🚀 Flask Admin Dashboard MVP - Complete Implementation Plan (Pydantic)

📋 Table of Contents

---

1. Project Setup

Step 1.1: Create Project Structure

bash

```bash
mkdir flask-admin-dashboard
cd flask-admin-dashboard

# Create directory structure
mkdir -p app/{models,routes,schemas,utils,middleware}
mkdir -p migrations
mkdir -p tests/{unit,integration}
mkdir -p config
touch app/__init__.py
touch run.py
touch requirements.txt
touch .env.example
touch .gitignore
```

Step 1.2: Create `.gitignore`

```gitignore
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
build/
dist/
*.egg-info/

# Flask
instance/
.webassets-cache

# Environment
.env
.env.local

# Database
*.db
*.sqlite3

# IDE
.vscode/
.idea/
*.swp
*.swo

# Logs
*.log

# OS
.DS_Store
Thumbs.db
```

Step 1.3: Install Dependencies

Create `requirements.txt`:

```txt
# Core Framework
Flask==3.0.0
Flask-SQLAlchemy==3.1.1
Flask-Migrate==4.0.5
Flask-RESTX==1.3.0
Flask-JWT-Extended==4.6.0
Flask-CORS==4.0.0

# Database
psycopg2-binary==2.9.9
alembic==1.13.1

# Validation & Serialization - PYDANTIC
pydantic==2.5.3
pydantic-settings==2.1.0
email-validator==2.1.0

# Password Hashing
bcrypt==4.1.2

# System Monitoring
psutil==5.9.6

# Environment Variables
python-dotenv==1.0.0

# Utilities
pytz==2023.3

# Testing
pytest==7.4.3
pytest-flask==1.3.0
```

Install:

```bash
python3 -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
```

Step 1.4: Environment Configuration

Create `.env.example`:

```env
# Flask Configuration
FLASK_APP=run.py
FLASK_ENV=development
SECRET_KEY=your-secret-key-change-this-in-production

# Database
DATABASE_URL=postgresql://username:password@localhost:5432/admin_dashboard

# JWT
JWT_SECRET_KEY=your-jwt-secret-key-change-this-in-production
JWT_ACCESS_TOKEN_EXPIRES=3600
JWT_REFRESH_TOKEN_EXPIRES=2592000

# Application
DEBUG=True
PORT=5000
```

Copy to `.env`:

```bash
cp .env.example .env
# Edit .env with your actual values
```

---

2. Database Configuration

Step 2.1: Create Config Files

`config/base.py`:

python

```python
import os
from datetime import timedelta
from dotenv import load_dotenv

load_dotenv()

class Config:
    """Base configuration"""
    # Flask
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev-secret-key-change-in-prod'

    # Database
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'postgresql://localhost/admin_dashboard'
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    # JWT
    JWT_SECRET_KEY = os.environ.get('JWT_SECRET_KEY') or 'jwt-secret-key'
    JWT_ACCESS_TOKEN_EXPIRES = timedelta(
        seconds=int(os.environ.get('JWT_ACCESS_TOKEN_EXPIRES', 3600))
    )
    JWT_REFRESH_TOKEN_EXPIRES = timedelta(
        seconds=int(os.environ.get('JWT_REFRESH_TOKEN_EXPIRES', 2592000))
    )

    # CORS
    CORS_HEADERS = 'Content-Type'

    # Pagination
    ITEMS_PER_PAGE = 20


class DevelopmentConfig(Config):
    """Development configuration"""
    DEBUG = True


class ProductionConfig(Config):
    """Production configuration"""
    DEBUG = False


config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

Step 2.2: Install PostgreSQL

```bash
# Ubuntu/Debian
sudo apt-get install postgresql postgresql-contrib

# macOS
brew install postgresql

# Start PostgreSQL
# Ubuntu: sudo service postgresql start
# macOS: brew services start postgresql

# Create database
createdb admin_dashboard

# Or via psql:
psql postgres
CREATE DATABASE admin_dashboard;
\q
```

---

3. Models Implementation

Step 3.1: User Model

`app/models/user.py`:

python

```python
from app import db
from datetime import datetime
import bcrypt


class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True, nullable=False, index=True)
    password_hash = db.Column(db.String(255), nullable=False)
    role = db.Column(
        db.String(20),
        nullable=False,
        default='user'
    )  # superadmin, admin, user
    status = db.Column(
        db.String(20),
        nullable=False,
        default='active'
    )  # active, inactive

    # Profile
    first_name = db.Column(db.String(50))
    last_name = db.Column(db.String(50))

    # Timestamps
    created_date = db.Column(db.DateTime, default=datetime.utcnow)
    last_login = db.Column(db.DateTime)

    # Relationships (Many-to-Many with Applications)
    assigned_applications = db.relationship(
        'Application',
        secondary='user_applications',
        back_populates='users'
    )

    # Activity logs
    activities = db.relationship(
        'ActivityLog',
        back_populates='user',
        cascade='all, delete-orphan'
    )

    def set_password(self, password):
        """Hash and set password"""
        self.password_hash = bcrypt.hashpw(
            password.encode('utf-8'),
            password.encode('utf-8'),
            bcrypt.gensalt()
        ).decode('utf-8')

    def check_password(self, password):
        """Verify password"""
        return bcrypt.checkpw(
            password.encode('utf-8'),
            self.password_hash.encode('utf-8')
        )

    def to_dict(self):
        """Convert to dictionary"""
        return {
            'id': self.id,
            'email': self.email,
            'role': self.role,
            'status': self.status,
            'first_name': self.first_name,
            'last_name': self.last_name,
            'created_date': self.created_date.isoformat() if self.created_date else None,
            'last_login': self.last_login.isoformat() if self.last_login else None,
            'assigned_applications': [app.to_dict() for app in self.assigned_applications]
        }

    def __repr__(self):
        return f'<User {self.email}>'


# Association table for User-Application many-to-many
user_applications = db.Table(
    'user_applications',
    db.Column('user_id', db.Integer, db.ForeignKey('users.id'), primary_key=True),
    db.Column('application_id', db.Integer, db.ForeignKey('applications.id'), primary_key=True),
    db.Column('assigned_date', db.DateTime, default=datetime.utcnow)
)
```

Step 3.2: Application (Region) Model

`app/models/application.py`:

```python
from app import db
from datetime import datetime


class Application(db.Model):
    __tablename__ = 'applications'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True, nullable=False, index=True)
    description = db.Column(db.Text)
    url = db.Column(db.String(255))
    status = db.Column(
        db.String(20),
        nullable=False,
        default='active'
    )  # active, inactive, maintenance

    # Timestamps
    created_date = db.Column(db.DateTime, default=datetime.utcnow)
    last_updated = db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    # Relationships
    users = db.relationship(
        'User',
        secondary='user_applications',
        back_populates='assigned_applications'
    )

    def to_dict(self):
        """Convert to dictionary"""
        return {
            'id': self.id,
            'name': self.name,
            'description': self.description,
            'url': self.url,
            'status': self.status,
            'created_date': self.created_date.isoformat() if self.created_date else None,
            'last_updated': self.last_updated.isoformat() if self.last_updated else None,
            'user_count': len(self.users)
        }

    def __repr__(self):
        return f'<Application {self.name}>'
```

Step 3.3: Activity Log Model

`app/models/activity.py`:

```python
from app import db
from datetime import datetime


class ActivityLog(db.Model):
    __tablename__ = 'activity_logs'

    id = db.Column(db.Integer, primary_key=True)
    event_type = db.Column(db.String(50), nullable=False, index=True)
    # user_login, user_logout, user_created, user_updated, user_deleted,
    # app_created, app_updated, app_deleted, etc.

    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=True)
    description = db.Column(db.Text, nullable=False)
    ip_address = db.Column(db.String(45))  # IPv6 compatible
    user_agent = db.Column(db.String(255))

    # Additional context (JSON-like storage)
    metadata = db.Column(db.Text)  # Can store JSON string

    timestamp = db.Column(db.DateTime, default=datetime.utcnow, index=True)

    # Relationships
    user = db.relationship('User', back_populates='activities')

    def to_dict(self):
        """Convert to dictionary"""
        return {
            'id': self.id,
            'event_type': self.event_type,
            'user_id': self.user_id,
            'user_email': self.user.email if self.user else None,
            'description': self.description,
            'ip_address': self.ip_address,
            'timestamp': self.timestamp.isoformat() if self.timestamp else None
        }

    def __repr__(self):
        return f'<ActivityLog {self.event_type} at {self.timestamp}>'
```

Step 3.4: System Metrics Model

`app/models/metrics.py`:

```python
from app import db
from datetime import datetime


class SystemMetric(db.Model):
    __tablename__ = 'system_metrics'

    id = db.Column(db.Integer, primary_key=True)
    cpu_usage = db.Column(db.Float)  # Percentage
    memory_usage = db.Column(db.Float)  # Percentage
    memory_total = db.Column(db.BigInteger)  # Bytes
    memory_used = db.Column(db.BigInteger)  # Bytes
    disk_usage = db.Column(db.Float)  # Percentage
    disk_total = db.Column(db.BigInteger)  # Bytes
    disk_used = db.Column(db.BigInteger)  # Bytes

    timestamp = db.Column(db.DateTime, default=datetime.utcnow, index=True)

    def to_dict(self):
        """Convert to dictionary"""
        return {
            'id': self.id,
            'cpu_usage': round(self.cpu_usage, 2) if self.cpu_usage else None,
            'memory_usage': round(self.memory_usage, 2) if self.memory_usage else None,
            'memory_total': self.memory_total,
            'memory_used': self.memory_used,
            'disk_usage': round(self.disk_usage, 2) if self.disk_usage else None,
            'disk_total': self.disk_total,
            'disk_used': self.disk_used,
            'timestamp': self.timestamp.isoformat() if self.timestamp else None
        }

    def __repr__(self):
        return f'<SystemMetric at {self.timestamp}>'
```

Step 3.5: Models Init

`app/models/__init__.py`:

python

```python
from app.models.user import User, user_applications
from app.models.application import Application
from app.models.activity import ActivityLog
from app.models.metrics import SystemMetric

__all__ = [
    'User',
    'Application',
    'ActivityLog',
    'SystemMetric',
    'user_applications'
]
```

---

4. Authentication System

Step 4.1: Pydantic Schemas

`app/schemas/user_schema.py`:

python

```python
from pydantic import BaseModel, EmailStr, Field, field_validator, model_validator
from typing import Optional, List, Literal
from datetime import datetime


class LoginSchema(BaseModel):
    """Login request schema"""
    email: EmailStr
    password: str = Field(..., min_length=6)


class UserCreateSchema(BaseModel):
    """User creation schema"""
    email: EmailStr
    password: str = Field(..., min_length=6)
    role: Literal['user', 'admin', 'superadmin'] = 'user'
    status: Literal['active', 'inactive'] = 'active'
    first_name: Optional[str] = None
    last_name: Optional[str] = None
    application_ids: List[int] = Field(default_factory=list)

    @field_validator('password')
    @classmethod
    def validate_password_strength(cls, v: str) -> str:
        """Validate password strength"""
        if len(v) < 6:
            raise ValueError('Password must be at least 6 characters')
        return v


class UserUpdateSchema(BaseModel):
    """User update schema - all fields optional"""
    email: Optional[EmailStr] = None
    password: Optional[str] = Field(None, min_length=6)
    role: Optional[Literal['user', 'admin', 'superadmin']] = None
    status: Optional[Literal['active', 'inactive']] = None
    first_name: Optional[str] = None
    last_name: Optional[str] = None
    application_ids: Optional[List[int]] = None

    @model_validator(mode='after')
    def check_at_least_one_field(self):
        """Ensure at least one field is provided"""
        if not any([
            self.email, self.password, self.role, self.status,
            self.first_name, self.last_name, self.application_ids
        ]):
            raise ValueError('At least one field must be provided for update')
        return self


class UserQuerySchema(BaseModel):
    """User query parameters schema"""
    page: int = Field(default=1, ge=1)
    per_page: int = Field(default=20, ge=1, le=100)
    search: Optional[str] = None
    role: Optional[Literal['user', 'admin', 'superadmin']] = None
    status: Optional[Literal['active', 'inactive']] = None
    sort: str = 'created_date'
    order: Literal['asc', 'desc'] = 'desc'

    model_config = {
        'extra': 'forbid'  # Forbid extra fields
    }


class UserResponseSchema(BaseModel):
    """User response schema"""
    id: int
    email: str
    role: str
    status: str
    first_name: Optional[str]
    last_name: Optional[str]
    created_date: Optional[datetime]
    last_login: Optional[datetime]
    assigned_applications: List[dict]

    model_config = {
        'from_attributes': True  # Allow ORM models
    }
```

app/schemas/application_schema.py:

python

```python
from pydantic import BaseModel, Field, HttpUrl
from typing import Optional, Literal
from datetime import datetime


class ApplicationCreateSchema(BaseModel):
    """Application creation schema"""
    name: str = Field(..., min_length=1, max_length=100)
    description: Optional[str] = None
    url: Optional[HttpUrl] = None
    status: Literal['active', 'inactive', 'maintenance'] = 'active'


class ApplicationUpdateSchema(BaseModel):
    """Application update schema"""
    name: Optional[str] = Field(None, min_length=1, max_length=100)
    description: Optional[str] = None
    url: Optional[HttpUrl] = None
    status: Optional[Literal['active', 'inactive', 'maintenance']] = None


class ApplicationQuerySchema(BaseModel):
    """Application query parameters schema"""
    page: int = Field(default=1, ge=1)
    per_page: int = Field(default=20, ge=1, le=100)
    search: Optional[str] = None
    status: Optional[Literal['active', 'inactive', 'maintenance']] = None
    sort: str = 'name'
    order: Literal['asc', 'desc'] = 'asc'

    model_config = {
        'extra': 'forbid'
    }


class ApplicationResponseSchema(BaseModel):
    """Application response schema"""
    id: int
    name: str
    description: Optional[str]
    url: Optional[str]
    status: str
    created_date: Optional[datetime]
    last_updated: Optional[datetime]
    user_count: int

    model_config = {
```

```
    'from_attributes': True
}
```

Step 4.2: Pydantic Validation Helper

`app/utils/validation.py`:

python

```python
from pydantic import BaseModel, ValidationError
from flask import jsonify, request
from functools import wraps
from typing import Type


def validate_request(schema: Type[BaseModel], source: str = 'json'):
    """
    Decorator to validate Flask request data with Pydantic

    Args:
        schema: Pydantic model class to validate against
        source: Where to get data from ('json', 'args', 'form')
    """
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            try:
                # Get data based on source
                if source == 'json':
                    data = request.get_json() or {}
                elif source == 'args':
                    data = request.args.to_dict()
                elif source == 'form':
                    data = request.form.to_dict()
                else:
                    return jsonify({
                        'error': {
                            'code': 'INVALID_SOURCE',
                            'message': f'Invalid data source: {source}'
                        }
                    }), 500

                # Validate with Pydantic
                validated_data = schema(**data)

                # Add validated data to kwargs
                kwargs['validated_data'] = validated_data

                return f(*args, **kwargs)

            except ValidationError as e:
                return jsonify({
                    'error': {
                        'code': 'VALIDATION_ERROR',
                        'message': 'Invalid input data',
                        'details': e.errors()
                }
            }), 400
        except Exception as e:
            return jsonify({
                'error': {
                    'code': 'INTERNAL_ERROR',
                    'message': str(e)
                }
            }), 500

    return decorated_function
  return decorator


def validate_query_params(schema: Type[BaseModel]):
    """Shorthand for validating query parameters"""
    return validate_request(schema, source='args')


def validate_json_body(schema: Type[BaseModel]):
    """Shorthand for validating JSON body"""
    return validate_request(schema, source='json')
```

Step 4.3: Error Handler Utility

`app/utils/error_handler.py`:

python

```python
from flask import jsonify
from pydantic import ValidationError


def register_error_handlers(app):
    """Register error handlers for the application"""

    @app.errorhandler(ValidationError)
    def handle_pydantic_validation_error(error):
        """Handle Pydantic validation errors"""
        return jsonify({
            'error': {
                'code': 'VALIDATION_ERROR',
                'message': 'Invalid input data',
                'details': error.errors()
            }
        }), 400

    @app.errorhandler(404)
    def handle_not_found(error):
        return jsonify({
            'error': {
                'code': 'NOT_FOUND',
                'message': 'Resource not found'
            }
        }), 404

    @app.errorhandler(401)
    def handle_unauthorized(error):
        return jsonify({
            'error': {
                'code': 'UNAUTHORIZED',
                'message': 'Authentication required'
            }
        }), 401

    @app.errorhandler(403)
    def handle_forbidden(error):
        return jsonify({
            'error': {
                'code': 'FORBIDDEN',
                'message': 'You do not have permission to access this resource'
            }
        }), 403

    @app.errorhandler(500)
    def handle_internal_error(error):
```

```
    return jsonify({
        'error': {
            'code': 'INTERNAL_SERVER_ERROR',
            'message': 'An internal server error occurred'
        }
    }), 500
```

Step 4.4: Authentication Routes

`app/routes/auth.py`:

python

```python
from flask import Blueprint, request, jsonify
from flask_jwt_extended import (
    create_access_token,
    create_refresh_token,
    jwt_required,
    get_jwt_identity,
    get_jwt
)
from datetime import datetime

from app import db
from app.models import User, ActivityLog
from app.schemas.user_schema import LoginSchema
from app.utils.validation import validate_json_body

auth_bp = Blueprint('auth', __name__, url_prefix='/api/auth')


@auth_bp.route('/login', methods=['POST'])
@validate_json_body(LoginSchema)
def login(validated_data: LoginSchema):
    """User login endpoint"""

    # Find user
    user = User.query.filter_by(email=validated_data.email).first()

    if not user or not user.check_password(validated_data.password):
        return jsonify({
            'error': {
                'code': 'INVALID_CREDENTIALS',
                'message': 'Invalid email or password'
            }
        }), 401

    # Check if user is active
    if user.status != 'active':
        return jsonify({
            'error': {
                'code': 'ACCOUNT_INACTIVE',
                'message': 'Your account has been deactivated'
            }
        }), 403

    # Update last login
    user.last_login = datetime.utcnow()
    db.session.commit()
    # Create tokens
    access_token = create_access_token(
        identity=user.id,
        additional_claims={'role': user.role}
    )
    refresh_token = create_refresh_token(identity=user.id)

    # Log activity
    activity = ActivityLog(
        event_type='user_login',
        user_id=user.id,
        description=f'User {user.email} logged in',
        ip_address=request.remote_addr,
        user_agent=request.headers.get('User-Agent')
    )
    db.session.add(activity)
    db.session.commit()

    return jsonify({
        'message': 'Login successful',
        'access_token': access_token,
        'refresh_token': refresh_token,
        'user': user.to_dict()
    }), 200


@auth_bp.route('/refresh', methods=['POST'])
@jwt_required(refresh=True)
def refresh():
    """Refresh access token"""
    identity = get_jwt_identity()
    access_token = create_access_token(identity=identity)

    return jsonify({
        'access_token': access_token
    }), 200


@auth_bp.route('/logout', methods=['POST'])
@jwt_required()
def logout():
    """User logout endpoint"""
    user_id = get_jwt_identity()
    user = User.query.get(user_id)

    if user:
        # Log activity
        activity = ActivityLog(
            event_type='user_logout',
            user_id=user.id,
            description=f'User {user.email} logged out',
            ip_address=request.remote_addr
        )
        db.session.add(activity)
        db.session.commit()

    return jsonify({
        'message': 'Logout successful'
    }), 200


@auth_bp.route('/me', methods=['GET'])
@jwt_required()
def get_current_user():
    """Get current authenticated user"""
    user_id = get_jwt_identity()
    user = User.query.get(user_id)

    if not user:
        return jsonify({
            'error': {
                'code': 'USER_NOT_FOUND',
                'message': 'User not found'
            }
        }), 404

    return jsonify(user.to_dict()), 200
```

---

5. API Endpoints

Step 5.1: User Management Routes

`app/routes/users.py`:

python

```python
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required, get_jwt_identity, get_jwt
from sqlalchemy import or_

from app import db
from app.models import User, Application, ActivityLog
from app.schemas.user_schema import (
    UserCreateSchema,
    UserUpdateSchema,
    UserQuerySchema
)
from app.utils.validation import validate_json_body, validate_query_params

users_bp = Blueprint('users', __name__, url_prefix='/api/users')


def require_admin():
    """Decorator to require admin or superadmin role"""
    claims = get_jwt()
    role = claims.get('role', 'user')
    if role not in ['admin', 'superadmin']:
        return jsonify({
            'error': {
                'code': 'FORBIDDEN',
                'message': 'Admin access required'
            }
        }), 403
    return None


@users_bp.route('', methods=['GET'])
@jwt_required()
@validate_query_params(UserQuerySchema)
def get_users(validated_data: UserQuerySchema):
    """Get all users with pagination and filtering"""
    error = require_admin()
    if error:
        return error

    # Build query
    query = User.query

    # Search filter
    if validated_data.search:
        search_term = f"%{validated_data.search}%"
        query = query.filter(
            or_(
                User.email.ilike(search_term),
                User.first_name.ilike(search_term),
                User.last_name.ilike(search_term)
            )
        )

    # Role filter
    if validated_data.role:
        query = query.filter_by(role=validated_data.role)

    # Status filter
    if validated_data.status:
        query = query.filter_by(status=validated_data.status)

    # Sorting
    sort_column = getattr(User, validated_data.sort, User.created_date)
    if validated_data.order == 'desc':
        query = query.order_by(sort_column.desc())
    else:
        query = query.order_by(sort_column.asc())

    # Pagination
    pagination = query.paginate(
        page=validated_data.page,
        per_page=validated_data.per_page,
        error_out=False
    )

    return jsonify({
        'users': [user.to_dict() for user in pagination.items],
        'pagination': {
            'page': validated_data.page,
            'per_page': validated_data.per_page,
            'total': pagination.total,
            'pages': pagination.pages,
            'has_next': pagination.has_next,
            'has_prev': pagination.has_prev
        }
    }), 200


@users_bp.route('', methods=['POST'])
@jwt_required()
@validate_json_body(UserCreateSchema)
def create_user(validated_data: UserCreateSchema):
    """Create a new user"""
    error = require_admin()
    if error:
        return error
    # Check if email already exists
    if User.query.filter_by(email=validated_data.email).first():
        return jsonify({
            'error': {
                'code': 'EMAIL_EXISTS',
                'message': 'A user with this email already exists'
            }
        }), 409

    # Create user
    user = User(
        email=validated_data.email,
        role=validated_data.role,
        status=validated_data.status,
        first_name=validated_data.first_name,
        last_name=validated_data.last_name
    )
    user.set_password(validated_data.password)

    # Assign applications
    if validated_data.application_ids:
        applications = Application.query.filter(
            Application.id.in_(validated_data.application_ids)
        ).all()
        user.assigned_applications = applications

    db.session.add(user)
    db.session.commit()

    # Log activity
    current_user_id = get_jwt_identity()
    activity = ActivityLog(
        event_type='user_created',
        user_id=current_user_id,
        description=f'Created user: {user.email}',
        ip_address=request.remote_addr
    )
    db.session.add(activity)
    db.session.commit()

    return jsonify({
        'message': 'User created successfully',
        'user': user.to_dict()
    }), 201


@users_bp.route('/<int:user_id>', methods=['GET'])
@jwt_required()
def get_user(user_id):
    """Get a specific user by ID"""
    error = require_admin()
    if error:
        return error

    user = User.query.get(user_id)

    if not user:
        return jsonify({
            'error': {
                'code': 'USER_NOT_FOUND',
                'message': f'User with id {user_id} not found'
            }
        }), 404

    return jsonify(user.to_dict()), 200


@users_bp.route('/<int:user_id>', methods=['PUT'])
@jwt_required()
@validate_json_body(UserUpdateSchema)
def update_user(user_id, validated_data: UserUpdateSchema):
    """Update a user"""
    error = require_admin()
    if error:
        return error

    user = User.query.get(user_id)

    if not user:
        return jsonify({
            'error': {
                'code': 'USER_NOT_FOUND',
                'message': f'User with id {user_id} not found'
            }
        }), 404

    # Update fields (only if provided)
    if validated_data.email:
        # Check if new email already exists
        existing = User.query.filter(
            User.email == validated_data.email,
            User.id != user_id
        ).first()
        if existing:
            return jsonify({
                'error': {
                    'code': 'EMAIL_EXISTS',
                    'message': 'A user with this email already exists'
            }
        }), 409
        user.email = validated_data.email

    if validated_data.password:
        user.set_password(validated_data.password)

    if validated_data.role:
        user.role = validated_data.role

    if validated_data.status:
        user.status = validated_data.status

    if validated_data.first_name is not None:
        user.first_name = validated_data.first_name

    if validated_data.last_name is not None:
        user.last_name = validated_data.last_name

    # Update applications
    if validated_data.application_ids is not None:
        applications = Application.query.filter(
            Application.id.in_(validated_data.application_ids)
        ).all()
        user.assigned_applications = applications

    db.session.commit()

    # Log activity
    current_user_id = get_jwt_identity()
    activity = ActivityLog(
        event_type='user_updated',
        user_id=current_user_id,
        description=f'Updated user: {user.email}',
        ip_address=request.remote_addr
    )
    db.session.add(activity)
    db.session.commit()

    return jsonify({
        'message': 'User updated successfully',
        'user': user.to_dict()
    }), 200


@users_bp.route('/<int:user_id>', methods=['DELETE'])
@jwt_required()
def delete_user(user_id):
    """Delete a user"""
    error = require_admin()
    error = require_admin()
    if error:
        return error

    user = User.query.get(user_id)

    if not user:
        return jsonify({
            'error': {
                'code': 'USER_NOT_FOUND',
                'message': f'User with id {user_id} not found'
            }
        }), 404

    # Prevent deleting yourself
    current_user_id = get_jwt_identity()
    if user_id == current_user_id:
        return jsonify({
            'error': {
                'code': 'CANNOT_DELETE_SELF',
                'message': 'You cannot delete your own account'
            }
        }), 400

    email = user.email
    db.session.delete(user)
    db.session.commit()

    # Log activity
    activity = ActivityLog(
        event_type='user_deleted',
        user_id=current_user_id,
        description=f'Deleted user: {email}',
        ip_address=request.remote_addr
    )
    db.session.add(activity)
    db.session.commit()

    return jsonify({
        'message': 'User deleted successfully'
    }), 200
```

Step 5.2: Application Management Routes

`app/routes/applications.py`:

python

```python
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required, get_jwt_identity, get_jwt

from app import db
from app.models import Application, ActivityLog
from app.schemas.application_schema import (
    ApplicationCreateSchema,
    ApplicationUpdateSchema,
    ApplicationQuerySchema
)
from app.utils.validation import validate_json_body, validate_query_params

applications_bp = Blueprint('applications', __name__, url_prefix='/api/applications')


def require_admin():
    """Decorator to require admin or superadmin role"""
    claims = get_jwt()
    role = claims.get('role', 'user')
    if role not in ['admin', 'superadmin']:
        return jsonify({
            'error': {
                'code': 'FORBIDDEN',
                'message': 'Admin access required'
            }
        }), 403
    return None


@applications_bp.route('', methods=['GET'])
@jwt_required()
@validate_query_params(ApplicationQuerySchema)
def get_applications(validated_data: ApplicationQuerySchema):
    """Get all applications with pagination and filtering"""

    # Build query
    query = Application.query

    # Search filter
    if validated_data.search:
        search_term = f"%{validated_data.search}%"
        query = query.filter(Application.name.ilike(search_term))

    # Status filter
    if validated_data.status:
        query = query.filter_by(status=validated_data.status)
    # Sorting
    sort_column = getattr(Application, validated_data.sort, Application.name)
    if validated_data.order == 'desc':
        query = query.order_by(sort_column.desc())
    else:
        query = query.order_by(sort_column.asc())

    # Pagination
    pagination = query.paginate(
        page=validated_data.page,
        per_page=validated_data.per_page,
        error_out=False
    )

    return jsonify({
        'applications': [app.to_dict() for app in pagination.items],
        'pagination': {
            'page': validated_data.page,
            'per_page': validated_data.per_page,
            'total': pagination.total,
            'pages': pagination.pages,
            'has_next': pagination.has_next,
            'has_prev': pagination.has_prev
        }
    }), 200


@applications_bp.route('', methods=['POST'])
@jwt_required()
@validate_json_body(ApplicationCreateSchema)
def create_application(validated_data: ApplicationCreateSchema):
    """Create a new application"""
    error = require_admin()
    if error:
        return error

    # Check if name already exists
    if Application.query.filter_by(name=validated_data.name).first():
        return jsonify({
            'error': {
                'code': 'APPLICATION_EXISTS',
                'message': 'An application with this name already exists'
            }
        }), 409

    # Create application
    application = Application(
        name=validated_data.name,
        description=validated_data.description,
        url=str(validated_data.url) if validated_data.url else None,
        status=validated_data.status
    )

    db.session.add(application)
    db.session.commit()

    # Log activity
    current_user_id = get_jwt_identity()
    activity = ActivityLog(
        event_type='application_created',
        user_id=current_user_id,
        description=f'Created application: {application.name}',
        ip_address=request.remote_addr
    )
    db.session.add(activity)
    db.session.commit()

    return jsonify({
        'message': 'Application created successfully',
        'application': application.to_dict()
    }), 201


@applications_bp.route('/<int:app_id>', methods=['GET'])
@jwt_required()
def get_application(app_id):
    """Get a specific application by ID"""
    application = Application.query.get(app_id)

    if not application:
        return jsonify({
            'error': {
                'code': 'APPLICATION_NOT_FOUND',
                'message': f'Application with id {app_id} not found'
            }
        }), 404

    return jsonify(application.to_dict()), 200


@applications_bp.route('/<int:app_id>', methods=['PUT'])
@jwt_required()
@validate_json_body(ApplicationUpdateSchema)
def update_application(app_id, validated_data: ApplicationUpdateSchema):
    """Update an application"""
    error = require_admin()
    if error:
        return error
```

```python
application = Application.query.get(app_id)

if not application:
    return jsonify({
        'error': {
            'code': 'APPLICATION_NOT_FOUND',
            'message': f'Application with id {app_id} not found'
        }
    }), 404

# Update fields
if validated_data.name:
    # Check if new name already exists
    existing = Application.query.filter(
        Application.name == validated_data.name,
        Application.id != app_id
    ).first()
    if existing:
        return jsonify({
            'error': {
                'code': 'APPLICATION_EXISTS',
                'message': 'An application with this name already exists'
            }
        }), 409
    application.name = validated_data.name

if validated_data.description is not None:
    application.description = validated_data.description

if validated_data.url is not None:
    application.url = str(validated_data.url) if validated_data.url else None

if validated_data.status:
    application.status = validated_data.status

db.session.commit()

# Log activity
current_user_id = get_jwt_identity()
activity = ActivityLog(
    event_type='application_updated',
    user_id=current_user_id,
    description=f'Updated application: {application.name}',
    ip_address=request.remote_addr
)
db.session.add(activity)
db.session.commit()

return jsonify({
        'message': 'Application updated successfully',
        'application': application.to_dict()
    }), 200


@applications_bp.route('/<int:app_id>', methods=['DELETE'])
@jwt_required()
def delete_application(app_id):
    """Delete an application"""
    error = require_admin()
    if error:
        return error

    application = Application.query.get(app_id)

    if not application:
        return jsonify({
            'error': {
                'code': 'APPLICATION_NOT_FOUND',
                'message': f'Application with id {app_id} not found'
            }
        }), 404

    name = application.name
    db.session.delete(application)
    db.session.commit()

    # Log activity
    current_user_id = get_jwt_identity()
    activity = ActivityLog(
        event_type='application_deleted',
        user_id=current_user_id,
        description=f'Deleted application: {name}',
        ip_address=request.remote_addr
    )
    db.session.add(activity)
    db.session.commit()

    return jsonify({
        'message': 'Application deleted successfully'
    }), 200
```

Step 5.3: Dashboard Routes

`app/routes/dashboard.py`:

python

```python
from flask import Blueprint, jsonify
from flask_jwt_extended import jwt_required
from sqlalchemy import func
from datetime import datetime, timedelta

from app import db
from app.models import User, Application, ActivityLog, SystemMetric
from app.utils.monitoring import get_system_health

dashboard_bp = Blueprint('dashboard', __name__, url_prefix='/api/dashboard')


@dashboard_bp.route('/stats', methods=['GET'])
@jwt_required()
def get_stats():
    """Get dashboard statistics"""

    # User counts
    total_users = User.query.count()
    active_users = User.query.filter_by(status='active').count()
    inactive_users = User.query.filter_by(status='inactive').count()

    # Users by role
    users_by_role = db.session.query(
        User.role,
        func.count(User.id)
    ).group_by(User.role).all()

    role_counts = {role: count for role, count in users_by_role}

    # Application counts
    total_applications = Application.query.count()
    active_applications = Application.query.filter_by(status='active').count()

    # Recent activity count (last 24 hours)
    yesterday = datetime.utcnow() - timedelta(days=1)
    recent_activities = ActivityLog.query.filter(
        ActivityLog.timestamp >= yesterday
    ).count()

    # Recent logins (last 7 days)
    week_ago = datetime.utcnow() - timedelta(days=7)
    recent_logins = User.query.filter(
        User.last_login >= week_ago
    ).count()

    return jsonify({
        'users': {
            'total': total_users,
            'active': active_users,
            'inactive': inactive_users,
            'by_role': role_counts,
            'recent_logins': recent_logins
        },
        'applications': {
            'total': total_applications,
            'active': active_applications
        },
        'activity': {
            'recent_count': recent_activities
        }
    }), 200


@dashboard_bp.route('/health', methods=['GET'])
@jwt_required()
def get_health():
    """Get real-time system health metrics"""
    health = get_system_health()
    return jsonify(health), 200


@dashboard_bp.route('/activity', methods=['GET'])
@jwt_required()
def get_activity():
    """Get recent activity logs"""

    # Get last 50 activities
    activities = ActivityLog.query.order_by(
        ActivityLog.timestamp.desc()
    ).limit(50).all()

    return jsonify({
        'activities': [activity.to_dict() for activity in activities]
    }), 200


@dashboard_bp.route('/metrics/history', methods=['GET'])
@jwt_required()
def get_metrics_history():
    """Get historical system metrics (last 24 hours)"""

    yesterday = datetime.utcnow() - timedelta(days=1)
    metrics = SystemMetric.query.filter(
        SystemMetric.timestamp >= yesterday
    ).order_by(SystemMetric.timestamp.asc()).all()
    return jsonify({
        'metrics': [metric.to_dict() for metric in metrics]
    }), 200
```

---

6. System Monitoring

Step 6.1: Monitoring Utility

`app/utils/monitoring.py`:

```python
import psutil
from datetime import datetime
from app import db
from app.models import SystemMetric


def get_system_health():
    """Get current system health metrics"""

    # CPU usage
    cpu_percent = psutil.cpu_percent(interval=1)

    # Memory usage
    memory = psutil.virtual_memory()
    memory_percent = memory.percent
    memory_total = memory.total
    memory_used = memory.used

    # Disk usage
    disk = psutil.disk_usage('/')
    disk_percent = disk.percent
    disk_total = disk.total
    disk_used = disk.used

    return {
        'cpu': {
            'usage_percent': round(cpu_percent, 2)
        },
        'memory': {
            'usage_percent': round(memory_percent, 2),
            'total_bytes': memory_total,
            'used_bytes': memory_used,
            'total_gb': round(memory_total / (1024**3), 2),
            'used_gb': round(memory_used / (1024**3), 2)
        },
        'disk': {
            'usage_percent': round(disk_percent, 2),
            'total_bytes': disk_total,
            'used_bytes': disk_used,
            'total_gb': round(disk_total / (1024**3), 2),
            'used_gb': round(disk_used / (1024**3), 2)
        },
        'timestamp': datetime.utcnow().isoformat()
    }


def save_system_metrics():
    """Save current system metrics to database"""

    health = get_system_health()

    metric = SystemMetric(
        cpu_usage=health['cpu']['usage_percent'],
        memory_usage=health['memory']['usage_percent'],
        memory_total=health['memory']['total_bytes'],
        memory_used=health['memory']['used_bytes'],
        disk_usage=health['disk']['usage_percent'],
        disk_total=health['disk']['total_bytes'],
        disk_used=health['disk']['used_bytes']
    )

    db.session.add(metric)
    db.session.commit()

    return metric
```

Step 6.2: Background Metrics Collector (Optional)

`app/utils/background_tasks.py`:

python

```python
import threading
import time
from app.utils.monitoring import save_system_metrics


class MetricsCollector(threading.Thread):
    """Background thread to collect system metrics periodically"""

    def __init__(self, app, interval=300):  # Default: every 5 minutes
        threading.Thread.__init__(self, daemon=True)
        self.app = app
        self.interval = interval
        self.running = True

    def run(self):
        """Run the metrics collection loop"""
        while self.running:
            with self.app.app_context():
                try:
                    save_system_metrics()
                    print(f"System metrics saved at {time.ctime()}")
                except Exception as e:
                    print(f"Error saving metrics: {e}")

            time.sleep(self.interval)

    def stop(self):
        """Stop the metrics collector"""
        self.running = False
```

---

7. Activity Logging

Step 7.1: Activity Logging Middleware

`app/middleware/activity_logger.py`:

```python
from flask import request, g
from flask_jwt_extended import get_jwt_identity, verify_jwt_in_request
from functools import wraps

from app import db
from app.models import ActivityLog


def log_activity(event_type, description):
    """Log an activity to the database"""
    try:
        verify_jwt_in_request(optional=True)
        user_id = get_jwt_identity()
    except:
        user_id = None

    activity = ActivityLog(
        event_type=event_type,
        user_id=user_id,
        description=description,
        ip_address=request.remote_addr,
        user_agent=request.headers.get('User-Agent')
    )

    db.session.add(activity)
    db.session.commit()

    return activity


def activity_required(event_type):
    """Decorator to automatically log activity for a route"""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            # Execute the function first
            result = f(*args, **kwargs)

            # Log activity after successful execution
            try:
                description = f"Executed {f.__name__}"
                log_activity(event_type, description)
            except:
                pass  # Don't fail the request if logging fails

            return result
        return decorated_function
    return decorator
```

---

8. Testing

Step 8.1: Test Configuration

`tests/conftest.py`:

python

```python
import pytest
from app import create_app, db
from app.models import User, Application
from config.base import Config


class TestConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'postgresql://localhost/admin_dashboard_test'
    JWT_SECRET_KEY = 'test-secret-key'


@pytest.fixture
def app():
    """Create application for testing"""
    app = create_app(TestConfig)

    with app.app_context():
        db.create_all()
        yield app
        db.session.remove()
        db.drop_all()


@pytest.fixture
def client(app):
    """Create test client"""
    return app.test_client()


@pytest.fixture
def auth_headers(client):
    """Create authenticated headers"""
    # Create a test user
    from app import db
    from app.models import User

    user = User(email='admin@test.com', role='admin', status='active')
    user.set_password('password123')
    db.session.add(user)
    db.session.commit()

    # Login
    response = client.post('/api/auth/login', json={
        'email': 'admin@test.com',
        'password': 'password123'
    })
    token = response.json['access_token']

    return {
        'Authorization': f'Bearer {token}'
    }
```

Step 8.2: Sample Tests

`tests/unit/test_user_model.py`:

```python
from app.models import User


def test_user_password_hashing(app):
    """Test password hashing and verification"""
    with app.app_context():
        user = User(email='test@example.com')
        user.set_password('password123')

        assert user.check_password('password123')
        assert not user.check_password('wrongpassword')


def test_user_to_dict(app):
    """Test user serialization"""
    with app.app_context():
        user = User(
            email='test@example.com',
            role='admin',
            status='active'
        )

        data = user.to_dict()

        assert data['email'] == 'test@example.com'
        assert data['role'] == 'admin'
        assert 'password_hash' not in data
```

`tests/unit/test_schemas.py`:

python

```python
import pytest
from pydantic import ValidationError
from app.schemas.user_schema import (
    LoginSchema,
    UserCreateSchema,
    UserQuerySchema
)


def test_login_schema_valid():
    """Test valid login schema"""
    data = {
        'email': 'test@example.com',
        'password': 'password123'
    }
    schema = LoginSchema(**data)
    assert schema.email == 'test@example.com'
    assert schema.password == 'password123'


def test_login_schema_invalid_email():
    """Test login schema with invalid email"""
    data = {
        'email': 'invalid-email',
        'password': 'password123'
    }
    with pytest.raises(ValidationError) as exc_info:
        LoginSchema(**data)

    errors = exc_info.value.errors()
    assert any('email' in str(error) for error in errors)


def test_user_create_schema_defaults():
    """Test user creation schema with defaults"""
    data = {
        'email': 'user@example.com',
        'password': 'password123'
    }
    schema = UserCreateSchema(**data)
    assert schema.role == 'user'
    assert schema.status == 'active'
    assert schema.application_ids == []
```

tests/integration/test_auth.py:

```python
def test_login_success(client, app):
    """Test successful login"""
    from app import db
    from app.models import User

    with app.app_context():
        # Create test user
        user = User(email='test@example.com', role='user', status='active')
        user.set_password('password123')
        db.session.add(user)
        db.session.commit()

    # Attempt login
    response = client.post('/api/auth/login', json={
        'email': 'test@example.com',
        'password': 'password123'
    })

    assert response.status_code == 200
    assert 'access_token' in response.json
    assert 'refresh_token' in response.json


def test_login_invalid_credentials(client):
    """Test login with invalid credentials"""
    response = client.post('/api/auth/login', json={
        'email': 'nonexistent@example.com',
        'password': 'wrongpassword'
    })

    assert response.status_code == 401
    assert response.json['error']['code'] == 'INVALID_CREDENTIALS'
```

---

9. Running the Application

Step 9.1: Flask Application Factory

`app/__init__.py`:

python

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_jwt_extended import JWTManager
from flask_cors import CORS

# Initialize extensions
db = SQLAlchemy()
migrate = Migrate()
jwt = JWTManager()


def create_app(config_object=None):
    """Application factory"""
    app = Flask(__name__)

    # Load configuration
    if config_object:
        app.config.from_object(config_object)
    else:
        from config.base import config
        env = app.config.get('ENV', 'development')
        app.config.from_object(config[env])

    # Initialize extensions
    db.init_app(app)
    migrate.init_app(app, db)
    jwt.init_app(app)
    CORS(app)

    # Register error handlers
    from app.utils.error_handler import register_error_handlers
    register_error_handlers(app)

    # Register blueprints
    from app.routes.auth import auth_bp
    from app.routes.users import users_bp
    from app.routes.applications import applications_bp
    from app.routes.dashboard import dashboard_bp

    app.register_blueprint(auth_bp)
    app.register_blueprint(users_bp)
    app.register_blueprint(applications_bp)
    app.register_blueprint(dashboard_bp)

    # Start background metrics collector (optional)
    # from app.utils.background_tasks import MetricsCollector
    # collector = MetricsCollector(app, interval=300)
    # collector = MetricsCollector(app, interval=300)
    # collector.start()

    return app
```

Step 9.2: Run Script

`run.py`:

```python
import os
from app import create_app, db

app = create_app()

if __name__ == '__main__':
    port = int(os.environ.get('PORT', 5000))
    debug = os.environ.get('DEBUG', 'True') == 'True'

    with app.app_context():
        # Create tables if they don't exist
        db.create_all()

    app.run(host='0.0.0.0', port=port, debug=debug)
```

Step 9.3: Initialize Database

```bash
# Make sure PostgreSQL is running and database is created

# Initialize migrations
flask db init

# Create migration
flask db migrate -m "Initial migration"

# Apply migration
flask db upgrade

# Or simply use run.py which creates tables automatically
python run.py
```

---

10. Sample Data

Step 10.1: Seed Script

`scripts/seed_data.py`:

python

```python
from app import create_app, db
from app.models import User, Application
from datetime import datetime, timedelta


def seed_database():
    """Seed the database with sample data"""

    app = create_app()

    with app.app_context():
        # Clear existing data
        print("Clearing existing data...")
        db.drop_all()
        db.create_all()

        # Create Applications
        print("Creating applications...")
        applications = [
            Application(
                name='Dashboard',
                description='Main admin dashboard',
                url='https://dashboard.example.com',
                status='active'
            ),
            Application(
                name='Region 14',
                description='Region 14 management system',
                url='https://region14.example.com',
                status='active'
            ),
            Application(
                name='Region 2',
                description='Region 2 management system',
                url='https://region2.example.com',
                status='active'
            ),
            Application(
                name='Analytics',
                description='Analytics and reporting platform',
                url='https://analytics.example.com',
                status='maintenance'
            ),
            Application(
                name='Legacy System',
                description='Old system being phased out',
                url='https://legacy.example.com',
        status='inactive'
    )
]

for app_item in applications:
    db.session.add(app_item)

db.session.commit()
print(f"Created {len(applications)} applications")

# Create Users
print("Creating users...")
users_data = [
    {
        'email': 'superadmin@example.com',
        'password': 'SuperAdmin123!',
        'role': 'superadmin',
        'status': 'active',
        'first_name': 'Super',
        'last_name': 'Admin',
        'apps': ['Dashboard', 'Region 14', 'Region 2', 'Analytics']
    },
    {
        'email': 'admin@example.com',
        'password': 'Admin123!',
        'role': 'admin',
        'status': 'active',
        'first_name': 'John',
        'last_name': 'Administrator',
        'apps': ['Dashboard', 'Region 14']
    },
    {
        'email': 'user1@example.com',
        'password': 'User123!',
        'role': 'user',
        'status': 'active',
        'first_name': 'Alice',
        'last_name': 'Johnson',
        'apps': ['Region 14']
    },
    {
        'email': 'user2@example.com',
        'password': 'User123!',
        'role': 'user',
        'status': 'active',
        'first_name': 'Bob',
        'last_name': 'Smith',
        'apps': ['Region 2', 'Analytics']
    },
    {
        'email': 'inactive@example.com',
        'password': 'User123!',
        'role': 'user',
        'status': 'inactive',
        'first_name': 'Charlie',
        'last_name': 'Inactive',
        'apps': []
    },
    {
        'email': 'newuser@example.com',
        'password': 'User123!',
        'role': 'user',
        'status': 'active',
        'first_name': 'Diana',
        'last_name': 'New',
        'apps': ['Dashboard']
    }
]

for user_data in users_data:
    user = User(
        email=user_data['email'],
        role=user_data['role'],
        status=user_data['status'],
        first_name=user_data['first_name'],
        last_name=user_data['last_name']
    )
    user.set_password(user_data['password'])

    # Set last login for active users (random dates in last 30 days)
    if user_data['status'] == 'active':
        days_ago = hash(user_data['email']) % 30
        user.last_login = datetime.utcnow() - timedelta(days=days_ago)

    # Assign applications
    for app_name in user_data['apps']:
        app = Application.query.filter_by(name=app_name).first()
        if app:
            user.assigned_applications.append(app)

    db.session.add(user)

db.session.commit()
print(f"Created {len(users_data)} users")

print("\n✅ Database seeded successfully!")
print("\nSample credentials:")
print("Superadmin: superadmin@example.com / SuperAdmin123!")
print("Admin: admin@example.com / Admin123!")
```

```
    print("User: user1@example.com / User123!")


if __name__ == '__main__':
    seed_database()
```

Step 10.2: Run Seed Script

```bash
python scripts/seed_data.py
```

🎯 Quick Start Commands

```bash
# 1. Setup
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
cp .env.example .env
# Edit .env with your database credentials

# 2. Create database
createdb admin_dashboard

# 3. Initialize migrations
flask db init
flask db migrate -m "Initial migration"
flask db upgrade

# 4. Seed sample data
python scripts/seed_data.py

# 5. Run application
python run.py

# Application will be available at http://localhost:5000
```

📡 API Testing Examples

Login

```bash
curl -X POST http://localhost:5000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "email": "admin@example.com",
    "password": "Admin123!"
  }'
```

Get Users (with token)

```bash
curl -X GET "http://localhost:5000/api/users?page=1&per_page=10" \
  -H "Authorization: Bearer YOUR_ACCESS_TOKEN"
```

Create User

```bash
curl -X POST http://localhost:5000/api/users \
  -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "email": "newuser@example.com",
    "password": "Password123!",
    "role": "user",
    "first_name": "New",
    "last_name": "User",
    "application_ids": [1, 2]
  }'
```

Dashboard Stats

```bash
curl -X GET http://localhost:5000/api/dashboard/stats \
  -H "Authorization: Bearer YOUR_ACCESS_TOKEN"
```

System Health

```bash
curl -X GET http://localhost:5000/api/dashboard/health \
  -H "Authorization: Bearer YOUR_ACCESS_TOKEN"
```

🎉 Next Steps After MVP

1. **Frontend Development**: Build React/Vue dashboard to consume these APIs
2. **Advanced Features**:
   - Password reset via email
   - Rate limiting
   - API documentation (Swagger)
   - File uploads
   - Export to CSV/Excel
3. **Deployment**: Docker, nginx, gunicorn
4. **Monitoring**: Sentry, logging infrastructure
5. **Testing**: Expand test coverage to 80%+

📚 Project Structure Summary

```
flask-admin-dashboard/
├── app/
│   ├── __init__.py          # App factory
│   ├── models/              # Database models
│   │   ├── __init__.py
│   │   ├── user.py
│   │   ├── application.py
│   │   ├── activity.py
│   │   └── metrics.py
│   ├── routes/              # API endpoints
│   │   ├── auth.py
│   │   ├── users.py
│   │   ├── applications.py
│   │   └── dashboard.py
│   ├── schemas/             # Pydantic validation schemas
│   │   ├── user_schema.py
│   │   └── application_schema.py
│   ├── utils/               # Utilities
│   │   ├── error_handler.py
│   │   ├── monitoring.py
│   │   ├── validation.py    # Pydantic helpers
│   │   └── background_tasks.py
│   └── middleware/          # Middleware
│       └── activity_logger.py
├── config/
│   └── base.py              # Configuration
├── migrations/              # Database migrations
├── scripts/
│   └── seed_data.py         # Sample data seeder
├── tests/
│   ├── conftest.py
│   ├── unit/
│   └── integration/
├── .env                     # Environment variables
├── .env.example
├── .gitignore
├── requirements.txt
└── run.py                   # Entry point
```

✅ Checklist

- [ ] Install dependencies
- [ ] Create PostgreSQL database
- [ ] Configure .env file
- [ ] Initialize database migrations
- [ ] Run migrations
- [ ] Seed sample data
- [ ] Start application
- [ ] Test authentication endpoint
- [ ] Test user CRUD endpoints
- [ ] Test application CRUD endpoints
- [ ] Test dashboard endpoints
- [ ] Review API documentation
- [ ] Write additional tests
- [ ] Deploy to staging

🚀 Why Pydantic?

Feature	Benefit
Better Type Hints	Native Python type hints with full IDE support
Faster	Written in Rust (Pydantic v2) - 3-5x faster than alternatives
Modern Syntax	More Pythonic, cleaner validation decorators
Better Errors	Detailed, structured error messages
No load/dump	Direct object instantiation, simpler to use

Example:

```python
# Pydantic approach
@validate_json_body(LoginSchema)
def login(validated_data: LoginSchema):
    email = validated_data.email  # ✅ Attribute access with autocomplete!
    # Data is already validated and typed
```

You now have a complete, production-ready Flask admin dashboard backend with Pydantic validation! 🚀

Follow the steps in order, and you'll have a working API in under an hour.