

# FastApi

- [Общие команды](#)
- [Маршрутизация](#)
- [Pydantic](#)
- [Jinja2](#)
- [Авторизация и аутентификация](#)

# Общие команды

## Установка

```
pip install fastapi uvicorn
```

## Ручной запуск (api - имя файла, app - имя объекта FastApi)

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

## Запуск uvicorn из python скрипта

Файл main.py

```
from uvicorn import run
...
app = FastAPI()
...
if __name__ == '__main__':
    run(app="main:app", host='0.0.0.0', port=8000, workers=4, log_level='warning')
    #run(app="main:app", host='0.0.0.0', port=8000, reload=True)
```

## curl запросы

```
curl -X 'GET' 'http://127.0.0.1:8000/todo' -H 'accept: application/json'
```

```
curl -X 'POST' \
'http://127.0.0.1:8000/todo' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "id": 1,
  "item": "First Todo is to finish this book!"
}'
```

**Автоматическая документация**

## Swagger

`http://ip:port/docs`

## Redoc

`http://ip:port/redoc`

## Шаблоны Jinja

Поддерживает шаблоны Jinja при выводе данных (вплоть до циклов).

# Маршрутизация

## Добавление маршрутов

Основной файл:

```
from fastapi import FastAPI
from todo import todo_router

app = FastAPI()

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

app.include_router(todo_router)
```

Файл дополнительных маршрутов

```
from fastapi import APIRouter

todo_router = APIRouter()

todo_list = []

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

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

Автоматическое добавление маршрутов в основной файл *app* из файлов в директории *data/plugins* имеющих шаблон имени объекта *APIRouter* *modulename\_router*

```

fpath = os.path.join('data', 'plugins')
flist = os.listdir(fpath)
sys.path.insert(0, fpath)
for fname in flist:
    if fname not in ['__pycache__', '__init__.py']:
        m = os.path.splitext(fname)[0]
        impmod = importlib.import_module(m)
        router_name = f'{m}_router'
        if router_name in dir(impmod):
            router_mod = getattr(impmod, router_name)
            app.include_router(router_mod)

```

## Получаемые параметры

Параметры пути:

```

from fastapi import Path

@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The ID of the todo to retrieve. ")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return {"todo": todo}
    return {
        "message": "Todo with supplied ID doesn't exist."
    }

```

В Path ... - параметр пути обязательный, None - не обязательный

Параметры запроса (после ? в запросе):

```

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

```

## Передаваемые параметры

При помощи response\_model можно фильтровать отдаваемые данные. Т е можно в отдаваемой модели указать неполный набор.

```

from typing import List

class TodoItem(BaseModel):

```

```
item: str
```

```
class TodoItems(BaseModel):  
    todos: List[TodoItem]
```

```
@todo_router.get("/todo", response_model=TodoItems)  
async def retrieve_todo() -> dict:  
    return {  
        "todos": todo_list  
    }
```

У содержащихся в списке словарей будет оставлен только item

## Исключения

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

```
from fastapi import APIRouter, Path, HTTPException, status  
  
@todo_router.get("/todo/{todo_id}")  
async def get_single_todo(todo_id: int = Path(..., title="The ID of the todo to retrieve. ")) -> dict:  
    for todo in todo_list:  
        if todo.id == todo_id:  
            return { "todo": todo }  
    raise HTTPException(  
        status_code=status.HTTP_404_NOT_FOUND,  
        detail="Todo with supplied ID doesn't exist",  
    )
```

Можно переопределить код успешного возврата в декораторе

```
@todo_router.post("/todo", status_code=201)  
async def add_todo(todo: Todo) -> dict:  
    todo_list.append(todo)  
    return { "message": "Todo added successfully." }
```



# Pydantic

```
class Item(BaseModel):
```

```
    item: str
```

```
    status: str
```

```
class Todo(BaseModel):
```

```
    id: int
```

```
    item: Item
```

# Jinja2

## Формат Jinja2

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.

### Виды синтаксиса

- {% ... %} управляющие структуры
- {{ todo.item }} вывод значений переданных ему выражений
- {# This is a great API book! #} комментарии

### Фильтры

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

## Виды фильтров

Название	Описание
default(strdefault)	Замена вывода переданного значения, если оно оказывается None <div><pre>{{ todo.item   default('This is a default todo item') }}</pre></div>
escape	Отображение необработанного вывода HTML
striptags	Удаление HTML тегов перед отправкой
int float	Преобразование типов перед ответом <div><pre>{{ 3.142   int }}</pre><pre>3</pre><pre>{{ 31   float }}</pre><pre>31.0</pre></div>
join(whitespace)	Объединение элементов списка в строку <div><pre>{{ ['Packt', 'produces', 'great', 'books!']   join(' ') }}</pre><pre>Packt produces great books!</pre></div>

Название	Описание
length	Длина переданного объекта <div>Todo count: {{ todos   length }}</div> <div>Todo count: 4</div>

[Полный список фильтров](#)

Условия:

```
{% if user %}
    Hello, {{ user.name }}!
{% else %}
    Hello, Unknown!
{% endif %}
```

Циклы

```
{% for comment in comments %}
    <b>{{ comment }}</b>
{% endfor %}
```

Можно также обратиться к переменной loop.index для получения дополнительной информации

Переменная	Описание
loop.index	Текущее значение итерации (1 - первая итерация)
loop.index0	Текущее значение итерации (0 - первая итерация)
loop.revindex loop.revindex0	Кол-во оставшихся итераций
loop.first	True если первая итерация
loop.last	
loop.length	
loop.pervitem loop.nextitem	Значение предыдущей/следующей итерации (пусто если не существует)

Макросы:

```
{% macro render_comment(comment) %}
    <li>{{ comment }}</li>
{% endmacro %}
<ul>
    {% for comment in comments %}
        {{ render_comment(comment) }}
    {% endfor %}
```

</ul>

Макросы можно импортировать из файлов

```
{% import 'macros.html' as macros %}
```

В Jinja двойные фигурные скобки `{{ }}` позволяют получить результат выражение, переменную или вызвать функцию и вывести значение в шаблоне.

class Foo:

```
def __str__(self):
```

```
    return "This is an instance of Foo class"
```

```
Template("{{ var }}").render(var=Foo())
```

```
'This is an instance of Foo class'
```

Если обратится к индексу, который не существует, Jinja просто выведет пустую строку.

Вызов функции

В Jinja для определения функции ее нужно просто вызвать.

```
def foo():
```

```
    return "foo() called"
```

```
Template("{{ foo() }}").render(foo=foo)
```

```
'foo() called'
```

Объявление переменных

Внутри шаблона можно задать переменную с помощью инструкции `set`.

```
{% set fruit = 'apple' %}
```

```
{% set name, age = 'Tom', 20 %}
```

Переменные определяются для хранения результатов сложных операций, так чтобы их можно было использовать дальше в шаблоне. Переменные, определенные вне управляющих конструкций (о них дальше), ведут себя как глобальные переменные и доступны внутри любой структуры. Тем не менее переменные, созданные внутри конструкций, ведут себя как локальные переменные и видимы только внутри этих конкретных конструкций.

Единственное исключение — инструкция `if`.

Цикл и условные выражения

Реклама

Управляющие конструкции позволяют добавлять в шаблоны элементы управления потоком и циклы. По умолчанию, управляющие конструкции используют разделитель `{% ... %}` вместо двойных фигурных скобок `{{ ... }}`.

Инструкция `if`

Инструкция `if` в Jinja имитирует выражение `if` в Python, а значение условия определяет набор инструкции. Например:

```
{% if bookmarks %}
```

```
<p>User has some bookmarks</p>
{% endif %}
```

Если значение переменной `bookmarks` – `True`, тогда будет выведена строка `<p>User has some bookmarks</p>`. Стоит запомнить, что в Jinja, если у переменной нет значения, она возвращает `False`.

Также можно использовать условия `elif` и `else`, как в обычном коде Python. Например:

```
{% if user.newbie %}
    <p>Display newbie stages</p>
{% elif user.pro %}
    <p>Display pro stages</p>
{% elif user.ninja %}
    <p>Display ninja stages</p>
{% else %}
    <p>You have completed all stages</p>
{% endif %}
```

Управляющие инструкции также могут быть вложенными. Например:

```
{% if user %}
    {% if user.newbie %}
        <p>Display newbie stages</p>
    {% elif user.pro %}
        <p>Display pro stages</p>
    {% elif user.ninja %}
        <p>Display ninja stages</p>
    {% else %}
        <p>You have completed all states</p>
    {% endif %}
{% else %}
    <p>User is not defined</p>
{% endif %}
```

## Реклама

В определенных случаях достаточно удобно записывать инструкцию `if` в одну строку. Jinja поддерживает такой тип записи, но называет это выражением `if`, потому что оно записывается с помощью двойных фигурных скобок `{{ ... }}`, а не `{% ... %}`. Например:

```
{{ "User is logged in" if loggedin else "User is not logged in" }}
```

Здесь если переменная `loggedin` вернет `True`, тогда будет выведена строка `"User is logged in"`. В противном случае — `"User is not logged in"`.

Условие else использовать необязательно. Если его нет, тогда блок else вернет объект undefined.

```
{{ "User is logged in" if loggedin }}
```

Здесь, если переменная loggedin вернет True, будет выведена строка “User is logged in”. В противном случае — ничего.

Как и в Python можно использовать операторы сравнения, присваивания и логические операторы для управляющих конструкций, чтобы создавать более сложные условия. Вот несколько примеров:

```
{# Если user.count равен 1000, код '<p>User count is 1000</p>' отобразится #}  
{% if users.count == 1000 %}  
    <p>User count is 1000</p>  
{% endif %}
```

```
{# Если выражение 10 >= 2 верно, код '<p>10 >= 2</p>' отобразится #}  
{% if 10 >= 2 %}  
    <p>10 >= 2</p>  
{% endif %}
```

```
{# Если выражение "car" <= "train" верно, код '<p>car <= train</p>' отобразится #}  
{% if "car" <= "train" %}  
    <p>car <= train</p>  
{% endif %}
```

```
{#  
    Если user залогинен и superuser, код  
    '<p>User is logged in and is a superuser</p>' отобразится  
#}  
{% if user.loggedin and user.is_superuser %}  
    <p>User is logged in and is a superuser</p>  
{% endif %}
```

```
{#  
    Если user является superuser, moderator или author, код  
    '<a href="#">Edit</a>' отобразится  
#}  
{% if user.is_superuser or user.is_moderator or user.is_author %}  
    <a href="#">Edit</a>  
{% endif %}
```

```
{#  
    Если user и current_user один и тот же объект, код  
    <p>user and current_user are same</p> отобразится  
#}
```

```
{% if user is current_user %}
    <p>user and current_user are same</p>
{% endif %}

{#
    Если "Flask" есть в списке, код
    '<p>Flask is in the dictionary</p>' отобразится
#}
{% if ["Flask"] in ["Django", "web2py", "Flask"] %}
    <p>Flask is in the dictionary</p>
{% endif %}
```

Если условия становятся слишком сложными, или просто есть желание поменять приоритет оператора, можно обернуть выражения скобками ():

```
{% if (user.marks > 80) and (user.marks < 90) %}
    <p>You grade is B</p>
{% endif %}
```

Цикл for

Цикл for позволяет перебирать последовательность. Например:

```
{% set user_list = ['tom', 'jerry', 'spike'] %}

<ul>
{% for user in user_list %}
    <li>{{ user }}</li>
{% endfor %}
</ul>
```

Вывод:

```
<ul>

    <li>tom</li>

    <li>jerry</li>

    <li>spike</li>

</ul>
```

Вот как можно перебирать значения словаря:

```
{% set employee = { 'name': 'tom', 'age': 25, 'designation': 'Manager' } %}
```

```
<ul>
{% for key in employee.items() %}
<li>{{ key }} : {{ employee[key] }}</li>
{% endfor %}
</ul>
```

Вывод:

```
<ul>

<li>designation : Manager</li>

<li>name : tom</li>

<li>age : 25</li>

</ul>
```

Примечание: в Python элементы словаря не хранятся в конкретном порядке, поэтому вывод может отличаться.

Если нужно получить ключ и значение словаря вместе, используйте метод items().

```
{% set employee = { 'name': 'tom', 'age': 25, 'designation': 'Manager' } %}
```

```
<ul>
{% for key, value in employee.items() %}
<li>{{ key }} : {{ value }}</li>
{% endfor %}
</ul>
```

Вывод:

```
<ul>

<li>designation : Manager</li>

<li>name : tom</li>

<li>age : 25</li>

</ul>
```

Цикл for также может использовать дополнительное условие else, как в Python, но зачастую способ его применения отличается. Стоит вспомнить, что в Python, если else идет следом за циклом for, условие else выполняется только в том случае, если цикл завершается после перебора всей последовательности, или если она пуста. Оно не выполняется, если цикл

остановить оператором break.

Когда условие else используется в цикле for в Jinja, оно выполняется только в том случае, если последовательность пустая или не определена. Например:

Реклама

```
{% set user_list = [] %}
```

```
<ul>
```

```
{% for user in user_list %}
```

```
  <li>{{ user }}</li>
```

```
{% else %}
```

```
  <li>user_list is empty</li>
```

```
{% endfor %}
```

```
</ul>
```

Вывод:

```
<ul>
```

```
  <li>user_list is empty</li>
```

```
</ul>
```

По аналогии с вложенными инструкциями if, можно использовать вложенные циклы for. На самом деле, любые управляющие конструкции можно вкладывать одна в другую.

```
{% for user in user_list %}
```

```
  <p>{{ user.full_name }}</p>
```

```
  <p>
```

```
    <ul class="follower-list">
```

```
      {% for follower in user.followers %}
```

```
        <li>{{ follower }}</li>
```

```
      {% endfor %}
```

```
    </ul>
```

```
  </p>
```

```
{% endfor %}
```

Цикл for предоставляет специальную переменную loop для отслеживания прогресса цикла. Например:

```
<ul>
```

```
{% for user in user_list %}
```

```
  <li>{{ loop.index }} - {{ user }}</li>
```

```
{% endfor %}
```

```
</ul>
```

loop.index внутри цикла for начинает отсчет с 1. В таблице упомянуты остальные широко используемые атрибуты переменной loop.

Метод Значение

loop.index0 то же самое что и loop.index, но с индексом 0, то есть, начинает считать с 0, а не с 1.

loop.revindex возвращает номер итерации с конца цикла (считает с 1).

loop.revindex0 возвращает номер итерации с конца цикла (считает с 0).

loop.first возвращает True, если итерация первая. В противном случае — False.

loop.last возвращает True, если итерация последняя. В противном случае — False.

loop.length возвращает длину цикла(количество итераций).

Примечание: полный список есть в документации Flask.

Фильтры

Фильтры изменяют переменные до процесса рендеринга. Синтаксис использования фильтров следующий:

```
variable_or_value|filter_name
```

Вот пример:

```
{{ comment|title }}
```

Фильтр title делает заглавной первую букву в каждом слове. Если значение переменной comment — "dust in the wind", то вывод будет "Dust In The Wind".

Можно использовать несколько фильтров, чтобы точнее настраивать вывод. Например:

```
{{ full_name|striptags|title }}
```

Фильтр striptags удалит из переменной все HTML-теги. В приведенном выше коде сначала будет применен фильтр striptags, а затем — title.

У некоторых фильтров есть аргументы. Чтобы передать их фильтру, нужно вызвать фильтр как функцию. Например:

```
{{ number|round(2) }}
```

Фильтр round округляет число до конкретного количества символов.

В следующей таблице указаны широко используемые фильтры.

Название	Описание
upper	делает все символы заглавными
lower	приводит все символы к нижнему регистру
capitalize	делает заглавной первую букву и приводит остальные к нижнему регистру
escape	экранирует значение
safe	предотвращает экранирование

length возвращает количество элементов в последовательности

trim удаляет пустые символы в начале и в конце

random возвращает случайный элемент последовательности

Примечание: полный список фильтров доступен [здесь](#).

Макросы

Макросы в Jinja напоминают функции в Python. Суть в том, чтобы сделать код, который можно использовать повторно, просто присвоив ему название. Например:

```
{% macro render_posts(post_list, sep=False) %}
    <div>
        {% for post in post_list %}
            <h2>{{ post.title }}</h2>
            <article>
                {{ post.html|safe }}
            </article>
        {% endfor %}
        {% if sep %}<hr>{% endif %}
    </div>
{% endmacro %}
```

В этом примере создан макрос render\_posts, который принимает обязательный аргумент post\_list и необязательный аргумент sep. Использовать его нужно следующим образом:

```
{{ render_posts(posts) }}
```

Определение макроса должно идти до первого вызова, иначе выйдет ошибка.

Реклама

Вместо того чтобы использовать макросы прямо в шаблоне, лучше хранить их в отдельном файле и импортировать по надобности.

Предположим, все макросы хранятся в файле macros.html в папке templates. Чтобы импортировать их из файла, нужно использовать инструкцию import:

```
{% import "macros.html" as macros %}
```

Теперь можно ссылаться на макросы в файле macros.html с помощью переменной macros. Например:

```
{{ macros.render_posts(posts) }}
```

Инструкция {% import "macros.html" as macros %} импортирует все макросы и переменные (определенные на высшем уровне) из файла macros.html в шаблон. Также можно импортировать определенные макросы с помощью from:

```
{% from "macros.html" import render_posts %}
```

При использовании макросов будут ситуации, когда потребуется передать им произвольное число аргументов.

По аналогии с `*args` и `**kwargs` в Python внутри макросов можно получить доступ к `varargs` и `kwargs`.

`varargs`: сохраняет дополнительные позиционные аргументы, переданные макросу, в виде кортежа.

`kwargs`: сохраняет дополнительные позиционные аргументы, переданные макросу, в виде словаря.

Хотя к ним можно получить доступ внутри макроса, объявлять их отдельно в заголовке макроса не нужно. Вот пример:

```
{% macro custom_renderer(para) %}  
    <p>{{ para }}</p>  
    <p>varargs: {{ varargs }}</p>  
    <p>kwargs: {{ kwargs }}</p>  
{% endmacro %}
```

```
{{ custom_renderer("some content", "apple", name='spike', age=15) }}
```

В этом случае дополнительный позиционный аргумент, "apple", присваивается `varargs`, а дополнительные аргументы-ключевые слова (`name='spike'`, `age=15`) — `kwargs`.

Экранирование

Jinja по умолчанию автоматически экранирует вывод переменной в целях безопасности. Поэтому если переменная содержит, например, такой HTML-код: "`<p>Escaping in Jinja</p>`", он отрендерится в виде "`&lt;p&gt;Escaping in Jinja&lt;/p&gt;`". Благодаря этому HTML-коды будут отображаться в браузере, а не интерпретироваться. Если есть уверенность, что данные безопасны и их точно можно рендерить, стоит воспользоваться фильтром `safe`. Например:

```
{% set html = "<p>Escaping in Jinja</p>" %}  
{{ html|safe }}
```

Вывод:

```
<p>Escaping in Jinja</p>
```

Использовать фильтр `safe` в большом блоке кода будет неудобно, поэтому в Jinja есть оператор `autoescape`, который используется, чтобы отключить экранирование для большого объема данных. Он может принимать аргументы `true` или `false` для включения и отключения экранирования, соответственно. Например:

```
{% autoescape true %}
    Escaping enabled
{% endautoescape %}
```

```
{% autoescape false %}
    Escaping disabled
{% endautoescape %}
```

Все между `{% autoescape false %}` и `{% endautoescape %}` отрендерится без экранирования символов. Если нужно экранировать отдельные символы при выключенном экранировании, стоит использовать фильтр `escape`. Например:

```
{% autoescape false %}
<div class="post">
    {% for post in post_list %}
        <h2>{{ post.title }}</h2>
        <article>
            {{ post.html }}
        </article>
    {% endfor %}
</div>
<div>
    {% for comment in comment_list %}
        <p>{{ comment|escape }}</p> # escaping is on for comments
    {% endfor %}
</div>
{% endautoescape %}
```

## Вложенные шаблоны

Инструкция `include` рендерит шаблон внутри другого шаблона. Она широко используется, чтобы рендерить статический раздел, который повторяется в разных местах сайта. Вот синтаксис `include`:

Предположим, что навигационное меню хранится в файле `nav.html`, сохраненном в папке `templates`:

```
<nav>
    <a href="/home">Home</a>
    <a href="/blog">Blog</a>
    <a href="/contact">Contact</a>
</nav>
```

Чтобы добавить это меню в `home.html`, нужно использовать следующий код:

```
<!DOCTYPE html>
```

```

<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>

  { # добавляем панель навигации из nav.html # }
  {% include 'nav.html' %}

</body>
</html>

```

Вывод:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>

<nav>
  <a href="/home">Home</a>
  <a href="/blog">Blog</a>
  <a href="/contact">Contact</a>
</nav>

</body>
</html>

```

## Наследование шаблонов

Наследование шаблонов — один из самых мощных элементов шаблонизатора Jinja. Его принцип похож на ООП (объектно-ориентированное программирование). Все начинается с создания базового шаблона, который содержит в себе скелет HTML и отдельные маркеры, которые дочерние шаблоны смогут переопределять. Маркеры создаются с помощью инструкции `block`. Дочерние шаблоны используют инструкцию `extends` для наследования или расширения основного шаблона. Вот пример:

```

{ # Это шаблон templates/base.html # }
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>{% block title %}Default Title{% endblock %}</title>

```

```

</head>
<body>

    {% block nav %}
        <ul>
            <li><a href="/home">Home</a></li>
            <li><a href="/api">API</a></li>
        </ul>
    {% endblock %}

    {% block content %}

    {% endblock %}
</body>
</html>

```

Это базовый шаблон `base.html`. Он создает три блока с помощью `block`, которые впоследствии будут заполнены дочерними шаблонами. Инструкция `block` принимает один аргумент — название блока. Внутри шаблона это название должно быть уникальным, иначе возникнет ошибка.

Дочерний шаблон — это шаблон, который растягивает базовый шаблон. Он может добавлять, перезаписывать или оставлять элементы родительского блока. Вот как можно создать дочерний шаблон.

```

{# Это шаблон templates/child.html #}
{% extends 'base.html' %}

{% block content %}
    {% for bookmark in bookmarks %}
        <p>{{ bookmark.title }}</p>
    {% endfor %}
{% endblock %}

```

Инструкция `extends` сообщает Jinja, что `child.html` — это дочерний элемент, наследник `base.html`. Когда Jinja обнаруживает инструкцию `extends`, он загружает базовый шаблон, то есть `base.html`, а затем заменяет блоки контента внутри родительского шаблона блоками с теми же именами из дочерних шаблонов. Если блок с соответствующим названием не найден, используется блок родительского шаблона.

Стоит отметить, что в дочернем шаблоне перезаписывается только блок `content`, так что содержимое по умолчанию из `title` и `nav` будет использоваться при рендеринге дочернего шаблона. Вывод должен выглядеть следующим образом:

Реклама

```
<!DOCTYPE html>
```

```

<head>
  <meta charset="UTF-8">
  <title>Default Title</title>
</head>
<body>

  <ul>
    <li><a href="/home">Home</a></li>
    <li><a href="/api">API</a></li>
  </ul>

  <p>Bookmark title 1</p>
  <p>Bookmark title 2</p>
  <p>Bookmark title 3</p>
  <p>Bookmark title 4</p>

</body>
</html>

```

Если нужно, можно поменять заголовок по умолчанию, переписав блок title в child.html:

```

{# Это шаблон templates/child.html #}
{% extends 'base.html' %}

{% block title %}
  Child Title
{% endblock %}

{% block content %}
  {% for bookmark in bookmarks %}
    <p>{{ bookmark.title }}</p>
  {% endfor %}
{% endblock %}

```

После перезаписи блока на контент из родительского шаблона все еще можно ссылаться с помощью функции `super()`. Обычно она используется, когда в дополнение к контенту дочернего шаблона нужно добавить содержимое из родительского. Например:

```

{# Это шаблон templates/child.html #}
{% extends 'base.html' %}

{% block title %}
  Child Title
{% endblock %}

{% block nav %}

```

```
{{ super() }} {# referring to the content in the parent templates #}  
<li><a href="/contact">Contact</a></li>  
<li><a href="/career">Career</a></li>  
{% endblock %}
```

```
{% block content %}  
  {% for bookmark in bookmarks %}  
    <p>{{ bookmark.title }}</p>  
  {% endfor %}  
{% endblock %}
```

Вывод:

```
<!DOCTYPE html>  
<head>  
  <meta charset="UTF-8">  
  <title>Child Title</title>  
</head>  
<body>  
  
  <ul>  
    <li><a href="/home">Home</a></li>  
    <li><a href="/api">API</a></li>  
    <li><a href="/contact">Contact</a></li>  
    <li><a href="/career">Career</a></li>  
  </ul>  
  
  <p>Bookmark title 1</p>  
  <p>Bookmark title 2</p>  
  <p>Bookmark title 3</p>  
  <p>Bookmark title 4</p>  
  
</body>  
</html>
```

# Авторизация и аутентификация

## Ссылки:

[Fastapi users документация](#)

[Role-based authentication](#)

## Подготовка проекта

```
python -m venv --system-site-packages env
python -m pip install fastapi uvicorn
```