

# Grok - szybkie tworzenie aplikacji webowych

Wojciech Lichota

Grok jest frameworkiem który daje mnóstwo frajdy podczas tworzenia aplikacji webowych. Osiągnięto to poprzez połączenie prostoty jego użycia z bogactwem oferowanych możliwości. Grok czerpie to, co najlepsze z Zope – najbardziej wszechstronnego serwera aplikacji napisanego w Pythonie. Dokłada do tego łatwą do nauczenia, a zarazem logiczną i przejrzystą składnię. Wszystkie te cechy powodują, że Grok jest przyjazny zarówno dla nowicjuszy jak i dla zaprawionych w bojach deweloperów. Niniejszy artykuł jest rozszerzeniem wykładu przedstawiającego podstawy tego środowiska.

## 1. Grok jest prosty w nauce.

Efekt prostoty otrzymano głównie dzięki temu, że świat Grok'a poznajemy stopniowo - robiąc małe kroki. Od razu po poznaniu nowej funkcjonalności można ją wykorzystać i zobaczyć jej rezultat. Takie podejście ułatwia naukę, ponieważ na bieżąco widzimy efekty naszej pracy.

Aby przekonać się o tym samemu polecam skorzystanie z samouczka (ang. tutorial) [1] zamieszczonego na stronie domowej projektu Grok. W tym dokumencie opisano jak zainstalować framework i utworzyć pusty projekt – szkielet dla nowej aplikacji webowej.

## 2. Grok jest prosty w budowie.

Aplikację webową w Groku opisuje się wyłącznie przy użyciu kodu Pythona. Wykorzystano tutaj podejście oparte o programowanie obiektowe [2] i architekturę MVC (ang. *Model-View-Controller*) [3] w nieco zmodyfikowanej postaci MTV (ang. *Model-Template-View*).

Model danych opisywany jest przy użyciu `zope.interface`:

```
import grok
from zope.interface import Interface
from zope.schema import TextLine, Text

class IDocument(Interface):
    """
    Document schema definition.
    """
    title = TextLine(title=u'Title', required=True)
    text = Text(title=u'Text of document', required=False)
```

Na podstawie tego modelu danych budowana jest implementacja – czyli faktyczny model. Grok wykorzystuje obiektową bazę danych ZODB [4] do przechowywania danych. Dzięki temu model może zawierać dowolne typy danych dostępne w języku Python. Nie jest wykorzystywana relacyjna baza danych, a więc nie trzeba się martwić czy dane z modelu możliwe są do zapisania z użyciem języka SQL. Daje to programiście pełną swobodę w definiowaniu struktur danych.

```
class Document(grok.Model):
    """
    Object that can contain text.
    """
    grok.implements(IDocument)
    title = text = ''
```

Reprezentacja modelu pokazywana użytkownikowi jest tworzona przy użyciu klasy widoku (*View*). W przypadku aplikacji webowych widok powinien generować kod możliwy do wyświetlenia przez przeglądarki internetowe. W Groku to klasa widoku zawiera większość kodu pozwalającego przekształcić dane zapisane w modelu na dane możliwe do osadzenia w szablonie.

```
from zope.dublincore.interfaces import IZopeDublinCore

class Index(grok.View):
    """
```

```

Default document view
"""
def getModificationTime(self):
    dc_data = IZopeDublinCore(self.context)
    formatter = self.request.locale.dates.getFormatter('date')
    return formatter.format(dc_data.modified)

```

Ostatnią warstwą architektury MTV jest szablon (*Template*). Grok domyślnie oferuje system szablonów ZPT (Zope Page Templates) [5], aczkolwiek w prosty sposób można wykorzystać swój ulubiony generator. Przykładowy kod ZPT takiego szablonu:

```

<html>
  <body>
    <div class="document.title" tal:content="context/title">
      Title
    </div>
    <div class="document.text" tal:content="structure context/text">
      Text
    </div>
    <div class="document.modified" tal:content="view/getModificationTime">
      Modification Time
    </div>
  </body>
</html>

```

Innym typem widoku są formularze pozwalające na edycję danych. Formularze w Groku generowane są automatycznie na podstawie modelu zdefiniowanego w interfejsie.

```

class Edit(grok.EditForm):
    """
    Edit a document.
    """
    form_fields = grok.AutoFields(IDocument)

    @grok.action(_('Save'))
    def save(self, **kw):
        self.applyData(self.context, **kw)

```

### 3. Grok jest prosty w konfiguracji.

Zastosowano tu regułę DRY (Don't Repeat Yourself) [6]. Tak więc nie jest konieczna dodatkowa konfiguracja aplikacji. Nie są potrzebne żadne pliki XML'owe, żadne reguły przetwarzania URL'i czy aktywatory, wystarczy trzymać się wyznaczonej konwencji.

Przedstawione powyżej klasy (tj.: IDocument, Document, Index, Edit) powinny znaleźć się w pliku `src/document.py`. Szablon natomiast w pliku `src/document_templates/index.pt`. Grok automatycznie skonfiguruje aplikację - ustawi klasę o nazwie `Index` domyślnym widok, a klasę `Edit` domyślnym formularzem edycji zdefiniowanego w tym samym pliku modelu. W podobny sposób zlokalizuje szablon.

Używanie konwencji jednak nie jest obowiązkowe. Jeżeli chcemy, aby któryś z komponentów miał inne zachowanie niż wynikało by to z konwencji, w prosty sposób nadpisać część lub całą konfigurację.

Konfiguracja samego środowiska (czyli serwera aplikacji) w którym uruchomiona będzie aplikacja webowa została opisana w pliku `buildout.cfg`. Takie parametry jak np. port na którym działa serwer, bądź wersja Grok użyta w projekcie znajdują się właśnie w tym pliku. Domyślny plik z konfiguracją tworzony jest przez komendę `grokproject`, a następnie jest on przetwarzany przy użyciu `zc.buildout` [9]. Poprzez wykorzystanie tak zwanych recept (ang. *recipe*) możliwe jest zwiększenie parametrów jakie możliwych do skonfigurowania poprzez plik `buildout.cfg`

### 4. Grok jest prosty w rozbudowie.

Framework Grok zbudowano na bazie serwer aplikacji Zope 3 [7]. Zope w wersji 3 tworzony jest od 2001 roku. Został napisany od podstaw przy użyciu architektury opartej o komponenty (ang. *component architecture*) [8]. Dzięki tej

architekturze możliwe jest osiągnięcie dużej modularności. Przyczyniło się to do powstania bardzo dużej liczby dodatkowych pakietów modyfikujących bądź rozszerzających funkcjonalność Zope.

Opis wszystkich właściwości aplikacji webowej znajduje się w pliku `setup.py` opisującym pakiet zgodnie z wytycznymi `distutils` [10]. Aby móc wykorzystać dodatkowy moduł w swoim projekcie wystarczy dopisać nazwę tego pakietu do listy zależności (`install_requires`).

Na zakończenie należy ponownie uruchomić skrypt projekt tworzący instancję:

```
$ ./bin/buildout
```

## 5. Grok jest prosty w rozwijaniu.

Architektura komponentowa pozwala na łatwą rozbudowę istniejącego modelu o nowe funkcjonalności. W toku prac najczęściej konieczne jest powiększenie modelu o kolejne atrybuty. Można to oczywiście zrobić w samym modelu (i interfejsie), ale po pewnym czasie może to doprowadzić do skomplikowania modelu. Innym przypadkiem przemawiającym za wykorzystaniem komponentów jest chęć ponownego wykorzystania wspólnych atrybutów przez wiele modeli. Można to osiągnąć poprzez wielokrotne dziedziczenie, ale to niepotrzebnie czyni hierarchię klas bardziej złożoną – utrudniając zarządzanie całym projektem. W opisywanym przypadku najlepszym rozwiązaniem będzie wykorzystanie adapterów. Adaptery to swego rodzaju konwertery pozwalające na przekształcenie danego modelu na inny format.

Opisany powyżej model `Document` zawiera dwa podstawowe atrybuty – tytuł i treść. Model ten implementuje bezpośrednio interfejs `IDocument`. Inny interfejs – `IZopeDublinCore` opisuje dodatkowe atrybuty (takie jak: twórca, opis, język, data modyfikacji) zgodnie z wytycznymi standardu Dublin Core [11]. Aby móc przypisać do modelu dokumentu tego typu dane konieczne jest aby istniał adapter konwertujący model na nowy interfejs.

Jak można zauważyć architektura komponentowa pozwala zatem, na wydzielenie pewnej funkcjonalności do oddzielnego modułu. Podobnie w tym przypadku – mechanizm odpowiedzialny za operacje na danych według standardu Dublin Core został wydzielony do modułu `zope.dublincore`.

## 6. Podsumowanie.

Grok jest frameworkiem pozwalającym szybko tworzyć aplikacje webowe, zarówno na początku jak i w dalszych etapach prac. Aby zacząć naukę nie jest wymaga specjalistycznej wiedzy z dziedziny tworzenia aplikacji webowych. Poznając małymi porcjami kolejne możliwości oferowane przez Grok'a można jednocześnie je wypróbować, eksperymentować, bawić się. Jednocześnie oferuje on potencjał jaki drzemie w Zope 3 – bardzo duża ilość dodatkowych modułów jakie dotychczas powstały i komponentowa architektura pozwala na łatwiejszy i szybszy rozwój tego środowiska.

- [1] - <http://grok.zope.org/documentation/book/>
- [2] - <http://pl.wikipedia.org/wiki/OOP>
- [3] - <http://pl.wikipedia.org/wiki/MVC>
- [4] - <http://plone.org/documentation/tutorial/introduction-to-the-zodb>
- [5] - <http://plone.org/documentation/tutorial/zpt>
- [6] - [http://pl.wikipedia.org/wiki/DRY\\_\(reguła\)](http://pl.wikipedia.org/wiki/DRY_(reguła))
- [7] - [http://en.wikipedia.org/wiki/Zope\\_3](http://en.wikipedia.org/wiki/Zope_3)
- [8] - <http://www.muthukadan.net/docs/zca.html>
- [9] - <http://pypi.python.org/pypi/zc.buildout>
- [10] - <http://docs.python.org/dist/module-distutils.core.html>
- [11] - [http://pl.wikipedia.org/wiki/Dublin\\_Core](http://pl.wikipedia.org/wiki/Dublin_Core)