ОБЗОР КУРСА ЗАДАЧИ

Приглашаем преподавателей поучаствовать в интервью с нашей командой исследований. Если вы готовы участвовать, пожалуйста, заполните форму. Приглашение на интервью.

Урок Tornado

# Асинхронный веб-фреймворк Tornado

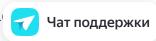
## 1. Введение

Почему Tornado? Ведь мы уже рассмотрели aiohttp, который казалось бы закрывает все потребности разработки веб-приложений: можно использовать и как веб-клиент, и как веб-сервер, и с вебсокетами можно поработать. Но, во-первых, асинхронное программирование дает наибольшее количество преимуществ (поэтому нас ждет еще один веб-фреймворк), а во вторых Tornado — это один из самых быстрых веб-фреймворков на Python и появился он еще до того, как asyncio появился в Python.

Tornado был разработан в 2009 году компанией FriendFeed, которую позже купил Facebook. Tornado был создан для обеспечения высокой производительности и масштабируемости социальных сервисов. Первоначально он был предназначен для обработки тысяч одновременных соединений благодаря своей неблокирующей модели ввода-вывода. Tornado остается популярным выбором для создания высокопроизводительных веб-приложений и реальных приложений.

Если провести сравнение с aiohttp, то можно отметить следующие моменты:

- У Tornado более высокая производительность. Он изначально разработан для работы с большим количеством одновременных соединений
- Высокая надежность благодаря долгой истории разработки
- Меньшее сообщество: меньше ресурсов и примеров по сравнению с аі-



• Нет возможности использовать как http-клиент

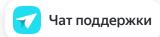
Установка Tornado:

```
pip install tornado
```

## 2. Работа с НТТР-запросами

Строго говоря, разные веб-фреймворки, по большей части, не особо отличаются друг от друга синтаксисом, поэтому примеры вызовут у вас чувство узнавания после aiohttp (или flask). Давайте посмотрим простой пример на Tornado:

Важной особенностью Tornado является то, что обработчики, которые мы делаем, оформляются в виде классов, наследующихся от tornado.web.RequestHandler. Внутри класса обработчика мы можем определить методы для обработки различных типов запросов (GET, POST, PUT, DELETE и т.д.). В примере выше мы определили метод get, который будет вызываться при обработке GET-запросов. Внутри метода мы можем получить параметры запроса с помощью метода get\_argument.



Чтобы создать приложение, мы создаем экземпляр класса

tornado.web.Application, передавая в конструктор список кортежей, где первый элемент — это регулярное выражение, которое определяет путь запроса, а второй элемент — это класс обработчика. После этого мы запускаем приложение, вызывая метод listen и передавая порт, на котором будет запущен наш сервер.

Встроенного шаблонизатора в tornado нет, но никто не запретит вам использованить сторонний, вроде прекрасного Jinja3.

Еще один пример с post-запросом и данными в json.

```
import tornado.ioloop
import tornado.web
import json
class MainHandler(tornado.web.RequestHandler):
    async def post(self):
        data = json.loads(self.request.body) # парсим json из стро
        name = data.get('name', 'Anonymous')
        age = data.get('age', 'Unknown')
        self.write(f"Hello, {name}! You are {age} years old.")
def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])
if name == " main ":
    app = make\_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

В отличие от **flask** и **aiohttp** y tornado нет встроенного парсера json, поэтому приходится использовать стандартный модуль json.

Работа с заголовками практически не отличается от других фреймворков.

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    async def get(self):
        user_agent = self.request.headers.get('User-Agent', 'Unкпо')
```

Работа с файлами организована через стандартные потоки в Python, и вообще, как вы могли заметить по работе с json, Tornado в значительной степени пологается на то, что уже есть в стандартной библиотеке, а не придумывает свои реализации.

Рассмотрим пример на загрузку файлов на сервер:

```
import tornado.ioloop
import tornado.web
import os
class UploadHandler(tornado.web.RequestHandler):
    async def post(self):
        file1 = self.request.files['file1'][0]
        original_fname = file1['filename']
        output_file = open(os.path.join("uploads", original_fname)
        output_file.write(file1['body'])
        self.write(f"File {original_fname} uploaded successfully."
def make_app():
    return tornado.web.Application([
        (r"/upload", UploadHandler),
    ])
if name == " main ":
    os.makedirs('uploads', exist_ok=True)
    app = make\_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
                                                             Чат поддержки
```

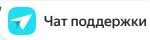
А теперь отправку файлов с сервера:

```
import tornado.ioloop
import tornado.web
import os
class DownloadHandler(tornado.web.RequestHandler):
    async def get(self, filename):
        filepath = os.path.join("uploads", filename)
        if not os.path.isfile(filepath):
            self.set status(404)
            self.write("File not found.")
            return
       with open(filepath, "rb") as f:
            self.set_header('Content-Type', 'application/octet-str
            self.set_header('Content-Disposition', f'attachment; f
            while chunk := f.read(8192):
                self.write(chunk)
        self.finish()
def make_app():
    return tornado.web.Application([
        (r"/download/(.*)", DownloadHandler),
    1)
if name == " main ":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

#### 3. Работа с WebSocket

Раз уж мы говорили про то, что aiohttp умеер работать с вебсокетами, то было бы неправильно обойти этот момент в Tornado.

В примерах с обработкой запросов мы с вами наследовали свои классы or tornado.web.RequestHandler, тут класс-родитель будет уже другой tornado.websocket.WebSocketHandler.



Фактически, для базовой работы нам необходимо определить всего три метода, которые будут определять поведения нашего приложения:

```
    open — при установке соединения

• on message — при получении сообщения
• on close — при закрытии соединения
  import tornado.ioloop
  import tornado.web
  import tornado.websocket
  class EchoWebSocket(tornado.websocket.WebSocketHandler):
      clients = set()
      def open(self):
          EchoWebSocket.clients.add(self)
          self.write_message("WebSocket opened")
          print("WebSocket opened")
      async def on_message(self, message):
          for client in EchoWebSocket.clients:
              if client is not self:
                  client.write_message(f"New message: {message}")
      def on_close(self):
          EchoWebSocket.clients.remove(self)
          print("WebSocket closed")
  def make_app():
      return tornado.web.Application([
          (r"/websocket", EchoWebSocket),
      ])
  if name == " main ":
      app = make\_app()
      app.listen(8888)
      tornado.ioloop.IOLoop.current().start()
```

Tornado позволяет сделать функцию on\_message синхронной (для этого надо просто убрать async в определении).

Чат поддержки

#### 4. Заключение

Tornado — это мощный асинхронный web-фреймворк для создания масштабируемых веб-приложений. Он поддерживает множество функций, включая работу с HTTP-запросами, обработку файлов и использование WebSocket. Мы рассмотрели историю фреймворка, его плюсы и минусы, а также сравнили его с aiohttp. Примеры кода охватывают различные аспекты работы с Tornado, предоставляя практические знания для создания веб-приложений.

Этот урок предоставляет базовые и расширенные примеры, которые помогут вам понять, как использовать **Tornado** для создания высокопроизводительных асинхронных веб-приложений.

Информационные материалы представлены Яндексом и АНО ДПО «Образовательные технологии Яндекса» в соответствии с <u>Условиями использования</u>. Не является образовательной услугой.

Справка

© 2018 - 2025 Яндекс

