Folder app

26 printable files

(file list disabled)

app/.env

```
DATABASE_URL=mariadb://jintian:1K403uvJqS84@vschool.ddns.net:3306/cplearn_test
SECRET_KEY=cc0804b0b7709c908e15635a1a47bd8f05e78a7f17bd1338fbcf7084146330a9
```

app/__init__.py

```

```

app/config.py

```

from dotenv import load_dotenv
from pydantic_settings import BaseSettings
# Automatically load .env from the current working directory (no need for os)
load_dotenv()

class Settings(BaseSettings):
    database_url: str
    secret_key: str

    class Config:
        env_file = ".env"  # Pydantic will also look for the .env file automatically in cwd
        extra = 'allow'  # Use a literal string instead of Extra.allow

settings = Settings()


```

app/crud/__init__.py

```

```

app/crud/role.py

```
from sqlalchemy.orm import Session
from app.models.role import Role
from app.schemas.role import RoleCreate

def create_role(db: Session, role: RoleCreate):
    db_role = Role(role_name=role.role_name)
    db.add(db_role)
    db.commit()
```

```
    db.refresh(db_role)
    return db_role

# Function to fetch all roles
def get_all_roles(db: Session):
    return db.query(Role).all()

def get_role(db: Session, role_id: int):
    return db.query(Role).filter(Role.id == role_id).first()

def get_roles(db: Session, skip: int = 0, limit: int = 10):
    return db.query(Role).offset(skip).limit(limit).all()

def delete_role(db: Session, role_id: int):
    db_role = get_role(db, role_id)
    if db_role:
        db.delete(db_role)
        db.commit()
    return db_role

```

app/crud/student.py

```
from sqlalchemy.orm import Session
from app.models.student import Student
from app.schemas.student import StudentCreate

def create_student(db: Session, student: StudentCreate):
    db_student = Student(**student.dict())
    db.add(db_student)
    db.commit()
    db.refresh(db_student)
    return db_student

def get_student(db: Session, student_id: int):
    return db.query(Student).filter(Student.id == student_id).first()

def get_students(db: Session, skip: int = 0, limit: int = 10):
    return db.query(Student).offset(skip).limit(limit).all()

def delete_student(db: Session, student_id: int):
    db_student = get_student(db, student_id)
    if db_student:
        db.delete(db_student)
        db.commit()
    return db_student

```

app/crud/user.py

```python
from sqlalchemy.orm import Session
from app.models.user import User
from app.schemas.user import UserCreate
from app.utils.security import hash_password

def create_user(db: Session, user: UserCreate):
    hashed_password = hash_password(user.password)
    db_user = User(**user.dict(exclude={'password'}))
    db_user.password_hash = hashed_password
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
# ... (rest of the file remains the same)
def get_user_by_account_id(db: Session, account_id: str):
    return db.query(User).filter(User.account_id == account_id).first()

def get_user(db: Session, user_id: int):
    return db.query(User).filter(User.id == user_id).first()

def get_all_users(db: Session):
    return db.query(User).all()  # This will return all users in the 'users' table

def get_users(db: Session, skip: int = 0, limit: int = 10):
    return db.query(User).offset(skip).limit(limit).all()

def delete_user(db: Session, user_id: int):
    db_user = get_user(db, user_id)
    if db_user:
        db.delete(db_user)
        db.commit()
    return db_user

```

app/db.py

```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from app.config import settings
from sqlalchemy.exc import SQLAlchemyError
import logging

# Configure logging for the database operations
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Database URL from your .env file or configuration
DATABASE_URL = settings.database_url
```

```

# Step 1: Create a SQLAlchemy engine with connection pooling
# Customize pool settings: pool_size, max_overflow, pool_recycle, pool_timeout
engine = create_engine(
    DATABASE_URL,
    pool_size=10,            # Max number of connections in the pool
    max_overflow=20,         # Max additional connections above pool_size
    pool_recycle=3600,       # Recycle connections after 1 hour (3600 seconds)
    pool_timeout=30          # Timeout to get a connection from the pool (30 seconds)
)

# Step 2: Create a session factory (SessionLocal) bound to the engine
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Step 3: Create a declarative base class for models
Base = declarative_base()

# Step 4: Dependency for getting a database session for each request
def get_db():
    """
    Dependency that provides a SQLAlchemy session for the duration of a request.
    It yields a session and ensures the session is closed after the request, return↩
ing
    the connection back to the connection pool.
    """
    db = SessionLocal()  # Create a new session
    try:
        yield db  # Yield the session for use in the API call
    except SQLAlchemyError as e:
        # Handle any database-related exceptions
        logger.error(f"Database error: {str(e)}")
        db.rollback()  # Rollback any changes if an error occurs
        raise
    finally:
        # Always close the session after use, returning the connection to the pool
        db.close()
        logger.info("Database session closed and connection returned to the pool")

```

app/logging_config.py

```

# app/logging_config.py

import logging

def setup_logging():
    """
    Configures logging for the entire application.
```

```
    """
    logging.basicConfig(
        level=logging.INFO,  # Set the minimum log level
        format='%(asctime)s %(levelname)s %(name)s %(message)s',  # Log format
        handlers=[
            logging.FileHandler("app.log"),  # Log to a file
            logging.StreamHandler()          # Also log to the console
        ]
    )

```

app/main.py

```
from fastapi import FastAPI
from app.routers import role, user, student, auth
from app.logging_config import setup_logging  # Import the logging setup
from dotenv import load_dotenv
import os

load_dotenv()

DATABASE_URL=os.getenv("DATABASE_URL")
SECRET_KEY=os.getenv("SECRET_KEY")

# Setup logging
setup_logging()


app = FastAPI()

app.include_router(role.router)
app.include_router(user.router)
app.include_router(student.router)
app.include_router(auth.router)

```

app/models/__init__.py

```

```

app/models/role.py

```
from sqlalchemy import Column, Integer, String
from app.db import Base

class Role(Base):
    __tablename__ = "roles"

    id = Column(Integer, primary_key=True, index=True)
    role_name = Column(String(50), unique=True, index=True, nullable=False)
```

9

app/models/student.py

```python
from sqlalchemy import Column, Integer, String, ForeignKey, Text
from app.db import Base

class Student(Base):
    __tablename__ = "students"

    id = Column(Integer, primary_key=True, index=True)
    student_id = Column(Integer, ForeignKey("users.id"), nullable=False, unique=True)
    grade_level = Column(String(255), nullable=False)
    main_parent_id = Column(Integer, ForeignKey("users.id"), nullable=False)
    secondary_parent_id = Column(Integer, ForeignKey("users.id"), nullable=True)
    addional_info = Column(Text)
    food_allergy = Column(Text)

```

app/models/user.py

```python
# app/models/user.py

from sqlalchemy import Column, Integer, String, Boolean, ForeignKey, TIMESTAMP, text
from sqlalchemy.orm import relationship
from app.db import Base

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    role_id = Column(Integer, ForeignKey("roles.id"), nullable=False)
    account_id = Column(String(255), unique=True, nullable=False)
    last_name = Column(String(100), nullable=False)
    first_name = Column(String(100), nullable=False)
    email = Column(String(100), unique=True, nullable=False)
    phone = Column(String(50))
    wechat_id = Column(String(50))
    password_hash = Column(String(255))
    created_at = Column(TIMESTAMP, server_default=text('CURRENT_TIMESTAMP'))
    updated_at = Column(
        TIMESTAMP,
        server_default=text('CURRENT_TIMESTAMP'),
        onupdate=text('CURRENT_TIMESTAMP')
    )
    is_active = Column(Boolean, default=True)

    role = relationship("Role")

```

app/routers/__init__.py

```

```

app/routers/auth.py

```python
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from app.db import get_db
from app.crud.user import get_user_by_account_id
from app.utils.security import verify_password
from datetime import datetime, timedelta
from jose import JWTError, jwt
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from app.config import settings
from app.schemas.user import User
from app.config import settings

router = APIRouter()

SECRET_KEY =DATABASE_URL = settings.secret_key #"6aa48ea5c4ff67e2fd7cdcdef7cd↩
d4554d80fa3b84110b10d23113eb1e85acb2"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/token")
def authenticate_user(db: Session, account_id: str, password: str):
    user = get_user_by_account_id(db, account_id=account_id)
    if not user or not verify_password(password, user.password_hash):
        return False
    return user

def create_access_token(data: dict, expires_delta: timedelta = None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=15))
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
@router.post("/token")
def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db)
):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect account_id or password",
            headers={"WWW-Authenticate": "Bearer"},
```

```
    )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.account_id, "role_id": user.role_id, "user_id": user.id},
        expires_delta=access_token_expires
)
    return {"access_token": access_token, "token_type": "bearer"}

def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db)
    ):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
)
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        account_id: str = payload.get("sub")
        if account_id is None:
            raise credentials_exception
        user = get_user_by_account_id(db, account_id=account_id)
        if user is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    return user
```

app/routers/role.py

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.crud import role as crud_role
from app.schemas import role as schema_role
from app.db import get_db

router = APIRouter()

#  get all roles
@router.get("/roles", response_model=list[schema_role.Role])
def get_all_roles(db: Session = Depends(get_db)):
    roles = crud_role.get_all_roles(db)
    return roles

@router.post("/roles/", response_model=schema_role.Role)
def create_role(role: schema_role.RoleCreate, db: Session = Depends(get_db)):
    return crud_role.create_role(db, role)

```

```python
@router.get("/roles/{role_id}", response_model=schema_role.Role)
def get_role(role_id: int, db: Session = Depends(get_db)):
    db_role = crud_role.get_role(db, role_id)
    if db_role is None:
        raise HTTPException(status_code=404, detail="Role not found")
    return db_role

@router.delete("/roles/{role_id}")
def delete_role(role_id: int, db: Session = Depends(get_db)):
    return crud_role.delete_role(db, role_id)

```

app/routers/student.py

```python
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.crud import student as crud_student
from app.schemas import student as schema_student
from app.db import get_db

router = APIRouter()

@router.post("/students/", response_model=schema_student.Student)
def create_student(student: schema_student.StudentCreate, db: Session =
Depends(get_db)):
    return crud_student.create_student(db, student)

@router.get("/students/{student_id}", response_model=schema_student.Student)
def get_student(student_id: int, db: Session = Depends(get_db)):
    db_student = crud_student.get_student(db, student_id)
    if db_student is None:
        raise HTTPException(status_code=404, detail="Student not found")
    return db_student

@router.delete("/students/{student_id}")
def delete_student(student_id: int, db: Session = Depends(get_db)):
    return crud_student.delete_student(db, student_id)

```

app/routers/user.py

```python
# app/routers/user.py

from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.crud import user as crud_user
from app.schemas import user as schema_user
from app.db import get_db
from app.routers.auth import get_current_user
```

```python
from app.utils.permissions import has_permission

router = APIRouter()

@router.post("/users/", response_model=schema_user.User)
def create_user(
    user: schema_user.UserCreate,
    db: Session = Depends(get_db),
    current_user: schema_user.User = Depends(get_current_user)
):
    if not has_permission(current_user.role_id, "users", "create"):
        raise HTTPException(status_code=403, detail="Not enough permissions")
    return crud_user.create_user(db, user)

@router.get("/users", response_model=list[schema_user.User])
def get_all_users(
    db: Session = Depends(get_db),
    current_user: schema_user.User = Depends(get_current_user)
):
    if not has_permission(current_user.role_id, "users", "read"):
        raise HTTPException(status_code=403, detail="Not enough permissions")
    users = crud_user.get_all_users(db)
    return users

@router.get("/users/{user_id}", response_model=schema_user.User)
def get_user(
    user_id: int,
    db: Session = Depends(get_db),
    current_user: schema_user.User = Depends(get_current_user)
):
    # Allow access if the user is accessing their own data
    if current_user.id != user_id and not has_permission(current_user.role_id,
"users", "read"):
        raise HTTPException(status_code=403, detail="Not enough permissions")
    db_user = crud_user.get_user(db, user_id)
    if db_user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return db_user

@router.delete("/users/{user_id}")
def delete_user(
    user_id: int,
    db: Session = Depends(get_db),
    current_user: schema_user.User = Depends(get_current_user)
):
    if not has_permission(current_user.role_id, "users", "delete"):
        raise HTTPException(status_code=403, detail="Not enough permissions")
    return crud_user.delete_user(db, user_id)
```

```
56
```

app/schemas/__init__.py

```
1
```

app/schemas/role.py

```python
from pydantic import BaseModel

class RoleBase(BaseModel):
    role_name: str

class RoleCreate(RoleBase):
    pass

class Role(RoleBase):
    id: int

    class Config:
        from_attributes = True

```

app/schemas/student.py

```python
from pydantic import BaseModel
from typing import Optional

class StudentBase(BaseModel):
    grade_level: str
    addional_info: Optional[str]
    food_allergy: Optional[str]

class StudentCreate(StudentBase):
    student_id: int
    main_parent_id: int
    secondary_parent_id: Optional[int]

class Student(StudentBase):
    id: int
    student_id: int

    class Config:
        from_attributes = True

```

app/schemas/user.py

```python
# app/schemas/user.py

from pydantic import BaseModel
```

```python
from typing import Optional

class UserBase(BaseModel):
    account_id: str
    last_name: str
    first_name: str
    email: Optional[str] = None
    phone: Optional[str] = None
    wechat_id: Optional[str] = None
    is_active: int = 0

class UserCreate(UserBase):
    password: str  # Changed from 'password_hash' to 'password'
    role_id: int   # Include role_id during user creation

class User(UserBase):
    id: int
    role_id: int

    class Config:
        from_attributes = True

```

app/utils/permissions.py

```python
# app/utils/permissions.py

role_permissions = {
    "admin": {
        "users": ["create", "read", "edit", "delete"],
        "students": ["create", "read", "edit", "delete"],
        "roles": ["create", "read", "edit", "delete"],
        # Add other resources as needed
    },
    "student": {
        "users": ["read", "edit"],
        "students": ["read", "edit"],
    },
    "teacher": {
        "users": ["read", "edit"],
        "students": ["read", "edit"],
        "classes": ["read", "edit"],
    },
    # Add other roles as needed
}

role_id_to_name = {
    1: "admin",
    2: "student",
```

```
    3: "teacher",
    4: "parent",
}

def has_permission(user_role_id, resource, action):
    role_name = role_id_to_name.get(user_role_id)
    if not role_name:
        return False
    permissions = role_permissions.get(role_name, {})
    actions = permissions.get(resource, [])
    return action in actions

```

app/utils/security.py

```
from passlib.context import CryptContext
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
def hash_password(plain_password):
    return pwd_context.hash(plain_password)
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)
```

app/utils/utils.py

```
import secrets

# Generate a random 32-byte hex string
secret_key = secrets.token_hex(32)
print(secret_key)

# on mac or linux
# openssl rand -hex 32
```