Universidade Federal do ABC

Tutorial Servidor HTTP utilizando Mônada em Python

Ian Nicolas Magatti da Silva
RA:11201922124

Santo André - São Paulo
25 de Março de 2024

Este tutorial descreve como construir um servidor HTTP simples em Python. Vamos utilizar conceitos de programação funcional como base para o servidor, capacitando-o a receber e responder a solicitações HTTP e exibir uma página da web. Embora Python permite vários paradigmas, empregaremos princípios de programação funcional para esta tarefa. Mesmo que Python não seja fortemente tipado e tenha tanto suporte para construção de novos tipos como Haskell, podemos imitar comportamentos equivalentes, sendo as formas apresentadas neste tutorial como um exemplo de implementação e uma nova maneira de design para o servidor.

As principais funções que nosso servidor precisa realizar é preparar uma porta para comunicação com um cliente, receber uma requisição HTTP, conseguir interpretá-la e responder a solicitação ao cliente.
O arquivo do código fonte está no link a seguir, juntamente com a página html de exemplo e um arquivo cliente para se comunicar com o servidor.
Link: https://github.com/IanNicolasMagattiDaSilva/Tutorial-Servidor-HTTP.git

Para o desenvolvimento de novos tipos vamos utilizar classes que representaram os novos tipo criados. A seguir temos o tipo que representa os métodos HTTP e quais deles podemos ter.

```
class HttpMetodo:
  GET = 'GET'
  POST = 'POST'
  PUT = 'PUT'
  DELETE = 'DELETE'
  PATCH = 'PATCH'
  HEAD = 'HEAD'
  OPTIONS = 'OPTIONS'
```

desta forma o método que irá para nossa requisição deve pertencer diretamente a este tipo *HttpMetodo*. Com isto conseguimos verificar e selecionar se a requisição enviada pelo cliente é válida ou não a partir dos valores possiveis da classe *HttpMetodo*.

```
class RequisicaoHTTP:
     def __init__(self, method: HttpMetodo | None, path: str | None,
headers: Optional[Dict[str, str]] = None, body: Optional[str] = None):
      if method not in HttpMetodo.__dict__.values():
          raise ValueError("Método de requisição inválido.")
      else:
        self.method = getattr(HttpMetodo, method.upper())
        self.path = path
        self.headers = headers
        self.body = body
```

Da mesma forma que foi feita no *HttpMetodo* a *RequisicaoHTTP* se tornou um tipo, onde guarda os valores referente ao método, caminho, cabeçalho e corpo de uma requisição. É importante destacar que foi atribuída uma camada de verificação para o valor do método para saber se ele está dentro do valor dos valores de *HttpMetodo* e a função getattr() garante que o método utilizado na requisição será retirado de *HttpMetodo*. Esta seria uma maneira simples de aproximar os tipos visto em Haskell e em outras linguagens de programação fortemente tipadas.

Em Python, uma mônada é uma estrutura que encapsula um valor e define duas operações principais: bind & return. A mônada é utilizada para gerir o fluxo de controle em situações que implicam cálculos sequenciais, facilitando uma composição mais clara e sucinta das operações.

```
class Monad(ABC):
    @abstractmethod
    def bind(self, func):
        pass

    @abstractmethod
    def return_(self, value):
        pass
```

A classe Monad fornecida é uma base de interface para implementar uma aproximação de mônadas. Aqui está uma explicação detalhada das operações: Aqui está uma breve explicação das operações:

bind(func): A função bind é incumbida de encadear operações em uma mônada. Ela aceita uma função *func* que mapeia o valor encapsulado na mônada e retorna uma nova mônada com o resultado dessa transformação. Bind tem como objetivo aplicar a função *func* ao valor encapsulado na mônada, mantendo sua estrutura monádica.

return_(value): A criação de uma nova instância da mônada com o valor fornecido é a responsabilidade da função *return_*. Ela envolve o valor especificado na mônada e devolve essa mônada como resultado. O retorno do *return_* é criar uma mônada a partir de um valor simples, permitindo introduzir valores em um contexto monádico.

A classe Monad busca seguir a estrutura comum encontrada em linguagens funcionais para implementar mônadas e servirá como base para criar o servidor. Entretanto, em Python, a utilização de mônadas não é tão frequente como em linguagens puramente funcionais como Haskell e a linguagem de programação não há suporte nativo para esta estrutura. A razão disso é que Python tem uma abordagem mais orientada a objetos e não inclui suporte nativo para mônadas. Dessa forma vamos estruturar um servidor a partir dessa estrutura de monada:

```
class ServerMonad(Monad):
    def __init__(self, host, port):
        self.host = host
        self.port = port
```

```
        self.server_socket = None

    def bind(self, func):
        if self.server_socket is None:
                    self.server_socket  =  socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            self.server_socket.bind((self.host, self.port))
            self.server_socket.listen(1)
            print("O servidor está ouvindo...")
        return func(self.server_socket)

    def return_(self, value):
        return value
```

A classe ServerMonad utiliza do padrão criado como a classe Monad, porém desta vez para realizar as atividades básicas de qualquer servidor: guardando seu endereço e porta para socket e se disponibilizando para conexão. Mas agora com esta configuração semelhante a uma mônada podemos encadear uma função ao invés de ter necessariamente de realizar todas ações do servidor sequencialmente.
Sendo a operação para criar uma instancia de ServerMonad a seguinte:

```
server = ServerMonad('', 8080)
    server.bind(lambda socket: msg_of_client(socket.accept()[0]))
```

Sendo estas as principais características ao implementar o servidor com estes conceitos mais orientado a tipos e um estrutura monástica para o servidor, podemos concluir este servidor da seguinte maneira:

```
import socket
from abc import ABC, abstractmethod
from typing import Union, Optional, Dict

class HttpMetodo: # Classe que representa o tipo Método de HTTP
  GET = 'GET'
  POST = 'POST'
  PUT = 'PUT'
  DELETE = 'DELETE'
  PATCH = 'PATCH'
  HEAD = 'HEAD'
  OPTIONS = 'OPTIONS'

class RequisicaoHTTP: # Classe que representa o tipo Requisição HTTP
     def __init__(self, method: str | None, path: str | None, headers:
Optional[Dict[str, str]] = None, body: Optional[str] = None):
```

```
      if method not in HttpMetodo.__dict__.values():
          raise ValueError("Método de requisição inválido.")
      else:
        self.method = getattr(HttpMetodo, method.upper())
        self.path = path
        self.headers = headers
        self.body = body

def  parse_http_request(mensagem):  #  Função  utilizada  para  tratar  a
mensagem enviada pelo cliente
    lines = mensagem.split("\n")
    method, path, _ = lines[0].split()
    headers = {}
    body = ""
    for line in lines[1:]:
        if line.strip():
            key, value = line.split(":", 1)
            headers[key.strip()] = value.strip()
        else:
            break
    if len(lines) > len(headers) + 1:
        body = "\n".join(lines[len(headers) + 1:])
    return RequisicaoHTTP(method, path, headers, body)

def Func_to_request(socketConnection ,request: RequisicaoHTTP) -> None:
      #Nesta  função  você  pode  implementar  a  função  do  seu  servidor:
Fornecer uma página web ou enviar uma mensagem
    print("Método:", request.method)
    print("Caminho:", request.path)
    print("Cabeçalhos:", request.headers)
    print("Corpo:", request.body)

    if request.method == HttpMetodo.GET:
        if request.path == '/':
            request.path = "index.html"
        arq = open(f"{request.path}","r")
        content = arq.read()
        arq.close()
    response = f"HTTP/1.1 200 OK\n\n{content}\n\n"
    socketConnection.send(response.encode('utf-8'))



#----------------------------------------
```

```
class Monad(ABC):
    @abstractmethod
    def bind(self, func):
        pass

    @abstractmethod
    def return_(self, value):
        pass

class ServerMonad(Monad):
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.server_socket = None

    def bind(self, func):
        if self.server_socket is None:
                    self.server_socket  =  socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            self.server_socket.bind((self.host, self.port))
            self.server_socket.listen(1)
            print("O servidor está ouvindo...")
        return func(self.server_socket)

    def return_(self, value):
        return value

#-----------------------------------------------------------------------

def msg_of_client(client_socket):
    request = client_socket.recv(1024).decode()
    request = parse_http_request(request)
    Func_to_request(client_socket,request)


def main():
    server = ServerMonad('', 8080)
    server.bind(lambda socket: msg_of_client(socket.accept()[0]))

if __name__ == "__main__":
    main()
```

Utilizando um cliente simples em python, que utilize o método ‘GET’, ou em seu navegador e acessar ‘http://localhost:8080’, será exibido a página web *index.html* que é uma página web para servir de exemplo ao nossa intenção de desenvolver um servidor básico com alguns conceitos de programação funcional. A partir desse código é possivel desenvolver outras aplicações e até mesmo novas funcionalidades para o servidor e utilizando a função *bind* encadear uma série de funções a serem aplicado ao mesmo socket ou a mesma instancia do servidor.