¡Excelente elección! Usaremos **Python con FastAPI**.

¿Por qué FastAPI? Porque es el estándar moderno para **Backends Científicos** (perfecto para tu *Pyrolysis Hub*) y tiene un rendimiento increíble para manejar peticiones asíncronas.

A continuación, presento el código completo para una implementación profesional simulada. Este código no solo hace "login", sino que implementa la lógica de **JWS (JSON Web Signature)** con la **versión del token** que definimos en la base de datos para máxima seguridad.

Requisitos Previos

Necesitarás instalar las siguientes librerías:

Bash
pip install fastapi uvicorn sqlalchemy passlib[bcrypt] python-jose

El Código (`main.py`)

Este script incluye la configuración de la base de datos (simulada con SQLite para que puedas ejecutarlo ya, pero mapeada a tu esquema PostgreSQL), el hasheo de contraseñas y la generación del token seguro.

Python
```
import time
from typing import List, Optional
from datetime import datetime, timedelta
from uuid import uuid4

from fastapi import FastAPI, HTTPException, Depends, status
from pydantic import BaseModel, EmailStr
from sqlalchemy import create_engine, Column, String, Integer, Boolean, ForeignKey, Table
from sqlalchemy.orm import sessionmaker, relationship, Session, declarative_base
from passlib.context import CryptContext
from jose import jwt, JWTError

# --- CONFIGURACIÓN & SECRETOS ---
# En producción, esto va en variables de entorno (.env)
SECRET_KEY = "tu_super_secreto_nexo_sinergico_key"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60

# Configuración DB (Usamos SQLite en memoria para esta demo, cambiable a PostgreSQL)
```

```
DATABASE_URL = "sqlite:///./nexo_sinergico.db"
# Para Postgres sería: "postgresql://user:pass@localhost/db_name"

engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# --- 1. MODELOS SQLALCHEMY (Reflejan tu esquema SQL) ---

# Tabla intermedia
user_roles = Table('user_roles', Base.metadata,
    Column('user_id', String, ForeignKey('users.id')),
    Column('role_id', Integer, ForeignKey('roles.id'))
)

class Role(Base):
    __tablename__ = "roles"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, unique=True, index=True)
    description = Column(String)

class User(Base):
    __tablename__ = "users"
    # Usamos String para UUID en SQLite, en Postgres usarías UUID nativo
    id = Column(String, primary_key=True, default=lambda: str(uuid4()))
    email = Column(String, unique=True, index=True)
    full_name = Column(String)
    password_hash = Column(String)
    is_active = Column(Boolean, default=True)
    token_version = Column(Integer, default=1) # CLAVE para seguridad

    # Relación Many-to-Many
    roles = relationship("Role", secondary=user_roles, backref="users")

# --- 2. UTILIDADES DE SEGURIDAD ---

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

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)
```

```
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)

    # JWS Standard Claims + Custom Claims
    to_encode.update({"exp": expire})

    # Firmamos el token
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

# --- 3. SCHEMAS PYDANTIC (Validación de entrada/salida) ---

class LoginRequest(BaseModel):
    email: EmailStr
    password: str

class TokenResponse(BaseModel):
    access_token: str
    token_type: str
    roles: List[str]
    user_name: str

# --- 4. API & ENDPOINTS ---

app = FastAPI(title="Nexo Sinérgico Auth System")

# Dependencia para obtener la DB
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.post("/auth/login", response_model=TokenResponse)
def login_for_access_token(form_data: LoginRequest, db: Session = Depends(get_db)):
    """
    Endpoint oficial de Login.
    Verifica credenciales y emite un JWS firmado con la versión del token del usuario.
    """
```

```
    # 1. Buscar usuario por email
    user = db.query(User).filter(User.email == form_data.email).first()

    # 2. Validaciones de Seguridad
    if not user:
        raise HTTPException(status_code=401, detail="Credenciales incorrectas")

    if not verify_password(form_data.password, user.password_hash):
        raise HTTPException(status_code=401, detail="Credenciales incorrectas")

    if not user.is_active:
        raise HTTPException(status_code=403, detail="Usuario inactivo. Contacte al Admin.")

    # 3. Preparar Payload del Token (La 'Fusión de Datos')
    # Extraemos los nombres de los roles para enviarlos al Frontend
    role_names = [role.name for role in user.roles]

    token_payload = {
        "sub": user.id,            # Subject (ID del usuario)
        "email": user.email,       # Contexto
        "roles": role_names,       # Para control de acceso en Frontend
        "ver": user.token_version  # CRÍTICO: Versión actual del token para invalidación
    }

    # 4. Generar JWS
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data=token_payload, expires_delta=access_token_expires
    )

    # 5. Respuesta
    return {
        "access_token": access_token,
        "token_type": "bearer",
        "roles": role_names,
        "user_name": user.full_name
    }

# --- ENDPOINT AUXILIAR PARA CREAR USUARIO DE PRUEBA (SOLO DEMO) ---
@app.post("/setup-demo")
def setup_demo(db: Session = Depends(get_db)):
    # Crear tablas
    Base.metadata.create_all(bind=engine)
```

```
    # Verificar si ya existe
    if db.query(User).filter(User.email == "cientifico@nexo.com").first():
        return {"msg": "Datos de demostración ya existen"}

    # Crear Roles
    r_admin = Role(name="Admin", description="Total control")
    r_academic = Role(name="Academico", description="Acceso científico")
    r_colab = Role(name="Colaborador", description="Acceso creativo")

    db.add_all([r_admin, r_academic, r_colab])
    db.commit()

    # Crear Usuario Híbrido (Científico que también crea contenido)
    hashed_pw = get_password_hash("ciencia123")
    user = User(
        email="cientifico@nexo.com",
        full_name="Dr. Nexo",
        password_hash=hashed_pw
    )
    # Asignar roles
    user.roles = [r_academic, r_colab]

    db.add(user)
    db.commit()

    return {"msg": "Usuario Demo Creado: cientifico@nexo.com / ciencia123"}
```

Análisis del "Nexo Sinérgico" en este código

Este no es un login genérico. Mira los detalles clave:

1. **El Campo `ver` (Token Version):** En la línea `token_payload`, incluimos `"ver": user.token_version`.
    - *El Truco de Seguridad:* Si mañana despides a un operador o un usuario reporta robo de cuenta, simplemente vas a la base de datos y cambias su `token_version` de 1 a 2.
    - Todos los tokens viejos (que tienen `ver: 1` dentro) serán rechazados automáticamente por tus *middlewares* de protección, aunque la fecha de expiración aún sea válida. ¡Invalidación instantánea sin borrar tokens!
2. **Carga de Roles en el Login:** El endpoint devuelve `roles: ["Academico", "Colaborador"]`.

    - Esto permite a tu Frontend (React/Vue/Angular) decidir inmediatamente: *"Ah, este es un Académico, muestro el dashboard de Pyrolysis Hub"* Y *"También es Colaborador, habilito el botón de 'Crear Narrativa'"*.
3. **Hash Seguro:** Se usa `bcrypt` a través de `passlib`. Nunca guardamos texto plano. Esto cumple con estándares industriales.

Cómo probarlo

1. Ejecuta el servidor: `uvicorn main:app --reload`
2. Ve a `http://127.0.0.1:8000/docs` (Swagger UI automático de FastAPI).
3. Ejecuta el endpoint `/setup-demo` para crear la DB y el usuario.
4. Usa el endpoint `/auth/login` con:
    - Email: `cientifico@nexo.com`
    - Password: `ciencia123`

Recibirás un Token JWS real.