gepardec

redhat.

PREMIER

BUSINESS PARTNER

Docker

Architectural Impact

START >

Author: Ing. Thomas Herzog B.Sc / Version 1.0

Docker Architectural Impact

- Applications
  - Configure
  - Deploy
  - Backup/Restore
- Monitoring
  - (Distributed) Logging
  - Classical Monitoring Systems
- Security
  - Docker Infrastructure
  - Docker Image-Registry
  - Docker Containers

https://docker.com

Application

Monolith:

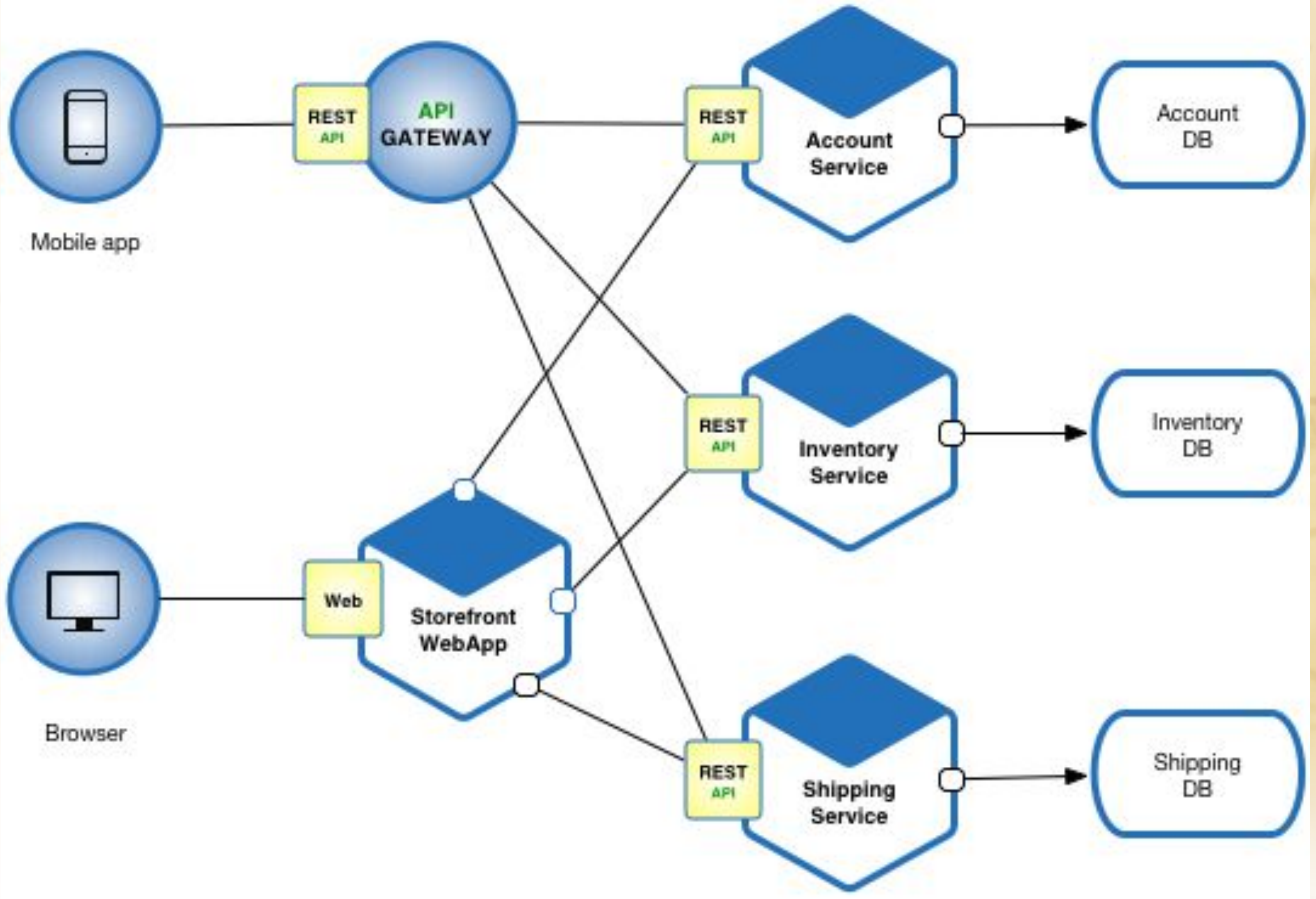
- One repository
- One artifact represents application
- One build, one deployment
- One configuration per deployment
- Touches ConfigSource
- Uses Secrets, and knows them
- Centralized logging
- Application in one runtime
- Fix/Deploy whole application
- ...

Microservice:

- N repositories
- N artifacts represents application
- N builds and deployments
- One configuration per N deployments
- No reference to ConfigSource *(MicroProfile Config)*
- Uses Secrets, but doesn’t know them
- Distributed Logging *(MicroProfile OpenTracing)*
- Application in N runtimes
- Fix/Deploy single Service
- ...

Application

- More flexible and complex
  (but not complicated)
- Easier to maintain
- Harder to manage
- Application is distributed
- New problems
  - Failures are distributed
  - Logs are distributed
  - Possibility of Cycles
  - Service resilience very important



https://www.youtube.com/watch?v=TvnZTi_gaNc

Application - Configure

- Configurations can be provided via:
  - **Environment variables**

    ```
    docker run -e MY_VAR=MY_VAL ...
    ```

  - **Files**

    ```
    docker run -v /config.cf:/conf/config.cf ...// Single file
    docker run -v /config:/conf ...             // Directory of files
    ```

  - **CMD as arguments for ENTRYPOINT**

    ```
    docker run image:latest ‘-Dswarm.project.stage=dev’
    ```

- Docker provides CLI for managing secrets/configs *(Swarm only)*
- Kubernetes provides ConfigMaps/Secrets
  - Developers don’t see configuration values and secrets anymore
  - Provides mechanisms for injecting configs and secrets into containers

Application - Configure

- Applications can still package all configurations *(not recommended)*
  - controlled via switch *(e.g.: via environment variable)*
- Application should expose configurations
  - So that application can be configured for N stages without rebuild
- One build, one artifact, N stages
  - We want to move artifact or container over all stages
- Eclipse MicroProfile Config-API *(https://microprofile.io/)*
  - Abstracts developer form ConfigSource
  - Mechanisms to consume configurations as usually provided in the cloud
  - Several ConfigSource Types supported such as URL

Application - Deploy

- Binary, Dockerfile and Scripts to build on target environment
- Ready to use Docker Image
  - Runs anywhere, where Docker is supported
- Docker Compose definition *(good for static application environments)*
  - docker-compose.yml, Binary and Dockerfile to build on target environment
  - or docker-compose.yml which references ready to use Docker Images
- Templates when running on CaaS/PaaS *(Kubernetes, Openshift, Azure, ...)*
  - Deploy provided Docker Image,
  - or deploy self built Docker Images *(BuildConfig - Openshift)*

Application - Deploy

- When Dockerfile and Binaries are provided ensure
  - compatibility with used Linux Kernel,
  - compatibility with Docker Version,
  - compatibility of the scripts,
  - and the actuality and security of the provided resources.
- When Docker Images are provided ensure
  - to use a safe Docker Base-Image source *(RHEL),*
  - to keep Docker Images as small as possible *(RHEL Atomic, Alpine),*
  - to provide Docker Images via secured Docker Image-Registry,
  - that there are no secrets in the Docker Image layers,
  - and that Docker Container is ephemeral *(Drop and recreatable with little config).*

Application - Backup

- Docker Container use Docker Volumes to keep data persistent
- It is not as easy as usual to get to the persistent data
- Prefer native backups over `docker container commit`
- Backup running container

  -
    ```
    docker container exec -i -u root
    -v /backup:/backup
    mysql-db /usr/bin/mysqldump mydb > /backup/backup.sql
    ```
- Backup stopped container

  -
    ```
    docker container run -i -u root
    -v /backup:/backup --volumes-from mysql-db
    backup:latest /usr/bin/tar -zcvf /data/dump.tar.gz /mysql/data
    ```

Application - Restore

- If `docker container commit` was used, create Container of commited image
- Restoring running container

  -
    ```
    docker container exec -i -u root
    -v /backup:/backup
    mysql-db /usr/bin/mysql < /backup/backup.sql
    ```
- Restoring stopped container

  -
    ```
    docker container run -i -u root
    -v /backup:/backup --volumes-from mysql-db
    backup:latest /usr/bin/tar -xvf /data/dump.tar.gz
    ```
- Kubernetes/Openshift provide no native backup support

Monitoring

- Docker CLI has little support for monitoring
  - `docker inspect <[Image-Id, Container-Id]>`
  - `docker logs -f <Container-Id>`
  - `docker stats <Container-Id>`
- Docker can contribute to Prometheus *(experimental)*
- Other tools available for monitoring
- Labels are crucial for monitoring containers *(Kubernetes, Openshift)*
- Application itself can contribute to any monitoring tool

Monitoring - Logging

- stdin/stderr are captured by Docker
  - `docker logs -f <Container-Id>`
- Docker provides several drivers
- Application itself can send logs to log server
- No log to file, only stdout/stderr
- Applications log must provide transaction-id *(MicroProfile OpenTracing)*
- Logs are very important to analyze failures in a distributed system
- Openshift provides EFK stack *(Elastic, FluentD, Kibana)*

Monitoring - Classical monitoring sys.

- Maybe they already support Docker/Kubernetes/Openshift
- Can, but doesn't have to run in a Docker Container
- New way of monitoring applications is via agent *(Java Agent)*
- Lot of provider already on the market:
    - CoScale Openshift Monitoring
    - Dynatrace Openshift Monitoring
    - hawt.io Java Web-Console *(Fuse Integration Services 2.0)*
    - …

Security - Docker Infrastructure

- Use build in Linux Security of Docker Host *(SELinux)*
- Restrict access Docker Host *(Linux user/group permissions)*
- Don’t enable remote access *(Docker API-Server)*
  - If, then only via Client-Certificate-Authentication
- Don’t run privileged containers *(no root access)*
- Don’t use legacy repositories *(don’t use registry v1)*
- Don’t use insecure registries *(no Docker Hub)*

Security - Docker Image-Registry

- Use encrypted transport protocol *(HTTPS only)*
- Restrict access to registry *(user/group permissions)*
- Restrict pushes to registry *(prevent distribution of insecure images)*
- Don’t mirror insecure registries *(no Docker Hub)*
- Allow only signed content *(especially for production)*
- Separate registries for usage
  - Production (Released, signed and production ready Docker Images)
  - Tooling (Tooling for development)
  - ...

Security - Docker Container

- Don’t use unsigned Docker Images
- Don’t use Docker Images from insecure/untrusted registries
- Don’t use Docker Images which contain secrets/configurations
  - Provide secrets/configurations on startup
- Use minimized Docker Images *(no curl, ping or such installed)*
- Never run Docker Containers as root user
- Expose only necessary resources

Security - Docker Container

- Security mostly applied outside, therefore:
  - Application uses http://localhost:8080 instead of https://localhost:443
  - User access controlled via *(OAuth2)*
- Never run Docker Container as root user
- Expose only necessary resources
- Keep backend container in backend network *(no external access)*