Formation Django,développement Web avec Python

Matthieu Falce

Décembre 2022

Au programme I

Programmation Orientée objet (POO)

- Concepts
- Association
- Modélisation
- POO en python
- Gestion des exceptions
- Classe ou pas ?
- Méthodes
- Bibliographie

Django

- Contexte
- Stockage des données – SQL
- ORM
- Vues
- Templates
- Formulaires
- Authentification

Au programme II

- Admin
- Aller plus loin
- Déploiement

A propos de moi – Qui suis-je ?

- Qui suis-je ?
  - Matthieu Falce
  - habite à Lille
  - ingénieur en bioinformatique (INSA Lyon)

A propos de moi – Qui suis-je ?

- Qu'est ce que j'ai fait ?
  - ingénieur R&D en Interaction Homme-Machine (IHM), Inria Lille, équipe Mint puis Mjolnir
  - développeur *fullstack* / *backend* à FUN-MOOC (France Université Numérique)

A propos de moi – Actuellement

- entrepreneur salarié dans une SCOP (Société COOPérative) : MFconsulting
  - conseil en python
  - rédaction de dossier de financement de l'innovation
  - formations

A propos de moi – Actuellement

- entrepreneur salarié dans une SCOP (Société COOPérative) : MFconsulting
  - conseil en python
  - rédaction de dossier de financement de l'innovation
  - formations
- créateur de *Oh Ce Cours Formation*

A propos de moi – Actuellement

- entrepreneur salarié dans une SCOP (Société COOPérative) : MFconsulting
  - conseil en python
  - rédaction de dossier de financement de l'innovation
  - formations
- créateur de *Oh Ce Cours Formation*

A propos de moi – Actuellement

- entrepreneur salarié dans une SCOP (Société COOPérative) : MFconsulting
  - conseil en python
  - rédaction de dossier de financement de l'innovation
  - formations
- créateur de *Oh Ce Cours Formation*
- cofondateur / CTO de ExcellencePriority (site de partage exclusif de petites annonces orienté luxe)

A propos de moi – Actuellement

- entrepreneur salarié dans une SCOP (Société COOPérative) : MFconsulting
  - conseil en python
  - rédaction de dossier de financement de l'innovation
  - formations
- créateur de *Oh Ce Cours Formation*
- cofondateur / CTO de ExcellencePriority (site de partage exclusif de petites annonces orienté luxe)
- coorganisateur de meetups à Lille
  - python
  - big data et machine learning

Où me trouver ?

- mail: matthieu@falce.net
- github : ice3
- twitter : @matthieufalce
- site: falce.net

Programmation Orientée objet (POO)

Matthieu Falce

1- Programmation Orientée objet (POO)

1.1. Concepts

Programmation orientée objet (POO)

Matthieu Falce

La POO consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs.

---

https://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_objet

Programmation orientée objet (POO)

Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.

---

https://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_objet

Constitution d'une classe

Une classe est constituée de 2 entités (en gros) :

- les méthodes : des "fonctions" qui s'appliquent sur un objet
- les attributs : des "variables" qui s'appliquent sur un objet

Constitution d'une classe

Une classe est constituée de 2 entités (en gros) :

- les méthodes : des "fonctions" qui s'appliquent sur un objet
- les attributs : des "variables" qui s'appliquent sur un objet

Cela permet de conserver le *comportement* et *l'état* à l'intérieur de l'instance.
Des appels à des méthodes vont modifier l'état interne en changeant les attributs.

Constitution d'une classe

Une classe est constituée de 2 entités (en gros) :

- les méthodes : des "fonctions" qui s'appliquent sur un objet
- les attributs : des "variables" qui s'appliquent sur un objet

Cela permet de conserver le *comportement* et *l'état* à l'intérieur de l'instance.
Des appels à des méthodes vont modifier l'état interne en changeant les attributs.

Une classe est une *boîte noire*. On interagit avec elle à l'aide de quelques leviers et boutons sans savoir ce qui se passe à l'intérieur.

Vocabulaire

- une classe défini un nouveau *type* (comme `int`)
- un *objet* est une *instance* d'une classe (comme 2 est une instance de *int*)

1- Programmation Orientée objet (POO)

1.2. Association

Association entre classes

2 grandes techniques pour associer des classes entre elles :

- *héritage* (*inheritence* en anglais): on étend une classe mère en faisant un nouveau type qui le restreint
  - modélise la relation *"est un"*
  - le type fille peut être utilisé à la place du type mère (*polymorphisme*)
  - on peut redéfinir ou *surcharger* certains comportements (méthodes, attributs)
  - les relations classe mère / classe fille définissent un *arbre d'héritage*

Association entre classes

2 grandes techniques pour associer des classes entre elles :

- *composition* : on étend une classe en l'utilisant comme attribut d'une classe
  - modélise la relation *"possède un"*
  - assouplit la relation de dépendance

1- Programmation Orientée objet (POO)

1.3. Modélisation

UML

Le Langage de Modélisation Unifié, de l'anglais Unified Modeling Language (UML), est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système. Il est couramment utilisé en développement logiciel et en conception orientée objet.

---

https://fr.wikipedia.org/wiki/UML_(informatique)

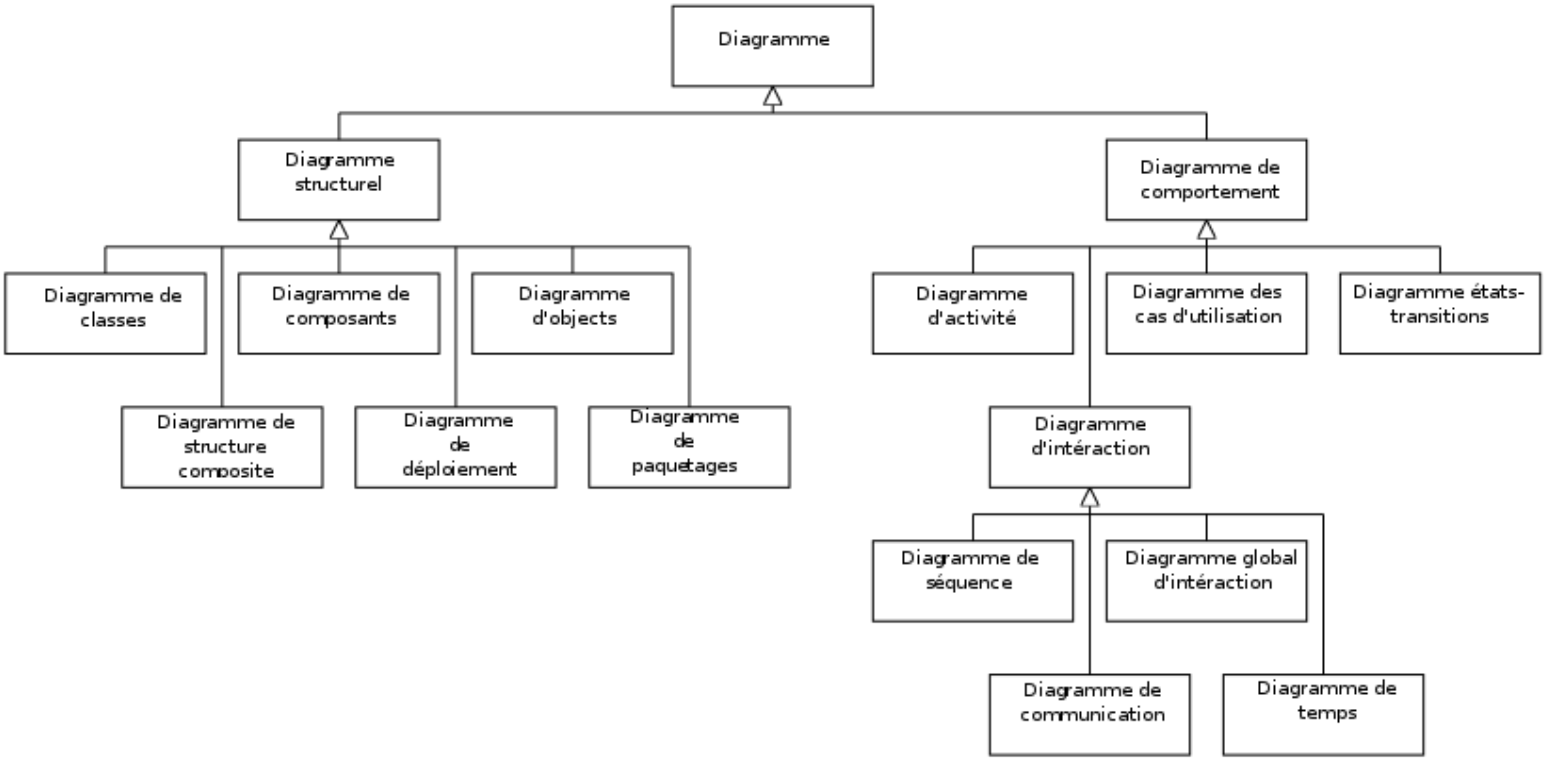
UML

Différents types de diagrammes

- *diagramme de classes* : représente les classes intervenant dans le système
- diagramme d'objets : représente les instances de classes
- diagramme d'activité : représente la suite des actions à effectuer dans le programme
- ...

UML

Diagramme montrant la hiérarchie de types de diagrammes UML

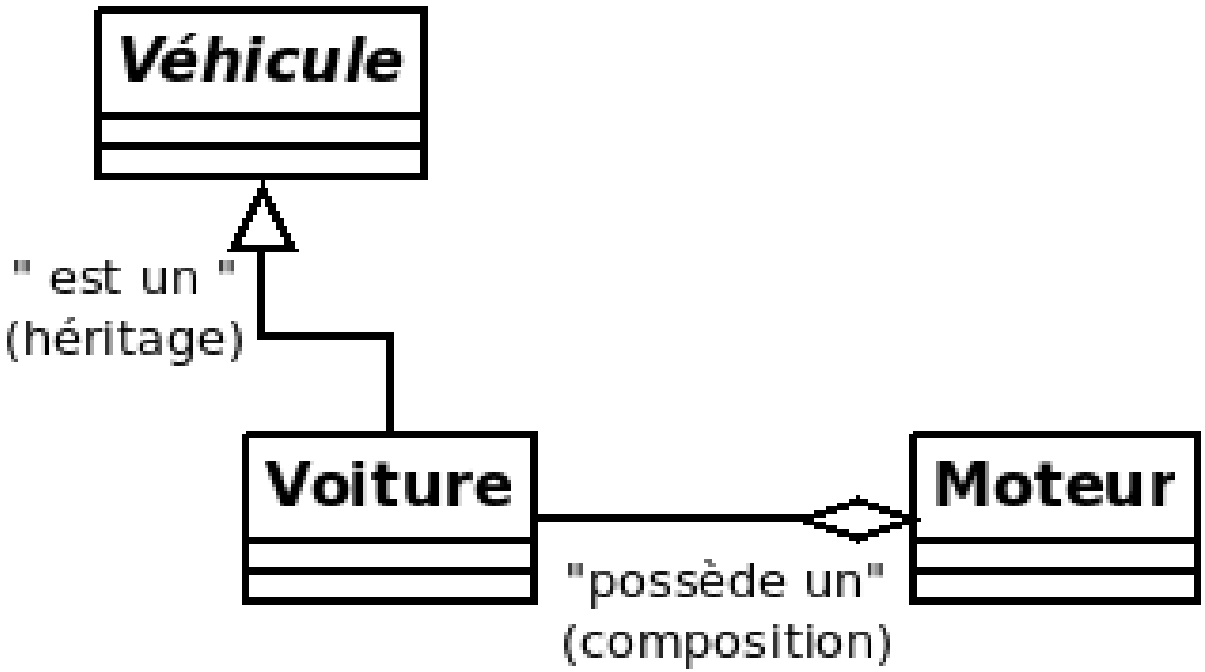


source: https://fr.wikipedia.org/wiki/UML_(informatique)#/media/File:Uml_diagram-fr.png

Matthieu Falce

Diagrammes de classe

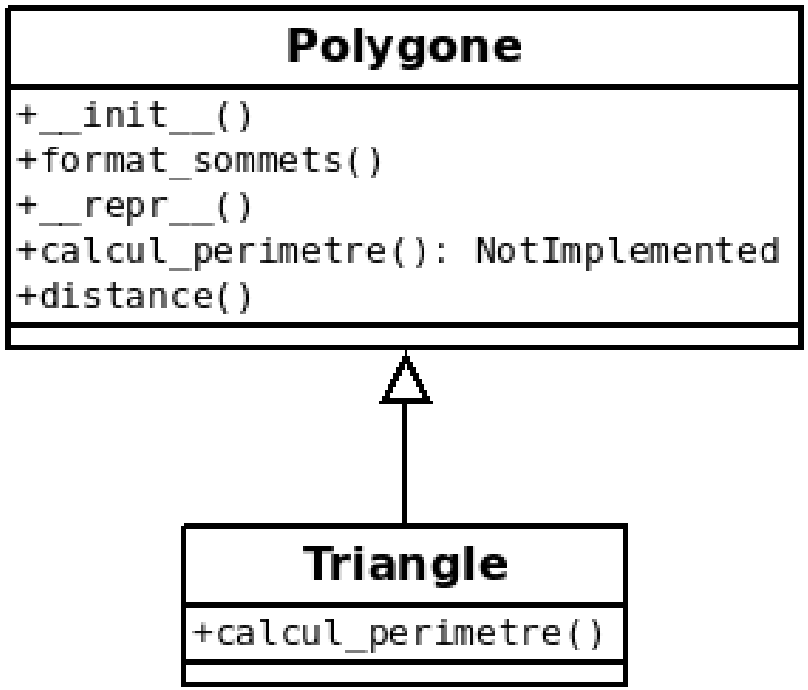
Diagramme de classes montrant composition et héritage



source: https://waytolearnx.com/2018/08/difference-entre-heritage-et-composition.html

Héritage

Diagramme de classes montrant un exemple d'héritage



1- Programmation Orientée objet (POO)

1.4. POO en python

Matthieu Falce

Créer une classe

```
class MonObjet():
    pass


o = MonObjet()
print(o)
```

Créer une classe

```python
#  Contructeur, méthodes et attributs


class MonAutreObjet:
    def __init__(self, nom):
        self.nom = nom

    def dis_ton_nom(self):
        print("Bonjour, je suis {}".format(self.nom))


o1 = MonAutreObjet(1)
o2 = MonAutreObjet(2)

print(o1.nom)
print(o2.nom)

o1.dis_ton_nom()
o2.dis_ton_nom()
```

Créer une classe

```
#  Les attributs sont dynamiques et ajoutable
#  TOUT EST PUBLIC (en première approximation)


class DisBonjour():
    def dis_bonjour(self):
        print("Bonjour : {}".format(self.nom))


d = DisBonjour()
try:
    # ne fonctionne pas ici, self.nom n'est pas défini
    d.dis_bonjour()
except NameError:
    pass

d.nom = "Toto"  # on définit un nom à qui dire bonjour
d.dis_bonjour()
d.nom = "Tata"
d.dis_bonjour()
```

Matthieu Falce

Méthodes magiques

Certaines méthodes (les ___*___) sont utilisées par l'interpréteur pour modifier le comportement des objets.

La plus connue est `__init__` qui permet d'initialiser l'objet.

Mais il y en a d'autres.

Méthodes magiques


```python
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        """Appelée lors de print(Point(1,1))."""
        return "({}, {})".format(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        print(self, other)
        return self.x < other.x  # bah

    def __gt__(self, other):
        return not self.__lt__(other)


p1 = Point(1, 1)
p2 = Point(2, 1)
assert (p1 < p2) is True
```

Héritage

```python
class Bonjour():
    """Classe "abstraite"
    """
    def __init__(self, nom):
        self.nom = nom

    def dis_ton_nom(self):
        # Méthode "abstraite"
        raise NotImplementedError


class BonjourFrancais(Bonjour):
    def dis_ton_nom(self):
        print("Bonjour, je suis {}".format(self.nom))


class BonjourItalien(Bonjour):
    def dis_ton_nom(self):
        print("Ciao, sono {}".format(self.nom))

# le __init__ et le nom sont gérés dans la classe mère
```

Matthieu Falce

Héritage

```python
import math

class Polygone():
    def __init__(self, sommets):
        self.sommets = [tuple(p) for p in sommets]
        self.name = "Polygone"

    def format_sommets(self):
        return " - ".join([str(point) for point in self.sommets])

    def __repr__(self):
        return "{}: {}".format(self.name, self.format_sommets())

    def calcule_perimetre(self):
        raise NotImplementedError

    def distance(self, a, b):
        return math.sqrt((a[0]-b[0]) ** 2 + (a[1] - b[1]) ** 2)


class Triangle(Polygone):
    def __init__(self, sommets):
        super().__init__(sommets)  # /!\
        self.name = "triangle"

    def calcule_perimetre(self):
        cotes = [
            (self.sommets[0], self.sommets[1]),
            (self.sommets[1], self.sommets[2]),
            (self.sommets[2], self.sommets[0])
        ]
        ds = [self.distance(p1, p2) for p1, p2 in cotes]
        return sum(ds)
```

Matthieu Falce

Accès aux attributs

Les attributs sont publics par défaut. Comment protéger certaines contraintes dans ce cas ?

- contrat avec les autres développeurs : variables "privées", préfixées par _ : (`_temperature`)
- on peut préfixer avec un double underscore (`__temperature`) pour les rendre inaccessible hors de l'instance (l'attribut est renommé automatiquement par l'interpréteur)
- `getters` / `setters` : utiliser les `properties`

1- Programmation Orientée objet (POO)

1.5. Gestion des exceptions

Capturer une exception

```python
# on peut capturer une exception
try:
    a = 1 / 0
except Exception as e:
    print(e)
else:
    print("Si pas d'exception")
finally:
    print("Dans tous les cas")

# il faut essayer d'être plus précis dans son exception
try:
    a = 1 / 0
    print(a)
except ZeroDivisionError as e:
    print(e)

# on peut capturer plusieurs exceptions
li = [0]
try:
    calcul = 1 / li[0]
    print(a)
except (IndexError, ZeroDivisionError) as e:
    print(e)
```

Matthieu Falce

Lever une exception – Personnalisation

```python
# On peut lever des exceptions dans certains cas
def notation(note):
    if 0 < note < 20:
        raise ValueError(
            "une note est entre 0 et 20, pas {}".format(note)
        )
    # faire des choses avec la note correcte


# ============================================

# On peut créer ses propres exceptions
# Les exceptions héritent toutes de Exception,
# c'est pour ça que 'except Exception' fonctionne


class MaBelleException(Exception):
    pass
```

Matthieu Falce

Taxonomie d'exceptions de la DB API

Taxonomie des exceptions d'après la PEP 249

```
StandardError
|__Warning
|__Error
    |__InterfaceError
    |__DatabaseError
        |__DataError
        |__OperationalError
        |__IntegrityError
        |__InternalError
        |__ProgrammingError
        |__NotSupportedError
```

1- Programmation Orientée objet (POO)

1.6. Classe ou pas ?

Quand utiliser une classe ?

```
class Bonjour():
    def __init__(self, nom):
        self.nom = nom

    def parle(self):
        return "Bonjour {}".format(self.nom)


bonjour = Bonjour("Matthieu")
print(bonjour.parle())
```

Quand utiliser une classe ?

```python
class Bonjour():
    def __init__(self, nom):
        self.nom = nom

    def parle(self):
        return "Bonjour {}".format(self.nom)


bonjour = Bonjour("Matthieu")
print(bonjour.parle())
```

```python
def bonjour(nom):
    return "Bonjour {}".format(nom)

print(bonjour("Matthieu"))
```

Quand utiliser une classe ?

- Ne pas utiliser
  - quand moins de 2 méthodes...
  - seulement conteneurs, pas de méthodes (utiliser plutôt `dict`, `namedtuple`, ...)
  - gestion des ressources (plutôt `context manager`)

Quand utiliser une classe ?

- Ne pas utiliser
  - quand moins de 2 méthodes...
  - seulement conteneurs, pas de méthodes (utiliser plutôt `dict`, `namedtuple`, ...)
  - gestion des ressources (plutôt `context manager`)
- Utiliser une classe
  - organisation (boîte noire)
  - conserver un état
  - profiter de l'OOP (héritage, ...)
  - surcharge d'opérateurs / méthodes magiques
  - produire une API définie

Conteneurs

```
Point2d = collections.namedtuple('Point2d', ['x', 'y'])
p1 = Point2d(3, 2)
p2 = Point2d(10, 1)

dist = math.sqrt(
    (p2.x - p1.x)**2 + (p2.y - p1.y)**2
)
```

Matthieu Falce

Dataclasses

Version python $\geqslant$ 3.7

```python
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

1- Programmation Orientée objet (POO)

1.7. Méthodes

Différents types de méthodes

```python
class Exemple():
    def __init__(self, attribut):
        self.attribut = attribut

    def methode(self, param):
        print(self, type(self))
        return self.attribut + param


e = Exemple(10)
print(e.methode(2))
```

`method`

- classique
- s'applique à une instance
- accès aux variables de classe et d'instance
- `self` est injecté automatiquement (`bound method`)

Différents types de méthodes

```
class Exemple():
    variable_de_classe = 1

    @classmethod
    def methode_de_classe(cls, param):
        print(cls, type(cls))
        return cls.variable_de_classe + param


print(Exemple.methode_de_classe(5))
```

classmethod

- s'applique sur une classe et pas une instance
- accès aux variables de classe
- `cls` est injecté automatiquement

Différents types de méthodes

```
class Galette():
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def complete(cls):
        return cls(["jambon", "fromage", "oeuf"])

    @classmethod
    def nature(cls):
        return cls(["beurre salé"])


print(Galette.complete().ingredients)
```

Différents types de méthodes

```
class Exemple():
    @staticmethod
    def methode_statique(param):
        return param


print(Exemple.methode_statique(5))
```

staticmethod

- permet de regrouper des fonctions dans l'objet
- n'a accès à aucune information classe ou instance
- ne va pas modifier l'état de la classe ou de l'instance

Résumé

Quel est le résultat ?

```
class MyClass:
    def method(self):
        return ”méthode d'instance”, self

    @classmethod
    def _classmethod(cls):
        return 'méthode de classe', cls

    @staticmethod
    def _staticmethod():
        return 'méthode statique'


print(MyClass._staticmethod())
print(MyClass._classmethod())
print(MyClass.method())

m = MyClass()
print(m._staticmethod())
print(m._classmethod())
print(m.method())
```

1- Programmation Orientée objet (POO)

1.8. Bibliographie

Bibliographie I

- Tous les sujets :
  - http://www.dabeaz.com/py3meta/Py3Meta.pdf
- Classe ou pas
  - https://eev.ee/blog/2013/03/03/the-controller-pattern-is-awful-and-other-oo-heresy/
  - https://www.youtube.com/watch?v=o9pEzgHorH0
  - http://lucumr.pocoo.org/2013/2/13/moar-classes/
- Méthodes de classe / statiques / méthode :
  - https://realpython.com/instance-class-and-static-methods-demystified/
  - commentaire de l'article http://sametmax.com/comprendre-les-decorateur-python-pas-a-pas-partie-2/
  - https://rushter.com/blog/python-class-internals/

Bibliographie II

- Loi de Demeter :
  - https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf

Django

2- Django

2.1. Contexte

Qu'est ce que django I

- framework web écrit en python à haut niveau, adapté à un développement rapide et pragmatique
- *"For perfectionists with deadlines*
- abstrait la plupart des difficultés du développement web pour permettre au développeur de se concentrer sur l'écriture du site en lui même
- *open source*
- développé en 2003 à la base pour un journal local
- depuis 2008, la DSF (*django software foundation*) s'occupe du développement et de la promotion du framework (site, conférence...)
- utilisé dans des gros sites grands public comme Disqus (gestion de commentaires), addons.mozilla.org, Pinterest ou Instagram
- version actuelle : *4.1* / python 3.8

Avantages de django

- système d'authentification intégré
- génération automatique d'un admin
- debug facile pendant le développement (les exceptions python apparaissent dans la page d'erreur)
- documentation très complète et en grande partie traduite en français
- ...

Principe de fonctionnement

Django utilise un patron de conception (*design pattern*) proche du *MVC (Model View Controller)* ou *MVT (Model View Template)*. Le système se découpe donc en 3 grandes parties :

- la vue ou le template : s'occupe de l'affichage des informations (le plus souvent du HTML)
- le contrôleur : géré par la partie de routage d'URL (savoir quelle fonction appeler pour quelle URL)
- le modèle : gérant les données à afficher et l'accès aux données depuis / vers base de données

Architecture d'un projet django

Matthieu Falce

Django est un framework basé sur le concept suivant :

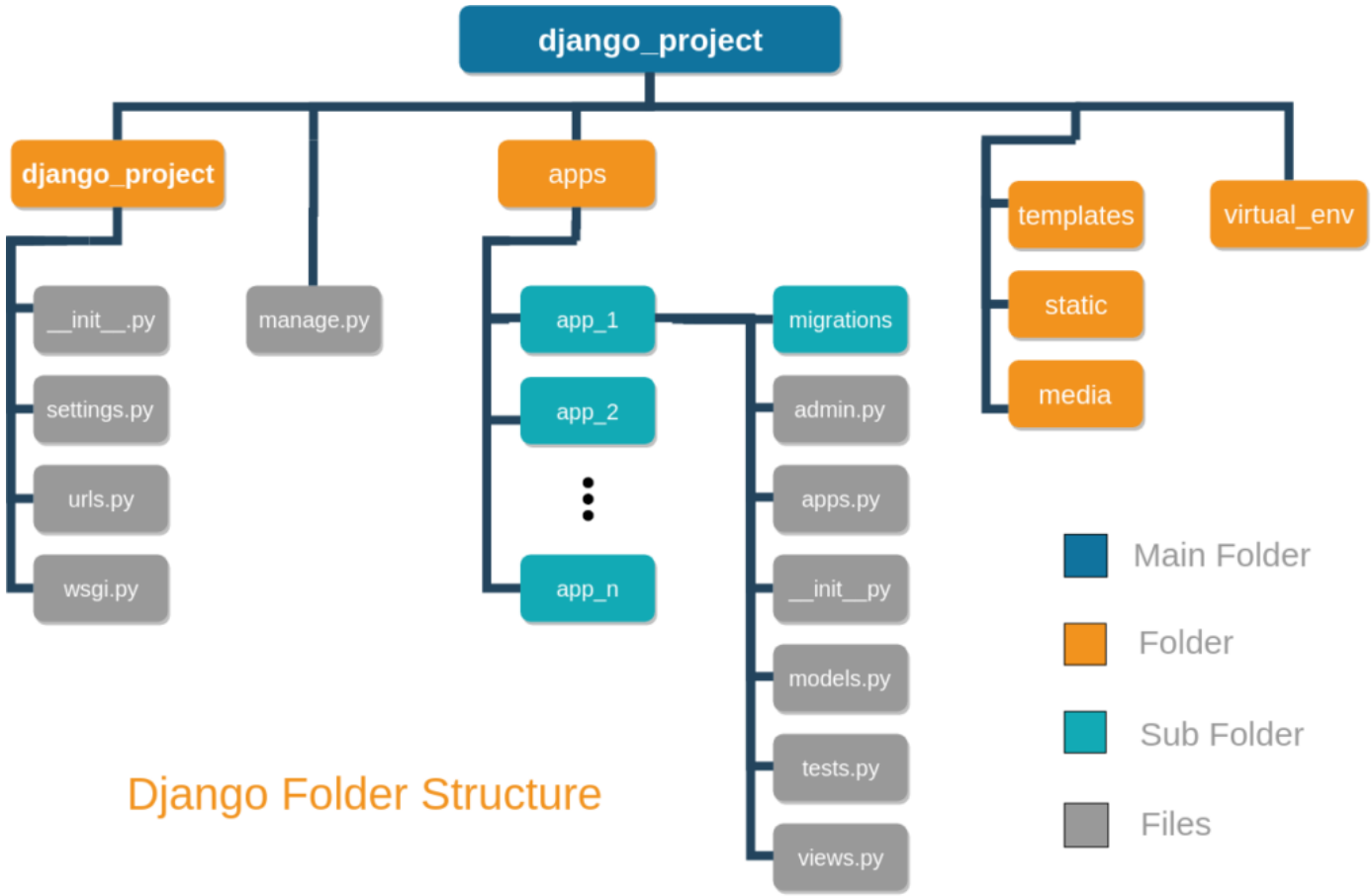
> Les conventions sont plus fortes que la configuration

Architecture d'un projet django

Un projet est organisé de la façon suivante :

- un projet
  - gère les paramètres (`settings.py`)
  - gère le routage de premier niveau (`urls.py`)
  - ...
- un ensemble d'`applications` (spécifique au projet ou génériques)
  - défini des tables SQL (`models.py`)
  - défini un routage spécifique (`urls.py`)
  - défini des vues spécifiques (`views.py`)

Architecture des dossiers en django



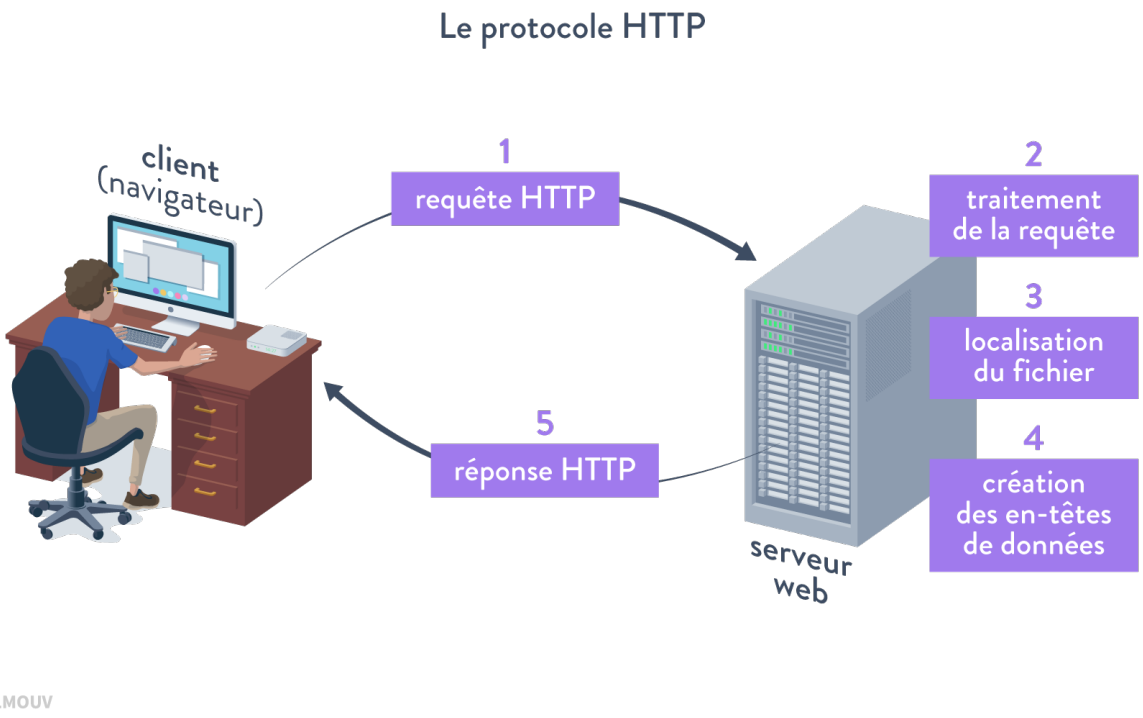
Source : https://studygyaan.com/django/best-practice-to-structure-django-project-directories-and-files

Matthieu Falce

Le protocole HTTP

- HTTP est un protocole d'échange d'informations sur internet
- ce protocole est sans état (stateless) : les requêtes sont indépendantes
- c'est dans les réponses HTTP que peut se trouver le HTML
- il s'agit d'un protocole textuel (jusqu'à HTTP 2 en tous cas)
- il peut être chiffré pour éviter les modifications et "l'espionnage" par des tiers (le HTTPS)

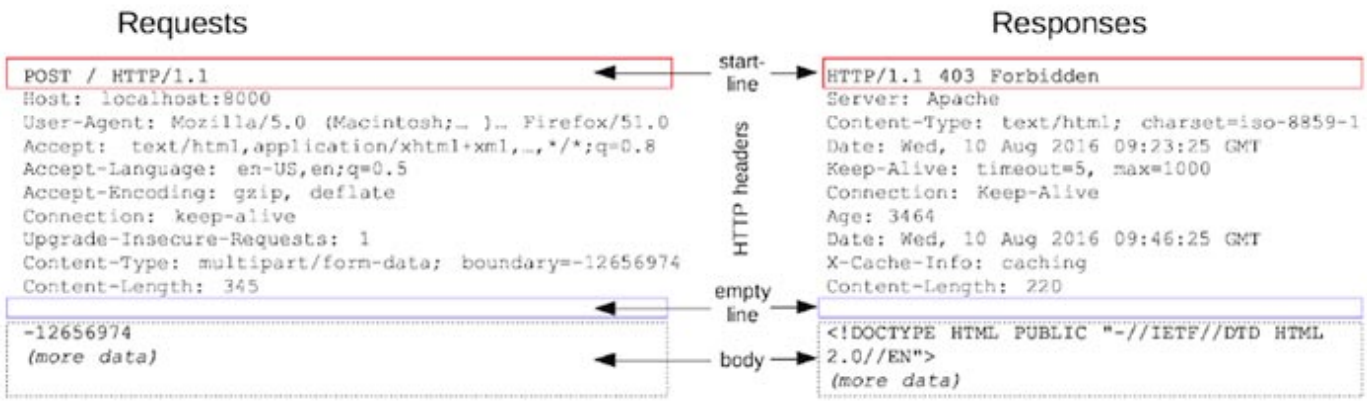
Parcours d'une requete HTTP



Source : https://www.schoolmouv.fr/cours/la-page-web-http-et-langages-html-et-css-/fiche-de-cours

Matthieu Falce

Détail d'une requête / réponse



Source : https://www.pierre-giraud.com/http-reseau-securite-cours/requete-reponse-session/

Matthieu Falce

Les verbes HTTP

HTTP Verb	Path	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

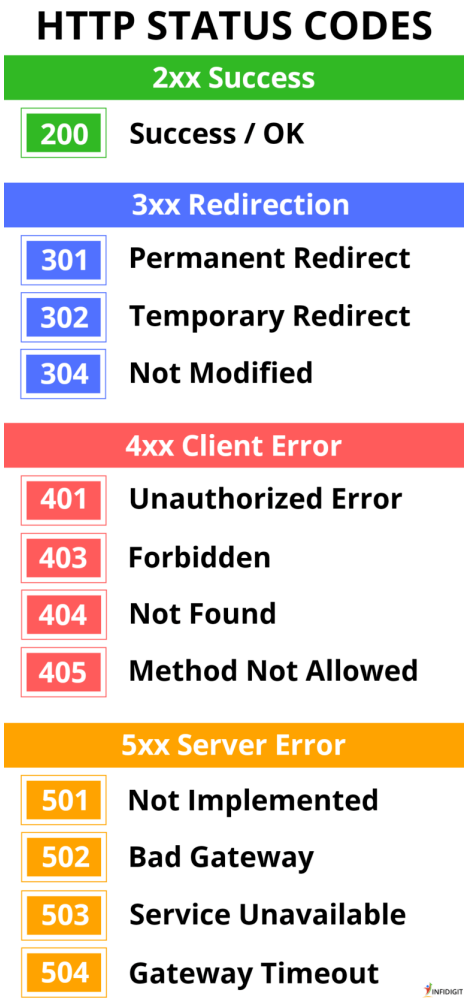
Source : https://medium.com/@ellehallal/week-9-%C2%BD-http-verbs-query-parameters-to-do-list-e56f469ff354

Le verbe GET est idempotent, il ne doit pas modifier les données sur le serveur.

Status code HTTP

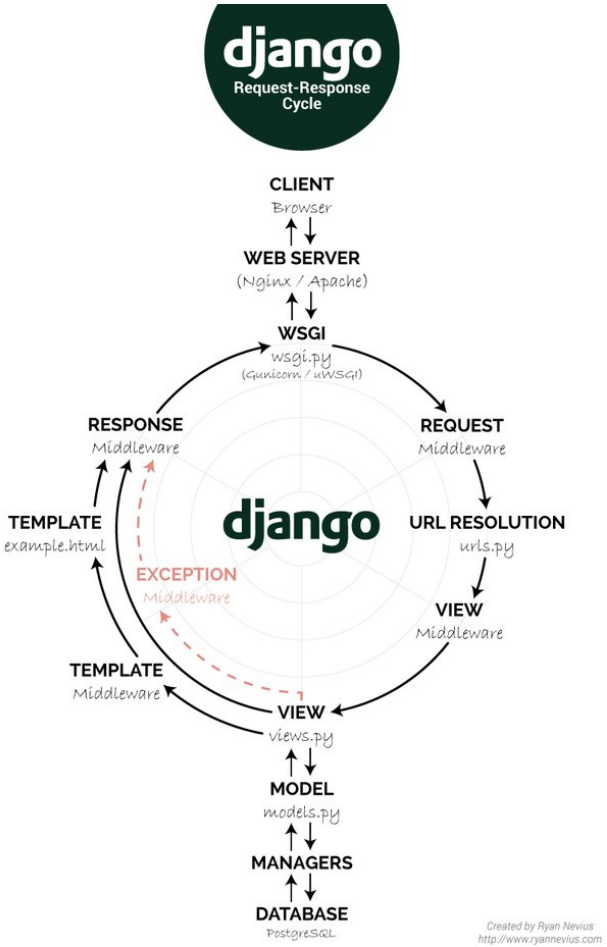
- les requêtes HTTP peuvent avoir différents retours (bien passé, erreur, ...)
- pour en informer le client, il existe des "status code" permettant de normaliser et d'identifier les différents cas
- vous en connaissez forcément (404, 500, 200, ...)
- ils sont classés en différentes catégories

Status code HTTP



Source : https://www.infidigit.com/blog/http-status-codes/

Et dans django



Source : https://www.quora.com/How-can-you-explain-the-basic-request-and-response-flow-in-Django-in-simple-language

2- Django

2.2. Stockage des données – SQL

SQL

- SQL (pour `Structured Query Language`) est le langage de requêtage de données utilisé pour manipuler des bases de données relationnelles
- il permet de faire les opérations de CRUD (`Create / Read / Update / Delete`)
- en django, nous ne manipulons pas directement de SQL mais les concepts classiques sont importants pour ne pas faire n'importe quoi (on parle de l'algèbre relationnelle)

Algèbre relationnelle

- l'algèbre relationnelle est un langage de requête de bases de données relationnelles
- dans ces bases, les informations sont stockées sous forme de tableaux à deux dimensions appelés `table` ou `relation` (penser à un tableau excel)
- les notions importantes sont les `tables` qui vont contenir
  - des colonnes : qui correspondent à une caractéristique typée que l'on veut sauvegarder
  - les lignes : qui correspondent à des individus, l'ordre des lignes n'est pas spécialement important

Algèbre relationnelle

Customer_Details

Customer_id	Name	Address	Age
1	Billie	NY	22
2	Eilish	London	19
3	Ariana	Miami	18
4	Selena	New Jersey	32
5	Kety	Hawaii	42
6	Adele	Miami	29

Source : https://afteracademy.com/blog/what-is-view-in-sql

Lien entre les tables

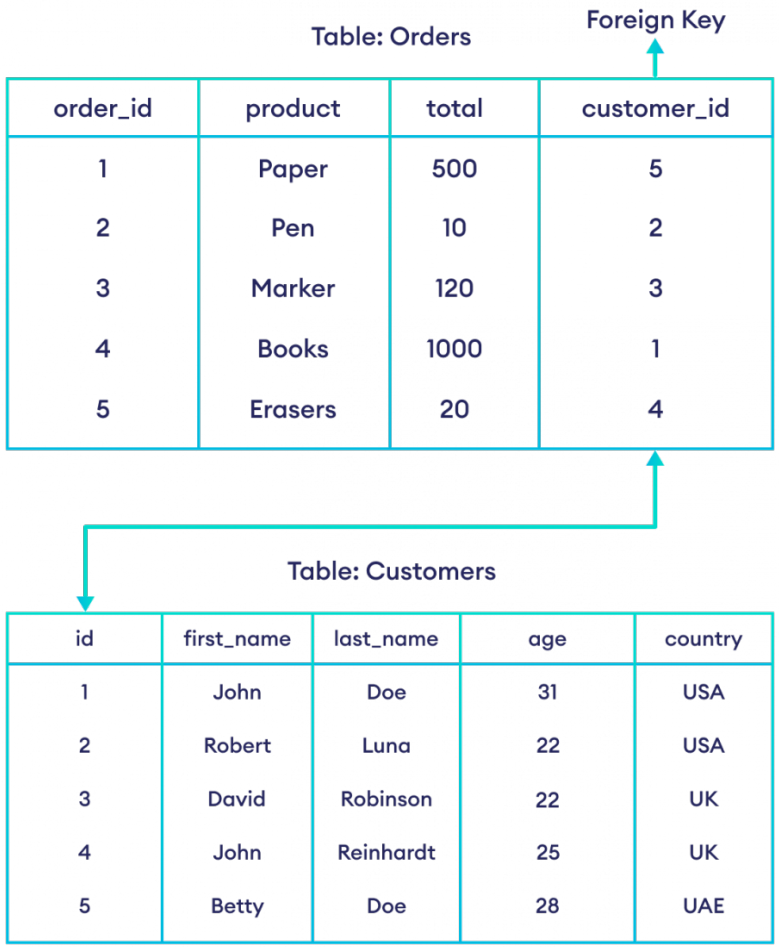
- dans certains cas, nous voulons regrouper des données stockées sur différentes tables
- pour cela nous utilisons des colonnes qui permettent de les relier pour faire une `jointure`
- ces colonnes sont appelées des `clés étrangères` (ou `foreign keys`) et permettent de relier les tables

Lien entre les tables

- dans certains cas, nous voulons regrouper des données stockées sur différentes tables
- pour cela nous utilisons des colonnes qui permettent de les relier pour faire une `jointure`
- ces colonnes sont appelées des `clés étrangères` (ou `foreign keys`) et permettent de relier les tables

Savoir modéliser les données en tables est fondamental en développement web. Il existe des critères : les `formes normales` qui permettent de donner un cadre théorique à cela. Cependant rien ne remplace la pratique

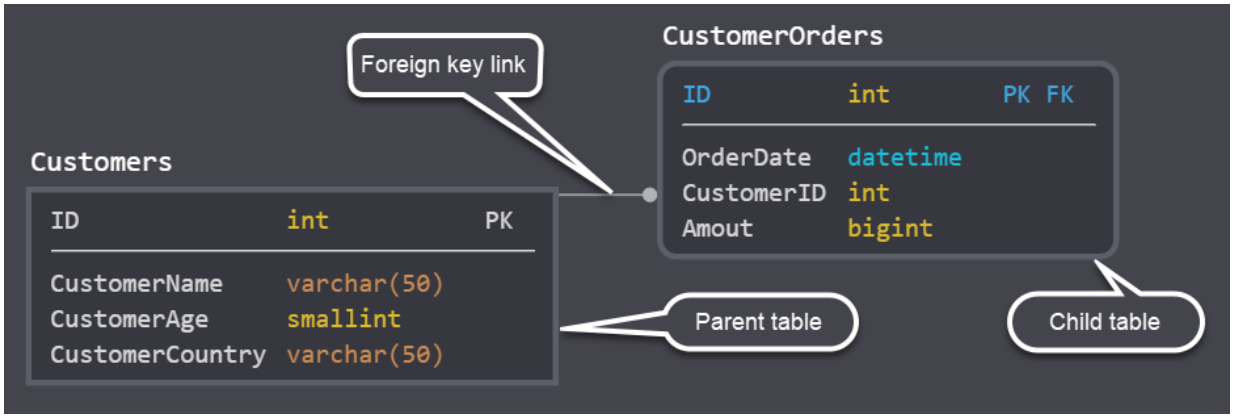
Clés étrangères



Source : https://www.programiz.com/sql/foreign-key

Matthieu Falce

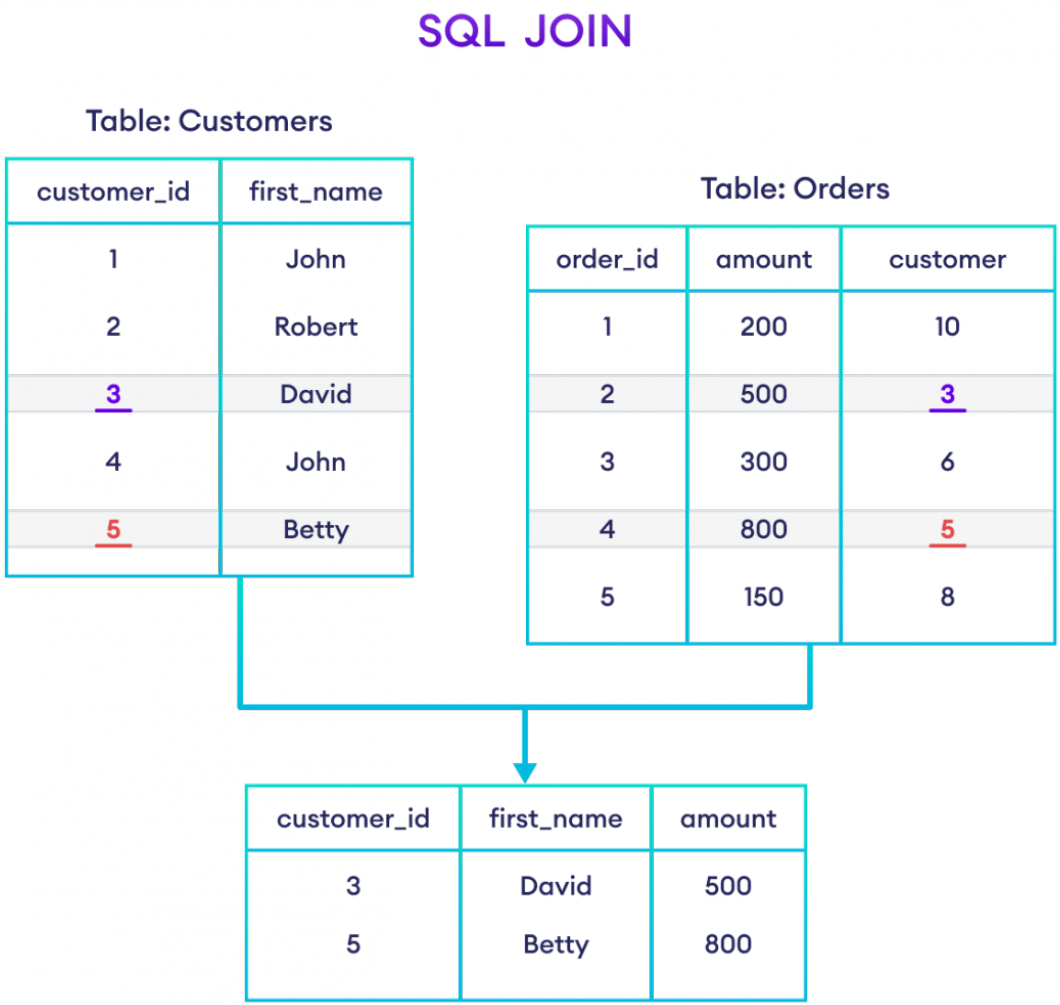
Clés étrangères



Source : https://www.sqlshack.com/what-is-a-foreign-key-in-sql-server/

Matthieu Falce

Jointure



Source : https://www.programiz.com/sql/join

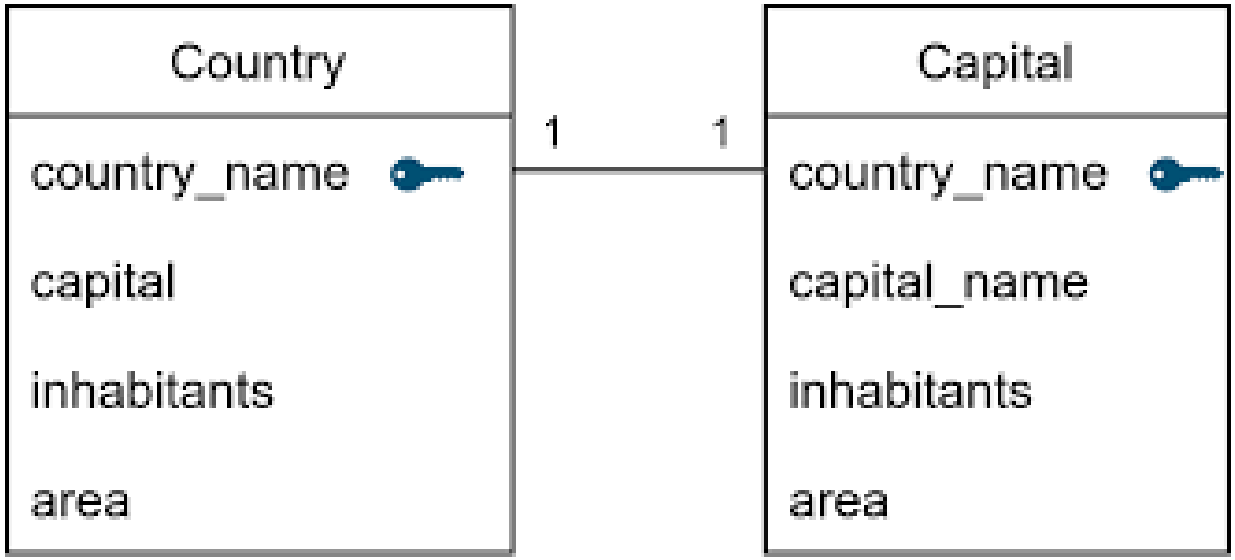
Matthieu Falce

Types de relations

Il existe 3 grands types de relations possibles :

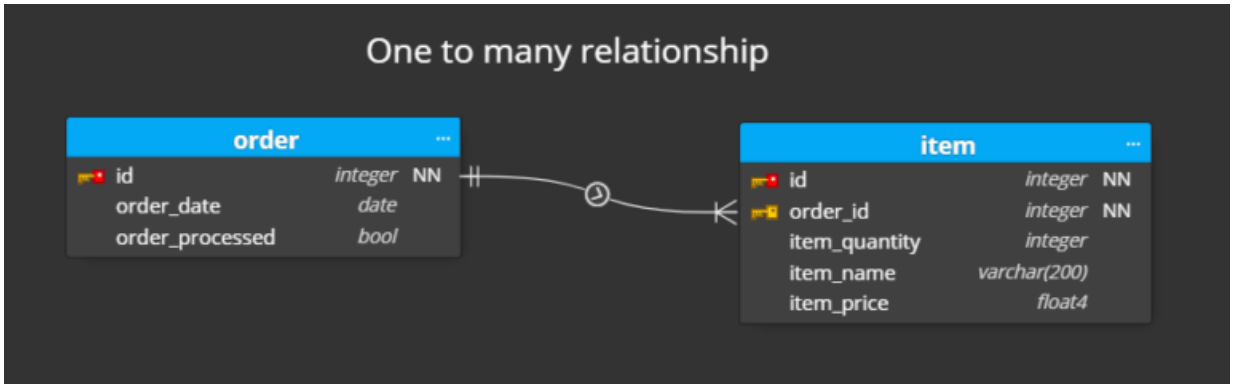
- 1 to 1 : 1 élément d'une table correspond à 1 élément d'une autre (par exemple entre un pays et sa capitale)
- 1 to Many : 1 élément d'une table correspond à N d'une autre table (par exemple 1 utilisateur qui achète N produits)
- Many to Many : N éléments peuvent correspondent à N d'une autre table (par exemple des élèves peuvent être inscrits à plusieurs cours)

1 to 1



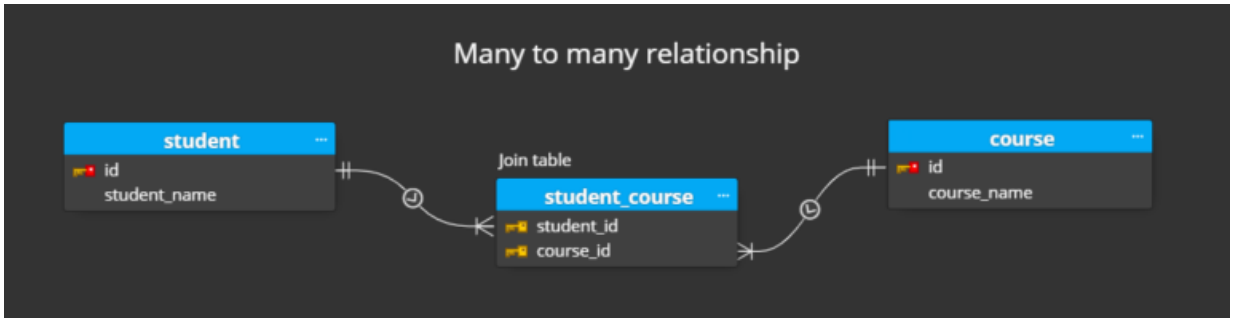
Source :
https://stackoverflow.com/questions/10292355/how-to-create-a-real-one-to-one-relationship-in-sql-server

1 to Many



Source : https://www.datensen.com/blog/er-diagram/one-to-many-relationships/

Many to Many



Source : https://www.datensen.com/blog/er-diagram/many-to-many-relationships/

2- Django

2.3. ORM

Le role d'un ORM

- ORM : `Objet Relation Mapper`
- permet d'abstraire le SQL (on ne l'écrit pas directement)
- on manipule des objets python à l'aide d'une API définie
- permet de faciliter le changement de type de bases de données

Connexion à une base de données

Django permet de se connecter aux SGBD (système de gestion de base de données) suivant :

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite

Connexion à une base de données

Cela se configure dans les settings du projet (connexion à un postgres) :

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "mydatabase",
        "USER": "mydatabaseuser",
        "PASSWORD": "mypassword",
        "HOST": "127.0.0.1",
        "PORT": "5432",
    }
}
```

Connexion à une base de données

Cela se configure dans les settings du projet (connexion à un sqlite) :

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": "le_nom_du_fichier_stockant_la_base",
    }
}
```

Les modèles django

Le fichier models.py est un fichier fondamental en django. Il va stocker les modèles d'une application. C'est là que va se trouver le "schéma" (la structure des tables).

- il s'agit de classes python standard (peuvent avoir des méthodes)
- elles héritent de `models.Models` (venant de django)
- les champs de la base de données sont stockés comme variables de classes (`title = models.CharField(max_length=200)`)
- les champs sont typés (la liste complète ici : https://docs.djangoproject.com/en/4.1/ref/models/fields/)
- option `blank=True` (valide pour django si reçu vide)
- option `null=True` (stocké comme vide en base)
- les champs peuvent avoir des valeurs par défaut et / ou des contraintes

Les modèles django

Plus d'infos sur la différence entre `blank` et `null`:

- https://docs.djangoproject.com/fr/4.1/ref/models/fields/#null
- https://docs.djangoproject.com/fr/4.1/ref/models/fields/#blank
- https://stackoverflow.com/questions/8609192/what-is-the-difference-between-null-true-and-blank-true-in-django

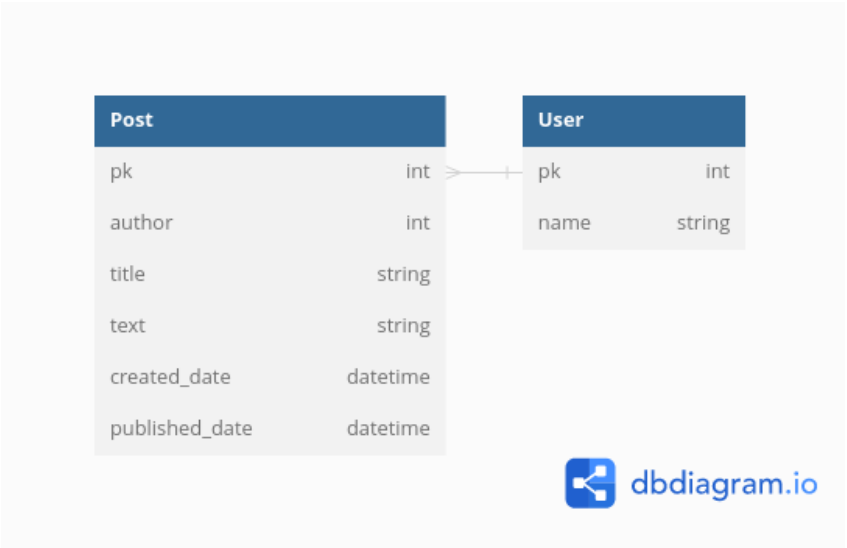
Les modèles django

```python
# Source : https://tutorial.djangogirls.org/en/django_models/
from django.conf import settings
from django.db import models
from django.utils import timezone


class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(default=timezone.now)
    published_date = models.DateTimeField(blank=True, null=True)

    def publish(self):
        self.published_date = timezone.now()
        self.save()

    def __str__(self):
        return self.title
```

Matthieu Falce

Les modèles django

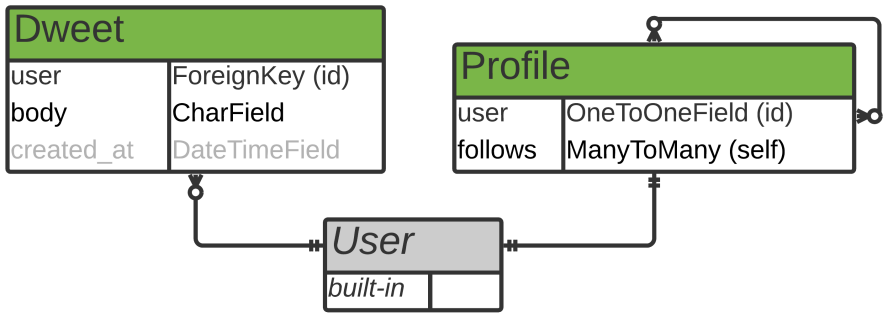
```python
# Source : https://tutorial.djangogirls.org/en/django_models/
from django.conf import settings
from django.db import models
from django.utils import timezone


class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(default=timezone.now)
    published_date = models.DateTimeField(blank=True, null=True)

    def publish(self):
        self.published_date = timezone.now()
        self.save()

    def __str__(self):
        return self.title
```



Matthieu Falce

Association entre les tables

Les différents types d'associations SQL (1 to 1, 1 to Many, Many to Many) peuvent s'exprimer en django.



Source : https://realpython.com/django-social-network-1/

Association entre les tables

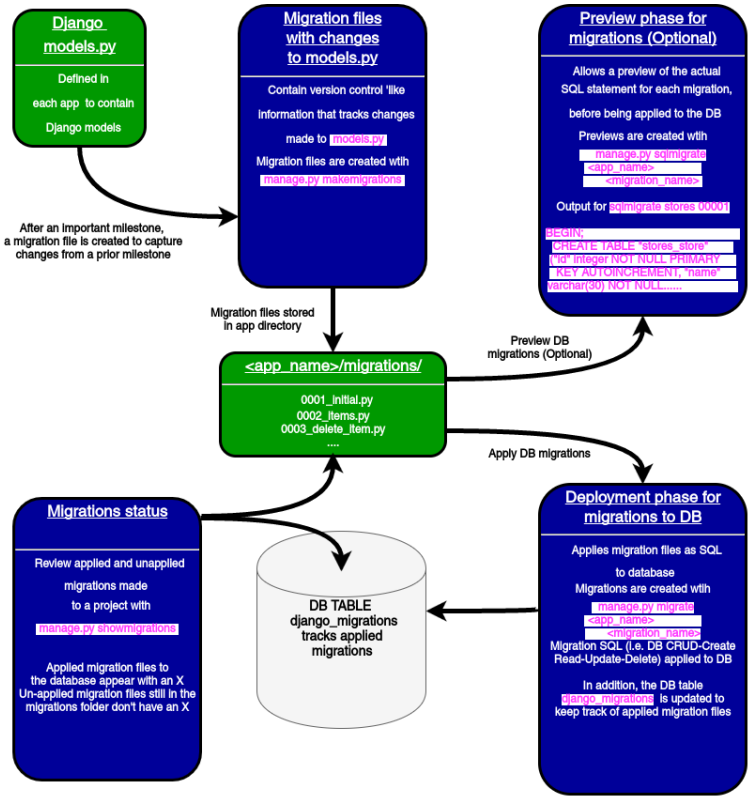
```python
from django.db import models
from django.contrib.auth.models import User
from django.utils import timezone


class Dweet(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    body = models.CharField()
    created_at = models.DateTimeField(default=timezone.now)


class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    follows = models.ManyToManyField(
        "self", related_name="followed_by", symmetrical=False, blank=True
    )

    def __str__(self):
        return self.user.username
```

Matthieu Falce

Migrations

- un schéma de base de données est beaucoup plus strict qu'un modèle objet python
- changer le modèle ne va pas mettre à jour le schéma automatiquement
- pour cela il faut utiliser les `migrations` (ce mécanisme est inclus dans django)
- cela se passe en 2 étapes :
  - `makemigrations` : liste les évolutions entre le schéma actuel et le modèle (va générer du code python permettant de mettre à jour le schéma)
  - `migrate` : met à jour le schéma à proprement parler

Migrations



Source : https://www.webforefront.com/django/setupdjangomodels.html

Matthieu Falce

Queryset et objets

- les `querysets` sont les mécanismes permettant de faire des requêtes à la base de données, ils vont permettre de construire la requête et stocker son résultat
- les objets sont les éléments individuels retournés (de vrais objets python)
- le `queryset` est un objet *paresseux* (les opérations sont effectuées au dernier moment)
- bonne pratique : faire le plus d'opérations possibles dans le `queryset` plutôt qu'en python

Opérations CRUD usuelles

```python
# https://docs.djangoproject.com/fr/4.1/topics/db/queries/

from datetime import date
from django.db import models


class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()


class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()


class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    mod_date = models.DateField(default=date.today)
    authors = models.ManyToManyField(Author)
    number_of_pingbacks = models.IntegerField(default=0)
    rating = models.IntegerField(default=5)
```

Matthieu Falce

Opérations CRUD usuelles

```python
# création d'un objet en base
from blog.models import Blog, Entry, Author

b = Blog(name="Beatles Blog", tagline="All the latest Beatles news.")
b.save()

# mise à jour (Update)
b.name = "New name"
b.save()

# gestion des clés étrangères
entry = Entry.objects.get(pk=1)
cheese_blog = Blog.objects.get(name="Cheddar Talk")
entry.blog = cheese_blog
entry.save()

# gestion des associations N à N
joe = Author.objects.create(name="Joe")
entry.authors.add(joe)

john = Author.objects.create(name="John")
paul = Author.objects.create(name="Paul")
george = Author.objects.create(name="George")
ringo = Author.objects.create(name="Ringo")
entry.authors.add(john, paul, george, ringo)
```

Matthieu Falce

Opérations CRUD usuelles

```python
import datetime
from blog.models import Blog, Entry, Author

# tous les objets
all_entries = Entry.objects.all()

# filtrage
Entry.objects.filter(pub_date__year=2006)

# chainage de filtres
Entry.objects.filter(headline__startswith="What").exclude(
    pub_date__gte=datetime.date.today()
).filter(pub_date__gte=datetime.date(2005, 1, 30))

# récupération d'un élément unique
one_entry = Entry.objects.get(pk=1)

# fenetrage
Entry.objects.all()[5:10]

# trie des données
Entry.objects.order_by("headline")
```

Matthieu Falce

Opérations CRUD usuelles

```python
from blog.models import Blog, Entry, Author

# recherche dans un champ
# plus d'infos : https://docs.djangoproject.com/fr/4.1/ref/models/querysets/#field-lookups
Entry.objects.filter(pub_date__lte="2006-01-01")
Entry.objects.get(headline__exact="Cat bites dog")

# recherche à travers une relation
Entry.objects.filter(blog__name="Beatles Blog")
Blog.objects.filter(entry__headline__contains="Lennon")
```

Matthieu Falce

Opérations CRUD usuelles

```python
from blog.models import Blog, Entry, Author

# suppression d'un objet
entry = Entry.objects.get(pk=1)
entry.delete()

# suppression d'un queryset
Entry.objects.filter(headline__startswith="BREAKING NEWS").delete()
```

2- Django

2.4. Vues

Routage d'URL

- le but de django est d'appeler la bonne fonction de vue après un appel à une URL
- ce mécanisme s'appelle le *routage d'URL* ou la *distribution d'URL*
- cela s'effectue dans plusieurs fichiers
  - urls.py du projet (qui peut définir des préfixes pour les différentes apps)
  - urls.py des apps
- plus d'informations ici : https://docs.djangoproject.com/fr/4.1/topics/http/urls/

Routage d'URL

```python
from django.urls import path

from . import views

urlpatterns = [
    path(
        "quelle-heure/", views.quelle_heure,
        name="quelle-heure"
    ),
    path(
        "quelle-date", views.quelle_date,
        name="quelle-date"
    ),
]
```

Fonction de vue

- les fonctions de vues sont parmi les plus importantes en django
- ce sont elles qui retournent des informations à l'utilisateur du site
- elles vont être appelées automatiquement suite au routage de l'URL demandée
- elles prennent au moins un paramètre : la requête HTTP (permettant d'avoir accès au verbe, à l'utilisateur logué, ...)
- une vue doit retourner une HttpResponse dans quasiment tous les cas
- si une fonction de vue "plante" (lève une exception non gérée), l'utilisateur verra une erreur 500

Fonction de vue

```
from django.http import HttpResponse
import datetime


def quelle_heure(request):
    html = f"""<html><body>
    It is now {datetime.datetime.now().time()}.
    </body></html>"""
    return HttpResponse(html)


def quelle_date(request):
    html = f"""<html><body>
    It is now {datetime.datetime.now()}.
    </body></html>"""
    return HttpResponse(html)
```

Ce que retourne une vue I

- une vue retourne une HttpResponse
- pour lever un autre code HTTP, on peut le préciser dans `HttpResponse(status=404)`
- dans le cadre des erreurs HTTP on peut lever des exceptions :
  - `raise Http404("Object doesn't exist")`
  - `raise PermissionDenied`
  - ...
- dans le cadre d'opérations courantes, la vue peut retourner autre chose (`shortcuts`)
  - `render()` pour faciliter la gestion des templates
  - `redirect()`
  - `get_object_or_404` / `get_list_or_404`
  - ...

Ce que retourne une vue II

Plus d'informations :
https://docs.djangoproject.com/fr/4.1/topics/http/views/
et
https://docs.djangoproject.com/en/4.1/topics/http/shortcuts/

Passage de paramètres

On peut passer des paramètres aux fonctions de vue depuis le routage (pour faire des vues plus génériques)

```python
# urls.py

from django.urls import path

from . import views

urlpatterns = [
    path("articles/2003/", views.special_case_2003),
    path("articles/<int:year>/", views.year_archive),
    path("articles/<int:year>/<int:month>/", views.month_archive),
    path("articles/<int:year>/<int:month>/<slug:slug>/", views.article_detail),
]
```

Passage de paramètres

```python
# views.py
from django.http import JsonResponse
from .models import Articles


def special_case_2003(request):
    data = Articles.objets.filter(publication_year="03")
    return JsonResponse({"data": data})


def year_archive(request, year: int):
    data = Articles.objets.filter(year=year)
    return JsonResponse({"data": data})


def month_archive(request, year: int, month: int):
    data = Articles.objets.filter(year=year, month=month)
    return JsonResponse({"data": data})


def article_detail(request, year: int, month: int, slug: str):
    data = Articles.objets.get(slug=slug, year=year, month=month)
    return JsonResponse({"article": data})
```

Gestion des différents verbes HTTP

Il est possible de gérer les différents verbes HTTP :

```
def vue_func(request):
    if request.method == "GET":
        do_something()
    elif request.method == "POST":
        do_something_else()
```

On peut utiliser le décorateur `require_http_methods` pour indiquer quels seront les verbes autorisés dans la vue.

Gestion des différents verbes HTTP

Il est possible de récupérer les paramètres GET des requêtes :

- requête de la forme : `domain/search/?q=haha`
- accès de la forme : `request.GET.get('q', '')`

Même chose pour les paramètres `POST`

Techniques d'écritures I

Il existe différentes méthodes pour écrire des vues en django.

- à base de fonctions, comme on l'a vu jusqu'à présent (on parle de `Function Based Views` ou `FBW`)
- à base de classe. On parle de vues génériques ou vues basées sur les classes (`CBW` pour `Class Based Views`)
  - `DetailView` pour accéder à une vue d'un élément
  - `ListView` pour assister à un listing d'éléments
  - savoir quelle technique utiliser est propre aux goûts de chacun
  - plus d'informations : https://docs.djangoproject.com/en/4.1/ref/class-based-views/
  - critique des CBW : `https://lukeplant.me.uk/blog/posts/djangos-cbvs-were-a-mistake/`
  - défense des CBW : `http://www.curiousefficiency.org/posts/2012/05/djangos-cbvs-are-not-mistake-but.html`

Classe Based Views

```python
# from : https://docs.djangoproject.com/en/4.1/ref/class-based-views/generic-display/
# views.py
from django.utils import timezone
from django.views.generic.detail import DetailView
from articles.models import Article


class ArticleDetailView(DetailView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["now"] = timezone.now()
        return context


# urls.py
from django.urls import path
from article.views import ArticleDetailView

urlpatterns = [path("<slug:slug>/", ArticleDetailView.as_view(), name="article-detail")]


<!-- myapp/article_detail.html -->
<h1>{{ object.headline }}</h1>
<p>{{ object.content }}</p>
<p>Reporter: {{ object.reporter }}</p>
<p>Published: {{ object.pub_date|date }}</p>
<p>Date: {{ now|date }}</p>
```

Comparaison CBV / FBV

- les codes suivants sont tirés de https://lukeplant.me.uk/blog/posts/djangos-cbvs-were-a-mistake/
- il s'agit de cas potentiellement pathologiques
- c'est au développeur de voir ce qui sera le plus pratique et le plus maintenable

Comparaison CBV / FBV

```python
from django.core.urlresolvers import reverse
from django.shortcuts import render


def contact(request):
    high_priority_user = (
        not request.user.is_anonymous() and request.user.get_profile().high_priority
    )
    form_class = HighPriorityContactForm if high_priority_user else ContactForm

    if request.method == "POST":
        form = form_class(request.POST)
        if form.is_valid():
            email, message = form.cleaned_data["email"], form.cleaned_data["message"]
            send_contact_message(email, message)
            if high_priority_user and form.cleaned_data["urgent"]:
                send_text_message(email, message)
            return HttpResponseRedirect(reverse("contact_thanks"))
    else:
        form = form_class(
            initial={"email": request.user.email}
            if not request.user.is_anonymous()
            else {}
        )

    return render(
        request,
        "contact.html",
        {"form": form, "high_priority_user": high_priority_user},
    )
```

Matthieu Falce

Comparaison CBV / FBV

```python
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import ProcessFormView


class ContactView(ProcessFormView):
    template_name = "contact.html"
    success_url = reverse_lazy("contact_thanks")

    def dispatch(self, request, *args, **kwargs):
        self.high_priority_user = (
            not request.user.is_anonymous() and request.user.get_profile().high_priority
        )
        return super(ContactView, self).dispatch(request, *args, **kwargs)

    def get_form_class(self):
        return HighPriorityContactForm if self.high_priority_user else ContactForm

    def get_initial(self):
        initial = super(ContactView, self).get_initial()
        if not request.user.is_anonymous():
            initial["email"] = request.user.email
        return initial

    def form_valid(self, form):
        email, message = form.cleaned_data["email"], form.cleaned_data["message"]
        send_contact_message(email, message)
        if self.high_priority_user and form.cleaned_data["urgent"]:
            send_text_message(email, message)
        return super(ContactView, self).form_valid(form)

    def get_context_data(self, **kwargs):
        context = super(ContactView, self).get_context_data(**kwargs)
        context["high_priority_user"] = self.high_priority_user
        return context
```

Matthieu Falce

2- Django

2.5. Templates

Présentation et intérêt

- les templates ou gabarits permettent de créer des chaines de caractères à partir de variables
- imaginez les comme des fstrings en plus puissantes (boucles, conditions, inclusions)
- vous allez en utiliser la majorité du temps pour générer vos pages HTML
- ils peuvent servir à bien plus (template de mails, ...)
- ils permettent de séparer la logique / programmation de la présentation (en théorie quelqu'un qui ne connaît rien à django pourrait mettre à jour un design dans les templates)
- par défaut django est fourni avec son moteur de gabarits, mais vous pouvez en utiliser d'autres

Création et configuration

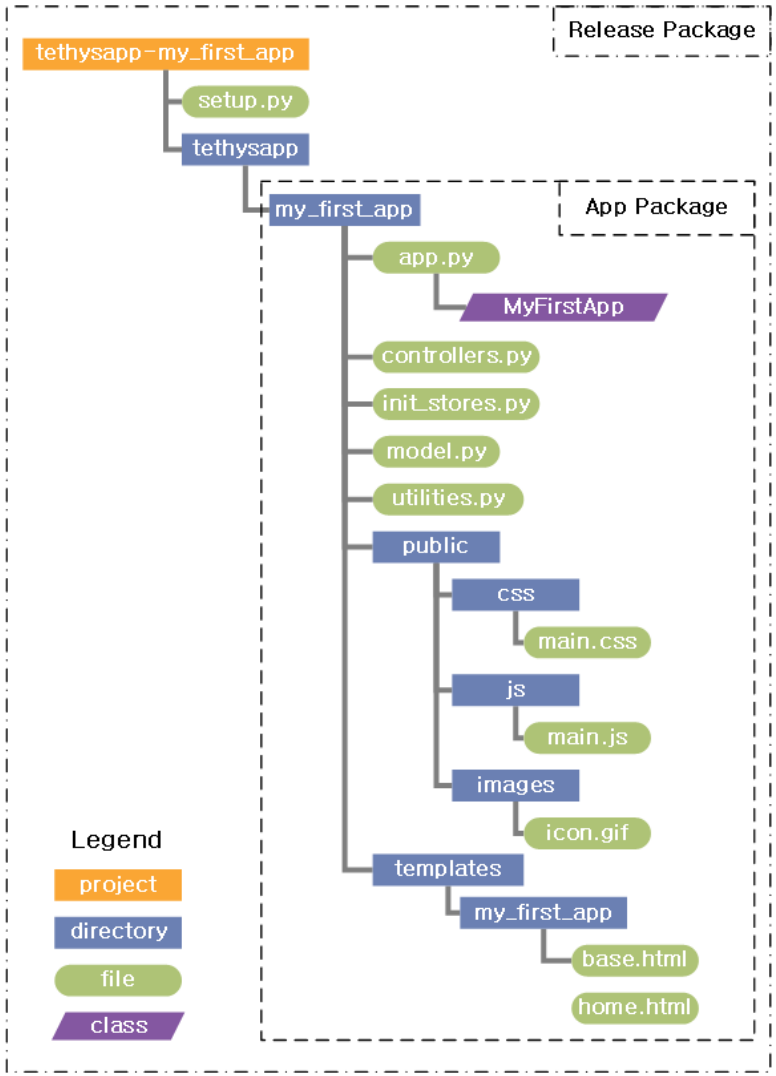
La configuration des templates s'effectue dans `settings.py`. Ils sont activés et configurés par défaut :

```
# source : https://docs.djangoproject.com/fr/4.1/topics/templates/#the-django-template-language
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [],
        "APP_DIRS": True,
        "OPTIONS": {
            # ... some options here ...
        },
    }
]
```

Création et configuration

- l'organisation des templates peut être un peu complexe (surtout quand on commence un nouveau projet)
- les templates sont auto-détectés par le moteur de rendu
- il faut cependant respecter une arborescence de dossiers précise
- il est préférable d'organiser les templates en sous-répertoires
- la convention est de créer un sous-répertoire par application django

Création et configuration



Source : http://docs.tethysplatform.org/en/stable/supplementary/app_project.html

Matthieu Falce

Langage de template

La syntaxe [1]:

- les variables sont entourées par `{{` et `}}` : "je suis {{prenom}}" → "Je suis Matthieu"
- les balises permettent d'appliquer une logique (produire du contenu, insérer des structures de contrôle, ...) : de la forme `{% ... %}`
  - `{% cycle 'odd' 'even' %}`
  - `{% if user.is_authenticated %}Hello, {{ user.username }}.{% endif %}`
- les filtres permettent de transformer les valeurs des variables : de la forme texttt{{ var | filtre }}

1.urlhttps://docs.djangoproject.com/fr/4.1/topics/templates/#the-django-template-language

Langage de template

Il est possible de :

- faire des conditions (if)
- faire des boucles (for)
- inclure d'autres templates / faire des héritages entre templates pour factoriser les parties communes
- créer vos propres balises et filtres pour faciliter vos cas précis

Langage de template

Je vous déconseille de mettre beaucoup de logique dans vos templates ou dans vos vues, c'est la porte ouverte à de nombreux problèmes de maintenance.

Utilisation

Source : https://docs.djangoproject.com/en/4.1/intro/tutorial03/

```python
# polls/urls.py

from django.urls import path

from . import views

app_name = "polls"
urlpatterns = [
    path("", views.index, name="index"),
    path("<int:question_id>/", views.detail, name="detail"),
    path("<int:question_id>/results/", views.results, name="results"),
    path("<int:question_id>/vote/", views.vote, name="vote"),
]
```

Utilisation

Source : https://docs.djangoproject.com/en/4.1/intro/tutorial03/

```python
# polls/views.py

from django.http import HttpResponse
from django.template import loader

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by("-pub_date")[:5]
    template = loader.get_template("polls/index.html")
    context = {"latest_question_list": latest_question_list}
    return HttpResponse(template.render(context, request))
```

Utilisation

Source : https://docs.djangoproject.com/en/4.1/intro/tutorial03/

```
{# polls/templates/polls/index.html #}

{% if latest_question_list %}
<ul>
  {% for question in latest_question_list %}
  <li>
    <a href="{% url 'polls:detail' question.id %}"
      >{{ question.question_text }}</a
    >
  </li>
  {% endfor %}
</ul>
{% else %}
<p>No polls are available.</p>
{% endif %}
```

Utilisation

Source : https://docs.djangoproject.com/en/4.1/intro/tutorial03/

```python
# polls/views.py

from django.shortcuts import render

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by("-pub_date")[:5]
    context = {"latest_question_list": latest_question_list}
    return render(request, "polls/index.html", context)
```

2- Django

2.6. Formulaires

Concept

- les formulaires HTTP permettent à l'utilisateur d'envoyer des informations à votre serveur
- vous en avez déjà manipulé en tant qu'utilisateur (formulaire de contact, champ de recherche, ...)
- il s'agit d'une des choses les plus compliquées à gérer correctement dans le développement web
  - présentation des informations déjà enregistrées
  - validation potentiellement complexe (champs liés)
  - gestion des erreurs
  - connexion avec des modèles en base de donnée
  - récupération des informations dans le formulaire
  - ...
- django permet de gérer une partie de cette difficulté

Utilisation des verbes HTTP

Les formulaires peuvent utiliser les 2 verbes HTTP suivant :

- GET :
  - pour les requêtes qui ne vont pas modifier l'état du système (on dit idempotente)
  - les requêtes contenant des informations confidentielles ne doivent pas être GET car les informations peuvent se trouver dans les logs
  - la requête ressemble à ça : `https://docs.djangoproject.com/search/?q=forms&release=1`
- POST :
  - les données sont contenues dans le corps du message HTTP
  - les formulaires POST peuvent être protégés contre des attaques classiques (CSRF, ...)
  - à utiliser dans la majorité des cas en pratique

Ecriture en HTML

```
{# https://docs.djangoproject.com/fr/4.1/topics/forms/#django-s-role-in-forms #}
<form action="/your-name/" method="post">
  <label for="your_name">Your name: </label>
  <input
    id="your_name"
    type="text"
    name="your_name"
    value="{{ current_name }}"
  />
  <input type="submit" value="OK" />
</form>
```

- url où envoyer les données : `/your-name/`
- méthode : POST
- données : `your_name`
- peut être pré-rempli avec la variable `current_name`

Rôle de django dans la gestion des formulaires

- simplifier et automatiser la gestion des formulaires
- permet de le faire plus facilement
- permet de le faire de manière plus sécurisée

Les 3 grandes étapes du travail :

- préparation et restructuration des données en vue de leur présentation
- création des formulaires HTML pour les données
- réception et traitement des formulaires et des données envoyés par le client

Rôle de django dans la gestion des formulaires

- simplifier et automatiser la gestion des formulaires
- permet de le faire plus facilement
- permet de le faire de manière plus sécurisée

Les 3 grandes étapes du travail :

- préparation et restructuration des données en vue de leur présentation
- création des formulaires HTML pour les données
- réception et traitement des formulaires et des données envoyés par le client

Source : https://docs.djangoproject.com/fr/4.1/topics/forms/#django-s-role-in-forms

Matthieu Falce

Presentation des Forms

- la classe `Form` est centrale à la gestion des formulaires
- un peu équivalent à la gestion des modèles, il va y avoir une liste de champs possibles, vérifiables et pouvant générer des erreurs
- les types indiqués dans les champs permettent de changer le `widget` qui sera utilisé (le composant HTML utilisé pour le rendu pour l'utilisateur)
- nous verrons une classe dérivée : les `ModelForm` qui permettent de générer un formulaire pour un modèle facilement

Presentation des Forms

```python
# form.py

from django import forms


class NameForm(forms.Form):
    your_name = forms.CharField(label="Your name", max_length=100)
```

Et le résultat dans le HTML :

```html
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required />
```

Matthieu Falce

Presentation des Forms

Vue permettant de gérer le informations :

```python
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import NameForm


def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == "POST":
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect("/thanks/")

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, "name.html", {"form": form})
```

Presentation des Forms

Template

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

Rendu dans un template

Le rendu peut être effectué de différentes façons [1] :

- form.as_div
- form.as_table
- form.as_p
- form.as_ul

1. https://docs.djangoproject.com/fr/4.1/topics/forms/#form-rendering-options

Rendu dans un template

Le rendu peut être effectué de différentes façons [1] :

- form.as_div
- form.as_table
- form.as_p
- form.as_ul

On peut également rendre les champs manuellement (explicitement ou en itérant sur les champs). Dans ce cas, attention à bien gérer les erreurs des champs

1. https://docs.djangoproject.com/fr/4.1/topics/forms/#form-rendering-options

Validation et gestion des erreurs

Il est possible de gérer la validation assez finement :

- transformation des données HTTP en python (transformer des chaines en liste par exemple)
- valider des champs uniques
- valider des champs dépendants l'un de l'autre
- envoyer des messages d'erreurs personnalisés
- ...

Validation et gestion des erreurs

```python
# Source : https://docs.djangoproject.com/fr/4.1/ref/
#          forms/validation/#form-field-default-cleaning

from django import forms
from django.core.validators import validate_email
from django.core.exceptions import ValidationError


class MultiEmailField(forms.Field):
    def to_python(self, value):
        """Normalize data to a list of strings."""
        # Return an empty list if no input was given.
        if not value:
            return []
        return value.split(",")

    def validate(self, value):
        """Check if value consists only of valid emails."""
        # Use the parent's handling of required fields, etc.
        super().validate(value)
        for email in value:
            validate_email(email)


class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)
```

Matthieu Falce

Validation et gestion des erreurs

```python
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)

    def clean_recipients(self):
        data = self.cleaned_data["recipients"]
        if "fred@example.com" not in data:
            raise ValidationError("You have forgotten about Fred!")

        # Always return a value to use as the new cleaned data, even if
        # this method didn't change it.
        return data

    def clean(self):
        cleaned_data = super().clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject and "help" not in subject:
            msg = "Must put 'help' in subject when cc'ing yourself."
            self.add_error("cc_myself", msg)
            self.add_error("subject", msg)
```

Matthieu Falce

Presentation des `ModelForm`

Les `ModelForms` permettent :

- de lier des formulaires à des modèles django
- cela facilite les modifications des modèles base de données car les formulaires sont mis à jour automatiquement
- ils évitent en effet de répéter ces informations de champs
- c'est ici que nous allons utiliser `blank=True` et le `verbose_name`
- on peut choisir quels sont les champs à utiliser dans le formulaire ou en rajouter d'autres

Presentation des ModelForm

```python
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = [("MR", "Mr."), ("MRS", "Mrs."), ("MS", "Ms.")]


class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name


class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)


class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(max_length=3, widget=forms.Select(choices=TITLE_CHOICES))
    birth_date = forms.DateField(required=False)


class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

Matthieu Falce

Presentation des ModelForm

```python
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = [("MR", "Mr."), ("MRS", "Mrs."), ("MS", "Ms.")]


class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):
        return self.name


class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)


class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ["name", "title", "birth_date"]


class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ["name", "authors"]
```

Matthieu Falce

2- Django

2.7. Authentification

Activer l'authentification [2]

- l'authentification est incluse par défaut dans django
- elle permet de gérer les comptes, groupes, permissions et session utilisateur (ne pas avoir besoin de se reconnecter à chaque page)
- elle s'occupe de vérifier la sécurité des mots de passes
- pour l'activer il faut (c'est le cas par défaut):
  - rajouter `django.contrib.auth` dans les `INSTALLED_APPS` de `settings.py`
  - rajouter les middlewares `SessionMiddleware` et `AuthenticationMiddleware`
  - faire une migration pour ajouter les tables en base

2.https://docs.djangoproject.com/en/4.1/topics/auth/

Le modèle utilisateur

Par défaut, le modèle utilisateur possède les attributs suivants :

- `username`
- `password`
- `email`
- `first_name`
- `last_name`

Pour un projet qui commence, il est conseillé de customiser l'utilisateur par défaut [3]:

- définir une nouvelle classe d'utilisateur
- l'indiquer en pointant `AUTH_USER_MODEL` dans les `settings.py` vers cette classe

3. https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#using-a-custom-user-model-when-starting-a-project

Gestion des utilisateurs (rôles)

Les permissions peuvent être gérées par 3 booléens :

- `is_staff` : permet l'accès à l'admin django
- `is_admin` : peut accéder à toutes les modifications car possède tous les droits
- `is_active` : si vrai alors l'utilisateur peut se logger
- si ces 2 champs `staff` et `admin` sont à faux mais que `active` est vrai, alors l'utilisateur est considéré comme normal

2- Django

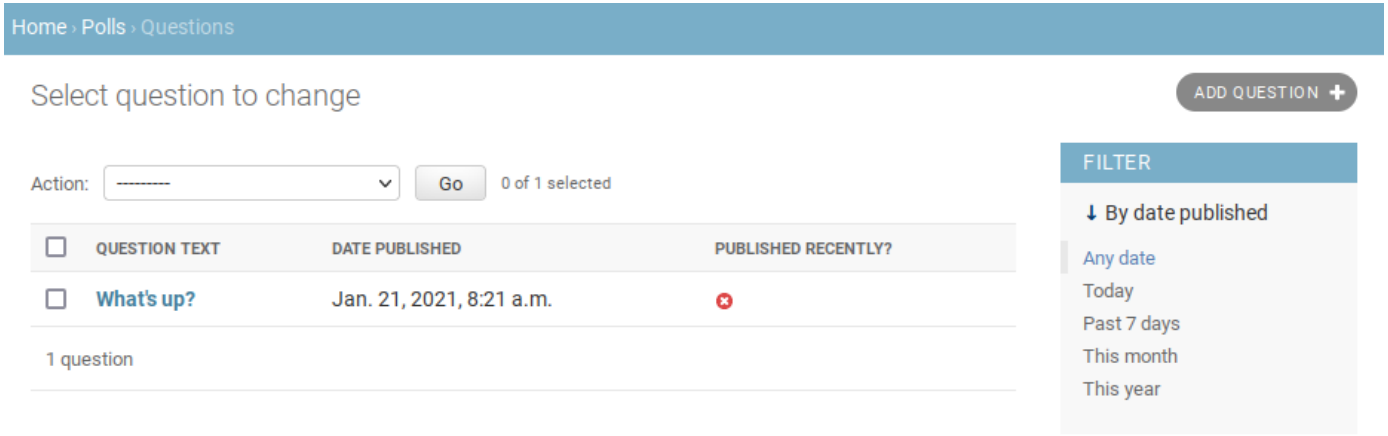
2.8. Admin

Intérêt de l'admin django [4]

- c'et l'une des parties les plus puissantes de django
- elle va lire les informations des modèles et en faire une interface pour éditer et créer rapidement des enregistrements
- possibilité de gérer les relations 1-N et N-N
- certains l'utilisent entièrement pour gérer leur backoffice avant d'en développer un
- vous pouvez en effet le personnaliser assez fortement (affichage, actions possibles, ...)

4. https://docs.djangoproject.com/fr/4.1/ref/contrib/admin/

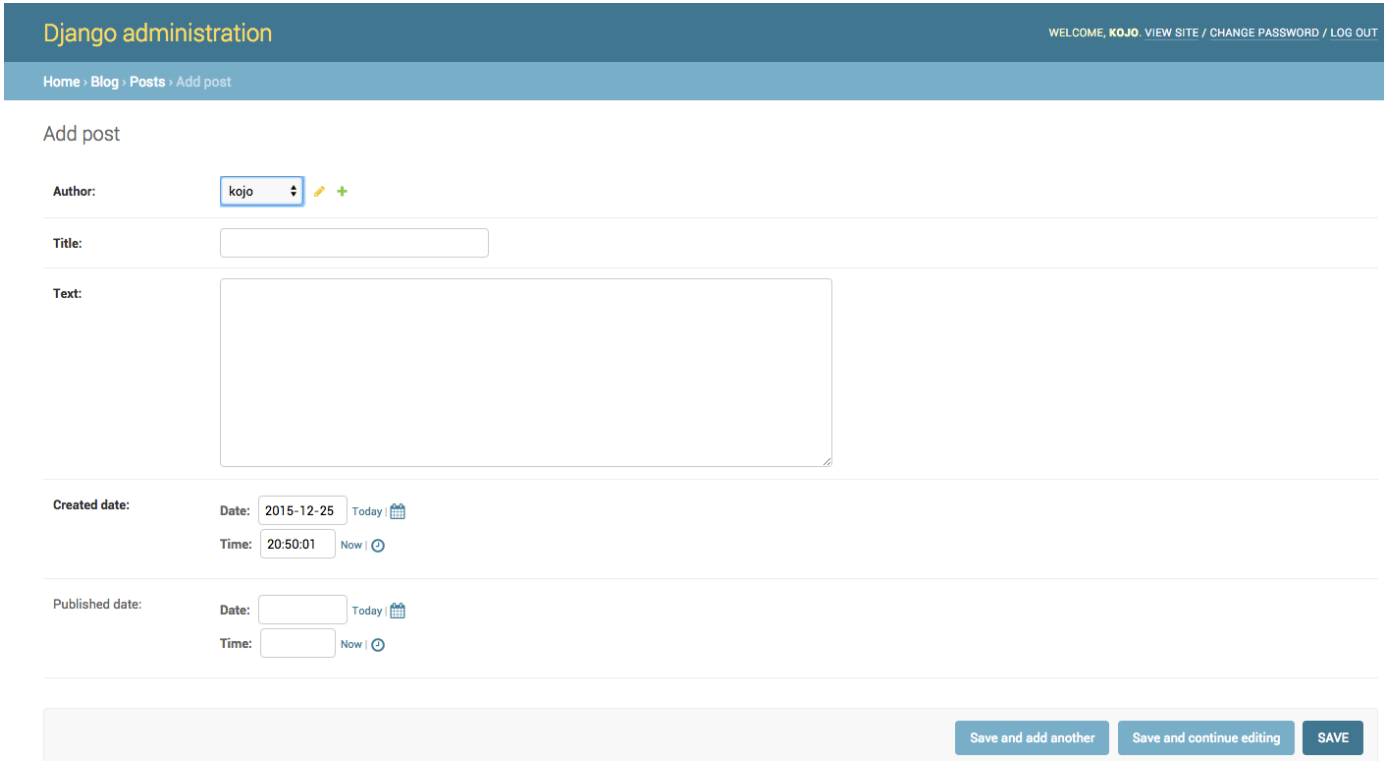
Intérêt de l'admin django [4]



Source :
https://docs.djangoproject.com/fr/4.1/intro/tutorial07/

4.https://docs.djangoproject.com/fr/4.1/ref/contrib/admin/

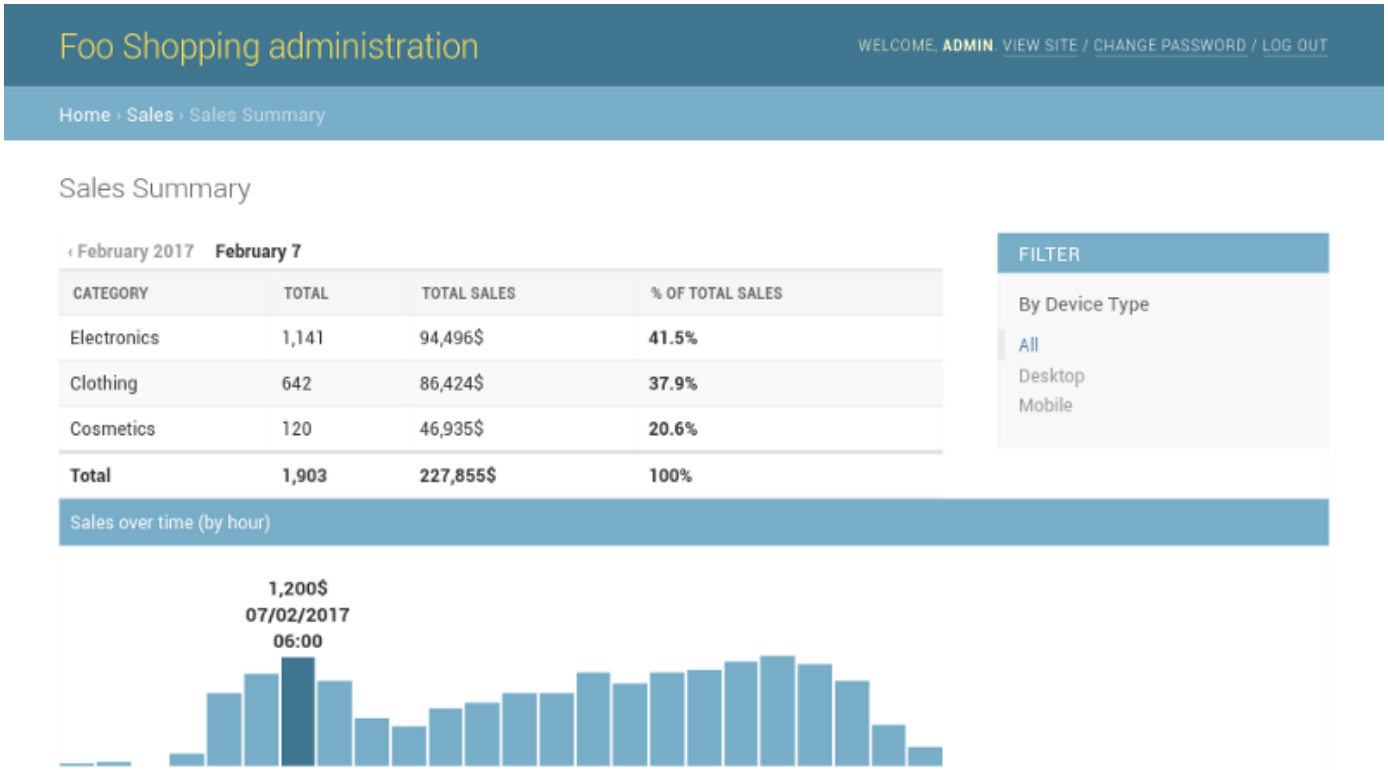
Intérêt de l'admin django [4]



Source :
https://tutorial.djangogirls.org/fr/django_admin/index.html

4.https://docs.djangoproject.com/fr/4.1/ref/contrib/admin/

Intérêt de l'admin django [4]



Source : https://hakibenita.com/how-to-turn-django-admin-into-a-lightweight-dashboard

4.https://docs.djangoproject.com/fr/4.1/ref/contrib/admin/

Activer l'admin

Il est activé par défaut (l'application `django.contrib.admin` est déjà dans les `INSTALLED_APPS`).

Il faut cependant rajouter les urls :

```python
# Source : https://docs.djangoproject.com/fr/4.1/ref/
#          contrib/admin/#hooking-adminsite-instances-into-your-urlconf
from django.contrib import admin
from django.urls import path

urlpatterns = [path("admin/", admin.site.urls)]
```

Un utilisateur pourra s'y connecter (sur `/admin/`) s'il est `is_staff=True`

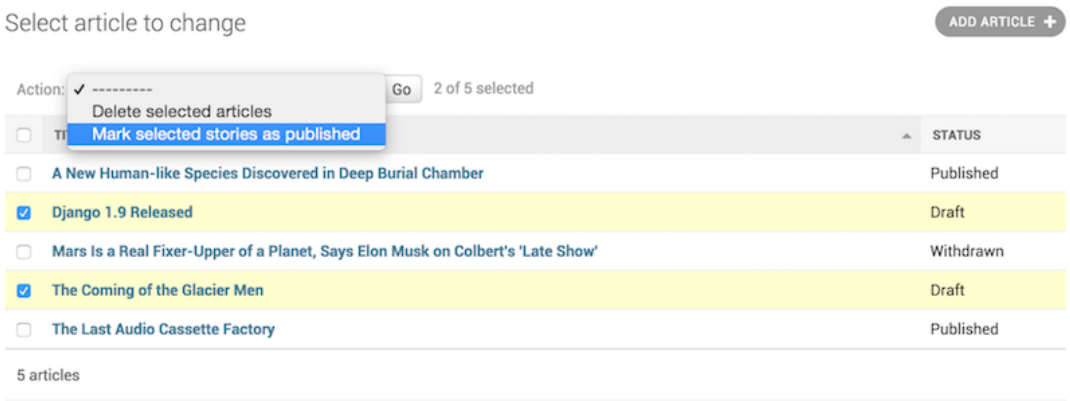
Ajouter un modèle à l'admin

```python
# app/admin.py
# source : https://docs.djangoproject.com/fr/
#          4.1/intro/tutorial07/
from django.contrib import admin

from .models import Question, Choice


class ChoiceInline(admin.TabularInline):
    model = Choice
    extra = 3


class QuestionAdmin(admin.ModelAdmin):
    # champs modifiables dans la partie édition
    fields = ["pub_date", "question_text"]

    # utilisé pour les listing
    list_display = ("question_text", "pub_date", "was_published_recently")

    # clés étrangères à gérer
    inlines = [ChoiceInline]


# on enregistre le modèle (très important)
admin.site.register(Question, QuestionAdmin)
```

Matthieu Falce

Actions personnalisées

```python
# Source : https://docs.djangoproject.com/en/
#          4.1/ref/contrib/admin/actions/#adding-actions-to-the-modeladmin

from django.contrib import admin
from myapp.models import Article


@admin.action(description="Mark selected stories as published")
def make_published(modeladmin, request, queryset):
    queryset.update(status="p")


class ArticleAdmin(admin.ModelAdmin):
    list_display = ["title", "status"]
    ordering = ["title"]
    actions = [make_published]


admin.site.register(Article, ArticleAdmin)
```



2- Django

2.9. Aller plus loin

L'internationalisation

Comment faire en sorte que son site soit accessible à toutes les audiences, quelque soit leur langue :

- internationalisation (i18n) : préparer le serveur pour la localisation (principalement utilisé par les développeurs)
- localisation (l10n): préparer les traductions (principalement géré par des traducteurs)

Django dans tout ça :

- permet de facilement afficher la bonne langue à l'utilisateur (header HTTP, paramètre, ...)
- permet de gérer les traductions

L'internationalisation

Utilisation dans des vues :

```
from django.http import HttpResponse
from django.utils.translation import gettext as _


def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

Et dans des templates :

```
Pour une chaine spécifique :
<title>{% translate "This is the title." %}</title>
ou pour un bloc
{% blocktranslate %}
This string will have {{ value }} inside.
{% endblocktranslate %}
```

L'internationalisation

Les 3 étapes de la gestion des fichiers de langue :

- générer le fichier de langues : `django-admin makemessages -l de` → `locale/de/LC_MESSAGES/django.po`
- on édite le fichier `.po` généré pour inclure les traductions (soit avec un éditeur de texte soit un logiciel spécialisé comme `poedit`[5])
- on compile le fichier `.po` dans un fichier rapide d'accès pour django : `django-admin compilemessages`

5. https://poedit.net/

Optimisation et caches

Il est possible d'améliorer les performances de django :

- l'utilisation d'un ORM peut provoquer le problème N+1 requêtes, il faut faire bien attention à utiliser `select_related` ou `prefetch_related` quand nécessaire (peut avoir des impacts impressionnants sur les performances) [6] [7]
- si cela ne suffit pas, on peut utiliser des mécanismes de caches pour ne pas régénérer des pages ou parties de pages [8]
  - il faut configurer un outil de mise en cache (memcache, redis, base de données, mémoire locale)
  - il faut indiquer quelles sont les vues ou les portions de templates que l'on veut mettre en cache et pour combien de temps
  - il faut faire très attention à l'invalidation de cache [9]

6. https://scoutapm.com/blog/django-and-the-n1-queries-problem
7. https://buildatscale.tech/fixing-n1-query-problem-in-django/
8. https://docs.djangoproject.com/fr/4.1/topics/cache/
9. https://www.quora.com/Why-is-cache-invalidation-considered-difficult

Intégration d'AJAX

- `AJAX` : `asynchronous JavaScript and XML`
- mécanisme permettant d'effectuer des requêtes sans recharger la page
- peut-être utilisé pour charger des morceaux de page (*infinite scrolling*) ou pour soumettre des formulaires
- vous l'avez déjà rencontré (commentaires facebook qui s'affichent automatiquement, page d'actualités qui se mettent à jour en temps réel, jeux en ligne, ...)
- cela est permis par du JavaScript
- django n'a pas à proprement parler de mécanismes facilitant l'AJAX (vous devez écrire du JS)

Intégration d'AJAX

Requete GET

```
{# source : https://testdriven.io/blog/django-ajax-xhr/ #}
<script>
  fetch(url, {
    method: "GET",
    headers: {
      "X-Requested-With": "XMLHttpRequest",
    },
  })
    .then((response) => response.json())
    .then((data) => {
      console.log(data);
    });
</script>
```

Intégration d'AJAX

Requete GET

```python
from django.http import HttpResponseBadRequest, JsonResponse

from todos.models import Todo


def todos(request):
    # request.is_ajax() is deprecated since django 3.1
    is_ajax = request.headers.get("X-Requested-With") == "XMLHttpRequest"

    if is_ajax:
        if request.method == "GET":
            todos = list(Todo.objects.all().values())
            return JsonResponse({"context": todos})
        return JsonResponse({"status": "Invalid request"}, status=400)
    else:
        return HttpResponseBadRequest("Invalid request")
```

Intégration d'AJAX

Requete POST

```
<script>
  fetch(url, {
    method: "POST",
    credentials: "same-origin",
    headers: {
      "X-Requested-With": "XMLHttpRequest",
      "X-CSRFToken": getCookie("csrftoken"),
    },
    body: JSON.stringify({ payload: "data to send" }),
  })
    .then((response) => response.json())
    .then((data) => {
      console.log(data);
    });

  function getCookie(name) {
    let cookieValue = null;
    if (document.cookie && document.cookie !== "") {
      const cookies = document.cookie.split(";");
      for (let i = 0; i < cookies.length; i++) {
        const cookie = cookies[i].trim();
        // Does this cookie string begin with the name we want?
        if (cookie.substring(0, name.length + 1) === name + "=") {
          cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
          break;
        }
      }
    }
    return cookieValue;
  }
</script>
```

Matthieu Falce

Intégration d'AJAX

Requete POST

```python
import json

from django.http import HttpResponseBadRequest, JsonResponse

from todos.models import Todo


def todos(request):
    # request.is_ajax() is deprecated since django 3.1
    is_ajax = request.headers.get("X-Requested-With") == "XMLHttpRequest"

    if is_ajax:
        if request.method == "POST":
            data = json.load(request)
            todo = data.get("payload")
            Todo.objects.create(task=todo["task"], completed=todo["completed"])
            return JsonResponse({"status": "Todo added!"})
        return JsonResponse({"status": "Invalid request"}, status=400)
    else:
        return HttpResponseBadRequest("Invalid request")
```

Intégration d'AJAX

- les requêtes que vous allez faire seront déjà authentifiées
- cependant, il faudra gérer correctement le CSRF des formulaires si vous soumettez des données
- il est possible d'utiliser d'autres verbes HTTP (`put`, `delete`)
- dans les cas les plus complexes, il est peut-être intéressant de passer à des techniques de développement plus "modernes" (front-end indépendant / réactif + API) et utiliser par exemple django-rest-framework

Les tests unitaires

Pour la qualité du code, il peut être intéressant d'écrire des tests unitaires sur ses projets

- il s'agit de tester les fonctions de votre projet
  - soit des fonctions utilitaires
  - soit des fonctions de vues (on parlera plus de tests d'intégration)
- les tests doivent être indépendants
- il faut qu'ils soient joués sur une autre base de données
- vous pouvez utiliser le module `unittest`[10] (bibliothèque standard) ou `pytest`[11] (bibliothèque tierce)

10. https://docs.djangoproject.com/en/4.1/topics/testing/overview/
11. https://pytest-django.readthedocs.io/en/latest/

2- Django

2.10. Déploiement

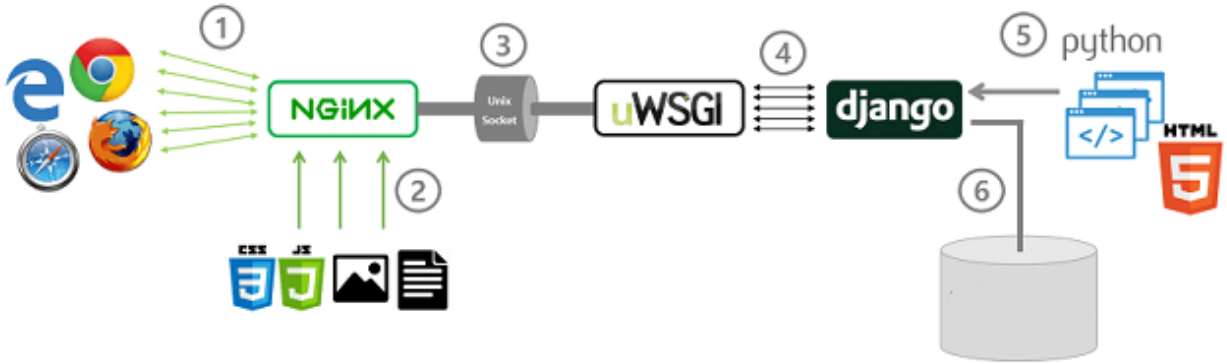
Concepts

- ne plus utiliser `debug=True`
- ne plus utiliser le serveur web de développement
- utiliser un serveur web de production (nginx, apache, caddy) en mode `reverse_proxy`
- faire servir les fichiers statiques directement par le serveur web
- utiliser un server WSGI HTTP pour faire le lien entre django et le serveur web (comme `gunicorn` [12])
- dans le cas de l'utilisation de vues asynchrones

Plus d'infos ici : https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-20-04

12. https://gunicorn.org/

Architecture



Source : https://www.futurefundamentals.com/django-application-deployment-with-nginx-and-uwsgi-ubuntu/

Gestion des fichiers statiques

Il faut gérer les fichiers statiques [13] pour :

- les regrouper à un endroit précis : `manage.py collectstatic` dire dans quels dossiers ils seront stockés : paramétrer `STATIC_ROOT` dans le `settings.py`
-

Il est également possible de stocker ces fichiers depuis un service cloud / CDN comme `AWS S3`

13. https://docs.djangoproject.com/en/4.1/howto/static-files/deployment/

Gunicorn

Gunicorn est un outil python permettant :

- de passer les requêtes entre django et nginx
- de faire tourner plusieurs serveurs django en parallèle (`pre-fork model`) pour avoir de meilleures performances et permettre de servir plusieurs clients en même temps

Configuration de `gunicorn` :

```
gunicorn --access-logfile - \
    --workers 3 --bind unix:/run/gunicorn.sock \
    myproject.wsgi:application
```

NGINX

Voilà une configuration simple de NGINX pour fonctionner avec django[14].

```
server {
    listen 80;
    server_name server_domain_or_IP;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root /home/.../myprojectdir;
    }
    location / {
        include proxy_params;
        proxy_pass http://unix:/run/gunicorn.sock;
    }
}
```

Remarquez que le projet ne se trouve pas dans `/var/www` comme en PHP

14. https://www.digitalocean.com/community/tutorials/how-to-set-up-django-with-postgres-nginx-and-gunicorn-on-ubuntu-20-04

Bibliographie I

Quelques ressources intéressantes pour approfondir vos connaissance en django :

- https://docs.djangoproject.com/en/4.1/
- https://docs.djangoproject.com/fr/4.1/intro/tutorial01/
- https://tutorial.djangogirls.org/fr/django_start_project/
- https://simpleisbetterthancomplex.com/
- https://testdriven.io/
- https://www.docstring.fr