Contents

Restaurant Management System - Complete Runbook

Version: 1.0 **Last Updated**: January 6, 2026 **Author**: System Documentation

---

Table of Contents

---

1. System Overview

What is this system?

The Restaurant Management System is a **multi-tenant SaaS platform** that helps restaurants manage their operations including:

- **Table Management**: Track table availability, reservations, and status
- **Menu Management**: Create and manage digital menus with categories and items
- **Order Processing**: Handle customer orders from dine-in and online channels
- **User Management**: Manage staff (chefs, waiters) and customer accounts
- **QR Code Ordering**: Enable contactless ordering via QR codes
- **Real-time Updates**: Live order status updates and kitchen notifications
- **Analytics**: Sales reports, popular items, revenue tracking

Multi-Tenant Architecture

What is Multi-Tenancy?

Instead of each restaurant having its own separate deployment, all restaurants share the same application infrastructure but their data is completely isolated.

How it works:

- Each restaurant gets a unique `restaurant_id`
- All database queries filter by `restaurant_id`
- Users belong to specific restaurants (except master admin)
- One codebase serves multiple restaurants

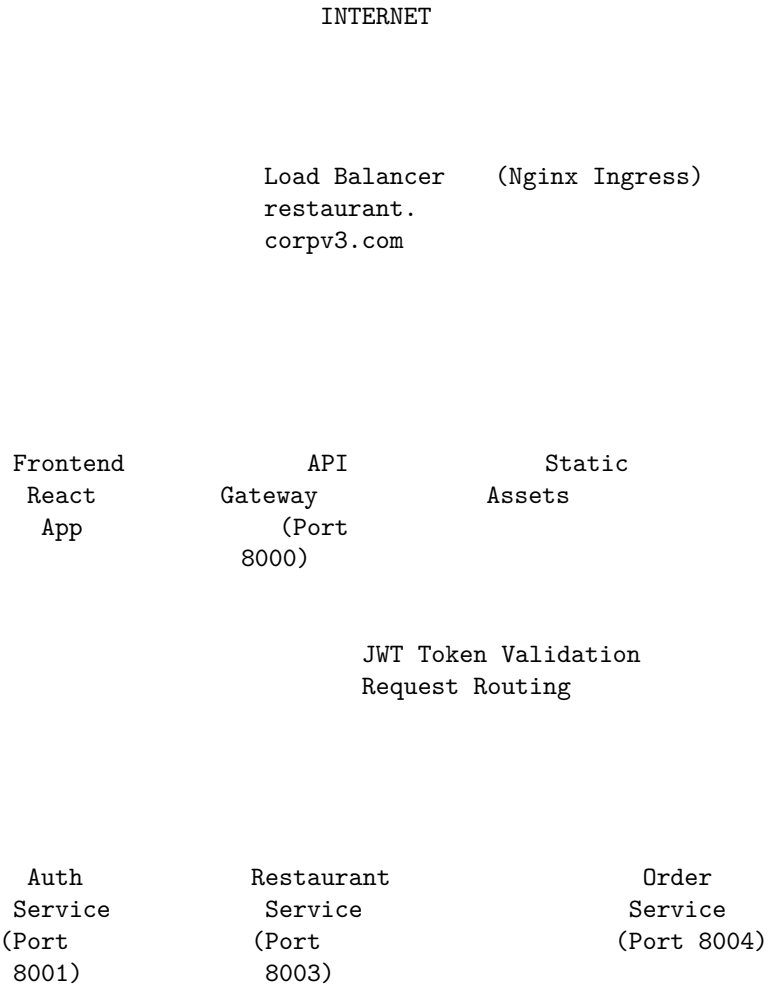
Benefits:

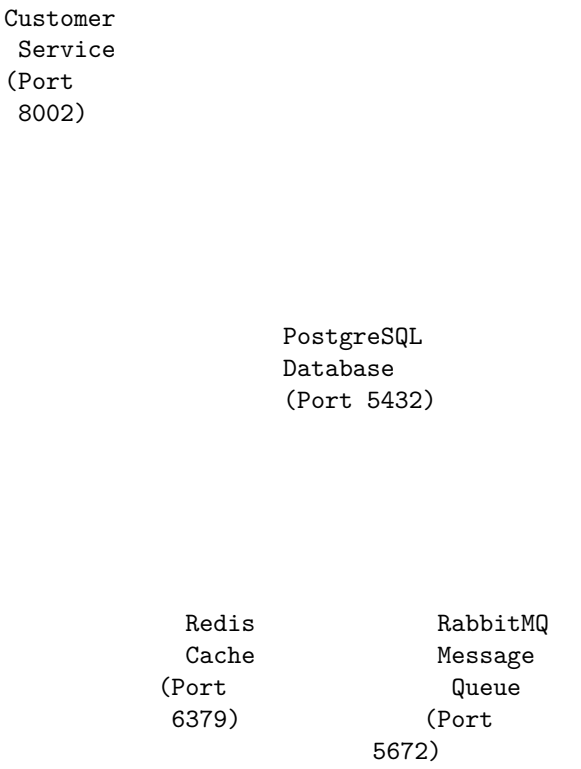
- Lower infrastructure costs (shared resources)
- Easier updates (deploy once, affects all tenants)
- Centralized monitoring and maintenance
- Faster onboarding for new restaurants

---

2. Architecture Diagram

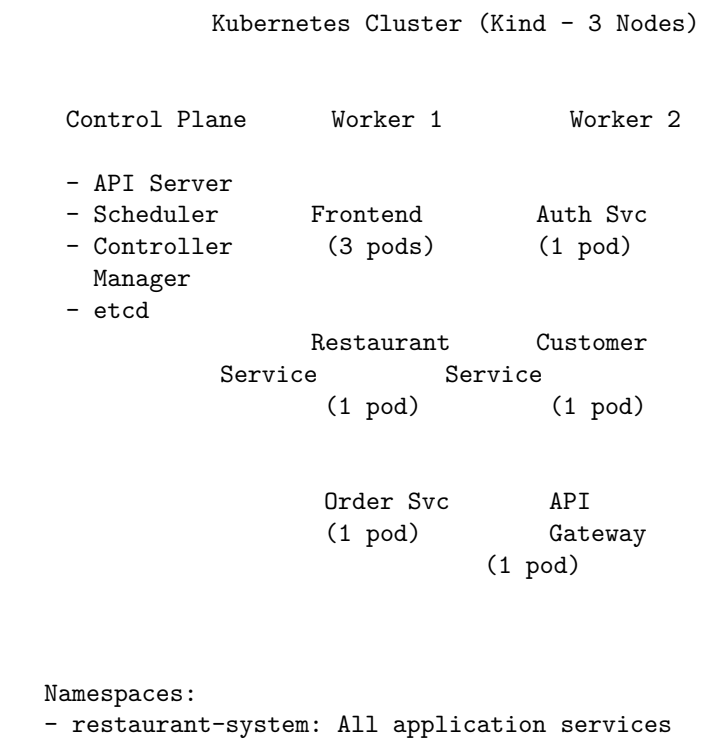
High-Level Architecture





Service Mesh Architecture (Istio)

Kubernetes Deployment Architecture



---

3. Technology Stack

Frontend Technologies

Technology	Version	Purpose	Why We Use It
React	18.x	UI Framework	Component-based architecture, virtual DOM for performance, huge ecosystem
TypeScript	5.x	Type Safety	Catch errors at compile time, better IDE support, self-documenting code
Vite	5.x	Build Tool	Fast HMR (Hot Module Replacement), optimized builds, modern dev experience
TailwindCSS	3.x	Styling	Utility-first CSS, consistent design, smaller bundle size
React Router	6.x	Routing	Client-side navigation, nested routes, URL parameter handling
Axios	1.x	HTTP Client	Promise-based, request/response interceptors, automatic JSON transformation

Zustand	4.x	State Management	Lightweight alternative to Redux, simple API, no boilerplate

Backend Technologies

Technology	Version	Purpose	Why We Use It
Python	3.11	Programming Language	Easy to read, rich ecosystem, great for rapid development
FastAPI	0.109	Web Framework	Automatic API docs, data validation with Pydantic, async support, fast performance
Pydantic	2.5	Data Validation	Type hints for validation, automatic schema generation, JSON serialization
SQLAlchemy	2.0	ORM	Database abstraction, async support, migration handling
Alembic	1.13	DB Migrations	Version control for database schema, rollback support
PostgreSQL	15	Database	ACID compliance, JSON support, full-text search, reliable
AsyncPG	0.29	DB Driver	High-performance async PostgreSQL driver

Infrastructure & DevOps

Technology	Version	Purpose	Why We Use It
Docker	24.x	Containerization	Package applications with dependencies, consistent environments
Kubernetes	1.28	Orchestration	Auto-scaling, self-healing, service discovery, rolling updates
Kind	0.20	Local K8s	Kubernetes in Docker, perfect for local development and testing
Istio	1.20	Service Mesh	mTLS security, traffic management, observability, circuit breaking
ArgoCD	2.9	GitOps CD	Automated deployments, git as source of truth, easy rollbacks
Helm	3.13	Package Manager	Template Kubernetes manifests, version applications, reusable charts

Monitoring & Observability

Technology	Version	Purpose	Why We Use It
Prometheus	2.48	Metrics Collection	Time-series database, powerful query language (PromQL), alerting
Grafana	10.2	Visualization	Beautiful dashboards, multiple data sources, alerting
Loki	2.9	Log Aggregation	Like Prometheus but for logs, efficient storage, LogQL queries
Jaeger	1.52	Distributed Tracing	Track requests across services, identify bottlenecks

Message Queue & Caching

Technology	Version	Purpose	Why We Use It
RabbitMQ	3.12	Message Broker	Reliable message delivery, multiple messaging patterns, dead letter queues
Redis	7.2	Cache & Session Store	In-memory speed, pub/sub support, data structures beyond key-value

---

4. Core Concepts

4.1 Microservices Architecture

What is it?

Instead of one big application (monolith), we split functionality into small, independent services that communicate over the network.

Our Microservices:

1. **API Gateway** - Entry point for all client requests
2. **Auth Service** - Handles authentication and user management
3. **Restaurant Service** - Manages restaurants, menus, and tables
4. **Order Service** - Processes orders and order sessions
5. **Customer Service** - Manages customer accounts

Why Microservices?

 Independent Scaling: Scale only the services that need it (e.g., Order Service during peak hours)

 Independent Deployment: Deploy Auth Service without affecting Restaurant Service

 Technology Freedom: Could use different languages for different services if needed

 Team Autonomy: Different teams can own different services

 Fault Isolation: If Order Service crashes, Restaurant Service keeps running

 Complexity: More moving parts, need service discovery, distributed tracing

 Network Latency: Inter-service calls are slower than in-process calls

 Data Consistency: Harder to maintain ACID transactions across services

4.2 API Gateway Pattern

What is it?

A single entry point that routes requests to appropriate microservices.

What our API Gateway does:

```
# Pseudo-code showing routing logic
if path.startswith("/api/v1/auth"):
    forward_to(auth_service)
elif path.startswith("/api/v1/restaurants"):
    forward_to(restaurant_service)
elif path.startswith("/api/v1/orders"):
    forward_to(order_service)
```

Benefits:

- **Single endpoint** for clients (https://restaurant.corpv3.com)
- **Authentication** happens once at the gateway
- **Rate limiting** applied centrally
- **Request/Response transformation**
- **Load balancing** across service instances

4.3 JWT (JSON Web Tokens)

What is it?

A secure way to transmit information between parties as a JSON object that can be verified and trusted.

Structure:

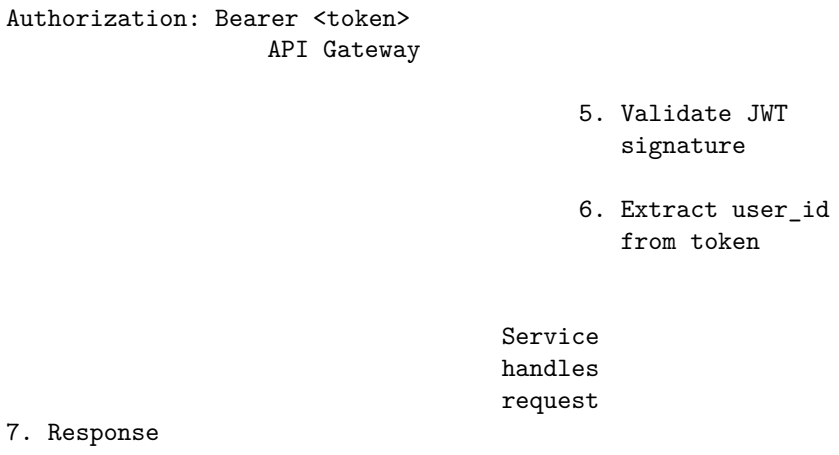
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIn0.dozjgNryP4J3jVmNHl0w5N_XgL0n3I9PlFUP0THsR8
    ^Header (base64)              ^Payload (base64)           ^Signature
```

Our JWT Payload:

```
{
  "sub": "user-uuid-here",
  "username": "admin",
  "role": "master_admin",
  "restaurant_id": "restaurant-uuid-or-null",
  "exp": 1736187096,
  "iat": 1736185296
}
```

How it works:

```
 Client                        Auth Service

       1. Login (username+password)

                                    2. Validate credentials
                                       Hash password & compare

       3. JWT Token + Refresh Token



       4. API Request with JWT
```



Why JWT?

- **Stateless**: Server doesn't need to store sessions
- **Scalable**: No session store to synchronize
- **Cross-domain**: Works across different domains
- **Self-contained**: All info in the token
- **Size**: Larger than session IDs
- **Can't revoke**: Token valid until expiry (we use short expiry + refresh tokens)

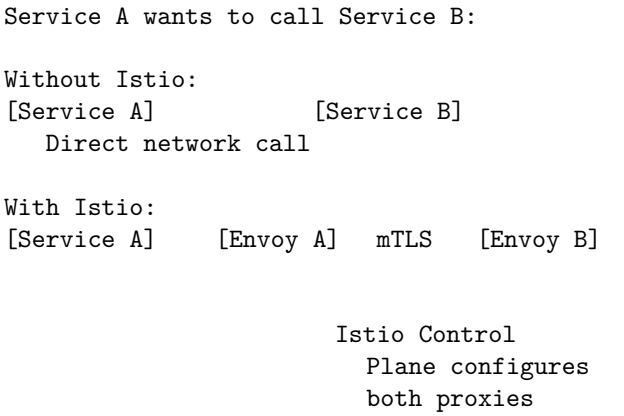
4.4 Service Mesh (Istio)

What is it?

A dedicated infrastructure layer that handles service-to-service communication.

How it works:

Every service pod gets an **Envoy sidecar proxy** that intercepts all network traffic.



What Istio gives us:

1. **Mutual TLS (mTLS)**: All service-to-service traffic encrypted automatically
2. **Traffic Management**: Canary deployments, A/B testing, circuit breaking
3. **Observability**: Automatic metrics, traces, and logs
4. **Security**: Authorization policies at network level

Example - Canary Deployment:

```yaml
# Route 90% traffic to v1, 10% to v2
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: order-service
spec:
  hosts:
  - order-service
  http:
  - match:
    - headers:
        user-type:
          exact: beta-tester
    route:
    - destination:
        host: order-service
        subset: v2
  - route:
    - destination:
        host: order-service
        subset: v1
      weight: 90
    - destination:
        host: order-service
        subset: v2
      weight: 10
```

4.5 Kubernetes Concepts

Pods:

The smallest deployable unit. Contains one or more containers (app + sidecar).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: auth-service
spec:
  containers:
  - name: auth-service      # Your application
    image: shadrach85/auth-service:v1.0
  - name: istio-proxy       # Envoy sidecar (injected by Istio)
    image: istio/proxyv2
```

Deployments:

Manages multiple pod replicas, handles rolling updates.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-service
spec:
  replicas: 3               # Run 3 copies
  strategy:
    type: RollingUpdate     # Update one at a time
    rollingUpdate:
      maxUnavailable: 1
```

Services:

Provides a stable network endpoint for pods (which can come and go).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: auth-service
spec:
  selector:
    app: auth-service
  ports:
  - port: 8001            # Service port
    targetPort: 8001      # Container port
```

Ingress:

Routes external traffic to services based on URL path/hostname.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: restaurant-ingress
spec:
  rules:
  - host: restaurant.corpv3.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: api-gateway
            port:
              number: 8000
```

---

5. Microservices Deep Dive

5.1 API Gateway (Port 8000)

Purpose: Single entry point for all client requests

Responsibilities:

- Route requests to appropriate backend services
- Validate JWT tokens
- Apply rate limiting
- Transform requests/responses
- Handle CORS

Technology: FastAPI + HTTPBearer security

Key Code Flow:

```python
@app.api_route("/{path:path}", methods=["GET", "POST", "PUT", "DELETE"])
async def gateway(
    request: Request,
    path: str,
    credentials: Optional[HTTPAuthorizationCredentials] = Depends(security)
):
    # 1. Apply rate limiting
    await rate_limit(request)

    # 2. Route based on path
    if path.startswith("api/v1/auth"):
        target_url = f"{AUTH_SERVICE_URL}/{path}"
    elif path.startswith("api/v1/restaurants"):
        target_url = f"{RESTAURANT_SERVICE_URL}/{path}"

    # 3. Forward Authorization header
    headers = dict(request.headers)
    if credentials:
        headers["authorization"] = f"Bearer {credentials.credentials}"

    # 4. Forward request to target service
    async with httpx.AsyncClient() as client:
        response = await client.request(
            method=request.method,
            url=target_url,
            headers=headers,
            content=await request.body()
        )

    return Response(content=response.content)
```

Environment Variables:

```
AUTH_SERVICE_URL=http://auth-service:8001
RESTAURANT_SERVICE_URL=http://restaurant-service:8003
ORDER_SERVICE_URL=http://order-service:8004
CUSTOMER_SERVICE_URL=http://customer-service:8002
```

5.2 Auth Service (Port 8001)

Purpose: Handle authentication, user management, authorization

Responsibilities:

- User login/logout
- JWT token generation
- Token refresh
- User CRUD operations
- Role-based access control (RBAC)

Database Tables:

```sql
-- Users table
CREATE TABLE users (
    id UUID PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    hashed_password VARCHAR(255) NOT NULL,
    role VARCHAR(20) NOT NULL,  -- master_admin, restaurant_admin, chef, customer
```

```
    restaurant_id UUID,          -- NULL for master_admin
    is_active BOOLEAN DEFAULT TRUE,
    is_verified BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP,
    last_login TIMESTAMP
);

-- Refresh tokens table
CREATE TABLE refresh_tokens (
    id UUID PRIMARY KEY,
    token VARCHAR(512) UNIQUE NOT NULL,
    user_id UUID REFERENCES users(id),
    expires_at TIMESTAMP,
    created_at TIMESTAMP
);
```

User Roles:

1. **master_admin**: Super admin, can manage all restaurants
2. **restaurant_admin**: Manages one specific restaurant
3. **chef**: Kitchen staff, sees orders for their restaurant
4. **customer**: End users who place orders

JWT Secret:

```
JWT_SECRET_KEY = "change-me-in-production"  # Shared across Auth Service & API Gateway
JWT_ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
REFRESH_TOKEN_EXPIRE_DAYS = 7
```

Login Flow:

```
@router.post("/login")
async def login(credentials: LoginRequest, db: AsyncSession):
    # 1. Find user by username
    user = await db.execute(
        select(User).where(User.username == credentials.username)
    )

    # 2. Verify password (bcrypt)
    if not verify_password(credentials.password, user.hashed_password):
        raise HTTPException(401, "Invalid credentials")

    # 3. Generate access token (30 min expiry)
    access_token = create_access_token({
        "sub": str(user.id),
        "username": user.username,
        "role": user.role.value,
        "restaurant_id": str(user.restaurant_id) if user.restaurant_id else None
    })

    # 4. Generate refresh token (7 days)
    refresh_token = create_refresh_token()

    # 5. Store refresh token in DB
    db_token = RefreshToken(
        token=refresh_token,
        user_id=user.id,
        expires_at=datetime.utcnow() + timedelta(days=7)
```

```
    )
    await db.add(db_token)

    # 6. Return tokens
    return TokenResponse(
        access_token=access_token,
        refresh_token=refresh_token,
        token_type="bearer",
        expires_in=1800
    )
```

5.3 Restaurant Service (Port 8003)

Purpose: Manage restaurants, menus, tables, and categories

Responsibilities:

- Restaurant CRUD
- Menu item management
- Table management
- Category management
- QR code generation
- Image upload handling

Database Tables:

```
CREATE TABLE restaurants (
    id UUID PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address TEXT,
    phone VARCHAR(20),
    email VARCHAR(255),
    description TEXT,
    logo_url VARCHAR(512),
    is_active BOOLEAN DEFAULT TRUE,
    subscription_status VARCHAR(20),
    subscription_plan VARCHAR(20),
    created_at TIMESTAMP
);

CREATE TABLE menu_items (
    id UUID PRIMARY KEY,
    restaurant_id UUID REFERENCES restaurants(id),
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    category VARCHAR(50),
    image_url VARCHAR(512),
    is_available BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP
);

CREATE TABLE tables (
    id UUID PRIMARY KEY,
    restaurant_id UUID REFERENCES restaurants(id),
    table_number VARCHAR(20) NOT NULL,
    capacity INTEGER,
    status VARCHAR(20),  -- available, occupied, reserved, cleaning
```

```
    qr_code TEXT,         -- Base64 encoded QR code
    created_at TIMESTAMP
);
```

Multi-Tenant Data Isolation:

```python
# Every query filters by restaurant_id
@router.get("/menu-items")
async def get_menu_items(
    current_user: User = Depends(get_current_user),
    db: AsyncSession = Depends(get_db)
):
    # Restaurant admin can only see their own menu
    query = select(MenuItem).where(
        MenuItem.restaurant_id == current_user.restaurant_id
    )

    result = await db.execute(query)
    return result.scalars().all()
```

5.4 Order Service (Port 8004)

Purpose: Handle orders, sessions, and analytics

Responsibilities:

- Order creation and management
- Order session handling (table orders)
- Real-time order status updates
- Kitchen notifications
- Sales analytics and reporting

Database Tables:

```
CREATE TABLE orders (
    id UUID PRIMARY KEY,
    restaurant_id UUID,
    table_id UUID,
    customer_id UUID,
    session_id UUID,
    order_type VARCHAR(20),   -- TABLE or ONLINE
    status VARCHAR(20),        -- pending, confirmed, preparing, ready, served
    total_amount DECIMAL(10, 2),
    items JSONB,              -- Array of order items
    created_at TIMESTAMP,
    updated_at TIMESTAMP
);

CREATE TABLE order_sessions (
    id UUID PRIMARY KEY,
    restaurant_id UUID,
    table_id UUID,
    status VARCHAR(20),       -- active, completed, cancelled
    total_amount DECIMAL(10, 2),
    started_at TIMESTAMP,
    ended_at TIMESTAMP
);
```

Order Status Flow:

```
Customer Places Order
        ↓
    [PENDING]
        ↓
Restaurant Confirms  → [CONFIRMED]
        ↓
Kitchen Starts       → [PREPARING]
        ↓
Food Ready           → [READY]
        ↓
Served to Table      → [SERVED]
        ↓
Payment Done         → [COMPLETED]
```

Real-time Updates with RabbitMQ:

```python
# When order status changes
async def update_order_status(order_id: UUID, new_status: str):
    # 1. Update in database
    order.status = new_status
    await db.commit()

    # 2. Publish event to RabbitMQ
    await rabbitmq.publish(
        exchange="orders",
        routing_key=f"order.{order.restaurant_id}.status_changed",
        message={
            "order_id": str(order_id),
            "status": new_status,
            "timestamp": datetime.utcnow().isoformat()
        }
    )
```

5.5 Customer Service (Port 8002)

Purpose: Manage customer-specific functionality

Responsibilities:

- Customer profile management
- Order history
- Feedback and ratings
- Loyalty points (future)

Database Tables:

```sql
CREATE TABLE customer_profiles (
    id UUID PRIMARY KEY,
    user_id UUID REFERENCES users(id),
    restaurant_id UUID,
    preferences JSONB,
    loyalty_points INTEGER DEFAULT 0,
    created_at TIMESTAMP
);

CREATE TABLE feedback (
    id UUID PRIMARY KEY,
    order_id UUID,
    customer_id UUID,
    restaurant_id UUID,
```

```
    rating INTEGER,  -- 1-5
    comment TEXT,
    created_at TIMESTAMP
);
```

---

6. Authentication & Authorization

6.1 Password Hashing

What we use: bcrypt

Why bcrypt?

- **Adaptive**: Can increase difficulty over time as computers get faster
- **Salted**: Each password gets unique salt, prevents rainbow table attacks
- **Slow**: Intentionally slow to prevent brute force

How it works:

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# When user signs up
hashed_password = pwd_context.hash("user_password123")
# Stores: $2b$12$KIJklEjLK23j4lk5j6lk7OuYT8.vQ.o2j3k4l5j6k7l8m9n0o1p2q
#         ^alg ^cost ^salt             ^hash

# When user logs in
is_valid = pwd_context.verify("user_password123", hashed_password)
```

6.2 JWT Token Flow

Access Token (30 minutes):

```
{
  "sub": "550e8400-e29b-41d4-a716-446655440000",
  "username": "admin",
  "role": "master_admin",
  "restaurant_id": null,
  "exp": 1736187096,
  "iat": 1736185296
}
```

Refresh Token (7 days):

Stored in database, used to get new access token when it expires.

Token Refresh Flow:

```
  Client                                Auth Service


      1. Access token expires
         (Client detects 401 error)

      2. POST /refresh
         Body: { refresh_token: "..." }
```

```
                                          3. Validate refresh token
                                             - Check in DB
                                             - Verify not expired

      4. New access token


      5. Retry original request
         with new access token
```

6.3 Role-Based Access Control (RBAC)

Implementation with FastAPI Dependencies:

```python
# Dependency to get current user from JWT
async def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db: AsyncSession = Depends(get_db)
) -> User:
    token = credentials.credentials

    # Decode JWT
    payload = decode_token(token)
    user_id = payload.get("sub")

    # Fetch user from DB
    user = await db.execute(
        select(User).where(User.id == user_id)
    )
    return user.scalar_one_or_none()

# Dependency to require master admin
async def require_master_admin(
    current_user: User = Depends(get_current_user)
) -> User:
    if current_user.role != UserRole.MASTER_ADMIN:
        raise HTTPException(403, "Master admin access required")
    return current_user

# Protected endpoint
@router.get("/users", response_model=List[UserResponse])
async def list_users(
    current_user: User = Depends(require_master_admin)
):
    # Only master_admin can access this
    return await get_all_users()
```

Permission Matrix:

Action	Master Admin	Restaurant Admin	Chef	Customer
Create Restaurant	✅	❌	❌	❌
Manage Menu	✅	✅ (own)	❌	❌
View Orders	✅	✅ (own)	✅ (own)	✅ (own)
Update Order Status	✅	✅	✅	❌

Action	Master Admin	Restaurant Admin	Chef	Customer
Create User (any role)	✅	✅	✅	✅
Create Chef	✅	✅ (own restaurant)	✅	✅

---

7. Infrastructure & Deployment

7.1 Docker Containerization

What is Docker?

Package your application with all dependencies into a portable container.

Dockerfile Example (Auth Service):

```
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y gcc postgresql-client

# Copy and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY shared/ /app/shared/
COPY services/auth-service/app /app/app

# Create non-root user for security
RUN useradd -m -u 1000 authuser && chown -R authuser:authuser /app
USER authuser

EXPOSE 8001

# Run application
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8001"]
```

Multi-stage Build (for smaller images):

```
# Build stage
FROM python:3.11-slim AS builder
WORKDIR /build
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Runtime stage
FROM python:3.11-slim
COPY --from=builder /root/.local /root/.local
COPY app /app
ENV PATH=/root/.local/bin:$PATH
CMD ["uvicorn", "app.main:app"]
```

7.2 Kubernetes Deployment

Why Kubernetes?

- **Auto-scaling**: Scale pods based on CPU/memory
- **Self-healing**: Restart failed pods automatically
- **Rolling updates**: Zero-downtime deployments
- **Service discovery**: Services find each other by name
- **Load balancing**: Distribute traffic across pods

Deployment YAML:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-service
  namespace: restaurant-system
spec:
  replicas: 2                    # Run 2 pods
  selector:
    matchLabels:
      app: auth-service
  template:
    metadata:
      labels:
        app: auth-service
        version: v1
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8001"
        sidecar.istio.io/inject: "true"  # Enable Istio
    spec:
      containers:
      - name: auth-service
        image: shadrach85/auth-service:v1.0.0
        ports:
        - containerPort: 8001
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: url
        - name: JWT_SECRET_KEY
          valueFrom:
            secretKeyRef:
              name: jwt-secret
              key: secret
        resources:
          requests:
            memory: "256Mi"
            cpu: "100m"
          limits:
            memory: "512Mi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8001
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
```

```
          httpGet:
            path: /health
            port: 8001
          initialDelaySeconds: 5
          periodSeconds: 5
```

Service YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: auth-service
  namespace: restaurant-system
spec:
  selector:
    app: auth-service
  type: ClusterIP              # Internal only
  ports:
  - port: 8001
    targetPort: 8001
    protocol: TCP
```

7.3 GitOps with ArgoCD

What is GitOps?

Git repository is the single source of truth for infrastructure and application state.

How ArgoCD works:

```
1. Developer commits K8s manifests to Git
      git push origin main

2. ArgoCD detects changes (polling or webhook)
      Compares Git state vs Cluster state

3. If different (Out of Sync):
      Shows drift in UI
      Manual or Auto-sync to apply changes

4. Applies changes to Kubernetes
      kubectl apply -f manifests/

5. Monitors health and sync status
      Shows in ArgoCD UI dashboard
```

ArgoCD Application:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: restaurant-system
  namespace: argocd
spec:
  project: default
```

```yaml
  source:
    repoURL: https://github.com/your-org/restaurant-k8s
    targetRevision: main
    path: manifests
  destination:
    server: https://kubernetes.default.svc
    namespace: restaurant-system
  syncPolicy:
    automated:
      prune: true              # Delete resources not in Git
      selfHeal: true           # Revert manual changes
    syncOptions:
    - CreateNamespace=true
```

Benefits:

 Audit trail: All changes tracked in Git history

 Rollback: `git revert` to undo changes

 Disaster recovery: Recreate entire cluster from Git

 Consistency: Dev/Staging/Prod from same manifests

7.4 Helm Charts

What is Helm?

Package manager for Kubernetes. Templates for K8s manifests with variables.

Chart Structure:

```
restaurant-chart/
   Chart.yaml                # Chart metadata
   values.yaml               # Default configuration
   values-dev.yaml           # Dev environment overrides
   values-prod.yaml          # Prod environment overrides
   templates/
       deployment.yaml       # Templated deployment
       service.yaml          # Templated service
       ingress.yaml          # Templated ingress
       _helpers.tpl          # Reusable template snippets
```

Template Example:

```yaml
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.serviceName }}
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
      - name: {{ .Values.serviceName }}
        image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
        env:
        - name: DATABASE_URL
          value: {{ .Values.database.url }}
```

values.yaml:

```yaml
serviceName: auth-service
replicaCount: 2

image:
  repository: shadrach85/auth-service
  tag: v1.0.0

database:
  url: postgresql://user:pass@postgres:5432/restaurant_db
```

Deploy:

```bash
# Install chart
helm install restaurant ./restaurant-chart -f values-prod.yaml

# Upgrade
helm upgrade restaurant ./restaurant-chart -f values-prod.yaml

# Rollback
helm rollback restaurant 1
```

---

8. Monitoring & Observability

8.1 Prometheus Metrics

What metrics do we collect?

System Metrics:

- CPU usage per pod
- Memory usage per pod
- Network I/O
- Disk I/O

Application Metrics:

- HTTP request rate
- HTTP error rate (4xx, 5xx)
- Request duration (p50, p95, p99)
- Active connections
- Database query duration

Business Metrics:

- Orders per minute
- Revenue per hour
- Active tables
- Menu item popularity

Prometheus Query Examples:

```
# Request rate per service
rate(http_requests_total{service="order-service"}[5m])

# 95th percentile latency
histogram_quantile(0.95, rate(http_request_duration_seconds_bucket[5m]))

# Error rate percentage
sum(rate(http_requests_total{status=~"5.."}[5m]))
  /
```

```
sum(rate(http_requests_total[5m])) * 100

# Orders per minute
rate(orders_created_total[1m]) * 60
```

8.2 Grafana Dashboards

Dashboard Layout:

```
  Restaurant Management System - Overview


    Total        Active       Revenue
    Orders       Tables       Today
     1,234          42        $12,450


          Orders Over Time

      30
      20
      10
       0


    Service Health          Error Rate
    Auth:                  Auth:      0.1%
    Restaurant:             Restaurant: 0.0%
    Order:                 Order:     0.5%
```

8.3 Loki for Logs

Log Aggregation:

Instead of SSH-ing into each pod to check logs, Loki centralizes everything.

LogQL Query Examples:

```
# All errors in the last hour
{namespace="restaurant-system"} |= "ERROR"

# Auth service login attempts
{app="auth-service"} |= "User logged in"

# Slow database queries (>1s)
{app=~".*-service"}
  | json
  | query_duration > 1000

# Failed orders
{app="order-service"}
```

```
  |= "Order failed"
  | json
  | restaurant_id="abc-123"
```

8.4 Jaeger Distributed Tracing

What is Distributed Tracing?

Track a single request as it flows through multiple services.

Trace Example:

```
Request: POST /api/v1/orders


 Span 1: API Gateway                          [0ms      245ms]
     Span 2: Auth Service - Validate JWT  [2ms    15ms]

     Span 3: Order Service                [20ms     240ms]
         Span 4: Restaurant Service       [25ms    45ms]
            Get menu items

         Span 5: Database Query           [50ms    180ms]
            INSERT INTO orders            [SLOW!]

         Span 6: RabbitMQ Publish         [185ms   190ms]
             Send order notification

Total: 245ms
```

Identifying Performance Issues:

- Span 5 (DB INSERT) takes 130ms - might need indexing
- Span 2 (JWT validation) is fast (13ms) - good!
- Overall request takes 245ms - acceptable for order creation

---

9. Data Flow & Connectivity

9.1 Complete Request Flow

Scenario: Customer places an order via QR code

```
 Customer   Scans QR code at table
  Phone


       1. HTTPS GET https://restaurant.corpv3.com/table/<table_id>


   Nginx         TLS termination, load balancing
   Ingress


         2. Forward to Frontend Service


   Frontend      Serve React app
```

```
   (Nginx)


         3. React app loads, fetches menu
            GET /api/v1/restaurants/<id>/menu


  API Gateway    Check JWT (if logged in), route request


         4. Authorization: Bearer <jwt_token>
            Forward to Restaurant Service


 Restaurant Service  Query menu items from DB


         5. SELECT * FROM menu_items WHERE restaurant_id=?


  PostgreSQL     Return menu items


         6. Menu data flows back to customer

      Display menu on phone

User selects items and clicks "Place Order"

         7. POST /api/v1/orders
            Body: { table_id, items: [...] }


  API Gateway    Validate JWT, route to Order Service


         8. Forward order creation request


 Order Service


          9. Validate restaurant_id matches JWT

          10. Calculate total price
              Query menu item prices from Restaurant Service
              Sum up order total

          11. INSERT INTO orders
              Save order to database
              Status: PENDING

          12. Publish event to RabbitMQ
              Exchange: orders
              Routing key: order.restaurant_123.created
              Message: { order_id, table_id, items, total }
```

```
          13. Return order confirmation
                  Order ID
                  Estimated time
                  Status: PENDING


   RabbitMQ      Order created event


          14. Kitchen display consumer receives event
              Updates real-time dashboard


 Kitchen App     Shows new order on chef's screen
  (WebSocket)    Chef marks as PREPARING


          15. PUT /api/v1/orders/<id>/status
              Body: { status: "PREPARING" }


 Order Service   Update status, publish event


           UPDATE orders SET status='PREPARING'

           Publish to RabbitMQ
               Customer app receives status update
```

9.2 Database Connections

Connection Pooling:

```python
# SQLAlchemy async engine with connection pooling
from sqlalchemy.ext.asyncio import create_async_engine

engine = create_async_engine(
    DATABASE_URL,
    pool_size=10,              # Keep 10 connections open
    max_overflow=20,           # Allow 20 more if needed
    pool_pre_ping=True,        # Verify connection before use
    pool_recycle=3600,         # Recycle connections every hour
    echo=False                 # Don't log SQL (use in dev only)
)
```

Why connection pooling?

Creating a new database connection is expensive (100-200ms). Connection pool keeps connections open and reuses them, reducing latency from 150ms to 1ms.

9.3 Inter-Service Communication

Service-to-Service HTTP Calls:

```python
# Order Service calling Restaurant Service
async with httpx.AsyncClient() as client:
    response = await client.get(
        f"http://restaurant-service:8003/api/v1/menu-items/{item_id}",
        headers={"Authorization": f"Bearer {internal_token}"}
```

```
    )
    menu_item = response.json()
```

Why internal service calls need auth?

Even though services are inside the cluster, we still validate to prevent unauthorized access if a service is compromised.

9.4 Message Queue Patterns

Pub/Sub with RabbitMQ:

```python
# Publisher (Order Service)
async def publish_order_event(order: Order):
    message = {
        "event": "order.created",
        "order_id": str(order.id),
        "restaurant_id": str(order.restaurant_id),
        "table_id": str(order.table_id),
        "items": order.items,
        "total": float(order.total_amount)
    }

    await rabbitmq.publish(
        exchange="orders",
        routing_key=f"order.{order.restaurant_id}.created",
        message=json.dumps(message)
    )

# Consumer (Kitchen Display Service)
async def consume_order_events():
    async for message in rabbitmq.consume(
        queue="kitchen_orders",
        routing_keys=["order.*.created", "order.*.status_changed"]
    ):
        order_data = json.loads(message.body)
        await update_kitchen_display(order_data)
```

Exchange Types:

- **Direct**: Route to queue with exact routing key match
- **Topic**: Route based on pattern (e.g., `order.*.created`)
- **Fanout**: Broadcast to all queues
- **Headers**: Route based on message headers

---

10. Troubleshooting Guide

10.1 Common Issues

Issue: Service returns 503 Service Unavailable

Possible Causes:

1. Target service is down
2. Service name typo in code
3. Network policy blocking traffic
4. Istio mTLS mismatch

Debug Steps:

```
# 1. Check if pods are running
kubectl get pods -n restaurant-system

# 2. Check service endpoints
kubectl get endpoints -n restaurant-system

# 3. Check service logs
kubectl logs -n restaurant-system deployment/order-service

# 4. Test service connectivity from another pod
kubectl run test-pod --rm -it --image=curlimages/curl -- \
  curl http://order-service:8004/health

# 5. Check Istio sidecar status
kubectl get pods -n restaurant-system -o jsonpath='{.items[*].spec.containers[*].name}'
```

Issue: JWT Token Invalid (401 Unauthorized)

Possible Causes:

1. Token expired
2. JWT secret mismatch between services
3. Authorization header not forwarded
4. Token signature invalid

Debug Steps:

```
# 1. Decode JWT to check expiry (use jwt.io)
# Copy token from browser DevTools

# 2. Check JWT secret in auth service
kubectl get secret jwt-secret -n restaurant-system -o jsonpath='{.data.secret}' | base64 -d

# 3. Check JWT secret in API gateway
kubectl get secret jwt-secret -n restaurant-system -o jsonpath='{.data.secret}' | base64 -d

# 4. Check API gateway logs for header forwarding
kubectl logs -n restaurant-system deployment/api-gateway | grep -i authorization

# 5. Manually test auth endpoint
curl -X POST https://restaurant.corpv3.com/api/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"admin123"}'
```

Issue: Database Connection Failed

Possible Causes:

1. Wrong database credentials
2. Database pod not ready
3. Connection pool exhausted
4. Network policy blocking DB access

Debug Steps:

```
# 1. Check PostgreSQL pod
kubectl get pods -n restaurant-system | grep postgres
```

```
# 2. Check database logs
kubectl logs -n restaurant-system statefulset/postgres

# 3. Test database connectivity
kubectl run psql-test --rm -it --image=postgres:15 -- \
  psql postgresql://user:pass@postgres:5432/restaurant_db

# 4. Check connection pool stats (in application logs)
kubectl logs -n restaurant-system deployment/auth-service | grep "pool"

# 5. Check service secret
kubectl get secret db-credentials -n restaurant-system -o yaml
```

Issue: ArgoCD Shows Out of Sync

Possible Causes:

1. Manual changes to cluster (kubectl apply)
2. Git repo has newer version
3. Auto-sync disabled

Resolution:

```
# 1. Check sync status
argocd app get restaurant-system

# 2. See what's different
argocd app diff restaurant-system

# 3. Sync manually
argocd app sync restaurant-system

# 4. Enable auto-sync
argocd app set restaurant-system --sync-policy automated

# 5. Revert manual changes (hard refresh)
argocd app sync restaurant-system --prune --force
```

10.2 Debugging Checklist

Service Not Responding:

- ☐ Check pod status: `kubectl get pods -n restaurant-system`
- ☐ Check pod logs: `kubectl logs -n restaurant-system <pod-name>`
- ☐ Check service endpoints: `kubectl get endpoints -n restaurant-system`
- ☐ Check ingress: `kubectl get ingress -n restaurant-system`
- ☐ Check Istio sidecar: Pod should have 2/2 containers

Authentication Issues:

- ☐ Verify JWT token not expired (check `exp` claim)
- ☐ Verify JWT secret matches across services
- ☐ Check Authorization header in API Gateway logs
- ☐ Test login endpoint directly
- ☐ Verify user exists and is active in database

Database Issues:

- ☐ Check PostgreSQL pod running
- ☐ Verify database credentials in secret
- ☐ Check connection pool not exhausted
- ☐ Verify database migrations applied
- ☐ Check disk space on postgres PVC

Performance Issues:

- ☐ Check Grafana dashboards for spikes
- ☐ Review Jaeger traces for slow operations
- ☐ Check resource limits (CPU/memory)
- ☐ Review database slow query log
- ☐ Check connection pool size

Appendix A: Environment Variables

Auth Service

```
# Database
DATABASE_URL=postgresql+asyncpg://restaurant_user:restaurant_pass@postgres:5432/restaurant_db

# JWT
JWT_SECRET_KEY=change-me-in-production
JWT_ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30
REFRESH_TOKEN_EXPIRE_DAYS=7

# Service
SERVICE_NAME=auth-service
PORT=8001
LOG_LEVEL=INFO
```

API Gateway

```
# Service URLs
AUTH_SERVICE_URL=http://auth-service:8001
RESTAURANT_SERVICE_URL=http://restaurant-service:8003
ORDER_SERVICE_URL=http://order-service:8004
CUSTOMER_SERVICE_URL=http://customer-service:8002

# JWT (same secret as auth service!)
JWT_SECRET_KEY=change-me-in-production
JWT_ALGORITHM=HS256

# Rate Limiting
RATE_LIMIT_REQUESTS=100
RATE_LIMIT_PERIOD=60
```

Appendix B: Useful Commands

Kubernetes

```
# Get all resources in namespace
kubectl get all -n restaurant-system

# Describe pod (shows events)
kubectl describe pod <pod-name> -n restaurant-system

# Execute command in pod
kubectl exec -it <pod-name> -n restaurant-system -- /bin/bash

# Port forward to local
kubectl port-forward -n restaurant-system svc/auth-service 8001:8001

# View logs (follow)
kubectl logs -f -n restaurant-system deployment/auth-service

# Scale deployment
kubectl scale deployment auth-service --replicas=3 -n restaurant-system

# Rollout status
kubectl rollout status deployment/auth-service -n restaurant-system

# Rollback deployment
kubectl rollout undo deployment/auth-service -n restaurant-system

# Apply manifests
kubectl apply -f infrastructure/kubernetes/

# Delete resource
kubectl delete pod <pod-name> -n restaurant-system --force
```

Docker

```
# Build image
docker build -t shadrach85/auth-service:v1.0.0 -f services/auth-service/Dockerfile .

# Push to DockerHub
docker push shadrach85/auth-service:v1.0.0

# Load image into Kind cluster
kind load docker-image shadrach85/auth-service:v1.0.0 --name restaurant-cluster

# View image layers
docker history shadrach85/auth-service:v1.0.0

# Remove unused images
docker image prune -a

# View container logs
docker logs <container-id>
```

ArgoCD

```
# Login to ArgoCD
argocd login localhost:8080 --username admin --password <password>
```

```
# List applications
argocd app list

# Get application details
argocd app get restaurant-system

# Sync application
argocd app sync restaurant-system

# View sync history
argocd app history restaurant-system

# Rollback to previous version
argocd app rollback restaurant-system <revision-number>

# Delete application
argocd app delete restaurant-system
```

Istio

```
# Check Istio installation
istioctl version

# Analyze configuration
istioctl analyze -n restaurant-system

# View proxy configuration
istioctl proxy-config routes <pod-name> -n restaurant-system

# Enable Istio injection
kubectl label namespace restaurant-system istio-injection=enabled

# Check mTLS status
istioctl authn tls-check <pod-name> -n restaurant-system

# Inject Envoy sidecar manually
istioctl kube-inject -f deployment.yaml | kubectl apply -f -
```

Appendix C: Glossary

API (Application Programming Interface): Interface that defines how software components interact

CRUD: Create, Read, Update, Delete - basic database operations

DNS (Domain Name System): Translates domain names to IP addresses

Horizontal Scaling: Adding more instances of a service

Vertical Scaling: Adding more CPU/RAM to existing instances

Ingress: Kubernetes resource for HTTP(S) routing from outside the cluster

JWT (JSON Web Token): Secure token format for authentication

Kind (Kubernetes in Docker): Run Kubernetes cluster in Docker containers

mTLS (Mutual TLS): Both client and server verify each other's identity

ORM (Object-Relational Mapping): Map database tables to Python objects

Pod: Smallest deployable unit in Kubernetes (one or more containers)

Prometheus: Time-series database for metrics

QR Code: Quick Response code - 2D barcode

RBAC (Role-Based Access Control): Permissions based on user roles

REST (Representational State Transfer): Architectural style for APIs

SaaS (Software as a Service): Cloud-hosted software subscription model

Sidecar: Container that runs alongside main application container

StatefulSet: Kubernetes resource for stateful applications (databases)

TLS (Transport Layer Security): Encryption for data in transit (HTTPS)

WebSocket: Full-duplex communication protocol for real-time updates

---

End of Runbook

For questions or issues, refer to: - GitHub Issues: https://github.com/your-org/restaurant-management/issues - Documentation: https://docs.restaurant-system.com - Slack Channel: #restaurant-system-support