Django
starting guide

(and much more...)

Alessandro Bucciarelli

Outline

- Lesson 1
    - Intro to versioning systems (Git)
    - Intro to Python and basic data structures
    - Django
- Lesson 2
    - Interaction between Django and REST API
    - Q&A's time

Intro to versioning systems (Git)

- What is versioning?
- Who cares about versioning!!
- Scenarios where Git is useful

What is versioning?

- A versioning system keeps track of your code (also every blank line you add/remove)

- Every time you modify something on files under revision it will compare current revision with former revision

Who cares about versioning!!

- It often happens that versioning is not used that much or not used at all because it is mainly believed to be difficult or useless
- A lot of projects are tracked down thanks to versioning, and probably they would not work without it (e.g Linux Kernel running on your PCs or Vs)

Scenarios where Git is useful

(and helps you to survive)

- You need to write some sort of code :
- Probably without Git you would start to code and :
- a certain point you realise that something is going wrong due to errors but :

Scenarios where Git is useful

(and helps you to survive)

- You need to write some sort of code :
- Probably without Git you would start to code and :
- a certain point you realise that something is going wrong due to errors but :

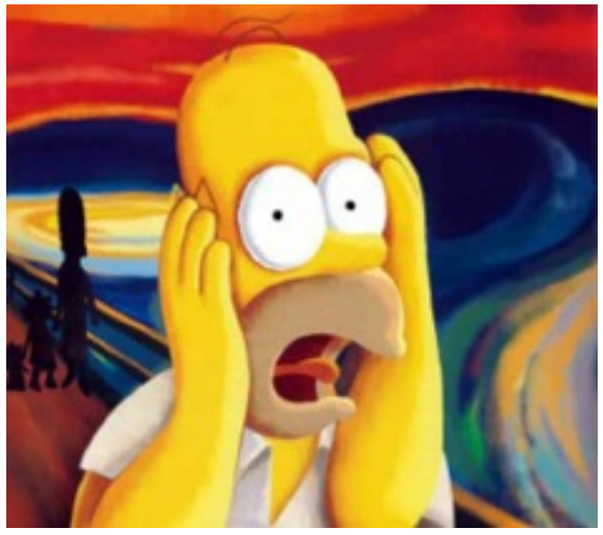
YOU CAN'T FIGURE OUT WHERE THE ERROR IS and..

Scenarios where Git is useful

(and helps you to survive)

- You need to write some sort of code :
- Probably without Git you would start to code and :
- a certain point you realise that something is going wrong due to errors but :

YOU CAN'T FIGURE OUT WHERE THE ERROR IS and..



Scenarios where Git is useful

(and helps you to survive)

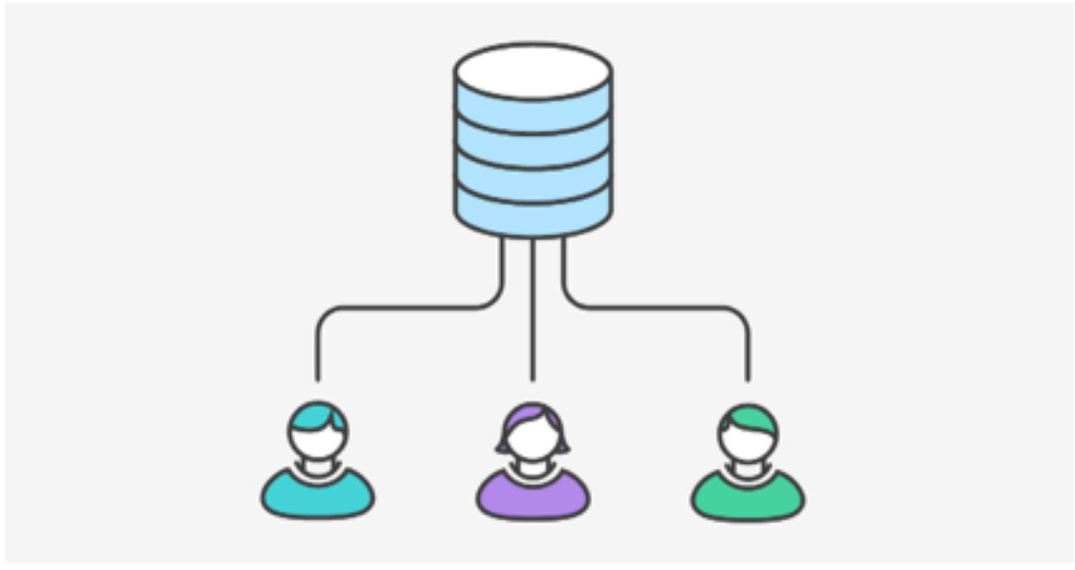
- Let's say you are assigned to a very big project with other people

- How do you ensure every one's work does not conflict with other's ?


git

Basic git commands

- git init .
- git add <filename>
- git commit -m "commit message"
- git log
- git reset <commit's unique hash>
- git checkout <file name>
- git checkout -b <new brach name>
- git stash / pop



Let's give it a (fast) try :)

1. Create an empty directory

Let's give it a (fast) try :)

1. Create an empty directory

2. Initialise a git repo

Let's give it a (fast) try :)

1. Create an empty directory
2. Initialise a git repo
3. Add a file and write something in it

Let's give it a (fast) try :)

1. Create an empty directory
2. Initialise a git repo
3. Add a file and write something in it
4. Commit your changes

Let's give it a (fast) try :)

1. Create an empty directory

2. Initialise a git repo

3. Add a file and write something in it

4. Commit your changes

5. Modify it and commit again

Let's give it a (fast) try :)

1. Create an empty directory

2. Initialise a git repo

3. Add a file and write something in it

4. Commit your changes

5. Modify it and commit again

6. Turn back to the previous commit

Let's give it a (fast) try :)

1. Create an empty directory
2. Initialise a git repo
3. Add a file and write something in it
4. Commit your changes
5. Modify it and commit again
6. Turn back to the previous commit
7. You are on the right way to become a git master!

Intro to Python

Python is:

- the fourth most used programming language

http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

- considered a scripting language but is more powerful than that, hence many great firms use it for a full stack development (e.g Dropbox)

http://stackshare.io/

- easy to learn/use

If you use Java or C++ forget it!

- In python, variables do not have type :)

Python :)

```
myCounter = 0
myString = str(myCounter)
if myString == "0": ...
```

Java :(

```
int    myCounter = 0;
String myString = String.valueOf(myCounter);
if (myString.equals("0")) ...
```

- It has its own syntax

Python :)

```
 print the integers from 1 to 9
for i in range(1,10):
    print i
```

Java :(

```
// print the integers from 1 to 9
for (int i = 1; i < 10; i++)
{
   System.out.println(i);
}
```

Don't panic :
it is object oriented

- you can define your classes like this:

```
class Student:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
```

- and methods for the class

```
    def add_age(self, new_age):
        self.age += new_age
```

Basic Python's data structures

Lists, Tuples and Dictionaries

- Lists are an heterogeneous set of objects, they are defined like this:

```
my_list_name = ["Bob", "Alice", 3, 4]
```

- every element in the list can be accessed with its index, for example:

```
bob = my_list_name[0]   # bob will contain the string "Bob"
alice = my_list_name[1] # alice will contain the string "Alice"
```

- negative indexes are supported as well:

```
variable = my_list_name[-1]   # what will variable contain?
```

Basic Python's data structures

Lists, Tuples and Dictionaries

- Tuples are mostly the same as lists :

```
my_tuple_name = (“Bob", "Alice", 3, 4)
```

- every element in the tuple can be accessed with its index, for example:

```
bob = my_list_name[0]   # bob will contain the string "Bob"
alice = my_list_name[1] # alice will contain the string "Alice"
```

- negative indexes are supported as well:

```
variable = my_list_name[-1]   # what will variable contain?
```

- if you compare this slide with the previous one, I hope you are wondering why introduce two data structures to do the same things :P

Basic Python's data structures

Lists, Tuples and Dictionaries

- Dictionaries are very different from lists and tuples

```
my_dict = {}
my_dict["key"] = value
```

- the dictionary can't be accessed with an index, you have to get the value referencing it by its key:

```
my_dict["key"] = value
```

- you can change a value pointed by a key simply typing:

```
my_dict["key"] = new_value
```

Python exception handling

- as every other programming language, python has its own exception handling system

- exceptions happens, when you write down code think to what might go wrong and handle it!!

Python exception handling

```
@login_required
def delete_object(request, key):
    try:
        object = mymodel.objects.get(id=key)
        object.delete()
        return redirect('objects')
    except:
        print "Something went wrong”
```

In this case is not specified what kind of exception you are trying to handle, by default the Exception class will be called. This is not the best thing you can do because you are including all the possible exceptions.
So, except the exceptions from the less to the most common and diversify the handling procedure :)

Let's dive into Django

- Django is an open-source MVT framework based on Python
- Django is an ORM
- Django is easy :)

Let's dive into Django

- What does MVT mean?
  - It is an acronym for Model View Template
  - Model = the model is the interface to the database, you define your custom classes will be used in your project
  - View = the view is where all the magic happens :) Here you define your logic and retrieve objects from the database
  - Template = is the presentation logic, where you can visualise the retrieved data or something else

Let's dive into Django

- What does ORM mean?
  - It means Object Relational Mapping
    - It is an abstraction to interact with objects in database without worrying about database connection and other boring stuff

Without ORM

```
book_list = new List();
sql = "SELECT book FROM library WHERE author = 'Linus'";
data = query(sql); // I over simplify ...
while (row = data.next())
{
     book = new Book();
     book.setAuthor(row.get('author');
     book_list.add(book);
}
```

With ORM :)

```
book_list = BookTable.query(author="Linus");
```

Let's dive into Django

- What does ORM mean?
  - It means Object Relational Mapping
    - It is an abstraction to interact with objects in database without worrying about database connection and other boring stuff

database connection (query)

Without ORM

```
book_list = new List();
sql = "SELECT book FROM library WHERE author = 'Linus'";
data = query(sql); // I over simplify ...
while (row = data.next())
{
     book = new Book();
     book.setAuthor(row.get('author');
     book_list.add(book);
}
```

Fetch results

With ORM :)

```
book_list = BookTable.query(author="Linus");
```

Let's dive into Django

- What does ORM mean?
  - It means Object Relational Mapping
    - It is an abstraction to interact with objects in database without worrying about database connection and other boring stuff

Without ORM :(

```
book_list = new List();
sql = "SELECT book FROM library WHERE author = 'Linus'";
data = query(sql); // I over simplify ...
while (row = data.next())
{
     book = new Book();
     book.setAuthor(row.get('author');
     book_list.add(book);
}
```

With ORM :)

```
book_list = BookTable.query(author="Linus");
```

The ORM took care of DB connection and result fetching saving them in a tuple ready to be iterated over

How is a Django project structured?

- The main brick of every Django app is the PROJECT. To start a project you just have to type from the main directory:

```
django-admin.py startproject mysite
```

- Once you create the project you will have the basic project structure:

```
mysite/

    manage.py

    mysite/

        __init__.py

        settings.py

        urls.py

        wsgi.py
```

How is a Django project structured?

- manage.py: it is a python script inside your main project's folder used to set the environment you are working on and to import some cool things you will need to use Django (e.g runserver, makemigrations, migrate). This script is UNIQUE among the whole project

- __init__.py: it is usually an empty file, used by python to treat files' content as modules to be imported by other files. You can write code to initialise your package.

- settings.py: this is the hearth of your Django project. This file contains your apps inside the project or other libraries installed with pip, moreover here is defined the database connection. The name of this file must be UNIQUE inside the whole project, but you can define you custom setting file (i.e my_custom_settings.py). It is particularly useful when you have difficult environments on develop and production machines.

- urls.py: this file is not unique among the whole project, you will find others urls.py files in every app. This file is used to route the urls to specific views (e.g when you type an address in your browser the matching url will route the request to a view). This file is NOT UNIQUE among the whole project, in fact you will usually use another version in every app.

- wsgi.py: this file must be used if you want to deploy your web app in a much resilient way (e.g on Apache or nginx) rather than running it with runserver. Doing this is a very wise choice because if something goes wrong, a service will restart while runserver won't

Much deeper in settings.py file

```
"""
Django settings for idroplanweb project.

Generated by 'django-admin startproject' using Django 1.8.

For more information on this file, see
https://docs.djangoproject.com/en/1.8/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/1.8/ref/settings/
"""

# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/1.8/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = '$0=b3#1ueriza3z(xj30p!zl=g#w7oe$-^uqq2elf#b+zvc@&a'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = ['*', ]


# Application definition

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)

MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
)

ROOT_URLCONF = 'idroplanweb.urls'

WSGI_APPLICATION = 'idroplanweb.wsgi.application'


# Database
# https://docs.djangoproject.com/en/1.8/ref/settings/#databases

#DATABASES = {
#    'default': {
#        'ENGINE': 'django.db.backends.sqlite3',
#        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
#    }
#}


# Internationalization
# https://docs.djangoproject.com/en/1.8/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_L10N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.8/howto/static-files/

TEMPLATE_DIRS = (
    os.path.join(BASE_DIR, 'templates'),
)

STATIC_URL = '/static/'

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static"),
)
```

this option when active will help you in debugging your code. When set to True and something goes wrong, you will see the error and a stack trace in your browser.

INSTALLED_APPS is a tuple when you MUST list all your app (if you want all to work)

MIDDLEWARE_CLASSES is a tuple when you list all the middle-wares your app will be using. What is a middleware? A middleware is some sort of light-weight plugin, allow to modify Views or Request, Response

ROOT_URL_CONF is the path to your project's url file

TEMPLATE_DIRS tells Django when the template files are located

Much deeper in urls.py file

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^myapp1/', include('app1.urls')),
)
```

This is the typical structure of an admin url file.

The first one cares to point all user's request
to localhost:8000/admin to the Django admin

The second one is more important to understand.
It means: include all the urls listed inside app1.urls and make them reachable with this path:

localhost:8000/myapp1/custom_url VVV

Remember that you have to type the whole address including myapp1, otherwise you will get an error:

localhost:8000/custom_url XXX

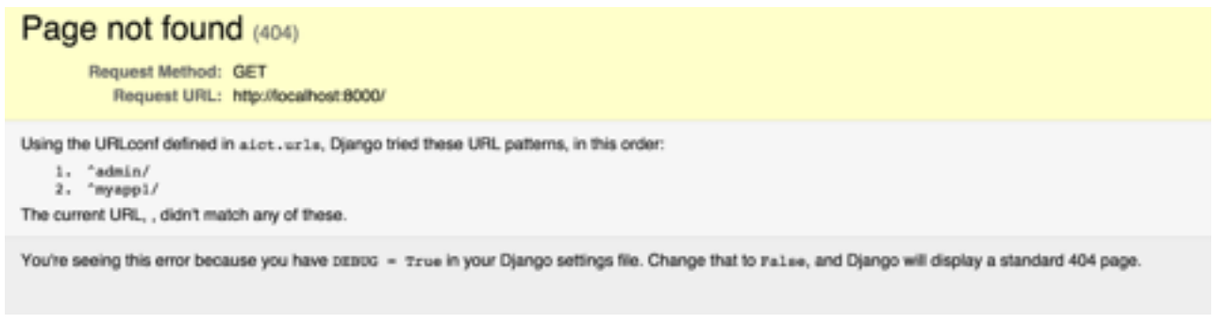
Much deeper in urls.py file

Django matching URL flow for:
localhost:8000/myapp1/custom_url

1. The first thing Django will match is the main urls file (inside your project directory)

2. After that he will include all the other root urls looking for the matching url (if any)

    1. if no url will be matched, Django will raise an Exception (HTTP 404) to alert that the requested url were not found. If you set DEBUG=True in your settings.py you should see something like:



Much deeper in urls.py file

This kind of urls' structure (a main urls.py file to include all the others) is not mandatory, but strongly advisable in order to maintain a separate structure among all the applications and a loosely coupled architecture.

Remember: the file name "urls.py" is just a convention. In fact, while you must not change it while defining the urls for your project, you CAN rename it as whatever you want inside your apps. Don't forget to register them properly in the "urls.py" file inside your project's folder

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^myapp1/', include('app1.whatever_you_want_urls')),
)
```

The Model

As we have seen, the model is the interface to the database.

- Every thing you write down in your model will be wrapped by the ORM and translated into SQL, Postgres or SQLite language, hence the database will reflect your model.

- So pay attention in doing the model, in particular when setting the relations between models if any.

The Model (relationships)

Django supports different type of relationships between models, the most common are:

- ForeignKey when a model A reference a model B (e.g a Car has one Manufacturer)

- ManyToManyField when a model A reference multiple models B, C, D, E, .., .., N (e.g a Car has many Manufacturer)

- OneToOneField when a model A reference a model B (e.g a Car has one Manufacturer)

The Model (relationships)

What is the difference between a OneToOneField (e.g one-to-one relationship) and a ForeignKey?

Conceptually, it is similar to a ForeignKey with unique=True, but the "reverse" side of the relation will directly return a single object instead of a tuple

The Model (queries)

We have defined our model, all perfectly work, how do we query our models?

Simply, with the ORM :)

The Model (queries)

- The result of every query we will do is returned into a query set.

- A queryset is a list

- A queryset cannot be modified by the user

- The queryset will contain model objects

The Model (queries)

There are a lot of functionalities Django gives us to query models:

1. all() - returns all the objects belonging to the queried model

2. delete() - delete one or more objects in the queryset

3. filter() - returns a query set as well, containing only the objects with the precise match in the filter

4. get() - returns **only one** object (not a tuple) based on the condition you provide

> Please note that in case of no object matching the condition in the get(),
> Django will raise an Exception
> so you have to properly handle it

The Model (queries)

Some methods of the ORM listed in the previous slide can be mixed up together, for instance you can:

delete all objects

delete all objects matching a query

The Model (queries)

Some methods of the ORM listed in the previous slide can be mixed up together, for instance you can:

delete all objects - my model.objects.all().delete()

delete all objects matching a query - my model.objects.filter(name="pippo").delete()

The View

The view is responsible to connect model to template and some other things, basically the view is the glue between the model (hence the DB) and the template when you want to show.

Remember that the view is called when Django matches the url, then the view does its things and then return a template.

The view can carry whatever data you want to the template, for example to the results of a query in a table :)

The View

This is an example of a very simple view:

```
from django.shortcuts import render, render_to_response
from app1.models import Student
from django.template import RequestContext


def students(request):
    extra_data = {}
    extra_data["students"] = Student.objects.all()
    return render_to_response('student_list.html', extra_data, context_instance=RequestContext(request))
```

The view is called by the url **students** and will return a template called **student_list.html**

What is the extra_data in second line?

It is a dictionary which allows you to transfer data from view to the template

The Template

- The template is every piece of html file you use to display data

- A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template

- Django templating language supports a lot of tags, and python-like syntax

```
<html>
     <body>
          <table id="students">
               <thead>
                    <tr>
                         <th>Name</th>
                         <th>Surname</th>
                    </tr>
               </thead>
               {% for student in students %}
               <tbody>
                    <tr>
                         <td>{{ student.name }}</td>
                         <td>{{ student.surname }}</td>
                    </tr>
               </tbody>
               {% endfor %}
          </table>
     </body>
</html>
```

The Template

- The template is every piece of html file you use to display data

- A template contains variables, which get replaced with values when the template is evaluated, and tags, which control the logic of the template

- Django templating language supports a lot of tags, and python-like syntax

```
<html>
    <body>
        <table id="students">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Surname</th>
                </tr>
            </thead>
            {% for student in students %}
            <tbody>
                <tr>
                    <td>{{ student.name }}</td>
                    <td>{{ student.surname }}</td>
                </tr>
            </tbody>
            {% endfor %}
        </table>
    </body>
</html>
```

we are simply iterating over the query set returned from the view

The Template (inheritance)

- As you can do with code that repeats inside your project, you can do it with the templates as well, thanks to inheritance

- For instance, if you have a piece of html code to define a table to display a students list, just define in an html file, and than import it in the main html file

```
<html>
    <body>
        {% include “table.html" %}
    </body>
</html>
```

table.html

```
<table id="students">
        <thead>
            <tr>
                <th>Name</th>
                <th>Surname</th>
            </tr>
        </thead>
        {% for student in students %}
        <tbody>
            <tr>
                <td>{{ student.name }}</td>
                <td>{{ student.surname }}</td>
            </tr>
        </tbody>
        {% endfor %}
    </table>
```