Modern Web Application Framework
Python, SQL Alchemy, Jinja2 & Flask

Devert Alexandre
Table of Contents

1 Model-View-Controller

2 Flask

3 First steps

4 Routing

5 Templates
   Basic template rendering
   Using ressources
   Template inheritance
   Template macros
   Template language

6 Requests
Model-View-Controller

Most of the modern web development frameworks follow the Model-View-Controller model (MVC model)

- The *model* : representation of data. Usually, have a strong relation with the database
- The *views* : what is shown to the user. Can be any kind of user interface, usually HTML pages with Javascript.
- The *controls* : what operation are done on the data.

It’s a rather convenient way to design software projects involving user interfaces presenting and manipulating data.
Model-View-Controller
Model-View-Controller

Example for Model-View-Controller: an online management game

- The rule of the game, updating the state of each player ⇒ the model
- The HTML pages, showing the various screen of the game ⇒ the views
- The methods called when a user click on the screen ⇒ the controllers
Model-View-Controller

Example for *Model-View-Controller*: an online shop

- The list of products, the payment rules, delivery orders ⇒ the model
- The HTML pages, showing the various screen of the shop ⇒ the views
- The methods for payment, order, shopping cart ⇒ the controllers
Model-View-Controller

*Model-View-Controller* also helps to organize the work

- Some work on the views ⇒ graphic designers, HTML, javascript
- Some work on the model ⇒ database, software architecture
- Some work on the controls ⇒ rather low-level and/or specialized code
- Some work on writing unit tests for at least the model and the views
Table of Contents

1 Model-View-Controller
2 Flask
3 First steps
4 Routing
5 Templates
   Basic template rendering
   Using ressources
   Template inheritance
   Template macros
   Template language
6 Requests
Web application with script language

Why using a scripting language for a web application?

- More adapted language to paste together various components (database, rendering, routing, ...)  
- Make it easier to release early & often  
- Easier to maintain & modify  
- Speed far enough for many use cases
Web application with script language

Why not PHP, or PHP framework?

- Designed to make simple web pages, not large web applications
- Awfully designed programming language
- very inconsistent libraries
- very little help for debugging
- many security issues
- many better alternatives

Detailed explanation here
http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design
Web application with script language

Why not using Java/JSP/JBoss/Apache/Hibernate/Spring?

- Even simple changes requires lots of coding
- Big changes takes a lot of planning
- Edit/Compile/Run takes more ressource
- General speed of development much reduced
- Working without a big fat IDE is tedious

But you can use those all this with a script-like language: Grails and Groovy
Flask

I am going to introduce the framework *Flask*

- It is small: quick to learn and master
- It is complete: you can use to do serious apps
- It is lean: a shell and a text editor are enough, no need for an IDE to be efficient with it
- It is very well documented

The same ideas can be found in most web development frameworks.
Flask

Flask is a nice glue around existing tools

- *Python* ⇒ programming language
- *SQL Alchemy* ⇒ database
- *Jinja2* ⇒ HTML template system
- *Werkzeug* ⇒ WSCGI handling (CGI, but better)
Table of Contents

1 Model-View-Controller
2 Flask
3 First steps
4 Routing
5 Templates
   Basic template rendering
   Using resources
   Template inheritance
   Template macros
   Template language
6 Requests
Hello, world!

A minimal Flask application

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello World!

if __name__ == '__main__':
    app.run()
```

Run this, and open your web browser at

http://127.0.0.1:5000
Hello, world!

You will see this

Hello World!
Hello, world!

This creates an *application* instance and run it

```python
from flask import Flask
app = Flask(__name__)

if __name__ == '__main__':
    app.run()
```
Hello, world!

This adds the *hello* method to the application instance

```python
@app.route('/

def hello():
    return 'Hello World!'
```

- *hello()** will be called every time the address /* is requested
- *hello()** returns the text data for the web browser
Debugging

Triggering the debug mode is easy

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(debug = True)
```

In debug mode, you can edit the code while the server runs: it will restart!
Debugging

The debug mode will also help a lot to point where the problem is.

```
TypeError: cannot concatenate 'str' and 'int' objects
```

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

You can execute arbitrary Python code in the stack frames and there are some extra helpers available for introspection:

- `dump()` shows all variables in the frame
- `dump(obj)` dumps all that's known about the object
Table of Contents

1. Model-View-Controller
2. Flask
3. First steps
4. Routing
5. Templates
   - Basic template rendering
   - Using ressources
   - Template inheritance
   - Template macros
   - Template language
6. Requests
Function / URL mapping

When an URL is requested, Flask will look for its corresponding function.

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Index Page'

@app.route('/welcome')
def hello():
    return 'Hello World'

if __name__ == '__main__':
    app.run()
```

One function return text data. It can be HTM, XML, JSON, etc.
Function / URL mapping

You can defines URL with parameters

```python
@app.route('/show_name/<name>,'

def print_name(name):
    return 'Hello, %s!' % name
```

It gives a nice way, intuitive way to define ressources on a website.
Function / URL mapping

You can make URL parameters optional

```python
@app.route('/hello/')
@app.route('/hello/<name> ')
def hello(name = None):
    if name is None:
        return 'A horse with no name'
    else:
        return 'A horse named %s' % name
```
Function / URL mapping

You can enforce the type of a parameter

```python
@app.route('/team/<int:team_id>')</def show_team(team_id):
    return 'team #%d' % team_id
```

Flask will check the type for you
Function / URL mapping

You can translate function names to URL with `url_for()`

```python
@app.route('/',)
def welcome():
    return 'Hello World!'

@app.route('/test')
def test():
    name = 'welcome'
    return 'url_for"%s"=_="%s" % (name, url_for(name))
```

Especially convenient when you might have to change the URL naming scheme
Function / URL mapping

`url_for()` also works for URL with parameters

```python
@app.route('/show_name/<name>')
def print_name(name):
    return 'Hello, %s!' % name

@app.route('/test')
def test():
    func_name, user_name = 'print_name', 'Alex'
    return 'url_for "%s" = "%s"' % (func_name, url_for(func_name, name = user_name))
```
## Catching HTTP errors

The HTTP protocol defines several status codes.

<table>
<thead>
<tr>
<th>status code</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td>402</td>
<td>Payment Required</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
</tr>
<tr>
<td>501</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>503</td>
<td>Service Unavailable</td>
</tr>
</tbody>
</table>
Catching HTTP errors

Using `@errorhandler`, you can catch such errors

```python
@app.errorhandler(403)
def page_forbidden(error):
    print 'Hey! You are not allowed to access this!'

@app.errorhandler(404)
def page_not_found(error):
    print 'No no! The resource you want to access does not exist:'
```
Throwing HTTP errors

It is also possible to throw HTTP errors with *abort*

```python
@app.route('/show_account_infos')
def show_account_infos():
    if not user.logged_in:
        abort(401)

    # Do things ...
```

For instance, an error 401 to deny access to resources
Table of Contents

1 Model-View-Controller
2 Flask
3 First steps
4 Routing
5 Templates
   Basic template rendering
   Using resources
   Template inheritance
   Template macros
   Template language
6 Requests
The need for templates

Generating HTML directly with code
- Easy to make very hard to read code
- Mix-up the control code with the view code

Text template system is a convenient and common way to separate the view code from the remaining code
The need for templates

Flask uses Jinja2 as template system. There are many others template system

- Mako, for Python (if you ask me, it’s better than Jinja2)
- JSP, for Java, THE standard for Java. Allow to mix Java & HTML.
- ASP, for Microsoft products. Allow to mix VBScript & HTML.
- XSLT is a template system based on XML. Plateform indepedent but not very convenient in practice.
- Maybe 10 different for every language you can think of
Basic template rendering

The function `render_template` takes a path to an HTML file, and arbitrary parameters

```python
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name = None):
    return render_template('hello.html', name = name)

if __name__ == '__main__':
    app.run()
```

What will be returned will the content of `hello.html`
Basic template rendering

The HTML file *hello.html*

```html
<!doctype html>
<html>
<head>
  <title>The website that says Hello to you</title>
</head>
<body>
  {% if name %}
    <h1>Hello, {{ name }} !</h1>
  {% else %}
    <h1>Hello, thing with no name !</h1>
  {% endif %}
</body>
</html>
```

It's no ordinary HTML ⇒ there are instruction mixed in !
Basic template rendering

The HTML file *hello.html*

```
<!doctype html>
<html>
<head>
  <title>The website that says Hello to you</title>
</head>
<body>
  {% if name %}
    <h1>Hello, {{ name }}!</h1>
  {% else %}
    <h1>Hello, thing with no name!</h1>
  {% endif %}
</body>
</html>
```

*hello.html* is processed to generate the HTML to send to a user. Here, we use the *name* variable, passed as a parameter of *render_template*
Basic template rendering

The HTML file *hello.html*

```html
<!doctype html>
<html>
<head>
  <title>The website that says Hello to you</title>
</head>
<body>
{% if name %}
  <h1>Hello, {{ name }} !</h1>
{% else %}
  <h1>Hello, thing with no name !</h1>
{% endif %}
</body>
</html>
```

Variables values can be rendered to text with `{{ }}`
Basic template rendering

The HTML file *hello.html*

```html
<!doctype html>
<html>
<head>
    <title>The website that says Hello to you</title>
</head>
<body>
    {% if name %}
        <h1>Hello, {{ name }} !</h1>
    {% else %}
        <h1>Hello, thing with no name !</h1>
    {% endif %}
</body>
</html>
```

Blocks of code are put between `{% %}`
Basic template rendering

Flask assumes that all your templates will be in a `template` directory, relative to your script

```
| templates
|   | hello.html
|   | test.py
```
Using resources

If you wish to use other file resources, like pictures or CSS files, you can put them in directory named static

```
- templates
  - hello.html
- static
  - style.css
- test.py
```

Those resource are not dynamic, not generated on the fly like the HTML code, hence the name "static"
Using ressources

Then, to use those ressources, you can again use `url_for`
Template inheritance

On a typical website, different views follow a similar design
Template inheritance

On a typical website, different views follow a similar design.
Template inheritance

On a typical website, different views follow a similar design
Template inheritance

On a typical website, different views follow a similar design.
Template inheritance

Jinja2 provides a simple way to share a common template and specialize it: *template inheritance*

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

{% block content %}
  {% if name %}
  <h2>Hello, {{ name }}!</h2>
  {% else %}
  <h2>Hello, thing with no name!</h2>
  {% endif %}
{% endblock %}

**hello.html** extends **base.html**
Template inheritance

Jinja2 provides a simple way to share a common template and specialize it: *template inheritance*

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

{% block content %}
    {% if name %}
        <h2>Goodbye, {{ name }}!</h2>
    {% else %}
        <h2>Goodbye, thing with no name!</h2>
    {% endif %}
{% endblock %}
```

goodbye.html extends base.html
Template inheritance

And *base.html* look like this

```html
<!DOCTYPE HTML PUBLIC "−//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
  <title>Salute.com, the website that salutes you</title>
  <link rel="stylesheet" type="text/css" href="{{url_for('static', filename='style.css')}}">
</head>
<body>
  <div id="container">
    <div id="header">
      <h1>Salute.com</h1>
      <p>The website that salutes you</p>
    </div>
    <div id="content">
      {% block content %}{% endblock %}
    </div>
    <div id="footer">
      <h2>Salute.com</h2>
      <p>Site design &amp; copyright © Alexandre Devert</p>
    </div>
  </div>
</body>
</html>
```
Template inheritance

On the Python side, `hello.html` and `goodbye.html` are just normal HTML pages, nothing special to do

```python
@app.route('/hello/')
@app.route('/hello/<name>/')
def hello(name = None):
    return render_template('hello.html', name = name)

@app.route('/goodbye/')
@app.route('/goodbye/<name>/')
def goodbye(name = None):
    return render_template('goodbye.html', name = name)
```
Template inheritance

In this example, extending *base.html* provides

- A common title
- Includes common resources (css, javascript, etc.)
- A common header
- A common footer
- The specialized part goes in the "content" block.

Coherent look, code reuse, and clean separation!
Template macros

On a website, the same user interface elements are often re-used.
Template macros

On a website, the same user interface elements are often re-used
Template macros

We can define reusable HTML bits of codes.

```html
{%- macro render_panel(title, style="left") %}
<div class="panel">
  <h1 class="{{{style}}}">{{ title }}</h1>
  <div class="panel-content">
    {{ caller() }}
  </div>
</div>
{%- endmacro %}
```

This define a box, containing whatever `caller()` will put in it, and with a title. We put this in `ui.html`
Template macros

Now, we can create lots of boxes.

{% extends "base.html" %}
{% import "ui.html" as ui %}

{% block content %}
<div class="three-columns-layout">
  <div class="left-column">
    {% call ui.render_panel("Lorem ipsum", "left") %}
    ... blabla ...
    {% endcall %}
  
    {% call ui.render_panel("Lorem ipsum", "left") %}
    ... blabla ...
    {% endcall %}
  </div>

  <div class="right-column">
    {% call ui.render_panel("History", "left") %}
    ... blabla ...
    {% endcall %}

    {% call ui.render_panel("Now is the time for all good men", "left") %}
    ... blabla ...
    {% endcall %}
  </div>
</div>

{% endblock %}

No need to copy paste the same HTML code around!
Template macros

To use a macro, first import the file that contains that macro

```
{% import "ui.html" as ui %}
```

Then you can call the macro

```
{% call ui.render_panel("My Title Here", "left") %}
... blabla ...
{% endcall %}
```

What is between `call` and `endcall` could be any valid HTML code. It will be placed in place of `caller` in the macro definition.
Template language

Jinja templates use their own language, more or less Python-like.

- It tries to imitate Python
- But it is not Python

Why not having full power of Python in a template?
Template language

Jinja provides a limited language because

- It’s a page that might be served for many different users. Should be fast.
Template language

The *if* block works like Python

```{%
if show_advertisement %}
<h1>Buy Drunk Panda, the best beer in Suzhou !</h1>
{%
endif %}
```
Template language

An optional `else` block works can be used

```%
% if show_advertisement %
<h1>Buy Drunk Panda, the best beer in Suzhou !</h1>
%
% else %
Do not buy anything
%
% endif %
```
Template language

An even `elif` blocks are available

{% if show_beer_advertisement %}
<h1>Buy Drunk Panda, the best beer in Suzhou !</h1>
{% elif show_pizza_advertisement %}
<h1>Buy Pizza Hut, the worst pizzas ever !</h1>
{% else %}
Do not buy anything
{% endif %}
Template language

The Jinja *for* loop works like the Python one

```jinja
{% for item in navigation %}
  <li>
    <a href="{{item.href}}">{{ item.caption }}</a>
  </li>
{% endfor %}
```

Note that

- *navigation* is a sequence, passed to the template
- *item* is one item of the sequence
- loop code is between `{% for %}` and `{% endfor %}`
Template language

Jinja provides a *loop* object that can be called inside a *for* loop

{% for item in navigation %}
 <li>
   <a href="{{ item.href }}">{{ loop.index }} {{ item.caption }}</a>
 </li>
{% endfor %}
Template language

This *loop* object provides some useful informations about the current item of the loop.

<table>
<thead>
<tr>
<th>loop variable</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop.index</td>
<td>Current index (1-indexed)</td>
</tr>
<tr>
<td>loop.index0</td>
<td>Current index (0-indexed)</td>
</tr>
<tr>
<td>loop.revindex</td>
<td>Current index, reversed order (1-indexed)</td>
</tr>
<tr>
<td>loop.revindex0</td>
<td>Current index, reversed order (0-indexed)</td>
</tr>
<tr>
<td>loop.last</td>
<td>True if last item</td>
</tr>
<tr>
<td>loop.first</td>
<td>True if first item</td>
</tr>
</tbody>
</table>
Template language

You can filter the `for` loop, as in Python

```%
% for user in user_list if not user.is_hidden %
<\li>
   {{ user.name }}
</\li>
%
% endfor %
%```
If the sequence you iterate turns out to be empty, you can catch this case with an else block

```plaintext
{% for user in user_list if not user.is_hidden %}
  <li>
    {{ user.name }}
  </li>
{% else %}
  No users found!
{% endfor %}
```
Table of Contents

1 Model-View-Controller
2 Flask
3 First steps
4 Routing
5 Templates
   - Basic template rendering
   - Using resources
   - Template inheritance
   - Template macros
   - Template language
6 Requests
Requests

We can send data (HTML, JSON, XML, any kind of text), but we also need to receive data

- passwords
- checkboxes
- values
- ...
Requests

The HTTP protocol defines different kind of requests

- $GET \Rightarrow$ request to send data
- $POST \Rightarrow$ request to accept data

So far, we only handled $GET$ requests: sending HTML data.
Requests

We can also handle POST requests, like this

```python
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    # GET request
    if request.method == 'GET':
        return render_template('login.html')
    # POST REQUEST
    else:
        email = request.form['email']
        password = request.form['password']

        # Check email & password
        # TODO

        return render_template('welcome.html')
```
Requests

The *request* object hold the information sent to the server

```html
<form name="login" method="post" action="{{url_for('login')}}">
  <label>Email</label>
  <input type="text" name="email" maxlength="254" />

  <label>Password</label>
  <input type="password" name="password" />

  <button type="submit">Enter</button>
</form>
```