Product Requirements Document: Fuel Delivery Management Platform

Overview

Problem Statement

Fuel delivery companies currently lack a modern, scalable SaaS platform to manage their operations efficiently. They need a centralized system to coordinate deliveries, track orders, manage clients, and allocate delivery trucks across their fleet.

Solution

A multi-tenant SaaS backend API that enables fuel delivery companies to manage their complete operational workflow - from order creation to delivery execution. The platform provides secure data segregation between different fuel delivery companies while sharing the same infrastructure.

Target Users

- **System Administrators**: Manage company-wide settings and user access
- **Dispatchers/Operations Staff**: Create and manage fuel delivery orders
- **Company Owners**: Monitor operations across their fuel delivery business

Value Proposition

- Centralized operational management for fuel delivery companies
- Multi-tenant architecture enabling SaaS business model
- Scalable API-first architecture for future mobile/web clients
- Modern technology stack (Symfony + API Platform) ensuring maintainability

---

Core Features

1. User Management & Authentication

What it does:

- Secure user authentication and authorization
- Role-based access control (RBAC)
- User profile management

Why it's important:

- Foundation for secure system access
- Enables proper audit trails (who created what order)
- Supports multi-user companies with different permission levels

How it works:

- JWT-based authentication via API Platform
- Users belong to a specific tenant (fuel delivery company)

- Different roles: Admin, Dispatcher, Driver (future), Read-only

2. Multi-Tenancy & Data Segregation

What it does:

- Complete data isolation between different fuel delivery companies
- Each company operates in its own secure environment
- Shared infrastructure with logical data separation

Why it's important:

- Core requirement for SaaS model
- Ensures data security and privacy
- Enables scaling to multiple fuel delivery companies

How it works:

- Tenant identifier on all entities
- Doctrine filters for automatic query filtering
- Tenant context set during authentication
- No cross-tenant data access possible

3. Client Management

What it does:

- Maintain database of fuel delivery clients
- Store delivery addresses and contact information
- Track client order history

Why it's important:

- Clients are the revenue source
- Delivery addresses must be accurate
- Historical data helps with route optimization

How it works:

- CRUD operations via REST API
- Clients belong to specific tenant
- Multiple delivery locations per client supported
- Soft delete for historical data preservation

4. Delivery Truck Fleet Management

What it does:

- Inventory of delivery trucks/vehicles
- Track truck capacity, status, and availability
- Associate trucks with specific tenants

Why it's important:

- Essential for dispatch planning
- Capacity constraints affect order fulfillment

- Vehicle maintenance scheduling

How it works:

- Truck entity with capacity (gallons/liters)
- Status tracking: Available, In-Transit, Maintenance, Out-of-Service
- Current location tracking (future enhancement)
- Assignment to orders

5. Order Management System

What it does:

- Create fuel delivery orders
- Specify fuel quantity and delivery location
- Track order lifecycle from creation to completion

Why it's important:

- Core business transaction
- Drives all other operations (dispatch, delivery, billing)
- Audit trail for business intelligence

How it works:

- Order created by authenticated User for a specific Client
- Contains: fuel quantity, delivery address (from Client), requested delivery date
- Order states: Pending, Assigned, In-Transit, Delivered, Cancelled
- Assigned to a Delivery Truck
- Immutable audit log of status changes

User Experience

User Personas

Persona 1: Sarah - Dispatch Manager

- Age: 32, 5 years in fuel delivery operations
- Goals: Efficiently manage daily orders, optimize truck assignments
- Pain Points: Paper-based systems, double-entry, no real-time visibility
- Tech Savvy: Moderate - comfortable with web applications

Persona 2: Mike - System Administrator

- Age: 45, IT manager at mid-size fuel delivery company
- Goals: Maintain system, manage users, ensure data security
- Pain Points: Legacy systems, lack of multi-user support
- Tech Savvy: High - understands APIs and system architecture

Persona 3: Jennifer - Company Owner

- Age: 51, owns small fuel delivery business
- Goals: Visibility into operations, growth without complexity
- Pain Points: Expensive custom software, vendor lock-in

- Tech Savvy: Low to Moderate - needs simple interfaces

Key User Flows

Flow 1: Create a Fuel Delivery Order

1. Dispatcher logs into system (API authentication)
2. Searches for existing client or creates new client
3. Selects client and delivery address
4. Specifies fuel quantity and requested delivery date
5. System validates order (client exists, quantity is reasonable)
6. Order is created in "Pending" state
7. Dispatcher receives confirmation with order ID

Flow 2: Assign Order to Truck

1. Dispatcher views pending orders
2. Checks available trucks (capacity, status)
3. Assigns order to specific truck
4. Order status changes to "Assigned"
5. Driver can view assigned orders (future mobile app)

Flow 3: User Onboarding

1. Admin creates new user account
2. Sets role and permissions
3. User receives credentials
4. User logs in and changes password
5. User can access system based on role

UI/UX Considerations

For MVP (API-focused):

- RESTful API with clear documentation (via API Platform)
- OpenAPI/Swagger specification for frontend developers
- Consistent error messages and status codes
- JSON response format with proper data structures

For Future Frontend:

- Mobile-first responsive design
- Real-time status updates (WebSocket consideration)
- Map integration for delivery locations
- Dashboard with key metrics (orders today, trucks available)
- Quick-action buttons for common tasks
- Search and filter capabilities on all list views

Development Standards & Architecture Principles

Code Quality Principles

All code must adhere to the following principles to ensure maintainability, scalability, and security:

SOLID Principles

Single Responsibility Principle (SRP)

- Each class/service has one reason to change
- Example: `OrderService` handles order business logic, `OrderRepository` handles data access
- Enforced through code review and architectural guidelines

Open/Closed Principle (OCP)

- Classes open for extension, closed for modification
- Use interfaces, events, and inheritance for extensibility
- Example: New order status types shouldn't require modifying existing status transition logic

Liskov Substitution Principle (LSP)

- Subtypes must be substitutable for their base types
- Respect interface contracts in implementations
- Critical for tenant-specific customizations

Interface Segregation Principle (ISP)

- Clients shouldn't depend on interfaces they don't use
- Create focused, specific interfaces
- Example: `OrderAssignmentInterface` separate from `OrderReportingInterface`

Dependency Inversion Principle (DIP)

- Depend on abstractions, not concretions
- Use Symfony's Dependency Injection container
- Example: Services depend on `OrderRepositoryInterface`, not concrete implementation

Clean Architecture

Layer Separation

1. **Presentation Layer** (API Platform Controllers, Serializers)
    - HTTP concerns only (request/response handling)
    - Thin controllers that delegate to Application Layer
    - No business logic in controllers
2. **Application Layer** (Services, Use Cases, DTOs)
    - Orchestrates business operations
    - Handles transactions and validation
    - Example: `CreateOrderService`, `AssignTruckService`
3. **Domain Layer** (Entities, Business Rules, Domain Events)
    - Core business logic and rules
    - Framework-agnostic (no Symfony-specific code)
    - Rich domain models (not anemic entities)
4. **Infrastructure Layer** (Repositories, External APIs, Database)
    - Technical implementations
    - Doctrine repositories, API clients
    - Tenant isolation filters

Dependency Flow: Presentation → Application → Domain ← Infrastructure

Additional Principles

Separation of Concerns (SoC)

- Each module/class addresses a specific concern
- Example: Authentication separate from Authorization separate from Business Logic

Don't Repeat Yourself (DRY)

- Eliminate code duplication
- Extract common patterns into reusable services/traits
- Be pragmatic: Don't over-abstract for 2-3 similar lines

You Aren't Gonna Need It (YAGNI)

- Don't build features "just in case"
- Wait for actual requirements before adding complexity
- Critical for MVP: Build what's needed now, not what might be needed later

Keep It Simple, Silly (KISS)

- Prefer simple solutions over complex ones
- If two approaches work, choose the simpler one
- Complex problems don't always require complex solutions

Implementation Guidelines

Service Layer Pattern

php

```php
// ✅ Good: Business logic in service
class OrderService
{
    public function createOrder(CreateOrderDTO $dto): Order
    {
        $this->validator->validate($dto);

        $order = new Order(
            client: $this->clientRepository->find($dto->clientId),
            quantity: $dto->quantity,
            deliveryLocation: $this->locationRepository->find($dto->locationId)
        );

        $this->eventDispatcher->dispatch(new OrderCreatedEvent($order));

        return $this->orderRepository->save($order);
    }
}

// ❌ Bad: Business logic in controller
class OrderController
{
    public function create(Request $request)
    {
        $order = new Order();
        $order->setClient($clientRepo->find($request->get('client_id')));
        // ... validation, saving, etc in controller
    }
}
```

Rich Domain Models

php

```php
// ✅ Good: Business behavior in entity
class Order
{
    public function assignToTruck(Truck $truck): void
    {
        if (!$this->isPending()) {
            throw new InvalidOrderStateException('Order must be pending to assign truck');
        }

        if (!$truck->hasCapacityFor($this->quantity)) {
            throw new InsufficientCapacityException();
        }

        $this->truck = $truck;
        $this->status = OrderStatus::ASSIGNED;
        $this->recordEvent(new OrderAssignedEvent($this));
    }

    public function isPending(): bool
    {
        return $this->status === OrderStatus::PENDING;
    }
}

// ❌ Bad: Anemic domain model (just getters/setters)
class Order
{
    public function setTruck(?Truck $truck): void
    {
        $this->truck = $truck;
    }

    public function setStatus(string $status): void
    {
        $this->status = $status;
    }
}
```

Repository Pattern

php

```php
// ✅ Good: Custom queries in repository
class OrderRepository extends ServiceEntityRepository
{
    public function findPendingOrdersForClient(Client $client): array
    {
        return $this->createQueryBuilder('o')
            ->where('o.client = :client')
            ->andWhere('o.status = :status')
            ->setParameter('client', $client)
            ->setParameter('status', OrderStatus::PENDING)
            ->getQuery()
            ->getResult();
    }
}

// ❌ Bad: Query logic in controller/service
$orders = $entityManager->createQuery('SELECT o FROM Order o WHERE ...');
```

Event-Driven Architecture

php

```php
// ✅ Good: Decouple side effects with events
class OrderService
{
    public function assignTruck(Order $order, Truck $truck): void
    {
        $order->assignToTruck($truck);
        $this->orderRepository->save($order);

        // Event triggers notifications, logging, etc.
        $this->eventDispatcher->dispatch(new OrderAssignedEvent($order, $truck));
    }
}

// Separate event listener for notifications
class OrderAssignedNotificationListener
{
    public function __invoke(OrderAssignedEvent $event): void
    {
        $this->notificationService->notifyDriver($event->getTruck());
    }
}
```

Pragmatic Application

Balance is Key:

- Apply these principles pragmatically
- Don't over-engineer for simple CRUD operations
- Add complexity only when it solves a real problem
- Refactor when patterns emerge, don't predict them

Code Review Checklist:

- ☐ Does each class have a single responsibility?
- ☐ Are business rules in domain entities/services?
- ☐ Is the code testable without database/framework?
- ☐ Are dependencies injected, not instantiated?
- ☐ Is there unnecessary code duplication?
- ☐ Is this the simplest solution that works?

Technical Debt Management:

- Document intentional shortcuts with TODO/FIXME comments
- Track technical debt in separate backlog
- Allocate 20% of sprint time for refactoring
- Never let technical debt block new features

Technical Architecture

System Components

Backend API (Core MVP)

- **Framework**: Symfony 7.x (latest stable)
- **API Layer**: API Platform 4.x
- **Language**: PHP 8.3+
- **Architecture**: RESTful API with JSON:API or Hydra format

Database

- **Primary DB**: PostgreSQL 16+ (recommended for multi-tenancy)
- **ORM**: Doctrine ORM
- **Migrations**: Doctrine Migrations

Authentication & Authorization

- **Auth Method**: JWT (JSON Web Tokens) via LexikJWTAuthenticationBundle
- **Authorization**: Role-based with Symfony Security
- **Password Hashing**: Symfony PasswordHasher (bcrypt/argon2)

Infrastructure Components

- **Web Server**: Nginx or Apache with PHP-FPM
- **Caching**: Redis (for JWT blacklist, API response caching)
- **Queue System**: Symfony Messenger (for async tasks - future)
- **Logging**: Monolog with structured logging

Data Models

Core Entities

1. Tenant (Company)

- id (UUID)
- name (string)
- subdomain (string, unique)
- status (active/suspended/trial)
- createdAt (datetime)
- updatedAt (datetime)

2. User

- id (UUID)
- tenant (ManyToOne -> Tenant)
- email (string, unique within tenant)
- password (hashed)
- firstName (string)
- lastName (string)
- roles (array - ROLE_ADMIN, ROLE_DISPATCHER, etc.)
- isActive (boolean)
- createdAt (datetime)
- lastLoginAt (datetime, nullable)

3. Client

- id (UUID)
- tenant (ManyToOne -> Tenant)
- companyName (string)
- contactName (string)
- email (string)
- phone (string)
- billingAddress (embedded object)
- deliveryAddresses (OneToMany -> Location)
- isActive (boolean)
- createdAt (datetime)
- updatedAt (datetime)

4. Location (Delivery Address)

- id (UUID)
- tenant (ManyToOne -> Tenant)
- client (ManyToOne -> Client)
- addressLine1 (string)
- addressLine2 (string, nullable)
- city (string)
- state (string)
- postalCode (string)
- country (string)
- latitude (decimal, nullable)
- longitude (decimal, nullable)
- specialInstructions (text, nullable)
- isPrimary (boolean)

5. Truck (Delivery Vehicle)

- id (UUID)
- tenant (ManyToOne -> Tenant)
- truckNumber (string, unique within tenant)
- licensePlate (string)
- capacity (decimal - in gallons/liters)
- status (enum: available, in_transit, maintenance, out_of_service)
- currentLocation (embedded object, nullable)
- lastMaintenanceDate (date, nullable)
- nextMaintenanceDate (date, nullable)
- createdAt (datetime)
- updatedAt (datetime)

6. Order

- id (UUID)
- tenant (ManyToOne -> Tenant)
- orderNumber (string, auto-generated, unique within tenant)
- client (ManyToOne -> Client)
- deliveryLocation (ManyToOne -> Location)
- truck (ManyToOne -> Truck, nullable)
- createdByUser (ManyToOne -> User)
- fuelQuantity (decimal)
- fuelUnit (enum: gallons, liters)
- requestedDeliveryDate (datetime)
- actualDeliveryDate (datetime, nullable)
- status (enum: pending, assigned, in_transit, delivered, cancelled)
- notes (text, nullable)
- createdAt (datetime)
- updatedAt (datetime)
- statusHistory (OneToMany -> OrderStatusHistory)

7. OrderStatusHistory (Audit Log)

- id (UUID)
- order (ManyToOne -> Order)
- changedByUser (ManyToOne -> User)
- fromStatus (enum)
- toStatus (enum)
- notes (text, nullable)
- changedAt (datetime)

API Endpoints Structure

Authentication

- POST `/api/auth/login` - Get JWT token
- POST `/api/auth/refresh` - Refresh token
- POST `/api/auth/logout` - Invalidate token

Tenants (Admin only)

- GET `/api/tenants` - List tenants
- POST `/api/tenants` - Create tenant
- GET `/api/tenants/{id}` - Get tenant details

- PATCH `/api/tenants/{id}` - Update tenant
- DELETE `/api/tenants/{id}` - Soft delete tenant

Users

- GET `/api/users` - List users (filtered by tenant)
- POST `/api/users` - Create user
- GET `/api/users/{id}` - Get user details
- PATCH `/api/users/{id}` - Update user
- DELETE `/api/users/{id}` - Deactivate user

Clients

- GET `/api/clients` - List clients (filtered by tenant)
- POST `/api/clients` - Create client
- GET `/api/clients/{id}` - Get client details
- PATCH `/api/clients/{id}` - Update client
- DELETE `/api/clients/{id}` - Soft delete client

Locations

- GET `/api/locations` - List all locations
- GET `/api/clients/{clientId}/locations` - List client locations
- POST `/api/locations` - Create location
- PATCH `/api/locations/{id}` - Update location
- DELETE `/api/locations/{id}` - Delete location

Trucks

- GET `/api/trucks` - List trucks (filtered by tenant)
- POST `/api/trucks` - Create truck
- GET `/api/trucks/{id}` - Get truck details
- PATCH `/api/trucks/{id}` - Update truck
- GET `/api/trucks/available` - List available trucks
- PATCH `/api/trucks/{id}/status` - Update truck status

Orders

- GET `/api/orders` - List orders (filtered by tenant, paginated)
- POST `/api/orders` - Create order
- GET `/api/orders/{id}` - Get order details
- PATCH `/api/orders/{id}` - Update order
- PATCH `/api/orders/{id}/assign` - Assign truck to order
- PATCH `/api/orders/{id}/status` - Update order status
- GET `/api/orders/{id}/history` - Get order status history
- DELETE `/api/orders/{id}` - Cancel order

Infrastructure Requirements

Development Environment

- Docker Compose setup with: PHP 8.3, PostgreSQL 16, Redis, Nginx
- Symfony CLI for local development

- Git for version control

Production Environment (Minimum)

- **Compute**: 2 vCPU, 4GB RAM (scalable)
- **Database**: PostgreSQL managed service or self-hosted with backup
- **Cache**: Redis instance
- **Storage**: Minimal (primarily database)
- **SSL**: Required for API endpoints
- **Monitoring**: Application logging, error tracking

Security Requirements

- HTTPS only (TLS 1.2+)
- JWT token expiration (15 min access, 7 day refresh)
- Rate limiting on API endpoints
- SQL injection protection (Doctrine parameterized queries)
- CORS configuration for frontend domains
- Input validation and sanitization
- Tenant isolation at query level

Development Roadmap

Phase 1: Foundation & MVP Core (Priority: Critical)

1.1 Project Setup & Infrastructure

- Initialize Symfony 7.x project
- Install and configure API Platform
- Setup Docker development environment
- Configure PostgreSQL database
- Setup Git repository and branching strategy
- Configure code quality tools (PHPStan, CS Fixer)

1.2 Multi-Tenancy Foundation

- Create Tenant entity
- Implement Doctrine filter for tenant isolation
- Create tenant context service
- Add tenant middleware/event subscriber
- Test data isolation thoroughly

1.3 Authentication System

- Create User entity with tenant relation
- Install LexikJWTAuthenticationBundle
- Implement JWT authentication
- Create login/refresh/logout endpoints
- Implement role-based authorization
- Create user management endpoints

1.4 Core Entity Structure

- Create Client entity with CRUD
- Create Location entity with CRUD
- Create Truck entity with CRUD
- Create Order entity with CRUD
- Create OrderStatusHistory entity
- Setup entity relationships and validation

1.5 Basic Order Management

- Implement order creation workflow
- Add order validation (client exists, quantity > 0)
- Implement order listing with filters
- Add order status transitions
- Create status history tracking
- Implement order assignment to trucks

MVP Deliverable: Functional API that allows users to log in, create clients, manage trucks, and create/track orders within their tenant.

Phase 2: Enhanced Functionality (Priority: High)

2.1 Advanced Querying & Filtering

- Implement complex filters on orders (date range, status, client)
- Add search capabilities (clients, orders by number)
- Implement pagination improvements
- Add sorting options on all list endpoints

2.2 Business Logic Enhancements

- Truck capacity validation when assigning orders
- Truck availability check (prevent double-booking)
- Order scheduling conflicts detection
- Automatic order number generation
- Client order history aggregation

2.3 Reporting & Analytics Endpoints

- GET `/api/reports/orders-summary` - Daily/weekly/monthly stats
- GET `/api/reports/truck-utilization` - Truck usage metrics
- GET `/api/reports/client-activity` - Top clients by volume
- Dashboard data endpoint for future frontend

2.4 Audit & Compliance

- Enhanced audit logging for all changes
- User activity tracking
- Export capabilities (CSV) for orders and clients
- Data retention policies

Phase 2 Deliverable: Robust API with advanced querying, better business logic, and basic reporting capabilities.

Phase 3: Optimization & Scalability (Priority: Medium)

3.1 Performance Optimization

- Implement API response caching (Redis)
- Database query optimization and indexing
- Add database connection pooling
- Implement lazy loading where appropriate

3.2 Async Processing

- Setup Symfony Messenger
- Move status change notifications to queue
- Implement background jobs for reports
- Add webhook support for order updates

3.3 Advanced Multi-Tenancy Features

- Tenant-specific configuration
- Custom branding per tenant (metadata)
- Tenant usage metrics and quotas
- Tenant suspension/reactivation workflows

3.4 Location Services Integration

- Geocoding for addresses (Google Maps/OpenStreetMap API)
- Distance calculation between locations
- Route optimization suggestions (future)

Phase 3 Deliverable: Highly optimized API ready for production scale with async capabilities.

Phase 4: Extended Features (Priority: Low/Future)

4.1 Mobile Driver App Support

- Driver entity and authentication
- Driver-specific endpoints
- Real-time location tracking endpoints
- Order completion workflow from driver perspective

4.2 Billing & Invoicing

- Invoice entity and generation
- Payment tracking
- Integration with payment processors
- Automated billing cycles

4.3 Inventory Management

- Fuel inventory tracking
- Depot/warehouse locations
- Stock level monitoring

- Reorder point notifications

4.4 Advanced Analytics

- Machine learning for demand forecasting
- Route optimization algorithms
- Predictive maintenance for trucks
- Customer behavior analytics

Phase 4 Deliverable: Full-featured platform supporting end-to-end fuel delivery operations.

Logical Dependency Chain

Layer 1: Absolute Foundation (Must Build First)

1. **Project Setup** → Without this, nothing else can exist
2. **Tenant Entity & Multi-Tenancy System** → All other entities depend on tenant isolation
3. **User Entity & Authentication** → Required to access any endpoints securely

Rationale: These three are the bedrock. Every other entity will have a tenant relation, and every API call needs authentication.

Layer 2: Core Business Entities (Build in This Order)

4. **Client Entity** → Orders reference clients, so clients must exist first
5. **Location Entity** → Clients have delivery locations, orders reference locations
6. **Truck Entity** → Orders can be assigned to trucks
7. **Order Entity** → The main business transaction (depends on User, Client, Location, Truck)
8. **OrderStatusHistory Entity** → Audit trail for orders (depends on Order)

Rationale: This order respects database foreign key relationships. Each entity builds on previous ones.

Layer 3: Getting to Usable MVP Quickly

9. **Basic CRUD for Client** → Dispatchers need to add clients first
10. **Basic CRUD for Truck** → Need trucks in the system to assign orders
11. **Basic CRUD for Location** → Usually created alongside clients
12. **Order Creation Endpoint** → First meaningful user action
13. **Order Listing & Details** → Users need to see what they created
14. **Order Assignment to Truck** → Makes orders actionable

Milestone: At this point, you have a functional MVP. Users can log in, add clients/trucks, create orders, and assign them. This is testable and demonstrates value.

Layer 4: Essential Business Logic

15. **Order Status Transitions** → Orders need to move through workflow
16. **Status History Tracking** → Automatic audit trail when status changes
17. **Validation Rules** → Prevent invalid data (capacity checks, business rules)
18. **Role-Based Access Control** → Not all users should do everything

Rationale: These make the system production-ready and prevent bad data/operations.

Layer 5: Polish & Discoverability

19. **Filtering & Search** → Find specific orders/clients quickly
20. **Pagination** → Handle large datasets
21. **Tenant Admin Endpoints** → Manage tenants (super admin feature)
22. **Documentation Generation** → API Platform auto-docs, but polish them

Rationale: These improve usability dramatically but aren't blockers for core functionality.

Layer 6: Advanced Features (Post-MVP)

23. **Caching Layer** → Improve performance
24. **Async Processing** → Handle heavy operations without blocking
25. **Reporting Endpoints** → Business intelligence
26. **Location Services** → Geocoding and mapping

Rationale: These are enhancements that significantly improve the product but require the core to be stable first.

Pacing Strategy for Each Feature

Atomic Feature Scoping

- **User Entity**: Build with authentication in one unit (don't separate entity creation from auth setup)
- **CRUD Operations**: For each entity, build Create → Read → Update → Delete as one feature
- **Order Management**: Split into: (1) Create order, (2) List orders, (3) Assign truck, (4) Status transitions
    - This allows testing order creation immediately without waiting for full workflow

Iteration Points

- After completing each layer, deploy to staging and test
- Each layer should be fully functional, not partial implementations
- Don't start Layer N+1 until Layer N is tested and working

Build-Upon Pattern

- **Example**: Order entity starts simple (just client, quantity, date)
- Later iterations add: truck assignment → status workflow → history tracking → advanced validation
- Each iteration is deployable and adds value without breaking existing functionality

---

Risks and Mitigations

Technical Challenges

Risk 1: Multi-Tenancy Data Leakage

Impact: Critical - Tenant A could see Tenant B's data **Probability**: Medium **Mitigation**:

- Implement Doctrine filters globally, not per-query
- Write integration tests that attempt cross-tenant access
- Code review checklist for all entity queries
- Automated testing suite that runs as different tenants

Risk 2: Performance at Scale

Impact: High - Slow API responses hurt UX **Probability**: Medium (as tenants/data grow) **Mitigation**:

- Database indexing strategy from day one (tenant_id + frequently queried columns)
- Implement caching early (Redis for frequent reads)
- Load testing before launch with realistic data volumes
- Plan for database sharding if single tenant grows very large

Risk 3: JWT Token Security

Impact: High - Compromised tokens = unauthorized access **Probability**: Low to Medium **Mitigation**:

- Short-lived access tokens (15 min)
- Refresh token rotation
- Token blacklist in Redis for logout
- HTTPS only, no token in URL parameters
- Rate limiting on auth endpoints

Risk 4: Complex Order State Machine

Impact: Medium - Bugs in status transitions could corrupt data **Probability**: Medium **Mitigation**:

- Use Symfony Workflow component for state management
- Comprehensive unit tests for all state transitions
- Audit log every status change with reason
- Build admin tool to manually correct states if needed

MVP Definition & Scope Management

Risk 5: Feature Creep

Impact: High - Delayed launch, increased cost **Probability**: High (common in new projects) **Mitigation**:

- Strict MVP definition: Authentication + Client CRUD + Truck CRUD + Order CRUD + Assignment
- Defer all "nice-to-have" features to Phase 2+
- Weekly scope review meetings
- "Parking lot" document for future ideas
- Time-box each phase (suggest 2-3 weeks per phase)

Risk 6: Over-Engineering Too Early

Impact: Medium - Wasted development time **Probability**: Medium **Mitigation**:

- Don't build for 1M users on day 1
- Start with simple solutions (direct DB queries, no caching initially)
- Add complexity only when proven necessary
- Measure first (profiling), optimize second

Resource Constraints

Risk 7: Single Developer/Small Team

Impact: High - Knowledge silos, slower development **Probability**: High (common in startups) **Mitigation**:

- Comprehensive documentation from day one
- Code comments for complex business logic
- Git commit messages explaining "why" not just "what"
- Pair programming sessions for critical components
- External code review if budget allows

Risk 8: Changing Requirements

Impact: Medium - Rework, technical debt **Probability**: High (typical in early-stage products) **Mitigation**:

- Modular architecture (easy to swap implementations)
- Avoid tight coupling between entities
- API versioning strategy (/api/v1/...)
- Database migrations (never modify, only add new)
- Feature flags for experimental functionality

Risk 9: Third-Party Dependencies

Impact: Medium - API Platform or Symfony breaking changes **Probability**: Low to Medium **Mitigation**:

- Pin exact versions in composer.json
- Test updates in staging before production
- Subscribe to Symfony/API Platform security advisories
- Budget time for dependency updates quarterly

Data & Compliance

Risk 10: Data Loss

Impact: Critical - Business operations halt **Probability**: Low (but catastrophic) **Mitigation**:

- Automated daily database backups
- Test restore procedures monthly
- Point-in-time recovery capability
- Soft deletes instead of hard deletes where possible
- Audit log all destructive operations

Risk 11: GDPR/Data Privacy (if applicable)

Impact: High - Legal liability **Probability**: Medium (if serving EU customers) **Mitigation**:

- Plan for data export (user requests their data)
- Implement data deletion workflows
- Document data retention policies
- Add consent tracking if needed
- Legal review before launch

Appendix

A. Technology Stack Justification

Why Symfony 7.x?

- Mature, enterprise-grade PHP framework
- Excellent documentation and large community
- Long-term support (LTS) versions available
- Built-in security features
- Doctrine ORM integration
- Strong ecosystem for API development

Why API Platform 4.x?

- Built on top of Symfony specifically for APIs
- Automatic OpenAPI documentation
- JSON:API or Hydra support out of the box
- Built-in pagination, filtering, validation
- Rapid development of REST/GraphQL APIs
- Active development and community

Why PostgreSQL?

- Superior support for multi-tenancy patterns
- JSONB for flexible data storage (future needs)
- Row-level security (RLS) for additional tenant isolation
- Better performance for complex queries vs MySQL
- Strong ACID compliance
- Excellent backup and replication tools

Why JWT over Session-Based Auth?

- Stateless authentication (scalable horizontally)
- Works seamlessly with mobile apps and SPAs
- No server-side session storage needed
- Can include custom claims (tenant_id, roles)
- Industry standard for API authentication

B. Database Indexing Strategy

Critical Indexes for Performance

sql

```sql
-- Tenant isolation (every table)
CREATE INDEX idx_tenant_id ON users(tenant_id);
CREATE INDEX idx_tenant_id ON clients(tenant_id);
CREATE INDEX idx_tenant_id ON trucks(tenant_id);
CREATE INDEX idx_tenant_id ON orders(tenant_id);

-- Frequently queried fields
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_orders_created_at ON orders(created_at DESC);
CREATE INDEX idx_trucks_status ON trucks(status);

-- Composite indexes for common queries
CREATE INDEX idx_orders_tenant_status ON orders(tenant_id, status);
CREATE INDEX idx_orders_tenant_created ON orders(tenant_id, created_at DESC);
CREATE INDEX idx_orders_client ON orders(client_id);

-- Unique constraints
CREATE UNIQUE INDEX idx_users_tenant_email ON users(tenant_id, email);
CREATE UNIQUE INDEX idx_tenants_subdomain ON tenants(subdomain);
```

C. API Response Format Examples

Successful Response (Single Resource)

json

```
{
  "@context": "/api/contexts/Order",
  "@id": "/api/orders/123e4567-e89b-12d3-a456-426614174000",
  "@type": "Order",
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "orderNumber": "ORD-2024-0001",
  "client": "/api/clients/456...",
  "fuelQuantity": 500.00,
  "fuelUnit": "gallons",
  "status": "pending",
  "createdAt": "2024-01-15T10:30:00+00:00"
}
```


Error Response

json


```
{
  "@context": "/api/contexts/Error",
  "@type": "hydra:Error",
  "hydra:title": "An error occurred",
  "hydra:description": "Order quantity must be greater than zero",
  "violations": [
    {
      "propertyPath": "fuelQuantity",
      "message": "This value should be greater than 0."
    }
  ]
}
```


D. Testing Strategy

Unit Tests

- Entity validation logic
- Business rules (capacity checks, status transitions)
- Custom services and utilities
- Target: 80%+ code coverage

Integration Tests

- API endpoint functionality
- Database interactions
- Multi-tenancy isolation
- Authentication flows

End-to-End Tests

- Complete user workflows (login → create order → assign truck)
- Cross-entity operations
- Error handling scenarios

Performance Tests

- Load testing with realistic data volumes
- Concurrent request handling
- Database query performance
- Target: <200ms for GET requests, <500ms for POST/PATCH

E. Environment Variables (Example .env)

```
APP_ENV=prod
APP_SECRET=your-secret-key

DATABASE_URL="postgresql://user:pass@localhost:5432/fuel_delivery?serverVersion=16"

JWT_SECRET_KEY=%kernel.project_dir%/config/jwt/private.pem
JWT_PUBLIC_KEY=%kernel.project_dir%/config/jwt/public.pem
JWT_PASSPHRASE=your-passphrase
JWT_TTL=900

REDIS_URL=redis://localhost:6379

CORS_ALLOW_ORIGIN='^https?://(localhost|127\.0\.0\.1|yourdomain\.com)(:[0-9]+)?

# Optional
SENTRY_DSN=your-sentry-dsn
GOOGLE_MAPS_API_KEY=your-api-key
```

F. Deployment Checklist

- [ ] Environment variables configured
- [ ] Database migrations run
- [ ] JWT keys generated and secured
- [ ] SSL certificate installed
- [ ] CORS origins configured
- [ ] Rate limiting enabled
- [ ] Logging configured (Sentry, CloudWatch, etc.)
- [ ] Database backups scheduled
- [ ] Health check endpoint (/api/health) responding
- [ ] API documentation accessible (/api/docs)
- [ ] Monitoring alerts configured
- [ ] First tenant created
- [ ] First admin user created

G. Future Considerations (Beyond Initial Scope)

- **GraphQL Support**: API Platform supports GraphQL natively
- **Real-time Updates**: WebSocket support for order status changes
- **Mobile SDKs**: Generate client SDKs for iOS/Android from OpenAPI spec
- **Webhook System**: Notify external systems of order events
- **Multi-language Support**: i18n for API responses
- **Advanced Reporting**: Business intelligence dashboard
- **Machine Learning**: Demand forecasting, route optimization
- **IoT Integration**: Truck telemetry, fuel level sensors

Document Version: 1.0
Last Updated: November 10, 2025
Status: Draft - Ready for Technical Review