Corsi Python 2016

Associazione POuL – Politecnico Open unix Labs
Emanuele Santoro – manu@santoro.tk

Slide disponibili presso http://slides.poul.org

Topics – Di cosa parleremo oggi

- Flask
  - MVC pattern
- SQLAlchemy
  - Definire dei models
  - Eseguire query
- Dokku
  - Cos'é
  - Come si usa

Una (semplice) applicazione completa:

- Controllers
- Models
- Views
- Deploy

Obiettivo di questa lezione

Alla fine di questa lezione saprete:

- Scrivere una semplice applicazione con Flask
  - Scrivere controller
  - Salvare/caricare dati in/da un database relazionale
  - Usare i templates
- Installare e configurare un (semplice) ambiente di produzione (dokku)

Flask

Flask è un **micro**-framework: offre meno cose rispetto ad altri framework, principalmente:

- Request dispatching
- Templating
- Unit-testing
- Poco altro

SQLAlchemy

É un **ORM** (Object-Relational Mapper):

(de|)serializzazione ragionando in termini di classi, oggetti e istanze.

SQLAlchemy

É un **ORM** (Object-Relational Mapper):

(de|)serializzazione ragionando in termini di classi, oggetti e istanze.

Genera automaticamente tabelle e query.

SQLAlchemy

É un **ORM** (Object-Relational Mapper):

(de|)serializzazione ragionando in termini di classi, oggetti e istanze.

Genera automaticamente tabelle e query.

Useremo SQLAlchemy attraverso il wrapper Flask-SQLAlchemy, che rende il tutto ancora piú semplice.

Dokku

Platform as a service (PaaS) di semplice installazione e gestione.

Punto di forza: far partire un'applicazione é semplice quanto fare git-push.

Nell'ultima parte di questa lezione vedremo come installare e configurare Dokku su DigitalOcean.

MVC Pattern

Scomposizione dell'applicazione in tre componenti principali:

- Model
- View
- Controller

MVC – Model

Nel modello MVC, il Model é l'astrazione dei meccanismi di serializzazione e deserializzazione dei dati.

MVC – View

Le Views corrispondono a come verranno presentati all'utente i dati.

Tipicamente corrispondono a i templates.

MVC – Controllers

I controllers sono la componente principale di un'applicazione web realizzata mediante il pattern MVC.

Il controller é il codice che gestisce effettivamente una richiesta HTTP. Effettua operazioni su i dati, riempie i templates e invia la risposta HTTP all'utente.

Flask – installazione

Su GNU/Linux (Debian e derivate):

```
apt-get install python3 python3-pip
pip3 install -user flask
```

Flask – Hello, world!

- Importazione dei moduli necessari
- Decorazione dei controllers
- Codice di avvio

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

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

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

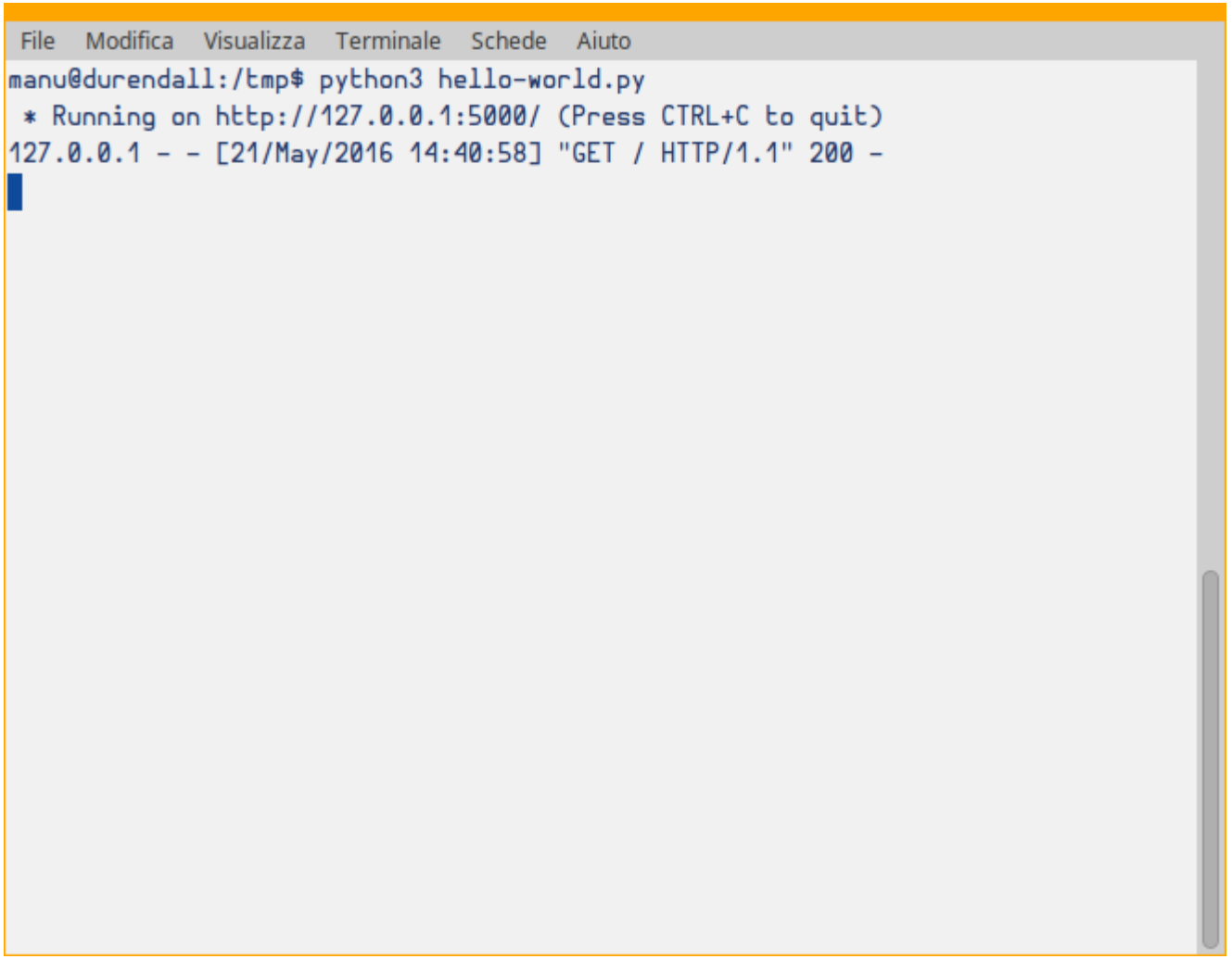
Flask – Hello, world!

Salvare il codice in un file (ad esempio “hello-world.py”) ed eseguire con:

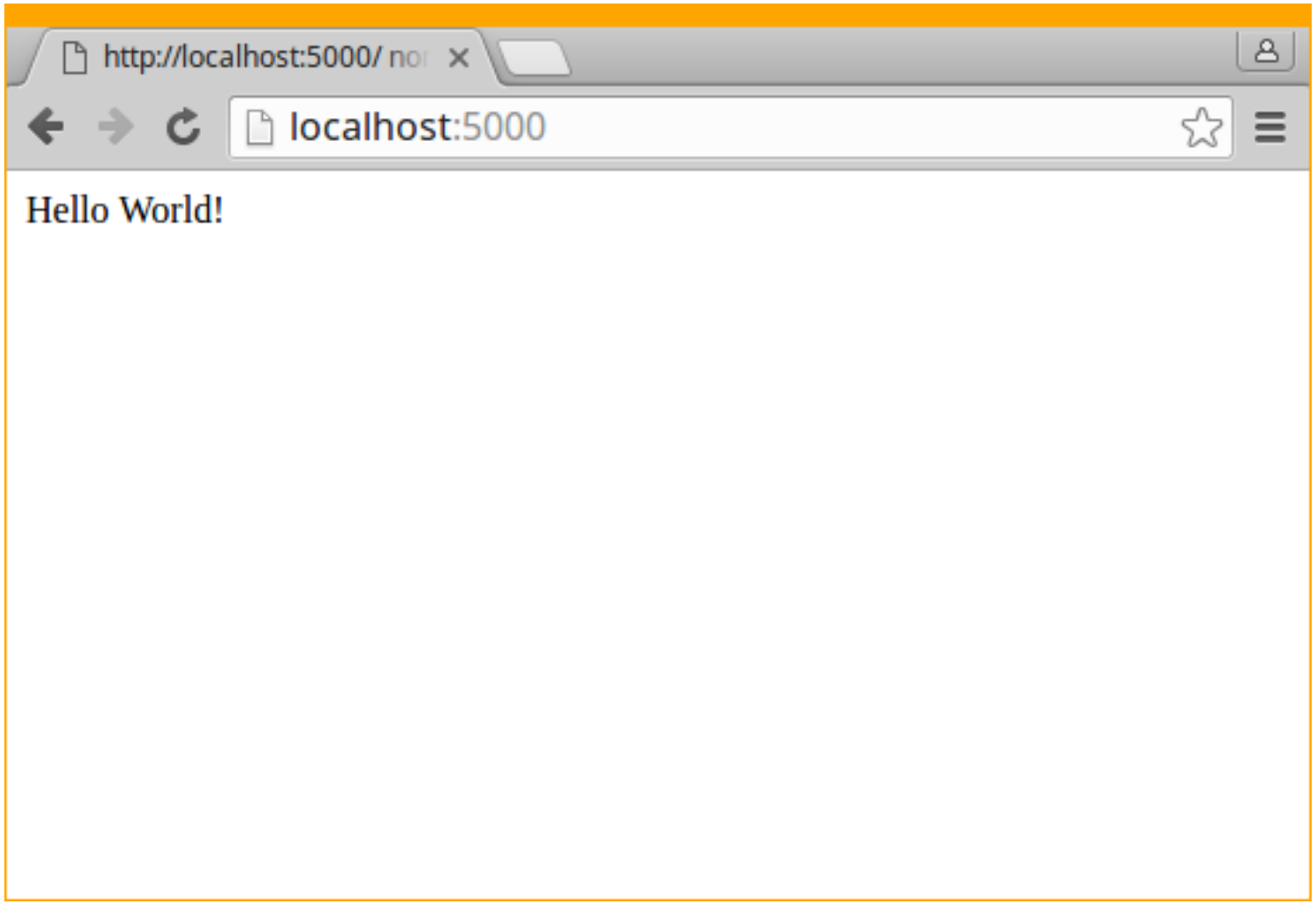
```
python3 hello-world.py
```

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

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

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

Flask – Hello, world!



Flask – Hello, world!



Flask – Anatomia di un controller

```
@app_instance.route(“/target/<var_type:var_name>”)
def controller(var_name) :
    # ‘varname’ sará convertita al tipo var_type
    # … codice
    content = <…>
    # … codice
    return content
```

Flask – anatomia di un controller

- Una normale funzione diventa un controller quando viene “decorata”
- La decorazione associa una parte di url (route) ad un a funzione
- Il corpo della risposta é costituito da ció che il controller ritorna

Flask – parametri della richiesta HTTP

L'oggetto **request** contiene una rappresentazione della richiesta http.

request.forms contiene tutti i vari parametri della richiesta HTTP

Flask – l'oggetto `request`

É una variabile globale che contiene una rappresentazione della richiesta HTTP.

Flask – l'oggetto `request`

É una variabile globale che contiene una rappresentazione della richiesta HTTP.

Tecnicamente, l'oggetto request é un proxy alla rappresentazione della richiesta HTTP per il thread corrente.

Flask – l'oggetto `request`

É una variabile globale che contiene una rappresentazione della richiesta HTTP.

Tecnicamente, l'oggetto request é un proxy alla rappresentazione della richiesta HTTP per il thread corrente.

Effetto: codice piú semplice da scrivere (e da leggere!)

Flask – l'oggetto `request`

Parti utili dell'oggetto **`request`**:

- **`request.form`**: dizionario che contiene i contenuti di un eventuale form
- request.method: GET, POST, PUT, HEAD...
- request.headers: dizionario con le intestazioni della richiesta HTTP
- request.cookies: dizionario con i cookie
- request.files: dizionario nome→file, per la gestione dell'upload di files

Flask – templating

Flask usa la libraria Jinja2 per renderizzare i templates.

Flask – templating

Flask fornisce delle funzioni per riempire dei template usando Jinja2.

```
from flask import render_template

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

Flask – templating

Flask fornisce delle funzioni per riempire dei template usando Jinja2.

```
from flask import render_template

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

La sintassi dei template é molto semplice.

```
<!-- templates/hello.html -->
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello World!</h1>
{% endif %}
```

Jinja2 – Riempire dei templates

Poche cose da sapere:

- Usare `render_template` per riempire un modello
- Il valore delle variabili deve essere passato a `render_template` come chiave=valore

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def index() :
    var1 = some_computation()
    var2 = some_other_computation()
    return render_template("page.html",
                                    var1=var1,
                                    var2=var2)
```

Jinja2 – Sintassi dei template

Jinja2 interpreterá tutto ció che é compreso tra ‘{{{’ e ‘}}}’ oppure ‘{%’ e ‘%}’.

Jinja2 – Sintassi dei template

Jinja2 interpreterá tutto ció che é compreso tra ‘{{{’ e ‘}}}’ oppure ‘{%’ e ‘%}’.

É possibile usare blocchi if/then/else e cicli for/while

Jinja2 – Sintassi dei template

Jinja2 interpreterá tutto ció che é compreso tra ‘{{{’ e ‘}}}’ oppure ‘{%’ e ‘%}’.

É possibile usare blocchi if/then/else e cicli for/while

Nello scope di un template sono disponibili gli oggetti request, session e g.

Jinja2 – Sintassi dei template

Jinja2 interpreterá tutto ció che é compreso tra ‘{{{’ e ‘}}}’ oppure ‘{%’ e ‘%}’.

É possibile usare blocchi if/then/else e cicli for/while

Nello scope di un template sono disponibili gli oggetti request, session e g.

Si assume che i templates siano nella cartella `templates/`

Flask-SQLAlchemy

- Semplifica l'uso di SQLAlchemy
- Effettua le scelte piú comuni
- Piú facile da importare
- Piú integrato con i controller e tutto il resto

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
                'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

## db adesso é un oggetto che ci consente
## di accedere al database
```

SQLAlchemy – definire un model

É possibile definire sia la classe che rappresenta i nostri dati sia le tabelle nel database in maniera dichiarativa e soprattutto in termini di classi Python.

```
class User(db.Model):
    id = db.Column(db.Integer,
                   primary_key=True)

    username = db.Column(db.String(80),
                         unique=True)

    email = db.Column(db.String(120),
                      unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

SQLAlchemy – creare le tabelle nel database

Una volta definite le classi é possibile generare le tabelle nel database.

Per fare questo é necessario chiamare il metodo `SQLAlchemy.create_all()`

```
from application import db
db.create_all()
```

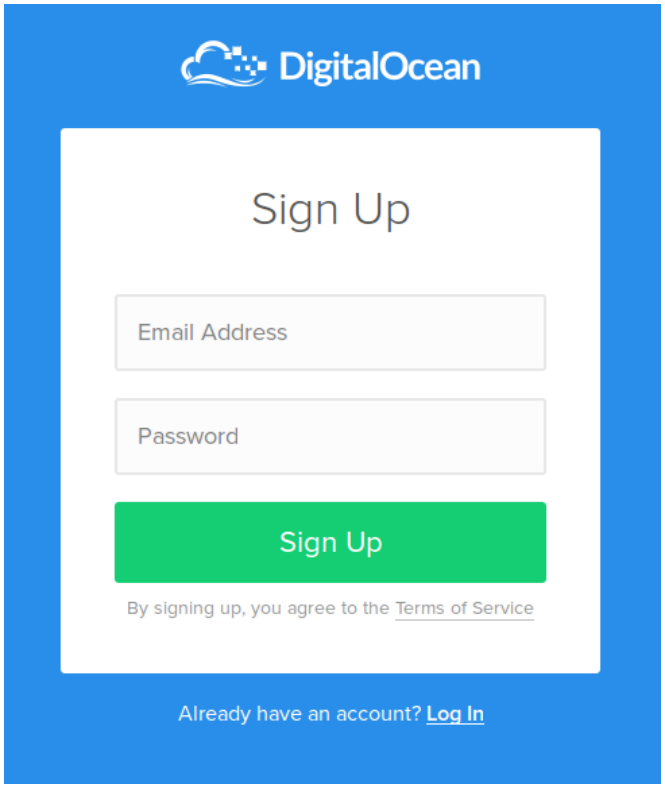
SQLAlchemy – salvare un oggetto nel DB

```
admin = User(“admin”, “admin@example.com”)
guest = User(“guest”, “admin@example.com”)
# crea due utenti

db.session.add(admin)
db.session.add(guest)
# aggiunge le due righe corrispondenti nella transazione

db.session.commit()
#effettua la transazione
```

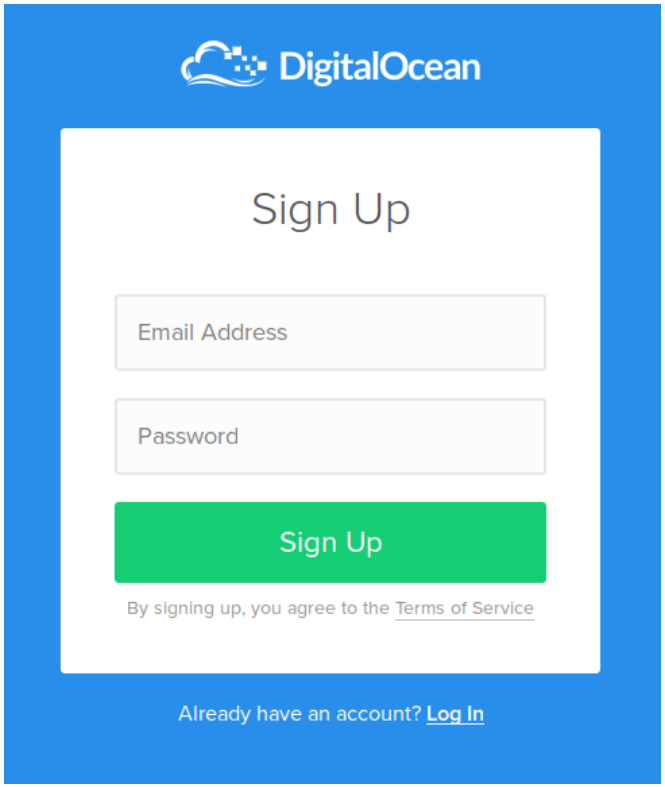
SQLAlchemy – eseguire una query

```
users = User.query.all()
[<User u'admin'>, <User u'guest'>]
admin =
User.query.filter_by(username='admin').first()
<User u'admin'>
```

Dokku – installazione



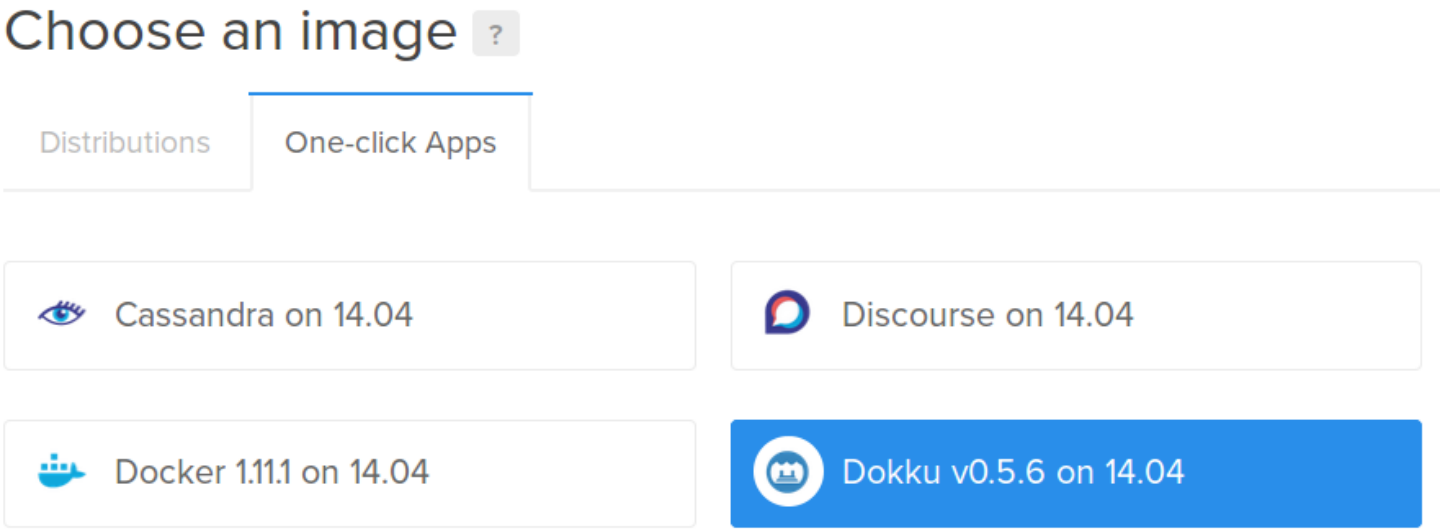
Dokku – installazione



Suggerimento: https://m.do.co/c/867be540644c

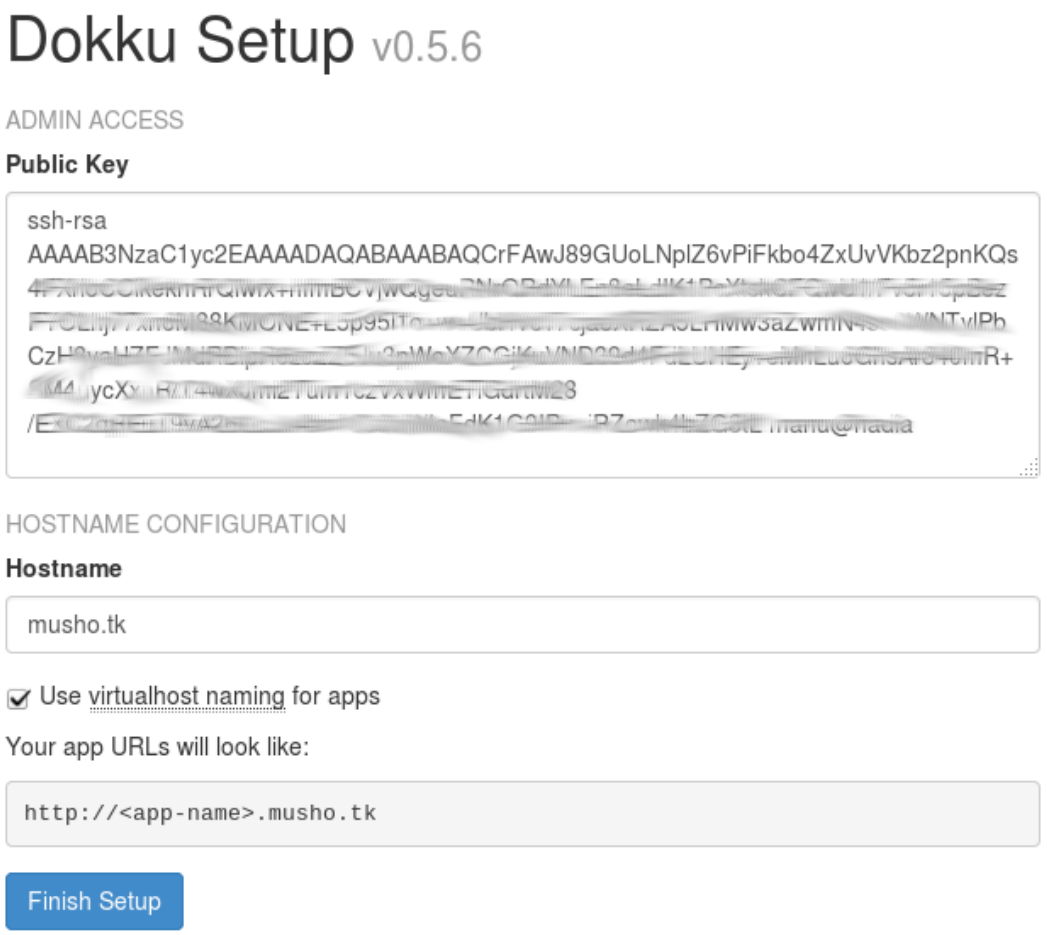
Referral link: ottenete 10$ di credito gratis (due mesi di hosting)

Dokku – installazione



Risultato: in circa un minuto otteniamo una macchina virtuale con Ubuntu e Dokku installati e configurati.

Dokku – configurazione



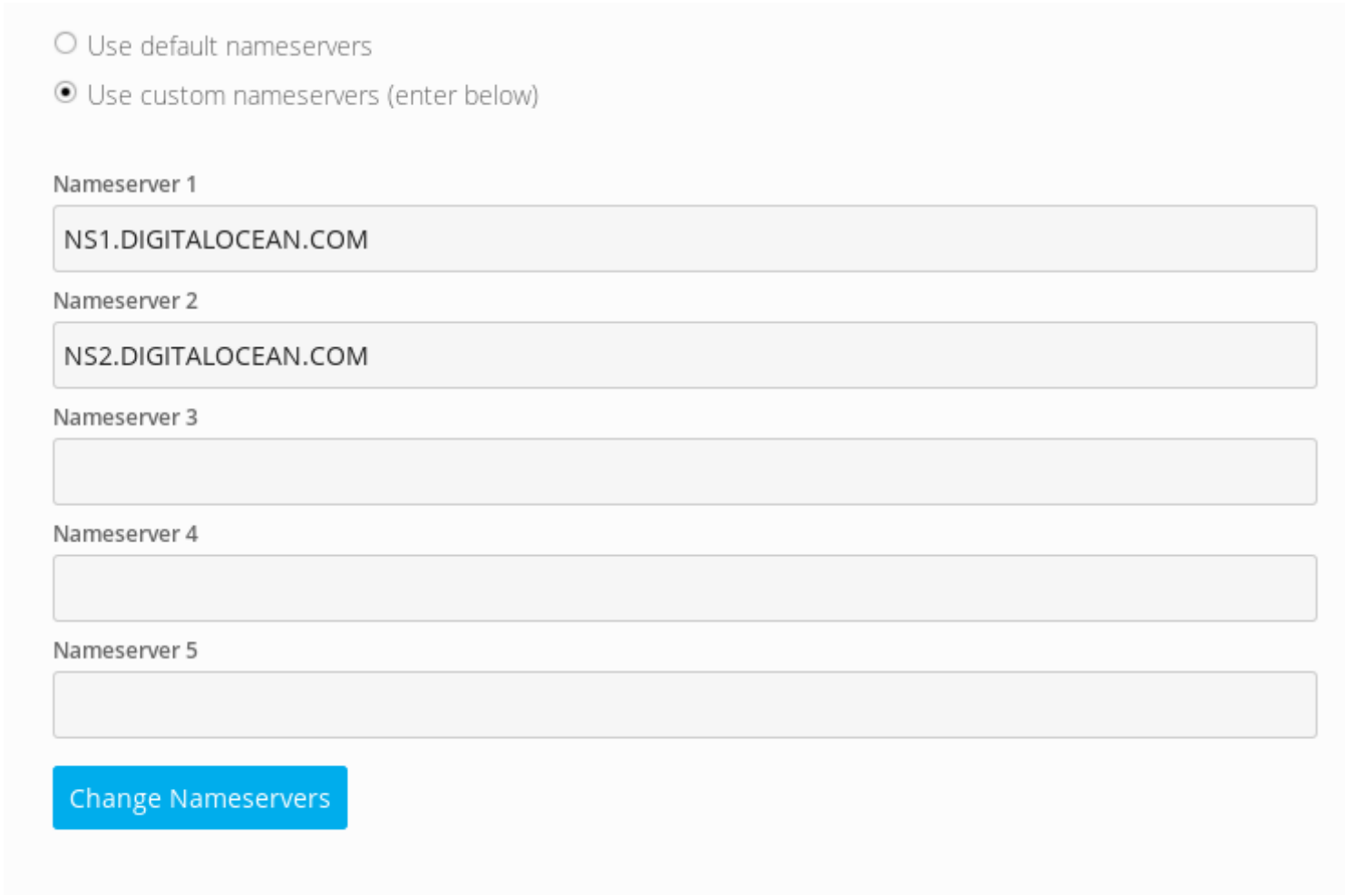
- Chiave RSA
- Dominio
- Virtualhost naming

Dokku – installazione

Img	Name	IP Address
● 	dokku 512 MB Memory / 20 GB Disk / FRA1	46.101.213.13

É arrivato il momento di associare il nostro dominio alla nostra macchina virtuale

Dokku – configurazione DNS (1)



Dokku – configurazione DNS (2)

A	@	178.62.110.83	Save	Remove
CNAME	*	musho.tk.	Save	Remove
NS	ns1.digitalocean.com.		Save	Remove
NS	ns2.digitalocean.com.		Save	Remove
NS	ns3.digitalocean.com.		Save	Remove

Dokku: Amministrazione

É sufficiente aprire una shell di root sull'host ed eseguire il comando **dokku**, eventualmente con i vari sottocomandi

```
root@dokku:~# dokku
Usage: dokku [--quiet|--trace|--rm-container|--rm|--force] COMMAND <app> [command-specific-options]

Primary help options, type "dokku COMMAND:help" for more details, or dokku help --all to see all commands.

Commands:

    apps            List your apps
    certs           Manage Dokku apps SSL (TLS) certs
    checks          Show zero-downtime status
    config          Display all global or app-specific config vars
    docker-options  Display app's Docker options for all phases (or comma separated phase list)
    domains         List domains
    enter           Connect to a specific app container
    events          Show the last events (-t follows)
    help            Print the list of commands
    logs            Show the last logs for an application
    ls              Pretty listing of deployed applications and containers
    nginx           Interact with Dokku's Nginx proxy
```

Dokku: struttura di un'applicazione

Alcuni accorgimenti sono necessari per far funzionare un'applicazione con Dokku:

Dokku: struttura di un'applicazione

Alcuni accorgimenti sono necessari per far funzionare un'applicazione con Dokku:

- Procfile

Dokku: struttura di un'applicazione

Alcuni accorgimenti sono necessari per far funzionare un'applicazione con Dokku:

- Procfile
- runtime.txt

Dokku: struttura di un'applicazione

Alcuni accorgimenti sono necessari per far funzionare un'applicazione con Dokku:

- Procfile
- runtime.txt
- requirements.txt

Dokku: Procfile

Contiene delle direttive per avviare l'applicazione

Esempio:

```
web: gunicorn hello:hello --preload -b 0.0.0.0:$PORT --log-file -
## Nota l'uso di $PORT
```

Dokku: requirements.txt

Dipendenze (librerie) dell'applicazione

Esempio:

```
flask==0.10.1
gunicorn==19.4.5
```

Dokku: runtime.txt

Serve a specificare una versione specifica del runtime che stiamo usando.

Nel nostro caso, usiamo Python 3.4.0

Se avete intenzione di usare Python 2.x, ripensateci.

```
python-3.4.0
```

Dokku: deploy di un'applicazione

Pochi step:

- Creare l'applicazione
- Installare i plugin necessari
- Creare le risorse/servizi necessari
- Collegare i vari servizi
- `git push dokku master`

Dokku: deploy di un’applicazione

Pochi step:

- Creare l’applicazione
- Installare i plugin necessari
- Creare le risorse/servizi necessari
- Collegare i vari servizi
- `git push dokku master`

- dokku apps:create <app>
- dokku plugins:install <...>
- dokku postgres:create <db>
- dokku

Dokku: hello, world!

Installiamo il classico “Hello, world!”

```
from flask import *

hello = Flask(__name__)

@hello.route("/")
def welcome() :
    return "Hello, world!"


if __name__ == "__main__" :
    hello.run()
```

Dokku: hello, world (deploy)

- `ssh root@musho.tk dokku apps:create hello`
- `git remote add dokku dokku@musho.tk:hello`
- `git push dokku master`

Alla fine del processo di deploy la nostra applicazione sará disponibile all'url http://hello.musho.tk

DEMO!

In questa demo scriveremo una piccola applicazione: un url-shortener.

Links