node.js on a PaaS

the awesome and the wonky

Patrick Mueller - **`@pmuellr`** - **`muellerware.org`**

developer advocate for IBM's Bluemix PaaS

http://pmuellr.github.io/slides/2014/10-node-js-on-a-paas
http://pmuellr.github.io/slides/ (all slides)

wat - node.js on a PaaS

- you know what node.js is
- PaaS == Platform as a Service
- examples:
    - Heroku
    - Nodejitsu
    - Cloud Foundry
    - OpenShift
    - many more!

PaaS fundamentals

- OS provided for you (Linux)
- you provide the application (web server)
- configure:
  - number of instances running
  - RAM per instance
  - ephemeral disk per instance
- run a few commands

voila!

web server running on the "cloud" (public internet)

PaaS usage scenarios

- special focus on web servers, typically with:
  - one open HTTP port open when app starts
  - HTTPS support
  - free domain, or use your own custom domain
  - WebSocket support
- or anything - arbitrary compute

let's deploy an app to Cloud Foundry

```
$ git clone https://github.com/pmuellr/cf-node-hello.git
$ cd cf-node-hello
$ cf push
Using manifest file .../manifest.yml
...
Installing IBM SDK for Node.js from admin cache
...
Installing dependencies
...
Uploading droplet (8.0M)
...
App started
...
urls: cf-node-hello-pjm.mybluemix.net
$ curl https://cf-node-hello-pjm.mybluemix.net
Hello World
```

what just happened?

- **`cf push`** uploaded your application files to a staging server
- staging server got node binaries, package dependencies, packaged into archive
- vm allocated to run the app, archive downloaded, expanded, started
- web server now running on the internet

how the staging server "builds" an app

- driven from **`package.json`**
- get node executable from **`engines.node`**
  - **`{ "engines" : { "node" : "0.10.x" } }`**
- **`npm install`** will be run to obtain packages
- for Heroku and Cloud Foundry, the build is scripted with a "buildpack"; you can also write your own

PaaS app development methodology

The Twelve Factor App - http://12factor.net/

patterns for building apps on the cloud

1. Codebase
2. Dependencies
3. Config
4. Backing Services
5. Build, release, run
6. Processes
7. Port binding
8. Concurrency
9. Disposability
10. Dev/prod parity
11. Logs
12. Admin processes

PaaS app development methodology

- **Codebase** - One codebase tracked in revision control, many deploys
- **Dependencies** - Explicitly declare and isolate dependencies
- **Config** - Store config in the environment
- **Backing Services** - Treat backing services as attached resources
- **Build, release, run** - Strictly separate build and run stages
- **Processes** - Execute the app as one or more stateless processes

PaaS app development methodology

- **Port binding** - Export services via port binding
- **Concurrency** - Scale out via the process model
- **Disposability** - Maximize robustness with fast startup and graceful shutdown
- **Dev/prod parity** - Keep development, staging, and production as similar as possible
- **Logs** - Treat logs as event streams
- **Admin processes** - Run admin/management tasks as one-off processes

PaaS tools

- web dashboard
- command-line tooling
    - Heroku toolbelt - `heroku`
    - Cloud Foundry - `cf`
- typically will be using both web and cli

domains

- typically PaaS provides you a free domain for your apps:
    - `foo.herokuapp.com`
    - `bar.mybluemix.net`
    - `yow.cfapps.io`
- host names must be unique across free domain!
- custom domains usually supported

https support

- most PaaS's provide SSL termination
- allows you to support http and https traffic with just an http server
- or do you want https all the way to your server?
- https support for custom domains not simple; upload certs, etc

using hosted services

Instead of running your own database, queueing server, etc, you'll be using hosted services, like:

- MongoLab
- Cloudant
- Redis Cloud

Some PaaS's co-locate hosted services in same datacenter to reduce latency.

using hosted services

- add service to your app via:
  - command-line tool
  - web dashboard
  - roll your own access any 3rd service however you can
- services exposed to app via environment variables
  - Heroku's **`MONGOLAB_URL`** env var
  - inside of Cloud Foundry's **`VCAP_SERVICES`** env var

using hosted services

Heroku:

```
$ heroku addons:add mongolab
```

Cloud Foundry:

```
$ cf create-service mongolab sandbox my-mongo-db
```

adapting your app

- configuration provided via environment variables
- **`process.env`** is your new best friend
- **`process.env.PORT`** - env var set to the port to bind server
- using Heroku's MongoLab add-on service:

```
process.env.MONGOLAB_URL // MongoLab db URL
   // mongodb://[user]:[pass]@[host]:[port]/[path]
```

adapting your app - Cloud Foundry

- **`VCAP_SERVICES`** - JSON string containing structured service info
- **`VCAP_APPLICATION`** - JSON string containing other environmental info
    - url(s) to server
    - ip address of server
    - start time
    - etc

adapting your app - Cloud Foundry

VCAP_SERVICES will look like this, but with even more data:

```
{
  "mongodb-2.2": [
    {
      "name": "mongodb",
      "label": "mongodb-2.2",
      "credentials": {
          "url": "mongodb://..."
      }
    }
  ]
}
```

adapting your app - Cloud Foundry

use the cfenv package to programmatically introspect over **VCAP_SERVICES** and **VCAP_APPLICATION**

instead of:

```
services = JSON.parse(process.env.VCAP_SERVICES)
mongoURL = services["mongodb-2.2"][0].credentials.url
```

use this:

```
cfenv = require("cfenv")
appEnv = cfenv.getAppEnv()
mongoURL = appEnv.getServiceURL("mongodb")
```

scaling

By default, PaaS will run one instance of your app. Want more?

Heroku:

```
$ heroku ps:scale web=42
```

Cloud Foundry:

```
$ cf scale my-app -i 42
```

scaling

if you want to scale, servers must be stateless

- no caching mutable data
- sometimes you want to scale **down**, so be prepared for servers to end
    - no long running, non-atomic transactions

the awesome: summary

- no special "cloud libraries" **required** for your app
- quick deploy of applications to the cloud
- don't need to worry about installing operating systems
- don't need to worry about installing services
- easy to scale up/down

the wonky

and how to de-wonk-ify

core issues

- can't configure base operating system
- often no ssh
- stdout/err via syslog
- ephemeral file system
  - **=>** database all the things

debugging

node-inspector difficult to run

- wants two ports open (app and debug)
- PaaS typically only allows one

=>

use a proxy splitter

- run two servers, split traffic with proxy by URL
- cf-node-debug

other diagnostic tools

- remember: no **`ssh`** to run diagnostic tools

=>

- New Relic
  - get account, apply light config
  - instrument via **`require("newrelic")`**
- StrongLoop
  - requires StrongLoop's tooling
- PaaS-specific
  - Bluemix Monitoring and Analytics

diagnosing startup problems

- app runs fine on your laptop
- fails when run on PaaS

=>

- most errors here are in startup
  - initializing databases, services, etc
- at startup:
  - add LOTS of logging
  - add LOTS of error checking

private packages

if the PaaS runs **`npm install`** for you, how do can you access private packages

`=>`

- manage them separately from the rest of your packages
    - see pmuellr/bluemix-private-packages for an example project structure
- arrange to use a private package manager (npm in the future)

logging

- typically only get last XX lines of your stdout/err
- but easy hook-ups to logging services:
    - Loggly
    - PaperTrail
    - splunk>storm
    - SumoLogic
    - Logentries
    - roll your own syslog support

dependency versions

```
{
  "name":            "my-awesome-app",
  "dependencies": {
    "express":     "*"
    }
}
```

What's wrong with this **`package.json`** file?

dependency versions

```
{
  "name":           "my-awesome-app",
  "dependencies": {
    "express":      "*"
    }
}
```

Guess what happened the day express 4.0 was released?

- removed from express:
  - properties on **`express`**, **`req`**, **`res`**
  - all bundled middleware except static

dependency versions

Lesson: lock down your dependencies

do one of these:

- version your node_modules with your app
- use fixed major/minor version specs: eg, **`"3.4.x"`**
- use npm shrinkwrap

fin