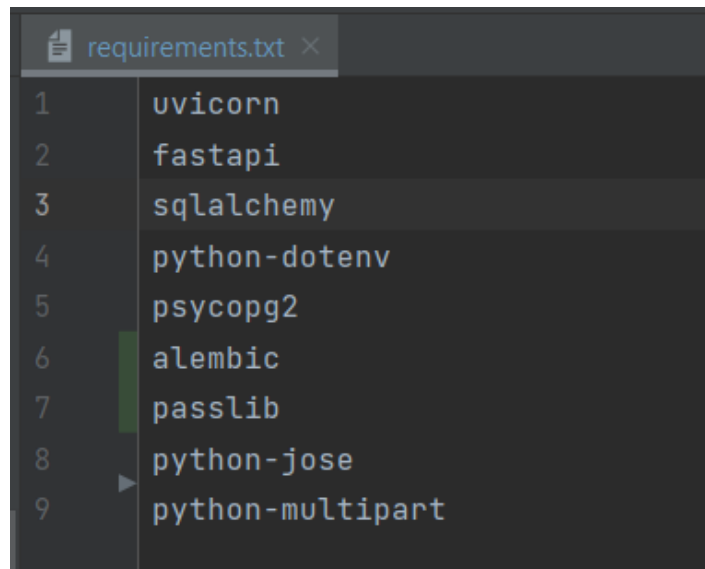


СПРИНТ 6
Разработка API на Python
Часть 2

Новые технологии и инструменты.....	2
Назначение миграций. Подход CodeFirst	2
Настройка Alembic	3
Создание и выполнение миграций.....	7
Создание сервиса авторизации. JWT	9
Создание эндпоинтов для регистрации и авторизации. Защита эндпоинтов.....	14
Загрузка и скачивание файлов	16

Новые технологии и инструменты

Во второй части спринта мы собираемся научиться выполнять миграции в базу данных, добавить в наш проект авторизацию и рассмотреть процессы загрузки и скачивания файлов. Для этого нам потребуется установка новых зависимостей. Дополним файл `requirements.txt`.



```
requirements.txt
1  uvicorn
2  fastapi
3  sqlalchemy
4  python-dotenv
5  psycpg2
6  alembic
7  passlib
8  python-jose
9  python-multipart
```

Разберемся, для чего новые зависимости:

- `alembic` – для создания и выполнения миграций в базу данных;
- `passlib` – для хэширования паролей и верификации пароля (сопоставление хэшированного и текстового представления пароля, для авторизации);
- `python-jose` – для реализации авторизации с использованием JWT;
- `python-multipart` – для возможности работы с файлами (загрузка файлов происходит через форму).

Назначение миграций. Подход CodeFirst

В первой части спринта мы начали с того, что у нас уже была созданная база данных. Мы подключили к ней наше приложение и все работало.

На практике, такой подход применяется редко. Требования заказчика могут изменяться быстро, поэтому куда удобнее синхронизировать модели данных, описанные в `python` средствами `sqlalchemy`, со схемой реальной базы данных.

Миграцией называется специальная структура данных, позволяющая перейти от одного состояния схемы БД к другому. Например, в начальном состоянии в БД нет никаких таблиц. Мы описали таблицы в виде моделей, создали миграцию. После ее

выполнения, в БД появились описанные средствами sqlalchemy таблицы (причем со всеми описанными ограничениями, внешними ключами и т. д.). Когда заказчик захотел добавить новое поле или новую таблицу – мы добавим поле в наш класс, создадим новую миграцию и выполним ее. Теперь в БД также появилось новое поле. Другой разработчик (или devops при развертывании) сможет скачать наш проект и перед его запуском выполнить все миграции, получив таким образом актуальную схему данных.

Подход CodeFirst – это подход к проектированию схемы БД, при котором схема БД получается в результате выполнения миграций, сформированных на основе классов исходного кода. Описанный подход применяется во многих технологиях разработки с использованием самых разных ORM.

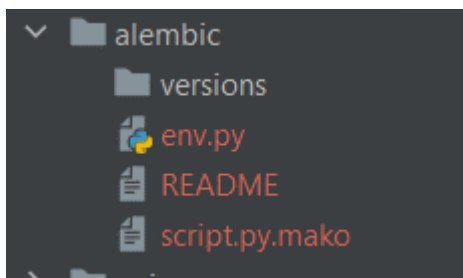
Настройка Alembic

Установив зависимости, приступим к настройке инструмента Alembic, который позволит нам выполнять миграции. На данный момент мы имеем пустую БД ascourse_2. Не забываем изменить эту информацию в строке подключения (файл .env).

Выполним в окружении следующую команду. В ней последний аргумент – путь к каталогу, где будет располагаться миграционное окружение (все, что касается миграций). Пусть это будет папка alembic в папке src.

```
(venv) PS D:\Programs\Python\as_course_2022\webapi_p2\src> alembic init alembic
Creating directory D:\Programs\Python\as_course_2022\webapi_p2\src\alembic ... done
Creating directory D:\Programs\Python\as_course_2022\webapi_p2\src\alembic\versions ... done
Generating D:\Programs\Python\as_course_2022\webapi_p2\src\alembic.ini ... done
Generating D:\Programs\Python\as_course_2022\webapi_p2\src\alembic\env.py ... done
Generating D:\Programs\Python\as_course_2022\webapi_p2\src\alembic\README ... done
Generating D:\Programs\Python\as_course_2022\webapi_p2\src\alembic\script.py.mako ... done
Please edit configuration/connection/logging settings in 'D:\Programs\Python\as_course_2022\webapi_p2\src\alembic.ini' before proceeding.
(venv) PS D:\Programs\Python\as_course_2022\webapi_p2\src>
```

Команда выполнена. В папке src появилась папка alembic.

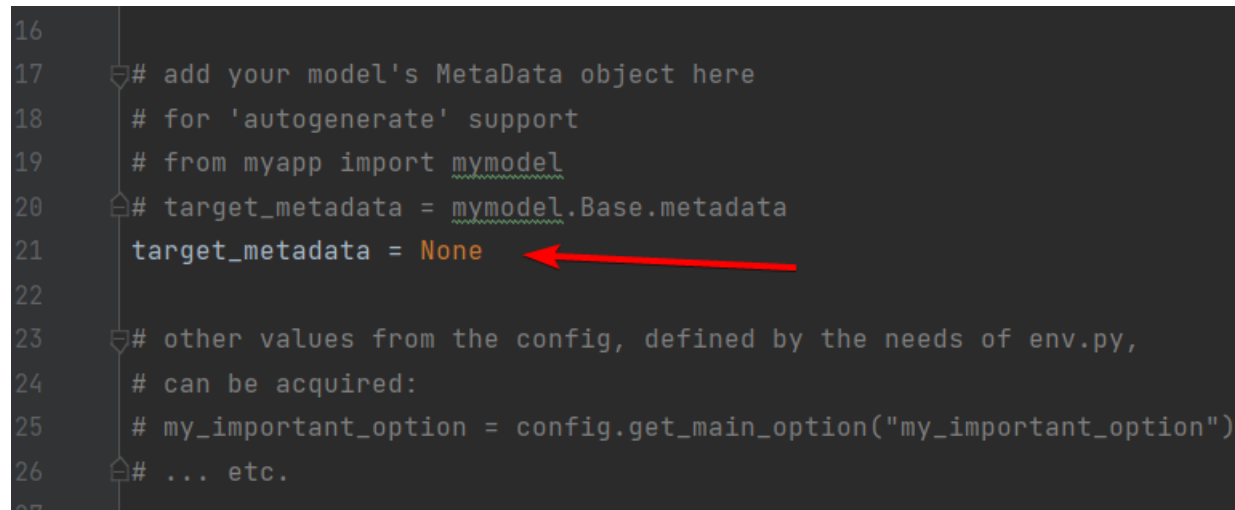


Также обратите внимание, что в самой папке `src` (откуда выполнялась команда) появился файл `alembic.ini`. Давайте разбираться, что за что отвечает и как нам с этим работать.

Файл `script.py.mako` содержит шаблон миграции. Мы его не будем использовать, также, как и файл `README` (который я удалю, он не нужен). В папку `versions` по умолчанию будут сохраняться миграции (мы можем указать место их сохранения).

Нам для конфигурации потребуется файл `env.py`. Этот скрипт будет выполняться при создании и выполнении миграции, здесь изначально много сгенерированного кода. Давайте сразу обратим внимание на несколько вещей, которые могут нам создать проблемы.

Во-первых, обратим внимание на переменную `target_metadata`.



```
16
17 # add your model's MetaData object here
18 # for 'autogenerate' support
19 # from myapp import mymodel
20 # target_metadata = mymodel.Base.metadata
21 target_metadata = None
22
23 # other values from the config, defined by the needs of env.py,
24 # can be acquired:
25 # my_important_option = config.get_main_option("my_important_option")
26 # ... etc.
```

Эта переменная отвечает за то, информация о каких моделях данных будет отслеживаться `alembic`. Вспомните, что мы наследовали все наши модели данных от единого класса `Base`, который изначально получили с помощью `sqlalchemy`. Таким образом, нам необходимо импортировать `Base` и изменить `target_metadata` следующим образом.

```

8  from src.models import base
9
10 # this is the Alembic Config object, which provides
11 # access to the values within the .ini file in use.
12 config = context.config
13
14 # Interpret the config file for Python logging.
15 # This line sets up loggers basically.
16 if config.config_file_name is not None:
17     fileConfig(config.config_file_name)
18
19 # add your model's MetaData object here
20 # for 'autogenerate' support
21 # from myapp import mymodel
22 # target_metadata = mymodel.Base.metadata
23 target_metadata = base.Base.metadata
24

```

Второй момент – оказывается этого недостаточно. Несмотря на то, что еще как-либо изменять `target_metadata` нам не нужно, мы должны импортировать в скрипт все модули, в которых описаны таблицы, которые мы собираемся учитывать в миграциях (это наши модели данных, `category`, `article`).

```

7
8  from src.models import base, article, category
9

```

Третий момент, нам нужно сконфигурировать подключение к БД при выполнении миграций. Здесь мы воспользуемся нашим объектом `settings`.

Важно понимать, что существует 2 метода выполнения миграция – `offline` и `online`. Результат их выполнения один и тот же, однако `online`-миграцию можно выполнить во время работы клиента с СУБД. В скрипте `env` прописаны методы для выполнения миграций двумя способами. Можно увидеть, что по умолчанию строка подключения к БД (URL) берется из некоторого `config`. На самом деле, оно берется из файла `alembic.ini` (про который я выше говорил). Там конфигурируется множество параметров, с которыми вы сможете разобраться самостоятельно. Однако строку подключения я рекомендую брать не из этого файла, а из нашего единого источника – объекта `settings`. Поэтому меняем скрипт в двух местах.

```

42
43     """
44     url = settings.connection_string
45     context.configure(
46         url=url,
47         target_metadata=target_metadata,
48         literal_binds=True,
49         dialect_opts={"paramstyle": "named"},
50     )
51
52     with context.begin_transaction():
53         context.run_migrations()
54

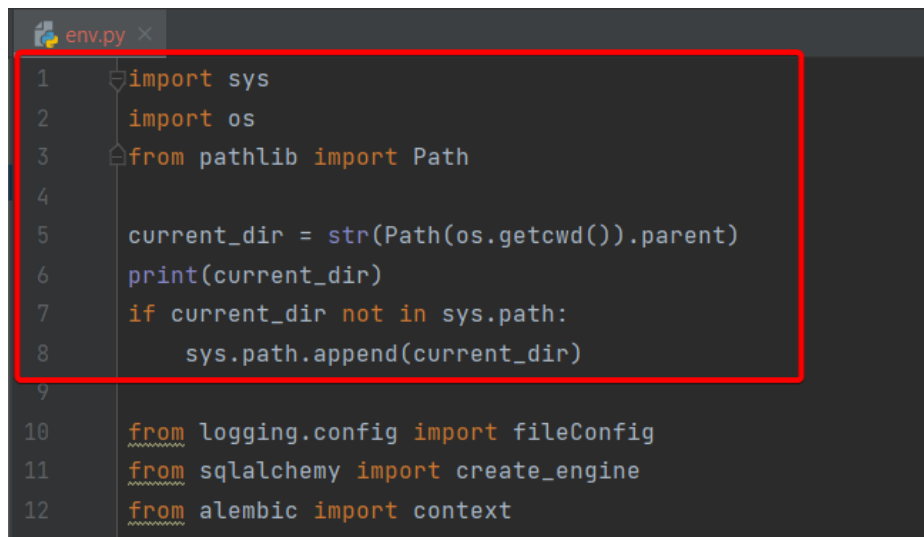
```

```

61
62     """
63     connectable = create_engine(
64         settings.connection_string
65     )
66
67     with connectable.connect() as connection:
68         context.configure(
69             connection=connection, target_metadata=target_metadata
70         )
71

```

Вроде как у нас все готово, однако попытка вызова скрипта `env` в ходе выполнения команд `alembic` приведет к ошибке. Python не сможет найти модуль `src`, из которого мы импортировали модели данных и настройки. Для решения этой проблемы допускается применить небольшой костыль – добавить родительскую для папки `src` директорию в переменную `PYTHONPATH` в процессе выполнения этого скрипта (если ее там нет). Идея в том, что существует переменная среды `PYTHONPATH`, в которой находится список директорий, который воспринимаются как папки с модулями Python, которые могут быть импортированы. В самом начале, до импортов (это важно), выполним следующие действия.



```

1 import sys
2 import os
3 from pathlib import Path
4
5 current_dir = str(Path(os.getcwd()).parent)
6 print(current_dir)
7 if current_dir not in sys.path:
8     sys.path.append(current_dir)
9
10 from logging.config import fileConfig
11 from sqlalchemy import create_engine
12 from alembic import context

```

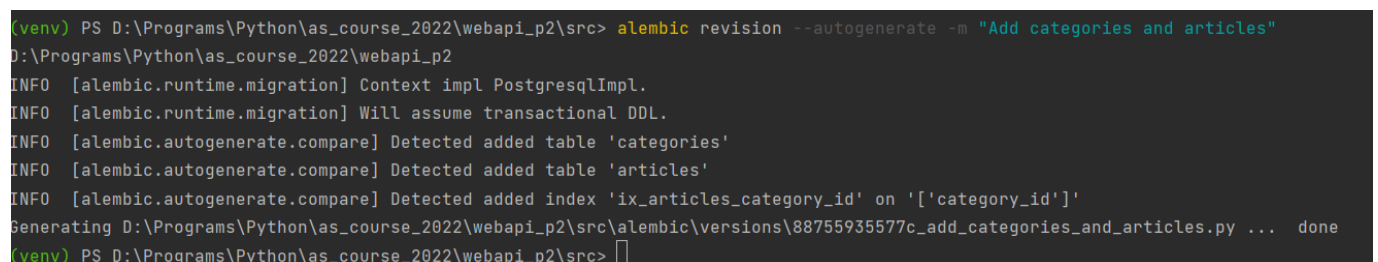
Используем модуль `pathlib`, находим родительскую для `src` папку и добавляем ее в `sys.path` (да, `current_dir` здесь – не самое удачное название; рекомендую придумать другое). Обращаю внимание – мы будем использовать этот скрипт в составе команды `alembic`, команду будем исполнять из папки `src`. Поэтому `parent` получаем один раз, ведь `getcwd()` вернет нам путь до папки `src`.

Базовая настройка `Alembic` завершена. Можно создавать и выполнять миграции.

Создание и выполнение миграций

Обращаю внимание – для выполнения миграция БД должна существовать! Я создал БД `ascourse_2`, она пустая. В `env` строку подключения поменял.

Для создания миграции выполним следующую команду в окружении (находясь при этом в папке `src`).



```

(venv) PS D:\Programs\Python\as_course_2022\webapi_p2\src> alembic revision --autogenerate -m "Add categories and articles"
D:\Programs\Python\as_course_2022\webapi_p2
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO [alembic.runtime.migration] Will assume transactional DDL.
INFO [alembic.autogenerate.compare] Detected added table 'categories'
INFO [alembic.autogenerate.compare] Detected added table 'articles'
INFO [alembic.autogenerate.compare] Detected added index 'ix_articles_category_id' on '['category_id']'
Generating D:\Programs\Python\as_course_2022\webapi_p2\src\alembic\versions\88755935577c_add_categories_and_articles.py ... done
(venv) PS D:\Programs\Python\as_course_2022\webapi_p2\src>

```

Можем увидеть, что в папке `versions` появилась миграция.

```

88755935577c_add_categories_and_articles.py x
17
18
19 def upgrade() -> None:
20     # ### commands auto generated by Alembic - please adjust! ###
21     op.create_table('categories',
22         sa.Column('id', sa.Integer(), nullable=False),
23         sa.Column('name', sa.String(), nullable=True),
24         sa.Column('description', sa.String(), nullable=True),
25         sa.PrimaryKeyConstraint('id')
26     )
27     op.create_table('articles',
28         sa.Column('id', sa.Integer(), nullable=False),
29         sa.Column('title', sa.String(), nullable=True),
30         sa.Column('body', sa.String(), nullable=True),
31         sa.Column('category_id', sa.Integer(), nullable=True),
32         sa.ForeignKeyConstraint(['category_id'], ['categories.id'], ),
33         sa.PrimaryKeyConstraint('id')
34     )

```

Видно, что alembic восстановил последовательность действий, которые необходимо произвести с БД, чтобы получить нужную нам схему таблиц. Но в самой БД пока ничего не произошло, поскольку мы только создали миграцию. Теперь ее необходимо применить. Для этого выполним следующую команду.

```

(env) PS D:\Programs\Python\as_course_2022\webapi_p2\src> alembic upgrade head
D:\Programs\Python\as_course_2022\webapi_p2
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO [alembic.runtime.migration] Will assume transactional DDL.
INFO [alembic.runtime.migration] Running upgrade -> 88755935577c, Add categories and articles
(env) PS D:\Programs\Python\as_course_2022\webapi_p2\src>

```

После выполнения команды видим, что в БД появились нужные нам таблицы. Попрактикуемся с миграциями — для дальнейшей работы с авторизацией нам понадобится таблица users. В пакете models создадим файл user и опишем модель.

```

user.py x
1 from sqlalchemy import Column, Integer, String
2 from src.models.base import Base
3
4 Mikhail Gunenkov
5 class User(Base):
6     __tablename__ = 'users'
7     id = Column(Integer, primary_key=True)
8     username = Column(String)
9     password_hash = Column(String)
10

```

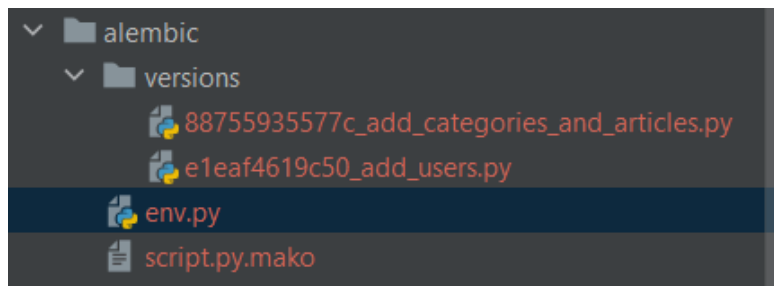
Для того, чтобы alembic увидел эту модель, вернемся в скрипт env и импортируем модуль user.


```

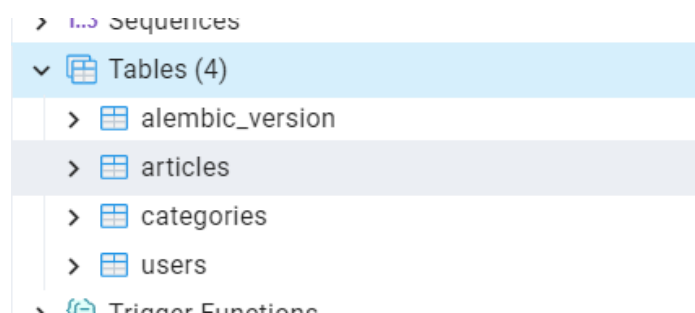
13 from src.core.settings import settings
14 from src.models import base, article, category, user
15

```

Создадим новую миграцию. В папке versions видим обе миграции.



Выполним ее. Схема базы данных готова.



Обращаю внимание — работать с миграциями следует крайне осторожно. Alembic отслеживает порядок создания миграций. При выполнении нескольких миграций они выполняются последовательно в том порядке, в котором создавались. Удаление одной из промежуточных миграций (или ошибка в выполнении промежуточной миграции) остановит процесс выполнения миграций. При этом процесс выполнения миграций является транзакцией — либо выполняются все, либо ни одна. Будьте осторожны!

Создание сервиса авторизации. JWT

Создадим файл users в пакете services. Начнем реализацию сервиса с определения конструктора (в котором будем получать объект сессии) и с пары статических методов для хэширования и верификации пароля.

```

21 class UserService:
22     def __init__(self, session: Session = Depends(get_session)):
23         self.session = session
24
25     new *
26     @staticmethod
27     def hash_password(password: str) -> str:
28         return pbkdf2_sha256.hash(password)
29
30     @staticmethod
31     def check_password(password_text: str, password_hash: str) -> bool:
32         return pbkdf2_sha256.verify(password_text, password_hash)

```

Итак, что происходит в конструкторе нам уже понятно. Метод `hash_password` принимает на вход пароль в текстовом виде и возвращает его хэш, полученный алгоритмом PBKDF2 на базе SHA256. Метод `check_password` принимает пароль в текстовом виде и в виде хэша (полученный, например, из БД) и сравнивает их.

Можно сделать промежуточный и крайне важный вывод – пароли от клиента будут приходить в открытом виде, после чего будут сравниваться с содержащимся в БД хэшем.

Далее необходимо поговорить о JWT (Json Web Token). Это токен, хранящий информацию в формате base64. Во многих современных приложениях авторизация работает с использованием JWT. Рассмотрим схему:

- все защищенные эндпоинты (требующие авторизацию) каким-либо образом проверяют наличие авторизационного заголовка в запросе (в случае использования JWT, заголовок `Authorization` должен содержать значение `Bearer <токен>`); в случае, если токен не предоставлен, запрос отменяется, считается, что клиент не авторизован;
- если токен есть, он валидируется; токен подписывается при помощи некоторого алгоритма (который неизвестен клиенту) с некоторым ключом (который тем более неизвестен клиенту); в процессе валидации проверяется подпись, а также время жизни токена (если токен устарел, то пользователю необходимо получить новый);

- токен можно получить, выполнив авторизацию (запрос на открытый эндпоинт авторизации с юзернеймом и паролем); токен хранится на клиенте и используется при выполнении запросов к защищенным эндпоинтам.

Нам необходимо добавить в наш сервис методы для создания и валидации токенов.

Для начала нам потребуется описать вспомогательную схему данных, которая будет представлять собой объект с токеном. В пакете `models/schemas` создадим пакет `utils`. В нем создадим файл `jwt-token`. Опишем структуру.

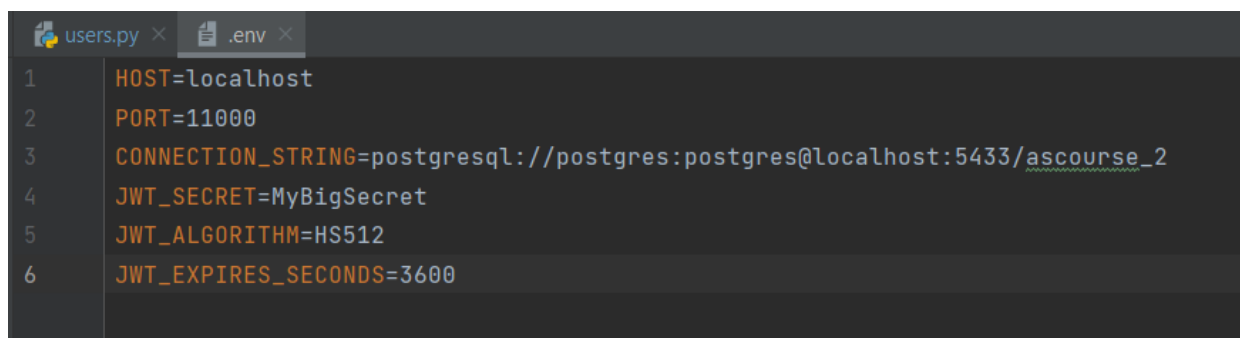
```
1 from pydantic import BaseModel
2
3
4 class JwtToken(BaseModel):
5     access_token: str
6
```

Для упрощения последующих действий важно, чтобы поле с токеном имело название `access_token`. Мы можем добавлять другие поля, однако в процессе создания токена нам следует вернуть объект данной структуры и положить токен в поле `access_token`.

Реализуем метод для создания токена на основе `id` пользователя (мы его положим в токен, чтобы потом, на основе этого токена, переданного в заголовке авторизации, можно было определить пользователя, от имени которого выполняется запрос).

```
users.py x
44 return payload.get('sub')
45
46 @staticmethod
47 def create_token(user_id: int) -> JwtToken:
48     now = datetime.utcnow()
49     payload = {
50         'iat': now,
51         'exp': now + timedelta(seconds=settings.jwt_expires_seconds),
52         'sub': str(user_id),
53     }
54     token = jwt.encode(payload, settings.jwt_secret, algorithm=settings.jwt_algorithm)
55     return JwtToken(access_token=token)
56
```

Обратите внимание, что в settings были добавлены новые поля (соответственно в файл .env тоже) – это секретный ключ токена (строка), алгоритм (строка) и время жизни в секундах (число).



```
users.py x .env x
1 HOST=localhost
2 PORT=11000
3 CONNECTION_STRING=postgresql://postgres:postgres@localhost:5433/ascourse_2
4 JWT_SECRET=MyBigSecret
5 JWT_ALGORITHM=HS512
6 JWT_EXPIRES_SECONDS=3600
```

В ходе заполнения полей токена, нам необходимо заполнить поля iat и exp. Остальные поля заполняются по желанию. Более того, можем добавлять в токен свои собственные поля, для сохранения, например, ролей пользователя (подсказка для домашнего задания). Полученный токен мы можем ввести на сайте <https://jwt.io/> и посмотреть всю содержащуюся в нем информацию.

Теперь реализуем метод для верификации (проверки) токена.



```
Mikhail Gunenkov *
34 @staticmethod
35 def verify_token(token: str) -> Optional[int]:
36     try:
37         payload = jwt.decode(token, settings.jwt_secret, algorithms=[settings.jwt_algorithm])
38     except JWTError:
39         raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Некорректный токен")
40
41     return payload.get('sub')
```

Если валидация токена провалится, то мы немедленно вернем ошибку с кодом 401 (не авторизован). Если все хорошо – вернем id пользователя (которое заведомо положили в поле sub).

Последние методы, которые мы реализуем в этом сервисе, это методы для выполнения регистрации и авторизации пользователя.

```

Mikhail Gunenkov
def register(self, user_schema: UserRequest) -> None:
    user = User(
        username=user_schema.username,
        password_hash=self.hash_password(user_schema.password_text),
    )
    self.session.add(user)
    self.session.commit()

```

```

Mikhail Gunenkov
2 def authorize(self, username: str, password_text: str) -> Optional[JwtToken]:
3     user = (
4         self.session
5         .query(User)
6         .filter(User.username == username)
7         .first()
8     )
9
10    if not user:
11        return None
12    if not self.check_password(password_text, user.password_hash):
13        return None
14
15    return self.create_token(user.id)

```

При регистрации мы просто добавляем информацию о новом пользователе в БД (не забывая о хэшировании пароля). В данном примере не обрабатывается ситуация, когда пытаемся зарегистрироваться с логином, который в БД уже есть. При авторизации мы пробуем сначала найти пользователя по имени. Если не нашли – возвращаем None. Если нашли – проверяем пароль. Можно None не возвращать, а прямо здесь кинуть исключение с кодом 401 (в данном случае, исключение будет выкидываться в роутере, как вы, наверное, догадались, если вернется None).

Но в этом сервисе нам потребуется сделать еще одну штуку.

```

15 oauth2_schema = OAuth2PasswordBearer(tokenUrl='/users/authorize')
16
17
Mikhail Gunenkov
18 def get_current_user_id(token: str = Depends(oauth2_schema)) -> int:
19     return UsersService.verify_token(token)

```

OAuth2PasswordBearer – это класс из FastAPI. С его помощью мы можем создать объект специальной схемы (указав url, по которому можно получить токен). Теперь мы можем использовать внедрение зависимостей для того, чтобы в любой

момент получить токен пользователя, переданный в заголовке авторизации. Но мы пойдем дальше – мы сделаем функцию для получения id текущего пользователя – и уже используя эту функцию как зависимость для внедрения будем внедрять в методы роутеров id текущего пользователя. Самое интересное, что такая схема работы для FastAPI является естественной. Мы это увидим, когда будем работать с эндпоинтами. Сервис готов.

Создание эндпоинтов для регистрации и авторизации. Защита эндпоинтов

Перед непосредственным созданием эндпоинтов, давайте создадим необходимые схемы данных (которые будем передавать при регистрации и авторизации). В пакете `models/schemas` создадим пакет `user`. В нем создадим файл `user_request`.

```
1  from pydantic import BaseModel
2
3
4  class UserRequest(BaseModel):
5      username: str
6      password_text: str
7
```

Теперь в пакете `api` создадим файл (роутер) `users` (не забудьте его включить в базовый роутер) и в первую очередь реализуем метод для регистрации.

```
8
9  router = APIRouter(
10     prefix='/users',
11     tags=['users'],
12 )
13
14
15  @router.post('/register', status_code=status.HTTP_201_CREATED, name='Регистрация')
16  def register(user_schema: UserRequest, users_service: UsersService = Depends()):
17      return users_service.register(user_schema)
18
```

Видно, что в нашей реализации создание пользователя не отличается от создания категории (разве что внедряем другой сервис). Куда интереснее обстоят дела с методом для выполнения авторизации.

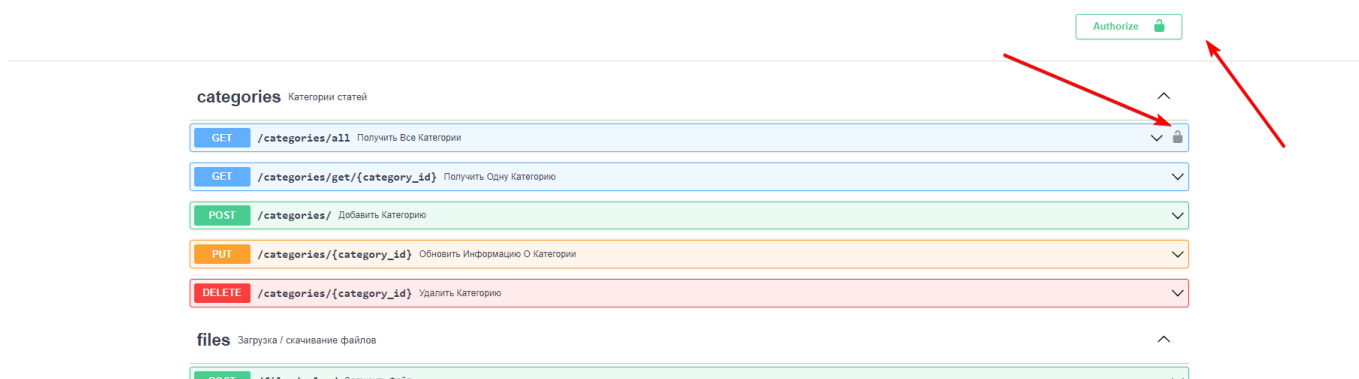
```
Mikhail Gunenkov
20 @router.post('/authorize', response_model=JwtToken, name='Авторизация')
21 def authorize(auth_schema: OAuth2PasswordRequestForm = Depends(), users_service: UsersService = Depends()):
22     result = users_service.authorize(auth_schema.username, auth_schema.password)
23     if not result:
24         raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail='Не авторизован')
25     return result
26
```

В качестве ответа сервера мы ожидаем получить описанную ранее схему `JwtToken`. Метод выполняет авторизацию и возвращает нам токен в случае успешной авторизации. Обратите также внимание на схему данных, которую принимает метод — это снова встроенный в FastAPI класс формы. Там обязательные поля — это логин и пароль. Почему используем его, а не просто схему с логином и паролем? Потому что использование этого класса облегчает работу с интерактивной документацией и позволяет корректно выполнять авторизацию внутри Swagger UI для выполнения запросов, требующих авторизацию. Вы это увидите. Как на данном этапе выглядит Swagger UI с новыми эндпоинтами спойлерить не буду 😊

Теперь давайте разберемся с последней вещью в данном разделе — как нам защитить эндпоинты (чтобы при обращении к эндпоинту требовалась авторизация). Откроем роутер для категорий и для примера сделаем возможность получения списка всех категорий только для авторизованных пользователей.

```
@router.get('/all', response_model=List[CategoryResponse], name='Получить все категории')
def get(categories_service: CategoriesService = Depends(), user_id: int = Depends(get_current_user_id)):
    """
    Получить все категории. Более подробное описание.
    """
    print(user_id)
    return categories_service.all()
```

Что называется, найдите 10 отличий! Чтобы разобраться в том, что мы сделали, посмотрим, как выглядит этот эндпоинт в сваггере.



В сваггере появилась кнопка авторизации (для авторизации в Swagger UI и выполнения запросов, требующих авторизации). Также все методы, у которых мы будем внедрять нашу новую зависимость (функцию для получения id пользователя) будут считаться защищенными, о чем будет свидетельствовать иконка замка. Но не само внедрение функции будет творить такую магию, а внедрение той самой магической схемы, которое мы сделали в функции на последнем этапе разработки сервиса.

Самое приятное в том, что мы теперь сможем легко узнать, от имени какого пользователя выполняется защищенный запрос. Если мы в jwt записали другую информацию — ее мы также сможем узнать. А используя показанное внедрение зависимости мы можем защищать методы API. На этом про авторизацию все.

Загрузка и скачивание файлов

В заключении этого вводного спринта рассмотрим вопросы загрузки и скачивания файлов. Будем работать с файлами в формате csv, которые часто используются для обучения моделей и получения предсказаний (когда данные структурированы — в Газпром нефти такое часто бывает).

Здесь нам не потребуются новые модели / схемы данных. Создадим сервис и роутер. Начнем с создания сервиса. Создадим в пакете services файл files.

```
1 import csv
2 from io import StringIO
3 from typing import BinaryIO
```



```

9 class FileService:
    Mikhail Gunenkov
10 @staticmethod
11 def upload(file: BinaryIO):
12     reader = csv.DictReader((line.decode() for line in file))
13     for row in reader:
14         print(row)
15
    Mikhail Gunenkov *
16 @staticmethod
17 def download():
18     output = StringIO()
19     writer = csv.DictWriter(output, fieldnames=["id", "name", "age"])
20     writer.writeheader() # если хотим записать названия столбцов
21     writer.writerow({"id": 3, "name": "Alina", "age": None})
22     output.seek(0)
23     return output

```

Здесь нам не понадобится получать сессию для работы с БД (хотя в домашнем задании понадобится).

Метод `upload` будет использоваться для построчного чтения файла и вывода информации в терминал. Метод `download` будет использоваться для формирования выходного файла в формате `csv`. В конце работы обязательно выполнить `seek(0)` — чтобы установить «виртуальный курсор» в файле на его начало.

Теперь создадим роутер в пакете `api` (назовем его `files`). Не забудем включить его в базовый роутер. Реализуем здесь нужные нам методы. Начнем с загрузки файла.

```

1 from fastapi import APIRouter, BackgroundTasks, UploadFile, Depends
2 from fastapi.responses import StreamingResponse
3
4 from src.services.files import FileService
5
6 router = APIRouter(
7     prefix='/files',
8     tags=['files'],
9 )
10
    Mikhail Gunenkov
11 @router.post('/upload', name='Загрузить файл')
12 def upload(background_tasks: BackgroundTasks, file: UploadFile, files_service: FileService = Depends()):
13     background_tasks.add_task(files_service.upload, file.file)

```

Обратите внимание на то, что здесь демонстрируется создание фоновой задачи. Файлы могут быть большими. Если задача будет выполняться в основном потоке, то

ответ на запрос придет клиенту только после окончания обработки всего файла. Если мы хотим вернуть пользователю результат, не дожидаясь обработки, то можем создать фоновую задачу на обработку файла и продолжить выполнение метода. Далее в методе можно выполнять другие действия – обработка с файлом будет производиться в фоне.

Если мы хотим отправить файл с клиента, то делаем это с использованием формы (FormData). При этом обращаю внимание – имя элемента с файлом в форме должно совпадать с названием переменной (в данном случае, «file»).

Что касается фоновой задачи – она запускается с помощью метода `add_task()`, куда передается сначала ссылка на метод (который требуется запустить в фоне), а затем, последовательно, принимаемые им параметры.

Теперь реализуем скачивание файла.

A screenshot of a code editor with a dark background. At the top left, it says 'Mikhail Gunenkov'. The code is in Python and defines a router view for downloading a file. It starts with a line number 7 and a decorator `@router.get('/download', name='Скачать файл')`. Line 8 has `def download(files_service: FilesService = Depends()):`. Line 9 has `report = files_service.download()`. Line 10 has `return StreamingResponse(report, media_type='text/csv',`. Line 11 has `headers={'Content-Disposition': 'attachment; filename=report.csv'})`. Line 12 is empty. There are line numbers 7, 8, 9, 10, 11, 12 on the left margin.

```
7 @router.get('/download', name='Скачать файл')
8 def download(files_service: FilesService = Depends()):
9     report = files_service.download()
10    return StreamingResponse(report, media_type='text/csv',
11                             headers={'Content-Disposition': 'attachment; filename=report.csv'})
12
```

Здесь мы будем возвращать клиенту объект `StreamingResponse`. Это позволит при обращении к эндпоинту сразу же начать скачивание файла после выполнения всех действий.

Действия с файлами можно протестировать в Swagger UI.

POST /files/upload Загрузить Файл

Parameters Cancel Reset

No parameters

Request body required multipart/form-data

file * required
string(\$binary) Выберите файл test.txt

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:11000/files/upload' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@test.txt;type-text/plain'
```

Request URL

http://localhost:11000/files/upload

Server response

Code	Details
200	Response body null

GET /files/download Скачать Файл

Parameters Cancel

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:11000/files/download' \
  -H 'accept: application/json'
```

Request URL

http://localhost:11000/files/download

Server response

Code	Details
200	Response body Download file Response headers

На этом спринт, посвященный разработке API, заканчивается. Мы разобрали большинство моментов, которые необходимо учитывать в реальной разработке. Конечно, о некоторых тонкостях мы не успели поговорить. Рекомендую вам самостоятельно обратить на них внимание:

- технология CORS, ее настройка (пригодится сразу, как только попытаетесь где-либо развернуть ваше приложение);
- работа с вебсокетами;
- работа с куки и заголовками запросов.

И даже это — всего лишь основы...

Желаю вам успехов в непрерывном обучении!