Open Source Frameworks for Rapid Application Development

Marek Krętowski
Krzysztof Bandurski, Tomasz Łukaszuk, Tomasz Rybak

Software Departament
Faculty of Computer Science
Bialystok University of Technology

m.kretowski@pb.edu.pl
k.bandurski@pb.edu.pl, t.lukaszuk@pb.edu.pl, t.rybak@pb.edu.pl

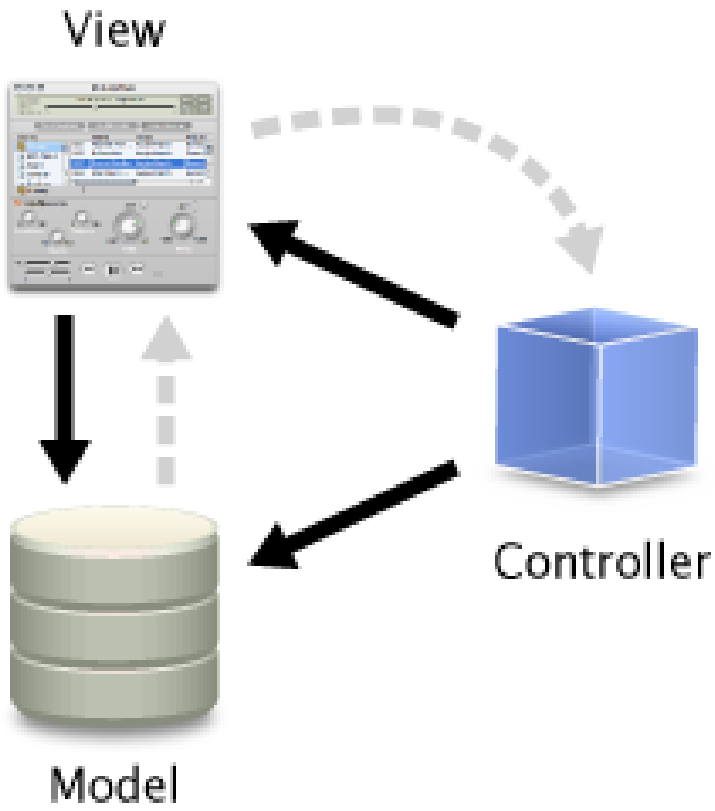
Lecture topic

Controllers

Controllers: Table of content

Introduction

Model-View-Controller pattern



MVC

- architectural pattern
- isolates ”domain logic” from the user interface
- proposed in 1979 by Trygve Reenskaug for Smalltalk

Model-View-Controller pattern

Model:

- is an object representing data or even activity
- manages the behavior and data of the application domain
- passive model, active model

View:

- is some form of visualization of the state of the model
- renders the model into a user interface element

Controller:

- offers facilities to change the state of the model
- receives user input and initiates a response by making calls on model objects
- accepts input from the user and instructs the model and viewport to perform actions based on that input

Ruby on Rails controllers

Controllers overview

- Action Controller module
- controller is a Ruby class which inherits from ApplicationController
- controllers are stored in `app/controllers/name_controller.rb` files
- routing has determined which controller should be use for a request
- controller is responsible for making sense of the request and producing the appropriate output

Methods and Actions

- controller is a Ruby class inherits from ApplicationController
- controller has methods just like any other class
- When your application receives a request, the routing will determine which controller and action to run, then Rails creates an instance of that controller and runs the method with the same name as the action.

```
1  class ClientsController < ApplicationController
2    # actions are public methods
3    def new
4    end
5
6     # private methods can not be used as actions
7     private
8
9     def foo
10    end
11 end
```

Controller for RESTful routing

```
1  class Controller < ApplicationController
2  def index
3      ...
4  end
5  def show
6      ...
7  end
8
9  def new
10     ...
11 end
12
13 def edit
14     ...
15 end
16 def update
17     ...
18 end
19
20 def destroy
21     ...
22 end
23 end
```

Parameters

- two kinds of parameters possible in a web application
  query string parameters (everything after "?" in the URL)
  POST data (usually comes from an HTML form)
- Rails does not make any distinction between query string and POST parameters
- `params` hash

```
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
```

Hash and Array Parameters

- The `params` hash is not limited to one-dimensional keys and values.
- sending an array of values

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

- sending a 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 City" />
6  </form>
```

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

Routing Parameters

- the `params` hash will always contain the `:controller` and `:action` keys
- methods `controller_name` and `action_name`
- other parameters defined by the routing, such as `:id` will also be available

```
1  map.connect "/clients/:status",
2    :controller => "clients",
3    :action => "index",
4    :foo => "bar"
```

Ways of creating HTTP responses

render	sends full response to the client
redirect_to	sends HTTP redirect status
head	sends header-only response

Hello world in RoR

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

Convention

- method `name` in the controller `mycontr` renders the view `app/views/mycontrs/name.html.erb` (unless explicitly stated otherwise)
- even if the controller `mycontr` there is no method `name`, action `name` renders the view `app/views/mycontrs/name.html.erb` (in case of proper routing)

```
1  class ClientsController < ApplicationController
2    def new
3      # renders the view app/views/mycontrs/new.html.erb
4    end
5
6     # action index renders the view app/views/mycontrs/index.html.erb
7  end
```

Filters

- Filters are methods that are run before, after or "around" a controller action.
- Filters are inherited, a filter on ApplicationController will be run on every controller in your application
- Before filters may halt the request cycle

```
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
```

The `request` and `response` Objects

- in every controller there are two accessor methods: `request` and `response`
- them are associated with the request cycle that is currently in execution
- the `request` method contains an instance of `AbstractRequest`
- the `response` method returns a response object representing what is going to be sent back to the client

The `request` Object

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.

The `response` Object

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

Overview

- Views in Django respond for the ”Controller” part of MVC pattern
- They are stored in `views.py` file
- This (initially empty) file is created by django-admin command
- View takes `django.http.HttpRequest` object and should return `django.http.HttpResponse`
- Name of the function does not matter
- Which function is called for which URL depends on `urls.py` file content
- First parameter of the view function is always `HttpRequest`
- Function can have more parameters, depending on content of `urls.py` file

The simplest view in Django

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

- View function is solely responsible for generating content send to the user
  - (ignoring middleware which can change this content)
- Each view function takes an `HttpRequest` object as its first parameter, which is typically named `request`
- Function must return `HttpResponse` object
- Namespace `django.http` contains many classes that can ease managing responses to the user
- `HttpResponse` accepts content string as one of its parameters
- This string is send as the response to the user
- Although name of the function does not matter, it is good practice to give meaningful names to ease maintaining of the code

View function parameters — unnamed groups

```
1  urls.py:
2  (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)
```

- Order of parameters is the same as order of regular expression groups in urls.py
- Group must be surrounded by parentheses
- Name of function parameters does not matter
- Only order is taken into consideration
- Types of all such parameters is string, regardless of regular expression used to capture them

View function parameters — named groups

```
1  urls.py:
2  (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)
```

- Names are used instead of order to distinguish parameters
- Names must be the same as those given to groups in regular expression in `urls.py`
- Order of those parameters does not matter

View function parameters — named groups

```
1  urls.py:
2  (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)
```

- As previously, types of all such parameters is string, regardless of regular expression used to capture them
- Similar syntax (question mark and additional parameters) is used in many regular expression engines to give parameters that change behaviour during matching process

Additional parameters

- Third (optional) fragment of tuple from `urls.py` may be dictionary
- It will be given to view function as named parameters
- This allows for giving additional parameters to the function

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

Returning errors

- subclasses of `HttpResponse` for a number of common HTTP status codes other than 200 (which means "OK")
- return an instance of one of those subclasses instead of a normal `HttpResponse` in order to signify an error

```
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>')
```

Returning errors (cd)

- there isn't a specialized subclass for every possible HTTP response code
- you can pass the HTTP status code into the constructor for `HttpResponse` to create a return class for any status code

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

The Http404 exception

- HTTP error code 404 - file/page not found error
- class `django.http.Http404`
- Django provides an `Http404` exception
- template `404.html` (located in the top level of your template tree)

```
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_to_response('polls/detail.html', {'poll': p})
```

Django shortcut functions

- package `django.shortcuts`
- functions

  `render`

  `render_to_response`

  `redirect`

  `get_object_or_404`

  `get_list_or_404`

`render` function

```
render(request, template[, dictionary][, context_instance]
       [, content_type][, status][, current_app])
```

```
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")
```

equivalent to

```
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")
```

render_to_response function

```
render_to_response(template[, dictionary][, context_instance][, mimetype])
```

```
from django.shortcuts import render_to_response

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

equivalent to

```
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 = Context({'foo': 'bar'})
    return HttpResponse(t.render(c),
        mimetype="application/xhtml+xml")
```

`redirect` function

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

- returns an `HttpResponseRedirect`
- the arguments could be
  - a model: the model's `get_absolute_url()` function will be called
  - a view name: uses `urlresolvers.reverse()`
  - a URL

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

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

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

`get_object_or_404` function

- calls `get()` on a given model manager, but it raises `Http404` instead of the model's `DoesNotExist` exception

```
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)
```

equivalent to

```
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
```

`get_list_or_404` function

- returns the result of `filter()` on a given model manager, raising `Http404` if the resulting list is empty

```
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)
```

equivalent to

```
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
```

View decorators

- decorators can be applied to views to support various HTTP features
- decorator `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 far
6      # ...
7      pass
```

- decorators `require_GET()`, `require_POST()`
- decorators `condition`, `etag`, `last_modified` (Conditional view processing)
- decorator `gzip_page` (GZip compression)

Class `HttpRequest`

Attributes:

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

Methods:

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

Class `HttpResponse`

Attributes:

- content
- status_code

Methods:

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