${Pylons} 0.9.7 - Przewodnik część 3

Czas na bazę danych.

Nasza aplikacja posiada już formularz, który wydaje się być perfekcyjnym narzędziem do zakładania użytkownikom kont. Posiadamy nawet metodę create. Brakuje nam jedynie bazy danych. Do komunikacji użyjemy narzędzia ORM o wdzięcznej nazwie SQLAlchemy[1]. Zainteresowanych odsyłam do odpowiedniej strony[2] dokumentacji Pylons. Na pewno pomoże lepiej zrozumieć to co będzie się działo na dalszych stronach tego przewodnika. Na początku musimy skonfigurować nasze narzędzie. Zajrzyjmy do pliku *~/Pylons/darc/development.ini* do sekcji *[app:main]*:

```
[app:main]
use = egg:darc
full_stack = true

cache_dir = %(here)s/data
beaker.session.key = darc
beaker.session.secret = somesecret

# If you'd like to fine-tune the individual locations of
the cache data dirs
# for the Cache data, or the Session saves, un-comment the
desired settings
# here:
#beaker.cache.data_dir = %(here)s/data/cache
#beaker.session.data_dir = %(here)s/data/sessions

# SQLAlchemy database URL
sqlalchemy.url = sqlite:///%(here)s/development.db

# WARNING: *THE LINE BELOW MUST BE UNCOMMENTED ON A
PRODUCTION ENVIRONMENT*
# Debug mode will enable the interactive debugging tool,
allowing ANYONE to
# execute malicious code after an exception is raised.
#set debug = false
```

Jak łatwo się domyślić interesuje nas linijka związana z SQLAlchemy. Zgodnie z naszą początkową prośbą przykładowa konfiguracja została stworzona. Domyślnie, zaproponowany został mechanizm SQLite[3]. W celu korzystania z tej opcji w systemie powinna znaleźć się biblioteka *pysqlite*. Aby zaspokoić szersze grono czytelników pokażę również jak skonfigurować MySQL i PostgreSQL. Umownie nasze dane dostępowe do bazy danych to: login *pylons*, hasło *passpy*, nazwa bazy danych *pylons*.

1 http://www.sqlalchemy.org/
2 http://docs.pylonshq.com/models.html
3 http://sqlite.org/

Konfiguracja dla MySQL

Poniżej konfiguracja dla standardowo zainstalowanej bazy danych:

```
[app:main]
use = egg:darc
full_stack = true

cache_dir = %(here)s/data
beaker.session.key = darc
beaker.session.secret = somesecret

# If you'd like to fine-tune the individual locations of
the cache data dirs
# for the Cache data, or the Session saves, un-comment the
desired settings
# here:
#beaker.cache.data_dir = %(here)s/data/cache
#beaker.session.data_dir = %(here)s/data/sessions

# SQLAlchemy database URL
sqlalchemy.url =
mysql://pylons:passpy@localhost:3306/pylons
sqlalchemy.pool_recycle = 3600

# WARNING: *THE LINE BELOW MUST BE UNCOMMENTED ON A
PRODUCTION ENVIRONMENT*
# Debug mode will enable the interactive debugging tool,
allowing ANYONE to
# execute malicious code after an exception is raised.
#set debug = false
```

Pierwsza linijka to nic innego jak podanie wszystkich informacji niezbędnych do połączenia się z bazą danych w formacie *DSN*[4]. W formie adresu URL jako protokół występuje nazwa sterownika, dalej mamy nazwę użytkownika i oddzieloną od niej dwukropkiem hasło. Po małpce widzimy nazwę hosta (w znacznej części przypadków będzie to localhost) i port, na którym działa MySQL (domyślnie: 3306). Po ukośniku umieszczamy nazwę bazy danych.

Druga linijka zapewnia nam iż połączenia z bazą będzie podtrzymywane przez godzinę co pozwoli uniknąć błędów *MySQL server has gone away*.

Nie wolno nam oczywiście zapomnieć o obecności biblioteki *MySQL-python*.

4 http://en.wikipedia.org/wiki/Database_Source_Name

PostgreSQL on board

```
[app:main]
use = egg:darc
full_stack = true

cache_dir = %(here)s/data
beaker.session.key = darc
beaker.session.secret = somesecret

# If you'd like to fine-tune the individual locations of
the cache data dirs
# for the Cache data, or the Session saves, un-comment the
desired settings
# here:
#beaker.cache.data_dir = %(here)s/data/cache
#beaker.session.data_dir = %(here)s/data/sessions

# SQLAlchemy database URL
sqlalchemy.url =
postgres://pylons:passpy@localhost:5432/pylons

# WARNING: *THE LINE BELOW MUST BE UNCOMMENTED ON A
PRODUCTION ENVIRONMENT*
# Debug mode will enable the interactive debugging tool,
allowing ANYONE to
# execute malicious code after an exception is raised.
#set debug = false
```

Tutaj jak widać zmienił się wyłącznie sterownik (teraz *postgres*) oraz port (w tym przypadku *5432*). Pozostałe elementy jak w opisie dla MySQL. Linijka, która deklaruje podtrzymywanie połączenia przez godzinę jest wymagana wyłącznie w przypadku MySQLa. Biblioteka, niezbędna do działania tej konfiguracji, z której będzie korzystał Python to *psycopg2*.

Użytkownika bazy danych, z którego ma korzystać nasz projekt musisz stworzyć samodzielnie, przydzielić mu odpowiednie prawa i dostęp do zadeklarowanej bazy. Pylons nie zrobi tego za Ciebie.

Pomimo upływu czasu i ewolucji standardów nawet w świecie *Unicode* oraz *UTF-8* nam Polakom nadal nie jest łatwo zadbać o narodowe ogonki. Również tutaj musimy wykazać się dodatkową wiedzą. Po utworzeniu bazy danych (MySQL) zmień metodę porównywani napisów na *utf8_polish_ci*. Wszystkie tworzone przez SQLAlchemy pola oraz tabele będą pomimo ustawień w samym kodzie dziedziczyły informacje po bazie danych. **Zawsze** pamiętaj o zrobieniu tego **przed** uruchomieniem projektu i wygenerowaniem tabel, oraz dodaniu do nich jakichkolwiek danych. Dzięki temu unikniesz problemów z sytuacją, w której na stronie widać polskie znaki zaś po zajrzeniu do tabel w miejscu ogonków zobaczysz wszechobecne robaczki. Przenoszenie takiej bazy danych jest istnym horrorem.

Pomimo wsparcia Pythona dla unicode jako Polacy musimy zadbać tutaj o kilka rzeczy. Już przy konfiguracji należy dodać do DSN ciąg *?use_unicode=1&charset=utf8* (wyłącznie dla PostgreSQL i MySQL), dzięki któremu będziemy mieć pewność iż do naszej tabeli zapiszą się polskie literki zamiast krzaczków. Zobowiązuje nas to jednak do używania kodowania *UTF-8* (domyślne w Pylons 0.9.7) oraz wstawiania do bazy danych obiektów unicodowych. Dla porządku powinnyśmy przy tworzeniu tabel używać pól typu *Unicode* zamiast *String* oraz opatrywać pola typu *Text* parametrem *convert_unicode=True*. Te wszystkie zabiegi pozwolą nam oddychać swobodnie i nie martwić się o nasze rodzime znaczki. Na koniec podam jeszcze formę zmienionej konfiguracji SQLAlchemy dla podanych powyżej trzech rodzajów tabel:

SQLite: sqlalchemy.url = sqlite:///%(here)s/development.db
MySQL: sqlalchemy.url = mysql://pylons:passpy@localhost:3306/pylons?use_unicode=1&charset=utf8
sqlalchemy.pool_recycle = 3600
PostreSQL: sqlalchemy.url = postgres://pylons:passpy@localhost:5432/pylons?use_unicode=1&charset=utf8

Jak wspomniałem powyżej baza SQLite nie wymaga zmian :)

Której bazy danych używać ?

Której Ci wygodniej. Mechanizm, którym będziemy się posługiwać opiera się na technice ORM[5] dzięki czemu dla kodu programu nie będzie miało to żadnego znaczenia. W opisie będę posługiwał się MySQL. Wierzę, że w ten sposób dotrę do najszerszego grona odbiorców.

Tworzymy model użytkownika.

Jak wspomniałem przy omówieniu MVC modele to opakowania dla tabel. Stwórzmy więc jeden dla naszego użytkownika. W odróżnieniu od kontrolerów, w tym przypadku, będziemy nadawać nazwy w liczbie pojedynczej. Jak łatwo się domyślić umieszczać je będziemy w folderze *model*. Tak więc wyedytujmy plik *model/user.py*.

5 http://pl.wikipedia.org/wiki/Mapowanie_obiektowo-relacyjne

```
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta
```

Na starcie zaimportowaliśmy do naszego pliku potrzebne nam moduły. Teraz w pliku będziemy definiowali dwie rzeczy. Pierwsza to struktura tabeli. Nasz ORM musi wiedzieć z jakich pól składa się nasza tabela w bazie danych aby móc ją stworzyć. Przypomnijmy sobie jakie mamy pola: *login*, *password*, *email*. Powiedzmy, że dla naszego kaprysu chcemy umieść informację o dacie utworzenia konta. Umówmy się, że będziemy ją trzymać w polu o nazwie *created_at*. Dajmy również możliwość napisania kilku słów o sobie (pole *about*) oraz daty ostatniej modyfikacji rekordu *updated_at*.

Ale po kolei. Zacznijmy od klucza głównego id.

```
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True)
    )
```

Dzięki temu w naszej tabeli MySQL znajdzie się pole będące *kluczem głównym*, z własnością *auto_increment* typu *Integer*. Teraz dodamy pole login. Będzie to zgodnie z wcześniejszymi wskazówkami pole typu *Unicode* (z obecnymi ustawieniami w MySQL zadziała również *String*), które wygeneruje nam rekord typu *VARCHAR*.

```
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=false, unique=True)
    )
```

Jak widać maksymalna długość to 25 znaków. Jest to zgodne z naszym mechanizmem weryfikacji (login nie dłuższy niż 25 znaków). *Nullable=false* jest informacją iż pole nie będzie mogło posiadać wartości *NULL*. Ostatnia wartość ustawione na *True*, powoduje iż pole będzie musiało być unikalne.

Skoro już wiemy jak wygląda pole login możemy w analogiczny sposób utworzyć *email* oraz *password*.

```
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True)
    )
```

Co nam przybyło ? Ograniczyliśmy długość e-mail,a do 150 znaków. Jeżeli ktoś wpisze dłuższą wartość, podczas zapisywania do bazy danych, zostanie ona obcięta. Jeżeli chcesz się przed tym zabezpieczyć możesz dodać odpowiednią regułkę przy weryfikacji formularza i zabronić tym samym wprowadzania tak długiego ciągu znaków. Myślę, jednak, że 150 znaków to aż zanadto i nie musimy się o to martwić o przekroczenie tej wielkości. Unikalność pola adresu skrzynki jest przydatna przy implementowaniu opcji "odzyskaj dane konta". W naszym polu *password* będziemy zaś trzymać skrót hasła. Aby było troszkę bardziej egzotycznie proponuję zastosować algorytm *SHA512*. W tym wypadku nasz pole powinno więc posiadać długość 128 znaków.

```
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False)
    )
```

W naszej tabeli w celu zapewnienia bezpieczeństwa godnego dobrze napisanej aplikacji powinno znaleźć się jeszcze pole *salt*, w celu utrudnienia życia posiadaczom tęczowych tablic[6].

Czas na *about*. Będzie to zwykły rekord typu *TEXT*, z konwersją na *Unicode*. Dzięki temu będziemy mogli trzymać w nim aż *64 kB* danych (MySQL). Powinno wystarczyć każdemu. About domyślnie nie będzie zawierało żadnej wartości.

6 http://pl.wikipedia.org/wiki/T%C4%99czowe_tablice

```
import sqlalchemy as sa
from sqlalchemy import orm

from myapp.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False)
    )
```

Na koniec wzbogacimy naszą tabelkę o pola *updated_at* oraz *created_at*.

```
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )
```

Czas na krótkie wyjaśnienia. Pierwsze pole używa typu *TIMESTAMP*. Za każdym razem gdy będziemy uaktualniać nasz rekord zostanie wstawiony aktualny znacznik czasu (data modyfikacji). W przypadku *created_at*, użycie typu *Date i sa.func.now()* jako wartości domyślnej spowoduje

jednokrotne wstawienie tam daty - będzie to więc data utworzenia danego rekordu. Warto zauważyć, że skorzystaliśmy z funkcji udostępnionej przez SQLAlchemy. Kilka z nich jest opisane w dokumentacji, po resztę warto zajrzeć do kodu źródłowego pliku */usr/lib/python2.x/site-packages/SQLAlchemy-0.x.0-py2.x.egg/sqlalchemy/sql/functions.py*.

Inną metodą dostania się do informacji o danej bibliotece Pythona jest uruchomienie konsoli, zaimportowanie modułu i użycia funkcji help(), która wyświetli dokumentację modułu. Treść wpisywaną przez nas w konsoli oznaczę żółtym tłem.

```
python
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>import sqlalchemy as sa
>>>help(sa)
>>>help(sa.func)
```

Mamy tabelę - chcemy ORM.

Czas przełożyć opis naszej tabeli na język klas.

```
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

 class User(object):
    pass

orm.mapper(User, t_user)
```

Doszły dwie nowe rzeczy. Pierwsza - stworzyliśmy klasę *User*. To nią będziemy posługiwać się w naszym kodzie. Jest to zwykła klasa do której będziemy z czasem dopisywać różne metody, które będą nam przydatne - *model* w czystym jego znaczeniu - nasz użytkownik. Ostatnia linijka jest jedną z tych magicznych. Opakowuje tabelę (podaną jako drugi parametr) w klasę w naszym przypadku o nazwie *User*. Innymi słowy od teraz obiekt instancji *User* poza zdefiniowanymi przez nas metodami, których na razie brak *(pass)*, będzie posiadał metody i pola, w które wyposażył go *SQLAlchemy* pozwalające na odwzorowanie podanej tabeli z bazy danych. Tyle z teorii. Wszystko wyjdzie w praniu - a więc do dzieła.

Urzeczywistniamy sen.

Mamy już wszystko idealnie opisane. Nasz model kipi doskonałością - niestety, nie wie jeszcze nic o nim nasza baza danych. Czas poprosić ją o zrealizowanie naszego schematu. Zrobimy to oczywiście z użyciem narzędzi dostępnych wraz z frameworkiem. Jest tylko jeszcze jeden drobiazg. Nasze klasy istnieją, ale nikt o nich jeszcze nie wie. Nie są ładowane "automatycznie", nie będzie więc wiedział o nich również nasz skrypt mapujący schematy *SQLAlchemy* na tabele w bazie danych. Aby to zmienić wystarczy iż zaimportujesz nasze klasy w pliku *model/__init__.py*

```
"""The application's model objects"""
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

def init_model(engine):
    """Call me before using any of the tables or classes
in the model"""
    ## Reflected tables must be defined and mapped here
    #global reflected_table
    #reflected_table = sa.Table("Reflected",
meta.metadata, autoload=True,
    #                           autoload_with=engine)
    #orm.mapper(Reflected, reflected_table)

    sm = orm.sessionmaker(autoflush=True,
transactional=True, bind=engine)

    meta.engine = engine
    meta.Session = orm.scoped_session(sm)

from darc.model import user
from darc.model.user import User

## Non-reflected tables may be defined and mapped at
module level
#foo_table = sa.Table("Foo", meta.metadata,
#    sa.Column("id", sa.types.Integer, primary_key=True),
#    sa.Column("bar", sa.types.String(255),
nullable=False),
#    )
#
#class Foo(object):
#    pass
#
#orm.mapper(Foo, foo_table)

## Classes for reflected tables may be defined here, but
the table and
## mapping itself must be done in the init_model function
#reflected_table = None
#
#class Reflected(object):
#    pass
```

Jak widać po komentarzach cały kod zawarty w *models/user.py* można było zapisać również w tym pliku. Jednak moim skromnym zdaniem, trzymanie każdego modelu w osobnym pliku i importowanie go w razie potrzeby znacznie bardziej służy przejrzystości kodu i zachowaniu w nim

porządku.

Staraj się zawsze importować najmniej jak to tylko możliwe wskazując dokładnie co z danego modułu jest Ci potrzebne. Nie używaj *from module import* * jeżeli nie musisz.

Skoro poinformowaliśmy już "świat" o istnieniu naszego modułu czas na urzeczywistnienie go w postaci tabeli.

```
cd ~/Pylons/darc
paster setup-app development.ini
```

Proszę bardzo. Skrypt pobrał konfigurację z pliku *developement.ini* i wykonał funkcję *setup_app* zawartą w *websetup.py*. Odpowiedni fragment przełożył nasze opisy na język bazy danych. Możesz zajrzeć teraz do bazy danych i nazwie *pylons* i sprawdzić czy rzeczywiście znajduje się w niej tabela *users*.

Chciałbym się w tym miejscu wytłumaczyć z przyjętej przeze mnie konwencji nazywania tabel, kontrolerów czy modeli. IMHO tabela przetrzymuje dane wielu użytkowników (liczba mnoga *users*), zaś po pobraniu rekordu z bazy danych, *User* jest reprezentacją jednego, konkretnego użytkownika (liczba pojedyncza w nazwie klasy modelu). Nazwanie kontrolera *users* czy *user* wynika już wyłącznie z przyzwyczajenia. Z jednej strony sensownie wygląda adres *http://serwer.pl/user/create* z drugiej *http://serwer.pl/users/list*. W różnych podręcznikach można spotkać się z innymi konwencjami. Myślę, że z czasem wypracujesz swoją własną metodę nazewnictwa, która będzie logiczna i wygodna dla Ciebie.

Dodajemy nasz pierwszy rekord.

Zanim wpiszemy do metody *create* kod dodający użytkownika do bazy danych sprawdźmy przetestujmy jej działanie. Stworzymy nasze dzieło w interpreterze pythona.

```
cd ~/Pylons/darc
python
```

Na starcie zaimportujemy wszystkie biblioteki, które są nam potrzebne. Aby oddzielić wyjście interpretera od tego co wpisujemy, wprowadzana przez nas treść będzie zaznaczana kolorem żółtym.

```
Python 2.5.2 (r252:60911, Jul 31 2008, 17:28:52)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import sqlalchemy as sa # Importujemy SQLAlchemy
>>> from darc.model import init_model # Func. inicjuj. modele
>>> from darc.model import meta # Obiekt do komunikacji z BD
>>> from darc.model.user import User # Nasz model
>>> import hashlib
```

Na starcie importujemy sobie warsztat ze stajni SQLAlchemy, następnie funkcję inicjalizujące nasze modele by na końcu zapewnić sobie dostęp do obiektu User. Ładowanie *meta* pozwala nam na dostęp do zmiennej, przez którą manipulujemy (między innymi zapisujemy) informacjami zawartymi w bazie danych, a z biblioteki *hashlib* skorzystamy w celu wygenerowania funkcji skrótu *sha512*. Czas połączyć się z bazą danych a następnie zainicjalizować modele:

```
>>> engine =
sa.create_engine("mysql://pylons:passpy@localhost:3306/pylons?
use_unicode=1&charset=utf8") # Połączenie z nasz BD
>>> init_model(engine) # Inicjalizujemy modele z połączeniem
do tej konkretnej BD
```

Jeżeli podczas inicjalizowania modeli otrzymałeś jakieś komunikaty typu *deprecated* nie przejmuj się - ostatecznie korzystamy z Pylons w wersji rozwojowej, więc może się to przydarzyć. Teraz stwórzmy nasz obiekt i dodajmy do niego podstawowe dane.

```
>>> user = User() # Tworzymy obiekt
>>> user.login = u"dziubdziub" # Nasz login
>>> user.password =
unicode(hashlib.sha512("dziubek").hexdigest()) # Hashujemy
nasze hasło
>>> user.email = u"email@email.pl" # Podajemy email
```

Do zakodowania hasła skorzystaliśmy z odpowiedniej funkcji modułu *hashlib*. Dodanie na końcu *hexdigest* pozwoliło nam uzyskać skrót w formie *stringu*. Proszę zwrócić uwagę, że przy każdej wartości staraliśmy się pamiętać aby przypisywać dane w formie unicodowej (*u""*). Pozostałe pola mają u nas charakter opcjonalny więc pozostawimy je wypełnione wartościami domyślnymi. Możemy już zapisać nasz obiekt do bazy danych. Użyjemy do tego zmiennej *meta*.

```
>>> meta.Session.save(user) # Zapisujemy obiekt
>>> meta.Session.commit() # Potwierdzamy transakcję
>>> quit() # Wychodzimy z powłoki
```

Brawo ! Do Twojej bazy danych właśnie został dodany pierwszy użytkownik. Możesz już obejrzeć wyniki swoich działań wyświetlając rekordy tabeli. Tak oto bez pisania kodu - w bezpośrednim tego sensie znaczeniu - dodałeś pierwszego użytkownika :) Na tak "niskim" poziomie operowania obiektami Pythona nie działają mechanizmy walidacji a nasze środowisko pracy jest okrojone do

niezbędnego minimum. Ten sam kod możemy zaimplementować w którejś z metod. Czas nauczyć tego naszą stronę.

Importowanie kolejnych modułów oraz uruchamianie SQLAlchemy w powłoce miało charakter wyłącznie edukacyjny. Istnieje specjalny skrypt *paster shell*, który po wywołaniu w *~/Pylons/darc* automatycznie załaduje nam konfigurację i niezbędne moduły dając nam gotowe środowisko pozwalające nam pracować z naszą aplikacją.

Jeżeli powłoka Pythona jest dla Ciebie niewystarczająca możesz skorzystać z *ipython*[7] - shella o znacznie większych możliwościach.

Włóż swoją wiedzę do kontrolera.

Oto jak mógłby wyglądać nasz kontroler.

7 http://ipython.scipy.org/

```
import logging
import hashlib

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
from darc.model import meta
from darc.model.user import User

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')

    @validate(schema=RegisterForm(), form="register")
    def create(self):
        user = User()
        user.login = request.POST['login']
        user.email = request.POST['email']
        user.password =
unicode(hashlib.sha512(request.POST['password']).hexdigest
())
        meta.Session.save(user)
        meta.Session.commit()
        return render('users/create.xhtml')
```

Ponieważ wszystkie zmiany zostały omówione już wcześniej ten kod zostawię bez komentarza. Co jednak nie pasuje w tym całym bałaganie? Otóż - nie korzystamy do końca z tego co MVC nam oferuje. Dokładniej: bawimy się w kontrolerze bazą danych, która powinna być domeną modelu. Dodatkowo jako programiści projektujący warstwę logiczną musimy pamiętać o tym aby zaszyfrować hasło. Dlaczego to jest złe ?

Wyobraź sobie na chwilkę, że nad naszym projektem pracując trzy osoby. Firma zatrudniła fantastycznego specjalistę od baz danych, świetnego znawcę systemów CMS oraz dobrej klasy grafika. Zdecydowano się na zastosowanie architektury MVC aby każdy z nich mógł pracować w swojej warstwie, nie musząc znać się na pozostałych elementach portalu. Grafik miesza w folderze

public i w widokach. Pan CMS siedzi przede wszystkim w kontrolerach. Jego używanie modeli polega głównie na korzystaniu z klas, które napisał gość od Baz danych. Nie wie jak dane są zapisywane, ani nawet przechowywane na serwerze, szyfrowane czy nie - dane mu są dobrze udokumentowane klasy, a wszystko zapisuje się "samo". Żadna z osób nie ma dostępu do plików swojego kolegi.

Jesteś gościem od modeli (baz danych). Wyobraź sobie, że zadzwoniono do Ciebie z informacją iż algorytm szyfrowania sha512 został właśnie złamany, jest dziecinnie prosty do obejścia. Kryptografowie opracowali jednak sha1024 pozbawiony błędów poprzednika i w celu zapewnienia bezpieczeństwa musisz na niego migrować ponieważ zagrożenie jest ogromne. W tym momencie albo włamiesz się na konto kolegi "kontroler", albo zhackujesz bibliotekę hashlib w celu podmienienia jej implementacji.

Inna sytuacja - firma zażyczyła sobie aby wszystkie dane były od dziś trzymane w plikach tekstowych ponieważ od jutra MySQL staje się płatny a nikogo nie stać na zakupienie odpowiedniej liczby licencji. Jako programista mający dostęp wyłącznie do klas modeli jesteś kompletnie bezradny (pół biedy gdy firma migruje na coś co jest wspierane przez SQLAlchemy - wtedy jest to kwestia przemigrowania danych i zmiany konfiguracji Pylons).

Podczas gdy wystarczyłoby wszystkie szczegóły ukryć w modelu. Nie jest to rozwiązanie samolubne. Stworzenie "warstwy abstrakcji" daje kolosalne korzyści w konserwacji i modyfikacji kodu o walorach pozwalających na współpracę wielu osób nie znających się zupełnie na "czymś innym poza swoją działką". Podsumowując - programista kontrolera nie powinien musieć wiedzieć jak działa stworzony przez Ciebie model - ta wiedza nie jest mu potrzebna. Powinieneś odciążyć go od pamiętania o męczących szczegółach implementacji Twojej warstwy abstrakcji. Jedyne czego potrzebuje to wiedzieć jak jej używać. Poza dostarczonym przez Ciebie API nie powinien wiedzieć nic więcej. Zróbmy to więc porządnie. Oto model:

```
import hashlib
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

class User(object):

    def __setattr__(self, key, value):
        if key == 'password':
            value =
unicode(hashlib.sha512(value).hexdigest())
        object.__setattr__(self, key, value)

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

orm.mapper(User, t_user)
```

Jak widać wszystkie informacje o kodowaniu hasła czy użytym mechanizmie (SQLAlchemy) ukryliśmy w modelu na zewnątrz zaś pokażemy tylko naszą klasę. Ponieważ chcemy kodować hasło skorzystaliśmy z możliwości "wpięcia się" w mechanizm przypisywania wartości. Jeżeli ktoś przypisuje coś do pola obiektu automatycznie wywoływana jest metoda *__setattr__*. Sprawdzamy w niej czy przypisywanym polem nie jest przypadkiem *password* i w zamian jawnego tekstu przyporządkowujemy mu zaszyfrowaną wartość.

```
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
from darc.model.user import User

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')

    @validate(schema=RegisterForm(), form="register")
    def create(self):
        user = User()
        user.login = request.POST['login']
        user.email = request.POST['email']
        user.password = request.POST['password']
        user.save()

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

W metodzie *create* zginął na pewno obiekt *c* i przypisane do niego wartości, wypadałoby więc zmienić również widok. Jak widać programista tworzący kontroler nie musi już wiedzieć o wszystkich szczegółach dotyczących kodowaniu hasła czy zabawie z bazą danych. Czas na widok. Ograniczymy się w nim do wyświetlenia wiadomości iż użytkownik został utworzony.

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

<h1>Twój użytkownik został pomyślnie stworzony</h1>
```

Od teraz Twój program posiada już umiejętność dodawania do bazy danych nowych użytkowników. Tak jak wspominałem na początku - nie widać tutaj żadnego kodu SQL, ani tego z jakiej bazy danych korzystamy. Wszystko załatwia za nas warstwa modelu, które używa SQLAlchemy - my bawimy się wyłącznie obiektami.

Psujemy czyli pola unique w bazie danych.

Jeżeli jesteś spostrzegawczy zauważyłeś, że niektóre pola w bazie danych są unikalne. Oznacza to nie mniej nie więcej a tyle iż podczas gdy wpiszesz login lub email znajdujący się już w tabeli pojawią się problemy. Spróbuj dodać ponownie użytkownika dziubdziub. Co się stało ? Pylons "wywalił się" i wyświetlił błąd ***IntegrityError***. Powiem szczerze, że gdyby spotkało mnie coś takiego - jako użytkownika portalu byłbym mocno zawiedziony. Spodziewałbym się raczej ładnego poinformowania o problemie. Spróbujmy więc.

Wyłap wyjątek

Najszybszą i najprostszą metodą jest wyłapać odpowiedni wyjątek i zwrócić informację o dublującej się wartości w tabeli. Wprowadźmy więc w naszym modelu i kontrolerze pewne zmiany. Pierwszą z nich będzie wyłapanie wyjątku *IntegrityError* w modelu i poinformowanie o tym, własnym wyjątkiem, kontrolera.

```
import hashlib
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

class LoginOrEmailExistsException(Exception):
    pass

class User(object):

    def save(self):
        self.password
unicode(hashlib.sha512(self.password).hexdigest())
        meta.Session.save(self)
        try:
            meta.Session.commit()
        except sa.exc.IntegrityError:
            raise LoginOrEmailExistsException()

orm.mapper(User, t_user)
```

Zmieniło się nie wiele. Po pierwsze stworzyliśmy nową klasę wyjątku, który oznacza fakt iż w bazie danych, któraś z dwóch wartości się powtórzyła. Następnie w kontrolerze wyłapujemy wyjątek zwracany przez metodę *commit()* oraz przekładamy go na język modelu. W tym przypadku oznacza to, że któraś z dwóch wartości istnieje już w naszej tabeli. Ktoś mógłby zapytać "a dlaczego nie wyłapiemy *IntegrityError* w kontrolerze? Nie musielibyśmy tworzyć klasy naszego własnego wyjątku? Służy to po pierwsze zwiększeniu czytelności kodu i tworzeniu wspomnianej wcześniej warstwy abstrakcji. Kontroler ma komunikować się z modelem - i model mówi "taki użytkownik już istnieje". O bazie danych mowy być nie może :) Właśnie o to chodzi - nasz "LoginOrEmailExistsException" to nic innego jak przełożenie języka bazy danych na język naszej

aplikacji. Staje się to też bardziej zrozumiałe dla programisty kontrolera, który nie musi nic wiedzieć o bazie danych natomiast o istnieniu użytkowników wie bardzo dobrze. Czas na zmiany w kontrolerze.

```
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
from darc.model.user import User,
LoginOrEmailExistsException

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')

    @validate(schema=RegisterForm(), form="register")
    def create(self):
        user = User()
        user.login = request.POST['login']
        user.email = request.POST['email']
        user.password = request.POST['password']
        try:
            user.save()
        except LoginOrEmailExistsException:
            return 'Twój e-mail lub login istnieje już w
bazie danych. Użyj innego.'

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

Ot i działa. Rozwiązanie jest oczywiście pisane "na kolanie" i jest jednym z tych, które stosuje się gdy dzwoni wściekły szef a Ty jesteś właśnie na randce. Polega ono na wyłapaniu wyjątku jaki zgłasza SQLAlchemy i wyświetlenie ludzkiego komunikatu. Jednak czy naprawdę potrzebny jest nam osobny widok do wyświetlenia informacji o dodaniu nowego użytkownika, czy pojawiającym

się problemie ? Nie :) Czas na wiadomości flash.

Gdyby chcieć pójść jeszcze bardziej w kierunku programowania defensywnego powinniśmy obsłużyć również wyjątek zwracany gdy nie wszystkie wymagane pola są wypełnione.

Cywilizowany sposób powiadamiania o problemach.

Aby skorzystać z wiadomości flash musimy dodać je do ładowanych automatycznie helperów. Czas wrócić do pliku *lib/helpers.py*.

```
"""Helper functions

Consists of functions to typically be used within
templates, but also
vailable 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
from webhelpers.pylonslib import Flash as _Flash
flash = _Flash()
```

Od teraz możemy korzystać już w kontrolerach z obiektu flash i dodawać do niego wiadomości. Zanim jednak to zrobimy znajdźmy miejsce w którym wyświetlimy użytkownik informację o naszym problemie. Na celownik weźmiemy plik *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>
    <% messages = h.flash.pop_messages() %>
    % if messages:
    <ul id="flash-messages">
    % for message in messages:
        <li>${message}</li>
    % endfor
    </ul>
    % endif

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

Co to za zamieszanie z tymi znaczkami "%" - ktoś zapyta. Już spieszę z wyjaśnieniem. Te symbole procentu to nic innego jak zakomunikowanie *Mako* iż teraz nastąpi kod, który ma być zinterpretowany jako linijka programu Pytona. Na początku pobieramy wszystkie wiadomości jakie dodaliśmy do naszego kontrolera i jeżeli lista ta nie jest pusta wyświetlamy ją. Czas skorzystać z tych dobrodziejstw.

```
# -*- encoding: utf-8 -*-
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.helpers import flash
from darc.lib.base import BaseController, render
#from darc import model
from darc.model.form import RegisterForm
from darc.model.user import User,
LoginOrEmailExistsException

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')

    @validate(schema=RegisterForm(), form="register")
    def create(self):
        user = User()
        user.login = request.POST['login']
        user.email = request.POST['email']
        user.password = request.POST['password']
        try:
            user.save()
        except LoginOrEmailExistsException:
            flash('Twój e-mail lub login istnieje już w
bazie danych. Użyj innego.')
        else:
            flash('Twój użytkownik został pomyślnie dodany
do bazy danych.')

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

Teraz po wykonaniu akcji *create* informacja zostanie zapisana do obiektu *flash* i wyświetlona po przekierowaniu z powrotem do *register*. Na samym początku pojawił się pewien komentarz, który wymaga wyjaśnienia. Jest to informacja dla Pythona mówiąca o kodowaniu pliku. Pojawiła się tam

z powodu użycia w pliku polskich ogonków. Bez niej otrzymalibyśmy nieprzyjemny komunikat błędu. Wracając do naszych powiadomień. Wszystko pięknie gdy operacja zakończy się powodzeniem, jednak w przypadku błędu co prawda lądujemy ponownie w formularzu jednak już bez wartości początkowych. Spróbujmy jeszcze bardziej ulepszyć nasze rozwiązanie - tym razem zrobimy to naprawdę porządnie.

Jeżeli zwróciłeś uwagę plik *create.xhtml* nie jest już nigdzie wykorzystywany. Możesz go z czystym sumieniem usunąć.

Rozszerzamy umiejętności walidatora.

Aby dane, które wpiszemy w nasz formularz nie ginęły, sprawdzanie warunków musimy przenieść z kontrolera do metod klasy walidacji. W tym celu stworzymy dwa niestandardowe walidatory: osobny dla *loginu*, drugi dla *emaila*. Każdy z nich będzie sprawdzał czy w bazie danych istnieje już pole zawierające daną wartość. Jeżeli nie istnieje - zostanie zwrócony wyjątek *NoResultFound*. Wtedy wszystko będzie grało - w przeciwny wypadku sami zgłosimy wyjątek *Invalid*, który poinformuje *FormEncode* o tym iż powinien wyświetlić ponownie odpowiedni formularz i komunikat błędu. Czas zobaczyć jak to będzie wyglądało w praktyce. Na starcie nauczymy nasz model sprawdzać czy istnieje login lub email w bazie danych.

```
import hashlib
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

class LoginExistsException(Exception):
    pass

class EmailExistsException(Exception):
    pass

class LoginOrEmailExistsException(Exception):
    pass

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

class User(object):

    def save(self):
        self.password =
unicode(hashlib.sha512(self.password).hexdigest())
        meta.Session.save(self)
        try:
          meta.Session.commit()
        except sa.exc.IntegrityError:
          raise LoginOrEmailExistsException()

orm.mapper(User, t_user)
```

Nasz model wzbogacił się o dwie klasy wyjątków *LoginExistsException*, który będziemy zgłaszać gdy okaże się że login wpisany w formularzu jest już zajęty oraz *EmailExistsException* w analogicznym przypadku . Dodajmy teraz metodę sprawdzającą czy w bazie danych znajdują się

nasze dane. Zacznijmy od loginu.

```
import hashlib
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

class LoginExistsException(Exception):
    pass

class EmailExistsException(Exception):
    pass

class LoginOrEmailExistsException(Exception):
    pass

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

class User(object):

    def loginExists(self):
        try:
            meta.Session.query(User).filter(User.login==se
lf.login).one()
        except orm.exc.NoResultFound:
            pass
        else:
            raise LoginExistsException()

    def save(self):
        self.password =
unicode(hashlib.sha512(self.password).hexdigest())
        meta.Session.save(self)
        try:
```

```
            meta.Session.commit()
          except sa.exc.IntegrityError:
            raise LoginOrEmailExistsException()

orm.mapper(User, t_user)
```

Pojawiła się więc metoda, która ma na celu sprawdzić czy w bazie danych występuje już wpisany przez nas login. Dzięki własności Pythona możemy to zrobić bardzo prostym łańcuszkiem. Spróbuję go teraz ciut objaśnić.

Na początek pobieramy obecną sesję połączenia z bazą danych: *meta.Session*. Nie jest to jednak obiekt, na którym możemy wykonywać zapytania. Aby go uzyskać wywołujemy metodę *query*. Jako jej parametr podajemy model, na którym chcemy operować. SQLAlchemy przekłada to sobie na informację o tabeli. *Filter* pozwala nam pobrać dane spełniające jakieś konkretne warunki. Ostatecznie używamy *one()*, który ma za zadanie zwrócić jeden rekord spełniający nasze kryteria - to właśnie ta metoda zwraca kluczowy wyjątek.

Wszystko to ujmujemy w blok *try* - *except*, w którym wyłapujemy *NoResultFound*. W tym przypadku, jest to informacja "dobra" - nie podejmujemy więc żadnych akcji. W przeciwnym zgłaszamy odpowiedni wyjątek. W sumie tyle. Zróbmy to samo dla pola *email*.

```
import hashlib
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

class LoginExistsException(Exception):
    pass

class EmailExistsException(Exception):
    pass

class LoginOrEmailExistsException(Exception):
    pass

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

class User(object):

    def emailExists(self):
        try:
            meta.Session.query(User).filter(User.email==se
lf.email).one()
        except orm.exc.NoResultFound:
            pass
        else:
            raise EmailExistsException()

    def loginExists(self):
        try:
            meta.Session.query(User).filter(User.login==se
lf.login).one()
        except orm.exc.NoResultFound:
            pass
```

```
            else:
                raise LoginExistsException()

        def save(self):
            self.password =
    unicode(hashlib.sha512(self.password).hexdigest())
            meta.Session.save(self)
            try:
              meta.Session.commit()
            except sa.exc.IntegrityError:
              raise LoginOrEmailExistsException()

    orm.mapper(User, t_user)
```

Proszę bardzo. Metoda dla Emaila różni się wyłącznie typem sprawdzanego pola i zwracanego wyjątku. Teraz zmodyfikujmy nasz walidator.

```
import formencode

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

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')]
```

Na początku pliku importujemy nasz model oraz wyjątki, które będziemy chcieli obsłużyć. Stworzymy wstępnie pustą klasę *LoginExistsValidator*.

```
import formencode

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

class
LoginExistsValidator(formencode.validators.FancyValidator)
:
    pass

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')]
```

Dziedziczy ona po *FancyValidator*. Dodamy do niej teraz metodę *validate_python*. To właśnie ona będzie wywołana podczas sprawdzana formularza.

```
import formencode

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

class
LoginExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        pass

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')]
```

Co powinna zrobić nasza metoda ? Sprawdzić czy w bazie danych istnieje rekord z wpisanym przez użytkownika *loginem* i w gdy odpowiedź będzie pozytywna zwrócić odpowiedni wyjątek.

```
import formencode

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

class
LoginExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.login = value
        try:
            user.loginExists()
        except LoginExistsException:
            raise formencode.Invalid(-'Your login is used
by another user. Please type another username.', value,
state)
        else:
            pass


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')]
```

Wszystko co robimy to przekazanie do naszego modelu wartość pola *login* i wyłapanie odpowiedniego wyjątku. Zgłoszenie *Invalid(...)* poinformuje właściwe mechanizmy odpowiedzialne za obsługę walidacji aby wrócić do formularza i wyrenderować go wyświetlając wpisany przez nas komunikat błędu. Powtórzmy tę samą sztuczkę z Emailem.

```
import formencode

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

class
LoginExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.login = value
        try:
            user.loginExists()
        except LoginExistsException:
            raise formencode.Invalid('Your login is used
by another user. Please type another username.', value,
state)
        else:
            pass


class
EmailExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.email = value
        try:
            user.emailExists()
        except EmailExistsException():
            raise formencode.Invalid('Your email is used
by another user. Please type another email address.',
value, state)
        else:
            pass


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')]
```

A więc mamy już nasze klasy, które w razie powtórzenia wartości w polach *login* lub *email* zwrócą nam odpowiedni wyjątek i informację. Czas je wykorzystać. Użyjemy ich dokładnie tak jak każdej innej klasy do walidacji.

```
import formencode

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

class
LoginExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.login = value
        try:
            user.loginExists()
        except LoginExistsException:
            raise formencode.Invalid('Your login is used
by another user. Please type another username.', value,
state)
        else:
            pass


class
EmailExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.email = value
        try:
            user.emailExists()
        except EmailExistsException():
            raise formencode.Invalid('Your email is used
by another user. Please type another email address.',
value, state)
        else:
            pass


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), LoginExistsValidator())
    email =
formencode.All(formencode.validators.Email(not_empty=True)
, EmailExistsValidator())
    password = formencode.validators.MinLength(8,
```

```
    not_empty=True)
        password_confirmation = formencode.validators.String()

        chained_validators =
    [formencode.validators.FieldsMatch('password',
    'password_confirmation')]
```

Tylko tyle ? Tak. Tylko tyle. Aby zastosować dodatkowy walidator w przypadku pola *email* musieliśmy posłużyć *formencode.All* tak jak zrobiliśmy to już wcześniej w przypadku *login*. Ponieważ komunikaty są w języku angielskim wprowadźmy jeszcze jedną zmianę.

```
import formencode

from pylons.i18n import _

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

class
LoginExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.login = value
        try:
            user.loginExists()
        except LoginExistsException:
            raise formencode.Invalid(_('Your login is used
by another user. Please type another username.'), value,
state)
        else:
            pass


class
EmailExistsValidator(formencode.validators.FancyValidator)
:

    def validate_python(self, value, state):
        user = User()
        user.email = value
        try:
            user.emailExists()
        except EmailExistsException():
            raise formencode.Invalid(_('Your email is used
by another user. Please type another email address.'),
value, state)
        else:
            pass


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), LoginExistsValidator())
    email =
formencode.All(formencode.validators.Email(not_empty=True)
```

```
, EmailExistsValidator())
    password = formencode.validators.MinLength(8,
not_empty=True)
    password_confirmation = formencode.validators.String()

    chained_validators =
[formencode.validators.FieldsMatch('password',
'password_confirmation')]
```

Pozwoli nam to za chwilkę przetłumaczyć nasze wiadomości o błędach. Zaimportowana funkcja "_" potrafi przy odpowiednim przygotowaniu portalu zwracać komunikaty w różnych językach. Do dzieła.

Tłumaczenie.

Dwa skróty i18n[8] oraz l10n[9]. Co One oznaczają? i18n to nic innego jak *internacjonalization (**i + 18** liter w środku **+ n**)*. Podobnie sprawa ma się sprawa z lolkalizacją (***l** + 10 liter w środku **+ n** - localization)*. Terminy są dość bliskoznaczne. Tutaj ograniczymy się do zadbania o ich część wspólną czyli przetłumaczenia komunikatów na obcy język. Narzędziem służącym do internacjonalizacji oprogramowania programów Pythona jest *Babel*[10]. Proces tłumaczenia aplikacji jest bardzo prosty i składa się z 3-4 etapów. Na pierwszym z nich specjalny skrypt wyłapuje wszystkie frazy, które są ujęte w funkcje "tłumaczenia", np *_()*. Zapisywane są one w pliku tekstowym *.pot*. Następnie tworzymy "kopię" takiego pliku z tłumaczeniami dla konkretnego języka - powiedzmy właśnie polskiego z rozszerzeniem *.po* i do obcojęzycznych znaczeń przypisujemy nasze przetłumaczone na dany język treści. W trzecim etapie plik tekstowy *.po* "kompilowany jest do" pliku binarnego *.mo*, z którego umie korzystać już Pylons. Za finał można uznać ustawienie języka, w którym chcemy aby nasza aplikacja komunikowała się z użytkownikiem.

Zacznijmy od odkomentowania linijek w pliku *~/Pylons/darc/setup.py*:

```
    message_extractors = {'darc': [
            ('**.py', 'python', None),
            ('templates/**.mako', 'mako', None),
            ('public/**', 'ignore', None)]},
```

W trzeciej linijce przeszukiwane pliki szablonów mają zdefiniowane rozszerzenie **.mako*. My zaś używamy plików z rozszerzeniem **.xhtml*. Zmień tą linijkę na: *('templates/**.xhtml', 'mako', None),*

Są one odpowiedzialne za wyłapywanie tłumaczonych wiadomości z najróżniejszych miejsc projektu. Na nasze potrzeby stworzymy jeszcze katalog, w którym skrypty będą umieszczały generowane przez siebie pliki.

8 http://pl.wikipedia.org/wiki/I18n
9 http://pl.wikipedia.org/wiki/L10n
10 http://babel.edgewall.org/

```
cd ~/Pylons/darc
mkdir darc/i18n
```

Zaczynamy od wygenerowania pliku *.pot*.

```
python setup.py extract_messages
```

Zobaczmy co znalazło się w *darc/i18n/darc.pot*. Na początku widzimy nagłówki informujące o dacie utworzenia pliku czy autorze, zaś zaraz po nich dwa komunikaty :) Dlaczego one się tutaj znalazły. Wszystko dzięki użyciu funkcji *_()*, która jest częścią biblioteki *i18n* do Pythona. Skrypt wyłapał taki zapis w kodzie i pobrał jej argument do pliku. Fantastycznie uzupełnijmy dane w nagłówkach i zajmijmy się stworzeniem pliku dla naszego języka. Wykonamy to z użyciem funkcji *init_catalog*.

```
python setup.py init_catalog -l pl
```

Tak oto dostaliśmy plik *.po*, z tymi samymi wiadomościami przeznaczony jednak do przetłumaczenia ich na język polski. Zajrzyj do pliku *darc/i18n/pl/LC_MESSAGES/darc.po* i przetłumacz komunikaty. Na przykład tak (nagłówki pominąłem):

```
msgid "Your login is used by another user. Please type
another username."
msgstr "Twój login jest zajęty. Spróbuj wpisać inną nazwę
użytkownika."

msgid "Your email is used by another user. Please type
another email address."
msgstr "Twój email występuje już w naszej bazie danych.
Spróbuj użyć innego."
```

Zapisz plik. Czas na ostatni etap.

```
python setup.py compile_catalog
```

Nasze tłumaczenia są już gotowe do użycia, musimy tylko poinformować aplikację o tym jakiego języka ma używać nasza aplikacja. W tym celu w dziale *[app:main]* pliku *~/Pylons/darc/development.ini* dodaj linijkę:

```
lang = pl
```

Możesz sprawdzić czy komunikaty wyświetlają się już po polsku :)

Myślę, że po raz kolejny Pylons pokazało iż korzysta z dobrych rozwiązań i nawet stworzenie wielojęzycznej aplikacji nie będzie teraz sprawiało komuś problemu. Oczywiście - komunikaty można było podać od razu po polsku jednak w naszym przypadku byłoby to pozbawione waloru edukacyjnego. Jest jeszcze jedna przyczyna. Jeżeli zajrzysz do źródeł *FormEncode* dostrzeżesz folder *i18n*. Tak - dokładnie, sama biblioteka korzysta z tego samego rozwiązania :] dzięki temu

może zawracać nam komunikaty błędów w naszym rodzimym języku. W naszym tutorialu liznęliśmy tylko temat aby pokazać, że cała operacja nie jest trudna i, że się da. Więcej informacji znajdziesz na stronach dokumentacji frameworka[11] oraz samego *Babel*[12].

Po wykonaniu tłumaczenia na powrót zakomentuj odpowiednie linijki z pliku *setup.py*.

Praca nad drobiazgami.

Czas poświęcić ciut czasu na poprawę estetyki naszej aplikacji oraz ułatwienie troszkę życia użytkownikowi. Chodzi o wyrównanie pól formularzy, sprawdzanie dostępności nazwy użytkownika czy email z użyciem AJAX. Na starcie stwórzmy dwa katalogi w folderze naszego projektu.

```
cd ~/Pylons/darc/darc
mkdir public/stylesheets
mkdir public/javascript
```

Będziemy w nich trzymać arkusze stylów oraz pliki z kodem JS. Przydadzą się nam jeszcze dwa helpery w *lib/helpers.py*:

```
"""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, stylesheet_link, javascript_link
from webhelpers.pylonslib import Flash as _Flash
flash = _Flash()
```

Wyrównanie pól formularzy

Nasze pola formularzy nie wyglądają obecnie zbyt zadbanie. Możemy to jednak szybko poprawić. Metoda na "naprawienie" tego bałaganu dokładnie opisana jest w kursie BrowseHappy[13] ja podam już tylko gotowe rozwiązanie - zainteresowanych odsyłam do wcześniej podanej witryn.

Zaczniemy od stworzenia pliku *public/stylesheets/forms.css*:

11 http://docs.pylonshq.com/i18n.html
12 http://babel.edgewall.org/wiki/Documentation/index.html
13 http://kurs.browsehappy.pl/Krok/Formularze

```
label {
  display: block;
  width: 160px;
  float: left;
}

input, textarea {
  display: block;
  float: left;
}

div {
  overflow: auto;
  clear: both;
  margin-bottom: 0.5em;
}

input.check,
input.submit {
  margin-left: 160px;
  display: inline;
}

label.check {
  width: auto;
}
```

Teraz wystarczy użyć go w naszym kodzie. Z jednej strony dla formularzy o różnych długościach pola <*label*> będą potrzebne różne marginesy więc może zdarzy się tak, że dla różnych formularzy stosować będziesz różne reguły.. Dla uproszczenia jednak zaimportujemy plik do samego *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" />
    ${h.stylesheet_link('/stylesheets/forms.css')}
  </head>
  <body>
    <% messages = h.flash.pop_messages() %>
    % if messages:
    <ul id="flash-messages">
    % for message in messages:
        <li>${message}</li>
    % endfor
    </ul>
    % endif

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

Cel został osiągnięty. Pola są wyrównane :) i nie musieliśmy używać tabelek. Być może w Twoim przypadku będziesz zmienić marginesy - w zależności od wielkości czcionki. Skoro już tutaj jesteśmy czas na AJAX.

Ajaxowe sprawdzanie obecności użytkownika w bazie.

Ponieważ nie jest to kurs JavaScript tylko Pylons ograniczę się tutaj tylko do pokazania zmian w odpowiednich plikach i krótkich komentarzy. Zacznijmy od stworzenia w formularzu dwóch przycisków i pól w których będziemy wyświetlali informację o tym czy login istnieje czy nie.

```
<%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")} <button type="button"
name="loginExists">sprawdź</button> <span
id="loginExists"></span></div>
    <div><label for="emailField">E-mail: </label>$
{h.text("email", id="emailField")} <button type="button"
name="emailExists">sprawdź</button> <span
id="emailExists"></span></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()}
```

Następnie dodajemy do naszego kontrolera dwie akcje, które będziemy wywoływać ajaxowo i poinformują nas o tym co znajduje się w bazie danych. Jest to niemalże ten sam kod, który wykorzystujmy w klasie walidatora.

```
# -*- encoding: utf-8 -*-
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.helpers import flash
from darc.lib.base import BaseController, render
#from darc import model
from darc.model.form import RegisterForm
from darc.model.user import User,
LoginOrEmailExistsException, LoginExistsException,
EmailExistsException

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')

    @validate(schema=RegisterForm(), form="register")
    def create(self):
        user = User()
        user.login = request.POST['login']
        user.email = request.POST['email']
        user.password = request.POST['password']
        try:
            user.save()
        except LoginOrEmailExistsException:
            flash('Twój e-mail lub login istnieje już w
bazie danych. Użyj innego.')
        else:
            flash('Twój użytkownik został pomyślnie dodany
do bazy danych.')

        return redirect_to(action='register')

    def loginExists(self):
        user = User()
        user.login = request.POST['login']
```

```
        try:
            user.loginExists()
        except LoginExistsException:
            return 'login jest zajęty'
        else:
            return 'login jest wolny'

    def emailExists(self):
        user = User()
        user.email = request.POST['email']
        try:
            user.emailExists()
        except EmailExistsException:
            return 'email jest zajęty'
        else:
            return 'email jest wolny'
```

Metoda zwracania stringów jest idealna do AJAXa.

Aby połączyć nasz formularz i metody "niewidzialną nicią" użyjemy biblioteki *jQuery*[14]. Ściągnij ją i zapisz jako *public/javascript/jquery,js*. Następnie stwórz plik *public/javascript/main.js* o następującej zawartości:

14 http://jquery.com/

```
jQuery.noConflict();

jQuery(document).ready(function() {

    // Check login exists
    jQuery('button[name=loginExists]').click(function() {
        var vLogin = jQuery('input#loginField').val();
        is (! vLogin) { return false }
        var message = '';
        jQuery.post('/users/loginExists', { login:
vLogin },
            function(data, textStatus) {
                if (textStatus != 'success') {
                    message = 'wystąpił błąd';
                } else {
                    message = data;
                }
                jQuery('span#loginExists').text(message);
            },
        'text');
    });

    // Check email exists
    jQuery('button[name=emailExists]').click(function() {
        var vEmail = jQuery('input#emailField').val();
        is (! vEmail) { return false }
        var message = '';
        jQuery.post('/users/emailExists', { email:
vEmail },
            function(data, textStatus) {
                if (textStatus != 'success') {
                    message = 'wystąpił błąd';
                } else {
                    message = data;
                }
                jQuery('span#emailExists').text(message);
            },
        'text');
    });
});
```

Ten fragmencik pozwoli komunikować się sprawnie z naszymi metodami w kontrolerze i informować o ich wynikach. Teraz wystarczy załączyć skrypty do kodu *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" />
    ${h.stylesheet_link('/stylesheets/forms.css')}
    ${h.javascript_link('/javascript/jquery.js')}
    ${h.javascript_link('/javascript/main.js')}
  </head>
  <body>
    <% messages = h.flash.pop_messages() %>
    % if messages:
    <ul id="flash-messages">
    % for message in messages:
        <li>${message}</li>
    % endfor
    </ul>
    % endif

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

Z elementów, o których warto byłoby tutaj wspomnieć brakuje nam jeszcze mechanizmu ochrony przed botami. Propozycji jest wiele. Mechanizm Captcha[15] czy filtr antyspamowy taki jak Sblam[16] to ważne mechanizmy zabezpieczające naszą stronę. Nie będziemy się zajmować się tymi zagadnieniami w nie mniejszym tutorialu, temat uważam jednak za wystarczająco ważny aby wzmianka pojawiła się w ramce ostrzeżenia.

Logowanie

Rejestracja działa już dość dobrze. Jednak nasi użytkownicy nie mogą się jeszcze zalogować. I w tej dziedzinie również otrzymujemy wsparcie ze strony naszego frameworka. Dobrodziejem okazuje się *AuthKit*[17]. Jest to narzędzie, które znacznie ułatwi nam pracę.

15 http://pl.wikipedia.org/wiki/Captcha
16 http://sblam.com/
17 http://authkit.org/

Stopień w jakim skorzystamy z AuthKit będzie skrajnie infinitezymalny i dziwny. W rozdziale PylonsBook[18] poświęcony autoryzacji możesz zobaczyć, że AuthKit posiada wiele własnych metod autoryzacji opartych na przykład o zmienne środowiskowe czy pliki tekstowe a nawet autorskie wzorce SQLAlchemy. My "wepniemy" go w system autentykacji.

Włączamy AuthKit.

Aby móc korzystać z naszego rozwiązania musimy je "załadować" do naszego frameworka. Jest to mechanizm dość niskopoziomowy więc wymaga dodania go do warstwy middleware, która jest wywoływana - zanim dostępna staje się aplikacja jaką znamy. Niestety Pylons nie robi tego domyślnie. Za obsługę dodatkowego oprogramowania, jakiego chcemy używać odpowiedzialny jest plik *config/middleware.py*. Wszystko co musimy zrobić to załadować odpowiednią bibliotekę i w bardzo konkretnym miejscu "wmontować" *middleware* AuthKita do środowiska naszej aplikacji.

18 http://pylonsbook.com/alpha1/authentication_and_authorization

```
"""Pylons middleware initialization"""
from beaker.middleware import CacheMiddleware,
SessionMiddleware
from paste.cascade import Cascade
from paste.registry import RegistryManager
from paste.urlparser import StaticURLParser
from paste.deploy.converters import asbool
from pylons import config
from pylons.middleware import ErrorHandler,
StatusCodeRedirect
from pylons.wsgiapp import PylonsApp
from routes.middleware import RoutesMiddleware
import authkit.authenticate

from darc.config.environment import load_environment

def make_app(global_conf, full_stack=True, **app_conf):
    """Create a Pylons WSGI application and return it

    ``global_conf``
        The inherited configuration for this application.
Normally from
        the [DEFAULT] section of the Paste ini file.

    ``full_stack``
        Whether or not this application provides a full
WSGI stack (by
        default, meaning it handles its own exceptions and
errors).
        Disable full_stack when this application is
"managed" by
        another WSGI middleware.

    ``app_conf``
        The application's local configuration. Normally
specified in
        the [app:<name>] section of the Paste ini file
(where <name>
        defaults to main).

    """
    # Configure the Pylons environment
    load_environment(global_conf, app_conf)

    # The Pylons WSGI app
    app = PylonsApp()

    # CUSTOM MIDDLEWARE HERE (filtered by error handling
middlewares)
```

```
    # Routing/Session/Cache Middleware
    app = RoutesMiddleware(app, config['routes.map'])
    app = SessionMiddleware(app, config)
    app = CacheMiddleware(app, config)

    if asbool(full_stack):
        # Authenticate
        app = authkit.authenticate.middleware(app,
app_conf)
        # Handle Python exceptions
        app = ErrorHandler(app, global_conf,
**config['pylons.errorware'])

        # Display error documents for 401, 403, 404 status
codes (and
        # 500 when debug is disabled)
        if asbool(config['debug']):
            app = StatusCodeRedirect(app)
        else:
            app = StatusCodeRedirect(app, [400, 401, 403,
404, 500])

    # Establish the Registry for this application
    app = RegistryManager(app)

    # Static files (If running in production, and Apache
or another web
    # server is handling this static content, remove the
following 3 lines)
    static_app = StaticURLParser(config['pylons.paths']
['static_files'])
    app = Cascade([static_app, app])
    return app
```

Ważne jest miejsce, w którym wstawimy linijkę *app = authkit.authenticate.middleware(app, app_conf)*. Musi być umieszczona przed obsługą błędów *ErrorHandler*, gdyż część zabezpieczeń opiera się na manipulowaniu ich numerami. Od teraz Pylons wie już co to *AuthKit* i mamy wolną rękę w korzystaniu z tego narzędzia. Aby móc go użyć musimy jeszcze podać kilka parametrów w plikach konfiguracyjnych.

Konfiguracja

```
authkit.signin = /users/signin
authkit.setup.method = redirect
authkit.redirect.url = /users/signin

authkit.cookie.name = dArcookie
authkit.cookie.secret = &hs9#Ma*7N%a
authkit.cookie.signout = /users/signout
authkit.cookie.params = expires: 10
authkit.cookie.enforce = True
authkit.cookie.includeip = True
```

Podane linijki wklejamy do pliku *development.ini* w dziale *[app:main]*. Pierwsza z nich to informacja o tym pod jakim adresem znajduje się nasz formularz logowania. Dalej wybieramy metodę "autoryzacji". Ja postawiłem na *redirect*. Polega ona na przenoszeniu pod podany (w parametrze *authkit.redirect.url*) URL niezalogowanych lub nie posiadających odpowiednich uprawnień gości.

Kolejna seria parametrów dotyczy ciasteczek. Jego nazwa. Potem podajemy string, za pomocą którego będzie zaszyfrowane, URL pod którym będzie znajdowała się nasza funkcja wylogowująca, parametry (u nas czas wygaśnięcia). Opcja *enforce* ustawiona na *True* wymusza sprawdzenie wygaśnięcia ciasteczka również po stronie serwera zaś *includeip* zapisuje sobie adres przynależne do ciasteczka w celu zapobiegania używaniu tego samego cookie z innym adresem *IP*.

Tworzymy stronę logowania.

Umówmy się na plik *users/signin.xhtml*:

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

${h.form('authorize', method='post')}
  <fieldset>
    <legend>Formularz logowania do systemu</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", id="loginField")} </div>
    <div><label for="passwordField">Hasło: </label>$
{h.password("password", id="passwordField")}</div>
  </fieldset>
  <div>${h.submit("login", "Zaloguj")}</div>
${h.end_form()}
```

Nie ma tu właściwie nic skomplikowanego. Prosty formularz. Czas na model.

Dodajemy metodę logowania do naszego modelu.

Nasz mechanizm autoryzacji oprzemy o informację zawartą w sesji. Umówmy się, że o tym czy ktoś jest zalogowany czy nie będzie decydowało pole *session['user']['id']*. Metoda będzie miała za zadanie pobrać login i hasło użytkownika, sprawdzić czy istnieje odpowiadający ich wartościom

rekord w bazie danych, jeżeli tak - zapamiętujemy *id* użytkownika w sesji.

```
import hashlib
import sqlalchemy as sa
from sqlalchemy import orm

from darc.model import meta

class LoginExistsException(Exception):
    pass

class EmailExistsException(Exception):
    pass

class LoginOrEmailExistsException(Exception):
    pass

class UserDoesntExistsException(Exception):
    pass

t_user = sa.Table("users", meta.metadata,
    sa.Column("id", sa.types.Integer, primary_key=True,
autoincrement=True),
    sa.Column("login", sa.types.Unicode(25),
nullable=False, unique=True),
    sa.Column("email", sa.types.Unicode(150),
nullable=False, unique=True),
    sa.Column("password", sa.types.Unicode(128),
nullable=False),
    sa.Column("about",
sa.types.Text(convert_unicode=True), default=u"",
nullable=False),
    sa.Column("updated_at", sa.types.TIMESTAMP,
default=sa.func.current_timestamp()),
    sa.Column("created_at", sa.types.Date,
default=sa.func.now())
    )

class User(object):
    def loginExists(self):
        try:
            meta.Session.query(User).filter(User.login==se
lf.login).one()
        except orm.exc.NoResultFound:
            pass
        else:
            raise LoginExistsException()

    def emailExists(self):
        try:
            meta.Session.query(User).filter(User.email==se
lf.email).one()
```

```
            except orm.exc.NoResultFound:
                pass
            else:
                raise EmailExistsException()

    def save(self):
            self.password =
unicode(hashlib.sha512(self.password).hexdigest())
            meta.Session.save(self)
            try:
                meta.Session.commit()
            except sa.exc.IntegrityError:
                raise LoginOrEmailExistsException();

    def exists(self):
            conditions = meta.Session.query(User)
            for key, value in self.__dict__.iteritems():
                if value and str(key)[0] != '_':
                    conditions =
conditions.filter(getattr(User, key)==value)
            try:
                user = conditions.one()
            except orm.exc.NoResultFound:
                raise UserDoesntExistsException()

            self.id = user.id

orm.mapper(User, t_user)
```

Metoda udaje "dość sprytną". Przechodzi ona po nie pustych elementach klasy i dodaje ich wartości jako warunki do zapytania. Zastosowanie wielokrotnego *filter* połączy całość spójnikiem *AND* dzięki czemu pobierzemy użytkownika spełniającego wszystkie kryteria. Zapis *str(key)[0] != '_'* służy eliminacji pewnego pola dodanego przez *SQLAlchemy*, które nie jest powiązane z żadnym z atrybutów naszej tabeli MySQL. Jeżeli rekord nie zostanie znaleziony *(NoResultFound)*, zgłosimy wyjątek mówiący o tym iż użytkownik nie istnieje. Musimy go obsłużyć i zignorować. Na koniec sprawa techniczna - ponieważ pliki się rozrastają od teraz będę już tylko mówił co modyfikujemy i wklejał wyłącznie zmienione fragmenty plików. Koniec komunikatów technicznych.

Dane o użytkowniku warto trzymać zapisane pod osobnym kluczem (u nas *user*). Przyda się to szczególnie wtedy gdy będziemy chcieli zapamiętać większą ilość informacji o użytkowniku. Taki podział na pewnego rodzaju przestrzenie nazw w bardziej rozbudowanych aplikacjach pozwala na utrzymanie porządku.

Dodatkowo informacja dla mniej spostrzegawczych. Metodę *exists* możemy wykorzystać w miejscu implementacji naszych ajaxowych *emailExists* czy *loginExists* o walidatorach sprawdzających obecność adresu skrzynki czy nazwy użytkownika w bazie danych nie wspominając.

Warto zauważyć wykorzystanie introspekcyjności języka jakim jest Python. Zamiast korzystać z niebezpiecznej instrukcji exec i konstruować potworka w stylu: *exec('conditions = conditions.filter(User.%s=="%s")' % (key, value))* możemy użyć odpowiednich mechanizmów dostarczonych przez język i w znacznie bezpieczniejszy i elegancki sposób sprostać zadaniu sprytnie wykorzystując: *conditions = conditions.filter(getattr(User, key)==value)*.

Identyfikacja w kontrolerze

Jedyne co musimy zrobić to zaimportować potrzebne nam klasy wyjątków i stworzyć akcję *authenticate,* która przekaże modelowi dane z formularza i wywoła odpowiednią metodę. Po odpytaniu modelu o to czy użytkownik o zadanych kryteriach istnieje w bazie pobieramy *id* i zapisujemy w sesji. Należy tutaj pamiętać o dwóch rzeczach. Primo: nie wolno zapomnieć o wywołaniu *session.save()*. Bez tego nasze zmiennej sesji nie zostaną zachowane. Secundo - musimy obsłużyć wyjątek *KeyError*, który pojawi się gdy będziemy przypisywać wartość do nie istniejącego jeszcze klucza. Obsłużenie wyjątku *UserDoesntExistsException* uznaję za oczywiste.

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

{...}

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

    def authenticate(self):
        user = User()
        user.login = request.POST['login']
        user.password = request.POST['password']
        try:
            user.exists()
        except UserDoesntExistsException:
            flash('Niepoprawne dane autoryzacyjne. Sprawdź
swój login i hasło.')
        else:
            flash('Witamy w systemie')
            try:
                session['user'] = {'id': user.id}
            except KeyError:
                pass
            else:
                session.save()

    def signout(self):
        try:
          del(session['user'])
        except KeyError:
          pass
        else:
          session.save()

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

Na końcu przekierowujemy użytkownika do akcji *index*. Przy okazji w kodzie dodałem jeszcze dwie metody. Pierwsza jest odpowiedzialna za wyświetlenie formularza do logowania a druga za usunięcie sesji, która odpowiada za oznaczenie użytkownika jako autoryzowanego. Kiedy możemy się już logować, a w meandrach *middleware* drzemie skonfigurowany i gotowy do pracy *AuthKit* czas połączyć nasze rozwiązania.

Zabezpieczamy akcje

Ten i kolejny podpunkt będzie pokazywał jak zaadaptować do naszych rozwiązań *AuthKit*.

Authkit pozwala nam na bardzo przyjemną i bezpieczną metodę zabezpieczania naszych metod. Przyjmuje Ona jeden parametr - klasę. Zanim akcja, opatrzona odpowiednim dekoratorem, zostanie wykonana *middleware AuthKita* wywoła metodę *check()* obiektu, która w razie stwierdzenia braku

dostępu zwróci wyjątek. Ponieważ wykorzystujemy mechanizm *redirect* (parametr *authkit.setup.method*) zostaniemy natychmiast przekierowaniu pod url wskazany w opcji *authkit.redirect.url*. Skoro znasz już metodę działania zastanówmy się, które akcje powinny być chronione. Na pewno nie może być to *signin*. Odpada również *register*, *create* czy *authenticate*. Właściwie do kandydatów można włączyć jedynie metodę *index* oraz, co mogłoby wydawać się wręcz intuicyjne *signout*. Wszystkie inne metody używane są przez użytkowników niezalogowanych. Zaimportujemy więc dekorator i do dzieła.

```
from pylons.decorators import validate
from authkit.pylons_adaptors import authorize
from darc.lib.base import BaseController, render
from darc.lib.helpers import flash
#from darc import model
from darc.model.form import RegisterForm, UpdateForm,
ChangePasswordForm
from darc.model.user import User,
LoginOrEmailExistsException, LoginExistsException,
EmailExistsException, UserDoesntExistsException
from darc.model.authenticated_user import
AuthenticatedUser

{...}

    @authorize(AuthenticatedUser())
    def signout(self):
        try:
          del(session['user'])
        except KeyError:
          pass
        else:
          session.save()

        return redirect_to(action='signin')

{...}

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

Jak zauważyłeś jako parametr autoryzacji przekazaliśmy nową, nie istniejącą jeszcze klasę. Uważniejsi zauważą iż ze wcześniejszymi deklaracjami powinien on posiadać metodę *check()*. To będzie nasz ostatni przystanek.

{...} oznacza pominięty fragment kodu. Nie zapisujemy tego w naszych plikach z kodem programu.

Klasa autoryzacji

Kod który teraz napiszemy jest wzorowany na klasie *RemoteUser* zawartej w źródłach *AuthKit*. Pliczek *model/authenticated_user.py* (nie będę kolorował wszystkie na żółto - calutka jest nowa):

```
from pylons import session
from authkit.permissions import RequestPermission,
NotAuthenticatedError, NotAuthorizedError

class AuthenticatedUser(RequestPermission):

    def __init__(self, accept_empty=False):
        self.accept_empty = accept_empty

    def check(self, app, environ, start_response):
        if 'user' not in session:
            raise NotAuthenticatedError('Not
Authenticated')
        elif self.accept_empty==False and not 'id' in
session['user']:
            raise NotAuthorizedError('Not Authorized')
        return app(environ, start_response)
```

Aby uzyskać efekt potrzebne było nam pobranie kilku klas z modułu *permissions* naszego narzędzia. Dziedziczymy po *RequestPermission*. Dodaliśmy również dwie metody. Zarówno metoda *__init__()* jak i *check()* to trawest z wzorowany na *UserRemote* który czyni nasz model kompatybilnym z klasą, użyteczną w dekoratorze *@autorize*. Zadaniem *check()* jest sprawdzenie czy w sesji znajdują się odpowiednie klucze a następnie, w przypadku stwierdzenia ich braku, zwrócenie wujątku. Teraz nasza strona obsługuje pełną autoryzację użytkowników. Aby zalogować się do systemu wywołaj *users/signin*. Wylogowania możesz zażądać przechodząc na stronę *users/signout* zaś to czy wszystko działa sprawdzić na metodzie *users/index*.

Modny błąd ...

Nasza aplikacja może wydawać się już całkiem "bezpieczna". Czy słyszałeś jednak o atakach *CSRF*[19]? Niestety - w obecnej formie stworzony przez nas jest na nią wciąż podatna. Dzięki Pylons możemy się jednak błyskawicznie zabezpieczyć.

Dodatkowy helper

Zanim zaczniemy przyda nam się dodatkowy helper:

19 http://pl.wikipedia.org/wiki/Cross-site_request_forgery

```
"""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.secure_form import secure_form
from webhelpers.html.tags import form, text, password,
submit, end_form, stylesheet_link, javascript_link
from webhelpers.pylonslib import Flash as _Flash
flash = _Flash()
```

Zmiany w widok

Pierwszym etapem jest wygenerowanie bezpiecznego formularza właśnie zaimportowaną funkcją. Dokonajmy więc zmian w plik *users/signin.xhtml*

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

${h.secure_form('authenticate', method='post')}
  <fieldset>
    <legend>Formularz logowania do systemu</legend>
    <div><label for="loginField">Login: </label>$
{h.text("login", id="loginField")} </div>
    <div><label for="passwordField">Hasło: </label>$
{h.password("password", id="passwordField")}</div>
  </fieldset>
  <div>${h.submit("login", "Zaloguj")}</div>
${h.end_form()}
```

Jeżeli zajrzysz w źródła okaże się, że zostało dodane ukryte pole o pewnej losowej wartości. Jednak nasz kontroler nie potrafi jeszcze z tego korzystać. Zaradźmy temu.

Na straży kontrolera - pełna ochrona

Kolejnym ogniwem zabezpieczenia *Pylons* jest dekorator o nazwie *authenticate_form*. Wystarczy zaimportować go i użyć przed akcją, która odbiera dane.

```
from pylons.decorators import validate
from pylons.decorators.secure import authenticate_form

{...}

    @authenticate_form
    def authenticate(self):
        user = User()
        user.login = request.POST['login']
        user.password = request.POST['password']
        try:
            user.exists()
        except UserDoesntExistsException:
            flash('Niepoprawne dane autoryzacyjne. Sprawdź
swój login i hasło.')
        else:
            flash('Witamy w systemie')
            try:
                session['user'] = {'id': user.id}
            except KeyError:
                pass
            else:
                session.save()

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

Od teraz wszystkie formularze, które będą dotyczyły zalogowanego użytkownika, aby były bezpieczne, muszą być tworzone w taki właśnie sposób.

Licencja

Spis treści