FlowEngine 5.0

Complete Project Explanation

Enterprise Email Management System

Multi-Tenant Architecture Documentation

Table of Contents

1. Project Overview

FlowEngine 5.0 is an **Enterprise Email Management System** with a **multi-tenant architecture**. This means multiple organizations (tenants) can use the same application, but their data is completely isolated from each other.

What Does This System Do?

The system manages:

• **Intents**: Email categorization/routing rules

• **Validation Rules**: Data validation logic with datasource lookups

• **Datasources**: External data connections (databases, APIs, files)

• **Tenants**: Organization/client management

Key Feature: Multi-Tenancy

Every piece of data in the system is associated with a `tenant_id`. This ensures:

• Company A cannot see Company B's data

• Each tenant has isolated configurations

• Shared application, separated data

2. Technology Stack

Backend

• **FastAPI**: Modern Python web framework (async support, automatic API docs)

• **SQLAlchemy**: ORM (Object-Relational Mapping) for database operations

• **PostgreSQL**: Relational database

• **Pydantic**: Data validation using Python type hints

• **Uvicorn**: ASGI server to run the application

Frontend

• **Vanilla JavaScript**: No frameworks, pure JS

• **HTML5/CSS3**: Modern responsive design

• **Font Awesome**: Icon library

• **Session Storage**: Browser-based tenant context management

3. Architecture Overview

The project follows a **Clean Architecture** pattern with clear separation of concerns:

```
FlowEngine5.0/ ■■■ backend/ # Python backend ■ ■■■ core/ # Core infrastructure ■ ■■■
modules/ # Business logic modules ■ ■■■ common/ # Shared utilities ■ ■■■ static/ # Frontend
files ■■■ .env # Environment configuration ■■■ requirements.txt # Python dependencies
```

Architectural Layers

Presentation Layer (Frontend HTML/JS)

API Layer (FastAPI Routes)

Service Layer (Business Logic)

Repository Layer (Database Access)

Data Layer (SQLAlchemy Models)

4. Core Components Deep Dive

4.1 Configuration Management (`core/config.py`)

```
class Settings(BaseSettings): DATABASE_URL: str APP_NAME: str = "FlowEngine" APP_VERSION: str
= "1.0.0" DEBUG: bool = True ENVIRONMENT: str = "development"
```

What's Happening:

• Uses `pydantic-settings` to load configuration from environment variables

• Reads from `.env` file automatically

• Type validation ensures DATABASE_URL is a string, DEBUG is boolean, etc.

• Creates a global `settings` instance used throughout the app

Why This Matters:

• No hardcoded values (security)

• Easy to change configuration per environment (dev/staging/prod)

• Type safety prevents configuration errors

4.2 Database Setup (`core/database.py`)

```
engine = create_engine( settings.DATABASE_URL, pool_pre_ping=True, pool_size=5,
max_overflow=10 )
```

Connection Pooling Explained:

• `pool_size=5`: Keep 5 persistent database connections open

• `max_overflow=10`: Can create 10 additional connections if needed

• `pool_pre_ping=True`: Check if connection is alive before using it

• **Why?** Database connections are expensive to create. Pooling reuses them.

```
@event.listens_for(engine, "connect") def create_schema(dbapi_conn, connection_record): cursor
= dbapi_conn.cursor() cursor.execute("CREATE SCHEMA IF NOT EXISTS eivs") cursor.close()
```

Schema Auto-Creation:

• **Event Listener**: Runs every time a new database connection is made

• Creates `eivs` schema if it doesn't exist

• All tables will be created in this schema (not the public schema)

• **Why?** Organizes database objects, prevents naming conflicts

```
SessionLocal = sessionmaker( autocommit=False, autoflush=False, bind=engine )
```

Session Factory:

• `autocommit=False`: Changes must be explicitly committed

• `autoflush=False`: Don't automatically sync with database

• Creates sessions that manage transactions

4.3 Dependency Injection (`core/dependencies.py`)

```
def get_db() -> Generator[Session, None, None]: db = SessionLocal() try: yield db finally:
db.close()
```

How This Works:

FastAPI calls `get_db()` when an endpoint needs a database session

Creates a new session from `SessionLocal`

`yield db` gives the session to the endpoint

After the endpoint finishes, `finally` block closes the session

This ensures connections are always cleaned up, even if errors occur

Usage in Routes:

```
@router.get("/intents") def get_intents(db: Session = Depends(get_db)): # db is automatically
injected here return db.query(Intent).all()
```

4.4 Exception Handling (`common/exceptions.py`)

```
class ResourceNotFoundError(HTTPException): def __init__(self, detail: str): super().__init__( status_code=status.HTTP_404_NOT_FOUND, detail=detail )
```

Custom Exceptions:

• Inherits from `HTTPException` (FastAPI's base exception)

• Sets appropriate HTTP status codes:

• `404 NOT FOUND` for missing resources

• `400 BAD REQUEST` for validation errors

• Allows consistent error responses across the API

Usage:

```
if not intent: raise ResourceNotFoundError(f"Intent {intent_id} not found")
```

This automatically returns:

```
{ "detail": "Intent 123 not found" }
```

with HTTP 404 status.

5. Multi-Tenancy Implementation

The Core Concept

Every database table has a `tenant_id` column. All queries MUST filter by this ID.

How Tenant Context is Managed

Backend: Query String Parameter

```
@router.get("/api/intents") def get_intents( tenant_id: str = Query(...), # Required query parameter db: Session = Depends(get_db) ): service = IntentService(db) return service.get_all(tenant_id)
```

The Flow:

Frontend makes request: `GET /api/intents?tenant_id=acme_corp`

FastAPI validates `tenant_id` is present

Passes it to the service layer

Service filters: `WHERE tenant_id = 'acme_corp'`

Frontend: Session Storage

```
// When switching tenants sessionStorage.setItem('selectedTenantId', tenantId); // When making
API calls const tenantId = sessionStorage.getItem('selectedTenantId');
fetch(`/api/intents?tenant_id=${tenantId}`)
```

Why Session Storage?

• Persists across page refreshes

• Cleared when tab closes (security)

• Specific to each browser tab (can work with multiple tenants in different tabs)

Database-Level Multi-Tenancy

Unique Constraints Per Tenant

```
UniqueConstraint( "tenant_id", "name", name="uq_eivs_intents_tenant_name" )
```

What This Does:

• Tenant A can have an intent named "Sales"

• Tenant B can also have an intent named "Sales"

• But Tenant A cannot create TWO intents named "Sales"

• Enforced at the database level (not just application code)

Indexes for Performance

```
Index("idx_eivs_intents_tenant", "tenant_id")
```

Why Index tenant_id?

• Every query filters by tenant_id

• Without an index: Database scans entire table

• With an index: Database jumps directly to tenant's rows

• **Performance difference:** Milliseconds vs. seconds for large datasets

6. Database Architecture

Schema: `eivs`

All tables are created in the `eivs` schema, not the default `public` schema.

Advantages:

• Logical grouping of related tables

• Prevents conflicts with other applications using the same database

• Cleaner namespace management

Table Relationships

```
Datasources (1) ■■■■■■■< (Many) Validation Rules (Many) >■■■■■■■ (1) Intents ■ ■■ config_key
■■■> (1) Datasource Configs
```

Relationships Explained:

Datasource → Validation Rules (One-to-Many)

• One datasource can be used by multiple validation rules

• Cascade delete: Deleting a datasource deletes its validation rules

Intent → Validation Rules (One-to-Many)

• One intent can have multiple validation rules

• Cascade delete: Deleting an intent deletes its validation rules

Datasource → Datasource Config (Many-to-One via connection_key)

• Multiple datasources can share the same configuration

• Not a foreign key, just a reference by string

Key Tables Explained

1. **Tenants Table**

```
class Tenant(Base): tenant_id: str # Primary key, e.g., "acme_corp" name: str # Display name,
e.g., "Acme Corporation" is_active: bool
```

Purpose:

• Master list of all tenants in the system

• Controls which tenants can access the application

• Does NOT have a `tenant_id` foreign key (it IS the tenant definition)

2. **Intents Table**

```
class Intent(Base): intent_id: int # Auto-increment ID tenant_id: str # Which tenant owns this
name: str # "Sales Inquiry" description: str # Optional details is_active: bool created_at:
datetime updated_at: datetime
```

Purpose:

• Defines categories for email classification

• Examples: "Support Ticket", "Sales Lead", "Complaint"

Key Constraints:

• `UniqueConstraint("tenant_id", "name")` → Each tenant has unique intent names

• `Index("tenant_id")` → Fast queries per tenant

3. **Datasources Table**

```
class Datasource(Base): datasource_id: int # Auto-increment ID tenant_id: str name: str #
"Customer Database" datasource_type: str # "database" | "api" | "file" connection_key: str #
References DatasourceConfig.name description: str is_active: bool
```

Purpose:

• Defines WHERE to get validation data from

• Examples: Customer database, CRM API, Excel file

Connection Pattern:

```
Datasource.connection_key → DatasourceConfig.name "prod_db" → "prod_db" config
```

4. **Datasource Configs Table**

```
class DatasourceConfig(Base): config_id: int tenant_id: str name: str # "prod_db" protocol:
str # "postgresql" | "http" | "file" driver_family: str # "psycopg2" | "rest" | "csv"
base_url: str # Connection string or URL auth_type: str # "basic" | "oauth" | "api_key"
auth_config: dict # JSON: {"username": "...", "password": "..."} connection_json: dict #
Additional connection parameters
```

Purpose:

• Stores HOW to connect to external systems

• Separates connection details from datasource definition

• Allows reusing configs across multiple datasources

Security Note:

• In production, `auth_config` should be encrypted

• Passwords should NEVER be stored in plain text

5. **Validation Rules Table**

```
class ValidationRule(Base): rule_id: int tenant_id: str intent_id: int # Which intent is this
for? datasource_id: int # Which datasource to query? field_name: str # "email" rule_type: str
# "exists" | "equals" | "regex" conditions: dict # JSON: query conditions priority: int #
Execution order is_active: bool
```

Purpose:

• Defines validation logic using external data

• Example: "Check if sender email exists in customer database"

How It Works:

Email arrives with intent "Support Ticket"

Find all validation rules for this intent

For each rule:

• Get the datasource

• Build a query using `conditions`

• Execute query

• Validate result based on `rule_type`

Example Rule:

```
{ "field_name": "sender_email", "rule_type": "exists", "datasource_id": 5, "conditions": {
"query": "SELECT * FROM customers WHERE email = ?", "params": ["{{sender_email}}"] } }
```

7. Module-by-Module Breakdown

7.1 Tenants Module

Models (`tenants/models.py`)

```
class Tenant(Base): __tablename__ = "tenants" __table_args__ = {"schema": "eivs"} tenant_id:
str = Column(Text, primary_key=True) name: str = Column(Text, nullable=False, unique=True)
is_active: bool = Column(Boolean, default=True)
```

Schemas (`tenants/schemas.py`)

```
class TenantResponse(BaseModel): tenant_id: str name: str is_active: bool class Config:
from_attributes = True # Allows SQLAlchemy models → Pydantic
```

Pydantic Schemas Purpose:

• **Validation**: Ensures API requests have correct data types

• **Documentation**: Auto-generates OpenAPI/Swagger docs

• **Serialization**: Converts database models to JSON

Repository (`tenants/repository.py`)

```
class TenantRepository: def get_all(self) -> List[Tenant]: return
self.db.query(Tenant).order_by(Tenant.tenant_id).all() def get_by_id(self, tenant_id: str) ->
Optional[Tenant]: return self.db.query(Tenant).filter( Tenant.tenant_id == tenant_id ).first()
```

Repository Pattern Benefits:

• Encapsulates all database queries

• Single place to change query logic

• Easy to mock for testing

Service (`tenants/service.py`)

```
class TenantService: def get_all(self, active_only: bool = False) -> List[Tenant]: tenants =
self.repository.get_all() if active_only: tenants = [t for t in tenants if t.is_active] return
tenants def create(self, payload: TenantCreate) -> Tenant: # Check if already exists if
self.repository.get_by_id(payload.tenant_id): raise ResourceAlreadyExistsError(...) return
self.repository.create(payload)
```

Service Layer Benefits:

• Business logic separate from HTTP and database

• Handles validation, error checking

• Can combine multiple repository calls in one transaction

Routes (`tenants/routes.py`)

```
@router.get("/api/tenants", response_model=List[TenantResponse]) def get_all_tenants(
active_only: bool = Query(False), db: Session = Depends(get_db) ): service = TenantService(db)
return service.get_all(active_only)
```

FastAPI Route Breakdown:

• `@router.get(...)` → HTTP GET method

• `response_model=List[TenantResponse]` → Auto-converts to JSON, validates output

• `active_only: bool = Query(False)` → Optional query parameter with default

• `db: Session = Depends(get_db)` → Injects database session

7.2 Intents Module

The **most complex module** due to validation rule relationships.

Key Model Relationship

```
class Intent(Base): validation_rules = relationship( "ValidationRule",
back_populates="intent", cascade="all, delete-orphan" )
```

Cascade Delete Explained:

• `cascade="all, delete-orphan"`: When intent is deleted, delete all its validation rules

• `back_populates="intent"`: Two-way relationship (can access from both sides)

Why This Matters:

```
# Can access validation rules from intent intent = db.query(Intent).first()
print(intent.validation_rules) # List of ValidationRule objects # Can access intent from
validation rule rule = db.query(ValidationRule).first() print(rule.intent) # Intent object
```

Service Complex Logic

```
def create_with_rules( self, tenant_id: str, payload: IntentCreateWithRules ) -> Intent: # 1.
Create intent intent = self.repository.create(tenant_id, payload.intent) # 2. Create
validation rules if payload.validation_rules: for rule_data in payload.validation_rules:
rule_data.intent_id = intent.intent_id self.validation_service.create(tenant_id, rule_data)
return intent
```

Transaction Safety:

• All operations happen in one database session

• If validation rule creation fails, intent creation is rolled back

• Ensures data consistency

7.3 Validation Rules Module

Complex Query Logic

```
def get_by_intent( self, tenant_id: str, intent_id: int, active_only: bool = False ) ->
List[ValidationRule]: query = self.db.query(ValidationRule).join(Intent).filter(
ValidationRule.tenant_id == tenant_id, ValidationRule.intent_id == intent_id ) if active_only:
query = query.filter(ValidationRule.is_active == True) return
query.order_by(ValidationRule.priority).all()
```

SQL Generated (roughly):

```
SELECT validation_rules.* FROM eivs.validation_rules JOIN eivs.intents ON
validation_rules.intent_id = intents.intent_id WHERE validation_rules.tenant_id = 'acme_corp'
AND validation_rules.intent_id = 5 AND validation_rules.is_active = TRUE ORDER BY
validation_rules.priority;
```

Why Join Intent?

• Ensures the intent belongs to the same tenant

• Prevents accessing validation rules from other tenants' intents

7.4 Datasources Module

Two Models, Two Repositories

Datasource: The logical reference

DatasourceConfig: The physical connection details

```
# Service coordinates both class DatasourceService: def __init__(self, db: Session):
self.datasource_repo = DatasourceRepository(db) self.config_repo =
DatasourceConfigRepository(db) def get_datasource_with_config( self, tenant_id: str,
```

```
datasource_id: int ): datasource = self.datasource_repo.get_by_id(tenant_id, datasource_id) if not datasource: raise ResourceNotFoundError(...) config = self.config_repo.get_by_name( tenant_id, datasource.connection_key ) return { "datasource": datasource, "config": config }
```

8. Frontend Architecture

Session-Based Tenant Management

Tenant Selector (`select_tenant.html`)

```
function switchTenant(tenantId) { sessionStorage.setItem('selectedTenantId', tenantId); window.location.href = '/static/index.html'; }
```

Flow:

User lands on `/static/select_tenant.html`

Fetches tenant list from API

User clicks a tenant

Saves `tenant_id` to session storage

Redirects to dashboard

All subsequent pages read `tenant_id` from session storage

Dashboard (`index.html`)

```
// Check if tenant is selected const tenantId = sessionStorage.getItem('selectedTenantId'); if (!tenantId) { window.location.href = '/static/select_tenant.html'; return; } // Load dashboard data loadStats(); loadRecentData();
```

Protection Against Missing Tenant:

• Every page checks for `selectedTenantId`

• If missing, redirects to tenant selector

• Prevents API errors from missing `tenant_id` parameter

CRUD Pattern (Example: Intents)

List View

```
async function loadIntents() { const tenantId = sessionStorage.getItem('selectedTenantId');
const response = await fetch(`/api/intents?tenant_id=${tenantId}`); const intents = await
response.json(); renderIntentsTable(intents); }
```

Create/Edit Modal

```
function showIntentModal(intentId = null) { const modal =
document.getElementById('intentModal'); if (intentId) { // Edit mode: fetch and populate
fetchIntent(intentId).then(intent => { document.getElementById('name').value = intent.name; //
... populate other fields }); } else { // Create mode: clear form
document.getElementById('intentForm').reset(); } modal.style.display = 'block'; }
```

Form Submission

```
async function saveIntent() { const tenantId = sessionStorage.getItem('selectedTenantId');
const intentId = document.getElementById('intentId').value; const data = { name:
document.getElementById('name').value, description:
document.getElementById('description').value }; const method = intentId ? 'PUT' : 'POST';
const url = intentId ? `/api/intents/${intentId}?tenant_id=${tenantId}` :
`/api/intents?tenant_id=${tenantId}`; const response = await fetch(url, { method, headers: {
'Content-Type': 'application/json' }, body: JSON.stringify(data) }); if (response.ok) {
closeModal(); loadIntents(); // Refresh list } }
```

Responsive Design Strategy

Desktop-First Approach

Base styles assume desktop:

```
.sidebar { width: 250px; position: fixed; left: 0; } .main-content { margin-left: 250px; }
```

Mobile Breakpoint

```
@media (max-width: 768px) { .sidebar { transform: translateX(-100%); /* Hide off-screen */
transition: transform 0.3s; } .sidebar.active { transform: translateX(0); /* Slide in */ }
.main-content { margin-left: 0; /* Full width */ } }
```

Mobile Menu Toggle

```
function toggleMenu() { const sidebar = document.querySelector('.sidebar');
sidebar.classList.toggle('active'); const overlay =
document.querySelector('.sidebar-overlay'); overlay.classList.toggle('active'); }
```

How It Works:

On mobile, sidebar is hidden by default (`translateX(-100%)`)

Hamburger button toggles `.active` class

CSS transform slides sidebar into view

Overlay appears to cover content

Clicking overlay closes menu

9. How Everything Works Together

Complete Request Flow

Let's trace a request to create an intent with validation rules:

Step 1: Frontend Form Submission

```
// User fills form and clicks "Save" const formData = { intent: { name: "Support Ticket", description: "Customer support inquiries" }, validation_rules: [ { field_name: "sender_email", rule_type: "exists", datasource_id: 3, conditions: { /* ... */ } } ] }; const tenantId = sessionStorage.getItem('selectedTenantId'); fetch(`/api/intents/with-rules?tenant_id=${tenantId}`, { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify(formData) });
```

Step 2: FastAPI Route Handler

```
@router.post("/api/intents/with-rules") def create_intent_with_rules( payload: IntentCreateWithRules, # Pydantic validates this tenant_id: str = Query(...), db: Session = Depends(get_db) ): service = IntentService(db) return service.create_with_rules(tenant_id, payload)
```

What Happens Here:

FastAPI receives HTTP POST request

Parses JSON body into `IntentCreateWithRules` schema

Pydantic validates all fields match expected types

Extracts `tenant_id` from query string

Gets database session from dependency

Passes to service layer

Step 3: Service Layer (Business Logic)

```
def create_with_rules(self, tenant_id: str, payload: IntentCreateWithRules): # Check for duplicates existing = self.repository.get_by_name(tenant_id, payload.intent.name) if existing: raise ResourceAlreadyExistsError( f"Intent '{payload.intent.name}' already exists" ) # Create intent intent = self.repository.create(tenant_id, payload.intent) # Create validation rules if payload.validation_rules: for rule_data in payload.validation_rules: rule_data.intent_id = intent.intent_id rule_data.tenant_id = tenant_id # Validate datasource exists datasource = self.datasource_service.get_by_id( tenant_id, rule_data.datasource_id ) if not datasource: raise ValidationException("Invalid datasource") self.validation_service.create(tenant_id, rule_data) return intent
```

Business Rules Enforced:

• No duplicate intent names per tenant

• All validation rules must reference valid datasources

• Transaction ensures atomicity (all or nothing)

Step 4: Repository Layer (Database Access)

```
def create(self, tenant_id: str, payload: IntentCreate): data = payload.model_dump() #
Pydantic → dict data['tenant_id'] = tenant_id # Inject tenant_id obj = Intent(**data) # Create
SQLAlchemy model self.db.add(obj) self.db.commit() # Save to database self.db.refresh(obj) #
Reload to get generated ID return obj
```

SQL Generated:

```
INSERT INTO eivs.intents (tenant_id, name, description, is_active, created_at, updated_at)
VALUES ('acme_corp', 'Support Ticket', 'Customer support inquiries', TRUE, NOW(), NOW())
RETURNING intent_id;
```

Step 5: Response Back to Frontend

```
# Service returns Intent object # FastAPI converts to JSON using response_model { "intent_id":
42, "tenant_id": "acme_corp", "name": "Support Ticket", "description": "Customer support
inquiries", "is_active": true, "created_at": "2024-01-15T10:30:00", "updated_at":
"2024-01-15T10:30:00" }
```

Step 6: Frontend Handles Response

```
const response = await fetch(...); if (response.ok) { const intent = await response.json(); //
Close modal closeModal(); // Refresh table loadIntents(); // Show success message
showToast('Intent created successfully'); } else { const error = await response.json();
alert(error.detail); // Show error message }
```

Multi-Tenant Data Isolation

Scenario: Two Tenants Using the System

Tenant A (acme_corp):

• Has intents: "Sales", "Support"

• Has datasources: "Customer DB", "CRM"

Tenant B (globex_inc):

• Has intents: "Sales", "Billing"

• Has datasources: "ERP System"

Database State:

```
-- Intents table intent_id | tenant_id | name ---------|-------------|---------- 1 | acme_corp | Sales 2 | acme_corp | Support 3 | globex_inc | Sales 4 | globex_inc | Billing
```

Query Isolation:

```
# Acme Corp requests intents service.get_all("acme_corp") # Returns: [{"intent_id": 1, "name": "Sales"}, {"intent_id": 2, "name": "Support"}] # Globex Inc requests intents service.get_all("globex_inc") # Returns: [{"intent_id": 3, "name": "Sales"}, {"intent_id": 4, "name": "Billing"}]
```

SQL Executed:

```
SELECT * FROM eivs.intents WHERE tenant_id = 'acme_corp' -- Only sees their data ORDER BY intent_id DESC;
```

10. Security & Best Practices

Current Implementation

1. **Multi-Tenancy Enforcement**

■ All database queries filter by `tenant_id`

■ Unique constraints include `tenant_id`

■ Indexes on `tenant_id` for performance

2. **Input Validation**

■ Pydantic schemas validate all API inputs

■ Type checking (string, int, bool, etc.)

■ Required vs. optional fields enforced

3. **Error Handling**

■ Custom exceptions with appropriate HTTP status codes

■ Database sessions properly closed in `finally` blocks

■ Cascade deletes prevent orphaned records

4. **Database Best Practices**

■ Connection pooling for efficiency

■ Schema organization (`eivs` schema)

■ Proper indexes for query performance

Security Gaps & Production Recommendations

■■ **Authentication & Authorization**

Current State: None

Risk: Anyone can access any tenant's data by changing `tenant_id` parameter

Recommended Fix:

```
# Add JWT authentication from fastapi_jwt_auth import AuthJWT @router.get("/api/intents") def get_intents( Authorize: AuthJWT = Depends(), db: Session = Depends(get_db) ): Authorize.jwt_required() # Get tenant_id from JWT token, not query parameter tenant_id = Authorize.get_jwt_subject() service = IntentService(db) return service.get_all(tenant_id)
```

■■ **Password Storage**

Current State: `DatasourceConfig.auth_config` stores credentials as plain JSON

Risk: Database breach exposes all credentials

Recommended Fix:

```
from cryptography.fernet import Fernet def encrypt_auth_config(auth_config: dict) -> str: key = settings.ENCRYPTION_KEY # Store securely! f = Fernet(key) json_str = json.dumps(auth_config) return f.encrypt(json_str.encode()).decode() def decrypt_auth_config(encrypted: str) -> dict: key = settings.ENCRYPTION_KEY f = Fernet(key) json_str = f.decrypt(encrypted.encode()).decode() return json.loads(json_str)
```

■■ **SQL Injection**

Current State: SQLAlchemy ORM prevents most SQL injection

Good Practice: Never use raw SQL with string concatenation

Safe (using ORM):

```
query = db.query(Intent).filter(Intent.name == user_input)
```

Unsafe (raw SQL):

```
# ■ DON'T DO THIS query = f"SELECT * FROM intents WHERE name = '{user_input}'"
```

■■ **CORS Configuration**

Current State: `allow_origins=["*"]` allows all domains

Risk: Any website can call your API

Production Fix:

```
app.add_middleware( CORSMiddleware, allow_origins=["https://yourdomain.com"], # Specific domain only allow_credentials=True, allow_methods=["GET", "POST", "PUT", "DELETE"], allow_headers=["*"], )
```

■■ **Environment Variables**

Current State: `.env` file in repository

Risk: Accidental commit exposes secrets

Best Practice:

Add `.env` to `.gitignore` ■ (already done)

Use secret management in production (AWS Secrets Manager, Vault)

Never commit `.env` to version control

Summary: The Big Picture

What This System Does

Multi-tenant email management with complete data isolation

Intent classification for routing emails

Validation rules that query external data sources

Datasource management for connecting to databases, APIs, files

Architecture Layers

Frontend (HTML/JS) → User interface, session management

Routes (FastAPI) → HTTP endpoints, request/response handling

Services → Business logic, validation, orchestration

Repositories → Database queries, CRUD operations

Models → SQLAlchemy ORM, relationships, constraints

Database (PostgreSQL) → Data persistence, multi-tenant storage

Key Technologies

• **FastAPI**: Modern async Python web framework

• **SQLAlchemy**: ORM for database operations

• **Pydantic**: Data validation and serialization

• **PostgreSQL**: Relational database with schema support

• **Session Storage**: Browser-based tenant context

Multi-Tenancy Strategy

• `tenant_id` in every table

• Session storage for frontend tenant context

• Query parameter for backend tenant filtering

• Database-level unique constraints per tenant

• Indexes for performance

Data Flow

User selects tenant → Saved to session storage

Frontend makes API call → Includes `tenant_id` parameter

FastAPI route receives request → Validates input

Service layer applies business logic → Checks rules

Repository queries database → Filters by `tenant_id`

Database returns results → Only that tenant's data

FastAPI serializes response → Converts to JSON

Frontend renders data → Updates UI

Next Steps for Production

Add Authentication (JWT tokens)

Encrypt Sensitive Data (credentials, passwords)

Implement Rate Limiting (prevent API abuse)

Add Logging (audit trail, debugging)

Set Up Monitoring (error tracking, performance)

Write Tests (unit tests, integration tests)

Database Migrations (Alembic for schema changes)

CI/CD Pipeline (automated testing, deployment)

Load Balancing (handle multiple servers)

Backup Strategy (database backups, disaster recovery)

This system is a solid foundation for a multi-tenant SaaS application with proper separation of concerns, clean architecture, and scalable design patterns.