

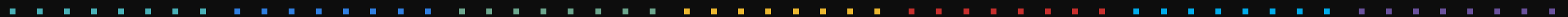
WMNBE-2203 | BACK-END DEVELOPMENT

Flask #2

Templates | Static assets



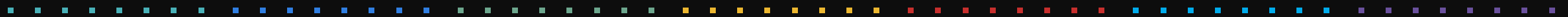
ΠΕΡΙΕΧΟΜΕΝΑ



Περιεχόμενα

- **Jinja2**
 - Control structures
 - Template Hierarchy
- **URL** generation
- Error pages

TEMPLATES



Templates

HTML rendering

Αν και μπορούμε να επιστρέφουμε **HTML** απευθείας από τα *view functions* του **Flask**, δεν είναι ιδιαίτερα πρακτικό.

```
@app.route('/hello/<username>')
def hello(username):
    return f'''
        <html>
        <head>
            <title>Hello</title>
        </head>
        <body>
            <h1>Hello {username}!</h1>
        </body>
        <html>
    '''
```

Templates

Template engine

Το **Flask** κάνει χρήση της *template engine* **Jinja2** η οποία ρυθμίζεται αυτόματα από το ίδιο το framework.

Για να γίνει *render* ένα *template* (πρότυπο) αρκεί να κληθεί η συνάρτηση **render_template()**.

```
from flask import Flask, render_template
...
@app.route('/hello/<username>')
def hello(username):
    return render_template('hello.html', username=username)
```

Templates

Folder structure

Τα *templates*, για να μπορέσει να τα βρει η μηχανή, πρέπει να βρίσκονται σε φάκελο **templates** ο οποίος πρέπει να βρίσκεται "δίπλα" στην *Flask* εφαρμογή.

Ομοίως τα *static* αρχεία πρέπει να βρίσκονται σε φάκελο **static**.

```
/app.py
/templates
  /hello.html
/static
  /css
    /style.css
  /js
    app.js
```

JINJA2

Jinja2

Jinja2

Η μηχανή **Jinja2** είναι ιδιαίτερα δυνατή και έχει τη δική της σύνταξη, που στα περισσότερα σημεία θυμίζει **Python**.

```
<html>
...
<body>
  <h1>Hello
    {% if username %}
      {{ username }}!
    {% else %}
      world!
    {% endif %}
  </h1>
</body>
</html>
```

Jinja2

Tags/Placeholders/Delimiters

Για την παρεμβολή "κώδικα" **Jinja2** μέσα στο πρότυπο, γίνεται χρήση των παρακάτω **tags**:

- **{% ... %}** για εντολές
- **{{ ... }}** για εκφράσεις (οι οποίες θα "τυπωθούν")
- **{# ... #}** για σχόλια (μη ορατά στο τελικό αποτέλεσμα)

Παρόλο που μπορούμε να έχουμε αρκετά σύνθετες εκφράσεις και εντολές μέσα στα ίδια τα *templates*, καλό είναι να το αποφεύγουμε.

Κρατάμε στα *templates* ότι έχει να κάνει με την εμφάνιση και τα υπόλοιπα ανήκουν μέσα στα *view functions*.

CONTROL STRUCTURES

Control structures

if

Για τον έλεγχο περιπτώσεων

```
<div>
{% if user %}
  <span>{{ user.username }}</span>
{% else %}
  <a href="/login">login</a>
{% endif %}
</div>
```

Control structures

for

Για βρόχο πάνω σε στοιχεία/αντικείμενα

```
<h1>Products</h1>
<ul>
{% for product in products %}
  <li>{{ product.title }}</li>
{% endfor %}
</ul>
```

Control structures

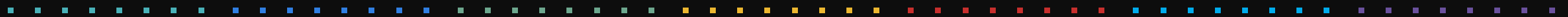
macro

Για την επαναχρησιμοποίηση τμημάτων

```
{% macro input(name, value='', type='text', size=20) %}
    <input type="{{ type }}" name="{{ name }}"
        value="{{ value }}" size="{{ size }}">
{% endmacro %}

<p>{{ input('username') }}</p>
<p>{{ input('password', type='password') }}</p>
```

FILTERS



Filters

Built-in Filters

Μέσα σε ένα **Jinja2 statement** μπορούν να χρησιμοποιηθούν εντολές και μέθοδοι της **Python**, για να διαμορφώσουμε κατάλληλα το παραγόμενο αποτέλεσμα.

```
{{ page.title.capitalize() }}
{{ "Hello, {}".format(name) }}
```

Παρόλα αυτά είναι προτιμότερη η χρήση των ενσωματωμένων φίλτρων, με τη χρήση του χαρακτήρα **|**, που είναι επαρκής για τις περισσότερες συνηθισμένες λειτουργίες.

```
{{ page.title|capitalize }}           {# This is nicer #}
{{ "%s, %s!"|format(greeting, name) }} {# This one not so much #}
```


Filters

HTML Escaping

Δύο από τα πιο χρήσιμα φίλτρα, βρίσκονται στο *module MarkupSafe*.

e: κάνει *escape* το *string* που θα του δοθεί, δηλαδή αντικαθιστά τους "ύποπτους" χαρακτήρες (**>**, **<**, **&**, **"**) με ασφαλείς.

safe: επισημαίνει ένα *string* ως ασφαλή, ώστε να μην γίνουν σε αυτό αντικαταστάσεις χαρακτήρων.

Η χρήση του **e** είναι, κυρίως, για την αποφυγή επιθέσεων, όπως **XSS** κ.λπ. Είναι αναγκαίο μόνο αν έχει ενεργοποιηθεί το **manual escaping**.

Το **safe** χρησιμοποιείται όταν κάποια έκφραση παράγει, με βεβαιότητα, ασφαλή κώδικα **HTML** και θέλουμε να εμφανίσουμε το περιεχόμενο ως έχει.

Filters

Custom Filters

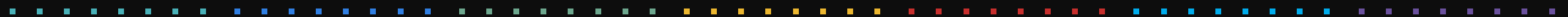
Η χρησιμότητα των φίλτρων φαίνεται, ιδιαίτερα, στη δυνατότητα ορισμού custom φίλτρων.

```
def format_currency(value):
    locale.setlocale(locale.LC_ALL, 'el_GR')
    return locale.currency(value,
                            symbol=True,
                            grouping=True)

filters.FILTERS["format_currency"] = format_currency
```

```
<span>{{ product.price|format_currency }}</span>
```

TEMPLATE HIERARCHY



Template Hierarchy

block

Για την καλύτερη οργάνωση των *templates*, συχνά χρειαζόμαστε ένα βασικό πρότυπο στο οποίο θα βασίζονται κάποια από/όλα τα υπόλοιπα.

Σε αυτό θα βρίσκονται τα κοινά στοιχεία μεταξύ των διαφόρων *views* ώστε να μην επαναλαμβάνονται.

Για να το επιτύχουμε αυτό χρησιμοποιούμε το *tag block* και την "εντολή" **extends**.

Template Hierarchy

block

Στο βασικό *template* ορίζουμε περιοχές στις οποίες μπορούν τα υπόλοιπα *templates* να παρεμβάλουν το δικό τους περιεχόμενο.

Αυτές οι περιοχές μπορούν να περιέχουν ήδη κάποιο *default* περιεχόμενο ή και όχι.

Ένα **block** μπορεί να οριστεί και ως **required**.

Template Hierarchy

Base Template

```
<html>
<head>
  <title>{% block title %}{% endblock %} - Site</title>
</head>
<body>
  <div id="main">
    {% block main required %}{% endblock %}
  </div>
  <div id="footer">
    {% block footer %}
    &copy; Copyright {{ year }}.
    {% endblock %}
  </div>
</body>
</html>
```

Template Hierarchy

Child Template

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

{% block title %}Home page{% endblock %}

{% block main %}
    <h1>This is the home page</h1>
{% endblock %}
```

Template Hierarchy

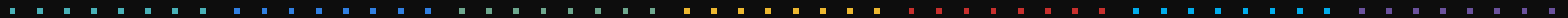
include

Στην οργάνωση των *templates* βοηθά και η χρήση της εντολής **include**.

Ο ρόλος της είναι προφανής, μας επιτρέπει την ενσωμάτωση άλλων αρχείων στο *template*.

```
<body>
  {% include "_header.html" %}
  {% block main required %}{% endblock %}
  {% include "_footer.html" %}
</body>
```


URL GENERATION



URL generation

url_for()

Μέσα στα *templates*, αλλά και στα *view functions*, συχνά αναφερόμαστε σε *URLs* που αφορούν, είτε σε κάποιο *route/view function*, είτε σε κάποιο *static asset*.

Αν και μπορούμε να ορίσουμε αυτά τα *URLs* εμείς, καλό είναι να τα δημιουργούμε μέσω της κατάλληλης *function* του **Flask**.

```
# for view functions
url_for('view_function_name', param1=value1, ...)

# for static assets
url_for('static', filename='...')
```

Παράδειγμα

URL generation

```
<html>
  <head>
    ...
    <link rel="stylesheet"
          href="{{ url_for('static', filename='css/style.css') }}">
  </head>
  <body>
    ...
    <ul>
      {% for key, product in products.items() %}
      <li>
        <a href="{{ url_for('product_details', sku=key) }}">
          {{ product.title }}
        </a>
      </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

URL generation

redirect

Στα πλαίσια ενός *view function* η **url_for** είναι χρήσιμη στην περίπτωση που πρέπει να γίνει ανακατεύθυνση σε άλλη διεύθυνση.

Για την ανακατεύθυνση θα γίνει χρήση της συνάρτησης **redirect**.

Προσοχή η **redirect** χρησιμοποιείται "πάντα" με την εντολή **return**, όλες οι *view function* **πρέπει** να επιστρέφουν κάτι.

```
return redirect(url_for('view_function_name',
                        param1=value1,
                        ...))
```

Παράδειγμα

URL generation

```
@app.route('/products/<sku>')
def product_details(sku):
    products = get_products()
    product = products.get(sku, None)

    if product is not None:
        return render_template('products/details.html',
                               product=product)
    else:
        return redirect(url_for('product_list'))
```

ERROR PAGES

Error pages

abort

Πέρα από την ανακατεύθυνση σε άλλο *view* μπορούμε, με κώδικα, να ανακατευθύνουμε το χρήστη σε σελίδα σφάλματος.

Σε αυτή την περίπτωση θα γίνει χρήση της συνάρτησης **abort**.

Η συνάρτηση **abort** του **Flask module** δέχεται, ως όρισμα, το κατάλληλο *HTTP status code*.

```
return abort(error_code)
```

Παράδειγμα

Error pages

```
@app.route('/profile/<username>')
def profile(username):
    user = get_user(username)

    if user is not None:
        return render_template('users/profile.html',
                               user=user)
    else:
        return abort(404)
```


Error pages

errorhandler

Τι είναι πιο σωστό, να εμφανίσω ένα σφάλμα, π.χ. **404**, ή να ανακατευθύνω σε μια *custom* σελίδα λάθους;

Μπορούν να γίνουν και τα δύο, με τη συνάρτηση **abort()** επιστρέφω το επιθυμητό σφάλμα και με τον *decorator* **errorhandler** αναθέτω τη διαχείριση των σφαλμάτων σε κατάλληλα *views*.

```
@app.errorhandler(404)
def page_not_found(e):
    app.logger.debug(e)
    # note that we set the 404 status explicitly
    return render_template('errors/404.html'), 404
```

Χρήσιμα links


🔗 Quickstart — Flask Documentation (2.1.x)
<https://flask.palletsprojects.com/en/2.1.x/quickstart/>

📖 Template Designer Documentation — Jinja
Documentation (3.1.x)
<https://jinja.palletsprojects.com/en/3.1.x/templates/>

🌐 Primer on Jinja Templating – Real Python
<https://realpython.com/primer-on-jinja-templating/>

📖 Jinja2 Tutorial - Part 1 - Introduction and variable
substitution |
<https://ttl255.com/jinja2-tutorial-part-1-introduction-a...>

Extra info

 List of HTTP status codes - Wikipedia
https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

 List of Builtin Filters — Jinja Documentation (3.1.x)
<https://jinja.palletsprojects.com/en/3.1.x/templates/#...>

THANK YOU!

