ADPC Project 1

Beatriz Escribano Cebrián

February 2025

1 Database Creation and Connection

1.1 Database Setup

This project uses **PostgreSQL** as the primary database and is provisioned on a cloud instance via **Tembo**.

1.2 Steps to Create the Database

1. **Provision a Cloud PostgreSQL Instance**
   - Sign up on **Tembo** and create a new PostgreSQL database instance.
   - Note down the host, port, database name, username, and password.
2. **Run the Database Initialization Script**
   - Execute the SQL script `init_db.sql` to create the necessary tables, relationships, and constraints.
   - Ensure the database schema includes entities such as `authors`, `repositories`, `packages`, `dependencies`, `users`, and `user_repository_access`.
   - Command:

     ```
     psql "postgresql://<username>:<password>@<host>:<port>/<database_name>" -f
     ```

3. **Configure the Connection in the Application**
   - The database connection is managed in `database/connection.py`.
   - Environment variables are stored in `.env`:

     ```
     DB_HOST=<host>
     DB_PORT=<port>
     DB_NAME=<database_name>
     DB_USER=<username>
     DB_PASSWORD=<password>
     ```

- The connection is established using **SQLAlchemy** with **lazy initialization**, a pattern where the database connection and session are only created when they are first needed, rather than at application startup, and each time they are needed again, the class returns the existing instance, avoiding unnecessary connections. This helps optimize resource usage and improves efficiency.
- **SQLAlchemy** was chosen as the Object-Relational Mapping (ORM) tool for this project due to its robustness, flexibility, and seamless integration with PostgreSQL.
- It integrates seamlessly with Python, as database tables can be represented as Python classes, improving readability and maintainability.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import os

DATABASE_URL = (f"postgresql://{os.getenv('DB_USER')}:"
                f"{os.getenv('DB_PASSWORD')}@{os.getenv('DB_HOST')}:"
                f"{os.getenv('DB_PORT')}/{os.getenv('DB_NAME')}")
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

4. **Apply Migrations with Alembic**

- **Alembic** was chosen as the database migration tool for this project because it integrates seamlessly with **SQLAlchemy**, allowing for efficient and structured database schema management.
- Alembic can automatically detect changes in SQLAlchemy models and generate migration scripts accordingly..
- Initialize Alembic:

```
alembic init database/migrations
```

- Generate an initial migration script:

```
alembic revision --autogenerate -m "Initial migration"
```

- Apply migrations:

```
alembic upgrade head
```

2 Application-Database Communication

The project follows a structured approach for **database interaction** using an **ORM-based repository pattern**.

2.1 Components Involved

- **Models** (`models.py`) - Defines database entities and their relationships.
- **Repository** (`repository.py`) - Implements CRUD operations using the ORM.
- **Service Layer** (`repository_factory.py`) - Serves as an abstract factory design pattern to manage database interactions.

2.2 How Data is Queried and Stored

Fetching Data:

```
class AbstractRepository(ABC):
    """Abstract repository enforcing implementation of required methods."""
    def __init__(self):
        self.conn = DatabaseConnection.get_connection()

    @abstractmethod
    def get_packages_by_repo(self, repo_name):
        pass

    @abstractmethod
    def get_package_info(self, package_name):
        pass

    @abstractmethod
    def find_packages(self, search_query):
        pass
```

Inserting Data:

```
def create_user(db_session, username: str, email: str):
    db_user = User(username=username, email=email)
    db_session.add(db_user)
    db_session.commit()
    db_session.refresh(db_user)
    return db_user
```

Updating and Deleting Data:

```
def update_user_email(db_session, user_id: int, new_email: str):
    db_session.query(User).filter(User.id == user_id).update({User.email: new_email})
    db_session.commit()


def delete_package(db_session, package_id: int):
    package = db_session.query(Package).filter(Package.id == package_id).first()
    if package:
        db_session.delete(package)
        db_session.commit()
```

3 Conceptual Model and Normalization

3.1 Database Design

The database adheres to **3rd Normal Form (3NF)** and is structured to optimize data integrity and minimize redundancy using different data relationships.

- **One-to-Many Relationship**: A **Repository** can have multiple **Packages**. This ensures that each package is associated with a specific repository while allowing a repository to contain multiple packages.
- **Many-to-Many Relationship**: A **User** can have multiple **Repository accesses**, with different permission levels (read, write, admin). This is implemented using a junction table (`user_repository_access`).
- **One-to-One Relationship**: A **Package** has exactly one **Detailed record**, storing additional metadata such as release notes and documentation links. This is represented by the `package_details` table.
- **Self-Referencing Relationship**: The **Dependencies** table establishes relationships where a **Package** can depend on other **Packages**, forming a self-referencing Many-to-Many relationship.

3.2 SQL Data Definition Language (DDL) for 3NF Compliance

```
-- Drop existing tables to avoid conflicts
DROP TABLE IF EXISTS user_repository_access CASCADE;
DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS dependencies CASCADE;
DROP TABLE IF EXISTS packages CASCADE;
DROP TABLE IF EXISTS authors CASCADE;
DROP TABLE IF EXISTS repositories CASCADE;

-- Create the authors table (One-to-Many relationship with packages)
CREATE TABLE authors (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) UNIQUE
);

-- Create the repositories table (One-to-Many relationship with packages)
CREATE TABLE repositories (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Create the packages table
CREATE TABLE packages (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    version VARCHAR(50) NOT NULL,
    description TEXT,
    author_id INT REFERENCES authors(id) ON DELETE SET NULL,
    repo_id INT REFERENCES repositories(id) ON DELETE CASCADE,
    artifact_path TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (name, version)
);

-- Create the package details table (One-to-One relationship)
CREATE TABLE package_details (
    package_id INT PRIMARY KEY REFERENCES packages(id),
    release_notes TEXT,
    documentation_link TEXT
);

-- Create a self-referencing dependencies table (Many-to-Many relationship within packages)
CREATE TABLE dependencies (
    package_id INT REFERENCES packages(id) ON DELETE CASCADE,
    dependency_id INT REFERENCES packages(id) ON DELETE CASCADE,
    PRIMARY KEY (package_id, dependency_id)
);

-- Create the users table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create a user repository access table (Many-to-Many relationship between users and reposi
CREATE TABLE user_repository_access (
    user_id INT REFERENCES users(id) ON DELETE CASCADE,
    repo_id INT REFERENCES repositories(id) ON DELETE CASCADE,
    access_level VARCHAR(50) CHECK (access_level IN ('read', 'write', 'admin')),
    PRIMARY KEY (user_id, repo_id)
);
```

4 Understanding ORM and Migrations

4.1 What is ORM?

An **Object-Relational Mapper (ORM)** allows developers to interact with databases using object-oriented code instead of raw SQL.

4.2 ORM Used: SQLAlchemy

- **Declarative Base**: Defines models using Python classes.
- **Session Management**: Handles transactions automatically.
- **Migration Support**: Uses Alembic for schema changes.

4.3 Example ORM Model

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from database.connection import Base

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    email = Column(String, unique=True, nullable=False)
    orders = relationship("Order", back_populates="user")
```

5 Demonstrating ORM Migrations

5.1 Applying a Migration

1. **Generate Migration**

   ```
   alembic revision --autogenerate -m "Add package tags"
   ```

2. **Apply Migration**

   ```
   alembic upgrade head
   ```

3. **Example Migration File** (`versions/ce6a5f4e0942_add_package_tags.py`)

   ```
   def upgrade():
       op.add_column('package', sa.Column('tag', sa.String(), nullable=True))
   ```

By following this ORM-based approach, the project ensures **maintainability, scalability, and abstraction from raw SQL queries**.