



DOCKER PATTERNS

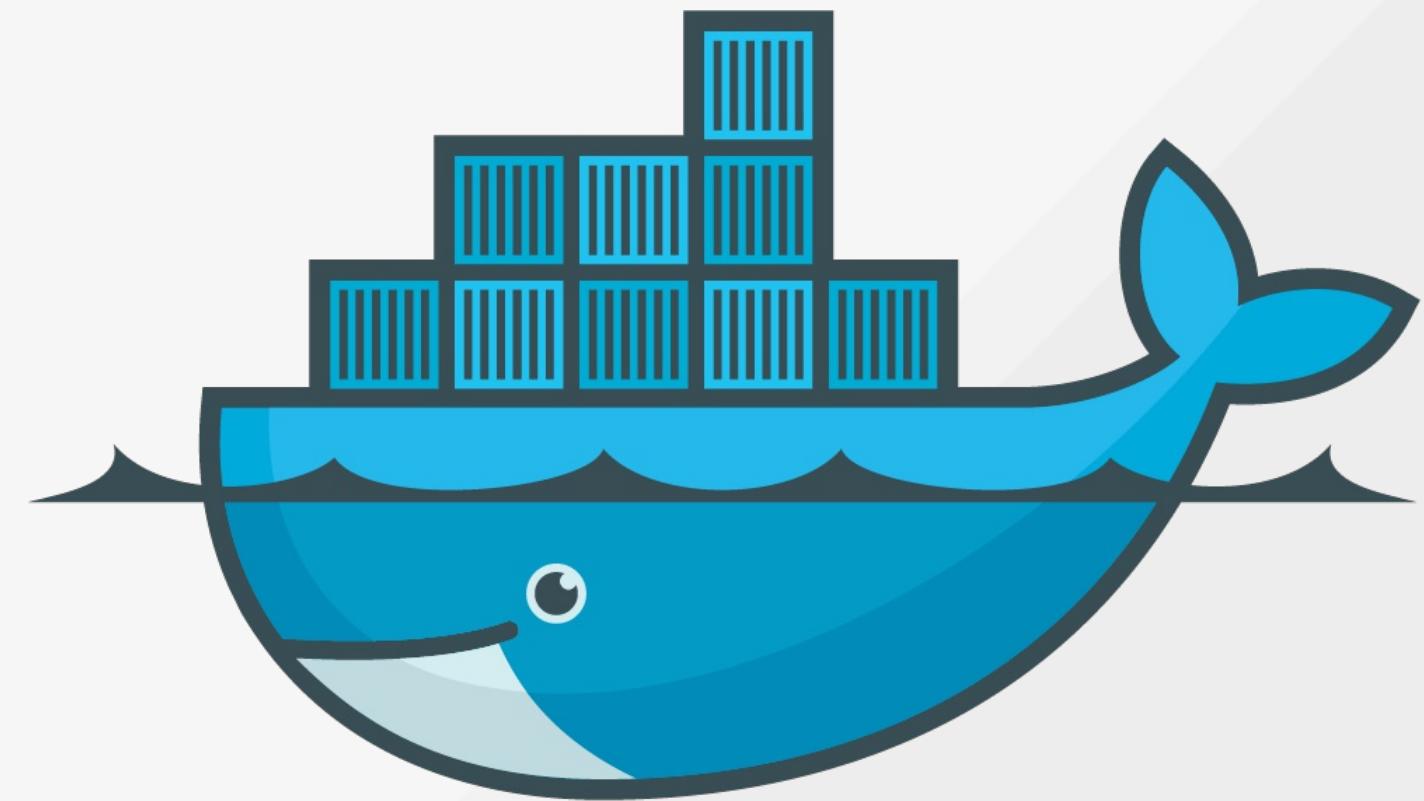
Dr. Roland Huß, Red Hat, @ro14nd



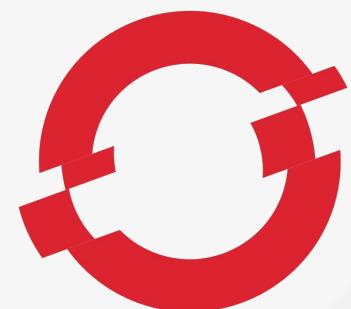
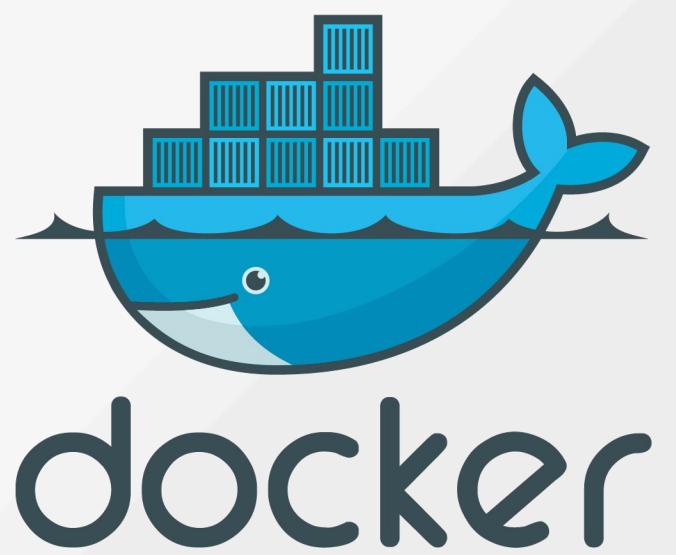
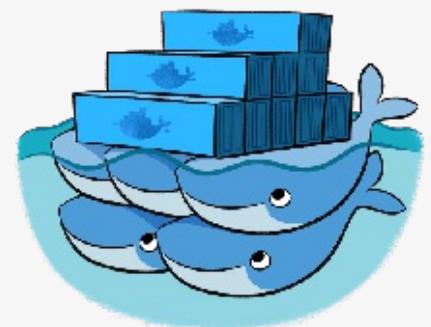
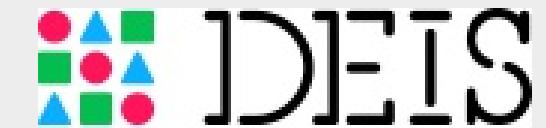
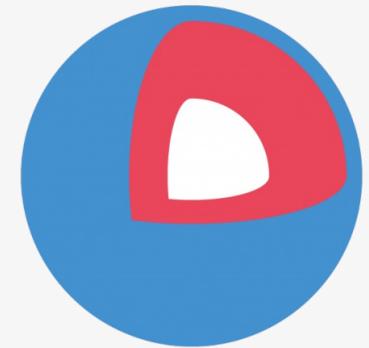
"m" for menu, "?" for other shortcuts

AGENDA

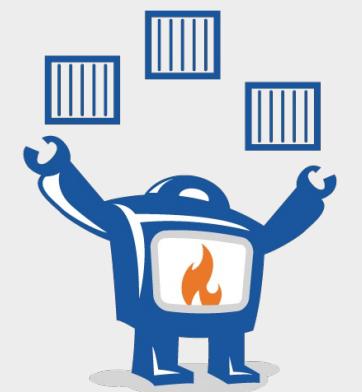
- Docker & Patterns
- Configuration
- Service Discovery
- Sidecar
- Image Builder
- Bonus



docker



OPENSHIFT



fabric8



Design Patterns

DESIGN PATTERN

A **Design Pattern** describes a
repeatable solution for a
software engineering **problem**.

A Pattern Language

Towns • Buildings • Construction



Christopher Alexander

Sara Ishikawa • Murray Silverstein

WITH

Max Jacobson • Ingrid Fiksdahl-King

Shlomo Angel

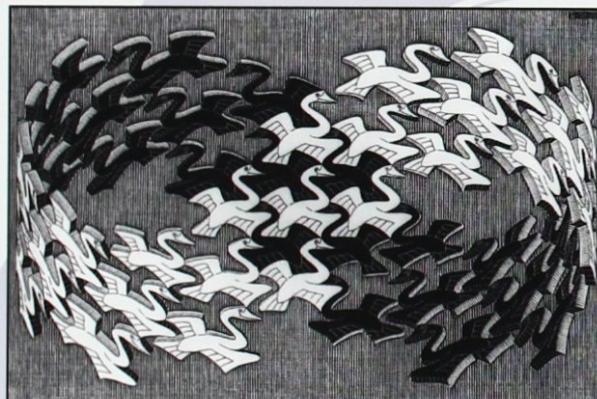


redhat

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



STRUCTURE

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

CONFIGURATION

How to configure containerized applications for different environments ?

- Env-Var Configuration
- Configuration Container
- Configuration Service

ENV-VAR CONFIGURATION

- 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
```

DEMO

PRO

- Supported everywhere
- Simple and Easy
- Explicit

CONTRA

- Static
- Hard to maintain

CONFIGURATION CONTAINER

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

CONFIG DOCKERFILE

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

CREATE CONTAINER

```
docker run --name "dev-config" \  
$(docker build -q . | tail -1)
```

START IMAGE

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

DEMO

KUBERNETES CONFIGMAP

- Kubernetes Resources that can be mounted as volumes
- ConfigMap is created on deployment time
- Analogous to plain Docker volume mounts

PRO

- Flexible
- Explicit

CONTRA

- Static
- Maintenance overhead
- Custom file layout

CONFIGURATION SERVICE

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

DEMO

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
 - Custom (e.g. envconsul + gliderlabs/registrator)

KUBERNETES

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

FABRIC8 CDI

```
@Factory  
@ServiceName  
public DataSource create(@ServiceName  
    @Protocol("jdbc:mysql")  
    @Path("db")  
    String url) {  
    MysqlDataSource mysqlDS = new MysqlDataSource();  
    mysqlDS.setUrl(url);  
    mysqlDS.setUser(System.getProperty(USERNAME, "root"));  
    mysqlDS.setPassword(System.getProperty(PASSWORD, "pass"));  
    return mysqlDS;  
}
```

```
@ServiceName("dev-mysql")  
DataSource mySqlD;
```

PRO

- Simple
- No lookup required

CONTRA

- Only services during startup available
- Supporting platform required
(Kubernetes)

SERVICE REST LOOKUP

- Key-Value Stores
- REST API for Registration & Query
- [gliderlabs/registrator](https://github.com/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()  
        .cachingLookups(true)  
        .retainingDataOnFailures(true)  
        .dnsLookupTimeoutMillis(1000)  
        .build();  
  
List<LookupResult> nodes =  
    resolver.resolve(  
        "redis.myproject.svc.cluster.local");  
  
for (LookupResult node : nodes) {  
    System.out.println(  
        node.host() + ":" + node.port());  
}
```

PRO

- Dynamic
- Standard
- Good infrastructure support

CONTRA

- Deep interaction with OS
- Rusty

SIDE CAR

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

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

SIDECAR

- Combine containers to a functional entity (e.g. **Pod** in Kubernetes)
- AOC : Aspect oriented container
- Separation of concerns
- Examples:
 - Logging
 - File Sync
 - Funktion: Event driven, lambda style coding

PROXY

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

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

PRO

- Decoupling
- Separation of concerns

CONTRA

- More complex

IMAGE BUILDING

How to build Docker images ?

- Dockerfile Template
- Image Factory
- Self-contained
- Artifact Container
- Build Integration

CLASSIC STYLE

- docker commit

```
docker commit 3e3d3cf39 redis-server:1.0
```

- docker build

```
FROM ubuntu:14.04
RUN apt-get update && \
    apt-get install -y redis-server
EXPOSE 6379
ENTRYPOINT ["/usr/bin/redis-server"]
```

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
- Since Docker 1.9
- 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
- Support for fragments
- Hierarchical
- Docker build and push support

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



FISH-PEPPER

```
# Configuration to fill in templates
fish-pepper:
  params:
    - "version"
    - "type"
  name: "jolokia/fish-pepper-java"
# Global template vars
# ...
config:
  version:
    openjdk7:
      fish-pepper:
        version: "1.7"
      # Template variables
      java: "java:7u79"
    openjdk8:
      fish-pepper:
        version: "1.8"
      java: "java:8u45"
  type:
    jre:
      extension: "-jre"
    jdk:
      extension: "-jdk"
```



FISH-PEPPER

```
FROM {{= fp.config.version.java + fp.config.type.extension }}

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
```

DEMO

PRO

- Simple
- Uses Docker builtin mechanism
- Good for many similar builds

CONTRA

- Restricted to Dockerfiles
- Standard solution very limited

IMAGE FACTORY

- Creating images without Dockerfiles
- Flow:
 - Start container from a **base image**
 - **Provision** stuff to the container
 - **Commit** container as image
- Provisioning must be **reproducible** and **automated**

ANSIBLE

```
- name: initialize provisioning
hosts: docker
- name: start up target container
  docker:
    image: java:8.0
    name: fis-karaf
    # ....
    command: sleep infinity
    state: started
# dynamically update inventory
- name: register new container hostname
  add_host: name=lab
- name: provision container
  connection: docker
  hosts: lab
  tasks:
    # ...
- name: finalize build
  hosts: docker
  tasks:
    - name: stop container
      docker:
        name: fis-karaf
        state: stopped
```

PRO

- Composition
- Parametrisation
- Less image layers

CONTRA

- Extra Tooling
- No builds by Docker Hub

SELF CONTAINED

- JEE 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 JEE container
- JEE 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:
 - <https://github.com/fabric8io/shootout-docker-maven>

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
- More than one way

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"

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



QUESTIONS ?

Blog <https://ro14nd.de>

Slides `firefox $(curl -sL http://bit.ly/docker-patterns | sh)`