Digital Career Institute

Python Course - Django Web Framework - Views and Templates

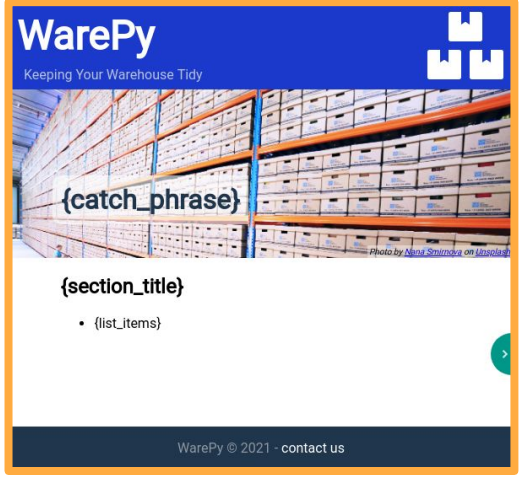
Digital Career Institute

Templates

What are they?

Templates are the final HTML,
except for the data.

TEMPLATE



+

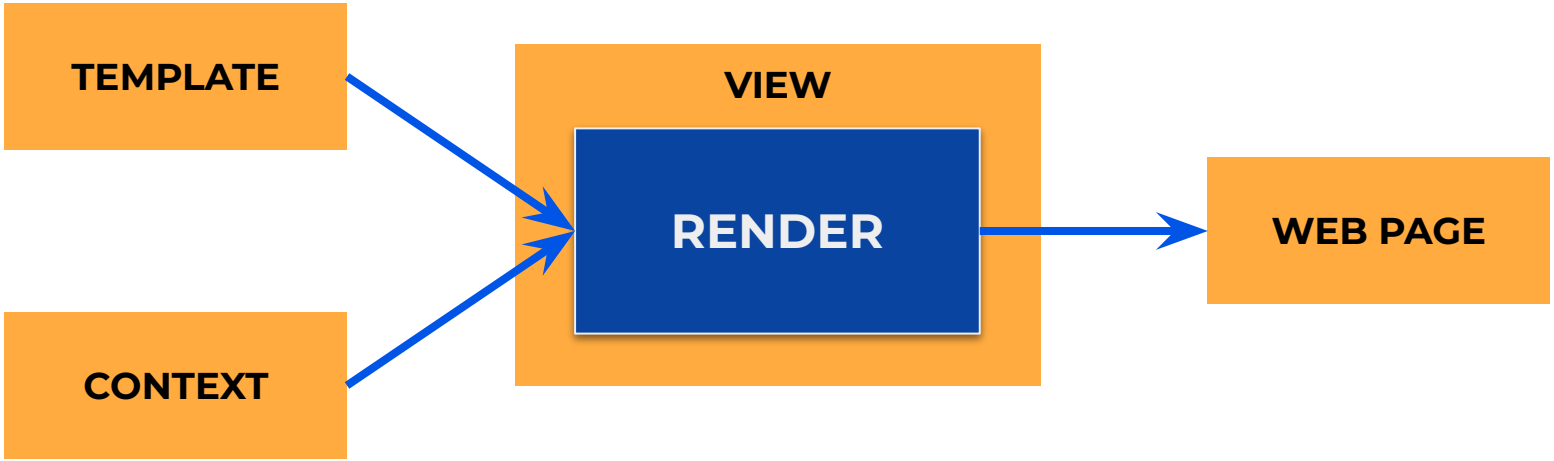
DATA

```
catch_phrase = "With WarePy
your warehouse will look
this good!"
section_title = "Feature
list"
list_items = ["A complete
role-based...", "..."]
```

=

WEB PAGE

Terminology



Templates

shop/views.py

```
from django.http import HttpResponse


def home(request):
    """The shop home view."""
    text = "Hello World!"
    if some condition:
        text = "Hey People!"
    content = f"<html><body><h1>{text}</h1></body></html>"
    return HttpResponse(content)
```

The HTML is part of the User Interface and **should not be** defined **in the view**.

Templates: Hello World

shop/views.py

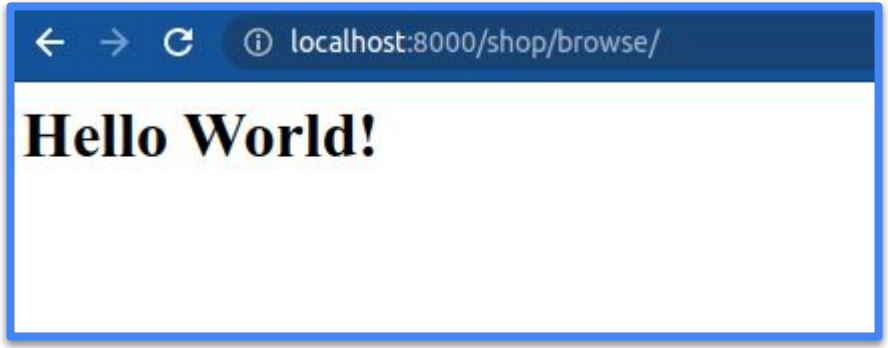
```
from django.template.loader import  get_template
from django.views import View


class Browse(View):
    template = get_template("browse_home.html")
    context = {"my_variable_name": "Hello World!")
    return HttpResponse( template.render(context))
```

A template object can be retrieved with **`get_template`** and the data can be passed as a dictionary to the **`render`** method.

Templates: Hello World

shop/templates/browse_home.html

```
<html>
<head>
  <title>Browse the shop!</title>
</head>
<body>
  <h1>{{ my_variable_name }}</h1>
</body>
</html>
```



The code in **{{ }}** is a special templating language used to embed content in the template.

Template File Location

```
+ shop
  + migrations
  + templates
    - browse_home.html
  - __init__.py
  - admin.py
  - apps.py
  - models.py
  - tests.py
  - views.py
```

hello/settings.py

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

For Django to find an app template, the app must be added to **`INSTALLED_APPS`** in **`settings.py`**.

Templates & Base Views

shop/views.py

```
from django.views.generic.base import  TemplateView


class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"my_variable_name": "Hello World!")
```

A simpler code can be used with the **`TemplateView`** base view.

Templates & Base Views

shop/views.py

```
from django.views.generic.base import TemplateView


class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self, url_argument):
        return {"my_variable_name": "Hello World!")
```

URL arguments are automatically passed to **`get_context_data`** if the path includes parameters.

Templates & Base Views

shop/views.py

```
from django.views.generic.base import TemplateView


class Browse(TemplateView):
    template_name = "shop/browse_home.html"
    def get_context_data(self):
        return {"my_variable_name": "Hello World!")
```

The templates may be organised in subdirectories and they will need to be included as part of the **`template_name`**.

To avoid naming conflicts, it is common practice to put the html file in a directory with the same name as the app.

Template Language Overview

shop/templates/browse_home.html

```
<body>
  {% if my_variable_name %}
    <ul>
        {% for item in items %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>
  {% endif %}
</body>
```

The template language is a language on its own.

Some basic control flow structures can be used, wrapping them in `{% %}`.

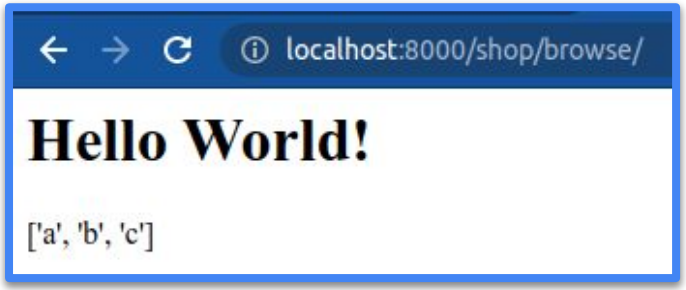
Writing the value of a variable into the HTML can be done with `{{ }}`.

Template Variables

shop/templates/browse_home.html

```
<body>
  <h1>{{ my_variable_name }}<h1>
  <p>{{ list }}</p>
</body>
```

Template **variables** output content into the template. They are wrapped in **{{ }}**.



The output is always converted into a string.

Template Variables

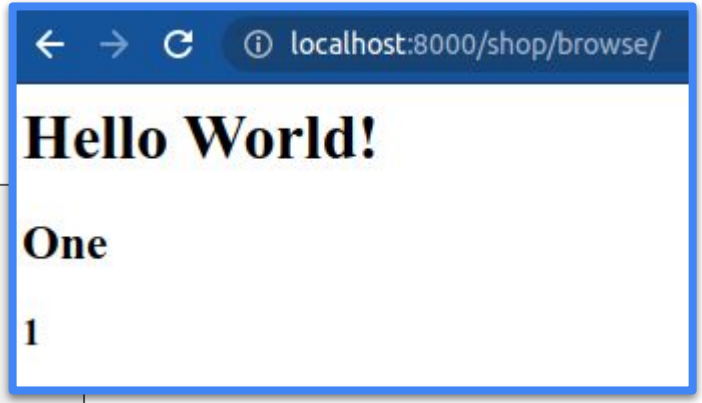
shop/templates/browse_home.html

```
<body>
  <h1>{{ text }}<h1>
  <h2>{{ items.0 }}</h2>
  <h3>{{ dic.one }}</h3>
</body>
```

The dot (.) can be used to access elements in lists and dictionaries.

shop/views.py

```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"text": "Hello World!",
                "items": ["One", "Two"],
                "dic": {"one": 1, "two": 2}}
```



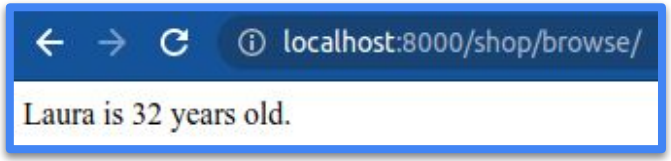
Template Variables

shop/templates/browse_home.html

```
<body>
  {{ user.name }} is {{ user.age }} years old.
</body>
```

shop/views.py

```
class User:
    name = "Laura"
    def age:
        return 32

class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return { "user": User() }
```



The dot (.) can also be used to access object **properties and methods**.

Template Variables

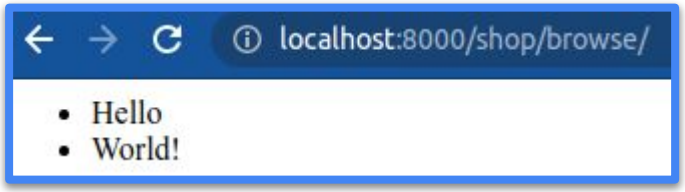
shop/templates/browse_home.html

```
<body>
  <ul>
    {% for word in text.split %}
      <li>{{ word }}</li>
    {% endfor %}
  </li>
</body>
```

shop/views.py

```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return { "text": "Hello World!" }
```

The same can be done with built-in data type methods.

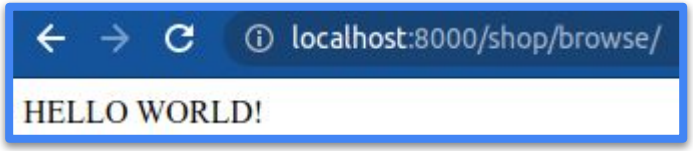


(Template Variable) Filters

shop/templates/browse_home.html

```
<body>
  {{ text|upper }}
</body>
```

Filters can be used to modify the output of a variable.



shop/views.py

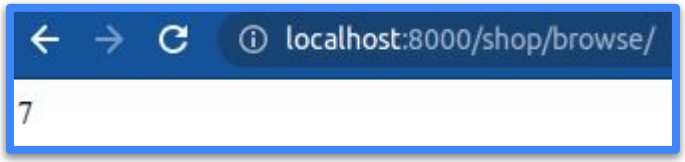
```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"text": "Hello World!"}
```

Filters

shop/templates/browse_home.html

```
<body>
  {{ number|add:2 }}
</body>
```

Filters are functions and can also accept arguments, by using a colon (:).



shop/views.py

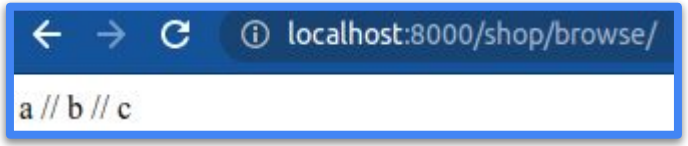
```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"number": 5}
```

Built-in Filter Examples: Join Lists

shop/templates/browse_home.html

```
<body>
  {{ list|join:" // " }}
</body>
```

The **join** filter replicates the behavior of the **join** method in **str** objects.



shop/views.py

```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"list": ["a", "b", "c"]}
```

Built-in Filter Examples: Unescaping

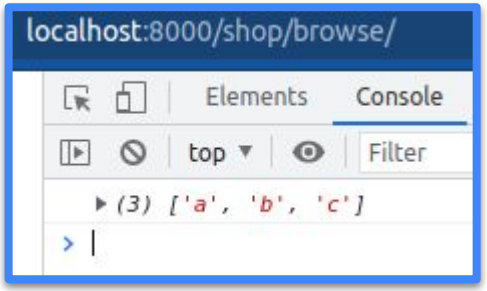
shop/templates/browse_home.html

```
<script>
  const array = {{ list |safe }};
  console.log(array);
</script>
```

Data can be embedded into the front-end JavaScript using the **safe** filter.

shop/views.py

```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"list": ["a", "b", "c"]}
```



Built-in Filter Examples: Unescaping

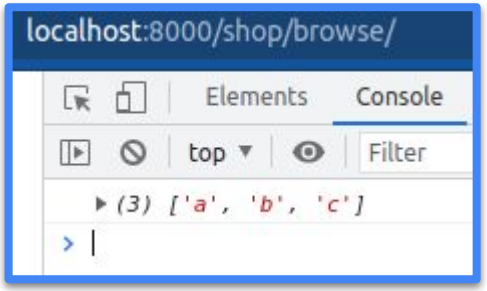
shop/templates/browse_home.html

```
<script>
  const array = {{ list |safe }};
  console.log(array);
</script>
```

Data can be embedded into the front-end JavaScript using the **safe** filter.

shop/views.py

```
class Browse(TemplateView):
    template_name = "browse_home.html"
    def get_context_data(self):
        return {"list": ["a", "b", "c"]}
```



Built-in Filters: And More ...

- add
- addslashes
- capfirst
- center
- cut
- date
- default
- default_if_none
- dictsort
- dictsortreversed
- divisibleby
- escape
- escapejs
- filesizeformat
- first
- floatformat
- force_escape
- get_digit
- iriencode
- join
- json_script
- last
- length
- length_is
- linebreaks
- linebreaksbr
- linenumbers
- ljust
- lower
- make_list
- phone2numeric
- pluralize
- pprint
- random
- rjust
- safe
- safeseq
- slice
- slugify
- stringformat
- striptags
- time
- timesince
- timeuntil
- title
- truncatechars
- truncatechars_html
- truncatewords
- truncatewords_html
- unordered_list
- upper
- urlencode
- urlize
- urlizetrunc
- wordcount
- wordwrap
- yesno

Template Tags

shop/templates/browse_home.html

```
<body>
  <ul>
    {% if items %}
      {% for item in items %}
        <li>{{ item }}</li>
      {% endfor %}
    {% else %}
      <li>No items found.</li>
    {% endif %}
  </ul>
</body>
```

Template tags are wrapped in `{% %}`

They are more complex than variables and can output content or perform other operations.

`if` and `for` are template tags.

Template Tag Examples: Empty Lists

shop/templates/browse_home.html

```
<body>
  <ul>
    {% for item in items %}
      <li>{{ item }}</li>
    {% empty %}
      <li>No items found.</li>
    {% endfor %}
  </ul>
</body>
```

The **`for`** tag allows for an **`empty`** clause that catches an empty list.

This produces a cleaner and better solution than the code in the previous slide.

(Built-in) Template Tag Examples: Firstof

shop/templates/browse_home.html

```
{% firstof a b c %}
```

The **firstof** tag prints the first available value.

=

```
{% if a %}
  {{ a }}
{% elif b %}
  {{ b }}
{% elif c %}
  {{ c }}
{% endif %}
```

This also produces a cleaner and better solution to this common situation.

Template Tag Examples: Commenting

shop/templates/browse_home.html

```
{% comment "Some internal note" %}
  <p>This will not be printed.</p>
{% endcomment %}
```

The **`comment`** tag allows us to comment code in a clean way.

Template Tag Examples: Including

shop/templates/browse_home.html

```
...

{% include "contact_form.html" %}
```

The **`include`** tag allows us to embed another html template.

The context data of **`browse_home.html`** *will be passed to* **`contact_form.html`**.

Template Tag Examples: App Links

shop/templates/browse_home.html

```
<a href="{% url "shop" %}">Home</a>
<a href="{% url "shop-browse" %}">All items</a>
<a href="{% url "shop-item" 1 %}">Offer!</a>
```

The **url** tag returns the URL matching the named path.

Rendered Output

```
<a href="/shop/">Home</a>
<a href="/shop/browse/">All Items</a>
<a href="/shop/1/">Offer!</a>
```

hello/urls.py

```
urlpatterns = [
    path('shop/', shop_home, name="shop"),
    path('shop/browse/', shop_browse, name="shop-browse"),
    path('shop/<item_id>/', shop_item, name="shop-item"),
]
```

Creating Custom Template Tags

shop/templates/browse_home.html

```
<h1>{% hello_world %}</h1>
```

Template tags are functions.

How do we tell Django that **this custom tag** should show the output of **this function**?

And where do we place **this file**? And how do we name it?

????/my_tags.py

```
def hello_world():
    Return "Hello World!"
```

Custom Template Tags & Filters

```
+ hello
  + hello
      - settings.py
      - urls.py
  + common
      + templatetags
          - __init__.py
          - my_tags.py
  + shop
      + templatetags
          - __init__.py
          - my_tags.py
      - views.py
  - manage.py
```

Custom template tags must be defined in a directory named **`templatetags`** inside an app directory.

The directory must have an empty **`__init__.py`** file.

The app must be in the **`INSTALLED_APPS`** settings constant.

We can have as many files as we want with any name we want.

Custom Template Tags & Filters

shop/templates/browse_home.html

```
{% load my_tags %}

<h1>{% hello_world %}</h1>
```

The file with the template tag definition must be loaded in the template, using the **`load`** built-in tag.

Multiple tags may be defined (and then loaded) in one same file.

common/templatetags/my_tags.py

```
from django import template

register = template.Library()

@register.simple_tag
def hello_world():
    return "Hello World!"
```

Template tags of general purpose may be placed in a different app.

The template tag function must be registered in the library. It can be done with the **`simple_tag`** decorator.

Custom Template Tags & Filters

shop/templates/browse_home.html

```
{% load mytags mytags2 static %}

{% chart context %}
```

Multiple template tag modules, custom and built-in, can be loaded.

Custom tags may also accept arguments.

common/templatetags/mytags2.py

```
from django import template

register = template.Library()

@register.inclusion_tag("chart.html")
def chart(context):
    return context
```

The `inclusion_tag` decorator is a shortcut for a template renderer.

Custom Template Tags & Filters

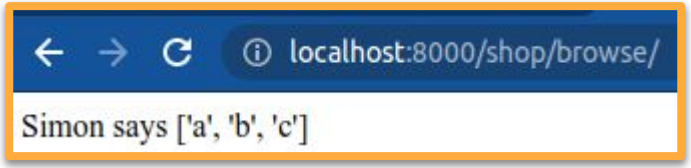
shop/templates/browse_home.html

```
{% load mytags mytags2 static %}

<h1>{{ list|simon }}</h1>
```

common/templatetags/mytags2.py

```
from django import template

register = template.Library()

@register.filter
def simon(says):
    return f"Simon says {says}."
```

Custom template filters are defined the same way as tags and placed in the same files.



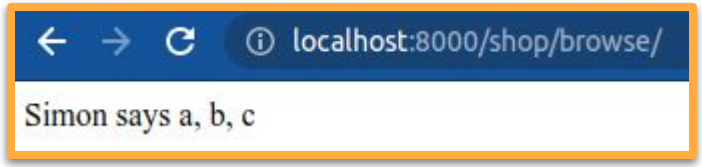
The template filter function must be registered in the library. It can be done with the `filter` decorator.

Custom Template Tags & Filters

shop/templates/browse_home.html

```
{% load mytags mytags2 static %}

<h1>{{ list|join:", "|simon }}</h1>
```

Multiple filters, custom and built-in, can be chained together.



Template Inheritance

common/templates/base.html

```
<html>
  <head><title>Hello World!</title></head>
  <body>
    <p>Common content</p>
    {% block page-content %}
      <p>Default page content</p>
    {% endblock %}
  </body>
</html>
```

Templates may be inheriting from another template.

shop/templates/home.html

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

{% block page-content %}
  <h1>Welcome to the shop!</h1>
  <p>We have some offers today.</p>
{% endblock %}
```

The **`extends`** tag tells Django to extend the **`base.html`** template.

Overrides the block named **`page-content`** in the parent template.

Template Inheritance

common/templates/base.html

```
<html>
  <head><title>Hello World!</title></head>
  <body>
    <p>Common content</p>
    {% block page-content %}
      <h1>Common title</h1>
    {% endblock %}
  </body>
</html>
```

Template inheritance is similar to class inheritance.

shop/templates/home.html

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

{% block page-content %}
  {{ block.super }}
  <h4>Welcome to the shop!</h4>
  <p>We have some offers today.</p>
{% endblock %}
```

The `block.super` variable returns the code from the parent template.

Can be used to append elements instead or replacing them.

File Strategy: Project vs. App

```
+ hello
  + hello
      - settings.py
      - urls.py
  + common
      + templates
  + shop
      + templates
  - manage.py
  + templates
```

Various template directories are often used to store app specific and project-wide templates.

Common project templates: `base.html`, `wide.html`,...

Templates specific for the `shop` app: `shop_home.html`, `shop_categories.html`, ...

It is also common practice to place project templates here.

Static URLs & Files

Static URLs

Static URLs trigger requests that **do not require any processing** on the server.

common/templates/base.html

```
<html>
<head>
  <script src="script.js"></script>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <img src="image.png">
</body>
</html>
```

http://localhost:8000/script.js

http://localhost:8000/style.css

http://localhost:8000/image.png

Using Static URLs

shop/templates/browse_home.html

```
{% load static %}

<img src="{% static 'img/my_logo.png' %}">
```

The tag **`url`** works with URL *endpoints* while the tag **`static`** works with *static* URL resources.

The **`static`** template tag, like the built-in **`url`** tag, returns a path in our website tree.

`static` returns a path to the static resources (mostly front-end resources like CSS, images and JavaScript files).

The path passed to **`static`** is directly matching the directory tree in the file system.

Static Files

shop/templates/browse_home.html

```
{% load static %}

<img src="{% static 'img/logo.png' %}">
```

Static files are often placed in a directory called **`static`**.

Each app often has its own set of static files, so it is good practice to keep them in the app directory.

There may be some common static files too, that can be kept in the **`common`** app directory.

```
+ hello
  + hello
      - settings.py
      - urls.py
  + common
      + static
          + img
              - logo.png
  + shop
      + static
          - ...
      - views.py
  - manage.py
```

Static Files in Development

While we are working on the app in development Django will search for the resource in the appropriate directory.

<u>In development it is better to keep static files organized by apps, but in production this produces some unnecessary overhead</u>, because the system needs to load the apps and search their directory trees.

It will be more efficient if we merge all the static files together and have a direct match between paths and a single root in the server's file system.

```
+  hello
   + hello
       - settings.py
       - urls.py
   + common
       + static
   + shop
       + static
       - views.py
   - manage.py
```

Static Files in Production

In production all the app static files will be merged into a common root directory so the web server can be optimized for.

This directory is often placed in the root directory of the project and named static, but it can be placed anywhere in the file system that is accessible by the web server.

It's location needs to be indicated in the `settings.py` file with the name `STATIC_ROOT`.

Django's `collectstatic` command will merge all static files into our `STATIC_ROOT` directory:

```
(env) $ python manage.py collectstatic
```

```
+ hello
  + hello
      - settings.py
      - urls.py
  + common
      + static
  + shop
      + static
      - views.py
  + static
  - manage.py
```

Static File Configuration Example

hello/settings.py

```
...

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/...

STATIC_URL = 'static/'
STATIC_ROOT = os.path.join(BASE_DIR, STATIC_URL)

...
```

All URLs produced with the **`static`** template tag will start always with this path.

If we place our production directory in the root of the project (**`BASE_DIR`**) this will get its subdirectory **`static`**.

Static Files: Path Collisions

shop/templates/browse_home.html

```
{% load static %}

<img src="{% static 'logo.png' %}">
```

blog/templates/blog_home.html

```
{% load static %}

<img src="{% static 'logo.png' %}">
```

```
+ hello
  + hello
  + shop
      + static
          - logo.png
  + blog
      + static
          - logo.png
  - manage.py
```

Both tags will produce **`/static/logo.png`**.

How does Django know which `logo.png` it needs to show?

Static Files: Path Collisions

shop/templates/b

```
{% load static %
<img src="{% sta
```

blog/templates/bl

```
{% load static %
<img src="{% sta
```

```
(env) $ python manage.py collectstatic
...
Found another file with the destination path
'styles.css'. It will be ignored since only
the first encountered file is collected. If
this is not what you want, make sure every
static file has a unique path.
```

Static files must have a **unique path**.

Static Files: Path Collisions

shop/templates/browse_home.html

```
{% load static %}

<img src="{% static 'shop/logo.png' %}">
```

blog/templates/blog_home.html

```
{% load static %}

<img src="{% static 'blog/logo.png' %}">
```

```
+ hello
  + hello
  + shop
      + static
          - logo.png
  + blog
      + static
          - logo.png
  - manage.py
```

It is common practice to namespace the static paths with the app name.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute