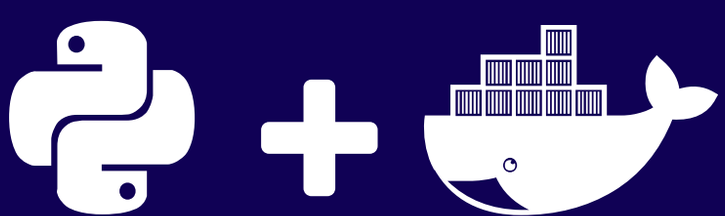
PYTHONS IN A CONTAINER

LESSONS LEARNED DOCKERIZING PYTHON MICROSERVICES...
...THE HARD WAY



Presented by Dorian Puła / @dorianpula

INTRODUCTION

WHO AM I?

Software Development Engineer @ Points

- ✈ Develop eCommerce platform for Loyalty Programs (Buy, Gift + Transfer points)
- ⚗ Flask REST APIs + Apps
- Dockerized microservices

Open Source

- </> Rookeries - Yet Another CMS
- Contributed to Fabric, Ansible & core Python
- Ansible roles for NGINX, UWSGI, NodeJS and Supervisor

WHAT IS THIS TALK ABOUT?

- Lessons learned using Docker for Flask REST API and apps.
- Incorporating various tools that Docker and docker-compose provide for better DevOps workflow.
- The usefulness of unlearning some accepted patterns in Python development, when working with Docker.

WHAT IS THIS TALK *NOT* ABOUT?

- An introduction to basic Docker or WSGI apps.
- Docker Machine (cool as it is).
- Advanced Docker wizardery. *See Dockercon next week for that.*
- An exposé on why you *must or must not* use Docker.

MICROSERVICES + DOCKER

EXAMPLE APP + API - POINTS FOR PYTHONISTAS

- Imagine having to build an app for a new hypothetical loyalty program for sprint contributers at PyCon.
- Earn points per commit or issue resolved. Redeem points for essential sprint goods. (e.g. coffee, poptarts or dogecoin.)
- Has the following components:
  - REST API
  - Frontend App
  - Redemption of Points
  - User + Project Registration/Linking
  - Database

WHY A MICROSERVICES ARCHITECTURE?

Imagine implementing said example using a microservices architecture, with multiple services built by multiple teams.

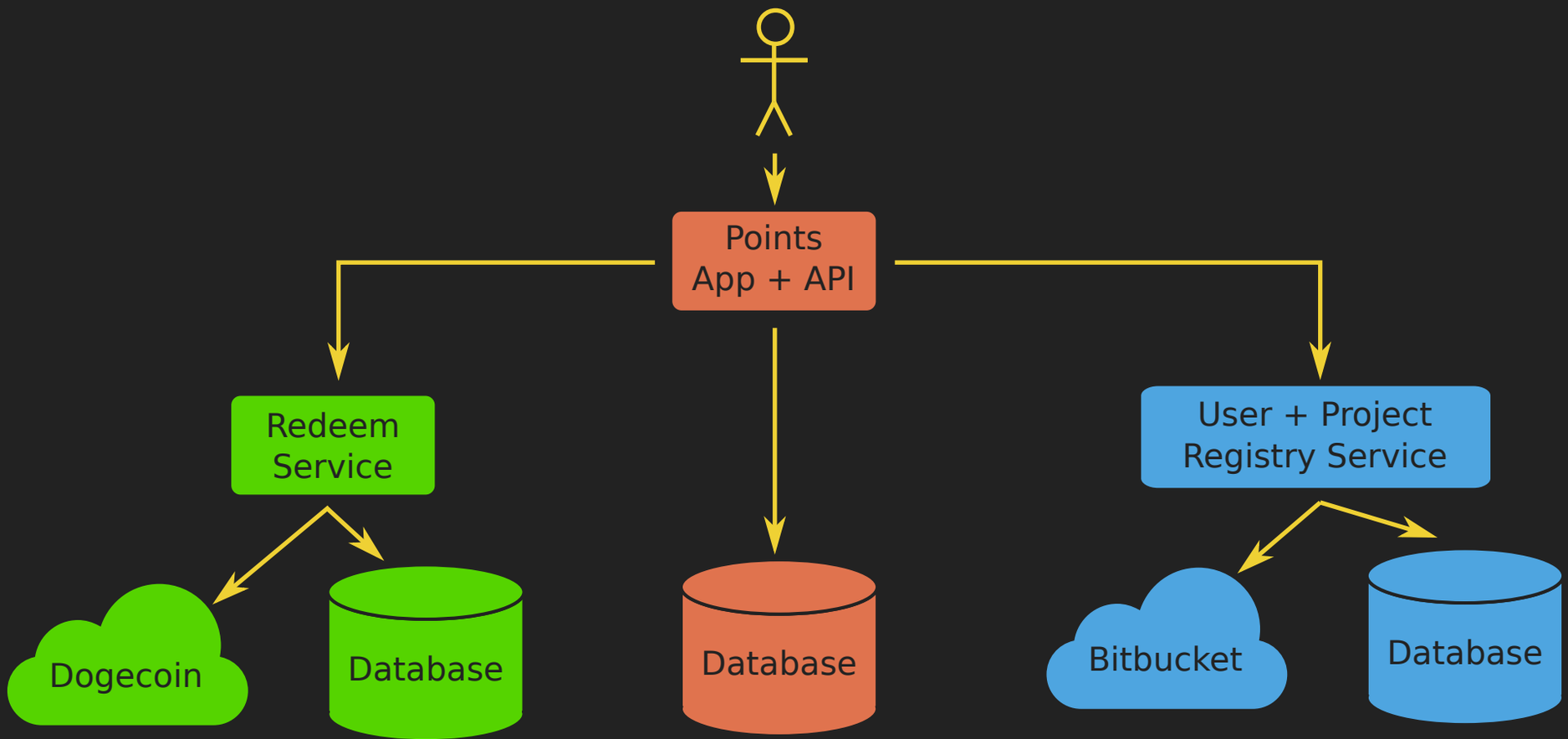
Benefits:

- ✔ Smaller less complex codebases.
- ✔ Enable independence between codebases & teams.
- ✔ More flexible scaling schemes (tech & organizational).

Drawbacks:

- ✖ Distributed codebases harder to infer, and may contain implicit inter-service dependencies.
- ✖ More complex orchestration, monitoring & provisioning.

EXAMPLE ARCHITECTURE



WHY USE DOCKER?

 Containers vs. Virtual machines

- Containers lighter in memory and processing than VMs.
  - Isolated user-space instances vs. machine emulation.
- Docker uses cached/immutable layered file systems.

 Tooling for Managing Containers

- Quick spin up of container/environments.
- Easily create, share and publish images to registries.
- Unified workflow that replaces other tools:
  - e.g. chroot jails, LXC, Vagrant, etc.

DEVELOPMENT AND TESTING

DOCKER COMPOSE

Specify with docker-compose.yaml...

```
points_app:
    build: .
    ports:
        - "5000:5000"
    environment:
        - API_KEY=MY_SUPER_SECRET_KEY
    hostname: app
    links:
        "couchdb:couch"
couchdb:
    image: couchdb
    ports:
        - "5984:5984"
    volumes:
        - data:/usr/local/var/lib/couchdb
other_services: ...
```

...and start up with:

```
docker-compose up
```

DOCKER WORKFLOW

- Docker + Compose replaces a Vagrant + VM workflow
- vagrant up + vagrant ssh + run $app_command ⟶ docker run $app_command
- vagrant halt ⟶ docker stop
- vagrant status ⟶ docker ps
- vagrant provision ⟶ docker build
- vagrant destroy ⟶ docker stop + docker rm
- vagrant box list, remove ⟶ docker images, docker rmi

BUILDING GOOD DOCKER IMAGES

Sample Dockerfile

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y python python-dev gcc \
    python-pip python-setuptools
ADD wsgi_app /app
WORKDIR /app
RUN pip install -r requirements.txt && pip install uwsgi
CMD uwsgi --http :5000 --master --processes 4 --wsgi-file app_wsgi.py
# CMD python app_wsgi.py
EXPOSE 5000
```

- Each step in a Dockerfile can create a new layer in filesystem.
  - Minimize steps number of separate RUN steps.
- Try to make layers cacheable:
  - Cached layer reused if no checksum change in source.
- Use base images for heavily repeated steps.
  - See ONBUILD command for making dynamic base images.
- Expose ports and volumes to document image.

PYTHON AND WSGI APPS

Web Servers

- Don't run a web server on your container. Use an external proxy or container instead.
- Just run WSGI apps using a WSGI app server:
  - uWSGI
  - Gunicorn

Virtualenvs

- Don't use virtualenvs inside Docker containers!
- Install directly into the system Python site packages.

DEBUGGING CONTAINERS

Want a minimal image, so no SSH daemon...
...so how do we debug a running container?

- Run Bash (or other command) on a Running Service

```
docker-compose exec $SERVICE_NAME /bin/bash
```

- Inspecting a Service's Logs (Standard Out & Error)

```
docker-compose logs $SERVICE_NAME
```

- Inspecting a Running Container's Setup

```
docker inspect $CONTAINER_ID
           > ...
```

```
docker inspect --format '{{json .Config.ExposedPorts }}' \
        $CONTAINER_ID
        > {"5000/tcp": {}}
```

PERSISTANCE, CONFIGS & PROCESSES

Volume Maps

- Changes to container lost after container destroyed.
- Volume maps to external host folder for persistence.
- Another pattern is using separate Docker data containers.

Configuration

- Prefer using environment variables for configuration.
- Volume mapped configs maybe a warning sign of a overly complex setup or a config in need of refactoring.

Managing Processes

- Use supervisord or runit to control multiple processes.
- Consider refactoring containers to not need that.

TESTING + TOOLING

Testing

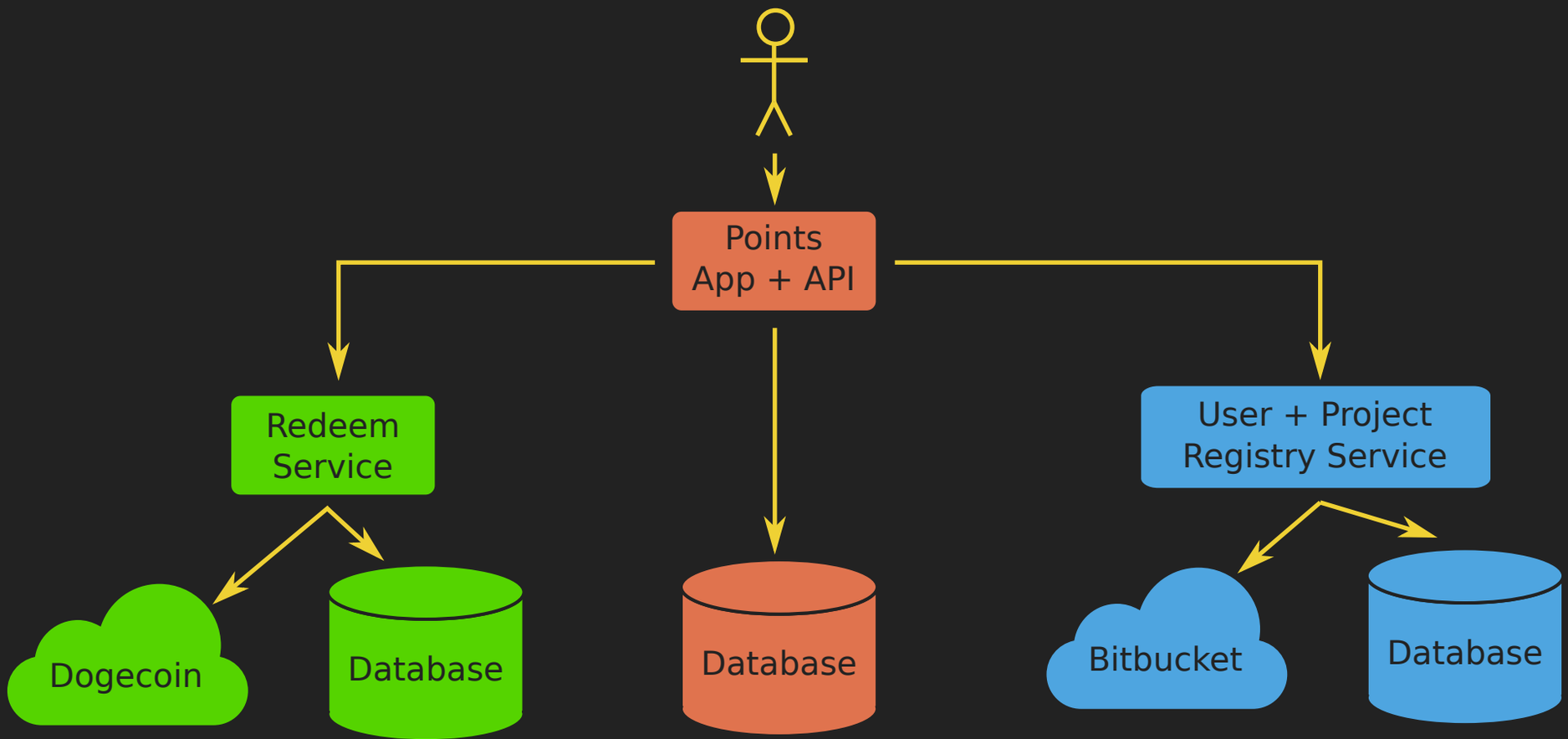
- Docker adds consistency in your CI environments!
- Simple setup for a Docker host.
- Control over what is in container = Repeatable workflow and simpler test environment.
- Cloud-based CI options with Docker support out there.

Tooling

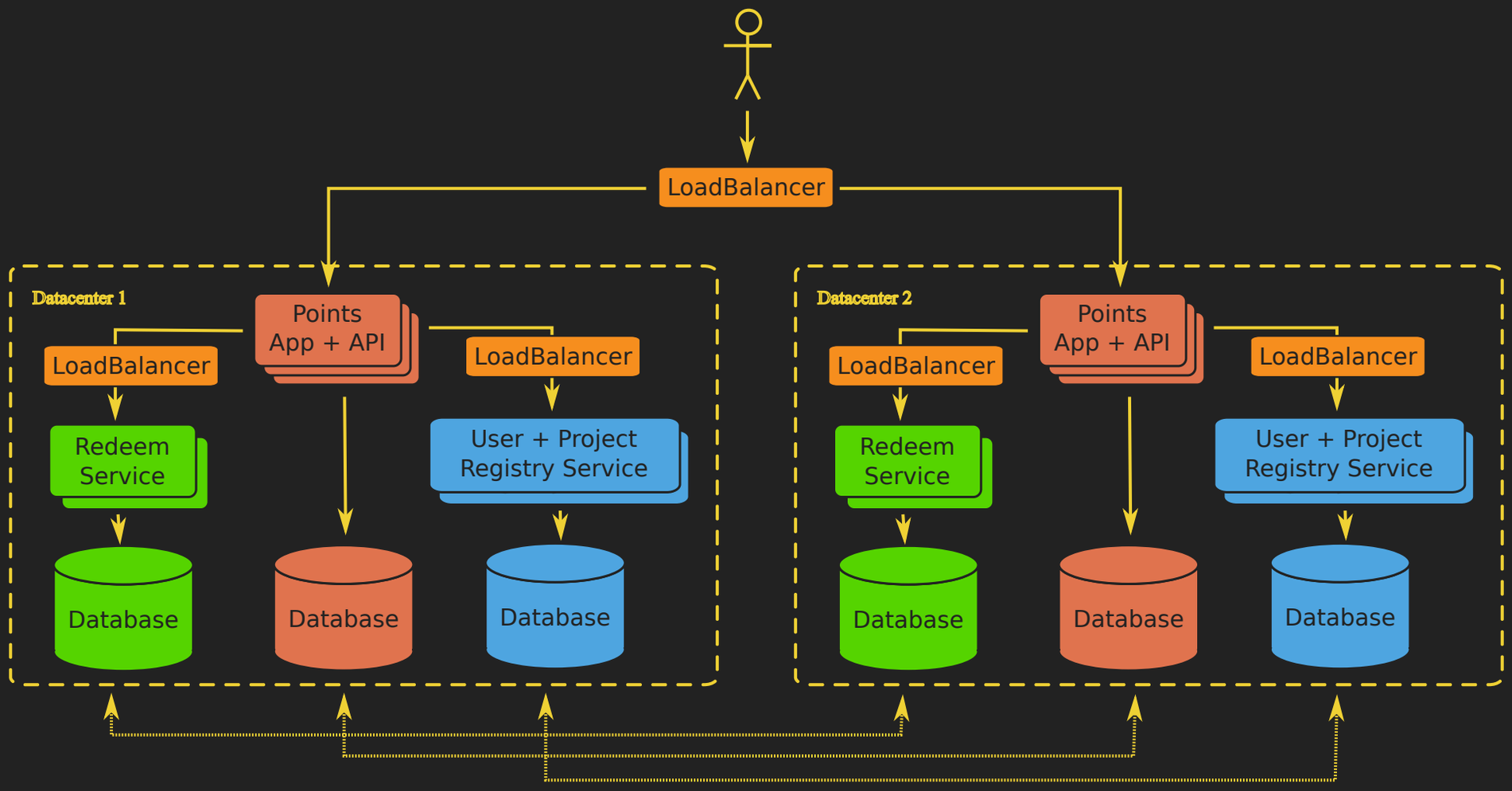
- Docker tool defaults, options, and internal API can radically from version to version.
- Don't build your own tooling! If you can avoid it...
- docker-py: a Python client library for working with Docker*

DEPLOYMENT AND SCALING

EXAMPLE ARCHITECTURE



EXAMPLE PROD ENVIRONMENT



SETTING UP A CLOUD

Looks like you're trying to build a cloud of microservices...

- Load Balancing + Network Topology:
  e.g. HAProxy & Nginx, etc.
- Provisioning:
  Automated, repeatable setup for non-Docker systems.
  e.g. Ansible, Puppet & Salt.
- Monitoring:
  Look at app health, app behaviour & system resources.
  e.g. Nagios, Pingdom & New Relic.
- Logging:
  Aggregate various logs and correlate events.
  e.g. Splunk.

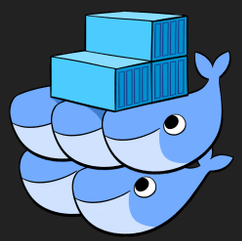
CLOUD INFRASTRUCTURE

Managing cloud infrastructure is hard!

Need tooling and automation for all that stuff.

Don't build your own tool unless you want to support it to end of time. *(Unless you're a cloud tech vendor.)*

Consider using one of these instead:



Docker Swarm

Kubernetes

OpenStack Magnum

CoreOS Fleet

LESSONS LEARNED

LESSONS LEARNED

- Microservices and Docker can improve building and deploying complex systems. But neither is a cure-all.
- Good development & deployment processes matter. Docker has a decent workflow to help shape those processes.
- Expect lots of additional infrastructure around microservices.
- Avoid building your own tooling.
- Use Docker containers to do effective isolation.
- Good app design goes a long way.

RESOURCES

- Jared Kerim's Django Docker template:
  - https://github.com/jaredkerim/django-docker-compose
- 12 Factor apps:
  - http://12factor.net/
- Rookeries - Dockerized Workflow Example:
  - https://bitbucket.org/dorianpula/rookeries/ (docker_compose_workflow branch)

THANK YOU!

Twitter - @dorianpula

WWW - http://dorianpula.ca/

ANY QUESTIONS?

GO FORTH AND BUILD AWESOME STUFF!!!