${Pylons} 0.9.7 - Przewodnik część 4

Menu

Aby móc wygodnie korzystać z dostępnych opcji przydałoby się jakieś menu. Podzielimy nasz dokument na dwie szpalety. Po lewej umieścimy listę o szerokości stu pięćdziesięciu pikseli, po prawej zawartość witryny.

Jeszcze kilka helperów

Na stracie przyda się garść funkcji pomocniczych. Zmieńmy odrobinę plik *lib/helpers.py*:

```
from webhelpers.html.tags import form, file, text,
textarea, password, image, submit, end_form,
stylesheet_link, javascript_link, link_to
from webhelpers.rails.secure_form_tag import
secure_button_to
```

Do tworzenia menu i generowania w nim linków użyteczny będzie *link_to*. Na przyszłość zaimportujemy również helper *textarea*. Przyda się do stworzenia pola dla *about*. *Button_to* to bardzo użyteczne narzędzie, które potrafi tworzyć formularze zawierające wyłącznie jeden guziczek. Idealny kandydat na na przykład "usuń konto". *Image* i *file* to już wcześniejsze przygotowania do obsługi awatarów.

Użyty *secure_button_to* ma status deprecated i przestanie być dostępny w kolejnych wersjach *Pylons*. Obecnie jednak jesteśmy zmuszeni z niego korzystać ponieważ podczas przeportowywania starej funkcjonalności przepisanie tej funkcji przeoczono.

Czas na menu

Naszą listę stworzymy w osobnym pliku. W zależności od tego czy użytkownik będzie zalogowany czy nie wyświetlać będziemy różne pozycje. Umówmy się na plik *templates/menu.xhtml*.

```
<ul style="float: left; width: 150px;">
    % if 'user' in session and session['user']['id']:
        <li>${h.link_to('Moje konto',
'/users/index')}</li>
        <li>${h.link_to('Wyloguj', '/users/signout')}</li>

    % else:
        <li>${h.link_to('Rejestracja',
'/users/register')}</li>
        <li>${h.link_to('Zaloguj', '/users/signout')}</li>
    % endif
</ul>
```

Na początku dodaliśmy fragment kodu CSS ustawiający menu po lewej stronie i nadający mu

cc BY NC ND

szerokość stu pięćdziesięciu pikseli. Dziwny może wydawać się warunek sprawdzający czy użytkownik jest zalogowany. Jest ułożony w taki sposób, aby pierwszy z nich nie wyświetlał błędu. Python jest na tyle "sprytny" iż widząc spójnik *and* kończy sprawdzanie kolejnych warunków gdy napotka pierwszy niespełniony.

Nasze menu w layoucie

```
{...}
<body>
    <%include file="/menu.xhtml" />
    <div style="clear: right;">
    <% messages = h.flash.pop_messages() %>
    % if messages:
    <ul id="flash-messages">
    % for message in messages:
        <li>${message}</li>
    % endfor
    </ul>
    % endif

    ${next.body()}
    </div>
  </body>
</html>
```

Użyliśmy tutaj dyrektywy *include* jaką udostępnia *Mako* w celu zamieszczenia pliku. Objęcie pozostałej części znacznikiem *div* służy ułożeniem treści strony w prawej kolumnie.

Panel użytkownika

Mamy już menu, działający system rejestracji użytkowników i logowania. Wszystko "pięknie". Przydałby się jednak jeszcze jakiś panel użytkowników umożliwiający wpisanie czegoś w pole about, zmianę e-maila, hasła czy ustawienie avataru. Pójdzie nam jak spłatka.

Formularz

Na starcie stwórzmy formularz użytkownika. Najlepszym kandydatem do edycji jest plik *templates/ users/index.xhtml*.

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

${h.secure_form('/users/update', method='post')}
  <fieldset>
    <legend>Aktualizacja danych</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", c.user.login, id="loginField",
readonly="readonly")}</div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", c.user.email,  id="emailField")}</div>
    <div><label for="aboutField">About: </label>$
{h.textarea("about", c.user.about, cols=25, rows=10,
id="aboutField")}</div>
  </fieldset>
  <div>${h.submit("update", "Aktualizuj dane")}</div>
${h.end_form()} ${h.secure_button_to(u'Usuń konto',
'/users/delete') | n}

${h.secure_form('/users/changePassword', method='post')}
  <fieldset>
    <legend>Zmiana hasła</legend>
<div><label for="currentPasswordField">Aktualne hasło:
</label>${h.password("current_password",
id="currentPasswordField")}</div>
    <div><label for="passwordField">Nowe hasło: </label>$
{h.password("password", id="passwordField")}</div>
    <div><label for="password_confirmationField">Potwierdź
nowe hasło: </label>${h.password("password_confirmation",
id="password_confirmationField")}</div>
  </fieldset>
  <div>${h.submit("update", u"Zmień hasło")}</div>
${h.end_form()}
```

Widać tutaj dwie części naszego panelu. Pierwsza z nich umożliwia zmianę *emaila* oraz uzupełnienia pola *about*. Aby wartość *login* uczynić z jednej strony obecnym, zaś z drugiej już na poziomie interfejsu uniemożliwić zmianę jej zawartości opatrzyłem je atrybutem *readonly*. Działa on podobnie do *disable* jednak w przeciwieństwie do niego daje on komfort kopiowania zawartości do schowka. Dalej rzuca się nam w oczy formularz do zmiany hasła. Pomiędzy ukrył się guziczek służący do usuwania konta. Jeżeli spojrzysz w źródła jest to kolejny formularz, z akcją ustawioną na drugi parametr. Z tajników powyższego widoku to już wszystko. Czas na działanie. Na początek zabierzemy się za usuwanie użytkowników.

W obecnym *Pylons* zabrakło niestety generatora dla *secure_button_to*. Używany przez nas, z modułu rails ma już status *deprecated*. Zwracana przez niego zawartość jest niestety escapowana. W celu uniknięcia problemów potrzebne było użycie *"| n"*.

Zmiany w modelu

Czas przygotować nasz model. Do tej pory uzyskując instancję klasy user zawsze prosiliśmy o

obiekt, nie powiązany z żadnym z rekordów. Było to wystarczające dla stworzenia nowego użytkownika czy też sprawdzenia czy pole o zadanych kryteriach istnieje w tabeli. Teraz jednak przydałoby się nam narzędzie umożliwiające dostęp do użytkownika o konkretnym *id*. Po krótkich przemyśleniach dojdziesz do wniosku, że logiczne wydaje się tutaj stworzenie metody statycznej, która zwróci nam obiekt konkretnego użytkownika. Python jako język wspierający metaprogramowanie posiada możliwość manipulowania obiektem, który jest zwracany podczas tworzenia jego nowej instancji. Cała sztuczka tkwi w implementacji metody *__new__*. Oto kod, który należy dodać do klasy modelu.

```
class User(object):
{...}
    def __new__(cls, id=None):
        if id:
            return meta.Session.query(User).get(id)
        else:
            return object.__new__(cls)

    def __init__(self, id=None):
        if id:
            self.id = id
{...}
```

Tworząc obiekt będziemy mogli teraz przekazać mu jako parametr *id* użytkownika, którego chcemy pobrać. Metoda *__init__* została zaimplementowana ponieważ wywołanie *__new__* w przypadku zwrócenia obiektu klasy *User* wywołuje automatycznie *__init__* z tą samą listą parametrów. Dodajemy jeszcze jedną metodę, tym razem usuwającą konkretnego użytkownika.

```
class User(object):
{...}
    def delete(self):
        meta.Session.delete(self)
        meta.Session.commit()
{...}
```

Kiedy wszystko jest już przygotowane - możemy zająć się kontrolerem.

Usuwanie użytkownika.

Od teraz schemat wszystkich działań będzie bardzo prosty. Pobieramy użytkownika i wykonujemy na nim jakieś działanie. Najwyższa pora zobaczyć jak będzie to wyglądało w praktyce:

cc BY NC ND

```
class UsersController(BaseController):
{...}

    @authorize(AuthenticatedUser())
    @authenticate_form
    def delete(self):
        user = User(session['user']['id'])
        user.delete()

        return self.signout()
{...}
```

Rozwiązanie jest wręcz banalne i bardzo czytelne. Na początku pobieramy użytkownika o zadanym *id* a następnie wywołujemy metodę *delete*. Ponieważ usera już nie ma wywołujemy akcję wylogowania. Można było to zrobić poprzez *redirect_to* jednak chciałem pokazać, że kontroler to też zwykła klasa :) i nie zawsze musimy wszystko robić z użyciem protokołu HTTP. W obecnym kodzie można byłoby dodatkowo zabezpieczyć się i spróbować wyłapać wyjątki odpowiedzialne za kasowanie użytkownika o nie istniejącym *id.* Ogromne ułatwienie uzyskujemy dzięki zastosowaniu metody __*new()*__ naszego modelu.

Jeżeli to kogoś interesuje dekoratory przetwarzane są po kolei z góry na dół. Najpierw sprawdzamy czy ktoś jest zalogowany. Jeżeli tak sprawdzamy czy nie jest to atak CSRF.

Zmiana hasła

Jak to zrealizować ?

Aby zaplanować działanie naszego modelu warto wiedzieć coś więcej o działaniu *SQLAlchemy*. Tak jak w przypadku tworzenia obiektów wymagane było wywołanie kolejno *meta.Session.save(obiekt)*, *meta.Session.commit(obiekt)*, a przy usuwaniu *meta.Session.delete(obiekt)*, *meta.Session.commit(obiekt)*, tak w przypadku uaktualniania (a czymże innym jest zmiana hasła) danych wystarcza po podmianie wartości pól samo *commit*. Jeżeli dobrze się przyjrzysz jest to metoda, która czyni naszą bazę aktualną (up to date), dodatkowo pojawia się tutaj już trzy razy (tworzenie, usuwanie - za moment zmiana hasła) i pojawi się czwarty (zmiana pól *email* i *about*). Jest to znak iż być może warto byłoby wydzielić temu poleceniu osobną, publiczną metodę. Jak się okazuje jest to dobry pomysł. Czas na małą restrukturyzację.

cc BY NC ND

Zmiany w modelu

```
class User(object):

    def update(self):
          meta.Session.commit()

    def save(self):
        meta.Session.save(self)
        try:
            self.update()
        except sa.exc.IntegrityError:
          raise LoginOrEmailExistsException();

    def delete(self):
        meta.Session.delete(self)
        self.update()
```

Obecnie nasz model udostępnia nam już trzy metody. Do zapisywania nowego obiektu, oraz usuwania i uaktualniania istniejącego.

Kontroler

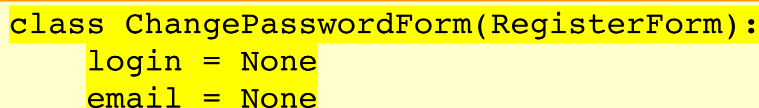
```
class UsersController(BaseController):
{...}

    @authorize(AuthenticatedUser())
    @authenticate_form
    def changePassword(self):
        user = User(session['user']['id'])
        user.password = request.POST['password']
        user.update()

        flash('Twoje hasło zostało zmienione.')

        return redirect_to(action='index')
{...}
```

W gruncie rzeczy jest to powtórzony schemat metody *delete* i wykorzystanie naszej zaminy polegającej na wyposażeniu kontrolera w możliwość wywołania *update()*. Coś tu jednak nie pasuje. Czy nic nie wzbudziło Twojego niepokoju ? Zastanów się.

Walidacja

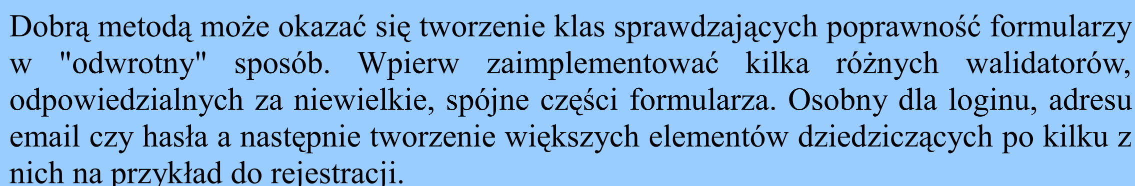
Gdybyśmy nie stworzyli formularza walidacji po zarejestrowaniu się istniałaby możliwość ustawienia hasła nie zgodnego z początkowymi regułami. Przy klasie *RegisterForm* i regułach w niej ustalonych, regulowanie tych samych fragmentów w innych formularzach to banał. W tym przypadku warto wykorzystać dziedziczenie. Naszym przodkiem może być *RegisterForm*. W takim razie plik *model/form.py* wzbogaci się o nową klasę.

cc BY NC ND



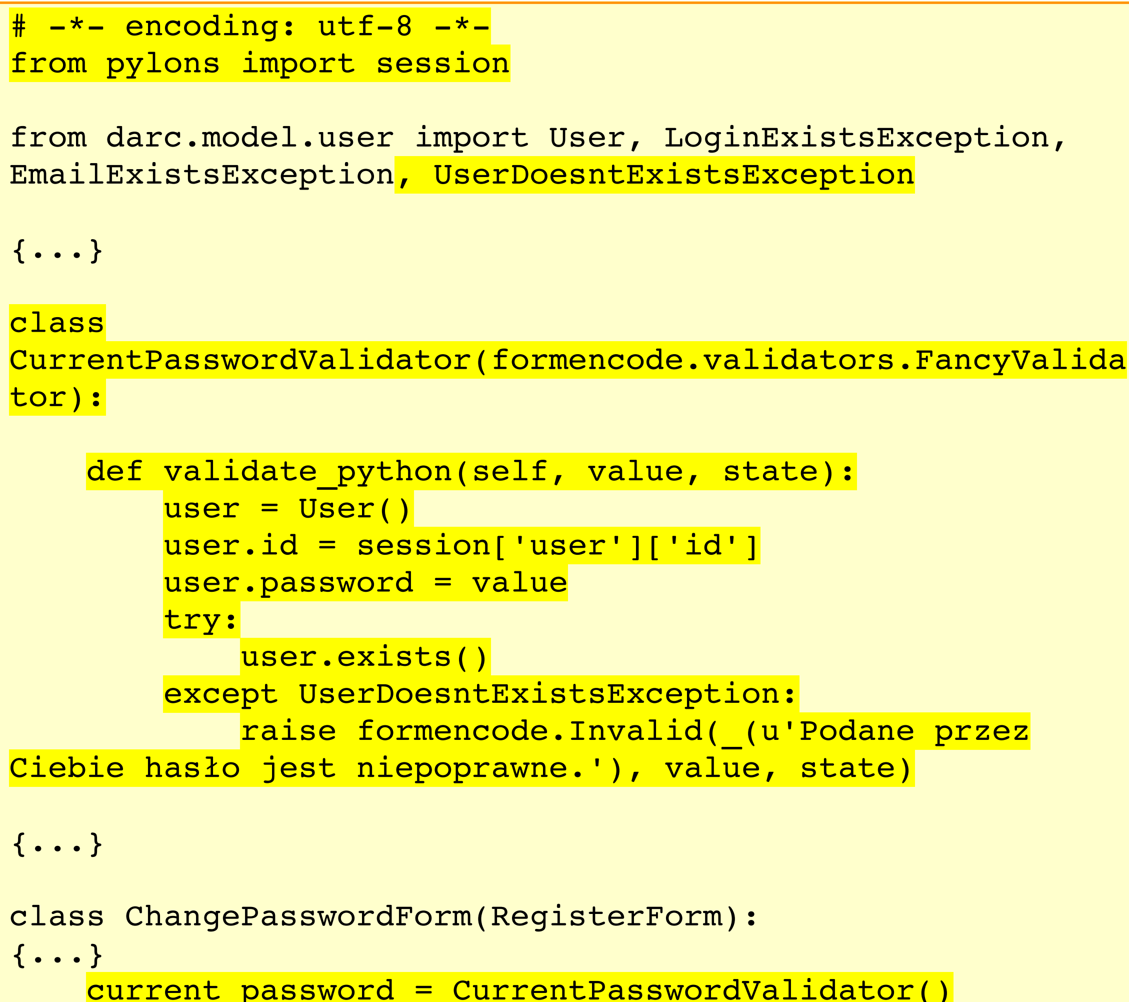
```
class ChangePasswordForm(RegisterForm):
    login = None
    email = None
```

Dzięki zastosowaniu dziedziczenia, w przypadku zmiany minimalnej długości hasła będziemy musieli dokonać modyfikacji wyłącznie w jednym miejscu - i automatycznie rozpropaguje się ona dalej. Minusem jest jak widać konieczność ignorowania pól *login* i *email*, które jak widać musimy wyłączyć. W momencie dodania kolejnych atrybutów w formularzu rejestracji jesteśmy zmuszeni do modyfikacji innych klas w celu ich ignorowania.



Dobrą metodą może okazać się tworzenie klas sprawdzających poprawność formularzy w "odwrotny" sposób. Wpierw zaimplementować kilka różnych walidatorów, odpowiedzialnych za niewielkie, spójne części formularza. Osobny dla loginu, adresu email czy hasła a następnie tworzenie większych elementów dziedziczących po kilku z nich na przykład do rejestracji.

Brakuje jeszcze sprawdzenia, czy podane obecnie hasło jest poprawne. Przyda nam się do tego klasa. Podamy wartości odpowiednich pól i wywołamy metodę *exists*. Jeżeli użytkownik nie zostanie znaleziony - oznacza to iż hasło zostało podane błędnie.



```
# -*- encoding: utf-8 -*-
from pylons import session

from darc.model.user import User, LoginExistsException,
EmailExistsException, UserDoesntExistsException

{...}

class
CurrentPasswordValidator(formencode.validators.FancyValida
tor):

    def validate_python(self, value, state):
        user = User()
        user.id = session['user']['id']
        user.password = value
        try:
            user.exists()
        except UserDoesntExistsException:
            raise formencode.Invalid(_(u'Podane przez
Ciebie hasło jest niepoprawne.'), value, state)

{...}

class ChangePasswordForm(RegisterForm):
{...}
    current_password = CurrentPasswordValidator()
```


cc BY NC ND

Skoro wszystko jest już gotowe sprawmy aby do kontrolera nie prześliznęły się hasła niezgodne z naszymi restrykcjami.

```
from darc.model.form import RegisterForm, ChangePasswordForm

class UsersController(BaseController):
{...}
    @authorize(AuthenticatedUser())
    @authenticate_form
    @validate(schema=ChangePasswordForm(), form="index")
    def changePassword(self):
{...}
```

Rozsądnym byłoby zaimplementowanie sprawdzanie obecnego hasła we wszystkich pozostałych formularzach.

Zmiana danych użytkownika

Walidacja

Pozostał nam jeszcze jeden formularz. Tym razem zaczniemy od razu od walidacji. Ponieważ używamy tutaj wyłącznie jednego pola występującego w rejestracji dziedziczenie jest nie warte zachodu - więcej narobimy się przypisując do pozostałych atrybutów wartości *None*. Dodatkowo, aby atrybut pola *about* był zachowywany stworzymy dla niego jakąś neutralną regułkę.

```
class UpdateForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    email = formencode.All(formencode.validators.Email(not_empty=True), EmailExistsValidator())

    about = formencode.validators.String()
```

Kontroler

Będzie to właściwie to samo działanie co w przypadku zmiany hasła. Różnica polega wyłącznie na fakcie zmiany innych pól.

```
from darc.model.form import RegisterForm, UpdateForm,
ChangePasswordForm

class UsersController(BaseController):
{...}
    @authorize(AuthenticatedUser())
    @authenticate_form
    @validate(schema=UpdateForm(), form="index")
    def update(self):
        user = User(session['user']['id'])
        user.email = request.POST['email']
        user.about = request.POST['about']
        user.update()

        flash('Twoje dane zostały pomyślnie
zmodyfikowane.')

        return redirect_to(action='index')
```

Uzupełniamy formularz danymi

Teraz kiedy zalogujesz się do systemu zauważysz, że niestety formularz jest pusty. Jednym z wielu pomysłów jest przekazanie do widoku całego obiektu klasy *User*. Tak więc zróbmy.

```
class UsersController(BaseController):
{...}
    @authorize(AuthenticatedUser())
    def index(self):
        # Return a rendered template
        #   return render('/template.mako')
        # or, Return a response
        c.user = User(session['user']['id'])

        return render('users/index.xhtml')
```

Zmienna *c.name* nie będzie nam już więcej potrzebna. Wprowadźmy jeszcze dane do widoku:

CC BY NC ND

```
${h.secure_form('/users/update', method='post')}
  <fieldset>
    <legend>Aktualizacja danych</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", c.user.login, id="loginField",
readonly="readonly")}</div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", c.user.email,  id="emailField")}</div>
    <div><label for="aboutField">About: </label>$
{h.textarea("about", c.user.about, cols=25, rows=10,
id="aboutField")}</div>
  </fieldset>
  <div>${h.submit("update", "Aktualizuj dane")}</div>
${h.end_form()} ${h.button_to(u'Usuń konto',
'/users/delete')}
```

Problemy z e-mailem

Czas przetestować nasze dokonania. O ile zmiana hasła oraz usuwanie konta działają bezproblemowo o tyle pierwszy od góry formularz sprawia pewne problemy. Kiedy pozostawimy wpisaną wartość e-mail zostaniemy poczęstowani komunikatem o błędzie iż podany e-mail istnieje w bazie danych. Ale ... ale jak to ... no to oczywiste, że istnieje - przecież to mój e-mail. Przeanalizujmy jak to działa. Jeżeli w polu formularza pozostawimy wartość pobraną z bazy danych walidator sprawdzi czy w tabeli występuje taki e-mail. Oczywiście dostanie wiadomość iż istnieje - no bo istnieje. Problem polega na tym, że nasz walidator nie umie odróżnić naszego e-maila od emaila innej osoby. Aby poprawić ten błąd stwórzmy walidator, który najpierw sprawdzi czy w bazie danych istnieje podany przez nas adres - jeżeli tak dodatkowo sprawdzi czy to nasz email.

```
class EmailEngagedValidator(EmailExistsValidator):

    def validate_python(self, value, state):
        try:
            super(EmailEngagedValidator,
self).validate_python(value, state)
        except formencode.Invalid, superException:
            user = User(session['user']['id'])
            if value != user.email:
                raise superException

class UpdateForm(formencode.Schema):
{...}
    email =
formencode.All(formencode.validators.Email(not_empty=True)
, EmailEngagedValidator())
```

Aby wykorzystać to co już zostało zrobione klasa dziedziczy po *EmailExistsValidator*. W metodzie walidacji wykonujemy test na obecność emaila w bazie danych (metodę o tej samej nazwie naszego rodzica). Jeżeli został zwrócony wyjątek *Invalid* oznacza to iż mail w bazie danych istnieje. Pobieramy obiekt tego wyjątku do zmiennej *superException*. Teraz naprawiamy błąd - skoro mail

cc BY NC ND

istnieje sprawdzamy czy nie jest to przypadkiem nasz adres. Sprawdzamy dane użytkownika aktualnie zalogowanego a następnie porównujemy pole *email* z wprowadzoną wartością. Jeżeli są różne oznacza to iż próbujemy zmienić e-mail na adres innego użytkownika istniejącego już w bazie danych. Jako wyjątek zgłaszamy ten same obiekt, który stworzył walidator *EmailExistsValidator (superException)*. Teraz wszystko będzie działało poprawnie :)

Uploadowaniu pliku

Ładowanie plików na serwer w Pylons[1] odbywa się w bardzo prosty sposób. Aby uzyskać pole odpowiedniego typu wystarczy użyć helpera *file*. Musimy tylko pamiętać aby formularzowi, w którym je umieścimy ustawić wartość *multipart* na *True*. W kontrolerze odbieramy obiekt typu *cgi.FieldStorage*[2]. W celu zapisania na dysk uploadowanego obiektu musimy otworzyć pusty deskryptor pliku i przekopiować do niego otrzymane dane z użyciem funkcji *copyfileobj* z modułu *shutil*. Przy używaniu uploadowanych obiektów warto wiedzieć o ich czterech podstawowych atrybutach

- **file** - obiekt podobny do typu *file* reprezentujący uploadowany plik
- **filename -** nazwa pliku
- **type -** typu zawartości
- **value -** zawartość przesyłanego pliku

Pole uploadu pliku

Dodajmy do naszego widoku *users/index.xhtml* formularz do obsługi awatarów.

```
${h.secure_form('/users/uploadAvatar', method='post',
multipart=True)}
  <fieldset>
    <legend>Ładowanie awataru</legend>
    <div><label for="avatarField">Obrazek: </label>$
{h.file("image", id="avatarField")}</div>
  </fieldset>
  <div>${h.submit("update", u"Zmień avatar")}</div>
${h.end_form()} ${h.secure_button_to(u'Usuń avatar',
'/users/removeAvatar') | n}
```

Skorzystaliśmy z helpera *file* - co automatycznie implikuje dodania *multipart=True* przy tworzeniu formularza. Jak widać istnieje też guzik służący do usuwania avataru. Zacznijmy jednak od uploadu, od stworzenia klasy *Avatar*.

Avatar

Będziemy tworzyć ją kroczek po kroczku. Zaimplementujmy mechanizm samego uploadu.

1 http://docs.pylonshq.com/forms.html#id2
2 http://docs.python.org/library/cgi.html

```
import os
import shutil

class Avatar(object):

    __base = os.path.join(os.path.sep, 'home', 'username',
'Pylons', 'darc', 'darc', 'public')
    __directory = os.path.join(os.path.sep, 'uploads',
'user', 'avatars', '')

    def __init__(self, id=None):
        self.id = id

    def upload(self, image):
      ext = os.path.splitext(image.filename)[1]
      path = self.__base + self.__directory + str(self.id)
+ ext.lower()

      avatar = open(path, 'w')
      shutil.copyfileobj(image.file, avatar)
      image.file.close()
      avatar.close()
```

Nasza klasa (.pl :P) potrafi już uplodować pliki. Pierwsze dwa prywatne pola to nic innego jak ścieżki do katalogu. Po stronie systemu musimy posługiwać się pełnymi, bezwzględnymi ścieżkami. Dzięki wykorzystaniu *os.path.join* nasza klasa będzie bardziej przenośna i to czy w ścieżce obecne będą backslahse (/) czy slashe (\) będzie zależało od systemu operacyjnego. Obecność *__init__()* jest tutaj niezbędna w celu przypisania do obiektu id konkretnego użytkownika.

Na początku metody upload pobieramy rozszerzenie naszego pliku i budujemy do niego ścieżkę. Plik przybierze nazwę *"id_użytkownika.rozszrzenie"*. Dzięki temu będziemy mogli błyskawicznie przyporządkować awatar do konta. Zapiszmy nasze dzieło jako *avatar.py* w folderze *model* i przejdźmy dalej.

Ponieważ w powyższym przykładzie użyta jest ścieżka bezwzględna musisz zmienić zmienną *__base* i dopasować ją do realiów swojego systemu.

Tworzymy katalog

Brakuje nam jeszcze folderu :) w którym będą trzymane pliki użytkowników. Oczywiście gdzieś w public. Osobiście pomyślałem, że nie będzie złym pomysłem wydzielenie katalogu *uploads* a w nim stworzenie *user/avatars*. Pozwoli to na przyszłość zachować porządek.

```
mkdir -p ~/Pylons/darc/darc/public/uploads/user/avatar
```


CC BY NC ND

Obsługa w kontrolerze

```
from darc.model.avatar import Avatar

class UsersController(BaseController):
{...}
    @authorize(AuthenticatedUser())
    @authenticate_form
    def uploadAvatar(self):
        avatar = Avatar(session['user']['id'])
        avatar.upload(request.POST['image'])

        flash('Twój avatar został załadowany.')

        return redirect_to(action='index')
```

Na starcie tworzymy obiekt, z id zalogowanego użytkownika a następnie wywołujemy metodę upload :) Spróbuj wgrać nasz pierwszy obrazek. Wszystko powinno zadziałać poprawnie.

Wyświetlamy nasz avatar

Obecnie o tym iż zdjęcie załadowało się na serwer musimy uwierzyć na słowo. Nie możemy go jednak jeszcze wyświetlić. Nic prostszego. Dodajmy kilka metod do naszego modelu:

```
import os
import glob
import shutil


class Avatar(object):

{...}

    def getPath(self):
        try:
            path = glob.glob(self.__base + 
self.__directory + str(self.id) + '.*')[0]
            return path
        except IndexError:
            return ''

    def getUrl(self):
        path = self.getPath()
        return path.replace(self.__base, '')

    def __str__(self):
        return self.getUrl()

    def exists(self):
        return self.getPath() != ''
```

Mamy tu już praktycznie wszystko co potrzeba. *getPath()* pobiera listę plików zaczynających się od zadanego id. Na początku nie znamy rozszerzenia pliku jaki jest przypisany użytkownikowi. Może być to png, jpg czy gif. Pobieramy więc wszystkie pliki, których nazwa to *id* konta a następnie wybieramy pierwszy. Funkcja *glob.glob* zwraca nam krotkę, której kolejne elementy to kolejne dopasowania (może być więcej niż jedno). Wybieramy pierwszą - jeżeli nie istnieje zwracamy pusty string (pliku nie ma). W *getUrl()* obcianamy człon *__base* naszej ścieżki. Zauważ, że po stronie przeglądarki będzie on automatycznie zastąpiony adresem strony. Sensowne staje się stworzenie również metody *__str__()*, która zostanie wywołana podczas gdy nasz obiekt będzie konwertowany na *string*. Umownie możemy przyjąć, że będzie to ścieżka do naszego pliku - u nas bardziej użyteczne okaże się zwrócenie w tym momencie wyjścia funkcji *getUrl()*. E*xists* to właściwie nazwanie tego co już jest. Aby upewnić się czy plik istnieje najprościej jest sprawdzić czy *getPath()* nie zwróciło pustego stringu. Wykorzystajmy to teraz w widoku. *users/index.xhtml* wzbogaci się o następujący fragment:

```
    <legend>Ładowanie awataru</legend>
    % if c.avatar.exists():
      ${h.image(c.avatar, c.user)}
    % endif
    <div><label for="avatarField">Obrazek: </label>$
{h.file("image", id="avatarField")}</div>
```

Użycie metody *exists* jest tutaj dodane wyłącznie dla czytelności kodu. *c.avatar* w tym kontekście wywoła *__str__()* i będzie równoważne z *str(c.avatar)*.

CC BY NC ND

Po uploadowaniu obrazka prawdopodobnie będziesz musiał odświeżyć stronę - aby zobaczyć zmiany w widoku.

Brakuje jeszcze obsług naszym kontrolerze. Oto ona:

```
class UsersController(BaseController):

{...}

    @authorize(AuthenticatedUser())
    def index(self):
        # Return a rendered template
        #   return render('/template.mako')
        # or, Return a response
        c.user = User(session['user']['id'])
        c.avatar = Avatar(session['user']['id'])
        return render('users/index.xhtml')
```

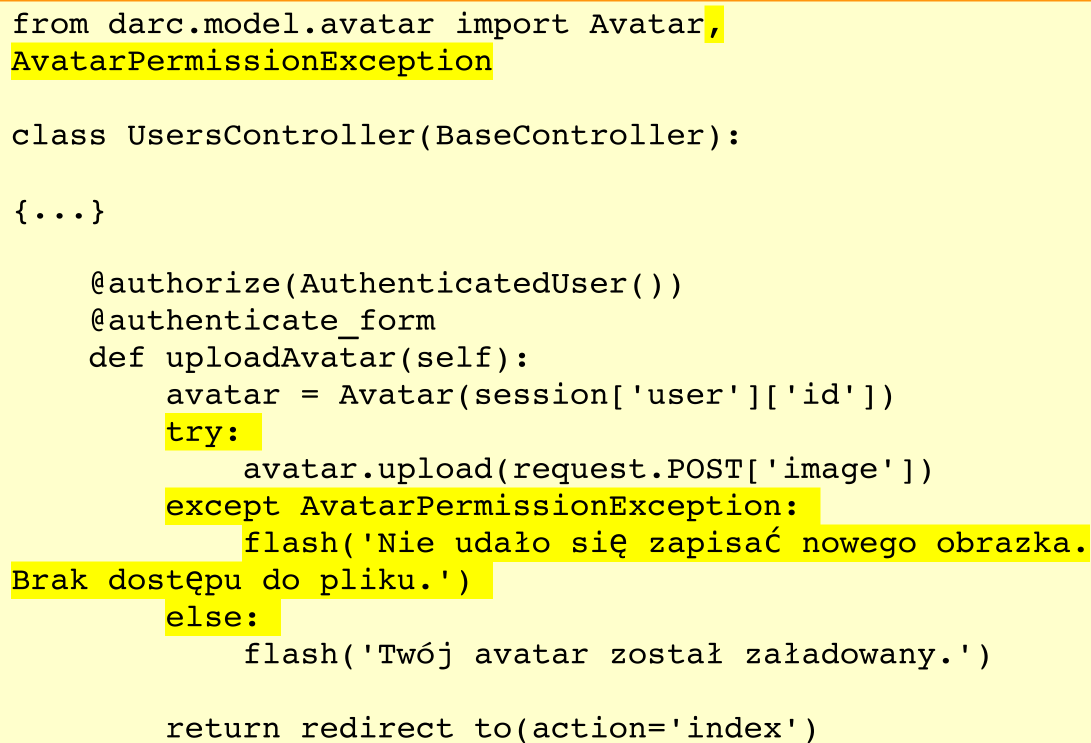
Wiele rozszerzeń - wiele plików

No tak - a co jeżeli najpierw uploadujemy plik *obrazek.jpg* a potem kolejny *obrazek.png*. Dla użytkownika o id przykładowo *7776* powstaną dwa pliki *7776.jpg* i *7776.png*. Jak temu zaradzić ? Najprostszą metodą jest przed samym zapisaniem nowego pliku wykasowanie poprzedniego lub wszystkich o danym id. Przy obrazach o różnych rozszerzeniach to jedyne wyjście. Ulepszmy więc odrobinę nasz obiekt:

cc BY NC ND

```
class AvatarPermissionException(Exception):
    pass

{...}

class Avatar(object):

{...}

    def removeAll(self):
        files = glob.glob(self.__base + self.__directory +
str(self.id) + '.*')
        try:
            for file in files:
                os.remove(file)
        except OSError:
            raise AvatarPermissionException()

    def upload(self, image):
      ext = os.path.splitext(image.filename)[1]
      path = self.__base + self.__directory + str(self.id)
+ ext.lower()

      self.removeAll()

      try:
          avatar = open(path, 'w')
          shutil.copyfileobj(image.file, avatar)
          image.file.close()
          avatar.close()
      except IOError:
            raise AvatarPermissionException()
```

Przy okazji dodałem troszkę więcej rzeczy. Przede wszystkim pojawiła się w klasie metoda, która usuwa wszystkie pliki, których nazwą jest zadane *id* użytkownika. Wywołujemy ją zaraz przed skopiowaniem zawartości uploadowanego pliku do katalogu. Doszła obsługa wyjątków. Otóż może się okazać, że plik lub folder nie posiadają odpowiednich uprawnień aby zapisać lub usunąć obrazek. Wtedy warto zwrócić odpowiedni wyjątek i poinformować o tym użytkownika. Musimy więc jeszcze ciut zmienić zachowanie się kontrolera.

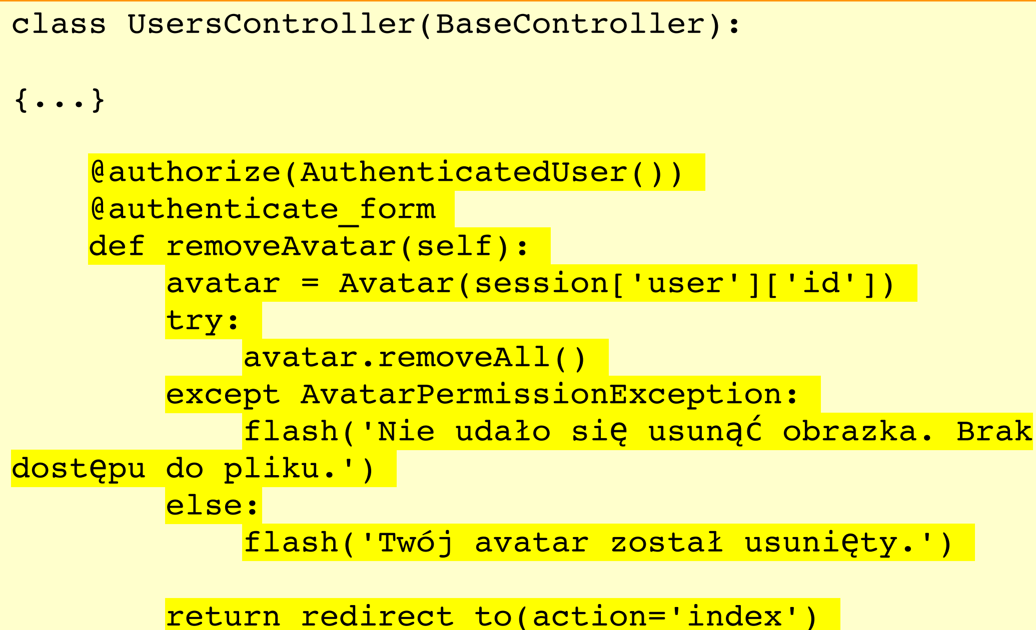


```
from darc.model.avatar import Avatar,
AvatarPermissionException

class UsersController(BaseController):

{...}

    @authorize(AuthenticatedUser())
    @authenticate_form
    def uploadAvatar(self):
        avatar = Avatar(session['user']['id'])
        try:
            avatar.upload(request.POST['image'])
        except AvatarPermissionException:
            flash('Nie udało się zapisać nowego obrazka.
Brak dostępu do pliku.')
        else:
            flash('Twój avatar został załadowany.')

        return redirect_to(action='index')
```

Wyjątek został obsłużony :)

Usuwanie plików.

Jeżeli zwróciłeś uwagę posiadamy już metodę, która pozwala usunąć obrazki dla zadanego *id*. Skoro od zaimplementowania usuwania avataru dzieli nas już tylko krok - dlaczego by nie zrobić tego teraz. Wystarczy, że dodamy jedną akcję.



```
class UsersController(BaseController):

{...}

    @authorize(AuthenticatedUser())
    @authenticate_form
    def removeAvatar(self):
        avatar = Avatar(session['user']['id'])
        try:
            avatar.removeAll()
        except AvatarPermissionException:
            flash('Nie udało się usunąć obrazka. Brak
dostępu do pliku.')
        else:
            flash('Twój avatar został usunięty.')

        return redirect_to(action='index')
```

I to właściwie tyle. Pobieramy obiekt o zadanym id i wywołujemy *removeAll()*.

cc BY NC ND

Pliki na serwerze - sprawdź zanim zapiszesz

Nasza obsługa awatarów nie doczekała się jeszcze walidacji formularza. Co należałoby sprawdzić ? Po pierwsze obrazek nie powinien być zbyt wielki (powiedzmy 200x200), rozmiar pliku powinien być również rozsądny - niech będą dwa mibibity. Dodatkowo warto sprawdzić typ pliku. Rozsądnym minimum wydaje się zezwolenie na formaty *png*, *gif* oraz *jpg*. Na starcie przegotujmy sobie klasy walidatorów. Umieszczamy je oczywiście w *model/form.py*.

Sprawdzanie typu pliku

Zacznijmy od upewnienia się czy nasz kandydat na awatar jest w ogóle obrazkiem. W tym celu możemy użyć pola *type* i sprawdzić czy jego wartość mieści się w zbiorze dozwolonych, przekazanych do naszego walidatora wartości:

```
class
MimeTypeValidator(formencode.validators.FancyValidator):

    __unpackargs__ = ('allowed_types',)

    def validate_python(self, value, state):
        if value.type not in self.allowed_types.values():
            raise formencode.Invalid(u'Dozwolone rodzaje
plików to: ' + ', '.join(self.allowed_types.keys()) + '.',
value, state)

{...}

class AvatarForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    image =
formencode.All(MimeTypeValidator(allowed_types={'jpg':
'image/jpeg', 'gif': 'image/gif', 'png': 'image/png'}))
```

Parametry do naszych walidatorów przekazuje się definiując je jako krotkę *__unpackargs__*. Zbiór dozwolonych parametrów to słownik, którego kluczami są rozszerzenia plików (użyte w komunikacie błędu) zaś wartościami typy mime. Wystarczy teraz sprawdzić czy pośród nich znajduje się zwracany nagłówek naszego pliku - jeżeli nie wyrzucamy wyjątek.

Wymiary obrazka

Do sprawdzenia wymiarów obrazka potrzebna będzie nam biblioteka *PIL*[3]. Prawdopodobnie znajduje się już w Twoim systemie. Z jej użyciem potraktujemy naszą zawartość pliku jako obrazek i pobierzemy jego wymiary. Następnie porównamy z maksymalnymi, dozwolonymi przez nas.

3 http://www.pythonware.com/products/pil/

cc BY NC ND

```
import IOString

from PIL import Image

{...}

class
ImageSizeValidator(formencode.validators.FancyValidator):

    __unpackargs__ = ('mx_dimensions',)

    def validate_python(self, value, state):
        max_width, max_height = self.max_dimensions

        img = Image.open(StringIO.StringIO(value.value))
        width, height = img.size

        if width > max_width or height > max_height:
            raise formencode.Invalid(u'Maksymane dozwolone
wymiary obrazka to: ' +
'x'.join(map(str,self.max_dimensions)) + ' .', value,
state)

{...}

class AvatarForm(formencode.Schema):

{...}

    image =
formencode.All(MimeTypeValidator(allowed_types={'jpg':
'image/jpeg', 'gif': 'image/gif', 'png': 'image/png'})) ,
ImageSizeValidator(max_dimensions=(200,200)))
```

Rozmiar obrazka

Badanie rozmiaru obrazka oprzemy o metodę zaproponowaną w dokumentacji Pylons. Poleganie na tym co zwróci nam funkcja *len()* wywołana na zawartości pliku powinna być wystarczająco dokładna.

```
class
FileSizeValidator(formencode.validators.FancyValidator):

    __unpackargs__ = ('max_size',)

    def validate_python(self, value, state):
        if len(value.value) > self.max_size:
            raise formencode.Invalid(u'Maksymalny
dozwolony rozmiar pliku to: ' + str(self.max_size) + ' .',
value, state)

class AvatarForm(formencode.Schema):

{...}

    image =
formencode.All(MimeTypeValidator(allowed_types={'jpg':
'image/jpeg', 'gif': 'image/gif', 'png': 'image/png'})) ,
ImageSizeValidator(max_dimensions=(200,200)),
FileSizeValidator(max_size=(8 * 1024 * 2)))
```

Zachowywanie wartości formularza pliku

To ostatnia rzecz, o której chciałbym wspomnieć w tej dziedzinie. *FormEncode* defniuje klasę *FileUploadKeeper()*[4], której zadaniem jest zachowanie "pliku" w formularzu - w razie wystąpienia błędów. Klasa jednak nie działa. Zainteresowanych odsyłam do wątku[5] na *pylons-discuss*[6]. W naszym przypadku funkcjonalność ta, pomijając iż spowalnia ładowanie się formularza po "powrocie", jest zupełnie zbyteczna. Jeżeli wystąpi jakikolwiek błąd oznaczać to będzie iż nasz rysunek jest niepoprawny i należy z niego zrezygnować. W tym momencie jego zawartość jest już niepotrzebna.

Walidacja w kontrolerze :)

Skoro mamy tak fantastycznie przygotowaną klasę walidacji czas wdrożyć ją do użytku:

4 http://tiny.pl/sgqk
5 http://tiny.pl/sgqp
6 http://groups.google.com/group/pylons-discuss

CC BY NC ND

```
from darc.model.avatar import Avatar,
AvatarPermissionException

{...}

class UsersController(BaseController):

{...}

    @authorize(AuthenticatedUser())
    @authenticate_form
    @validate(schema=AvatarForm(), form="index")
    def uploadAvatar(self):
{...}
```

HOP na serwer - Apache 2 i mod_wsgi

Na koniec warto omówić uruchamianie naszej aplikacji na prawdziwym serwerze. Skrypt odpalany przez *paster* świetnie nadaje się do tworzenia aplikacji. Każde wywołanie *print* możemy obejrzeć na wyjściu konsoli naszego "serwera", ułatwia to na przykład debugowanie. Jednak w środowisku produkcyjnym przydałoby się uruchomienie naszego cuda na serwerze takim jak Apache 2. Na koniec pokażę jak go skonfigurować.

mod_wsgi[7] na Apache 2

Na początku zadbajmy o to aby w naszym systemie znalazł się *mod_wsgi*. Jest to moduł do apache, który pozwoli nam podpiąć po jedną z jego subdomen aplikację *Pylons*. Załaduj plik mod_wsgi do apache. W moim systemie (Ubuntu) polega to wyłącznie na dowiązaniu dwóch plików.

```
sudo ln -s /etc/apache2/mods-available/mod-wsgi.*
/etc/apache2/mods-enabled/
```

i zrestartowaniu serwera

```
sudo apache2ctl -k graceful
```

Plik konfiguracyjny

W przypadku produkcyjnej aplikacji komunikowanie o błędach powinno zostać pominięte. W tym celu stwórz kopię pliku *development.ini*

```
cp ~/Pylons/darc/development.ini ~/Pylons/darc/production.ini
```

i odkomentuj linijkę

7 http://www.modwsgi.org/

CC BY NC ND

```
set debug = false
```

Dla ujednolicenia dokonywanych zmian można zakomentować dodatkowo linijkę *debug = true* z sekcji *[DEFAULT]*.

Użyteczny skrypt

Do naszych celów będzie potrzebny dodatkowy skrypt. Nazwijmy go *~/Pylons/darc/darc.wsgi*

```
WORKING_ENV = "/home/username/Pylons/darc"
APP_CONFIG = "/home/username/Pylons/darc/production.ini"

import sys
import os
sys.path.append(WORKING_ENV)
sys.path.append(WORKING_ENV + '/wsgi')
os.environ['PYTHON_EGG_CACHE'] = '/usr/lib/python2.5/site-packages'

from paste.deploy import loadapp
application = loadapp("config:" + APP_CONFIG)
```

Bardzo ważne jest aby podać właściwą ścieżkę do używanych przez nas bibliotek Pythona. Zazwyczaj znajdują się one właśnie w */usr/lib/pythonX.X/site-packages* gdzie *X.X* to wersja używanego języka.

Na dokładkę - wirtual host

Czas na skonfigurowanie naszego wirtualhosta. Nic prostszego:

```
<VirtualHost *>
  ServerName darc.dev
  ServerAlias www.darc.dev

  WSGIScriptAlias / /home/username/Pylons/darc/wsgi/darc.wsgi
</VirtualHost>
```

Nasza strona będzie od teraz dostępna pod adresami *darc.dev* oraz *www.darc.dev*. Ponieważ jednak komputer nic nie wie o naszych zamiarach wypada wspomnieć o nich dopisując do pliku */etc/hosts* linijkę:

```
127.0.0.1 darc.dev www.darc.dev
```

i zrestartować Apache:

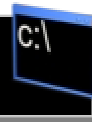
```
sudo apache2ctl -k stop && sudo apache2ctl -k start
```


cc BY NC ND

Instalujemy nasz portal w systemie

Aby ukończyć dzieło musimy jeszcze zainstalować naszą aplikację w systemie. Jest to ostatni etap działań :)

```
cd ~/Pylons/darc
sudo python setup.py install
```



Skrypt zacznie instalować w systemie najróżniejsze biblioteki. Teraz możemy już spokojnie wpisać w przeglądarce adres *http://www.darc.dev* i cieszyć się naszą aplikacją.

To tylko przykładowa procedura uruchomienia produkcyjnie *Pylons* na serwerze Apache. Więcej informacji znajdziesz na stronie[8]

Uwagi końcowe

Pamiętaj o docstringach

Właściwie - powinna być to uwaga na początek. Pomimo iż w kursie, aby zmniejszyć objętość i tak dużych ilości kodu, nigdzie nie używałem *Docstringów*, Ty rób to zawsze. Nigdy nie wiadomo kto będzie posługiwał się stworzonym przez Ciebie kodem. Być może i Tobie uratują kiedyś życie. Nawet jeżeli kod kontrolera wydaje Ci się oczywisty - opisz go choćby jednym zdaniem. Pisanie takich komentarzy to również czas na refleksję, moment, w którym być może często zauważysz, że zmiana koncepcji okaże się dużo korzystniejsza dla projektu i wpadniesz na fenomenalny pomysł, który szalenie ułatwi dalszy rozwój aplikacji.

PEP 8 i PEP 20

i nie tylko te. Standardy kodowania, nie mają na celu zabraniać, ale raczej pokazywać co przyniesie Ci korzyść. Pamiętaj, że kod jest pisany raz, a czytany wielokrotnie.

Komunikaty o błędach "z dołu".

Jeżeli zwróciłeś uwagę, zauważyłeś, że komunikaty o błędach ustalane były dopiero w kontrolerze. Na wcześniejszych warstwach wszystko działo się w materii wyjątków, których nazwy miały za zadanie opisywać problem. Można jednak wraz z wyjątkiem przekazywać również komunikat błędu oraz wywindować go aż do samej góry (widoku).

Pamiętaj jednak, że ostatecznym odbiorcom komunikatów jest użytkownik - nie będący zazwyczaj programistą i to dla niego przeznaczona jest informacja o tym co się stało. Oznacza to nie mniej nie więcej tyle iż komunikaty zawierające szczegóły techniczne powinny pozostać do wiadomości programistów. Nic nie stoi na przeszkodzie aby stworzyć klasę wyjątku, przekazującą więcej niż jeden string.

Do zaimplementowania swojej własnej klasy wyjątku może Cię zmusić również sam *Python.* Inwazyjne w tym przypadku okazują się komunikatu błędu, zawierające polskie znaki, i ich oczytanie po stronie kontrolera. Rozwiązaniem jest trzymanie go w polu klasy stworzonej samodzielnie i będącej rodzicem wszystkich innych tworzonych przez Ciebie klas wyjątków.

8 http://tiny.pl/sg7t

cc BY NC ND

Aplikacja produkcyjna - mod_wsgi

Dodatek

Wysyłanie emaili

W Pylons zawsze brakowało mi funkcji, która wysyłałaby emaile. Metodę warto umieścić w *lib/mail.py* i w miarę potrzeby importować tam gdzie musimy wysłać jakąś wiadomość. Oto dwie przykładowe implementacje.

Wysyłanie maili przez serwer gmail

```
import smtplib
from email.MIMEText import MIMEText

from pylons import config

def sendmail(sender, to, subject, text):
    sender = sender.encode('utf-8', 'ignore')
    to = to.encode('utf-8', 'ignore')
    subject = subject.encode('utf-8', 'ignore')
    text = text.encode('utf-8', 'ignore')

    msg = MIMEText(text)
    msg['Subject'] = subject
    msg['From'] =  sender
    msg['Reply-to'] = sender
    msg['To'] = to

    smtp = config.get('sendmail.smtp')
    port = config.get('sendmail.port')
    username = config.get('sendmail.username')
    password = config.get('sendmail.password')

    server = smtplib.SMTP(smtp, port)
    server.ehlo()
    server.starttls()
    server.ehlo()
    server.login(username, password)
    server.sendmail(sender, to, msg.as_string())
    server.close()
```

Wyjaśnienia wymaga pojawienie się obiektu *config*. Niewątpliwą wygodą jest móc trzymać hasło czy login użytkownika gmail w pliku konfiguracyjnym - nie zaś w ciele funkcji. Zaimportowanie *config* z modułu *pylons* pozwoliło nam na dostęp do sekcji konfiguracyjnej naszej aplikacji, którą definiujemy w *development.ini* w sekcji *[DEFAULT]*.

Lokalny postfix

W przypadku wysyłania przez lokalnego, domyślnie (nie)skonfigurowanego postfixa czy innego sendmaila funkcja będzie wyglądała mniej więcej tak:

```
import smtplib
from email.MIMEText import MIMEText

def sendmail(sender, to, subject, text):
    sender = sender.encode('utf-8', 'ignore')
    to = to.encode('utf-8', 'ignore')
    subject = subject.encode('utf-8', 'ignore')
    text = text.encode('utf-8', 'ignore')

    msg = MIMEText(text)
    msg['Subject'] = subject
    msg['From'] =  sender
    msg['Reply-to'] = sender
    msg['To'] = to

    server = smtplib.SMTP('localhost')
    server.sendmail(sender, to, msg.as_string())
    server.quit()
```

Licencja

http://creativecommons.org/licenses/by-nc-nd/2.5/pl/

Spis treści

Menu....1
Jeszcze kilka helperów....1
Czas na menu....1
Nasze menu w layoucie....2
Panel użytkownika....2
Formularz....2
Zmiany w modelu....3
Usuwanie użytkownika....4
Zmiana hasła....5
Jak to zrealizować ?....5
Zmiany w modelu....6
Kontroler....6
Walidacja....6
Zmiana danych użytkownika....8
Walidacja....8
Kontroler....8
Uzupełniamy formularz danymi....9
Problemy z e-mailem....10
Uploadowaniu pliku....11
Pole uploadu pliku....11
Avatar....11
Tworzymy katalog....12
Obsługa w kontrolerze....13
Wyświetlamy nasz avatar....13
Wiele rozszerzeń - wiele plików....15
Usuwanie plików....17
Pliki na serwerze - sprawdź zanim zapiszesz....18
Sprawdzanie typu pliku....18
Wymiary obrazka....18
Rozmiar obrazka....19
Zachowywanie wartości formularza pliku....20
Walidacja w kontrolerze :)....20
HOP na serwer - Apache 2 i mod_wsgi....21
mod_wsgi na Apache 2....21
Plik konfiguracyjny....21
Użyteczny skrypt....22
Na dokładkę - wirtual host....22
Instalujemy nasz portal w systemie....23
Uwagi końcowe....23
Pamiętaj o docstringach....23
PEP 8 i PEP 20....23
Komunikaty o błędach "z dołu"....23
Aplikacja produkcyjna - mod_wsgi....24
Dodatek....24
Wysyłanie emaili....24
Wysyłanie maili przez serwer gmail....24
Lokalny postfix....25
Licencja....25