1: Develop menu management system (CRUD operations)

- **Priority**: Critical
- **Estimated effort**: 8-10 days
- **Dependencies**: S1
- **Description**: Build comprehensive menu management with items, categories, pricing, ingredients, and nutritional information
- **Acceptance criteria**:
  - Full CRUD operations for menu items
  - Category and subcategory management
  - Ingredient tracking and allergen information
  - Pricing and availability management

CF2: Build inventory tracking system

- **Priority**: Critical
- **Estimated effort**: 10-12 days
- **Dependencies**: CF1
- **Description**: Implement real-time inventory tracking with automatic reorder points and supplier integration
- **Acceptance criteria**:
  - Real-time stock level tracking
  - Low-stock alerts and automatic reordering
  - Supplier management and purchase orders
  - Inventory reports and analytics

CF3: Implement order management workflow

- **Priority**: Critical
- **Estimated effort**: 12-15 days
- **Dependencies**: CF1, CF2
- **Description**: Create complete order lifecycle from creation to completion with status tracking
- **Acceptance criteria**:

    - Order creation and modification
    - Kitchen workflow integration
    - Status tracking and updates
    - Order history and reporting

CF4: Create staff management with role-based access

- **Priority**: High
- **Estimated effort**: 6-8 days
- **Dependencies**: S1
- **Description**: Implement comprehensive staff management with roles, permissions, and shift scheduling
- **Acceptance criteria**:
    - Role-based access control (Chef, Server, Manager, etc.)
    - Shift scheduling and management
    - Staff performance tracking
    - Permission management system

CF5: Design table/reservation system

- **Priority**: High
- **Estimated effort**: 8-10 days
- **Dependencies**: CF4
- **Description**: Build table management and reservation system with real-time availability
- **Acceptance criteria**:
    - Table layout and capacity management
    - Reservation booking and management
    - Real-time table status updates
    - Waitlist and notification system

🔧 **Technical Enhancements (4 tasks)**

TE1: Add WebSockets for real-time order updates

- **Priority**: High
- **Estimated effort**: 5-7 days
- **Dependencies**: CF3
- **Description**: Implement WebSocket connections for real-time order status updates across kitchen and front-of-house
- **Acceptance criteria**:
    - Real-time order status broadcasting
    - Kitchen display system integration
    - Connection management and reconnection logic
    - Scalable WebSocket architecture

TE2: Develop mobile app API endpoints

- **Priority**: High

- **Estimated effort**: 6-8 days
- **Dependencies**: CF1, CF3, CF4
- **Description**: Create mobile-optimized API endpoints for staff applications and management tools
- **Acceptance criteria**:
  - Mobile-optimized response formats
  - Offline capability support
  - Push notification integration
  - Mobile authentication flows

TE3: Build kitchen display system interface

- **Priority**: High
- **Estimated effort**: 7-9 days
- **Dependencies**: CF3, TE1
- **Description**: Create dedicated interface for kitchen display systems showing orders, timing, and preparation status
- **Acceptance criteria**:
  - Real-time order display
  - Preparation time tracking
  - Kitchen workflow optimization
  - Multi-screen support

TE4: Implement reporting and analytics dashboard

- **Priority**: Medium
- **Estimated effort**: 8-10 days
- **Dependencies**: CF1, CF2, CF3
- **Description**: Build comprehensive analytics dashboard with sales, inventory, and performance metrics
- **Acceptance criteria**:
  - Sales reporting and analytics
  - Inventory turnover analysis
  - Staff performance metrics
  - Customizable dashboard views

🚀 Operational Improvements (3 tasks)

OI1: Set up CI/CD pipeline with GitHub Actions

- **Priority**: High
- **Estimated effort**: 3-4 days
- **Dependencies**: T1, T2
- **Description**: Implement comprehensive CI/CD pipeline with automated testing, security scanning, and deployment
- **Acceptance criteria**:
  - Automated testing on pull requests
  - Security vulnerability scanning
  - Automated deployment to staging/production

- Rollback capabilities

OI2: Implement feature flags for gradual rollouts

- **Priority**: Medium
- **Estimated effort**: 4-5 days
- **Dependencies**: OI1
- **Description**: Add feature flag system for safe deployment of new features and A/B testing
- **Acceptance criteria**:
  - Runtime feature toggle system
  - User-based feature rollouts
  - A/B testing capabilities
  - Feature flag management interface

OI3: Configure monitoring and alerting with Prometheus/Grafana

- **Priority**: High
- **Estimated effort**: 5-6 days
- **Dependencies**: None
- **Description**: Set up comprehensive monitoring, alerting, and observability for production systems
- **Acceptance criteria**:
  - Application metrics collection
  - Business metrics dashboards
  - Alerting for critical issues
  - Log aggregation and analysis

🎯 Implementation Strategy

Phase 1: Foundation (Weeks 1-4)

- **Focus**: Security, Testing, and Core Infrastructure
- **Tasks**: T1, T2, S1, S2, D1, P1, P2
- **Goal**: Establish robust foundation with comprehensive testing and security

Phase 2: Core Kitchen Features (Weeks 5-10)

- **Focus**: Essential kitchen management functionality
- **Tasks**: CF1, CF2, CF3, CF4, TE1
- **Goal**: Deliver core restaurant operations capabilities

Phase 3: Advanced Features (Weeks 11-14)

- **Focus**: Advanced functionality and user experience
- **Tasks**: CF5, TE2, TE3, T3
- **Goal**: Complete feature set with mobile and real-time capabilities

Phase 4: Production Ready (Weeks 15-16)

- **Focus**: Operations, monitoring, and documentation
- **Tasks**: OI1, OI2, OI3, D2, D3, TE4, P3, S3
- **Goal**: Production-ready system with full observability

📊 Success Metrics

- **Performance**: <200ms API response times, 99.9% uptime
- **Scalability**: Support for 1000+ concurrent orders
- **Security**: Zero critical vulnerabilities, complete audit trails
- **User Experience**: ❤️ second page load times, intuitive workflows
- **Business Impact**: 30% reduction in order processing time, 25% improvement in inventory accuracy

---

Built with ❤️ by the Platform Engineering Team

👨‍💻 API Usage Examples (with curl)

> All endpoints assume the server is running locally on http://localhost:3000 and the database is configured via .env.

Health Checks

```
curl -i http://localhost:3000/health/live
curl -i http://localhost:3000/health/ready
```

Authentication

Register

curl -i -X POST http://localhost:3000/api/v1/auth/register
-H 'Content-Type: application/json'
-d '{"email":"user@example.com","password":"SecurePassword123!","full_name":"John Doe"}'

Login

curl -i -X POST http://localhost:3000/api/v1/auth/login
-H 'Content-Type: application/json'
-d '{"email":"user@example.com","password":"SecurePassword123!"}'

Refresh Token

curl -i -X POST http://localhost:3000/api/v1/auth/refresh
-H 'Authorization: Bearer <refresh_token>'

User CRUD

Create User

curl -i -X POST http://localhost:3000/api/v1/users
-H 'Content-Type: application/json'
-d '{"id":"","email":"user@example.com","password_hash":"","full_name":"John Doe","preferences":null,"created_at":"2024-01-01T00:00:00Z","updated_at":"2024-01-01T00:00:00Z"}'

Get User (requires JWT)

curl -i http://localhost:3000/api/v1/users/
-H 'Authorization: Bearer <access_token>'

Update User (full_name)

curl -i -X PUT http://localhost:3000/api/v1/users/
-H 'Authorization: Bearer <access_token>'
-H 'Content-Type: application/json'
-d '"Jane Doe"'

Delete User

curl -i -X DELETE http://localhost:3000/api/v1/users/
-H 'Authorization: Bearer <access_token>'

Refresh Token CRUD

Create Refresh Token

curl -i -X POST http://localhost:3000/api/v1/refresh_tokens
-H 'Content-Type: application/json'
-d '{"id":"","user_id":"","token":"","expires_at":"2024-01-01T00:00:00Z","created_at":"2024-01-01T00:00:00Z"}'

Get Refresh Token

curl -i http://localhost:3000/api/v1/refresh_tokens/

Update Refresh Token (token string)

curl -i -X PUT http://localhost:3000/api/v1/refresh_tokens/
-H 'Content-Type: application/json'
-d '"new_token_string"'

Delete Refresh Token

curl -i -X DELETE http://localhost:3000/api/v1/refresh_tokens/