Python-Flask Micro framework for web development

Markus Veijola

June 2015

opiframe

Pre-requirements

* In this training our focus is to learn how to use Python-Flask micro framework to develop dynamic, rich web applications with use of databases and many helpful modules alongside.
* At this point you should have a Python-Flask environment installed, the basic flask project template created and tested. If not you can find step by step instructions from python_installation_in_windows.pdf material.
* You should also be somewhat comfortable writing Python code: know syntax, class definition, python script environment variables, python configuration etc.
* At this point if you are not familiar with those keywords you can visit the next web site: Python tutorial

The Application

- Application will include next features:
  - Flask configuration
  - Use of HTML templates
  - Web form support, including field validation.
  - Database management
  - User management with logins, session handling etc.
  - And few other handy things….

Introduction to Flask

- You might wonder what is ”micro framework”? In Python-Flask this does not mean that Flask is lacking some functionality needed to build up fully working web application. Instead it means that in basis Flask offers you a core functionality -> when you need more you take modules in use (like we did when installed Flask. Remember those commands pip install xxx?)
- Flask won’t make many decisions for you, such as what database to use. Those decisions that it does make, such as what templating engine to use, are easy to change. Everything else is up to you, so that Flask can be everything you need and nothing you don’t.
- By default Flask uses Jinja2 templating engine, which we use in this tutorial/training.

Introduction to Flask:Security

* Always keep security in mind when building web applications in ANY framework!!
* People using your application belives that data they store is safe and secured.
* Flask protects you against one of the most common security problems of modern web applications: cross-site scripting (XSS). Unless you deliberately mark insecure HTML as secure, Flask and the underlying Jinja2 template engine have you covered.
* There are many security aspects that can be configured in Flask modules. For example when one uses Flask-WTF extension you can configure it to protect you from CSRF (cross-site request forgery) attacks.
* Also SQLAlchemy database tool protects you against basic SQL injections.
* Best practise to get secure application is to see the module documentation before using it. Like you would do when using Flask-WTF module.

Introduction to Flask

* Flask application is easy to set up. You install the ”flask” module and create an object from Flask class (ofcourse you need to import it first). Here is all you need to get started with flask:

```
from flask import Flask

app = Flask(__name__)
```

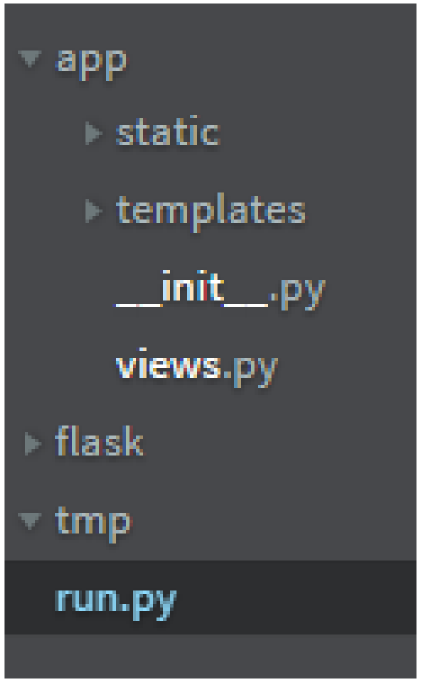
* In the code above the ”app” object is what you use to configure and set up your Flask web application. The ”__name__” module is what Python interpreter sets to be in value ”__main__” if the file contains the main program of your application. If the module is imported then the ”__name__” will be the same as module name.

Flask Configuration

* Normally you need to configure Flask module at least in bigger applications, or perhaps you want to use some other templating engine than Jinja2. The configuration can be done in several ways but the easiest and most common ways is to use own module for this.
* Before we do this go see the builtin configuration values of Flask in here: Flask Configuration
* Note that all these configuration variables have a default values that you can override in your own configuration as we will do in next sections.

Flask Configuration

- As you remeber we created next kind of folder hierarchy for our application in installation section:



```
▾ app
    ▸ static
    ▸ templates
      __init__.py
      views.py
▸ flask
▾ tmp
  run.py
```

Flask Configuration

* Now create a file named config.py in the root of our project. This file will include all configuration for our application.
* Let’s make a little modification and set the port where our application is running from 5000 to 3000.
* Append next line in our config.py file

```
SERVER_NAME=‘localhost:3000‘
```

* Then the line with red color below in our __init__.py file. This line will tell to Flask where to read the configurations.

```
from flask import Flask
app = Flask(__name__)
app.config.from_object('config')
from app import views
```

opiframe

Flask Configuration

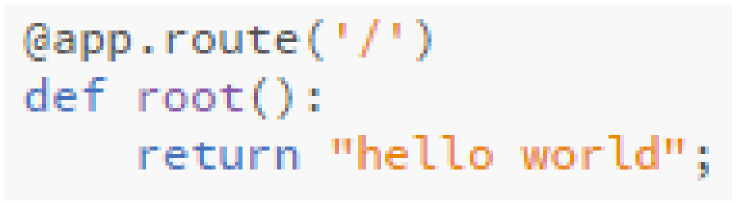
- Restart the application and you should see from command prompt that our application is now running on port 3000!
- In future we will set more configuration flags in this file.

opiframe

Routers

* One very important concept in web programming are routers.
* Routers in Python-Flask are defined using decorators. You have already done this in installation phase in file called views.py by using @app.route('/') decorator

Decorator →



```
@app.route('/')
def root():
    return "hello world";
```

opiframe

Routers

* How it works? Well consider the URL address. It consits of several parts:

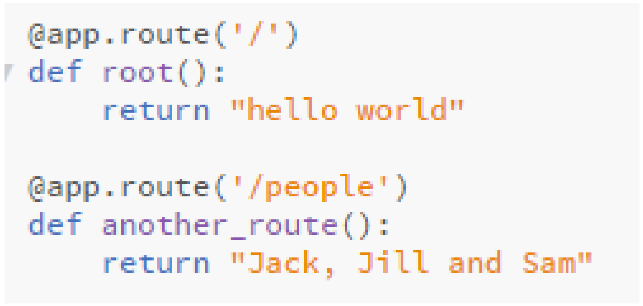
  protocol://domain:port/path

  i.e.

  http:localhost:3000/
* When you use decorator like in our example it works as follow:
  * When ever our application gets a request where the path part of URL is ”/” call the function root.
* This operation is called routing.

Routers

- Here is another enlightning example:



```
@app.route('/')
def root():
    return "hello world"

@app.route('/people')
def another_route():
    return "Jack, Jill and Sam"
```

- Now we have two routers in our application. If one now enters the url http://localhost:3000/people in browser it will invoke the function another_route() in our application. This will retrun a sting containing person names in browser.
- Defining routes in your application, you actually define the resources what your application offers.

HTML Templates

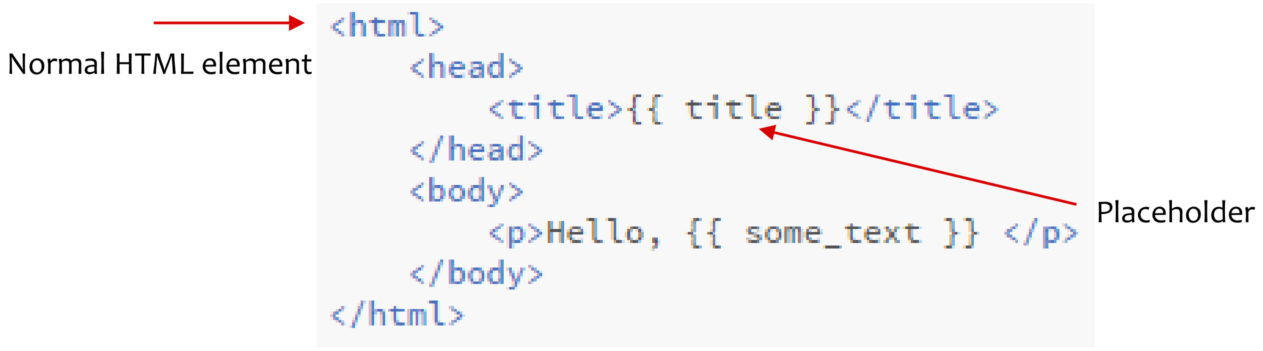
* Templates are here to save us. HTML templates are used to build up html pages from dynamic data, like data from database.
* Template engines are ”interpreters” that generates html from the dynamic data.
* Jinja2 is the default template engine used by Flask, but there are many other available also:
  * AngularJs
  * Jade
  * Django
  * JSP
  * Mustache
  * And many many more…

HTML Templates

* In Flask project templates and static files are stored in subdirectories within the application’s Python source tree, with the names *templates* and *static* respectively.
* These are the directories where Flask searches for html tempaltes and static files, like your JavaScript and CSS files.
* This is why our projects ”app” folder contains also folders ”static” and ”templates”.
* We will store our html template files in our template folder as you propably already expected.

HTML Templates

- When we create templates we use so called ”placeholders” along with HTML elements. These placheolders are then replaced with the backend data when template engine interprets these files.
- Create a file ”index.html” under ”templates folder and fill it with next simple code:



```
<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
        <p>Hello, {{ some_text }} </p>
    </body>
</html>
```

HTML Templates

* Then change the code in views.py as follow:

```
from flask import render_template
from app import app

@app.route('/')
def root():
    return render_template('index.html',title='MyBlog',some_text='Markus')
```

HTML Templates

* In previous example we use render_template function to wake up Jinja2 template engine. The first argument tells Jinja what file should be rendered. The following arguments contains the data for placeholders in .html file. Jinja uses this data to fill up the placholders ”title” and ”some_text” in html template.

Dynamic Routes

* To make application more exciting, lets create a dynamic route for our application (meaning route containing URL attribute).
* To do this make next kind of router that takes one url attribute user and passes that value to template_renderer function

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

Dynamic Routes

- Test the route with next url from browser:

http://localhost:3000/user/markus

Application Request Context

- When application receives a request from client, sometimes the application wants to know more information about request. One can use *request object which encapsulates the HTTP requests sent by the client.*
- Next example shows you how to use request object to read the user agent info from request….(NOTE request object is destroyed after we send the response!!!!!)

Application Request Context

```
from app import app
from flask import render_template
from flask import request
@app.route('/')
def index():
        return render_template('index.html',name='Markus Veijola')

@app.route('/user/<user>')
def user(user):
        agent = request.headers.get('User-Agent')
        return render_template('index.html',name=user,user_agent=agent)
```

HTTP Status Codes

* By default flask sets staus code 200 for each response. There are cases where you want to set the status code explicitly to response. This is done by setting the status code as second argument in response, as show in code snippet below…

```
@app.route('/bad')
def bad():
        return render_template('error.html'),400
```

Creating response object

- Sometimes you may have a need to add some custom headers in you HTTP response, or some set cookie in the response . In this case you need flask make_response method. Using this method one is able to set headers, cookies or some other data to response before it is sent. Next code snippet shows you how to do this….

Creating response object

```
from app import app
from flask import render_template
from flask import request
from flask import make_response
@app.route('/')
def index():
        return render_template('index.html',name='Markus Veijola')

@app.route('/custom')
def custom():
        response = make_response(render_template('index.html'))
        response.headers.add('Cache-Control','no-cache')
        return response
```

Redirecting

* There is also a special type of response called redirect. The response DOES NOT include page document, it just gives a browser a new URL from which to load new page. Redirects are commonly used with web forms….

```
from app import app
from flask import render_template
from flask import request
from flask import make_response
from flask import redirect

@app.route('/goto')
def someroute():
        return redirect('http://www.kaleva.fi')
```

HTTP method in router

* To define what HTTP method router should accept (GET,POST,PUT,DELETE,HEADER, OPTIONS), you can define the router as follow…

```
@app.route('/entries/<int:id>', methods=['GET'])
def get_entry(id):
   ...

@app.route('/entries/<int:id>', methods=['POST'])
def update_entry(id):
   ...

@app.route('/entries/<int:id>', methods=['DELETE'])
def delete_entry(id):
```

Handling POST request

* Post request usually contains a Form data. Flask framework offers few ways to handle from data in server side. You can directly use request object to read the data, or you can use Flask From for this.
* Next example shows you how to use request object for reading POST values, later in this material you will see how to handle the form data using WTF –form components (…and yes it is WTF and it does not mean that bad thing even if you feel like that when using this sometimes. I 'don't know where the abbreviation comes form….)

opiframe

Handling POST request

* Consider you have next kind of web form in your application (NOTE! Both id and name attributes MUST be present in form elements!!!)

```
<form action="/login" method="POST">
		<input type="text" id="username" 
name="username"/><br/>
		<input type="password" id="password" 
name="password"/><br/>
		<input type="submit" value="Login"/>
</form>
```

Handling POST request

- Then you handle the POST request in server router like this…

```
@app.route('/login',methods=['POST'])
def login():
        username = request.form['username']
        password = request.form['password']
        print(username)
        print(password)
        return redirect('/users')
```

Jinja2 Template Engine

- You have already used the render_template to generate HTML from given data.
- Next sections of this material shows you the most used Jinja control structures like if, for, inheritance etc.
- To test these create user.html file in your projects templates folder.
- Make next kind of template for it….

user.html (if)

```
<h1>Welcome</h1>
{% if user %}
        <h2>Hello, {{user}}</h2>
{% else %}
        <h2>We dont know you</h2>
{% endif %}
```

opiframe

Router for it…

* Make next router in your server

```
@app.route('/users')
def testJinja():
        return render_template('user.html',user="Heikki")
```

* Test the application also so, that you put empty string in user variable user=“”

user.html (for)

* Append next code in end of your user.html file

```
<ul>
{% for name in names %}
		<li>{{name}}</li>
{% endfor %}
</ul>
```

opiframe

Router

* Modify the router like below…

```
@app.route('/users')
def testJinja():
        name_array=['Dave','James','Tim','Jane','Stine','Amber']
        return render_template('user.html',user="",names=name_array)
```

* And test the application again

Template Inheritance

- A very powerful feature in Jinja is template inheritance.
- Portions of template code that need to be repeated in several places can be stored in a separate file and included from all templates to avoid repetition.
- First you need to create a base.html file containing the basic template code for your pages…

base.html

```
<!doctype html>
<html>
	<head>
		{% block head %}
		<link href="/static/css/style.css" type="text/css" 
rel="stylesheet"/>
		<title>{% block title %}{% endblock %} Title</title>
		{% endblock %}
	</head>
	<body>
		{% block body %}
		{% endblock %}
	</body>
</html>
```

Extend some other page (i.e. user.html)

```
{% extends "base.html" %}
{% block title %}User page{% endblock %}
{% block head %}
    {{super()}}
{% endblock %}
{% block body %}
{% if user %}
            <h2>Hello, {{user}}</h2>
{% else %}
            <h2>We dont know you</h2>
{% endif %}

<ul>
{% for name in names %}
            <li>{{name}}</li>
{% endfor %}
</ul>
{% endblock %}
```

Using Jinja modifiers

* Variables in html page can be modified with filters, whch are added after variable name with a pipe character as separator. For example next example will capitalize the first letter of string inside the name variable:

  ```
  <h1>Hello {{name|capitalize}}</h1>
  ```

Jinja modifiers

Filter name	Description
safe	Renders the value without applying escaping
capitalize	Converts first character to uppercase
lower	Converts characters to lowercase
upper	Converts characters to uppercase
title	Capitalize each word in the value
trim	Removes leading/trailing whitespace
striptags	Removes any HTML tags from value before rendering the value

Twitter Bootstrap integration with Flask-Bootstrap

* Twitter bootstrap is one of the common components used in modern web applications.
* Flask has it own extension for it called flask-bootstrap.
* You can install this extension with command:

  ```
  pip install flask-bootstrap
  ```
* The previous command installs all the needed dependencies (bootstrap.css and bootstrap.js files etc).
* The next example shows how you create the responsive navbar in our application

Twitter Bootstrap integration with Flask-Bootstrap

* Then you need to initialize the flask bootstrap in your application. Append next lines in __init__.py file.

```
from flask import Flask
from flask.ext.bootstrap import Bootstrap
app = Flask(__name__)
app.config.from_object('config')
bootstrap = Bootstrap(app)
from app import views
```

Creating twitter responsive navbar

- Next create a file named navbar.htmlCopy the content of next slide in that file…(Please after copy paste fix the intendation)

Creating twitter responsive navbar

```
{% extends "bootstrap/base.html" %}
{% block title %}Hello{% endblock %}
{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
       <button type="button" class="navbar-toggle"
                 data-toggle="collapse" data-target=".navbar-collapse">
                 <span class="sr-only">Toggle navigation</span>
                 <span class="icon-bar"></span>
                 <span class="icon-bar"></span>
                 <span class="icon-bar"></span>
         </button>
                 <a class="navbar-brand" href="/">Flasky</a>
                 </div>
                 <div class="navbar-collapse collapse">
                        <ul class="nav navbar-nav">
                          <li><a href="/">Home</a></li>
                         </ul>
                   </div>
                 </div>
</div>
{% endblock %}
```

opiframe

Use the navbar in your pages…

```
<!doctype html>
<html>
        <head>
                <meta charset="utf-8">
                <title>Jinja2 Example</title>
        </head>
        <body>
                {% include "navbar.html" %}
                <h1>Hello {{name}}</h1>
                <h2>You are using browser:{{user_agent}}</h2>
        </body>
</html>
```

Jinja2 support next block elements

* doc
* html_attributes
* html
* head
* title
* metas
* styles
* body_attributes
* body
* navbar
* content
* scripts

Web Forms

* Flask-WTF is an extension that makes working with web forms little easier.
* Flask-WTF and its dependencies can be installed with pip:

  pip install flask-wtf
* By default Flask-WTF protects all form against Cross-Site Request Forgery (CSRF) attacks.
* To implement CSFR protections you need to configure an encryption key to application configuration file.
* Just add next line in your config.py file

  SECRET_KEY = “some hard string to guess”

Web forms

- When using Flask-WTF, EACH web form is represented by defining a class that inherits from class Form.
- Next example shows you how to implement and use a simple form containing a text field and a submit button.
- Under the app folder create a filed called i.e. my_froms.py
- Inside that file write next sample code

Web forms

```
from flask.ext.wtf import Form
from wtforms import StringField, SubmitField
from wtforms.validators import Required

class FormName(Form):
	name = StringField('Your name',validators=[Required()])
	submit = SubmitField('Submit');
```

Web forms

* Make next modifications in your “/” router (remember also include next line in that file: from app.my_forms import FormName

```
@app.route('/‘, methods=[‘GET’,'POST'])
def index():
        name = None
        form = FormName()
        if form.validate_on_submit():
                name=form.name.data
                form.name.data = ''
        return render_template('index.html',name=name,form=form)
```

Web forms

* Make next changes in your index.html file

```
<form method="POST">
        {{form.csrf_token}}
        {{form.name.label}} {{form.name(id="style")}}
        {{form.submit}}
</form>
```

Flask-WTF supported HTML fields

- Flask-WTF supports next fields:
  - StringField
  - TextAreaField
  - PasswordField
  - HiddenField
  - DateField
  - DateTimeField
  - IntegerField
  - DecimalField
  - FloatField
  - BooleanField
  - RadioField
  - SelectField
  - SelectMultipleField
  - FileField
  - SubmitField
  - FormField
  - FieldList

Flask-WTF supported validators

- Flask-WTF supports next validators
  - Email
  - EqualTo
  - IPAddress
  - Length
  - NumberRange
  - Optional
  - Required
  - Regexp
  - URL
  - AnyOf
  - NoneOf

Flash Messages

* Flash messages are neat way to inform user about their action (like entering a wrong username, or password) or invalid user input.
* First of all you need to import that module. On your routers file add next line in the top of the file…

  from flask import flash

* Then define the flash message in some of your routers. In my example (in next slide), the message is defined in "/" router, after checking if form is valid

Flash Messages

```
@app.route('/',methods=['GET','POST'])
def index():
        name = None
        form = FormName()
        if form.validate_on_submit():
                name=form.name.data
                form.name.data = ''
        else:
                flash('Hey, why not giving your name?')

        return render_template('index.html',name=name,form=form)
```

opiframe

Flash Messages

- Calling flash() is not enough still: The templates used needs to render these messages. Best place to put the rendering code would be the base.html, because then any page inheriting from it cloud display flash messages. The next code snippet shows you the code what template needs to render the message

Flash Messages

```
{% for message in get_flashed_messages() %}
        <div class="alert alert-warning">
          <button type="button" class="close" data-dismiss="alert">&times;</button>
          {{message}}
        </div>
{% endfor %}
```

SqLite Integration

* Most web applications uses some database implementation to store and retrieve information, and then render a view from that data.
* With flask you can use basically any database implementation available: MySQL, SQLite, MongoDB…etc.
* In this material we integrate SqLite database to our application, since configuring MySql in windows can be a nightmare.
* You need to install needed tools with the following command:

```
pip install flask-sqlalchemy sqlalchemy-migrate
```

SqLite Integration

* To configure access to your SqLite database server by using these setting (in your config.py):

SQLALCHEMY_DATABASE_URI = ‘sqlite:///absolute/path/to/database‘

SQLALCHEMY_MIGRATE_REPO = “path to migrate repo”

* For example it can be as follow in windows:

SQLALCHEMY_DATABASE_URI =‘sqlite:///’ + os.path.join(basedir,’data.db’)

SQLALCHEMY_MIGRATE_REPO = os.path.join(basedir, 'db_repository')

* More config flags can be found from here: http://flask-sqlalchemy.pocoo.org/2.1/config/

SqLite Integration

* Instantiating SqlAlchemy can be done in __init__.py file like following:

```
from flask import Flask
from flask.ext.bootstrap import Bootstrap
from flask.ext.sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config.from_object('config')
bootstrap = Bootstrap(app)
db = SQLAlchemy(app)
from app import views
from app import my_forms
```

Defining Tables

* You define your database tables as classes. Next example will create a table named user with columns id (primary key) username,password
* Create a file named models in app folder.
* Fill it with next code….

Defining Tables

```
from app import db

class User(db.Model):
        id = db.Column(db.Integer, primary_key=True)
        username = db.Column(db.String(128))
        password = db.Column(db.String(128))
        """Define the class constructor"""
        def __init__(self, username, password):
            self.username = username
            self.password = password
```

Creating Database

- With the configuration and model in place we are now ready to create our database file. The SQLAlchemy-migrate package comes with command line tools and APIs to create databases in a way that allows easy updates in the future, so that is what we will use.
- The this script is completely generic. All the application specific pathnames are imported from the config file.
- Create a file with name db_create.py (in root folder of your project) and copy the code from next slide.

Creating Database

```
from migrate.versioning import api
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
from app import db
import os.path
db.create_all()
if not os.path.exists(SQLALCHEMY_MIGRATE_REPO):
   api.create(SQLALCHEMY_MIGRATE_REPO, 'database repository')
   api.version_control(SQLALCHEMY_DATABASE_URI,
SQLALCHEMY_MIGRATE_REPO)
else:
   api.version_control(SQLALCHEMY_DATABASE_URI,
SQLALCHEMY_MIGRATE_REPO, api.version(SQLALCHEMY_MIGRATE_REPO))
```

Creating Database

* Now execute the script from CMD:

  python db_create.py

* This will create you an database: a file called data.db as you defined in your config.py file.

Migration Files

- We have defined our model (User), we can store it into our database.
- We will consider any changes to the structure of the application database a an migration, so this is our first, which will take us from an empty database to a database that can store users.

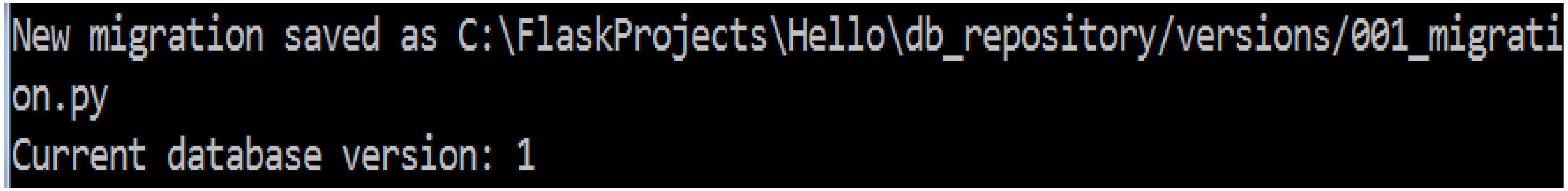
- To generate a migration we use another little Python helper script (create file db_migrate.py in root and copy the next code in it):

Migration

```
import imp
from migrate.versioning import api
from app import db
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
migration = SQLALCHEMY_MIGRATE_REPO + ('/versions/%03d_migration.py' % (v+1))
tmp_module = imp.new_module('old_model')
old_model = api.create_model(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
exec(old_model, tmp_module.__dict__)
script = api.make_update_script_for_model(SQLALCHEMY_DATABASE_URI,
SQLALCHEMY_MIGRATE_REPO, tmp_module.meta, db.metadata)
open(migration, "wt").write(script)
api.upgrade(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
print('New migration saved as ' + migration)
print('Current database version: ' + str(v))
```

Migration

- Now you can run the script : **python db_migrate.py**



```
New migration saved as C:\FlaskProjects\Hello\db_repository/versions/001_migrati
on.py
Current database version: 1
```

Database upgrades and downgrades

* Let's say that for the next release of your application you have to introduce a change to your models, for example a new table needs to be added. Without migrations you would need to figure out how to change the format of your database, both in your development machine and then again in your server, and this could be a lot of work.
* For this we create a few new scripts to help us update and downgrade the database… .

upgrade

- Create db_upgrade.py file in the root of your project and add next content to it:

```
from migrate.versioning import api
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
api.upgrade(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
print('Current database version: ' + str(v))
```

downgrade

* Create db_downgrade.py file in the root of your project and add next content to it:

```
from migrate.versioning import api
from config import SQLALCHEMY_DATABASE_URI
from config import SQLALCHEMY_MIGRATE_REPO
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
api.downgrade(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO, v - 1)
v = api.db_version(SQLALCHEMY_DATABASE_URI, SQLALCHEMY_MIGRATE_REPO)
print('Current database version: ' + str(v))
```

Database relations

- Relational databases are good at storing relations between data items. This can be achieved also in Python Flask. We already have seen how to create a model to database, the User. Next we see how we can create a relation between two tables User and Friends

Database relations

```
class User(db.Model):
		id = db.Column(db.Integer, primary_key=True)
		username = db.Column(db.String(128),unique=True)
		password = db.Column(db.String(128))
		friends = db.relationship('Friends',backref="user",lazy='dynamic')
		"""Define the class constructor"""
		def __init__(self, username, password):
				self.username = username
				self.password = password


class Friends(db.Model):
		id = db.Column(db.Integer, primary_key = True)
		name = db.Column(db.String)
		address = db.Column(db.String)
		age = db.Column(db.Integer)
		user_id= db.Column(db.Integer,db.ForeignKey('user.id'))
```

Database relations

- After you have made the changes run the: python db_migrate.py

Database operations

* Read operation from database is done with using the models we have defined (like User and Friends).
* Here is an example to get list of all Users (note call for all() function returns ALWAYS an array)

```
@app.route('/users',methods=['GET'])
def listUsers():
        user = User.query.all();
        print(user[0].username);
```

Database operations (Read)

- Quering with username, and get only first record:
  `User.query.filter_by(username='kalle').first()`
- Selecting a bunch of users by a more complex expression:
  `User.query.filter(User.username.endswith(‘lle')).all()`
- Ordering users by something:
  `User.query.order_by(User.username)`

Database operations (Create)

* Creating is easy, just create a instance of model you want to add and use add() and commit() as in following example

```
me = User('test', 'test')
db.session.add(me)
db.session.commit()
```

Database operations (Delete)

* Deleting is also easy. If we have the same instance of user as in previous example (me) deletion would be done as follow:

```
db.session.delete(me)
db.session.commit()
```

opiframe

Database operations (Update)

* There are several ways to update, but these ways are the most used ones:

```
User.query.filter_by(username='admin').update(dict(password='ad
min'))

//or

user = User.query.get(id)
user.name = 'New Name'
db.session.commit()
```