Книга по FastAPI

ССЫЛКА - https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI
GITHUB - https://github.com/Kematin/FastAPI-book

Команды по установке и сбору пакетов:

```
pip install fastapi uvicorn
pip freeze > requirements.txt
pip install -r requirements.txt
```

> chapter 1
>
> ## CHAPTER 1
>
> ---
>
> ## Docker
>
> ---
>
> Использование Dockerfile и файла docker-compose избавляет от необходимости загружать образы наших приложений и делиться ими. Новые версии наших приложений можно создавать из файла Dockerfile и развертывать с помощью файла docker-compose. Образы приложений также можно хранить и извлекать из Docker Hub. Это известно, как операция толкания и вытягивания.
>
> **Dockerfile**
>
> ```
> FROM PYTHON:3.8
> # Set working directory to /usr/src/app
> WORKDIR /usr/src/app
> # Copy the contents of the current local directory into the container's working
> directory
> ADD . /usr/src/app
> # Run a command
> CMD ["python", "hello.py"]
> ```
>
> **Создание и запуск контейнера**
>
> ```
> docker build -t getting_started .
> # docker build -t api api/Dockerfile
> ```

```
docker run getting_started
```

Первое приложение

api.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def welcome() -> dict:
    return {
        "message": "Hello world!"
    }
```

Запуск

```
uvicorn api:app --port 8000 --reload
```

chapter 2

Маршрутизация

API ROUTER - класс fastapi, который помогает создавать сложные маршруты ,т.к. в дефолтном классе FastAPI это недоступно.

todo.py

```
from fastapi import APIRouter

todo_router = APIRouter()
todo_list = list()


@todo_router.post("/todo")
async def add_task(todo: dict) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
```

```
    }


@todo_router.get("/todo")
async def retrieve_tasks() -> dict:
    return {
        "tasks": todo_list
    }
```

api.py

```
from fastapi import FastAPI
from todo import todo_router

app = FastAPI()

@app.get("/")
async def welcome() -> dict:
    return {
        "message": "Hello world!"
    }

# Include todo router to main route
app.include_router(todo_router)
```

Валидация

В FastAPI тела запросов могут быть проверены, чтобы гарантировать отправку только определенных данных. Это крайне важно, поскольку служит для очистки данных запросов и снижения рисков вредоносных атак. Этот процесс известен как валидация.

Что такое Pydantic?
Pydantic — это библиотека Python, которая выполняет проверку данных с помощью аннотаций типа Python.

models.py

```
from pydantic import BaseModel

class Item(BaseModel):
    item: str
```

```
    status: bool

class TodoItem(BaseModel):
    id: int
    item: Item
```

todo.py

```
from models import TodoItem
...

@todo_router.post("/todo")
async def add_task(todo: TodoItem) -> dict:
        ...
```

Параметры пути

Параметры пути — это параметры, включенные в маршрут API для идентификации ресурсов. Эти параметры служат идентификатором, а иногда и связующим звеном, позволяющим выполнять дальнейшие операции в веб-приложении.

todo.py

```
from fastapi import APIRouter, Path
...

...
@todo_router.get("/todo/{todo_id}")
async def retrieve_single_task(todo_id:
                               int = Path(...,
                                          title="The ID of the task to
retrieve.")
                               ) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return {"task": todo}
    else:
        return {"error": f"Task with id {todo_id} not found."}
```

Модуль Path из FastAPI

Typehint = Path(..., kwargs)
Класс Path принимает первый позиционный аргумент, равный None или многоточие (...). Если в качестве первого аргумента задано многоточие (...), параметр пути становится обязательным. Класс Path также содержит аргументы, используемые для числовой проверки, если параметр пути является числом. Определения включают gt и le – gt означает больше, а le означает меньше. При использовании маршрут будет проверять параметр пути на соответствие этим аргументам.
Он так-же помогает в создание автоматической документации swagger.

Параметры запроса

Параметр запроса — это необязательный параметр, который обычно появляется после вопросительного знака в URL-адресе. Он используется для фильтрации запросов и возврата определенных данных на основе предоставленных запросов.

Пример из книги

```
async query_route(query: str = Query(None):
        return query
```

Тело запроса

---

*Тело запроса — это данные, которые вы отправляете в свой API, используя метод маршрутизации, такой как POST и UPDATE.

Когда мы делаем запрос по пути:

```
@todo_router.post("/todo")
async def add_task(todo: TodoItem):
    todo_list.append(todo)
    return {"message": f"Add {todo}"}
```

Тело запроса у выглядит так:

```
{
        "id": 2,
        "item": "Validation models help with input types.."
}
```

chapter 3

Модели ответов и обработка ошибок

_Модели ответов служат шаблонами для возврата данных из пути маршрута API. Они построены на Pydantic для правильной обработки ответов на запросы, отправленные на сервер.

Обработка ошибок включает методы и действия, связанные с обработкой ошибок в приложении. Эти методы включают возврат адекватных кодов состояния ошибки и сообщений об ошибках.

Коды состояния

- 1XX: Запрос получен.
- 2XX: Запрос выполнен успешно.
- 3XX: Запрос перенаправлен.
- 4XX: Ошибка клиента.
- 5XX: Ошибка сервера.

Полный список кодов - https://httpstatuses.com/

Построение моделей ответов

*Чтобы указать
возвращаемую информацию, нам пришлось бы либо отделить отображаемые данные, либо ввести дополнительную логику. К счастью, мы можем создать модель,
содержащую поля, которые мы хотим вернуть, и добавить ее в определение нашего
маршрута, используя аргумент response_model.*

models.py

```
# Model for the record to db
class Todo(BaseModel):
    id: int
    item: str

    class Config:
```

```
        json_schema_extra = {
            "example": {
                "id": 1,
                "item": "example schema!"
            }
        }


# Response model
class TodoItem(BaseModel):
    item: str

    class Config:
        json_schema_extra = {
            "example": {"item": "example schema!"}
        }


# Response model
class TodoItems(BaseModel):
    tasks: List[TodoItem]

    class Config:
        json_schema_extra = {
            "example": {
                "tasks": [
                    {"item": "schema 1"},
                    {"item": "schema 2"},
                    {"item": "schema 3"},
                ]
            }
        }
```

todo.py

```
# add response model
@todo_router.get("/todo", response_model=TodoItems)
async def retrieve_tasks() -> dict:
    return {"tasks": todo_list}
```

Обработка ошибок

Класс HTTPException

Класс HTTPException принимает три аргумента:
• *status_code*: Код состояния, который будет возвращен для этого сбоя
• *detail:* Сопроводительное сообщение для отправки клиенту
• *headers:* Необязательный параметр для ответов, требующих заголовков

chapter 4

Шаблоны, понимание Jinja

*Шаблонирование — это процесс отображения данных, полученных от API, в различных форматах. Шаблоны действуют как компонент интерфейса в веб-приложениях. **Как в Django***.

Jinja — это механизм шаблонов, написанный на Python, предназначенный для облегчения процесса рендеринга ответов API. В каждом языке шаблонов есть переменные, которые заменяются фактическими значениями, переданными им при отображении шаблона, и есть теги, управляющие логикой шаблона.

Основные блоки синтаксиса:

1. Синтаксис {{ }} - блок переменных.
2. Синтаксис {% %} - блок конструкций.
3. Синтаксис {# #} - блок комментария.

Переменные шаблона Jinja могут относиться к любому типу или объекту Python, если их можно преобразовать в строки. Тип модели, списка или словаря можно передать шаблону и отобразить его атрибуты, поместив эти атрибуты во второй блок, указанный ранее.

Фильтры

*Несмотря на сходство синтаксиса Python и Jinja, такие модификации, как объединение строк, установка первого символа строки в верхний регистр и т. д., не могут быть выполнены с использованием синтаксиса Python в Jinja. **Поэтому для выполнения таких модификаций у нас в Jinja есть фильтры.***

Формат фильтра:

```
{{ variable | filter_name(*args) }}
```

Формат фильтра без аргумента:

```
{{ variable | filter_name }}
```

Фильтр по умолчанию

Фильтр, который возвращает заданную строку, если в входных данных нет переменной.

```
{{ todo.item | default("Default value for todo item.") }}
```

Эвакуационный фильтр

Используется для отображения необработанного вывода HTML

```
{{ "<h1>Item</h1>" | escape }}
{# In page will be: <h1>Item</h1> (header will not render) #}
```

Фильтр преобразования

Используется для преобразования между float и int

```
{{ 3.14 | int }}
{# 3 #}

{{ 2 | float }}
{# 2.0 #}
```

Фильтр объединения

Фильтр, преобразовывающий список в строку (как array.join("") в python).

```
{{ ["Hello", "my", "name", "Mike."] | join(" ") }}
{# Hello my name Mike. #}
```

Фильтр длинны

Фильтр, который возвращает длину объекта (как len в python).

```
{# todos = [1,2,3,4] #}
Todo count: {{ todos | length }}
{# Todo count: 4 #}
```

Использование оператора if

```
{% if todos | length < 5 %}
        You don`t have much items today!
{% else %}
        You have busy day!
{% endif %}
```

Использование циклов

```
{% for todo in todos %}
        <b>{{ todo.item }}</b>
{% endfor %}
```

Так-же можно получить доступ к специальным переменным цикла:

Variable	Description
loop.index	The current iteration of the loop (1 indexed)
loop.index0	The current iteration of the loop (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	The number of iterations from the end of the loop (0 indexed)
loop.first	True if first iteration

Variable	Description
`loop.last`	True if last iteration
`loop.length`	The number of items in the sequence
`loop.cycle`	A helper function to cycle between a list of sequences
`loop.depth`	Indicates how deep in a recursive loop the rendering currently is; starts at level 1
`loop.depth0`	Indicates how deep in a recursive loop the rendering currently is; starts at level 0
`loop.previtem`	The item from the previous iteration of the loop; undefined during the first iteration
`loop.nextitem`	The item from the following iteration of the loop; undefined during the last iteration
`loop.changed(*val)`	True if previously called with a different value (or not called at all)

Макросы

Макрос в Jinja — это функция, которая возвращает строку HTML. Основной вариант использования макросов — избежать повторения кода и вместо этого использовать один вызов функции. Например, макрос ввода определен для сокращения непрерывного определения тегов ввода в HTML-форме:

```
{% macro input(name, value='', type='text', size=20 %}
        <div class="form">
                <input type="{{ type }}" name="{{ name }}"
                        value="{{ value|escape }}" size="{{ size }}">
        </div>
{% endmacro %}
```

Теперь, чтобы быстро создать ввод в вашей форме, вызывается макрос:

```
{{ input('item') }}
```

Это вернет следующее:

```
<div class="form">
        <input type="text" name="item" value="" size="20">
</div>
```

Наследование шаблонов

Самая мощная функция Jinja — наследование шаблонов. Эта функция продвигает принцип «не повторяйся» (DRY) и удобна в больших веб-приложениях. Наследование шаблона — это ситуация, когда базовый шаблон определен, а дочерние шаблоны могут взаимодействовать, наследовать и заменять определенные разделы базового шаблона.

***Примечание**
Вы можете узнать больше о наследовании шаблонов Jinja на https://jinja.palletsprojects.com/en/3.0.x/template*

chapter 5

Структурирование в приложениях FastAPI

- planner/
  - main.py
  - database/
    - init.py
      connection.py
  - routes/
    - init.py
    - events.py
    - users.py
  - models/
    - init.py
    - events.py
    - users.py

chapter 6

Настройка SQLModel

```
pip install sqlmodel
```

Создание таблицы:

```
class Event(SQLModel, table=True):
        id: Optional[int] = Field(default=None,
        primary_key=True)
        title: str
        image: str
        description: str
        location: str
        tags: List[str]
```

Создание строки таблицы:

```
new_event = Event(title="Book Launch",
        image="src/fastapi.png",
        description="The book launch event will be held at Packt HQ, Packt
city",
        location="Google Meet",
        tags=["packt", "book"])
```

Создание транзакции:

```
with Session(engine) as session:
        session.add(new_event)
        session.commit()
```

Сессии

Сессии в этом случае отвечает за то, что таблицы и строки попадут в базу данных

- **add()**: Этот метод отвечает за добавление объекта базы данных в память в ожидании дальнейших операций. В предыдущем блоке кода объект new_event добавляется в память сеанса, ожидая фиксации в базе данных методом commit().
  • **commit()**: Этот метод отвечает за сброс транзакций, присутствующих в сеансе.
  • **get()**: Этот метод принимает два параметра — модель и идентификатор запрошенного документа. Этот метод используется для извлечения одной строки из базы данных.

Создание базы данных

В SQLModel подключение к базе данных осуществляется с помощью механизма SQLAlchemy. Движок создается методом create_engine(), импортированным из библиотеки SQLModel.

```
database_file = "database.db"
engine = create_engine(database_file, echo=True)
SQLModel.metadata.create_all(engine)
```

connection.py:

```
from sqlmodel import SQLModel, Session, create_engine
# from models.events import Event

database_file = "database/planner.db"
database_connection_string = f"sqlite:///{database_file}"
connect_args = {"check_same_thread": False}
engine_url = create_engine(database_connection_string,
                           echo=True, connect_args=connect_args)


def conn():
    SQLModel.metadata.create_all(engine_url)


def get_session():
    with Session(engine_url) as session:
        yield session
```

routes/events.py:

```

@event_router.get("/", response_model=EventsResponse)
async def retrieve_all_events(session=Depends(get_session)) -> EventsResponse:
    statement = select(Event)
    events = session.exec(statement).all()
    return events


@event_router.get("/{id}", response_model=Event)
async def retrieve_single_event(id: int, session=Depends(get_session)) -> Event:
    event = session.get(Event, id)
    if event:
        return event
```

```
    else:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Event with id {id} not found."
        )


@event_router.post("/new")
async def create_event(new_event: Event = Body(...),
                       session=Depends(get_session)) -> dict:
    session.add(new_event)
    session.commit()
    session.refresh(new_event)
    return {"message": "Event created successfully."}


@event_router.delete("/{id}")
async def delete_event(id: int, session=Depends(get_session)) -> dict:
    event = session.get(Event, id)
    if event:
        session.delete(event)
        session.commit()
        return {"message": "Event deleted successfully."}
    else:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Event with id {id} not found."
        )


@event_router.delete("/")
async def delete_all_events(session=Depends(get_session)) -> dict:
    statement = select(Event)
    events = session.exec(statement).all()
    for event_row in events:
        event_dict = dict(event_row)
        event = session.get(Event, event_dict["Event"].id)
        session.delete(event)

    session.commit()

    return {"message": "Events delete successfully."}


@event_router.put("/edit/{id}", response_model=Event)
async def update_event(id: int, new_event: EventUpdate,
```

```
                      session=Depends(get_session)) -> Event:
    event = session.get(Event, id)
    if event:
        event_data = new_event.dict(exclude_unset=True)
        for key, value in event_data.items():
            setattr(event, key, value)
        session.add(event)
        session.commit()
        session.refresh(event)

        return event
    else:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Event with id {id} not found."
        )
```

MongoDB

```
pip install beanie
```

MongoDB хранит данные в базе с помощью документов (NoSQL).
В SQL данные, хранящиеся в строках и столбцах, содержатся в таблице. В базе данных NoSQL это называется документом. Документ представляет, как данные будут храниться в коллекции базы данных. Документы определяются так же, как и модель Pydantic, за исключением того, что вместо этого наследуется класс Document из библиотеки Beanie.
Создание документа:

```
from beanie import Document

class Event(Document):
    name: str
    location: str

    class Settings:
        name = "events"
```

Методы для работы с CRUD

- create(), insert(), insert_one(), insert_many() - для создание записей в документе.

```
data = {...}
event = Event(**data)
await event.create() # for object of model
await Event.insert_one(event) # for model
```

- find(), find_one(), get() - поиск записей в базе данных по заданным данным (find - критерии, get - id).

```
uuid = "2468129468901fgfa"
event = await Event.get(uuid) # return 1
event = await Event.find(Event.location=="Google").to_list() # return n
event = await Event.find_one(Event.location=="Google") # return 1
```

- save(), update(), upsert() - для обновления записи.

```
event = await Event.get("342525")
update_query = {"$set": {"location": "Hybrid"}}
await event.update(update_query)
```

- delete() - удаление записи.

```
event = await Event.get("5213451")
await event.delete()
```

Инициализация

```
from beanie import init_beanie
from motor.motor_asyncio import AsyncIOMotorClient
from typing import Optional
from pydantic import BaseSettings

from models.users import User
from models.events import Event


class Settings(BaseSettings):
    DATABASE_URL = Optional[str] = None
```

```
    async def initialize_database(self):
        client = AsyncIOMotorClient(self.DATABASE_URL)
        await init_beanie(
            database=client.get_default_database(),
            document_models=[User, Event])

    class Config:
        env_file = ".env"
```

Создание CRUD операций для базы

```
class Database:
    def __init__(self, model):
        self.model = model

    async def save(self, document) -> None:
        await document.create()

    async def get(self, id: PydanticObjectId) -> Any:
        doc = await self.model.get(id)
        if doc:
            return doc
        else:
            return False

    async def get_all(self) -> List[Any]:
        docs = await self.model.find_all().to_list()
        return docs

    async def update(self, id: PydanticObjectId, body: BaseModel) -> Any:
        doc = await self.get(id)
        if not doc:
            return False

        des_body = body.dict()
        des_body = {key: value for key,
                    value in des_body.items() if value is not None}

        update_query = {"$set": {
            field: value for field, value in des_body.items()
        }}

        doc.update(update_query)
```

```
        return doc

    async def delete(self, id: PydanticObjectId) -> bool:
        doc = await self.get(id)
        if not doc:
            return False
        await doc.delete()
        return True
```

Запуск

```
(venv)$ mkdir store
(venv)$ mongod --dbpath store
```

chapter 7

Методы аутентификации в FastAPI

• **Базовая HTTP-аутентификация:** В этом методе аутентификации учетные данные пользователя, которые обычно представляют собой имя пользователя и пароль, отправляются через HTTP-заголовок авторизации. Запрос, в свою очередь, возвращает заголовок WWW-Authenticate содержащий, базовое значение и необязательный параметр области, который указывает ресурс, к которому выполняется запрос аутентификации.
• **Cookies:** Файлы cookie используются, когда данные должны храниться на стороне клиента, например, в веб-браузерах. Приложения FastAPI также могут использовать файлы cookie для хранения пользовательских данных, которые могут быть получены сервером в целях аутентификации.
• **Аутентификацию на основе токенов:** Этот метод аутентификации включает использование токенов безопасности, называемых токенами носителя. Эти токены отправляются вместе с ключевым словом Bearer в запросе заголовка авторизации. Наиболее часто используемым токеном является JWT, который обычно представляет собой словарь, содержащий идентификатор пользователя и срок действия токена.

Зависимости (Depends)

Зависимости - это те функции или классы, которые должны выполнятся перед конечной точкой запроса, если через запрос не проходит условия зависимости, то

сервер вернет ошибку.

Пример:

```
from fastapi import ..., Depends

# Create Depends
async def get_user(token: str):
    user = decode_token(token)
    return user


# Use depends in fn arg (user). If we cant decode token, endpoint will return
error.
@user_router.get("/user/me")
async def get_user_detail(user: User = Depends(get_user)):
    return user
```

Хэширование паролей

```
pip install passlib
```

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")


class HashPassowrd:
    def create_hash(self, password: str):
        return pwd_context.hash(password)

    def verify_hash(self, password: str, hash: str):
        return pwd_context.verify(password, hash)
```

Создание JWT токенов

```
pip install wheel jose crypto social-auth-core
```

```
# FastAPI, jwt libs
from fastapi import HTTPException, status
```

```python
from jose import JWTError, jwt

# settings
from database.connection import Settings

# time
import time
from datetime import datetime


settings = Settings()


def create_access_token(user: str) -> str:
    payload = {
        "user": user,
        "expires": time.time() + 3600*3
    }
    token = jwt.encode(payload, settings.SECRET_KEY, algorithm="HS256")
    return token


def verify_access_token(token: str) -> dict:
    try:
        data = jwt.decode(token, settings.SECRET_KEY, algorithms=["HS256"])
        expire = data.expires

        if expire is None:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="No access token suplied."
            )
        if datetime.utcnow() > datetime.utcfromtimestamp(expire):
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Token expired!"
            )

        return data

    except JWTError:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Invalid token."
        )
```

Проверка аутентификации

```python
# FastAPI
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer

# jwt handler
from auth.jwt_handler import verify_access_token


oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/user/signin")


async def authenticate(token: str = Depends(oauth2_scheme)) -> str:
    if not token:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Sign in for access."
        )

    decoded_token = verify_access_token(token)
    return decoded_token["user"]
```

views.py

```python
# Depends to auth
@event_router.put("/edit/{id}", response_model=Event)
async def update_event(id: PydanticObjectId, new_event: EventUpdate,
                       user: str = Depends(authenticate)) -> Event:
        ...
```

Настройка CORS

FastAPI предоставляет CORS middleware, CORSMiddleware, которое позволяет нам регистрировать домены, которые могут получить доступ к нашему API. Middleware принимает массив источников, которым будет разрешен доступ к ресурсам на сервере.

Что такое middleware?

Middleware — это функция, выступающая посредником

между операциями. В веб-API middleware служит посредником в операции запрос-ответ.

```
from fastapi.middleware.cors import CORSMiddleware

...

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

chapter 8

Pytest

```
pip install pytest
```

Создание простых тестов:

```
class Arichmetic:
    def add(self, a: int, b: int) -> int:
        return a + b

    def subtract(self, a: int, b: int) -> int:
        return b - a

        ...

class TestsArithmetic:
    Ar = Arichmetic()

    def test_add(self) -> None:
        assert self.Ar.add(1, 1) == 2

    def test_subtract(self) -> None:
        assert self.Ar.subtract(2, 5) == 3
```

```
        ...
```

```
pytest test_module.py
# 2 passed tests.
```

Фикстуры

Фикстуры - это повторно используемые функции, которые определяют возвратные данные для тестов.

Пример:

```
import pytest

from models.events import EventUpdate


@pytest.fixture
def event() -> EventUpdate:
    return EventUpdate(
        title="FastAPI Book Launch",
        image="https://packt.com/fastapi.png",
        description="We will be discussing the contents of \
        the FastAPI book in this event.Ensure to come with \
        your own copy to win gifts!",
        tags=["python", "fastapi", "book", "launch"],
        location="Google Meet"
    )


def test_event_name(event: EventUpdate) -> None:
    assert event.title == "FastAPI Book Launch"
```

Подготовка тестовой среды

confest.py

```
# test libs
import asyncio
import httpx
```

```python
import pytest

# App modules
from main import app
from database.connection import Settings
from models.events import Event
from models.users import User


@pytest.fixture
def event_loop():
    loop = asyncio.get_event_loop()
    yield loop
    loop.close()


async def init_db():
    test_settings = Settings()
    test_settings.DATABASE_URL = "mongodb://localhost:27017/testdb"
    await test_settings.initialize_database()


@pytest.fixture(scope="session")
async def default_client():
    await init_db()
    async with httpx.AsyncClient(app=app, base_url="http://app") as client:
        yield client
        # Clean db
        await Event.find_all().delete()
        await User.find_all().delete()
```

Примеры тестов

Тесты конечных точек для входа в систему:

```python
import httpx
import pytest


@pytest.mark.asyncio
async def test_sign_new_user(default_client: httpx.AsyncClient) -> None:
    headers = {
        "accept": "application/json",
        "Content-Type": "application/json"
```

```
    }
    payload = {
        "email": "testuser@packt.com",
        "username": "testusername",
        "password": "testpassword"
    }

    test_response = {
        "message": "User successfully registered!"
    }

    response = await default_client.post("user/signup", json=payload,
headers=headers)
    assert response.status_code == 200
    assert response.json() == test_response

...
```

Тесты конечных точек для получения данных об событиях:

```
@pytest.fixture(scope="module")
async def access_token() -> str:
    return create_access_token("testuser@packt.com")


@pytest.fixture(scope="module")
async def mock_event() -> Event:
    data = {
        "creator": "testuser@packt.com",
        "title": "FastAPI book launch test",
        "image_url": "https://site.com/image.png",
        "description": "content about fastapi",
        "tags": ["python", "web", "rest"],
        "location": "Google meet"
    }
    new_event = Event(**data)
    await Event.insert_one(new_event)
    yield new_event


@pytest.mark.asyncio
async def test_get_events(default_client: client, mock_event: Event) -> None:
    response = await default_client.get("/event/")
    response = response
```

```
    assert response.status_code == 200
    assert response.json()[0]["_id"] == str(mock_event.id)
```

Покрытие тестами

Отчет о покрытии тестами полезен для определения процента нашего кода, который был выполнен в ходе тестирования. Давайте установим модуль coverage, чтобы мы могли измерить, был ли наш API адекватно протестирован:

```
pip install coverage
coverage run -m pytest
coverage report
coverage html # create html page with report
```

chapter 9

Развертывание приложения

Создание докерфайла и инструкций

DOCKERFILE:

```
FROM python:3.10

# set workdir
WORKDIR /planner_app

# install requirements
COPY requirements.txt /planner_app
RUN pip install --upgrade pip && pip install -r /planner_app/requirements.txt

# Give open port
EXPOSE 8000

# add project to workdir
ADD ./ /planner_app

# run app
CMD ["python", "main.py"]
```

.dockerignore:

```
venv
.env
.git
```

Создание docker образов

```
docker build -t event-planner-api .
docker pull mongo
```

*Команда **docker pull** отвечает за загрузку образов из реестра. Если не указано иное, эти образы загружаются из общедоступного реестра Docker Hub.*

Локальное развертывание приложения

```
touch docker-compose.yml # create manifest
```

docker-compose.yml:

```
version: "3"

services:
  api:
    build: .
    image: event-planner-api:latest
    ports:
      - "8000:8000"
    env_file:
      - .env.prod

  database:
    image: mongo
    ports:
      - "27017"
    volumes:
      - data:/data/db
```

```
volumes:
  data:
```

.env.prod:

```
DATABASE_URL=mongodb://database:27017/planner
SECRET_KEY=HI5HL3V3L$3CR3T
```

Запуск приложения

```
docker-compose up -d # start containers
docker ps # check containers
docker-compose down # stop and remove containers
```