python™ Seminar

Australian Synchrotron
Turning bright ideas into brilliant outcomes

Flask

web development,
one drop at a time

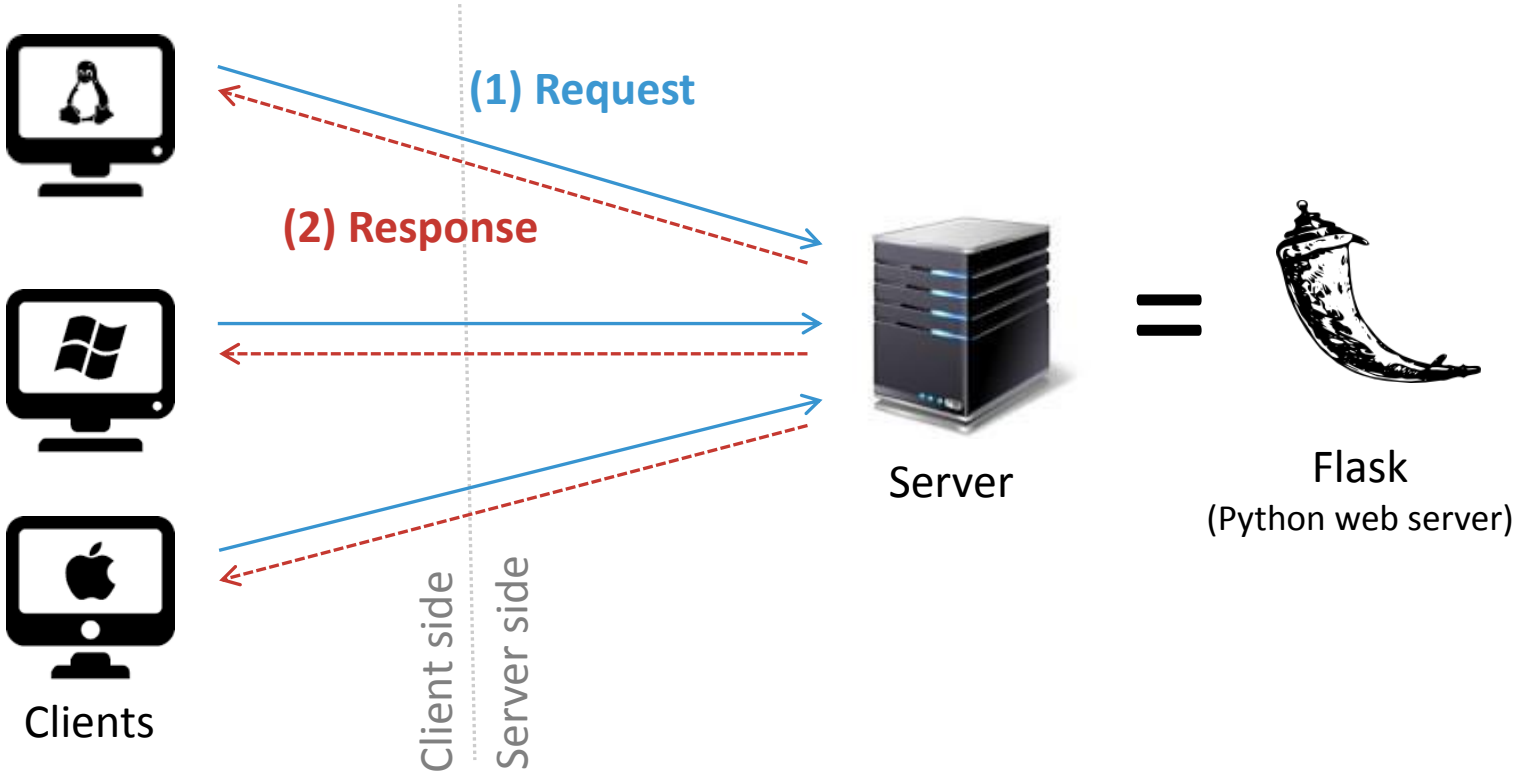
Supported by
Australian Government
State Government Victoria

What is Flask?

- Python library
- Server-side web micro-framework for Python
- Very small
- Easy to code and configure
- Doesn’t stand in your way
- Highly extensible (lots and lots of extensions available)
- Excellent documentation
- Open Source (http://flask.pocoo.org/)

Web services in a nutshell

The web follows a client server model with HTTP as its message protocol.



Types of requests and responses

HTTP defines different **request** methods. The most used request methods and their typical use are:

GET	*Request data from the server*
POST	*Update data that is already present on the server*
	Often also used for submitting data and starting "things"
PUT	*Tell the server to create data*
DELETE	*Delete data on the server*

A request can be made with command line tools (such as curl), a web browser, Python code, or any other HTTP client.

Types of responses

The **response** consists of meta data and the actual data. The most famous meta data is the response code (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes).

Remember the 404 website you saw recently?

The response data can be simple text, structured text (such as JSON, XML), **HTML**, …

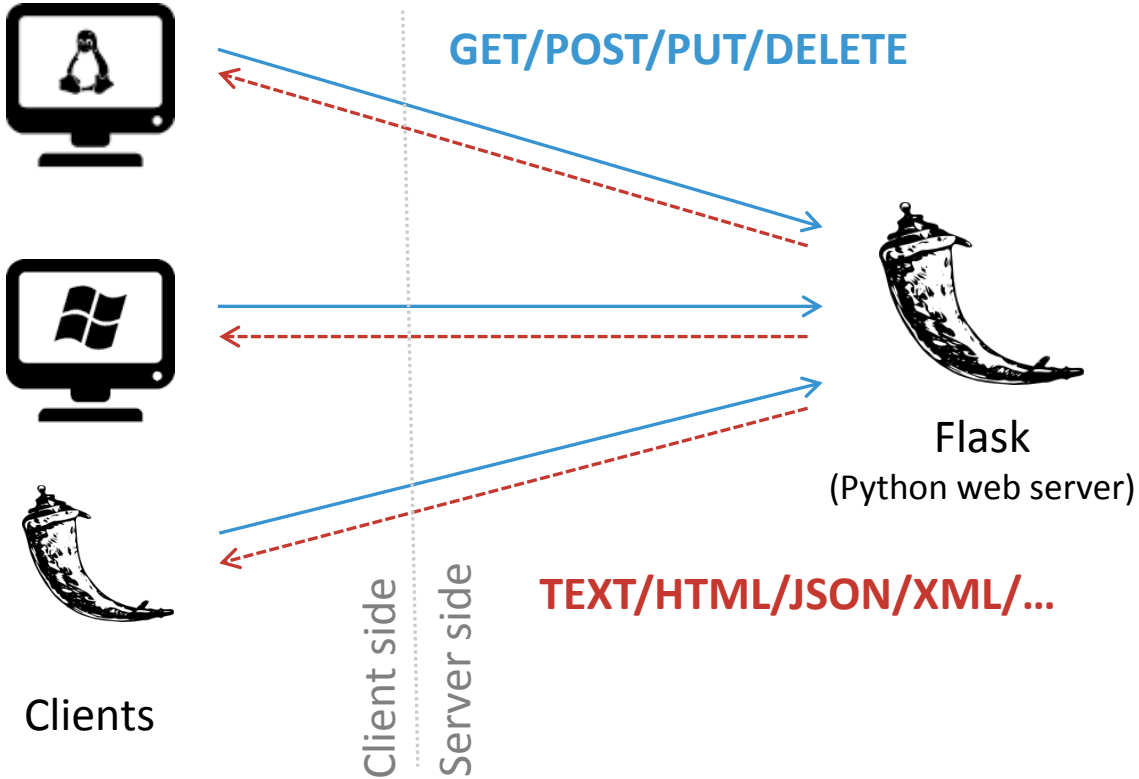
```
{"fruit": [
    {"name": "Apple"},
    {"name": "Pear"},
    {"name": "Mango"}
]}
```

JSON

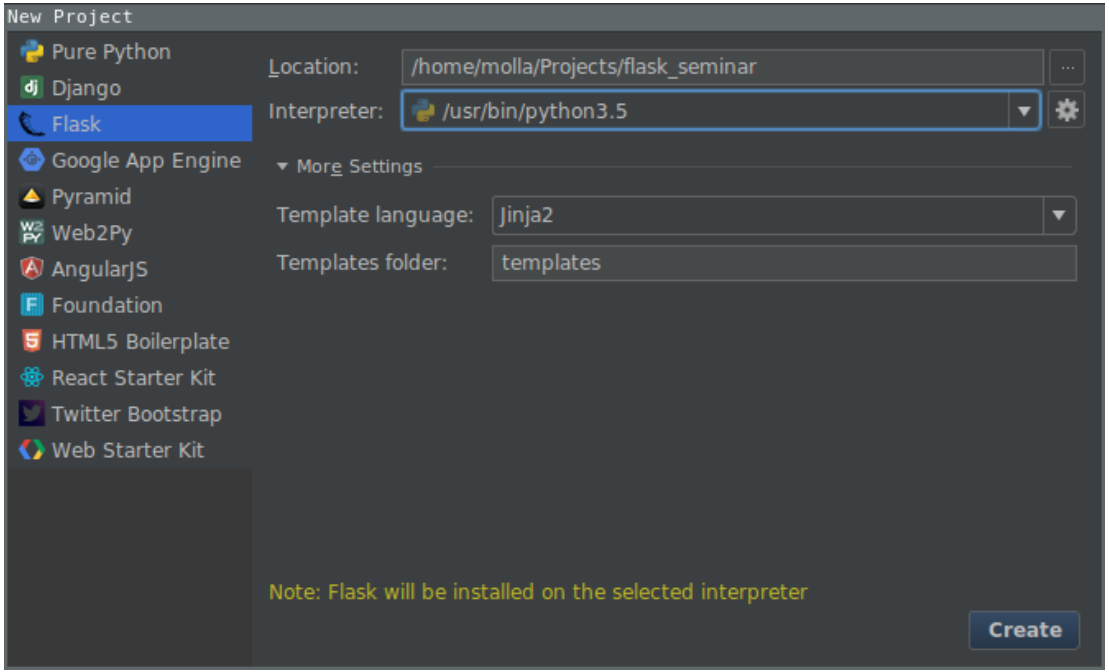
```
<fruit>
    <item>
        <name>Apple</name>
    </item>
    <item>
        <name>Pear</name>
    </item>
    <item>
        <name>Mango</name>
    </item>
</fruit>
```

The big picture

- curl/httpie
- web browser
- Python code
- another Flask server
- …



PyCharm – Create a Flask Project

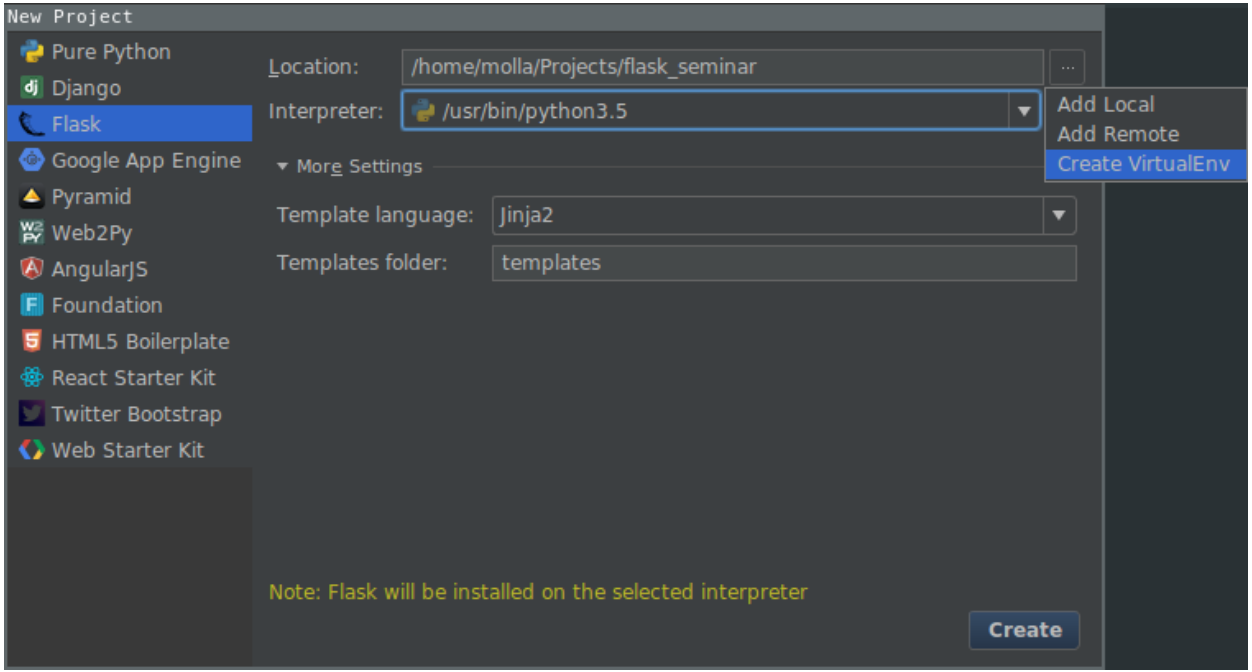


Top Menu: **File >> New Project…**

Template language: **Jinja2**
Template folder: **templates**

PyCharm will automatically install the Flask library for you

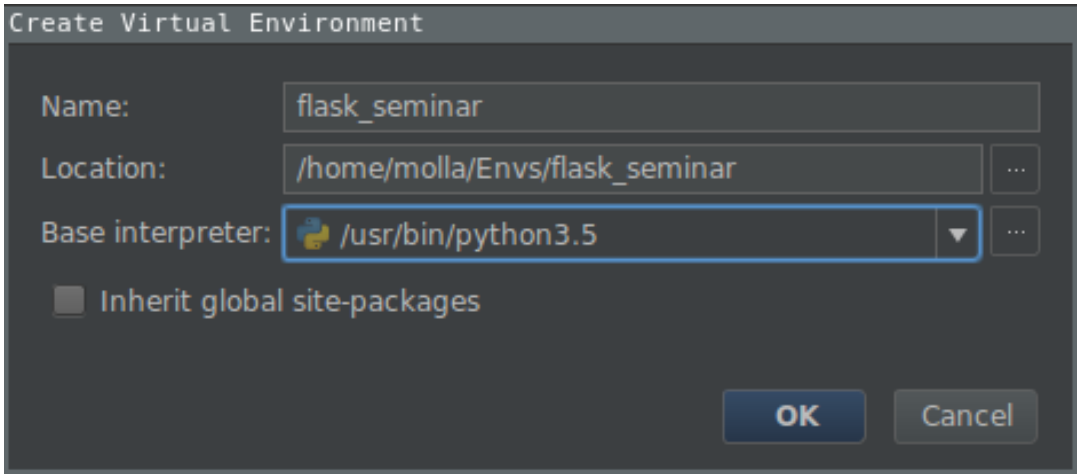
PyCharm – Create a Flask Project



Create a virtual env

This will keep our installed libraries clean and avoids clashes with libraries installed globally.

PyCharm – Create a Flask Project



Set the name and the location to your **preferred values**.

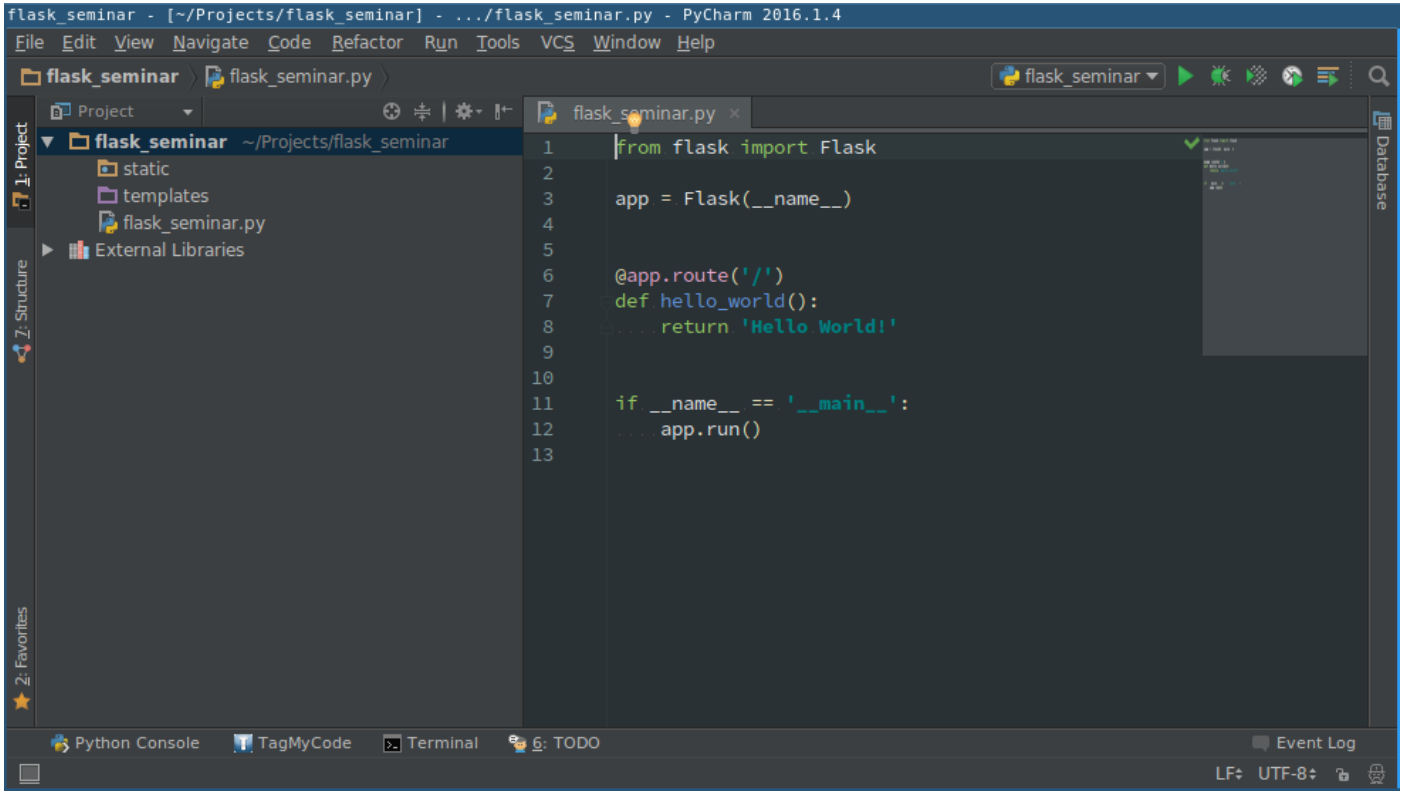
Set the base interpreter to: **Python 3**

Once you click OK, PyCharm will create the virtual env and set the interpreter to Python 3.

Click **Create** in the **New Project** dialog on order to create the Flask project

PyCharm – Create a Flask Project

Your Hello World Flask application is created and immediately ready to run



Start the Flask server

Run the Python code by clicking on the green triangle or using **Run >> Run ‘flask_seminar’**

The server will listen for requests on

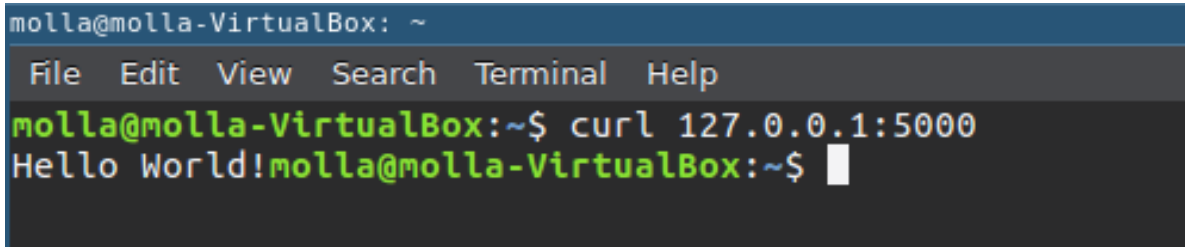
IP address: **127.0.0.1** (localhost)
port: **5000**

by default.

Send a request to the server

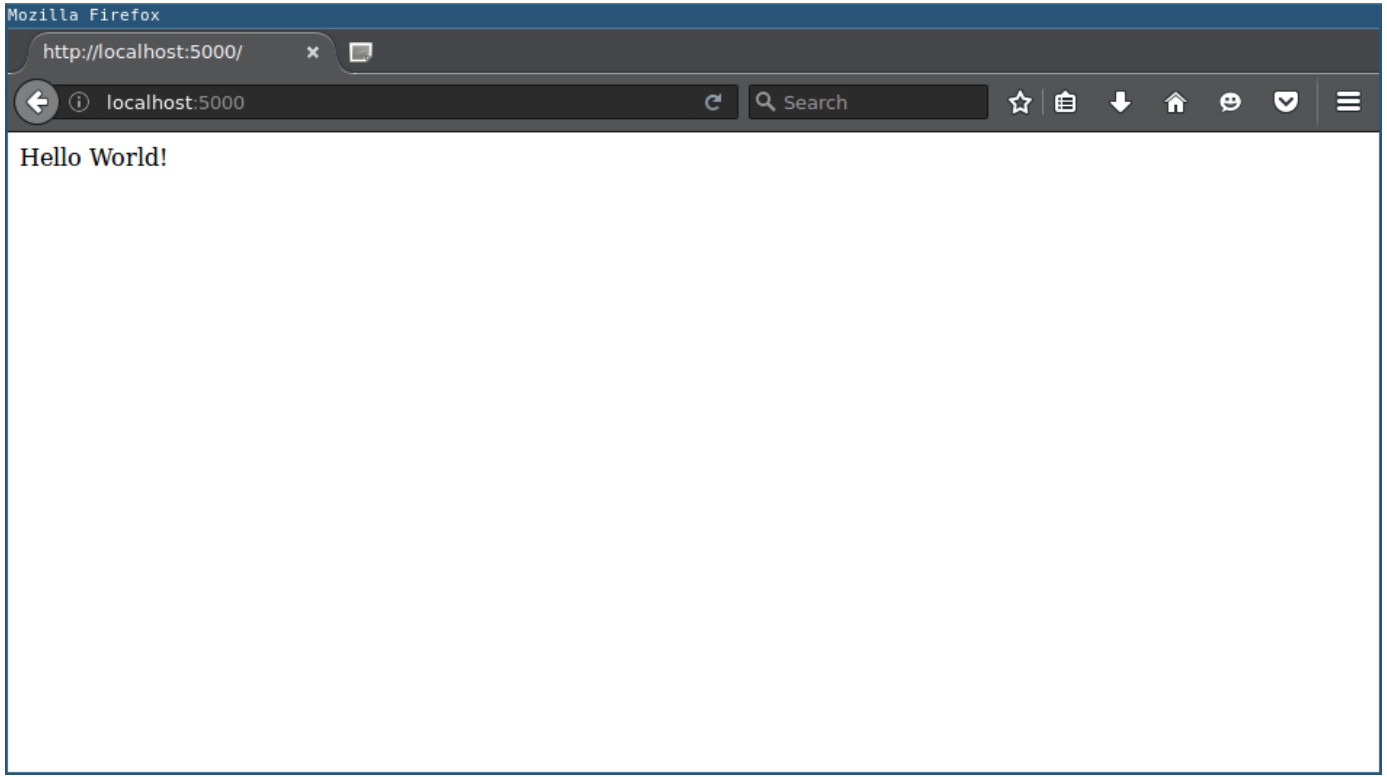
In a terminal send a **GET** request to the server using **curl:**

```
curl 127.0.0.1:5000
```



Pro Tip: Instead of curl you can use the beautiful httpie tool (https://github.com/jkbrzt/httpie)

Or just use a browser!



The code in more detail

```
from flask import Flask

app = Flask(__name__)


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


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

Import the Flask class from the flask library

Create a web application and give it a name (here we give it the name of the current file)

Define the URL this method is responsible for

Return a response

Run the web application if the python script is executed and not imported by another Python script.

Request methods in Flask

If you don’t specify a request method, Flask uses GET.

Let’s implement a route that you can send data to. We will use the POST request method. (https://en.wikipedia.org/wiki/POST_(HTTP))

```
@app.route('/store', methods=['POST'])
def store_data():
    return 'Received the data'
```

In order to access the data that has been sent with the request we import the Flask *request* object

```
from flask import Flask, request
```

Request methods in Flask

Let's send some data. For POST requests, the data is usually sent as *key=value* pairs (so called **form-encoded data**). We will use the httpie tool this time:

```
http --form POST 127.0.0.1:5000/store fruit=Apple
```

Result:

```
molla@molla-VirtualBox:~$ http --form POST 127.0.0.1:5000/store fruit=Apple
HTTP/1.0 200 OK
Content-Length: 17
Content-Type: text/html; charset=utf-8
Date: Thu, 23 Jun 2016 04:44:54 GMT
Server: Werkzeug/0.11.10 Python/3.5.1+

Received the data
```

Request methods in Flask

In order to access the form data that was sent with the request, the *request* object provides the data in a dictionary called *form*:

```
@app.route('/store', methods=['POST'])
def store_data():
    fruit = request.form['fruit']
    return 'Received the data: {}'.format(fruit)
```

```
molla@molla-VirtualBox:~$ http --form POST 127.0.0.1:5000/store fruit=Apple
HTTP/1.0 200 OK
Content-Length: 24
Content-Type: text/html; charset=utf-8
Date: Thu, 23 Jun 2016 04:51:27 GMT
Server: Werkzeug/0.11.10 Python/3.5.1+

Received the data: Apple
```

Request methods in Flask

Alternatively, return the full *form* dictionary as JSON structured text, using Flask's handy *jsonify* method:

```
from flask import Flask, request, jsonify

@app.route('/store', methods=['POST'])
def store_data():
    return jsonify(request.form)
```

```
molla@molla-VirtualBox:~$ http --form POST 127.0.0.1:5000/store fruit=Apple
HTTP/1.0 200 OK
Content-Length: 23
Content-Type: application/json
Date: Thu, 23 Jun 2016 04:56:26 GMT
Server: Werkzeug/0.11.10 Python/3.5.1+

{
    "fruit": "Apple"
}
```

Return some HTML

Let’s go back to our original route and return some HTML:

```
@app.route('/')
def hello_world():
    html_body = '<h1>Hello World!</h1>' + 'How is it going'
    return html_body
```

It works, but for a larger HTML page this approach gets ugly pretty quickly.

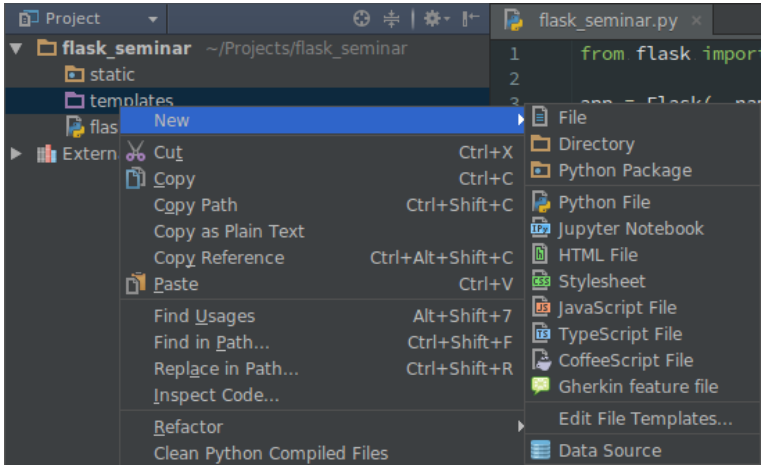
Solution: Use templates!

Jinja2 Templates

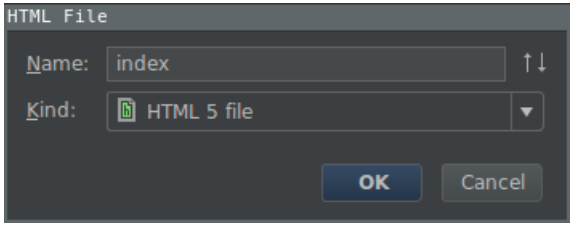
Flask uses the Jinja2 template engine.
Let's create a HTML page that we will turn into a dynamic page using templates.

Right click in the **templates** folder

New >> HTML File



Enter a name. We use *index* thus it will create a file named *index.html*



Jinja2 Templates

In order to return the newly created HTML page, import the `render_template` method

```
from flask import Flask, request, jsonify, render_template
```

and return the page in the `hello_world` method

```
@app.route('/')
def hello_world():
    return render_template('index.html')
```

Refresh the web browser and an empty page with the title “Title” is shown.

Let’s fill the page with dynamic content!

Dynamic pages

Extend the route with a template parameter (parameter name sandwiched by “<>”):

```
@app.route('/<name>')
```

This placeholder will allow for dynamic URLs such as

```
http://localhost:5000/Andreas
http://localhost:5000/Robbie
```

Add a parameter *name* to the method definition:

```
def hello_world(name):
```

This will make the value specified in an URL available in the method via the variable *name*.

Dynamic pages

The variable *name* is now available in the method body. But we want to print it on the webpage. Therefore pass the variable as an argument to the *render_template* method.

```
@app.route('/<name>')
def hello_world(name):
    return render_template('index.html', name=name)
```

Now when the page index.html is rendered a variable *name* containing the text specified after the “/” in the URL is available.

Dynamic pages

Let’s make use of our variable *name* and print it in the HTML file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
Hello {{ name }}, how is it going?
</body>
</html>
```

In Jinja2 a variable sandwiched by two curly braces is replaced with its value when the page is rendered.

Even more dynamic page content

Jinja2 allows any Python object to be passed as an argument to the `render_template` method.

Let's try a list:

```
@app.route('/<name>')
def hello_world(name):
    todo = ['cleaning windows', 'cooking dinner']
    return render_template('index.html',
                           name=name, jobs=todo)
```

Even more dynamic page content

In the HTML page we can loop over the list with the special Jinja2 statement

```
{% for %} {% endfor %}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
{% for job in jobs %}
{{ name }} is {{ job }} <br>
{% endfor %}
</body>
</html>
```

Loop over the jobs list
Print the name and the job
End of the body of the for loop

Even more dynamic page content

You can also use conditions to show certain parts of the HTML page:

```
@app.route('/<name>')
def hello_world(name):
    todo = ['cleaning windows', 'cooking dinner']
    return render_template('index.html',
                           name=name, jobs=todo, admin=True)
```

Use the `{% if %}` `{% endif %}` Jinja2 statement in your HTML code:

```
{% if admin %}
Only admins can see this
{% endif %}
```

Client-side dynamic page design

So far the page “dynamics” are created on the server-side and thus require a page refresh.

What if you like to create a dynamic page that magically refreshes itself and updates content in-place? Think of all the pages you visit regularly: *Facebook, Gmail, Twitter, …*

This is were we have to leave the beautiful Python-land and enter the dark corners of client-side scripting: **JavaScript**, **AJAX** calls, **websockets**, …

Unfortunately, this is beyond the seminar (it’s not really Python either).

The next slide lists our current recommendations for client-side scripting.

Client-side scripting recommendations

JavaScript libraries that we use at the Synchrotron and that make your life easier:

jQuery: https://jquery.com/
Knockout.js: http://knockoutjs.com/
Moment.js: http://momentjs.com/
Sammy.js: http://sammyjs.org/

New kid on the block:

Polymer: https://www.polymer-project.org

You hate writing JavaScript? Try CoffeeScript and the flask-assets extension:

CoffeeScript: http://coffeescript.org/
Flask-Assets: https://flask-assets.readthedocs.io/en/latest/

Styling your HTML page

If you use Polymer, you already have a decent looking page (thanks to Google)

Otherwise, check out the following CSS libraries:

Bootstrap: http://getbootstrap.com/
FontAwesome: http://fontawesome.io/

Further reading

Objectively the best Flask tutorial ever written:
http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

Miguel also wrote the excellent Flask Web Development book:
http://shop.oreilly.com/product/0636920031116.do

Want to connect services with each other? Try implementing a REST interface:
http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

All PyCon videos are highly recommended. For example:
https://www.youtube.com/watch?v=DIcpEg77gdE
https://www.youtube.com/watch?v=px_vg9Far1Y
https://www.youtube.com/watch?v=pZYRC8IbCwk