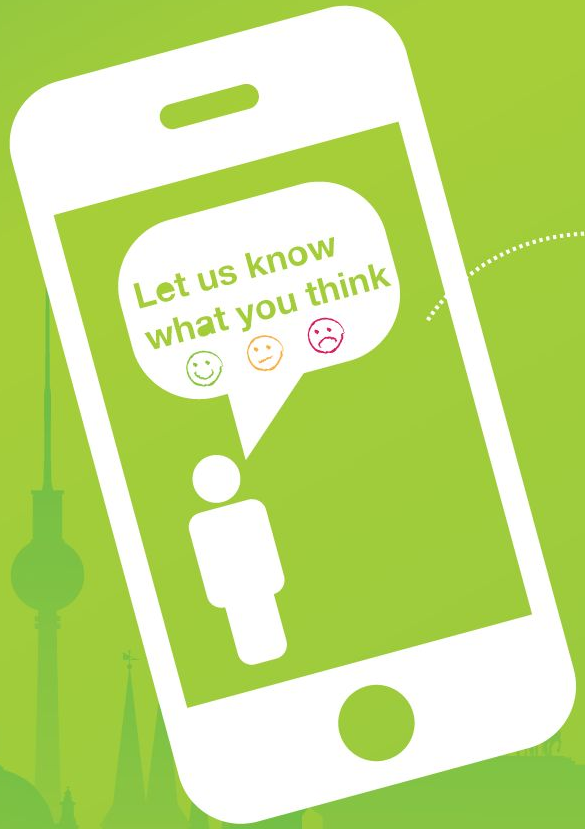
Container Patterns

Matthias Lübken

Giant Swarm

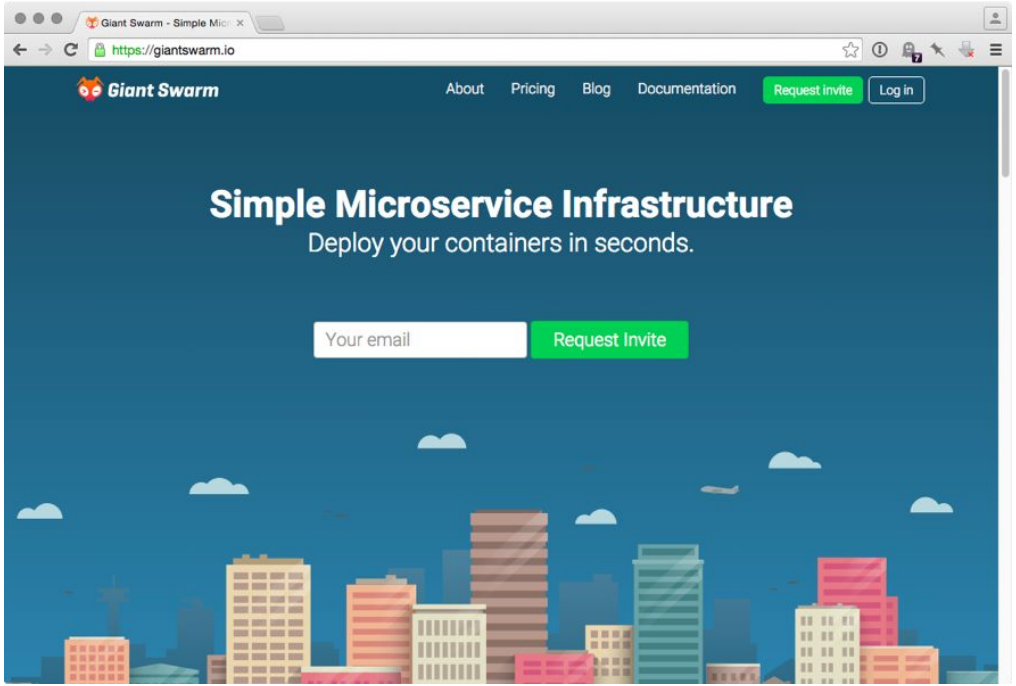
goto; conference



Click ‘engage’
to rate sessions
and ask questions
plus give feedback


Giant Swarm

GiantSwarm.io

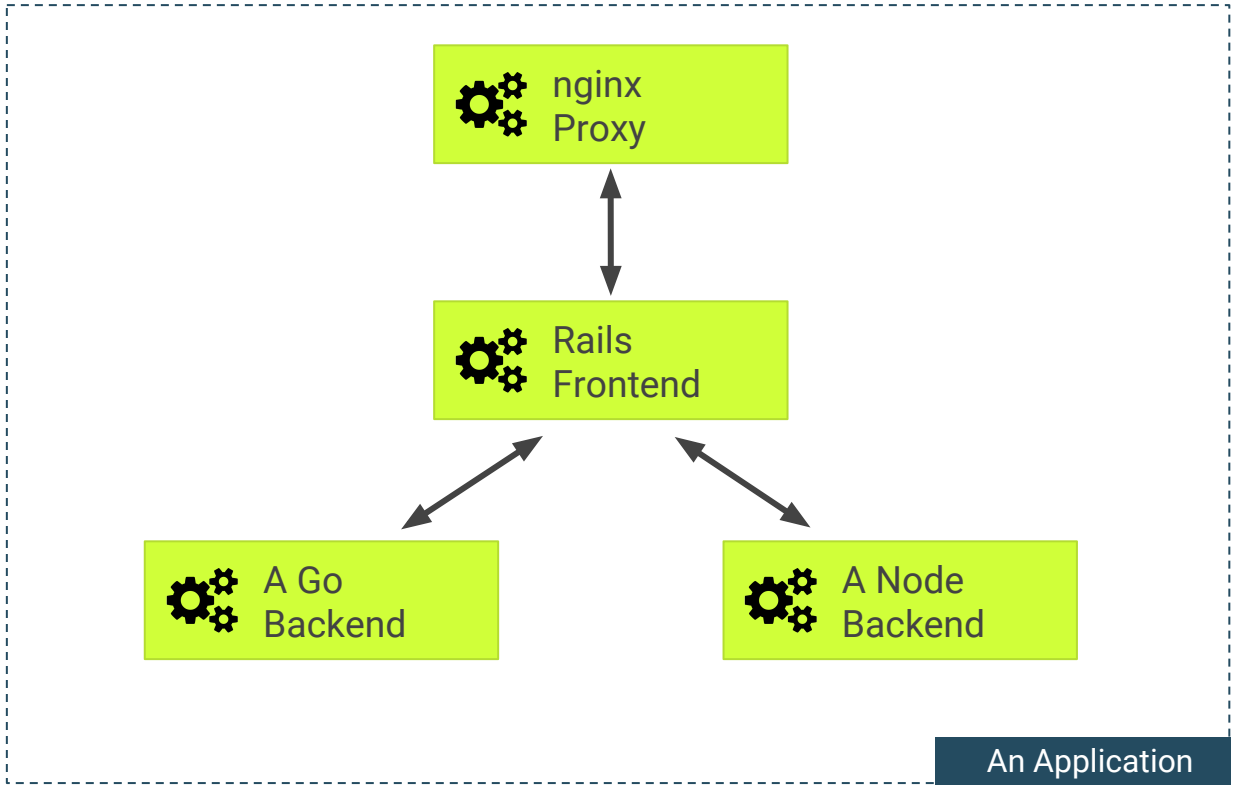


Simple Microservice Infrastructure build for developers.

Deploy your containers in seconds.

Scaling with your needs: Public, Private, On-Prem

"Docker is an open-source project to easily create lightweight, portable, self-sufficient containers from any application."


nginx
Proxy
Rails
Frontend
A Go
Backend
A Node
Backend
An Application

“Use multiple containers to modularize your application.”

Some reasons

- Independently releasable
- Separate processing types
- Different loads
- Different teams
- Reuse of containers
- Crash isolation
- Different release cycles
- Use different languages / versions / libraries

Container Patterns?

- Are there **general applicable** patterns?
- How would we **describe** them?
- What are **concrete** examples and best-practices

- Context:
  - Cloud cluster applications
  - They should be container runtime agnostic

Related work

- 12-Factor apps
- Cloud-native application architectures
- Microservices
- Continuous Delivery

Outline

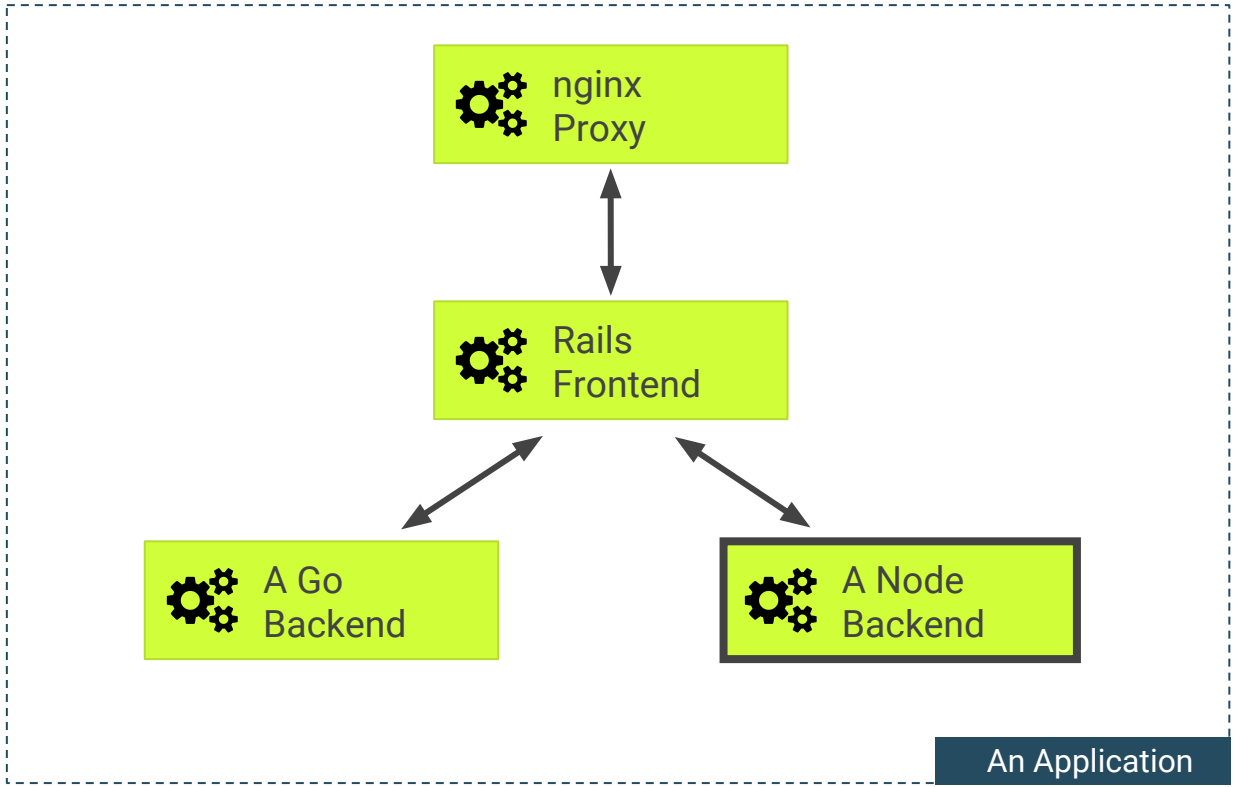
Building blocks

- Modular container
- Pods

Composite patterns

- Sidecar
- Ambassador
- Adapter
- Chains

Modular container


nginx
Proxy
Rails
Frontend
A Go
Backend
A Node
Backend
An Application

Modular Container

We define a modular container as the collection of these 6 properties:

1. Proper Linux process
2. Explicit interfaces
3. Disposable
4. Immutable
5. Self-contained
6. Small

1. Proper Linux Process

Containers should behave as a proper Linux process and be nice to their init process.

- React to signals
- Return proper exit codes
- Use standard streams

Best practices (Proper Linux Process)

- React to signals:
  - React on e.g. SIGINT, SIGTERM, etc.
  - Don’t daemonize your processes
  - Make your process foreground (e.g. use exec)

- Return proper exit codes:
  - 0 (OK), 1 (General error) …

- Use stdin, stdout, stderr:
  - Log to stdout. Don’t concern with routing and storage

2. Explicit interfaces

Dependencies to other containers should be made explicit by defining its interfaces.

- CLI arguments
- Environment variables
- Network / Port
- Document via labels

Best practices (Explicit interfaces)

- CLI arguments
  - Use a lib for parsing / validating
- Environment variables
  - Set defaults in the image
  - Overwrite with `docker -e`
- Network / Ports
  - Expose port via EXPOSE in Dockerfile
- Document via labels
  - E.g. LABEL INSTALL="docker run ...

3. Disposable Containers

Containers should be treated as disposable artefacts. The application shouldn’t rely on a particular container instance to be running.

Pets vs. Cattle:
Treat your container as part of a cattle. You number them and when get sick you shoot them.

Best practices (Disposable Containers)

- Only keep ephemeral state
  - Don’t assume this state between two requests
- Robust against sudden death
  - If the container gets interrupted pass on your current job.
- Minimal setup
  - If more setup needed let the scheduler know

4. Immutable

Once a container image is build it shouldn’t be changed. State should be extracted and changes to the container should be applied by rebuilding.

Best practices (Immutable)

- Strive for dev / prod parity
- Extract runtime state in volumes
- Anti-pattern: **`docker exec`**

5. Self-contained

The container should only rely on the Linux kernel. All other dependencies should be made explicit and added dynamically.

Best practices (Self-contained)

- Add dependencies at build time
  - Build Uber-Jar and include webserver
- Strive for zero-config deployment
- Generate dynamic config files on the fly
- Anti-Patterns:
  - Put config into a volume
  - Put code into a volume *

6. Small

A container should have the least amount of code possible to fulfill its job.

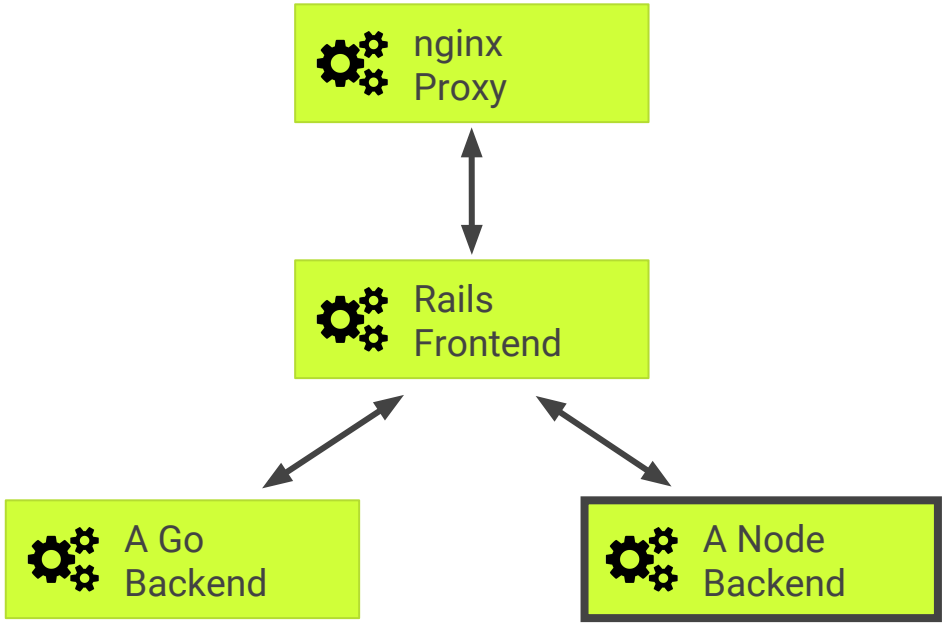
Best practices (Small)

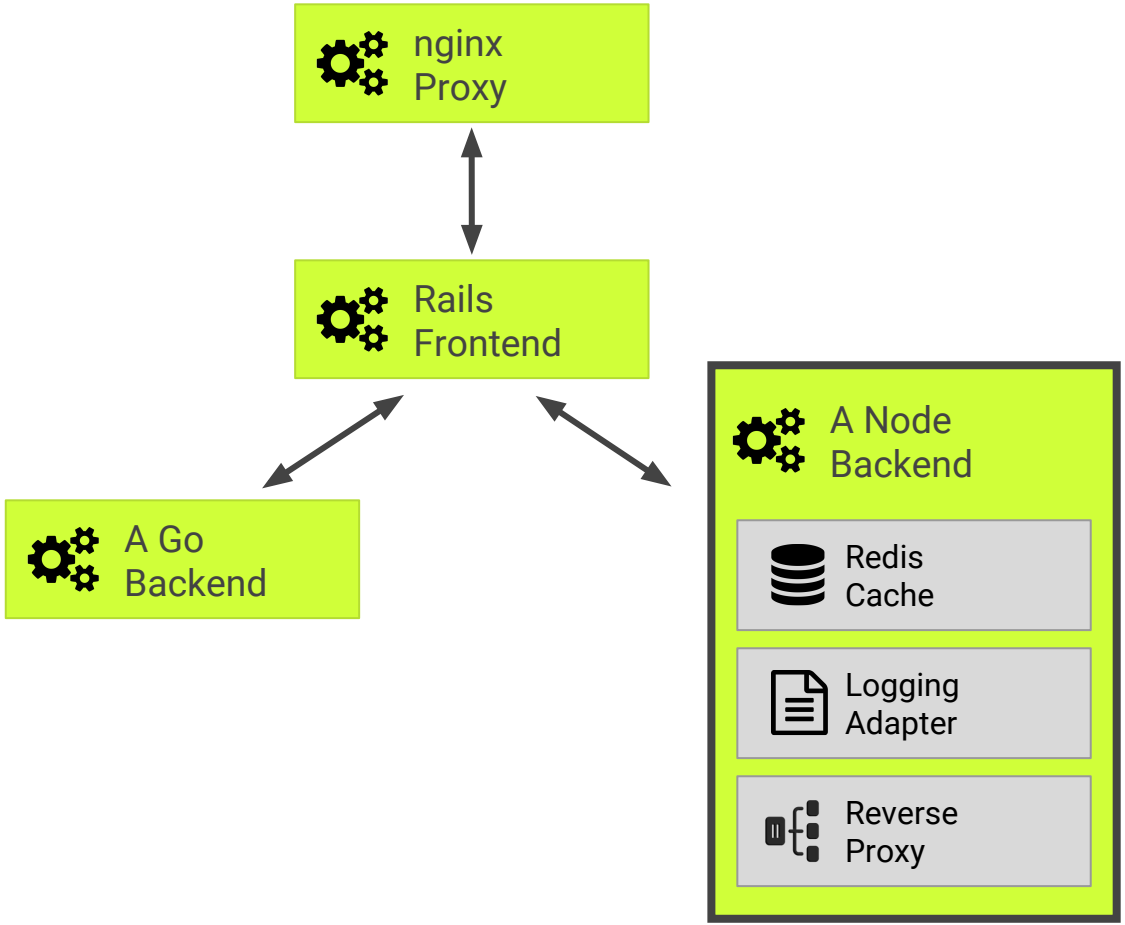
- Build from scratch
- Use small base-image
  - **`busybox`**, **`alpine`**
- Reuse custom base image
- Anti-Pattern: VM Container

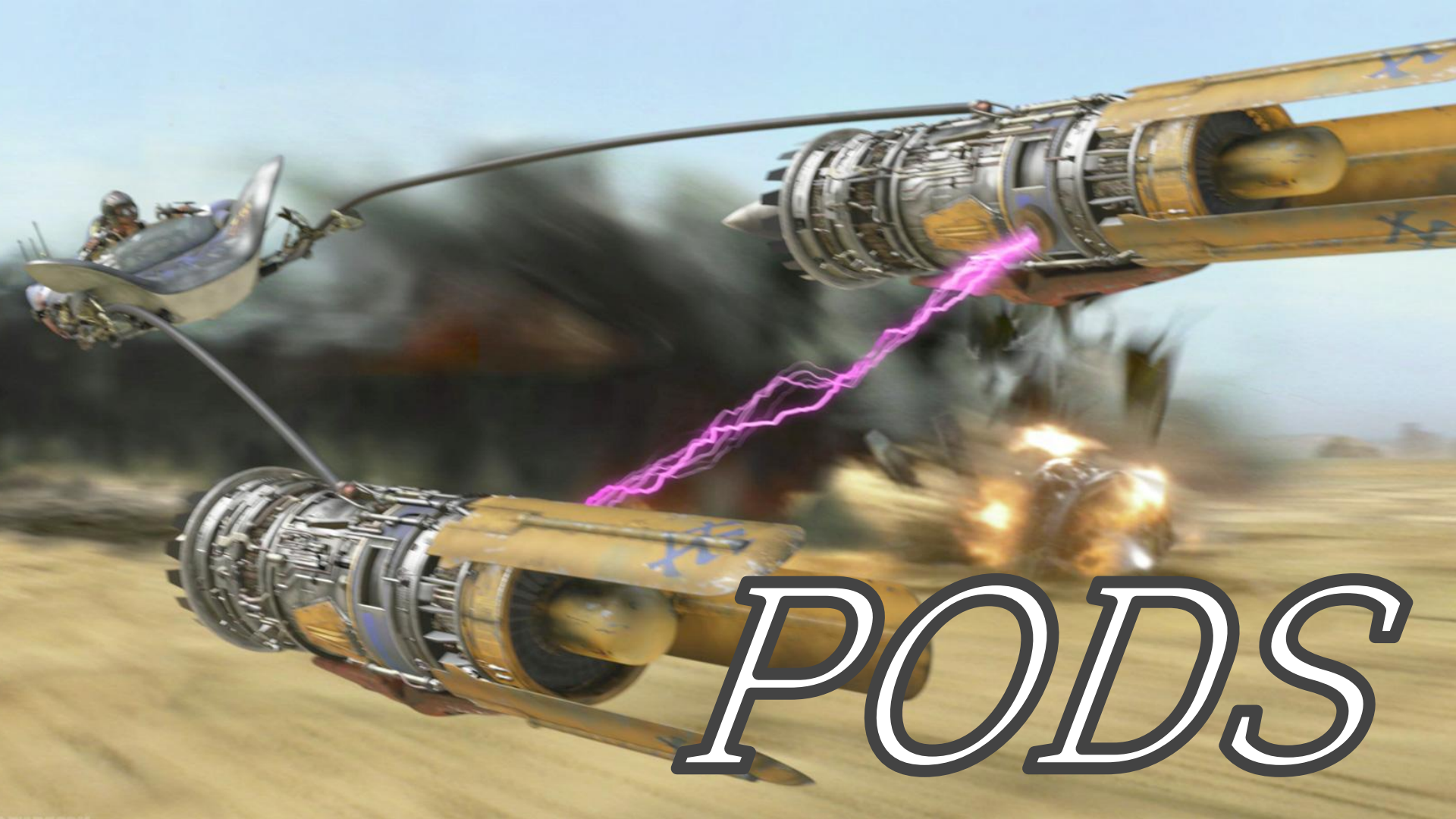
Recap: Modular Container

We define a modular container as the collection of these 6 properties:

1. Proper Linux process
2. Explicit interfaces
3. Disposable
4. Immutable
5. Self-contained
6. Small


nginx
Proxy
Rails
Frontend
A Go
Backend
A Node
Backend


nginx
Proxy
Rails
Frontend
A Go
Backend
A Node
Backend
Redis
Cache
Logging
Adapter
Reverse
Proxy


PODS

Pods

- Group closely related containers
- A **single deployable unit**
- **Share** all available **namespaces**
- The pod as a whole and the individual containers can be limited

Share namespace

- Sharing the same **network** namespace and access to the same IP and port namespace
- Sharing the **IPC** namespace for communicating e.g. Unix sockets, shared memory, system message queues
- Share the same hostname via the **UTS** namespace
- Share the **PID** namespace and can see each others processes (not supported by docker)
- Sharing the same **volumes**

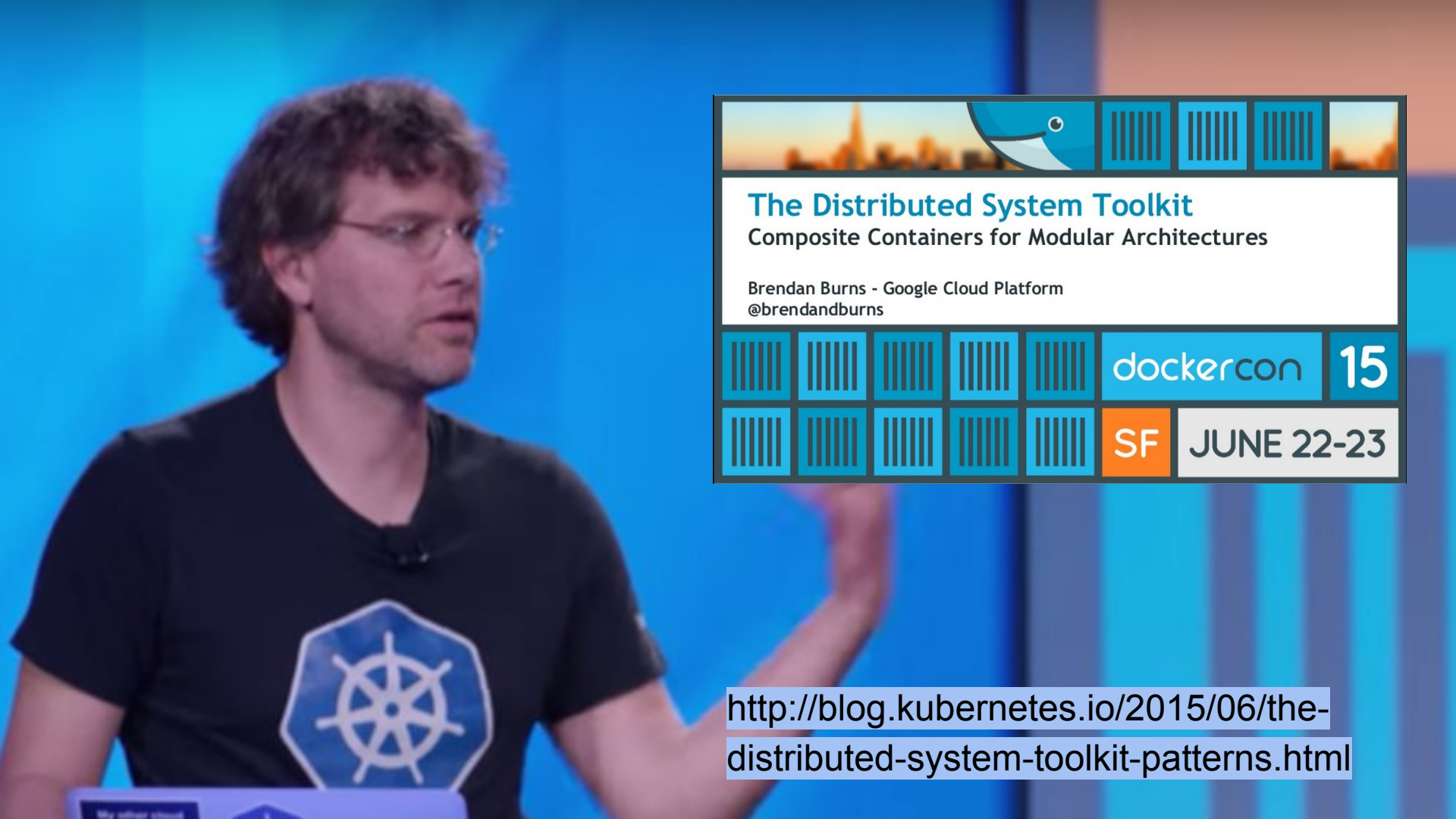
Outline

Building blocks

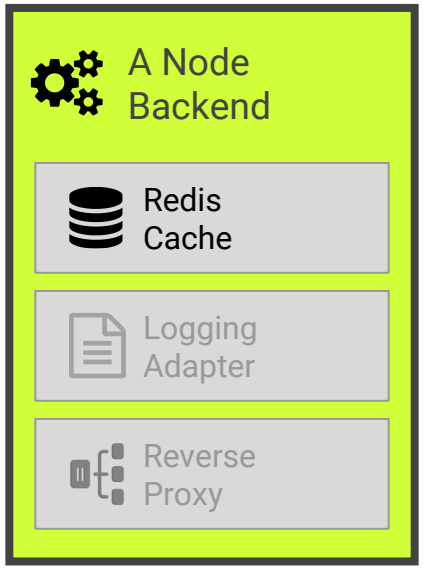
- Modular container
- Pods

Composite patterns

- Sidecar
- Ambassador
- Adapter
- Chains

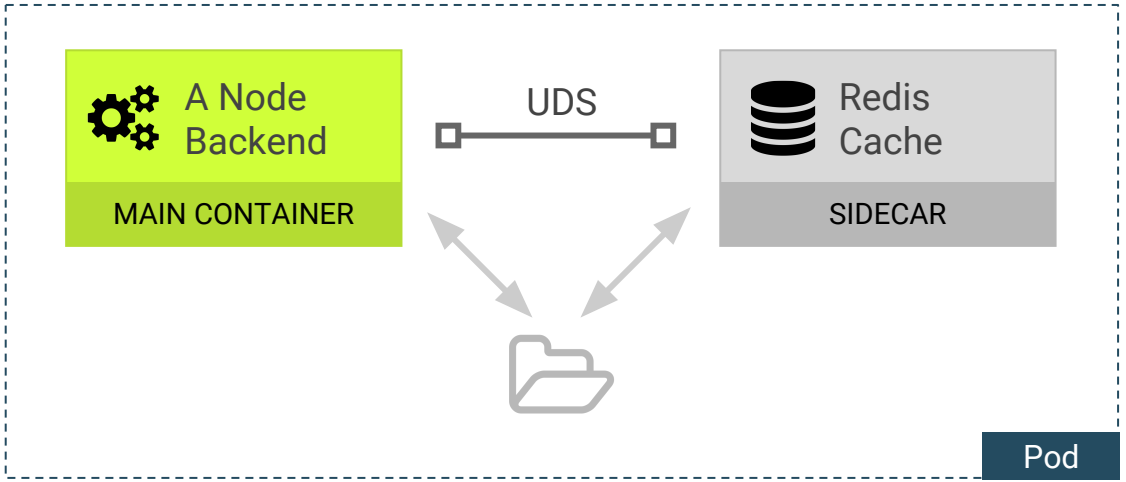


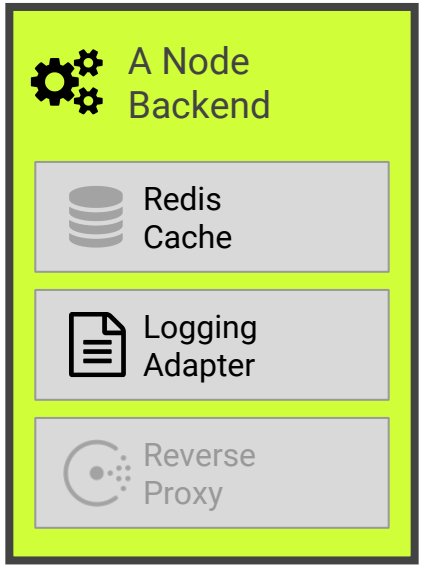
http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html


A Node
Backend
Redis
Cache
Logging
Adapter
Reverse
Proxy

Pattern: Sidecar / Sidekick

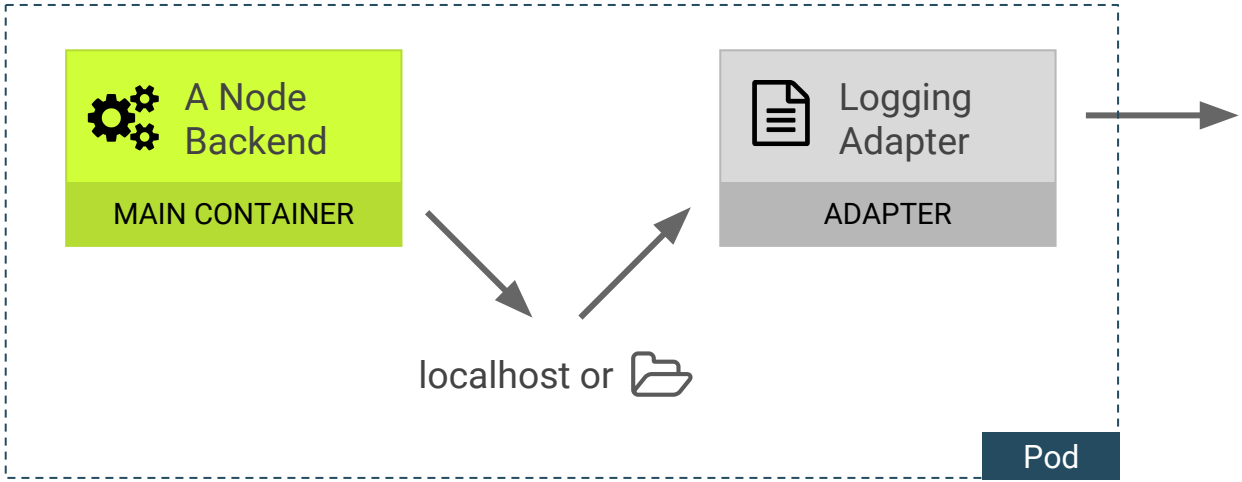
- Enhance & extend the main container.
- K8S: transparently. Netflix: platform features.

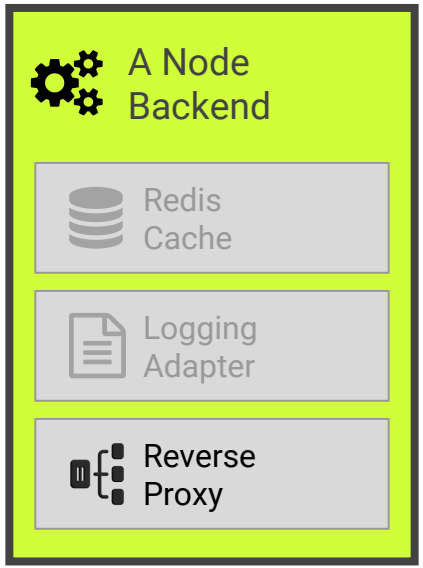



A Node
Backend
Redis
Cache
Logging
Adapter
Reverse
Proxy

Pattern: Adapter

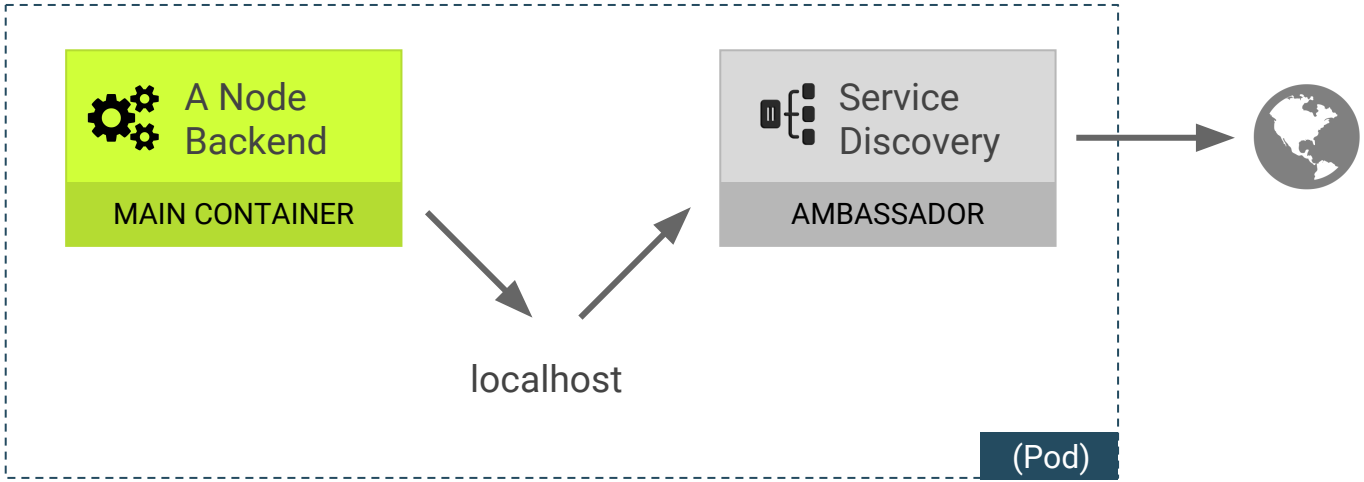
Standardise and normalize output. E.g. logging and metrics.



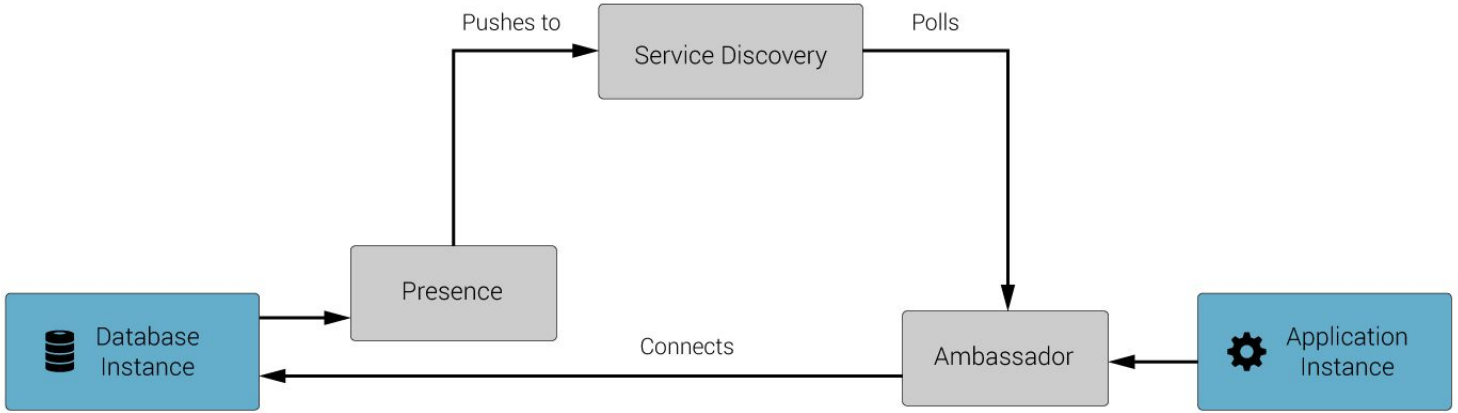

A Node
Backend
Redis
Cache
Logging
Adapter
Reverse
Proxy

Pattern: Ambassador

Proxy a local connection to the world: Service Discovery, Client Side LB, Circuit Breaker



Service Discovery



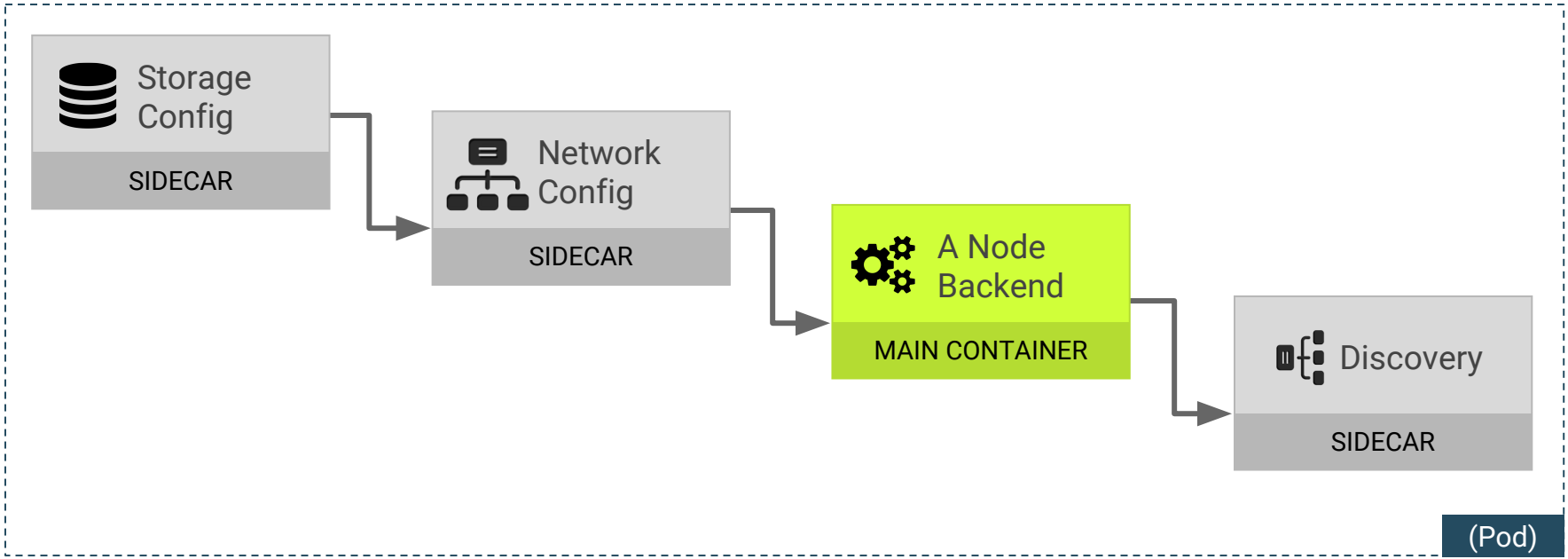
Giant Swarm

More info:

- https://docs.giantswarm.io/fundamentals/user-services/container-injection/
- https://docs.giantswarm.io/fundamentals/user-services/service-discovery/

Pattern: Container chains

Defined order of starting and stopping sidecar container.



Recap

Building blocks

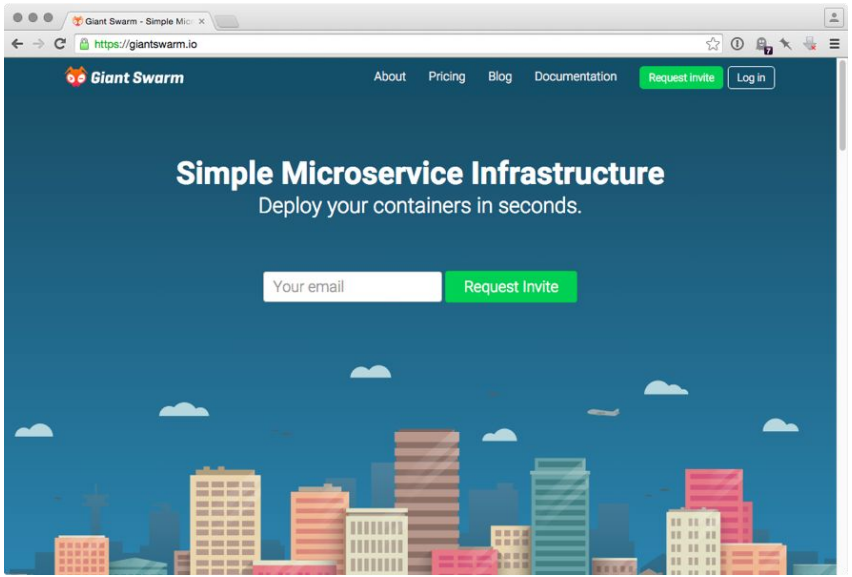
- Modular container
- Pods

Composite patterns

- Sidecar
- Ambassador
- Adapter
- Chains


I WANT MORE
MORE! MOAR!! MOOOAAARRR!!!
quickmeme.com

GiantSwarm.io



bit.ly/container-patterns

@luebken

Giant Swarm

Links / References

- http://blog.james-carr.org/2013/09/04/parameterized-docker-containers/
- https://docs.docker.com/articles/dockerfile_best-practices/
- http://tldp.org/LDP/abs/html/exitcodes.html (Exit Codes for “Proper Linux Process”)
- http://www.theregister.co.uk/2013/03/18/servers_pets_or_cattle_cern/ (Pets vs Cattle)
- http://www.projectatomic.io/docs/docker-image-author-guidance/ (Dockerfile)
- http://www.hokstad.com/docker/patterns (Dev patterns)
- http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html (Composite Patterns)
- http://static.googleusercontent.com/media/research.google.com/de//pubs/archive/43438.pdf (Borg by Google inspiration for Kubernetes / Pods)
- http://techblog.netflix.com/2014/11/prana-sidecar-for-your-netflix-paas.html (Sidecar Netflix)

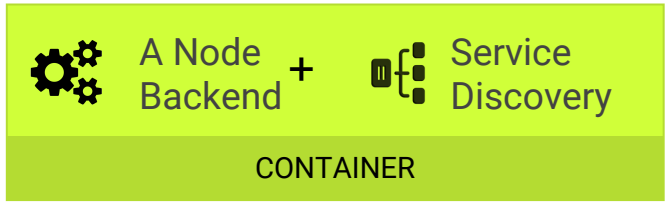
Credits

- https://www.flickr.com/photos/skynoir/8241460998 (Cover image)
- https://www.flickr.com/photos/tinker-tailor/8378048032/ (Help us image)

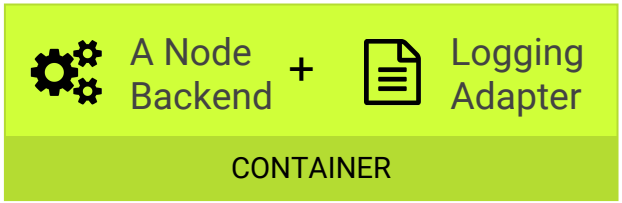
old slides

Fat Container

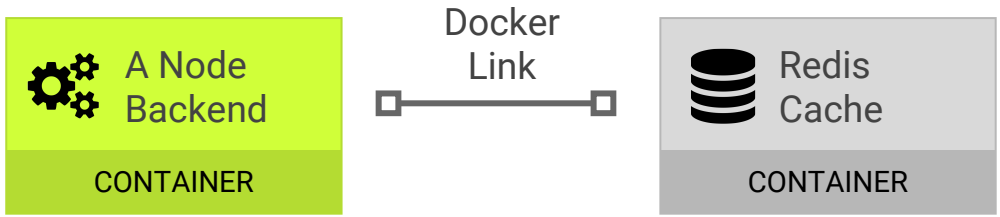
Fat Container



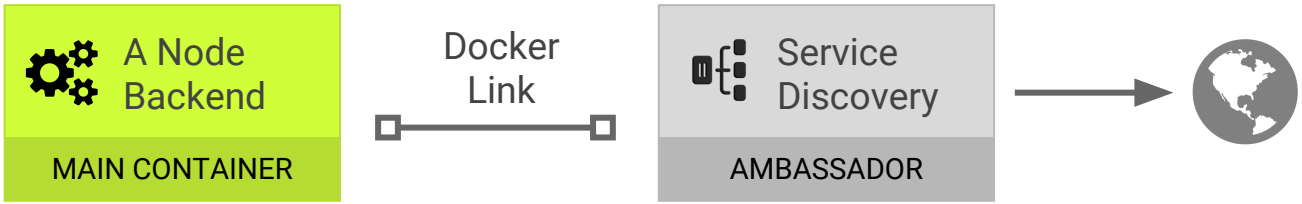
Fat Container



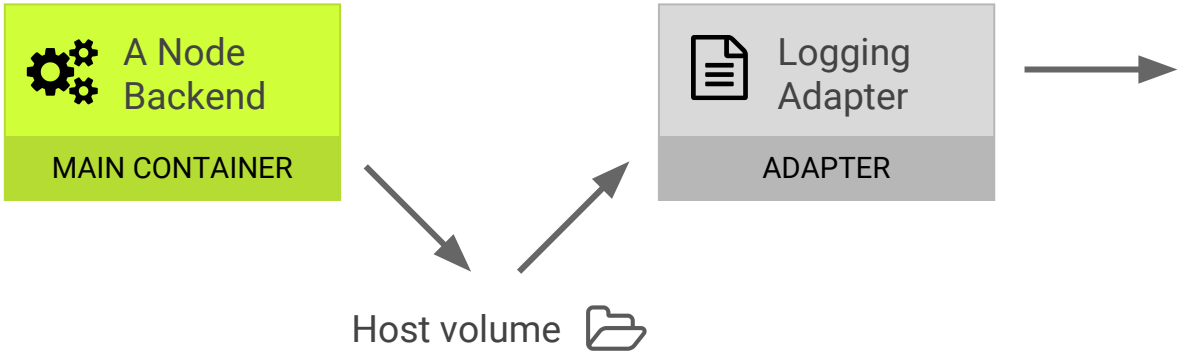
Linked Containers



Linked Container



Shared volume



NodeJS Example

```
server.listen(httpPort, httpAddress);

process.on('SIGTERM', function() {
    console.log("Received SIGTERM. Exiting.");
    server.close(function () {
        process.exit(0);
    });
});
```

https://github.com/giantswarm/giantswarm-firstapp-nodejs/blob/master/server.js

Pods Examples

- Redis cache via unix socket
- Monitoring adapters
- Cache init via named pipe

Best practices (2) (Explicit dependencies)

- Volumes
  -

Container Runtime (Explicit contracts)

- Start containers with --icc==false && --link:other-container