Workshop django

Laurent Peuch

July 13, 2012

Plan

- Installation
- Hello world
- MTV basique
- Avancé
- Jquery

Objectif

- vous apprendre la base de django et la structure classique d'un projet django
- vous donner une vision d'ensemble de ce qui est possible/disponible
- pour pouvoir continuer à apprendre la suite par vous même
- que ces slides puissent être un support pour vous
- plus ou moins vous donner ce qu'il faut pour bosser sur la partie django de p402

Requirements

- savoir coder
- connaitre la programmation orienté objet
- python (la base)
- sql (pas forcement poussée)
- html (la base)
- regexp (la base)
- savoir utiliser un shell (la base)

https://djangoproject.com

Introduction

C'est quoi django ?

- framework web en python

Introduction

C'est quoi django ?

- framework web en python
- full stacked (vs microframework comme flask, bottle et les 15 milles autres)

Introduction

C'est quoi django ?

- framework web en python
- full stacked (vs microframework comme flask, bottle et les 15 milles autres)
- vous "impose" un orm et un système de templates (changeable tous les 2 mais pas pensé pour)

Introduction

C'est quoi django ?

- framework web en python
- full stacked (vs microframework comme flask, bottle et les 15 milles autres)
- vous "impose" un orm et un système de templates (changeable tous les 2 mais pas pensé pour)
- mais en échange vous avez les django app

Introduction

C'est quoi django ?

- framework web en python
- full stacked (vs microframework comme flask, bottle et les 15 milles autres)
- vous "impose" un orm et un système de templates (changeable tous les 2 mais pas pensé pour)
- mais en échange vous avez les django app
- facile à utiliser (quasi tout), gros gain de productivité mais beaucoup à apprendre, vous allez souvent utiliser la documentation

Installation

Installation

2 façons de faire

- la classique avec le pkg manager de la distribution (eg: sudo apt-get install django)
- en utilisant pypi via pip (on va utiliser celle là) dans un virtualenv

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)
- aller dans un nouveau répertoire (mkdir blog && cd blog)

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)
- aller dans un nouveau répertoire (mkdir blog && cd blog)
- virtualenv - -no-site-packages - -distribute ve

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)
- aller dans un nouveau répertoire (mkdir blog && cd blog)
- virtualenv - -no-site-packages - -distribute ve
- source ve/bin/activate # pour rentrer dans le venv

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)
- aller dans un nouveau répertoire (mkdir blog && cd blog)
- virtualenv - -no-site-packages - -distribute ve
- source ve/bin/activate # pour rentrer dans le venv
- pip install django

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)
- aller dans un nouveau répertoire (mkdir blog && cd blog)
- virtualenv - -no-site-packages - -distribute ve
- source ve/bin/activate # pour rentrer dans le venv
- pip install django
- rehash # pour zsh

Installation

Faire:

- Installation de pip (sudo apt-get install python-pip)
- sudo pip install virtualenv (ou alors avec le package python-virtualenv)
- aller dans un nouveau répertoire (mkdir blog && cd blog)
- virtualenv - -no-site-packages - -distribute ve
- source ve/bin/activate # pour rentrer dans le venv
- pip install django
- rehash # pour zsh

Si vous voulez sortir du venv:

- deactivate

Mon premier projet django

Faire dans le virtualenv:

```
django-admin.py startproject <nom du projet>
```

Mon premier projet django

Faire dans le virtualenv:

```
django-admin.py startproject <nom du projet>
```

Lancer le serveur de développement:

```
cd <nom du projet> && python manage.py runserver
```

Puis aller sur http://0.0.0.0:8000

Django 1.4

Vous allez obtenir une hiérarchie du style:

```
project/urls.py
project/settings.py
project/__init__.py
project/wsgi.py
manage.py
```

Django $\leq$ 1.3

Remarque: vous allez aussi rencontrer des projets ayant la forme suivante (django $\leq$ 1.3)

```
urls.py
settings.py
manage.py
__init__.py
```

Hello World

urls.py

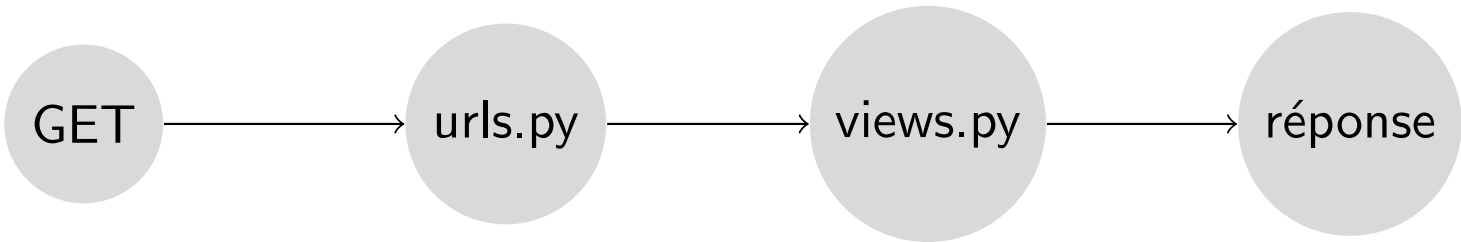
```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'project.views.home', name='home'),
    # url(r'^project/', include('project.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
)
```

Hello World

```
from django.conf.urls import patterns, include, url
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello World!")

urlpatterns = patterns('',
    url(r'^$', hello),
)
```

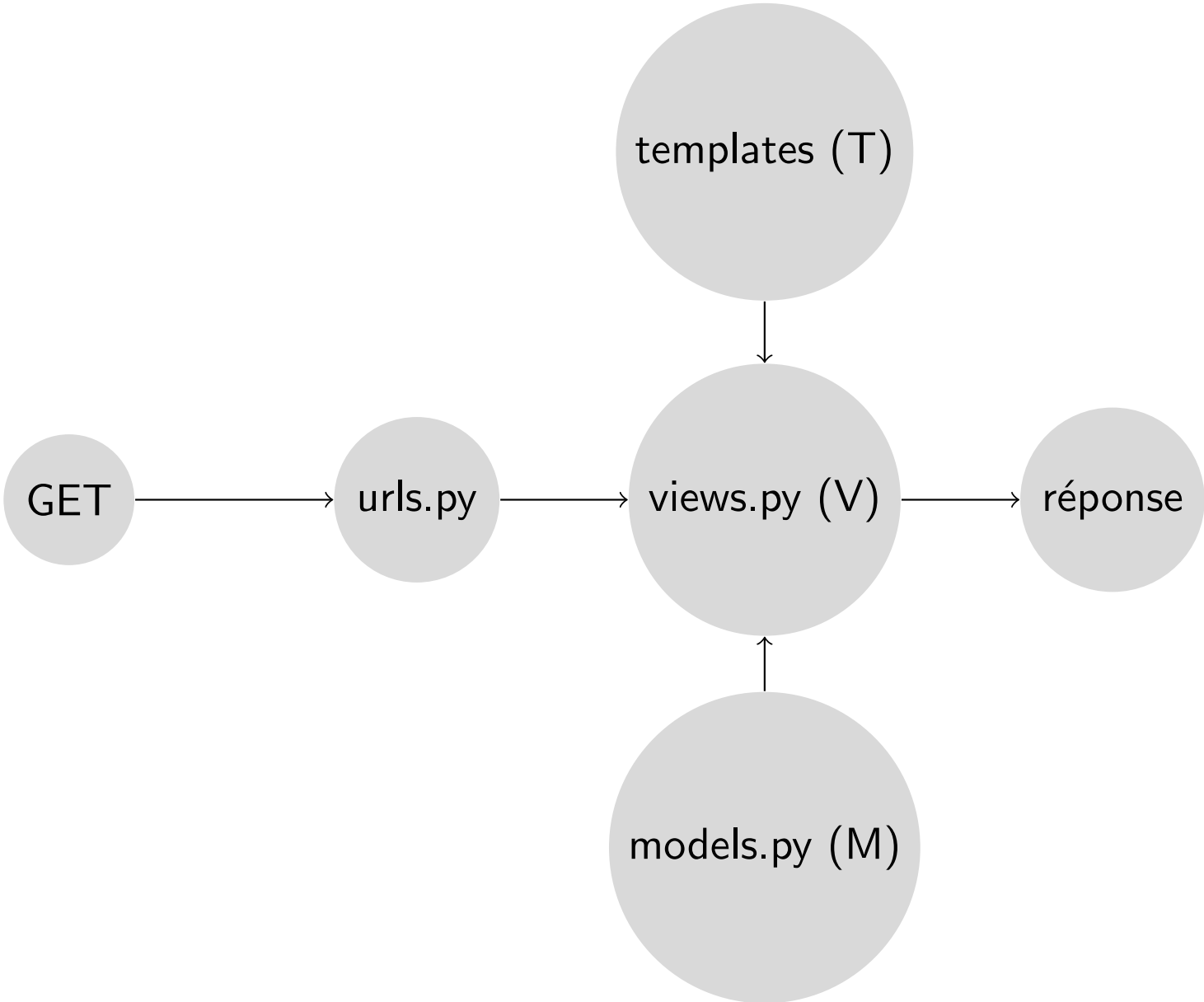
Remarque: traditionnelement ces fonctions sont dans un fichier *views.py*

La vie d'une requète



MTV

MTV



Plan pour la partie mtv

- templates
- models
- urls.py
- interface admin

Objectif: créer un mini blog (que sur la racine (/) s'affiche la liste de postes et sur $/ < post\_id > /$ s'affiche un poste).

Templates

Templates

Dans le répertoire de votre projet, au niveau où il y a le *manage.py* faire:

```
mkdir templates
```

Puis créer un fichier *templates/hello.html* contenant:

```
<p>Hello {{ name }}</p>
```

Utiliser un template

Utiliser un template version moyen âge:

```
from django.template import Context, loader

def hello(request):
    t = loader.get_template('hello.html')
    c = Context({
        'name': 'argument',
    })
    return HttpResponse(t.render(c))
```

On utilisera **jamais ça.**

Utiliser un template

Ce que tout le monde utilise:

```
from django.shortcuts import render

def hello(request):
    return render(request, "hello.html", {"name": "World"})
```

Utiliser un template

Si vous essayez d'aller sur *http* : //0.0.0.0 : 8000/ vous allez avoir une erreur *TemplateDoesNotExistat*/.

C'est parce que django demande qu'on lui dise où sont les templates (sauf pour les apps, cf plus tard).

Cela se fait dans le fichier *settings.py*.

Utiliser un template

Dans *settings.py* chercher un truc qui ressemble à ça:

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/d
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

Utiliser un template

Dans *settings.py* chercher un truc qui ressemble à ça:

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/d
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

Et mettre quelque chose du style:

```
TEMPLATE_DIRS = (
    '/home/urlab/code/slides/workshop-django/1.4/project/templates',
)
```

Utiliser un template

Dans *settings.py* chercher un truc qui ressemble à ça:

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/d
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

Et mettre quelque chose du style:

```
TEMPLATE_DIRS = (
    '/home/urlab/code/slides/workshop-django/1.4/project/templates',
)
```

Remarque: ce n'est pas portable, en vrai on utilise des paths relatifs, ici par simplicité je ne l'ai pas montré.

Utiliser un template

Dans *settings.py* chercher un truc qui ressemble à ça:

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/d
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

Et mettre quelque chose du style:

```
TEMPLATE_DIRS = (
    '/home/urlab/code/slides/workshop-django/1.4/project/templates',
)
```

Remarque: ce n'est pas portable, en vrai on utilise des paths relatifs, ici par simplicité je ne l'ai pas montré.

La virgule à la fin du string est OBLIGATOIRE sinon python ne considère pas que c'est un tuple (une liste à taille fixe).

Utiliser un template

Retourner sur *http* : //0.0.0.0 : 8000/, cela devrait fonctionner désormait.
Remarque: vérifier que le serveur de dev se lance bien et qu'il ne montre pas d'erreurs.

Syntaxe des templates

In a nutshell:

```
{{ nom_de_variable }} <- pour accèder à des variables

{% ... %} <- pour le reste (les "templatetags")
```

Volontairement simple, le but c'est de ne pas y avoir de logique. C'est pensé pour qu'un designer qui ne connaisse pas la programmation puisse s'en servir.

Syntaxe des templates

Les appels aux attributs d'un objet pythons sont uniformisés, cela veut dire en gros que:

```
objet.attribut
objet.methode()
objet[clef]
```

Syntaxe des templates

Les appels aux attributs d'un objet pythons sont uniformisés, cela veut dire en gros que ... s'écrivent tous:

```
objet.attribut  -> {{ objet.attribut }}
objet.methode() -> {{ objet.methode }}
objet[clef]     -> {{ objet.clef }}
```

Syntaxe des templates

Les appels aux attributs d'un objet pythons sont uniformisés, cela veut dire en gros que ... s'écrivent tous:

```
objet.attribut  -> {{ objet.attribut }}
objet.methode() -> {{ objet.methode }}
objet[clef]     -> {{ objet.clef }}
```

Remarque: **on ne peut pas passer d'arguments à une méthode dans un template, il faut passer par des templatetags.**

Syntaxe des templates

Une liste des templatetags de base:

Syntaxe des templates

Une liste des templatetags de base:

```
{% for element in list %}
    ...
{% endfor %}
```

Syntaxe des templates

Une liste des templatetags de base:

```
{% for element in list %}
    ...
{% endfor %}

{% if condition %}
    ...
{% elif autre_condition %}
    ...
{% endif %}
```

Les conditions sont très similaire à celles de python.

ORM

Modèles

On veut pouvoir stocker des données, ici on utilise une base de données SQL via l'ORM de django.

(ORM == object-relationnal mapping, une abstraction de la base de données)

Mon premier modèle

Exemple:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField()
```

À mettre dans *nom_du_projet/models.py*.

Remarque: ici c'est pour apprendre, en vrai ce fichier sera dans une app.

Autre remarque: si ce n'est pas spécifié django créera une clef primaire pour vous qui s'appelera *id* (et elle a un alias *pk*).

Créer la base de données

Maintenant qu'on a décrit un modèle, on doit demander à django de le créer dans la base de donner. Pour cela on doit faire la commande:

```
python manage.py syncdb
```

Créer la base de données

Maintenant qu'on a décrit un modèle, on doit demander à django de le créer dans la base de donner. Pour cela on doit faire la commande:

```
python manage.py syncdb
```

Cela va produire une erreur car on a pas dit à django quelle base de donnée utiliser.

Cela se fait dans *settings.py*.

Configurer la base de donnée

Dans *settings.py* changer:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.', # Add 'postgresql_psycopg2', '
        'NAME': '',                      # Or path to database file if
        'USER': '',                      # Not used with sqlite3.
        'PASSWORD': '',                  # Not used with sqlite3.
        'HOST': '',                      # Set to empty string for loca
        'PORT': '',                      # Set to empty string for defa
    }
```

En:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db.sqlite',
    }
```

Et refaire un:

```
python manage.py syncdb
```

Configurer la base de donnée

Django va vous demander si vous voulez créer un *superuser* (un admin), comme vous voulez, vous pourrez le refaire après avec:

```
python manage.py createsuperuser
```

En fait ici il y a un piège, django va créer plein de tables dans la base de données, mais pas celle que l'on vient de déclarer. C'est parce que django ne créer les tables que des apps qui sont spécifiés dans *settings.py* et la notre n'y est pas.

Configurer la base de donnée

Dans *settings.py* chercher:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    ...
)
```

Et rajouter:

```
INSTALLED_APPS = (
    ...
    'nom_du_projet',
)
```

Et refaire un:

```
python manage.py syncdb
```

Migration

Remarque: **si vous modifiez vos modèles par après,** *syncdb* **ne modifiera pas la base de données !**

Migration

Remarque: **si vous modifiez vos modèles par après,** *syncdb* **ne modifiera pas la base de données !**

Le SQL est assez rigide, pour faire cela il faut passer par des migrations (avec *django south* par exemple). Pas détaillers ici.

Migration

Remarque: **si vous modifiez vos modèles par après,** *syncdb* **ne modifiera pas la base de données !**

Le SQL est assez rigide, pour faire cela il faut passer par des migrations (avec *django south* par exemple). Pas détaillers ici.

Ou alors supprimer la db et refaire un syncdb (c'est ce qu'on fera ici). (Non, on ne fait pas ça en production).

Comment jouer avec des modèles

Remarque: je vous conseil de faire ça dans le shell django pour vous entrainer/amuser.

Comment jouer avec des modèles

Remarque: je vous conseil de faire ça dans le shell django pour vous entrainer/amuser.

Remarque #2: je vous conseil vivement d'installer *ipython*, cela rend les choses bien plus facile (dans le virtualenv faire: "pip install ipython").

Comment jouer avec des modèles

Remarque: je vous conseil de faire ça dans le shell django pour vous entrainer/amuser.

Remarque #2: je vous conseil vivement d'installer *ipython*, cela rend les choses bien plus facile (dans le virtualenv faire: "pip install ipython").

Pour lancer le shell:

```
python manage.py shell
```

Créer un model

Cela se fait de 2 façons. L'intuitive:

```
from project.models import Post

post = Post(title="titre", content="foobar")
post.save()
```

Créer un model

Cela se fait de 2 façons. L'intuitive:

```
from project.models import Post


post = Post(title="titre", content="foobar")
post.save()

# ou
post = Post()
post.title = "titre"
post.content = "foobar"
post.save()
```

Créer un model

Cela se fait de 2 façons. L'intuitive:

```
from project.models import Post


post = Post(title="titre", content="foobar")
post.save()

# ou
post = Post()
post.title = "titre"
post.content = "foobar"
post.save()
```

Remarque: le .*save*() est **obligatoire** pour sauver l'objet dans la base de donnée. On l'oublie souvent, c'est pour ça qu'il y a une autre méthode qui n'en nécessite pas:

```
Post.objects.create(title="titre", content="foobar")
```

Opérations classiques

Remarque: une grande partie des opérations va se faire via l'attribut *objetcts*. C'est appelé un manager.

Accèder à des modèles:

```
Post.objects.all()
Post.objects.filter(title="pouet")
Post.objects.count()
```

Opérations classiques

Remarque: une grande partie des opérations va se faire via l'attribut *objetcts*. C'est appelé un manager.

Accèder à des modèles:

```
Post.objects.all()
Post.objects.filter(title="pouet")
Post.objects.count()
```

Cela peut se chainer (les queries sont *lazy*):

```
Post.objects.all().filter(title="pouet").count()
```

Opérations classique

Accèder à un seul objet:

```
Post.objects.get(id=1)
```

Opérations classique

Accèder à un seul objet:

```
Post.objects.get(id=1)
```

Attention: si .get ne retourne rien ou plusieurs objets une exception est lancé.

Opérations classique

Accèder à un seul objet:

```
Post.objects.get(id=1)
```

Attention: si .get ne retourne rien ou plusieurs objets une exception est lancé.

Modifier un modèle:

```
post = Post.objects.get(id=1)
post.title = "autre titre"
post.save() # <- !!!
```

Opérations classique

Supprimer:

```
Post.objects.all().delete()
post = Post.objects.get(id=1)
post.delete()
```

urls.py

urls.py

Comment choper un argument:

```
from django.conf.urls import patterns, include, url
from django.http import HttpResponse

...

def with_arg(request, argument):
    return HttpResponse("Hello %s" % argument)

urlpatterns = patterns('',
    ...
    url(r'^(.*)/$', with_arg),
)
```

Pour tester aller sur *http* : //0.0.0.0 : 8000/*world*/

Syntaxe des templates

Une liste des templatetags de base:

Syntaxe des templates

Une liste des templatetags de base:

```
{% for element in list %}
    ...
{% endfor %}
```

Syntaxe des templates

Une liste des templatetags de base:

```
{% for element in list %}
    ...
{% endfor %}

{% if condition %}
    ...
{% elif autre_condition %}
    ...
{% endif %}
```

Les conditions sont très similaire à celles de python.

Interface admin

Blabla

- Interface généré automatiquement pour les modèles
- Les modèles doivent y être enregistré
- Pas mal customisable
- J'entrerai pas dans les détails

L'activer

- Dans *project/urls.py* décommenter les lignes correspondantes.
- Dans *project/settings.py* dans la liste d'applications décommencer les lignes correspondantes.

L'activer

urls.py

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    ...
    url(r'^admin/', include(admin.site.urls)),
)
```

settings.py

```
INSTALLED_APPS = (
    ...
    'django.contrib.admin',
)
```

Rajouter un modèle

Faire un fichier *admin.py* et y mettre:

```
from models import Post
from django.contrib import admin

admin.site.register(Post)
```

Exercice

- Activer l'interface admin.
- Se logger et se balader un peu.
- Rajouter le model Post dans l'interface admin.

Django apps

blabla

- aide à structurer le code

blabla

- aide à structurer le code
- réutilisable $->$ les gens ont déjà résolu plein de problèmes pour nous

blabla

- aide à structurer le code
- réutilisable $->$ les gens ont déjà résolu plein de problèmes pour nous
- grosse killer feature de django

blabla

- aide à structurer le code
- réutilisable $->$ les gens ont déjà résolu plein de problèmes pour nous
- grosse killer feature de django
- il est important qu'une app se concentre sur une et une seule chose, pas hésiter à en faire plein

blabla

- aide à structurer le code
- réutilisable $->$ les gens ont déjà résolu plein de problèmes pour nous
- grosse killer feature de django
- il est important qu'une app se concentre sur une et une seule chose, pas hésiter à en faire plein
- On en trouve un peut partout, surtout sur github, une bonne partie est listé sur *djangopackages.com*

blabla

- aide à structurer le code
- réutilisable $->$ les gens ont déjà résolu plein de problèmes pour nous
- grosse killer feature de django
- il est important qu'une app se concentre sur une et une seule chose, pas hésiter à en faire plein
- On en trouve un peut partout, surtout sur github, une bonne partie est listé sur *djangopackages.com*
- Aller voir ce talk http://youtu.be/A-S0tqpPga4

blabla

- aide à structurer le code
- réutilisable $->$ les gens ont déjà résolu plein de problèmes pour nous
- grosse killer feature de django
- il est important qu'une app se concentre sur une et une seule chose, pas hésiter à en faire plein
- On en trouve un peut partout, surtout sur github, une bonne partie est listé sur *djangopackages.com*
- Aller voir ce talk http://youtu.be/A-S0tqpPga4

Remarque: l'interface admin est une app.

Ma première app

Faire:

```
python manage.py startapp blog
```

Ma première app

Faire:

```
python manage.py startapp blog
```

Cela va vous donner:

```
blog/views.py
blog/models.py
blog/__init__.py
blog/tests.py
```

Migrer sur l'app

Objectif: bouger tout ce qu'on a fait dans l'app. Cela va impliquer:

- de bouger *models.py*, *admin.py*, *views.py* et d'adapter le *urls.py*
- d'avoir à enregistrer l'app dans *settings.py*
- d'avoir à recréer la db (les modèles sont plus ou moins lié à une app via un namespace (modifiable))

urls.py

L'*urls.py* de l'app sera quasi le même que le premier, il faudra juste retirer ce qui est lié à l'interface admin.

L'ancien *urls.py* devra être un peu modifié.

Approche naïve

On peut faire quelque chose comme ça (dans l'*urls.py* du projet, pas de l'app):

```
from django.conf.urls import patterns, include, url

from blog.views import hello, with_arg

urlpatterns = patterns('',
    url(r'^$', hello),
    url(r'^(.*)/$', with_arg),
    ...
)
```

Approche naïve

Ou comme ça (dans l'*urls.py* du projet, pas de l'app):

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
    url(r'^$', 'blog.views.hello'),
    url(r'^(.*)/$', 'blog.views.with_arg'),
    ...
)
```

Approche naïve

Ou encore comme ça (mieux) (dans l'*urls.py* du projet, pas de l'app):

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('blog.views',
    url(r'^$', 'hello'),
    url(r'^(.*)/$', 'with_arg'),
    ...
)
```

Approche plus mieux

Mais le mieux c'est de prendre le dernier fichier, de virer ce qui est lié à l'admin, de le mettre dans *blog/urls.py* et de faire:

```
from django.conf.urls import patterns, include, url

from blog import urls

urlpatterns = patterns('',
    url(r'^', include(urls)),
    url(r'^admin/', include(admin.site.urls)),
)
```

Approche plus mieux

Mais le mieux c'est de prendre le dernier fichier, de virer ce qui est lié à l'admin, de le mettre dans *blog/urls.py* et de faire:

```
from django.conf.urls import patterns, include, url

from blog import urls

urlpatterns = patterns('',
    url(r'^', include(urls)),
    url(r'^admin/', include(admin.site.urls)),
)
```

Remarque: **attention** pas de "$" à la fin de la regex !

Approche plus mieux

Mais le mieux c'est de prendre le dernier fichier, de virer ce qui est lié à l'admin, de le mettre dans *blog/urls.py* et de faire:

```
from django.conf.urls import patterns, include, url

from blog import urls

urlpatterns = patterns('',
    url(r'^', include(urls)),
    url(r'^admin/', include(admin.site.urls)),
)
```

Remarque: **attention** pas de "$" à la fin de la regex !

```
url(r'^', include('blog.urls')), # marche aussi
```

Approche plus mieux

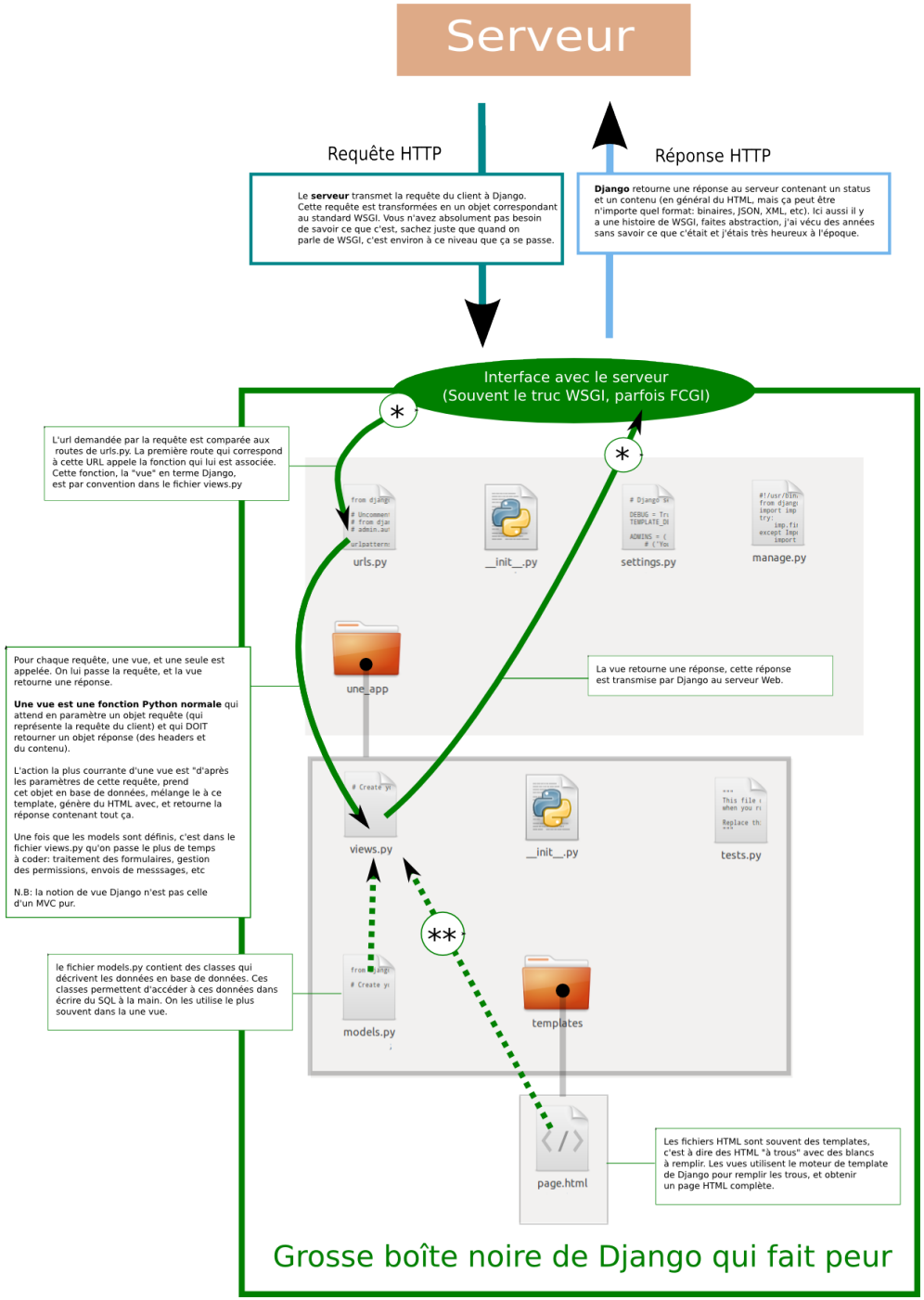
Pour référence voici à quoi peut ressembler le fichier *urls.py* de l'application (à mettre dans *application/urls.py* eg: *blog/urls.py*):

```
from django.conf.urls import patterns, include, url

from views import hello, with_arg

urlpatterns = patterns('',
    url(r'^$', hello),
    url(r'^(.*)/$', with_arg),
)
```

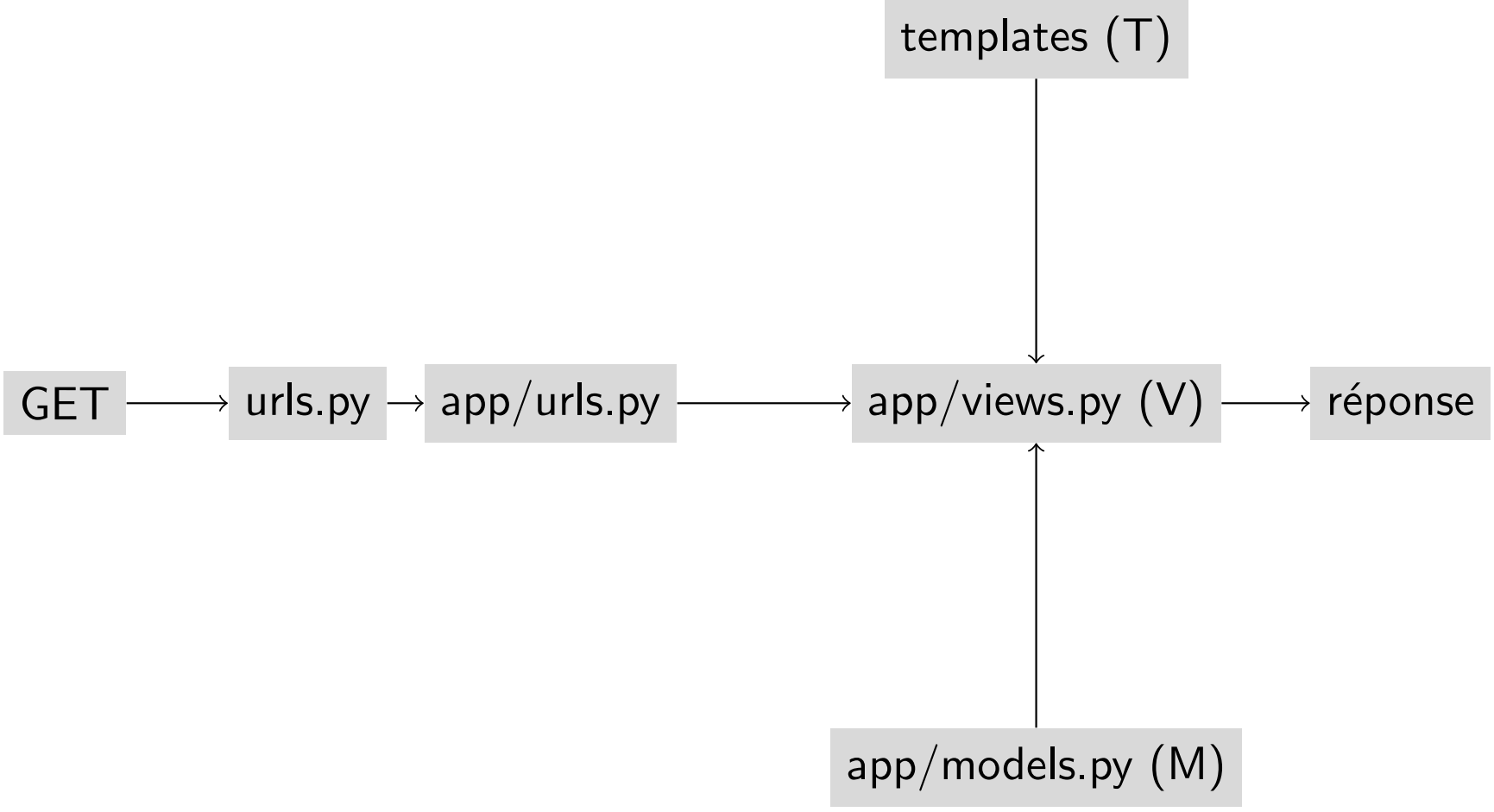
Remarque: seul le "blog." a disparu de l'import et les urls non liés à l'application ont disparues.

Remarques

- Vous pouvez créer un répertoire *templates* dans l'app et vous n'aurez pas à le déclarer dans le *settings.py*
- Pareil pour un répertoire *static* (pour les fichiers static)



Source: http://sametmax.com/schema-de-fonctionnement-general-de-django/


templates (T)
GET
urls.py
app/urls.py
app/views.py (V)
réponse
app/models.py (M)

Exercice

Migrer dans une django app.

Avancé

Arguments nomées dans les urls

Donner un nom aux arguments dans les urls:

```
url(r'^post/(?P<post_id>\d+)/$', 'show_post')
```

Utile pour les génériques views.

POST et GET dans les requests

Les arguments POST et GET sont mit dans des dictionnaires sur l'objet request:

```
def vue(request):
    request.POST["clef"]

def autre_vue(request):
    request.GET.get("clef")
```

Il est préférable d'utiliser .get pour éviter une exception

Nomer ses urls

Cela permet de faire des reverses pour avoir des urls relatives.

```
url(r'^(?P<post_id>\d+)/$', 'show_post', name="nom_url")
```

Dans les templates:

```
<a href="{% url nom_url une_variable.id %}">Lien vers un post</a>
```

Dans une vue:

```
from django.core.urlresolvers import reverse

def vue(request):
    ...
    reverse('post', args=[post.id])
    ...
```

Avec un préfixe

Pour éviter les conflits de nom entre apps:

```
(r'^blog/', include('blog.urls', namespace='blog', app_name='blog')
```

Ce qui donne:

```
{% url blog:post post.id %}
```

Et pareil pour reverse:

```
reverse("blog:post", args=[post.id])
```

Template, filtres

Il existe toute une série de filtres applicables à une variable comme ça (marche aussi sur les variables dans les templatetags):

```
{{ variable|filtre }}
```

Ils peuvent se chainer:

```
{{ variable|filtre1:"argument"|filtre2:"autre argument"
```

On y retrouve la plupart des opérations classiques sur les strings (cut, strip, slugify, lower, capitalize ...) et d'autres trucs plus spéciaux (slice par exemple).

Template, include

Le templatetag *include* permet d'inclure un template dans un autre de la façon suivante:

```
{% include "autre-template.html" %}
```

Remarque: préférez l'héritage pour les choses tels que les headers et les footers.

L'héritage de templates

Beaucoup plus intéressant que l'include:

base.html

```
<html>
    <head>
        ...
        {% block head %}{% endblock %}
    </head>
    <body>
        <div id="supermenu">
            ...
        </div>
        <div class="content">
        {% block content %}
        {% endblock %}
        </div>
    </body>
</html>
```

L'héritage de templates

example.html

```
{% extends "base.html" %}

{% block content %}
<p>Super content</p>
{% endblock %}
```

Champs des modèles classique

Il y a normalement tout ce qu'on peut trouver dans une bdd SQL classique et d'autres

- CharField
- TextField
- BooleanField
- DateField et DateTimeField
- EmailField
- URLField
- ForeignKeyField
- ManyToManyField(..., through="...")
- OneToOne

Et les options classique: null=True, blank=True, max_lenght=255

__unicode__ pour un modèle

C'est intéressant de surcharger cette méthode car c'est elle qui sera appelé par défaut dans un template (et ça rend la lecture dans un shell plus facile):

```
Class Post(models.Model):
    ...
    def __unicode__(self):
        return self.title

{{ post }} <- affiche le titre du poste
```

Les reverses des modèles

Si vous faites des relations entre modèles avec des foreigns key, django créer automatiquement des méthodes pour vous aider.

Les reverses des modèles

Si vous faites des relations entre modèles avec des foreigns key, django créer automatiquement des méthodes pour vous aider.

Par exemple si vous avez un modèle Post et un modèle Comment qui a une foreign key vert Post, vous pouvez faire:

```
post = Post.objects.get(id=1)
post.comment_set.all()
```

model_set est un manager comme *objects* et vous pouvez faire tout ce que vous pouvez faire avec objects dessus.

get_object_or_404

Un shortcut sympa:

```
from django.shortcuts import get_object_or_404

def vue(request):
    post = get_object_or_404(Post, id=1)
```

get_object_or_404

Un shortcut sympa:

```
from django.shortcuts import get_object_or_404

def vue(request):
    post = get_object_or_404(Post, id=1)
```

Remplace:

```
from django.http import Http404

def my_view(request):
    try:
        post = Post.objects.get(pk=1)
    except Post.DoesNotExist:
        raise Http404
```

Les génériques views

Très pratique mais un peu difficile à comprendre, y en a pour plusieurs types de choses:

- afficher le résultat d'un select
- afficher un objet
- créer u n nouvel objet
- updater un objet
- supprimer un objet
- afficher un template et afficher par rapport à un type de date

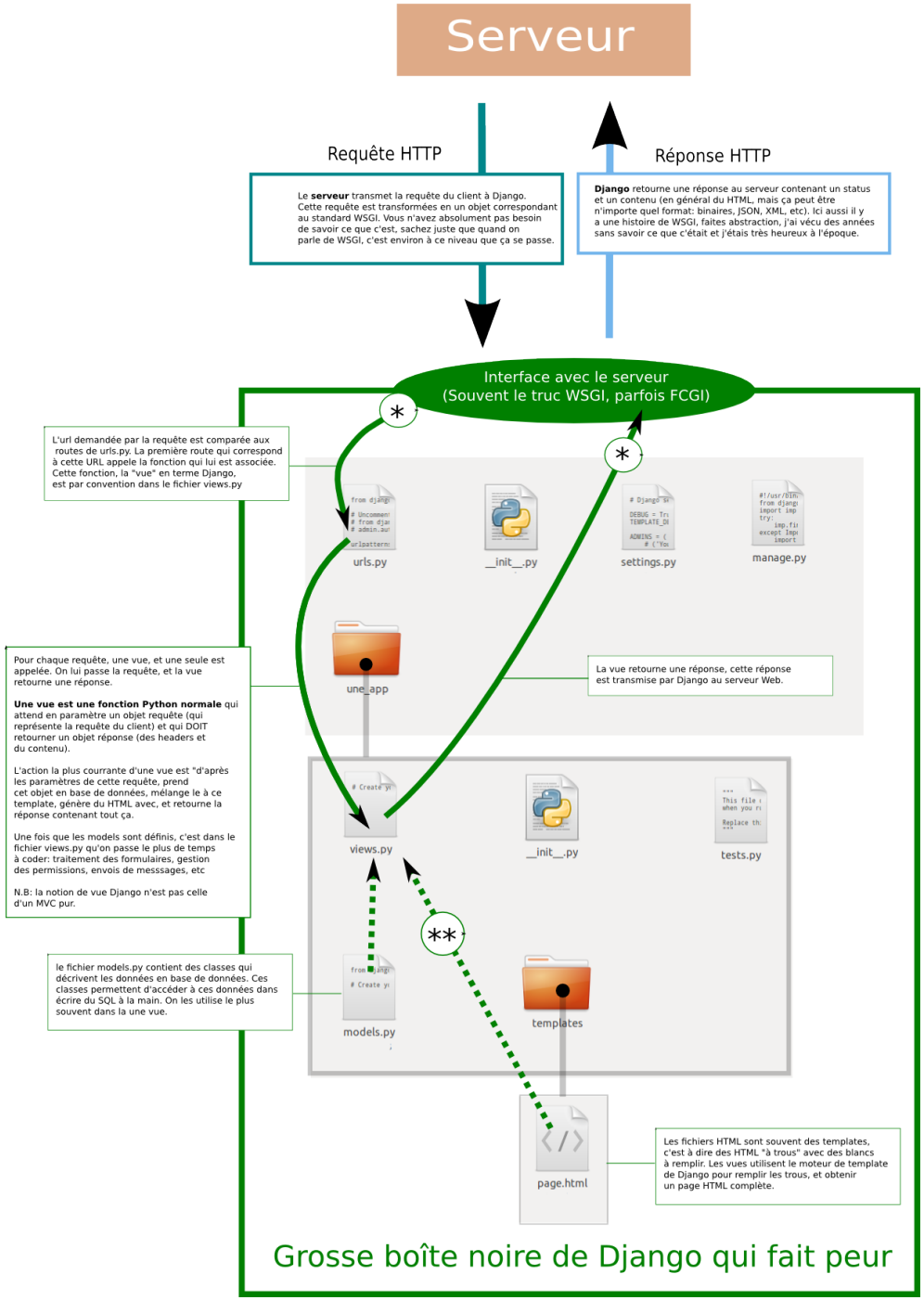
Les génériques views

Ex: pour du listing et le detail (nom des templates **important**):

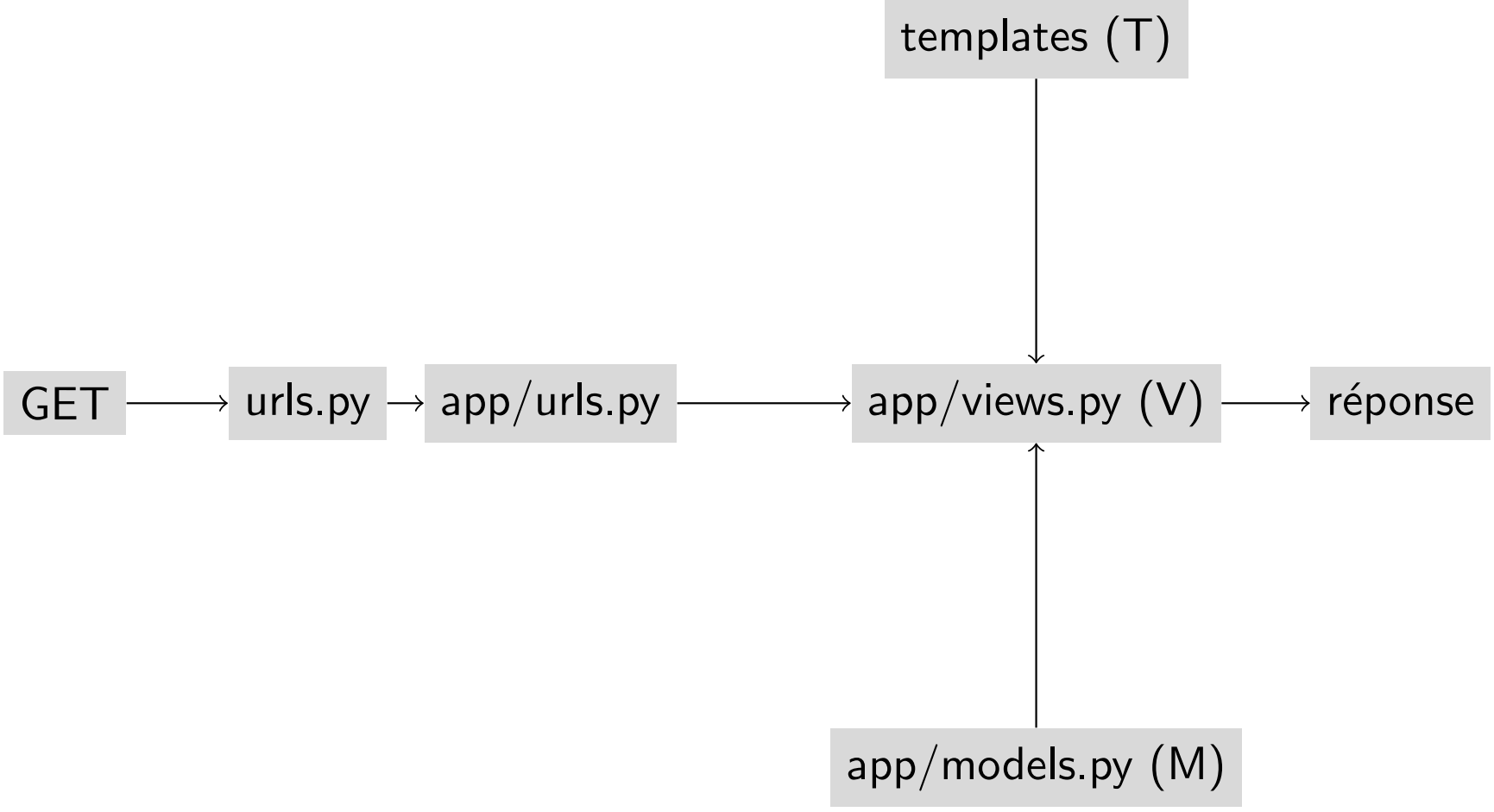
```
from django.views.generic import ListModel

urlpatterns = patterns('',
    url(r'^$', ListModel.as_view(model=Post)),
    url(r'^post/(?P<pk>\d+)/$', DetailModel.as_view(model=Post)),
    )

# est l'equivalent de

def list_posts(request):
    return render(request, "blog/post_list.html",
                  {"object_list": Post.objects.all()})

def post_detail(request, pk):
    return render(request, "blog/post_detail.html",
                  {"object": get_object_or_404(Post, pk=pk)})

urlpatterns = patterns('',
    url(r'^$', list_posts),
    url(r'^post/(?P<pk>\d+)/$', post_detail),
    )
```

Ce dont j'ai pas parlé

- les fichiers statiques
- forms
- middleware
- templatestags customs et filters custom
- rss/atom
- commandes perso
- gestion des utilisateurs (login_required)
- messages
- i18n
- fixtures
- trucs avancés dans les modèles (et les contexts managers (with))
- django-nonrel
- les trucs à savoir quand on passe en prod
- django-south
- le caching
- les django-apps cool
- et tout le reste



Source: http://sametmax.com/schema-de-fonctionnement-general-de-django/


templates (T)
GET
urls.py
app/urls.py
app/views.py (V)
réponse
app/models.py (M)

Jquery

Présentation

- javascript web pour les humains

Présentation

- javascript web pour les humains
- still javascript ...

Présentation

- javascript web pour les humains
- still javascript ...
- surtout intéressant pour rendre une page dynamique

Présentation

- javascript web pour les humains
- still javascript ...
- surtout intéressant pour rendre une page dynamique
- et faire de l'AJAX (en gros du POST/GET http sans recharger la page)

Présentation

- javascript web pour les humains
- still javascript ...
- surtout intéressant pour rendre une page dynamique
- et faire de l'AJAX (en gros du POST/GET http sans recharger la page)
- firebug (ou le brole buildin de chrome) sera votre compagnon d'infortune

Présentation

- javascript web pour les humains
- still javascript ...
- surtout intéressant pour rendre une page dynamique
- et faire de l'AJAX (en gros du POST/GET http sans recharger la page)
- firebug (ou le brole buildin de chrome) sera votre compagnon d'infortune
- utilisé par p402

Présentation

- javascript web pour les humains
- still javascript ...
- surtout intéressant pour rendre une page dynamique
- et faire de l'AJAX (en gros du POST/GET http sans recharger la page)
- firebug (ou le brole buildin de chrome) sera votre compagnon d'infortune
- utilisé par p402
- surtout vous montrer le principe

Présentation

- javascript web pour les humains
- still javascript ...
- surtout intéressant pour rendre une page dynamique
- et faire de l'AJAX (en gros du POST/GET http sans recharger la page)
- firebug (ou le brole buildin de chrome) sera votre compagnon d'infortune
- utilisé par p402
- surtout vous montrer le principe
- pas oublier que c'est executé par le naviguateur de l'utilisateur qui peut le modifier (ou en empêcher son execution)

Playground

Template avec lequel on va jouer:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Demo</title>
    <style>
        .test { font-weight: bold; }
    </style>
  </head>
  <body>
    <script src="jquery-1.7.2.min.js"></script>
    <script>
       $(document).ready(function(){
            // on met le code jquery ici
       });
     </script>
  </body>
</html>
```

Principe

Les uses cases de base sont assez simple:

- selectionner un élément du DOM (l'arbre html) et le modifier
- selectionner un élément du DOM et lui attacher un événement
- faire une requète (POST/GET) et potentielement utiliser le coder de retour

Selectionner

C'est relativement simple:

```
$("ici on met la query")
```

Ça ressemble à du xpath simplifié:

```
$("p") // tous les elements <p>
$("p.nom_de_class") // tous les elements <p class="nom_de_class">
$("p#pouet") // tous les elements <p id="pouet">
             // (normalement y en a qu'un seul sur du bon html)
$(".nom_de_class") // tous les elements ayant la class correspondante
$("#pouet") // tous les elements avec cet id
```

Exemple

Exemple idiot (à tester si ça vous amuse):

```
$("p").remove();
// va supprimer tous les <p> de votre page
```

Autre:

```
$("p").text("Atchoum");
```

Événements

Exemple, dans l'html:

```
<p><a href="#" id="hide">hide</a> -
   <a href="#" id="show">show</a></p>

<p id="pouet">Foobarbaz</p>
```

Dans la partie jquery:

```
$("#hide").click(function(event) {
    $("#pouet").hide("slow");
});
$("#show").click(function(event) {
    $("#pouet").show("slow");
});
```

Exemple réel: un menu déroulant.

AJAX: GET

GET (remarque, l'ajax n'est permit que sur le domaine où s'excecute le code):

```
$.get("/url/", function(data) {
    $("#pouet").html(data);
});
```

Pratique pour le chargement asynchrone du contenu d'une page.

AJAX: POST

POST similaire au GET:

```
$.post("/url/", function(data) {
    // ...
});
```

On peut lui donner des données (au GET aussi):

```
$.post("/url/", {name: "pouet", password: "toto"}, function(data) {
    // ...
});
```

Ou le contenu d'une form (au GET aussi):

```
$.post("/url/", $("#my_form").serialize(), function(data) {
    // ...
})
```

Remarques: le callback est optionnel et il existe un $.getJSON qui marche pareil.

Remarques

Vous aurez pas besoin d'énormement plus, avec ça vous savez déjà fait énormement.

Je vous recommande vraiment d'apprendre à vous servir de firebug, de sa console et de la fenêtre qui permet de mettre des breakpoints.

L'unique commande à retenir pour la console:

```
console.dir(truc);
```

Affiche les méthodes/attributs d'un objet.

Aussi intéressant de savoir que *console.log*("*foobar*"); affichera ça dans la console firebug.

Faim. Questions/Remarques ?