

Lista ToDo – Aplikacja internetowa

Opis: Realizacja prostej listy ToDo (lista zadań do zrobienia), jako aplikacji internetowej, z wykorzystaniem Pythona i frameworka Flask w wersji 0.10.1.

Autorzy: Tomasz Nowacki, Robert Bednarz

Czas realizacji: 90 min

Poziom trudności: Poziom 2

Spis treści

Lista ToDo – Aplikacja internetowa.....	1
I. Katalog, plik i przeznaczenie aplikacji.....	2
Terminal I.1.....	2
II. Szkielet aplikacji.....	2
Kod II.1.....	2
III. Definiowanie widoków.....	3
Kod III.1.....	3
IV. Model bazy danych.....	3
Kod IV.1.....	3
Terminal IV.1.....	3
V. Połączenie z bazą danych.....	4
Kod V.1.....	4
VI. Pobieranie i wyświetlanie danych.....	5
Kod VI.1.....	5
Kod VI.2.....	5
VII. Formularz dodawania zadań.....	6
Kod VII.1.....	6
Kod VII.2.....	7
VIII. Wygląd aplikacji (opcja).....	8
Kod VIII.1.....	8
Kod VIII.2.....	9
IX. Oznaczanie zadań jako wykonane (opcja).....	9
Kod IX.1.....	9
Kod IX.2.....	10
Zadania dodatkowe:.....	11

I. Katalog, plik i przeznaczenie aplikacji

Zaczynamy od utworzenia katalogu projektu **ToDo**, w którym zamieścimy wszystkie pliki niezbędne do realizacji tej implementacji. W katalogu użytkownika tworzymy nowy katalog **todo**, a w nim plik **todo.py**:

Terminal I.1

```
~ $ mkdir todo; cd todo; touch todo.py
```

Nasza aplikacja ma pozwalać na dodawanie z określoną datą, przeglądanie i oznaczanie jako wykonane różnych zadań, które zapisywane będą w bazie danych SQLite.

II. Szkielet aplikacji

Utworzenie minimalnej aplikacji Flask pozwoli na uruchomienie serwera deweloperskiego, umożliwiającego wygodne rozwijanie kodu. W pliku **todo.py** wpisujemy:

Kod II.1

```
# -*- coding: utf-8 -*-
# todo/todo.py

from flask import Flask

app = Flask(__name__)

if __name__ == '__main__':
    app.run(debug=True)
```

Serwer uruchamiamy komendą: **python todo.py**

```
* Running on http://127.0.0.1:5000/
* Restarting with reloader
```

Domyślnie serwer uruchamia się pod adresem **127.0.0.1:5000**. Po wpisaniu adresu do przeglądarki internetowej otrzymamy stronę z błędem HTTP 404, co wynika z faktu, że nasza aplikacja nie ma jeszcze zdefiniowanego żadnego zachowania (widoku) dla tego adresu. W uproszczeniu możemy **widok** utożsamiać z pojedynczą stroną w ramach aplikacji internetowej.

Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

III. Definiowanie widoków

W pliku **todo.py** umieścimy funkcję `index()`, domyślny widok naszej strony:

Kod III.1

```
# -*- coding: utf-8 -*-
# todo/todo.py

from flask import Flask
app = Flask(__name__)

# dekorator łączący adres główny z widokiem index
@app.route('/')
def index():
    return 'Hello, SWOI'

if __name__ == '__main__':
    app.run(debug=True)
```

Widok `index()` za pomocą dekoratora związaliśmy z adresem głównym (/). Po odświeżeniu adresu `127.0.0.1:5000` zamiast błędu powinniśmy zobaczyć napis: "Hello, SWOI"

IV. Model bazy danych

W katalogu aplikacji tworzymy plik `schema.sql`, który zawiera opis struktury tabel z zadaniami. Do tabel wprowadzimy przykładowe dane.

Kod IV.1

```
-- todo/schema.sql

-- tabela z zadaniami
drop table if exists entries;
create table entries (
    id integer primary key autoincrement, -- unikalny indentyfikator
    title text not null, -- opis zadania do wykonania
    is_done boolean not null, -- informacja czy zadania zostalo juz wykonane
    created_at datetime not null -- data dodania zadania
);

-- pierwsze dane
insert into entries (id, title, is_done, created_at)
values (null, 'Wyrzucić śmieci', 0, datetime(current_timestamp));
insert into entries (id, title, is_done, created_at)
values (null, 'Nakarmić psa', 0, datetime(current_timestamp));
```

Tworzymy bazę danych w pliku `db.sqlite`, łączymy się z nią i próbujemy wyświetlić dane, które powinny być zostać zapisane w tabeli `entries`:

Terminal IV.1

```
sqlite3 db.sqlite < schema.sql
sqlite3 db.sqlite
select * from entries;
```

```
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from entries;
1|Wyrzucić śmieci|0|2014-08-13 10:39:58
2|Nakarmić psa|0|2014-08-13 10:39:58
sqlite>
```

Połączenie z bazą zamykamy poleceniem `.quit`.

V. Połączenie z bazą danych

W pliku **todo.py** dodajemy:

Kod V.1

```
# -*- coding: utf-8 -*-
# todo/todo.py

# importujemy biblioteki potrzebne do nawiązania połączenia z baza
import os
import sqlite3

from flask import Flask, g

app = Flask(__name__)

# konfiguracja aplikacji
app.config.update(dict(
    # nieznany nikomu sekret dla mechanizmu sesji
    SECRET_KEY = 'bardzosekretnawartosc',
    # położenie naszej bazy
    DATABASE = os.path.join(app.root_path, 'db.sqlite'),
    # nazwa aplikacji
    SITE_NAME = 'Moja lista ToDo'
))

def connect_db():
    """Nawiązywanie połączenia z baza danych określona w konfiguracji."""
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv

def get_db():
    """Funkcja pomocnicza, która tworzy połączenia z baza przy pierwszym
    wywołaniu i umieszcza ją w kontekście aplikacji (obiekt g). W kolejnych
    wywołaniach zwraca połączenie z kontekstu."""
    if not hasattr(g, 'db'):
        g.db = connect_db() # jeżeli kontekst nie zawiera informacji o połączeniu to je tworzymy
    return g.db # zwracamy połączenie z baza

# dekorator wykonujący funkcje po wysłaniu odpowiedzi do klienta
@app.teardown_request
```

```
def close_db(error):
    """Zamykanie połączenia z baza."""
    if hasattr(g, 'db'):
        g.db.close()

# dekorator łączący adres główny z widokiem index
@app.route('/')
def index():
    return 'Hello, SWOI'

if __name__ == '__main__':
    app.run(debug=True)
```

Dodaliśmy sekretny klucz zabezpieczający mechanizm sesji, ustawiliśmy ścieżkę do pliku bazy danych (w katalogu aplikacji, dlatego funkcja `app.root_path`) oraz nazwę aplikacji. Utworzyliśmy trzy funkcje odpowiedzialne za nawiązywanie (`connect_db`, `get_db`) i kończenie (`close_db`) połączenia z bazą danych.

VI. Pobieranie i wyświetlanie danych

Wyświetlanie danych umożliwia wbudowany we Flask system **szablonów** (templatek), czyli mechanizm renderowania kodu HTML na podstawie plików zawierających kod wstawiający wybrane dane z aplikacji oraz znaczniki HTML. Modyfikujemy funkcję `index()` w pliku **todo.py**:

Kod VI.1

```
# dodajemy nowe importy do pozostałych
from flask import render_template

# dekorator łączący adres główny z widokiem index
@app.route('/')
def index():
    db = get_db() # łączymy się z baza
    # pobieramy wszystkie wpisy z bazy:
    cur = db.execute('select id, title, is_done, created_at from entries order
by created_at desc;')
    entries = cur.fetchall()
    # renderujemy tempaltke i zwracamy ja do klienta
    return render_template('show_entries.html', entries=entries)
```

W widoku `index()` pobieramy obiekt bazy danych (`db`) i wykonujemy zapytanie (`select...`), by wyciągnąć wszystkie zapisane zadania. Na koniec renderujemy szablon przekazując do niej pobrane zadania (`entries`). Szablon, czyli plik `show_entries.html`, umieszczamy w podkatalogu `templates` aplikacji. Poniżej jego zawartość:

Kod VI.2

```
<!-- todo/templates/show_entries.html -->
<html>
  <head>
    <!-- automatycznie przekazana informacja z konfiguracji aplikacji -->
    <title>{{ config.SITE_NAME }}</title>
  </head>
```

```
<body>
  <h1>{{ config.SITE_NAME }}:</h1>
  <ol>
    <!-- wypisujemy kolejno wszystkie zdania -->
    {% for entry in entries %}
      <li>
        {{ entry.title }}<em>{{ entry.created_at }}</em>
      </li>
    {% endfor %}
  </ol>
</body>
</html>
```

Wewnątrz szablonu przeglądamy wszystkie wpisy (**entries**) i umieszczamy je na liście HTML. Do szablonu automatycznie przekazywany jest obiekt **config** (dane konfiguracyjne), z którego pobieramy tytuł strony (**site_name**). Po odwiedzeniu strony 127.0.0.1:5000 powinniśmy zobaczyć listę zadań.

Moja lista ToDo:

1. Wyrzucić śmieci2014-08-13 10:39:58
2. Nakarmić psa2014-08-13 10:39:58

VII. Formularz dodawania zadań

Aby umożliwić dodawanie i zapisywanie w bazie nowych zadań, modyfikujemy widok **index()**, tak aby obsługiwał żądania POST, które zawierają dane wysłane z formularza na serwer.

Kod VII.1

```
# dodajemy importy
from datetime import datetime
from flask import flash, redirect, url_for, request

# dekorator łączący adres główny z widokiem index
# poza powiązaniem adresu / z funkcją index, dodajemy możliwość akceptacji
# zadan HTTP POST (domyslnie dozwolone sa tylko zadania GET)
@app.route('/', methods=['GET', 'POST'])
def index():
    """Główny widok strony. Obsługuje wyświetlanie i dodawanie zadan."""
    # zmienna przechowująca informacje o ewentualnych błędach
    error = None

    # jeżeli otrzymujemy dane POST z formularza, dodajemy nowe zadanie
    if request.method == 'POST':
        if len(request.form['entry']) > 0: # sprawdzamy poprawność przesłanych danych
            db = get_db() # nawiązujemy połączenie z bazą danych
            new_entry = request.form['entry'] # wyciągamy treść zadania z przesłanego formularza
            is_done = '0' # ustalamy, że nowo dodane zadanie nie jest jeszcze wykonane
```

```
        created_at = datetime.now() # data dodania
        # zapytanie do bazy, ktore wstawia nowy wiersz z danymi
        db.execute('insert into entries (title, is_done, created_at)
values (?, ?, ?);', [new_entry, is_done, created_at])
        db.commit() # wykonujemy zapytanie
        flash('Dodano nowe zadanie') # informacja o pomyslnym dodaniu nowego zadania
        return redirect(url_for('index')) # przekierowujemy na strone glowna
        error = u'Nie mozesz dodac pustego zadania' # komunikat o bledzie

db = get_db() # laczymy sie z baza
# pobieramy wszystkie wpisy z bazy:
cur = db.execute('select id, title, is_done, created_at from entries order
by created_at desc;')
entries = cur.fetchall()
# renderujemy tempaltke i zwracamy do klienta:
return render_template('show_entries.html', entries=entries, error=error)
```

Wpisując adres w polu adresu przeglądarki, wysyłamy do serwera żądanie typu GET, które obsługujemy zwracając klientowi odpowiednie dane (listę zadań). Natomiast żądania typu POST są wykorzystywane do zmiany informacji na serwerze (np. dodania nowego wpisu). Dlatego widok `index()` rozszerzyliśmy o sprawdzanie typu żądania, w razie wykrycia danych POST, sprawdzamy poprawność danych przesłanych z formularz i jeżeli są poprawne, dodajemy nowe zadanie do bazy. W przeciwnym razie zwracamy użytkownikowi informację o błędzie.

Szablon **show_entries.html** aktualizujemy, dodając odpowiedni formularz:

Kod VII.2

```
<!-- todo/templates/show_entries.html -->
<html>
  <head>
    <title>{{ config.SITE_NAME }}</title>
  </head>

  <body>
    <h1>{{ config.SITE_NAME }}:</h1>

    <!-- formularz dodawania zadania -->
    <form class="add-form" method="POST" action="{{ url_for('index') }}">
      <input name="entry" value=""/>
      <button type="submit">Dodaj zadanie</button>
    </form>

    <!-- informacje o sukcesie lub bledzie -->
    <p>
      {% if error %}
        <strong class="error">Błąd: {{ error }}</strong>
      {% endif %}

      {% for message in get_flashed_messages() %}
        <strong class="success">{{ message }}</strong>
      {% endfor %}
    </p>
```

```
<ol>
  {% for entry in entries %}
    <li>
      {{ entry.title }}<em>{{ entry.created_at }}</em>
    </li>
  {% endfor %}
</ol>

</body>
</html>
```

W szablonie dodaliśmy formularz oraz informację o błędzie lub sukcesie przy próbie dodawania zadania. Określając atrybut `action` w formularzu, skorzystaliśmy z wbudowanej funkcji `url_for`, która zamienia nazwę widoku (w tym wypadku `index`) na odpowiadający jej adres URL (w tym wypadku `/`). W ten sposób łączymy formularz z konkretną funkcją Pythonową (widokiem), a nie z adresem URL.

Moja lista ToDo:

Dodano nowe zadanie

1. nowe zadanie2014-08-13 14:12:47.358445
2. Wyrzucić śmieci2014-08-13 10:39:58
3. Nakarmić psa2014-08-13 10:39:58

VIII. Wygląd aplikacji (opcja)

Wygląd aplikacji możemy zdefiniować w arkuszu stylów CSS o nazwie **style.css**, który zapisujemy w podkatalogu `static` aplikacji:

Kod VIII.1

```
/* todo/static/style.css */

body { margin-top: 20px; }
h1, p { margin-left: 20px; }
.add-form { margin-left: 20px; }
ol { text-align: left; }
em { font-size: 11px; margin-left: 10px; }
form { display: inline-block; margin-bottom: 0; }
input[name="entry"] { width: 300px; }
input[name="entry"]:focus {
  border-color: blue;
  border-radius: 5px;
}
li { margin-bottom: 5px; }
button {
  padding: 0;
```



```
cursor: pointer;
font-size: 11px;
background: white;
border: none;
color: blue;
}
.error { color: red; }
.success { color: green; }
.done { text-decoration: line-through; }
```

Zdefiniowane style podpinamy do pliku **show_entries.html**, dodając w sekcji head wpis **<link... >**:

Kod VIII.2

```
<head>
  <title>{{ config.SITE_NAME }}</title>
  <link rel="stylesheet" type="text/css" href="{{ url_for('static',
filename='style.css') }}">
</head>
```

Dzięki temu nasza aplikacja nabierze nieco lepszego wyglądu.

Moja lista ToDo:

[Dodaj zadanie](#)

Dodano nowe zadanie

1. kolejne ważne zadanie 2014-08-13 14:32:51.449445
2. nowe zadanie 2014-08-13 14:29:45.798655
3. Wyrzucić śmieci 2014-08-13 12:29:32
4. Nakarmić psa 2014-08-13 12:29:32

IX. Oznaczanie zadań jako wykonane (opcja)

Do każdego zadania dodamy formularz, którego wysłanie będzie oznaczało, że wykonaliśmy dane zadanie, czyli zmienimy atrybut **is_done** wpisu z 0 (niewykonane) na 1 (wykonane). Odpowiednie żądanie typu POST obsłuży nowy widok w pliku **todo.py**. W szablonie **show_entries.html** dodamy kod wyróżniający zadania wykonane.

Kod IX.1

```
# todo/todo.py

# dodajemy ponad definicja if __main__(...)

# nadajemy osobny adres, oraz zezwalamy jedynie na zadania typu POST
@app.route('/mark_as_done', methods=['POST'])
```

```
def mark_as_done():
    """Zmiana statusu zadania na wykonane."""
    # z przeslanego formularza pobieramy identyfikator zadania
    entry_id = request.form['id']
    db = get_db() # laczymy sie z baza danych
    # przygotowujemy zapytanie aktualizujace pole is_done zadania o danym identyfikatorze
    db.execute('update entries set is_done=1 where id=?', [entry_id,])
    db.commit() # zapisujemy nowe dane
    return redirect(url_for('index')) # na koniec przekierowujemy na liste wszystkich zadan
```

W szablonie **show_entries.html** modyfikujemy fragment wyświetlający listę zadań i dodajemy formularz:

Kod IX.2

```
<ol>
  {% for entry in entries %}
    <li>
      <!-- dodatkowe dekoracje dla zadan zakonczonych -->
      {% if entry.is_done %}
        <span class="done">
      {% endif %}

      {{ entry.title }}<em>{{ entry.created_at }}</em>

      <!-- dodatkowe dekoracje dla zadan zakonczonych -->
      {% if entry.is_done %}
        </span>
      {% endif %}

      <!-- formularz zmiany statusu zadania -->
      {% if not entry.is_done %}
        <form method="POST" action="{{ url_for('mark_as_done') }}">

          <!-- wysylamy jedynie informacje o id zadania -->
          <input type="hidden" name="id"

value="{{ entry.id }}" />

          <button type="submit">Wykonane</button>
        </form>
      {% endif %}
    </li>
  {% endfor %}
</ol>
```

Aplikację można uznać za skończoną. Możemy dodawać zadania oraz zmieniać ich status.

Moja lista ToDo:

[Dodaj zadanie](#)

Błąd: Nie możesz dodać pustego zadania

1. kolejne ważne zadanie 2014-08-13 14:32:51.449445 [Wykonane](#)
2. nowe zadanie 2014-08-13 14:29:45.798655
3. Wyrzucić śmieci 2014-08-13 12:29:32 [Wykonane](#)
4. Nakarmić psa 2014-08-13 12:29:32

Zadania dodatkowe:

- Dodać możliwość usuwania zadań.
- Dodać mechanizm logowania użytkownika tak, aby użytkownik mógł dodawać i edytować tylko swoją listę zadań.
- Można rozważyć wprowadzenie osobnych list dla każdego użytkownika.

Film instruktażowy: <http://youtu.be/1Fik9AShSpo>

Słownik pojęć:

- Aplikacja – program komputerowy.
- Baza danych – program przeznaczony do przechowywania i przetwarzania danych.
- CSS – język służący do opisu formy prezentacji stron WWW.
- Framework – zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji.
- GET – typ żądania HTTP, służący do pobierania zasobów z serwera WWW.
- HTML – język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.
- HTTP – protokół przesyłania dokumentów WWW.
- POST – typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW.
- Serwer deweloperski – serwer używany w czasie prac nad oprogramowaniem.
- Serwer WWW – serwer obsługujący protokół HTTP.
- Templatka – szablon strony WWW wykorzystywany przez Flask do renderowania widoków.
- URL – ustandaryzowany format adresowania zasobów w internecie (przykład: adres strony WWW).
- Widok – fragment danych, który jest reprezentowany użytkownikowi.

Materiały pomocnicze:

1. Strona projektu Flask <http://flask.pocoo.org/>
2. Informacje o SQLite <http://pl.wikipedia.org/wiki/SQLite>
3. Co to jest framework? <http://pl.wikipedia.org/wiki/Framework>
4. Co nieco o HTTP i żądaniach GET i POST <http://pl.wikipedia.org/wiki/Http>