

Technologies avancées





1. Introduction

INTRODUCTION

- Installation et prérequis
- Le modèle MVC
- Architecture d'une application Web

PRISE EN MAIN

- 1er PROJET
- Commandes d'administration
- Création du projet
- Configuration
- Création d'une application
- Interface d'administration
- Création d'un modèle
- Ajout d'un modèle dans l'admin
- Création de vues
- Création d'un gabarit (template)

LES MODELES

- Les types de champs
- Relations inverses
- Syntaxe de requêtage Django

ADMINISTRATION

- Personnalisation de l'interface
- Gestion évoluée des modèles

LES TEMPLATES (GABARITS)

- Principe
- Choix du moteur
- Le langage DTL
- Les filtres
- Les balises (tags)
- Les commentaires
- Création de filtres + balises
- Héritage de gabarits

LES VUES

- Les "Class Based Views"
- Les vues génériques
- Les Context Processors
- Réécriture des pages d'erreur

LES FORMULAIRES

- Principes
- Validations
- Enregistrement
- Templating

DIVERS

- Internationalisation
- Flash Message
- Optimisations

DÉPLOIEMENT

- Intégration Nginx
- Checklist de déploiement



1. Introduction

1. Installation et prérequis

Création d'un environnement virtuel (Linux)

```
python3 -m venv my_venv3.8
```

→ (il va créer un dossier nommé my_venv3.8)

```
my_venv3.8/bin/activate ↔ source my_venv3.8/bin/activate
```

```
pip install --upgrade pip
```

```
pip install django
```

```
Installing collected packages: sqlparse, pytz, asgiref, django
Successfully installed asgiref-3.2.3 django-3.0 pytz-2019.3 sqlparse-0.3.0
(my_venv3.8) 16:59:23 olivier@olivier-mint ~ $
```



1. Introduction

1. Installation et prérequis

Création d'un environnement virtuel (Windows)

Presque la même chose excepté l'activation

```
python3 -m venv my_venv3.8
```

→ (il va créer un dossier nommé my_venv3.8)

```
my_venv3.8\Scripts\activate
```

```
pip install --upgrade pip
```

```
pip install django
```



1. Introduction

2. Le modèle MVC vs MVT

Architecture MVC : Modèle-Vue-Contrôleur.

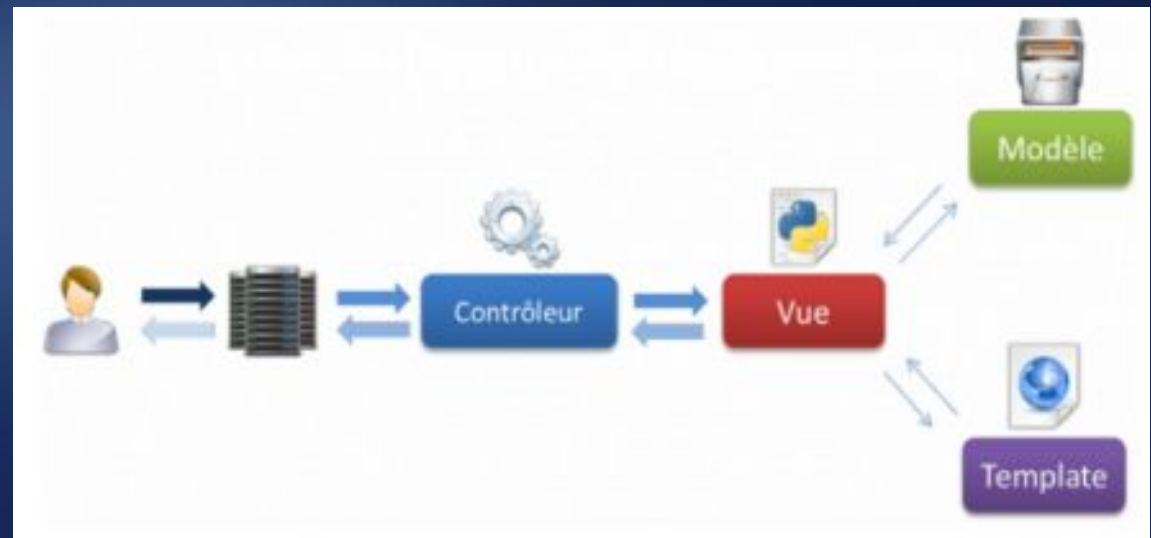
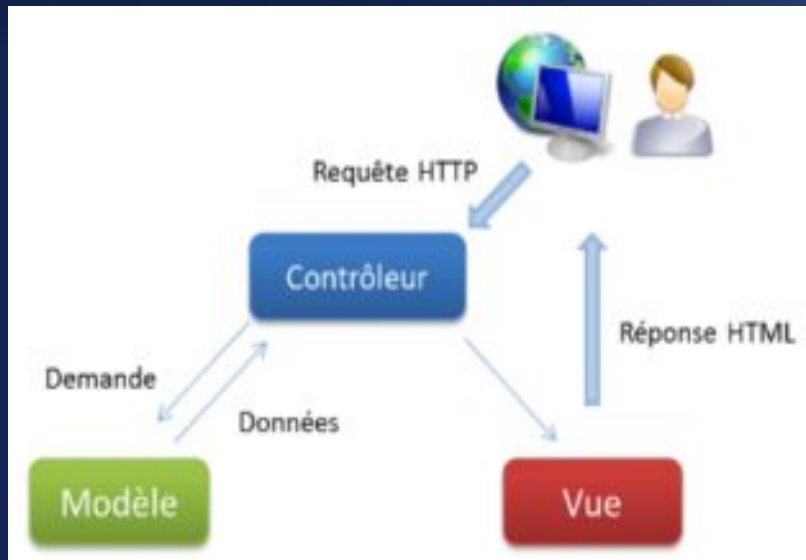
La communauté Django préfère parler de MVT, pour Modèle-Vue-Template : le contrôleur est presque totalement pris en charge par Django !

De ce fait, la vue prend en charge la récupération des données et est scindée en deux, avec un template afin de gérer l'affichage.

1. Introduction

2. Le modèle MVC vs MVT

MVC ↔ MVT : le contrôleur est géré par Django





1. Introduction

3. Architecture d'une application Web

Projet Django	→ Un projet
├─ App 1	→ Une application
├─ App 2	→ Une autre application
├─ ...	→ ...
└─ Apps	→ etc

Un projet est constitué de une ou plusieurs applications.
Attention ! Une application n'est pas forcément un serveur Web !
Cela peut être une gestion de client, un ensemble de modèles etc.



2. Prise en main

1. Premier projet

Par où commencer ?

Il est possible de commencer par n'importe quel centre d'intérêt !

- Création de tous les modèles ;
- Création de la présentation visuelle ;
- Mise en place du cadre de communication de base (REST ou autre etc.).

Ici nous suivrons ceci :

- Création des modèles
- Compréhension de l'interface d'administration
- Présentation visuelle



2. Prise en main

2. Commandes d'administration

`django-admin`

utilitaire en ligne de commande pour les tâches administratives.

`manage.py`

dans le dossier du projet, même chose mais définit en plus `DJANGO_SETTINGS_MODULE` = le fichier de réglages à utiliser

`django-admin`

utilitaire en ligne de commande pour les tâches administratives.

```
$ django-admin <command> [options]
```

```
$ manage.py <command> [options]
```

```
$ python -m django <command> [options]
```



2. Prise en main

2. Commandes d'administration

`django-admin help`

- Informations d'utilisation
- Liste des commandes de chaque application.

`django-admin help -commands`

Afficher une liste des commandes disponibles.

`django-admin help <command>`

Description de la commande concernée et ses options



2. Prise en main

2. Commandes d'administration

Pour les informations sur la création d'un projet :

```
django-admin help startproject  
python -m django help startproject
```

```
Creates a Django project directory structure for the given project name in the  
positional arguments:  
  name                Name of the application or project.  
  directory            Optional destination directory  
  
optional arguments:  
  -h, --help          show this help message and exit  
  --template TEMPLATE The path or URL to load the template from.  
  --extension EXTENSIONS, -e EXTENSIONS  
                      The file extension(s) to render (default: "py"). Separate  
  --name FILES, -n FILES  
                      The file name(s) to render. Separate multiple file name  
  --version           show program's version number and exit  
  -v {0,1,2,3}, --verbosity {0,1,2,3}
```

Etc.

→ capture
volontairement
incomplète
... car beaucoup
d'aide !



2. Prise en main

3. Création du projet

Le fichier `manage.py` est une sorte d'enveloppe de "django-admin"

L'interface d'administration n'existe pas car...
elle est générée automatiquement !

En attendant il faut demander à générer la base de données
dont se servent les modules de base...

Puis générer une application.

En général : un projet contient plusieurs "applications"



2. Prise en main

3. Création du projet

```
> django-admin startproject my_project  
> tree my_project
```

```
my_project/  
├── manage.py  
└── my_project  
    ├── __init__.py  
    ├── settings.py  
    ├── urls.py  
    ├── asgi.py  
    └── wsgi.py
```

- Dossier racine
- outil d'admin.
- module
- config. globale
- résolution des routes
- Asynchronous Server Gateway Interface
- Web Server Gateway Interface



2. Prise en main

4. Configuration

Éditez `my_project/settings.py`

Mettez le projet en français `LANGUAGE_CODE = 'fr'`
puis `TIME_ZONE` à `"Europe/Paris"`

Lancez ensuite :

```
python3 manage.py makemigrations  
python3 manage.py migrate  
python3 manage.py createsuperuser
```

Explications détaillées dans les slides suivants



2. Prise en main

4. Configuration

Lancez finalement :

```
python3 manage.py runserver
```

```
Django version 3.0, using settings 'my_project.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CONTROL-C.
```

Si tout fonctionne →





2. Prise en main

5. Création d'un application

Un projet Django contient des *applications*.

Une application décrit un paquet Python qui fournit un certain ensemble de fonctionnalités.

Les *applications* peuvent être réutilisées dans différents projets.

Les *applications* comprennent une combinaison de modèles, vues, gabarits (templates), fichiers statiques, URL, etc.

Elles sont généralement liées à des projets via le réglage `INSTALLED_APPS`

Plus d'information : <https://docs.djangoproject.com/en/dev/ref/applications/>



2. Prise en main

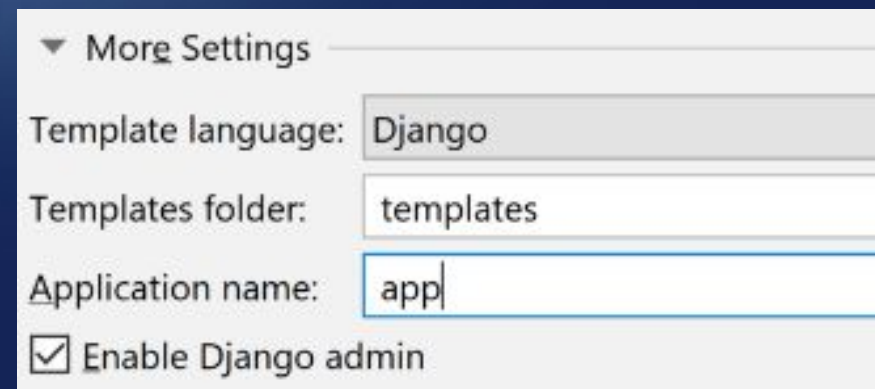
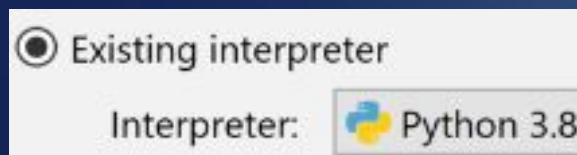
5. Création d'un application

Pour créer une application dans un projet existant
Il suffit d'appeler « `manage.py startapp nom_application` » .

PyCharm Pro le fait pour nous au démarrage d'un nouveau projet.

Dans les slides ci-après, on voit comment le faire manuellement.

Nouveau projet + application via PyCharm :





2. Prise en main

5. Création d'un application

Créer une application

- Créer le dossier concerné, ici, par exemple « app »
- Créer le fichier « `__init__.py` » afin que « app » soit un module
- Créer le fichier « `app/apps.py` » (nom défini par Django)
- Dans ce fichier, préciser les informations sur l'application :

```
from django import apps
```

```
class AppConfig(apps.AppConfig):  
    name = 'app'  
    verbose_name = "My own application"
```



2. Prise en main

5. Création d'un application

- Editer « `settings.py` » et y ajouter notre application nommée « `app` » dans « `INSTALLED_APPS` », *en notation pointée* package python soit :

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    # ... ,  
    'app.apps.AppConfig',  
]
```

Plus d'information : <https://docs.djangoproject.com/en/dev/ref/applications/>



2. Prise en main

6. Interface d'administration

- Créez un super utilisateur :
`python3 manage.py createsuperuser`
- Relancez le serveur
- Allez sur l'interface d'administration :
`http://127.0.0.1:8000/admin/`

A screenshot of the Django administration interface. At the top, there is a teal header bar with the text "Administration de Django" in white. Below the header, the page is white. There are two input fields: the first is labeled "Nom d'utilisateur :" and the second is labeled "Mot de passe :". Below these fields is a teal button with the text "Connexion" in white.



2. Prise en main

6. Interface d'administration

L'interface d'administration représente tous les modèles que vous voulez afficher en CRUD.

Pour l'instant, il n'y a que les modèles « livrés » avec Django c'est à dire « Group » et « User ».

Nous allons en créer d'autres.





2. Prise en main

7. Création d'un modèle

La conception des modèles est *primordiale* pour tout type d'application : représenter le(s) métier(s) concernés correctement permet de faire des applications *pérennes*.

Django passe par un ORM (Mapping objet-relationnel).

On doit pouvoir se passer *presque toujours d'écrire des requêtes SQL manuellement* : c'est Django qui construit et optimise les requêtes à la base de données.



2. Prise en main

7. Création d'un modèle

Un modèle est une classe qui descend de `models.Model`. Les champs sont des propriétés statiques initialisées au démarrage de l'application qui sont directement mappées dans la base de données :

Pour une application nommée « app » :

<code>class Recipe(models.Model):</code>	→	Table	→	<code>app_recipe</code>
<code>title = models.CharField(...)</code>	→	champ	→	<code>title</code>

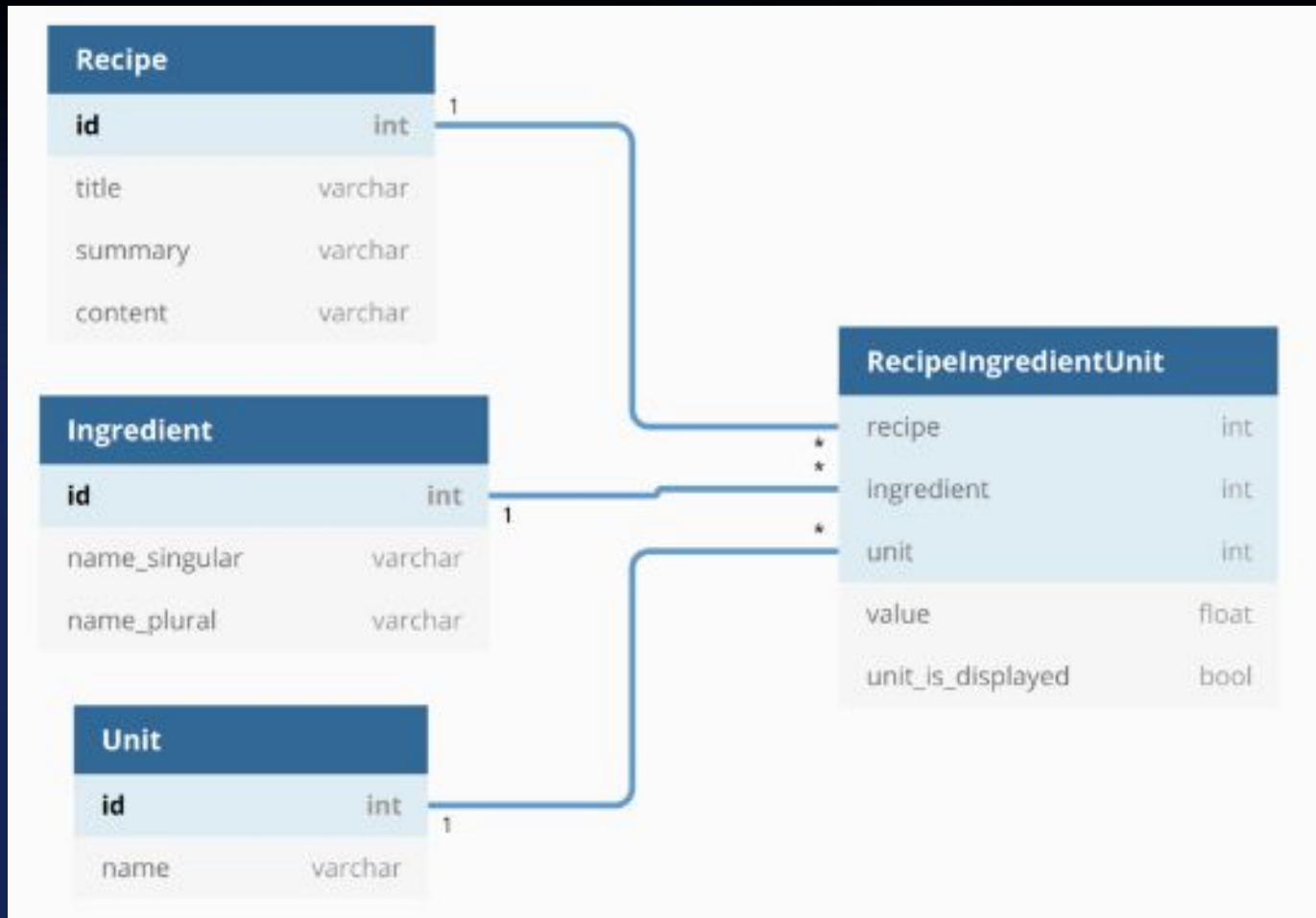


Code python



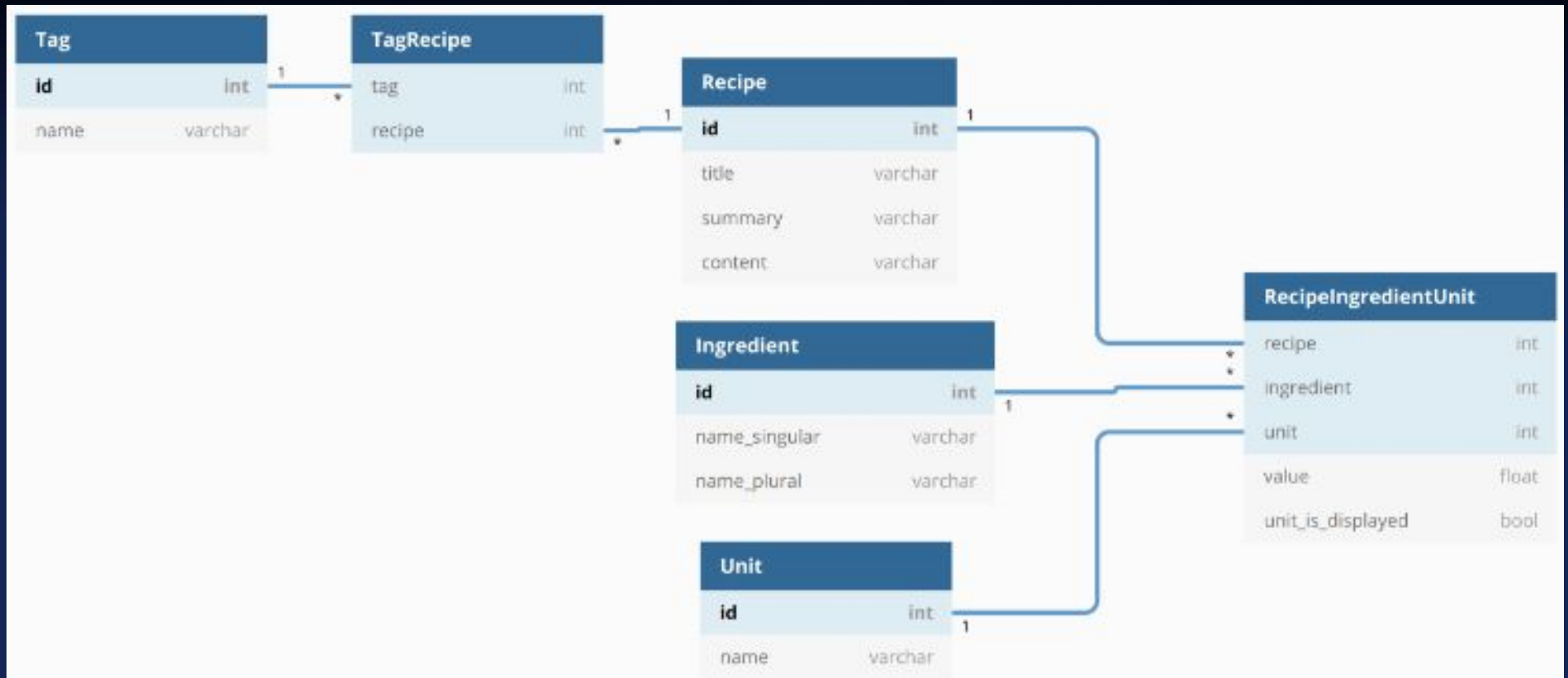
2. Prise en main

7. Création d'un modèle



2. Prise en main

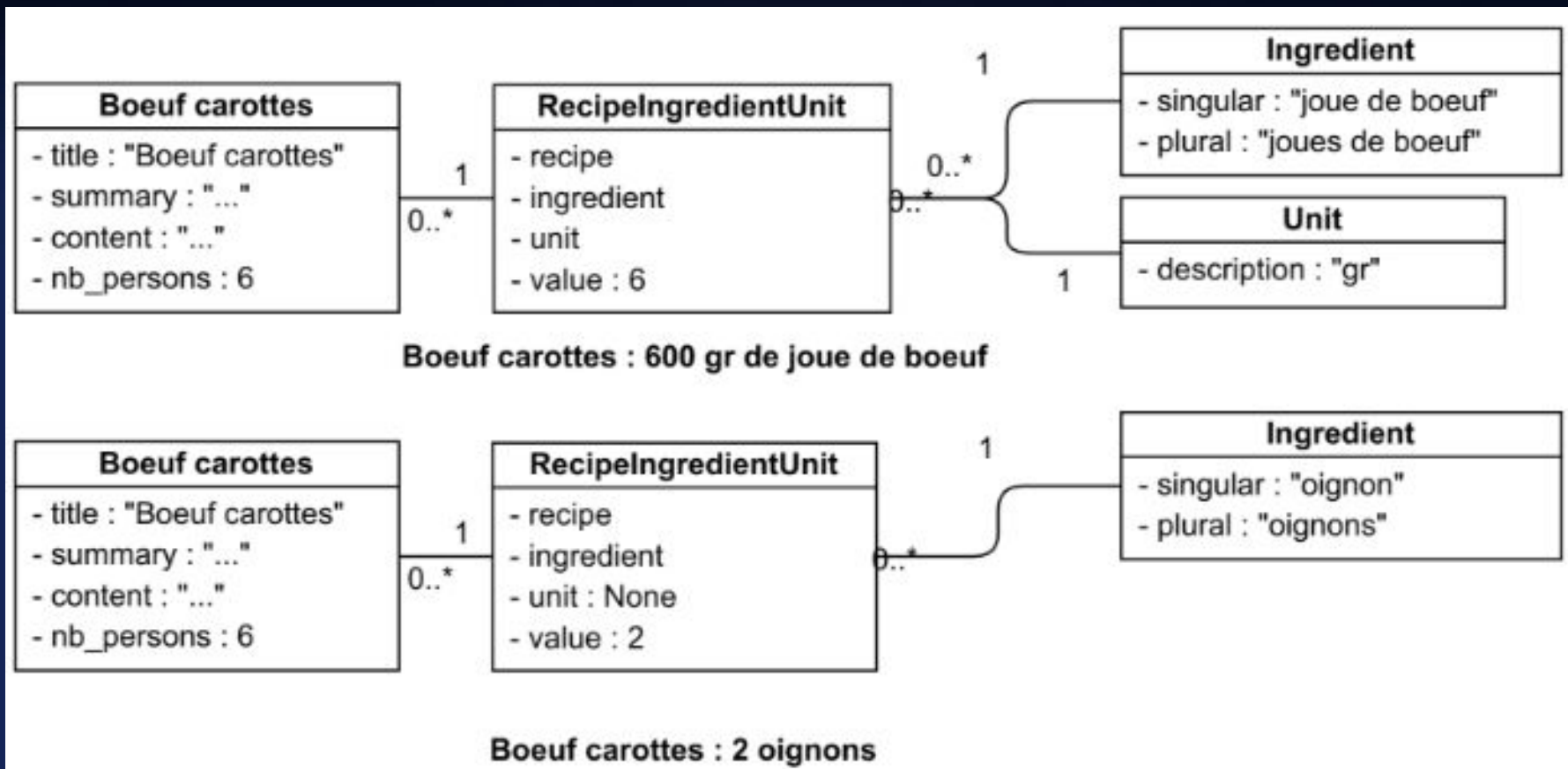
7. Création d'un modèle



2. Prise en main

7. Création d'un modèle

Exemple de deux ingrédients pour une recette :





2. Prise en main

7. Création d'un modèle

- Créer un fichier « `app/models.py` » : il contiendra tous les modèles.
NB : s'il devient trop grand il suffit de le transformer en package.
- Y ajouter notre premier modèle :

```
from django.db import models
```

```
class Recipe(models.Model):  
    title = models.CharField(max_length=200, blank=True,  
                             default=None, null=True)  
    summary = models.CharField(max_length=200, blank=True,  
                               default=None, null=True)  
    content = models.TextField(blank=True, default=None, null=True)  
    nb_persons = models.IntegerField(blank=True, default=2, null=True)
```



2. Prise en main

7. Création d'un modèle

Synchronisation modèles ↔ base de données : deux étapes :

- `makemigrations` pour noter les modifications dans un log
- `migrate` pour appliquer les log

→ Créer le dossier de log = package Python appelé « migrations ».
Il y en a un par application. Ici, donc :

```
app/  
├── migrations  
│   └── __init__.py  
├── ...  
└── models.py
```

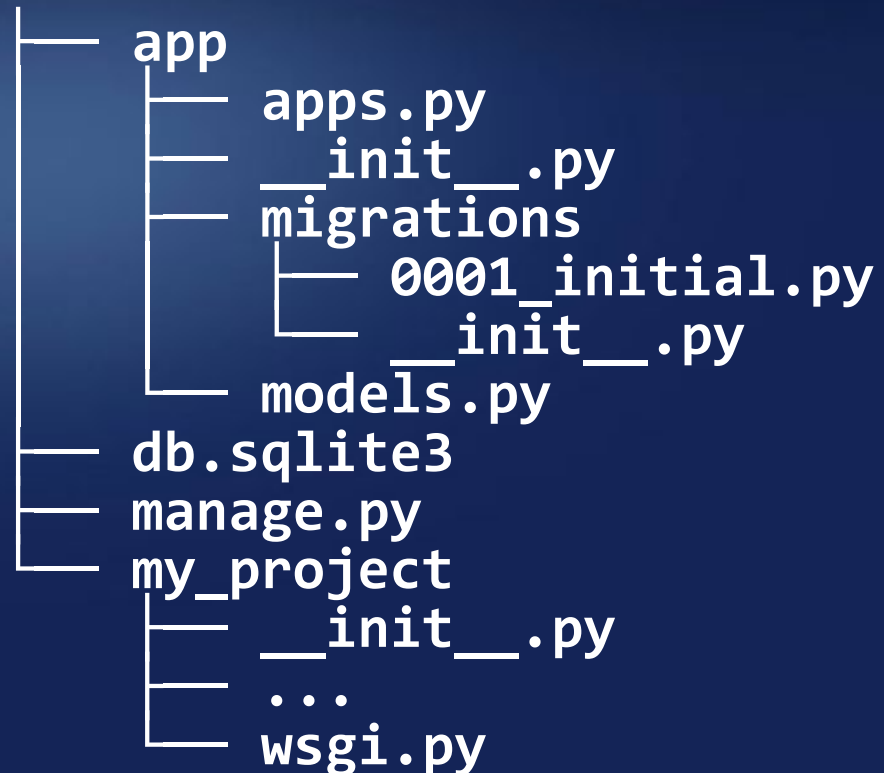


2. Prise en main

7. Création d'un modèle

NB : à chaque modification des modèles, ne pas oublier :
`python3 manage.py makemigrations`
`python3 manage.py migrate`

Ici, le contenu du projet doit ressembler à ceci :

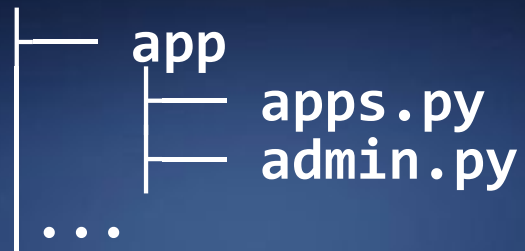




2. Prise en main

8. Ajout d'un modèle dans l'admin

Pour personnaliser l'administration, et y ajouter nos modèles, créer un fichier « `admin.py` » s'il n'est pas présent :



Puis déclarer le modèle avec ce code (à adapter selon le besoin) :

```
from django.contrib import admin
```

```
from app.models import Recipe
```

```
admin.site.register(Recipe)
```



2. Prise en main

8. Ajout d'un modèle dans l'admin

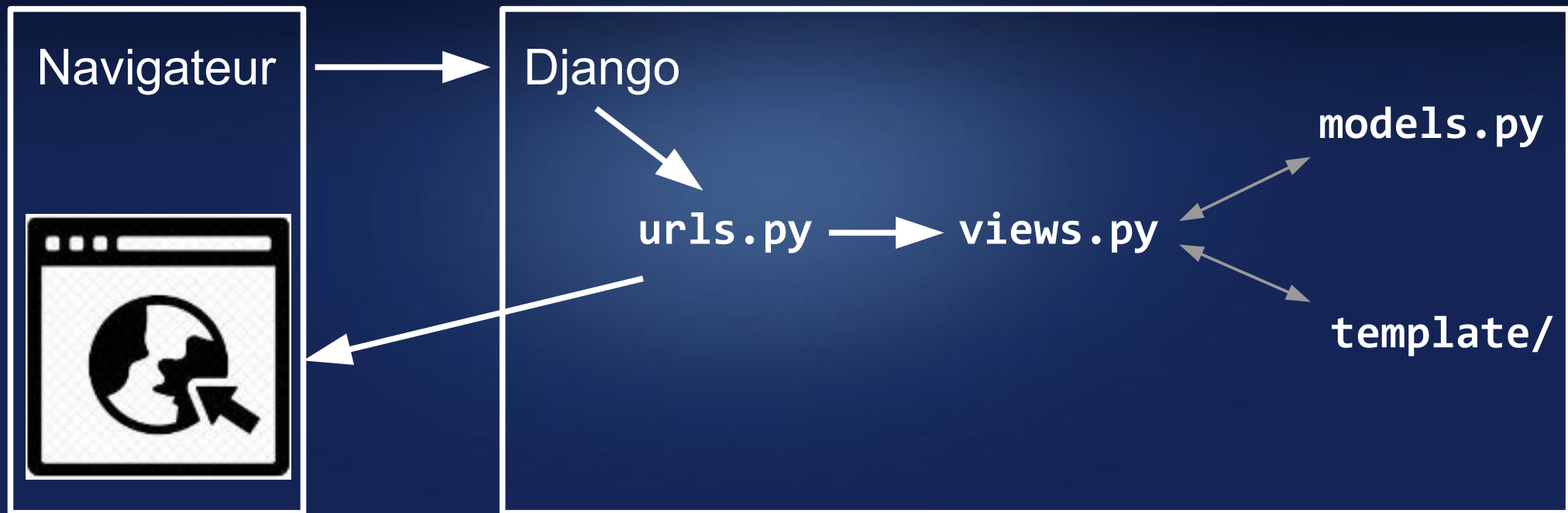
En allant sur l'interface d'administration, on doit voir le modèle et toute son interface CRUD sans ajouter de code !



2. Prise en main

9. Création de vues

Lorsqu'on affiche une page Web qui passe par Django, le trajet se fait – de manière simplifiée – ainsi :





2. Prise en main

9. Création de vues

Suite à l'explication, il faut donc :

- créer une vue dans `views.py`
- la déclarer dans `urls.py`

La vue dans `views.py` peut être très simple (2 lignes !) ou très complexe : dans ce cas, transformer `views.py` en package et mettre chaque vue dans fichier qui porte le nom de la vue.

(!) Tous les exemples « simples » utilisent des vues qui passent par des *fonctions*.

Ici nous ne ferons que des vues basées sur des classes, appelées ***class-based views*** :

class-based views = code plus lisible, évolutif et donc maintenable.



2. Prise en main

9. Création de vues

Créer la vue dans `views.py` :

```
from django.views import generic

class IndexView(generic.TemplateView):
    template_name = 'index.html'
```

Puis la déclarer dans `urls.py` :

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', IndexView.as_view()),
]
```



2. Prise en main

10. Création d'un gabarit (template)

La vue précédente ne va pas fonctionner car il faut créer le gabarit (= template) correspondant.

Pour cela :

- déclarer où aller chercher les gabarits dans `settings.py`
- créer le gabarit dans le dossier concerné

Dans `settings.py`, changer la déclaration du dossier (vide) des gabarits : `'DIRS': []` par :

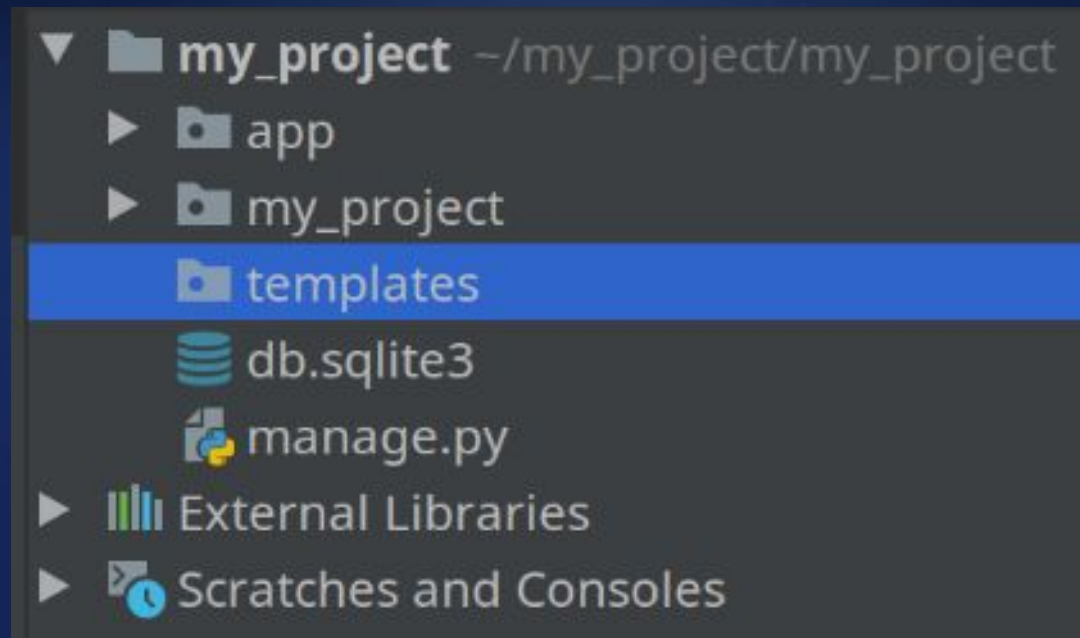
```
TEMPLATES = [{  
    ...  
    'DIRS': [os.path.join(BASE_DIR, 'templates')],  
    ...  
}, ]
```



2. Prise en main

10. Création d'un gabarit (template)

Créer le dossier 'templates' correspondant





2. Prise en main

10. Création d'un gabarit (template)

Dans le dossier 'templates' créer le fichier html correspondant à notre vue (« index.html ») et y mettre ce code html « simple » :

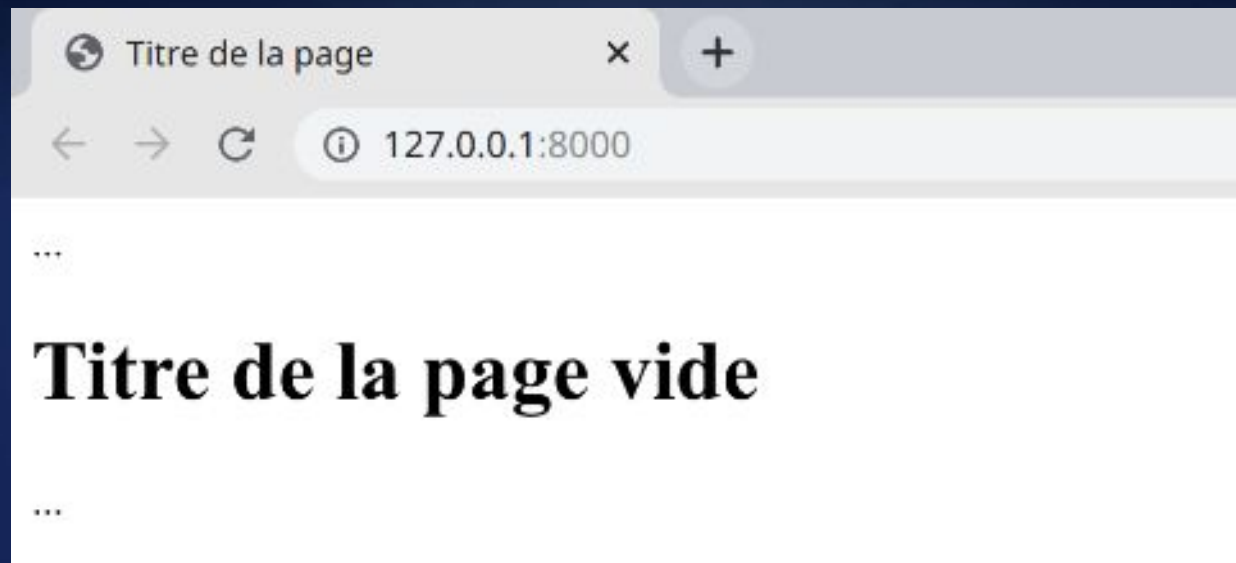
```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Titre de la page</title>
</head>
<body>
  ...
  <h1>Titre de la page vide</h1>
  ...
</body>
</html>
```



2. Prise en main

10. Création d'un gabarit (template)

Afficher la page web fonctionnelle :





3. Les modèles

1. Les types de champs

Dans tous les modèles, Django crée une clé unique « `id` » cachée. De la même manière, la déclaration des clés étrangères ne se fait pas en déclarant le champ, mais *la classe du modèle*.

Ainsi, lors de la création d'un modèle, si on veut déclarer une clé étrangère, l'écriture peut paraître surprenante mais elle est beaucoup plus lisible, exemple :

```
class Color(models.Model) :
    title = models.CharField(max_length=200, blank=True,
                             default=None, null=True)

class AutreModele(models.Model) :
    color = models.ForeignKey(Color, on_delete=models.CASCADE,
                              default=None)
```



3. Les modèles

1. Les types de champs

Application sur notre projet personnel : éditer « models.py » et créer tous les modèles. Solution ici (en petit, à faire seul !)

```
class Recipe(models.Model):
    title = models.CharField(max_length=200, blank=True,
                             default=None, null=True)
    summary = models.CharField(max_length=200, blank=True,
                               default=None, null=True)
    content = models.TextField(blank=True, default=None, null=True)
    nb_persons = models.IntegerField(blank=True, default=2, null=True)

class Ingredient(models.Model):
    singular = models.CharField(max_length=200, blank=True,
                                default=None, null=True)
    plural = models.CharField(max_length=200, blank=True,
                              default=None, null=True)

class Unit(models.Model):
    description = models.CharField(max_length=200, blank=True,
                                   default=None, null=True)

class RecipeIngredientUnit(models.Model):
    recipe = models.ForeignKey(Recipe, on_delete=models.CASCADE,
                               null=True, default=None)
    ingredient = models.ForeignKey(Ingredient, on_delete=models.CASCADE,
                                   null=True, default=None)
    unit = models.ForeignKey(Unit, on_delete=models.CASCADE,
                              default=None, null=True)
    value = models.FloatField(default=0.0, null=True)
```




3. Les modèles

1. Les types de champs

Types de champs complexes :

- clé étrangère :

```
my_other_model = models.ForeignKey(
    other_model, on_delete=models.CASCADE,)
```

- clé « un à un » :

```
my_other_model = models.OneToOne(
    other_model, on_delete=models.CASCADE,)
```

- clé « n – n » :

```
my_other_model = models.ManyToManyField(other_model)
```



3. Les modèles

2. Relations inverses (1/2)

Sur les types de champs complexes, en imaginant partir de la table *opposée*, il est possible de préciser le nom à utiliser via `related_name` : imaginons les modèles `Address` et `Person` :

```
class Address(models.Model) :
    text = models.TextField(default=None,
                            blank=True, null=True)

class Person(models.Model) :
    address = models.ForeignKey(
        Address, on_delete=models.CASCADE,
        related_name="persons")
```

Il serait possible d'accéder à toutes les personnes vivant à une adresse donnée comme ceci :

```
tab = Address.objects.get(text__contains="Xxx").persons
```



3. Les modèles

2. Relations inverses (2/2)

Exemple précédent avec une relation de type ManyToMany :

```
class Address(models.Model) :  
    text = models.TextField(default=None,  
                            blank=True, null=True)  
  
class Person(models.Model) :  
    addresses = models.ManyToManyField(  
        Address, on_delete=models.CASCADE,  
        related_name="persons",)
```

Il serait possible d'accéder à toutes les personnes vivant à une adresse donnée comme ceci :

```
tab = Address.objects.get(text__contains="Xxx").persons
```



3. Les modèles

3. Syntaxe de requêtage Django

Toutes les requêtes passent par la propriété statique « `objects` ». Cet objet est une instance qui fait l'inspection de la classe en cours afin de permettre une écriture de requête simple.

Exemples de requête :

```
Address.objects.get(pk=12)
```

La clause est « `pk=12` ». Il est possible d'enchaîner les clauses sur d'autres modèles via « `__` » (double underscore) et la jointure est construite automatiquement :

```
[modele_a]__[modele_b]__[modele_c]__[clause]=[valeur]
```

Exemple :

```
Address.objects.filter(country__icontains="ance")
```

→ JOIN entre (Address et Country)

```
Person.objects.filter(addresses__way__isnull=False)
```

→ JOIN entre (Person, Address et Way)



3. Les modèles

3. Syntaxe de requêtage Django

Au vu du slide précédent, il est possible de faire des requêtes avancées. Exemples de requête : dire ce qu'elle fait, et faire une vue adéquate :

```
search = #... code
return Recipe.objects.filter(
    Q(recipeingredientunit__ingredient__singular__icontains=search) |
    Q(recipeingredientunit__ingredient__plural__icontains=search)
)
```



3. Les modèles

3. CRUD manuel

Exemple de création / mise à jour / suppression d'un Recipe :

```
>>> from app.models import Recipe
>>> r = Recipe.objects.create(title="Boeuf carottes",
...                           summary=None,)
>>> r
<Recipe: Recipe object (1)>
>>> r.save()
>>> for test in Recipe.objects.all():
...     print(test.title)
...
Boeuf carottes
>>> r.update(summary="Test")
>>> r.delete()
(1, {'app.RecipeIngredientUnit': 0, 'app.Recipe': 1})
>>>
```



4. Administration

1. Personnalisation de l'interface

Dans le fichier `admin.py` :

- soit on déclare « simplement » les modèles pour du CRUD simple
- soit, pour le modèle concerné, on surcharge la classe d'affichage de l'administration Django.

Exemple de surcharge :

```
class RecipeAdmin(admin.ModelAdmin):  
    pass
```

```
admin.site.register(Recipe, RecipeAdmin)
```

Il est possible de faire une interface d'administration entièrement sur mesure, et de ré-écrire même le templating de l'administration !

Voir <https://docs.djangoproject.com/en/dev/ref/contrib/admin/>



4. Administration

1. Personnalisation de l'interface

Listing des modèles « sur mesure » :

Exemple de surcharge :

```
class RecipeAdmin(admin.ModelAdmin):  
  
    def custom_content(self, obj):  
        if len(obj.content) > 80:  
            return f'{obj.content[:80]}...'  
        return obj.content  
  
    custom_content.short_description = "Contenu"  
  
    list_display = ['title', 'summary', 'custom_content']
```




4. Administration

2. Gestion évoluée des modèles

Les `Inline` : dans l'administration Django, il est possible d'éditer un modèle, et si ce dernier a des champs de type `ForeignKey`, il est possible de les éditer via les `Inline`. Exemple :

```
class RecipeIngredientUnitInlineAdmin(admin.StackedInline):  
    model = RecipeIngredientUnit  
    extra = 0
```

```
class RecipeAdmin(admin.ModelAdmin):  
    fields = ('title',)  
    inlines = (RecipeIngredientUnitInlineAdmin,)
```

```
admin.site.register(Recipe, RecipeAdmin)  
admin.site.register(Ingredient)  
admin.site.register(Unit)  
admin.site.register(RecipeIngredientUnit)
```

5. Les templates

1. Principes

Lorsqu'un client (navigateur Web habituellement) demande une URL :





5. Les templates

2. Choix du moteur

Les moteurs de gabarit sont configurés dans le réglage `TEMPLATES`.
C'est une liste de configurations, une par moteur.

Par défaut :

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            # ... options ...  
        },  
    },  
]
```

Jusqu'à Django 3 : Les moteurs intégrés sont :

- `django.template.backends.django.DjangoTemplates`
- `django.template.backends.jinja2.Jinja2`



5. Les templates

3. Le langage DTL

Un gabarit contient des **variables** qui sont remplacées par des valeurs lorsque le gabarit est évalué, ainsi que des **balises = tags** qui contrôlent la logique du gabarit.

Les **variables**

- double accolades exemple : `{{ recipe }}`
- le point = attributs d'une variable exemple : `{{ recipe.title }}`

Les **balises** (ordres) :

- accolades « % » exemple : `{% url "recipes_list" %}`
- notes :
 - `{% block %}` se termine par `{% endblock %}`
 - `{% for ... %}` se termine par `{% endfor %}`

5. Les templates

4. Les filtres

On peut modifier l'affichage des variables en utilisant des filtres.

Les filtres ressemblent à ceci : `{{ nom | filtre }}`.

Ceci affichera le contenu de `nom` après avoir été filtrée par "`filtre`".

Barre verticale = « pipe » = " | " = appliquer un filtre.

Ajouter un paramètre au filtre = " : " = `{{ liste | join:", " }}`.

On peut les additionner : `{{ nom | filtre1 | filtre2 }}`

Quelques filtres natifs dans Django :

- `{{ valeur | lower }}` → valeur en minuscules
- `{{ liste | join:", " }}` → les éléments séparés par ", "
- `{{ valeur | default:"rien" }}` → valeur ou "rien" si valeur vide

→ <https://docs.djangoproject.com/en/dev/ref/templates/language/>



5. Les templates

5. Les balises (tags)

Les balises (tags en anglais) ressemblent à ceci : `{% tag %}`.

Plus complexes que les variables :

- certaines produisent du texte,
- d'autres contrôlent le flux (boucles ou la logique),
- d'autres encore chargent des informations externes (pour que des variables puissent les utiliser ensuite).

Quelques balises natives

- `{% for xxx in yyy %} {% endfor %}`
- `{% if %} ... {% elif %} ... {% endif %}`
- `{% trans "my string" %}`
- `{% blocktrans trimmed %} {% endblocktrans %}`

5. Les templates

6. Les commentaires

Deux types de commentaires :

- avec « # » : `{# mon commentaire #}`.

- avec « comment » :

`{% comment %}`

Mon long commentaire
sur plusieurs lignes

`{% endcomment %}`

Note PyCharm : un commentaire est une région,
→ on peut diminuer n'importe quelle région

```
{% comment %}  
My comment  
{% endcomment %}
```



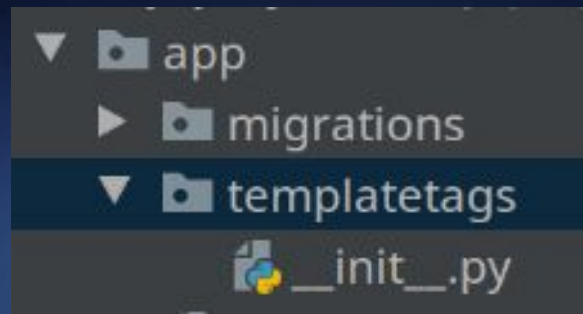
```
{% comment %} My comment {% endcomment ... %}
```



5. Les templates

7. Création de filtres et balises

Dans un dossier « `templatetags` » = en dur, au même niveau que `models.py`, `views.py`...



Exemple de déclaration d'un filtre :

```
@register.filter(name='addstr')
def addstr(arg1, arg2):
    return str(arg1) + str(arg2)
```

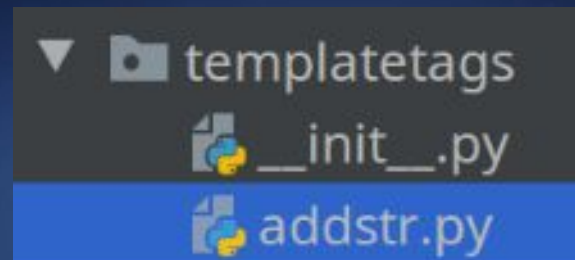
Dans un template, utilisation :

```
id="{ "btn-edit-travel- " | addstr:v.obj.pk }"
```


5. Les templates

7. Création de filtres et balises

Le nom du fichier correspondra au nom « à charger » dans le template : ici, fichier `addstr.py` :



À utiliser dans le template : chargement au début
`{% load addstr %}`

Puis par la suite :

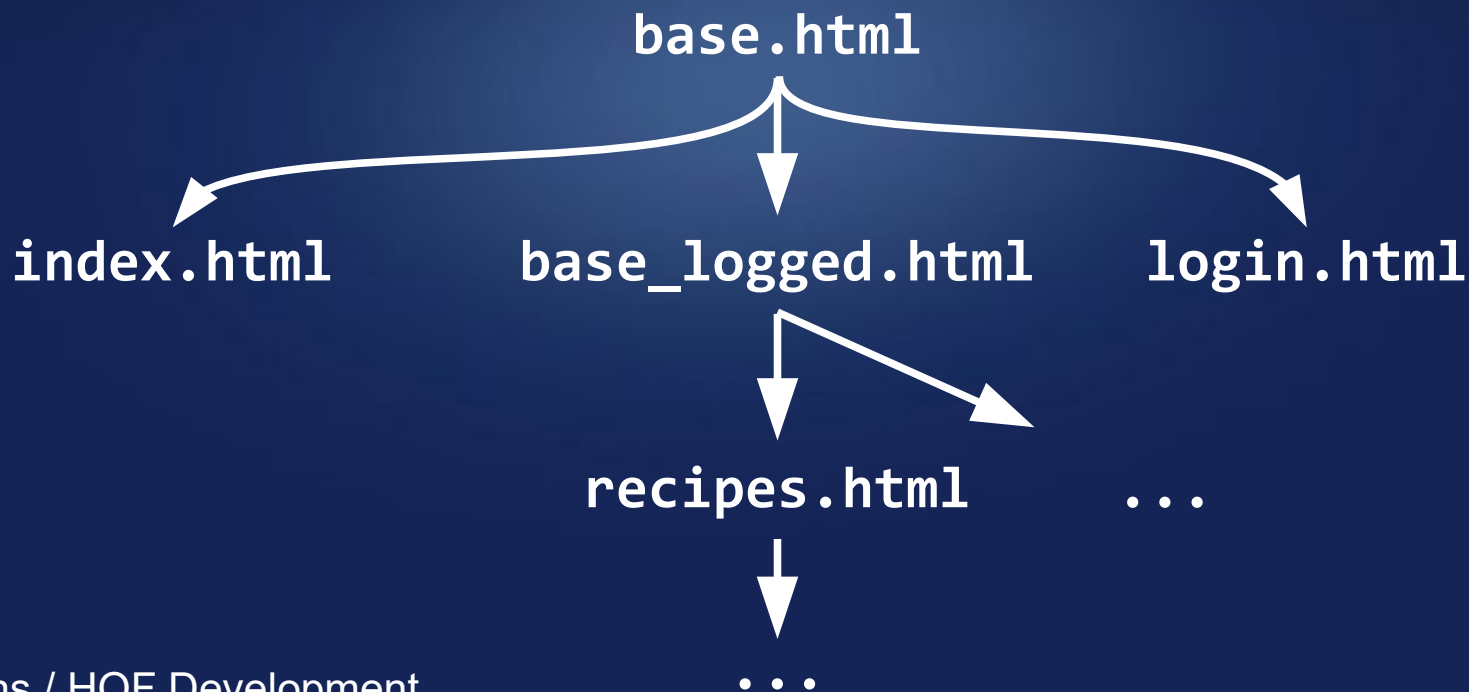
```
id="{ { "btn-edit-travel-" | addstr:v.obj.pk } }
```

5. Les templates

8. Héritage de gabarits

Principe d'organisation global :

- Préparer tout ce qui est commun dans un fichier « principal » (ce fichier est souvent appelé « `base.html` »)
- Préparer des « emplacements » qui seront remplis par les pages qui descendent de cette page « `base.html` ».



5. Les templates

8. Héritage de gabarits

```
base.html
<html>
  <head></head>
  <body>
    {% block content %}
    {% endblock %}
  </body>
</html>
```



```
index.html
{% extends 'base.html' %}
{% block content %}
  <h1>I'm the index!</h1>
{% endblock %}
```

Exemple :
index.html
Il ne reste
qu'à *remplir* les
« **blocks** » →



6. Les vues

1. Les vues basées sur les classes

Habituellement, tous les tutoriels montrent comment afficher une vue ainsi :

```
my_app/urls.py
urlpatterns = [
    # ex: /polls/
    path('', views.index, name='index'),
    # ex: /polls/5/
    path('<int:question_id>/', views.detail, name='detail'),
]
my_app/views.py
def detail(request, question_id):
    return HttpResponse("Question %s." % question_id)
```

Cela fonctionne, mais *ce n'est pas la méthode la plus pérenne. Il faut passer par des vues basées sur les classes.*



6. Les vues

1. Les vues basées sur les classes

Django a prévu un système de vues basées sur des classes très bien organisé : la classe « mère » est **View** puis les descendants héritent de **View** et des mixins « outils » afin d'atteindre les objectifs de la vue en question :

`django.views.generic.base.View`

Vues génériques

TemplateView

RedirectView

Vues génériques d'affichage

ListView

DetailView

CreateView

UpdateView

DeleteView



6. Les vues

2. Les vues génériques d'affichage

Les vues CRUD (`DetailView`, `CreateView`, `UpdateView`, `DeleteView`) demandent à préciser :

- soit en propriété statique un modèle : `model = xx` (le plus simple)
- soit surcharger la méthode `get_object()`

Elles créent une variable nommée « **object** » pour le template qui contient le modèle.

Dans le template, on peut utiliser `{{ object }}`, mais aussi ses propriétés / méthodes : `{{ object.xx }}`.



6. Les vues

2. Les vues génériques d'affichage

Avec le modèle `Recipe` :

`urls.py` :

```
urlpatterns = [  
    path('recipes/<int:pk>/',  
        RecipeDetailView.as_view(),  
        name='recipe_detail'), ]
```

`views.py` :

```
class RecipeDetail(generic.DetailView):  
    model = Recipe
```

ou bien

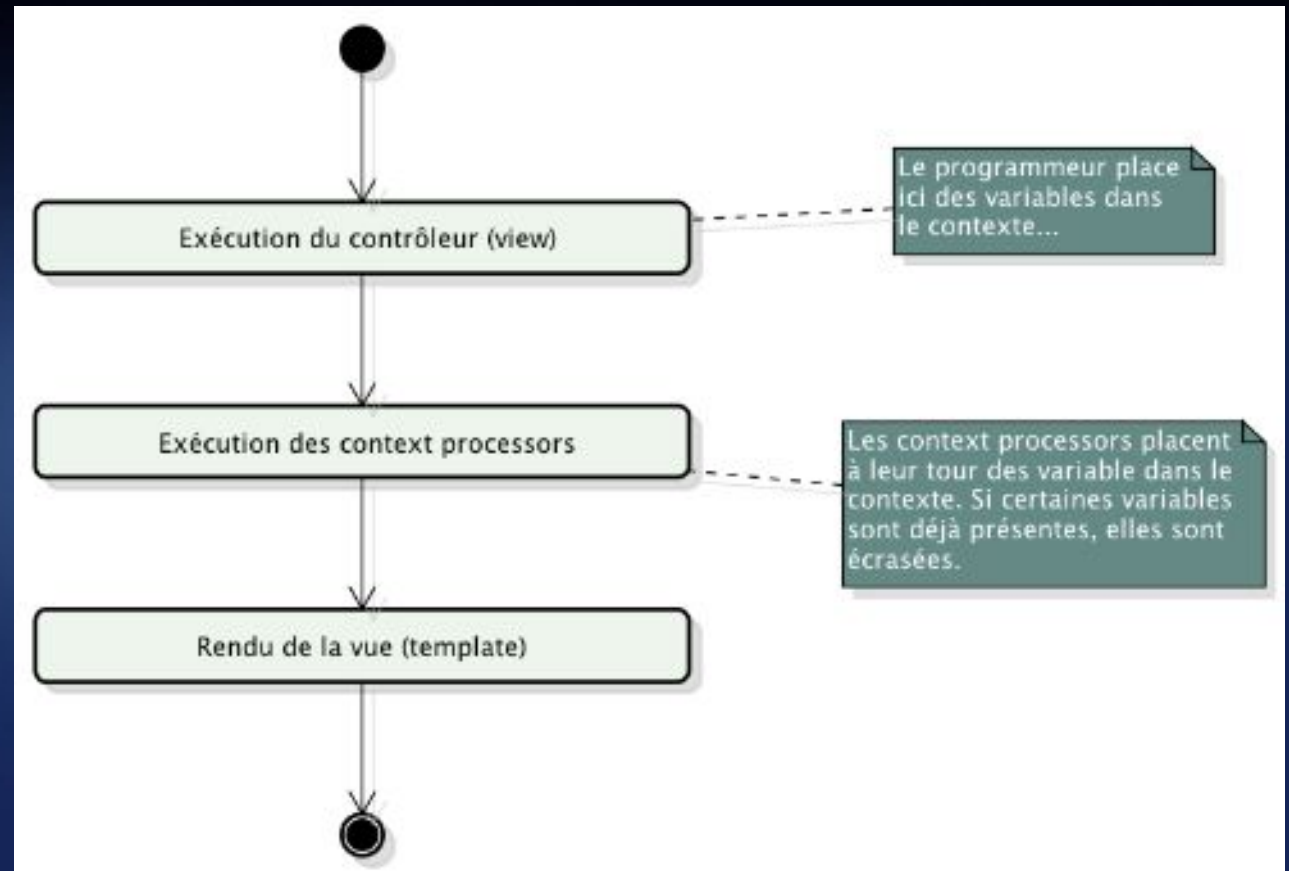
```
class RecipeDetail(generic.DetailView):  
    def get_object(self, queryset=None):  
        return Recipe.objects.get(pk=self.kwargs['pk'])
```

→ Dans le template : on peut utiliser `{{ object }}`,
mais aussi ses propriétés et méthodes : `{{ object.title }}` etc.

6. Les vues

3. Les context processors

Lorsqu'un contrôleur (view) demande le rendu d'une vue (template), elle fournit à cette vue un *contexte*.
Le contexte est un *ensemble de variables et de valeurs*, qui pourront être utilisés dans la vue (template).





6. Les vues

3. Les context processors

Dans les class-based views, cet ensemble de variables arrive dans la méthode `get_context_data()`. Il suffit d'appeler le parent qui constitue le dictionnaire à passer à la vue, et d'y ajouter la valeur que l'on veut.

Exemple qui crée la variable pour le template « `title` » :

```
class IndexView(generic.View):
    def get_context_data(self, **kwargs):
        result = super().get_context_data(**kwargs)
        result['title'] = 'My title'
        return result
```

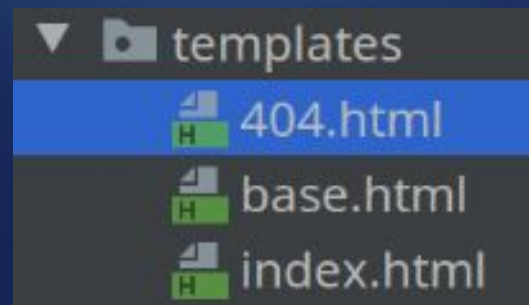


6. Les vues

4. Ré-écriture des pages d'erreur

Pour afficher une 404 sur mesure :

- si `DEBUG = False`, créer un gabarit HTML nommé **404.html** et le placer au premier niveau de l'arborescence des gabarits.
- si `DEBUG = True`, on peut fournir un message aux exceptions `Http404` et il apparaîtra dans le gabarit 404 *standard de débogage*. Ces messages sont à des fins de *débogage*, et *pas adaptés* aux gabarits 404 de production (`DEBUG = False`).

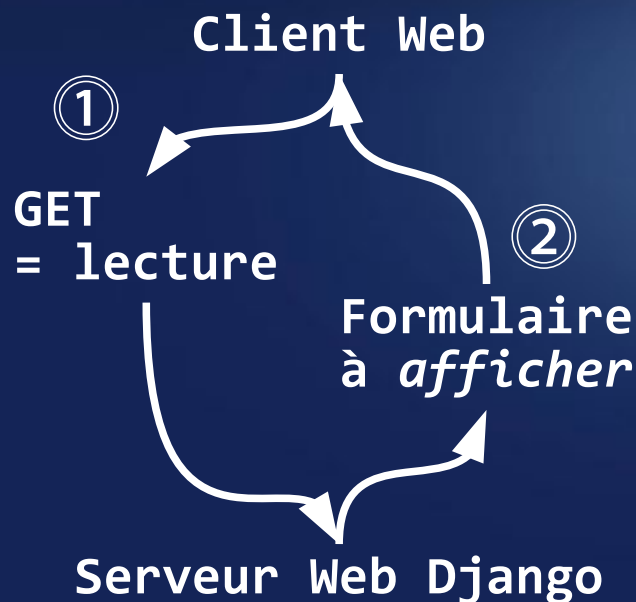


7. Les formulaires

1. Principe

Django gère les formulaires :

- en l'*affichant* via la méthode HTTP « GET »
- en le *validant* + l'*enregistrant* via la méthode HTTP « POST ».



7. Les formulaires

1. Principe

La classe « **Form** » de Django se situe au cœur de ce système. Elle décrit un formulaire et détermine son fonctionnement et son apparence.

Faire un nouveau fichier « `forms.py` » et y mettre ceci :

```
from django import forms
```

```
class RegisterForm(forms.Form):  
    first_name = forms.CharField(label="First name",  
                                max_length=100)  
    last_name = forms.CharField(label="Last name",  
                                max_length=100)
```

Puis, dans « `views.py` », ajouter la vue :

```
class RegisterFormView(generic.FormView):  
    form_class = RegisterForm
```

7. Les formulaires

2. Validation

Lorsque le client (navigateur) envoie le formulaire via la méthode HTTP « POST », voici les étapes appliquées :

- Création de la vue (**XxFormView**)
- Dans cette instance, création du formulaire (propriété **form_class**)
- Appel de **is_valid()** du formulaire
- → pour chaque champ :
 - appel de **clean_[champ]** (si elle existe)
- → appel « final » de **clean()**
- Si le formulaire est valide, appel de la méthode **form_valid(self, form)** de la vue (**XxFormView**) : cf slides suivants

7. Les formulaires

2. Validation

Amélioration de la classe du formulaire : exemple d'un `clean_xx` :

```
def clean_first_name(self)
    data = self.cleaned_data['first_name']
    if not data.isalnum():
        raise forms.ValidationError("Only alpha num!")
    # ou bien : forms.add_error('first_name',
    #                               "Only alpha num!")
    # toujours renvoyer une valeur,
    # même si elle n'a pas changé :
    return data
```

- Ajouter les champs `password_1`, `password_2`
- Créer les méthodes `clean_password_1`, `clean_password_2`
- Créer la méthode « globale » `clean`

7. Les formulaires

3. Enregistrement

Si le formulaire est valide, appel de la méthode `form_valid(self, form)` de la vue (`XxFormView`) : elle peut utiliser toutes les données du formulaire qui sont dans le dictionnaire `form.cleaned_data` (= données validées ci-avant).

Exemple :

```
class RegisterFormView(FormView):
    template_name = 'register.html'
    form_class = RegisterForm
    success_url = '/register/thanks/'

    def form_valid(self, form):
        # appelée quand le formulaire est valide
        # accéder à form.cleaned_data
        ... (code) ...
        # renvoyer un HttpResponseRedirect = appel du parent suffit :
        return super().form_valid(form)
```

7. Les formulaires

4. Templating

Template de formulaire : le plus simple :

```
<form action="" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

Template de formulaire : champ par champ : au lieu de {{ form }} :

```
{{ form.non_field_errors }}
<div>
    {{ form.first_name.errors }}
    <label for="{{ form.first_name.id_for_label }}">
        First name:
    </label>
    {{ form.first_name }}
</div>
```


7. Les formulaires

4. Templating

Template de formulaire : la méthode « générique » :

```
{{ form.non_field_errors }}  
<form action="" method="post">  
    {% csrf_token %}  
    {% for field in form %}  
        <label>{{ field.label }}</label>  
        {{ field }}  
    {% endfor %}  
    <input type="submit" value="Submit">  
</form>
```

7. Les formulaires

4. Templating

Template de formulaire : le plus simple :

```
<form action="" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>
```

Template de formulaire : champ par champ : au lieu de {{ form }} :

```
{{ form.non_field_errors }}
<div>
    {{ form.first_name.errors }}
    <label for="{{ form.first_name.id_for_label }}">
        First name:
    </label>
    {{ form.first_name }}
</div>
```



8. Divers

1. Internationalisation

Pour choisir la langue, Django se base sur l'entête reçu **Accept-Language** par la demande Web.

Internationalisation = tâche sur Django, par les développeurs

Régionalisation = tâche par les traducteurs.

Traduction dans le code :

```
from django.views import generic
from django.utils.translation import gettext_lazy as _
from app.models import Recipe

class IndexView(generic.TemplateView):
    template_name = 'index.html'

    def get_context_data(self, **kwargs):
        result = super().get_context_data(**kwargs)
        result['title'] = _('My title')
        return result
```



8. Divers

1. Internationalisation

Pour activer l'internationalisation, il faut installer le middleware « `LocaleMiddleware` », qui n'est pas installé par défaut. Ce dernier doit être entre « `SessionMiddleware` » et « `CommonMiddleware` », *pas avant ni après par rapport au traitement de la requête dans Django.*

```
MIDDLEWARE = [  
    # maybe some middleware before  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    # maybe some middleware in-between  
    'django.middleware.locale.LocaleMiddleware',  
    # maybe some middleware in-between  
    'django.middleware.common.CommonMiddleware',  
    # maybe some middleware after  
]
```



8. Divers

1. Internationalisation

- Créer un dossier « `locale` », à la racine du projet.
Dans « `settings.py` », ajouter le code suivant, qui :
- déclare où chercher les chaînes de traduction,
 - déclare les langages activables sur le site

```
LOCALE_PATHS = (  
    os.path.join(BASE_DIR, 'locale'),  
)  
LANGUAGES = (  
    ('en', _('English')),  
    ('fr', _('French')),  
)
```



8. Divers

1. Internationalisation

`gettext()` vs `gettext_lazy()`

La version « paresseuse » = « lazy » contient une *référence* à la chaîne de traduction au lieu du texte traduit, de sorte que la traduction se produit lors de l'accès à la valeur plutôt que lors de son appel.

Dans un projet Django, il y a plusieurs cas où le code n'est exécuté qu'*une seule fois* (au démarrage de Django).

Cela se produit avec des modules de définition tels que des modèles, des formulaires et des formulaires de modèle.



8. Divers

1. Internationalisation

`gettext()` vs `gettext_lazy()`

Problème classique

- Django démarre, la langue par défaut est l'anglais ;
- Django choisit la version anglaise des étiquettes de champ ;
- L'utilisateur change la langue du site Web en espagnol ;
- Les étiquettes sont toujours affichées en anglais !

C'est parce que la définition de champ n'est appelée qu'une seule fois.

Solution : utiliser `gettext_lazy()`

En général, **pour toute traduction susceptible de changer après le démarrage du serveur Django, il est préférable d'appeler les méthodes « `_lazy()`. »**



8. Divers

1. Internationalisation

Étapes systématiques à suivre pour la traduction :

- ajouter des chaînes (dans code ou dans les templates)
- lancer la recherche / mise à jour des fichiers de traduction :
`makemessages --locale fr --locale en`
- compiler les messages pour qu'ils soient utilisables par Django :
`compilemessages`
- relancer le serveur Web (il ne se relance pas automatiquement!)

<https://docs.djangoproject.com/en/dev/topics/i18n/translation/>



8. Divers

1. Internationalisation

Dans le code :

```
from django.utils.translation import gettext_lazy as _  
# utilisation via _("My string")  
# forcer la traduction : str(_("My string"))
```

- Dans les templates :

```
{% trans "My string" %}  
{% blocktrans trimmed %}  
This is my block on many lines  
{% endblocktrans %}
```



8. Divers

2. Flash messages

Souvent, les applications Web ont besoin d'afficher des messages de notification (aussi appelés « messages flash »). Elles ne s'afficheront qu'*une seule fois* lors de la demande de la prochaine page Web.

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

Raccourcis utiles :

```
messages.debug(request, 'Total users: %s' % count)
messages.info(request, 'Your account is inactive.')
messages.success(request, 'Profile details updated.')
messages.warning(request, '3 dangerous messages.')
messages.error(request, 'Document not found, maybe deleted.')
```



8. Divers

2. Flash messages

Gestion dans le code Python :

```
from django.contrib.messages import get_messages

storage = get_messages(request)
for message in storage:
    do_something_with_the_message(message)
```

Gestion dans les templates :

```
{% if messages %}
<ul class="messages">
  {% for message in messages %}
  <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
    {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}
      Important:
    {% endif %}
    {{ message }}
  </li>
  {% endfor %}
</ul>
{% endif %}
```

8. Divers

3. Optimisations

Code à mettre dans `settings.py` pour logger toutes les requêtes que Django fait dans la console :

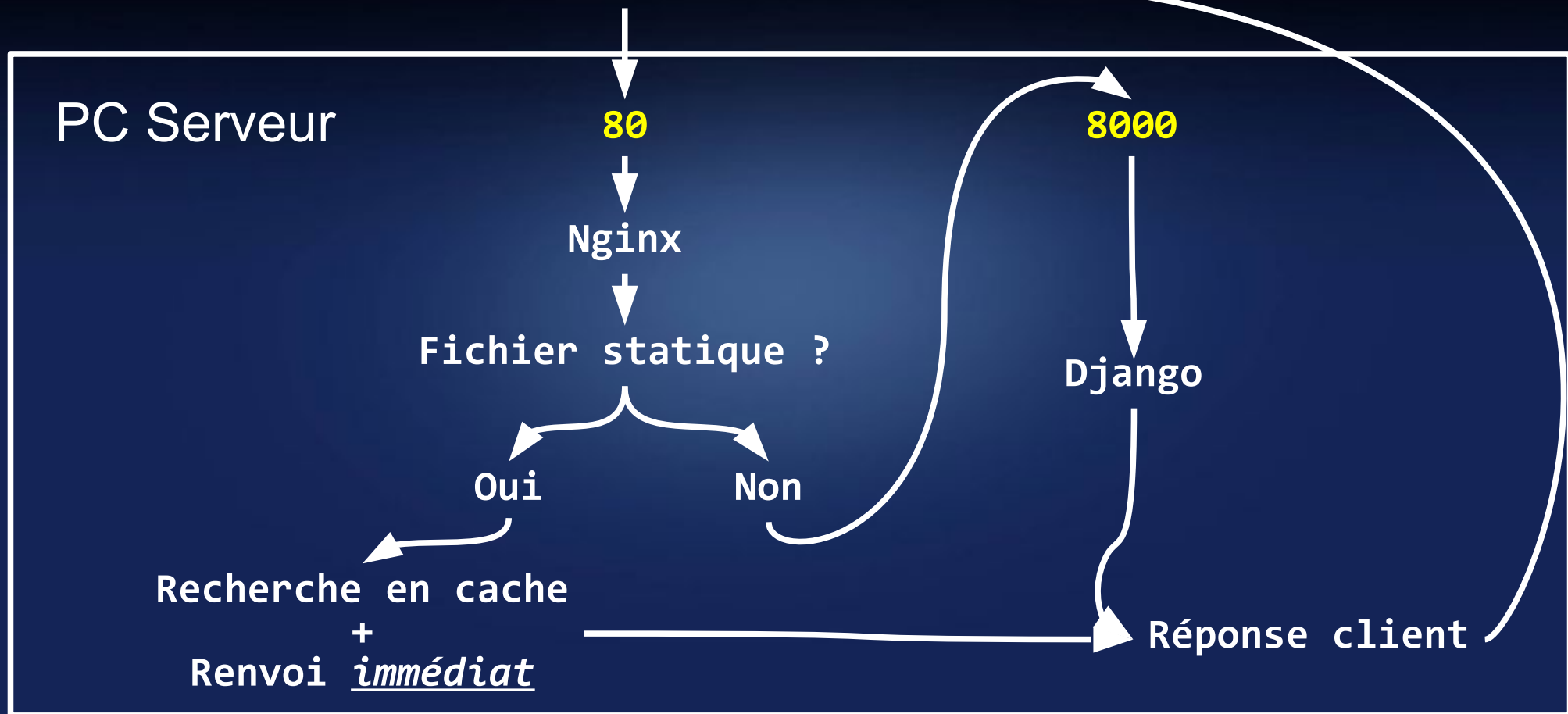
```
LOGGING = {
    'disable_existing_loggers': False,
    'version': 1,
    'handlers': {
        'console': {
            # logging handler that outputs log messages to terminal
            'class': 'logging.StreamHandler',
            'level': 'DEBUG', # message level to be written to console
        },
    },
    'loggers': {
        '': {
            # this sets root level logger to log debug and higher level
            # logs to console. All other loggers inherit settings from
            # root level logger.
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': False, # this tells logger to send logging message
                               # to its parent (will send if set to True)
        },
        'django.db': {
            # django also has database level logging
            'level': 'DEBUG'
        },
    },
}
```



9. Déploiement

1. Intégration Nginx (1/4)

Client = Navigateur Web





9. Déploiement

1. Intégration Nginx (2/4)

Configuration de nginx dans `/etc/nginx/sites-available/`

```
upstream monsite {
    ip_hash;
    server 127.0.0.1:8006;
}
server {
    listen *:80;
    server_name monsite.fr monsite.com www.monsite.fr;
    index index.html index.htm;
    access_log /var/log/nginx/proxy-access-monsite.log combined;
    error_log /var/log/nginx/proxy-error-monsite.log error;

```

...



9. Déploiement

1. Intégration Nginx (3/4)

Configuration de nginx dans `/etc/nginx/sites-available/`

...

```
# pour les statiques
# ~ = expression régulière
# ~* = expression régulière case *insensitive*
location ~* ^/static/(?<p>.+ ) {
    root /web/htdocs/monsite/htdocs/static;
    try_files /$p /production/$p =403;
    access_log off;
    expires 1h;
}
location ~* ^/public/(?<p>.+ ) {
    root /web/htdocs/monsite/htdocs/uploads;
    try_files /$p =403;
    access_log off;
    expires 1h;
}
```

...



9. Déploiement

1. Intégration Nginx (4/4)

Configuration de nginx dans `/etc/nginx/sites-available/`

```
...  
location / {  
    expires -1;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    proxy_set_header X-Server $host;  
    proxy_pass http://monsite/;  
}  
}
```




9. Déploiement

2. Checklist de déploiement

- Mettez tous les settings en mode « production », rappel :

```
DEBUG = False
```

```
ALLOWED_HOSTS = [  
    'siteweb.fr',  
    '127.0.0.1',  
    'localhost:8000',  
    'localhost',  
]
```

```
ADMINS = (  
    ('Olivier Pons', 'olivier.pons@gmail.com'),  
)
```

- Lancez `manage.py check --deploy`

- Regardez tous les conseils