${Pylons} 0.9.7 - Przewodnik część 2

Tworzymy formularz.

Spróbujmy stworzyć w naszym projekcie formularz do rejestracji użytkowników. Będzie on posiadał cztery pola: login, hasło, hasło_potwierdzone oraz e-mail. W tym celu dodajmy nową akcję kontrolera users:

```
class UsersController(BaseController):
    def index(self):
        # Return a rendered template
        #   return render('/template.mako')
        # or, Return a response
        c.name = "d'Arc"
        return render('users/index.xhtml')

    def register(self):
        return render('users/register.xhtml')
```

a następnie widok *users/register.xhtml*:

```
<form action="create" method="post">
  <fieldset>
    <legend>Formularz rejestracyjny</legend>
    <div><label for="loginField">Login: </label><input
type="text" name="login" id="loginField" /></div>
    <div><label for="emailField">E-mail: </label><input
type="text" name="email" id="emailField" /></div>
    <div><label for="passwordField">Hasło: </label><input
type="password" name="password" id="passwordField"
/></div>
    <div><label for="password_confirmationField">Potwierdź
hasło: </label><input type="password"
name="password_confirmation"
id="password_confirmationField" /></div>
  </fieldset>
  <div><input type="submit" name="create"
value="Zarejestruj" /></div>
</form>
```

Przyjrzyjmy się kodowi HTML. Pierwszym elementem, na który warto zwrócić uwagę to parametr *action*. Jest ona po prostu nazwą akcji. Dane z forumlarza zostaną wysłane do metody *create*. Nie stworzyliśmy jeszcze niczego o tej nazwie, jednak na to przyjdzie czas później. Dane będą przesyłane metodą post, choćby dlatego, aby informacje takie jak hasło nie pojawiły się przypadkiem w pasku adresu.

Ale czy Pylons może nam jakoś pomóc przy tworzeniu kodu formularza ? Oczywiście :) W skład

Pylons wchodzi między innymi *WebHelpers*[1]. Jest to zbiór funkcji, które, między innymi, próbują skróci czas jaki poświęcamy na klepanie kodu HTML starając się wygenerować go za nas. Po kolei.

Ładujemy helpery.

Aby móc korzystać z helperów musimy na starcie je załadować. W tym celu wyedtujmy plik *lib/helpers.py* i dopiszmy na końcu linijkę:

```
"""Helper functions

Consists of functions to typically be used within
templates, but also available to Controllers. This module
is available to both as 'h'.
"""

# Import helpers as desired, or define your own, ie:
# from webhelpers.html.tags import checkbox, password
from webhelpers.html.tags import form, text, password,
submit, end_form
```

Dzięki temu w naszych szablonach będą dostępne funkcje potrzebne nam do wygenerowania formularzy. Czas na edycję naszego widoku. Zrobimy to krok po kroku.

Przechodzimy na helpery.

Na starcie wygenerujmy sam tag <*form*>:

```
${h.form('/create', method='post')}
  <fieldset>
    <legend>Formularz rejestracyjny</legend>
    <div><label for="loginField">Login: </label><input
type="text" name="login" id="loginField" /></div>
    <div><label for="emailField">E-mail: </label><input
type="text" name="email" id="emailField" /></div>
    <div><label for="passwordField">Hasło: </label><input
type="pasword" name="password" id="passwordField" /></div>
    <div><label for="password_confirmationField">Potwierdź
hasło: </label><input type="password"
name="password_confirmation"
id="password_confirmationField" /></div>
  </fieldset>
  <div><input type="submit" name="create"
value="Zarejestruj" /></div>
</form>
```

Powyższa linijka generuje dokładnie taki sam kod HTML jak ten pisany przez nas wcześniej. Dużo krócej - prawda ? W porządku. Teraz pozamieniajmy pola tekstowe:

1 http://docs.pylonshq.com/thirdparty/webhelpers/index.html

```
${h.form('/create', method='post')}
  <fieldset>
    <legend>Formularz rejestracyjny</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", id="loginField")}</div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", id="emailField")}</div>
    <div><label for="passwordField">Hasło: </label><input
type="password" name="password" id="passwordField"
/></div>
    <div><label for="password_confirmationField">Potwierdź
hasło: </label><input type="password"
name="password_confirmation"
id="password_confirmationField" /></div>
  </fieldset>
  <div><input type="submit" name="create"
value="Zarejestruj" /></div>
</form>
```

Ilość kodu, jaki musimy napisać aby nasz formularz ukazał się światu - systematycznie maleje :) O to chodzi. Teraz czas na pola haseł:

```
${h.form('/create', method='post')}
  <fieldset>
    <legend>Formularz rejestracyjny</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", id="loginField")}</div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", id="emailField")}</div>
    <div><label for="passwordField">Hasło: </label>$
{h.password("password", id="passwordField")}</div>
    <div><label for="password_confirmationField">Potwierdź
hasło: </label>${h.password("password_confirmation",
id="password_confirmationField")}</div>
  </fieldset>
  <div><input type="submit" name="create"
value="Zarejestruj" /></div>
</form>
```

Na koniec wygenerujemy jeszcze przycisk *submit* oraz znacznik końca formularza.

```
${h.form('/create', method='post')}
  <fieldset>
    <legend>Formularz rejestracyjny</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", id="loginField")}</div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", id="emailField")}</div>
    <div><label for="passwordField">Hasło: </label>$
{h.password("password", id="passwordField")}</div>
    <div><label for="password_confirmationField">Potwierdź
hasło: </label>${h.password("password_confirmation",
id="password_confirmationField")}</div>
  </fieldset>
  <div>${h.submit("create", "Zarejestruj")}</div>
${h.end_form()}
```

Jak rozumieć istnienie helperów ?

No właśnie. Przecież to samo możemy napisać ręcznie. Można na to spojrzeć z kilku stron. Po pierwsze: dobrze napisane helpery dają nam pewność, że generowany przez nie kod będzie zgodny ze standardami, na przykład W3C. Jeżeli zależy nam na standardach i chcemy ograniczyć czas poświęcony na sesje spędzone z walidatorem można to postrzegać jako jakiś zysk. Helpery tworzą też swoistą *warstwę abstrackji*. Jeżeli powiedzmy za kilka lat tworzenie stron internetowych w niczym nie będzie przypominało dzisiejszego kodu HTML. Wtedy wystarczy podmienić w całym Framerowku to co generują helpery, wgrać nową jego wersję i nasza strona spełnia już najnowsze standardy (o ile oczywiście 100% portalu zostało wygenerowane - co jest tylko teoretycznie możliwe). Czy więc zawsze należy używać helperów ? Nie. Należy ich używać z rozsądkiem. Dobrym przykładem "zbędnego" użycia jest ostatnia linijka naszego formularza. *${h.end_form()}* jest znacznie dłuższe niż *</form>*.

Istnieją frameworki, w których nie znajdziesz funkcji odpowiedzialnej za wygenerowanie "nieopłacalnych" z punktu widzenia długości kodu tagów: przykładowo *</form>*.

Wyświetlamy zawartość formularza.

Aby wyświetlić zawartość formularza stworzymy w kontrolerze *users* akcję *create* a następnie wywołamy szablon, do którego przekażemy wysłane wartości. Do dzieła. Na start wyedytujmy plik *controllers/users.py*:

```
class UsersController(BaseController):
    def index(self):
        # Return a rendered template
        #   return render('/template.mako')
        # or, Return a response
        c.name = "d'Arc"
        return render('users/index.xhtml')

    def register(self):
        return render('users/register.xhtml')

    def create(self):
        c.login = request.POST['login']
        c.email = request.POST['email']
        c.password = request.POST['password']
        c.password_confirmation =
request.POST['password_confirmation']
        return render('users/create.xhtml')
```

Zatrzymajmy się na chwilkę i przeanalizujmy co się tutaj tak właściwie dzieje. Po pierwsze tworzymy cztery zmienne globalne, które za moment wyświetlimy w naszym widoku. Nowym elementem jest niewątpliwie *request.POST*. Obiekt *request* zawiera między innymi słownik *POST*, z którego pobieramy dane wysłane wcześniej formularzem. Na koniec renderujemy widok, który za moment stworzymy.

Osoby znające troszkę lepiej Pylons mogą zastanawiać się dlaczego nie użyto metody *request.params*. W naszym formularzu jawnie zarządaliśmy wysyłani danych metodą *POST*. Dane pobierane za pomocą *request.params* mogłyby zostać równie dobrze przesłane metodą *GET*. Warto o tym pamiętać. Mając to na uwadze możemy upewnić się skąd pochodzą informacje, konkretyzując z jakiego źródła je pobieramy. Użycie *request.POST*, jest w tym momencie celowe.

Czas na widok. Stwórz plik *users/create.xhtml*.

```
<strong>Login</strong>: ${c.login}<br />
<strong>E-mail</strong>: ${c.email}<br />
<strong>Hasło</strong>: ${c.password}<br />
<strong>Potwierdzenie hasła</strong>: $
{c.password_confirmation}
```

Teraz po wysłaniu formularza zobaczymy wpisane przez nas dane.

Walidacja formularza.

Nigdy nie polegaj **wyłącznie** na walidacji po stronie klienta opartej o *JavaScript* czy *AJAX*. Technologie te mogą znacznie uprzyjemnić i ułatwić wprowadzenie poprawnych danych użytkownikowi, "na bieżąco" informując go o nieprawidłowościach, bez potrzeby wysyłani formularza jednak metody stosowane po stronie klienta w żaden sposób nie gwarantują wysłania formularzem poprawnych wartości. Wniosek ? **Zawsze sprawdzaj dane po stronie serwera**: mechanizmy *client-side* traktuj jako dodatek ale nigdy im nie ufaj.
Warto zauważyć, że szalenie użyteczne jest tutaj wykorzystanie zapytań *AJAX* dla reguł walidacji. Dzięki temu mamy szansę przestrzegać zasady *DRY* i nie implementować ponownie tych samych reguł walidacji w innym języku programowania, po stronie klienta. Daje nam to automatyczną koniunkcję iż formularz zwalidowany po stronie użytkownika poprawnie przejdzie testy po stronie serwera.

Zazwyczaj formularze rejestracyjne wyposażone są w mechanizm walidacji, który wymusza na nas aby hasła były zgodne, login nie był za krótki, e-mail posiadał formę e-maila i wiele innych. Spróbujemy wyposażyć w takie mechanizmy również naszą stronę. *Pylons* potrafi korzystać, ze stworzonego specjalnie w tym celu mechanizmu zwanego *FormEncode*[2]. Spróbujmy użyć go w naszym przypadku. Zacznijmy od zdefiniowania kryteriów. Umówmy się, że login nie będzie mógł być krótszy niż cztery znaki i dłuższy niż 25, E-mail będzie musiał przypominać e-mail, a hasła będą musiały być nie krótsze niż 8 znaków i zgodne ze sobą. Wszystkie pola będą wymagane. Ok.

Walidacja naszego formularza będzie polegała na stworzeniu klasy opisującej wcześniej założone przez nas obostrzenia i dodaniu *dekoratora* do metody mającej odebrać dane z naszego widoku. Skoro wiemy już jak wszystko ma się zachowywać czas na przejście do konkretu. Zaczniemy od stworzenia klasy. W tym celu dodajmy do naszego projektu plik *model/form.py*:

```
import formencode

class RegisterForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
```

Powyższy kod to nic innego jak zaimportowanie na starcie *formencode*, z którego będziemy korzystać i stworzenie klasy. Dziedziczy po klasie *Schema*. Dodaliśmy na początek dwa atrybuty. Otóż chodzi o to, że w naszym formularzu istnieją pola, które nie powinny być poddane walidacji. Jest to na przykład nasz przycisk *submit*. Ustawienie wartości *allow_extra_fields = True* oraz *filter_extra_fields = True* pozwoli ominąć te, dla których nie ustawimy żadnych restrykcji.

Na starcie zacznijmy od minimalnej długości hasła. Ma być nie krótsze niż 8 znaków:

2 http://formencode.org/

```
import formencode

class RegisterForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    password = formencode.validators.MinLength(8,
not_empty=True)
```

Proszę bardzo. Powyższa linjka to wywołanie walidatora *MinLength* dla pola *password*. Pierwszym parametrem jest liczba osiem. Dodatkowo dodaliśmy parametr *not_empty=True*, który wywoła odpowiedni błąd gdy będziemy chcieli zostawić puste pole. Jak łatwo się domyślić skoro istnieje *MinLength* to istnieje również *MaxLength*. Mamy więc komplet potrzebny do walidacji pola loginu. Jak jedna połączyć dwa warunki. Służy do tego metoda *All*:

```
import formencode

class RegisterForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    login =
formencode.All(formencode.validators.MinLength(4,
not_empty=True), formencode.validators.MaxLength(25,
not_empty=True))
    password = formencode.validators.MinLength(8,
not_empty=True)
```

Co tutaj się wydarzyło ? Skorzystaliśmy z metody *All* aby spiąć klamrą nasze dwa warunki. Pierwszy z nich mówi o tym, że minimalna długość loginu jest równa 4 i pole nie może być puste. Drugi, że maksymalna długość loginu jest równa 25 i pole nie może być puste. Oczywiście - dublowanie warunku *not_empty=True* nie jest konieczne.

W porządku. Mamy login, mamy sprawdzanie długości hasła. Brakuje jeszcze sprawdzania poprawności e-mail-a i sprawdzanie zgodności powtórzonego hasła. Zacznijmy od adresu skrzynki pocztowej:

```
import formencode

class RegisterForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    login =
formencode.All(formencode.validators.MinLength(4,
not_empty=True), formencode.validators.MaxLength(25,
not_empty=True))
    email = formencode.validators.Email(not_empty=True)
    password = formencode.validators.MinLength(8,
not_empty=True)
```

Tym razem skorzystaliśmy z walidatora *Email*, i dodaliśmy znanym nam już skądinąd warunek iż

pole jest wymagane. Teraz czas na potwierdzenie poprawności hasła. Po pierwsze atrybuty, dla których nie stworzyliśmy regułek nie są dostępne w naszej klasie (dwie pierwsze wartości), tak więc nasza klasa nie wie jeszcze nic o polu *password_confirmation*. Łatwo temu zaradzić tworząc jaką neutralną zasadę. Powiedzmy, że hasło powtórzone będzie musiało być s*tringiemi*:

```
class RegisterForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    login =
formencode.All(formencode.validators.MinLength(4,
not_empty=True), formencode.validators.MaxLength(25,
not_empty=True))
    email = formencode.validators.Email(not_empty=True)
    password = formencode.validators.MinLength(8,
not_empty=True)
    password_confirmation = formencode.validators.String()
```

Fantastycznie. Skoro już zaradziliśmy naszemu "problemowi: i nasza klasa wie już o polu *password_confirmation* możemy sprawdzić czy zawiera to samo co *password.* Skorzystamy tutaj z metody w walidacji o nazwie *FieldsMatch*, który przyjmuje jako parametry nazwy atrybutów, które mają być ze sobą zgodne. Aby jednak porównać pola muszą one przejść wszystkie poprzednie testy. Wskażemy to przypisując regułkę do pola *chained_validators*. Ale szkoda gadać. Kod będzie mówił sam za siebie:

```
class RegisterForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    login =
formencode.All(formencode.validators.MinLength(4,
not_empty=True), formencode.validators.MaxLength(25,
not_empty=True))
    email = formencode.validators.Email(not_empty=True)
    password = formencode.validators.MinLength(8,
not_empty=True)
    password_confirmation = formencode.validators.String()
    chained_validators =
[formencode.validators.FieldsMatch('password',
'password_confirmation')]
```

Wiele więcej informacji o możliwych do zastosowania walidatorach oraz ich działaniu znajdziesz w *PylonsBook*[3] oraz *dokumentacji FormEncode*[4].

3 http://pylonsbook.com/
4 http://formencode.org/Validator.html

Podczas pisania kursu próbowałem zastosować wersję walidowania e-maili z włączonym sprawdzaniem domeny w DNS, parametrem *resolve_domain=True*. Pierwszym błędem jaki zaczął zwracać *Pylons* był rzekomy brak moduły *DNS* pochodzącego z projektu *pydns*[5] (oczywiście w konsoli Pythona importowanie modułu było bezproblemowe). Próba odszukania plików modułu skończyła się fiaskiem: katalog, na który wskazywała dokumentacja był traktowany przez system jako plik binarny. Ostatecznie "pomogło" ściągnięcie najnowszych źródeł ze strony projektu i zainstalowanie ich z użycie pliku setup.py *(sudo python setup.py install)* jednak i tym razem biblioteka okazała się wadliwa - pojawiały się problem z kodowanie znaków, domyślam się zawartych w adresie e-mail. Wiadomości z bugiem wysłana do developerów jest odrzucana przez serwer pocztowy.

Walidacja w akcji.

Czas na nasz kontroler tutaj zabiegi będą czysto kosmetyczne. Na początek przygotujmy naszemu kontrolerowi warsztat i zaimportujmy niezbędne moduły.

```
import logging

from pylons import request, response, session,
tmpl_context as c
from pylons.controllers.util import abort, redirect_to
from pylons.decorators import validate


from darc.lib.base import BaseController, render
#from darc import model
from darc.model.form import RegisterForm

log = logging.getLogger(__name__)

class UsersController(BaseController):
    def index(self):
        # Return a rendered template
        #   return render('/template.mako')
        # or, Return a response
        c.name = "d'Arc"
        return render('users/index.xhtml')

    def register(self):
        return render('users/register.xhtml')

    def create(self):
        c.login = request.POST['login']
        c.email = request.POST['email']
```

5 http://pydns.sf.net/

Z kwestii formalnych: na starcie zaimportowaliśmy dekorator *validate*, a następnie naszą klasę, w której zapisaliśmy informacje o wymaganiach jakie chcemy aby spełniały dane wysyłane przez nasz formularz. To jednak ciut za mało aby wszystko zaczęło współgrać zaczęła działać. Czas na magię:

```
class UsersController(BaseController):
    def index(self):
        # Return a rendered template
        #   return render('/template.mako')
        # or, Return a response
        c.name = "d'Arc"
        return render('users/index.xhtml')

    def register(self):
        return render('users/register.xhtml')

    @validate(schema=RegisterForm(), form="register")
    def create(self):
        c.login = request.POST['login']
        c.email = request.POST['email']
```

I tylko tyle ? Zapytacie. Tak. Tylko tyle. Użyliśmy składni *dekoratora*. Pierwszym parametrem jest szablon, do którego mają pasować dane wysłane przez nasz formularz. Tworzyliśmy go w poprzednim rozdziale. Drugi to nazwa akcji kontrolera, do której należy wrócić kiedy coś się nie powiedzie. Inaczej: nazwa akcji zawierającej formularz. Możesz już przetestować swoje dzieło.

Składnia dekoratorów została wprowadzona w języku Python dopiero po wersji 2.3. Jeżeli chcesz używasz wersji Pythona nie obsługującej dekoratorów zamiast linijki *@validate* wpisz *validate(schema=RegisterForm(), form="register")(create)* **po** deklaracji metody *create*.

Zauważ, że podczas nieudanej walidacji, któregoś z pól, dane w formularzu pozostają na swoim miejscu.

Psujemy dalej czyli XSS w natarciu.

No to się narobiliśmy ! Mysz się nie prześliźnie - mogłoby się wydawać sielanka, fajrant ! Domyślnie w wersji 0.9.7 włączona jest nawet ochrona przed atakami *XSS*. Winowajca nazywa się *default_filters=['escape']* i znajdziemy go w pliku *config/environment.py*.

Zobaczmy co się stanie jeżeli wyłączymy escapowanie. W tym celu wyedtujemy plik widoku *users/create.xhtml*

```
<strong>Login</strong>: ${c.login}<br />
<strong>E-mail</strong>: ${c.email}<br />
<strong>Hasło</strong>: ${c.password | n}<br />
<strong>Potwierdzenie hasła</strong>: $
{c.password_confirmation}
```

Opcja *n* powoduje wyłączenie domyślnego filtru. Teraz wpisz w pole hasła *<script>alert('akuku')</script>*. Nasz tekst zostanie wstawiony bez zamiany znaczników < i > na encje HTML przez co przeglądarka potraktuje go jak kod *JavaScript*. Pojawi się więc okienko z informacją "akuku". Pozostawmy opcję *n* i dodajmy do niej przykładowo h:

```
<strong>Login</strong>: ${c.login}<br />
<strong>E-mail</strong>: ${c.email}<br />
<strong>Hasło</strong>: ${c.password | n,h}<br />
<strong>Potwierdzenie hasła</strong>: $
{c.password_confirmation}
```

Opcja *h* to włączenie opcji esacpowania HTML. Uzyskaliśmy więc taki sam efekt jak na początku. Użycie literki *u* spowoduje bezpieczne wyświetlanie charakterystyczne dla adresów URL, zaś literka *x* przyda nam się gdy będzie potrzebowali filtrować kod XML. Inne przydatne opcje to *trim*, które wycina spacje z początku i końca stringu czy *unicode*, które zwróci obiekt unicode Pythona. Użyteczną opcją może okazać się również *decode.<jakieś kodowanie>*, które zwracając string użyje wskazanego kodowania.

Zawsze escapuj dane. Jeżeli masz wątpliwości w którym miejscu, rób to najbliżej wyjścia (w widoku gdy wyświetlasz dane).

Tworzymy layout.

Zazwyczaj strona posiada jakiś powtarzający się na każdej stronie element. Menu, czy chociażby nagłówki, które generalnie dla każdej strony są takie same. Spróbujmy więc stworzyć przykładowy layout. Niech będzie to plik *templates/layout.xhtml*.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html>
<head>
<title>d'Arc</title>
<meta http-equiv="Content-Type"
content="application/xhtml+xml; charset=utf-8" />
<meta name="author" content="Imię Nazwisko" />
</head>
<body>
${next.body()}
</body>
</html>
```

Poza kodem XHTML, który widać na pierwszy rzut oka wkradł się *${next.body()}*. *${next.body()}* to nic innego jak informacja dla *Mako*, iż życzymy sobie tutaj wstawić treść naszego następnego dokumentu. Już pokazuję jak to działa. Aby to sprawdzić musimy powiedzieć naszym widokom,z którego szablonu mają skorzystać. W tym celu edytujemy kolejno pliki *users/create.xhtml*, *users/index.xhtml* oraz *users/register.xhtml*:

```
<%inherit file="/layout.xhtml"/>

<strong>Login</strong>: ${c.login}<br />
<strong>E-mail</strong>: ${c.email}<br />
<strong>Hasło</strong>: ${c.password | n,entity}<br />
<strong>Potwierdzenie hasła</strong>: $
{c.password_confirmation}
```

```
<%inherit file="/layout.xhtml"/>

<h1>Indeks uzytkowników</h1>
<ul>
  <li>${c.name}</li>
</ul>
```

```
<%inherit file="/layout.xhtml"/>

${h.form('create', method='post')}
  <fieldset>
    <legend>Formularz rejestracyjny</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", id="loginField")}</div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", id="emailField")}</div>
    <div><label for="passwordField">Hasło: </label>$
{h.password("password", id="passwordField")}</div>
    <div><label for="password_confirmationField">PotwierdŹ
hasło: </label>${h.password("password_confirmation",
id="password_confirmationField")}</div>
  </fieldset>
  <div>${h.submit("create", "Zarejestruj")}</div>
${h.end_form()}
```

Proszę bardzo. Znacznik *inherit* na początku każdego z plików poinformuje *Mako* iż chcemy zawrzeć ten plik wewnątrz wskazanego w parametrze. Aby udowodnić, że wszystko gra podejrzyj źródła swojej strony :)

Piszemy testy.

Testowanie aplikacji często bywa traktowane po macoszemu. Być może dlatego, że zostawiane są zawsze na koniec. I rzeczywiście - w małych projektach, pisanych adhoc, w których kod pisze się tylko raz i mają działać - pomysł testowania może okazać się stratą czasu. W dużych przedsięwzięciach okazuje się przydatny. Wyobraź sobie, że masz całkiem rozbudowany system (ze 40 kontrolerów każdy po 7-13 akcji) i wprowadzasz w losowej ich części jakąś nową funkcjonalność. Perspektywa odwiedzania *40 * 7+13/2* stron i sprawdzanie czy wszystko działa ... jest strasznie czasochłonna. Lepiej napisać testy i je "odpalić" - co trwa dużo szybciej. Do sprawdzania poprawności dostajemy ponownie *Paste*[6].

Nasze przykłady oprzemy o jedną metodę, polegającą na sprawdzeniu czy to co zwróciła strona internetowa zawiera jakieś wyrażenie. Czyli czy kod strony posiada jakiś, zadany przez nas wcześniej, fragment. Otwieramy plik *tests/functional/test_users.py*. Kod ten został utworzony przez generator gdy tworzyliśmy kontroler. Czas wyjaśnić co się w nim znajduje:

```
from darc.tests import *

class TestUsersController(TestController):

    def test_index(self):
        response = self.app.get(url(controller='users',
action='index'))
        # Test response...
```

Widzimy tutaj klasę testującą *TestUsersController*. Została wygenerowana automatycznie metoda wywołująca akcję *index*. Nazewnictwo jest proste: test_nazwaakcji, którą testujemy. Fragment *response = self.app.get(url(controller='users', action='index'))* "odwiedza" stronę *http://127.0.0.1:5000/users/index* i kod, który został zwrócony do przeglądarki zapisuje w zmiennej *response*. Mamy więc tam źródło witryny. Najprostszą metodą sprawdzenia czy dostaliśmy to czego oczekujemy jest wykonanie sprawdzenia obecności w źródle dobrze nam znanego fragmentu. W tym przypadku spodziewamy się *<h1>Indeks*.

```
from darc.tests import *

class TestUsersController(TestController):

    def test_index(self):
        response = self.app.get(url(controller='users',
action='index'))
        assert '<h1>Indeks' in response
        # Test response...
```

Proste - prawda ? Nasza nowa linijka sprawdzi czy w zmiennej *response* znajduje się ciąg *<h1>Indeks*. W podobny sposób potraktujmy kolejne elementy naszego serwisu. Najprościej bo w analogiczny sposób możemy napisać metodę dla register:

6 http://pythonpaste.org/testing-applications.html
http://docs.pylonshq.com/testing.html

```
from darc.tests import *

class TestUsersController(TestController):

    def test_index(self):
        response = self.app.get(url(controller='users',
action='index'))
        assert '<h1>Indeks' in response
        # Test response...

    def test_register(self):
        response = self.app.get(url(controller='users',
action='register'))
        assert '<legend>Formularz rejestracyjny</legend>'
in response
```

Oczywiście możesz wybrać inny charakterystyczny dla danej witryny fragment.

To było banalnie proste. Sprawa zaczyna się komplikować przy chęci przetestowania metody *create*. Po pierwsze, jak pamiętamy działa ona wyłącznie dla danych przesłanych z użyciem *POST* poza tym jakoś te dane trzeba przesłać. Pierwszy warunek możemy szybko spełnić, jak łatwo się domyśleć, zamieniając *self.app.get* na *self.app.post*. Do przesłania przykładowych danych wykorzystamy argument *params*.

```
from darc.tests import *

class TestUsersController(TestController):

    def test_index(self):
        response = self.app.get(url(controller='users',
action='index'))
        assert '<h1>Indeks' in response
        # Test response...

    def test_register(self):
        response = self.app.get(url(controller='users',
action='register'))
        assert '<legend>Formularz rejestracyjny</legend>'
in response

    def test_create(self):
        response = self.app.post(url(controller='users',
action='create'), params={'login': 'johny', 'email':
'jan.koprowski@gmail.com', 'password': 'haslo1234',
'password_confirmation': 'haslo1234'})
        assert '<strong>Login</strong>' in response
```

Tym razem wywołaliśmy naszą akcję z użyciem metody *POST*, dodatkowo przekazując z użyciem

słownika, odpowiednie wartości. Przypomnę, że test powinien dać wynik pozytywny. Tak więc przesłane przez nas parametry muszą być "poprawne". Walidacja nadal działa i dla błędnych wartości zwróci nam ponownie formularz - test się nie powiedzie.

Jeżeli chciałbyś testować czy coś się nie powiedzie (np. podając złe dane sprawdzić czy wywołał się formularz) możesz to zrobić stosując słówko *not* np. *assert 'Login' not in response*. Oczywiście istnieje inne metody będące dokładniejsze i mniej podatne na zmiany źródeł witryny.

Zauważ, że w testach nie wykorzystujemy polskich liter. Niestety, podczas próby ich użycia nawet z nagłówkiem *# -*- encoding: utf-8 -*-* otrzymywałem błędy kodera. Jak widać można sobie jednak dać bez nich radę.

Koniec części drugiej.

Na "dziś" to już wszystko. Nauczyłeś się naprawdę sporo. Czas na własne eksperymenty i oswojenie się z nabytą wiedzą. Warto poszerzyć informacje o metodzie pozyskiwania danych użytecznych przy prowadzeniu testów. Linki podałem w stopce na odpowiedniej stronie. Powodzenia !

Licencja

Spis treści