Building a Comprehensive FastAPI Application with PostgreSQL, OAuth2, and NiceGUI

I. Introduction

Purpose

This report provides a comprehensive guide and reference implementation for developing a robust FastAPI application. It details the integration of an asynchronous PostgreSQL database hosted on Azure, the implementation of full Create, Read, Update, Delete (CRUD) operations for multiple data models using SQLAlchemy's asynchronous capabilities, user authentication via OAuth2 social logins (Google, Facebook, Microsoft), and the creation of a basic web-based frontend using NiceGUI for data interaction. The structure and implementation follow specific user requirements for project organization and functionality.

Technology Stack Overview

The application leverages a modern, asynchronous Python technology stack designed for performance and developer efficiency:

- **FastAPI:** A high-performance web framework for building APIs with Python 3.7+ based on standard Python type hints. Its asynchronous nature makes it suitable for I/O-bound tasks like database interactions and external API calls.[1] FastAPI's design emphasizes ease of use, speed, and automatic interactive documentation generation.[1]
- **SQLAlchemy (Async):** The premier SQL toolkit and Object-Relational Mapper (ORM) for Python. Version 2.0 provides a mature asynchronous interface, allowing non-blocking database operations crucial for high-concurrency applications.[1] It maps Python classes to database tables, enabling developers to work with database records as objects.[1]
- **Asyncpg:** A high-performance asynchronous PostgreSQL database client library for Python's asyncio framework. It is the recommended driver for use with SQLAlchemy's async PostgreSQL dialect due to its speed and native async support.[3]
- **PostgreSQL (Azure):** A powerful, open-source object-relational database system. Hosting on Azure provides scalability, reliability, and managed services. Specific connection requirements, such as SSL, must be handled.[9]
- **Pydantic:** A data validation and settings management library using Python type annotations. FastAPI uses Pydantic models extensively to define request and response data structures, providing automatic data validation, serialization, and

documentation.[10]

- **OAuth2 Libraries (fastapi-sso, Authlib):** Libraries designed to simplify the implementation of OAuth2 flows. These handle the complexities of redirecting users to identity providers (Google, Facebook, Microsoft), processing callbacks, exchanging codes for tokens, and fetching user information, enabling secure social logins.[13]
- **NiceGUI:** A Python library for building web-based graphical user interfaces. It allows developers to create interactive UIs using only Python, integrating well with frameworks like FastAPI for backend operations.[19] It uses FastAPI, Vue.js, and Tailwind CSS under the hood.[19]

Key Features

The application encompasses the following core functionalities:

1. **Structured API:** Asynchronous RESTful API endpoints for CRUD operations on ten distinct database models (act, book, image, location, thing, persona, scene, story, user, world).
2. **Database Integration:** Asynchronous connection and interaction with an Azure PostgreSQL database using SQLAlchemy and asyncpg.
3. **Data Modeling:** Definition of SQLAlchemy ORM models with UUID primary keys automatically generated by the database and server-managed creation/update timestamps.
4. **Social Authentication:** Secure user login via external OAuth2 providers (Google, Facebook, Microsoft).
5. **User Profile Management:** Collection and storage of basic user profile information (Email, First Name, Last Name) after initial OAuth authentication.
6. **Basic Frontend:** A simple, interactive web interface built with NiceGUI, allowing authenticated users to view and perform basic edits on the database data through the API.
7. **Specific Project Structure:** Adherence to a predefined directory layout separating models, schemas, CRUD logic, and API endpoints for enhanced modularity and maintainability.

II. Project Setup and Configuration

Establishing a solid foundation is crucial for building a maintainable and scalable application. This section details the project structure, dependencies, configuration management, and database connection setup.

A. Finalized Directory Structure

A well-defined project structure promotes separation of concerns, making the codebase easier to understand, navigate, and maintain. The following structure adheres to the user's requirements and aligns with common practices like the Repository Pattern or Clean Architecture, separating data persistence, business logic, and API presentation layers.[21]

Table: Project Directory Structure

Path	Description
/	Project Root
├── .env	Environment variables (Database URL, secrets, etc.) - **DO NOT COMMIT**
├── pyproject.toml	Project metadata and dependencies (using Poetry)
├── main.py	Main FastAPI application instance, middleware, router inclusion, NiceGUI init
├── core/	Core application logic and configuration
│ ├── __init__.py	
│ ├── config.py	Pydantic settings management (loads .env)
│ └── database.py	SQLAlchemy async engine, session factory, get_db dependency
├── models/	SQLAlchemy ORM models
│ ├── __init__.py	
│ ├── base.py	Declarative base, common mixins (optional)
│ ├── models_act.py	Act model definition
│ ├── models_book.py	Book model definition

│ ├──... (etc.)	(One file per model: image, location, thing, persona, scene, story, user, world)
├── schemas/	Pydantic schemas for data validation and serialization
│ ├── __init__.py	
│ ├── schemas_act.py	Act schemas (Base, Create, Update, Read)
│ ├── schemas_book.py	Book schemas
│ ├──... (etc.)	(One file per model)
├── crud/	Data Access Layer (Repository Pattern)
│ ├── __init__.py	
│ ├── base.py	Generic Base Repository class (optional but recommended)
│ ├── crud_act.py	Act specific CRUD functions/repository class
│ ├── crud_book.py	Book specific CRUD functions/repository class
│ ├──... (etc.)	(One file per model)
├── EndPoint/	FastAPI API Routers
│ ├── __init__.py	
│ ├── ep_act.py	APIRouter for Act CRUD endpoints
│ ├── ep_book.py	APIRouter for Book CRUD endpoints
│ ├──... (etc.)	(One file per model)
├── auth/	Authentication related logic

│ ├── __init__.py	
│ ├── router.py	APIRouter for OAuth login/callback endpoints
│ └── dependencies.py	Security dependencies (e.g., get_current_user)
└── ui/	NiceGUI page definitions (optional, can start in main.py)
├── __init__.py	
└── pages.py	NiceGUI page functions (@ui.page)

This structure clearly separates database models (models), API data contracts (schemas), data access logic (crud), and API route definitions (EndPoint), facilitating independent development and testing of different application layers.

B. Required Dependencies

The following Python packages are required. They can be managed using pip with a requirements.txt file or, preferably, using a dependency manager like Poetry via pyproject.toml.

- fastapi: The core web framework.[1]
- uvicorn[standard]: The ASGI server to run FastAPI.[2]
- sqlalchemy[asyncio]: The ORM toolkit with async support enabled.[1]
- asyncpg: Asynchronous PostgreSQL driver.[3]
- pydantic: Data validation and settings management.[10]
- pydantic-settings: For loading configuration from environment variables (used with Pydantic v2).
- python-dotenv: To load environment variables from a .env file during development.
- httpx: An asynchronous HTTP client, needed for making API calls from NiceGUI and potentially for OAuth interactions.[15]
- nicegui: The frontend UI library.[19]
- fastapi-sso: A library simplifying OAuth2 social login integration.[14] (Alternatively, Authlib [16] could be used).
- python-jose[cryptography]: For handling JWTs if implementing token-based sessions after OAuth.[16]
- passlib[bcrypt]: For password hashing if local password authentication were

added (useful even if starting with only OAuth).[26]

- email-validator: Often required by Pydantic for email string validation.

A sample pyproject.toml section using Poetry:

Ini, TOML

```toml
[tool.poetry.dependencies]
python = "^3.9"
fastapi = "^0.110.0"
uvicorn = {extras = ["standard"], version = "^0.29.0"}
sqlalchemy = {extras = ["asyncio"], version = "^2.0.29"}
asyncpg = "^0.29.0"
pydantic = "^2.7.0"
pydantic-settings = "^2.2.1"
python-dotenv = "^1.0.1"
httpx = "^0.27.0"
nicegui = "^1.4.14"
fastapi-sso = "^0.16.0" # Or authlib
python-jose = {extras = ["cryptography"], version = "^3.3.0"}
passlib = {extras = ["bcrypt"], version = "^1.7.4"}
email-validator = "^2.1.1"
```

C. Environment Variable Management (core/config.py)

Hardcoding sensitive information like database credentials or API secrets directly into the source code is a significant security risk and hinders deployment flexibility. A standard practice is to use environment variables, managed via a .env file for local development and set directly in the deployment environment for production. Pydantic's BaseSettings provides a convenient way to load and validate these settings.

Python

```python
# core/config.py
```

```python
import os
from pydantic_settings import BaseSettings
from pydantic import Field, PostgresDsn, AnyHttpUrl
from typing import List, Union
from pathlib import Path

# Load.env file from project root
env_path = Path(".") / ".env"
if env_path.is_file():
    from dotenv import load_dotenv
    load_dotenv(dotenv_path=env_path)

class Settings(BaseSettings):
    # Database Configuration
    DB_USER: str = Field(..., validation_alias='POSTGRES_USER')
    DB_PASSWORD: str = Field(..., validation_alias='POSTGRES_PASSWORD')
    DB_HOST: str = Field(..., validation_alias='POSTGRES_SERVER')
    DB_PORT: int = Field(default=5432, validation_alias='POSTGRES_PORT')
    DB_NAME: str = Field(..., validation_alias='POSTGRES_DB')
    # Construct PostgreSQL DSN including SSL requirement for Azure
    DATABASE_URL: Union[PostgresDsn, str] = ""

    # OAuth2 Configuration (using fastapi-sso examples)
    GOOGLE_CLIENT_ID: str = Field(..., validation_alias='GOOGLE_CLIENT_ID')
    GOOGLE_CLIENT_SECRET: str = Field(..., validation_alias='GOOGLE_CLIENT_SECRET')
    FACEBOOK_CLIENT_ID: str | None = Field(default=None,
validation_alias='FACEBOOK_CLIENT_ID')
    FACEBOOK_CLIENT_SECRET: str | None = Field(default=None,
validation_alias='FACEBOOK_CLIENT_SECRET')
    MICROSOFT_CLIENT_ID: str | None = Field(default=None,
validation_alias='MICROSOFT_CLIENT_ID')
    MICROSOFT_CLIENT_SECRET: str | None = Field(default=None,
validation_alias='MICROSOFT_CLIENT_SECRET')
    # Define redirect URIs - these must match provider configuration
    # Example: http://localhost:8000/auth/google/callback
    GOOGLE_REDIRECT_URI: AnyHttpUrl | None = Field(default=None,
validation_alias='GOOGLE_REDIRECT_URI')
    FACEBOOK_REDIRECT_URI: AnyHttpUrl | None = Field(default=None,
validation_alias='FACEBOOK_REDIRECT_URI')
```

```
    MICROSOFT_REDIRECT_URI: AnyHttpUrl | None = Field(default=None,
validation_alias='MICROSOFT_REDIRECT_URI')

    # Application Secrets
    SESSION_SECRET_KEY: str = Field(..., validation_alias='SESSION_SECRET_KEY') # For
session middleware / NiceGUI storage
    JWT_SECRET_KEY: str = Field(..., validation_alias='JWT_SECRET_KEY') # If using JWTs after
OAuth
    ALGORITHM: str = "HS256" # Algorithm for JWT

    # CORS Origins (adjust as needed for frontend deployment)
    BACKEND_CORS_ORIGINS: List[str] = ["http://localhost:8000", "http://localhost:8080",
"http://127.0.0.1:8000", "http://127.0.0.1:8080"]

    class Config:
        env_file = '.env'
        env_file_encoding = 'utf-8'
        case_sensitive = True
        # Allow extra fields if needed, though BaseSettings usually handles this
        # extra = 'ignore'

    def __init__(self, **values):
        super().__init__(**values)
        # Construct DATABASE_URL after loading other DB variables
        self.DATABASE_URL =
f"postgresql+asyncpg://{self.DB_USER}:{self.DB_PASSWORD}@{self.DB_HOST}:{self.DB_PORT}/{self.DB
_NAME}?ssl=require"

settings = Settings()
```

Create a .env file in the project root (and add it to .gitignore):

Code snippet

```
#.env - DO NOT COMMIT THIS FILE
POSTGRES_USER=testuser
```

```
POSTGRES_PASSWORD=testuser123
POSTGRES_SERVER=hack1database.postgres.database.azure.com
POSTGRES_PORT=5432
POSTGRES_DB=testdb

# Get these from Google Cloud Console -> APIs & Services -> Credentials
GOOGLE_CLIENT_ID="YOUR_GOOGLE_CLIENT_ID"
GOOGLE_CLIENT_SECRET="YOUR_GOOGLE_CLIENT_SECRET"
GOOGLE_REDIRECT_URI="http://localhost:8000/auth/google/callback" # Or your deployed URI

# Get these from Meta for Developers
FACEBOOK_CLIENT_ID="YOUR_FACEBOOK_CLIENT_ID"
FACEBOOK_CLIENT_SECRET="YOUR_FACEBOOK_CLIENT_SECRET"
FACEBOOK_REDIRECT_URI="http://localhost:8000/auth/facebook/callback"

# Get these from Azure Portal -> App registrations
MICROSOFT_CLIENT_ID="YOUR_MICROSOFT_CLIENT_ID"
MICROSOFT_CLIENT_SECRET="YOUR_MICROSOFT_CLIENT_SECRET"
MICROSOFT_REDIRECT_URI="http://localhost:8000/auth/microsoft/callback"

# Generate strong random secrets for these
SESSION_SECRET_KEY="a_very_strong_random_secret_for_sessions"
JWT_SECRET_KEY="another_very_strong_random_secret_for_jwt"
```

Table: Environment Variables

Variable Name	**Description**	**Example Value (Masked/Placeholder)**
POSTGRES_USER	PostgreSQL database username	testuser
POSTGRES_PASSWORD	PostgreSQL database password	testuser123
POSTGRES_SERVER	PostgreSQL database host name (Azure)	hack1database.postgres.database.azure.com

POSTGRES_PORT	PostgreSQL database port	5432
POSTGRES_DB	PostgreSQL database name	testdb
GOOGLE_CLIENT_ID	Google OAuth2 Client ID	YOUR_GOOGLE_CLIENT_ID
GOOGLE_CLIENT_SECRET	Google OAuth2 Client Secret	YOUR_GOOGLE_CLIENT_SECRET
GOOGLE_REDIRECT_URI	Redirect URI registered with Google	http://localhost:8000/auth/google/callback
FACEBOOK_CLIENT_ID	Facebook OAuth2 Client ID	YOUR_FACEBOOK_CLIENT_ID
FACEBOOK_CLIENT_SECRET	Facebook OAuth2 Client Secret	YOUR_FACEBOOK_CLIENT_SECRET
FACEBOOK_REDIRECT_URI	Redirect URI registered with Facebook	http://localhost:8000/auth/facebook/callback
MICROSOFT_CLIENT_ID	Microsoft OAuth2 Client ID (Application ID)	YOUR_MICROSOFT_CLIENT_ID
MICROSOFT_CLIENT_SECRET	Microsoft OAuth2 Client Secret	YOUR_MICROSOFT_CLIENT_SECRET
MICROSOFT_REDIRECT_URI	Redirect URI registered with Microsoft	http://localhost:8000/auth/microsoft/callback
SESSION_SECRET_KEY	Secret key for signing session cookies (used by middleware/NiceGUI)	generate_a_strong_random_secret
JWT_SECRET_KEY	Secret key for signing JWTs (if used post-OAuth)	generate_another_strong_random_secret

This centralized configuration approach enhances security and simplifies deployment across different environments.

D. Asynchronous Database Engine and Session Setup (core/database.py)

Connecting to the database asynchronously requires careful setup of the engine and session management.

- **Database URL Construction:** The connection string needs the correct format for asyncpg and must include the ssl=require parameter for secure connections to Azure PostgreSQL.[9] While manual string formatting works, using SQLAlchemy's URL.create can be more robust, especially if passwords contain special characters that require URL encoding.[9] The Settings class above demonstrates constructing the URL string directly after loading components, which is often sufficient. Adding ?ssl=require is crucial for cloud database connections.
- **Async Engine Creation:** The create_async_engine function establishes the connection pool and dialect handling.[4] Setting echo=True during development is invaluable for debugging as it logs all generated SQL statements.[4] For production, consider tuning pool settings like pool_size, max_overflow, and especially pool_pre_ping=True.[6] Enabling pool_pre_ping adds a minimal check (e.g., SELECT 1) before lending a connection from the pool, ensuring its validity and preventing errors caused by stale connections, which can occur in long-running applications or due to network issues or database timeouts.[29] This proactive check enhances application resilience.
- **Async Session Factory:** async_sessionmaker provides a configured factory for creating AsyncSession instances.[4] Setting expire_on_commit=False is generally recommended for web applications using the session-per-request pattern. It prevents SQLAlchemy from expiring objects after a transaction commits, avoiding DetachedInstanceError if objects are accessed later in the request lifecycle (e.g., during response serialization).[31]
- **Dependency for Session Management:** FastAPI's dependency injection system is the standard way to manage resources like database sessions on a per-request basis.[1] An asynchronous generator function using yield ensures the session is created before the request handler runs and properly closed or rolled back afterwards, even if errors occur.[5]

Python

```python
# core/database.py
from collections.abc import AsyncGenerator
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker,
AsyncSession
```

```python
from sqlalchemy.orm import declarative_base
from sqlalchemy.exc import SQLAlchemyError
import logging

from.config import settings

logger = logging.getLogger(__name__)

# Define a base for declarative models
Base = declarative_base()

# Create the asynchronous engine
# echo=True is useful for development to see generated SQL
engine = create_async_engine(
    str(settings.DATABASE_URL), # Pydantic v2 returns URL object, ensure it's string
    echo=True, # Log SQL queries (consider False for production)
    pool_pre_ping=True, # Check connection validity before use
    # pool_size=10, # Example: Adjust pool size if needed
    # max_overflow=20, # Example: Adjust overflow if needed
)

# Create a configured "Session" class
# expire_on_commit=False prevents detached instance errors after commit
async_session_factory = async_sessionmaker(
    engine,
    class_=AsyncSession,
    expire_on_commit=False
)

async def get_db() -> AsyncGenerator:
    """
   FastAPI dependency that provides an asynchronous database session
   per request. Ensures the session is closed afterwards.
   """
    async with async_session_factory() as session:
        try:
            yield session
            await session.commit() # Commit if no exceptions occurred
        except SQLAlchemyError as e:
            logger.error(f"Database error occurred: {e}")
```

```
            await session.rollback() # Rollback on error
            # Optionally re-raise or handle specific exceptions
            raise
        except Exception as e:
            logger.error(f"An unexpected error occurred during DB session: {e}")
            await session.rollback() # Rollback on any other error
            raise # Re-raise the exception
        finally:
            # The session is automatically closed by the async context manager
            # await session.close() # Not strictly needed with 'async with'
            pass

# Optional: Function to create tables (can be called from lifespan event or migration tool)
async def create_db_and_tables():
    async with engine.begin() as conn:
        # Ensure pgcrypto extension exists (as requested in user query)
        # Note: Running CREATE EXTENSION might require superuser privileges
        # It's often better to ensure this is done manually or via infrastructure setup.
        try:
            await conn.execute(text('CREATE EXTENSION IF NOT EXISTS "pgcrypto";'))
        except Exception as e:
            logger.warning(f"Could not ensure pgcrypto extension exists (might require DB admin): {e}")

        # Create all tables defined inheriting from Base
        await conn.run_sync(Base.metadata.create_all)
```

This setup provides a robust and efficient mechanism for handling asynchronous database connections and sessions within the FastAPI application.

E. Main Application Initialization (main.py)

The main.py file serves as the entry point for the application, bringing together configuration, middleware, routers, and the NiceGUI integration.

Python

```
# main.py
```

```
from fastapi import FastAPI, Request, Depends
from fastapi.middleware.cors import CORSMiddleware
from starlette.middleware.sessions import SessionMiddleware # Needed for fastapi-sso state
from contextlib import asynccontextmanager
import uvicorn
import logging

from core.config import settings
from core.database import create_db_and_tables, engine # Import engine for disposal
# Import API routers (assuming they are defined in EndPoint/*.py)
from EndPoint import ep_act, ep_book, ep_image, ep_location, ep_thing, ep_persona,
ep_scene, ep_story, ep_user, ep_world
from auth.router import router as auth_router
# Import NiceGUI
from nicegui import ui, app as nicegui_app
# Import NiceGUI UI pages (example)
# from ui import pages

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@asynccontextmanager
async def lifespan(app: FastAPI):
    """
    Application lifespan manager. Executes startup and shutdown events.
    """
    logger.info("Application startup...")
    # Optional: Create database tables on startup (consider Alembic for production)
    # Be cautious with create_all in production environments.
    # await create_db_and_tables()
    # logger.info("Database tables checked/created.")
    yield
    logger.info("Application shutdown...")
    # Clean up resources, like closing the database engine's connection pool
    await engine.dispose()
    logger.info("Database engine disposed.")

app = FastAPI(
    title="Comprehensive FastAPI Application",
    description="API for managing various world-building elements with OAuth2 and NiceGUI
```

```python
frontend.",
    version="0.1.0",
    lifespan=lifespan
)

# --- Middleware Configuration ---

# CORS Middleware: Allows requests from specified origins (e.g., your frontend)
app.add_middleware(
    CORSMiddleware,
    allow_origins=,
    allow_credentials=True,
    allow_methods=["*"], # Allows all methods
    allow_headers=["*"], # Allows all headers
)

# Session Middleware: Required by fastapi-sso to store OAuth state temporarily
# Ensure SESSION_SECRET_KEY is set in your.env file
app.add_middleware(
    SessionMiddleware,
    secret_key=settings.SESSION_SECRET_KEY
)

# --- API Router Integration ---

# Include authentication routes
app.include_router(auth_router, prefix="/auth", tags=["Authentication"])

# Include CRUD endpoints for each model
api_prefix = "/api/v1" # Optional global prefix for API routes
app.include_router(ep_act.router, prefix=f"{api_prefix}/acts", tags=["Acts"])
app.include_router(ep_book.router, prefix=f"{api_prefix}/books", tags=)
app.include_router(ep_image.router, prefix=f"{api_prefix}/images", tags=["Images"])
app.include_router(ep_location.router, prefix=f"{api_prefix}/locations", tags=["Locations"])
app.include_router(ep_thing.router, prefix=f"{api_prefix}/things", tags=)
app.include_router(ep_persona.router, prefix=f"{api_prefix}/personas", tags=["Personas"])
app.include_router(ep_scene.router, prefix=f"{api_prefix}/scenes", tags=)
app.include_router(ep_story.router, prefix=f"{api_prefix}/stories", tags=)
app.include_router(ep_user.router, prefix=f"{api_prefix}/users", tags=["Users"])
app.include_router(ep_world.router, prefix=f"{api_prefix}/worlds", tags=["Worlds"])
```

```python
# --- NiceGUI Integration ---

# Define NiceGUI UI pages (can be in main.py or imported from ui/pages.py)
@ui.page('/')
def main_page() -> None:
    ui.label('Welcome to the Application').classes('text-h4')
    ui.link('View Acts', '/ui/acts')
    # Add links to other UI pages as needed

    # Example: Display Login Buttons
    with ui.row().classes('mt-4'):
        ui.button('Login with Google', on_click=lambda: ui.open('/auth/google/login',
new_tab=False))
        # Add buttons for Facebook and Microsoft similarly if configured
        if settings.FACEBOOK_CLIENT_ID:
            ui.button('Login with Facebook', on_click=lambda: ui.open('/auth/facebook/login',
new_tab=False))
        if settings.MICROSOFT_CLIENT_ID:
            ui.button('Login with Microsoft', on_click=lambda: ui.open('/auth/microsoft/login',
new_tab=False))

# Example placeholder for a data display page
@ui.page('/ui/acts')
async def acts_page():
    ui.label('Acts Management').classes('text-h4')
    # UI elements to display and interact with Acts data will go here
    # This page will need to make calls to the /api/v1/acts endpoints
    ui.label('Act data will be displayed here.')
    # TODO: Implement table display and CRUD interactions using httpx

# Mount NiceGUI
# Ensure SESSION_SECRET_KEY is set for storage_secret
ui.run_with(
    app,
    mount_path="/ui", # Serve NiceGUI at the /ui path
    storage_secret=settings.SESSION_SECRET_KEY,
    title="App Frontend"
)
```

```python
# --- Root Endpoint (Optional - for basic API check) ---
@app.get("/")
async def root():
    return {"message": "API is running. Access NiceGUI at /ui"}

# --- Uvicorn Runner (for direct execution) ---
if __name__ == "__main__":
    uvicorn.run(
        "main:app",
        host="0.0.0.0",
        port=8000,
        reload=True # Enable auto-reload for development
    )
```

This main.py sets up the FastAPI application, configures necessary middleware (CORS for frontend access, SessionMiddleware for OAuth state), integrates all the API routers defined in the EndPoint/ directory, and mounts the NiceGUI interface at the /ui path. The lifespan manager ensures proper startup and shutdown procedures, including database engine disposal.[5]

III. Database Modeling with SQLAlchemy (models/)

Defining accurate and efficient database models is fundamental. SQLAlchemy's ORM allows mapping Python classes directly to database tables, simplifying data interaction.

A. Base Model Configuration (models/base.py)

Using a declarative base is standard practice. SQLAlchemy 2.0 encourages using DeclarativeBase. A common pattern is to define mixin classes to add recurring columns like primary keys and timestamps, promoting DRY (Don't Repeat Yourself) principles.[35]

Python

```python
# models/base.py
import uuid
```

```python
from datetime import datetime
from sqlalchemy import Column, DateTime, func, text
from sqlalchemy.dialects.postgresql import UUID as PG_UUID # Use specific dialect type
from sqlalchemy.orm import declarative_base, declared_attr, Mapped, mapped_column

# Base class using SQLAlchemy 2.0 style
class Base(declarative_base()):
    # Optional: Define a type map for common Python types to SQL types
    # type_annotation_map = {
    #     int: Integer,
    #     str: String(255),
    #     datetime: DateTime(timezone=True),
    #     uuid.UUID: PG_UUID(as_uuid=True),
    # }
    pass

# Mixin for UUID primary key using server-side generation
class UUIDPrimaryKeyMixin:
    # Use Mapped and mapped_column for SQLAlchemy 2.0 style
    id: Mapped[uuid.UUID] = mapped_column(
        PG_UUID(as_uuid=True),
        primary_key=True,
        server_default=text("gen_random_uuid()") # Use PostgreSQL function
    )

# Mixin for timestamp columns using server-side defaults
class TimestampMixin:
    # Use Mapped and mapped_column
    created_at: Mapped[datetime] = mapped_column(
        DateTime(timezone=False), # Match the schema definition (timestamp(0))
        server_default=func.now(), # Database server's current time on insert
        nullable=False
    )
    updated_at: Mapped[datetime] = mapped_column(
        DateTime(timezone=False), # Match the schema definition (timestamp(0))
        server_default=func.now(), # Database server's current time on insert
        onupdate=func.now(),      # Update timestamp on record update
        nullable=False
    )
```

```
# Optional: Define a common base class incorporating mixins
# class AppBaseModel(Base, UUIDPrimaryKeyMixin, TimestampMixin):
#     __abstract__ = True # Make this an abstract base, not a table itself
#
#     # Auto-generate snake_case table name from class name (optional convenience)
#     @declared_attr.directive
#     def __tablename__(cls) -> str:
#         import re
#         name = cls.__name__
#         # Convert CamelCase to snake_case
#         name = re.sub('(.)([A-Z][a-z]+)', r'\1_\2', name)
#         name = re.sub('__([A-Z])', r'_\1', name)
#         name = re.sub('([a-z0-9])([A-Z])', r'\1_\2', name)
#         # Handle potential quoting needed for reserved words
#         if name in ["user", "location"]:
#             return f'"{name.lower()}"'
#         return name.lower()
```

Using mixins like UUIDPrimaryKeyMixin and TimestampMixin standardizes the definition of these common columns across all models. The server_default and onupdate parameters leverage the database's capabilities for generating UUIDs and timestamps, which is generally more efficient and reliable than application-side generation.[37]

B. Individual Model Definitions (models/models_*.py)

Each table provided in the user query needs a corresponding SQLAlchemy model defined in its own file within the models/ directory. These models inherit from the Base (or the combined AppBaseModel if used) and define the table structure, columns, and relationships.

Handling Quoted Names: Tables named location and user are potential SQL reserved words. Their __tablename__ must be explicitly quoted (e.g., __tablename__ = '"user"') to ensure SQLAlchemy generates correct SQL syntax.[39]

Primary Keys & Timestamps: The id, created_at, and updated_at columns should be defined using Mapped and mapped_column (SQLAlchemy 2.0 style) and leverage the mixins or be defined explicitly as shown in the mixins, using server_default and onupdate with database functions (text('gen_random_uuid()'), func.now()).[35] The pgcrypto extension must be enabled in the PostgreSQL database for gen_random_uuid() to work.

Relationships: Foreign key relationships should be mirrored using relationship().

Using back_populates establishes bidirectional links, preventing warnings and ensuring clarity.[41] The lazy loading strategy dictates how related objects are fetched. While 'select' (the default) performs lazy loading, 'selectin' loading is often a good choice for async applications as it fetches related items eagerly using a separate SELECT... WHERE IN (...) query, avoiding the N+1 problem without requiring complex JOINs in the initial query.[43]

Example Model (models/models_act.py):

Python

```python
# models/models_act.py
import uuid
from datetime import datetime
from sqlalchemy import Column, String, ForeignKey, DateTime, func, text
from sqlalchemy.orm import relationship, Mapped, mapped_column
from.base import Base, UUIDPrimaryKeyMixin, TimestampMixin # Import base and mixins

class Act(Base, UUIDPrimaryKeyMixin, TimestampMixin):
    __tablename__ = 'act' # Explicitly define table name

    # Columns defined via Mapped/mapped_column (SQLAlchemy 2.0)
    name: Mapped[str] = mapped_column(String(255), nullable=False, unique=True)
    display_name: Mapped[str | None] = mapped_column(String(255), nullable=True)
    description: Mapped[str | None] = mapped_column(String(255), nullable=True)

    # Foreign Keys
    storyid: Mapped[uuid.UUID] = mapped_column(ForeignKey('story.id'), nullable=False)
    ownerid: Mapped[uuid.UUID] = mapped_column(ForeignKey('"user".id'),
nullable=False) # Reference quoted table name

    # Relationships
    # Establish relationship to Story model
    story: Mapped = relationship(back_populates="acts", lazy="selectin")
    # Establish relationship to User model (owner)
    owner: Mapped["User"] = relationship(back_populates="acts", lazy="selectin")
```

```
    # Relationship back from Scene (one-to-many)
    scenes: Mapped] = relationship(back_populates="act", lazy="selectin", cascade="all,
delete-orphan")

    def __repr__(self):
        return f"<Act(id={self.id}, name='{self.name}')>"

# Add forward references for type hinting if needed at the end of files or use string quotes
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from.models_story import Story
    from.models_user import User
    from.models_scene import Scene
```

Similarly, create models_book.py, models_image.py, models_location.py, models_thing.py, models_persona.py, models_scene.py, models_story.py, models_user.py, and models_world.py, defining their respective columns and relationships based on the provided SQL DDL. Remember to handle the quoted table names "location" and "user" correctly in ForeignKey and relationship definitions.

IV. Data Validation with Pydantic Schemas (schemas/)

Pydantic schemas define the expected structure and types for data entering (request bodies) and leaving (responses) the API. They provide automatic validation and serialization, integrating seamlessly with FastAPI.

A. Schema Structure (schemas/schemas_*.py)

For each model, it's best practice to define multiple schemas in a corresponding file (e.g., schemas_act.py) to handle different use cases:

- **SchemaBase:** Contains fields common to all operations (e.g., name, description).
- **SchemaCreate:** Inherits from SchemaBase. Includes fields required for creation, typically excluding database-generated fields like id, created_at, updated_at. It should include necessary foreign key fields (e.g., storyid, ownerid).
- **SchemaUpdate:** Inherits from SchemaBase (or a custom base). Fields are usually optional (Optional[...]) as updates might be partial.
- **SchemaRead (or Schema):** Inherits from SchemaBase. Includes all fields returned by the API, including id, created_at, updated_at, and potentially nested schemas for related objects.

Example Schemas (schemas/schemas_act.py):

Python

```python
# schemas/schemas_act.py
import uuid
from datetime import datetime
from pydantic import BaseModel, Field, ConfigDict
from typing import Optional, List

# --- Forward references for nested schemas ---
# Import related Read schemas here or use postponed evaluation (requires Python 3.7+)
# from.schemas_story import StoryRead
# from.schemas_user import UserRead
# from.schemas_scene import SceneRead

# --- Base Schema ---
class ActBase(BaseModel):
    name: str = Field(..., max_length=255)
    display_name: Optional[str] = Field(default=None, max_length=255)
    description: Optional[str] = Field(default=None, max_length=255)
    storyid: uuid.UUID
    ownerid: uuid.UUID

# --- Create Schema ---
# Fields required when creating a new Act via the API
class ActCreate(ActBase):
    pass # Inherits all fields from ActBase

# --- Update Schema ---
# Fields allowed when updating an Act via the API (all optional)
class ActUpdate(BaseModel):
    name: Optional[str] = Field(default=None, max_length=255)
    display_name: Optional[str] = Field(default=None, max_length=255)
    description: Optional[str] = Field(default=None, max_length=255)
    storyid: Optional[uuid.UUID] = None
    # ownerid might not be updatable depending on logic
    # ownerid: Optional[uuid.UUID] = None
```

```python
# --- Read Schema ---
# Fields returned when reading an Act from the API
class ActRead(ActBase):
    id: uuid.UUID
    created_at: datetime
    updated_at: datetime

    # Nested relationships (ensure related schemas also use from_attributes=True)
    # Use forward references (strings) if schemas are defined later or in different files
    # story: Optional = None # Example - uncomment and import if needed
    # owner: Optional = None # Example - uncomment and import if needed
    # scenes: List =   # Example - uncomment and import if needed

    # Enable ORM mode (Pydantic v2: from_attributes)
    model_config = ConfigDict(from_attributes=True)

# --- Update nested schemas if using forward references ---
# ActRead.model_rebuild() # Call after all related schemas are defined
```

B. ORM Compatibility

The key to bridging SQLAlchemy models and Pydantic schemas for API responses is the from_attributes=True setting (in Pydantic V2's model_config) or orm_mode = True (in Pydantic V1's Config class).[10] When FastAPI uses a Pydantic model with this setting as a response_model, it can automatically create an instance of the Pydantic schema directly from the attributes of the SQLAlchemy ORM object returned by the database query or CRUD operation. This significantly simplifies the process of converting database objects into JSON responses.

C. Field Types and Validation

Schema fields should use appropriate Python types (e.g., str, int, uuid.UUID, datetime.datetime, Optional[...], List[...]). Pydantic automatically validates incoming data against these types. Additional validation rules (e.g., max_length, min_length, constraints) can be added using pydantic.Field.[43]

D. Handling Relationships (Nested Schemas)

To represent relationships in API responses (e.g., showing the story details when fetching an act), the SchemaRead model should include fields typed with the corresponding SchemaRead of the related model.[41] For example, ActRead might have story: Optional = None. It is crucial that these nested schemas *also* have from_attributes=True enabled. When SQLAlchemy loads the relationship (e.g., via lazy='selectin'), the parent ORM object will have an attribute containing the related ORM object(s). Pydantic, guided by the nested schema definition and from_attributes=True, will then recursively populate the nested schema from the related object's attributes, resulting in the desired nested JSON output. Forward references (using string type hints like "StoryRead") might be necessary if schemas are defined across multiple files or depend on each other circularly; model_rebuild() can be called after all schemas are defined to resolve these references.

V. Data Access Layer: CRUD Operations (crud/)

Implementing a dedicated data access layer, often following the Repository Pattern, isolates database interaction logic from the API endpoint handlers. This improves code organization, testability, and maintainability.[7]

A. Generic Asynchronous Base Repository (crud/base.py)

Creating a generic base repository class significantly reduces repetitive code for standard CRUD operations.[23] This base class can be parameterized using Python's typing.TypeVar and typing.Generic to work with different SQLAlchemy models and Pydantic schemas.

Python

```python
# crud/base.py
from typing import Any, Dict, Generic, List, Optional, Type, TypeVar, Union
from uuid import UUID
from pydantic import BaseModel
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from sqlalchemy import update as sqlalchemy_update, delete as sqlalchemy_delete
from sqlalchemy.exc import NoResultFound

from models.base import Base # Import your SQLAlchemy Base

# Define TypeVars for generic repository
ModelType = TypeVar("ModelType", bound=Base)
```

```python
CreateSchemaType = TypeVar("CreateSchemaType", bound=BaseModel)
UpdateSchemaType = TypeVar("UpdateSchemaType", bound=BaseModel)

class CRUDBase(Generic):
    def __init__(self, model: Type):
        """
    CRUD object with default methods to Create, Read, Update, Delete (CRUD).

    **Parameters**

    * `model`: A SQLAlchemy model class
    """
        self.model = model

    async def get(self, db: AsyncSession, id: Union[UUID, int, str]) -> Optional:
        """Get a single record by ID."""
        stmt = select(self.model).where(self.model.id == id)
        result = await db.execute(stmt)
        return result.scalar_one_or_none()

    async def get_multi(
        self, db: AsyncSession, *, skip: int = 0, limit: int = 100
    ) -> List:
        """Get multiple records with pagination."""
        stmt = select(self.model).offset(skip).limit(limit)
        result = await db.execute(stmt)
        return result.scalars().all() # Use scalars() for single-column primary key or if selecting the
whole model

    async def create(self, db: AsyncSession, *, obj_in: CreateSchemaType) -> ModelType:
        """Create a new record."""
        # Convert Pydantic schema to dict, excluding unset fields if necessary
        obj_in_data = obj_in.model_dump(exclude_unset=True)
        db_obj = self.model(**obj_in_data)
        db.add(db_obj)
        # Note: commit is usually handled by the session dependency (get_db)
        # await db.commit() # Avoid commit here if using session-per-request pattern
        await db.flush() # Flush to get potential DB defaults like ID before refresh
        await db.refresh(db_obj)
        return db_obj
```

```python
    async def update(
        self,
        db: AsyncSession,
        *,
        db_obj: ModelType,
        obj_in: Union]
    ) -> ModelType:
        """Update an existing record."""
        if isinstance(obj_in, dict):
            update_data = obj_in
        else:
            # Exclude unset fields to allow partial updates
            update_data = obj_in.model_dump(exclude_unset=True)

        # Update model instance attributes
        for field, value in update_data.items():
            if hasattr(db_obj, field):
                setattr(db_obj, field, value)

        db.add(db_obj) # Add the updated object to the session
        # await db.commit() # Avoid commit here
        await db.flush() # Flush changes
        await db.refresh(db_obj) # Refresh to get any DB updates (like onupdate timestamps)
        return db_obj

    async def remove(self, db: AsyncSession, *, id: Union[UUID, int, str]) -> Optional:
        """Delete a record by ID."""
        db_obj = await self.get(db, id)
        if db_obj:
            await db.delete(db_obj)
            # await db.commit() # Avoid commit here
            await db.flush() # Ensure delete operation is flushed
        return db_obj # Return the deleted object or None
```

This base class provides asynchronous implementations for common CRUD operations, taking an AsyncSession as an argument. It uses SQLAlchemy Core select/update/delete constructs for efficiency and compatibility with async execution. Commits are generally handled by the get_db dependency at the end of a successful request, while flush is used within methods to synchronize changes with the database and retrieve generated values (like IDs or default timestamps) when needed before returning the object.

B. Specific Repository Implementations (crud/crud_*.py)

For each model, create a specific CRUD class that inherits from CRUDBase. This class specifies the concrete model and schema types. Any custom query logic specific to that model can be added here.

Python

```python
# crud/crud_act.py
from typing import List, Optional
from uuid import UUID
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select

from.base import CRUDBase # Import the generic base
from models.models_act import Act # Import the SQLAlchemy model
from schemas.schemas_act import ActCreate, ActUpdate # Import Pydantic schemas

class CRUDAct(CRUDBase[Act, ActCreate, ActUpdate]):
    # Inherits get, get_multi, create, update, remove from CRUDBase

    # --- Add model-specific methods here ---

    async def get_by_name(self, db: AsyncSession, *, name: str) -> Optional[Act]:
        """Get an Act by its unique name."""
        stmt = select(self.model).where(self.model.name == name)
        result = await db.execute(stmt)
        return result.scalar_one_or_none()

    async def get_acts_by_story(self, db: AsyncSession, *, story_id: UUID, skip: int = 0, limit: int = 100) ->
List[Act]:
        """Get all Acts belonging to a specific Story."""
        stmt = select(self.model).where(self.model.storyid ==
story_id).offset(skip).limit(limit)
        result = await db.execute(stmt)
```

```python
        return result.scalars().all()

    # Add other specific queries as needed (e.g., get_acts_by_owner)

# Instantiate the CRUD object for Acts
crud_act = CRUDAct(Act)
```

Repeat this pattern for all other models (crud_book.py, crud_image.py, etc.), creating specific CRUD classes and instantiating them.

C. Asynchronous Operations

All database interactions within the repository methods must use await with the asynchronous session methods like session.execute(), session.add(), session.commit(), session.refresh(), and session.delete().[5] Queries should be constructed using the select() construct from SQLAlchemy Core or ORM. Error handling for database-specific exceptions (e.g., IntegrityError on unique constraint violations, NoResultFound if scalar_one() expects a result) should be considered, although basic error handling might occur at the API endpoint level (e.g., catching NoResultFound to return a 404).

VI. API Endpoints with FastAPI (EndPoint/)

API endpoints are the interface through which clients interact with the application. FastAPI's APIRouter helps organize these endpoints logically.

A. Defining APIRouter (EndPoint/ep_*.py)

Using APIRouter is essential for structuring larger FastAPI applications.[49] Create a separate file for each model's endpoints (e.g., ep_act.py). Instantiate APIRouter within each file, specifying a prefix (e.g., /acts) to prepend to all routes defined in that router and tags (e.g., ["Acts"]) to group operations in the OpenAPI documentation (Swagger UI / ReDoc).

Python

```python
# EndPoint/ep_act.py
from fastapi import APIRouter, Depends, HTTPException, status, Query
```

```python
from sqlalchemy.ext.asyncio import AsyncSession
from typing import List, Optional
from uuid import UUID

from core.database import get_db # Dependency for DB session
from schemas import schemas_act # Import Pydantic schemas for Act
from crud import crud_act # Import CRUD operations for Act
# Import dependency for checking authenticated user (defined in auth/dependencies.py)
# from auth.dependencies import get_current_active_user
# from models.models_user import User as UserModel # Import User model for dependency type hint

router = APIRouter(
    prefix="/acts", # Prefix for all routes in this router
    tags=["Acts"], # Tag for grouping in API docs
    # dependencies=[Depends(get_current_active_user)] # Uncomment to protect all routes in this router
)

# --- Define CRUD endpoints ---
# (Endpoints will be defined below)
```

B. Implementing RESTful CRUD Endpoints

Within each router file, define asynchronous functions for the standard RESTful CRUD operations:

- **Create (POST /)**: Accepts a SchemaCreate model in the request body. Uses the corresponding crud function to create the item. Returns the created item using SchemaRead as the response_model. Returns status code 201 (Created) on success.
- **Read Multiple (GET /)**: Accepts optional skip and limit query parameters for pagination. Uses the crud function to retrieve a list of items. Returns a list of SchemaRead models.
- **Read Single (GET /{item_id})**: Accepts the item's ID as a path parameter. Uses the crud function to retrieve the item. Returns the SchemaRead model or raises an HTTPException with status code 404 (Not Found) if the item doesn't exist.
- **Update (PUT /{item_id})**: Accepts the item's ID and a SchemaUpdate model in the request body. Retrieves the existing item, updates it using the crud function, and returns the updated SchemaRead model. Raises 404 if the item doesn't exist. (PATCH can also be used for partial updates).

* **Delete (DELETE /{item_id})**: Accepts the item's ID. Uses the crud function to delete the item. Returns the deleted SchemaRead model or raises 404 if not found. Returns status code 200 or 204 (No Content).

Inject the AsyncSession using db: AsyncSession = Depends(get_db). Use the corresponding instantiated crud object (e.g., crud_act) to perform database operations. Handle potential errors, especially "Not Found" scenarios, by raising HTTPException.[1] FastAPI automatically handles validation errors based on Pydantic schemas, returning 422 (Unprocessable Entity).

Example Endpoints in EndPoint/ep_act.py:

Python

```python
# EndPoint/ep_act.py (continued)

@router.post("/", response_model=schemas_act.ActRead, status_code=status.HTTP_201_CREATED)
async def create_act(
    *,
    db: AsyncSession = Depends(get_db),
    act_in: schemas_act.ActCreate,
    # current_user: UserModel = Depends(get_current_active_user) # Uncomment if ownership needs to be set/checked
):
    """
    Create a new act.
    """
    # Optional: Add ownership check or set ownerid based on current_user
    # if act_in.ownerid!= current_user.id:
    #     raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Not allowed to create act for another user")
    try:
        act = await crud_act.create(db=db, obj_in=act_in)
        return act
    except Exception as e: # Catch potential IntegrityErrors etc.
        # Log the error e
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST, detail=f"Error creating act: {e}")
```

```
@router.get("/", response_model=List)
async def read_acts(
    db: AsyncSession = Depends(get_db),
    skip: int = Query(0, ge=0),
    limit: int = Query(100, ge=1, le=200),
    story_id: Optional[UUID] = Query(None, description="Filter acts by story ID"),
    # current_user: UserModel = Depends(get_current_active_user) # Uncomment if needed for
filtering/auth
):
    """
    Retrieve a list of acts, optionally filtered by story ID.
    """
    if story_id:
        acts = await crud_act.get_acts_by_story(db=db, story_id=story_id, skip=skip,
limit=limit)
    else:
        acts = await crud_act.get_multi(db=db, skip=skip, limit=limit)
    return acts


@router.get("/{act_id}", response_model=schemas_act.ActRead)
async def read_act(
    *,
    db: AsyncSession = Depends(get_db),
    act_id: UUID,
    # current_user: UserModel = Depends(get_current_active_user) # Uncomment if needed
):
    """
    Get a specific act by its ID.
    """
    act = await crud_act.get(db=db, id=act_id)
    if not act:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Act not
found")
    # Optional: Check if current_user owns the act
    # if act.ownerid!= current_user.id:
    #     raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Not authorized to
access this act")
    return act
```

```python
@router.put("/{act_id}", response_model=schemas_act.ActRead)
async def update_act(
    *,
    db: AsyncSession = Depends(get_db),
    act_id: UUID,
    act_in: schemas_act.ActUpdate,
    # current_user: UserModel = Depends(get_current_active_user) # Uncomment if needed
):
    """
    Update an existing act.
    """
    db_act = await crud_act.get(db=db, id=act_id)
    if not db_act:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Act not found")
    # Optional: Check ownership before update
    # if db_act.ownerid!= current_user.id:
    #     raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Not authorized to update this act")

    updated_act = await crud_act.update(db=db, db_obj=db_act, obj_in=act_in)
    return updated_act


@router.delete("/{act_id}", response_model=schemas_act.ActRead)
async def delete_act(
    *,
    db: AsyncSession = Depends(get_db),
    act_id: UUID,
    # current_user: UserModel = Depends(get_current_active_user) # Uncomment if needed
):
    """
    Delete an act.
    """
    db_act = await crud_act.get(db=db, id=act_id)
    if not db_act:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Act not found")
    # Optional: Check ownership before delete
    # if db_act.ownerid!= current_user.id:
    #     raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Not authorized to delete this act")
```

```
    deleted_act = await crud_act.remove(db=db, id=act_id)
    # Return the deleted object data (or modify response_model=None, status_code=204)
    return deleted_act
```

Implement similar endpoint files (ep_book.py, ep_image.py, etc.) for all other models.

C. Integrating Routers in main.py

As shown in Section II.E, the main application file (main.py) needs to import each APIRouter instance from the EndPoint/ files and include it using app.include_router(). This registers all the defined endpoints with the main FastAPI application.[49] Applying a common prefix like /api/v1 during inclusion is good practice for API versioning.

VII. Authentication: OAuth2 Social Logins

Implementing OAuth2 allows users to log in using their existing accounts from providers like Google, Facebook, or Microsoft, enhancing user experience and security.

A. Library Selection and Integration

Several libraries can simplify OAuth2 integration in FastAPI.

- **fastapi-sso**: Appears specifically designed for adding multiple common SSO providers (Google, Facebook, Microsoft, etc.) to FastAPI with minimal boilerplate.[14] It handles redirects and token processing.
- **Authlib**: A more general and powerful OAuth/OpenID Connect client and provider library. It offers flexibility but might require slightly more configuration for basic social login.[16]
- **fastapi-users**: A comprehensive user management library that includes OAuth support. It handles user registration, login, password recovery, and linking multiple OAuth accounts to a single user.[55] It might be more involved than necessary if only social login is required initially, but its user and OAuth account linking features [57] are relevant to the user profile requirements.

For this implementation, **fastapi-sso** [14] is recommended due to its direct focus on simplifying the integration of the specified social providers.

Install the chosen library (pip install fastapi-sso[standard] or poetry add fastapi-sso -E standard).

Configuration involves obtaining **Client IDs** and **Client Secrets** from each provider's developer console (Google Cloud Console, Meta for Developers, Azure App Registrations) and configuring the correct **Redirect URIs** (the /callback/{provider} endpoint in our FastAPI app).[15] These credentials must be stored securely using the environment variable system established in Section II.C.

The OAuth2 **Authorization Code Flow** is typically used for web applications. This involves:

1. Redirecting the user to the provider's login page.
2. User logs in and authorizes the application.
3. Provider redirects the user back to the application's callback URI with an authorization code.
4. The application exchanges this code (along with its client secret) for an access token and potentially an ID token.
5. The application uses the access token to request user information from the provider.

Libraries like fastapi-sso abstract most of these steps.[13]

B. Provider Configuration

Using fastapi-sso, instantiate clients for each provider using the credentials loaded from settings.

Python

```python
# auth/router.py (or a dedicated auth/sso.py)
from fastapi_sso.sso.google import GoogleSSO
from fastapi_sso.sso.facebook import FacebookSSO
from fastapi_sso.sso.microsoft import MicrosoftSSO
from core.config import settings

# Ensure Redirect URIs match exactly what's registered with the provider
google_sso = GoogleSSO(
    client_id=settings.GOOGLE_CLIENT_ID,
    client_secret=settings.GOOGLE_CLIENT_SECRET,
    redirect_uri=str(settings.GOOGLE_REDIRECT_URI), # Ensure string
    allow_insecure_http=True, # Set to False in production (requires HTTPS)
```

```
    scope=["openid", "email", "profile"] # Request necessary user info
)

facebook_sso = None
if settings.FACEBOOK_CLIENT_ID and settings.FACEBOOK_CLIENT_SECRET and
settings.FACEBOOK_REDIRECT_URI:
    facebook_sso = FacebookSSO(
        client_id=settings.FACEBOOK_CLIENT_ID,
        client_secret=settings.FACEBOOK_CLIENT_SECRET,
        redirect_uri=str(settings.FACEBOOK_REDIRECT_URI),
        allow_insecure_http=True, # Set to False in production
        scope=["email", "public_profile"]
    )

microsoft_sso = None
if settings.MICROSOFT_CLIENT_ID and settings.MICROSOFT_CLIENT_SECRET and
settings.MICROSOFT_REDIRECT_URI:
    microsoft_sso = MicrosoftSSO(
        client_id=settings.MICROSOFT_CLIENT_ID,
        client_secret=settings.MICROSOFT_CLIENT_SECRET,
        redirect_uri=str(settings.MICROSOFT_REDIRECT_URI),
        allow_insecure_http=True, # Set to False in production
        scope=, # Basic scope for profile info
        tenant="common" # Or specific tenant ID if needed
    )

# Dictionary to easily access providers
sso_providers = {
    "google": google_sso,
    "facebook": facebook_sso,
    "microsoft": microsoft_sso,
}
sso_providers = {k: v for k, v in sso_providers.items() if v is not None} # Filter out
unconfigured providers
```

The scope parameter requests specific permissions from the user (e.g., access to email and profile information).[15] allow_insecure_http=True is often needed for local development using HTTP, but **must** be set to False in production where HTTPS is required for OAuth2 security.

C. Authentication Routes (auth/router.py)

Create an APIRouter to handle the login initiation and callback processing.

Python

```python
# auth/router.py
from fastapi import APIRouter, Request, Depends, HTTPException, status
from fastapi.responses import RedirectResponse
from sqlalchemy.ext.asyncio import AsyncSession
from typing import Optional

# Import SSO provider instances (assuming defined in auth/sso.py or here)
from.sso import sso_providers, GoogleSSO, FacebookSSO, MicrosoftSSO # Adjust import
as needed
from core.database import get_db
from crud import crud_user # Import user CRUD operations
from schemas import schemas_user # Import user schemas
from models.models_user import User as UserModel # Import user model
# Import function to create session token/JWT (to be defined)
from.security import create_access_token # Example function name
from.dependencies import get_current_active_user # Import dependency

router = APIRouter()

@router.get("/{provider}/login")
async def sso_login(provider: str, request: Request):
    """Initiates the SSO login flow by redirecting the user to the provider."""
    if provider not in sso_providers:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"Provider
'{provider}' not supported")

    sso = sso_providers[provider]
    # Use library's method to get the authorization redirect URL
    return await sso.get_login_redirect()
```

```
@router.get("/{provider}/callback")
async def sso_callback(provider: str, request: Request, db: AsyncSession = Depends(get_db)):
    """
    Handles the callback from the SSO provider after user authentication.
    Verifies the user, finds or creates a local user record, generates a session/JWT,
    and redirects to the frontend.
    """
    if provider not in sso_providers:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=f"Provider
'{provider}' not supported")

    sso = sso_providers[provider]
    try:
        # Process the callback and verify the user with the provider
        # Note: Use async context manager for fastapi-sso >= 0.16.0 [14]
        async with sso:
            openid_user = await sso.verify_and_process(request)
        if openid_user is None:
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Failed to verify SSO user")

        # --- User Linking/Creation Logic ---
        user_email = openid_user.email
        provider_id = openid_user.id # Provider-specific ID
        display_name = openid_user.display_name or user_email # Fallback name
        # Attempt to parse first/last name if available (varies by provider/scope)
        first_name = getattr(openid_user, 'first_name', None)
        last_name = getattr(openid_user, 'last_name', None)

        # 1. Find user by provider ID and provider name (most reliable)
        #    (Requires adding provider and provider_id columns to User model/OAuthAccount table)
        #    db_user = await crud_user.get_by_provider(db, provider=provider, provider_id=provider_id)

        # 2. If not found by provider ID, find by email (allows linking accounts)
        # if not db_user and user_email:
        db_user = await crud_user.get_by_email(db, email=user_email)

        if db_user:
            # User exists, potentially link provider if not already linked
```

```
            # (Add logic here if tracking multiple OAuth accounts per user)
            # Update names if they were missing and are now provided
            update_data = {}
            if not db_user.first_name and first_name:
                update_data["first_name"] = first_name
            if not db_user.last_name and last_name:
                update_data["last_name"] = last_name
            if not db_user.display_name and display_name:
                 update_data["display_name"] = display_name
            if update_data:
                db_user = await crud_user.update(db, db_obj=db_user, obj_in=update_data)
            pass
        else:
            # User does not exist, create a new user record
            if not user_email:
                 raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail="Email not provided by SSO provider.")

            user_in = schemas_user.UserCreateOAuth(
                email=user_email,
                display_name=display_name,
                first_name=first_name,
                last_name=last_name,
                # Set provider details if model supports it
                # oauth_provider=provider,
                # oauth_provider_id=provider_id,
                is_active=True, # Activate user immediately
                is_verified=True # Assume email is verified by provider
            )
            db_user = await crud_user.create_oauth_user(db, obj_in=user_in) # Use a specific
CRUD method

        # --- Session/Token Generation ---
        # Generate a JWT or session token for the authenticated user
        access_token = create_access_token(
            data={"sub": str(db_user.id)} # Use user ID as subject
            # Add other claims like email, roles if needed
        )
```

```
        # --- Redirect to Frontend ---
        # Redirect back to the main UI page or a dashboard
        response = RedirectResponse(url="/ui",
status_code=status.HTTP_303_SEE_OTHER)
        # Set the token in an HttpOnly cookie (more secure than local storage)
        response.set_cookie(
            key="access_token",
            value=f"Bearer {access_token}",
            httponly=True,
            max_age=1800, # Example: 30 minutes
            samesite="lax", # Or "strict"
            # secure=True, # Set to True in production (requires HTTPS)
        )
        return response

    except HTTPException as e:
        # Re-raise HTTPExceptions
        raise e
    except Exception as e:
        # Log the error
        logger.error(f"Error during SSO callback for {provider}: {e}")
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail=f"Authentication failed: {e}")
```

This callback logic handles finding an existing user by email (allowing users to link accounts if they use the same email across providers, similar to fastapi-users's approach [57]) or creating a new user if they don't exist. It then generates an access token (implementation details for create_access_token using JWT are omitted here but standard patterns apply [16]) and sets it in an HttpOnly cookie for the frontend to use.

D. Endpoint Protection

To secure the CRUD API endpoints, a dependency function is needed to verify the access token (sent via the cookie or an Authorization header) and retrieve the currently authenticated user.

Python

```python
# auth/dependencies.py
from fastapi import Depends, HTTPException, status, Request
from fastapi.security import OAuth2PasswordBearer # Can be adapted for cookie auth
from jose import JWTError, jwt
from pydantic import ValidationError
from sqlalchemy.ext.asyncio import AsyncSession
from uuid import UUID

from core.config import settings
from core.database import get_db
from crud import crud_user
from models.models_user import User as UserModel
from schemas import schemas_token # Assuming a TokenData schema exists

# --- Token Verification Logic (Example using Cookie) ---

async def get_token_from_cookie(request: Request) -> Optional[str]:
    token = request.cookies.get("access_token")
    if token and token.startswith("Bearer "):
        return token.split("Bearer ")[1]
    return None

async def get_current_user(
    token: Optional[str] = Depends(get_token_from_cookie),
    db: AsyncSession = Depends(get_db)
) -> UserModel:
    """Dependency to get the current user from the token."""
    if token is None:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Not authenticated",
            headers={"WWW-Authenticate": "Bearer"},
        )
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
```

```
    )
    try:
        payload = jwt.decode(
            token, settings.JWT_SECRET_KEY, algorithms=
        )
        user_id_str: str = payload.get("sub")
        if user_id_str is None:
            raise credentials_exception
        # Validate payload structure if using a TokenData schema
        # token_data = schemas_token.TokenData(id=user_id_str)
        user_id = UUID(user_id_str) # Convert string back to UUID
    except (JWTError, ValidationError, ValueError): # Catch JWT errors, Pydantic validation
errors, or UUID conversion errors
        raise credentials_exception

    user = await crud_user.get(db, id=user_id)
    if user is None:
        raise credentials_exception
    return user

async def get_current_active_user(
    current_user: UserModel = Depends(get_current_user)
) -> UserModel:
    """Dependency to get the current *active* user."""
    if not current_user.is_active: # Assuming an is_active field exists
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail="Inactive user")
    return current_user
```

This get_current_active_user dependency can now be added to the Depends() list for any API endpoint that requires an authenticated and active user.[26] The example shows retrieving the token from a cookie, but OAuth2PasswordBearer(tokenUrl="/auth/token") could be used if tokens are sent in the Authorization: Bearer header.

VIII. User Profile Enhancement

The initial OAuth login provides basic user identification (email, provider ID, potentially a display name). The requirement to collect First Name and Last Name necessitates an additional step after the user is authenticated.

A. Strategy for Collecting Post-OAuth User Details

The most straightforward approach is:

1. **Check Profile on Callback:** In the /auth/{provider}/callback handler (Section VII.C), after finding or creating the db_user, check if db_user.first_name or db_user.last_name are None or empty.
2. **Redirect or Signal Frontend:**
   - **Option A (Redirect):** If profile details are missing, instead of redirecting to the main UI (/ui), redirect the user to a dedicated profile completion page (e.g., /ui/complete-profile).
   - **Option B (Frontend Check):** Always redirect to the main UI (/ui). The frontend, upon receiving the user (e.g., by calling a /users/me endpoint after login), checks if the profile is complete. If not, it prompts the user or navigates them to the profile completion form.

Option B is generally more flexible as it decouples the profile completion flow from the immediate OAuth callback. The frontend can handle the prompting logic. This requires an endpoint for the frontend to fetch the current user's details.

This post-authentication data collection is common as OAuth primarily confirms identity, not application-specific profile completeness.[57]

B. Updating User Model, Schema, and CRUD

To store the additional information:

1. **Model (models/models_user.py):**
   - Ensure the User model includes nullable columns for email (VARCHAR, potentially unique), first_name (VARCHAR), and last_name (VARCHAR).
   - Add is_active (BOOLEAN, default True) and is_verified (BOOLEAN, default False, set to True on OAuth login) fields.
   - Consider adding oauth_provider (VARCHAR) and oauth_provider_id (VARCHAR, indexed) columns to explicitly link OAuth accounts, especially if a user might log in via multiple providers.[55] Alternatively, rely on unique email linkage.

   Python
   ```python
   # models/models_user.py (additions/modifications)
   class User(Base, UUIDPrimaryKeyMixin, TimestampMixin):
       __tablename__ = '"user"' # Quoted table name
   ```

```
    email: Mapped[str] = mapped_column(String(255), unique=True, index=True,
nullable=False)
    first_name: Mapped[str | None] = mapped_column(String(100), nullable=True)
    last_name: Mapped[str | None] = mapped_column(String(100), nullable=True)
    display_name: Mapped[str | None] = mapped_column(String(255), nullable=True)
# From OAuth or constructed

    is_active: Mapped[bool] = mapped_column(Boolean, default=True,
nullable=False)
    is_verified: Mapped[bool] = mapped_column(Boolean, default=False,
nullable=False) # Verified via OAuth

    # Optional: Fields for linking specific OAuth accounts
    # oauth_provider: Mapped[str | None] = mapped_column(String(50), nullable=True)
    # oauth_provider_id: Mapped[str | None] = mapped_column(String(255), nullable=True,
index=True)

    # Relationships back to other models where user is owner
    acts: Mapped[List["Act"]] = relationship(back_populates="owner", lazy="selectin")
    books: Mapped] = relationship(back_populates="owner", lazy="selectin")
    images: Mapped[List["Image"]] = relationship(back_populates="owner",
lazy="selectin")
    locations: Mapped[List["Location"]] = relationship(back_populates="owner",
lazy="selectin")
    things: Mapped] = relationship(back_populates="owner", lazy="selectin")
    personas: Mapped[List["Persona"]] = relationship(back_populates="owner",
lazy="selectin")
    scenes: Mapped] = relationship(back_populates="owner", lazy="selectin")
    stories: Mapped] = relationship(back_populates="owner", lazy="selectin")
    worlds: Mapped[List["World"]] = relationship(back_populates="owner",
lazy="selectin") # Assuming ownerid on world

    def __repr__(self):
        return f"<User(id={self.id}, email='{self.email}')>"
```

2. **Schema (schemas/schemas_user.py):**
    - Update UserBase, UserCreate, UserUpdate, and UserRead schemas to include email, first_name, last_name, display_name, is_active, is_verified.
    - Create a specific schema for creating users via OAuth (UserCreateOAuth) and another for profile updates (UserProfileUpdate).

```python
# schemas/schemas_user.py (additions/modifications)
class UserBase(BaseModel):
    email: EmailStr = Field(...)
    first_name: Optional[str] = Field(None, max_length=100)
```

```
    last_name: Optional[str] = Field(None, max_length=100)
    display_name: Optional[str] = Field(None, max_length=255)
    is_active: bool = True
    is_verified: bool = False

class UserCreateOAuth(UserBase): # Schema for creating user from OAuth callback
    pass # Inherits fields, defaults are set in model or here if needed

class UserProfileUpdate(BaseModel): # Schema for updating profile
    first_name: Optional[str] = Field(None, min_length=1, max_length=100)
    last_name: Optional[str] = Field(None, min_length=1, max_length=100)
    display_name: Optional[str] = Field(None, max_length=255)

class UserRead(UserBase):
    id: uuid.UUID
    created_at: datetime
    updated_at: datetime
    model_config = ConfigDict(from_attributes=True)

# Add other schemas (UserCreate, UserUpdate for admin/other purposes if needed)
```

3. **CRUD (crud/crud_user.py):**
   - Implement get_by_email method.
   - Implement create_oauth_user method using the UserCreateOAuth schema.
   - Ensure the generic update method works correctly with the UserProfileUpdate schema or implement a specific update_profile method.

Python
```python
# crud/crud_user.py (additions/modifications)
from schemas.schemas_user import UserCreateOAuth, UserProfileUpdate # Import new schemas

class CRUDUser(CRUDBase[User, UserCreateOAuth, UserProfileUpdate]): # Adjust generic types
    async def get_by_email(self, db: AsyncSession, *, email: str) -> Optional[User]:
        stmt = select(self.model).where(self.model.email == email)
        result = await db.execute(stmt)
        return result.scalar_one_or_none()

    async def create_oauth_user(self, db: AsyncSession, *, obj_in: UserCreateOAuth) -> User:
        # Use the generic create method or customize if needed
        # Ensure is_verified is handled correctly
        create_data = obj_in.model_dump()
        create_data['is_verified'] = True # Mark as verified from OAuth
        db_obj = self.model(**create_data)
        db.add(db_obj)
        await db.flush()
```

```
        await db.refresh(db_obj)
        return db_obj

    # Generic update method from CRUDBase should handle UserProfileUpdate
    # async def update_profile(self, db: AsyncSession, *, db_obj: User, obj_in: UserProfileUpdate) ->
User:
    #     return await self.update(db, db_obj=db_obj, obj_in=obj_in)

crud_user = CRUDUser(User)
```

C. Profile Completion Endpoint

Create a protected API endpoint allowing authenticated users to submit their first and last names.

```python
# EndPoint/ep_user.py (additions)
from schemas.schemas_user import UserProfileUpdate # Import profile update schema
from auth.dependencies import get_current_active_user # Import auth dependency

@router.put("/me/profile", response_model=schemas_user.UserRead)
async def update_my_profile(
    *,
    db: AsyncSession = Depends(get_db),
    profile_in: UserProfileUpdate,
    current_user: UserModel = Depends(get_current_active_user)
):
    """
    Update the current authenticated user's profile (first name, last name).
    """
    updated_user = await crud_user.update(db=db, db_obj=current_user,
obj_in=profile_in)
    return updated_user

@router.get("/me", response_model=schemas_user.UserRead)
async def read_users_me(
    current_user: UserModel = Depends(get_current_active_user)
):
    """
```

```
    Get current logged in user.
    """
    return current_user
```

The frontend can call the /users/me endpoint after login to get user details. If first_name or last_name is missing, it presents a form that submits data to the /users/me/profile endpoint.

IX. Frontend Interface with NiceGUI

NiceGUI provides a way to build a simple web UI using Python, suitable for basic data editing tasks.

A. Integrating NiceGUI with FastAPI

As shown in Section II.E, integrating NiceGUI involves importing ui and calling ui.run_with(app,...) in main.py.[20] A storage_secret (taken from settings.SESSION_SECRET_KEY) is crucial for managing user-specific state securely across requests. Mounting NiceGUI at a specific path like /ui keeps it separate from the API endpoints.

B. Building UI Components (ui/pages.py or main.py)

NiceGUI uses functions decorated with @ui.page('/path') to define different pages or views.[20]

- **Login Page (/ui/login or integrated into /ui):** Display buttons that link directly to the backend's OAuth initiation routes (/auth/{provider}/login). NiceGUI's ui.open() can be used in button on_click handlers to trigger this navigation.
- **Data Display Pages (e.g., /ui/acts, /ui/books):**
    - Use ui.table to present data fetched from the corresponding API list endpoints (e.g., GET /api/v1/acts).
    - Implement controls (e.g., ui.button, ui.pagination) to handle pagination by making API calls with appropriate skip and limit parameters.
    - Add buttons for "Create New", "Edit", and "Delete" alongside the table or rows.
- **Data Editing Forms (e.g., /ui/acts/new, /ui/acts/{id}/edit or using dialogs):**
    - Use layout elements like ui.card, ui.row, ui.column.[19]
    - Use input elements (ui.input, ui.textarea, ui.select for foreign keys if needed) bound to variables holding the form data.[20]
    - Include "Save" and "Cancel" buttons.

C. Connecting UI Actions to Backend API Endpoints

While NiceGUI runs within the same Python process as FastAPI, UI interactions that modify data should generally communicate through the defined API endpoints. This ensures that all API-level logic (validation, authentication, authorization) is applied consistently. The httpx library is well-suited for making these asynchronous HTTP requests from within NiceGUI event handlers.[25]

Python

```python
# Example within a NiceGUI page function (e.g., in ui/pages.py)
import httpx
from nicegui import ui, app as nicegui_app, Client # Import Client for accessing storage/cookies
from core.config import settings # To get API base URL if needed
from schemas import schemas_act # Import schemas for type hints
from uuid import UUID

# Assume API_BASE_URL is configured, e.g., http://localhost:8000/api/v1
API_BASE_URL = f"http://localhost:8000{api_prefix}" # Use the prefix defined in main.py

async def get_auth_headers(client: Client) -> dict:
    """Helper to get authorization headers from stored token."""
    token = await client.cookies.get("access_token") # Read token from cookie
    if token:
        return {"Authorization": token} # Assumes cookie value includes "Bearer "
    return {}

@ui.page('/ui/acts')
async def acts_page(client: Client): # Inject Client to access cookies/storage
    ui.label('Acts Management').classes('text-h4')

    columns =
    act_table = ui.table(columns=columns, rows=, row_key='id').classes('w-full')

    async def load_acts():
        """Fetch acts from the API and update the table."""
        try:
```

```
        headers = await get_auth_headers(client)
        if not headers:
            ui.notify("Authentication token not found. Please log in.", type='negative')
            # Optionally redirect to login: ui.open('/ui')
            return

        async with httpx.AsyncClient() as http_client:
            response = await http_client.get(f"{API_BASE_URL}/acts", headers=headers,
params={'limit': 50}) # Add pagination later
            response.raise_for_status() # Raise exception for 4xx/5xx errors
            acts_data = response.json()
            # Add action buttons data (or handle via slots)
            for act in acts_data:
                act['actions'] = act['id'] # Pass ID for action handlers
            act_table.rows = acts_data
            act_table.update()
            ui.notify('Acts loaded successfully!', type='positive')
    except httpx.HTTPStatusError as e:
        detail = e.response.json().get('detail', e.response.text)
        ui.notify(f'Error loading acts: {detail} (Status: {e.response.status_code})', type='negative')
    except Exception as e:
        ui.notify(f'An unexpected error occurred: {e}', type='negative')

async def delete_act(act_id: UUID):
    """Delete an act via the API."""
    try:
        headers = await get_auth_headers(client)
        if not headers: ui.notify("Not authenticated.", type='negative'); return

        async with httpx.AsyncClient() as http_client:
            response = await http_client.delete(f"{API_BASE_URL}/acts/{act_id}",
headers=headers)
            response.raise_for_status()
            ui.notify(f"Act {act_id} deleted successfully.", type='positive')
            await load_acts() # Refresh table
    except httpx.HTTPStatusError as e:
        detail = e.response.json().get('detail', 'Unknown error')
        ui.notify(f'Error deleting act: {detail} (Status: {e.response.status_code})', type='negative')
    except Exception as e:
```

```python
            ui.notify(f'An error occurred: {e}', type='negative')

    # Add action buttons to the table (example using slots)
    act_table.add_slot('body-cell-actions', '''
      <q-td :props="props">
        <q-btn flat dense round icon="edit" @click="$parent.$emit('edit', props.row)" />
        <q-btn flat dense round icon="delete" color="negative" @click="$parent.$emit('delete',
props.row.id)" />
      </q-td>
    ''')

    # Handle delete event emitted from table slot
    act_table.on('delete', lambda e: delete_act(e.args))

    # TODO: Implement edit functionality (e.g., open a dialog with inputs)
    # act_table.on('edit', lambda e: open_edit_dialog(e.args))

    # TODO: Implement create functionality (e.g., button to open create dialog)
    ui.button("Create New Act", on_click=lambda: open_create_dialog())

    # Load initial data
    await load_acts()

# TODO: Define functions open_edit_dialog() and open_create_dialog()
# These functions would create ui.dialog() containing input fields and a save button.
# The save button's on_click handler would call httpx.post or httpx.put to the API.
```

This example demonstrates fetching data using httpx.get, updating a ui.table, and making a httpx.delete call from a button click handler. The get_auth_headers helper retrieves the token stored in the cookie to authenticate API requests. Error handling using try...except blocks and ui.notify provides feedback to the user.[19] Similar patterns would be used for create and update operations, typically involving ui.dialog to display forms.

X. Running and Testing the Application

With the components in place, the application can be set up and run.

A. Setup Instructions

1. **Clone Repository:** Obtain the project code.

2. **Environment Setup:** Create a Python virtual environment (e.g., using venv or conda) and activate it.
3. **Install Dependencies:** Run pip install -r requirements.txt or poetry install.
4. **Configure .env:** Create a .env file in the project root. Copy the contents from the example in Section II.C and fill in the actual database credentials and OAuth Client IDs/Secrets obtained from Azure, Google, Facebook, and Microsoft developer consoles. Generate strong random strings for SESSION_SECRET_KEY and JWT_SECRET_KEY.
5. **Database Initialization:**
   - Ensure the Azure PostgreSQL database (testdb on hack1database.postgres.database.azure.com) exists and is accessible with the provided credentials (testuser/testuser123).
   - Ensure the pgcrypto extension is enabled in the testdb database. This might require running CREATE EXTENSION IF NOT EXISTS "pgcrypto"; using a tool like psql or Azure Data Studio, potentially requiring admin privileges.
   - Initial table creation can be handled by uncommenting await create_db_and_tables() in the lifespan function in main.py for the first run. However, for production environments, using a dedicated migration tool like **Alembic** is strongly recommended for managing schema changes systematically.[1]

B. Running the Server

Execute the application using Uvicorn from the project root directory:

Bash

```bash
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

- main:app: Tells Uvicorn to find the FastAPI instance named app inside the main.py file.
- --reload: Enables auto-reloading the server when code changes are detected (useful for development).[2]
- --host 0.0.0.0: Makes the server accessible from other devices on the network (use 127.0.0.1 for local access only).
- --port 8000: Specifies the port number the server will listen on.

C. Accessing the Application

- **API Documentation:** Open a web browser and navigate to http://localhost:8000/docs (Swagger UI) or http://localhost:8000/redoc (ReDoc) to explore the automatically generated interactive API documentation.
- **NiceGUI Frontend:** Navigate to http://localhost:8000/ui (or the root / if NiceGUI is mounted there) to access the web interface.

XI. Conclusion and Next Steps

A. Summary

This report has detailed the design and implementation of a comprehensive FastAPI application featuring asynchronous communication with an Azure PostgreSQL database, full CRUD API endpoints for ten distinct models, and secure user authentication via Google, Facebook, and Microsoft OAuth2 providers. The application adheres to a specified modular structure, separating concerns into models, schemas, data access logic (CRUD/Repository), and API endpoints. Pydantic ensures data validation, while SQLAlchemy manages asynchronous ORM interactions. A basic frontend interface using NiceGUI provides capabilities for data viewing and editing by interacting with the backend API. Key considerations such as environment variable management, secure database connections (SSL), server-side default value generation (UUIDs, timestamps), and asynchronous session management have been addressed using established best practices and libraries like fastapi-sso and httpx.

B. Recommendations for Further Development

While the current implementation provides a solid foundation, several areas can be enhanced for production readiness and further functionality:

1. **Comprehensive Testing:** Implement a robust testing suite using pytest. Include unit tests for CRUD operations (potentially mocking the database session) and integration tests using FastAPI's TestClient (or httpx directly) to verify API endpoint behavior, including authentication and validation.[1]
2. **Database Migrations:** Integrate **Alembic** to manage database schema changes systematically.[1] create_all is suitable for initial setup or testing but unsafe for evolving production schemas. Alembic provides version control for the database schema.
3. **Enhanced Error Handling:** Implement more granular exception handling in CRUD and API layers. Provide clearer error messages to the frontend and implement robust logging for diagnostics.
4. **Advanced Frontend:** If UI requirements grow beyond basic CRUD operations,

consider developing a dedicated frontend application using frameworks like React, Vue, or Angular, which offer more extensive UI component libraries and state management solutions. NiceGUI is excellent for rapid development and Python-centric interfaces but may become limiting for complex UIs.[65]

5. **Deployment Strategy:** Define a clear deployment process. Containerize the application using Docker [3] and deploy to a suitable platform (e.g., Azure App Service, Kubernetes). Configure production-grade components like Gunicorn with Uvicorn workers, HTTPS termination (e.g., via Nginx or a load balancer), and proper environment variable injection.
6. **Security Hardening:** Conduct thorough security reviews. Implement measures like input sanitization beyond Pydantic's validation, rate limiting on API endpoints (e.g., using slowapi), Cross-Site Request Forgery (CSRF) protection (especially if using session cookies), and regular dependency vulnerability scanning. Ensure HTTPS is enforced in production.
7. **Asynchronous Task Queues:** For long-running operations initiated by API requests (e.g., complex report generation, sending emails), integrate an asynchronous task queue like Celery (with Redis or RabbitMQ) or ARQ to process them in the background without blocking API responses.
8. **Caching:** Implement caching strategies (e.g., using Redis) for frequently accessed, rarely changing data to improve API response times and reduce database load.

Works cited

1. How to Integrate FastAPI With SQLAlchemy - Neurelo, accessed April 19, 2025, https://www.neurelo.com/post/how-to-integrate-fastapi-with-sqlalchemy
2. Building a CRUD FastAPI app with SQLAlchemy - Mattermost, accessed April 19, 2025, https://mattermost.com/blog/building-a-crud-fastapi-app-with-sqlalchemy/
3. Fastapi Postgres Example Guide | Restackio, accessed April 19, 2025, https://www.restack.io/p/fastapi-answer-postgres-example
4. Connect to PostgreSQL with SQLAlchemy and asyncio - Makimo, accessed April 19, 2025, https://makimo.com/blog/connect-to-postgresql-with-sqlalchemy-and-asyncio/
5. From Zero to Production: Setting Up a SQL Database with Async Engine in FastAPI, accessed April 19, 2025, https://timothy.hashnode.dev/from-zero-to-production-setting-up-a-sql-database-with-async-engine-in-fastapi
6. Asynchronous I/O (asyncio) — SQLAlchemy 2.0 Documentation, accessed April 19, 2025, http://docs.sqlalchemy.org/en/latest/orm/extensions/asyncio.html
7. Implementation of Repository Pattern in Python? - Stack Overflow, accessed April 19, 2025,

https://stackoverflow.com/questions/9699598/implementation-of-repository-pattern-in-python
8. Is anyone using asyncpg to connect from Python? : r/PostgreSQL - Reddit, accessed April 19, 2025, https://www.reddit.com/r/PostgreSQL/comments/7r2fzg/is_anyone_using_asyncpg_to_connect_from_python/
9. How to connect to an Azure Postgres DB that requires ssl using asyncpg - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/79230394/how-to-connect-to-an-azure-postgres-db-that-requires-ssl-using-asyncpg
10. Databases - Pydantic, accessed April 19, 2025, https://docs.pydantic.dev/latest/examples/orms/
11. Interaction between Pydantic models/schemas in the FastAPI Tutorial - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/73700879/interaction-between-pydantic-models-schemas-in-the-fastapi-tutorial
12. Models - Pydantic, accessed April 19, 2025, https://docs.pydantic.dev/latest/concepts/models/
13. SSO with Social Login Integration & FastAPI Simplified - SlideShare, accessed April 19, 2025, https://www.slideshare.net/slideshow/social-login-integration-with-fastapi-simplified/274633899
14. FastAPI plugin to enable SSO to most common providers (such as Facebook login, Google login and login via Microsoft Office 365 Account) - GitHub, accessed April 19, 2025, https://github.com/tomasvotava/fastapi-sso
15. Fastapi Oauth2 Google Integration | Restackio, accessed April 19, 2025, https://www.restack.io/p/fastapi-answer-oauth2-google
16. Examples on how to implement the OAuth2 authorization code flow using FastAPI - GitHub, accessed April 19, 2025, https://github.com/lukasthaler/fastapi-oauth-examples
17. FastAPI OAuth Client - Authlib 1.5.2 documentation, accessed April 19, 2025, https://docs.authlib.org/en/latest/client/fastapi.html
18. How to implement OAuth to FastAPI with client ID & Secret - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/63248112/how-to-implement-oauth-to-fastapi-with-client-id-secret
19. Python NiceGUI: Build Powerful Web Interfaces with Ease - DataCamp, accessed April 19, 2025, https://www.datacamp.com/tutorial/nicegui
20. NiceGUI, accessed April 19, 2025, https://nicegui.io/
21. Adopting the Repository Pattern for Enhanced Backend Development With FastAPI, accessed April 19, 2025, https://hackernoon.com/adopting-the-repository-pattern-for-enhanced-backend-development-with-fastapi
22. 0xTheProDev/fastapi-clean-example - GitHub, accessed April 19, 2025, https://github.com/0xTheProDev/fastapi-clean-example

23. A Python Implementation of the Unit of Work and Repository Design Pattern using SQLModel - DEV Community, accessed April 19, 2025, https://dev.to/manukanne/a-python-implementation-of-the-unit-of-work-and-repository-design-pattern-using-sqlmodel-3mb5
24. CRUD API Dependency Injection using Repository Pattern - Avoiding poor patterns and over-engineering : r/learnpython - Reddit, accessed April 19, 2025, https://www.reddit.com/r/learnpython/comments/1iu1qz4/crud_api_dependency_injection_using_repository/
25. How to Conect to RESTful Interface? · zauberzeug nicegui · Discussion #447 - GitHub, accessed April 19, 2025, https://github.com/zauberzeug/nicegui/discussions/447
26. Fastapi Multiple Authentication Methods | Restackio, accessed April 19, 2025, https://www.restack.io/p/ai-for-identity-verification-answer-fastapi-multiple-authentication-methods-cat-ai
27. OAuth2 scopes - FastAPI, accessed April 19, 2025, https://fastapi.tiangolo.com/advanced/security/oauth2-scopes/
28. Engine Configuration — SQLAlchemy 2.0 Documentation, accessed April 19, 2025, http://docs.sqlalchemy.org/en/latest/core/engines.html
29. Asynchronous Database Sessions in FastAPI with SQLAlchemy - DEV Community, accessed April 19, 2025, https://dev.to/akarshan/asynchronous-database-sessions-in-fastapi-with-sqlalchemy-1o7e
30. Slow DB ORM operations? PostgresSQL+ SQLAlchemy + asyncpg : r/FastAPI - Reddit, accessed April 19, 2025, https://www.reddit.com/r/FastAPI/comments/1hjordn/slow_db_orm_operations_postgressql_sqlalchemy/
31. How to get a session from async_session() generator FastApi Sqlalchemy - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/75150942/how-to-get-a-session-from-async-session-generator-fastapi-sqlalchemy
32. Session with FastAPI Dependency - SQLModel, accessed April 19, 2025, https://sqlmodel.tiangolo.com/tutorial/fastapi/session-with-dependency/
33. What is the benefit of letting FastAPI handle SQLAlchemy's session vs the provided context manager? - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/78648595/what-is-the-benefit-of-letting-fastapi-handle-sqlalchemys-session-vs-the-provid
34. FastApi Sqlalchemy how to manage transaction (session and multiple commits), accessed April 19, 2025, https://stackoverflow.com/questions/65699977/fastapi-sqlalchemy-how-to-manage-transaction-session-and-multiple-commits
35. Modeling - Advanced Alchemy - Litestar, accessed April 19, 2025, https://docs.advanced-alchemy.litestar.dev/latest/usage/modeling.html
36. Correct way to autogenerate UUID primary key on server side, without a database roundtrip? #10698 - GitHub, accessed April 19, 2025, https://github.com/sqlalchemy/sqlalchemy/discussions/10698

37. How to add current date time by default on a table declaration? · Issue #594 · fastapi/sqlmodel - GitHub, accessed April 19, 2025, https://github.com/tiangolo/sqlmodel/issues/594
38. Column INSERT/UPDATE Defaults — SQLAlchemy 2.0 ..., accessed April 19, 2025, http://docs.sqlalchemy.org/en/latest/core/defaults.html
39. How can I use UUIDs in SQLAlchemy? - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/183042/how-can-i-use-uuids-in-sqlalchemy
40. GUID Type - FastAPI Utilities, accessed April 19, 2025, https://fastapi-utils.davidmontague.xyz/user-guide/basics/guid-type/
41. How to use pydantic and sqlalchemy models with relationship #9084 - GitHub, accessed April 19, 2025, https://github.com/fastapi/fastapi/discussions/9084
42. Many-To-Many Relationships In FastAPI - GormAnalysis, accessed April 19, 2025, https://www.gormanalysis.com/blog/many-to-many-relationships-in-fastapi/
43. Patterns and Practices for using SQLAlchemy 2.0 with FastAPI - The Chaotic Engineer, accessed April 19, 2025, https://chaoticengineer.hashnode.dev/fastapi-sqlalchemy
44. Define relationships — FastAPI-JSONAPI 3.0.0 documentation - Read the Docs, accessed April 19, 2025, https://fastapi-jsonapi.readthedocs.io/en/latest/relationships.html
45. SQLAlchemy+FastAPI: Set up a relationship in model.py to match a return Pydantic Union in schemas - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/78840414/sqlalchemyfastapi-set-up-a-relationship-in-model-py-to-match-a-return-pydantic
46. returning nested data in fastapi + pydantic v2 + sqlalchemy (orm_mode deprecated), accessed April 19, 2025, https://stackoverflow.com/questions/77705444/returning-nested-data-in-fastapi-pydantic-v2-sqlalchemy-orm-mode-deprecated
47. How do I convert a many-to-many sqlalchemy model to a Pydantic model? - Reddit, accessed April 19, 2025, https://www.reddit.com/r/FastAPI/comments/rf2n1i/how_do_i_convert_a_manyto_many_sqlalchemy_model_to/
48. Example data repository using Async Postgres, SQLAlchemy, Pydantic : r/Python - Reddit, accessed April 19, 2025, https://www.reddit.com/r/Python/comments/1j6zwrh/example_data_repository_using_async_postgres/
49. How to Use FastAPI APIRouters - Apidog, accessed April 19, 2025, https://apidog.com/articles/how-to-use-fastapi-apirouter/
50. Fastapi Multiple Route Files | Restackio, accessed April 19, 2025, https://www.restack.io/p/fastapi-knowledge-multiple-route-files
51. APIRouter class - FastAPI, accessed April 19, 2025, https://fastapi.tiangolo.com/reference/apirouter/
52. Fastapi Multiple Routers With Same Prefix | Restackio, accessed April 19, 2025, https://www.restack.io/p/fastapi-knowledge-multiple-routers-answer
53. Can multiple routers be registered at once in FastAPI? - Stack Overflow, accessed April 19, 2025,

https://stackoverflow.com/questions/77445628/can-multiple-routers-be-registered-at-once-in-fastapi
54. python 3.x - Best Practice FastAPI - including routers - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/75610309/best-practice-fastapi-including-routers
55. OAuth2 - FastAPI Users, accessed April 19, 2025, https://fastapi-users.github.io/fastapi-users/10.1/configuration/oauth/
56. FastAPI-users and Google oauth - Cannot retrieve user profile, accessed April 19, 2025, https://www.reddit.com/r/FastAPI/comments/1f9im5d/fastapiusers_and_google_oauth_cannot_retrieve/
57. How to handle same user registration from different OAuth #549 - GitHub, accessed April 19, 2025, https://github.com/fastapi-users/fastapi-users/discussions/549
58. Google oauth with fastapi-users procedure - python - Stack Overflow, accessed April 19, 2025, https://stackoverflow.com/questions/76647367/google-oauth-with-fastapi-users-procedure
59. Seeking Guidance on Implementing OAuth for React + FastAPI Application (Google & Microsoft) - Reddit, accessed April 19, 2025, https://www.reddit.com/r/FastAPI/comments/1868m6r/seeking_guidance_on_implementing_oauth_for_react/
60. OAuth2 scopes - FastAPI, accessed April 19, 2025, https://fastapi.tiangolo.com/id/advanced/security/oauth2-scopes/
61. Security - FastAPI, accessed April 19, 2025, https://fastapi.tiangolo.com/tutorial/security/
62. Get Current User - FastAPI, accessed April 19, 2025, https://fastapi.tiangolo.com/tutorial/security/get-current-user/
63. FastAPI Authentication by Example - Developer Center - Auth0, accessed April 19, 2025, https://developer.auth0.com/resources/guides/web-app/fastapi/basic-authentication
64. FastApi and nicegui #2223 - GitHub, accessed April 19, 2025, https://github.com/zauberzeug/nicegui/discussions/2223
65. FastAPI & NiceGUI integration - Reddit, accessed April 19, 2025, https://www.reddit.com/r/nicegui/comments/1dlx3or/fastapi_nicegui_integration/