Deploying the Application

CITS3403 and CITS5505 Agile Web Development

The Flask Mega-Tutorial
Miguel Grinberg
Chapters 17 and 18

Semester 1, 2024

Deploying a project

- In this unit we have gone through the process of building a web application and deploying it via the computers local host.
- However, web applications work best on the web, so in this lecture we'll go through some options for deploying the application via a URL accessible worldwide.
- We will look at using industrial strength servers, and consider deploying via:

1. A Linux virtual machine
2. Your own server
3. Heroku.

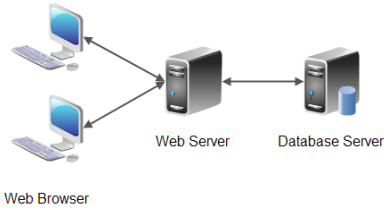
Upgrading to production-grade tools

Upgrading to production-grade tools

- We used the Flask development webserver and SQLite to allow fast development.
- However, these aren't suitable for deploying publicly, for security and scalability reasons.
- As we have been using SQL-Alchemy, we will look at using MySQL instead of SQLite.
- We will use Gunicorn and nginx as alternatives to the Flask development server to run the Flask app.
- Only tweaks to the configuration should be needed to set these up!

Why replace SQLite?

- The crucial difference between MySQL and SQLite is that the former is a server database rather than an embedded database like the latter.
- This means that it can run on an entirely separate machine (on hardware optimised for intensive IO operations), and act like a server for multiple web-servers.



- This means it performs better in production, and scales well with multiple web servers.
- It also has inbuilt authentication methods that keeps your data secure even if your virtual machine is compromised.

Replacing SQLite with MySQL

- For the moment we will keep the database on the same machine. We can install MySQL on our web server using `apt` (a Linux package manager) as shown on the previous slide.

```
$ sudo apt-get -y update
$ sudo apt-get -y install python3 python3-venv python3-dev
$ sudo apt-get -y install mysql-server postfix supervisor nginx git
```

- We open MySQL and create a special user to handle the database transactions.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)
```

- Set the username to your app name and insert an appropriate password.

```
mysql> create user 'pair-up'@'localhost' identified by 'top_secret';
Query OK, 0 rows affected (0.01 sec)

mysql> grant all privileges on pairup.* to 'pair-up'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

Running MySQL on the webserver

- We need to install a driver for MySQL.

```
pip install pymysql
```

- We then set the DATABASE_URL to the new database

```
export DATABASE_URL="mysql+pymysql://pair-up:top_secret@localhost:3306/pairup"
```

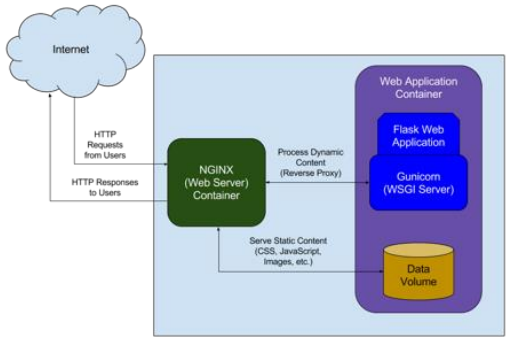
- As Flask automatically reads the DATABASE_URL variable we can simply run `flask db migrate` and `flask db upgrade` to create the database.

```
(cits3403-env) tim@server ~/CITS3403/cits3403-pair-up $ flask db migrate
INFO  [alembic.runtime.migration] Context impl MySQLImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.env] No changes in schema detected.
(cits3403-env) tim@server ~/CITS3403/cits3403-pair-up $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl MySQLImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
(cits3403-env) tim@server ~/CITS3403/cits3403-pair-up $
```

- Now the app will work as before, but we now have a full database server.

Why replace the Flask server?

- In a production system, it is once again common to decompose the actual web-server into two separate servers:
  1. A public facing reverse-proxy server that: i) serves cached static content, ii) handles load balancing, iii) handles SSL encryption
  2. One or more private origin servers that contain the actual logic of the web application.



Gunicorn as the origin server

- We will use Gunicorn as the origin server (installable as Pip).
- Gunicorn implements the Web Service Gateway Interface, a Python standard for accessing web-requests.

```
(venv) $ gunicorn -b localhost:8000 -w 4 microblog:app
```

- Just like Flask, Gunicorn also comes with the ability to configure the webserver itself via a `.conf` file.

/etc/supervisor/conf.d/microblog.conf: Supervisor configuration.

```
[program:microblog]
command=/home/ubuntu/microblog/venv/bin/gunicorn -b localhost:8000 -w 4 microblog:app
directory=/home/ubuntu/microblog
user=ubuntu
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
```

Nginx as the reverse-proxy server

- For dealing with external traffic and serving static content, we will use nginx as the outward facing reverse-proxy server (*install with apt)*.
- Nginx does several things:

1. It routes all traffic through HTTPS (port 443) so it is encrypted.
2. It caches any static data served, to improve efficiency.
3. It handle the public-key encryption/decryption, using a secure certificate. This means the origin server doesn't have to perform the expensive decryption/encryption stage itself.

Configuring nginx

- Again, nginx can be configured using a configuration file.
- SSL certificates can be generated locally, or if you can get an externally signed one from LetsEncrypt.org.
- This requires you knowing the final domain name for your server.

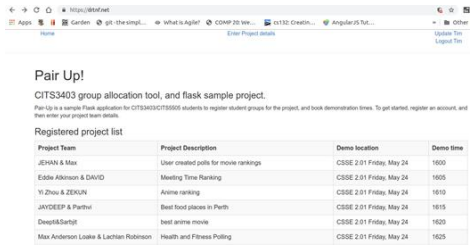
```
server {
    # listen on port 443 (https)
    listen 443 ssl;
    server_name _;

    # location of the self-signed SSL certificate
    ssl_certificate /home/ubuntu/microblog/certs/cert.pem;
    ssl_certificate_key /home/ubuntu/microblog/certs/key.pem;

    # write access and error logs to /var/log
    access_log /var/log/microblog_access.log;
    error_log /var/log/microblog_error.log;

    location / {
        # forward application requests to the gunicorn server
        proxy_pass http://localhost:8000;
        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /static {
        # handle static files directly, without forwarding to the application
        alias /home/ubuntu/microblog/app/static;
        expires 30d;
    }
}
```

Deploying the website

- Although usually the reverse-proxy and the origin servers are deployed on different machines, it's perfectly possible to deploy them on the same machine.
- When we deploy the website, the server will listen for requests on port 80 and forward them through port 443 so they are encrypted end to end.

```
sudo nginx start
```

- The servers run in a daemon thread, so they persist after the session has ended (i.e. you log out).



Deploying via Linux virtual machines

A Linux virtual machine

- A Linux server is the traditional way of deploying a web application.
- The server runs applications listening to ports for requests and serves those requests.
- As most people don't want to worry about the physical infrastructure they use hosting solutions.
- Amazon, Digital Ocean, A2, Azure, AliBaba all offer hosted virtual machines, rentable for about $10 a month.
- Typically, you register an account and request an instance. You have a user account with login details, that allows you to ssh to the virtual machine and configure and deploy your application from the command line.

Securing the virtual machine

- It's sensible to be reasonably paranoid when using a publicly accessible machine. Steps that should be taken to secure to server are:
  1. Remove passwords for login and use key files instead. You can generate a public-private key pairing that is stored in your personal machine.
  2. Disable root logins. Someone with root user credentials will have complete access to your server.
  3. Use a firewall to only accept traffic on ports 22 (SSH) 80 (HTTP) and 443 (HTTPS). People will scan open ports for any vulnerabilities.
  4. Route all web requests through HTTPS. HTTP traffic is transmitted in plaintext and is visible to intermediate nodes.

```
198.108.67.48 - - [20/May/2019 12:25:52] "GET / HTTP/1.1" 200 -
195.154.78.242 - - [20/May/2019 12:58:41] code 400, message Bad HTTP/0.9 request type ('\x03\x00\x00/*à\x00\x00\x00\x00\x00Cookie:')
195.154.78.242 - - [20/May/2019 12:58:41] "□/*àCookie: mstshash=Administr" 400 -
185.92.73.88 - - [20/May/2019 13:22:08] code 400, message Bad HTTP/0.9 request type ('\x03\x00\x00*%à\x00\x00\x00\x00\x00Cookie:')
185.92.73.88 - - [20/May/2019 13:22:08] "□*%àCookie: mstshash=Test" 400 -
185.92.73.88 - - [20/May/2019 13:22:09] code 400, message Bad HTTP/0.9 request type ('\x03\x00\x00*%à\x00\x00\x00\x00\x00Cookie:')
185.92.73.88 - - [20/May/2019 13:22:09] "□*%àCookie: mstshash=Test" 400 -
196.50.30.3 - - [20/May/2019 17:02:59] code 400, message Bad HTTP/0.9 request type ('\x03\x00\x00+&à\x00\x00\x00\x00\x00Cookie:')
196.50.30.3 - - [20/May/2019 17:02:59] "□+&àCookie: mstshash=hello" 400 -
130.95.254.4 - - [20/May/2019 21:26:54] "GET / HTTP/1.1" 200 -
```

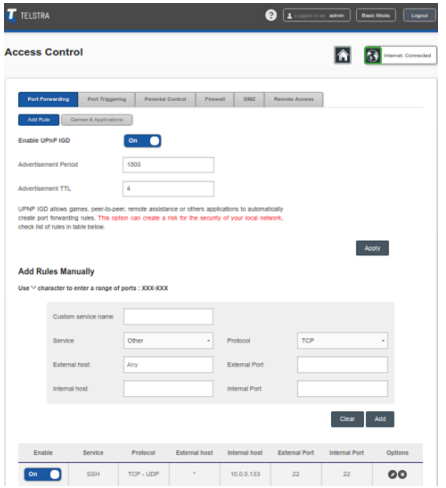
Deploying via your own physical server

Running your own server

- In the past, running your own server would have been very expensive.
- It is now much cheaper, using a wireless router and home broadband connection and Raspberry Pi, or similar small computer.
- A Raspberry Pi is a single board computer costing less than $60, with 1GB Memory and 1.4 Ghz processor. This is easily enough to power a small web server.
- Raspberry Pi's come with a variation of Linux (Raspbian) and can be configured in the same way as the Linux servers we have discussed.
- You normally need a keyboard and monitor to configure the Raspberry Pi, but once you have it on your home network, you can use ssh and treat it like any normal Linux server.

Deploying to Raspberry Pi

- You can connect a Raspberry Pi (or any computer) to a wireless router, and then access the administrator interface of the router.
- Using that you can open ports to the computer (80, 23 and 443 are usually enough), and then access the application via the IP address of the router.



Deploying via Cloud containers

Platform-as-a-service

- When deploying apps (especially in less-mature / early-stage projects), it's easiest at time of writing to use the cloud
- There exist many Platform-as-a-Service (PaaS) products, largely doing away with having to set up a server, manage storage and maintain infrastructure
- There are almost always pay to play and scale with playtime, but almost all have free or hobby tiers
- They also often are limited to different OS / languages

render

DigitalOcean

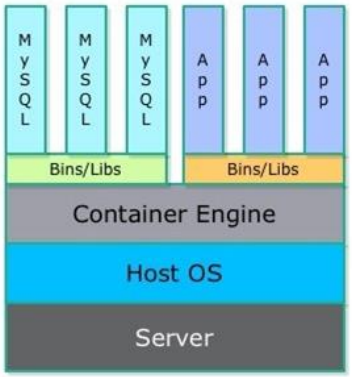
HEROKU

Fly.io

There are many services available: the choice is an individual one, but we've been using Heroku the longest, so we'll be using that for demonstration. At time of writing HN hates Heroku and espouses Fly.io

Containers

- When creating software such as a web app, there can be issues with versioning, package management and cross-platform reliability.
- Solution: put your web app in its own little transportable box that stays the same no matter what.
- This is known as a container
- Containers standardise the software you've written, along with all the dependencies, so it runs the same from one environment to another



An example of a container stack: anything above is bundled into the container, to be run on the Host OS and deployed on the server [1]

[1] https://www.mystechnologies.com/blog/a-complete-guide-to-cloud-containers/

Containers in the Cloud

- We will look at two micro-hosting solutions: Docker and Heroku.

- These services don't offer full Linux virtual machines, but rather containers which run a single process as a service (*software as a service*).
- The containers will install code directly from a Git repository, install all the dependencies, and execute and initialise the app.
- As the containers don't have persistent memory, 3rd party cloud databases are used.

Docker

- Docker software includes functionality to create Docker container "images", as well as the Docker Engine as a run environment
- These container images are run on the Docker Engine, and no matter what the environment is, the app will be the same
- Docker has different tiers of containers depending on your needs (security, space etc.)

Dockerfiles

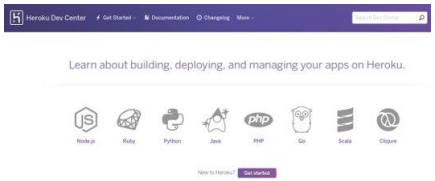
- Docker (as a program) can create an environment automatically from a script called a Dockerfile
- Each project can contain multiple files (depending on environments such as development and production), and these files are basically a script to create the container
- `docker build' can run this script, and the container will be created
- For further information on Dockerfile scripts and best practices, go here

```
Dockerfile > ...
1   # Get the base image
2   FROM python:3.10.3-slim-buster
3
4   # Set the working directory
5   WORKDIR /usr/src/app
6
7   # Set any relevant environment variables
8   ENV PYTHONDONTWRITEBYTECODE 1
9   ENV PYTHONUNBUFFERED 1
10
11  # Add any system dependencies
12  RUN apt-get update \
13      && apt-get -y install netcat gcc postgresql \
14      && apt-get clean
15
16  # Install requirements from file
17  COPY ./requirements.txt .
18  RUN pip install -r requirements.txt
19
20  # Add the Flask app
21  COPY . .
22
23  # Set the entrypoint with a script
24  COPY ./entrypoint.sh .
25  RUN chmod +x /usr/src/app/entrypoint.sh
```

Dockerfile from a deployed Flask project

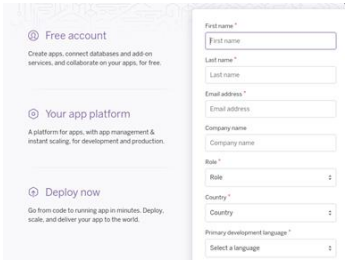
Heroku: a heroic haiku

- Heroku is a cloud platform that you can upload your (ideally containerised) app to and then use built-in functionality to build, maintain and scale your project
- Popular as very easy to use: you basically commit and push your project to the host URL like you would a Git repo. There are also good tutorials available.
- Popular and well recognised even though reputation for being a bit expensive.
- There used to be a free tier but no longer. Very similar free alternatives, usually with less features, are available, e.g. PythonAnywhere.



Setting up Heroku

- To launch your app on Heroku, you need a Heroku account, and a Git repo of your project.
- Once you register an account you should install Heroku CLI, a command line interface that lets you set up and configure your Heroku instance from your local machine.
- Build your app as usual using Git.
- Heroku does not have persistent memory, but offers free Postgres databases as a service, so we need to include psycopg2, and Gunicorn.
- Freeze the requirements and then create the app. This initialises a git remote to push our code to.




```
$ heroku apps:create flask-microblog
Creating flask-microblog... done
http://flask-microblog.herokuapp.com/ | https://git.heroku.com/flask-microblog.git
```

Databases as a service

- As the Heroku container does not have persistent memory, we need an external database.
- We can use Heroku addons to add a PostgreSQL database (very similar in features to MySQL).
- When we request a database for our project, Heroku initialises it, and sets DATABASE_URL in our project settings to point to it, so we don't have to do anything expect migrate our database structure.
- There are many other Database-as-a-service providers we could use: MLab for mongo databases, Azure and AWS offer every type of database as a service.

```
$ heroku addons:add heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on flask-microblog... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-parallel-56076 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

Deploying on Heroku

- Given we went through many steps to set up the Flask environment on our local machine deploying to Heroku is surprisingly easy.
- We set any system variables we need (in this case FLASK_APP), and Heroku has already set others we require (DATABASE_URL and PORT)
- We give it the basic commands to run on initialisation in a Procfile, a configuration file, that is stored in the root of our Git repo.
- To launch the application, we just push our Git remote to the remote that was created with the project. Heroku will detect what language we are using (Python), install Python, pip and all our requirements from reuqirements.txt, and then run the commands in the Procfile.

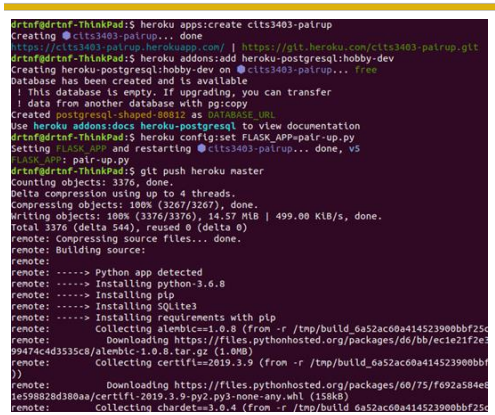
```
$ heroku config:set FLASK_APP=microblog.py
Setting FLASK_APP and restarting flask-microblog... done, v4
FLASK_APP: microblog.py
```

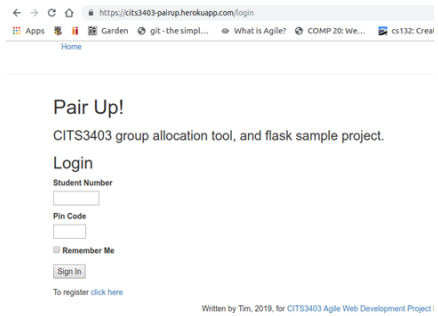
Procfile: Heroku Procfile.

```
web: flask db upgrade; flask translate compile; gunicorn microblog:app
```

```
$ git push heroku deploy:master
Counting objects: 247, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (238/238), done.
Writing objects: 100% (247/247), 53.26 KiB | 3.80 MiB/s, done.
Total 247 (delta 136), reused 3 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-3.6.2
remote: -----> Installing pip
remote: -----> Installing requirements with pip
...
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 57M
remote: -----> Launching...
remote:        Released v5
remote:        https://flask-microblog.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/flask-microblog.git
 * [new branch]      deploy -> master
```

Full deployment





The Future of the Web

The Future of the Web

- In this course we have tried to focus on the fundamental technologies of the web: HTML, CSS, JavaScript, and web application frameworks, RESTful architectures, AJAX, Sockets.
- We have also looked at the agile software development process, and key tools like GitHub.
- While the core technologies such as REST, HTTP, and AJAX are reasonably persistent, the web is a rapidly changing domain, with many trends, and many new emerging technologies.
- What will the web look like in the future?

Cloud computing

- As with deployment on Heroku… cloud services are everywhere.
- High bandwidth permanent online devices and the principles of REST between the line between your personal device and a cloud service is increasingly blurred.
- Interestingly cloud is blurring into desktop apps, such as word processing, and the desktop SOE is disappearing.
- Many music/movie streaming services are changing the notion of libraries and owning media entirely.
- Software as a service is a huge industry and many of the emerging business models use this concept.

Want to know more?
Study CITS5503, Cloud Computing

The Internet of Things

Want to know more?
Study CITS5506,
Internet of Things

- The internet of things is the extension of the internet to smart devices. It focuses on the message passing interfaces and infrastructure, rather than the end user interface.
- Smart homes, adaptive lighting, smart vehicles, and connected devices all use basic web technologies for remote control.
- Similarly, they can gather analytics and adapt to their usage to improve user experience.
- But this has big implications for privacy: the web extends seamlessly to our day-to-day life without us being conscious of it.

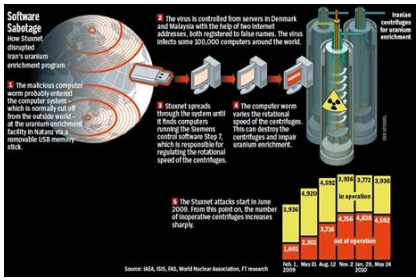
The Semantic Web

- Sir Tim Berners-Lee was one of the creators of the web in 1989 but has for a long time been championing the semantic web.
- This highlights the difference between content and meaning. Semantics is meaning. We use HTML and CSS to differentiate content and presentation, but if we also have a standard for meaning, then a smart search engine, can understand whether web services are selling tickets, or promoting events, or documenting history etc.
- This has big implications for artificial intelligence and automatic assistants.

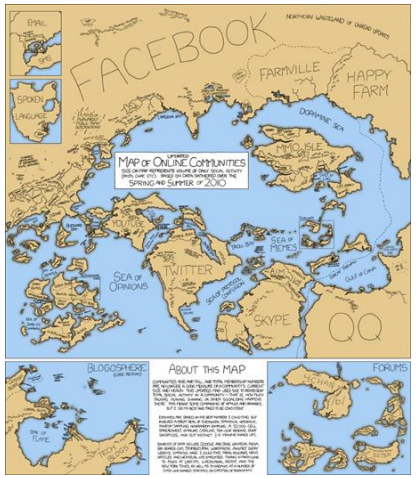
Want to know more?
Study CITS3011 – Intelligent Agents, and
CITS3005 – Knowledge Representation.

Cybersecurity

- Security is the ongoing challenge of the web. The web has evolved and wasn't designed for security, so securing services is a constant battle.
- The prevalence of the web means that small flaws can have major implications in defence and society and cyberwarfare is consuming significant defence spending.
- Big examples are the Stuxnet attacks on Iran's nuclear program, the Ashley Madison hack, and Anonymous's attack on HB Gary.



Want to know more?
Study CITS3006 - penetration testing

Final exam

- The exam is 2 hours long.
- In-person exam sat on Saturday 1st of June at 9am (sorry, didn't choose the time!)
- Covers all topics covered by lectures.
- The Group project has already assessed your ability to write code.
- The exam will instead examine your understanding of the theoretical topics covered, and the functioning of the web.
- Therefore no calculators allowed and no cheatsheets provided.
- Good luck!