Formation Django

Session I

Présenté par : **JADLI AISSAM**

Formulaires HTML

En HTML, un formulaire est un ensemble d'éléments à l'intérieur des balises **<form>...</form>** qui permettent à un visiteur d'effectuer des actions comme saisir du texte, sélectionner des options, manipuler des objets ou des contrôles, et ainsi de suite, puis d'envoyer ces informations au serveur.

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
    <input type="submit" value="OK">
</form>
```

Formulaires HTML

En plus de ses éléments **<input>**, un formulaire doit préciser deux choses :

- **comment** : la méthode HTTP utilisée pour renvoyer les données (attribut **method**)
- **où** : l'URL vers laquelle les données correspondant à la saisie de l'utilisateur doivent être renvoyées (attribut **action**).

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
    <input type="submit" value="OK">
</form>
```

Exemple **:** Lorsque le bouton **<input type="submit" value="Connexion">** est déclenché, les données sont renvoyées à **/your-name/** en utilisant **POST**

Formulaires HTML et Vues

Vous pouvez utiliser les formulaire HTML classiques directement avec les vues Django en manipulant l'objet **`request`**:

- L'attribut **`request.method`** contient la méthode utilisée
- L'attribut **`request.body`** permet d'accéder au données envoyés dans le Body.
- L'attribut **`request.POST`** contient les données soumises avec une requête POST
- L'attribut **`request.FILES`** contient les fichiers téléversés par utilisateur.

Formulaires HTML et Vues

Exemples

```
data = request.POST.dict()
name = data.get('name')
email = data.get('email')
subject = data.get('subject')
```

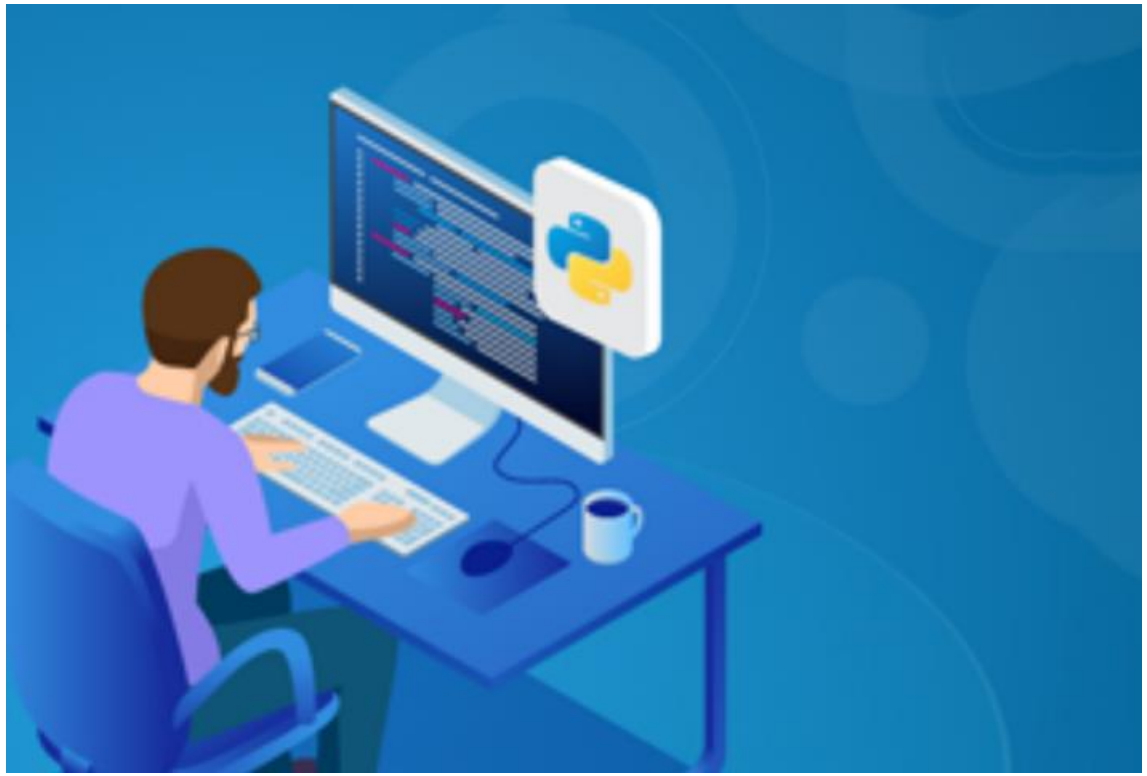
```
file = request.FILES['image']
```

```
m = Message.objects.create(name=name, subject=subject, email=email, image=file)
```

```
file = request.FILES['image']
default_storage.save("filename.txt", ContentFile(file.read()))
```

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

TRAVAUX PRATIQUES



Le rôle de Django dans les formulaires

Django gère **trois parties distinctes** du travail induit par les formulaires :

- Création des formulaires HTML pour les données;
- Préparation et restructuration des données en vue de leur Affichage;
- Réception et traitement des formulaires et des données envoyés par le client.

N.B: Il est possible d'écrire du code qui fait tout cela manuellement, mais Django peut s'en charger à votre place.

La classe Form de Django

De la même façon que la classe **Model** décrit la structure logique d'un objet et son comportement, la classe **Form** décrit un **formulaire** et détermine son **fonctionnement** et son **apparence**.

```python
from django import forms

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

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

Les Champs (Fields)

La bibliothèque des formulaires est livrée avec une série de classes Field représentant les besoins de validation les plus courants.

- **CharField**
- **BooleanField**
- **ChoiceField**
- **DateField**
- **DateTimeField**
- **DecimalField**
- **EmailField**
- **URLField**
- **FileField**
- **ImageField, …etc**

```python
class ArticleForm(ModelForm):
    headline = MyFormField(
        max_length=200,
        required=False,
        help_text='Use puns liberally',
    )
```

Paramètres des Champs

Chaque Champ accepte au moins **trois paramètres**. Certaines classes acceptent d'autres paramètres spécifiques à la classe selon les cas d'usage.

- **required**
- **label**
- **initial**
- **widget**
- **help_text**
- **error_messages**
- **validators**
- **disabled**

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" required></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" value="http://" required></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" required></td></tr>
```

Paramètres des Champs

Exemple

```
>>> from django import forms
>>> generic = forms.CharField()
>>> generic.clean('')
Traceback (most recent call last):
  ...
ValidationError: ['This field is required.']
```

```
>>> name = forms.CharField(error_messages={'required': 'Please enter your name'})
>>> name.clean('')
Traceback (most recent call last):
  ...
ValidationError: ['Please enter your name']
```

Champs dédiés aux relations

Deux champs sont disponibles pour représenter les relations entre modèles : **ModelChoiceField** and **ModelMultipleChoiceField**. Ces deux champs exigent un seul paramètre **queryset** utilisé pour créer les choix du champ.

```
# No custom to_field_name
field1 = forms.ModelChoiceField(queryset=...)
```

```
<select id="id_field1" name="field1">
<option value="obj1.pk">Object1</option>
<option value="obj2.pk">Object2</option>
...
</select>
```

Traitement par la Vue

Les données de formulaire renvoyés à une vue, **en principe la même qui a servi à produire le formulaire**.

Cela permet de réutiliser une partie de la même logique.

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

from .forms import NameForm

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

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

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

Affichage dans un Template

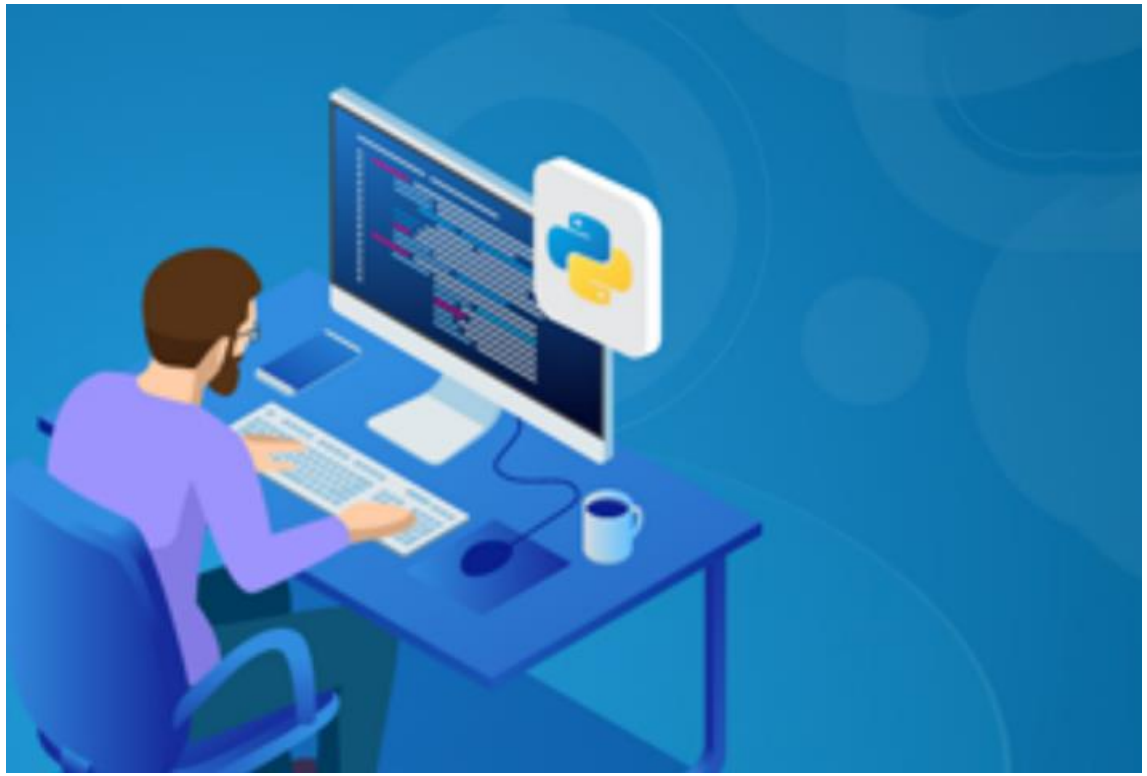
Tous les champs de formulaire et leurs attributs seront convertis en balises HTML à partir de **{{ form }}** par le langage de gabarit de Django.

La balise du formulaire **<form>** et le bouton de soumission **<input type="submit">** ne font pas partie du formulaire Django (il faut les ajouter manuellement).

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

N.B: Lors de l'envoi d'un formulaire par la méthode POST et la protection CSRF active, vous devez utiliser la balise de gabarit csrf_token.

TRAVAUX PRATIQUES



Validation des Données des Formulaires

La tâche principale d'un objet **Form** est de valider les données. Appelez la méthode **is_valid()** pour procéder à la validation et renvoyer un booléen indiquant si les données sont valides.

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
```

```
from django import forms

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

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
```

Validation des Données des Formulaires

Vous pouvez surcharger la définition de la méthode **`clean()`** pour personnaliser la validation de tout le formulaire ou bien appeler la méthode spécifique pour chacun des champs du formulaire sous la forme **clean_XXX().**

```python
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

    def clean_name(self):
        # custom validation for the name field
        ...
```

Validation Manuel du Formulaires

Pour spécifier manuellement la logique de validation de tout le formulaire, surcharger la méthode **clean()** du formulaire.

```python
from django import forms
from django.core.exceptions import ValidationError

class ContactForm(forms.Form):
    # Everything as before.
    ...

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

        if cc_myself and subject:
            # Only do something if both fields are valid so far.
            if "help" not in subject:
                raise ValidationError(
                    "Did not send for 'help' in the subject despite "
                    "CC'ing yourself."
                )
```

Données de champ

Au moment où les données ont été validées avec succès suite à l'appel de **is_valid()**, les données de formulaire validées se trouvent dans le dictionnaire **cleaned_data**. Ces données seront converties par défaut en en types Python.

```
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

Affichage dans les Template

Il existe toutefois d'autres options pour les paires <label>/<input>:

- `{{ form.as_table }}` affiche les composants sous forme de cellules de tableau à l'intérieur de balises **<tr>.**
- `{{ form.as_p }}` affiche les composants dans des balises **<p>.**
- `{{ form.as_ul }}` affiche les composants dans des balises **<li>.**

```
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" required></p>
<p><label for="id_message">Message:</label>
    <textarea name="message" id="id_message" required></textarea></p>
<p><label for="id_sender">Sender:</label>
    <input type="email" name="sender" id="id_sender" required></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself"></p>
```

Affichage manuel des champs

On peut tout à fait afficher soi-même les champs (par exemple pour changer leur ordre d'apparition). Chaque champ est accessible en tant qu'attribut du formulaire avec la syntaxe **{{ form.nom_du_champ }}.**

```
<div class="fieldWrapper">

    <label for="message"  >Your message:</label>
    {{ form.message }}
</div>
```

Affichage des messages d'erreur de formulaires

La syntaxe **{{ form.nom_du_champ.errors }}** affiche une liste des erreurs du formulaire, sous forme de liste non ordonnée.

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email
subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">

    <label for="message"  >Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email
address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC
yourself?</label>
    {{ form.cc_myself }}
</div>
```

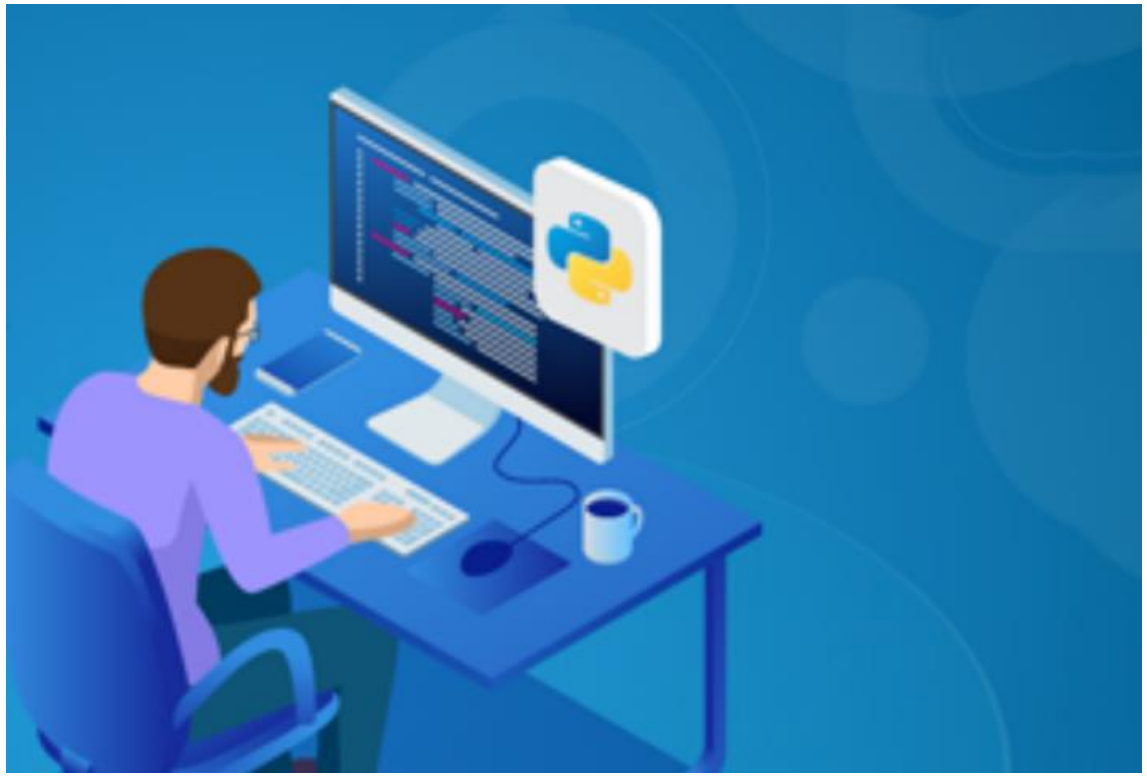
Gabarits de formulaire réutilisables

Si l'application utilise la même logique d'affichage des formulaires à plusieurs endroits, vous pouvez réduire la duplication en enregistrant la boucle de formulaire dans un gabarit autonome et en employant la balise **include** afin de réutiliser ce contenu dans d'autres gabarits.

```
# In your form template:
{% include "form_snippet.html" %}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

TRAVAUX PRATIQUES



Les Composants de formulaires (« widgets »)

Un composant de formulaire est la **représentation Django d'un élément de saisie HTML**. Le composant se charge de produire le code HTML et d'extraire les données qui lui sont propres dans un dictionnaire GET/POST.

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

Paramètres des composants

Beaucoup de composants acceptent des paramètres supplémentaires facultatif; ils peuvent être définis lors de l'attribution du composant au champ de formulaire.

```python
from django import forms

BIRTH_YEAR_CHOICES = ['1980', '1981', '1982']
FAVORITE_COLORS_CHOICES = [
    ('blue', 'Blue'),
    ('green', 'Green'),
    ('black', 'Black'),
]

class SimpleForm(forms.Form):
    birth_year = forms.DateField(widget=forms.SelectDateWidget(years=BIRTH_YEAR_CHOICES))
    favorite_colors = forms.MultipleChoiceField(
        required=False,
        widget=forms.CheckboxSelectMultiple,
        choices=FAVORITE_COLORS_CHOICES,
    )
```

Personnalisation des instances de composants

Si on souhaite qu'un composant apparaisse différemment d'un autre, il sera nécessaire d'indiquer des attributs supplémentaires lors de la définition du champ de formulaire

```python
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

```html
<tr><th>Name:</th><td><input type="text" name="name" class="special" required></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40" required></td></tr>
```

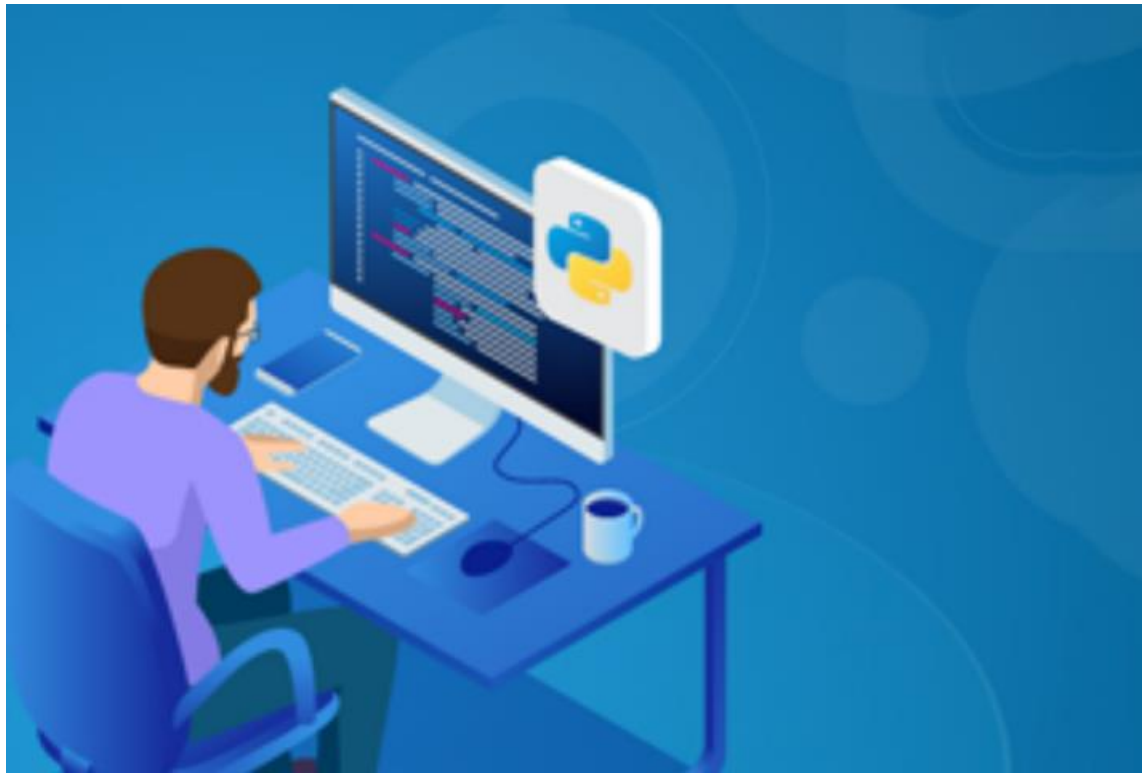
Personnalisation des instances de composants

Si on souhaite qu'un composant apparaisse différemment d'un autre, il sera nécessaire d'indiquer des attributs supplémentaires lors de la définition du champ de formulaire

```python
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

```html
<tr><th>Name:</th><td><input type="text" name="name" class="special" required></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40" required></td></tr>
```

TRAVAUX PRATIQUES



Création de formulaires à partir de modèles

il y a des chances pour que les formulaires correspondent étroitement avec les modèles Django. Dans ce cas, il serait redondant de devoir définir les types de champs du formulaire, car ils sont déjà défini des champs au niveau du modèle.

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Création de formulaires à partir de modèles

Exemple

```python
from django import forms

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

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

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

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

Validation d'un ModelForm

La validation d'un **ModelForm** se distingue en deux étapes importantes :

- La validation du formulaire
- La validation de l'instance de modèle

Comme pour la validation de formulaire normale, la validation des formulaires de modèle est déclenchée implicitement lors de l'appel à **is_valid()** ou par l'accession à l'attribut **errors**, ou explicitement en appelant **full_clean().**

La méthode save()

Chaque **ModelForm** possède aussi une méthode **save()**. Celle-ci crée et enregistre un objet en base de données à partir des données saisies dans le formulaire.

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

La méthode save()

Chaque **ModelForm** possède aussi une méthode **save()**. Celle-ci crée et enregistre un objet en base de données à partir des données saisies dans le formulaire.

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Sélection des champs à utiliser

Il est fortement recommandé de définir explicitement tous les champs qui doivent être présents dans le formulaire en utilisant l'attribut **fields** ou **exclude**.

```python
from django.forms import ModelForm

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'
```

```python
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ['title']
```

```python
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name']
```

Surcharge des champs par défaut

La classe **ModelForm** apporte la souplesse de pouvoir modifier les champs de formulaire pour un modèle défini.

```python
from django.forms import ModelForm, Textarea
from myapp.models import Author

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        widgets = {
            'name': Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

```python
from django.utils.translation import gettext_lazy as _

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        labels = {
            'name': _('Writer'),
        }
        help_texts = {
            'name': _('Some useful help text.'),
        }
        error_messages = {
            'name': {
                'max_length': _("This writer's name is too
long."),
            },
        }
```

Héritage des Formulaires

Si vous avez plusieurs classes **Form** dont les champs **sont partagés**, il est possible d'utiliser **l'héritage pour éviter la redondance**.

```
>>> from django import forms
>>> class PersonForm(forms.Form):
...     first_name = forms.CharField()
...     last_name = forms.CharField()
>>> class InstrumentForm(forms.Form):
...     instrument = forms.CharField()
>>> class BeatleForm(InstrumentForm, PersonForm):
...     haircut_type = forms.CharField()
>>> b = BeatleForm(auto_id=False)
>>> print(b.as_ul())
<li>First name: <input type="text" name="first_name" required></li>
<li>Last name: <input type="text" name="last_name" required></li>
<li>Instrument: <input type="text" name="instrument" required></li>
<li>Haircut type: <input type="text" name="haircut_type" required></li>
```

```
>>> class ContactFormWithPriority(ContactForm):
...     priority = forms.CharField()
>>> f = ContactFormWithPriority(auto_id=False)
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" required></li>
<li>Message: <input type="text" name="message" required></li>
<li>Sender: <input type="email" name="sender" required></li>
<li>Cc myself: <input type="checkbox" name="cc_myself"></li>
<li>Priority: <input type="text" name="priority" required></li>
```

Téléversement de Fichiers avec les Formulaires

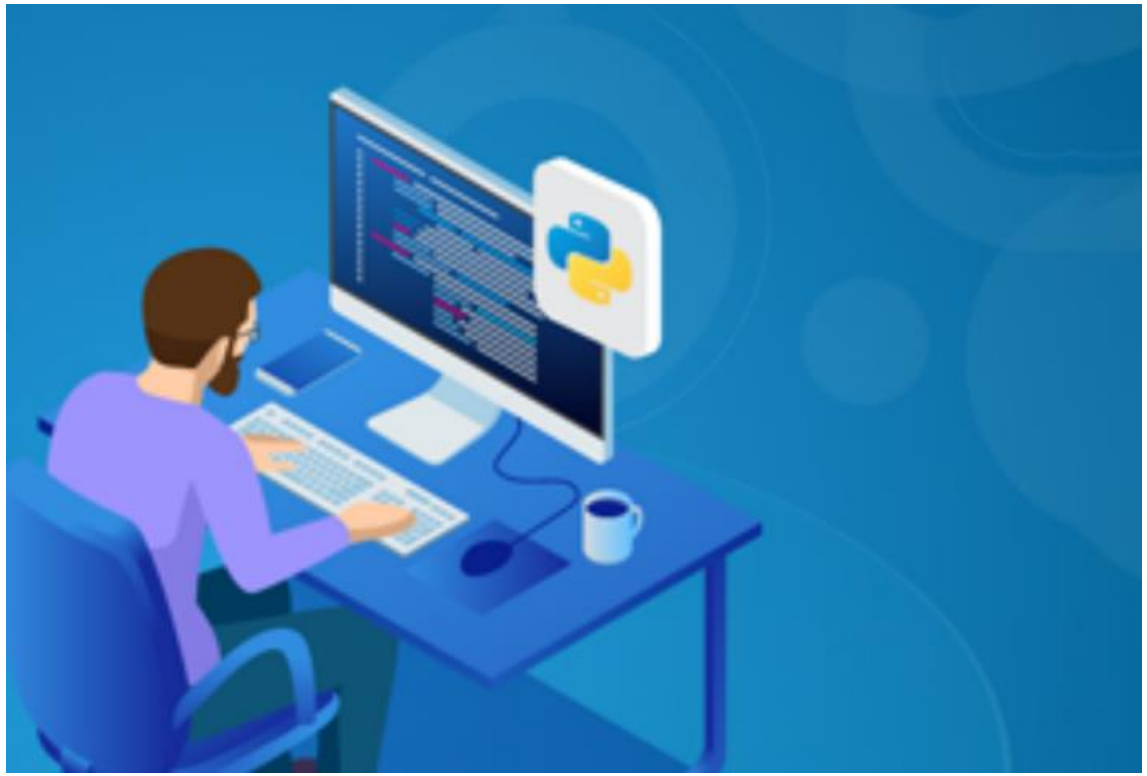
Premièrement, pour pouvoir envoyer des fichiers, il est important que la balise **<form>** du formulaire définisse correctement son attribut **enctype** à **"multipart/form-data".**

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

vous indiquez **request.FILES** comme source des données de fichier (comme pour **request.POST** représentant la source des données de formulaire)

```
# Bound form with an image field, data from the request
>>> f = ContactFormWithImage(request.POST, request.FILES)
```

TRAVAUX PRATIQUES

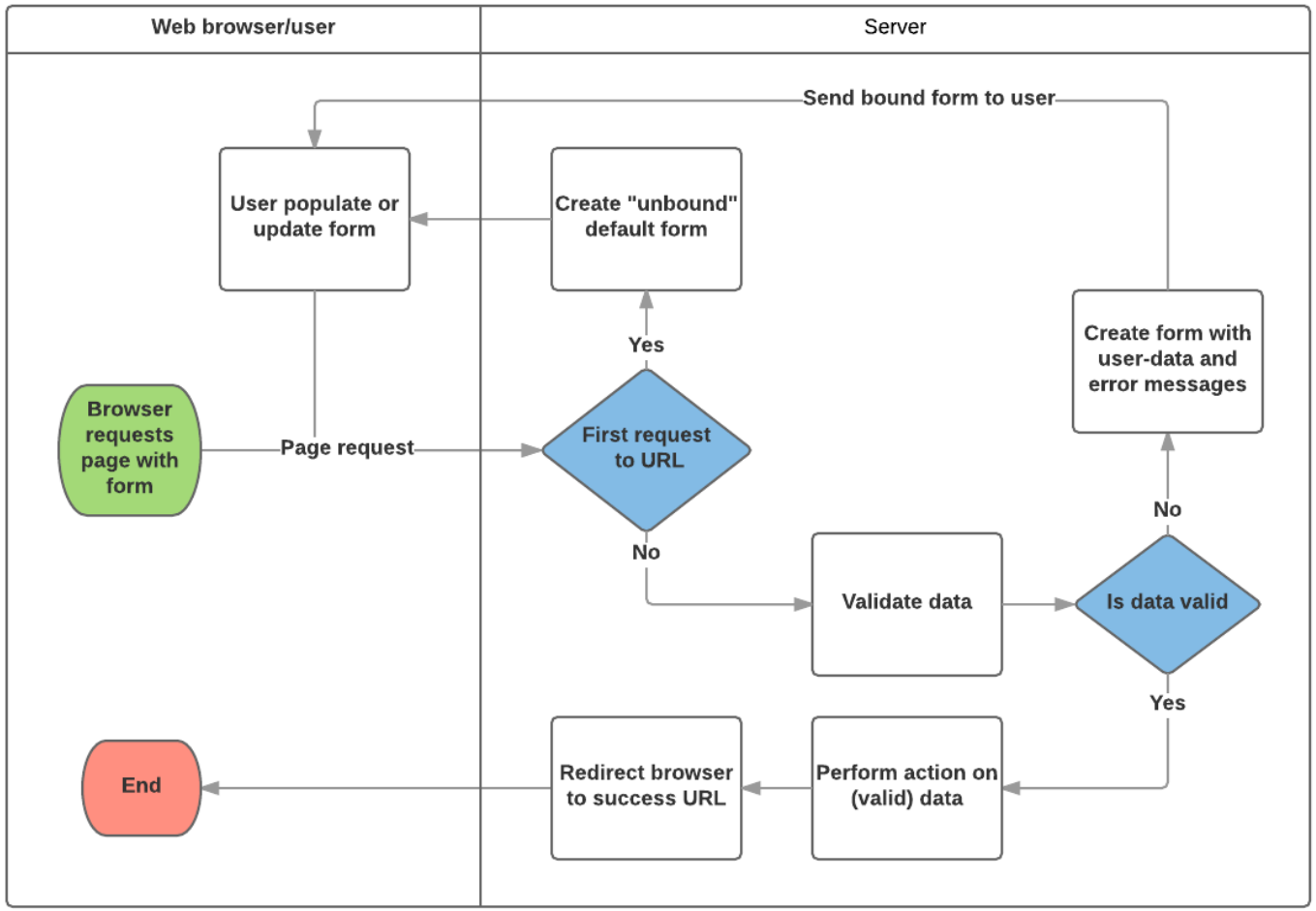


Forms & Class-based Views

Le traitement de formulaires utilise généralement trois parcours :

- Affichage initial **GET** (**vierge** ou **contenu pré-rempli**)
- Envoi **POST** avec données non valides (réaffiche normalement le formulaire avec **indication des erreurs**)
- Envoi **POST** avec données valides (**traitement des données normalement suivi par une redirection**)

Forms & Class-based Views



Forms & Class-based Views

Exemple

views.py

```python
from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactFormView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super().form_valid(form)
```

forms.py

```python
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

Forms & Class-based Views

Exemple

Remarquez que nous avons ici juste à **configurer les vues génériques fondées sur les classes** ; nous n'avons pas à écrire nous-même de logique de vue.

views.py

```python
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView, DeleteView, UpdateView
from myapp.models import Author

class AuthorCreateView(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdateView(UpdateView):
    model = Author
    fields = ['name']

class AuthorDeleteView(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

Forms & Class-based Views

Pour garder trace de l'utilisateur ayant créé un objet en utilisant CreateView, vous pouvez utiliser un formulaire ModelForm personnalisé.

views.py

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreateView(LoginRequiredMixin, CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super().form_valid(form)
```

models.py

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

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)

    # ...
```

Création de champs personnalisés

Si les classes **Field** intégrées ne correspondent pas aux besoins, vous pouvez créer des classes Field personnalisées qui héritent de **django.forms.Field**.

Les seules exigences sont queune des méthodes **clean() ou validate()** **doit être implémentée.**

```python
from django import forms
from django.core.validators import validate_email

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

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

TRAVAUX PRATIQUES