

Django

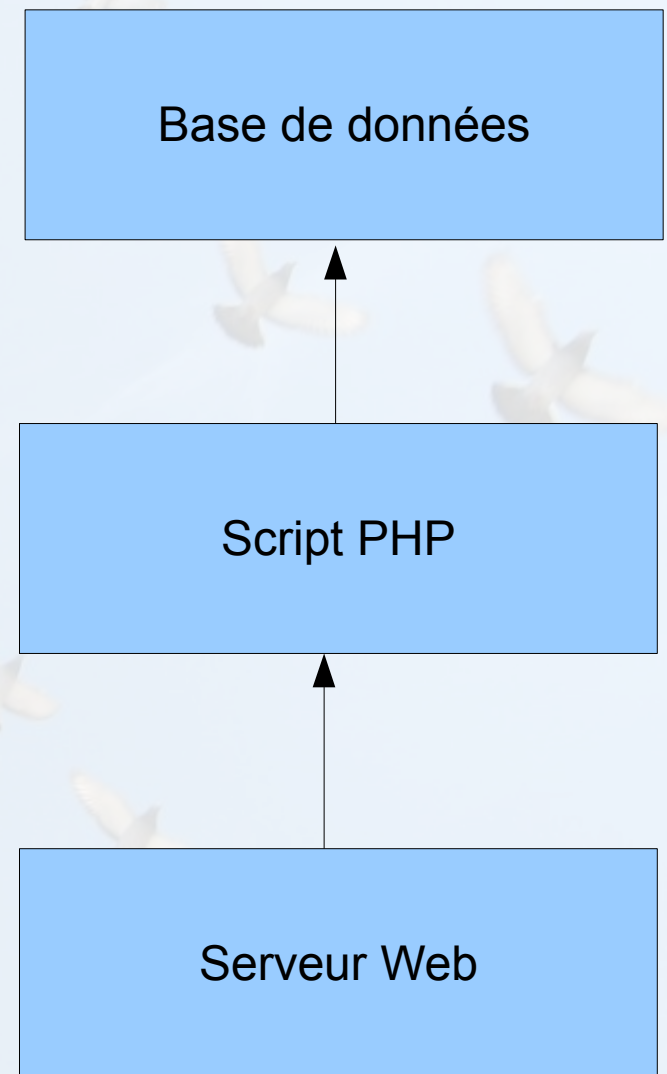
Framework de développement Web

Disclaimer

- Je ne suis pas un expert Python
- Je ne suis pas un expert Django
- Expérience basée sur le développement de deux petits sites en Django
 - Trivialibre
 - MapOSMatic
- Objectif
 - Inciter des développeurs à investir du temps dans la compréhension d'un framework
 - Donner les premières pistes

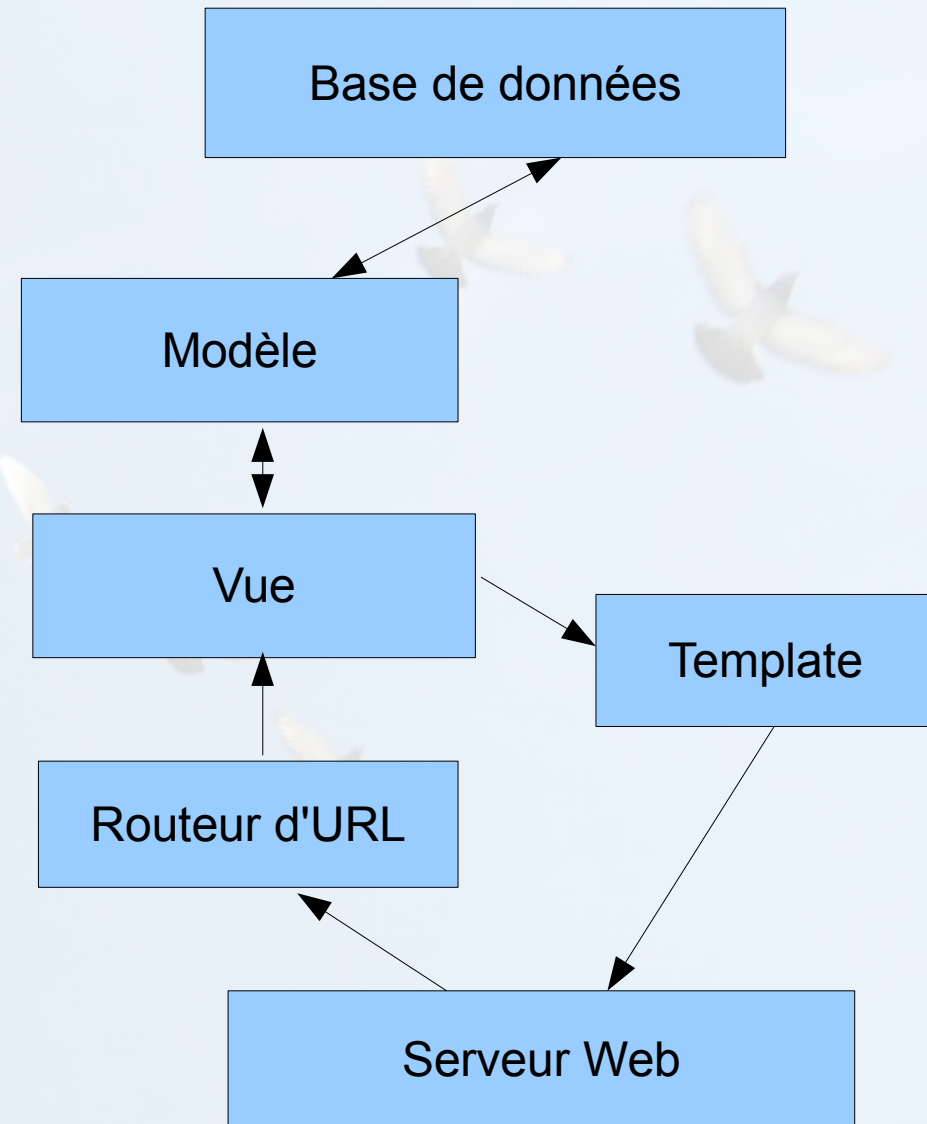
Développement Web sur le métal

- Une URL = un script
- Le script, linéairement
 - analyse la requête
 - consulte une base de données
 - produit un résultat HTML
 - retourne ce résultat au serveur
- Pas d'infrastructure pour le développement de l'application Web



Framework Web

- Infrastructure qui va faciliter le développement de l'application Web
 - Décomposer l'application en plusieurs couches claires
 - Offrir des solutions à des problématiques courantes
- Attention: un framework n'est pas un CMS !



Investir dans un framework

- Oui, apprendre un framework Web est initialement un peu plus long et plus compliqué que d'apprendre « seulement » le PHP
- Mais oui, investir du temps pour comprendre et apprendre un framework fera gagner du temps et de la qualité
 - Même pour un développeur Web « du dimanche »
- Une fois l'étape de compréhension franchie, le retour en arrière vers du PHP brut-de-pomme fait mal tellement c'est bien le framework :-)

Django

- Framework de développement Web en Python
- Avec Turbogears, probablement l'un des deux frameworks Python les plus populaires
- Choisi parce que
 - Python
 - Populaire
 - La documentation semblait pas mal
- Pas regardé/testé les autres frameworks
 - Semblent assez similaires dans les principes
- Principe de Django : ça marche par défaut, mais on peut personnaliser ensuite

Installation

- Se présente sous la forme
 - De modules Python
 - D'une commande `django-admin`, qui permet de faire différents opérations d'administration sur un projet
- Sur toutes les bonnes distributions
 - `apt-get install python-django`
- Et on est prêt à commencer !

Notre projet

- Pour illustrer le propos, nous allons ce soir construire une application d'inscription à des évènements
 - On peut lister les évènements, s'y inscrire, voir la liste des inscrits
 - On peut ajouter, modifier, supprimer des évènements
- Notre projet s'appellera « events »

Création d'un projet

- Pour créer un projet :
 - `django-admin startproject events`
- Va créer un répertoire `events` qui contient
 - `__init__.py`, le fichier classique pour être vu comme un module Python. Ici, il est vide.
 - `manage.py`, un script exécutable qui va nous permettre de gérer notre projet. C'est tout simplement un `django-admin` qui connaît notre projet
 - `settings.py`, qui contient les paramètres de notre projet
 - `urls.py`, le routeur d'URLs

Test du projet

- Notre projet est déjà fonctionnel !
- Django intègre un petit serveur Web bien pratique en phase de développement
- Pour le lancer :
 - `./manage.py runserver`
- Puis, on pointe son navigateur Web sur
 - <http://localhost:8000>
- Django nous accueille !

Notion d'application

- À l'intérieur d'un projet, on peut avoir plusieurs « applications »
 - Chaque application peut être un bout du site Web (front-end, back-end, ou autre.)
 - Une application peut aussi être un composant utilisé par le site Web: django-registration (pour gérer l'enregistrement des utilisateurs), django-dmigrations (pour gérer les migrations de bases de données), etc.
- Pour notre projet, nous allons créer une seule application « `www` » pour le front-end
 - `./manage.py startapp www`
- Un répertoire `www/` créé avec : `views.py` pour les vues, `models.py` pour les modèles.

Configurons tout ça

- La configuration est stockée dans settings.py.
- Plein d'informations, et notamment
 - Des informations sur la base de données (variables DATABASE_*)
 - Où sont les fichiers de media (les fichiers statiques, CSS, images, etc.), variables MEDIA_ROOT et MEDIA_URL
 - Quel est le routeur d'URLs, ROOT_URLCONF
 - Quelles applications sont installées, INSTALLED_APPS
- Configurons la base de données
 - DATABASE_ENGINE='sqlite3'
 - DATABASE_NAME='events.db'
- Installation de notre application
 - Ajout de « www » à INSTALLED_APPS

Modèle

Le modèle représente le comportement de l'application : traitements des données, interactions avec la base de données, etc. Il décrit ou contient les données manipulées par l'application. Il assure la gestion de ces données et garantit leur intégrité. Dans le cas typique d'une base de données, c'est le modèle qui la contient.

Wikipédia.

Modèles dans Django

- Django utilise l'ORM (Object Relational Mapping) avec une définition très légère des objets et de la relation
- On crée des classes qui héritent de `django.db.models.Model`
- Chaque instance de cette classe va être sauvegardée en base de données
- Une table par classe, une entrée dans la table par objet
- Les attributs de la classe, définis grâce à des `models.XXXField()` sont les champs de la table
 - On peut ajouter ses propres types de champ
- Au niveau programmation, on ne fait que manipuler des objets Python
- Pas de SQL à écrire, sauf si des requêtes particulièrement pointues sont nécessaires

Définition des modèles

```
class Event(models.Model):  
    description = models.CharField(max_length=256)  
    start_date = models.DateTimeField()  
    end_date = models.DateTimeField()
```

```
class Participant(models.Model):  
    lastname = models.CharField(max_length=256)  
    firstname = models.CharField(max_length=256)  
    email = models.EmailField()  
    event = models.ForeignKey(Event)
```


Initialisation de la base de données

- Django va dériver automatiquement de la définition du modèle la création des tables nécessaires
 - `./manage.py syncdb` ~~syncdb~~ `makemigrations then ./manage.py migrate`
- L'application « auth » étant également configurée, création d'un compte d'administrateur
- Si de nouveaux modèles sont créés, « syncdb » créera les nouvelles tables
- Si des modèles existants sont modifiés
 - Il faut répercuter à la main le changement dans la base (ajout ou suppression de colonne ou de table)
 - Ou alors regarder les outils de migration de base

Jouons avec l'API du modèle (1)

- Le modèle de Django nous fournit une API Python
 - Qu'on peut utiliser dans l'application Web évidemment
 - Mais aussi dans des applications Python « classiques »
- `./manage.py shell`
 - `from eventswww.models import Event, Participant`
- Création d'un objet
 - `e = Event()`
 - `e.description = 'Qjelt'`
 - `e.start_date = '2009-11-26 20:00'`
 - `e.end_date = '2009-11-26 23:00'`
 - `e.save()`

Jouons avec l'API du modèle (2)

- Sélection des évènements
 - `Event.objects.all()` retourne la liste de tous les évènements
 - Définition de la méthode `__str__` pour avoir quelque chose de plus joli
- Ajout d'un nouvel évènement: Ubuntu Party
- Compter
 - `Events.objects.count()`
- Filtrer
 - `Events.objects.filter(description='Ubuntu Party')`
 - `Event.objects.filter(start_date__gte=datetime(2009,12,1))`
`import datetime`
`.datetime`

Jouons avec l'API du modèle (3)

- Récupérer un objet
 - `e = Event.objects.get(description='Ubuntu Party')`
 - `q = Event.objects.get(id=1)`
- Modification
 - `e.description = 'Ubuntu Party 2009'`
 - `e.save()`
- Suppression
 - `e.delete()`

Jouons avec l'API du modèle (4)

- On récupère un événement
 - `e = Event.objects.get(id=1)`
- On ajoute un participant
 - `p = Participant()`
 - `p.firstname = 'Richard'`
 - `p.lastname = 'Stallman'`
 - `p.email = 'rms@gnu.org'`
 - `p.event = e`
 - `p.save()`
- Pour lister les participants à un événement
 - `e.participant_set.all()`

Étendons le modèle (1)

- On peut ajouter des méthodes aux objets du modèle pour implémenter diverses opérations
- Nombre de participants dans un événement

```
def participant_count(self):  
    return self.participant_set.count()
```
- Liste des adresses e-mail des participants d'un événement

```
def participant_mail_list(self):  
    return ','.join(  
        [p.email for p in self.participant_set.all()])
```

Étendons le modèle (2)

- Envoi d'un e-mail aux participants

```
def participant_notify(self):  
    for p in self.participant_set.all():  
        send_mail(u'Votre participation à %s' % \  
                self.description,  
                u'Vous êtes inscrit à %s le %s' % \  
                (self.description, self.start_date),  
                'info@superevenements.org', [ p.email ])
```


Étendons le modèle (3)

- On peut également ajouter des méthodes au niveau de la classe du modèle
- Cela permet d'ajouter des fonctions de sélection des évènements
- Pour cela
 - On définit une classe qui hérite de `models.Manager`
 - On implémente nos méthodes dans cette classe
 - On crée une instance de ce Manager dans notre classe de modèle, en l'appelant « `objects` »

Étendons le modèle (4)

```
class EventManager(models.Manager):  
    def by_participant(self, email):  
        return  
        self.filter(participant__email__exact=email)
```

```
class Event(models.Model):  
    [...]  
    objects = EventManager()
```

Euh, on était pas venu entendre parler de Web ?

Routeur d'URL

- Lorsque les requêtes arrivent, elles sont dispatchées vers les fonctions de vue par l'intermédiaire du routeur d'URL
- Dans `urls.py`
- Il associe une expression régulière matchant l'adresse de la requête et une fonction de vue

```
from events.www.views import index, about,  
event_info  
  
urlpatterns = patterns('',  
    (r'^$', index),  
    (r'^about/$', about),  
    (r'^event/(?P<eid>\d+)/$', event_info),  
)
```

Vues

- Dans `views.py`
- Une fonction de vue est appelée lorsqu'une requête parvient au serveur Web
- Elle analyse la requête, effectue les traitements correspondants et renvoie une réponse
- Les fonctions de vue prennent en paramètre
 - `request`, qui contient toutes les informations sur la requête HTTP
 - Les paramètres matchés par l'expression régulière du routeur d'URL
 - Ici, `eid` dans le cas de `event_info`
- Elles doivent retourner un objet `HttpResponse`

Vue minimale

```
from django.http import HttpResponse

def index(request):
    return HttpResponse(u"<h1>Bienvenue</h1>");

def about(request):
    return HttpResponse(u"<h1>À propos</h1>");

def event_info(request, eid):
    return HttpResponse(u"Évènement %s" % eid);
```

Templates

- On ne souhaite pas mélanger le code HTML/CSS/Javascript avec le code Python qui traite les requêtes
- La fonction de vue génère le code HTML renvoyé au client grâce à :
 - Un template, qui contient le code HTML et des « utilisations » de variables
 - L'assignation de valeurs à une liste de variables qui seront remplacées dans le template
- Ces variables peuvent être des valeurs simples, des listes, des objets Python, etc.
- Permet de séparer la présentation du traitement

Templates coté vue

- La méthode « longue »

```
def mafonctiondevue(request):  
    t = loader.get_template('foo.html')  
    c = Context({  
        'bar': 42,  
        'acme' : "Bonjour",  
    })  
    return HttpResponse(t.render(c))
```

- La méthode « courte »

```
from django.shortcuts import render_to_response  
  
def mafonctiondevue(request):  
    return render_to_response("foo.html",  
        { 'bar': 42, 'acme': "Bonjour" })
```

Templates

- Définir le répertoire qui contient les templates
- `TEMPLATE_DIRS` dans `settings.py` contient une liste de chemins absolus
- Ajout dans `settings.py` de

```
import os.path
```

```
PROJECT_PATH = \  
    os.path.dirname(os.path.abspath(__file__))
```

```
TEMPLATE_DIRS = (  
    os.path.join(PROJECT_PATH, 'templates')  
)
```

Premier template

- templates/event_info.html

```
<html>
<head><title>Évènement {{ eventid }}</title></head>

<body>
<h1>Informations sur évènement {{ eventid }}</h1>
<p>Rien à dire.</p>
</body>
</html>
```

- www/views.py

```
def event_info(request, eid):
    return render_to_response("event_info.html",
                              { 'eventid' : eid })
```

Dans les templates

- Accéder à des variables passées par la vue
 - `{{ variable }}`
 - `{{ objet.champ }}`
 - `{{ dictionnaire.cle }}`
- Tags
 - Pour faire des tests
 - `{% if ... %} ... {% else %} ... {% endif %}`
 - Faire des boucles
 - `{% for foo in foo_list %} ... {% endfor %}`
- Filtres
 - Pour traiter les données
 - `{{ value|date:"D d M Y" }}`
- On peut écrire ses propres tags/filtres

Héritage de templates

- Pour partager du code entre templates
 - Typiquement l'ossature du site
- Un template hérite d'un autre en utilisant `{% extends "foo.html" %}`
- Le template hérité « appelle » des blocs `{% block machin %}{% endblock %}`
- Le template héritant « définit » le contenu de ces blocs `{% block machin %}Bidule{% endblock %}`

Template de base : base.html

```
<html>
<head>
<title>{% block title %}{% endblock %}</title>
<link rel="stylesheet" type="text/css" href="/media/style.css"
media="screen" />
</head>
<body>
<div id="header">
  
  <h1>Évènements</h1>
  <div id="menu">
    <ul id="nav">
      <li><a href="/">Accueil</a></li>
      <li><a href="/events">Évènements</a></li>
      <li><a href="/about">À propos</a></li>
    </ul>
  </div>
</div>
<div id="content">
  {% block contents %}{% endblock %}
</div>
</body></html>
```

Index : index.html

Template

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

```
{% block title %}
```

```
Accueil
```

```
{% endblock %}
```

```
{% block contents %}
```

```
<h1>Bienvenue</h1>
```

```
<p>Rien à dire</p>
```

```
{% endblock %}
```

Vue

```
def index(request):
```

```
    return render_to_response("index.html")
```


Média

- Les fichiers statiques, appelés « media », ne sont pas gérés par Django
- Si on utilise l'URL /media/, il faut que le serveur Web soit configuré pour servir le bon répertoire à cette adresse
- Attention au conflit avec ADMIN_MEDIA_PREFIX
- En développement, on peut utiliser la vue `django.views.static.serve`
 - `document_root` est le répertoire qui contient les fichiers de média
 - `path` est le fichier accédé

Média

urls.py

```
import settings
```

```
urlpatterns = patterns('',  
    [...]
```

```
(r'^media/(?P<path>.*)$', django.views.static.serve,  
    { 'document_root': settings.MEDIA_PATH,  
      'show_indexes': True } ),  
    [...]
```

```
)
```

settings.py

```
ADMIN_MEDIA_PREFIX = '/admin-media/'
```

```
MEDIA_PATH = os.path.join(PROJECT_PATH, "media")
```

Page « Liste des évènements »

- Ajoutons une page qui liste les évènements
- Accessible à l'adresse /event/
- Liée à la fonction de vue `event_list()`
 - Qui récupérera en base la liste des évènements
 - Et la passera à un template `event_list.html` pour affichage

Page « Liste des évènements »

urls.py

```
urlpatterns = patterns('',  
[...]  
    (r'^events/$', event_list),  
[...]  
)
```

views.py

```
def event_list(request):  
    el = Event.objects.all()  
    return render_to_response("event_list.html",  
        { 'eventlist' : el })
```

Page « Liste des évènements »

```
{% extends "base.html" %}
{% block title %}
Liste des évènements
{% endblock %}
```

```
{% block contents %}
<h1>Liste des évènements</h1>
```

```
<ul>
```

```
{% for event in eventlist %}
  <li><a href="/event/{{ event.id }}">
    {{ event.description }}</a>,
    du {{ event.start_date }}
    au {{ event.end_date }}</li>
```

```
{% endfor %}
```

```
</ul>
```

```
<a href="/event/add">Ajouter un évènement</a>
{% endblock %}
```

Page « Évènement »

- Utilisation du shortcut « `get_object_or_404` » qui récupère un objet en base et s'il n'est pas trouvé, affiche une erreur 404.

views.py

```
from django.shortcuts import get_object_or_404

def event_info(request, eid):
    e = get_object_or_404(Event, id=1)
    return render_to_response("event_info.html",
        { 'event' : e })
```

Page « événement »

```
{% block title %}  
Évènement {{ event.description }}  
{% endblock %}
```

```
{% block contents %}  
<h1>{{ event.description }}</h1>
```

```
<p>L'évènement <i>{{ event.description }}</i> démarre le  
{{event.start_date}} et se termine le {{event.end_date}}.</p>
```

```
{% if event.participant_set %}  
<ul>  
  {% for p in event.participant_set.all %}  
    <li><a href="mailto:{{p.email}}">{{ p.firstname }}  
    {{ p.lastname }}</a></li>  
  {% endfor %}  
</ul>  
  {% else %}  
<p>Pas de participants inscrits pour l'instant.</p>  
  {% endif %}
```

```
<a href="/event/{{event.id}}/subscribe">S'inscrire à cet évènement</a>
```


Formulaire

- Il faut maintenant s'attaquer à l'ajout d'évènement et à l'inscription
- Django offre la classe `forms.Form` pour déclarer des formulaires
 - Chaque formulaire comprend un certain nombre de champ, grâce aux classes `forms.XXXField()`
 - Chaque champ est associé à un widget
 - Et une API de validation et manipulation du formulaire
- On peut également générer le formulaire automatiquement à partir du modèle
 - Les champs du formulaire sont dérivés à partir des champs du modèle

Ajout d'évènement

Ajout de l'URL (urls.py)

```
urlpatterns += patterns('',  
    [...]  
    (r'^event/add/$', event_add),  
    [...]  
)
```

Déclaration du formulaire (views.py)

```
from django import forms  
  
class EventForm(forms.Form):  
    desc = forms.CharField()  
    start = forms.DateTimeField()  
    end = forms.DateTimeField()
```

Ajout d'évènement

Méthode de vue (views.py)

```
def event_add(request):  
    if request.method == 'POST':  
        form = EventForm(request.POST)  
        if form.is_valid():  
            e = Event()  
            e.description = form.cleaned_data['desc']  
            e.start_date = form.cleaned_data['start']  
            e.end_date = form.cleaned_data['end']  
            e.save()  
            return HttpResponseRedirect('/event/')  
        else:  
            form = EventForm()  
  
    return render_to_response("event_add.html",  
        { 'form': form })
```

Ajout d'évènement

Template (event_add.html)

```
{% block contents %}
<h1>Ajout d'un évènement</h1>

<form method="post">
<table>
{{ form.as_table }}
    <tr>
        <th></th>
        <td><input type="submit" value="Ajouter"/></td>
    </tr>
</table>
</form>

{% endblock %}
```

Inscription à un évènement

Ajout de l'URL (urls.py)

```
urlpatterns += patterns('',  
    [...]  
    (r'^event/(?P<eid>\d+)/submit$', event_subscribe),  
    [...]  
)
```

Déclaration du formulaire (views.py)

```
class ParticipantSubmitForm(forms.ModelForm):  
    class Meta:  
        model = Participant  
        exclude = ('event')
```

Inscription à un évènement

```
def event_subscribe(request, eid):
    e = get_object_or_404(Event, id=eid)
    if request.method == 'POST':
        form = ParticipantSubmitForm(request.POST)
        if form.is_valid():
            p = Participant()
            p.firstname = form.cleaned_data['firstname']
            p.lastname = form.cleaned_data['lastname']
            p.email = form.cleaned_data['email']
            p.event = e
            p.save()
            return HttpResponseRedirect('/event/%s' % eid)
    else:
        form = ParticipantSubmitForm()
    return render_to_response("event_subscribe.html",
                              { 'form': form })
```

Déploiement



Autres sujets

- Gestion des utilisateurs
 - Django intègre tout ce qui est authentification, gestion des sessions, stockage de données propres à l'utilisateur, etc.
- Interface d'administration
- Personnalisation des champs de formulaire, de modèles, des widgets
- Internationalisation