



Attention, ce cours était fait sur Django 1 et début de la v2.  
Il faut adapter beaucoup de choses pour qu'il marche en v3 mais  
vous aurez l'idée au moins

# DJANGO

Pour faire du web comme les grands

# Comment ça marche ?

- ❖ Django considère un projet comme un ensemble d'applications
  - ❖ Structuration du code : séparation des composants par application, etc.
  - ❖ Possibilité de produire des applications indépendantes et réutilisables  
surcharge des templates, etc.

# Comment ça marche ?

- ❖ Tout projet est basé sur un design pattern Model View Template
  - ❖ Un template est un fichier HTML presque normal
  - ❖ Un modèle est une classe Python :
    - 1 modèle  $\Leftrightarrow$  1 table
    - 1 attribut  $\Leftrightarrow$  1 champ dans la table
  - ❖ Une vue est une fonction python permettant de mettre en forme des données
    - 1 instance de modèle + 1 template  $\rightarrow$  1 page HTML



# Hello world

Chapitre 1

# Créer un projet

- ❖ Création de l'arborescence du projet :
  - ❖ manage.py
  - ❖ projet/settings.py
  - ❖ projet/urls.py            (les Views)

```
django-admin.py startproject pokemon
```

# Démarrer le projet

```
manage.py createsuperuser
```

```
manage.py runserver
```

- ❖ Création d'un superuser
- ❖ Démarrage d'un serveur HTTP sur <http://localhost:8000>
- ❖ Interface d'administration accessible sur  
<http://localhost:8000/admin>

# Bilan !

- ❖ Nous avons un projet Django nommé pokemon
- ❖ Il est accessible, via manage.py runserver, sur <http://localhost:8000>
- ❖ Sauf que... Un projet Django est composé d'applications et nous n'en avons toujours pas

# Notre première application

```
manage.py startapp store
```

- ❖ Création d'une application nommée store à la racine de notre arborescence
- ❖ Vérifiez que l'application apparaisse dans settings.py

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'store',  
]
```

# Notre première application : créer une vue

- ❖ Modifions le fichier `views.py` de l'application pour créer notre première ..?

```
from django.shortcuts import render
from django.http import HttpResponse

def startPage(request):
    response = """<html>
        <head></head>
        <body>
            <h1>Hello worldz</h1>
            </body>
        </html>"""

    return HttpResponse(response)
```

# Notre première application : l'associer à une URL

- ❖ Modifions le fichier `urls.py` du projet pour lier notre vue à une URL :

```
from django.conf.urls import url
from django.contrib import admin
from store import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^accueil$', views.startPage),
]
```

*store est considéré comme un package du fait de l'existence du fichier `__init__.py`*

*Le fait que ce fichier vide existe, python permet donc son utilisation au même titre que tout package python*

# Bilan !

- ❖ Nous avons créé un projet, composé d'une seule application
- ❖ Cette application décrit une vue (statique)
- ❖ Cette vue est accessible via l'url <http://localhost:8000/accueil>



Move it

Chapitre 2

# Dynamisons ça : paramétrer les vues

- ❖ Il est possible de décrire des urls sous la forme de **regex**
- ❖ Les groupes de nos regex peuvent ensuite être utilisées par nos vues

```
url(r'^accueil/(\w+)$', views.secret)
```

- ❖ Ici, on passe un mot en paramètre
- ❖ Mot qui devra être récupéré en argument de notre vue :  
**view.secret(request, mot)**

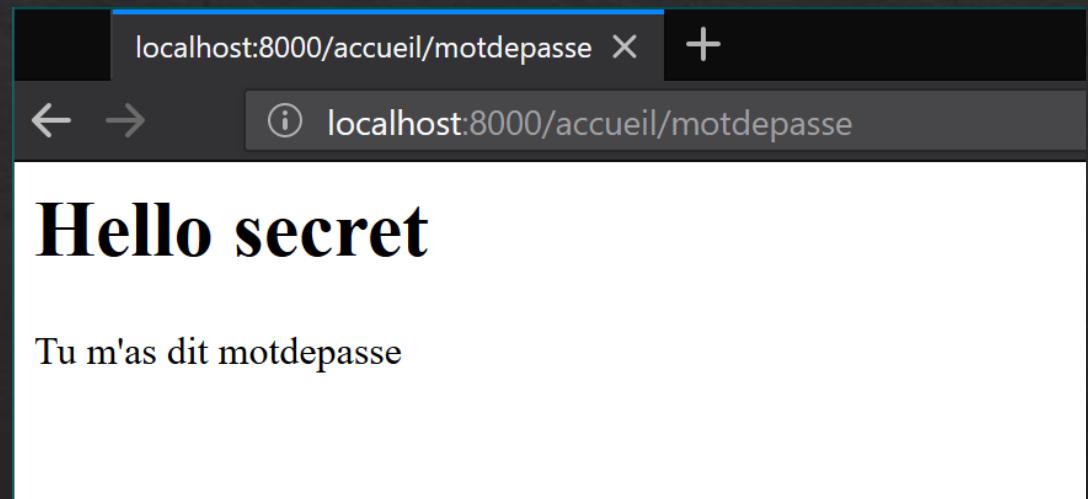
# Dynamisons ça : paramétrer les vues

urls.py

```
urlpatterns = [
    # ...
    url(r'^accueil/(\w+)$', views.secret),
]
```

store/secret.py

```
def secret(request, word):
    response = """<html>
        <head></head>
        <body>
            <h1>Hello secret</h1>
            <p>Tu m'as dit {0}</p>
        </body>
    </html>""".format(word)
    return HttpResponse(response)
```

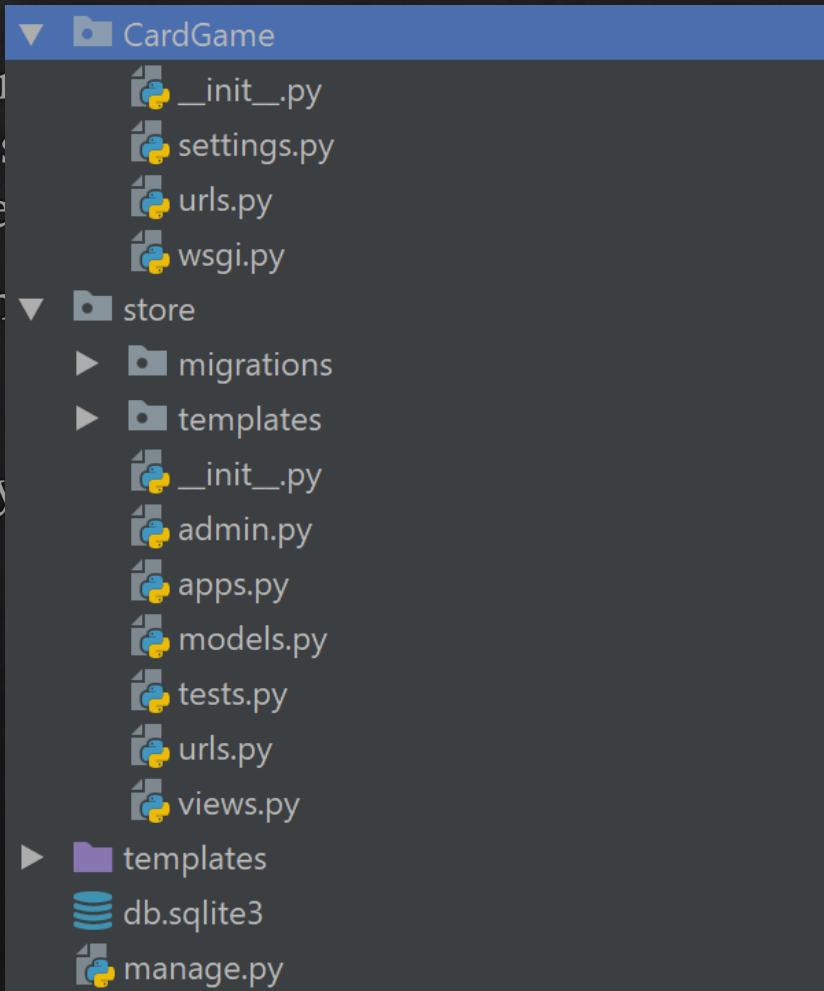


# Remarque

- ❖ Cette méthode de gestion des templates est intéressante pour la compréhension des mécanismes Django
- ❖ Cela étant, il n'est pas souhaitable de créer les templates comme des chaînes de caractères au sein du code pour des raisons évidentes...

# Organiser ses templates

- ❖ Par convention, nous allons créer un dossier nommé `templates` à l'entrée de notre projet.
- ❖ Ce dossier contiendra tous les templates communs aux différentes applications de notre projet.
- ❖ Nous allons aussi créer un dossier nommé `templates` dans chaque application du projet.
- ❖ Tous les fichiers que l'on y placera seront des fichiers HTML : pas de Python ici.



# Organiser ses templates

- ❖ Nous allons créer un premier template spécifique à notre application store qui nous souhaitera la bienvenue dans une langue donnée :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Bienvenue sur le sto
</head>
<body>
    <h1>{{ welcomeText }}</h1>
    <p>
        Lang : {{ lang }}
    </p>
</body>
</html>
```

*Ce template invoque deux variables :*

- *welcomeText*
- *Lang*

*Il faudra donc les définir avant de renvoyer le template*

# Organiser ses templates

- ❖ Maintenant, il faut créer la vue qui va définir nos deux variables et retourner le template configuré :

```
from django.shortcuts import render

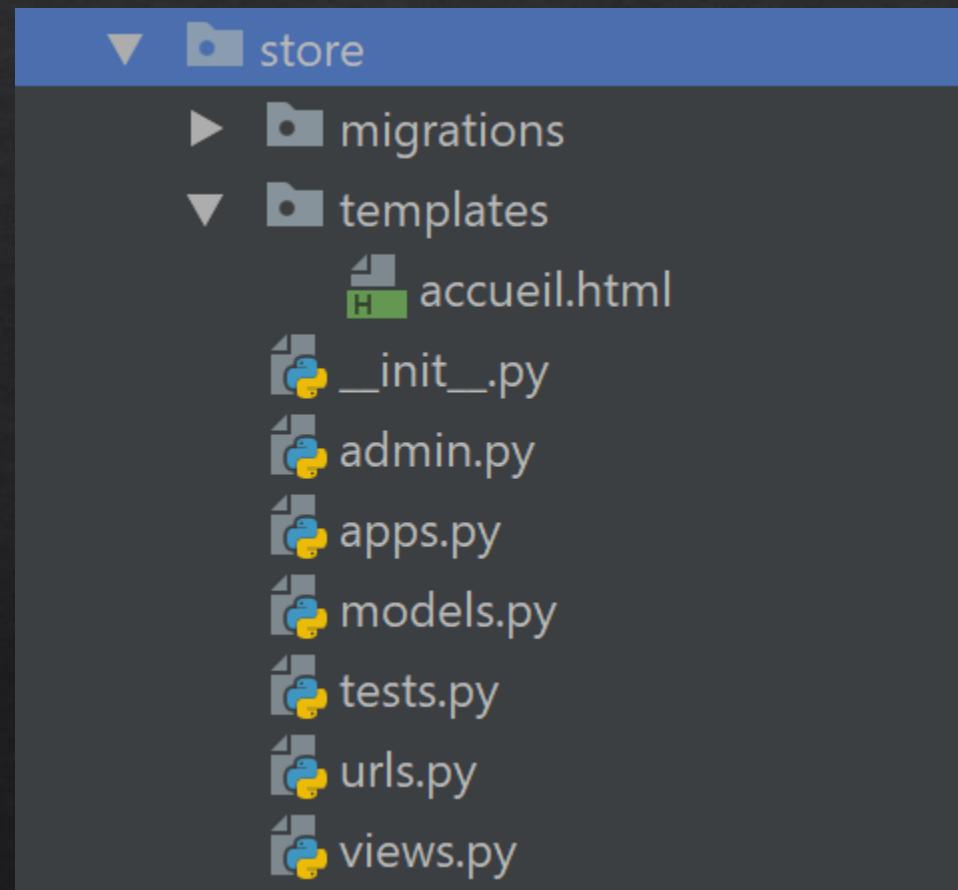
def welcome(request, lang):
    if lang == 'fr':
        welcomeText = 'Bienvenue'
    elif lang == 'en':
        welcomeText = 'Welcome'
    elif lang == 'de':
        welcomeText = 'Wilkommen'
    else:
        welcomeText = '??????'

    return render(request, 'accueil.html', locals())
```

*On utilise ici une nouvelle méthode de l'API Django : render() Elle prends trois paramètres :*

- *La requête initiale*
- *Le nom/chemin du template à afficher*
- *Le dictionnaire des variables à intégrer au template, ici locals() signifie que l'on travaille sur les variables locale à notre vue (lang et welcomeText)*

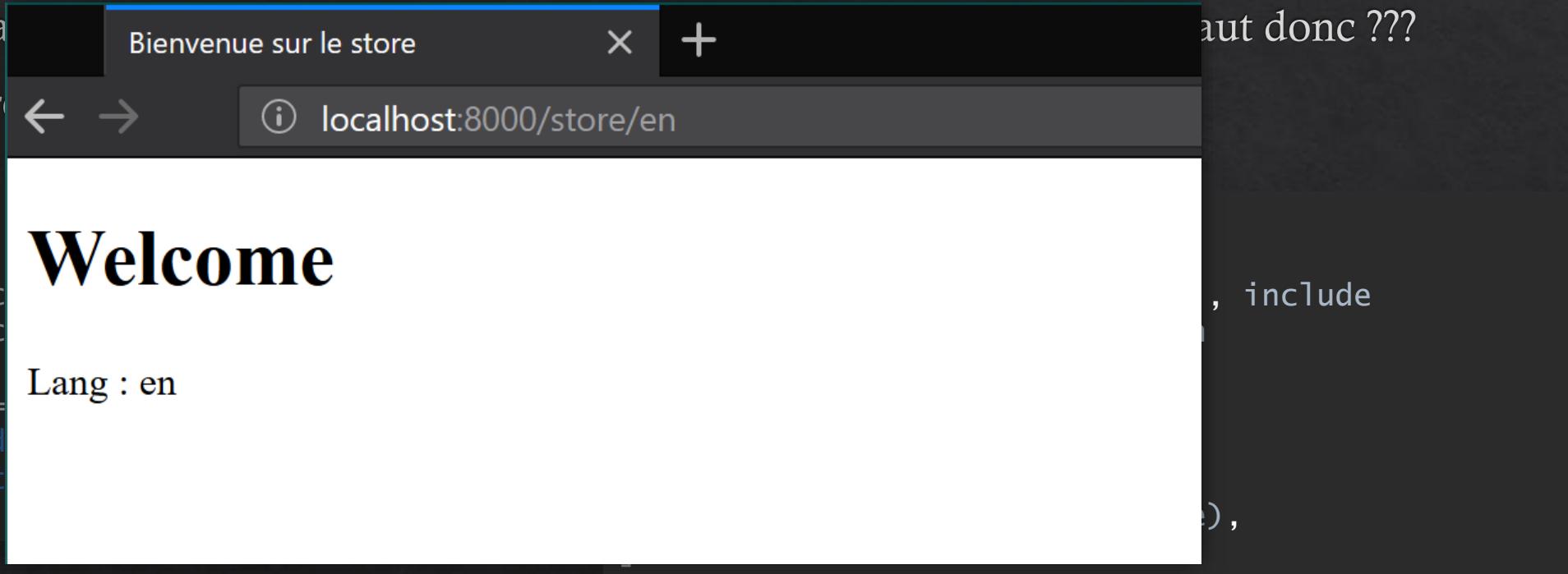
# Organiser ses templates



- ❖ L’arborescence de notre application devrait ressembler à ça
- ❖ Les fichiers de l’application qui nous intéressent sont ici :
  - ❖ templates/accueil.html
  - ❖ views.py
  - ❖ urls.py
- ❖ Nous allons aussi travailler sur le paramétrage de notre application avec le fichier ../settings.py pour pouvoir utiliser nos templates avec la méthode render()

# Organiser ses templates

- ❖ Notre template
- ❖ Créer notre r

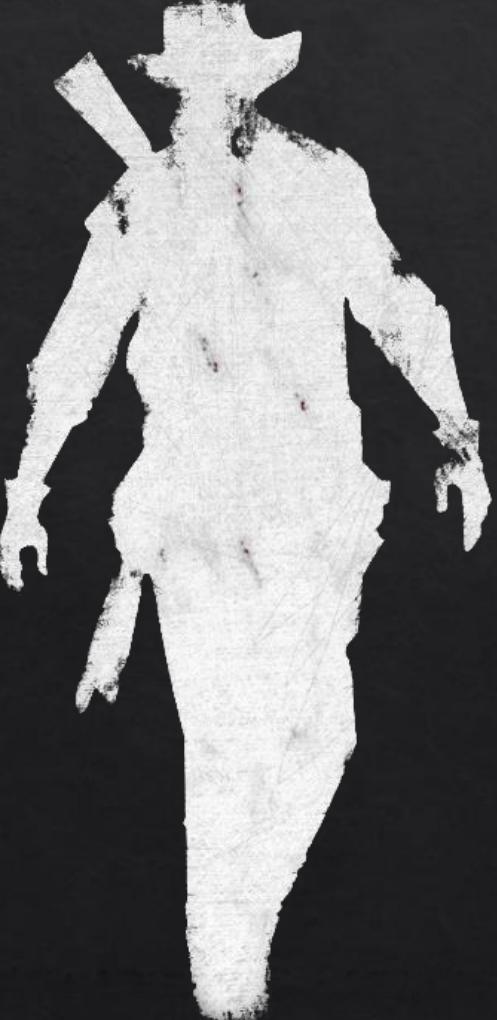


# La syntaxe des templates (30 minutes)

- ❖ Afficher conditionnellement une partie de votre template (en fonction de l'heure par exemple, ou d'un paramètre que votre vue reçoit)
  - ❖ utilisation des opérateurs not, and, or, >, <, ==, in
- ❖ Répéter un bout de template plusieurs fois (utilisation des boucles)

Documentation claire sur :

<https://overiq.com/django/1.10/template-tags-in-django>



# ???, vues, templates

Chapitre 3

Ressources utiles

<https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-django/les-modeles-8>

<https://overiq.com/django/1.10/basics-of-models-in-django>

store\_welcome [db.sqlite3] ×

<Filter criteria>

	id	welc...	lang
1	1	Undefined	new
2	2	Undefined	fs
3	3	Undefined	ps
4	4	Undefined	ps
5	5	Bonjour	accueil
6	6	Bonjour	lol
7	7	Bonjour	lol
8	8	Bonjour	DBa
9	9	Bonjour	a
10	10	Bonjour	a
11	11	Bonjour	DBa
12	12	Bonjour	a
13	13	Bonjour	a

Database

+

db.sqlite3

Schemas...

main

- auth\_group
- auth\_group\_permissions
- auth\_permission
- auth\_user
- auth\_user\_groups
- auth\_user\_user\_permissions
- django\_admin\_log
- django\_content\_type
- django\_migrations
- django\_session
- sqlite\_master
- sqlite\_sequence
- store\_welcome

- ❖ Nouvelles bases de données
- ❖ L'ensemble des tables de la base de données
- ❖ TIP : Utilisez la fonctionnalité "qui" pour trouver ce que vous recherchez.

# Les modèles

- ❖ Nous travaillons par défaut avec SQLite  
(déjà configuré dans settings.py)
- ❖ Vous pouvez installer et configurer  
une connexion MySQL ou PostgreSQL par exemple



# Configurer une base MySQL

- ❖ Dans settings.py, trouvez la variables DATABASES et remplacez la par ce contenu, adapté à votre connexion MySQL :

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'CardGame',  
        'USER': 'Login',  
        'PASSWORD': 'incassable',  
        'HOST': '127.0.0.1',  
        'PORT': '',  
    }  
}
```



# Les modèles

- ◊ Chaque application Django possède un fichier models.py
- ◊ Chaque classe, héritant de models.Model se trouvant dans ces fichiers conduira à la création d'une table identifiée par le nom du module et celui de cette classe :

store/models.py

```
from django.db import models
class Welcome(models.Model):
    welcomeText = models.CharField(max_length=50)
    lang = models.CharField(max_length=2)
```

*L'héritage en Python est  
explicite comme ceci*

conduira à la construction d'une table `store_welcome`, composée de deux champs :

- ◊ `welcomeText`
- ◊ `lang`

# Les modèles

- ❖ De cette manière, nous avons créé un modèle `welcome` pour l'application `store`
- ❖ A ce stade, il est nécessaire de faire appel à de la commande `migrate` pour mettre à jour de notre base de données

```
manage.py makemigrations [store]  
manage.py migrate
```

*Tips :*

*Le fait de spécifier un nom d'application (sans les crochets) permet de nous assurer que `makemigrations` a bien détecté notre application*

*Voir la déclaration des `INSTALLED_APPS` dans `settings.py` pour s'en assurer*

- ❖ `makemigrations` permet d'identifier tous les modèles de toutes nos applications (et de vérifier les modifications par rapport aux versions précédentes éventuelles) `store` est ici optionnel et permet de focaliser le calcul sur une seule application
- ❖ `migrate` réalise effectivement la migration

# Les modèles

- ❖ Après exécution de ces deux commandes (et un peu de débogage...) on devrait avoir une table store\_welcome effectivement créée dans notre base de données
- ❖ Assurez-vous en !
  
- ❖ Maintenant, ce serait pas mal d'écrire dedans <3

# Les modèles

- ❖ Pour cela, travaillons salement :

- ❖ Reprenez votre fichier `store/views.py`
- ❖ On va insérer en base la langue que l'utilisateur envoie si ce n'est ni `fr`, ni `en`, ni `de` :

```
from django.shortcuts import render
```

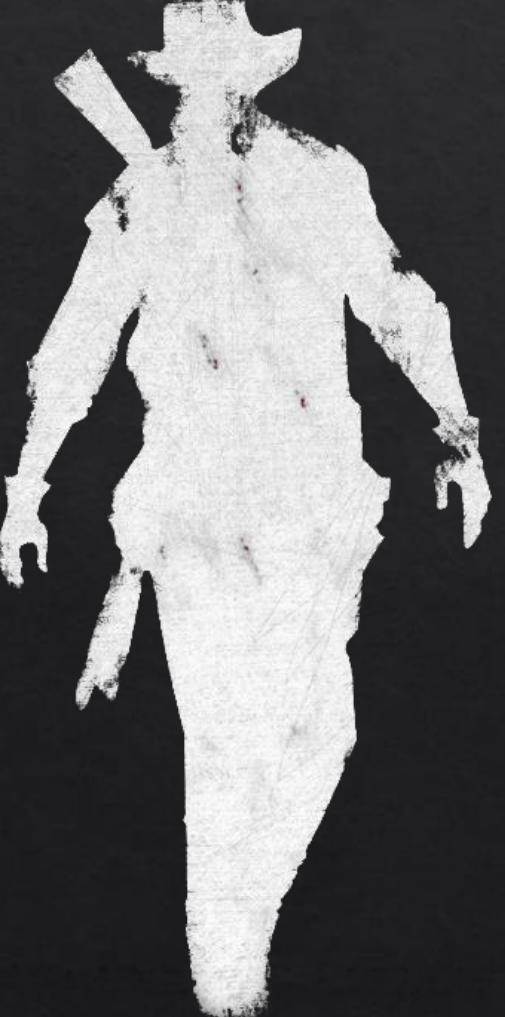
```
def welcome(request, lang):  
    if lang == 'fr':  
        welcomeText = 'Bienvenue'  
    elif lang == 'en':  
        welcomeText = 'Welcome'  
    elif lang == 'de':  
        welcomeText = 'Willkommen'  
    else:  
        from store.models import Welcome  
        welcomeInstance = Welcome(lang=lang)  
        welcomeInstance.save()  
        return render(request, 'newlang.html', locals())  
    return render(request, 'accueil.html', locals())
```

Jamais d'import sauvage de ce type dans ce cours...

Faire un insert dans une vue d'affichage... Mouai

# Bilan !

- ❖ Toute instance d'un modèle est enregistrable en base à l'aide la méthode save()
- ❖ Save réalisera un insert ou un update selon le critère suivant :
  - ❖ Si un id est défini (évalué à True par Python)
    - ❖ Si l'update est possible (ie. que l'ID existe déjà en base :  
**UPDATE**)
    - ❖ Sinon  
**INSERT**
  - ❖ Sinon : **INSERT**



# Les modèles 2 : récupérer des informations

Chapitre 3 et demi

# Accéder aux enregistrements

- ❖ Chaque modèle dispose d'une sous classe `objects`
- ❖ Cette sous classe fournit tous les outils nécessaires à la manipulation des données :
  - ❖ Sélection
  - ❖ Filtres
  - ❖ Ordonnancement
  - ...
- ❖ Pour récupérer tous les enregistrements :

```
from store.models import Welcome
for message in Welcome.objects.all():
    print(str(message))
```

# Accéder aux enregistrements

- ❖ Le résultat de Model.objects.all() est directement supporté dans un template :

```
from store.models import Welcome
resultSet = Welcome.objects.all()
```

```
<table>
  {% for message in resultSet %}
    <tr>
      <td>{{ message.lang }}</td>
      <td>{{ message.welcomeText }}</td>
    </tr>
  {% endfor %}
</table>
```

# Avec des filtres ?

- ❖ La sous classe objects propose notamment la méthode `filter()` (équivalent de la clause SQL WHERE)
- ❖ Il est possible de filtrer sur tous les attributs décrits dans le modèle, séparés par des virgules
- ❖ Si on reprend notre exemple précédent :

```
from store.models import Welcome
resultSet = Welcome.objects.filter(lang=lang)
```

```
from store.models import Welcome
resultSet = Welcome.objects.exclude(lang=lang)
```

# Avec des filtres plus élaborés ?

- ❖ Pour le moment, nous ne pouvons faire de la récupération que sur des égalités
- ❖ `filter()` permet la production de filtres plus élaborés au prix d'une syntaxe dont vous vous ferez une opinion :

```
Welcome.objects.filter(FOREIGNTABLE__FOREIGNFIELD=value)
```

```
Welcome.objects.filter(lang__startswith='f')
```

- ❖ Il existe de nombreux mécanismes permettant une recherche poussée dans les champs  
Source : <https://docs.djangoproject.com/fr/1.11/topics/db/queries/#field-lookups>

# Les models et la base de données (60 minutes)

- ❖ Reprenez la vue welcome pour une gestion totalement en base de données (sans le if du début)
- ❖ Créer des modèles faisant appel à des clés étrangères par exemple, un model Author et un model Articles (un article étant rédigé par un auteur)
- ❖ Créer diverses vues :
  - ❖ Informations détaillées sur un auteur
  - ❖ Liste des auteurs (par ordre Alphabétique, ou autre)
  - ❖ Articles complet
  - ❖ Liste d'articles « raccourcis »
- ❖ Filtrer les articles par mots clés, par titre, etc...



# Le backoffice : intro

Chapitre 4

# En deux mots

- ❖ De base Django offre un espace d'administration pas trop mal
- ❖ Il est normalement accessible via <http://localhost:8000/admin>
- ❖ Son accès est restreint aux utilisateurs identifiés « membres de l'équipe » ou « Staff status »
- ❖ Il offre par défaut le support des utilisateurs et des groupes
- ❖ Il est extensible et paramétrable pour supporter toutes nos applications

# Je découvre l'admin Django (10 minutes)

- ❖ Créer en console un super user si ce n'est pas déjà fait  
**manage.py createsuperuser**
- ❖ Se connecter avec cet utilisateur dans l'administration et :
  - ❖ Créer un utilisateur sans droit particulier
  - ❖ Créer un admin (utilisateur ayant tous les droits)
  - ❖ Créer un membre du staff (pouvant se connecter à l'administration)
  - ❖ Créer un ou plusieurs groupes, avec des droits pas piqués des hanetons
  - ❖ Affecter un groupe à l'utilisateur sans droit particulier
  - ❖ ...

# Les droits

- ❖ Au fur et à mesure que l'on crée des applications et des modules, les droits disponibles dans l'administration lors de la création des groupes augmentent
- ❖ Ils ont toujours la forme :  
`<application> | <model> | <add|change|remove>`

# Administre tes models

- ❖ Passé l'émoi de cette rencontre avec notre backoffice, on aimerait pouvoir en faire un peu plus
- ❖ Dans votre application, créez un fichier admin.py :

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

```
admin.site.register(Model)
```

*Importez le ou les modèles à administrer  
Puis enregistrez les Un à un comme site de  
l'administration*

- ❖ Il faut que vos models soient cohérents avec votre base de données

# Administre tes models

- ❖ Connectez-vous à votre backoffice et vous aurez quelque chose comme ça :

The screenshot shows the Django administration interface. At the top, a blue header bar displays "Django administration". Below it, a white main area starts with "Site administration". A blue navigation bar labeled "AUTHENTICATION AND AUTHORIZATION" contains links for "Groups" and "Users", each with "Add" and "Change" buttons. At the bottom of this bar is a "STORE" button. To the right, a dark callout box contains the text: "Vos models enregistrés apparaissent regroupés par application dans votre administration".

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups      + Add      Change

Users      + Add      Change

STORE

Attacks

Vos models enregistrés apparaissent regroupés par application dans votre administration

# Administre tes models

- ❖ Lorsque vous cliquez sur un model, vous avez un CRUD pas trop dégueu :
  - ❖ lister toutes ses instances en base de données ( + filtrer + trier)
  - ❖ afficher des formulaires de création/modification tout faits
  - ❖ Supprimer vos enregistrements

# Administre tes models

Django administration

Home › Store › Effects

Select effect to change

Action:  Go 0 of 3 selected

EFFECT  
 Effect object  
 Effect object  
 Effect object

3 effects

Django administration

WELCOME, ADMI

Home › Store › Effects › Effect object

Change effect

Name: Leader

Strength: 10

Type: Magie

Delete

Save and add another

# Je gère l'admin Django (10 minutes)

# L'inconvénient du jour

- ❖ C'est très figé et pas si exploitable que ça :
  - ❖ les libellés sont pas explicites
  - ❖ Les formulaires de création/modification pas paramétrables ?
- ❖ Jusqu'à l'arrivée des modelDescriptors...

# Les model descriptors\*

- ❖ Les **models**, pour bénéficier de l'aspect ORM doivent hériter de **models.Model**
- ❖ Les **modelDescriptors** eux, doivent hériter de **admin.ModelAdmin**
- ❖ Ils permettent de personnaliser assez finement notre CRUD pour chaque model

Un exemple ?

\* Ce mot n'existe pas officiellement dans le jargon Django, mais il fait bien

# Personnalisons l'affichage de notre liste

```
models.py
class Attack(models.Model):
    name = models.CharField(max_length=50)
    strength = models.IntegerField()
    type = models.CharField(max_length=50)

    def __str__(self):
        return self.name + ' (' + str(self.strength) + ')'
```

*On surcharge ici la méthode  
« `toString()` » de notre classe `Attack`,  
on verra pourquoi*

```
models.py
class AttackAdmin(admin.ModelAdmin):
    # Attributs à afficher dans la liste de notre administration
    list_display = ['name', 'strength', 'type']
    # Permet de n'afficher que la/les catégories qui nous intéressent
    list_filter = ['type']
    # Tri par défaut (peut porter sur plusieurs colonnes
    ordering = ['name']
```

```
admin.py
admin.site.register(Attack, AttackAdmin)
```

# Personnalisons l'affichage de notre liste

Django administration

WELCOME, **ADMIN**. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home › Store › Attacks

Select attack to change

Action:   0 of 3 selected

ATTACK

Attack object

Attack object

Attack object

3 attacks

[ADD ATTACK +](#)

# Personnalisons l'affichage de notre liste

Django administration

WELCOME, **ADMIN**. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home › Store › Attacks

Select attack to change

Action:  Go 0 of 3 selected

<input type="checkbox"/>	NAME	ordering	STRENGTH	TYPE
<input type="checkbox"/>	Brise matinale		9	Lumière
<input type="checkbox"/>	La fureur du chaton		12	Meow
<input type="checkbox"/>	Poney qui tousse		7	Virus

3 attacks

**list\_filter**

**ADD ATTACK +**

**list\_display**

**FILTER**

By type

- All
- Lumière
- Meow
- Virus

# Personnalisons l'affichage des formulaires

- ❖ On peut personnaliser les champs à afficher dans ses formulaires (côté admin) en spécifiant un champ attribut **fields** ou **fieldsets** dans notre model Descriptor :

```
class AttackAdmin(admin.ModelAdmin):  
    list_display = ['name', 'strength', 'type']  
    list_filter = ['type']  
    ordering = ['name']  
  
    # Personnalisation des champs du formulaire  
    fieldsets = (  
        ('Caractéristiques générales', {  
            'description': 'Nom de l\'attaque',  
            'fields': ['name']}),  
        ('Spécificités', {  
            'description': 'Description technique',  
            'fields': ['strength', 'type']}))
```

# Personnalisons l'affichage des formulaires

Django administration

WELCOME, AD

Home > Store > Attacks > Add attack

Add attack

Name:

Strength:

Type:

[Save and add another](#)

Django administration

WELCOME, ADMI

Home > Store > Attacks > Add attack

Add attack

**Caractéristiques générales**

Nom de l'attaque

Name:

**Spécificités**

Description technique

Strength:

Type:

[Save and add another](#)

# Personnalisons l'affichage des formulaires

- ❖ Petite note du jour :
  - ❖ Les contraintes de validations s'appliquent automatiquement
  - ❖ Si vous masquez certains champs ➔ ils doivent être optionnels

# Je domine l'admin Django (20 minutes)

- ❖ Personnalisez des listes de vos models ainsi que vos formulaires de création/ modification :
  - ❖ personnalisation des colonnes affichées dans la liste
  - ❖ champs de filtrage
  - ❖ champs d'ordre par défaut
  - ❖ champs de recherche
  
- ❖ personnalisation des champs affichés dans vos formulaires
- ❖ décomposition en fieldsets



# Backoffice & relations

Chapitre 4-5

# Elargir ses modèles

- ❖ Dans les faits, la majorité de nos modèles ne se limitera pas à une table
- ❖ On distingue deux cas de relations :
  - ❖ Les clés étrangères (OneToMany)
  - ❖ Les jointures ManyToMany
- ❖ Django admin intègre un support à ces mécanismes

# Clés étrangères

- ❖ Créons un model décrivant une clé étrangère sur un autre model :

**models.py**

```
class Card(models.Model):  
    name = models.CharField(max_length=50)  
    category = models.IntegerField()  
    lifePoints = models.IntegerField()  
    magicalPoints = models.IntegerField()  
    defensePoints = models.IntegerField()  
    specialEffect = models.CharField(max_length=50)  
    attack = models.ForeignKey('Attack')
```

**models.py**

```
class CardAdmin(admin.ModelAdmin):  
    list_display = ['name', 'category', 'lifePoints', 'magicalPoints', 'attack']  
    list_filter = ['category', 'attack']  
    ordering = ['name']  
    search_fields = ['name', 'specialEffect']
```

**admin.py**

```
admin.site.register(Card, CardAdmin)
```

# Clés étrangères

Django administration

WELCOME, AD

Home › Store › Cards › Canarenplastik

Change card

Name:

Category:

LifePoints:

MagicalPoints:

DefensePoints:

Attack:

Delete

Save and add another

The screenshot shows a Django admin change form for a 'Cards' model. The card has the name 'Canarenplastik', category '2', 12 life points, 900 magical points, and 900 defense points. The attack field contains 'La fureur du chaton (12)' and includes a dropdown arrow, a pencil icon, and a plus icon. At the bottom, there are 'Delete' and 'Save and add another' buttons.

- ❖ Notre dernier champ propose une liste déroulante de toutes les attaques connues dans la table store\_attack
- ❖ Il est possible d'en créer, d'en modifier et enfin, d'en sélectionner une
- ❖ **Note** : le texte apparaissant dans la liste pour chaque attaque résulte de l'appel à la méthode `__str__` que l'on a redéfini

# ManyToMany

- ❖ Il est possible de déclarer dans un model un ou plusieurs attributs de type ManyToMany
- ❖ Comme pour les clés étrangères, assurez-vous au préalable de l'existence du model et de la table (ici on se base sur un model **Effect**)

```
class Card(models.Model):  
    name = models.CharField(max_length=50)  
    category = models.IntegerField()  
    lifePoints = models.IntegerField()  
    magicalPoints = models.IntegerField()  
    defensePoints = models.IntegerField()  
    specialEffect = models.CharField(max_length=50)  
    attack = models.ForeignKey('Attack')  
    effects = models.ManyToManyField(Effect)  
  
def __str__(self):  
    return self.name
```

# ManyToMany

- ❖ En ajoutant dans notre modelDescriptor le champ effects que nous venons de créer, on obtient un attribut fieldsets de ce type :

```
fieldsets = (
    ('Caractéristiques générales', {
        'description': 'Caractéristiques principales de la carte',
        'fields': ['name', 'category', 'lifePoints', 'magicalPoints', 'defensePoints']
    }),
    ('Attaques et effets de la carte', {
        'description': 'Attaques',
        'fields': ['attack', 'effects']
    })
)
```

# ManyToMany

Django administration

WELCOME, AD

Home › Store › Cards › Canarenplastik

Change card

Caractéristiques générales

Caractéristiques principales de la carte

Name: Canarenplastik

Category: 2

LifePoints: 12

MagicalPoints: 900

DefensePoints: 900

Attaques et effets de la carte

Attaques

Attack: La fureur du chaton (12)

Effects:

- Disparition (1000)
- Illumination (-100)
- Leader (10)

oneToMany

manyToMany

Hold down "Control", or "Command" on a Mac, to select more than one.

Delete

Save and add another

- ❖ Si tout s'est bien passé, vous pourriez avoir quelque chose qui ressemble à ça.
- ❖ Les relations oneToMany sont matérialisées par des listes déroulantes (1-1 choix)
- ❖ Les relations manyToMany sont matérialisées par des listes combo (0-n choix)

# PGM de l'admin Django (20 minutes)

- ❖ Créez deux models indépendants  
l'un sera utilisé en Foreignkey, l'autre en jointure MTM
- ❖ Créez un troisième model référencant les deux premiers
- ❖ Créez un modelDescriptor pour chacun d'eux
- ❖ Amusez-vous :
  - ❖ créez/modifiez des instances
  - ❖ affectez-les comme clé étrangère ou jointure MTM
  - ❖ ...



# Parler à son backend

Chapitre 5

# Les forms Django

- ❖ Django intègre deux mécanismes permettant la création de formulaires dans `forms.py`
  - ❖ Création de formulaires from scratch héritant de `forms.Form`
  - ❖ Création de formulaires depuis un modèle héritant de `forms.ModelForm`

```
class Product(forms.Form):  
    """ Formulaire décrit de manière exhaustive """  
    name = forms.CharField(label="Nom du produit", max_length=150)  
    weight = forms.CharField(label="Poids en Kg", max_length=3)  
    description = forms.CharField(label="Description", widget=forms.Textarea, required=False)  
    available = forms.BooleanField(label="Votre adresse mail")
```

```
class Card(forms.ModelForm):  
    """ Formulaire basé sur un modèle existant """  
class Meta:  
    model = Card  
    exclude = [] # ou fields[] pour inclure explicitement des champs
```

# Les forms Django

## Réponse

Nom du produit:

Poids en Kg:

Description:

Votre adresse mail:

## Fiche produit

Name:

Category:

LifePoints:

MagicalPoints:

DefensePoints:

SpecialEffect:

Attack:

Effects:

# Les forms Django

*Notre token CSRF, convertit en HTML*

```
<form action="addproduct" method="post">
    <input type='hidden' name='csrfmiddlewaretoken'
value='XB1FpVGhgUTyMoPVEAwpT46JbLbiYhisVQKyQ5mRZvv4QCa2Lktkr5mSRY4u76vu' />
    <p>
        <label for="id_name">Nom du produit:</label>
        <input type="text" name="name" maxlength="150" required id="id_name" />
```

*Nos autres champs*

```
<p>
    <label for="id_weight">Poids en Kg:</label>
    <input type="text" name="weight" maxlength="3" required id="id_weight" />
</p>
<p>
    <label for="id_description">Description:</label>
    <textarea name="description" cols="40" rows="10" id="id_description"></textarea>
</p>
<p>
    <label for="id_available">Votre adresse mail:</label>
    <input type="checkbox" name="available" required id="id_available" />
</p>
    <input type="submit" value="Submit" />
</form>
```

# Les forms Django

- ❖ Chaque application peut contenir son propre fichier forms.py
- ❖ On peut ensuite y décrire tous les formulaires que l'on souhaite utiliser  
Ceux-ci doivent bien hériter des classes forms.Form ou forms.ModelForm
- ❖ On peut ensuite instancier nos formulaire au sein de nos views :

```
def addproduct(request):
    from .forms import Model
    form = Model(request.POST or None)

    if form.is_valid():
        # faire des trucs si le contenu du formulaire est ok
        return render(request, 'addproductconfirm.html', locals())
    else:
        # faire des trucs si l'on veut afficher le formulaire à l'utilisateur
        return render(request, 'addproductform.html', locals())
```

# Les forms Django

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</head>
<body>
    <h1>Fiche produit</h1>
    <form action="addproduct" method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

*Ici, on passe l'URL qui pointera vers notre vue, mais on pourrait directement spécifier la vue*

- ❖ Il faut ensuite décrire dans notre/nos template(s) l'intégration de notre formulaire
  - ❖ intégration d'un token CSRF
  - ❖ Intégration du formulaire
  - ❖ L'attribut action du <form> doit renvoyer sur l'URL / la view à même de traiter le POST

# Les forms Django

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
  </head>
  <body>
    <h1>Fiche produit</h1>
    <form action="addproduct" method="post">
      <input type="text" name="name"/>
      <input type="text" name="category"/>
      ...
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

*Pourquoi pas décrire tout ça directement en full HTML ?  
Et pourquoi ne pas envoyer une requête HTTP directement en Js*

- ❖ Inclure un form Django dans un template permet principalement d'adapter automatiquement vos formulaires au fur et à mesure que vos models évoluent
- ❖ Cela étant, vous pouvez décrire vos formulaires HTML tout à fait classiquement

Quoi qu'il en soit, on va devoir traiter le contenu de ces formulaires !

# Les forms Django

*Instanciation du form, soit avec le contenu de notre POST (ou d'un GET selon), soit vide*

```
def addCard(request):
    from .forms import CardForm
    form = Card(request.POST or None)

    if form.is_valid():
        name = form.cleaned_data['name']
        weight = form.cleaned_data['weight']
        description = form.cleaned_data['description']
        available = form.cleaned_data['available']

    return render(request, 'cardFeedback.html',
                  locals())
```

- ❖ Lorsque l'on envoie un formulaire, il existe deux manières d'en extraire les informations dans notre view
- ❖ La première consiste à instancier notre form, initialisé avec les attributs de request.POST
- ❖ On peut alors accéder de manière propre aux données via la méthode cleaned\_data() de notre formulaire

# Les forms Django

```
def addproduct(request):
    from .forms import CardForm
    from .models import Card
    form = CardForm(request.POST)

    if form.is_valid():
        name = request.POST.get("name")
        attack = request.POST.get("attack")
        card = Card(name=name)
        card.save()

    return render(request, 'confirm.html', locals())
```

*On peut utiliser directement les attributs décrits dans request.POST pour traiter le contenu du formulaire*

- ❖ La deuxième méthode consiste à manipuler directement le contenu de notre requête (request.POST si notre formulaire HTML est de type POST, GET sinon)
- ❖ Quelque soit la méthode utilisée, on peut alors instancier notre model selon les informations envoyées, et l'enregistrer en base par exemple

# Bilan !

L'envoie de données sur le serveur peut se faire de manière variées

- ❖ La méthode la plus classique impose
  - ❖ Emission d'une requête HTTP POST ou GET comme vous savez le faire (formulaire, AJAX, etc.)
  - ❖ Manipulation du contenu POST et/ou GET de l'objet request au sein d'une view dédiée
- ❖ La seconde consiste à utiliser les outils Django :
  - ❖ Déclaration exhaustive d'un formulaire dans une classe héritant de forms.Form ou Déclaration automatisée dans une classe héritant de forms.ModelForm
  - ❖ Création d'un template permettant son inclusion
  - ❖ Instanciation du modèle correspondant depuis le formulaire au sein d'une view dédiée



REST in peace

Chapitre 6

# Une API REST ?

- ❖ Permet de manipuler de manière très structurée à votre base de données
- ❖ Généralement, via une URL du type :
  - ❖ <http://monurl/<tableOuModel>>  
pour manipuler tous les éléments de votre collection
  - ❖ <http://monurl/<tableOuModel>/<id>>  
pour manipuler un élément de votre collection
- ❖ Basée sur les méthodes HTTP :
  - ❖ GET = récupérer un élément ou une liste d'éléments
  - ❖ POST = créer un élément
  - ❖ PUT = mettre à jour un élément ou une liste d'éléments
  - ❖ DELETE = supprimer un élément ou une liste d'éléments

# Django et REST : DRF

- ❖ Commençons par installer Django REST Framework :

```
pip install djangorestframework
```

- ❖ Maintenant, un œil dans settings.py :

```
INSTALLED_APPS = [  
    ...  
    'rest_framework',  
    'myapi',  
]
```

# Maintenant, un petit modèle ou deux

models.py

```
class PokemonAttack(models.Model):
    name = models.CharField(max_length=250)
    strength = models.IntegerField()

    def __str__(self):
        return self.name + ' : ' + str(self.strength)

class Pokemon(models.Model):
    name = models.CharField(max_length=250)
    hp = models.IntegerField()
    type = models.CharField(max_length=250)
    attacks = models.ManyToManyField(PokemonAttack)

    def __str__(self):
        return self.name + ' (' + str(self.type) + ')'
```

*Tips :*

*Enregistrez votre/vos modèle(s) dans le backoffice pour tester plus facilement votre API*

# Retourner nos objets : les serializers

- ❖ Un serializer permet de transformer notre objet Python (certainement un dict) et un format plus praticable par notre scripts Js : du Json
- ❖ On pourrait produire du XML à la place, si on avait travaillé avec SOAP par exemple

serializer.py

```
from rest_framework import serializers
from .models import Pokemon

class PokemonSerializer(serializers.ModelSerializer):
    class Meta:
        model = Pokemon
        exclude = []
```

*En héritant de ModelSerializer, votre serializer bénéficiera des outils dont nous avons besoin pour la production de réponses satisfaisantes dans notre API*

# Manipuler notre serializer dans notre view

- ❖ Nous devons produire une view indiquant les méthodes HTTP supportées (ici GET et POST)  
Cela se fait avec des **décorateurs**
- ❖ Cette view doit ici récupérer une collection, la sérialiser et la retourner :

```
from rest_framework.decorators import api_view
from rest_framework import status
from rest_framework.response import Response
from .models import Pokemon
from .serializer import PokemonSerializer

@api_view(['GET', 'POST'])
def pokemonList(request):
    if request.method == 'GET':
        pokemons = Pokemon.objects.all()
        serializer = PokemonSerializer(pokemons, many=True)
        return Response(serializer.data)
```

*Les décorateurs sont des concepts Python (pas spécifiquement Django). Il s'agit de fonctions prenant en paramètre la fonction décrite après et lui affecte un nouveau comportement*

`@decorateur(args)`  
`myFunc ...`  
*Equivaut à :*  
`myFunc = decorateur(myFunc, args)`

# Et une seconde view

- ❖ On récupère toute la liste, mais on voudrait pouvoir en récupérer un seul, et pourquoi pas le supprimer :

```
@api_view(['GET', 'DELETE'])
def pokemonIdentity(request, name=None, id=None):
    if request.method == 'GET':
        if name:
            pokemon = Pokemon.objects.get(name=name)
        else:
            pokemon = Pokemon.objects.get(id=id)

    serializer = PokemonSerializer(pokemon, many=False)
    return Response(serializer.data)

    elif request.method == 'DELETE':
        if name:
            pokemon = Pokemon.objects.get(name=name)
        else:
            pokemon = Pokemon.objects.get(id=id)
        pokemon.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

# Nos URLs

- ❖ En résumé :
  - ❖ Nous configuré notre API, produit nos models, notre view orientée API
  - ❖ Il nous faut maintenant linker tout ça à des URLs :

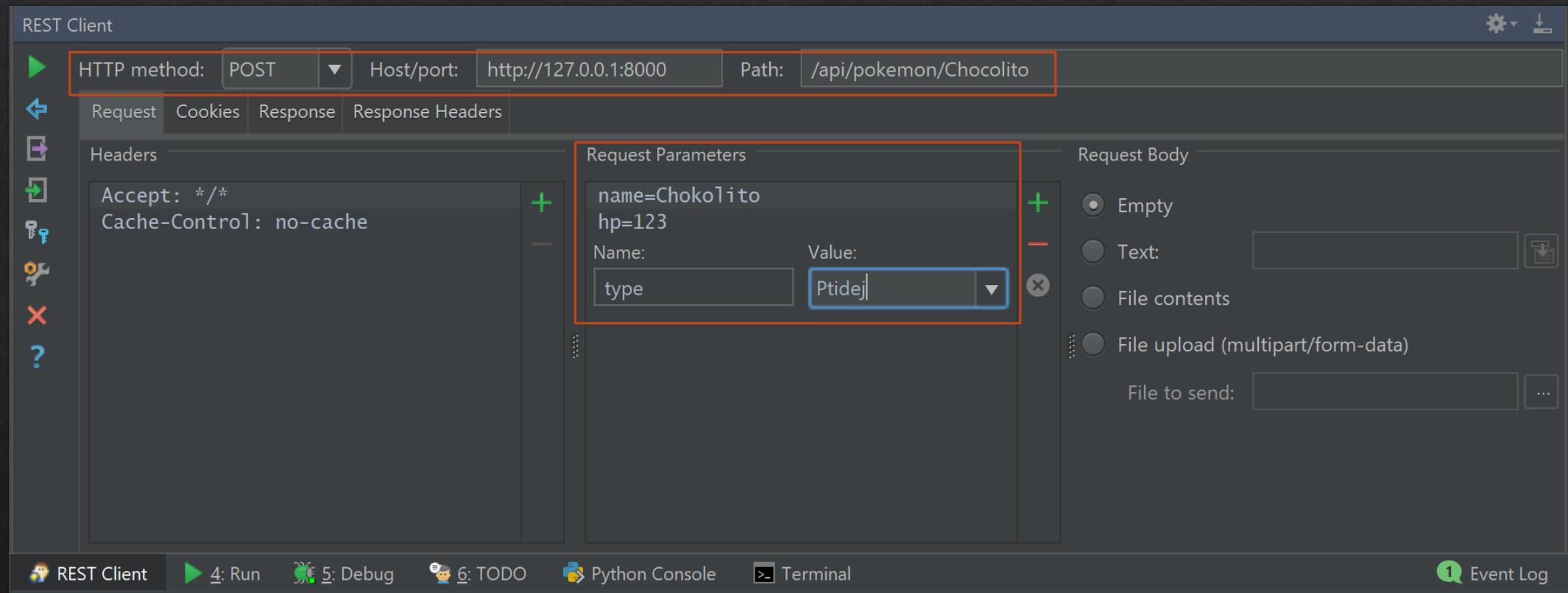
```
from django.conf.urls import url, include
from . import views

urlpatterns = [
    url(r'^pokemon$', views.pokemonList),
    url(r'^pokemon/(?P<name>[A-z]+)$', views.pokemonIdentity),
]
```

# Utiliser notre API !

- ❖ Etat actuel de notre API :
  - ❖ **api/pokemon**  
GET : retourne l'ensemble de nos pokemon
  - ❖ **api/pokemon/<nom>**  
GET : retourne le pokemon dont le nom est stipulé en paramètre  
DELETE : suppriemr le pokemon dont le nom est stipulé en paramètre
  - ❖ **api/pokemon/<id>**  
GET : retourne le pokemon dont l'id est stipulé en paramètre  
DELETE : suppriemr le pokemon dont l'id est stipulé en paramètre
- ❖ On requête !

# Le client REST de votre IDE ?



# Le client REST de Django

The screenshot shows a browser window with the URL `127.0.0.1:8000/api/pokemon` in the address bar. The page title is "Django REST framework". On the right, there is a user icon labeled "admin". The main content area is titled "Pokemon List". It features a "GET /api/pokemon" button and "OPTIONS" and "GET" buttons. Below these buttons, the response status is "HTTP 200 OK" and the headers are "Allow: POST, GET, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body is a JSON array representing a list of Pokemons, starting with the first entry:

```
[{"id": 1, "name": "Pikanartichaut", "hp": 120, "type": "Oignon", "attacks": [{"name": "Attaque Oignon", "power": 50}, {"name": "Attaque Oignon", "power": 50}, {"name": "Attaque Oignon", "power": 50}], "abilities": [{"name": "Abilit\u00e9 Oignon"}, {"name": "Abilit\u00e9 Oignon"}]}
```

# On découvre les API (40 minutes)

- ❖ Installez DjangoRestFramework
- ❖ Créez une API :
  - ❖ retournant tous les éléments d'une collection
  - ❖ retournant un élément spécifique
  - ❖ supprimant un ou tous les éléments

# Poster des infos

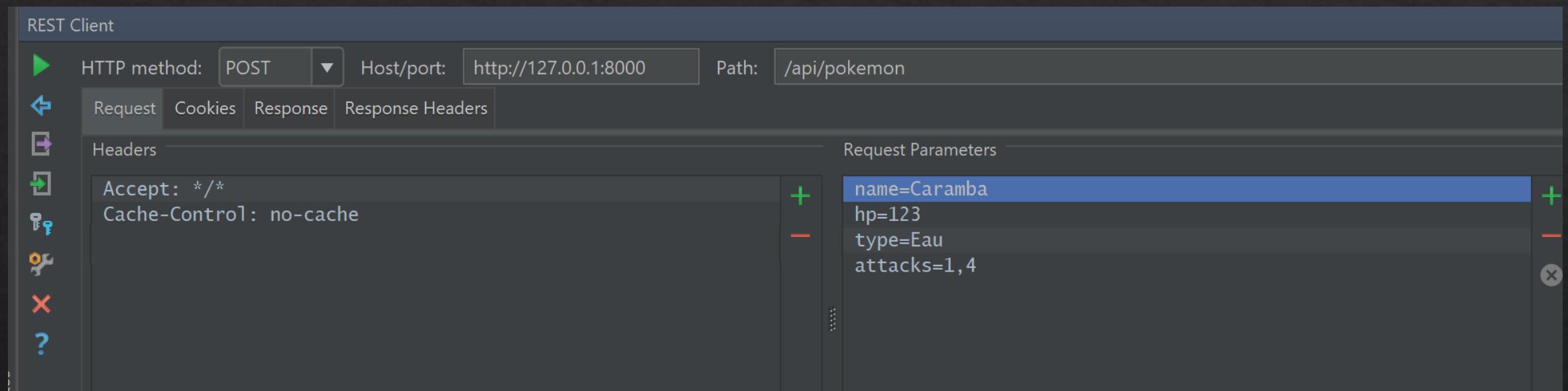
- ❖ Nous avons appris à lire et supprimer des données, maintenant on écrit

```
@api_view(['GET', 'POST'])
def pokemonList(request):
    if request.method == 'GET':
        pokemon = Pokemon.objects.all()
        serializer = PokemonSerializer(pokemon, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        postData = {
            'name': request.data.get('name'),
            'hp': request.data.get('hp'),
            'type': request.data.get('type'),
            'attacks': request.data.get('attacks').split(',')
        }
        serializer = PokemonSerializer(data=postData)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

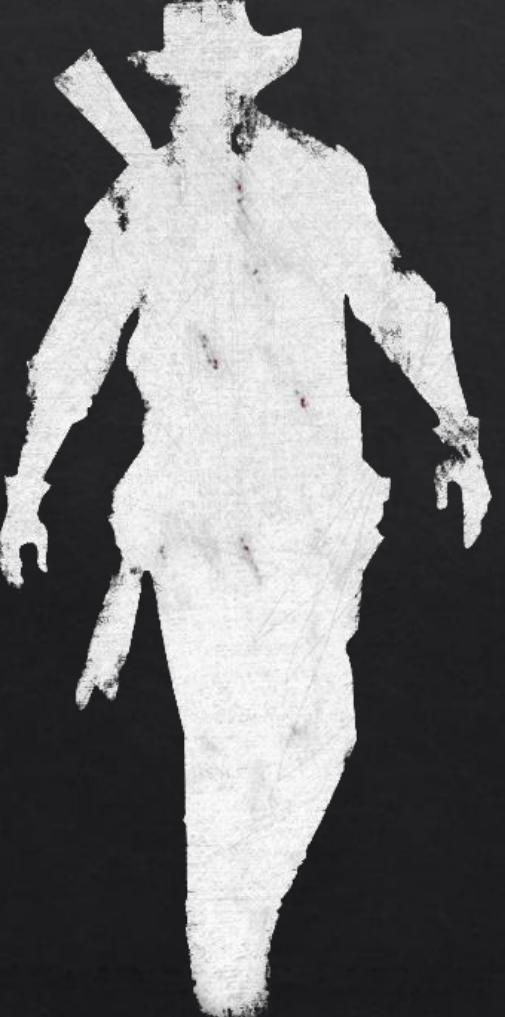
# Poster des infos

- ❖ Tester notre API



# On poste (20 minutes)

- ❖ On fait évoluer notre API pour poster des données
- ❖ Pourquoi pas du PUT pour faire des mises à jour ?



# REST et l'authentification

Chapitre 6-5

# Les tokens utilisateur

- ❖ On déclare notre mécanisme d'authentification dans settings.py

```
INSTALLED_APPS = [
    ...
    'myapi',
]

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
    )
}
```

# Les tokens utilisateur

- ❖ Dans models.py, ajouter les imports permettant de supporter les tokens et mécanismes d'authentification :

```
from django.db import models
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token
from django.conf import settings
```

*Il faudra créer un nouveau supersuer pour qu'il bénéficie d'un token*

```
# This code is triggered whenever a new user has been created and saved
@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

# Les tokens d'authentification

- ❖ Il faut à présent bénéficier d'un token pour pouvoir exécuter des requêtes
- ❖ Cela se configure dans notre fichier urls.py via l'utilisation de views.obtain\_auth\_token fournie par DRF

```
from django.conf.urls import url, include
from rest_framework.authtoken import views as apiViews
from . import views
```

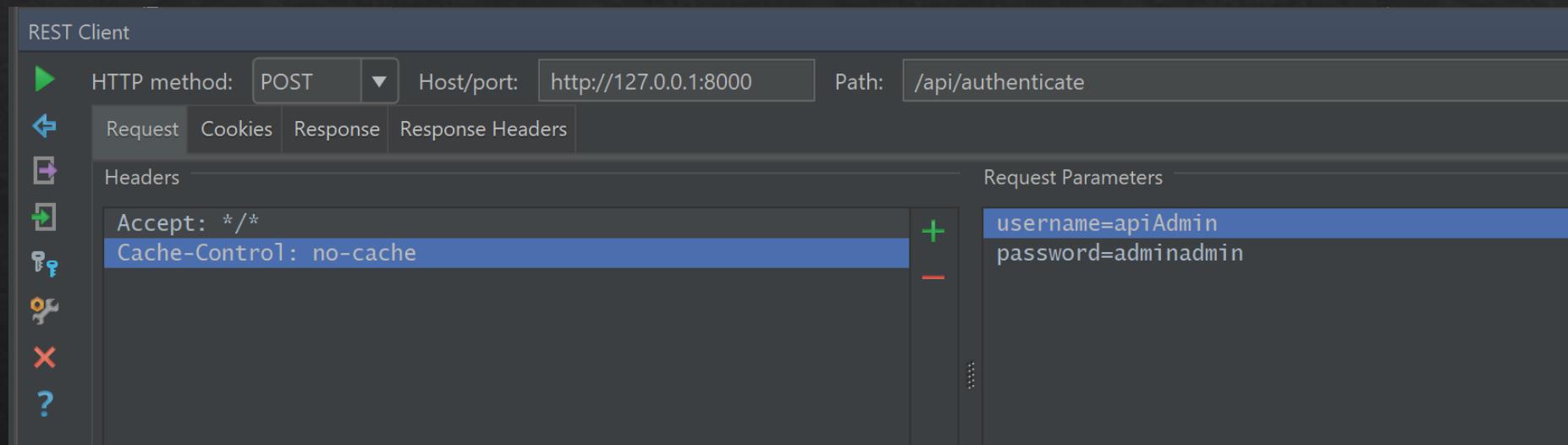
```
urlpatterns = [
    url(r'^pokemon$', views.pokemonList),
    url(r'^pokemon/(?P<name>[A-z]+)$', views.pokemonIdentity),
    url(r'^pokemon/(?P<id>[0-9]+)$', views.pokemonIdentity),

    url(r'^authenticate$', apiViews.obtain_auth_token)
]
```

*Utilisant deux packages views, on spécifie un alias pour l'un d'eux*

# Les tokens d'authentification

- ❖ Nous venons de décrire une URL permettant d'obtenir un token nécessaire à l'exécution des requêtes suivantes
- ❖ Cette requête de méthode **POST**, doit décrire deux variables
  - ❖ **username** pour le login
  - ❖ **password** pour le mot de passe

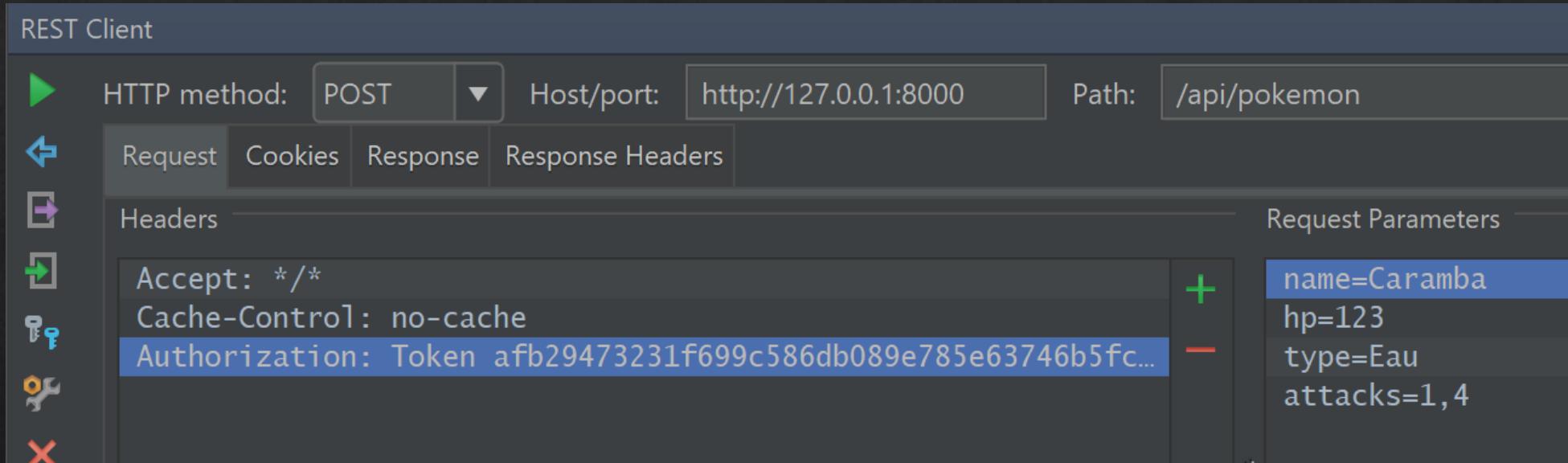


# Les tokens d'authentification

- ❖ La réponse, si tout se passe bien, est simple et de la forme suivante :

```
{"token": "afb29473231f699c586db089e785e63746b5fc24"}
```

- ❖ Charge à vous de stocker ce token et de stipuler dans le head de vos requêtes suivantes sous la forme **Authorization: Token votretoken**



# On sécurise (30 minutes)

- ❖ Sécuriser l'accès à notre API (`settings.py`)
- ❖ Vérifier que vous n'y avez plus accès sans authentification
- ❖ Créer un nouveau superuser
  - ❖ Récupérer son token
  - ❖ Envoyer des requêtes avec cet utilisateur



# Les web sockets

Chapitre 7

# Un framework qui marche : Channels

- ❖ Django intègre une couche de gestion des requêtes HTTP
- ❖ Il est nécessaire pour un framework websocket qu'il vienne se positionner en amont, ou à côté de ce serveur HTTP
- ❖ Les messages sont traités en FIFO
- ❖ La queue est stockée dans un layer

# Un framework qui marche : Channels

- ❖ Django intègre une couche de gestion des requêtes HTTP
- ❖ Il est nécessaire pour un framework websocket qu'il vienne se positionner en amont, ou à côté de ce serveur HTTP
- ❖ Les messages sont traités en FIFO

Commençons par installer Channels

# Channels : installation / configuration

- ❖ Sous Windows, Channels requiert Visual C++ Build Tools  
<http://landinghub.visualstudio.com/visual-cpp-build-tools>
- ❖ On l'installe :  
`pip install -U channels`

# Channels : installation / configuration

- ❖ Le **channel layer** décrit le « driver » de notre queue et la façon dont les messages sont traités

```
CHANNEL_LAYERS = {
    "default": {
        # Gestion du layer en RAM, ne permet pas un
        # fonctionnement ASYNC/Cross process
        "BACKEND": "asgiref.inmemory.ChannelLayer",
        "ROUTING": "sockets.routing.channel_routing",
    }
}
```

# Dans les grandes lignes

- ❖ Un **Consumer** est l'équivalent d'une View Django
- ❖ Il s'agit d'une fonction :
  - ❖ appelée depuis notre routeur
  - ❖ effectuant des traitements
  - ❖ retournant une réponse au client
- ❖ On parle de Consumer dans le sens où ces fonctions consomment les messages qui s'accumulent dans la queue

# Dans les grandes lignes

- ❖ Un Channel est similaire dans son utilisation d'une URL Django
- ❖ Il permet de lier un type de demande utilisateur à un Consumer
- ❖ Pourtant conceptuellement, un Channel et une URL ne se situent pas du tout au même niveau :
  - ❖ Toutes les URLs Django prennent place dans le Channel HTTP Request

# Un exemple parle plus que de mauvaises diapos

- ❖ Créons une application qui accepte les connexions websockets et fournit trois services :
  - ❖ connexion
  - ❖ diffusion d'un message
  - ❖ déconnexion
- ❖ Pour cela, nous allons créer **trois consumers**,  
**un routeur** les référençant  
et un petit script **Js websockets** pour y accéder

# Notre fichier consumers.py

```
from django.http import HttpResponse
from channels.handler import AsgiHandler
from channels import Group

def ws_connect(message):
    # On doit stipuler que la connexion est acceptée
    message.reply_channel.send({"accept": True})
    # On abonne l'utilisateur à un groupe
    Group("msnmessenger").add(message.reply_channel)

    Diffuser seulement vers l'émetteur

def ws_message(message):
    Group("msnmessenger").send({
        "text": "Quelq'un a dit " + message.content['text'],
    })

    La notion d'abonnement à un/des groupe(s) est essentielle à la diffusion de messages à plusieurs personnes

def ws_disconnect(message):
    # Désabonnement du groupe
    Group("msnmessenger").discard(message.reply_channel)
```

# Notre fichier routing.py

```
from channels.routing import route
from sockets.consumers import ws_connect, ws_message, ws_disconnect

channel_routing = [
    route("websocket.connect", ws_connect),
    route("websocket.receive", ws_message),
    route("websocket.disconnect", ws_disconnect),
]
```

# Notre fichier test.js

```
socket = new WebSocket("ws://localhost:8000");
```

*On travaille en websockets natives*

```
socket.onopen = function() {
    /* Méthode appelée juste après la connexion */
    socket.send("* Patou vient de se connecter");
};
```

```
socket.onmessage = function(e) {
    /* A la réception d'un message */
    console.log(Date.now() + ' - ' + e.data);
};
```

```
if (socket.readyState === WebSocket.OPEN) {
    /* On spamme quelques messages et on part */
    socket.send("Lu sava?");
    socket.send("Ya kkun?");
    socket.send("....");
    socket.send("Bsx");
    socket.close();
}
```

# On communique (30 min – 4 h...)

- ❖ Installer Channels (et les dépendances nécessaires)
- ❖ Monter quelques consumers et quelques routes
- ❖ Monter un petit client Js

# WebSockets Js

- ❖ La méthode send() prend en paramètre une chaîne de caractère
- ❖ Il est possible de produire un Json que l'on serialize :

```
if (socket.readyState === WebSocket.OPEN) {  
    data = {  
        "username": "John",  
        "password": 'doe'  
    };  
    socket.send(JSON.stringify(data));  
    socket.close();  
}
```

# WebSockets Js

- ❖ La méthode send() prend en paramètre une chaîne de caractère
- ❖ Il est possible de produire un Json que l'on serialize :

```
def ws_message(message):
    import json
    obj = json.loads(message.content['text'])

    if "action" not in obj:
        message.reply_channel.send({"text": 'Nope'})

    elif obj['action'] == 'join':
        Group("msn").send({
            "text": "* " + obj['username'] + " a rejoint la conversation",
        })
```

# Bilan

- ❖ Le mécanisme de consumers/routing de Channels est très similaire aux views/urls de Django à l'usage
- ❖ Il est très simple de démarrer une connexion via WebSockets à l'aide de ces outils
- ❖ Les messages qui transitent doivent être des chaînes de caractères  
Il faut donc sérialiser/désérialiser vos objets manuellement pour communiquer des données structurées