redhat

DOCKER PATTERNS

W-JAX 2017

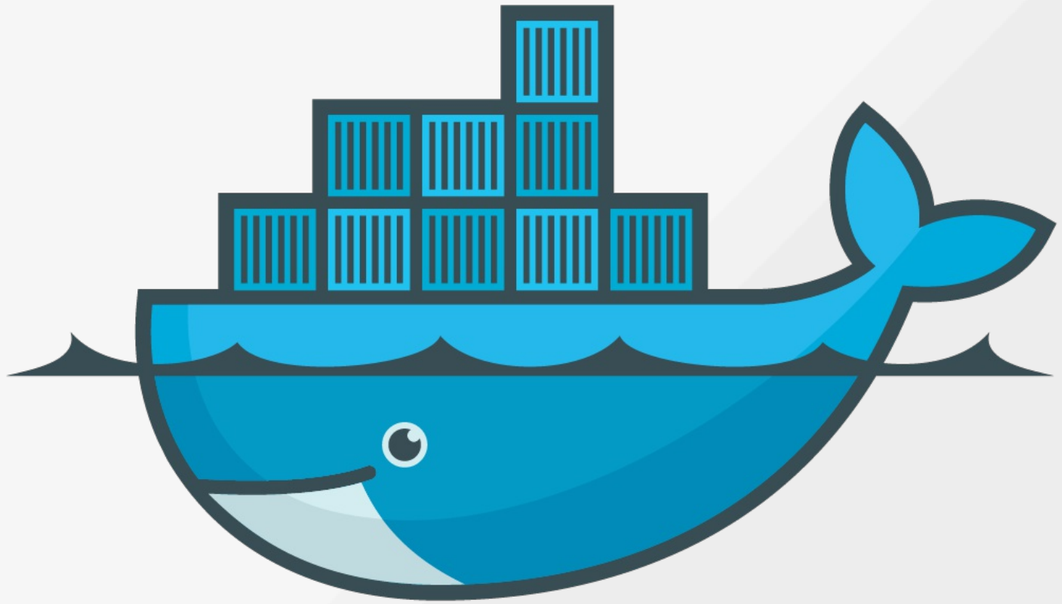
Dr. Roland Huß, Red Hat, @ro14nd

redhat

"**m**" for menu, "**?**" for other shortcuts

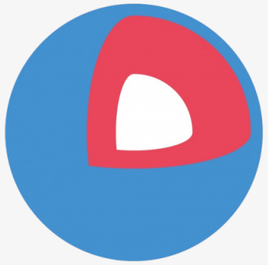
AGENDA

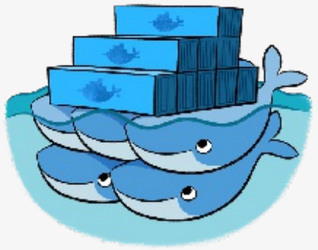
- Docker & Patterns
- Configuration
- Service Discovery
- Sidecar
- Image Builder

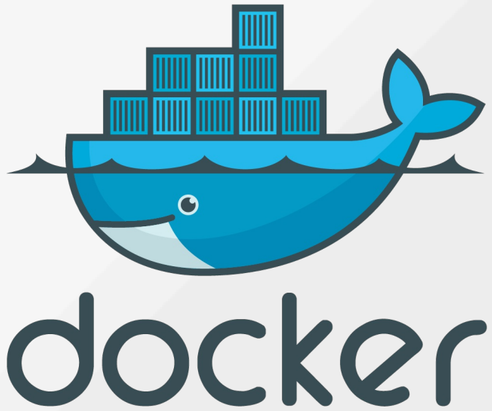


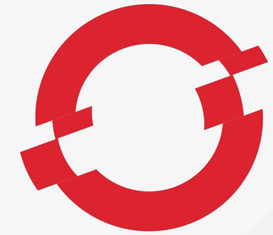
docker

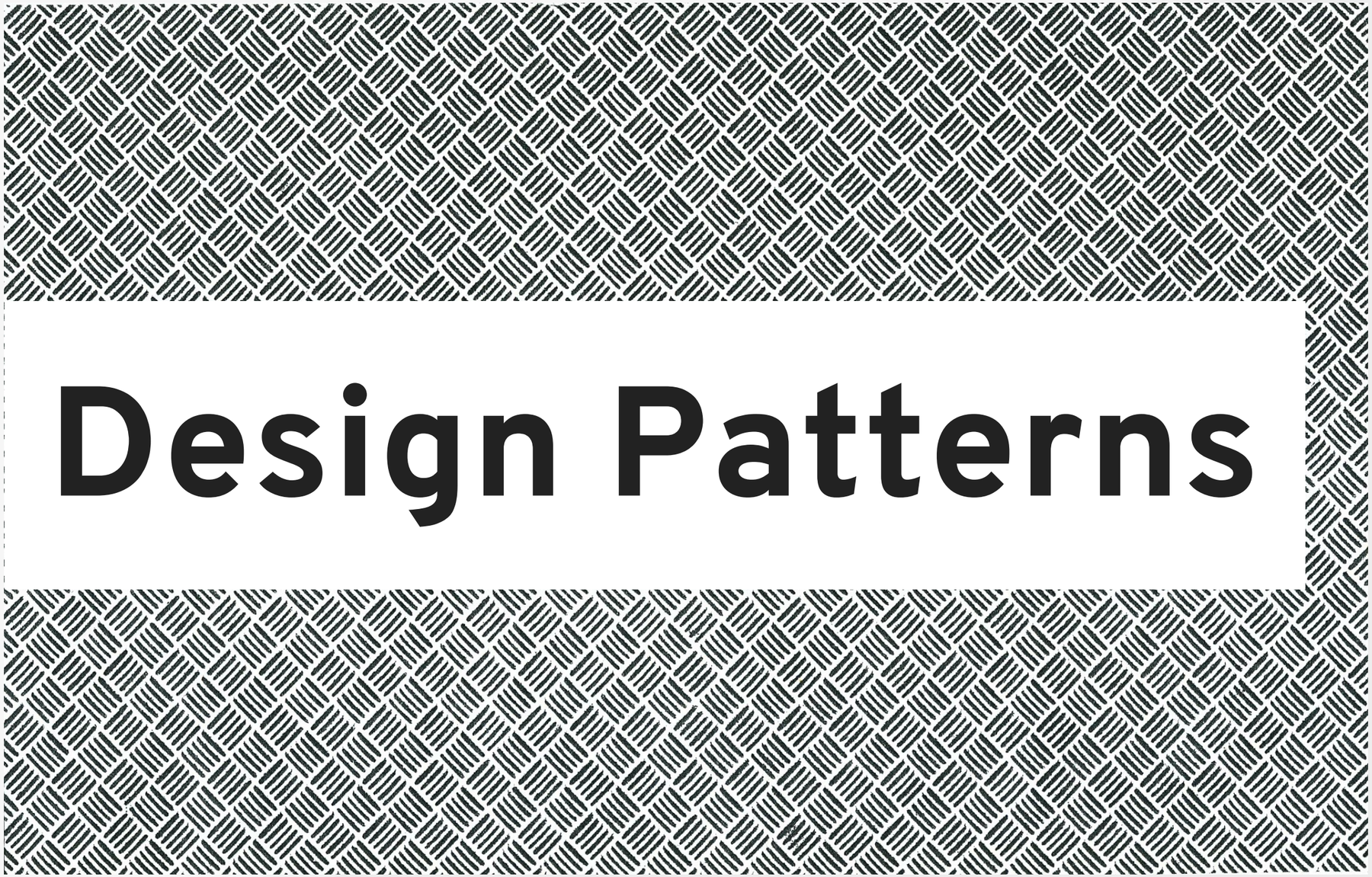
redhat











DESIGN PATTERN

A **Design Pattern** describes a **reusable solution** for a problem within a given context

A Pattern Language

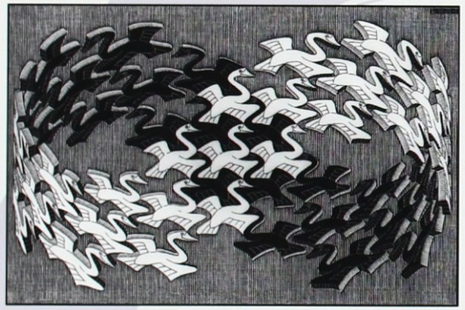
Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

STRUCTURE

- Problem
- Patterns:
  - Name
  - Solution
  - Pro & Contra

http://www.martinfowler.com/articles/writingPatterns.html

CONFIGURATION

How to configure containerized applications for different environments ?

- **Env-Var Configuration**
- **Immutable Configuration**
- **Configuration Service**

ENV-VAR CONFIGURATION

- The Twelve Factor App, Chapter 3
- Standard configuration method for Docker containers
- Specified during **build** or **run** time
- Universal

BUILD

```
FROM jboss/base-jdk:8

ENV DB_HOST "dev-database.dev.intranet"
ENV DB_USER "db-develop"
ENV DB_PASSWORD "s3cr3t"
....
```

RUN

```
docker run \
  -e DB_HOST "prod-db.mycompany.com" \
  -e DB_USER "dbuser" \
  -e DB_PASSWORD "3QkgbLXWPZ2sJcQ" \
  ....
  mycompany/cool-app
```

KUBERNETES

```
---
apiVersion: v1
kind: ReplicationController
spec:
 replicas: 1
 template:
  spec:
   containers:
    - env:
      - name: MYSQL_USER
       value: "mysql"
      - name: MYSQL_PASSWORD
       valueFrom:
        secretKeyRef:
         name: "db-passwords"
         key: "mysql"
      - name: SPECIAL_TYPE_KEY
       valueFrom:
        configMapKeyRef:
         name: "special-config"
         key: "special.type"
```

DOCKER COMPOSE

```
---
version: '2'
services:
 web:
  depends_on:
   - db
  environment:
   DB_USER: "dbuser"
   DB_PASS:
 db:
  image: "orchardup/mysql"
  environment:
   MYSQL_DATABASE: "wordpress"
```

EVALUATION

- E.g. in Java: System.getenv()
- Environment "profiles"
  - Predefined profiles
  - EnvVar selects profile

```
spring:
  profiles: dev
server:
  port: 9001
---
spring:
  profiles: prod
server:
  port: 8080
```

```
docker run \
 -e SPRING_PROFILES_ACTIVE=dev \
 my-springboot-app
```

PRO

- Supported everywhere
- Simple
- Explicit

CONTRA

- Static
- Hard to maintain

IMMUTABLE CONFIGURATION

- Configuration in extra Docker container
- Volume linked during **runtime**
- Example of a **Sidecar**

EXAMPLE

CONFIG DOCKERFILE

```
FROM scratch

ADD "dev_conf" "/usr/local/tomcat/conf"
VOLUME "/usr/local/tomcat/conf"
```

CREATE CONTAINER

```
docker build -t "config-dev-img"
docker create --name "config-dev" "config-dev-img" .
```

START IMAGE

```
docker run --volumes-from="config-dev" -P tomcat
```

DEMO

PRO

- Flexible
- Shareable
- Versioned

CONTRA

- Static
- Maintenance overhead

CONFIGURATION SERVICE

- Active Configuration lookup
- "Configuration-as-a-Service"
- Tools:
  - Apache ZooKeeper
  - Consul
  - etcd
  - Redis
  - ...

PRO

- Flexible
- Dynamic

CONTRA

- External Service (Latency)
- Maintenance overhead
- Weakens Immutability

SERVICE DISCOVERY

How can an application discover its runtime dependencies ?

- **Env-Var Injection**
- **Service Lookup**
- **Dynamic DNS**

ENV-VAR INJECTION

- Injection of services as **Environment Variables**
- Must be set during container start
- Examples:
  - Docker links / Docker network
  - Kubernetes
  - Docker compose

KUBERNETES

- For each active **Service** sets:
  - svcname_SERVICE_HOST
  - svcname_SERVICE_PORT
- Also Docker-style link variables
- Only when **Pod** starts

PRO

- Simple
- No lookup required

CONTRA

- Only Services which exist before startup are available

SERVICE LOOKUP

- Key-Value Stores
- REST API for Registration & Query
- gliderlabs/registrator
  - Automatic registration of Docker containers
  - Backends for Consul, etcd, SkyDNS

PRO

- Explicit
- Dynamic

CONTRA

- Support from platform required
- Vendor lock-in

DYNAMIC DNS

- Standard Protocol for Service Lookup (SRV records)
- Tools for DNS service registry:
  - Consul
  - SkyDNS
- DNS Lookup for Java Clients: JNDI or custom

JNDI DNS

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
     "com.sun.jndi.dns.DnsContextFactory");
env.put("java.naming.provider.url", "dns:");

DirContext ctx = new InitialDirContext(this.env);
Attributes attrs =
 ctx.getAttributes("redis.myproject.svc.cluster.local",
             new String[] { "SRV" });
NamingEnumeration e = attrs.getAll();

while (e.hasMore()) {
 Attribute attr = (Attribute) e.next();
 System.out.println(attr.get());
}
e.close();
```

```
10 33 6379 _redis._tcp.redis.myproject.svc.cluster.local.
```

SPOTIFY/DNS-JAVA

```
import com.spotify.dns.*;

DnsSrvResolver resolver =
    DnsSrvResolvers.newBuilder().build();

List<LookupResult> nodes =
            resolver.resolve("redis");

for (LookupResult node : nodes) {
    System.out.println(
        node.host() + ":" + node.port());
}
```

PRO

- Dynamic
- Standard
- Good infrastructure support

CONTRA

- Deep interaction with OS
- Rusty

DEMO

SIDECAR

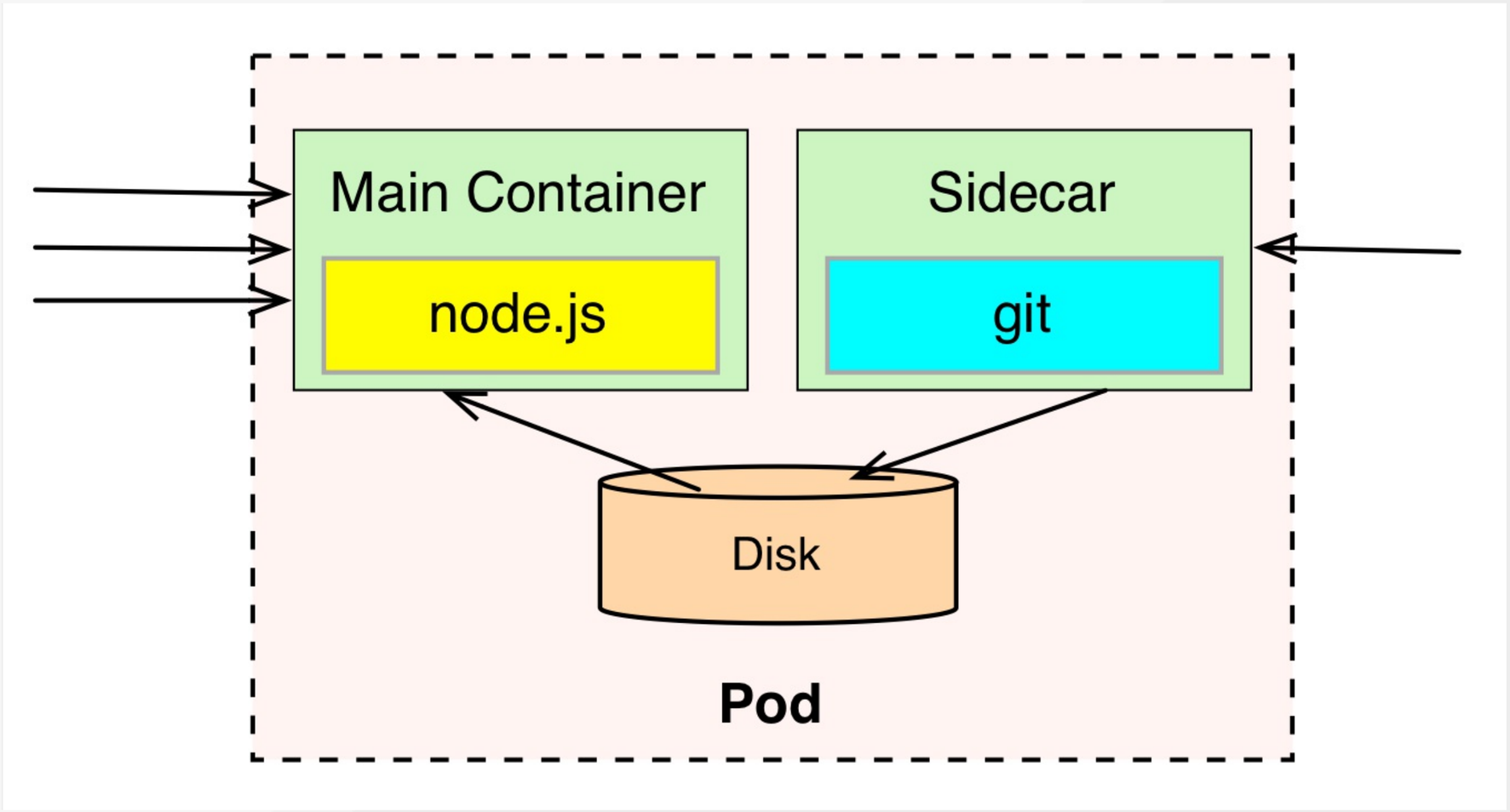
How can I add and decouple functionality for a container without changing it ?

- **Sidecar**
- **Ambassador** (aka Proxy)
- **Adapter**

SIDECAR

- Combine containers to a functional entity (e.g. **Pod** in Kubernetes)
- AOC : Aspect oriented container
- Separation of concerns
- Examples:
  - Logging
  - File Sync
  - Circuit Breaker

SIDECAR

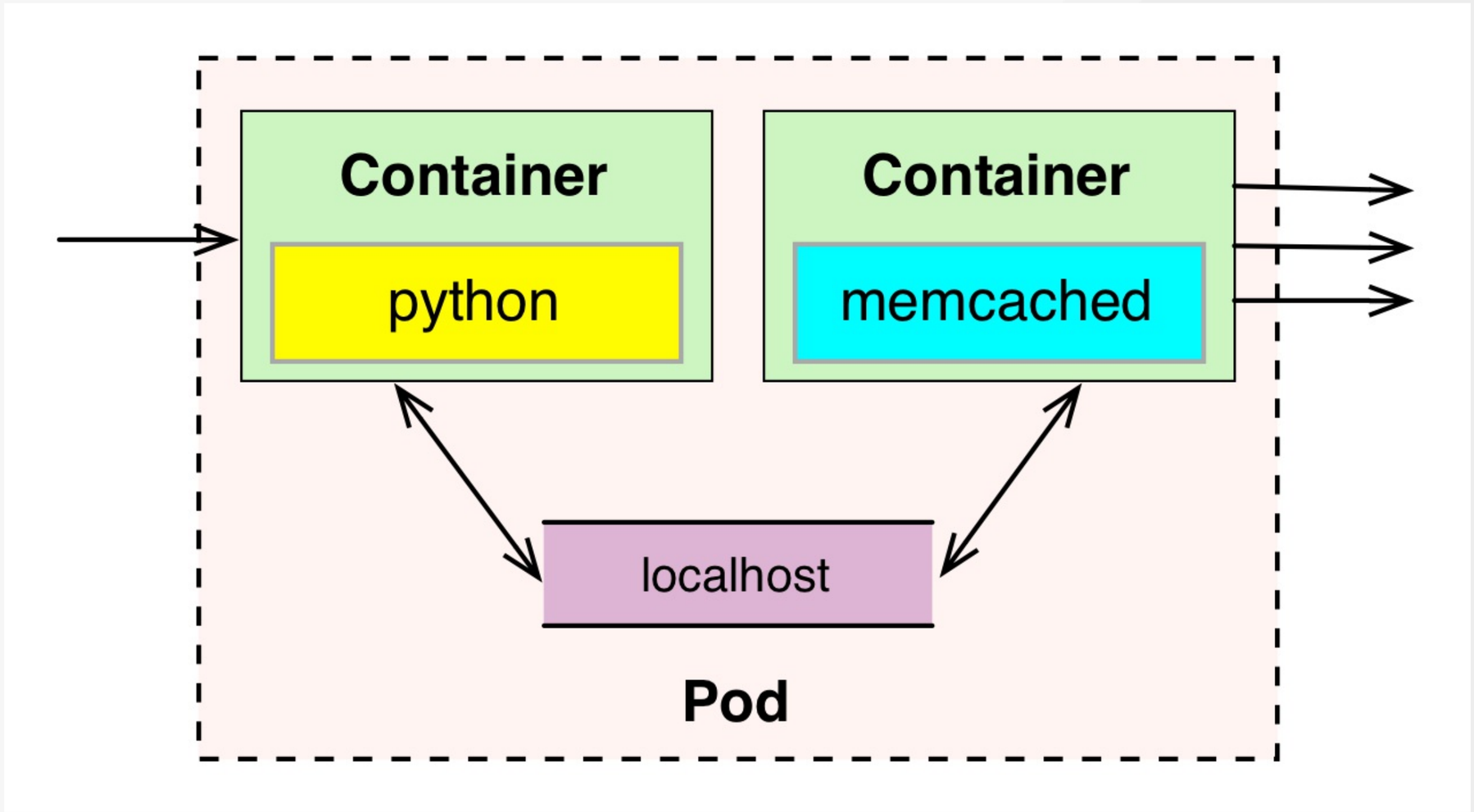


DEMO

AMBASSADOR

- Decoupling of services via a local proxy
- Example:
  - Proxy container connects to target DB and listens at **localhost**
  - Application connects to **localhost**
  - Only change to proxy container required when database changes

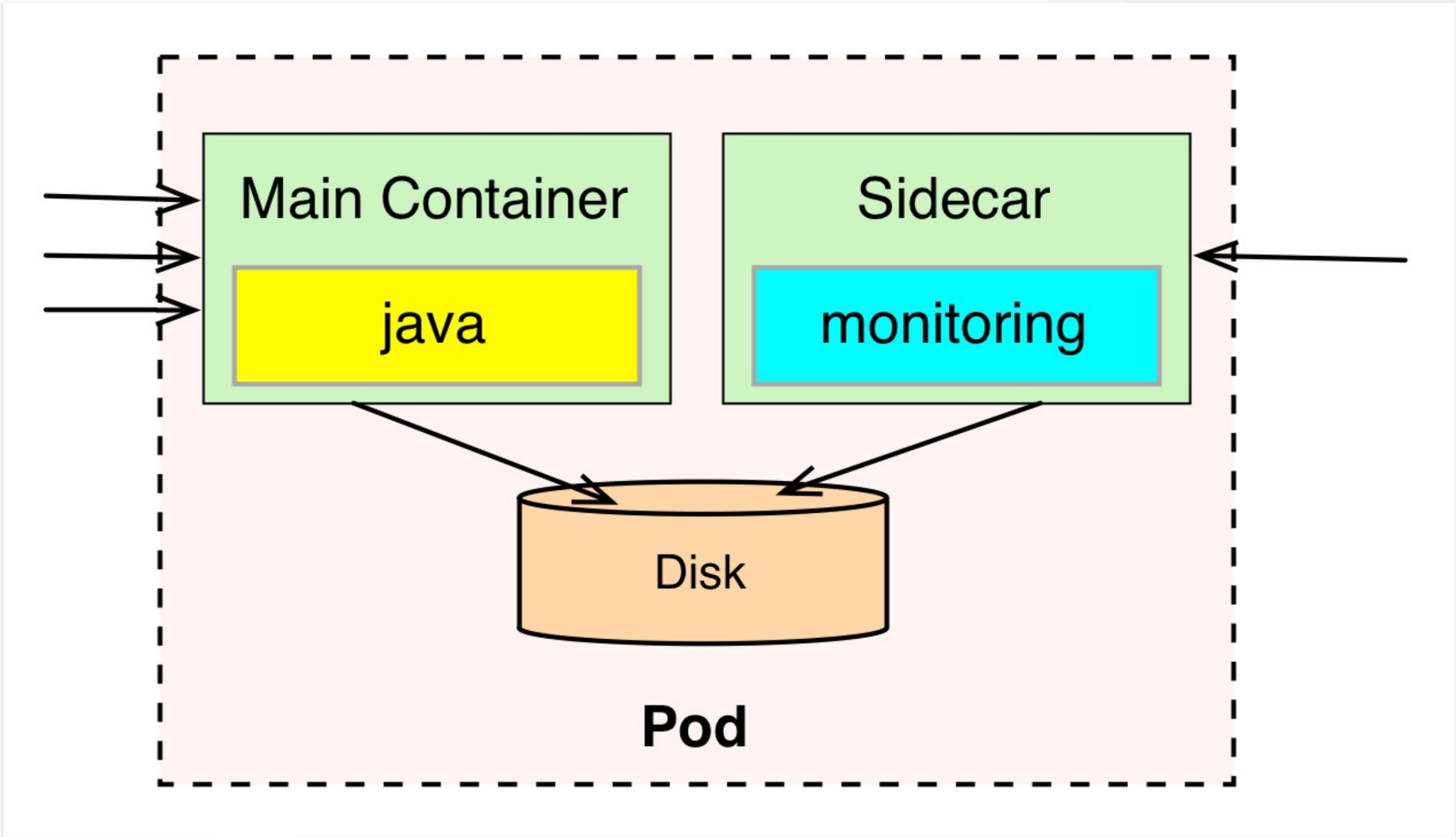
AMBASSADOR



ADAPTER

- Standardize and normalize output
- Example: Monitoring
  - Monitoring data exported in different formats
  - Monitoring backend expects one specific format
  - Different adapters normalize to required format

ADAPTER



PRO

- Decoupling
- Separation of concerns

CONTRA

- More complex
- No direct support in Docker

IMAGE BUILDING

How to create Docker images for Java EE Apps ?

- **Self-contained Application**
- **Artifact Container**
- **Build Integration**

SELF CONTAINED APPLICATION

- Java EE Server and Application in **one** Image
- Server base image

```
FROM fabric8/tomcat-8.0

COPY chat-app.war /opt/tomcat/webapps/chat-app.war
```

PRO

- Single deployment unit

CONTRA

- Coupled lifecycle

ARTIFACT CONTAINER

- Container holding the application artifact
- Linked to a Java EE container
- Java EE server deploys artifact during startup

PRO

- Independent lifecycles
- Separate responsibilities

CONTRA

- Two images to manage

BUILD INTEGRATION

- Create Docker images from within a build
- Plugins for Maven & Gradle
- Maven:

fabric8/docker-maven-plugin

CONFIGURATION

```
<image>
 <name>jolokia/jolokia-itest</name>
 <build>
  <from>consol/tomcat-7.0</from>
  <assemblyDescriptor>
   assembly.xml
  </assemblyDescriptor>
 </build>
 <run>
  <ports>
   <port>jolokia.port:8080</port>
  </ports>
 </run>
</image>
```

DEMO

PRO

- Self contained builds
- No external requirements
- Reuse of existing build configuration

CONTRA

- Own configuration syntax
  - but also supports plain Dockerfiles

BONUS

WORMHOLE

How can a container access its managing Docker daemon ?

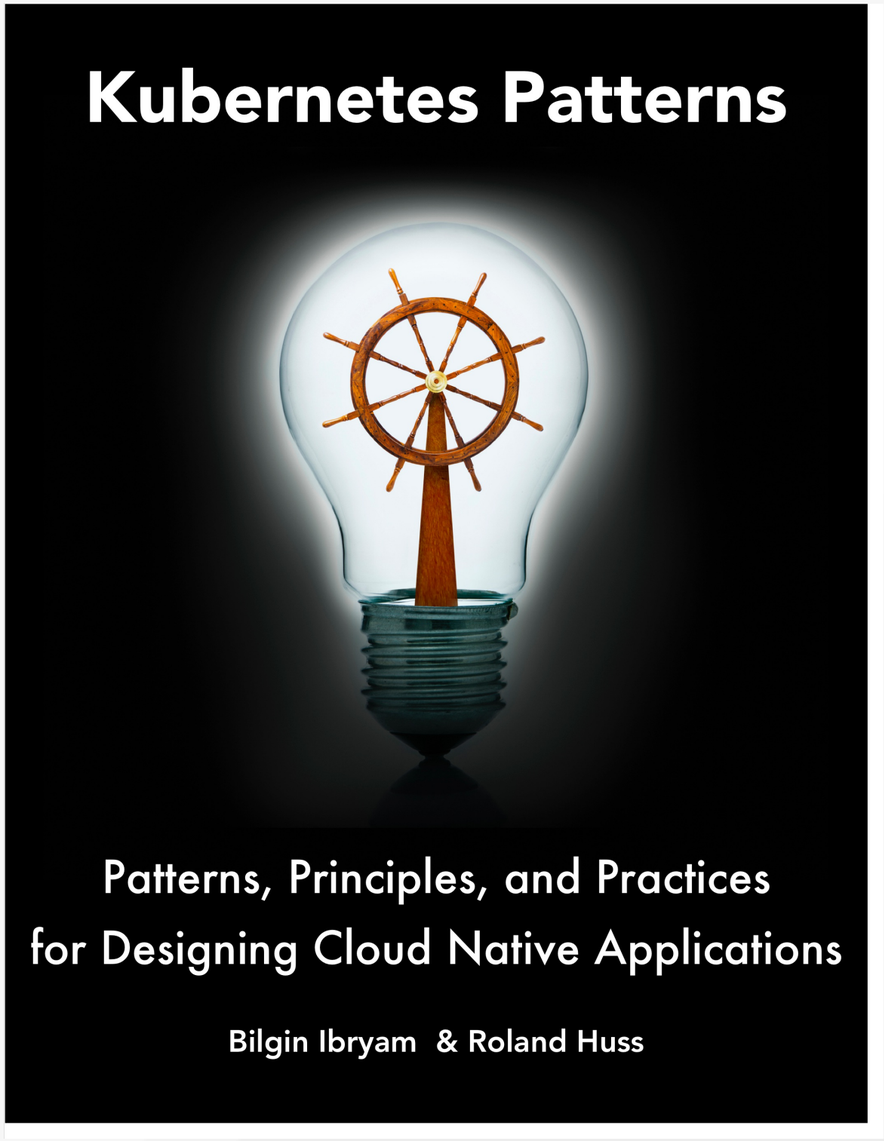
```
docker run \
 -v /var/run/docker.sock:/var/run/docker.sock  \
 ...
```

- Alternative: "Docker-in-Docker"

https://ro14nd.de/Docker-Wormhole-Pattern

WRAP UP

- Patterns can help in solving recurring Docker challenges.
- Multiple patterns exists for **Configuration**, **Service Discovery** and **Image building** which are complementary.
- A **Sidecar** can easily decouple stuff



https://leanpub.com/k8spatterns

redhat

redhat

QUESTIONS ?

Twitter @ro14nd

Slides https://github.com/ro14nd-talks/docker-patterns

redhat

DOCKERFILE TEMPLATE

- Dockerfile generation from templates
- Build with **docker build**
- Simple built-in support (**ARG**)
- Engines:

  fish-pepper, dogen, crane, App::Dockerfile::Template ...

DOCKERFILE ARGS

- Dockerfile variables
- ~~Can not be used in FROM~~
- Filled in during build time

```
FROM busybox
ARG uid
USER ${uid:-jboss}
# ...
```

```
docker build --build-arg uid=daemon ....
```

FISH-PEPPER

- **Template** system based on node.js
- **Multidimensional** parameters
- Support for **fragments**
- Docker **build** and **push** support

https://github.com/fabric8io-images/fish-pepper

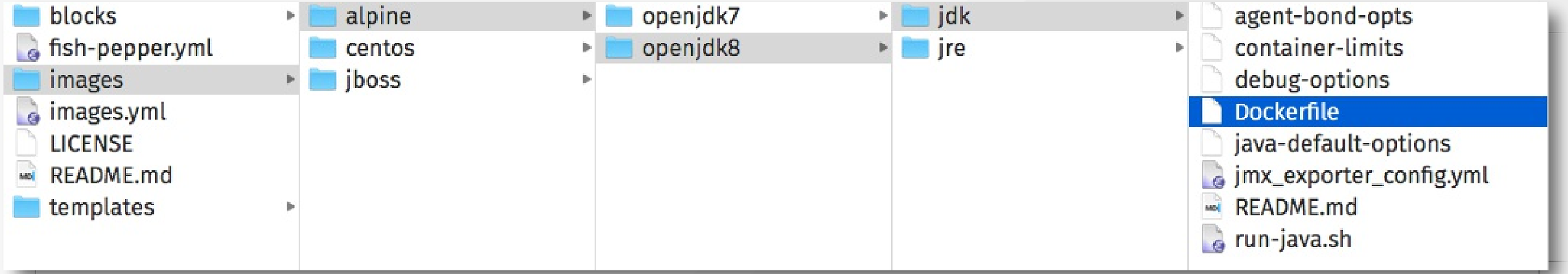
IMAGES.YML

```
fish-pepper:
 params:
  - "base"
  - "version"
  - "type"
 name: "jolokia/fish-pepper-java"
 maintainer: "Roland Huss <roland@jolokia.org>"
 # ... additional global params
config:
 version:
  openjdk7:
   java: "java:7u79"
  openjdk8:
   java: "java:8u45"
 type:
  jre:
   extension: "-jre"
  jdk:
   extension: "-jdk"
 base:
  alpine:
   from: "alpine:3.4"
  # ... additional base definitions for 'centos', 'jboss'
```

DOCKERFILE TEMPLATE

```
FROM {{= fp.config.base.from }}

MAINTAINER {{= fp.maintainer }}

ENV JOLOKIA_VERSION {{= fp.jolokiaVersion }}

RUN chmod 755 /bin/jolokia_opts \
  && mkdir /opt/jolokia \
  && wget {{= fp.jolokiaUrl}} -O /opt/jolokia/jolokia.jar

CMD java -jar /opt/jolokia/jolokia.jar --version
```

GENERATED BUILDS



redhat

PRO

- Simple
- Uses Docker builtin mechanism
- Docker Hub automated builds still possible
- Good for many similar builds

CONTRA

- Restricted to Dockerfiles

IMAGE FACTORY

- Creating images without Dockerfiles
- Provisioning must be **reproducible** and **automated**
- Tools:
  - ansible-container
  - Packer

ANSIBLE-CONTAINER

- Tool for building Docker images with Ansible
- Simple orchestration for containers
- Uses **docker exec** and **docker cp** as Ansible connector

HOW IT WORKS

- **Start up base containers** from the specification given in container.yaml
- Create Ansible **inventory** on the fly
- Runs an Ansible playbook for **provisioning**
- **Stop** containers
- **Commit** containers as images

PRO

- Flexible
- Reuse of Ansible roles
- Single layered images

CONTRA

- Complex system
- Longer Build times
- Single layered images