Rozwiązania szkieletowe w tworzeniu aplikacji www

Tomasz Łukaszuk

Katedra Oprogramowania
Wydział Informatyki
Politechnika Białostocka

Temat wykładu:

Kontrolery

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.

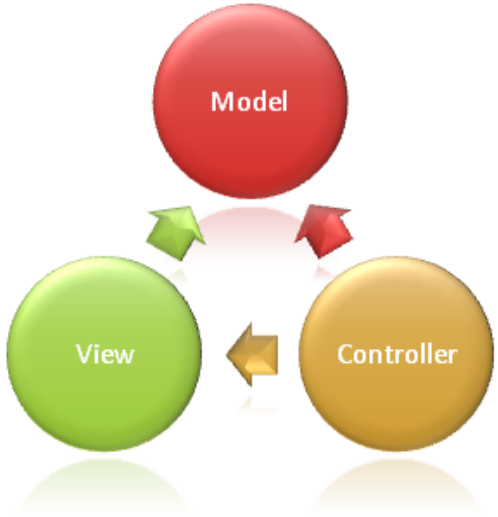
Kontrolery: Agenda

1. MVC - Kontroler
2. Kontrolery w RoR
   - Podstawowe wiadomości
   - Parametry
   - Tworzenie odpowiedzi HTTP
   - Filtry
   - Obiekty Request i Response
3. Django views
   - Podstawy
   - Parametry
   - Zwracanie błędów
   - Funkcje shortcut w django
   - Dekoratory do funkcji widoków
   - Obiekty request i response

MVC - Kontroler

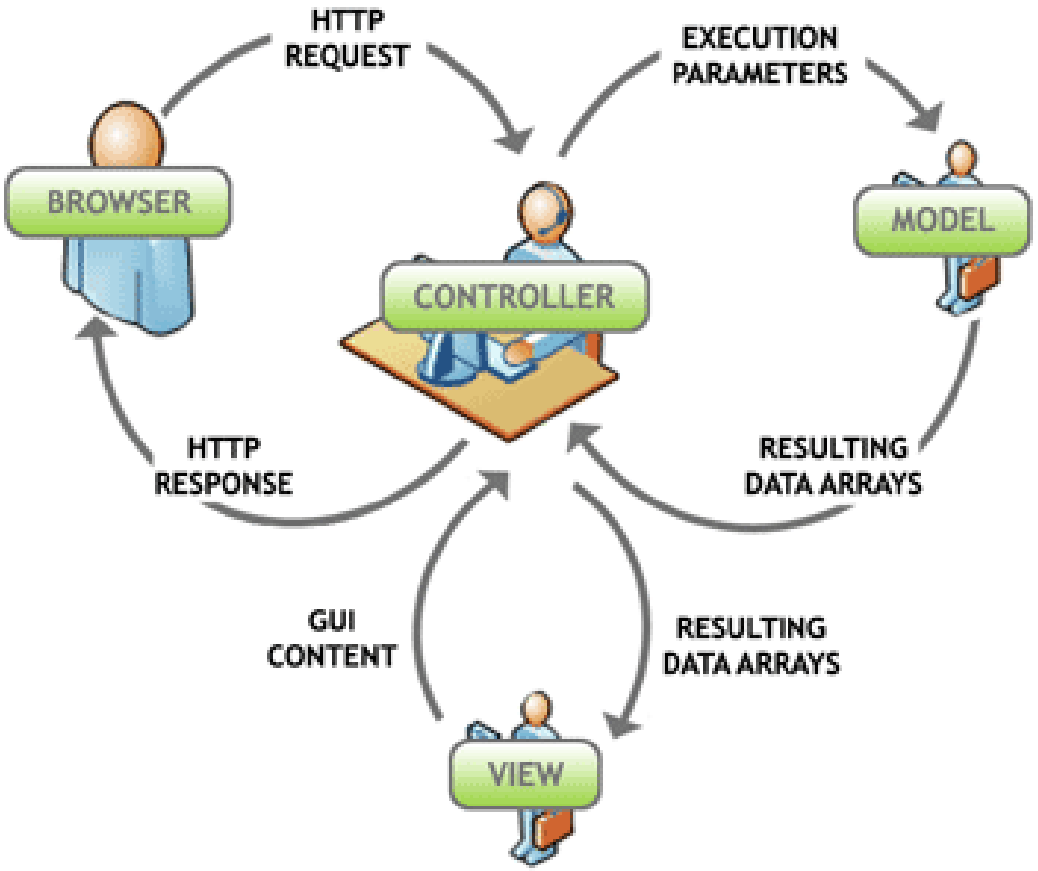
MVC: Kontroler - przypomnienie

- zadaniem kontrolera jest odbiór, przetworzenie oraz analiza danych wejściowych
- może zmienić stan modelu, odświeżyć widok, przełączyć sterowanie na inny kontroler



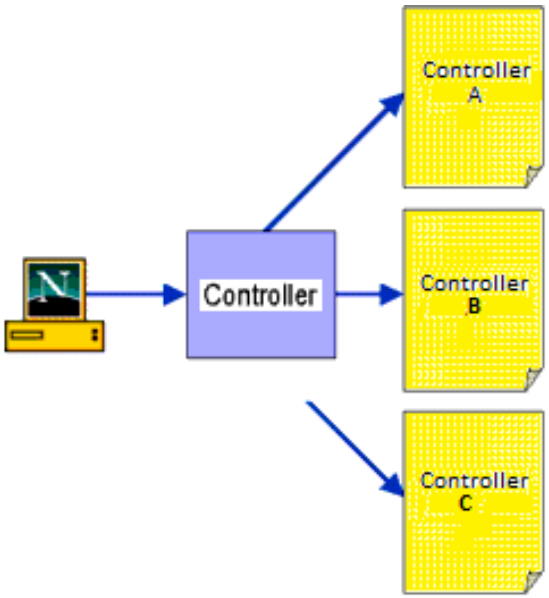
- może istnieć wiele kontrolerów, w danym momencie tylko jeden z nich steruje aplikacją
- kontroler posiada bezpośrednie wskazania na modele i widoki, z którymi współpracuje
- jest zasadniczą częścią każdej biblioteki implementującej MVC
- posiada akcje, tj. czynności wykonywane przez aplikację i określające jaki widok należy wyświetlić

Zadania kontrolera



1. odbiór żądania
2. decyzja o realizacji żądanych operacji określonych w parametrach żądania
3. rozdysponowanie zadań
4. określenie widoku do zwrócenia w odpowiedzi na żądanie
5. zwrot odpowiedzi

Realizacja kontrolera



- wzorzec projektowy Front Controller - jeden, centralny obiekt, który zarządza wszystkimi żądaniami przychodzącymi od klienta
- podkontrolery - kontrolery odpowiedzialne z reguły za jedną dziedzinę pracy aplikacji, np. zarządzanie użytkownikami, artykułami, zamówieniami
- akcje kontrolera - operacje odpowiedzialne za obsługę konkretnego żądania, implementowane jako metody w klasie kontrolera lub oddzielne klasy akcji (w zależności od konwencji frameworku)
- duży nacisk na konwencję, np. odpowiednia konstrukcja i nazwy akcji

Współpraca z modelem

Odczyt danych

- utworzenie instancji klasy modelu lub odwołanie się do klasy modelu w celu wywołania statycznej metody klasy modelu
- wywołanie odpowiedniej metody z instancji klasy modelu
- przypisanie wyniku do zmiennej

Modyfikacja danych

- odczyt parametrów przekazanych w żądaniu
- utworzenie instancji klasy modelu lub odwołanie się do klasy modelu w celu wywołania statycznej metody klasy modelu
- wywołanie odpowiedniej metody z instancji klasy modelu

Renderowanie widoku

- różne sposoby generowania odpowiedzi: renderowanie tekstu, renderowanie pliku
- różne formaty odpowiedzi
- przekazywanie danych do widoku: kontekst szablonu

Kontrolery w RoR

Podstawy kontrolerów w RoR

- moduł Action Controller
- kontroler jest klasą języka Ruby dziedziczącą z `ApplicationController`
- kontrolery są zapisywane w plikach `app/controllers/name_controller.rb`
- routing określa, który kontroler powinien zostać użyty do wygenerowania odpowiedzi na żądanie klienta
- kontroler jest odpowiedzialny za przetworzenie żądania i wygenerowanie odpowiedniej odpowiedzi (wyjścia)

Metody i Akcje

- kontroler jest klasą języka Ruby dziedziczącą z `ApplicationController`
- kontroler posiada metody tak jak każda inna klasa
- Kiedy aplikacja odbiera żądanie, routing określa który kontroler i akcja mają być uruchomione, Rails tworzy instancję kontrolera i uruchamia metodę o takiej samej nazwie jak nazwa akcji.

```
1  class ClientsController < ApplicationController
2    # akcje są metodami publicznymi
3    def new
4    end
5
6     # metody prywatne nie mogą być używane jako akcje
7     private
8
9     def foo
10    end
11 end
```

Parametry

- dwa rodzaje parametrów w aplikacjach web'owych
  parametry query string (wszystko po "?" w adresie URL)
  POST data (zwykle pochodzą z formularzy HTML)
- Rails nie rozróżnia parametrów query string i POST
- tablica asocjacyjna `params`

```
1  class ClientsController < ActionController::Base
2
3    def index
4      if params[:status] == "activated"
5        @clients = Client.activated
6      else
7        @clients = Client.unactivated
8      end
9    end
10
11 end
```

Parametry Hash i Array

- tablica asocjacyjna `params` nie jest ograniczona do jednowymiarowych par klucz i wartość
- przesyłanie tablicy (array) wartości

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
params[:ids] == ["1", "2", "3"]
```

- przesyłanie tablicy asocjacyjnej (hash)

```
1  <form action="/clients" method="post">
2    <input type="text" name="client[name]" value="Acme" />
3    <input type="text" name="client[phone]" value="12345" />
4    <input type="text" name="client[address][postcode]" value="12345"
5    <input type="text" name="client[address][city]" value="Carrot Cit
6  </form>
```

```
params[:client] == {"name" => "Acme", "phone" =>
"12345", "address" => {"postcode" => "12345",
"city" => "Carrot City"}}
```

Parametry z routing'u

- tablica asocjacyjna `params` zawsze zawiera klucze `:controller` i `:action`
- metody `controller_name` i `action_name`
- pozostałe parametry zdefiniowane przez routing, takie jak `:id` także są dostępne

```
routing
get '/clients/:status' => 'clients#index', foo: 'bar'

url
/clients/active

params
params[:status] = "active"
params[:foo] = "bar"
params[:controller] = "clients"
params[:action] = "index"
```

Sposoby generowania odpowiedzi HTTP

render	przesyła pełną odpowiedź do klienta
redirect_to	przesyła status przekierowania HTTP
head	przesyła odpowiedź zawierającą tylko nagłówek (head)

Hello world w RoR

```
1  class Controller < ApplicationController
2    def index
3      render :text => "Hello world!", :content_type = "text/plain"
4    end
5  end
```

Konwencja

- metoda `name` w kontrolerze `mycontr` renderuje widok `app/views/mycontrs/name.html.erb` (chyba że jawnie określono inaczej)
- nawet jeżeli w kontrolerze `mycontr` nie ma metody `name`, akcja `name` renderuje widok `app/views/mycontrs/name.html.erb` (w przypadku poprawie ustawionego routingu)

```
1  class ClientsController < ApplicationController
2    def new
3      # renderuje widok app/views/clients/new.html.erb
4    end
5
6     # akcja index renderuje widok app/views/clients/index.html.erb
7  end
```

Filters

- Filtry są metodami, które są uruchamiane przed (before), po (after) albo razem ("around") z akcją kontrolera.
- Filtry są dziedziczone, filtr z `ApplicationController` będzie uruchamiany dla każdego kontrolera w aplikacji
- Filtry typu before mogą wstrzymywać cykl przetwarzania żądania

```
1  class ApplicationController < ActionController::Base
2    before_filter :require_login
3    skip_before_filter :require_login, :only => [:new, :create]
4
5    private
6
7    def require_login
8      unless logged_in?
9        flash[:error] = "You must be logged in to access this section"
10       redirect_to new_login_url # halts request cycle
11     end
12   end
13
14 end
```

Obiekty `request` i `response`

- w każdym kontrolerze dostępne są dwa atrybuty: `request` i `response`
- są one związane z cyklem przetwarzania aktualnego żądania
- atrybut `request` odnosi się do instancji obiektu `AbstractRequest`
- atrybut `response` wskazuje na obiekt odpowiedz, która ma zostać zwrócona klientowi

Obiekt `request`

host - The hostname used for this request.
domain(n=2) - The hostname first n segments, starting from the right.
format - The content type requested by the client.
method - The HTTP method used for the request.
get?, post?, put?, delete?, head? - Returns true if the HTTP method is GET|POST|PUT|DELETE|HEAD.
headers - Returns a hash containing the headers associated with the request.
port - The port number (integer) used for the request.
protocol - Returns a string containing the protocol used plus "://", for example "http://".
query_string - The query string part of the URL, everything after "?".
remote_ip - The IP address of the client.
url - The entire URL used for the request.

Obiekt `response`

body - This is the string of data being sent back to the client. This is most often HTML.
status - The HTTP status code for the response, like 200 for a successful request or 404 for file not found.
location - The URL the client is being redirected to, if any.
content_type - The content type of the response.
charset - The character set being used for the response. Default is "utf-8".
headers - Headers used for the response.

Django views

Podstawy

- Widoki w Django odpowiadają części ”Kontroler” wzorca MVC
- Są one zapisywane w plikach `views.py`
- Ten (początkowo pusty) plik jest tworzony przez komendę `django-admin`
- Widoki pobierają obiekt `django.http.HttpRequest` i powinny zwracać obiekt `django.http.HttpResponse`
- Nazwa funkcji widoku nie ma znaczenia
- Która funkcja widoku będzie wywołana po otrzymaniu żądania zależy od zawartości pliku `urls.py`
- Pierwszym parametrem funkcji widoku zawsze jest obiekt `HttpRequest`
- Funkcja maże mieć więcej parametrów, w zależności od zawartości pliku `urls.py`

Najprostsza funkcja widoku w django

```
1  from django.http import HttpResponse
2
3  def function(request):
4      return HttpResponse('Hello world!')
```

- Funkcja widoku jest odpowiedzialna za generowanie treści wysłanych do użytkownika
  - (ignorując middleware, który może zmienić zawartość strony)
- Każda funkcja widoku dostaje obiekt `HttpRequest` jako pierwszy parametr, typowo nazywany jest on `request`
- Funkcja musi zwracać obiekt `HttpResponse`
- Pakiet `django.http` zawiera wiele klas, które ułatwiają generowanie odpowiedzi
- `HttpResponse` akceptuje ciąg znaków jako jeden z parametrów
- Ten ciąg znaków jest wysyłany jako odpowiedź do użytkownika
- Mimo, że nazwa funkcji nie ma znaczenia, to dobrą praktyką jest, aby nadawać znaczące nazwy ułatwiające utrzymywanie kodu

Parametry funkcji widoku — nienazwane grupy

```
1  urls.py:
2  re_path(r'^path/(\d+)/([a-z]{2,5})/$', function),
3
4  views.py:
5  from django.http import HttpResponse
6
7  def function(request, param1, param2):
8      string = 'Number: ' + param1
9      string = string + ' String: ' + param2
10     return HttpResponse(string)
```

- Kolejność parametrów jest taka sama jak kolejność grup wyrażeń regularnych w `urls.py`
- Grupy muszą być zamknięte w nawiasach
- Nazwa parametrów funkcji nie ma znaczenia
- Tylko kolejność jest brana pod uwagę
- Typem wszystkich parametrów jest string, niezależnie od wyrażenia regularnego użytego do ich przechwytywania

Parametry funkcji widoku — nazwane grupy

```
1  urls.py:
2  re_path(r'^path/(?P<number>\d+)/(?P<string>[a-z]{2,5})/$', function),
3
4  views.py:
5  from django.http import HttpResponse
6
7  def function(request, string, number):
8      string = 'Number: ' + number
9      string = string + ' String: ' + string
10     return HttpResponse(string)
```

- Nazwy zamiast kolejności są używane do odróżnienia parametrów
- Nazwy parametrów funkcji muszą być takie same jak nazwy występujące w wyrażeniach regularnych w `urls.py`
- Kolejność parametrów nie ma znaczenia

Parametry funkcji widoku — nazwane grupy

```
1  urls.py:
2  re_path(r'^path/(?P<number>\d+)/(?P<string>[a-z]{2,5})/$', function),
3
4  views.py:
5  from django.http import HttpResponse
6
7  def function(request, string, number):
8      string = 'Number: ' + number
9      string = string + ' String: ' + string
10     return HttpResponse(string)
```

- Jak poprzednio, typem wszystkich tych parametrów jest string, niezależnie od wyrażenia regularnego użytego do ich przechwytywania
- Podobna składnia (znak zapytania i dodatkowe parametry) jest używana w wielu silnikach wyrażeń regularnych do pobierania parametrów, które mogą zmieniać się podczas procesu dopasowania

Dodatkowe parametry

- Trzecim (opcjonalnym) fragmentem krotki z `urls.py` może być słownik
- Będzie on przekazany do funkcji widoku jako nazwane parametry
- Umożliwia to przekazanie dodatkowych parametrów funkcji

```
1  urls.py:
2  re_path(r'^path/(\d+)/([a-z]{2,5})/$', function,
3          {'name1': 'string', 'name2': 'string'}),
4
5  views.py:
6  from django.http import HttpResponse
7
8  def function(request, param1, param2, name1, name2):
9      string = 'Number: ' + param1
10     string = string + ' String: ' + param2
11     string = string + ' Name: ' + name2
12     return HttpResponse(string)
```

Zwracanie błędów

- podklasy `HttpResponse` dla wielu popularnych kodów statusów innych niż 200 (oznaczającego sukces)
- zwróć instancję jednej z tych podklas zamiast normalnego `HttpResponse` w celu zasygnalizowania błędu

```
1  def my_view(request):
2      # ...
3      if foo:
4          return HttpResponseNotFound('<h1>Page not found</h1>')
5      else:
6          return HttpResponse('<h1>Page was found</h1>')
```

Zwracanie błędów (cd)

- nie ma specjalnej podklasy dla każdego możliwego statusu odpowiedzi HTTP
- możliwe jest przekazanie kodu statusu HTTP do konstruktora `HttpResponse` aby utworzyć odpowiedź z dowolnym kodem statusu

```
1  def my_view(request):
2      # ...
3
4      # Return a "created" (201) response code.
5      return HttpResponse(status=201)
```

Wyjatek Http404

- błąd HTTP o kodzie 404 - file/page not found error
- klasa `django.http.Http404`
- Django posiada wyjątek `Http404`
- templatka `404.html` (umieszczona w najwyższym poziomie drzewa templatek)

```
1  from django.http import Http404
2
3  def detail(request, poll_id):
4      try:
5          p = Poll.objects.get(pk=poll_id)
6      except Poll.DoesNotExist:
7          raise Http404
8      return render(request, 'polls/detail.html', {'poll': p})
```

Funkcje shortcut w django

- pakiet `django.shortcuts`
- funkcje

  `render`

  `redirect`

  `get_object_or_404`

  `get_list_or_404`

Funkcja `render`

```
render(request, template_name, context=None, content_type=None,
               status=None, using=None)
```

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {"foo": "bar"},
        content_type="application/xhtml+xml")
```

równoważne z

```
from django.http import HttpResponse
from django.template import Context, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/template.html')
    c = RequestContext(request, {'foo': 'bar'})
    return HttpResponse(t.render(c),
        content_type="application/xhtml+xml")
```

Funkcja `redirect`

```
redirect(to, *args, permanent=False, **kwargs)
```

- zwraca `HttpResponseRedirect`
- argumenty mogą być
  - modelem: funkcja `get_absolute_url()` modelu zostanie wywołana
  - nazwą funkcji widoku: używa `urlresolvers.reverse()`
  - URLem

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object)

    #or
    return redirect('some-view-name', foo='bar')

    #or
    return redirect('/some/url/')
```

Funkcja `get_object_or_404`

- wywołuje `get()` na danym modelu, ale rzuca wyjątek `Http404` zamiast wyjątku model `DoesNotExist`

```
1  from django.shortcuts import get_object_or_404
2
3  def my_view(request):
4      my_object = get_object_or_404(MyModel, pk=1)
```

równoważne z

```
1  from django.http import Http404
2
3  def my_view(request):
4      try:
5          my_object = MyModel.objects.get(pk=1)
6      except MyModel.DoesNotExist:
7          raise Http404
```

Funkcja `get_list_or_404`

- zwraca wynik z metody `filter()` na danym modelu, rzuca `Http404` jeżeli wynikowa lista jest pusta

```
1  from django.shortcuts import get_list_or_404
2
3  def my_view(request):
4      my_objects = get_list_or_404(MyModel, published=True)
```

równoważne z

```
1  from django.http import Http404
2
3  def my_view(request):
4      my_objects = list(MyModel.objects.filter(published=True))
5      if not my_objects:
6          raise Http404
```

Dekoratory do funkcji widoków

- dekoratory mogą być stosowane do funkcji widoków do obsługi różnych funkcji HTTP
- dekorator `require_http_methods`

```
1  from django.views.decorators.http import require_http_methods
2
3  @require_http_methods(["GET", "POST"])
4  def my_view(request):
5      # I can assume now that only GET or POST requests make it this
6      # ...
7      pass
```

- dekoratory `require_GET()`, `require_POST()`
- dekoratory `condition`, `etag`, `last_modified` (warunkowe przetwarzanie widoku)
- dekorator `gzip_page` (kompresja GZip)

Klasa `HttpRequest`

Atrybuty:

- path
- path_info
- method
- encoding
- GET, POST, REQUEST
- COOKIES
- FILES
- META
- user
- session
- raw_post_data
- urlconf

Metody:

- get_host()
- get_full_path()
- build_absolute_uri(location)
- is_secure()
- is_ajax()
- ...

Klasa `HttpResponse`

Atrybuty:

- content
- status_code

Metody:

- has_header(header)
- set_cookie(key, value)
- delete_cookie(key)
- ...