Rozwiązania szkieletowe w tworzeniu aplikacji www

Tomasz Łukaszuk

Katedra Oprogramowania
Wydział Informatyki
Politechnika Białostocka

Temat wykładu:

Template

Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską
NCBR Narodowe Centrum Badań i Rozwoju

Materiały przygotowane w ramach projektu "PB 5.0 – dostosowanie oferty dydaktycznej Politechniki Białostockiej do potrzeb nowoczesnej gospodarki oraz zielonej i cyfrowej transformacji" (nr umowy FERS.01.05-IP.08-0327/23-00) w ramach programu Fundusze Europejskie dla Rozwoju Społecznego 2021-2027 współfinansowanego ze środków Europejskiego Funduszu Społecznego Plus.

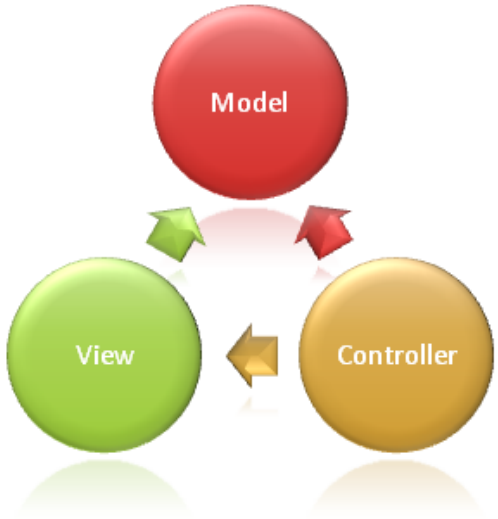
Template: Agenda

1. MVC - Widok
2. Django Template System
   - Wprowadzenie
   - Template'y dla designer'ów
   - Template'y dla programistów
   - Praktyka
3. Szablony i widoki w RoR
   - Rendering
   - Pliki widoków
   - Szablony (layouts)

MVC - Widok

MVC: Widok - przypomnienie

- odpowiada za prezentację, wyświetlanie danych użytkownikom
- może prezentować wyniki w postaci HTML, XML, WML, obrazków, plików PDF, itd.
- wykorzystuje model do pobrania danych, które będą wyświetlone (tworzy instancje klas modeli i wywołuje z nich metody)
  - nie powinien zmieniać stanu aplikacji
  - może być implementowany w postaci wielu klas, po jednej na każdy zwracany element
  - w dobrze zaprojektowanej aplikacji widok powinien być "wymienny"



Wymienność widoku

- generowanie różnej odpowiedzi (format, wygląd) dla różnych odbiorców
- formaty odpowiedzi: HTML, XML, JSON, PDF, RSS
- zmiana widoku wykonywana tylko przez zamianę klas implementujących sposób wyświetlania danych
- wymienność widoku ma obecnie coraz większe znaczenie, ponieważ od aplikacji zaczynamy wymagać obsługi formatów wyjściowych innych niż tradycyjny HTML, takich jak: PDF do drukowania, RSS do newsów, WML dla użytkowników telefonów, SOAP i XML-RPC dla implementacji Web Services
- wraz z rozwojem informatyki będą pojawiały się nowe technologie wyjściowe

Zalecenia odnoszące się do widoku w aplikacjach MVC

- kategorycznie Widok nie powinien zmieniać danych dostarczanych do niego w celu wyświetlenia
- Widok w żaden sposób nie powinien zmieniać stanu aplikacji

Django Template System

Wprowadzenie

Co to są szablony?

- Szablon jest po prostu plikiem tekstowym, który silnik szablonów może użyć do wygenerowania DOWOLNEGO formatu tekstowego (HTML, XML, CSV, JavaScript itd.)
- `{{ variables }}` są przekazywane do szablonów jako normalny słownik par `variable : value` i są zastępowane wartościami gdy szablon jest przetwarzany
- `{{ variables }}` mogą być modyfikowane przez filtry w następujący sposób: `{{ variable|filter }}`
- `{% tags %}` sterują logiką szablonu
- `{% tags %}` działają podobnie do konstrukcji programistycznych . . .
- . . . ale nie są one po prostu kodem Pythona wbudowanym w HTML.
- Filozofia: system szablonów jest przeznaczony do wyrażania prezentacji, a nie logiki programu

Przykład

Zauważ użycie kombinacji `{{ variables }}`, `{% tags %}` i `{{ variable|filter }}`

```
1  {% extends "base_generic.html" %}
2
3  {% block title %}{{ section.title }}{% endblock %}
4
5  {% block content %}
6  <h1>{{ section.title }}</h1>
7
8  {% for story in story_list %}
9  <h2>
10   <a href="{{ story.get_absolute_url }}">
11     {{ story.headline|upper }}
12   </a>
13 </h2>
14 <p>{{ story.tease|truncatewords:"100" }}</p>
15 {% endfor %}
16 {% endblock %}
```

Podstawowe zastosowanie w funkcjach widoków

```
1  from django.shortcuts import render
2
3  def my_view(request):
4      # View code here...
5      return render(request, 'myapp/index.html', {"foo": "bar"},
6          mimetype="application/xhtml+xml")
```

`render()` jest funkcją helper. Bez tego powinniśmy użyć konstrukcji:

```
1  from django.http import HttpResponse
2  from django.template import loader
3
4  def my_view(request):
5      # View code here...
6      t = loader.get_template('myapp/index.html')
7      c = {'foo': 'bar'}
8      return HttpResponse(t.render(c, request), content_type='application/xhtml+xml'
```

Należy pamiętać, że ścieżka do szablonu jest względna

Ustawienia silnika szablonów

```
1  # settings.py
2  TEMPLATES = [
3      {
4          'BACKEND': 'django.template.backends.django.DjangoTemplates',
5          'DIRS': [],
6          'APP_DIRS': True,
7          'OPTIONS': {
8              # ... some options here ...
9          },
10     },
11 ]
```

- `BACKEND` - ścieżka do klasy silnika szablonów implementującego API backendu szablonów Django
- `DIRS` - definiuje listę katalogów, w których silnik powinien szukać plików źródłowych szablonów, w kolejności wyszukiwania
- `APP_DIRS` - mówi, czy silnik powinien szukać szablonów wewnątrz zainstalowanych aplikacji, w przypadku beckendu `DjangoTemplates` w katalogu `templates` w katalogu aplikacji
- `OPTIONS` - zawiera ustawienia specyficzne dla backendu

Template'y dla designer'ów

Więcej na temat zmiennych

- Gdy silnik szablonów napotka zmienną, ewaluuje ją i zamienia na wartość uzyskanego wyniku.
- Użyj kropki, by odwołać się do atrybutów zmiennej (np. `{{ section.title }}`). Następujące warianty odwołań będą rozważane:
  1. odwołanie do słownika
  2. odwołanie do atrybutu
  3. odwołanie do metody
  4. odwołanie do indeksu listy
- Jeśli zmienna nie istnieje, stystem wstawi wartość ustawienia `TEMPLATES - OPTIONS - string_if_invalid` (domyślnie pusty napis)

Więcej na temat filtrów

- Filtry służą do modyfikacji zmiennych
- Filtry mogą być łączone w łańcuchy, np. `{{ text|escape|linebreaks }}`
- Niektóre filtry mogą przyjmować argumenty, np. `{{ list|join:", " }}` (tylko argumenty ze spacjami muszą być umieszczone w cudysłowiu)
- Popularne filtry:

```
1  {{ value|default:"nothing" }}
```

Jeżeli `value` jest pusta lub jej brak, wyświetl "nothing"

```
1  {{ value|length }}
```

Jeżeli `value` to "cat", zostanie zwrócone 3

```
1  {{ value|striptags }}
```

Jeżeli `value` to "Joel<button>is</button> a slug", zostanie zwrócone "Joel is a slug"

Więcej na temat tagów

- Niektóre tagi generują tekst, niektóre sterują wykonywaniem kodu obsługując pętle i logikę, niektóre ładują do szablonu informacje z zewnątrz
- Niektóre tagi wymagają tagu początkowego i końcowego:

  `{% tag %} ... tag contents ... {% endtag %}`
- Popularne tegi:

```
1  {% for athlete in athlete_list %}
2      <li>{{ athlete.name }}</li>
3  {% endfor %}
```

Pętla po każdej pozycji w sekwencji.

```
1  {% if athlete_list %}
2      Number of athletes: {{ athlete_list|length }}
3  {% else %}
4      No athletes.
5  {% endif %}
```

Ewaluuje zmienną i wyświetla jej zawartość jeżeli ewaluacja zwróci "True"

Dziedziczenie szablonów

- Najmocniejsza i najbardziej złożona część silnika szablonów
- Pozwala zbudować bazę "szkielet" szablonów zawierający wszystkie wspólne elementy witryny i definiujący bloki, które szablon-potomek może zmienić
- Rozszerz szablon używając tagu `{% extends %}` np.
  ```
  {% extends "base.html" %}
  ```
- Zdefiniuj bloki używając tagu `{% block %}` np.
  ```
  {% block content %}{% endblock %}
  ```

Przykład dziedziczenia

base.html

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4  <head>
5      <link rel="stylesheet" href="style.css" />
6      <title>{% block title %}My amazing site{% endblock %}</title>
7  </head>
8  <body><div id="content">
9          {% block content %}{% endblock %}
10 </body></div>
```

szablon potomny:

```
1  {% extends "base.html" %}
2
3  {% block title %}My amazing blog{% endblock %}
4  {% block content %}
5  {% for entry in blog_entries %}
6      <h2>{{ entry.title }}</h2>
7  {% endfor %}
8  {% endblock %}
```

Porady na temat dziedziczenia

- Jeżeli używamy dziedziczenia, `{% extends %}` musi być pierwszym tagiem w szablonie
- Szablony potomne nie muszą definiować wszystkich bloków rodzica.
- Jeżeli powtarzasz zawartość w wielu szablonach, powinieneś przenieść tą zawartości do `{% block %}` w szablonie nadrzędnym
- Pobranie zawartości z bloku szablonu nadrzędnego odbywa się poprzez użycie zmiennej `{{ block.super }}` .
- Dla poprawy czytelności, zastosuj *nazwę* do `{% endblock %}`, w następujący sposób:

```
1   {% block content %}
2   ...
3   {% endblock content %}
```

Automatyczny HTML escaping

Rozważmy ten fragment szablonu:

```
1   Hello, {{ name }}.
```

Jeśli użytkownik wprowadzi swoje nazwisko jako:

```
1   <script>alert('hello')</script>
```

szablon będzie wyświetlany jako:

```
1   Hello, <script>alert('hello')</script>
```

i przeglądarka wyświetli JavaScript'owe okno alertu!

- Danym przesyłanym przez użytkownika nie można ufać
- Domyślnie, Django konwertuje każdy znacznik ze zmiennej ("<" jest konwertowane "&lt", itp.)
- Takie zachowanie może być wyłączone dla zmiennej lub bloku.

Biblioteki niestandardowych tagów i filtrów

Tag `{% load %}` umożliwia dostęp do niestandardowych tagów i filtrów:

```
1  {% load comments %}
2
3  {% comment_form for blogs.entries entry.id with is_public yes %}
```

- `{% load %}` może pobrać kilka nazw bibliotek na raz
- Ma to wpływ tylko na bieżący szablon

Template'y dla programistów

Jak działają szablony?

Korzystanie z systemu szablonów w Pythonie jest procesem dwuetapowym:

1. Kompilacja surowego kodu szablonu do obiektu Template.

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print t
<django.template.Template instance>
```

2. Wywołanie metody `render()` na obiekcie Template z określonym kontekstem.

```
>>> c = Context({"my_name": "Adrian"})
>>> t.render(c)
"My name is Adrian."

>>> c = Context({"my_name": "Dolores"})
>>> t.render(c)
"My name is Dolores."
```

Context Processors i `RequestContext`

`RequestContext` pobiera `request` jako pierwszy argument:

```
1  c = RequestContext(request, {
2      'foo': 'bar',
3  })
```

Wypełnia on kontekst zmiennymi dostarczonymi przez context processors zdefiniowanymi w ustawieniu `TEMPLATES - OPTIONS - context_processors`:

```
1  [
2      'django.template.context_processors.debug',
3      'django.template.context_processors.request',
4      'django.contrib.auth.context_processors.auth',
5      'django.contrib.messages.context_processors.messages',
6  ]
7
8  # plus zawsze 'django.template.context_processors.csrf'
```

- Context processors wypełniają kontekst danymi z `request`
- Context processors zastąpią zmienne dostarczone przez użytkownika, jeżeli ich nazwy pokrywają się.

Pisanie własnych bibliotek tagów i filtrów

Własne tagi i filtry powinny być umieszczone w katalogu "templatetags" aplikacji, np.:

```
polls/
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

Aby być prawidłową biblioteką tagów poll_extras.py musi posiadać następującą zmienną:

```
from django import template

register = template.Library()
```

Zmienna ta będzie wykorzystana do rejestracji wszystkie własnych tagów i filtrów.

Pisanie własnych filtrów

Własne filtry są funkcjami Pythona, które mają jeden lub dwa argumenty:

- Wartość zmiennej (wejście) - niekoniecznie string.
- Wartość argumentu - może mieć wartość domyślną, lub być całkowicie pominięta.

Przykłady:

```
1  def cut(value, arg):
2      "Removes all values of arg from the given string"
3      return value.replace(arg, '')
```

```
1  def lower(value): # Only one argument.
2      "Converts a string into all lowercase"
3      return value.lower()
```

Rejestracja filtrów:

```
1  register.filter('cut', cut)
2  register.filter('lower', lower)
```

Pisanie własnych tagów

- Tagi są o wiele bardziej złożone niż filtry, ponieważ mogą one zrobić wszystko.
- Pisanie własnych tagów od podstaw jest skomplikowane
- Na szczęście, Django dostarcza funkcji pomocniczych pozwalających na pisanie powszechnie stosowanych rodzajów tagów: tagi proste i tagi integracji (inclusion)

Tagi proste

Tagi proste pobierają ciąg argumentów i zwracają string po wykonaniu określonego przetwarzania.
Przykład: tag który zwraca aktualny czas sformatowany zgodnie z podanym argumentem:

```
1           <p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

Tag ten może być zapisany jako:

```
1   def current_time(format_string):
2       return datetime.datetime.now().strftime(format_string)
3
4   register.simple_tag(current_time)
```

lub, z użyciem składni dekoratora z Python 2.4:

```
1   @register.simple_tag
2   def current_time(format_string):
3       return datetime.datetime.now().strftime(format_string)
```

Tagi typu inclusion

Tagi inclusion tags wyświetlają jakieś dane przez renderowanie innego szablonu. Przykład: tag, który wyświetla wyniki ankiety:

```
{% show_results poll %}
```

Tag ten może być zapisany jako:

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

i zarejestrowany:

```
register.inclusion_tag('results.html')(show_results)
```

results.html:

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

Praktyka

Praktyka

- katalog `templates` w katalogu aplikacji

```
mainapp
    templates
        base.html
        main_page.html
        other_page.html
    models.py
    views.py
    ...
```

- w funkcjach widoków renderowanie szablonów z użyciem funkcji `render`, drugi argument ścieżka do szablonu, ścieżka względna od katalogu `templates`

```
1  from django.shortcuts import render
2
3  def show_main_page(request):
4      users = User.objects.all()
5      return render(request, "main_page.html", {"users":users})
```

Praktyka

- base.html - szablon główny - layout strony

```
1  <html>
2      <head>
3          <title>{% block title %}My page{% endblock %}</title>
4      </head>
5
6      <body>
7          {% block body_content %}Default content.{% endblock %}
8      </body>
9  </html>
```

- podstrony - szablony dziedziczące po szablonie głównym other_page.html

```
1  {% extends "base.html" %}
2
3  {% block title %}My other page{% endblock %}
4  {% block body_content %}Concrete content{% endblock %}
```

Szablony i widoki w RoR

Podstawy widoków w RoR

- metoda ActionController::Base#render
- moduł ActionView
- folder app/views/controller_name
- pliki action_name.html.erb
- folder app/views/layouts

Rendering

Tworzenie odpowiedzi `HTTP response` I

- trzy sposoby tworzenia odpowiedzi: render, redirect_to, head
- domyślnie kontroler automatycznie renderuje widok, którego nazwa jest taka jak nazwa akcji

```
1  class BooksController < ApplicationController
2     def show
3        @book = Book.find(params[:id])
4        #automatycznie renderuje app/views/books/show.html.erb
5     end
6  end
```

- użycie metody `render`

```
1  render :nothing => true
2  render "name"     #akcja (template) w tym samym kontrolerze
3  render :template => 'controller_name/template_name'
4  render :file => "/path/to/file/filename"
5  render :text => "ok"
6  render :xml => @product
```

Tworzenie odpowiedzi `HTTP response` II

- opcje metody `render`:
  :content_type
  :layout
  :status
  :location

```
1  render :file => filename, :content_type => 'application/rss'
2  render :layout => 'special_layout'
3  render :status => 500
4  render :xml => photo, :location => photo_url(photo)
```

Unikanie błędu podwójnego renderowania

- komunikat błędu ”Can only render or redirect once per action”
- przykład błędu

```
1  def show
2     @book = Book.find(params[:id])
3     if @book.special?
4        render :action => "special_show"
5     end
6  end
```

- użycie and return

```
1  def show
2     @book = Book.find(params[:id])
3     if @book.special?
4        render :action => "special_show" and return
5     end
6  end
```

Przekazywanie parametrów z kontrolera do widoku

- zmienna instancji (zaczynająca się znakiem @) w kontrolerze jest dostępna w renderowanym widoku

Listing 1: app/controllers/books_controller.rb

```
1  class BooksController < ApplicationController
2     def index
3        @library = "University Library"
4        @books = Book.all
5     end
6  end
```

Listing 2: app/views/books/index.html.erb

```
1  <%= @library %><br />
2  <% @books.each do |book| %>
3     <%= h book.author %>: <%= h book.title %><br />
4  <% end %>
```

Pliki widoków

Zawartość pliku widoku

- HTML z wbudowanym kodem języka Ruby
- specjane tagi w html'u
  <% kod języka Ruby %>
  <%= wyrażenie języka Ruby %>
  <%# komentarz %>

```
1  <%# view file example %>
2  <h1><%= @name %> (<%= @code %>)</h1>
3  <p><%= @desc %></p>
4
5  <ul>
6     <% @features.each do |f| %>
7        <li><b><%= f %></b></li>
8     <% end %>
9  </ul>
```

Szablony (layouts)

Co to jest layout?

- layout - definiuje otoczenie strony HTML
- layout - miejsce do definiowania wspólnych elementów wyglądu i zachowania stron naszej aplikacji
- pliki layout'ów umieszczane są w `app/views/layouts`

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
5  <head>
6    <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
7    <title>Books: <%= controller.action_name %></title>
8    <%= stylesheet_link_tag 'scaffold' %>
9  </head>
10 <body>
11
12 <p style="color: green"><%= flash[:notice] %></p>
13
14 <%= yield %>
15
16 </body>
17 </html>
```

Znajdowanie layout'ów

```
1  class BooksController < ApplicationController
2     def show
3        @book = Book.find(params[:id])
4     end
5  end
```

- Rails szuka katalogu `app/views/layouts` pliku o tej samej nazwie co kontroler `/app/views/layouts/books.html.erb`
- jeżeli nie znajdzie takiego pliku, używany jest `/app/views/layouts/application.html.erb`

Używanie określonych layout'ów I

- określanie layout'u na poziomie kontrolera

```
1  class BooksController < ApplicationController
2     layout "spec_layout"
3     #...
4  end
```

- wybieranie layout'u w czasie działania aplikacji

```
1  class ProductsController < ApplicationController
2     layout :products_layout
3
4     def show
5        @product = Product.find(params[:id])
6     end
7
8    private
9      def products_layout
10        @current_user.special? ? "special" : "products"
11     end
12 end
```

Używanie określonych layout'ów II

- warunkowe layout'y

```
1  class ProductsController < ApplicationController
2     layout "inventory", :only => :index
3     layout "product", :except => [:index, :rss]
4     #...
5  end
```

- określanie layout'u dla akcji

```
1  class BooksController < ApplicationController
2     def show
3        @book = Book.find(params[:id])
4        render :layout => "special"
5     end
6  end
```

- dziedziczenie layout'ów przez klasy kontrolerów

Strukturyzacja layoutów

- asset tags
- yield i content_for
- widoki częściowe

Asset tags

- Helpery asset tags dostarczają metod do generowania kodu HTML, który łączy szablon z elementami zewnętrznymi takimi jak obrazki, javascript, arkusze stylów i kanały.
  - auto_discovery_link_tag
  - javascript_include_tag
  - stylesheet_link_tag
  - image_tag
  - vidio_tag
  - audio_tag
- Możesz używać tych tagów w layoutach lub innych plikach widoków
- Helpery asset tags nie weryfikują tego czy dany element zewnętrzny istnieje w określonym miejscu

javascript_include_tag

```
1  <%= javascript_include_tag "main" %>
2  #app/assets/javascripts/main.js
3
4  <%= javascript_include_tag "main", "/photos/columns" %>
5  #app/assets/javascripts/main.js and app/assets//photos/columns.js
6
7  <%= javascript_include_tag "http://example.com/main.js" %>
8  #http://example.com/main.js
9
10 <%= javascript_include_tag :defaults %>
11 #Prototype and Scriptaculous libraries
12
13 <%= javascript_include_tag :all %>
14 #every javascript file in app/assets/javascripts
```

stylesheet_link_tag

```
1  <%= stylesheet_link_tag "main" %>
2  #app/assets/stylesheets/main.css
3
4  <%= stylesheet_link_tag "main", "/photos/columns" %>
5  #app/assets/stylesheets/main.css and app/assets/photos/columns.css
6
7  <%= stylesheet_link_tag "http://example.com/main.css" %>
8  #http://example.com/main.css
9
10 <%= stylesheet_link_tag :all %>
11 #every css file in app/assets/stylesheets
```

image_tag

```
1  <%= image_tag "header" %>
2  #app/assets/images/header.png
3
4  <%= image_tag "icons/delete.gif" %>
5
6  <%= image_tag "icons/delete.gif", {:height => 45} %>
7
8  <%= image_tag "home.gif", :onmouseover => "menu/home_highlight.gif"  %>
9  <%= image_tag "home.gif",
10     :alt => "Go Home", :id => "HomeImage", :class => 'nav_bar' %>
```

yield i content_for

- yield identyfikuje sekcję do której powinna być wstawiona zawartość pliku widoku

```
1  <html>
2    <head>
3    </head>
4    <body>
5          <%= yield %>
6    </body>
7  </html>
```

```
1  <html>
2    <head>
3         <%= yield :head %>
4    </head>
5    <body>
6          <%= yield %>
7    <\body>
8  </html>
```

yield i content_for II

- główna część szablonu będzie zawsze renderowana do `yield` bez argumentu
- aby wyrenderować zawartość w nazwanym `yield`, należy użyć metody `content_for`

```
1  <% content_for :head do %>
2    <title>A simple page</title>
3  <% end %>
4
5  <p>Hello, Rails!</p>
```

```
1  <html>
2    <head>
3         <title>A simple page</title>
4    </head>
5    <body>
6          <p>Hello, Rails!</p>
7    <\body>
8  </html>
```

Widoki częściowe

- Widoki częściowe - są kolejnym narzędziem służącym do dzielenia procesu renderowania na mniejsze kawałki
- renderowanie widoku częściowego jako części pliku widoku

```
1  <%= render :partial => "menu" %>
2  #render a file named _menu.html.erb
3
4  <%= render :partial => "shared/menu" %>
5  #render app/views/shared/_menu.html.erb
```

- użycie widoków częściowych w pliku widoku

```
1  <%= render :partial => "shared/ad_banner" %>
2
3  <h1>Products</h1>
4
5  <p>Here are a few of our fine products:</p>
6  ...
7
8  <%= render :partial => "shared/footer" %>
```