CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Building APIs

Outcomes

- Setting up a development environment
- Using a framework
- Separation of Concerns
- Using URL paths
- Authentication and Authorization

File Structure

```
.
├── src/
│   ├── controllers/
│   ├── models/
│   ├── routes/
│   └── services/
├── .gitignore
├── .env
├── dockerfile
├── main.py
├── readme.md
└── requirements.txt
```

Benefits

- Separate the backend operations from the API endpoints
- Make API versioning easier
- Makes the code easier to test
- Makes the code easier to manage

Installing Packages

We will be using

- fastapi
- uvicorn
- pydantic
- python-dotenv
- python-multipart
- jose
- sqlalchemy

Requirements.txt

What is this

- File containing all of the dependencies for our application
- Makes working with other developers easier as everyone can have the same packages
- Best used with a virtual environment

Setting up the project

1. Create virtual environment
2. Install Packages
3. Create files and folders
4. Set up FastAPI
5. Create models
6. Create services
7. Create routers
8. Add authentication and authorization

Create virtual environment

Create environment

```
python -m venv .venv
```

Start environment

Windows

```
source .venv/Script/activate
```

Mac/Linux

```
source .venv/bin/activate
```

Stop the Environment

```
deactivate
```

Install Packages

Run virtual environment

windows

```
pip install -r requirements.txt
```

Mac/Linux

```
pip3 install -r requirements.txt
```

File

```
.
├── src/
│   ├── controllers/
│   ├── models/
│   ├── routes/
│   └── services/
├── .gitignore
├── .env
├── dockerfile
├── main.py
├── readme.md
└── requirements.txt
```

Files : Environment Files

`.env`

Stores sensitive or dynamic values for our applications.

`.gitignore`

References the files that we do not want to add to our GitHub repository

`dockerfile`

Used to create a docker image, useful when you want to host the application

`requirements.txt`

References the packages required for the application.

`readme.md`

Information about the repository

Files: Folders

`main.py`

Main entry point of the application, stores the FastAPI set up

`src/`

Stores the files that will be used in the application.

`models/`

Represets the objects we use in our application.

`services/`

Stores the business logic for the application

`repository/`

Stores the connection to the database

`routers/`

Stores the endpoints for the API

`controllers/`

Links the `services` to the `routers`

Setup FastAPI

In the `main.py`

IMPORT PACKAGES

```
# Import system services
import os
from dotenv import load_dotenv

# Import FastAPI stuff
import uvicorn
from fastapi.middleware.cors import CORSMiddleware
from fastapi import FastAPI
from fastapi import APIRouter

# Import routers
# ADD ROUTERS HERE
```

Setup FastAPI

In the `main.py`

CREATE THE FASTAPI OBJECT

```
app = FastAPI()
```

Setup FastAPI

In the `main.py`

SET UP MIDDLEWARE

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=['*'],
    allow_credentials=True,
    allow_methods=['*'],
    allow_headers=['*']
)
```

Setup FastAPI

In the `main.py`

SET UP SERVER

```
if __name__ == "__main__":
    uvicorn.run('main:app', host=os.getenv("HOST"),
                port=int(os.getenv("PORT")),
                reload=os.getenv("REOLOAD"))
```

Notes

- We are using the `.env` file to get the host, port and reload values.
- 'main:app' refers to the `main.py` file name and `app = FastAPI()`

Set up FastAPI

In `.env`

```
HOST=localhost
PORT=8080
RELOAD=True
```

Create Model

Create `user_model.py` in the `src/models` directory

in `src/models/user_model.py`

IMPORT MODULES

```
from pydantic import BaseModel, Field
```

Notes

- pydantic allows us to easily create model classes for representing data
- It handles data validation for us and makes passing the data easier

Create Model

in `src/models/user_model.py`

CREATE CLASS

```python
class User(BaseModel):
    user_id: str
    role: Role
    first_name: str = Field(default=None)
    last_name: str = Field(default=None)
    email: str
    cart: list[str] = Field(default=[])
    wishlist: list[str] = Field(default=[])
```

Notes

- Inherit from the `BaseModel` to make the class a pydantic classs
- By default all fields are mandetory, we can use `Field()` to set some paramters to our attributes including adding default values which will make the values optional

Create Repository

Create the `database_repository.py` file in the `src/repository/` directory

In `src/repository/database_repository.py`

IMPORT MODULES

```
from sqlalchemy import create_engine, text, Table, MetaData
from sqlalchemy import select, insert, update, delete
```

Create Repository

In `src/repository/database_repository.py`

CREATE REPOSITORY CLASS

```python
class DatabaseRepository:

    def __init__(self, connection_string: str):
        self.__engine = create_engine(connection_string)

    def create(self, table_name: str, values: dict):
        table_name = Table(table_name, MetaData(), autoload_with=self.engine)

        with self.engine.connect() as conn:
            stmt = insert(table_name).values(values)
            conn.execute(stmt)
            conn.commit()

    def update(self, table_name: str, id: str, values: dict):
        table_name = Table(table_name, MetaData(), autoload_with=self.engine)

        with self.engine.connect() as conn:
            stmt = update(table_name).where(table_name.columns.id == id).values(values)
            conn.execute(stmt)
            conn.commit()
```

Create Repository

In `src/repository/database_repository.py`

CREATE REPOSITORY CLASS

```
class DatabaseRepository:
    ...

    def get_all(self, table_name: str,):
        table_name = Table(table_name, MetaData(), autoload_with=self.engine)

        with self.engine.connect() as conn:
            stmt = select(table_name)
            result = conn.execute(stmt)
            return result.fetchall()

    def get(self, table_name: str, id: str):
        table_name = Table(table_name, MetaData(), autoload_with=self.engine)

        with self.engine.connect() as conn:
            stmt = select(table_name).where(table_name.columns.id == id)
            result = conn.execute(stmt)
            return result.fetchone()
```

Create Repository

In `src/repository/database_repository.py`

CREATE REPOSITORY CLASS

```
class DatabaseRepository:
    ...

    def delete(self, table_name: str, id: str):
        table_name = Table(table_name, MetaData(), autoload_with=self.engine)

        with self.engine.connect() as conn:
            stmt = delete(table_name).where(table_name.columns.id == id)
            conn.execute(stmt)
            conn.commit()
```

Create Service

Create `user_service.py` in the `src/services/` directory

In `src/services/user_service.py`

IMPORT MODULES

```
import src.models.user import User
from src.repository.database_repository import DatabaseRepository
from sqlalchemy import select
import uuid
```

Create Service

In `src/services/user_service.py`

CREATE SERVICE CLASS

```
class UserService():
    TABLE_NAME = 'user'
    COLUMNS = ['id', 'role', 'first_name',
               'last_name', 'email', 'cart', 'wishlist']

    # If we were using dependency injection, we would pass the instance of Db Repo
    # and not create it in the constructor
    def __init__(self, connection_string: str) -> None:
        self.__db_repo = DatabaseRepository(connection_string)
```

Create Service In `src/services/user_service.py`

CREATE SERVICE CLASS

```
class UserService():
    ...
    def create(self, user: User):
        return self.__db_repo.create(self.TABLE_NAME, user.model_dump())

    def update(self, user: User):
        return self.__db_repo.update(self.TABLE_NAME, user.model_dump())

    def get_user(self, user_id: str):
        user_details = self.__db_repo.get(self.TABLE_NAME, user_id)
        cart = [value[1] for value in self.__get_shopping_cart(user_id)]
        wishlist = [value[1] for value in self.__get_wishlist(user_id)]

        if user_details is None:
            return None

        user_details = dict(zip(self.COLUMNS, user_details))
        user_details["cart"] = cart
        user_details["wishlist"] = wishlist

        return User(**user_details)
```

Create Service In `src/services/user_service.py`

CREATE SERVICE CLASS

```
class UserService():
    ...
    def __get_shopping_cart(self, user_id: str):
        table = self.__db_repo.get_table("cart")

        statment = select(table).where(table.c.user_id == user_id)

        shopping_cart = self.__db_repo.execute_statement(statment)

        return shopping_cart

    def __get_wishlist(self, user_id: str):
        table = self.__db_repo.get_table("wishlist")

        statment = select(table).where(table.c.user_id == user_id)
        wishlist = self.__db_repo.execute_statement(statment)

        return wishlist
```

Create Service In `src/services/user_service.py`

CREATE SERVICE CLASS

```
class UserService():
    ...
    def add_to_cart(self, user_id: str, product_id: str):
        id = str(uuid.uuid4())

        details = {
            "id": id,
            "user_id": user_id,
            "product_id": product_id
        }

        self.__db_repo.insert("cart", details)

    def add_to_wishlist(self, user_id: str, product_id: str):
        id = str(uuid.uuid4())

        details = {
            "id": id,
            "user_id": user_id,
            "product_id": product_id
        }

        self.__db_repo.insert("wishlist", details)
```

Create Controller

Create `user_controller.py` in the `src/controller/` directory

In `src/controller/user_controller.py`

IMPORT MODULES

```
from src.models.user import User
from src.services.user_service import UserService
```

Create Controller

In `src/controller/user_controller.py`

CREATE CLASS

```
class UserController():
    TABLE_NAME = "user"
    COLUMNS = ['user_id', 'role', 'first_name',
               'last_name', 'email', 'cart', 'wishlist']

    def __init__(self, connection_string):
        self.__service = UserService(connection_string)
```

Create Controller

In `src/controller/user_controller.py`

CREATE METHODS

```
def update(self, user: User):
        return self.__service.update(user)


    def get_user(self, user_id: str):
        return self.__service.get_user(user_id)

    def add_to_cart(self, user_id: str, product_id: str):
        return self.__service.add_to_cart(user_id, product_id)


    def add_to_wishlist(self, user_id: str, product_id: str):
        return self.__service.add_to_wishlist(user_id, product_id)

    def is_admin(self, user_id):
        return self.__service.is_admin(user_id)
```

Create Router

Create `user_router.py` in the `src/router/` directory

In `src/router/user_router.py`

IMPORT MODULES

```
from fastapi import APIRouter, Depends, HTTPException
from src.models.user import User
from src.controllers.user_controller import UserController
```

Create Router

In `src/router/user_router.py`

CREATE ROUTER OBJECT

```
router = APIRouter(
    prefix="/user",
    tags=["user"]
)
```

Notes

- `prefix` - The endpoint that we want to hit
  - If the base user is `localhost:8080`
  - user router will be at `localhost:8080/user`
- The endpoints that we create will build off the `prefix`

Create Router

In `src/router/user_router.py`

CREATE ENDPOINTS

```
controller = UserController('sqlite:///resources/store.db')

@router.get("/")
async def get_user(user: str ):
    result = controller.get_user(user)

    if result is None:
        raise HTTPException(status_code=404, detail="User not found")

    return {
        "user": result
    }

@router.put("/")
def update_user(user_details: User):

    controller.update(user_details)
    return {
        "message": "updated"
    }
```

Create Router

In `src/router/user_router.py`

CREATE ENDPOINTS

```
@router.post("/{id}/cart")
def add_to_cart(id: str, product_id: str):
    controller.add_to_cart(id, product_id)
    return {
        "message": "added to cart"
    }

@router.post("/{id}/wishlist")
def add_to_wishlist(id: str, product_id: str):
    controller.add_to_wishlist(id, product_id)
    return {
        "message": "added to wishlist"
    }
```

Notes

Now that we have all of the code, we can run the application. But before running the application, we need to make sure that we can reach our endpoints.

To do this, we will need to import our router in the `main.py` file and set up the routers to the app.

Setting up routers

In the `main.py` file

IMPORT ROUTER

```
from src.routers import user_router
```

Setting up routers

In the `main.py` file

ADD ROUTES

```
# ADD ROUTERS
app.include_router(user_router.router)
```

Running the API

To run the API, you can either run the following command in your terminal, or hit the run button in the `main.py` file.

```
python main.py
```

Authentication and Authorization

Authentication allows us to manage who has access to our application and authorization allows us to manage who has access to which parts of the application.

We will implement authorisation using emails and passwrod (username and password), and implement authorization using API Keys and JWT.

The packages required to do this will be in the `requirements.txt` file

- python-jose
- passlib
- bcrypt

Authentication and Authorization

Before we start, add a random series of characters as the `SECRET_KEY` in the `.env` file:

```
HOST=localhost
PORT=8080
RELOAD=True
SECRET_KEY=09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7
```

Authorization

In the `src/routers/user_router.py` file:

IMPORT MODULES

```
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
```

Authorization

create `src/routers/user_router.py` file

In the `src/routers/user_router.py` file:

IMPORT MODULES

```
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm, OAuth2PasswordBearer
from jose import JWTError, jwt

from src.controllers.user_auth_controller import UserAuthController
from src.models.user_auth_request_model import UserAuthRequest

import os
from dotenv import load_dotenv

load_dotenv()
```

Authorization

In the `src/routers/user_router.py` file:

CREATE API ROUTER

```
router = APIRouter(
    prefix="/auth",
    tags=["auth"]
)
```

Authorization In the `src/routers/user_router.py` file:

IMPORT MODULES

```
contoller = UserAuthController('sqlite:///resources/store.db')

@router.post("/", status_code=status.HTTP_201_CREATED)
async def create_user(user: UserAuthRequest):
    contoller.create_user(user)
    return {
        "message": "User created successfully"
    }

@router.post("/login")
async def login(form_data: OAuth2PasswordRequestForm = Depends()):
    user = contoller.authenticate_user(form_data.username, form_data.password)

    if not user:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid username or password")

    token = contoller.create_access_token(user['user_id'])

    return {
        "access_token": token,
        "token_type": "bearer"
    }
```

Authentication

Create the `src/services/auth_service.py` file

IMPORT MODULES

```
import os
from dotenv import load_dotenv

from datetime import timedelta, datetime
from passlib.context import CryptContext
from jose import jwt
from sqlalchemy import select

from src.models.user_auth_request_model import UserAuthRequest
from src.models.user_auth import UserAuth
from src.models.user import User

from src.repository.database_repository import DatabaseRepository
import uuid

load_dotenv()
```

Authentication

in the `src/services/auth_service.py` file

CREATE CLASS

```
class AuthService():
    SECRET_KEY = os.getenv("SECRET_KEY")
    ALGORITHM = "HS256"

    def __init__(self, connection_string: str) -> None:
        bcrypt = CryptContext(schemes=["bcrypt"], deprecated="auto")
        self.bcrypt = bcrypt

        self.__db_repo = DatabaseRepository(connection_string)
```

Authentication

in the `src/services/auth_service.py` file

ADD METHODS

```
class AuthService():
    ...

    def create_user(self, user: UserAuthRequest):
        user_id = str(uuid.uuid4())
        user_model = UserAuth(user_id=user_id, email=user.email, password_hash=self.bcrypt.hash(user.password))
        self.__db_repo.create("user_auth",  user_model.model_dump())

        self.__add_to_user_table(user_model)

    def __add_to_user_table(self, user: UserAuth):
        new_user = User(id=user.user_id, email=user.email, role=1)

        new_user = new_user.model_dump()
        new_user.pop('cart')
        new_user.pop('wishlist')

        self.__db_repo.create("user", new_user)
```

Authentication

in the `src/services/auth_service.py` file

ADD METHODS

```
class AuthService():
    ...

    def authenticate_user(self, email: str, password: str):
        table = self.__db_repo.get_table("user_auth")
        statement = select(table).where(table.c.email == email)
        result = self.__db_repo.execute_statement(statement)

        if not result:
            return None

        columns = ['user_id', 'email', 'password_hash']
        user = result[0]

        user = dict(zip(columns, user))

        if not self.bcrypt.verify(password, user['password_hash']):
            return None

        user.pop('password_hash')
        return user
```

Authentication

in the `src/services/auth_service.py` file

ADD METHODS

```
class AuthService():
    ...

    def create_access_token(self, user_id: str):
        expires_delta = timedelta(minutes=30)
        expires = datetime.utcnow() + expires_delta

        encode = {"id": user_id, "exp": expires}

        return jwt.encode(encode, self.SECRET_KEY, algorithm=self.ALGORITHM)
```

Create the `src/controllers/auth_controller.py` file:

IMPORT MODULES

```
from src.models.user_auth_request_model import UserAuthRequest
from src.services.api_token_service import ApiTokenService

from src.services.auth_service import AuthService
```

Create the `src/controllers/auth_controller.py` file:

CREATE CLASS

```
class UserAuthController():
    TABLE_NAME = "user_auth"
    ALGORITHM = "HS256"

    def __init__(self, connection_string: str):
        self.__api_token_service = ApiTokenService(connection_string)
        self.__auth_service = AuthService(connection_string)

    def create_user(self, user: UserAuthRequest):
        return self.__auth_service.create_user(user)

    def authenticate_user(self, email: str, password: str):
        return self.__auth_service.authenticate_user(email, password)
```

Authorization

In the `src/routers/user_router.py` file:

ADD METHOD (BEFORE ALL ENDPOINTS)

```
async def validate_user_token(token: str = Depends(OAuth2PasswordBearer(tokenUrl="auth/login"))):
    try:
        payload = jwt.decode(token, os.getenv("SECRET_KEY"), algorithms=['HS256'])
        user_id = payload.get("id")

        if not user_id:
            return False

    except JWTError:
        raise HTTPException(status_code=401, detail="Invalid token")

    return user_id
```

Authorization

In the `src/routers/user_router.py` file:

ADD AUTHORIZATION (UPDATE FUNCTIONS)

```
@router.put("/")
def update_user(user_details: User, user: str = Depends(validate_user_token)):
    if not user:
        raise HTTPException(status_code=401, detail="Unauthorized")

    if not controller.is_admin(user):
        raise HTTPException(status_code=401, detail="Unauthorized")


    controller.update(user_details)
    return {
        "message": "updated"
    }
```

Authorization

In the `src/routers/user_router.py` file:

ADD AUTHORIZATION (UPDATE FUNCTIONS)

```
@router.put("/")
def update_user(user_details: User, user: str = Depends(validate_user_token)):
    if not user:
        raise HTTPException(status_code=401, detail="Unauthorized")

    if not controller.is_admin(user):
        raise HTTPException(status_code=401, detail="Unauthorized")


    controller.update(user_details)
    return {
        "message": "updated"
    }
```

Authorization

In the `src/routers/user_router.py` file:

ADD AUTHORIZATION (UPDATE FUNCTIONS)

```
@router.post("/cart")
def add_to_cart(product_id: str, user: str = Depends(validate_user_token)):
    if not user:
        raise HTTPException(status_code=401, detail="Unauthorized")

    controller.add_to_cart(user, product_id)
    return {
        "message": "added to cart"
    }
```

Authorization

In the `src/routers/user_router.py` file:

ADD AUTHORIZATION (UPDATE FUNCTIONS)

```
@router.post("/{id}/wishlist")
def add_to_wishlist(id: str, product_id: str, user: str = Depends(validate_user_token)):
    if not user:
        raise HTTPException(status_code=401, detail="Unauthorized")

    controller.add_to_wishlist(id, product_id)
    return {
        "message": "added to wishlist"
    }
```

TEST THE API