Django, développement Web avec Python

Votre formation

- Horaires
- Repas
- Objectifs

Faisons les présentations

Makina Corpus

Experts en logiciels libres, cartographie et analyse de données, nous concevons des applications métiers innovantes.

Nos valeurs :

- Les logiciels libres et les données ouvertes
- L'agilité
- Le développement durable

Le formateur

```
"{} : dev Django, {} ans d'expérience à votre écoute".format("Julien MARZIN", 5)
```

Et vous ?

Maintenant qu'on se connait : un petit quiz ?

Programme

- Rappel des principes de la POO
- Introduction
- Installation
- Structure
- Requêtes et réponses
- Modèles
- Vues, Templates et Urls
- Formulaires
- ORM
- raw SQL
- Ressources

Rappel des principes de la POO

Définition

Elle « 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. », *Wikipédia*.

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

la POO en python

Définition d'une classe

Pour définir une **classe** en Python, on utilise le mot-clé `class` suivi du nom de la classe.

Tout comme pour les fonctions, tout ce qui est indenté après cette déclaration fera partie de la classe.

```
class Formateur:
    # tout ce qui est après
    # est la définition de
    # la classe
```

Par convention, les noms de classe sont en *CamelCase* c'est à dire que chaque mot commence par une majuscule.

Création d'une instance

Pour créer une **instance** de classe, on l'appelle comme s'il s'agissait d'une fonction.

```
mon_formateur = Formateur()
```

Attributs

On accède à l'attribut d'un objet via la syntaxe

`instance.nom_de_lattribut`

Pour créer un attribut sur une instance, il suffit de lui attribuer une valeur.

```
mon_formateur.prenom = "Julien"
```

Méthodes

Les **méthodes** d'une classe sont **les actions** qui peuvent s'appliquer sur l'objet.

- Pour définir une méthode, on crée **une fonction dans la classe**
- Le premier paramètre de la fonction représente TOUJOURS l'instance courante sur laquelle s'applique la fonction (`self` par convention)
- Les autres paramètres sont libres comme toutes les autres fonctions

Exemple

```
class Formateur:
    # ...
    def se_presenter(self, xp):
        return (
          f"{self.nom} {self.prenom} : dev Django, "
          f"{xp} ans d'expérience à votre écoute"
        )
```

Invocation

Pour invoquer une méthode sur une instance on écrit : `instance.methode(param1, param2, ...)`

Le paramètre `self` ne doit pas être passé en argument, il est fourni automatiquement et c'est l'instance sur laquelle on exécute la méthode.

```
formateur = Formateur()
formateur.nom = "MARZIN"
formateur.prenom = "Julien"
formateur.se_presenter(4)
```

Constructeur

Pour gérer l'initialisation d'une instance, il existe une méthode spéciale, nommée `__init__` :

- Elle est **appelée implicitement** par Python chaque fois que vous créez une instance
- On appelle ce type de méthode un **constructeur**
- Elle permet d'**initialiser** les valeurs par défaut des **attributs** de votre instance.
- Le constructeur peut accepter des paramètres.
- Comme toutes les méthodes, son premier paramètre représente l'instance.
- Les paramètres reçus par cette méthode sont ceux utilisés lors de la création de l'instance.

Exemple

```
class Formateur:
    def __init__(self, nom, prenom, xp=5):
        # On initialise les attributs d'instance ici
        self.nom = nom
        self.prenom = prenom
        self.xp = xp

formateur = Formateur("MARZIN", "Julien")
formateur_2 = Formateur("CORBIN", prenom="Sebastien", 8)

print(formateur.xp)  # Affiche 5
print(formateur_2.xp)  # Affiche 8
```

Méthode spéciale `__str__`

Pour spécifier la manière dont votre instance est convertie en chaîne de caractères lorsque vous l'affichez, il faut définir la méthode `__str__` dans la classe, celle-ci doit retourner une chaîne.

```
class Formateur:
    def __init__(self, prenom):
        self.prenom = prenom

    def __str__(self):
        return f"{self.prenom}"

formateur = Formateur("Julien")
print(formateur)  # Affiche "Julien"
print(f"Votre formateur a pour prenom : {formateur}")  # Affiche :
# "Votre formateur a pour prenom : Julien"
```

Introduction

Qu'est-ce-que Django ?

- Framework en Python pour le Web
- Encourage le développement rapide et propre avec une conception pragmatique
- Django permet de construire des applications web rapidement et avec peu de code
- Malgré son haut niveau d'abstraction, il est toujours possible de descendre dans les couches

Historique

- Créé en 2003 par le journal local de Lawrence (Kansas, USA), basé sur le langage Python créé en 1990
- Rendu Open Source (BSD) en 2005
- Django 1.x : compatible python 2 et 3
- Django 2+ : compatible python 3 uniquement
- Aujourd'hui utilisé par de très nombreuses entreprises/sites : Mozilla, Instagram, Pinterest, Disqus, National Geographic, ...

Philosophie

> *La Plateforme de développement Web pour les perfectionnistes sous pression.*
>
> — *https://docs.djangoproject.com/fr/*

KISS (*Keep It Simple, Stupid*)

> *Simplicity should be a key goal in design and unnecessary complexity should be avoided.*

DRY (*Don't Repeat Yourself*)

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Conventions de codage

La documentation précise certaines conventions de codage spécifiques à Django. La PEP 8 fait référence pour le reste.

Développement rapide

Le but d'un système Web au 21ème siècle est de rendre rapide les aspects fastidieux du développement Web. Django permet un développement Web rapide, fiable et simplifié.

Couplage faible

- Les différentes couches du framework sont indépendantes
- Le socle d'applications peut être réduit au strict minimum

Peu de code à écrire

- Ecriture "automatique" de code
- Utilisation des possibilités d'introspection de Django

Les bonnes raisons d'utiliser Django

- Facile à installer
- Très complet
- Excellente documentation (en français)
- Modèles en Python et ORM faciles à utiliser
- Interface d'administration auto-générée
- Gestion de formulaires
- Serveur de développement inclus
- Extensible, nombreux modules existants
- Communauté autour du projet très active

Architecture MVC, ou plutôt MTV

L'architecture de Django s'inspire du principe MVC (*Model, View, Controller*) ou plutôt MTV (*Model, Template, View*) :

- **Model** : Les modèles sont écrits en Python et Django fournit un ORM (*Django ORM*) complet pour accéder à la base de données **= données**
- **Template** : Django possède son propre moteur de template (*Django Template Engine*) **= comment présenter**
- **View** : Les vues Django peuvent être de simples fonctions Python retournant des réponses HTTP ou être basées sur des classes **= quoi présenter**

cf l'explication selon les créateurs

Environnement

Avec quoi un projet peut fonctionner ?

- Django 2.2+
- Python : 3.5+
- Base de données : SQLite, PostgreSQL, MariaDB / MySQL, Oracle

Côté python

Python parcourt sys.path pour chercher les modules à importer

- Par défaut ce path contient les répertoires systèmes tels que `/usr/lib/python`, `/usr/local/lib/python`, `~/.local/lib/python` ainsi que le répertoire courant en général
- Comme tout module python, il faut que Django soit accessible dans le path pour pouvoir l'utiliser
- **Virtualenv** permet de créer un environnement python en isolation du système, c'est la méthode préférable pour développer avec python

Introduction au virtualenv

```
$ python3 -m venv venv  # crée l'environnement virtuel dans le dossier `venv`
$ ./venv/bin/python3  # lance le python de l'environnement virtuel
$ source venv/bin/activate  # ajoute ./venv/bin en tête du PATH
(venv) $ python3  # lance le python de l'environnement virtuel
(venv) $ deactivate  # rétablit le path
$ python3  # lance le python du système
```

Cela permet ainsi de créer plusieurs environnements avec différentes versions de python, de Django, etc.

Quelques alternatives/extensions : *virtualenvwrapper*, *anaconda*, *pyenv*, *pipenv*.

Installation d'un IDE

Nous utiliserons PyCharm comme IDE pour cette formation.

Rendez-vous sur https://www.jetbrains.com/fr-fr/pycharm/download/#section=linux

Libre à vous d'en choisir un autre par la suite, le meilleur IDE est celui que vous savez utiliser !

Tutoriel fil rouge

Nous créerons une application de gestion de *Todo lists* :

- différentes listes de tâches
- affectation à des utilisateurs
- notifications par e-mail
- catégorisation

Mirroir, mon beau mirroir

Sur les slides, des exemples basés sur la gestion d'une bibliothèque seront utilisés. Ainsi vous pourrez adapter ces exemples à notre application.

Installer Django

Création et activation du *virtualenv*

```
$ python3 -m venv venv
$ source venv/bin/activate
```

ou

```
$ python -m venv venv
```

ou encore sur windows

```
venv\Scripts\activate.bat
```

Installation de Django

```
(venv) $ pip install django
```

Cette commande met à disposition **`django-admin`**, un outil en ligne de commande.

Création du projet

Cela sert à créer une structure de projet avec en argument le nom du paquet python du projet et le nom du répertoire dans lequel le créer (optionnel), par exemple :

```
(venv) $ django-admin startproject library bibliotheque_municipale
```

Lancement du serveur de développement

Une fois le squelette créé, on peut lancer le serveur de développement :

```
(venv) $ cd bibliotheque_municipale
(venv) $ ./manage.py runserver
```

django

View release notes for Django 2.1

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's

Tutorial: A Polling App
Get started with Django

Django Community
Connect, get help, or contribute

Le serveur de développement de Django

Django inclut un serveur HTTP de développement (à ne surtout pas utiliser en production pour des raisons de performances et de sécurité)

- Ce serveur se relance (presque toujours) automatiquement lorsqu'il détecte un changement de fichier
- Par défaut il écoute sur l'interface localhost (`127.0.0.1`) sur le port `8000`
- Le script `manage.py` fait la même chose que `django-admin` mais après avoir lu la configuration du projet (dans `settings.py`)
- Attention ! Il peut s'arrêter inopénément si vous écrivez des bêtises (erreur de syntaxe, classe mal déclarée, etc.), dans ces cas, il faudra le redémarrer

Structure d'un projet Django

Arborescence

```
└── bibliotheque_municipale
    ├── library
    │   ├── __init__.py
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    └── manage.py
```

- `bibliotheque_municipale` : conteneur du projet (le nom est sans importance)
- `manage.py` : utilitaire en ligne de commande permettant différentes actions sur le projet
- `library` : paquet Python effectif du projet
- `library/settings.py` : réglages et configuration du projet
- `library/urls.py` : déclaration des URLs du projet
- `library/wsgi.py` : point d'entrée pour déployer le projet avec WSGI

Projet vs. Application

Il est important de différencier la notion de **projet** et d'**application**.

Une application

Application Web qui fait quelque chose – par exemple un système de blog ou une application de sondage

Un projet

Ensemble de réglages et d'applications pour un site Web particulier.

Projets et applications

Un projet peut contenir plusieurs applications. Une application peut apparaître dans plusieurs projets.

Un paquet de projet peut aussi être considéré comme une application (pour qu'il contienne des modèles, etc.).

> *Un doute ? Consulter la documentation sur le sujet*

Un projet est une combinaison d'applications

- Le projet peut être découpé en différentes apps
- Une même app peut être réutilisée dans plusieurs projets
- Django fournit par défaut des apps, par exemple pour gérer l'authentification
- De nombreuses autres apps sont mises à disposition par la communauté (installation via `pip`)
- Un projet django typique combine des apps de Django, d'autres provenant de la communauté, et enfin une ou des apps spécifiques au projet
- Ce sont des modules python
- La commande `manage.py startapp` crée automatiquement un patron d'app dans un nouveau répertoire
- Les apps sont à déclarer dans les settings (`INSTALLED_APPS = [...]`)
- C'est grace à ce settings que django sait différencier les applications des modules python disponibles dans le path.

Structure d'une application : chaque chose à sa place

Django "impose" une organisation du code (noms et emplacements des fichiers)

- C'est une contrainte mais c'est aussi très pratique pour retrouver son code, quand c'est le sien ou quand c'est celui des autres : ce qui revient au même au bout d'un certain temps...
- Par exemple les vues vont dans le `views.py` dans le répertoire de l'app (ou dans le package `views` de l'app)
- Et les URLs vont dans le module `urls` (fichier `urls.py`)
- On peut inclure la liste des URLs d'une app dans les URLs du projet

```
from django.urls import include, path
urlpatterns = [
    # soit on définit des couples url-vue
    path('', une_vue),
    # ... ou on inclut les urls depuis une autre app
    path('', include('books.urls')),
]
```

Création d'une application

```
$ ./manage.py startapp books
```

Arborescence de l'application créée

```
├── books/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/__init__.py
│   ├── models.py
│   ├── tests.py
│   ├── views.py
```

- `models.py` : déclaration des modèles de l'application
- `views.py` : écriture des vues de l'application
- `admin.py` : comportement de l'application dans l'interface d'administration
- `tests.py` : Il. Faut. Tester.
- `migrations`: modifications successives du schéma de la base de données

Activation de l'application

Déclaration de l'application dans les *settings*

```
# settings.py
INSTALLED_APPS = (
    'django.contrib.admin',
    ...
    'books.apps.BooksConfig,
)
```

Des applications prêtes à l'emploi

Django arrive avec des applications permettant de gérer divers concepts

- `django.contrib.admin`: interface d'administration
- `django.contrib.auth`: gestion d'utilisateurs (connexion, mot de passe)
- `django.contrib.flatpages`: des pages HTML, simplement
- `django.contrib.redirects`: gestion de redirections d'url
- `django.contrib.sites`: gestion de domaines multiples
- et d'autres ...

Certaines sont déjà activées dans notre projet, on verra comment les utiliser

Configuration de la BDD

Django propose une configuration par défaut pour une base SQLite (cf : `settings.py`).

Voici un exemple de configuration pour une base Postgresql :

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.postgresql',
    'NAME': 'library_db',
    'USER': 'library_user',
    'PASSWORD': 'Cx12%a03oa',
    'HOST': 'localhost'
  }
}
```

Création de la structure de la base de données

```
$ ./manage.py migrate
```

Tutoriel fil rouge

Créer le projet (ex: *formation*)

Puis une application (ex: *todo*)

Enfin activer l'application

Tutoriel fil rouge (correction)

Créer le projet (exemple de nom : *formation*)

```
$ django-admin startproject formation
```

Puis une application (ex: *todo*)

```
$ ./manage.py startapp todo
```

Enfin activer l'application

- Déclarer l'application dans les settings

```
# settings.py
INSTALLED_APPS = (
    'django.contrib.admin',
    ...
    'todo.apps.TodoConfig',
)
```

- Créer la structure de la base de données

```
$ ./manage.py migrate
```

Démarrer le serveur

```
$ ./manage.py runserver
```

Fonctionnement général

Déroulement d'une requête HTTP

- Une **vue** est une fonction qui prend un objet `HttpRequest` et renvoie un objet `HttpResponse`
- Quand Django reçoit une requête HTTP, il crée l'objet `HttpRequest` correspondant à la requête du client
- Il cherche la fonction de vue associée à l'URL
- Il appelle cette fonction en lui passant l'objet `HttpRequest` en paramètre
- Il récupère un objet `HttpResponse` en retour de la fonction ou de la classe
- Il répond au client

L'objet `HttpRequest`

Permet d'accéder à de nombreux attributs tels que

- Le schéma (ex. `http` ou `https`), le domaine, et le chemin formant l'URL
- La méthode (ex. `GET`, `POST`, `PUT`, `DELETE`)
- Les headers HTTP (ex. `Content-Type`)
- Les paramètres et les fichiers uploadés
- Les cookies

Peut être lu comme un fichier/flux

- `request.read()`
- `request.readline()`
- `for line in request:`

cf. documentation objet `HttpRequest`

L'objet `HttpResponse`

Permet de régler de nombreux attributs tels que

- Le statut HTTP (ex. `200 OK`, `404 Not Found`)
- Le contenu de la réponse (ex. du code HTML, des données sérialisées en JSON)
- Les headers HTTP (ex. `Content-Type`)
- Les cookies

Peut être instancié directement avec le contenu comme paramètre

```
response = HttpResponse("foobar")
```

Peut être écrit comme un flux

```
response.write()
```

Est dérivé en sous-classes (ex. `HttpResponseRedirect`)

cf. documentation objet `HttpResponse`

Vue simple basée sur une fonction

En somme, une vue se résume à :

- déclarer une URL

```
# library/urls.py
from django.urls import path
import books.views

urlpatterns = [
    path('ma-vue', books.views.ma_vue),
]
```

- retourner un contenu en fonction d'une requête

```
# books/views.py
from django.http import HttpResponse

def ma_vue(request):
    return HttpResponse("mon contenu")
```

TP - Exercice

Créer une vue affichant "Bienvenue dans la Todo List"

Ce sera notre page d'accueil, elle sera accessible sur `http://127.0.0.1:8000/`

TP – Exercice (correction)

- Ajouter une URL

```
# formationtp/urls.py
import todo.views

urlpatterns = [
    path('', todo.views.home_view),
    …
]
```

- Créer une fonction pour la vue

```
# todo/views.py
from django.http import HttpResponse

def home_view(request):
    return HttpResponse("Bienvenue dans la Todo List")
```

Rendez-vous sur http://127.0.0.1:8000/

Les modèles

Déclaration d'un modèle

Les modèles représentent une structure de données stockée en base

- un **modèle** représente généralement une table en BDD et chaque champ une colonne.
- chaque **instance** (objet) de ce modèle sera une ligne dans cette table.

```python
# books/models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    release = models.DateField(blank=True, null=True)
    borrowed = models.BooleanField(default=False)

    # Surcharge de la manière d'afficher un objet Book
    def __str__(self):
        return self.title
```

Django attribue automatiquement une colonne `id` pour la clé primaire sur cette table.

Quelques options pour les modèles

L'ajout de la classe **`Meta`** dans un modèle permet de déclarer des *options de métadonnées* sur le modèle. Celles-ci sont **optionnelles**, un exemple :

```
class Book(models.Model):
    ...

    class Meta:
        db_table = 'book'  # Permet de personnaliser le nom de la table en BDD
        verbose_name = 'Book'  # Le nom lisbile du modèle
        verbose_name_plural = 'Books'  # Le nom au pluriel du modèle
        ordering = ('-release', )  # Le tri par défaut dans les listes
```

D'autres options permettent par exemple de :

- rendre le modèle abstrait
- demander à Django de ne pas gérer ce modèle en base de données
- préciser des critères de tri
- déclarer des permissions relatives au modèle

cf documentation sur les modèles

Quelques types de champs de modèle

- Les champs texte :
  - `CharField` (une ligne avec longueur max)
  - `TextField` (multiligne)
  - `EmailField` (vérifie la syntaxe de l'adresse)
- Les champs pour les nombres :
  - `IntegerField` et `PositiveIntegerField`
  - `FloatField`
  - `DecimalField` (précision fixe, non soumis aux arrondis)
  - `AutoField` (`IntegerField` incrémenté automatiquement)
- Les champs booléens : `BooleanField` et `NullBooleandField`
- Les champs pour la gestion des dates :
  - `DateField`, `TimeField` et `DateTimeField`
  - `DurationField`
- Les champs pour la gestion des fichiers :
  - `FileField` et `ImageField`
  - `FilePathField`

Quelques options pour les champs

Chaque type de champs possède ses propres propriétés. Cependant, certaines sont communes et souvent utilisées comme :

- `verbose_name`: libellé du champ
- `null` : valeur NULL autorisée ou non en base de données
- `blank` : valeur vide autorisée lors de la validation du champ dans un formulaire
- `default` : valeur par défaut pour une nouvelle instance
- `editable` : si le champ doit apparaître automatiquement dans les formulaires
- `choices` permet d'expliciter la liste de valeurs possibles
- `primary_key` pour spécifier qu'un champ est la clé primaire (remplace *id*)
- `unique` ajoute une contrainte d'unicité
- `validators` permet d'ajouter des contraintes de validation au niveau du modèle

cf documentation sur les champs de modèle

Les migrations

- Django permet de faire évoluer les modèles sans devoir effacer les données en génèrant des « diffs » appelés migrations qu'il applique ensuite à la base de données
- Il compare la dernière des migrations existantes aux modèles déclarés en python (peu importe ce qui est dans la base de données)
- Lors de l'application des migrations, il les convertit en SQL et applique toutes les migrations qui n'ont pas déjà été exécutées (la liste des migrations déjà exécutées est stockée dans la table `django_migrations` en BDD)
- Ces migrations sont numérotées et rangées dans les apps dans le sous-répertoire `migrations/`. Il est conseillé d'enregistrer les migrations avec le code

Création d'une migration automatique

Ceci lit la structure des modèles pour créer un fichier de migration correspondant

```
(venv) $ ./manage.py makemigrations
```

Application de la migration

Ceci va lire les fichiers de migration non executés, et les appliquer à la BDD.

```
(venv) $ ./manage.py migrate
```

NB:

- Il est possible de modifier ces fichiers python pour faire des migrations de données. *Exemple : conversion d'un champ texte adresse en plusieurs champs (n°, rue, ville, cp)*
- On peut "désappliquer" une migration, c'est à dire revenir à un état antérieur de la BDD

cf documentation sur les migrations

Déclaration dans l'interface d'administration

L'interface d'administration est le "back-office automatique" de Django. Il liste les instances et par introspection des modèles, crée les formulaires de création/modification correspondants.

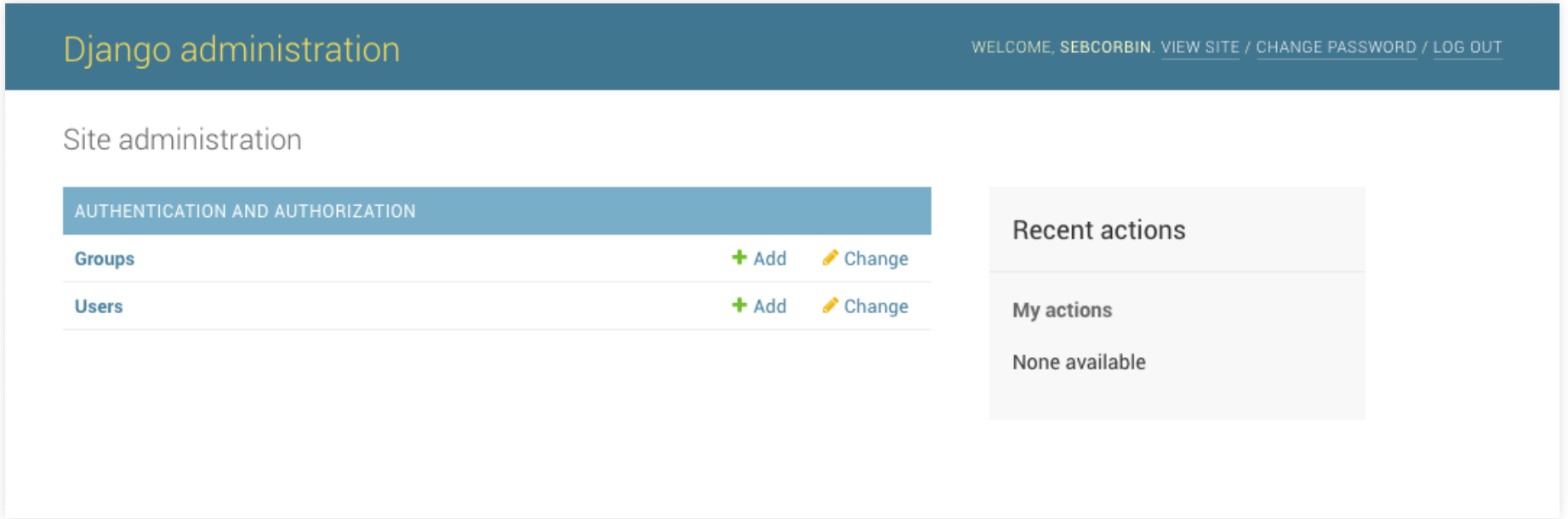
```
# books/admin.py
from django.contrib import admin
from books.models import Book

admin.site.register(Book)
```

Elle est personnalisable et permet de modifier :

- les filtres et l'ordre des listes
- l'affichage des listes
- les formulaires et l'ordre des champs
- ajouter des actions en masse sur les listes

cf documentation sur l'interface d'administration

L'interface d'administration Django



TP - Exercice

Par défaut: Créer le modèle *tâche* ayant notamment les champs :

- titre (texte)
- description (texte long)
- deadline (date et heure)
- réalisée (booléen)

Pour accèder à l'admin de Django, vous aurez besoin d'un superutilisateur :

```
(venv) $ ./manage.py createsuperuser
```

But: Le modèle doit apparaitre dans l'interface d'administration avec les bons champs

TP – Exercice (correction 1/3)

- Déclarer le modèle *tâche*

```
# todo/models.py
from django.db import models


class Task(models.Model):
    """Model for Tasks"""

    title = models.CharField(max_length=100)
    description = models.TextField(blank=True)
    deadline = models.DateTimeField(blank=True, null=True)
    done = models.BooleanField(default=False)
```

TP – Exercice (correction 2/3)

- Générer la migration correspondante

```
(venv) $ ./manage.py makemigrations
Migrations for 'todo':
todo/migrations/0001_initial.py
- Create model Task
```

- Appliquer la migration générée

```
(venv) $ ./manage.py migrate
Operations to perform:
Apply all migrations: admin, auth, contenttypes, sessions, todo
Running migrations:
Applying todo.0001_initial... OK
```

TP – Exercice (correction 3/3)

- Déclarer le modèle dans l'admin de Django

```
# todo/admin.py
from todo.models import Task

admin.site.register(Task)
```

- Pour afficher les titres des tâches dans la liste, il faut utiliser la méthode `__str__` du modèle

```
# todo/models.py
class Task(models.Model):
    ...
    def __str__(self):
        return self.title
```

Le modèle utilisateur

La classe `User` est le coeur du système d'authentification Django. Une instance de `User` représente un utilisateur, une personne qui interagit avec le site. Elle permet plusieurs choses comme :

- gérer des restriction d'accès ;
- personnaliser des profils utilisateurs ;
- associer des contenus à leur créateur.

Quelques propriétés

La classe `User` fournit quelques propriétés de base comme `first_name`, `last_name`, `username`, `password`, `email`. D'autres propriétés, plus *fonctionnelles*, sont à connaitre :

- `is_active` : booléen précisant si le compte est actif ;
- `is_staff` : booléen précisant si l'utilisateur peut accéder à l'interface d'administration ;
- `is_superuser` : booléen spécifiant si l'utilisateur est un super-utilisateur.

L'interface de programmation

Le shell permet d'accéder à l'environnement de django en ligne de commande

```
$ ./manage.py shell
```

Accéder aux objets

Pour lister tous les objets d'un modèle

```
>>> Books.objects.all()
```

Accéder à un objet et à ses paramètres

```
>>> b = Books.objects.all()[0]
>>> b.title
```

D'autres méthodes sont disponibles, elles constituent l'ORM (Object Relation Model) de Django.

TP – Exercice

Accéder aux tâches en utilisant l'interface de commande

TP – Exercice (correction)

Accéder à l'interface de commande

```
./manage.py shell
```

Afficher les objets `Task`

```
>>> from todo.models import Task
>>> Task.objects.all()
<QuerySet [<Task: Faire le TP>]>
>>> t = Task.objects.all()[0]
>>> t
<Task: Faire le TP>
>>> t.title
'Faire le TP'
>>> t.done
False
```

Vue, template et URL

Exemple complet : la liste des livres

Exemple : Création de la vue

Une vue contient la logique nécessaire pour renvoyer une réponse, elle pourrait collecter des données, soumettre un formulaire, afficher du HTML, du JSON ou du XML, générer un fichier zip ou même envoyer un e-mail durant son traitement.

```
# books/views.py
from django.shortcuts import render
from books.models import Book

def book_list(request):
    # Récupération des livres
    books = Book.objects.all()

    # Définition d'un contexte à fournir au template
    context = {
        'books': books
    }

    # Rendu du gabarit
    return render(request, 'books/book_list.html', context)
```

NB: Ce style de vue est dit *function-based* (par opposition à *class-based*).

Exemple : Création d'un template

Un template est spécialement conçu pour renvoyer du HTML en fonction d'un contexte (dans lequel sont contenus les variables).

```
{# books/templates/books/book_list.html #}

<h1>Liste des livres</h1>

{% if books %}
    <ul>
        {% for book in books %}
        <li>{{ book }}</li>
        {% endfor %}
    </ul>
{% else %}
    <p>Aucun livre !</p>
{% endif %}
```

Documentation: https://docs.djangoproject.com/fr/stable/topics/templates/

Exemple : Mapping de l'URL

Django possède un routeur basé sur des regex, avec un préfixe par application. Ce préfixe n'est pas obligatoire mais permet de classer les vues afin d'éviter les conflits, par exemple :

- `book/list` et `book/add`
- `movie/list` et `movie/add`

Déclaration d'une URL

```
# books/urls.py
from django.urls import path
import books.views

app_name = 'books'
urlpatterns = [
    path('list', books.views.book_list, name='list'),
]
```

Inclusion des URLs de l'application au projet

```
# library/urls.py
from django.urls import include, path
from library.views import home

urlpatterns = [
    path('', home, name='home'),
    path('books/', include('books.urls')),
]
```

cf. https://docs.djangoproject.com/fr/stable/topics/http/urls/

Les vues

En détail

Function-based views

Une vue *basée sur une fonction* est une fonction Python qui prend en entrée une **requête HTTP** et retourne une **réponse HTTP**.

Cette réponse peut être une page HTML, un document XML, une redirection, une erreur 404, un fichier PDF ou zip généré à la volée…

Ces vues sont écrites dans le fichier `views.py` ou dans un package `views/` de l'application.

Un exemple tiré de la documention Django

```python
# some_app/views.py
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Class-based views

Une vue *basée sur une classe* est une classe qui prend en entrée une **requête HTTP** et retourne une **réponse HTTP** à travers sa méthode `dispatch()` qui appelle la méthode `get` ou `post()` selon le verbe HTTP.

Un exemple tiré de la documentation Django

```
# some_app/views.py
from django.http import HttpResponse
from django.views.generic import View

class CurrentDatetimeView(View):
    def get(self, request, *args, **kwargs):
        now = datetime.datetime.now()
        html = "<html><body>It is now %s.</body></html>" % now
        return HttpResponse(html)
```

- possibilité d'organiser le code dans différentes méthodes (notamment selon la méthode HTTP entrante) ;
- possibilité d'utiliser l'héritage et les *mixins* pour factoriser et réutiliser le code.

Django propose une bibliothèque riche permettant de travailler avec des vues basées sur des classes, dont la classe `View` est le point central.

Worfklow de base

- La méthode `as_view()` est appelée par l'`URLDispatcher` ;
- Cette méthode instancie la classe et appelle la méthode `dispatch()` de l'instance créée ;
- Celle-ci appelle la méthode `get()`, `post()`, ... en fonction de la méthode HTTP entrante (GET, POST, ...) ;
- Le traitement qui suit dépend du cas d'utilisation, puis une `HttpResponse` est relayée par `dispatch()`.

Comment choisir

Class-based view

Il faut probablement utiliser une vue basée sur une classe ...

- si une des classes de vues génériques fournies par Django s'approche vraiment du besoin
- si la vue peut être créée par héritage d'une autre en surchargeant seulement des attributs
- si la vue à créer peut être réutilisée par héritage et avec peu de modifications par la suite

Function-based views

Il faut probablement utiliser une vue basée sur une fonction ...

- si une implémentation basée sur une classe semble complexe
- si la vue n'a pas vocation à être réutilisée

Rappel : Quelques classes de base

- `View` : classe *mère* et fournit le workflow vu précédemment
- `TemplateView` : faire le rendu d'un template avec un contexte
- `FormView` : gestion d'un formulaire en permettant une bonne organisation du code

Les vues permettant d'afficher des instances

- `ListView` permet de lister très simplement des instances d'un modèle.
- `DetailView` permet d'afficher le détail d'une instance d'un modèle.

Les vues permettant de modifier des instances

- `CreateView` et `UpdateView` sont très utiles pour la création/modification d'instance, de l'affichage du formulaire jusqu'à l'enregistrement.
- `DeleteView` facilite l'implémentation de vues pour la suppression d'instance.

Un excellent site permettant d'avoir un aperçu complet : http://ccbv.co.uk/

Une regex d'URL avec CBV aura cette forme :

```
# some_app/urls.py
from django.urls import path
from some_app.views import CurrentDatetimeView
urlpatterns = [
    path('date-courante', CurrentDatetimeView.as_view()),
]
```

Le moteur de template

En détail

Qu'est-ce qu'un template Django ?

Comme créer du HTML en utilisant des chaînes python n'est pas très pratique, on préfère utilise un système de templates (gabarits en français).

C'est un simple fichier texte qui peut générer n'importe quel format de texte (HTML, XML, CSV, ...).

Un template a accès à des **variables** qui lui auront été passées via un **contexte** par la vue.

NB: Par défaut, Django fournit sa propre syntaxe de template mais il est possible de la remplacer par un autre moteur comme Jinja2.

Où écrire ses templates ?

Pour retrouver les templates d'un projet, Django se base sur le réglage `TEMPLATES`. Le plus souvent on stocke les templates :

- dans chaque application, en suivant l'arborescence `<app>/templates/<app>`. Ils seront retrouvés grâce au loader activé par défaut quand la clé `APP_DIRS` vaut `True`.
- dans un répertoire `templates/` à la racine du projet qu'il faudra déclarer dans la clé `DIRS`.

Dans ce mécanisme de découverte, l'ordre importe : cela permet de surcharger les templates d'autres applications.

cf. https://docs.djangoproject.com/fr/stable/topics/templates/#configuration

Syntaxe de Django template

Affichage d'une variable

```
{{ ma_variable }}
```

Les filtres

Il est possible de modifier l'affichage d'une variable en appliquant des filtres. Un filtre peut prendre (ou non) un argument. Les filtres peuvent être appliqués en cascade. Quelques exemples :

```
{{ name|lower }}
{{ text|linebreaksbr }}
{{ current_time|time:"H:i" }}
{{ weight|floatformat:2|default_if_none:0 }}
```

Django fournit nativement une liste de filtres assez intéressante et il est possible d'écrire des filtres personnalisés facilement.

Les tags

Les tags sont plus complexes que les variables, ils peuvent créer du texte ou de la logique (boucle, condition, ...) dans le tempate.

- Une condition *if*

```
{% if condition %} .. {% else %} .. {% endif %}
```

- Une boucle *for*

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

- Un lien avec *url*

```
<a href="{% url 'books:book_detail' book.pk %}">Django book</a>
```

Django fournit aussi plusieurs tags nativement et il est possible d'écrire ses propres tags.

L'héritage de template

L'intérêt de l'héritage de template est par exemple de pouvoir créer un squelette HTML contenant tous les éléments communs du site et définir des blocs que chaque template pourra surcharger.

Dans un template *parent*, la balise `{% block %}` permet de définir les blocs surchargeables.

Dans un template *enfant*, la balise `{% extends %}` permet de préciser de quel template celui-ci doit hériter.

Exemple de template *parent*

```
{# templates/base.html #}
<html>
  <head>
    <title>
      {% block title %}
        ...
      {% endblock %}
    </title>
    <link href="styles.css" rel="stylesheet" />
  </head>
  <body>
    <header>Entête commune à tout le site</header>
    <section>
      {% block content %}
        ...
      {% endblock %}
    </section>
    <footer>Pied de page commun à tout le site</footer>
  </body>
</html>
```

Exemple de template *enfant*

```
{# books/templates/books/book_list.html #}

{% extends "base.html" %}

{% block title %}
  Liste des livres
{% endblock %}

{% block content %}
  {% if books %}
    <ul>
      {% for book in books %}
        <li>{{ book }}</li>
      {% endfor %}
    </ul>
  {% else %}
    <p>Aucun livre !</p>
  {% endif %}
{% endblock %}
```

L'inclusion de template

L'intérêt de l'inclusion de template est de pouvoir factoriser du code de template :

- pour éviter d'avoir des fichiers de templates trop long
- pour le réutiliser facilement tout en évitant la duplication de code

Cela peut être utile dans différents cas :

- pour certains éléments communs de la page (menu, entête, pied de page, ...)
- pour certaines macros réutilisables (structure d'onglets, affichage en liste d'éléments, structure HTML d'une pop-in, ...)

Exemple de template *appelant*

```
{# templates/base.html #}
<html>
  <head>
    <title>
      {% block title %}
        ...
      {% endblock %}
    </title>
    <link href="styles.css" rel="stylesheet" />
  </head>
  <body>
    {% include 'header.html' %}
    <section>
      {% block content %}
        ...
      {% endblock %}
    </section>
    {% include 'footer.html' %}
  </body>
</html>
```

L'URL dispatcher

En détail

Processus de traitement des requêtes

1. Django identifie le module *URLconf* racine à utiliser (cf **`ROOT_URLCONF`** dans les *settings*), souvent **`projet/urls.py`**
2. Django charge ce module et cherche la variable **`urlpatterns`**
3. Django parcourt chaque motif d'URL dans l'ordre et s'arrête dès la première correspondance avec l'URL demandée. Il tient compte des *include* d'autres **`urls.py`** de chaque application.
4. Une fois qu'une des regex correspond, Django appelle la vue correspondante avec en paramètre la requête HTTP (objet Python **`HttpRequest`**) puis toutes les valeurs capturées dans la regex.
5. Si aucune regex ne correspond, ou si une exception est levée durant ce processus, Django appelle une vue d'erreur appropriée.

Écriture d'une configuration d'URL

Le paramètre **`ROOT_URLCONF`** est un module (sourvent **`projet/urls.py`**) contenant une variable **`urlpatterns`** :

```
# library/urls.py
from django.urls import path
urlpatterns = [
    path('', home, name='home'),
    path('contact', ContactView.as_view(), name='contact'),
    # autant de motifs d'URL que nécessaire...
]
```

À chaque motif peut être associé un nom système qui pourra servir lors de l'inversion d'une URL.

Inclusion d'*URLconf*

Souvent, l'*URLconf* racine incluera les modules URLconf de chaque application :

```
# library/urls.py
from django.urls import include, path
urlpatterns = [
    path('', home, name='home'),
    path('contact', ContactView.as_view(), name='contact'),
    path('books/', include('books.urls')),
]
```

À chaque application peut être associé un namespace qui pourra servir lors de l'inversion d'une URL.

Syntaxe de déclaration d'une URL

URL sans paramètre

```
path('myview', my_view, name='my_view')
```

La vue aura en argument seulement l'objet **`HttpRequest`**.

```
def my_view(request):
    pass
```

URL avec paramètres

La capture de paramètres peut se faire avec **`re_path`** et une expression régulière

```
re_path(r'^archives/(?P<year>\d{4})/(?P<month>\d{2})/$',
    ArchiveView.as_view(), name='archive'),
```

La vue aura en argument l'objet **`HttpRequest`**, puis les valeurs trouvées dans l'expression régulière.

Depuis Django 2.0, il possible d'utiliser une syntaxe spéciale, plus simple :

```
path('archives/<int:year>/<int:month>/', ArchiveView.as_view(), name='archive'),
# types possibles : str, int, slug, uuid
```

Exemple avec l'URL **`http://127.0.0.1:8000/archives/2014/12/`**

```
class ArchiveView(View):
    def get(self, request, year, month):
        print(year)  # 2014
        print(month)  # 12
```

Résolution inversée d'une URL

La résolution consiste à partir d'une URL et trouver le motif correspondant ainsi que ses paramètres.

`/archives/2018/12/` -> `'archive' year=2018 month=12`

La résolution inversée part d'un nom système de motif avec des paramètres pour arriver à une URL.

`'archive' year=2019 month=3` -> `/archives/2019/03/`

Ceci permet de modifier les motifs sans avoir à retoucher toutes les fois où l'URL est appelée (par exemple dans un `<a href>`).

```python
# library/urls.py
urlpatterns = [
    path('books/', include('books.urls')),
]
```

```python
# books/urls.py
app_name = 'books'  # Ceci déterminera le namespace pour les URLs inclues
urlpatterns = [
    path('list', books.views.list, name='list'),
    path('edit/<int:pk>', books.views.edit, name='edit'),
]
```

On pourra utiliser dans du code python

```python
from django.urls import reverse
return HttpResponseRedirect(reverse('books:list'))
return HttpResponseRedirect(reverse('books:edit', kwargs={'pk': 12}))
from django.shortcuts import resolve_url
return HttpResponseRedirect(resolve_url('books:edit', pk=12))  # plus simple
from django.shortcuts import redirect
return redirect('books:edit', pk=12)  # plus simple
```

Ou dans un template

```
<a href="{% url 'books:list' %}">Liste des livres</a>
<a href="{% url 'books:edit' pk=12 %}">Editer le livre 12</a>
```

Tutoriel fil rouge

Créer deux vues :

- *liste des tâches*
- *détail d'une tâche*

Créer auparavant quelques tâches via l'interface d'administration

Pour récupérer les instances :

- liste de toutes les tâches : `Task.objects.all()`
- une seule tâche : `Task.objects.get(pk=<pk>)`

Créer des liens entre la liste et le détail

Pour les plus rapides : amusez-vous à rendre votre site plus beau avec un header/footer

Les formulaires

La bibliothèque `django.forms`

Django possède une bibliothèque assez complète de gestion de formulaires : `django.forms`.

Les concepts principaux sont les suivants :

- La classe `Widget` : permet de gérer et faire le rendu d'un widget HTML (ex: un champ `<input>`, `<textarea>`, ...)
- La classe `Field` : permet de gérer l'initialisation et la validation d'un champ de formulaire
- La classe `Form` : permet de gérer un ensemble de champs de formulaire, ainsi que l'initialisation, le rendu et la validation du formulaire global
- La classe `ModelForm` : permet de gérer des formulaires basés sur des modèles (création / modification d'une instance du modèle)

Déclaration d'un formulaire simple

Un exemple tiré de la documentation Django

```
# contact/forms.py
from django import forms

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

Quelques méthodes souvent utilisées

- La méthode `__init__` : permet de personnaliser l'initialisation du formulaire (par exemple : pré-remplir le champ `sender` par l'email de l'utilisateur connecté)
- La méthode `clean` : permet de personnaliser la validation du formulaire (par exemple : vérifier que `sender` a bien été fourni si `cc_myself` a été coché)

Quelques champs de formulaire

La bibliothèque `django.forms` fournit plus de 20 types de champs différents, dont voici les principaux :

- Champs texte : `CharField`, `SlugField`, `RegexField`, `EmailField`, `UrlField`
- Champs pour les nombres : `FloatField`, `IntegerField`
- Champs booléens : `BooleanField`, `NullBooleandField`
- Champs de sélection : `ChoiceField`, `MultipleChoiceField`
- Champs pour les dates : `DateField`, `DateTimeField`, `TimeField`, `DurationField`
- Champs pour les fichiers : `FileField`, `FilePathField`, `ImageField`
- Champs pour les relation : `ModelChoiceField`, `ModelMultipleChoiceField`

cf https://docs.djangoproject.com/fr/stable/ref/forms/fields/

Champs et Widgets

Chaque champ peut être initialisé avec des options, les plus utilisées sont `label`, `required`, `initial` et `widget`.

Les champs de formulaire gèrent les données, les widgets gèrent la manière dont elles sont saisies.

Par exemple, un `CharField` gérera des données de texte, et aura par défaut un widget `TextInput`, mais il est possible de spécifier un widget `Textarea`.

```
class CommentForm(forms.Form):
    title = forms.CharField(label="Donner un titre à votre commentaire")
    message = forms.CharField(widget=forms.Textarea)
```

Certains modules annexes fournissent leurs propres champs et il est possible d'écrire des champs personnalisés.

Formulaire dans une *Function-based view*

```
from django.shortcuts import render, redirect
from myapp.forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)  # instanciation avec données
        if form.is_valid():
            # Traite les données dans form.cleaned_data puis redirige
            ...
            return redirect('/thanks/')
    else:
        form = ContactForm()  # instanciation sans données
    # Affichage du formulaire via un template
    return render(request, 'contact.html', {'form': form})
```

Méthode simplifiée

```
def contact(request):
    form = ContactForm(request.POST or None)
    if form.is_valid():
        ...
        return redirect('/thanks/')
    return render(request, 'contact.html', {'form': form})
```

Rendu du formulaire dans un template

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

L'utilisation du tag `{% csrf_token %}` est importante car elle permet de protéger le formulaire des attaques de type CSRF (*Cross Site Request Forgeries*).

Un formulaire peut être rendu de différentes manières :

- `as_p`: chaque champ est rendu dans un paragraphe
- `as_ul`: chaque champ est rendu dans une ligne de liste
- `as_table`: chaque champ est rendu dans une ligne de tableau

Des packages de la communauté (_cripsy*forms*, *floppyforms*, *material*) permettent d'afficher le bon markup pour un framework HTML/CSS (Bootstrap, Foundation, Uniform, ...).

Les formulaires de modèles

La classe `ModelForm` permet de créer automatiquement des formulaires basés sur des modèles.

Le fonctionnement est assez semblable à celui des formulaires classiques à quelques différences près :

- La déclaration d'une classe `Meta` est nécessaire pour préciser sur quel modèle doit se baser le formulaire
- La méthode `__init__` prend en argument l'instance du modèle à modifier (ou `None` dans le cas d'une création)
- Le formulaire fournit une méthode `save` qui permet d'enregistrer l'instance éditée via le formulaire
- Les champs sont automatiquement listés dès lors qu'ils n'ont pas la propriété `editable=False`
- On peut quand même ajouter d'autres champs qui ne seront pas liés au modèle

Un exemple d'utilisation d'un ModelForm

En reprenant le modèle d'exemple :

```
### books/models.py
class Book(models.Model):
    title = models.CharField(max_length=100)
    release = models.DateField()
    borrowed = models.BooleanField(default=False)
```

Le formulaire basé sur `ModelForm`

```
### books/forms.py
class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ('title', 'release')
        #fields = '__all__'
```

Avec une Function-based view

```
### books/views.py
def add_book(request):
    form = BookForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('/books/')
    return render(request, 'add_book.html', {'form': form})
```

Note : pour modifier une instance, il faut la passer en paramètre du formulaire

```
book = Book.objects.get(pk=pk)
form = BookForm(instance=book)
```

Utilisation de ModelForm avec une Class-Based View

```
class AddBookView(CreateView):
    form_class = BookForm
    success_url = reverse_lazy('book:list')
    template_name = 'add_book.html'
```

ou même plus simple, puisque le formulaire est facultatif

```
class AddBookView(CreateView):
    model = Book
    fields = ('title', 'release')
    success_url = reverse_lazy('book:list')
    template_name = 'add_book.html'
```

Tutoriel

Créer les vues :

- ajout de tâche
- modification d'une tâche

Ajouter des liens depuis la liste de tâches vers chaque tâche et vers le formulaire d'ajout.

Pour les plus rapides, convertissez ces vues en Class-Based View (aide)

Tutoriel – Exercice (correction 1/3)

Le formulaire

```
### todo/forms.py
from django import forms
from todo.models import Task

class AddTaskForm(forms.ModelForm):

    class Meta:
        model = Task
        fields = ("title", "description",)
```

Tutoriel – Exercice (correction 2/3)

La vue

```
### todo/views.py
def add_task(request):
    """Add task view"""
    form = AddTaskForm(request.POST or None)
    if form.is_valid():
        form.save()
        return redirect('/todo/list')
    return render(request, 'todo/add_task.html', {'form': form})
```

L'URL

```
### todo/urls.py, dans urlpatterns
path('add', views.add_task, name='add'),
```

Tutoriel – Exercice (correction 3/3)

Le template d'ajout

```
<!-- todo/templates/todo/add_task.html -->
<h1>Ajouter une tâche</h1>
<form method="post">
    {% csrf_token %} {{ form.as_p }}
    <input type="submit" value="Submit" />
</form>
```

Le lien vers le détail de la tâche

```
<!-- todo/templates/todo/add_task.html -->
<a href={% url 'todo:detail' task.pk %}>{{ task }}</a>
```

Le lien pour ajouter une tâche

```
<!-- todo/templates/todo/add_task.html -->
<a href={% url 'todo:add' %}>Ajouter une tâche</a>
```

Tutoriel – Bonus (correction)

La classe `TaskCreateView` hérite de `CreateView`. Comme la vue, elle requiert les informations sur le template, le formulaire et la redirection.

```
### todo/views.py
from django.views.generic.edit import CreateView

class TaskCreateView(CreateView):
    template_name = 'todo/add_task.html' # par défaut : 'todo/task_form.html'
    form_class = AddTaskForm
    success_url = '/todo/list'
```

L'URL

```
### todo/urls.py, dans urlpatterns
('add', views.TaskCreateView.as_view(), name='add'),
```

Bonne pratique : les validateurs

Lorsqu'on travaille sur un champ de modèle, c'est une bonne pratique de créer un validateur au niveau du modèle et non au niveau du formulaire, sinon il faudrait copier-coller la validation du champ sur chaque formulaire travaillant sur le modèle.

```
from django.core.exceptions import ValidationError

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError(
            _('%(value)s is not an even number'),
            params={'value': value},
        )

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])

class MyForm(forms.Form):
    even_field = forms.IntegerField(validators=[validate_even])
```

L'avantage, c'est qu'ils fonctionnent sur les modèles et les formulaires.

Liste de validateurs déjà présents

- `RegexValidator`
- `EmailValidator`
- `URLValidator`
- `validate_ipv4_address`
- `validate_comma_separated_integer_list`
- `MaxValueValidator`, `MaxLengthValidator`, `DecimalValidator`…
- `FileExtensionValidator`, `validate_image_file_extension`
- `ProhibitNullCharactersValidator`

Créer un champ de formulaire personnalisé

Ainsi avec ces techniques, il facile de créer des sous-classes de champs existants. Par exemple, un champ *N° de téléphone* qui vérifie le format.

TP

Ajouter un validateur pour empêcher que la deadline d'une tâche soit antérieur à aujourd'hui

Relations entre les modèles

Les différents champs de relations

La bibliothèque `django.models` fournit différents champs spécifiques pour représenter les relations entre modèles.

- `models.ForeignKey` : représente une relation de type 1-N
- `models.ManyToManyField` : représente une relation de type N-N
- `models.OneToOneField` : représente une relation de type 1-1

Le champ ForeignKey

Le champ **`ForeignKey`** doit être déclaré avec comme premier argument le modèle auquel il est lié par cette relation 1-N. L'argument optionnel **`related_name`** permet de nommer la relation inverse à partir de ce modèle lié.

La représentation de ce champ en base de données est une contrainte de type clé étrangère.

Exemple

Un livre est associé à un auteur, un auteur peut avoir écrit plusieurs livres.

```
# books/models.py
class Author(models.Model):
    name = models.CharField(max_length=50)

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author, related_name='books',
                                       on_delete=models.PROTECT)
```

Le champ ManyToManyField

Le champ `ManyToManyField` doit être déclaré de la même manière que le champ `ForeignKey`.

La représentation de ce champ en base de données est une table contenant deux clés étrangères vers les deux tables des modèles liés.

Exemple

Un livre est associé à plusieurs catégories, plusieurs livres peuvent appartenir à une même catégorie.

```
# books/models.py
class Category(models.Model):
    label = models.CharField(max_length=50)

class Book(models.Model):
    title = models.CharField(max_length=100)
    categories = models.ManyToManyField(Category, related_name='books')
```

Le champ OneToOneField

La déclaration du **`OneToOneField`** est similaire.

La représentation de ce champ en base de données est une clé étrangère possédant une contrainte d'unicité.

Exemple

Un livre est associé à un seul code barre, un code barre correspond à un seul livre.

```
# books/models.py
class BarCode(models.Model):
    code = models.CharField(max_length=50)

class Book(models.Model):
    title = models.CharField(max_length=100)
    barcode = models.OneToOneField(BarCode, related_name='book')
```

Tutoriel

Mettre en place une modélisation gérant des listes de tâches partagées entre utilisateurs

L'ORM

Les moteurs de base de données

- 4 moteurs disponibles dans l'ORM de Django
  - PostgreSQL `django.db.backends.postgresql`
  - MySQL `django.db.backends.mysql` supporte aussi MariaDB
  - Oracle `django.db.backends.oracle`
  - SQLite `django.db.backends.sqlite3`
- SQLite non recommandé en production
- Django est d'abord pensé pour PostgreSQL (champs `DateRangeField`, `JSONField`, extension PostGIS, recherche plein texte, etc.)
- Possible de changer de moteur sans réécrire le code (mais il faut migrer les éventuelles données), sauf pour certaines spécificités
- Recommandé de développer (ou au moins de tester) avec le moteur utilisé en production
- Moteur spécifié dans `settings.py` (variable `DATABASES`), ainsi que la configuration du nom de la base, du serveur, et de l'authentification

cf https://docs.djangoproject.com/fr/stable/ref/databases/

ORM (Object-relational mapping)

- Fait correspondre une classe Python à une table SQL
- Fait correspondre un objet python, instance de cette classe, à un enregistrement de cette table SQL
- Il y a donc juste des classes et objets python à manipuler, aucun SQL à écrire, que ce soit :
  - Pour créer et modifier les tables
  - Pour créer et modifier les données
  - Pour interroger la base
- Facilite la gestion des relations entre modèles (jointures)
- A sa propre "opinion", nécessite souvent des optimisations

Manipulation d'une instance de modèle

Pour créer un objet en base, il suffit de l'instancier en passant en argument les noms des attributs du modèle. L'instance dispose ensuite d'une méthode `save` qui permet de l'enregistrer en base de données.

```
>>> b = Book(name='Two scoops of django',
             release=date(2013, 8, 31))
>>> b.save()
```

La même méthode `save` est utilisée pour enregistrer en base de données des modifications sur l'instance.

```
>>> b.name ='Two scoops of django - Best practices'
>>> b.save()
```

Pour supprimer une instance, il suffit d'appeler la méthode `delete()` qui permet de supprimer directement la ligne en base de données.

```
# Suppression
>>> b.delete()
```

Les concepts `Manager` & `Queryset`

Pour récupérer une ou plusieurs instances, il faut construire un `Queryset` via un `Manager` associé au modèle.

Qu'est ce qu'un `Manager` ?

Un `Manager` est l'interface à travers laquelle les opérations de requêtage en base de données sont mises à disposition d'un modèle Django. Chaque modèle possède un `Manager` par défaut accessible via la propriété `objects`.

Qu'est ce qu'un `Queryset` ?

Un `Queryset` représente une collection d'objets provenant de la base de données. Cette collection peut être filtrée, limitée, ordonnée, … grâce à des méthodes qui correspondent à des clauses SQL.

A partir d'un queryset il est possible d'obtenir un autre queryset plus spécialisé. Un queryset est "paresseux" : la requête SQL n'est exécutée que lorsqu'il n'est plus possible de la retarder.

Retrouver une liste d'instances

Retrouver toutes les instances d'un modèle

```
>>> Book.objects.all()
```

Retrouver une liste filtrée d'instances

Les méthodes de filtrage principalement utilisées sont **`filter`** et **`exclude`**. Il est possible de les chaîner.

```
>>> Book.objects.filter(
        release__gte=date(2013, 1, 1)
      ).exclude(
        borrowed=True
      )
```

Retrouver une liste ordonnée d'instances

```
>>> Book.objects.exclude(borrowed=True).order_by('title')
```

ORM - Filtrage

- Les paramètres nommés sont le nom du champ et la valeur
- On peut ajouter derrière le nom du champ deux undescores et un lookup
  - `__iexact` pour une recherche insensible à la casse
  - `__contains` pour chercher à l'intérieur
  - `__lt`, `__lte`, `__gt`,`__gte` pour les inégalités
- Avec deux undescores on peut aussi suivre une relation
- Il y a un **ET** logique entre les différentes conditions

cf https://docs.djangoproject.com/fr/stable/ref/models/querysets/

```
books = Book.objects.filter(title__startswith="Le")
books = Book.objects.filter(release__year__lt=1950) \
                    .exclude(title__icontains="fleurs")
```

Encapsuler plusieurs conditions

Pour l'opérateur OU ou des requêtes plus complexes, utiliser **`django.db.models.F`** et **`django.db.models.Q`**, qui permettent des combinaisons avant exécution.

```
Book.objects.filter(
    Q(release__date__gt=date.today()) | Q(title__contains=F('author__name')),
)
```

cf https://docs.djangoproject.com/fr/stable/ref/models/expressions/

Retrouver une instance en particulier

La méthode `get` permet de récupérer une instance particulière.

```
>>> Book.objects.get(pk=12)
```

La méthode ne peut retourner qu'une instance précise, il faut donc que le filtre fourni ne soit pas ambigu. Il faut veiller à filtrer sur un champ `unique` (ou plusieurs champs uniques ensemble).

Exceptions potentielles

- Si l'instance n'est pas trouvée, une exception `Book.DoesNotExist` sera levée (de manière générique : `<Model>.DoesNotExist`).
- Si plusieurs instances ont été trouvées, l'exception levée sera `Book.MultipleObjectsReturned` (`<Model>.MultipleObjectsReturned`).

Référence à des objets associés

Pour les relations entre instances (`ForeignKey`, `ManyToManyField`), Django fournit un `Manager` spécifique nommé `RelatedManager`. Il permet de :

- retrouver les instances liées par une `ForeignKey`
- ajouter une liaison entre deux instances dans le cas d'un `ManyToManyField`
- supprimer toutes les liaisons d'une instance vers d'autres

On peut également utiliser la relation inversée avec `related_name`.

Exemples

Récupérer toutes les livres d'une catégorie

```
# Récupérer toutes les livres d'une catégorie
>>> category = Category.objects.get(pk=5)
>>> category.books.all()


# Retrouver les livres disponibles d'un auteur
>>> author = Author.objects.get(pk=25)
>>> author.books.filter(borrowed=False)


# Ajouter un livre à une catégorie
>>> book = Book.objects.get(pk=12)
>>> book.categories.add(category)


# Supprimer tous les livres d'une catégorie (seulement l'association)
>>> category.books.clear()
```

Tutoriel

Mettre en place un formulaire de filtrage de tâches :

- entre deux dates
- sur le titre
- sur le fait qu'elles soient terminées ou non

Note : il sera plus facile d'utiliser un formulaire GET que POST

Quelques fonctionnalités avancées

Internationalisation et localisation

Quelques réglages

Plusieurs réglages dans `settings.py` permettent d'activer ou non certaines fonctionnalités liées à l'internationalisation et la localisation :

- `USE_I18N` : active ou non le module de traduction
- `USE_L10N` : active ou non l'affichage des dates et des nombres selon la langue
- `USE_TZ` : active ou non la gestion des fuseaux horaires
- `LANGUAGE_CODE` : langue par défaut
- `LANGUAGES` : liste des langues connues par l'application
- `LOCALE_PATHS` : chemins vers les fichiers de traduction

Traduire l'interface (fichiers python)

Utilisation de la bibliothèque **`gettext`**

Pour traduire les différents textes de l'interface, on utilise les fonctions `getttext`, ou plus souvent `gettext_lazy`. Pour la simplicité de l'écriture, on importe généralement cette fonction avec l'alias _.

```
# models.py
from django.utils.translation import gettext_lazy as _

class Book(models.Model):
    name = models.CharField(max_length=100, verbose_name=_('Title'))

    class Meta:
        db_table = 'task'
        verbose_name = _('Book')
        verbose_name_plural = _('Books')
```

Traduire l'interface (templates)

Deux tags permettant de traduire l'interface directement dans les templates sont disponibles :

Le tag {% trans %}

Il permet de traduire une chaine de caractères simple ou le contenu d'une variable.

```
{% load i18n %}
<title>{% trans "List of books" %}</title>
<title>{% trans page_title %}</title>
```

Le tag {% blocktrans %}

Il permet de mixer chaînes de caractères et variables pour traduire des chaînes complexes.

```
{% load i18n %}
{% blocktrans with book_t=book|title author_t=author|title %}
<p>This is {{ book_t }} by {{ author_t }}</p>
{% endblocktrans %}
```

Gérer les fichiers de traduction

Créer / mettre à jour le fichier de traduction

La commande `makemessages` permet de créer le fichier traduction pour une langue donnée (fichier texte avec l'extension `.po` contenant les identifiants de messages et les traductions correspondantes). Cette commande doit être lancée depuis la racine de l'application ou du projet pour lequel on crée le fichier car elle génère une arborescence de dossiers `locale/LANG/LC_MESSAGES`.

```
$ ./manage.py makemessages -l fr
```

Compiler le fichier de traduction

La commande `compilemessages` permet de compiler le fichier de traduction pour qu'il soit utilisable dans le code Python.

```
$ ./manage.py compilemessages -l fr
```

Tutoriel

Mettre en place la traduction et traduire quelques chaînes

Les vues

Django apporte nativement quelques vues facilitant l'authentification et la gestion du mot de passe des utilisateurs, principalement :

- `LoginView`
- `LogoutView`
- `logout_then_login`
- `PasswordChangeView`
- `PasswordResetView`

https://docs.djangoproject.com/fr/stable/topics/auth/default/#module-django.contrib.auth.views

Quelques settings permettent aussi de simplifier cette gestion :

- **`LOGIN_URL`** : URL vers la vue de connexion
- **`LOGIN_REDIRECT_URL`** : URL de redirection après l'authentification de l'utilisateur
- **`LOGOUT_URL`** : URL de la vue de déconnexion

Tutoriel

- Créer une page de connexion

Django et AJAX

Il est possible d'ajouter du "dynamisme" à nos applications en y implémentant des méthodes dites "Ajax". Ajax est l'acronyme d'asynchronous JavaScript and XML.

Tutoriel

Nous allons ajouter un petit peu de "dynamisme" sur notre page de création d'une tâche.

Nous allons essayer de voir si le nom de la tâche renseigné par l'utilisateur AVANT sa création est déjà pris.

Ajouter jQuery à la fin du template de base ainsi qu'un nouveau block :

```
[...]
    <script
      src="https://code.jquery.com/jquery-3.6.1.min.js"
      integrity="sha256-o88AwQnZB+VDvE9tvIXrMQaPlFFSUTR+nldQm1LuPXQ="
      crossorigin="anonymous">
    </script>
    {% block javascript %}{% endblock %}
  </body>
</html>
```

suite Ajax

Rendez-vous sur le template d'ajout d'une tâche et y ajouter :

```
{% block javascript %}
  <script>
    $("#id_name").change(function () {
      console.log( $(this).val() );
    });
  </script>
{% endblock %}
```

suite Ajax

Avant de continuer côté client, créons la vue nous permettant de vérifier si le nom renseigné est disponible :

```
from django.http import JsonResponse

def validate_task_name(request):
    task_name = request.GET.get('task_name', None)
    data = {
        'is_taken': Task.objects.filter(name__iexact=task_name).exists()
    }
    return JsonResponse(data)
```

suite Ajax

Il ne reste plus qu'à implémenter la requête Ajax côté client, retournons dans notre template :

```
$("#id_name").change(function () {
  var name = $(this).val();

  $.ajax({
    url: '/ajax/validate_task_name/',
    data: { 'task_name': name },
    dataType: 'json',
    success: function (data) {
      if (data.is_taken) alert("A task with this name already exists.");
    }
  });
});
```

Et voilà !

Introduction aux tests automatisés

Arborescence

Une architecture possible pour organiser les tests d'une application consiste à créer, dans un dossier **`tests`**, un fichier de tests (**`test_views.py`**, **`test_models.py`**, ...) par fichier de l'application (**`views.py`**, **`models.py`**, ...).

```
├── todolist
│   ├── __init__.py
│   ├── forms.py
│   ├── models.py
│   ├── views.py
│   ├── tests
│   │   ├── __init__.py
│   │   ├── test_forms.py
│   │   ├── test_models.py
│   │   └── test_views.py
```

Les tests automatisés sont les premières briques indispensables pour garantir une application fiable et éviter les régressions au fil du temps.

cf : https://docs.djangoproject.com/fr/stable/topics/testing/overview/

Tester un modèle

Le modèle

Voici un modèle très basique :

```
# models.py

class Author(models.Model):
    firstname = models.CharField(max_length=100, null=True, blank=True)
    lastname = models.CharField(max_length=100)

    def __str__(self):
        return '%s %s' % (self.firstname, self.lastname)
```

L'idée n'est pas de tester Django (création d'instance, vérification que les différents champs fonctionnent, etc) mais bien de tester notre code personnel. Ici, seule la fonction `__str__` est donc à tester.

Il faut prendre soin de tester les différents cas possibles d'exécution (en l'occurrence, la présence d'un `firstname` ou non).

Le test

```
# tests/test_models.py
from django.test import TestCase
from library.models import Author


class AuthorAsStringTest(TestCase):

    def test_with_first_name(self):
        author = Author.objects.create(firstname='René',
                                       lastname='Descartes')
        self.assertEqual(str(author), 'René Descartes')

    def test_without_first_name(self):
        author = Author.objects.create(lastname='Platon')
        self.assertEqual(str(author), 'Platon')
```

Un test rapide est un test qui crée peu d'instances en base de données.

Exécution des tests

```
$ ./manage.py test library
Creating test database for alias 'default'...
.F
======================================================================
FAIL: test_without_first_name (library.tests.test_models.AuthorAsStringTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/al/makina-slides/django/example_projects/libraryproject/library/tests/test_
    self.assertEqual(str(author), 'Platon')
AssertionError: 'None Platon' != 'Platon'
- None Platon
+ Platon

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Le modèle corrigé

```
class Author(models.Model):
    firstname = models.CharField(max_length=100, null=True, blank=True)
    lastname = models.CharField(max_length=100)

    def __str__(self):
        if self.firstname:
            return '%s %s' % (self.firstname, self.lastname)
        else:
            return self.lastname
```

Exécution des tests :

```
$ ./manage.py test library.tests.test_models
Creating test database for alias 'default'...
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
Destroying test database for alias 'default'...
```

Tester un formulaire

Le formulaire

```
# forms.py
from django import forms

class PeriodForm(forms.Form):
    begin = forms.DateField()
    end = forms.DateField()

    def __init__(self, *args, **kwargs):
        super(PeriodForm, self).__init__(*args, **kwargs)

        begin = self.initial.get('begin', None)
        if begin:
            self.initial['end'] = begin.replace(month=begin.month + 1)
```

Comme pour le modèle, l'idée n'est pas de tester ce qui est du ressort de Django. Ici, il est simplement nécessaire de s'assurer que la fonction `__init__` fonctionne correctement dans les différents cas possibles (présence ou non d'une valeur initiale pour le champ `begin`).

Le test d'un formulaire

```
# tests/test_forms.py
from datetime import date
from django.test import TestCase
from library.forms import PeriodForm


class PeriodFormTest(TestCase):
    def test_init_without_begin(self):
        f = PeriodForm()
        self.assertIsNone(f.initial.get('end'))

    def test_init_with_begin(self):
        initial = {'begin': date(2014, 1, 1)}
        f = PeriodForm(initial=initial)
        self.assertEqual(f.initial.get('begin'), date(2014, 1, 1))
        self.assertEqual(f.initial.get('end'), date(2014, 2, 1))
```

Tester une vue

La vue

```
from django.shortcuts import render
from datetime import date
from .models import Book

def recent_books(request):
    today = date.today()
    threshold = today.replace(year=today.year - 1)

    results = Book.objects.filter(published__gt=threshold)

    return render(request, 'library/book_list.html', {
        'results': results
    })
```

Le modèle

```
class Book(models.Model):
    title = models.CharField(max_length=200)
    published = models.DateField(null=True, blank=True)
```

Le test d'une vue

```
# tests/test_views.py
class BookViewTest(TestCase):
    def test_recent_books_view(self):
        recent_date = date.today()
        recent_book = Book.objects.create(title='Titre du livre récent',
                                          published=recent_date)

        old_date = recent_date.replace(year=recent_date.year - 1)
        old_book = Book.objects.create(title='Titre du vieux livre',
                                       published=old_date)

        response = self.client.get("/library/recent/")

        self.assertNotContains(response, old_book.title)
        self.assertContains(response, recent_book.title)
```

On a vérifié :

1. que la vue fonctionne correctement (pas d'erreur 500);
2. que la requête renvoit les résultats attendus.

A savoir

La classe `dango.test.TestCase` :

- reset la base de données à l'état initial à chaque fin de test
- possède l'attribut `client` qui permet de faire des requêtes et conserver les cookies de session
  - `self.client.get()` et `self.client.post()`
  - `self.client.login()` et `self.client.force_login()`
- permet de créer des données de test pour toutes les méthodes en surchargeant `setUp()` ou `setUpTestData()`
- permet de charger des fixtures (données initiales en début de test)
- fournit quantités d'assertions (https://docs.djangoproject.com/en/2.1/topics/testing/tools/#assertions)
  - `self.assertEqual(1, 3 - 2)`
  - `self.assertContains(response, "Mon titre")`

Ressources

Pour aller plus loin

Quelques modules indispensables

- `django_extensions` : plusieurs extensions et outils d'administration très pratiques
- `django_debug_toolbar` : une barre latérale permettant de faire du *debug* et du *profiling* page par page
- `django_hijack`: permet de se connecter avec un autre utilisateur sans se déconnecter
- `django_extra_views`: apporte d'autres CBV pour des formulaires et vues toujours plus rapides
- `django_braces`: apporte des mixins pour vos CBV

Les plus gros projets

- `wagtail` : CMS
- `django-rest-framework` : pour développer des APIs REST
- `sentry` : gestion de rapport d'erreurs
- `taiga` : gestion de projet en mode scrum
- `django-oscar` : e-commerce

Quelques modules souvent utilisés

- `django_compressor` : compression des fichiers statiques
- `django_pagination` : affichage de listes paginées
- `django_sorting` : affichage de tableaux triables
- `django_filters` : création de liste filtrées
- `django_crispy_forms` : affichage de forms avec Bootstrap/Foundation/Uniform
- `django_xworkflows` : gestion de workflows
- `django_modeltranslation` : gestion de modèles multilingues
- `easy_thumbnails` ou `versatileimagefield` ou `sorl.thumbnail` : gestion de miniatures pour les images
- `django_ckeditor` : intégration d'un *widget* CKEditor
- `django_allauth` : gérer plus finement la connexion utilisateur
- …

Où obtenir des informations ?

- http://www.djangoproject.com [EN]
- http://docs.djangoproject.com/fr [FR]
- http://stackoverflow.com/questions/tagged/django

Modules

- Comparaison/référencement : https://www.djangopackages.com/
- Liste de modules django : https://awesome-django.com/
- Liste de packages python : https://awesome-python.com/

Contacts/Evenements

- La *mailing list* Django : django@lists.afpy.org
- Les *channels* IRC : #django, #django-fr
- Les meetups python et les PyConFr
- Djangocong : Conférence annuelle française
- Djangocon-eu : Conférence annuelle européenne

Merci !

Des questions ? Des remarques ?

Gardons le contact

Site : https://makina-corpus.com

Twitter : https://twitter.com/makina_corpus