Python średnio zaawansowany

Dzień 17

Blok nr 5:

Aplikacja webowa

AGENDA

- Szerzej o przypisywaniu adresów do widoków
- Widoki oparte o funkcje
- Widoki oparte o klasy
- Podejście generyczne

Widoki oparte o funkcje

Przypisywanie adresu do widoku

Na potrzeby obsługi statycznego żądania o zasób:

```
@app.route('/')   # dekorujemy widok (funkcję) informacją o (jego) URL
def index():
        return 'Hello! Here is Flask’
```

W efekcie Flask wykona funkcję *index()* dla żądania: GET http://serwer/

To widok oparty o funkcję.

Przypisywanie adresu do widoku

Żądania mogą zbudowane i obsługiwane dynamicznie:

```
@app.route('/hello/<string:imie>')    # string: nie jest obowiązkowy
def hello(imie):
    return 'Hello' + imie
```

W efekcie, Flask wykona funkcję *hello()* dla żądania: http://serwer**/hello/Jan**.

Funkcja *hello()* otrzyma parametr *imie* z żądania wysłanego przez aplikację klienta.

Więcej info nt. typów parametrów: http://flask.pocoo.org/docs/1.0/quickstart/#variable-rules

Przypisywanie adresu do widoku

Żądanie (ścieżka do zasobów) może zawierać wiele parametrów:

```
@app.route('/adres/<miasto>/<ulica>/<kod>’)
def adres(miasto, ulica, kod):
    return ’Przesłany adres: %s %s %s’ % (miasto, ulica, kod)
```

Praktyka: 01

Przypisywanie adresu do widoku

Alternatywne podejście do przypisywania widoków do obsługiwanych adresów:

```
def index():
        return 'Hello!'


app = Flask(__name__)
app.add_url_rule('/', view_func=index)
```

Praktyka: 02

Przypisywanie adresu do widoku

Przedstawione podejście jest stosowane gdy chcemy zdefiniować mapowania URL na funkcje w jednym bloku kodu, a nawet w pętli iterującej po np. słowniku.

W przeciwnym przypadku, mapowania są „rozsypane” po całym pliku co może utrudniać pracę nad kodem.

Widoki oparte o klasy, podejście generyczne

Dotychczasowe rozwiązanie

```
@app.route('/')
def hello():
    clients = Client.query.all()
    ctx = {'clients': clients}
    return render_template('clients.html', **ctx)
```

ścieżka URL

queryset

kontekst

szablon HTML

Wada: dla praktycznie każdego modelu trzeba stworzyć osobny widok.

Wada: skomplikowana logika (m.in. GET/POST) obsługiwana jest przez tylko jedną funkcję.

Rozwiązanie generyczne

Flask realizuje także podejście zwane „pluggable views”, które umożliwia bardzo elastyczne podejście do obsługi widoków.

Podejście to umożliwia tworzenie widoków w formie klas, a w efekcie umożliwia łatwe i przejrzyste ich dziedziczenie.

Rozwiązanie generyczne

Koncepcja bazuje na tworzeniu klas dziedziczących po klasie **View.**

Klasa dziedzicząca rozszerza funkcjonalność o obsługę specyficznych przypadków, czyli modyfikuje tylko fragment dotychczasowej funkcjonalności zachowując pozostałe właściwości klasy rodzica.

Więcej informacji: http://flask.pocoo.org/docs/1.0/views/

Rozwiązanie generyczne

```
class ListView(View):
    def get_template_name(self):
        raise NotImplementedError()

    def get_context(self):
        return {'objects': self.get_objects()}

    def get_objects(self):
        raise NotImplementedError()

    def render_template(self, context):
        return render_template(self.get_template_name(), **context)

    def dispatch_request(self):
        return self.render_template(self.get_context())
```

kontekst

queryset

szablon HTML

obsługuje wywołanie widoku

Wykorzystanie widoków klas

```
class ClientView(ListView):
    def get_template_name(self):
        return 'clients.html'

    def get_objects(self):
        return Client.query.all()
```

szablon HTML

queryset

```
app.add_url_rule('/clients/', view_func=ClientView.as_view('clients_list'))
```

ścieżka URL

klasa realizująca widok

widok dostępny pod tą nazwą dla url_for()

Wykorzystanie widoków klas

- bazowy widok implementuje metody generyczne niezbędne do poprawnego działania
- klasy dziedziczące implementują tylko te metody, które są wymagane
- istnieje możliwość tworzenia rozbudowanych widoków poprzez wielodziedziczenie

Wykorzystanie widoków klas

- podejście to ułatwia modyfikacje - np.:
  - wymianę szablonu w widoku dziedziczącym,
  - rozdzielenie obsługi GET od POST
- gdy widoki rozrastają się, podejście klasowe może pomóc zapanować nad kodem przez jego dekompozycję i ponowne użycie

Wykorzystanie widoków klas

+ gotowe rozwiązanie dla powtarzalnych operacji
+ testowalność kodu
+ przejrzysty kod

- w przypadku skomplikowanego modelu należy rozważyć rezygnację z dziedziczenia i zaimplementować niezależny model

Dzięki!