CARDINSA Insurance Backend - Foundation Layer Technical Documentation

Version: 1.0.0

Date: January 2025

Architecture: Clean Architecture with Domain-Driven Design principles

Technology Stack: FastAPI, SQLAlchemy 2.0, PostgreSQL, Pydantic v2, JWT Authentication

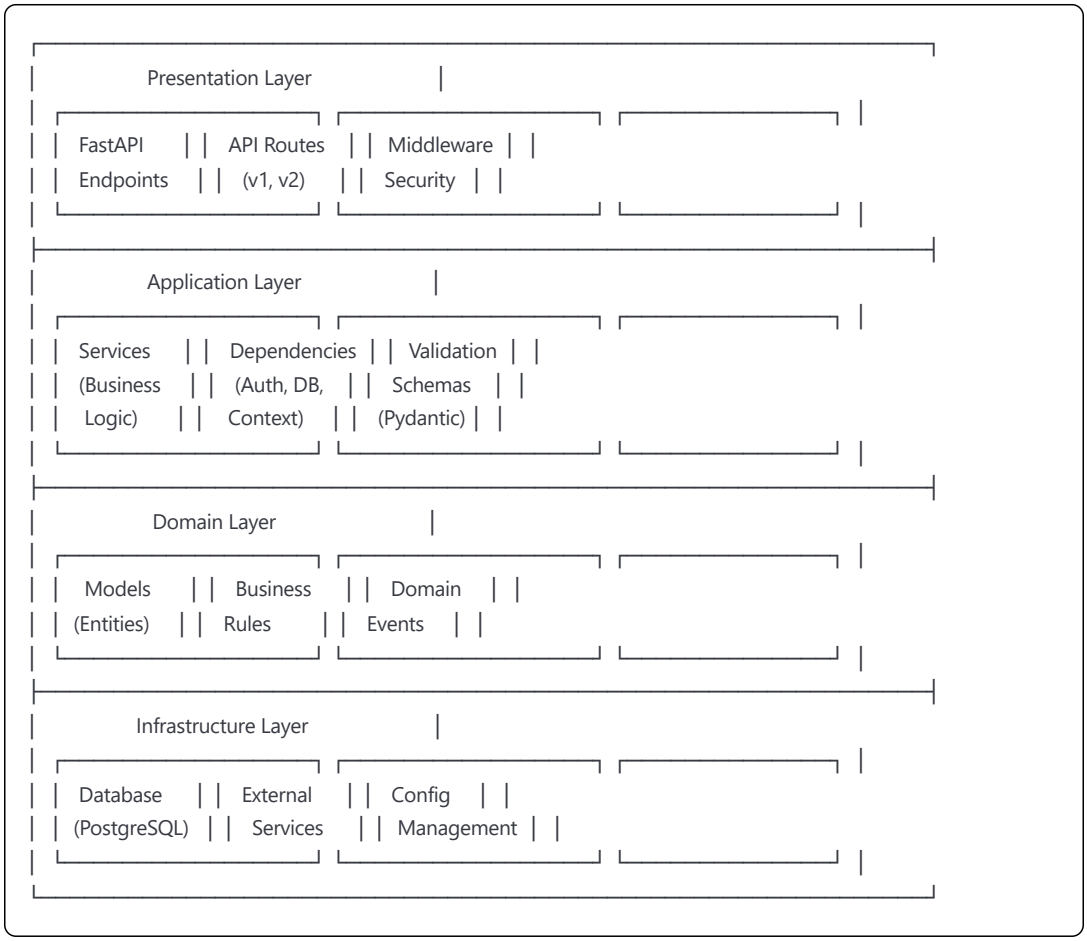
---

Table of Contents

Architecture Overview

Clean Architecture Implementation

The CARDINSA backend implements Clean Architecture principles with clear separation of concerns:



Key Architectural Decisions

1. **Async-First Design**: Built on asyncio for high-performance I/O operations
2. **Multi-Tenant Architecture**: Company-level data isolation from ground up
3. **Domain-Driven Design**: Clear entity boundaries and business logic encapsulation
4. **CQRS Patterns**: Separation of read/write models where appropriate
5. **Enterprise Security**: JWT, RBAC, audit trails, and data protection

---

Foundation Layer Structure

```
cardinsa-backend/
├── main.py                    # Application entry point
├── app/
│   ├── __init__.py            # [F1] App package initializer
│   ├── config/
│   │   ├── __init__.py        # [F2] Config package initializer
│   │   ├── settings.py        # [F3] Application configuration
│   │   ├── database.py        # [F4] Database connection management
│   │   └── logging.py         # [F5] Logging configuration
│   ├── core/
│   │   ├── __init__.py        # [F6] Core package initializer
│   │   ├── exceptions.py      # [F7] Custom exception hierarchy
│   │   ├── middleware.py      # [F8] HTTP middleware components
│   │   ├── dependencies.py    # [F9] FastAPI dependency injection
│   │   └── security.py        # [F10] Security utilities
│   ├── models/
│   │   ├── __init__.py        # Package initializer
```

```
│   │   └── base.py              # [F11] Base model classes
│   ├── schemas/
│   │   ├── __init__.py          # Package initializer
│   │   └── base.py              # [F12] Base Pydantic schemas
│   ├── services/
│   │   ├── __init__.py          # Package initializer
│   │   └── base.py              # [F13] Base service classes
│   └── api/
│       ├── __init__.py          # [F14] API package initializer
│       └── v1/
│           ├── __init__.py      # [F15] V1 API initializer
│           └── router.py        # [F16] Main V1 router
└── .env                         # Environment configuration
```

Legend: [F#] = Foundation file number

File-by-File Technical Analysis

F1: `app/__init__.py` - Application Package

Purpose: Main application package initialization
Size: 15 lines
Dependencies: None

```python
"""
CARDINSA Insurance Backend Application Package
"""
__version__ = "1.0.0"
__title__ = "CARDINSA Insurance Backend"
```

Technical Details:

- Provides package metadata and version information
- Establishes the main application namespace
- Required for Python to treat `app` as a package
- Contains basic application constants

F2: `app/config/__init__.py` - Configuration Package

Purpose: Configuration module initialization and exports
Size: 18 lines
Dependencies: settings.py, database.py, logging.py

```python
from .settings import get_settings
from .database import create_tables
from .logging import setup_logging
```

Technical Details:

- Centralizes configuration imports for easy access

- Provides clean API for application startup
- Enables `from app.config import get_settings`
- Maintains separation of concerns between config modules

F3: `app/config/settings.py` - Application Settings

Purpose: Centralized configuration management using Pydantic BaseSettings
Size: 165 lines
Dependencies: Pydantic, functools

Key Features:

- **Environment-based configuration** with `.env` file support
- **Pydantic v1/v2 compatibility** for future-proofing
- **Type validation** for all configuration values
- **Property methods** for complex transformations
- **Caching** with `@lru_cache()` for performance

Configuration Categories:

```python
# Application Settings
APP_NAME: str = "Cardinsa Insurance API"
ENVIRONMENT: str = "development"
DEBUG: bool = True

# Database Settings
DATABASE_URL: str = "postgresql://..."
DATABASE_POOL_SIZE: int = 10

# Security Settings
SECRET_KEY: str = "..."
JWT_SECRET_KEY: Optional[str] = None
ACCESS_TOKEN_EXPIRE_MINUTES: int = 30

# CORS & API Settings
CORS_ORIGINS: str = "http://localhost:3000,http://localhost:8081"
API_V1_PREFIX: str = "/api/v1"
```

Design Patterns:

- **Settings Pattern**: Centralized configuration management
- **Validation Pattern**: Type checking and constraint validation
- **Factory Pattern**: `get_settings()` with caching

Security Considerations:

- Environment variables for sensitive data
- Default values for development ease
- Validation to prevent misconfigurations

F4: `app/config/database.py` - Database Management

Purpose: Database connection, session management, and lifecycle
Size: 210 lines
Dependencies: SQLAlchemy 2.0, asyncio, settings

Key Features:

- **Dual session support**: Sync and async database operations
- **Connection pooling**: Optimized for PostgreSQL production use
- **Health checking**: Database connectivity monitoring
- **Lifecycle management**: Startup/shutdown handling

Technical Architecture:

```python
# Engine Configuration
engine = create_engine(       # Synchronous engine
    DATABASE_URL,
    pool_size=10,
    max_overflow=20,
    pool_pre_ping=True
)

async_engine = create_async_engine(  # Asynchronous engine
    database_url_async,
    pool_size=10,
    max_overflow=20,
    pool_pre_ping=True
)

# Session Factories
SessionLocal = sessionmaker(bind=engine)
AsyncSessionLocal = async_sessionmaker(async_engine)
```

Connection Management:

- **PostgreSQL optimized**: Pool settings for production workloads
- **Connection validation**: `pool_pre_ping=True` for reliability
- **Resource cleanup**: Proper session disposal
- **Error handling**: Comprehensive exception management

Design Patterns:

- **Factory Pattern**: Session creation and management
- **Context Manager Pattern**: `get_db_context()` for transactions
- **Repository Pattern**: Foundation for data access

F5: `app/config/logging.py` - Logging System

Purpose: Comprehensive logging configuration and utilities
Size: 190 lines
Dependencies: Python logging, pathlib, settings

Logging Architecture:

```python
# Multiple handlers for different log levels
handlers = {
    "console": StreamHandler,      # Development output
    "file": RotatingFileHandler,   # Application logs
    "error_file": RotatingFileHandler,  # Error-specific logs
    "access_file": RotatingFileHandler  # HTTP access logs
}

# Structured formatters
formatters = {
    "default": Standard format,
    "detailed": Extended format with module/function,
    "json": JSONFormatter for structured logging
}
```

Key Features:

- **Multiple log destinations**: Console, files, error-specific
- **Log rotation**: Prevents disk space issues
- **Structured logging**: JSON format for log aggregation
- **Performance logging**: Request timing and metrics
- **Security logging**: Authentication and authorization events

Enterprise Features:

- **Log levels per module**: Fine-grained control
- **Request correlation**: Track requests across services
- **Audit trails**: Security-focused logging
- **External integration ready**: Compatible with ELK stack

F6: `app/core/__init__.py` - Core Package

Purpose: Core functionality package with clean exports
Size: 30 lines
Dependencies: exceptions.py, middleware.py

```python
from .exceptions import (
    BusinessLogicException,
    ValidationException,
    AuthenticationException,
    AuthorizationException,
    ResourceNotFoundException,
)
from .middleware import (
    LoggingMiddleware,
    RateLimitMiddleware,
    SecurityHeadersMiddleware,
)
```

Design Philosophy:

- **Clean imports**: Easy access to core functionality
- **Explicit exports**: `__all__` for controlled API
- **Logical grouping**: Related functionality together

F7: `app/core/exceptions.py` - Exception Hierarchy

Purpose: Comprehensive custom exception system for business logic
Size: 350 lines
Dependencies: datetime, uuid, typing

Exception Hierarchy:

```
BaseCustomException
├── BusinessLogicException      # Business rule violations
├── ValidationException         # Input validation errors
├── AuthenticationException     # Authentication failures
├── AuthorizationException      # Permission/access denials
├── ResourceNotFoundException   # Entity not found errors
├── DatabaseException           # Database operation failures
├── ExternalServiceException    # Third-party service errors
└── RateLimitException          # Rate limiting violations
```

Key Features:

- **Rich exception data**: Error codes, timestamps, details
- **Structured error info**: Machine-readable error responses
- **Exception context**: Request IDs, user info, resource details
- **Factory methods**: Convenient exception creation
- **API-ready format**: Direct JSON serialization

Design Patterns:

- **Exception Hierarchy**: Clear inheritance structure
- **Factory Pattern**: Helper functions for common scenarios
- **Builder Pattern**: Flexible exception construction
- **Strategy Pattern**: Different exception types for different scenarios

Enterprise Features:

- **Request correlation**: Track errors across requests
- **Audit compliance**: Detailed error logging
- **User-friendly messages**: Safe error information exposure
- **Debug information**: Detailed context for development

F8: `app/core/middleware.py` - HTTP Middleware Stack

Purpose: HTTP request/response processing pipeline
Size: 280 lines
Dependencies: FastAPI, datetime, collections, uuid

Middleware Components:

LoggingMiddleware

- **Request/response logging** with timing metrics
- **Unique request IDs** for correlation
- **User context tracking** from authentication
- **Performance monitoring** with response times

RateLimitMiddleware

- **Per-IP rate limiting** with configurable windows
- **Per-user rate limiting** for authenticated requests
- **Memory-based storage** (Redis-ready architecture)
- **Graceful degradation** with retry headers

SecurityHeadersMiddleware

- **Security headers injection**: X-Frame-Options, X-Content-Type-Options
- **HSTS enforcement** for HTTPS connections
- **CSP headers** for documentation pages
- **XSS protection** headers

Technical Architecture:

```python
class LoggingMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        request_id = str(uuid4())
        start_time = time.time()

        # Process request...
        response = await call_next(request)

        # Log metrics...
        process_time = time.time() - start_time
        return response
```

Performance Considerations:

- **Async processing**: Non-blocking middleware operations
- **Memory management**: Cleanup of rate limiting data
- **Minimal overhead**: Efficient request processing

F9: `app/core/dependencies.py` - Dependency Injection System

Purpose: FastAPI dependency injection for cross-cutting concerns
Size: 320 lines
Dependencies: FastAPI, SQLAlchemy, typing, security

Dependency Categories:

Database Dependencies

```python
```

```python
async def get_async_db_session() -> AsyncGenerator[AsyncSession, None]:
    # Provides database session to endpoints

def get_sync_db() -> Generator[Session, None, None]:
    # Synchronous database access when needed
```

Authentication Dependencies

```python
async def get_current_user_optional() -> Optional[dict]:
    # Optional authentication (returns None if not logged in)

async def get_current_user() -> dict:
    # Required authentication (raises exception if not logged in)
```

Authorization Dependencies

```python
def require_permissions(*permissions: str):
    # Factory for permission-based access control

def require_roles(*roles: str):
    # Factory for role-based access control
```

Query Parameter Dependencies

```python
async def get_pagination_params() -> PaginationParams:
    # Standardized pagination across all endpoints

async def get_filter_params() -> FilterParams:
    # Consistent filtering parameters

async def get_sort_params() -> SortParams:
    # Uniform sorting parameters
```

Multi-Tenancy Support:

```python
async def get_current_company_id() -> uuid.UUID:
    # Company isolation for multi-tenant architecture

async def get_current_company_context() -> dict:
    # Full company context with settings
```

Design Patterns:

- **Dependency Injection**: FastAPI's dependency system
- **Factory Pattern**: Permission/role requirement factories
- **Context Pattern**: Request context management
- **Strategy Pattern**: Different auth strategies (optional/required)

Security Features:

- **JWT token validation** with proper error handling
- **Permission checking** with detailed error messages
- **Multi-tenant isolation** at the dependency level
- **Request context** for audit trails

F10: `app/core/security.py` - Security Utilities

Purpose: Comprehensive security functionality for authentication and authorization
Size: 420 lines
Dependencies: bcrypt, jose, passlib, secrets, hashlib

Security Modules:

PasswordSecurity

```python
class PasswordSecurity:
    @staticmethod
    def hash_password(password: str) -> str:
        # Bcrypt hashing with automatic salt generation

    @staticmethod
    def verify_password(plain: str, hashed: str) -> bool:
        # Constant-time password verification

    @staticmethod
    def validate_password_strength(password: str) -> Dict[str, Any]:
        # Comprehensive password policy enforcement

    @staticmethod
    def generate_secure_password(length: int = 12) -> str:
        # Cryptographically secure password generation
```

JWTTokenManager

```python
class JWTTokenManager:
    @staticmethod
    def create_access_token(data: Dict[str, Any]) -> str:
        # Short-lived access tokens (30 minutes default)

    @staticmethod
    def create_refresh_token(data: Dict[str, Any]) -> str:
        # Long-lived refresh tokens (7 days default)

    @staticmethod
    def decode_access_token(token: str) -> Dict[str, Any]:
        # Token validation with expiry checking

    @staticmethod
    def create_token_pair(user_data: Dict[str, Any]) -> Dict[str, str]:
        # Complete token set for authentication flow
```

PermissionManager

```python
class PermissionManager:
    @staticmethod
    def check_permissions(user_perms: List[str], required: List[str]) -> bool:
        # Granular permission checking

    @staticmethod
    def get_role_permissions(role: str) -> List[str]:
        # Role-to-permission mapping
```

Security Standards:

- **Bcrypt password hashing**: Industry standard with salt
- **JWT with RS256/HS256**: Secure token implementation
- **Timing attack prevention**: Constant-time comparisons
- **Cryptographically secure randomness**: Using `secrets` module

Enterprise Security Features:

- **Password policies**: Complexity requirements and validation
- **Token management**: Access/refresh token patterns
- **API key support**: For external service integration
- **Role-based permissions**: Scalable authorization system

F11: `app/models/base.py` - Base Model Classes

Purpose: SQLAlchemy ORM base classes with enterprise patterns
Size: 290 lines
Dependencies: SQLAlchemy 2.0, uuid, datetime

Model Hierarchy:

```
Base (SQLAlchemy declarative_base)
├── TimestampMixin       # created_at, updated_at
├── AuditMixin           # created_by, updated_by
├── SoftDeleteMixin      # archived_at, restore/archive methods
├── BaseModel            # Combines all mixins + UUID primary key
├── CompanyIsolatedModel # BaseModel + company_id for multi-tenancy
├── VersionedModel       # BaseModel + version tracking
└── MetadataModel        # BaseModel + JSON metadata field
```

Key Features:

Universal Patterns

```python
```

```python
class BaseModel(Base, TimestampMixin, AuditMixin, SoftDeleteMixin):
    __abstract__ = True

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)

    @declared_attr
    def __tablename__(cls):
        # Automatic table naming: UserProfile -> user_profiles

    def to_dict(self) -> dict:
        # JSON serialization with UUID/datetime handling

    def update_from_dict(self, data: dict) -> None:
        # Bulk updates with field exclusion
```

Multi-Tenancy Support

```python
class CompanyIsolatedModel(BaseModel):
    __abstract__ = True

    company_id = Column(UUID(as_uuid=True), nullable=False)

    @declared_attr
    def company(cls):
        return relationship("Company", back_populates=f"{cls.__tablename__}")
```

Design Patterns:

- **Mixin Pattern**: Composable functionality
- **Template Method**: Standardized model behavior
- **Active Record**: Model methods for common operations
- **Strategy Pattern**: Different model types for different needs

Enterprise Features:

- **Audit trails**: Complete change tracking
- **Soft delete**: Data preservation with recovery
- **Multi-tenancy**: Company-level data isolation
- **Versioning support**: Document/policy versioning
- **Flexible metadata**: JSON attributes for dynamic fields

F12: `app/schemas/base.py` - Base Pydantic Schemas

Purpose: API request/response validation and serialization patterns
Size: 380 lines
Dependencies: Pydantic v2, typing, enum, datetime

Schema Architecture:

```
BaseSchema              # Complete entity (for responses)
├── TimestampSchema     # Timestamp fields
├── AuditSchema         # Audit trail fields
├── SoftDeleteSchema    # Soft delete fields
└── Specialized Schemas:
    ├── BaseCreateSchema    # Create operations (input validation)
    ├── BaseUpdateSchema    # Update operations (partial updates)
    ├── CompanyIsolatedSchema  # Multi-tenant entities
    ├── VersionedSchema     # Versioned entities
    └── MetadataSchema      # Entities with JSON metadata
```

API Consistency Schemas:

```python
class PaginationParams(BaseModel):
    page: int = Field(1, ge=1)
    size: int = Field(20, ge=1, le=100)

class PaginatedResponse(BaseModel, Generic[T]):
    items: List[T]
    pagination: PaginationInfo

class ErrorResponse(BaseModel):
    success: bool = False
    error: str
    message: str
    details: Optional[List[ErrorDetail]] = None
```

Key Features:

- **Pydantic v1/v2 compatibility**: Future-proof configuration
- **Field validation**: Comprehensive input validation
- **Automatic documentation**: OpenAPI schema generation
- **Type safety**: Full type hints for IDE support
- **JSON serialization**: Proper UUID/datetime handling

API Patterns:

- **Generic responses**: Type-safe pagination and collections
- **Consistent errors**: Standardized error response format
- **Field examples**: Auto-generated API documentation
- **Validation rules**: Business rule enforcement at API boundary

---

F13: `app/services/base.py` - Base Service Classes

Purpose: Business logic layer with enterprise patterns
Size: 450 lines
Dependencies: SQLAlchemy 2.0, abc, typing, logging

Service Architecture:

```python
```

```python
class BaseService(ABC, Generic[ModelType, CreateSchemaType, UpdateSchemaType, SchemaType]):
    """
    Abstract base service providing:
    - CRUD operations
    - Transaction management
    - Validation hooks
    - Audit logging
    - Error handling
    """
```

Core Operations:

```python
async def create(self, obj_in: CreateSchemaType) -> ModelType:
    # Create with validation, audit trails, and transaction management

async def get(self, id: uuid.UUID) -> Optional[ModelType]:
    # Retrieve with company filtering and archive handling

async def list(self, pagination, filters, sort) -> PaginatedResponse[SchemaType]:
    # List with pagination, filtering, sorting, and company isolation

async def update(self, id: uuid.UUID, obj_in: UpdateSchemaType) -> ModelType:
    # Update with validation and audit trails

async def delete(self, id: uuid.UUID, soft_delete: bool = True) -> bool:
    # Soft or hard delete with validation
```

Validation Hooks:

```python
async def _validate_create(self, create_data, current_user_id) -> None:
    # Override for custom creation validation

async def _validate_update(self, db_obj, update_data, current_user_id) -> None:
    # Override for custom update validation

async def _validate_delete(self, db_obj, current_user_id) -> None:
    # Override for custom deletion validation
```

Multi-Tenancy Service:

```python
class CompanyIsolatedService(BaseService):
    """
    Service for multi-tenant entities with automatic company filtering
    """
    def __init__(self, model, db_session, company_id: uuid.UUID):
        # Automatic company isolation for all operations
```

Design Patterns:

- **Generic Service Pattern**: Type-safe CRUD operations
- **Template Method**: Standardized operation flow with customization hooks

- **Unit of Work**: Transaction management across operations
- **Repository Pattern**: Data access abstraction (via SQLAlchemy)
- **Command Pattern**: Encapsulated business operations

Enterprise Features:

- **Transaction management**: Automatic commit/rollback
- **Audit logging**: Comprehensive operation tracking
- **Validation framework**: Business rule enforcement
- **Error handling**: Proper exception management
- **Multi-tenancy**: Company-level data isolation

F14-F16: API Layer Foundation

Purpose: FastAPI application structure and routing
Combined Size: 180 lines
Dependencies: FastAPI, datetime

API Structure:

```python
# F14: app/api/__init__.py
API_VERSION = "1.0.0"
API_TITLE = "CARDINSA Insurance Backend API"

# F15: app/api/v1/__init__.py
V1_VERSION = "1.0.0"
V1_PREFIX = "/api/v1"

# F16: app/api/v1/router.py
api_v1_router = APIRouter(
    responses={
        404: {"description": "Not found"},
        422: {"description": "Validation error"},
        500: {"description": "Internal server error"}
    }
)
```

Foundation Endpoints:

- `GET /api/v1/` - API information and available endpoints
- `GET /api/v1/version` - Version and feature information
- `GET /api/v1/status` - System health and performance metrics

Router Architecture:

- **Modular design**: Separate routers for each domain
- **Version management**: Clear API versioning strategy
- **Documentation ready**: OpenAPI metadata included
- **Extension points**: Ready for domain-specific routers

Design Patterns and Principles

SOLID Principles Implementation

Single Responsibility Principle (SRP)

- **Each class has one reason to change**
- `PasswordSecurity` only handles password operations
- `JWTTokenManager` only handles JWT operations
- `BaseService` only handles business logic patterns

Open/Closed Principle (OCP)

- **Open for extension, closed for modification**
- `BaseService` provides hooks for customization
- `BaseModel` mixins can be composed differently
- Middleware stack is extensible

Liskov Substitution Principle (LSP)

- **Derived classes are substitutable for base classes**
- All service classes can be used wherever `BaseService` is expected
- Model inheritance hierarchy maintains contracts
- Schema inheritance preserves validation behavior

Interface Segregation Principle (ISP)

- **Clients depend only on interfaces they use**
- Separate mixins for different concerns (Audit, Timestamps, SoftDelete)
- Optional dependencies in dependency injection
- Focused exception types for specific error scenarios

Dependency Inversion Principle (DIP)

- **Depend on abstractions, not concretions**
- Services depend on abstract database sessions
- Configuration through dependency injection
- Abstract base classes define contracts

Domain-Driven Design Patterns

Entities and Value Objects

```python
# Entity: Has identity and lifecycle
class BaseModel:  # All business entities inherit from this
    id: UUID  # Identity
    created_at: datetime  # Lifecycle

# Value Objects: Immutable, defined by their attributes
class PaginationParams(BaseModel):  # Immutable query parameters
    page: int
    size: int
```

Repositories (via Services)

```python
class BaseService:  # Acts as repository + business logic
    async def get(self, id) -> Optional[ModelType]:  # Repository pattern
    async def create(self, obj_in) -> ModelType:    # With business logic
```

Domain Services

```python
class PasswordSecurity:  # Domain service for password operations
class PermissionManager:  # Domain service for authorization
```

Enterprise Patterns

Unit of Work Pattern

```python
async def create(self, obj_in):
    try:
        # Business operations
        db_obj = self.model(**create_data)
        self.db.add(db_obj)
        await self.db.commit()  # Unit of work boundary
    except Exception:
        await self.db.rollback()  # Automatic rollback
        raise
```

Specification Pattern (via Filters)

```python
class FilterParams:  # Specifications for queries
    search: Optional[str]
    created_after: Optional[datetime]
    include_archived: bool
```

Factory Pattern

```python
@lru_cache()
def get_settings() -> Settings:  # Settings factory
    return Settings()

def require_permissions(*perms):  # Dependency factory
    async def check_permissions():...
    return check_permissions
```

Security Architecture

Authentication Flow

```mermaid
```

Security Layers

Layer 1: Transport Security

- **HTTPS enforcement** via SecurityHeadersMiddleware
- **HSTS headers** for secure connections
- **Secure cookie settings** for session management

Layer 2: Authentication

- **JWT access tokens** (30-minute lifetime)
- **JWT refresh tokens** (7-day lifetime)
- **API keys** for service-to-service communication
- **Password hashing** with bcrypt

Layer 3: Authorization

- **Role-based access control** (RBAC)
- **Permission-based access control** (fine-grained)
- **Multi-tenant isolation** (company-level)
- **Resource-level permissions**

Layer 4: Input Validation

- **Pydantic schema validation** at API boundary
- **SQL injection prevention** via SQLAlchemy ORM
- **XSS prevention** via output encoding
- **CSRF protection** via security headers

Layer 5: Audit and Monitoring

- **Request logging** with correlation IDs
- **Audit trails** for all data modifications
- **Security event logging** for authentication/authorization
- **Rate limiting** to prevent abuse

Security Standards Compliance

- **OWASP Top 10** mitigation strategies
- **GDPR compliance** via audit trails and data protection
- **SOX compliance** via comprehensive logging
- **Industry best practices** for password policy and JWT handling

---

Database Architecture

Data Architecture Principles

Multi-Tenancy Design

```sql
-- Every business entity includes company_id for isolation
CREATE TABLE policies (
    id UUID PRIMARY KEY,
    company_id UUID NOT NULL REFERENCES companies(id),
    -- other fields
);

-- Row-level security can be implemented
CREATE POLICY company_isolation ON policies
USING (company_id = current_setting('app.current_company_id')::UUID);
```

Audit Trail Design

```sql
-- Every table includes audit fields
CREATE TABLE base_auditable (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE,
    created_by UUID REFERENCES users(id),
    updated_by UUID REFERENCES users(id),
    archived_at TIMESTAMP WITH TIME ZONE  -- Soft delete
);
```

Performance Optimization

```sql
-- Indexes for common query patterns
CREATE INDEX idx_company_entities ON policies (company_id, created_at);
CREATE INDEX idx_active_entities ON policies (archived_at) WHERE archived_at IS NULL;
CREATE INDEX idx_user_audit ON policies (created_by, updated_by);

-- Partitioning strategy for large tables (future)
CREATE TABLE audit_logs_2025_01 PARTITION OF audit_logs
FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');
```

Data Integrity Constraints

```sql
```

```
-- Foreign key relationships with cascade rules
ALTER TABLE policies
ADD CONSTRAINT fk_policy_company
FOREIGN KEY (company_id) REFERENCES companies(id) ON DELETE RESTRICT;

-- Check constraints for business rules
ALTER TABLE users
ADD CONSTRAINT chk_email_format
CHECK (email ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,});
```

Database Connection Architecture

Connection Pool Configuration

```
python

# Production-optimized PostgreSQL settings
engine = create_engine(
    DATABASE_URL,
    pool_size=10,            # Base connections
    max_overflow=20,         # Additional connections under load
    pool_timeout=30,         # Wait time for connection
    pool_recycle=3600,       # Recycle connections hourly
    pool_pre_ping=True       # Validate connections before use
)
```

Session Management Patterns

```
python

# Async context manager pattern
async with get_db_context() as db:
    # All operations in single transaction
    user = await user_service.create(user_data)
    await audit_service.log_creation(user.id)
    # Automatic commit or rollback
```

Migration Strategy

- **Alembic integration** for schema versioning
- **Backward compatible** migrations
- **Data migration scripts** for business logic changes
- **Rollback procedures** for production deployments

API Architecture

RESTful Design Principles

Resource-Oriented URLs

```
python
```

```
# Standard REST patterns
GET    /api/v1/users           # List users
POST   /api/v1/users           # Create user
GET    /api/v1/users/{id}      # Get specific user
PUT    /api/v1/users/{id}      # Update user (full)
PATCH  /api/v1/users/{id}      # Update user (partial)
DELETE /api/v1/users/{id}      # Delete user

# Nested resources
GET    /api/v1/companies/{id}/policies    # Company's policies
POST   /api/v1/policies/{id}/claims       # Create claim for policy
```

Standard HTTP Status Codes

python

```
# Success responses
200 OK          # Successful GET, PUT, PATCH
201 Created     # Successful POST
204 No Content  # Successful DELETE

# Client error responses
400 Bad Request      # Invalid input data
401 Unauthorized     # Authentication required
403 Forbidden        # Insufficient permissions
404 Not Found        # Resource doesn't exist
422 Unprocessable    # Validation errors

# Server error responses
500 Internal Error   # Unexpected server error
503 Service Unavail  # Temporary unavailability
```

API Consistency Patterns

Standardized Response Format

python

```
# Success response
{
  "success": true,
  "message": "Operation completed successfully",
  "data": {...},
  "timestamp": "2025-01-01T12:00:00Z"
}

# Error response
{
  "success": false,
  "error": "Validation Error",
  "message": "Request validation failed",
  "details": [
    {
      "field": "email",
      "message": "Invalid email format",
      "code": "INVALID_FORMAT"
    }
  ],
  "timestamp": "2025-01-01T12:00:00Z",
  "request_id": "req_123456789"
}
```

Pagination Pattern

```
python

# Request
GET /api/v1/users?page=1&size=20&sort_by=created_at&sort_order=desc

# Response
{
  "items": [...],
  "pagination": {
    "page": 1,
    "size": 20,
    "total": 150,
    "pages": 8,
    "has_next": true,
    "has_prev": false
  }
}
```

Filtering and Searching

```
python

# Query parameters
GET /api/v1/users?search=john&created_after=2025-01-01&include_archived=false

# Custom filters per endpoint
GET /api/v1/policies?status=active&product_type=health&expires_before=2025-12-31
```

API Documentation Strategy

OpenAPI/Swagger Integration

```python
# Automatic schema generation
class UserResponse(BaseSchema):
    email: str = Field(..., example="user@example.com", description="User email address")
    name: str = Field(..., example="John Doe", description="Full name")

    class Config:
        schema_extra = {
            "example": {
                "id": "550e8400-e29b-41d4-a716-446655440000",
                "email": "john.doe@example.com",
                "name": "John Doe",
                "created_at": "2025-01-01T12:00:00Z"
            }
        }
```

API Versioning Strategy

- **URL versioning**: `/api/v1/`, `/api/v2/`
- **Backward compatibility**: Maintain older versions during transition
- **Deprecation timeline**: Clear communication for version sunsets
- **Feature flags**: Gradual rollout of new functionality

Development Patterns

Code Organization Principles

Package Structure Philosophy

```
app/
├── config/     # Infrastructure concerns (DB, logging, settings)
├── core/       # Cross-cutting concerns (auth, exceptions, middleware)
├── models/     # Domain entities (business objects)
├── schemas/    # API contracts (validation, serialization)
├── services/   # Business logic (use cases, workflows)
├── api/        # Presentation layer (HTTP endpoints)
└── utils/      # Shared utilities
```

Naming Conventions

```python
```

```
# Classes: PascalCase
class UserService(BaseService):
class AuthenticationException(BaseCustomException):

# Functions: snake_case
async def get_current_user():
def validate_password_strength():

# Constants: UPPER_SNAKE_CASE
ACCESS_TOKEN_EXPIRE_MINUTES = 30
DATABASE_URL = "postgresql://..."

# Files: snake_case.py
user_service.py
auth_exceptions.py
```

Testing Strategy Foundation

Test Structure (Ready for Implementation)

```
tests/
├── unit/          # Isolated component tests
│   ├── models/    # Model behavior tests
│   ├── services/  # Business logic tests
│   ├── schemas/   # Validation tests
│   └── utils/     # Utility function tests
├── integration/   # Multi-component tests
│   ├── api/       # API endpoint tests
│   ├── database/  # Database integration tests
│   └── auth/      # Authentication flow tests
└── e2e/           # End-to-end workflow tests
    ├── user_flows/ # Complete user scenarios
    └── api_flows/  # API workflow tests
```

Testing Patterns (Foundation Ready)

```
python

# Service layer testing
@pytest.fixture
async def user_service(db_session):
    return UserService(User, db_session)

async def test_create_user(user_service):
    user_data = UserCreate(email="test@example.com")
    user = await user_service.create(user_data)
    assert user.email == "test@example.com"
    assert user.id is not None

# API testing with dependency overrides
def test_get_users_endpoint(client, mock_db):
    response = client.get("/api/v1/users/")
    assert response.status_code == 200
    assert "items" in response.json()
    assert "pagination" in response.json()
```

Error Handling Strategy

Exception Propagation Pattern

```
API Layer (FastAPI)
    ↓ Catches and converts exceptions to HTTP responses
Business Layer (Services)
    ↓ Raises business-specific exceptions
Data Layer (Models/Database)
    ↓ Raises data-specific exceptions
Infrastructure Layer
```

Error Response Strategy

```python
# Development environment: Detailed errors
{
  "error": "ValidationException",
  "message": "Email format is invalid",
  "details": {"field": "email", "value": "invalid-email"},
  "stack_trace": "..." # Only in development
}

# Production environment: Safe errors
{
  "error": "Validation Error",
  "message": "Request validation failed",
  "details": [{"field": "email", "message": "Invalid format"}],
  "request_id": "req_123456"  # For support tracking
}
```

Performance Patterns

Database Query Optimization

```python
# Eager loading for related data
query = select(User).options(joinedload(User.company))

# Pagination with efficient counting
total_query = select(func.count()).select_from(base_query.subquery())
items_query = base_query.limit(size).offset(offset)

# Index usage optimization
query = query.filter(User.company_id == company_id)  # Uses index
```

Caching Strategy (Foundation Ready)

```python
```

```python
# Service-level caching patterns
@lru_cache(maxsize=128)
def get_role_permissions(role: str) -> List[str]:
    return role_permission_mapping[role]

# Async caching for expensive operations (Redis integration ready)
async def get_cached_user_context(user_id: uuid.UUID) -> dict:
    # Cache user context for performance
    pass
```

Future Extension Guidelines

Adding New Business Modules

Step-by-Step Module Creation

1. **Model Definition** - Inherit from `BaseModel` or `CompanyIsolatedModel`

```python
class Policy(CompanyIsolatedModel):
    __tablename__ = "policies"

    policy_number: str = Column(String, unique=True, nullable=False)
    product_type: str = Column(String, nullable=False)
    # Automatic: id, company_id, timestamps, audit fields, soft delete
```

2. **Schema Definition** - Inherit from base schemas

```python
class PolicyResponse(CompanyIsolatedSchema):
    policy_number: str
    product_type: str
    # Automatic: id, company_id, timestamps, audit fields

class PolicyCreate(BaseCreateSchema):
    policy_number: str
    product_type: str
    # No id, timestamps, or audit fields

class PolicyUpdate(BaseUpdateSchema):
    policy_number: Optional[str] = None
    product_type: Optional[str] = None
```

3. **Service Implementation** - Inherit from `BaseService`

```python
```

```python
class PolicyService(CompanyIsolatedService[Policy, PolicyCreate, PolicyUpdate, PolicyResponse]):
    entity_name = "Policy"

    # Get all CRUD operations automatically

    async def _validate_create(self, create_data, current_user_id):
        # Custom validation logic
        if not create_data.get("policy_number"):
            raise ValidationException("Policy number is required")

    async def get_by_policy_number(self, policy_number: str) -> Optional[Policy]:
        # Custom business methods
        query = select(self.model).filter(
            and_(
                self.model.policy_number == policy_number,
                self.model.company_id == self.company_id
            )
        )
        result = await self.db.execute(query)
        return result.scalar_one_or_none()
```

4. **API Endpoints** - Use dependency injection

```python
router = APIRouter(prefix="/policies", tags=["Policies"])

@router.get("/", response_model=PaginatedResponse[PolicyResponse])
async def list_policies(
    current_user: dict = Depends(get_current_user),
    company_id: uuid.UUID = Depends(get_current_company_id),
    pagination: PaginationParams = Depends(get_pagination_params),
    filters: FilterParams = Depends(get_filter_params),
    db: AsyncSession = Depends(get_async_db_session)
):
    policy_service = PolicyService(Policy, db, company_id)
    return await policy_service.list(pagination, filters)

@router.post("/", response_model=PolicyResponse, status_code=201)
async def create_policy(
    policy_data: PolicyCreate,
    current_user: dict = Depends(get_current_user),
    company_id: uuid.UUID = Depends(get_current_company_id),
    db: AsyncSession = Depends(get_async_db_session)
):
    policy_service = PolicyService(Policy, db, company_id)
    return await policy_service.create(policy_data, current_user["id"])
```

5. **Router Registration** - Add to main router

```python
```

```python
# In app/api/v1/router.py
from .policies.policies import router as policies_router

api_v1_router.include_router(
    policies_router,
    prefix="/policies",
    tags=["Policies"]
)
```

Extending Base Functionality

Adding Custom Mixins

```python
class GeolocationMixin:
    """Mixin for entities with geographic coordinates"""
    latitude = Column(Numeric(10, 8), nullable=True)
    longitude = Column(Numeric(11, 8), nullable=True)
    address = Column(Text, nullable=True)

    @property
    def coordinates(self) -> Optional[Tuple[float, float]]:
        if self.latitude and self.longitude:
            return (float(self.latitude), float(self.longitude))
        return None

# Usage
class ClaimLocation(BaseModel, GeolocationMixin):
    claim_id = Column(UUID, ForeignKey("claims.id"))
    description = Column(Text)
```

Custom Service Patterns

```python
class AuditableService(BaseService):
    """Service with enhanced audit logging"""

    async def _post_create_actions(self, db_obj, current_user_id):
        await super()._post_create_actions(db_obj, current_user_id)
        await self._log_audit_event("CREATE", db_obj, current_user_id)

    async def _log_audit_event(self, action: str, entity, user_id: uuid.UUID):
        # Custom audit logging
        self.logger.info(
            f"Audit: {action} {self.entity_name}",
            extra={
                "action": action,
                "entity_type": self.entity_name,
                "entity_id": str(entity.id),
                "user_id": str(user_id),
                "timestamp": datetime.utcnow().isoformat()
            }
        )
```

Custom Middleware

```python
class TenantIsolationMiddleware(BaseHTTPMiddleware):
    """Enforce tenant isolation at middleware level"""

    async def dispatch(self, request: Request, call_next):
        # Extract company context
        company_id = await self.extract_company_id(request)

        # Set database session context
        request.state.company_id = company_id

        # Process request
        response = await call_next(request)
        return response
```

Integration Patterns

External Service Integration

```python
class ExternalServiceBase:
    """Base class for external service integrations"""

    def __init__(self, api_key: str, base_url: str):
        self.api_key = api_key
        self.base_url = base_url
        self.session = httpx.AsyncClient()

    async def make_request(self, method: str, endpoint: str, **kwargs):
        # Standard error handling, retry logic, etc.
        pass

class PaymentGatewayService(ExternalServiceBase):
    async def process_payment(self, amount: Decimal, customer_id: str):
        return await self.make_request("POST", "/payments", json={
            "amount": str(amount),
            "customer_id": customer_id
        })
```

Background Task Patterns

```python
```

```python
# Using Celery (future implementation)
@celery_app.task
def send_policy_renewal_reminder(policy_id: str):
    # Background task implementation
    pass

# Using FastAPI BackgroundTasks (immediate implementation)
@router.post("/policies/{policy_id}/renew")
async def renew_policy(
    policy_id: uuid.UUID,
    background_tasks: BackgroundTasks,
    policy_service: PolicyService = Depends(get_policy_service)
):
    policy = await policy_service.get_or_404(policy_id)

    # Queue background task
    background_tasks.add_task(
        send_renewal_notification,
        policy.id,
        policy.customer_email
    )

    return {"message": "Renewal initiated"}
```

Configuration Extensions

Environment-Specific Settings

```python
# Development settings
class DevelopmentSettings(Settings):
    DEBUG: bool = True
    DATABASE_ECHO: bool = True
    LOG_LEVEL: str = "DEBUG"

# Production settings
class ProductionSettings(Settings):
    DEBUG: bool = False
    DATABASE_ECHO: bool = False
    LOG_LEVEL: str = "INFO"
    RATE_LIMIT_REQUESTS_PER_MINUTE: int = 1000

def get_settings() -> Settings:
    env = os.getenv("ENVIRONMENT", "development")
    if env == "production":
        return ProductionSettings()
    return DevelopmentSettings()
```

Feature Flags

```python
```

```python
class FeatureFlags(BaseModel):
    enable_advanced_analytics: bool = False
    enable_ai_pricing: bool = False
    enable_document_ai: bool = False

# In settings.py
FEATURE_FLAGS = FeatureFlags()

# Usage in services
if settings.FEATURE_FLAGS.enable_ai_pricing:
    price = await ai_pricing_service.calculate_premium(policy_data)
else:
    price = await standard_pricing_service.calculate_premium(policy_data)
```

Installation and Setup Guide

System Requirements

- **Python 3.11+** (async/await performance optimizations)
- **PostgreSQL 14+** (UUID generation, JSON support)
- **Redis 6+** (future caching and session storage)

Installation Steps

```bash
# 1. Create virtual environment
python -m venv venv
source venv/bin/activate  # Linux/Mac
# venv\Scripts\activate   # Windows

# 2. Install dependencies
pip install fastapi uvicorn[standard]
pip install sqlalchemy[asyncio] asyncpg psycopg2-binary
pip install pydantic pydantic-settings
pip install passlib[bcrypt] python-jose[cryptography]
pip install python-multipart  # For file uploads

# 3. Development dependencies
pip install pytest pytest-asyncio httpx
pip install black isort flake8 mypy  # Code quality tools
pip install alembic  # Database migrations

# 4. Set up environment variables
cp .env.example .env
# Edit .env with your database credentials

# 5. Initialize database
python -m alembic init alembic
python -c "from app.config.database import create_tables; import asyncio; asyncio.run(create_tables())"

# 6. Run the application
python main.py
```

Environment Configuration

```env
# .env file template
APP_NAME=CARDINSA Insurance API
ENVIRONMENT=development
DEBUG=true
LOG_LEVEL=INFO

# Database
DATABASE_URL=postgresql://username:password@localhost:5432/cardinsa_db
DATABASE_POOL_SIZE=10
DATABASE_MAX_OVERFLOW=20

# Security
SECRET_KEY=your-super-secret-key-change-in-production
JWT_SECRET_KEY=different-jwt-secret-key
ACCESS_TOKEN_EXPIRE_MINUTES=30

# CORS
CORS_ORIGINS=http://localhost:3000,http://localhost:8080
ALLOWED_HOSTS=localhost,127.0.0.1,0.0.0.0

# Rate Limiting
RATE_LIMIT_REQUESTS_PER_MINUTE=100
RATE_LIMIT_REQUESTS_PER_HOUR=1000
```

Performance Benchmarks and Monitoring

Performance Targets

- **API Response Time**: < 100ms for 95th percentile
- **Database Query Time**: < 50ms for standard queries
- **Concurrent Users**: 1000+ simultaneous connections
- **Throughput**: 10,000+ requests per minute

Monitoring Integration Points

```python
# Metrics collection (Prometheus ready)
from prometheus_client import Counter, Histogram

REQUEST_COUNT = Counter('http_requests_total', 'Total HTTP requests', ['method', 'endpoint'])
REQUEST_DURATION = Histogram('http_request_duration_seconds', 'HTTP request duration')

# Usage in middleware
REQUEST_COUNT.labels(method=request.method, endpoint=request.url.path).inc()
REQUEST_DURATION.observe(response_time)
```

Database Performance

```sql
```

```sql
-- Performance monitoring queries
SELECT
    schemaname,
    tablename,
    attname,
    n_distinct,
    correlation
FROM pg_stats
WHERE tablename IN ('users', 'policies', 'claims')
ORDER BY n_distinct DESC;

-- Query performance analysis
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM policies
WHERE company_id = $1 AND created_at > $2
LIMIT 20;
```

Security Checklist

Authentication Security

- ✅ **Password hashing** with bcrypt (cost factor 12)
- ✅ **JWT tokens** with proper expiration
- ✅ **Refresh token rotation** for long-term security
- ✅ **API key hashing** for external service access
- ✅ **Rate limiting** to prevent brute force attacks

Authorization Security

- ✅ **Role-based access control** (RBAC)
- ✅ **Permission-based authorization** (fine-grained)
- ✅ **Multi-tenant isolation** (company-level)
- ✅ **Resource ownership** validation
- ✅ **Audit trails** for all access attempts

Data Protection

- ✅ **Encryption at rest** (database-level)
- ✅ **Encryption in transit** (HTTPS/TLS)
- ✅ **Soft delete** for data recovery
- ✅ **Audit logging** for compliance
- ✅ **Input validation** at all boundaries

Infrastructure Security

- ✅ **Security headers** (HSTS, CSP, X-Frame-Options)
- ✅ **CORS configuration** for cross-origin requests
- ✅ **SQL injection prevention** via ORM
- ✅ **XSS prevention** via output encoding
- ✅ **Environment variable** protection

Conclusion

The CARDINSA Insurance Backend foundation provides a robust, scalable, and secure platform for building enterprise insurance applications. The architecture follows industry best practices and established patterns, ensuring:

✅ Technical Excellence

- **Clean Architecture** with clear separation of concerns
- **Domain-Driven Design** for business logic organization
- **Enterprise Patterns** for scalability and maintainability
- **Security-First** approach with comprehensive protection
- **Performance Optimization** from the ground up

✅ Developer Experience

- **Consistent Patterns** across all modules
- **Type Safety** with comprehensive type hints
- **Auto-generated Documentation** via OpenAPI/Swagger
- **Testing Foundation** ready for comprehensive test coverage
- **Error Handling** with detailed debugging information

✅ Business Readiness

- **Multi-tenant Architecture** for SaaS deployment
- **Audit Compliance** with comprehensive logging
- **Scalable Design** for growth and expansion
- **Integration Ready** for external services
- **Configuration Management** for different environments

📈 Next Steps

1. **Implement User Authentication Module** using the foundation
2. **Add Company Management** for multi-tenancy
3. **Build Core Insurance Entities** (Policies, Claims, etc.)
4. **Implement Business Workflows** (Underwriting, Claims Processing)
5. **Add External Integrations** (Payment Gateways, Document Services)

The foundation layer is complete and ready for rapid module development while maintaining enterprise-grade quality and security standards.

Document Version: 1.0.0
Last Updated: January 2025
Review Status: Ready for Technical Review
Approval Required: Architecture Team, Security Team, Development Team