Django Templating

Most of the views we are going to create with **Django** will require us to render an HTML. The best way to create an HTML is to use templates. Templates allows us to create an HTML of our view with dynamic content inserted to it. Our view is in charge of rendering the template and injecting the dynamic content to that template.
Django comes with two built in engines for our templating.

- **Django Template Language** - The built in templating engine of **Django**
- **Jinja2 -** after **Django1.8** there was added built in option to create a template in the popular **Jinja** templating engine.

In this lesson we will cover **DTL (Django Template Language)** and we will cover the different things we can do in our template

New Django App

Create a new folder for your project called **django-dtl** and **cd** into that directory.
Create a new virtual environment to install your project:

> virtualenv venv

Activate your virtual environment by typing:

> source venv/bin/activate

now install django by typing:

> pip install django

Now start a new **Django** project by typing:

> django-admin startproject dtl

Now we need to create a new application in our project so in the terminal type:

> cd tdl
> python manage.py startapp dtl_intro

Modify the **settings.py INSTALLED_APPS** to include the new application we just created:

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

notice also that in the **settings.py** file there are default options created for the template engine. The options are located at the **TEMPLATES** dictionary and should look like this:

```
TEMPLATES = [
  {
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
  },
]
```

Let's cover those default options:

- **BACKEND -** which template engine we want to use, there is another built in engine called **Jinja** which is also very popular.
- **DIRS -** defines a list of directories where the engine should look for the template
- **APP_DIRS -** whether the engine should look for the template inside app packages. For **DTL** if this option is set to **True** it will search for the template in a directory called **templates** in the app.
- **context_processors -** string path to functions that gets the request and inject additional context to the template

We will now add a route in our created app to map to a view that will render a simple template. In the project **urls.py** we need to add url to map to the new app so change it to look like this:

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

```
from django.contrib import admin

urlpatterns = [
  url(r'^admin/', admin.site.urls),
  url(r'', include('dtl_intro.urls'))
]
```

the root url will direct to the urls of our app.
We still need to create the **urls.py** file in our app, so in the **dtl_intro** package create a file called: **urls.py** with the following code:

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

urlpatterns = [
  url(r'^$', views.index, name='home')
]
```

We set the root url to point to the **views.index** file, which we will create now.
Modify the **views.py** file to look like this:

```
from django.shortcuts import render

def index(request):
  return render(request, 'dtl_intro/intro.html')
```

the render method gets the **request** and a string of the template to load. Since in the options for the templates in the **settings.py** the option for **APP_DIRS** is set to **True** then the **DTL** engine will look for the template in the **templates** folder of our app.
Create the **templates** folder in the **dtl_intro** packages and create another directory in the templates folder called **dtl_intro** add a file in that directory called **intro.html** with the following content:

```
<h1>
  Hello intro template!
</h1>
```

Note that it's common to put the templates in a folder with the appname, this will prevent a collision of the template names with other apps.
Let's try and run our application, in your terminal type: **> python manage.py runserver**
and open you browser at the url **localhost:8000**
You should see the greeting message which means our template is now rendering.
From this bootstrap let's try to examine the common things we can do with **DTL**

Passing Variables & Functions

we can pass data to be injected in our template, let's try and experiment with this feature.
Modify your **views.py** file to look like this:

```
from django.shortcuts import render

def index(request):
  return render(request, 'dtl_intro/intro.html', context={
      'greeting': 'hello from greeting variable',
      'obj': {'greeting': 'we can also pass classes and objects'},
      'func': lambda: 'hello from function'
  })
```

notice that we are now passing a dictionary in the **render** method **context** parameter. Which means we are passing the dictionary keys to the template.
Modify the template **intro.html** to look like this:

```
<h1>
  Hello intro template!
</h1>

<h2>
  Injecting variables and functions
</h2>

<p>
  {{ greeting }}
</p>
<p>
  {{ obj.greeting }}
</p>
<p>
  {{ func }}
</p>
```

Note that we can inject variables and objects or classes and even functions that will be called when the template is rendered, notice that when placing a function we didn't have to call it the python way and just place it in the template, while rendering the template will execute automatically callable objects.

Template Inheritance

As a general rule when you write your **Django** templates, if you find yourself having a content which is repeated on different templates and you are doing copy paste, then you probably are not using the **inheritance** feature of **DTL** properly.
Template inheritance gives you the power to define a parent template, and define blocks in that template that the child templates that inherit from the parent can fill those blocks.
You can inherit one base template but that base template can also inherit and this flow can go on and on. You can also define a default value in the parent for a block and optionally override the content of that block.
Let's try to do the following exercise with template inheritance. We will now create a parent template with header block and content block. We will also create 3 child templates:

- one will grab the default header
- one will override the header with an empty header
- one will add items to the header

Create the file **base.html** with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Template Inheritance</title>
</head>
<body>


  <nav>
    <ul>
      {% block navbar %}
        <li>
          <a href="{% url 'home' %}">
            Home
          </a>
        </li>
        <li>
          <a href="{% url 'no-header' %}">
            No header
          </a>
        </li>
        <li>
          <a href="{% url 'add-to-header' %}">
```

```
                Add to header
              </a>
            </li>
        {% endblock %}
      </ul>
  </nav>


  {% block content %}
  {% endblock %}

</body>
</html>
```

We created the general skeleton of our page and also added two blocks to be extended in the child templates.
The navbar block has a default value so if a child won't extend it it will present the navbar in the **base.html** file.
The second block is empty and need to be completed in the child templates.
Note that for the navbar links we are using the **url** tag to grab the url of a view by name, this is the prefered way to get path url instead of putting constant values there.

Change the **urls.py** file in the package **dtl_intro** to this:

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

urlpatterns = [
  url(r'^$', views.index, name='home'),
  url(r'^no-header$', views.no_header, name='no-header'),
  url(r'^add-to-header$', views.add_to_header, name='add-to-header'),
]
```

we added additional two urls and also added names to those urls to match the url tag we added in the base

In the **views.py** add the missing view functions that are defined in the urls:

```
from django.shortcuts import render

def index(request):
  return render(request, 'dtl_intro/intro.html', context={
      'greeting': 'hello from greeting variable',
```

```
    'obj': {'greeting': 'we can also pass classes and objects'},
    'func': lambda: 'hello from function'
  })

def no_header(request):
  return render(request, 'dtl_intro/no-header.html')

def add_to_header(request):
  return render(request, 'dtl_intro/add-to-header.html')
```

we are rendering in each function a different template which we will now add those files in the **templates/dtl_intro** folder:

intro.html

```
{% extends 'dtl_intro/base.html' %}


{% block content %}
<h1>
  Hello intro template!
</h1>

<h2>
  Injecting variables and functions
</h2>

<p>
  {{ greeting }}
</p>
<p>
  {{ obj.greeting }}
</p>
<p>
  {{ func }}
</p>
{% endblock %}
```

In this template we are extending the base template we created earlier, notice that the extend should be placed at the top of the file. We are not overriding the **navbar** block so the default in the base will be used. For the **content** block we are overriding it with the content we had before.

add-to-header.html

```
{% extends 'dtl_intro/base.html' %}

{% block navbar %}
  {{ block.super }}
  <li>
     This is added to the navbar
  </li>
{% endblock %}
```

once again we are extending the base only this time we are overriding the **navbar** block and we are using the **block.super** variable to get the content of the parent and adding values to that content.

no-header.html

```
{% extends 'dtl_intro/base.html' %}

{% block navbar %}
{% endblock %}

{% block content %}
  <h1>
     No header template
  </h1>
{% endblock %}
```

We are extending the base template only this time we are overriding the navbar block with empty content.

Run your app and you should see the navigation bar we just created

include

Another very useful feature that you can use to avoid code duplication is the **include** which gives you the ability to reuse a template and inject it.
While inheritance is recommended for features that recur in every page of your app, the include is useful when you have repeated features that are not common in all the pages.
For example sake let's say that the footer in our app is only available in the root url and in the no-header url.

In your **templates/dtl_intro** open another directory called **includes** and place the file **footer.html** in that folder with the following code:

```
<h1>
  Hey Footer!
</h1>
```

Now modify the **templates/dtl_intro/intro.html**

```
{% extends 'dtl_intro/base.html' %}


{% block content %}
<h1>
  Hello intro template!
</h1>

<h2>
  Injecting variables and functions
</h2>

<p>
  {{ greeting }}
</p>
<p>
  {{ obj.greeting }}
</p>
<p>
  {{ func }}
</p>

{% include 'dtl_intro/includes/footer.html' %}
{% endblock %}
```

notice the **include** tag before the end of the content block with the path to the **footer.html**

We will do the same for the file: **templates/dtl_intro/no-header.html**

```
{% extends 'dtl_intro/base.html' %}

{% block navbar %}
{% endblock %}
```

```
{% block content %}
  <h1>
     No header template
  </h1>

  {% include 'dtl_intro/includes/footer.html' %}
{% endblock %}
```

run your app and you should see the footer displayed in the two pages we added.
note that the included template will be with the context of the page we are loading, you can disable it by adding an attribute of **only** you can also pass data to the template as attribute.

if - condition in template

DTL gives us the ability to branch our template and add condition to our html.
The syntax for the if tag is:

```
{% if condition %}
        some template code…
{% elif condition2 %}
        some template code…
{% else %}
        other conditions are not met…
{% endif %}
```

the **elif** and **else** is optional and you can also use **and or** and **operators** similar to python in the conditions.

Let's add a simple condition in the homepage. modify the **index** method in the **views.py** to look like this:

```
def index(request):
  return render(request, 'dtl_intro/intro.html', context={
     'greeting': 'hello from greeting variable',
     'obj': {'greeting': 'we can also pass classes and objects'},
     'func': lambda: 'hello from function',
     'condition_sample': True
  })
```

notice that we added a boolean variable to the context.
Now modify the **intro.html** template and add the following before the include of the footer:

```
{% if condition_sample %}
  <p>
     Condition sample variable is true
  </p>
{% else %}
  <p>
     Condition sample var is false
  </p>
{% endif %}
```

adding condition to our template is really simple

For loop

let's see how we can iterate on a list of items in our template. The syntax for loops in **DTL** is:

```
{% for item in list %}
        ...we can now print item {{item}}
{% endfor %}
```

let's try and and add a list in our **views.py** change the **index** method in that file to this:

```
def index(request):
  return render(request, 'dtl_intro/intro.html', context={
     'greeting': 'hello from greeting variable',
     'obj': {'greeting': 'we can also pass classes and objects'},
     'func': lambda: 'hello from function',
     'condition_sample': True,
     'list_sample': ['item1', 'item2', 'item3']
  })
```

notice that we added a variable called **list_sample** to the template.
Now in the template **intro.html** add the following before the footer include.

```
<ul>
  {% for item in list_sample %}
      <li>
         {{ item }}
      </li>
  {% endfor %}
</ul>
```

Filters

With filters we can take data and manipulate it to present it differently to the user. The general syntax of using a filter is:
{% data | filter:arg %}

you can also chain filters together to pass the data to the next filter

{% data | filter1 | filter2 %}

you can also use **filter** in a block:

{% filter lower %}
THIS TEXT WILL BE LOWER CASED
{% endfilter %}

date filter

displays a date in a certain format:

{% some_date | date:”j/n/Y %}

will display the date in format of **day/month/year**

default

if expression evaluates to **False** then will display the default value:

{% value | default:“display this string if value evaluates to false” %}

dictsort

takes a dictionary and sort it by a key

{% some_dict | dictsort:”title” %}

if dictionary contains a title key it will sort it by that key

random

returns a random item from a list

{% some_list|random %}

Custom Filter

Let's try and create our own custom filter.
Our filter will be called **zigzag** and it will get a data of a string and will uppercase lowercase each letter the first uppercase the second lowercase and so on.

Out filters has a conventional place they should be placed inside our app, **Django** will look in that place when it sees the **load** tag**.** the place we need to put our filters is a package in our app called **templatetags**. In your **dtl_intro** package, create another package called: **templatetags**.

The name of the python file you place in this package is important and should be identical to the name you use in your template with the **load** tag to load the filter. It's recommended to prefix the name of the file with the app name so it won't collide with any existing custom filters or tags of another app. Create a file in the **templatetags** directory called: **dtl_intro_filters.py** with the following code:

```
from django import template

register = template.Library()

def zigzag(value):
  result = ''
  for i in range(0, len(value)):
      if i % 2 == 0:
          result += value[i].upper()
      else:
          result += value[i].lower()
  return result


register.filter('zigzag', zigzag)
```

Our filter is a simple function, that gets as first argument the data passed to our filter, and the second argument as the param passed to the filter.

The function iterates on the string we are getting and creating from it another string with upper and lower cases. Note that in order for our filter to be registered we have to create an instance of **template.Library** and call the **register** method of that instance.

Now to use the filter we just created, in the **intro.html** template before the **include** tag add the following:

```
<h2>
  Custom filter
</h2>
{% load dtl_intro_filters %}

{{ greeting | zigzag }}
```

launch your app and see the result.
Custom filters are really easy way to customize what the user sees in the template.

CSRF

Let's try an understand what **CSRF** is and how we can easily deal with it in **Django** applications. **CSRF** stands for **Cross Site Request Forgery**. The security problem best be demonstrated with an example.
Let's say you have a form in your application that the user enters credit card details. The form sends a **POST** request to a url in your application with the data from the user. Everything is working well and your app is popular and a lot of users are shopping and using the form you created. Your app is so popular that a hacker is trying to infect your users with a maleware that when they get into the payment form they are directed to a malicious site that the hacker wrote with a form that also sends a **POST** request back to our site to the same url like our original form. That is exactly what the tag **csrf_token** will easily protect us from. Notice in the **settings.py** file in the **MIDDLEWARE** array by default there is a **CsrfViewMiddleware** already activated for us by default when we started a new project.

Now let's try and create a **POST** form in our **intro.html** file place the following code before the **include** tag:

```
<h2>
  CSRF
</h2>

<form method="post">
  <div>
    <label>Username</label>
```

```
    <input type="text" name="username" />
  </div>
  <div>
    <label>Password</label>
    <input type="password" name="password" />
  </div>
  <div>
    <button type="submit">Login</button>
  </div>
</form>
```

this will send a post request to the same url we are in, meaning to the **index** function in our **views.py**.
Modify the **views.py** file **index** method to look like this:

```
def index(request):
  if request.method == 'POST':
    return HttpResponse('form posted')
  return render(request, 'dtl_intro/intro.html', context={
    'greeting': 'hello from greeting variable',
    'obj': {'greeting': 'we can also pass classes and objects'},
    'func': lambda: 'hello from function',
    'condition_sample': True,
    'list_sample': ['item1', 'item2', 'item3']
```

Try your application and try to submit the form we just created.
Our **index** method is not even called and we are getting a missing **csrf token** message.
This means that the form we just created is not **CSRF secure** , so to secure it all we need to do is change the template and add the **csrf_token** tag. Modify **intro.html** and change the form you created to this:

```
<h2>
  CSRF
</h2>

<form method="post">
  {% csrf_token %}
  <div>
    <label>Username</label>
    <input type="text" name="username" />
  </div>
  <div>
    <label>Password</label>
```

```
    <input type="password" name="password" />
  </div>
  <div>
    <button type="submit">Login</button>
  </div>
</form>
```

notice that we added the **csrf_token** tag right at the beginning of the form.
reload your app and try to submit the form again.
Now the request should reach the **index** method and you should see the message **form posted**.
If we have an implementation of an **ajax** request we have to deal with **csrf** as well. For ajax request it's common to set the **X-CSRF** header of your request with the **csrf** token. notice that **Django** automatically by using the **csrf** middleware placed a cookie with the **csrf token** you need to grab that token and set it as a header in your ajax requests, otherwise you will get a csrf error on every ajax call.

Custom Tags

Django let's you build your own tags that you can use in your templates. We will now create a tag called **search**. This tag will get a list of strings and it will get a search string and the tag will output unordered list with all the items in the list that contains the search string.
For example if we have the list:

```
[
  "hello world",
  "yariv said hello",
  "foo bar",
  "hello yariv"
]
```

activating the filter on the above list:
{% search list "hell" %}

Should output the following:

```
<ul>
  <li>
    hello world
  </li>
  <li>
    yariv said hello
  </li>
```

```
    <li>
        hello yariv
    </li>
</ul>
```

our tag should also be placed in the **templatetags** package so create a file called **dtl_intro_customtags.py** in that package with the following code:

```
from django.template import Library

register = Library()

@register.inclusion_tag('dtl_intro/tags/search.html')
def search(list, q):
    return {
        'list': [x for x in list if x.find(q) != -1]
    }
```

We are creating a custom tag with a template called **search.html** attached to it, we are passing context to the search template by filtering the list with the search string located in the variable **q**.

Now in the **templates** directory inside the **dtl_intro** directory create another directory called **tags** and create a file in it called **search.html** with the following content:

```
<ul>
    {% for item in list %}
        <li>
            {{ item }}
        </li>
    {% empty %}
        <li>The list is empty</li>
    {% endfor %}
</ul>
```

now in the **intro.html** place the following code at the bottom of the file before the **include.**

```
<h2>
    Custom Tags
</h2>

{% load dtl_into_customtags %}

{% search list_for_search "hell" %}
```

the last thing we need to do is pass the **list_for_search** in the context of this template, and this is done in the **views.py** file in the **index** method:

```
def index(request):
  if request.method == 'POST':
      return HttpResponse('form posted')
  return render(request, 'dtl_intro/intro.html', context={
      'greeting': 'hello from greeting variable',
      'obj': {'greeting': 'we can also pass classes and objects'},
      'func': lambda: 'hello from function',
      'condition_sample': True,
      'list_sample': ['item1', 'item2', 'item3'],
      'list_for_search': [
          "hello world",
          "yariv said hello",
          "foo bar",
          "hello yariv"
      ]
  })
```

now reload your app and you should see the list is displayed. So a custom tag can have its own context and template and we can also have the context passed from the main template context if we need to. So the custom tags gives us more power in including template content and logic for repeated tasks.

Summary

We just witnessed how massive the **DTL** is and just touched on the main features we can do with it. From branching and for loops to customizing my own tags and filters we can do a lot and we can also create a very clean separation from our template and what the user see and interact with and with our view sections.