# Rapid Web Development with Python/Django

## Templates and Forms

B. Wakim

## Outline

# Templates

## Template definition

- A template is simply a text file containing:
  - Variables: get replaced with values
  - Tags: control the logic of the template
- A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags.
- Everything else is output as is.

## Variables

- A variable outputs a value from the context, which is a dict-like object mapping keys to values. Variables are surrounded by {{ and }}.
- Example:
  - My first name is {{ first_name }}. My last name is {{ last_name }}.
  - *With a context of {'first_name': 'John', 'last_name': 'Doe'}, this template renders to:*
    - My first name is John. My last name is Doe.
- Example:

```
class book(models.Model):
      title = models.CharField(max_length=50)
      author = models.CharField(max_length=35)
      obj = book(title = “My life”, author = “Unknow”)
```

- ***Template***

```
<h3> {{ obj.title }} </h3>
 <p> {{ obj.author }} </p>
```

- ***Result***

```
<h3> My life </h3>
<p> Unknow </p>
```

## Tags¶

- Tags provide arbitrary logic in the rendering process.
- Tags are surrounded by {% and %} like this:

```
{% if list | length > 0 %}
        List: {% for i in list %} {{ i }} {% endfor %}
{% else %}
        List is empty
{% endif %}
```

- If list = [1, 2, 3, 4] ➔ display: List: 1 2 3 4
- If list = [] ➔ display: List is empty

## Tags - Examples¶

- {% csrf_token %}: This tag is used for CSRF protection.
  - Django features a percent csrf token percent tag that is used to prevent malicious attacks. When generating the page on the server, it generates a token and ensures that any requests coming back in are cross-checked against this token.
  - You need to add django. middleware. csrf. CsrfViewMiddleware in the settings.py file to enable it.

- Extends: signals that this template extends a parent template.

  {% extends "base.html" %} ➔ (with quotes) uses the literal value "base.html" as the name of the parent template to extend.

## Tags - Comment

- Ignores everything between {% comment %} and {% endcomment %}. An optional note may be inserted in the first tag.
- For example, this is useful when commenting out code for documenting why the code was disabled.

```
<p>Rendered text with {{ pub_date|date:"c" }}</p>
{% comment "Optional note" %}
    <p>Commented out text with {{ create_date|date:"c" }}</p>
{% endcomment %}
```

- comment tags cannot be nested.

- "c" : ISO 8601 format. For Date filter used in template: [https://docs.djangoproject.com/en/4.2/ref/templates/builtins/#std-templatefilter-date](https://docs.djangoproject.com/en/4.2/ref/templates/builtins/#std-templatefilter-date)

## Tags - For

- Loops over each item in an array, making the item available in a context variable.
- For example, to display a list of athletes provided in athlete_list:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

- This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary data, the following would display the keys and values of the dictionary:

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

# Tags - If

- The {% if %} tag evaluates a variable, and if that variable is "true" (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

- If tags may use and, or not to test a number of variables or to negate a given variable:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}
```

## Tags If

- If tags may also use the operators ==, !=, <, >, <=, >=, in, not in, is, and is not which work as follows:

```
{% if somevar == "x" %}
  This appears if variable somevar equals the string "x"
{% endif %}

{% if "bc" in "abcdef" %}
  This appears since "bc" is a substring of "abcdef"
{% endif %}

{% if messages|length >= 100 %}
   You have lots of messages today!
{% endif %}
```

## Filters

- Filters transform the values of variables and tag arguments.
- Syntax: {{ variable | filter [| filter ...] }}

- Some filters take an argument: {{ my_date|date:"Y-m-d" }} used to change date format
- Example:

  value1 = “” value2 = [1, 4, 2, 6]

  <p>{{ value1 | default: "nothing" }}</p> ➔ <p> nothing </p>

  <p> Length: {{ value2 | length }}</p> ➔ <p> Length: 4 </p>

## Some Built-in filters

- Add: Adds the argument to the value.
  {{ value|add:"2" }} ➔ If value is 4, then the output will be 6.
- Capfirst: Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.
  {{ value|capfirst }} ➔ If value is "django", the output will be "Django".
- Center: Centers the value in a field of a given width.
  "{{ value|center:"15" }}“ ➔ If value is "Django", the output will be "     Django    ".
- cut: Removes all values of arg from the given string.
  {{ value|cut:" " }} ➔ If value is "String with spaces", the output will be "Stringwithspaces".
- First: Returns the first item in a list.
  {{ value|first }} ➔ If value is the list ['a', 'b', 'c'], the output will be 'a'.
- Slice: Returns a slice of the list.
  {{ some_list|slice:":2" }} ➔
  If some_list is ['a', 'b', 'c'], the output will be ['a', 'b'].

# Loading templates

- **setting.py**

```
TEMPLATE_DIR =
        ( “mysite/app/template”,
        “home/default”,
)
```

- **Views.py**

```
def  viewExample(request, title, author):
        obj = book(title, author)
        return render (request, “template.html”, {“book” : obj})
```

- **Template.html**

```
<p> This is a book </p>
<p> Title: {{ book.title }} </p>
<p> Author: {{ book.author }}</p>
```

- **Result**

```
<p> This is a book </p>
<p> Title: My life </p>
<p> Author: H.Anh </p>
```

## Forms

## Forms in Django

- Forms in Django¶
  - HTML forms
  - Django forms

- **The Django Form class** describes a form and determines how it works and appears.
  - A form class's fields map to HTML form <input> elements.
  - A ModelForm maps a model class's fields to HTML form <input> elements via a Form; this is what the Django admin is based upon.
  - A form's fields are themselves classes; they manage form data and perform validation when a form is submitted.
  - Each field type has an appropriate default Widget class, but these can be overridden as required.

## Building a form in Django

- A Form object encapsulates a sequence of form fields and a collection of validation rules that must be fulfilled in order for the form to be accepted.
- **forms.py¶**

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

- The field's maximum allowable length is defined by max_length. It puts a maxlength="100" on the HTML <input>. It also means that when Django receives the form back from the browser, it will validate the length of the data.

## Building a form in Django

- An unbound form does not have any data associated with it; when rendered to the user, it will be empty or will contain default values.

```
>>> from blog.forms import AuthorForm
>>> f = AuthorForm()
```

- A bound form does have submitted data, and hence can be used to tell if that data is valid.

```
>>> data = {
... 'name': 'jon',
... 'created_on': 'today',
... 'active': True,
... }
>>> f = AuthorForm(data)
```

## Building a form in Django

- **Cleaning data**
  - Any data the user submits through a form will be passed to the server as strings. It doesn't matter which type of form field was used to create the form.
  - When Django cleans the data it automatically converts data to the appropriate type. For example IntegerField data would be converted to an integer, CharField data would be converted to a string, BooleanField data would be converted to a bool i.e True or False and so on.
  - We can access cleaned data via cleaned_data dictiona
  - Tryrying to access cleaned_data before invoking is_valid() will throw an AttributeError exception.
- A Form instance has an is_valid() method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:
  - return True
  - place the form's data in its cleaned_data attribute.

## Building a form in Django

```
>>> import datetime
>>> data = {
...  'name': 'tim',
...  'email': 'tim@mail.com',
...  'active': True,
...  'created_on': datetime.datetime.now(),
...  'last_logged_in': datetime.datetime.now()
... }
>>> f = AuthorForm(data)
>>> f.is_bound
True
>>> f.is_valid()
True
>>> f.cleaned_data
{'name': 'tim', 'created_on': datetime.datetime(2017, 4, 29, 14, 11, 59, 433661,
 tzinfo=<UTC>), 'last_logged_in': datetime.datetime(2017, 4, 29, 14, 11, 59, 433
661, tzinfo=<UTC>), 'email': 'tim@mail.com', 'active': True}
```

## Using Form in a view

- To handle the form we need to instantiate it in the view for the URL where we want it to be published.
- If the form has been submitted, a bound instance of the form is created using request.POST.
- If the submitted data is valid, it is processed and the user is re-directed to a "thanks" page.
- If the form has been submitted but is invalid, the bound form instance is passed on to the template.

```
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
            your_name= form.cleaned_data[‘your-name']
            # 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})
```

## Using form in a template

- **name.html** template:

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

- All the form’s fields and their attributes will be unpacked into HTML markup from that {{ form }} by Django’s template language.
- We now have a working web form, described by a Django Form, processed by a view, and rendered as an HTML <form>.

## Form fields – widgets

```
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)
```

- **Widgets¶**
- Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such as <input type="text">.
- For example, by default, a CharField will have a TextInput widget, that produces an <input type="text"> in the HTML.
- If you needed <textarea> instead, you'd specify the appropriate widget when defining your form field, as we have done for the message field.

## Form fields - Cleaned data

- Each field in a Form class is responsible not only for validating data, but also for "cleaning" it – normalizing it to a consistent format.
- For example, DateField normalizes input into a Python datetime.date object. Regardless of whether you pass it a string in the format '1994-07-15', a datetime.date object, or a number of other formats, DateField will always normalize it to a datetime.date object as long as it's valid.
- Once you've created a Form instance with a set of data and validated it, you can **access** the clean data via its cleaned_data attribute:

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

## Displaying a form using a template

- All you need to do to get your form into a template is to place the form instance into the template context.
- So if your form is called form in the context, {{ form }} will render its <label> and <input> elements appropriately.

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

## Displaying a form using a template

- There are other output options though for the <label>/<input> pairs:
  - {{ form.as_table }} will render them as table cells wrapped in <tr> tags
  - {{ form.as_p }} will render them wrapped in <p> tags
  - {{ form.as_ul }} will render them wrapped in <li> tags

- Note that each form field has an ID attribute set to id_<field-name>, which is referenced by the accompanying label tag.

```
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>
```

## Useful attributes of field

- {{ field.label }} : The label of the field, e.g. Email address.
- {{ field.label_tag }}: The field's label wrapped in the appropriate HTML <label> tag.
- {{ field.value }}: The value of the field. e.g *someone@example.com*.
- {{ field.html_name }}: The name of the field that will be used in the input element's name field.
- {{ field.help_text }}: Any help text that has been associated with the field.
- {{ field.errors }}: Outputs a <ul class="errorlist"> containing any validation errors corresponding to this field.

## Form fields validation

- **Field.clean(value)¶**
- Each Field instance has a clean() method, which takes a single argument and either raises a django.core.exceptions.ValidationError exception or returns the clean value:

```
>>> from django import forms
>>> f = forms.EmailField()
>>> f.clean('foo@example.com')
'foo@example.com'
>>> f.clean('invalid email address')
Traceback (most recent call last):
...
ValidationError: ['Enter a valid email address.']
```

## Core Form fields arguments - Required

- By default, each Field class assumes the value is required, so if you pass an empty value .
- To specify that a field is not required, pass required=False to the Field constructor.

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

## Core Form fields arguments - Label

- The label argument lets you specify the “human-friendly” label for this field. This is used when the Field is displayed in a Form.
- The default label for a Field is generated from the field name by converting all underscores to spaces and upper-casing the first letter.

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

## Form auto_id argument

- Use the auto_id argument to the Form constructor to control the id and label behavior. This argument must be True, False or a string.
- If auto_id is False, then the form output will not include <label> tags nor id attributes:

```
div>Subject:<input type="text" name="subject" maxlength="100" required></div>
<div>Message:<textarea name="message" cols="40" rows="10" required></textarea></div>
```

- If auto_id is set to True, then the form output will include <label> tags and will use the field name as its id for each form field:

```
div><label for="subject">Subject:</label><input type="text" name="subject"
maxlength="100" required id="subject"></div>
<div><label for="message">Message:</label><textarea name="message"
cols="40" rows="10" required id="message"></textarea></div>
```

## Core Form fields arguments - Initial

- The initial argument lets you specify the initial value to use when rendering this Field in an unbound Form.

```
>>> 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>
```

## Core Form fields arguments - Help

- The help_text argument lets you specify descriptive text for this Field.
- If you provide help_text, it will be displayed next to the Field when the Field is rendered by one of the convenience Form methods (e.g., as_ul()).

```
>>> from django import forms
>>> class HelpTextContactForm(forms.Form):
...     subject = forms.CharField(max_length=100, help_text='100 characters max.')
...     message = forms.CharField()
...     sender = forms.EmailField(help_text='A valid email address, please.')
...     cc_myself = forms.BooleanField(required=False)
>>> f = HelpTextContactForm(auto_id=False)
>>> print(f.as_ul()))
<li>Subject: <input type="text" name="subject" maxlength="100" required> <span
class="helptext">100 characters max.</span></li>
<li>Message: <input type="text" name="message" required></li>
<li>Sender: <input type="email" name="sender" required> A valid email address,
please.</li>
<li>Cc myself: <input type="checkbox" name="cc_myself"></li>
```

## Core Form fields arguments – Error messages

- The error_messages argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override.

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

- And here is a custom error message:

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

## Looping over the fields

- If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a {% for %} loop:

```
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
        {% if field.help_text %}
        <p class="help">{{ field.help_text|safe }}</p>
        {% endif %}
    </div>
{% endfor %}
```

## Creating forms from models¶

## ModelForm¶

- Django provides a helper class that lets you create a Form class from a Django model, to avoid redundancy in defining the field types in your form.

```
>>> 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)
```

## Field types conversion

- The generated Form class will have a form field for every model field specified, in the order specified in the fields attribute.
- Each model field has a corresponding default form field.
  - a CharField on a model is represented as a CharField on a form.
  - ForeignKey is represented by django.forms.ModelChoiceField, which is a ChoiceField whose choices are a model QuerySet.
  - ManyToManyField is represented by django.forms.ModelMultipleChoiceField, which is a MultipleChoiceField whose choices are a model QuerySet.
- In addition, each generated form field has attributes set as follows:
  - If the model field has blank=True, then required is set to False on the form field. Otherwise, required=True.
  - The form field's label is set to the verbose_name of the model field, with the first character capitalized.
  - The form field's help_text is set to the help_text of the model field.

## Field types conversion

| Model field | Form field |
|---|---|
| BigIntegerField | IntegerField with min_value set to -9223372036854775808 and max_value set to 9223372036854775807. |
| BooleanField | BooleanField, or NullBooleanField if null=True. |
| CharField | CharField with max_length set to the model field’s max_length and empty_value set to None if null=True. |
| DateField | DateField |
| DateTimeField | DateTimeField |
| DecimalField | DecimalField |
| DurationField | DurationField |
| EmailField | EmailField |
| FileField | FileField |
| FilePathField | FilePathField |
| FloatField | FloatField |
| ForeignKey | ModelChoiceField (see below) |
| ImageField | ImageField |
| IntegerField | IntegerField |
| IPAddressField | IPAddressField |
| JSONField | JSONField |
| ManyToManyField | ModelMultipleChoiceField (see below) |
| NullBooleanField | NullBooleanField |
| TextField | CharField with widget=forms.Textarea |
| TimeField | TimeField |
| URLField | URLField |

## ModelForm - Save method

- Every ModelForm also has a save() method. This method creates and saves a database object from the data bound to the form.

```
>>> 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()
```

## ModelForms – Selecting the fields to use

- Set the fields attribute to the special value '__all__' to indicate that all fields in the model should be used.

```
from django.forms import ModelForm
class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'
```

- Set the exclude attribute of the ModelForm's inner Meta class to a list of fields to be excluded from the form.

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

## ModelForms – Overriding the default fields

- To specify a custom widget for a field, use the widgets attribute of the inner Meta class.
- For example, if you want the CharField for the name attribute of Author to be represented by a <textarea> instead of its default <input type="text">, you can override the field's widget:

```
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}),
        }
```

## ModelForms – Customize a field

- Similarly, you can specify the labels, help_texts and error_messages attributes of the inner Meta class if you want to further customize a field.

```
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."),
            },
        }
```

## Ressources

- **Built-in filters:** https://docs.djangoproject.com/en/3.2/ref/templates/builtins/#ref-templates-builtins-filters
- **Templates tags**
  https://docs.djangoproject.com/en/3.2/howto/custom-template-tags/
- **Working with forms:**
  https://docs.djangoproject.com/en/3.2/topics/forms/
- **Form API:**
  https://docs.djangoproject.com/en/3.2/ref/forms/api/
- **FormFields Reference:**
  https://docs.djangoproject.com/en/3.2/ref/forms/fields/
- **Form from models:** https://docs.djangoproject.com/en/3.2/topics/forms/modelforms/