DOCKER FOR ENTERPRISE DEVELOPERS

HOW WE TEACH

- Docker believes in learning by doing, with support.
- The course is lab driven with lecture.
- Work together
- Ask questions at any time

SESSION LOGISTICS

- 2 days duration
- mostly exercises
  - Take time to read the code examples closely
- regular breaks

ASSUMED KNOWLEDGE AND REQUIREMENTS

- Familiarity with using the Linux command line
- A UCP License (download one at https://store.docker.com/bundles/docker-datacenter/purchase?plan=free-trial)
- You should know the basics of Docker
  - Run a Docker container
  - Search for and pull images from Docker Store
  - Use Docker for Mac / Windows on your local machine

YOUR LAB ENVIRONMENT

- You have been given several instances for use in exercises.
- Ask instructor for access credentials if you don't have them already.

COURSE LEARNING OBJECTIVES

By the end of this course, learners will be able to:

- Describe the essential patterns used in a highly distributed EE application
- Understand how to configure EE applications for different environments without code change
- Produce and containerize scalable, accessible, and fault-tolerant EE applications
- Apply different debugging and testing techniques to containerized EE applications

DISTRIBUTED APPLICATION ARCHITECTURE

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

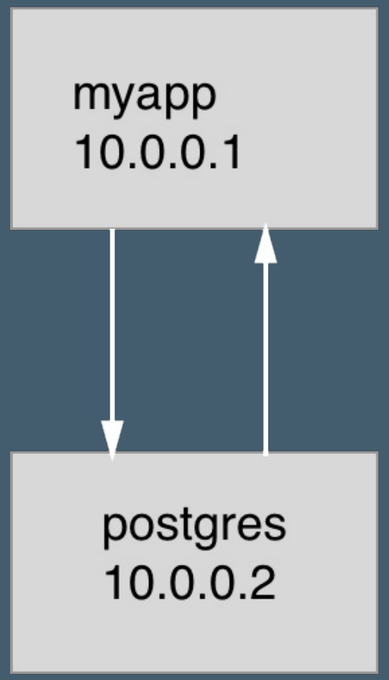
- Describe how familiar design patterns relate to containerized applications
- Enumerate key challenges in containerized application design

DISTRIBUTED APPLICATION ARCHITECTURE

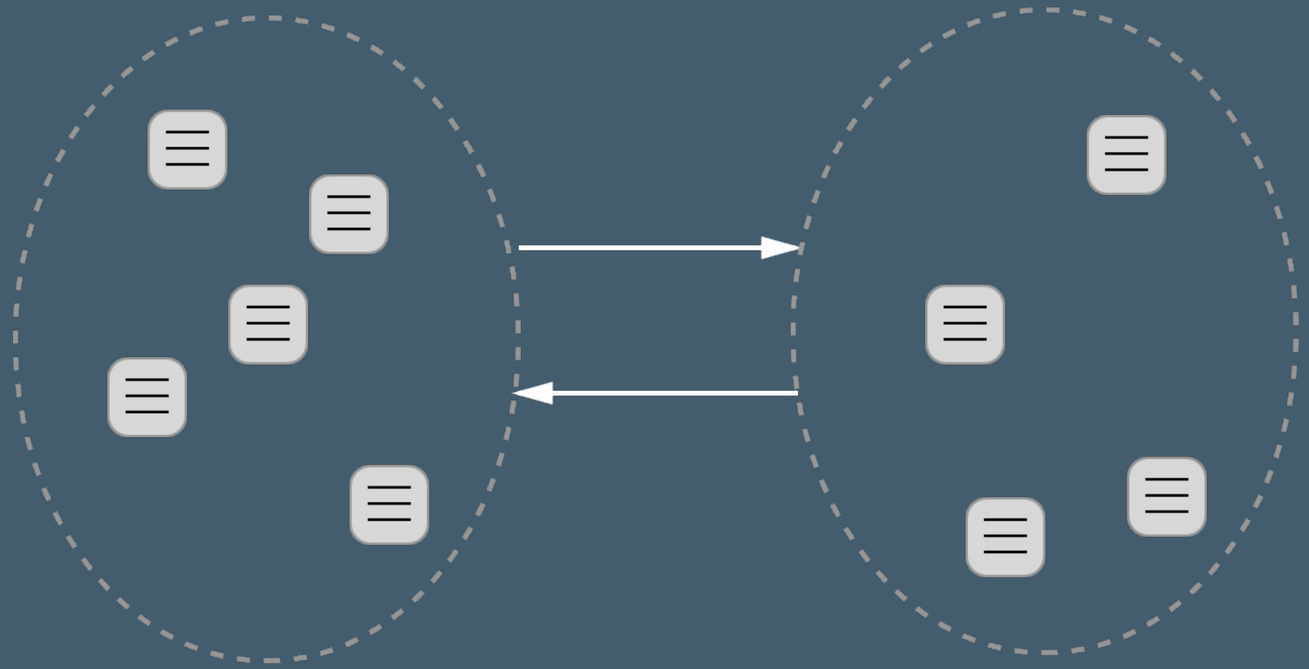
Definition: An application consisting of one or more processes running on one or more nodes.

SERVICES OVER PROCESSES

process to process



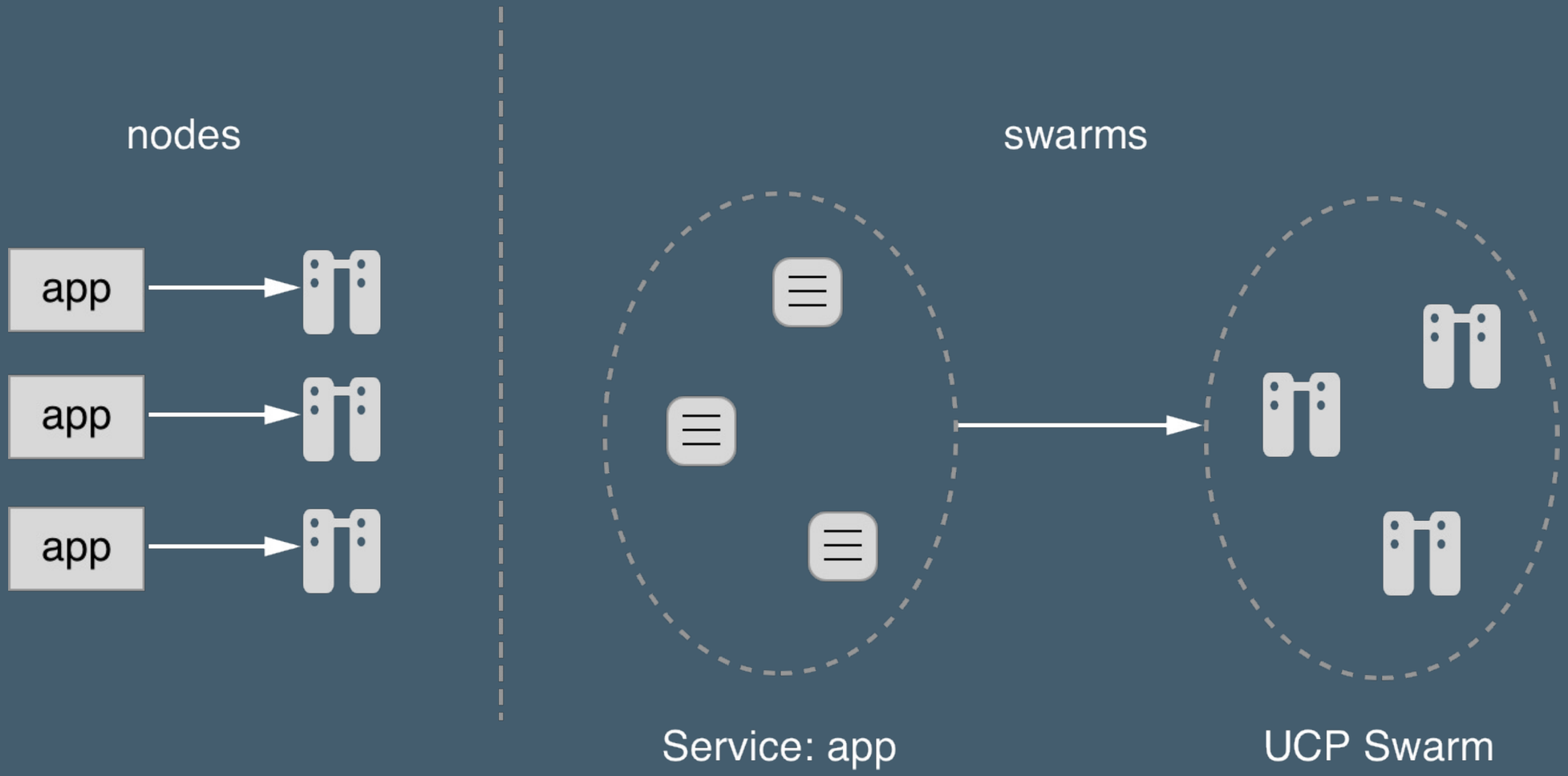
service to service



Service: myapp
10.0.0.1

Service: postgres
10.0.0.2

CLUSTERS OVER NODES



CHARACTERISTICS & REQUIREMENTS

- Bandwidth and latency
- Ephemeral components
- Stateless versus stateful
- Service discovery
- Load balancing
- Health checking
- Logging and monitoring
- Circuit breakers

BANDWIDTH & LATENCY

Action	Real Time	Scaled Time
1 CPU Cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid state disk I/O	50-150 us	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF -> NYC	40 ms	4 years
Internet: SF -> UK	81 ms	8 years
Internet: SF -> Australia	183 ms	19 years
OS Virtualization Reboot	4 s	423 years
SCSI Command timeout	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 min	32 millenia

- Enterprise apps consist of many components
- Remote calls are always expensive
- Keep distances short
- Co-locate where possible
- Use high bandwith connections

EPHEMERAL CONTAINERS

- Pets versus cattle
- Short lifecycle
- Don't rely on specific containers
- Fast initialization / shutdown

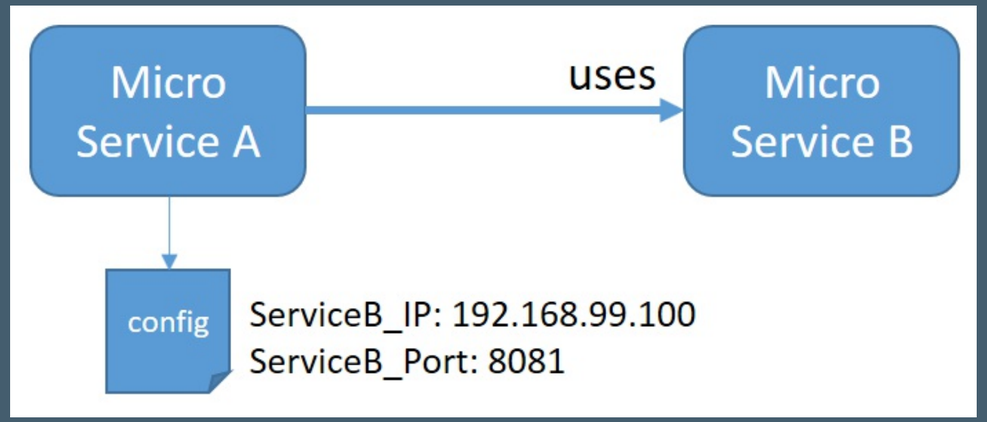
STATELESS VERSUS STATEFUL

- Only stateless components scale well
- Scaling stateful components is hard
- Stateful: DB, Filesystem, Blob storage, Cache, etc
- Developers: push stateful info out to volumes and databases

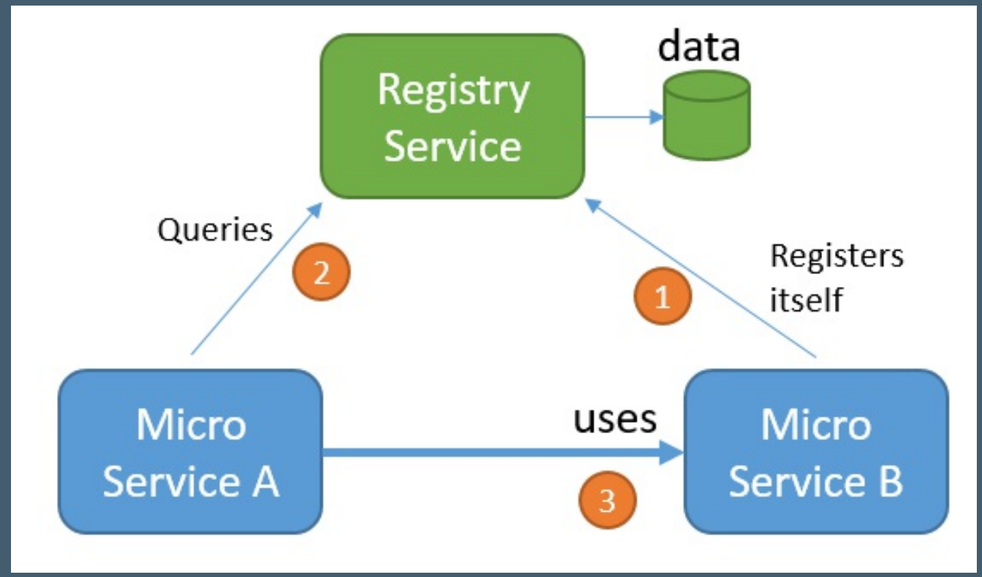
SERVICE DISCOVERY

- Containers are rescheduled frequently
- Hard-coded routes to specific containers will constantly break
- Use Docker's built-in DNS resolver for service names

Bad



Good



LOAD BALANCING

- Service-to-service communication essentially built-in load balancing
- Another reason for statelessness
- Design containers to participate in round-robin LB

HEALTH CHECKING

- Docker EE has no insight into your application logic
- Must provide healthcheck orchestrator can probe for application health
- Unhealthy containers/pods killed and restarted

LOGGING AND MONITORING

- How do we troubleshoot distributed app?
- Centralized logging, log collation & attribution
- Dashboards showing key metrics

CIRCUIT BREAKERS

Avoid cascading failures

- Use cached (stale) data
- Gracefully degrade functionality
- Give failed service time to recover

ARCHITECTURE TAKEAWAYS

Container Developer To-Dos:

- Design for services interacting with other services
- Be radically stateless
- Expect containers to start and stop all the time
- Use Docker EE's service discovery mechanisms
- Provide healthchecks for everything
- Use circuit breaking techniques for damage control
- Design logs for distributed systems

DISCUSSION

- What are the relative strengths and weaknesses of monolithic versus microservice architecture?
- Questions?

FURTHER READING

- View logs for a container or service: http://dockr.ly/2ezdZdI
- Docker Reference Architecture: Docker Logging Design and Best Practices: http://dockr.ly/2gG6ZjG
- Monitor Docker Trusted Registry: https://dockr.ly/2HIGTIw
- High availability architecture and apps with Docker EE: http://dockr.ly/1sqPrIH

SAMPLE APPLICATION

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Describe the components of a demonstration three tier web app
- Build and run the sample app using Docker Swarm or Kubernetes

SAMPLE APPLICATION ARCHITECTURE

- Backend
  - Spring Boot Java API
  - Postgres Database
- Frontend
  - Node/Express JS
  - Mustache Templates

SAMPLE APPLICATION - COMPONENTS

FRONTEND

- Node/Express JS
- Dockerfile, inherit from Node

```
FROM node:8-alpine
RUN mkdir /app
WORKDIR /app
COPY package.json /app/
RUN npm install
COPY ./src /app/src
EXPOSE 3000
CMD node src/server.js
```

BACKEND

- Maven, pom.xml
- Dockerfile, inherit from Maven
- Multi-stage build

```
FROM maven:3.5.0-jdk-8-alpine AS appserver
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:resolve
COPY . .
RUN mvn package

FROM java:8-jdk-alpine
WORKDIR /app
COPY --from=appserver /app/target/pets-api-1.0.0.jar .
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app/pets-api-1.0.0.jar"]
```

SAMPLE APPLICATION - APP DEFINITION

Docker Compose

```
version: "3.1"

services:
  database:
    build:
       context: database
    image: ddev_db
    environment:
      POSTGRES_USER: gordonuser
      POSTGRES_DB: ddev
    ports:
      - "5432:5432"
    networks:
      - back-tier
    secrets:
      - postgres_password

  api:
    build:
       context: api
    image: ddev_api
    ports:
      - "8080:8080"
      - "5005:5005"
    networks:
      - front-tier
      - back-tier
    secrets:
      - postgres_password

  ui:
    build:
      context: ui
    image: ddev_ui
    ports:
      - "3000:3000"
    networks:
      - front-tier

secrets:
  postgres_password:
    file: ./devsecrets/postgres_password

networks:
  front-tier:
  back-tier:
```

EXERCISE: WORKSHOP SAMPLE APPLICATION

Work through the 'Workshop Sample Application' exercise in the Docker for Enterprise Developers Exercises book.

FURTHER READING

- Sample applications: http://dockr.ly/2eU89rn
- Dockerize a .NET core application: http://dockr.ly/2jb6GOQ
- Dockerizing sample applications: http://dockr.ly/2jb4B5s

EDIT AND CONTINUE

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Mount a development environment inside a containerized execution environment
- Configure automatic rebuilds and restarts of containerized processes

CONTAINERS ADD FRICTION?

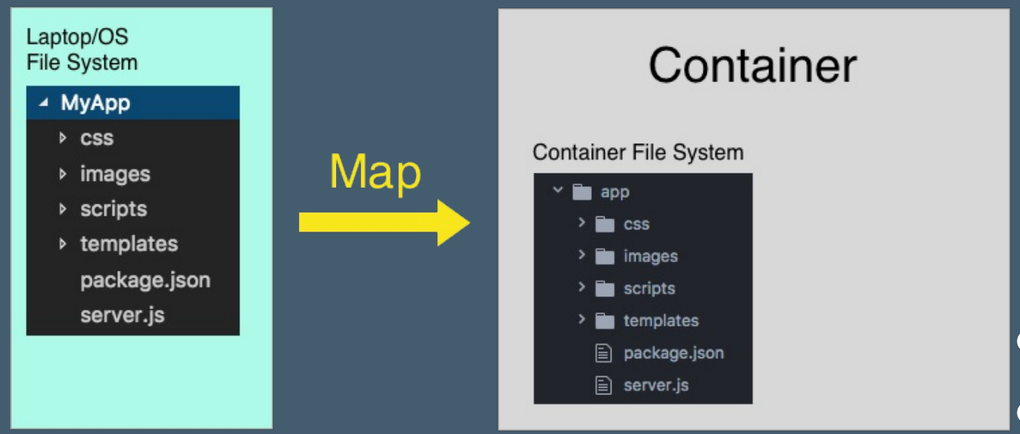
- Use your favorite code editor
- Code lives on host
- Application runs in container

Resulting process:

1. Add or edit code
2. Build the container image
3. Run a container with the new image
4. Test
5. Start over at 1...

MOUNTING CODE

Mounting the source folder into the container:




```
docker container run -v $(pwd):/app ... <image name>
```

Benefits:

- Container always has latest code
- No need to rebuild image

AUTO RESTART APP

Restart web server upon file change...

```
nodemon ./server.js 0.0.0.0 3000
```

Node/Express Application

Package.json

```
{
  "name": "pets",
  "version": "1.0.0",
  "description": "Pets application in Node JS",
  "main": "server.js",
  "scripts": {
    "start-dev": "nodemon ./server.js 0.0.0.0 3000"
  },
  "author": "Gabriel Schenker",
  "license": "ISC",
  "dependencies": {
    "express": "^4.14.1",
    "mustache-express": "^1.2.4"
  }
}
```

EXERCISE: EDIT AND CONTINUE

Work through the 'Edit and Continue' exercise in the Docker for Enterprise Developers Exercises book.

FURTHER READING

- Use volumes: http://dockr.ly/2vRZBDG
- Use bind mounts: http://dockr.ly/2wdstvn

DEBUGGING

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Connect an IDE to a process running in a container.
- Debug containerized processes as if they were remote processes.

DEBUGGING

- We want:
  - Line by line debugging
  - Variable inspection and modification
  - Stack trace
- How to attach an IDE to a containerized process?
- Debugging in container == Remote debugging

DEBUGGING

SUPPORT THROUGH IDE

Language	IDE/Tool	Example
Java	IntelliJ, Eclipse, Netbeans	http://dockr.ly/2rmLhlU, http://bit.ly/2rFoALR
.NET Core	Visual Studio Code	http://bit.ly/2rdVAKX
Node JS	Visual Studio Code	http://dockr.ly/2ac5TVw
	Webstorm	http://bit.ly/2qDcNe1
Ruby	RubyMine	http://bit.ly/2rmPgzd
C++	VisualGDB	http://bit.ly/2rHvcdN
Python/Django	PyCharm	http://bit.ly/2rdFnFg, http://bit.ly/2rn3zDy

EXERCISE: DEBUGGING

Work through the 'Debugging in a Container' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What would you use to keep an eye on the metrics of your container? Why?
- What would you do if you wanted to debug why your container won't start at all (e.g.: initial command or entrypoint that immediately crashes)?
- Questions?

FURTHER READING

- Live debugging with Docker: http://dockr.ly/2ac5TVw
- Live debugging Java with Docker: http://dockr.ly/2rmLhlU

DOCKER COMPOSE

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Describe and deploy a multi-service app using Docker Compose
- Capture configuration differences in environment-specific compose files

DOCKER COMPOSE

Define multiple Docker Compose files

- For development (front- vs. back-end)
- Running unit- and/or integration tests
- Running end-to-end tests
- Build artifacts on CI server
- Run tests on CI server

BACKEND DEVELOPMENT

```
version: '3.1'
services:
    api:
        build:
            context: api
            dockerfile: Dockerfile.dev
        ports:
            - 8080:8080
        volumes:
            - ./api:/usr/src/pets-api
        networks:
            - pets
    watcher:
        build:
            context: api
            dockerfile: Dockerfile.dev
        volumes:
            - ./api:/usr/src/pets-api
        command: mvn fizzed-watcher:run
        networks:
            - pets
networks:
    pets:
```

- Development specific Dockerfile
- Mounting code into container
- Opening port to access via e.g. curl
- Watcher container to auto-compile

FRONTEND DEVELOPMENT

```
version: '3.1'
services:
  api:
    image: <username>/pets-api:1.4
    networks:
      - pets
  ui:
    build:
      context: ui
      dockerfile: Dockerfile-dev
    ports:
      - 3000:3000
    volumes:
      - ./ui/src:/app/src
      - ./ui/package.json:/app/package.json
    command: nodemon src/server.js 0.0.0.0 3000
    networks:
      - pets
networks:
  pets:
```

- Use specific version of backend image
- Use development specific Dockerfile
- Mount UI code into container
- Use auto restart in case of change

EXERCISE: DOCKER COMPOSE

Work through the 'Docker Compose' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What are some pros and cons of having different compose files for different environments?
- Questions?

FURTHER READING

- Docker compose reference: http://dockr.ly/2iHUpeX
- Overview of Docker compose: http://dockr.ly/2jh9kjV

TESTING

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Create unit, API, stress and integration tests appropriate for containerized applications.

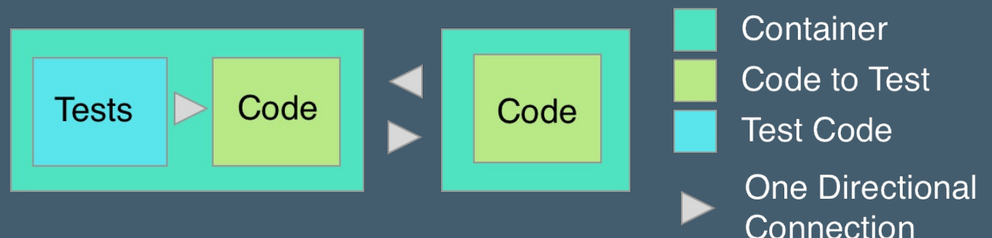
CATEGORIES & CHARACTERISTICS OF TESTS

- Unit Tests
- Integration Tests
- API Tests
- Stress and Load Tests
- End-to-End Tests

UNIT TESTS

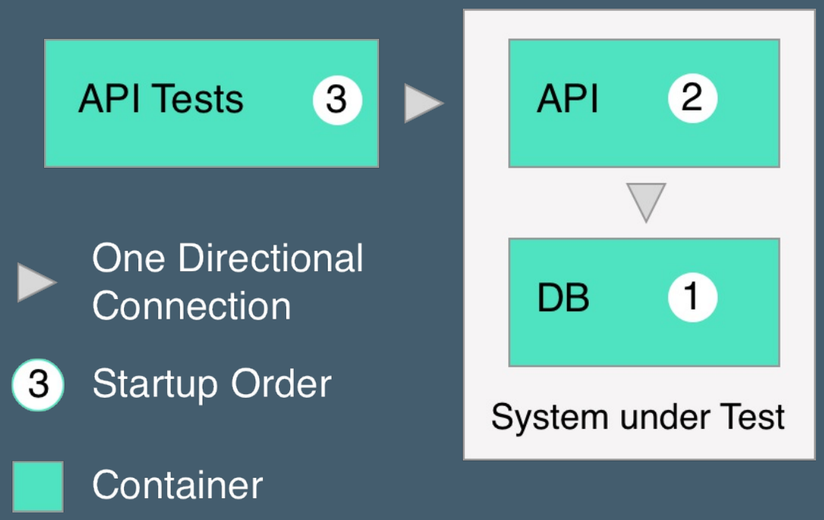
- Test a small & isolated piece of code
- All external dependencies are mocked
- Test code is tightly coupled with SUT
- Test code and SUT run in same container

INTEGRATION TESTS



- Test & validate multiple interacting components
- Some external dependencies are mocked
- Test code is tightly coupled with SUT
- Test code and SUT run in same container

API TESTS

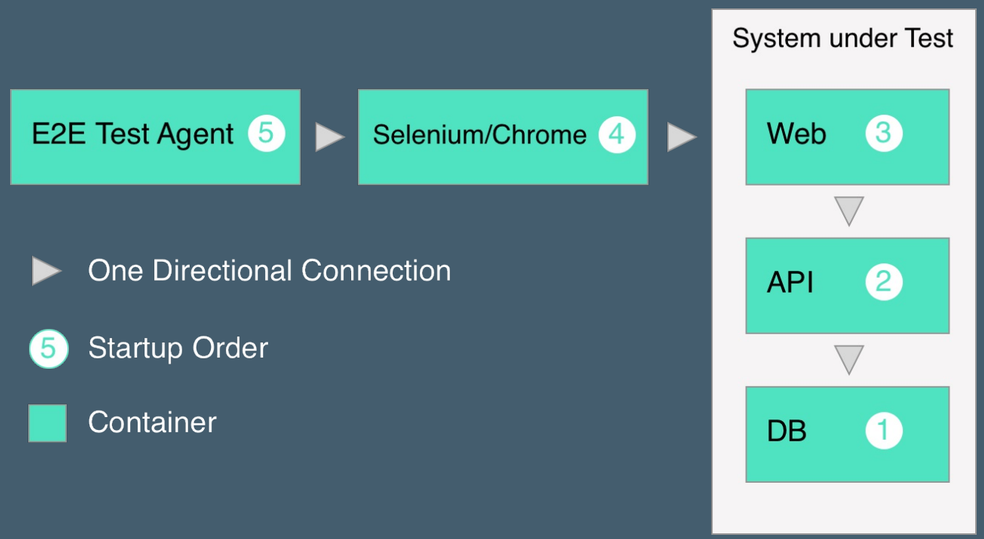


- Test & validate service via public API
- Some (expensive) external dependencies can be mocked
- Test code is loosley coupled with SUT
- Test code and SUT run in different containers

STRESS & LOAD TESTS

- Test code under load
- Use production like environments, data quantities and concurrent connections
- Test code is loosley coupled with SUT
- Test code and SUT run in different containers

END-TO-END TESTS



- Test whole application end-2-end
- Use production like environments, data quantities and concurrent connections
- Test code automates UI
- Test code and SUT run in different containers

EXERCISE: UNIT, API & E2E TESTS

Work through the exercises

- Unit Tests
- API Tests
- End-to-End Tests

in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What new tests become more important after refactoring a monolithic application into multiple containerized services?
- Questions?

FURTHER READING

- Continuous integration testing with Docker: http://dockr.ly/2xSlLXM
- Scale testing Docker swarm to 30000 containers: http://dockr.ly/2smWgvf

SERVICE DISCOVERY

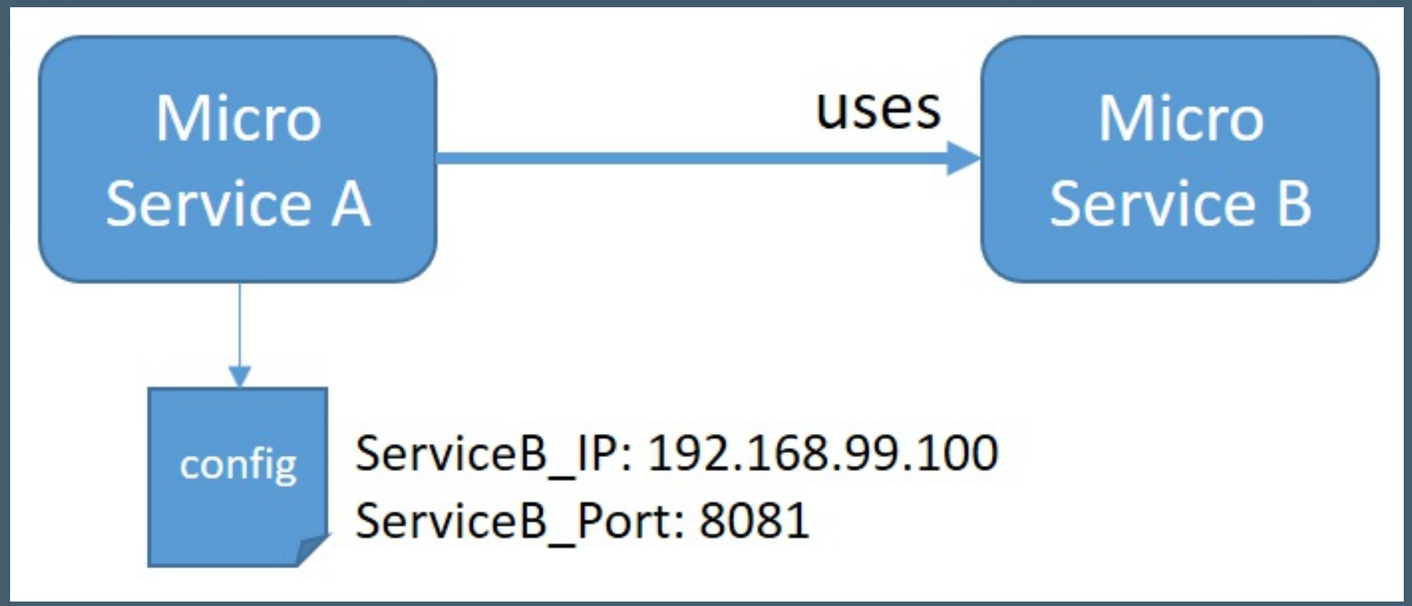
LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Route traffic between containers using orchestrator-driven service discovery
- Compare and contrast service discovery in Swarm and Kubernetes

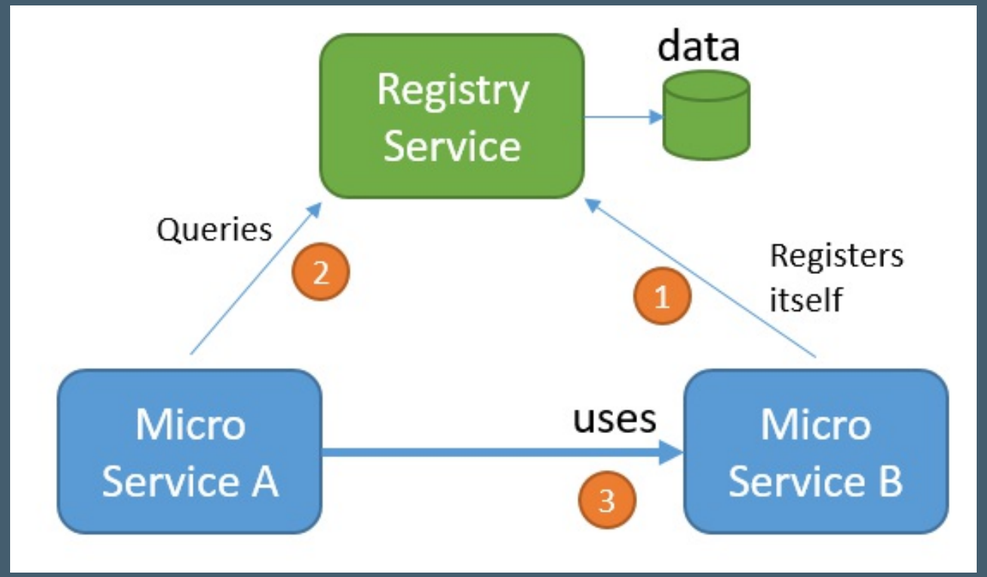
SERVICE DISCOVERY

Bad: Hard Wiring...



SERVICE DISCOVERY

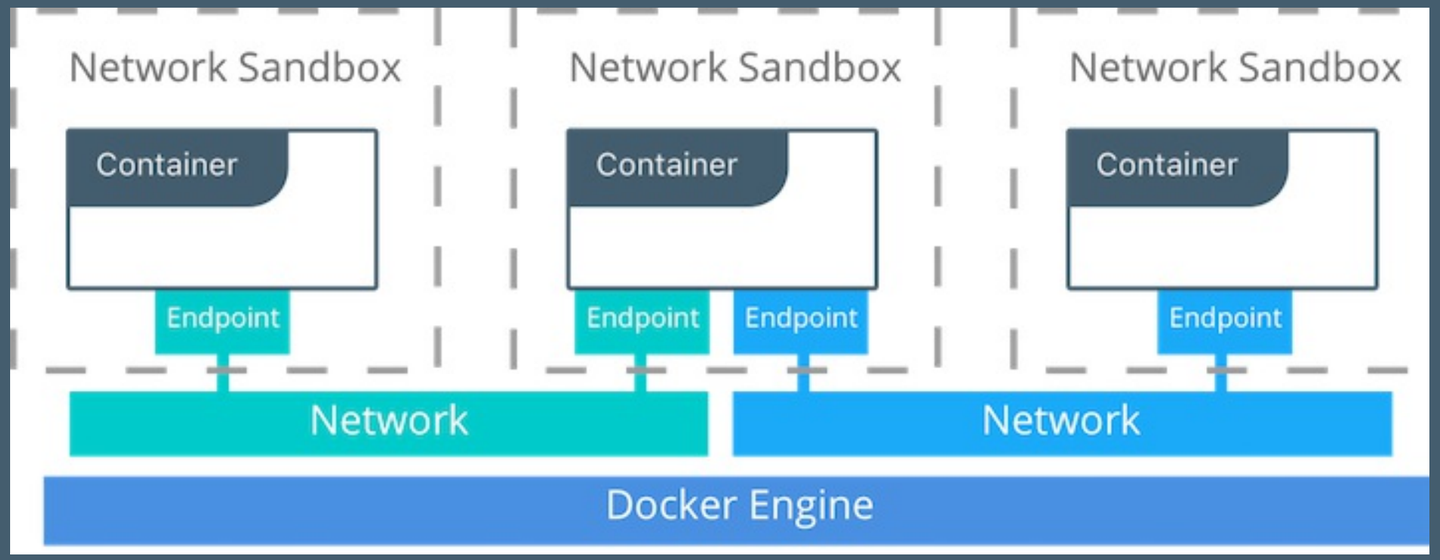
Good: Registry Service



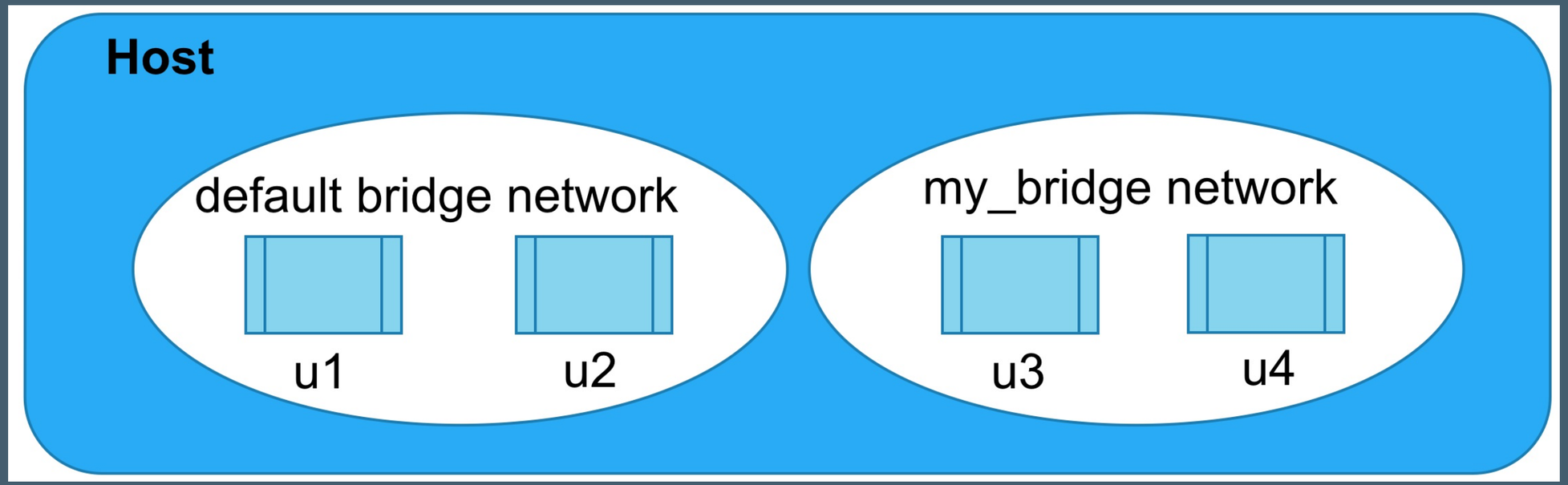
- Docker provides embedded DNS for container and service names
- Address traffic to services by name

DOCKER SWARM

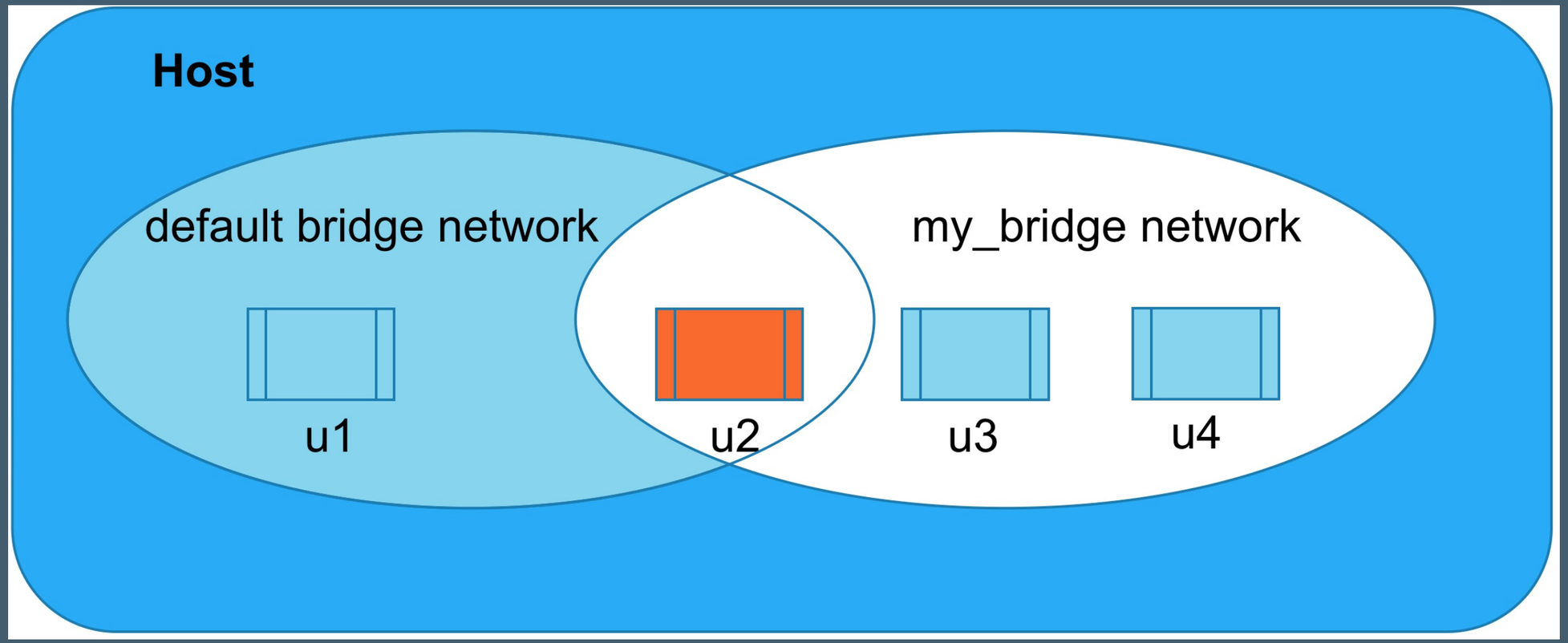
CONTAINER NETWORK MODEL



DOCKER SWARM: NETWORK FIREWALLS

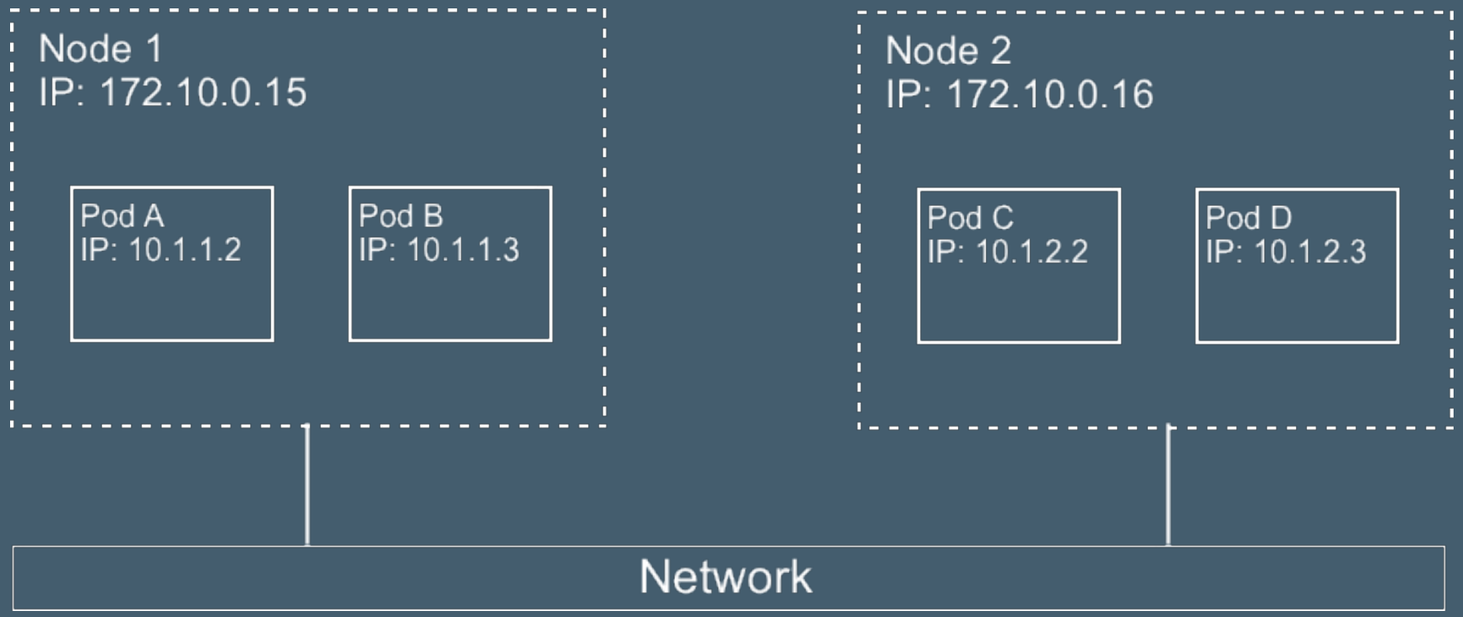


DOCKER SWARM: NETWORK FIREWALLS




```
docker network connect my_bridge u2
```

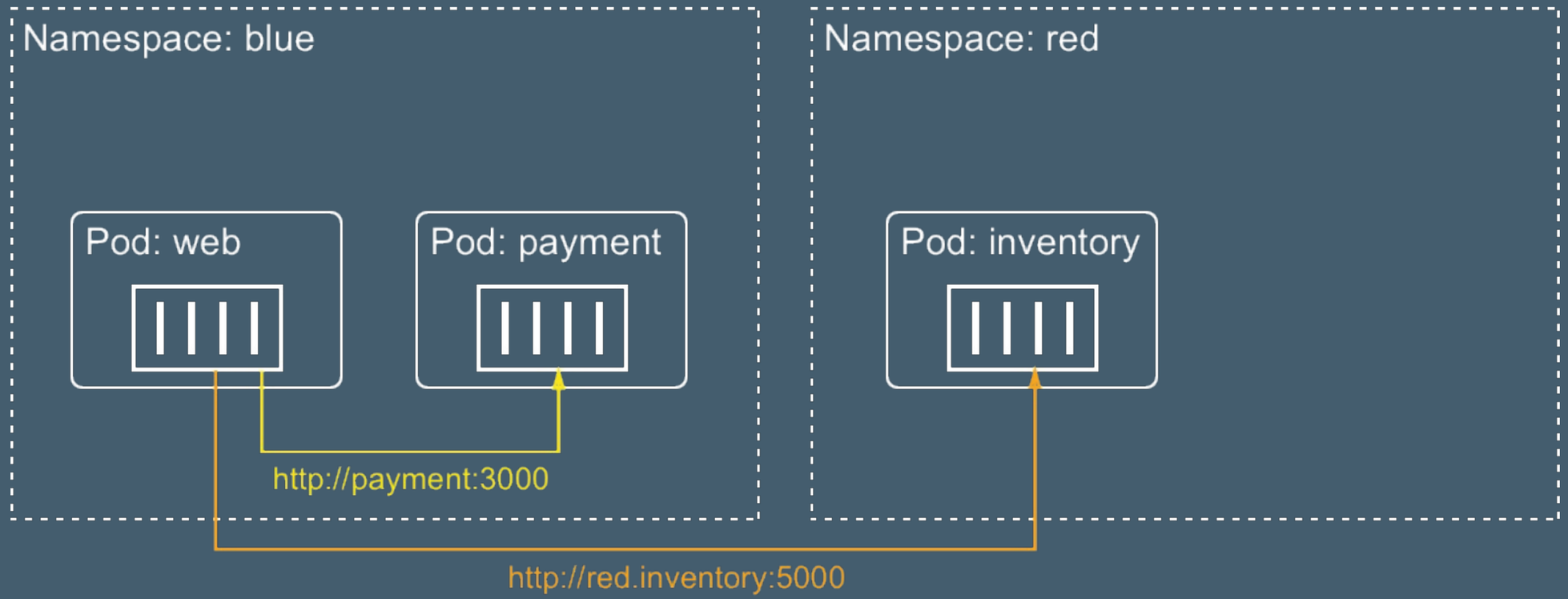
KUBERNETES: NETWORKING



Requirements

- Pod <--> Pod without NAT
- Node <--> Pod without NAT
- Pod's peers find it at the same IP it finds itself

KUBERNETES: NAMESPACES



EXERCISE: SERVICE DISCOVERY

Work through the 'Service Discovery' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What are some pros and cons of orchestrator-based service discovery?
- Questions?

FURTHER READING

- Service discovery and links: http://dockr.ly/2gQIctq
- Manage swarm service networks: http://dockr.ly/2gQjG82
- Docker Reference Architecture: UCP 2.0 Service Discovery and Load Balancing: http://dockr.ly/2rbxDDX

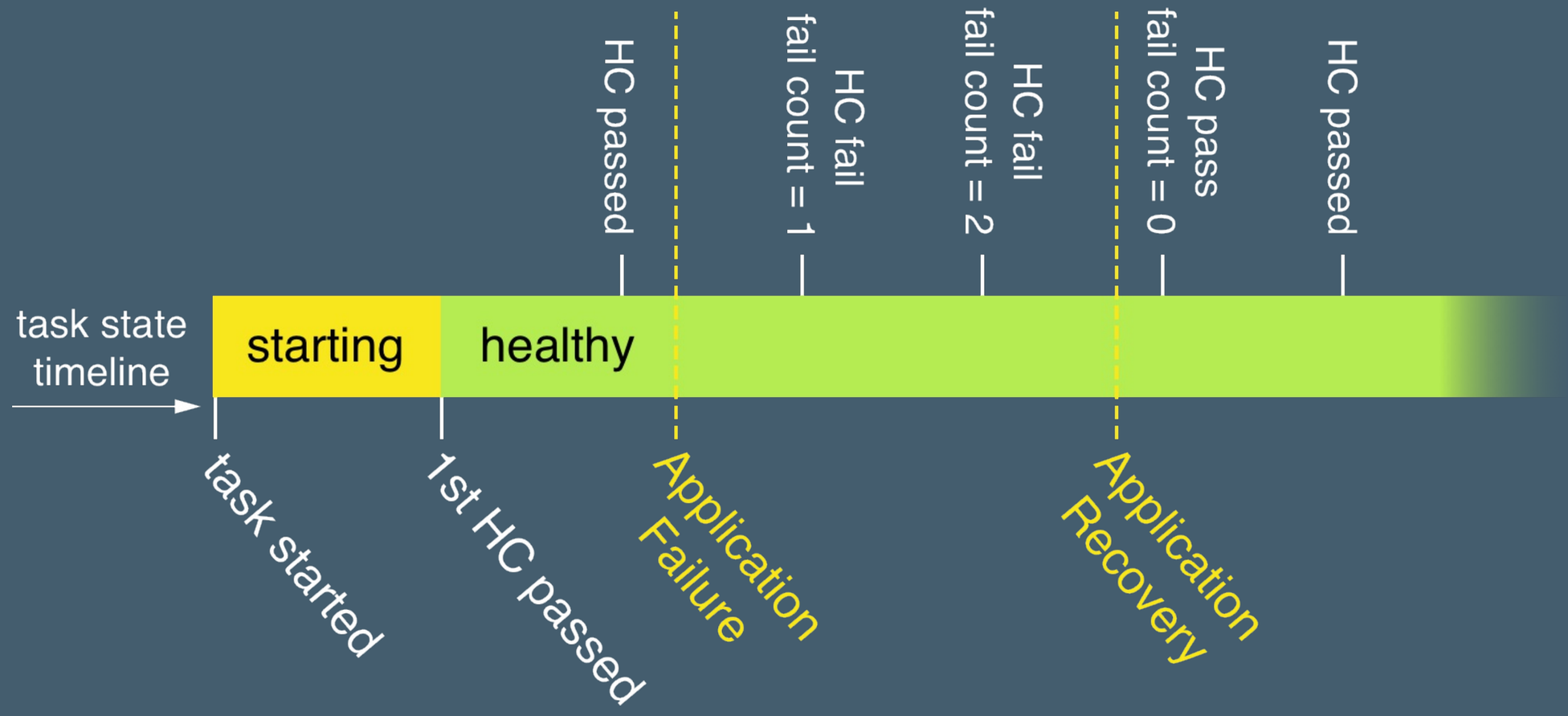
HEALTH CHECKS

LEARNING OBJECTIVES

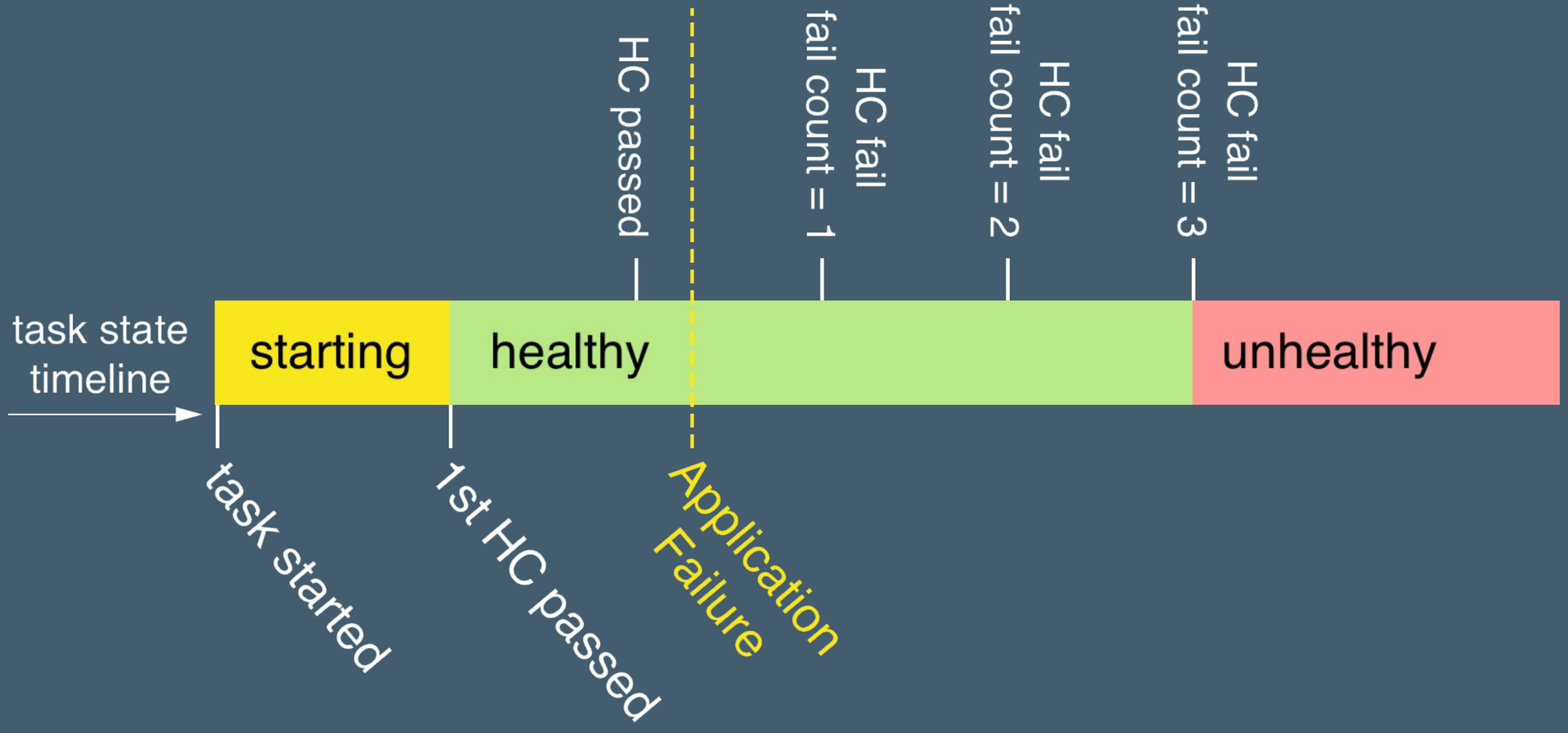
By the end of this module, learners will be able to:

- Implement healthcheck logic in application code for integration into Swarm and Kubernetes' healthcheck protocols
- Configure automated healthchecks in Swarm and Kubernetes

HEALTH CHECK LIFECYCLE - HEALTHY



HEALTH CHECK LIFECYCLE - UNHEALTHY



APPLICATION HEALTH CHECKS

- Orchestration engine periodically runs healthcheck command in service container
- Too many failures -> container/pod killed and rescheduled
- Healthcheck logic supplied by developers
- healthcheck exit 0: healthy
- healthcheck exit 1: unhealthy

IMPLEMENT HEALTH ENDPOINT

/health [GET] endpoint

- Python/Flask

```
@app.route("/health", methods=['GET'])
def get_health():
    if healthy == True:
        return 'OK', 200
    else:
        return 'NOT OK', 500
```

- Node JS/Restify

```
server.get('/health', function create(req, res, next) {
  if(healthy){
    res.send(200, 'OK');
  } else {
    res.send(500, 'NOT OK');
  }
  return next();
});
```

Alternatives: https://github.com/docker-library/healthcheck

DOCKER SWARM: HOW TO CHECK HEALTH

- Dockerfile

```
HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit 1
```

Command should return 0 for healthy, 1 for unhealthy

- Docker Compose File

```
healthcheck:
    interval: 10s
    timeout: 2s
    retries: 3
```

KUBERNETES: HOW TO CHECK HEALTH

Liveness Probe:

```
kind: Pod
...
spec:
  containers:
  - name: demo
    image: ...
    livenessProbe:
      periodSeconds: 10
      timeoutSeconds: 2
      failureThreshold: 3
      initialDelaySeconds: 30
      successThreshold: 1
      exec:
        command:
        - cat
        - /tmp/healthy
    ...
```

Readiness Probe:

```
kind: Pod
    ...
spec:
  containers:
  - name: demo
    image: ...
    readinessProbe:
      periodSeconds: 60
      timeoutSeconds: 2
      ...
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
    ...
```

EXERCISE: HEALTH CHECKS

Work through the Health Checks exercise in your Docker for Enterprise Developers Exercises book.

DISCUSSION

- Healthcheck endpoints must provide a measure of application health - but what other related logic might you want to implement around your healthcheck?
- Questions?

FURTHER READING

- Dockerfile reference: http://dockr.ly/1PallBK

DEFENSIVE PROGRAMMING

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Identify hazards posed by container orchestration that need to be handled effectively
- Write defensive logic that handles microservice outages
- Define and implement circuit breaking patterns in containers

DEFENSIVE PROGRAMMING

- SIGTERM versus SIGKILL
- Availability of external service not guaranteed
- Stale data vs. no data
- Fail fast
- Circuit breakers

SIGTERM VERSUS SIGKILL

- SIGTERM and SIGKILL issued by orchestration engine
- Stateless services are best
- Make failure recoverable:
  - Transactions
  - Compensating actions
  - Idempotency

EXTERNAL SERVICE NOT AVAILABLE

- Distributed services live and die independently
- Docker DNS propagation eventually consistent
- Retry
- Increase wait time between calls
- Log warning
- Also see: Circuit Breaker

STALE DATA VERSUS NO DATA

If external service is not available:

- Fail if critical data
- Serve stale data if freshness is less important
- Serve no data if freshness is important

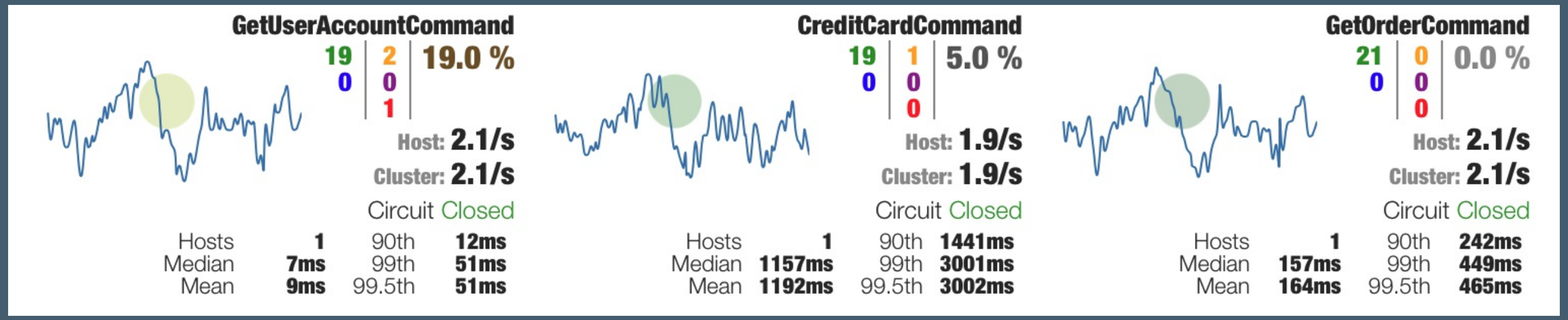
FAIL FAST

- Always check input data first
- Log clear failure reason
- Yet, do not disclose too much
- Take advantage of service rescheduling

CIRCUIT BREAKERS

- Service separation facilitates damage control
- Avoid Domino Effect
- Give failed service/system time to recover
- Gracefully degrade experience
- Serve stale (or no) data instead

Netflix Hystrix



EXERCISE: DEFENSIVE PROGRAMMING

Work through the 'Defensive Programming' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What are some container-specific situations that should be defended against?
- Questions?

FURTHER READING

- Controlling startup order in Compose: http://dockr.ly/2vTPjYE
- Trapping signals in Docker containers: http://bit.ly/2wZVKZK
- Netflix Hystrix: https://github.com/Netflix/Hystrix
- Managing Microservices with the Istio Service Mesh: http://bit.ly/2nJjpYH

LOGGING & ERROR HANDLING

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Instrument a containerized application with logging output
- Configure the Docker daemon's logging driver
- Name principles of error handling fit for orchestrated applications

LOGGING

Why does Logging matter?

- Highly distributed environment
- Many moving parts
- High probability of failure

Why is central Logging crucial?

- Correlate events
- Slice and dice
- No need to access hosts
- No need to backup logs
- Improved security

LOGGING

Where and What to log?

- Send logging info to StdOut and StdErr
- Add severity (debug, info, warn, err, fatal)
- Add service relevant unique identifier
- Do NOT add sensitive or personal information!

LOGGING

Python

```
from flask import Flask
app = Flask(__name__)
app.logger.setLevel(logging.DEBUG)
...
@app.route('/warning')
def warning():
    app.logger.warning('This is a warning message')
    return "warning"
```

Java

```
public class main{
    static Logger logger = LogManager.getLogger("main");

    public static void main(String[] args) {
        logger.debug("Hello this is a debug message");
        logger.info("Hello this is an info message");
    }
}
```

DOCKER EE LOGGING

- Logs come from three sources:
  - container runtime: system-level
  - kubelet: system-level
  - containers: STDOUT & STDERR
- Globally configured by /etc/docker/daemon.json:

```
{
   "log-driver": "journald",
   "log-level": "debug",
   "log-opts": {
      "tag":"{{.ImageName}}/{{.Name}}/{{.ID}}"
   }
}
```

- Override for containers with --log-opt

CENTRALIZED LOGGING - ELK STACK


```
version: "3"
services:
    elasticsearch:
        image: elasticsearch:2

    logstash:
        image: logstash
        command: |
        -e '
            input {...}
            filter {...}
            output {...}'

        ports:
        - "12201:12201/udp"

    kibana:
        image: kibana:4
        ports:
            - "5601:5601"
        environment:
            ELASTICSEARCH_URL: http://elasticsearch:9200
```

ERROR HANDLING

- Fail fast!
- Provide good quality error messages
  - reason for failure
  - app / service specific unique ID
  - stack trace
- Use defensive coding style
- Sometimes stale results are better than no results
- Implement health endpoint

EXERCISE: LOGGING & ERROR HANDLING

Work through the exercises

- Logging
- Error Handling

in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What additional logging information should your application provide after containerization?
- Questions?

FURTHER READING

- Configure logging drivers: http://dockr.ly/2gSFMdS
- Kubernetes Logging Architecture: http://bit.ly/2veExYy

BUILDER

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- List several Dockerfile best practices concisely
- Author a multi-stage Dockerfile for an application to minimize the final image size

IMAGE SIZE MATTERS

Image	Size (MB)
java:8-jdk	643
java:8-jdk-alpine	145
java:8-jre	311
java:8-jre-alpine	108
node:7.9	665
node:7.9-alpine	59
python:2.7	676
python:2.7-alpine	72
golang:1.7	676
golang:1.7-alpine	241
microsoft/windowsservercore	~9,000
microsoft/nanoserver	~450
busybox	1.1

Smaller images =

- smaller attack surface
- faster download & build times
- often faster container startup times

OPTIMIZE THE DOCKERFILE

Rely on image layer cache

Seldom changing --> beginning of Dockerfile

Frequently changing --> end of Dockerfile

```
FROM node:8-jdk-alpine
RUN yum install <somelibrary>
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
EXPOSE 3000
CMD npm start
```

MULTI STAGE BUILD

Multiple FROM statements per Dockerfile

Only last FROM results in an image

All layers are cached

```
FROM ubuntu AS build-env
RUN apt-get install make
ADD . /src
RUN cd /src && make

FROM busybox
COPY --from=build-env /src/build/app /usr/local/bin/app
EXPOSE 80
ENTRYPOINT /usr/local/bin/app
```

EXERCISE: BUILDER

Work through the 'Builder' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What are the tradeoffs between more versus fewer image layers?
- Questions?

FURTHER READING

- Best practices for writing Dockerfiles: http://dockr.ly/22WiJiO
- Use multistage builds: http://dockr.ly/2ewcUY3
- Containers and layers: http://dockr.ly/2vOwbap

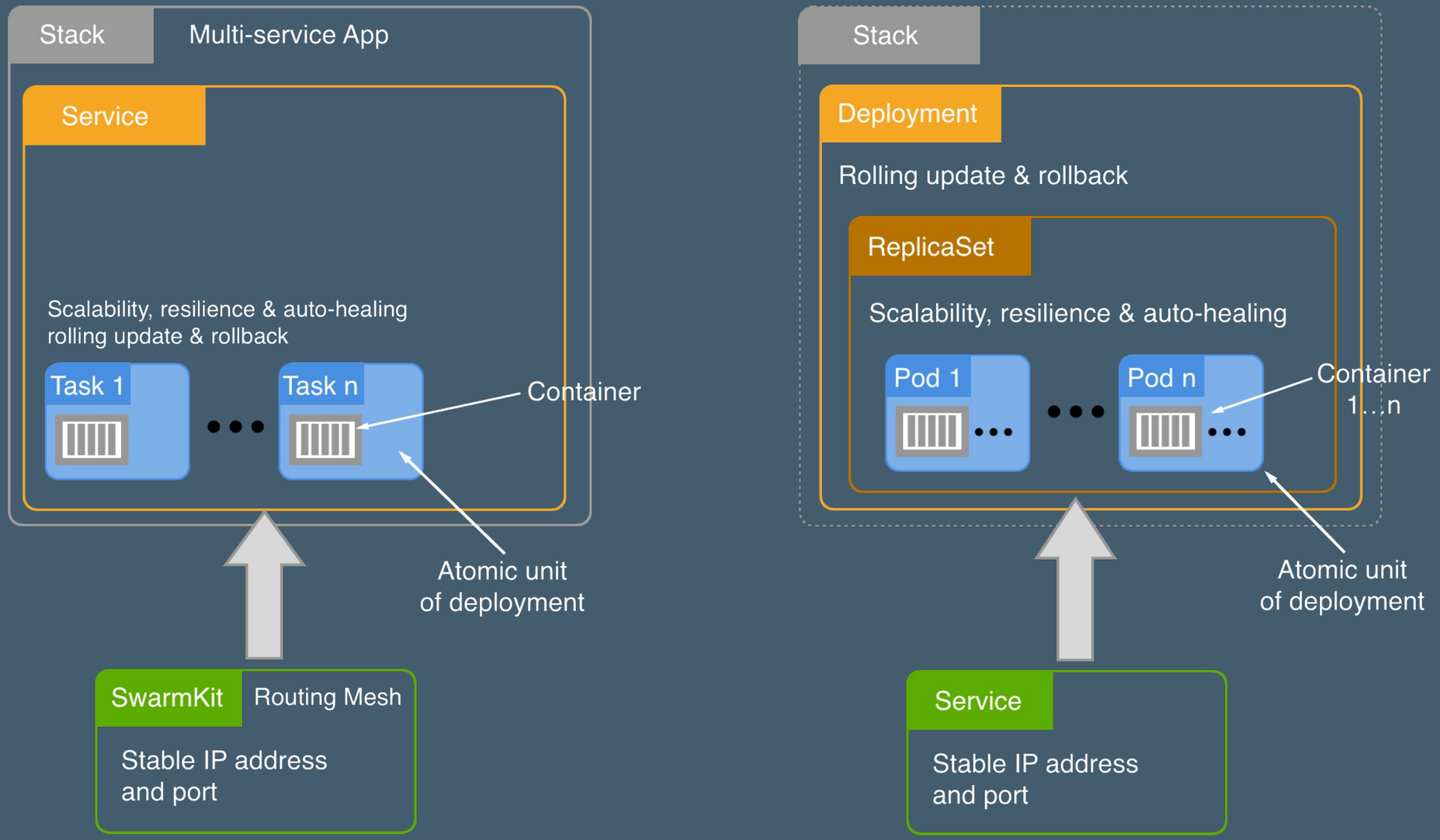
DOCKER SWARM & KUBERNETES

LEARNING OBJECTIVES

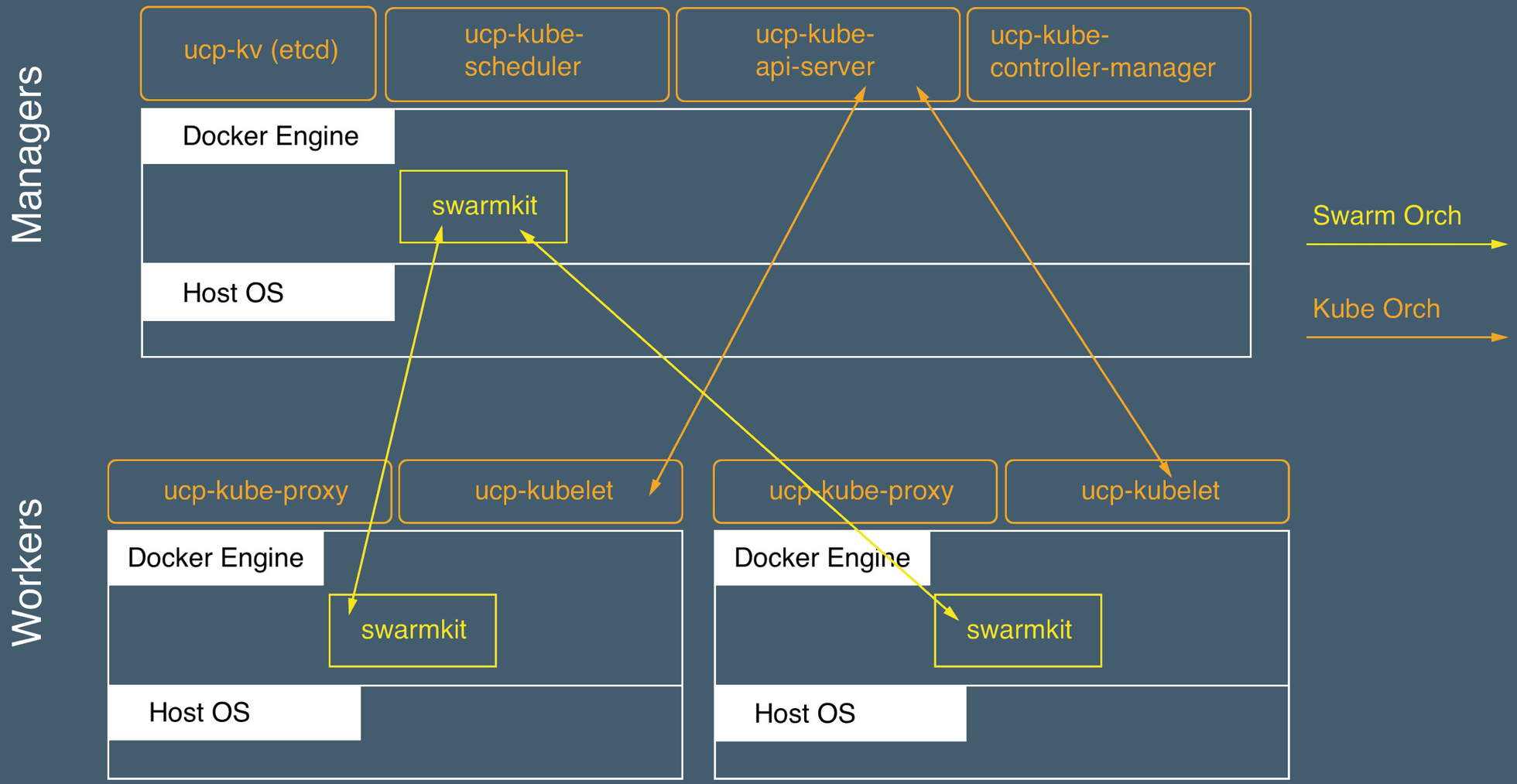
By the end of this module, learners will be able to:

- Draw correct analogies between fundamental Swarm and Kubernetes objects
- Understand how UCP manages both orchestrators
- Implement layer 4 routing to applications in both orchestrators

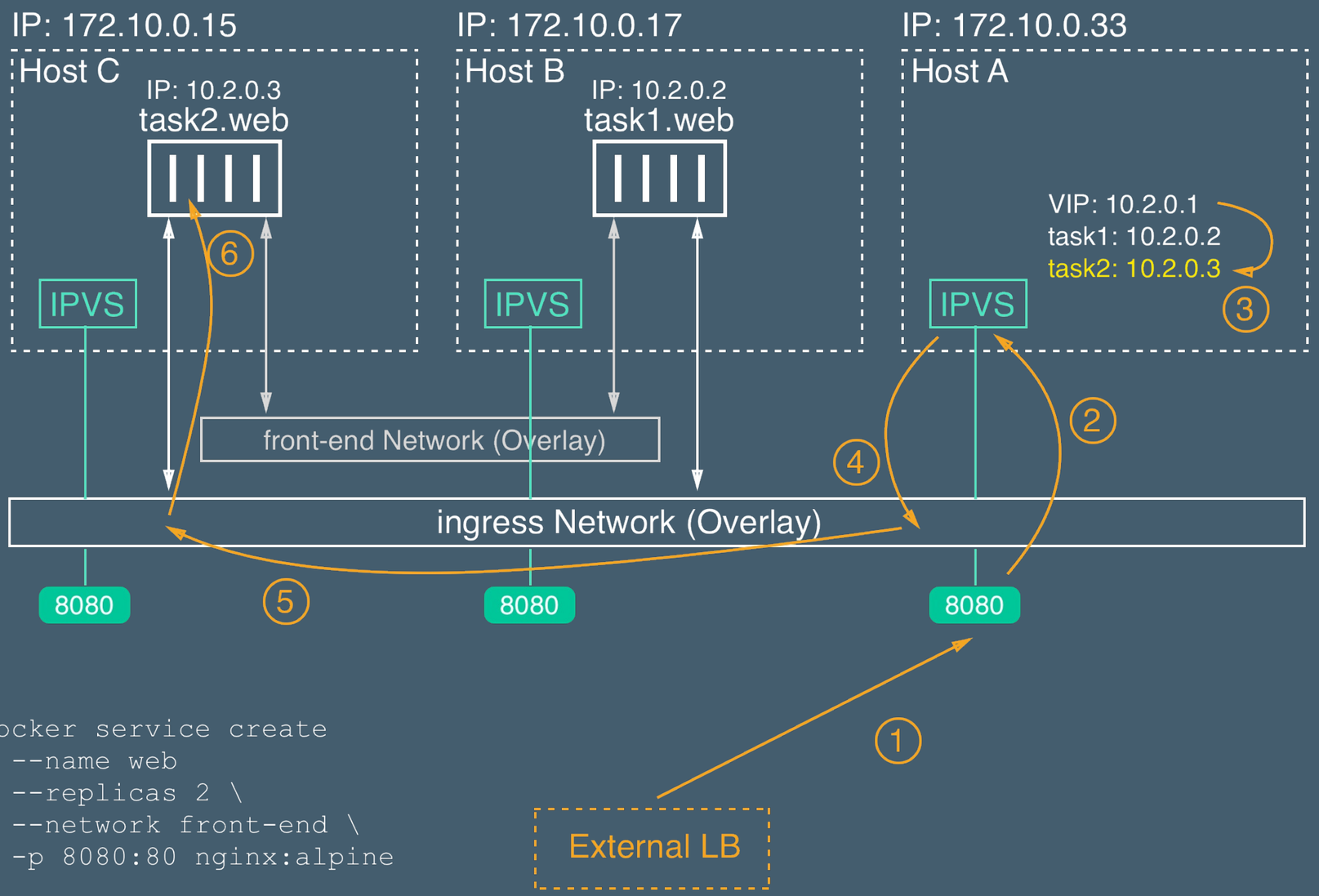
SWARM VERSUS KUBERNETES



ORCHESTRATORS IN UCP

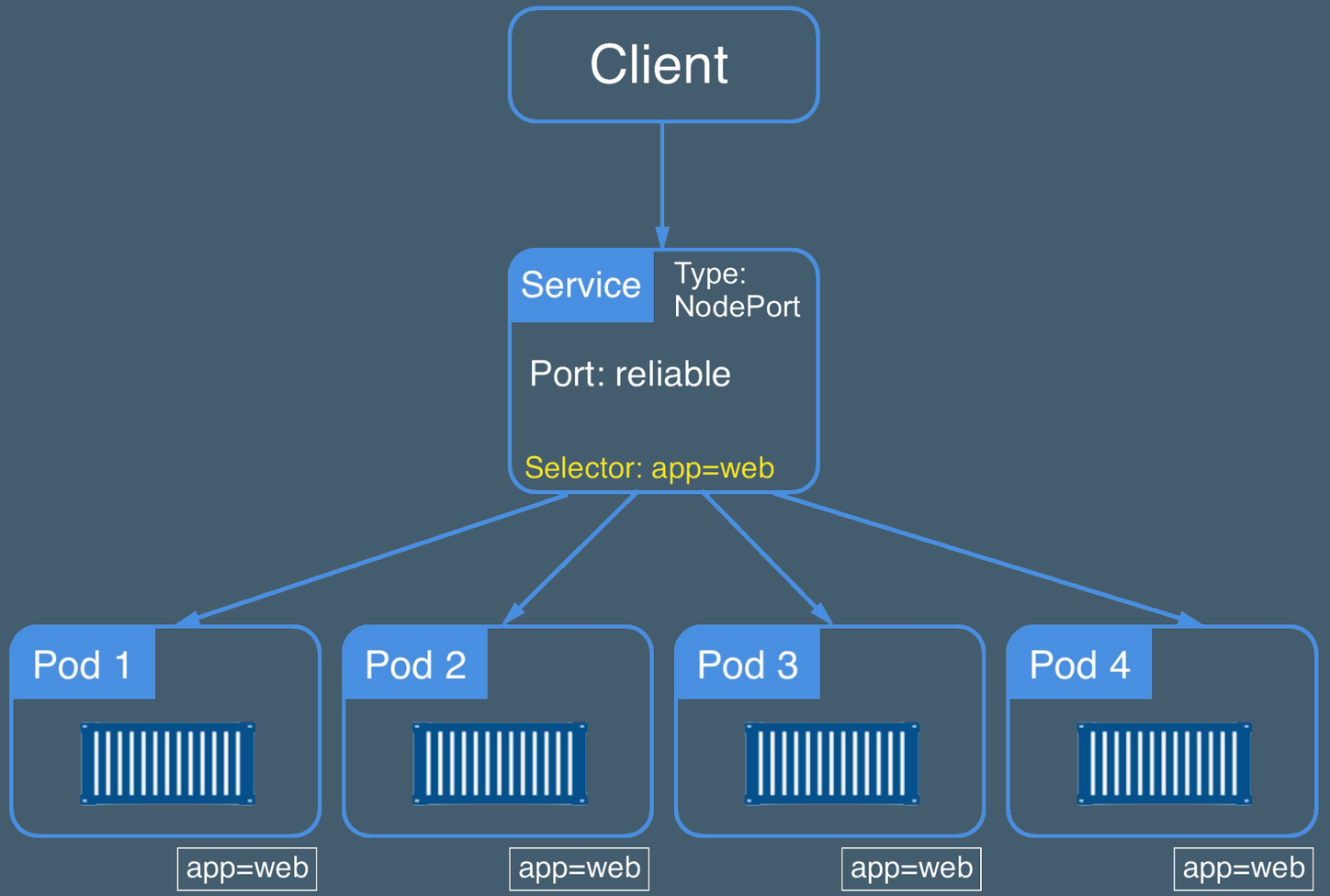


SWARM ROUTING MESH




```
docker service create
  --name web
  --replicas 2 \
  --network front-end \
  -p 8080:80 nginx:alpine
```

L4 ROUTING - KUBERNETES: SERVICE OBJECT



EXERCISE: DOCKER SWARM

Work through the exercises:

- Creating a Docker Swarm
- Routing Mesh

in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What sort of applications would you prefer to design for Swarm? Kubernetes?
- Questions?

FURTHER READING

- Swarm mode overview: http://dockr.ly/2jh11Vd
- Swarm mode key concepts: http://dockr.ly/2jgMT0f
- Getting started with swarm mode: http://dockr.ly/2vYFNU4
- High availability in Docker swarm: http://dockr.ly/2gVH6MS

SECRETS

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Use secrets in Swarm and Kubernetes
- Manage and mock secrets across environments
- Identify the encryption strategies used to protect secrets throughout their lifecycle

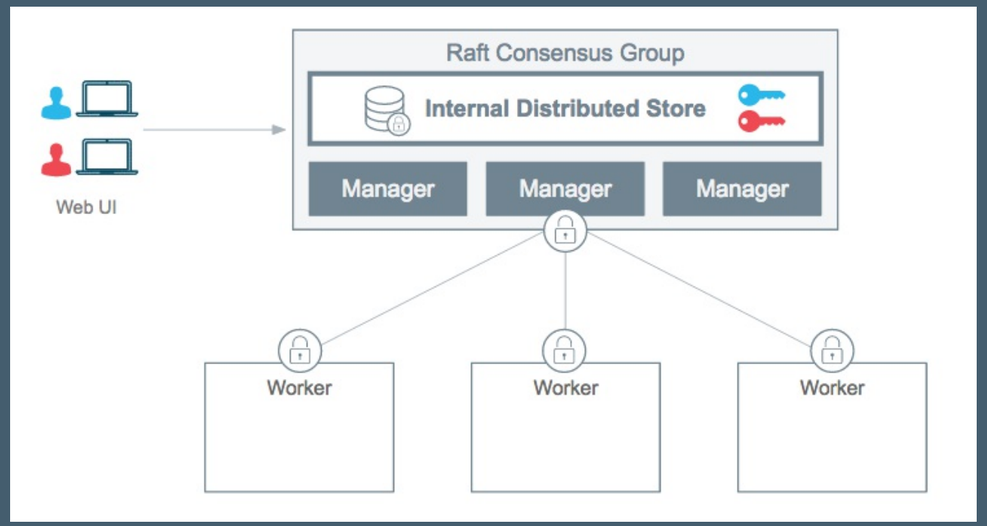
DISTRIBUTING SECRETS

- Apps need sensitive data like access tokens, ssh keys etc
- How to securely store and distribute secrets across a cluster?

SECRET REQUIREMENTS

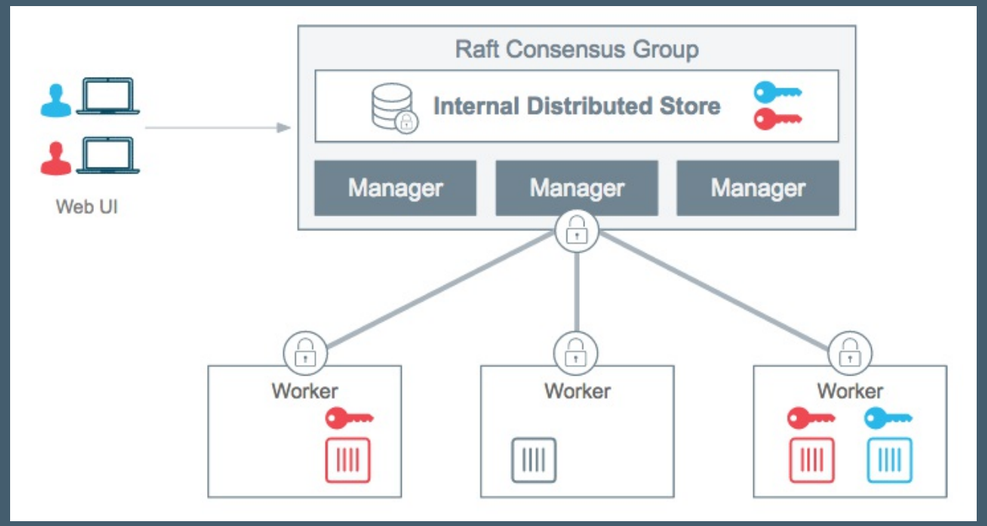
- Encrypted at rest
- Encrypted in transit
- Accessible only to intended containers

SECRETS WORKFLOW 1: CREATION & STORAGE



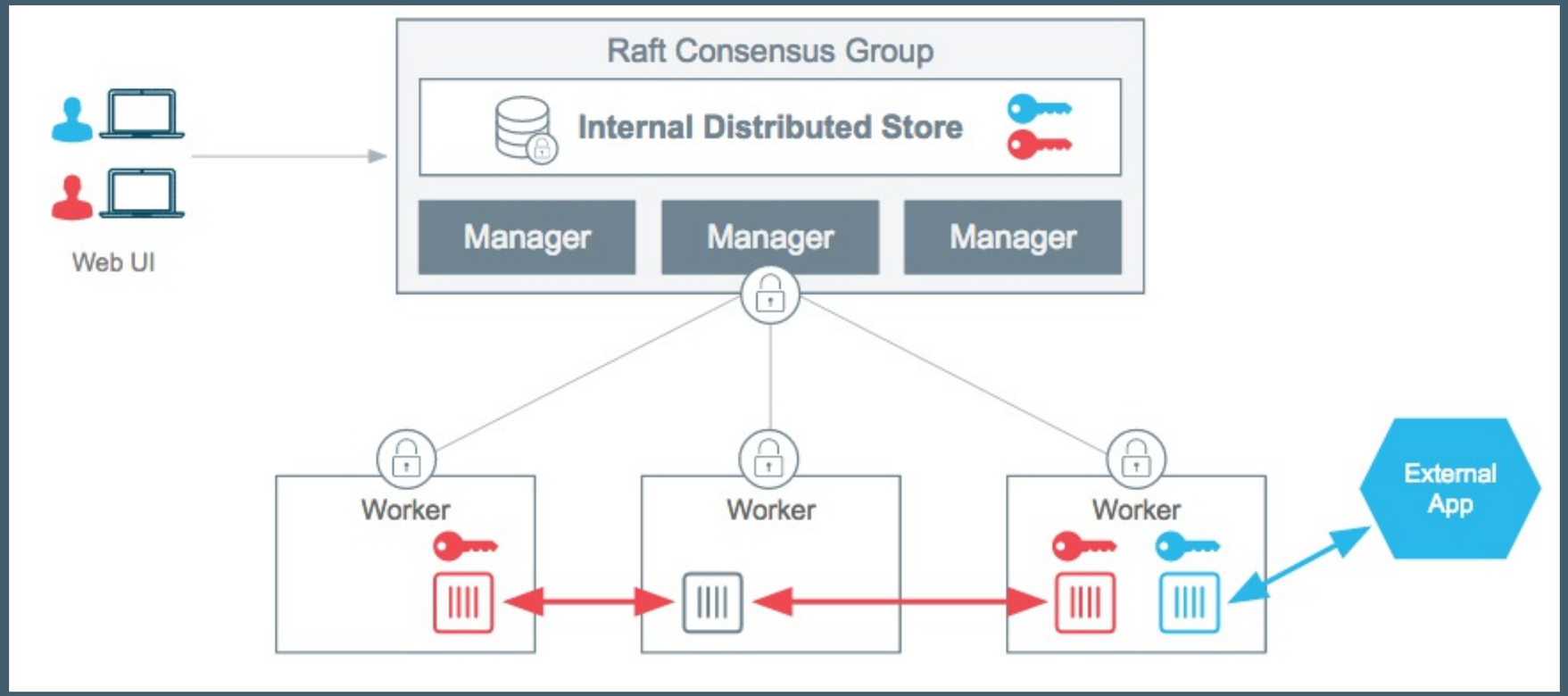
- Swarm:
  - Stored in state db
  - Automatically encrypted at rest
- Kubernetes:
  - Stored in etcd
  - Must configure encryption: http://bit.ly/2GCoPwE

SECRETS WORKFLOW 2: DISTRIBUTION



- Swarm:
  - Default mutual TLS
  - Associated with services
- Kubernetes:
  - TLS optional
  - Associated with pods or deployments
- Both:
  - Least Privilege: secrets available only exactly where needed
  - Deleted along with container

SECRETS WORKFLOW 3: SECRET USAGE



SECRETS IN DEVELOPMENT

Swarm: Service

```
docker service create --name foo \
    --secret source=DB_PASSWORD_V1,target=DB_PASSWORD \
    --env DB_PASSWORD_FILE=/run/secrets/DB_PASSWORD \
    <IMAGE_NAME>
```

Development: Container

```
docker container run --name foo \
    -v ./dev-secrets/DEV_DB_PASSWORD:/app/my-secrets/DB_PASSWORD \
    -e DB_PASSWORD_FILE=/app/my-secrets/DB_PASSWORD \
    ...
    <IMAGE_NAME>
```

Code (Node JS)

```
var fs = require('fs');
var db_password = fs.readFileSync(process.env.DB_PASSWORD_FILE, 'utf8');
...
```

SECRETS

Work through the 'Secrets' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- Secrets are one way to inject config into a container. What are some others, and when should each be used?
- Questions?

FURTHER READING

- Manage sensitive data with Docker secrets: http://dockr.ly/2vUNbuH
- Docker secret reference: http://dockr.ly/2iSsNJC
- Introducing Docker secrets management: http://dockr.ly/2k7zwzE
- Securing the AtSea app with Docker secrets: http://dockr.ly/2wx5MyV
- Manage secrets: https://dockr.ly/2HmznE3
- Kubernetes Secrets: http://bit.ly/2C6hMZF

CONFIGURATION MANAGEMENT

LEARNING OBJECTIVES

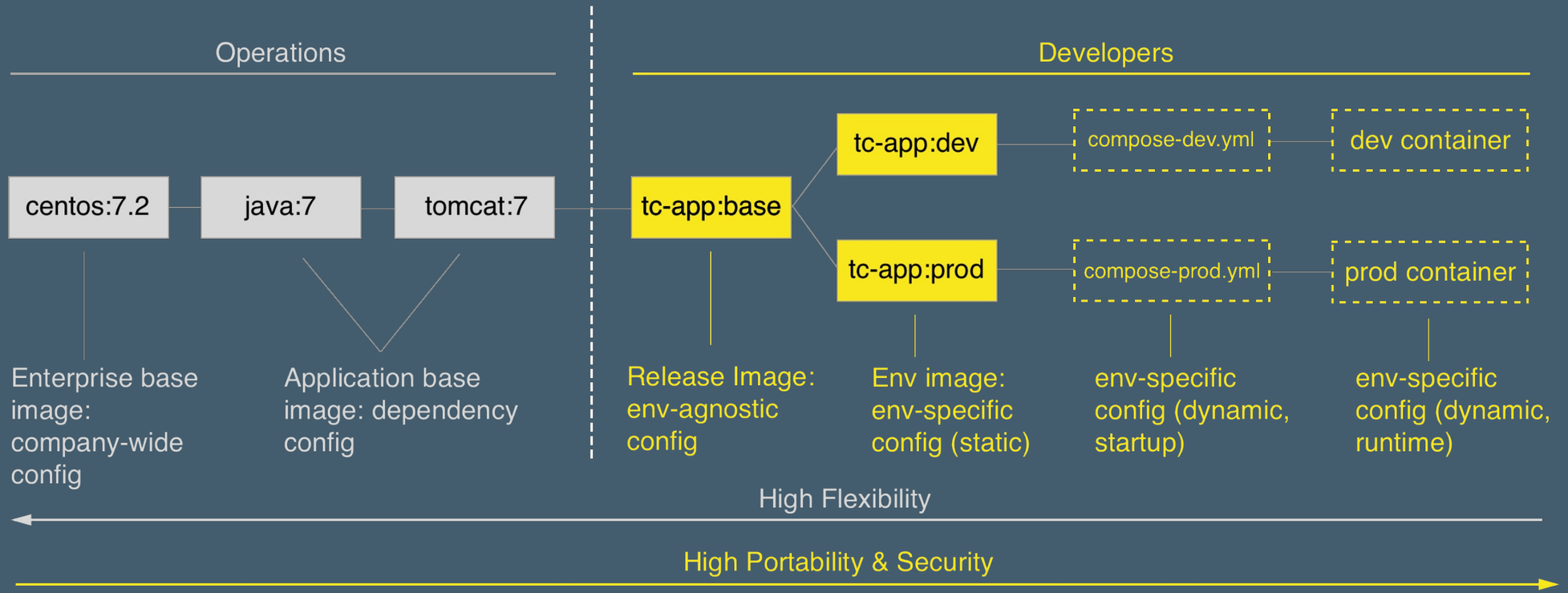
By the end of this module, learners will be able to:

- List a few places in a containerized enterprise application to put configuration information
- Categorize the configuration data type usually used in enterprise applications
- Explain and justify where to put sensitive configuration data

WHERE SHOULD MY CONFIG LIVE?

- Base image
- Derived image
- Stack or Object YAML files
- parameter file in image
- parameter file mounted externally
- entrypoint script
- external service

CONFIGURATION BUCKETS



ENTERPRISE BASE IMAGE

- Provided by Ops (CaaS)
- Start with Official Image
- Add company wide Tools & Utilities
- Push to DTR
- Scan image --> zero vulnerabilities
- Mind Image Size!

```
FROM centos:7
RUN yum -y --noplugins install bzip2 tar sudo curl net-tools
```

APPLICATION BASE IMAGE

- Define as Dockerfile
- Start with Enterprise Base Image
- Add App Type specific Tools & Utilities
- Mind Image Size!

```
FROM <DTR_FQDN>/<ORG_Name>/enterprise-base:1.5
COPY files/dynatrace-agent-6.1.0.7880-unix.jar /opt/dynatrace/
...
```

RELEASE IMAGE

- Define as Dockerfile
- Start with Application Base Image
- Add Application Artifacts
- Add environment-agnostic config
- Mind Image Size!

```
FROM <DTR_FQDN>/<ORG_Name>/app-base:2.2
COPY files/MY_APP_1.3.1-M24_1.war /opt/jboss/standalone/deployments/
...
```

ENVIRONMENT IMAGE

- Define as Dockerfile
- Start with Release Image
- Add environment-specific config
- Mind Image Size!
- Could alternatively mount config at runtime

DYNAMIC CONFIG: STARTUP

- Config defined at startup
- provided in stack or object YAML files
  - environment variables
  - Docker EE-managed secrets
  - Volume mounts
- or defined in entrypoint.sh
  - Hashicorp Vault-managed secret
  -

```
$ curl -H "X-Vault-Token: f3b09679-3001-009d-2b80-9c306ab81aa6" -X GET \
    https://vlt.example.com:8200/v1/secret/db
```

DYNAMIC CONFIG: RUNTIME

- Config to be adjusted in-flight
- example: Consul KV service discovery
-

```
$ consul-template -consul consul.example.com:6124 \
    -template "/tmp/nginx.ctmpl:/var/nginx/nginx.conf:service nginx restart"
```

EXERCISE: CONFIGURATION MANAGEMENT

Work through the 'Configuration Management' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What are some pitfalls of configuration mismanagement?
- Questions?

FURTHER READING

- Store configuration data using Docker configs: http://dockr.ly/2wTjFrv
- Kubernetes Configuration Management with Containers: http://bit.ly/2E8Atl9

DEVELOPMENT PIPELINE OVERVIEW

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Explain what container-as-a-service means in the context of a typical enterprise using DDC
- Sketch a diagram containing all essential elements of a Docker development pipeline
- List a handful of important differences between a traditional and a containerized CI/CD pipeline

TRADITIONAL CI/CD

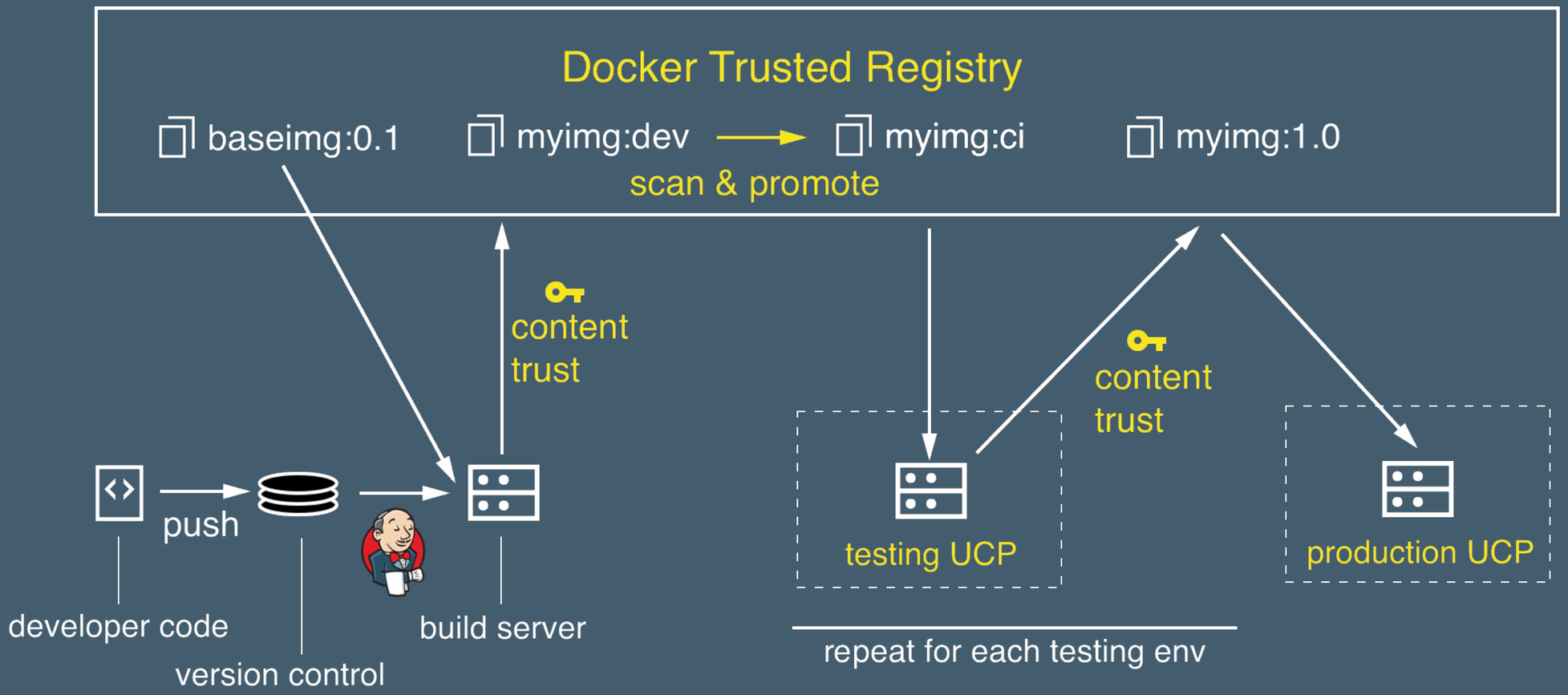
- Push code to a repo
- CI/CD manager pulls code through tests in multiple environments
- Code (automatically?) deployed to production when passing

But what does this look like with containers and images?

CONTAINERS IN CI/CD

- Images are full stack, not code alone
- Must check for dependency vulnerabilities
- Must ensure provenance
- UCP Clusters must reflect pipeline environments

A DOCKER PIPELINE



PIPELINE ALTERNATIVES

- Dev UCP
- Node RBAC instead of separate UCPs
- Build-serverless pipeline
- DTR Webhooks

DEPLOYMENT PHASE

- All familiar deployment strategies possible
- Need to think about how load balancing works on a swarm (external and internal)
- Consider routing meshes (L4 vs L7)

CONTAINERS AS A SERVICE

- (Dev)ops teams can provide base images
- Typically contain OS, middleware, enterprise credentials
- Developers use these as starting point

FURTHER READING

- Dockerized CI/CD Reference Architecture: http://dockr.ly/2tETN0V
- Docker EE Best Practices Reference Architecture: http://dockr.ly/2ohLad6
- Docker Reference Architecture: Development Pipeline Best Practices Using Docker EE http://dockr.ly/2tETN0V

UNIVERSAL CONTROL PLANE

LEARNING OBJECTIVES

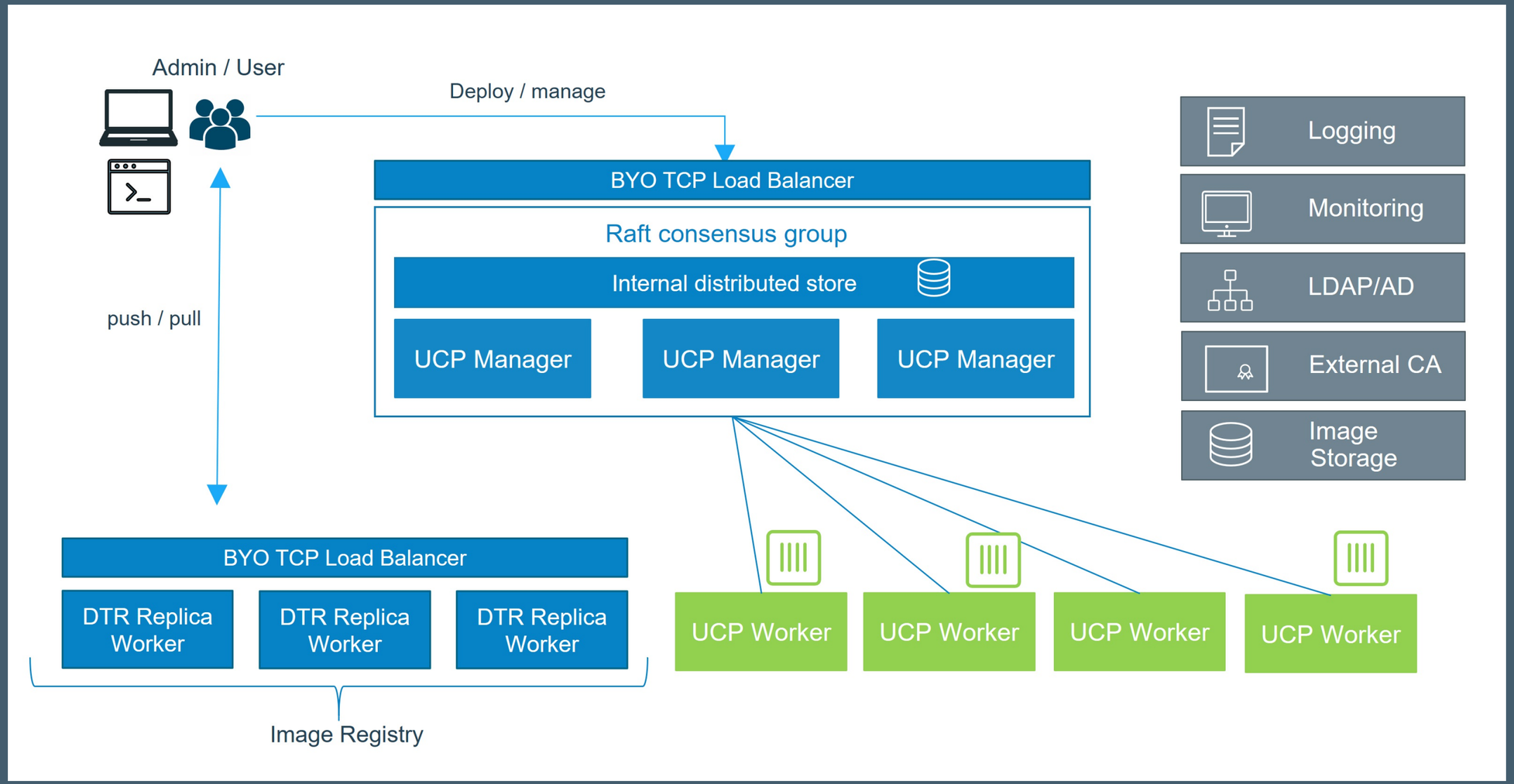
By the end of this module, learners will be able to:

- Install and uninstall a highly available UCP cluster

DOCKER ENTERPRISE EDITION

- Security compliance tooling
- Containerized CI/CD
- Secure image distribution
- Role based access control

DOCKER EE ARCHITECTURE



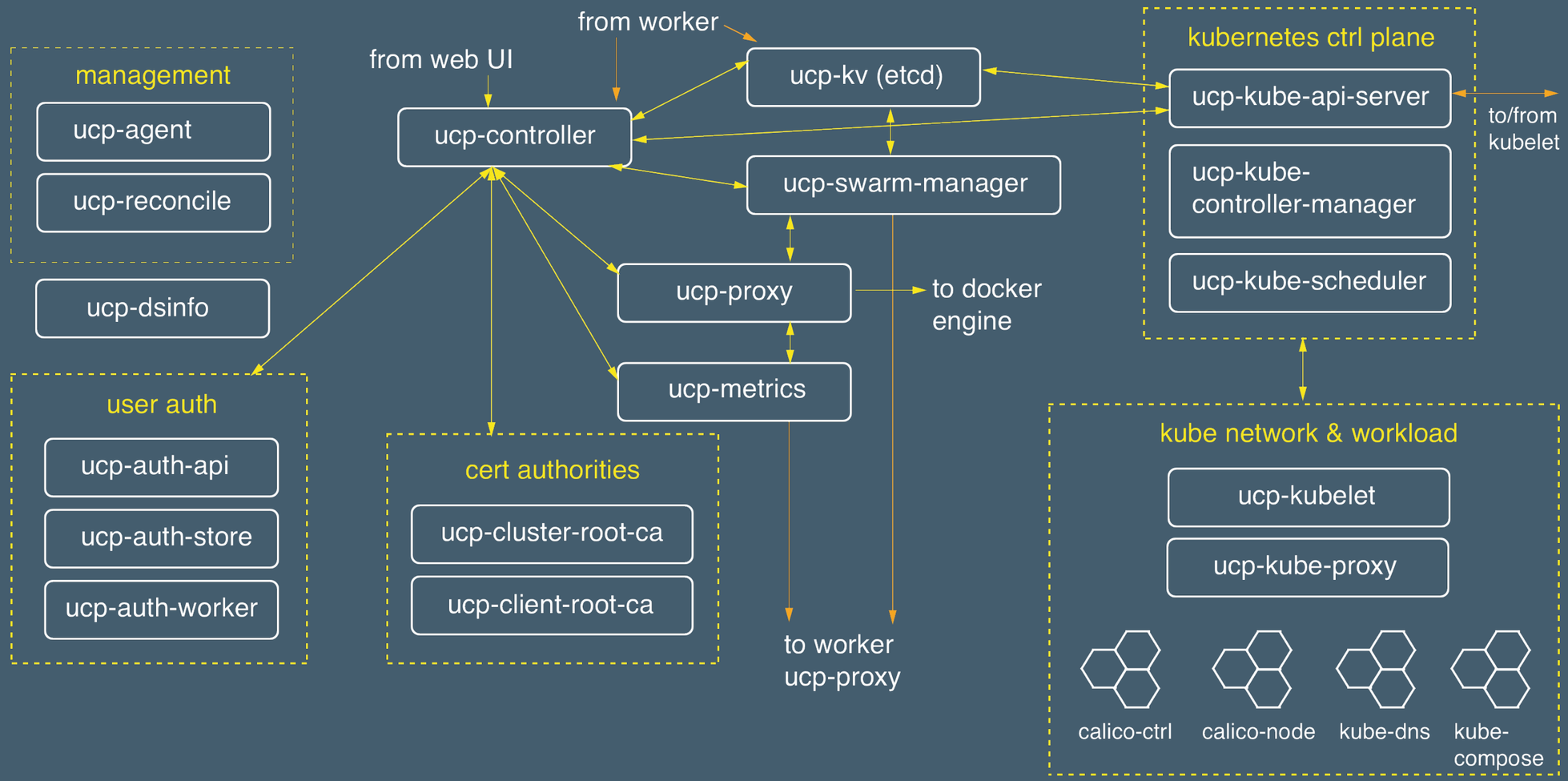
UCP IS A SWARM

- UCP sits on top of a Swarm
- Automatically create Kubernetes Cluster
- Adds UI, RBAC, registry integration...
- Start with one UCP manager == Swarm manager leader
- New UCP nodes joined exactly as new Swarm nodes joined

CHOOSE YOUR ORCHESTRATION ENGINE

- Swarm
- Kubernetes (containerized and running on top of the Swarm)
- Classic Swarm (legacy only)

UCP MANAGER ARCHITECTURE OVERVIEW



EXERCISE: INSTALLING UCP

Work through the 'Installing UCP' exercise in the Docker for Enterprise Developers Exercises book.

FURTHER READING

- High availability architecture and apps with DDC: http://dockr.ly/1sqPrIH
- UCP architecture: https://dockr.ly/2qTOgTb

CONTEXT BASED ROUTING

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Diagram the components that provide layer 7 routing in UCP and Kubernetes
- Configure Swarm services and Kubernetes deployments to use layer 7 routing

HOW IT WORKS

We don't like:

```
https://example.com:8080
https://example.com:8081
...
```

Much better:

```
https://foo.example.com
https://bar.example.com
or
https://example.com/cats
https://example.com/dogs
...
```

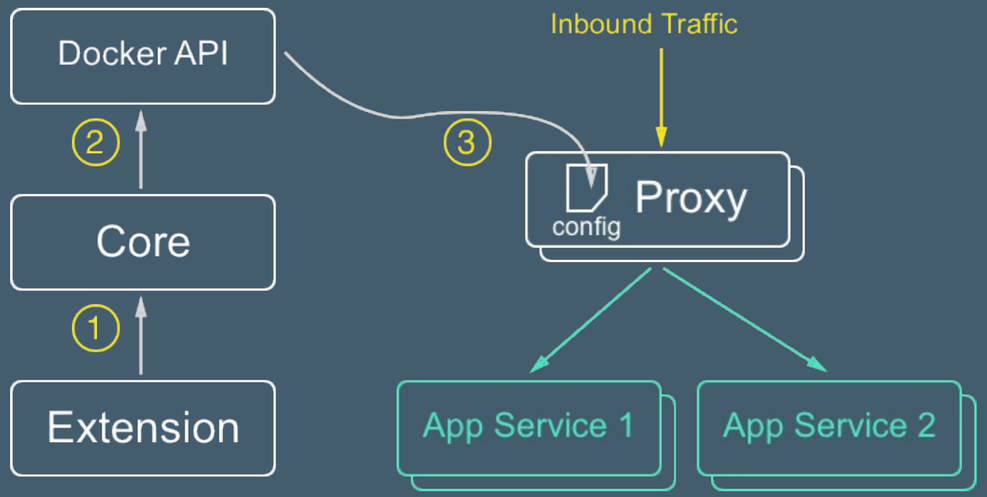
LAYER 7 ROUTING

- L4 routing requires LB be aware of each service
- Application layer (L7) load balancing avoids this
- Route traffic to services based on
  - header information
  - path

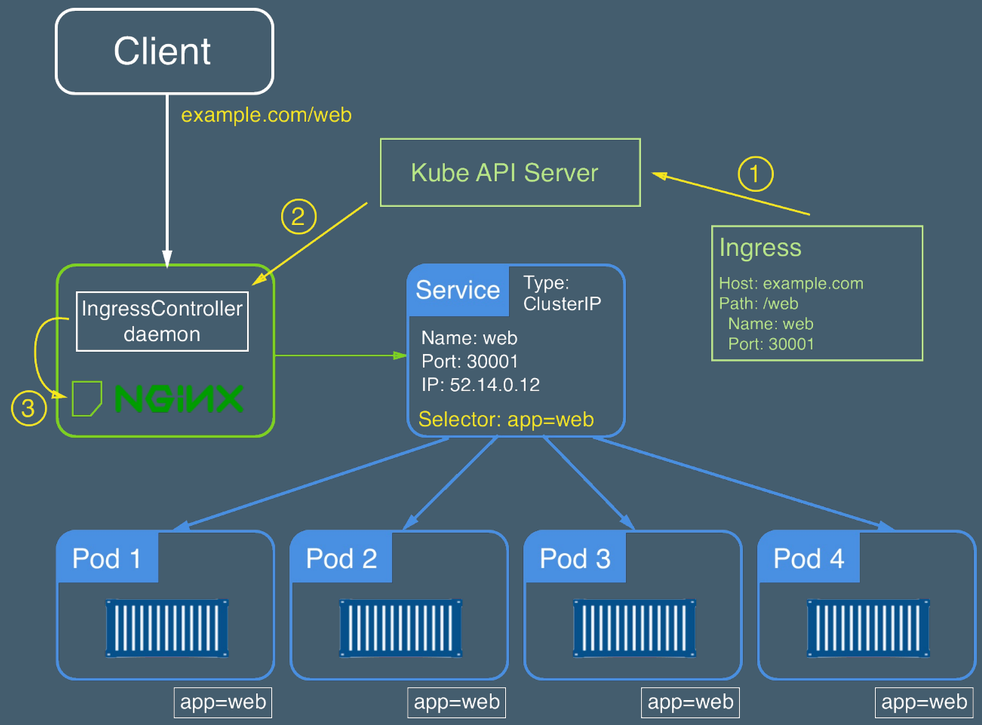
```
Accept:*/*
Accept-Encoding:gzip, deflate, sdch
Accept-Language:en-US,en;q=0.8,de;q=0.6,nb;q=0.4
Cache-Control:no-cache
Connection:keep-alive
Cookie:...
DNT:1
Host:pets.my-company.com
Pragma:no-cache
Referer:...
```

INTERLOCK 2 & INGRESSCONTROLLER

Docker Swarm



Kubernetes



EXERCISE: LAYER 7 LOAD BALANCING

Work through the 'Layer 7 Load Balancing' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- In what situations would L7 routing be preferable to L4 routing? Vice-versa?
- Questions?

FURTHER READING

- Interlock 2: http://bit.ly/2ETb4cx
- Kubernetes IngressController: http://bit.ly/2nMFaqL
- Nginx IngressController: http://bit.ly/2skeOko
- Use swarm mode routing mesh: http://dockr.ly/2ePQeOM
- Docker Reference Architecture: UCP 2.0 Service Discovery and Load Balancing: http://dockr.ly/2rbxDDX
- Get to know the DDC networking updates: http://dockr.ly/2fbZ61F

DOCKER TRUSTED REGISTRY

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Install a highly available DTR
- Configure a UCP node so it trusts DTR
- Push an arbitrary Docker image to a DTR repository
- Uninstall DTR

SECURE SOFTWARE SUPPLY CHAIN

- Image Creation
- Image Distribution
- Container Execution

DTR KEY FEATURES

- Image Creation:
  - Image Security Scanning
  - Repository Automation
  - Image Promotion
- Image Distribution:
  - Content Trust / Notary
  - Content Cache
- Image Storage:
  - Pluggable Storage Drivers

PUSHING IMAGES TO DTR

- Configure your Docker engine to trust DTR

```
$ sudo curl -k https://<DTR_FQDN>/ca \
    -o /etc/pki/ca-trust/source/anchors/<DTR_FQDN>.crt
$ sudo update-ca-trust
$ sudo /bin/systemctl restart docker.service
```

- Login to DTR:

```
$ docker login <DTR_FQDN>
```

- Tag your images with:

```
<DTR_FQDN>/<Org Name>/<Image Name>:<Tag>
```

- Push image

Note: **<Image Name>** and **<Repository Name>** are equivalent.

EXERCISE: INSTALLING DTR

Work through the 'Installing Docker Trusted Registry' exercise in the Docker for Enterprise Developers Exercises book.

FURTHER READING

- DTR overview: https://dockr.ly/2F84rkN
- Troubleshoot DTR: https://dockr.ly/2JhT78m
- Integrate with multiple registries: https://dockr.ly/2HqBzX7

CONTENT TRUST

LEARNING OBJECTIVES

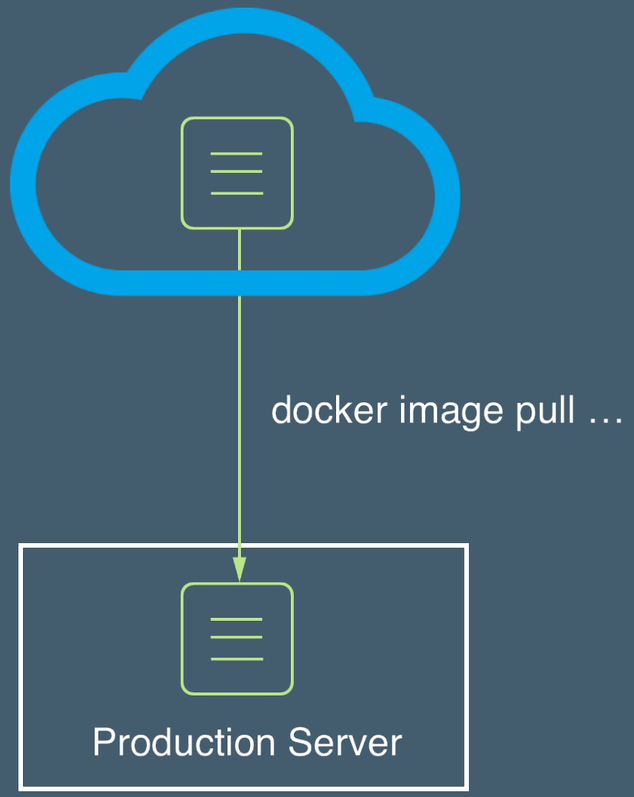
By the end of this module, learners will be able to:

- Identify the kinds of attacks and countermeasures considered by The Update Framework
- Describe the differences in Docker's behavior with content trust enabled versus disabled
- Establish content trust for a repository in DTR
- Demand UCP only runs images signed by a set of teams

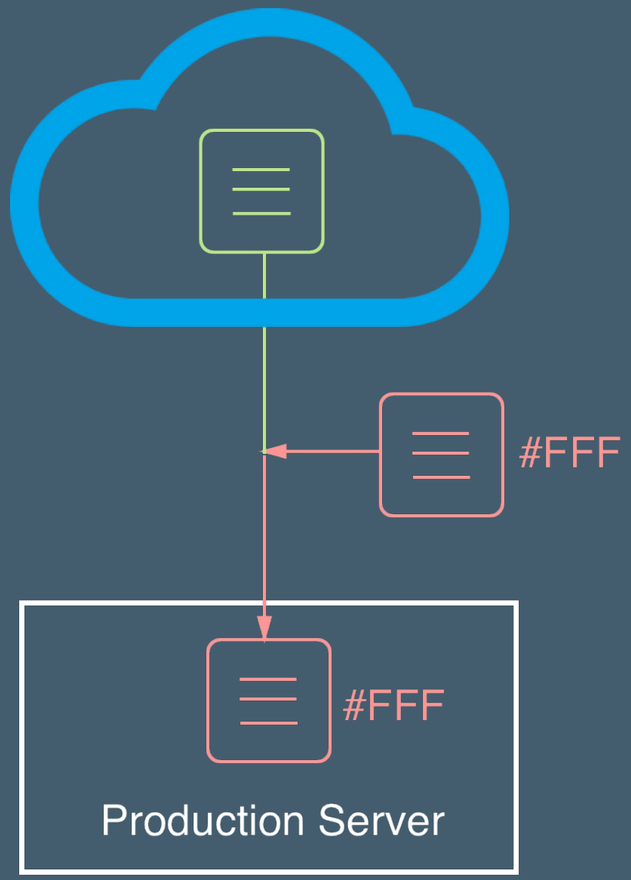
WHAT IMAGE DO YOU MEAN?

- Mutable: ref by tag: alpine:latest
- Immutable: ref by sha: alpine@sha256:FFFF....

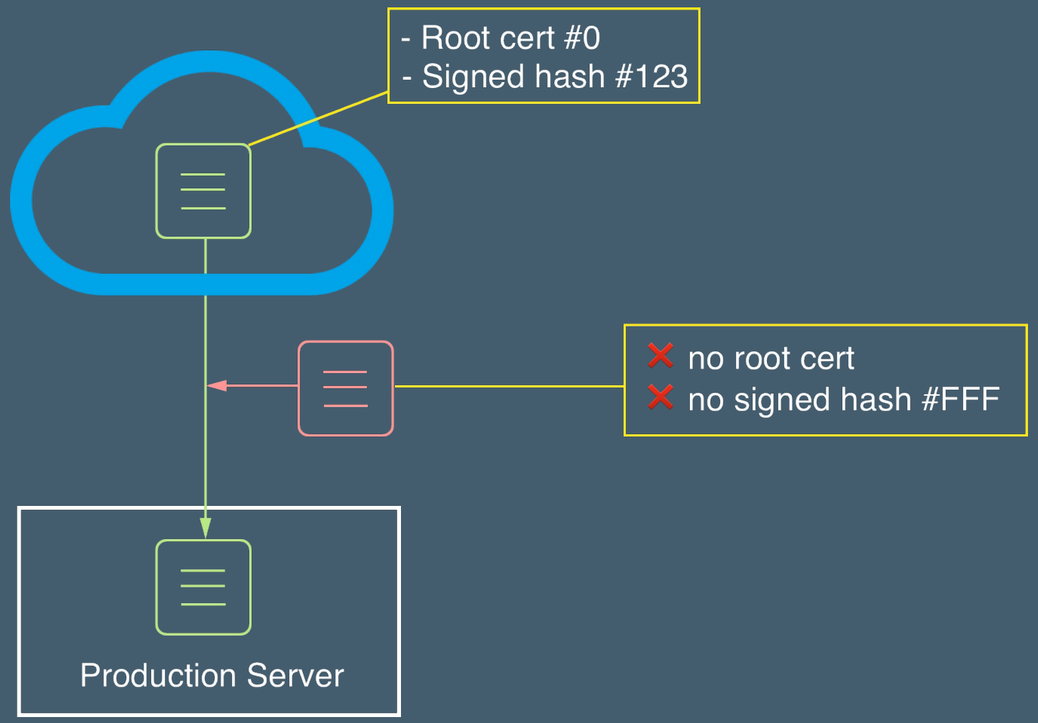
DOWNLOADING SOFTWARE



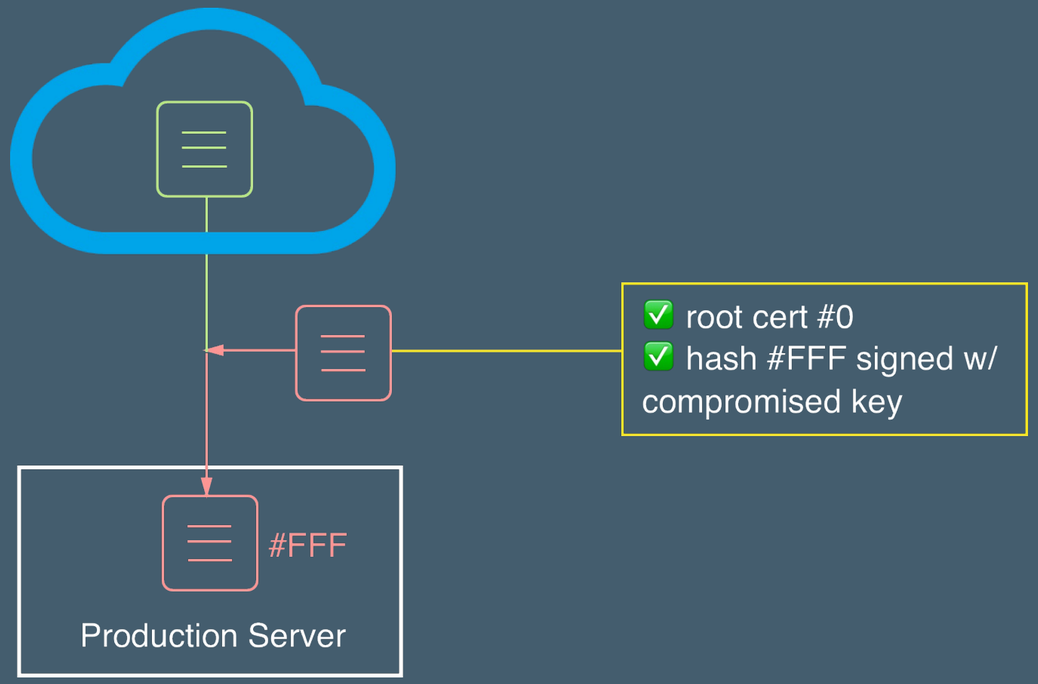
SIMPLE INJECTION ATTACK



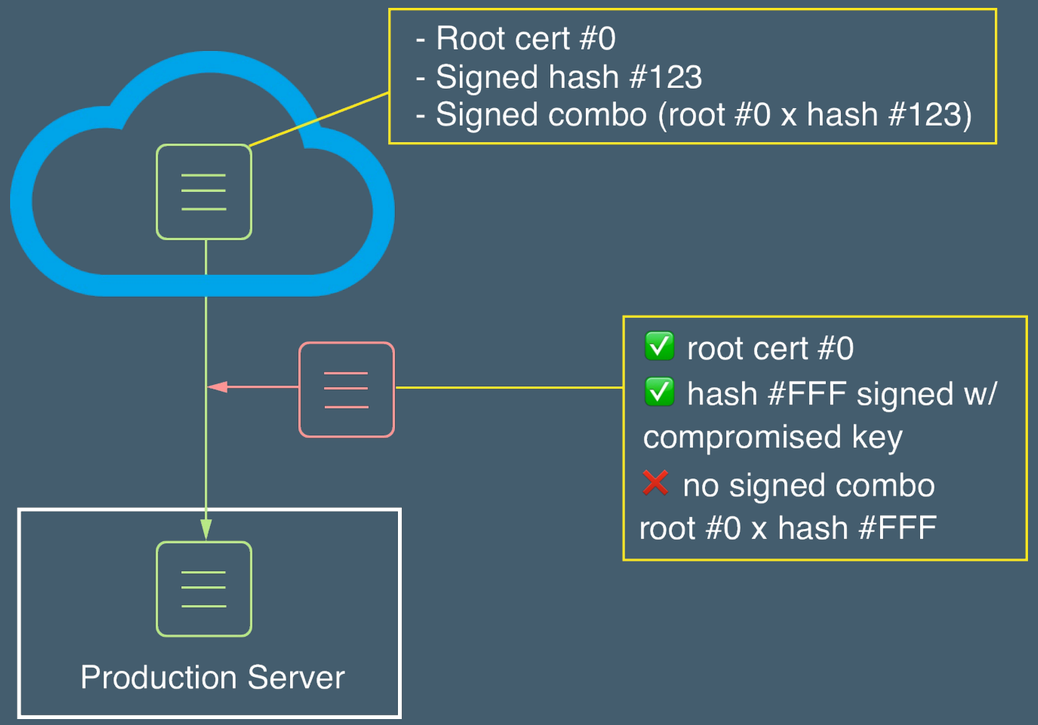
SIGNED CONTENT



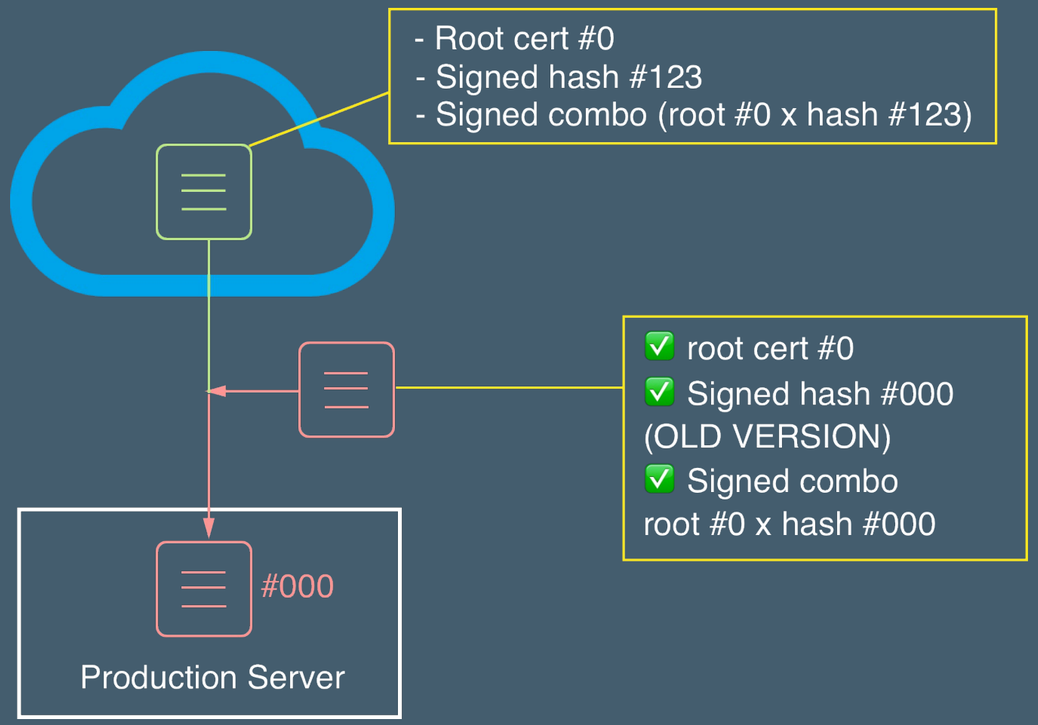
MIX AND MATCH ATTACK



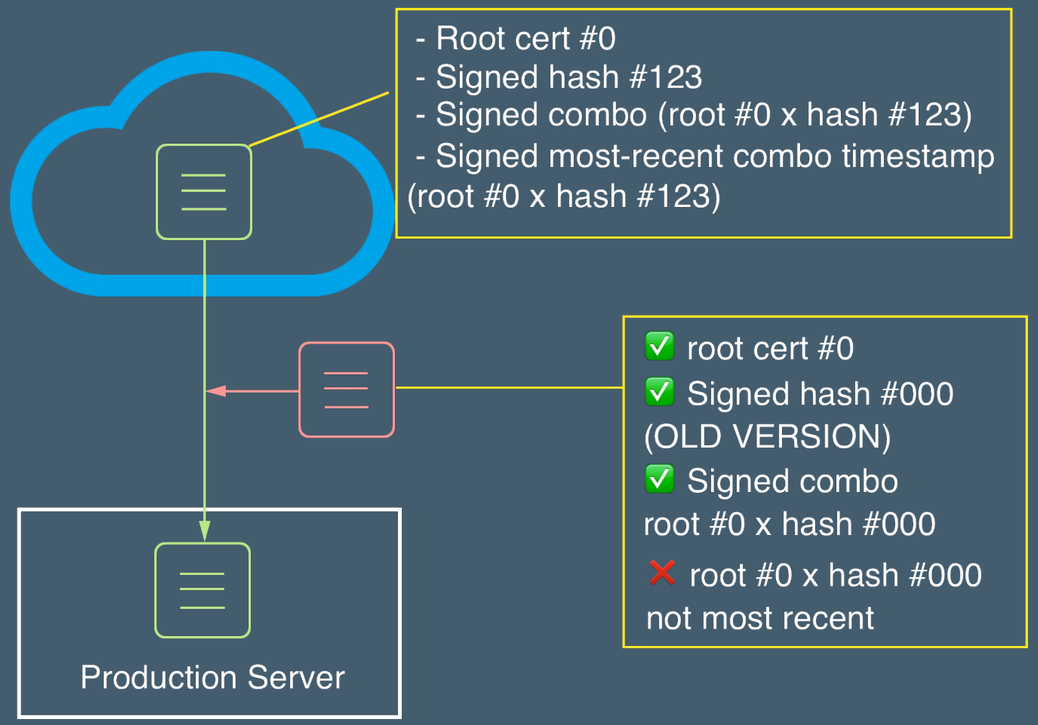
METADATA VALIDATION



DOWNGRADE ATTACK



FRESHNESS GUARANTEED



DOCKER CONTENT TRUST

- Image publishers sign their images
- Image consumers can ensure their images are signed
- Implements The Update Framework (TUF) into Docker as Notary
  - http://theupdateframework.com/
  - https://github.com/docker/notary

CONTENT TRUST FOR IMAGE PUBLISHERS

- Metadata generated and signed on image push certifying authenticity
- Four (+1) different keys are used:
  - Root key: fundamental signing authority
  - Target key: repository content integrity
  - Snapshot key: metadata integrity
  - Timestamp key: repository freshness
  - Delegation key: multiple signing entities (optional)

(Jargon aside: the target key and snapshot key are sometimes collectively called the repository key, and the root key is sometimes called the offline key).

CONTENT TRUST FOR IMAGE CONSUMERS

If content trust is enabled, only signed images are available for use with:

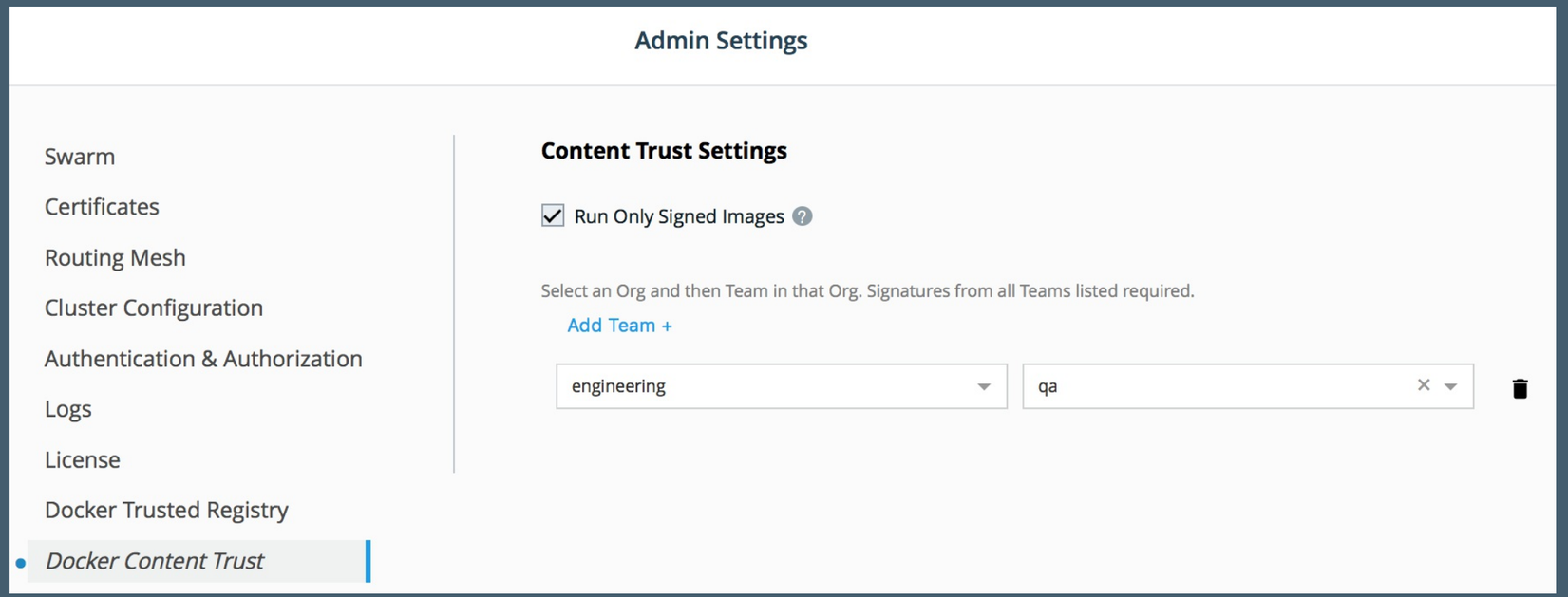
- docker image push
- docker image pull
- docker image build
- docker container create
- docker container run
- docker service create

CONTENT TRUST IN DTR

- DTR comes with a built in Notary server
- Can enable trust directly from Docker Engine
- Optional: install Notary client side for finer-grained control

INTEGRATION WITH UCP

- UCP can be configured to only run signed images
- Checked during application deployment
- Any unsigned image will be rejected
- Multiple signatures possible



EXERCISE: CONTENT TRUST

Work through the 'Content Trust' exercise in your exercise book.

DISCUSSION

- What is one simple alternative to content trust?
- Questions?

FURTHER READING

- Intro to content trust: http://dockr.ly/2gAgIo3
- Features of Docker content trust: http://dockr.ly/1EyCxrR
- Automation with content trust: http://dockr.ly/2x5lLaC

IMAGE SECURITY SCANNING

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Describe the process DTR takes to scan images for vulnerabilities
- Set up security scanning on DTR, and configure it for an individual repository
- Use DTR’s UI to find out an image's vulnerabilities, and more information about that vulnerability

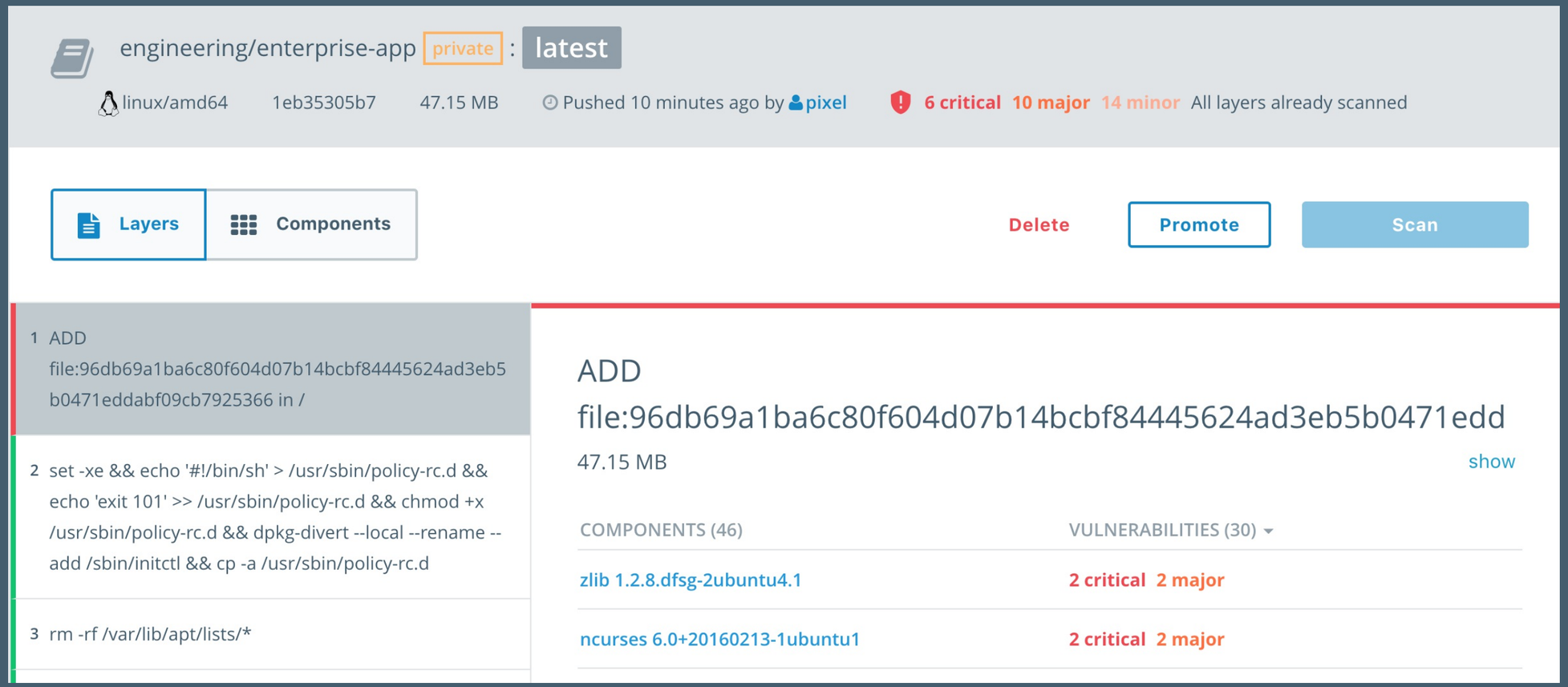
WHAT'S IN YOUR IMAGES?

- Components installed in base layer
- Components installed in custom layers
- Known vulnerabilities?
- (Currently) unknown vulnerabilities?

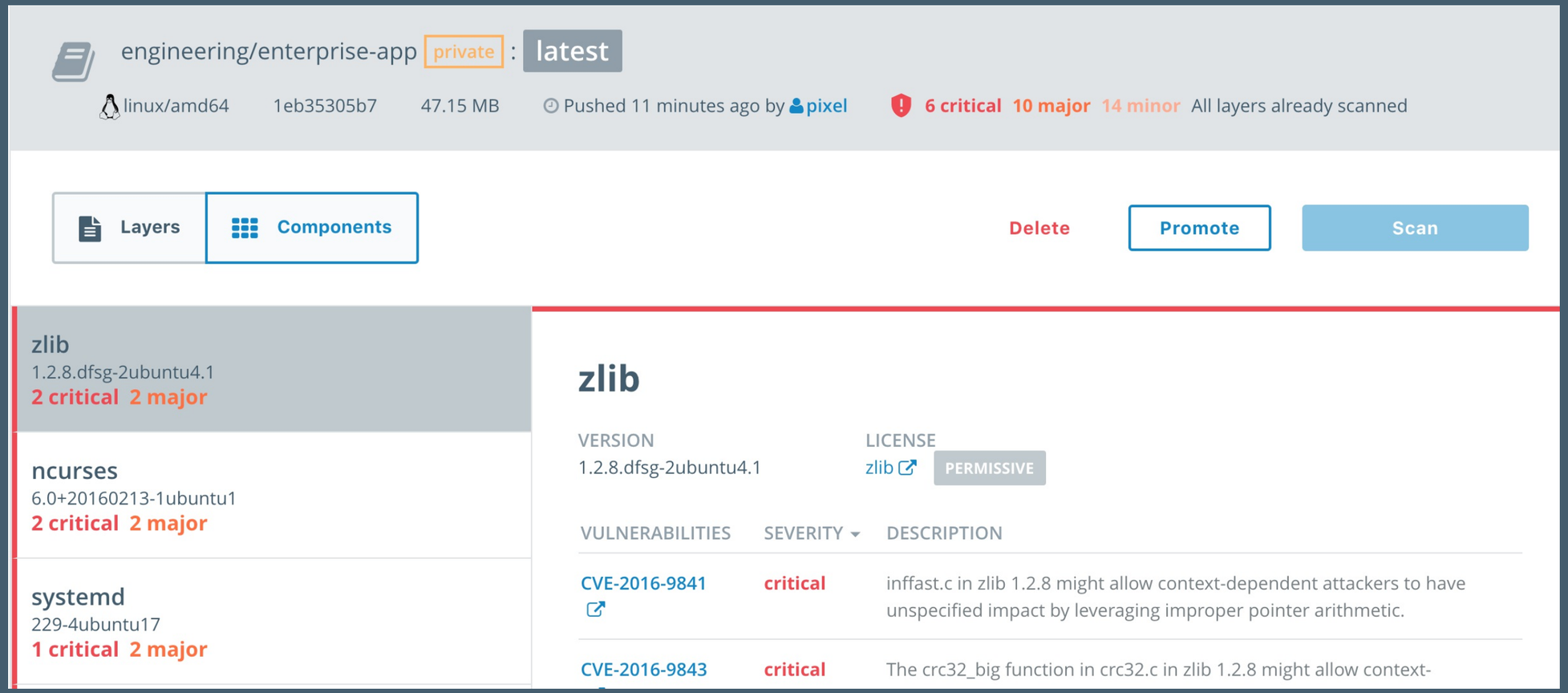
DTR SECURITY SCANNING FLOW

1. New layer pushed to DTR
2. Scan triggered (auto or manual)
3. Bill-of-materials generated and saved (slow, resource intensive)
4. BoM re-checked against CVE database every time the db is updated (fast)
5. Regular database updates from https://dss-cve-updates.docker.com/

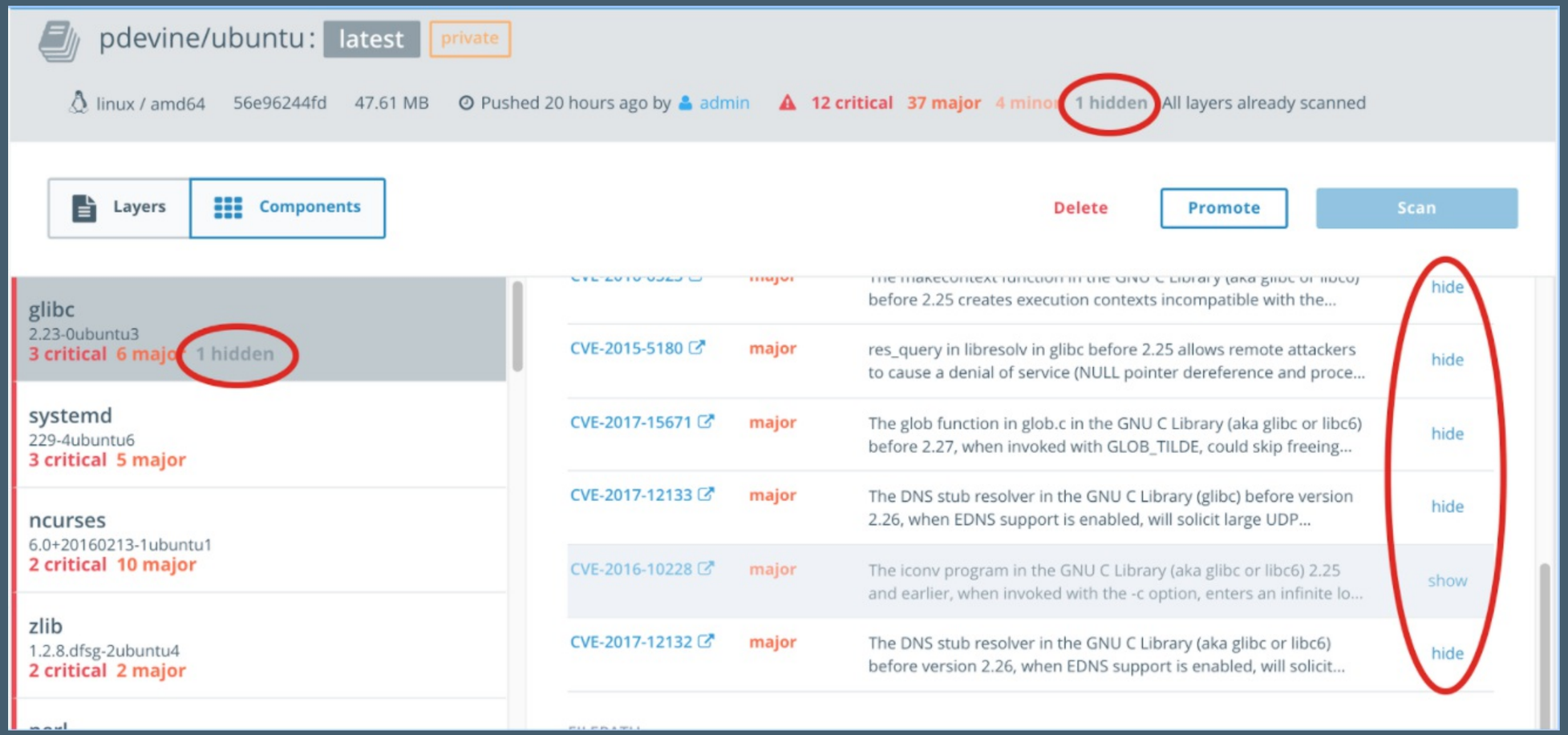
SCAN RESULTS PER LAYER



SCAN RESULTS PER COMPONENT

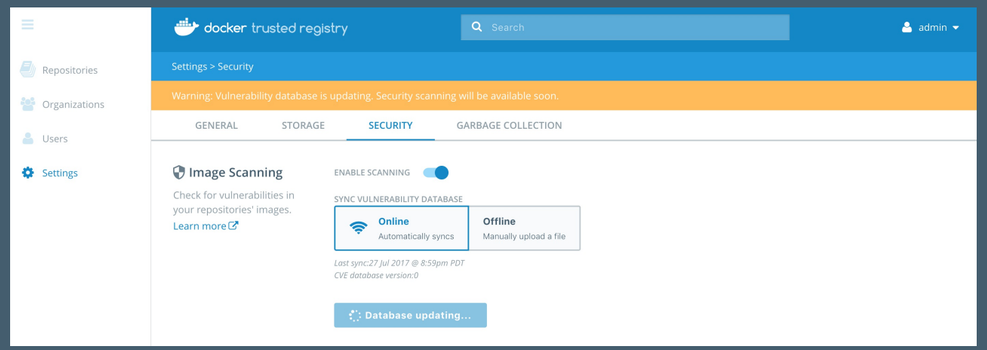


VULNERABILITY OVERRIDES



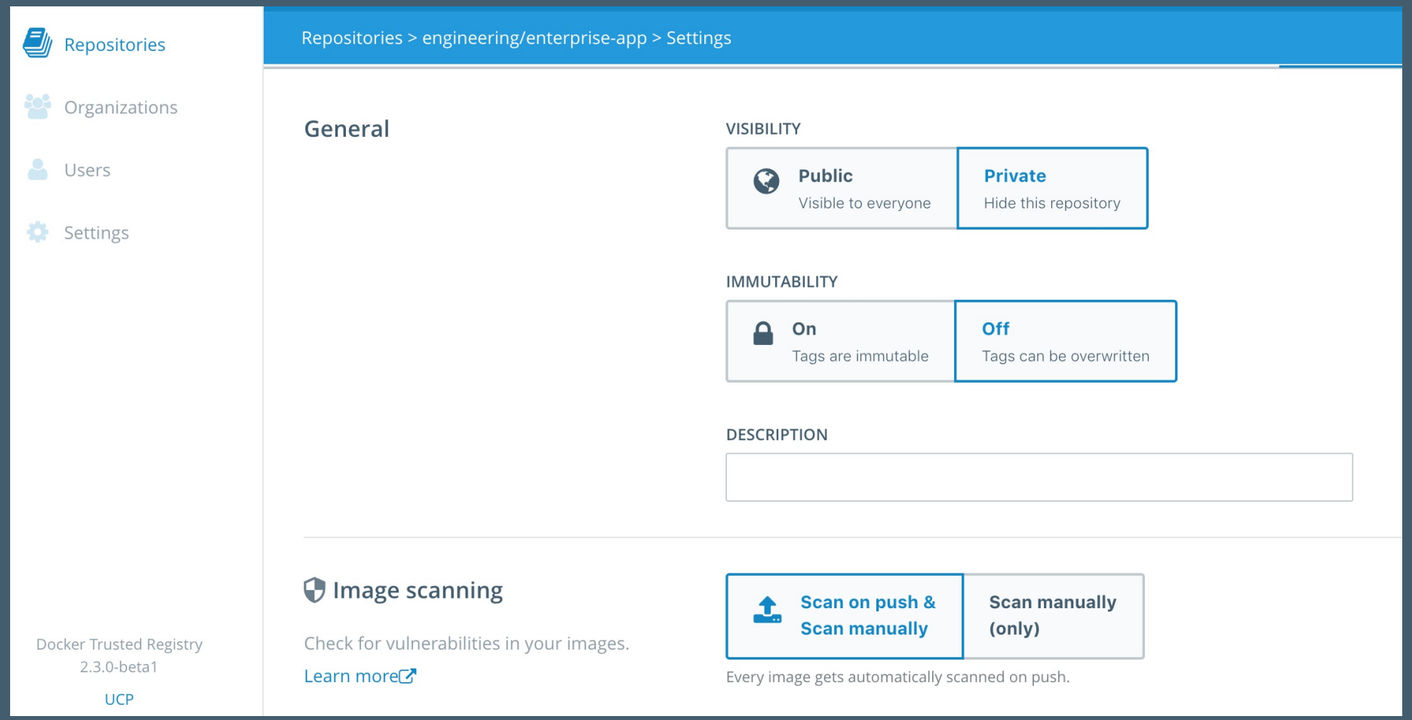
- Allows vulnerabilities to be hidden
- Matches based on different components

CVE DATABASE UPDATES



- DTR -> System -> Security
- Automatic updates: daily 3 AM UTC; must be able to reach https://dss-cve-updates.docker.com/ on port 443.
- Manual updates: uploaded through DTR, downloaded through store.docker.com.

SCANNING AUTOMATION



COMMON SCANNING MISTAKES

- Make sure initial CVE database is downloaded before starting scans
- Make sure all DTR replicas can reach storage backend
- Consider manual-only scans as part of pipeline

EXERCISE: IMAGE SCANNING

Work through the 'Image Scanning in DTR' exercise in your exercise book.

DISCUSSION

- Scanning identifies a vulnerability. What are some generic steps to mitigate?
- Questions?

FURTHER READING

- Set up security scanning in DTR: https://dockr.ly/2HMgp9d
- Scan images for vulnerabilities: https://dockr.ly/2HNOdCK

REPOSITORY AUTOMATION

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Automatically retag an image from one DTR repo to another
- Define webhooks triggered by DTR events
- Integrate DTR into a CI/CD chain using the above

AUTOMATION TOOLS

- Image Promotion & Mirroring
- Webhooks

IMAGE PROMOTION

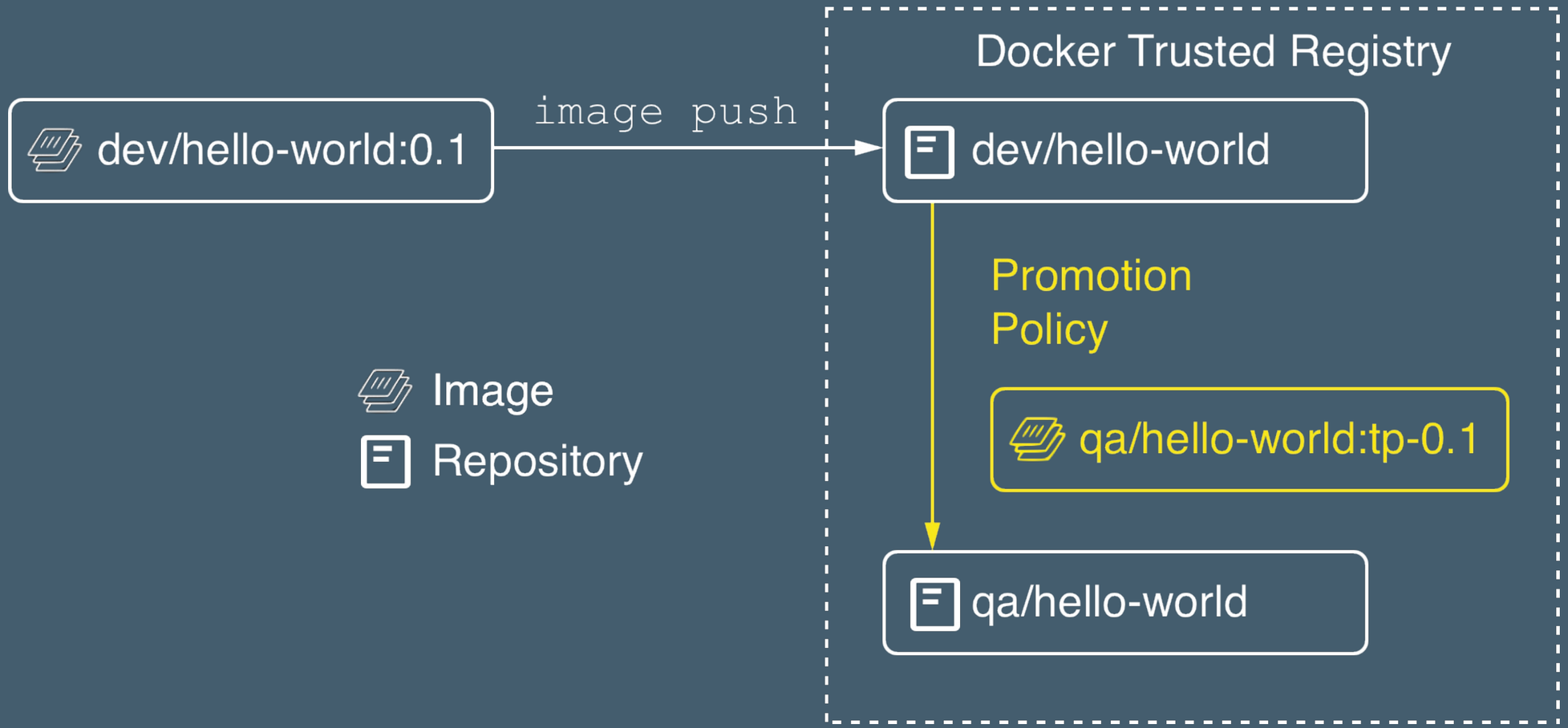


IMAGE PROMOTION POLICIES

- Manual or automatic
- Automatic promotion can be triggered on:
  - Tag name
  - Package name
  - Minor / Major / Critical / All vulnerabilities
  - Component licenses
- No limit to number of policies that can be defined

IMAGE MIRRORING

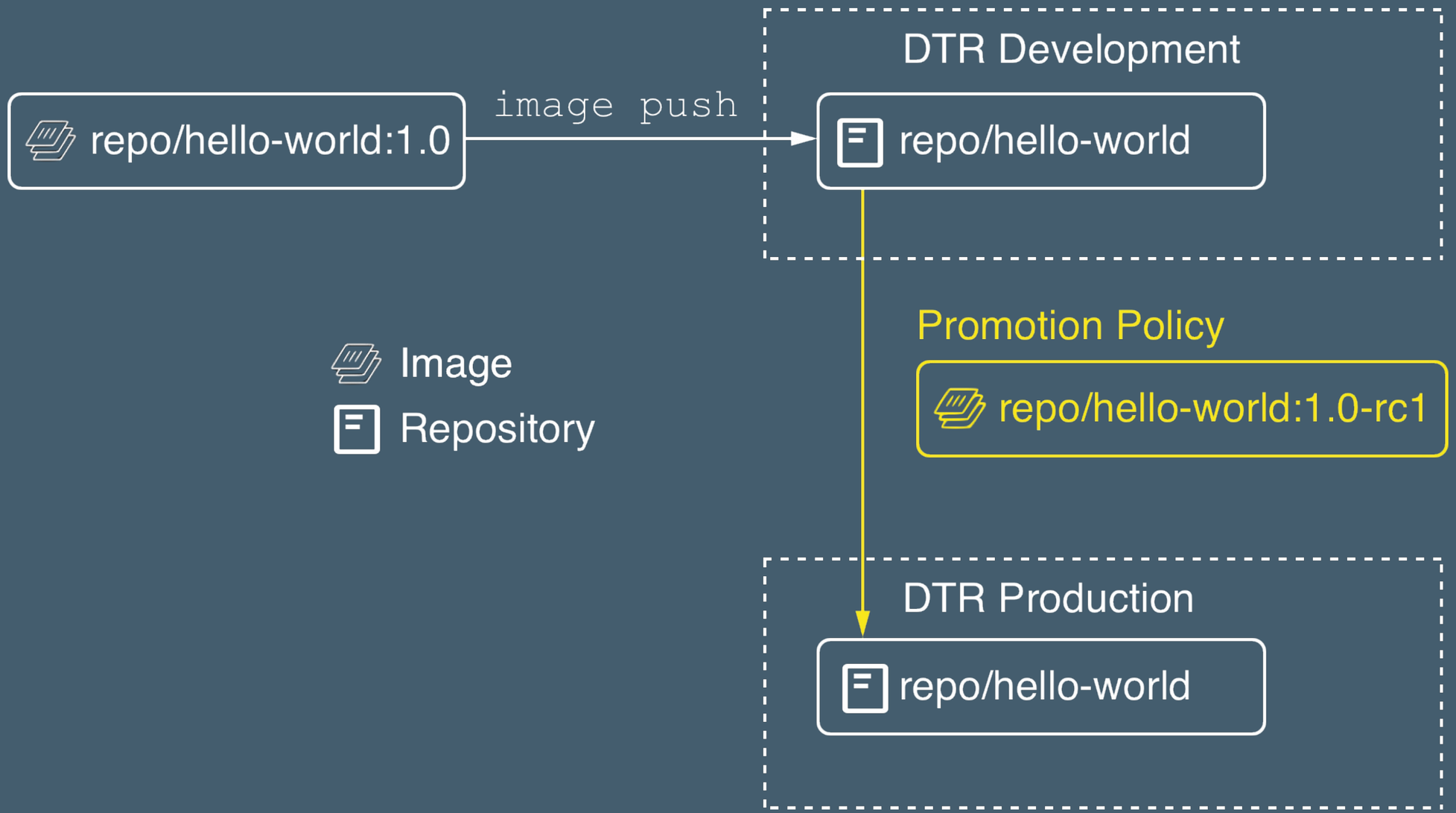
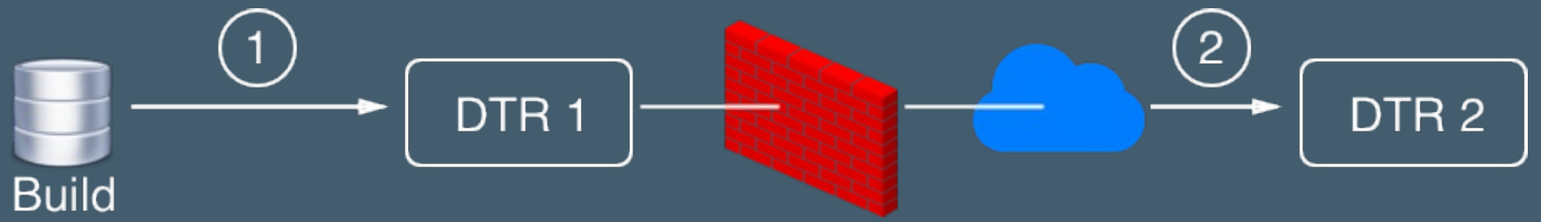


IMAGE MIRRORING - PUSH BASED



- Image pushed to DTR 1
- If policies met => push to DTR 2
- AuthN & AuthZ managed by each DTR
- Signing & scan data not (yet) preserved

IMAGE MIRRORING - PULL BASED

- Image pushed to DTR 1
- DTR 2 polls DTR 1 for updates
- New image found => pull to DTR 2
- Combine with Promotion Policies

WEBHOOKS

POST message with JSON payload, triggered on:

- Tag push or delete
- Manifest push or delete
- Security scan failed
- Security scan complete

Defined per repository.

WEBHOOK PAYLOAD

- Webhook payloads always come in a wrapper:

```
{
 "type": "...",
 "createdAt": "2012-04-23T18:25:43.511Z",
 "contents": {...}
}
```

- The contents key depends on the event type; see https://dockr.ly/2JjcAW7 for the full spec.

EXERCISE: REPOSITORY AUTOMATION

Work through the 'Image Promotion & Webhooks' exercise in your exercise book.

DISCUSSION

- What are the pros and cons of promoting images in a single DTR, versus mirroring them across multiple DTRs?
- Questions?

FURTHER READING

- Managing webhooks: https://dockr.ly/2JjcAW7
- Promotion policies overview: https://dockr.ly/2KarnDN
- Image Promotions and Immutable Repos: http://bit.ly/2eEz7TH

TAGGING & VERSIONING STRATEGIES

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Recognize and use some common image tagging patterns.

TAGGING IMAGES

<URL-to-registry>/<org-or-user>/<repo-name>:<tag>

https://dtr.example.com/engineering/web-ui:1.0.2

Tag is NOT unique!

Each image get SHA-256 associated == unique!

VERSIONING STRATEGIES

- Semantic Versioning: <MAJOR.MINOR.PATCH>
- Release Date Versioning: <YY.MM.PATCH>
- Other
- Company specific versioning

TAGGING & VERSIONING STRATEGIES

Version in Tag is not enough...

Mongo DB

- mongo:3.0.15
- mongo:3.0.15-windowsservercore

Python

- python:2.7.13
- python:2.7.13-wheezy
- python:2.7.13-alpine
- python:2.7.13-windowsservercore

FURTHER READING

- Docker tag reference: http://dockr.ly/2vP3vhw
- Docker Reference Architecture: Development Pipeline Best Practices Using Docker EE: http://dockr.ly/2tETN0V

BUILD SERVER

LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Combine Jenkins, DTR and GitHub to automatically build and push images upon code updates.

BUILD SERVER

Typical tasks of a CI server:

- Building images
- Testing images
- Pushing images
- Signing images
- Promoting images
  - to other environments
  - to other repositories
- Provisioning of infrastructure
- Notifying team members upon success or failure
- Auditing activity

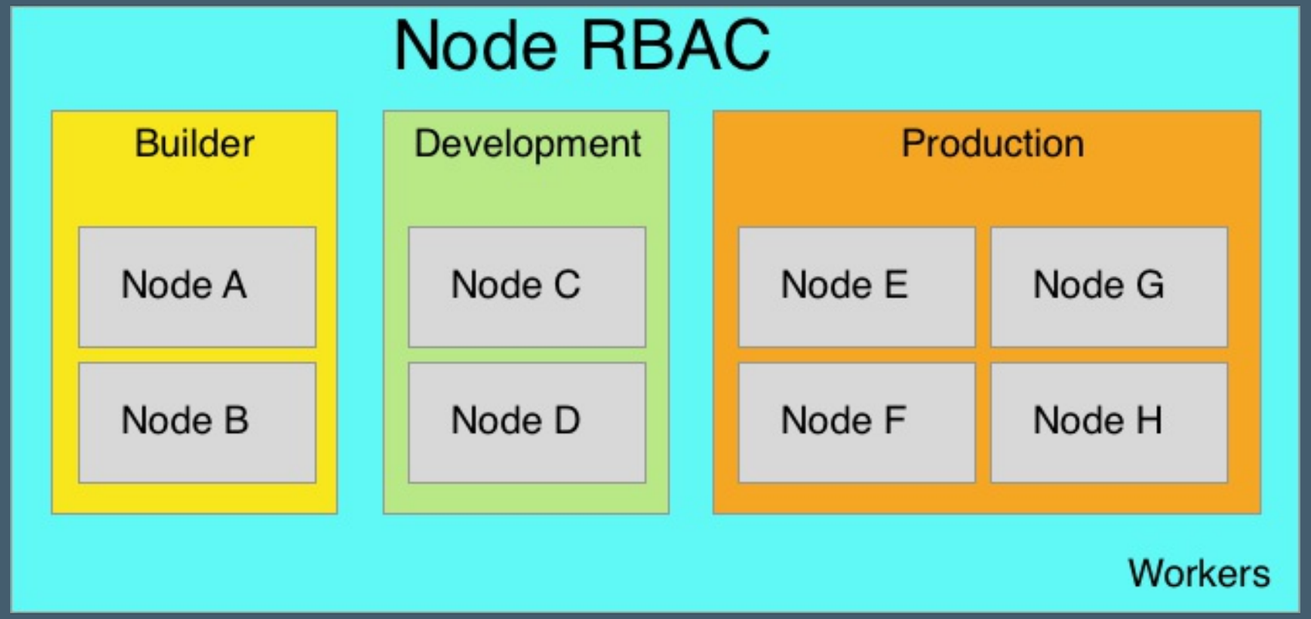
BUILD SERVERS

CI servers often used on premise:

- Cloud Bees: Jenkins
- Jetbrains: TeamCity
- Atlassian: Bamboo
- Microsoft: TeamFoundation server

BUILD SERVER - UCP

- Run build master/agents as containers
- Mount Docker socket to build, sign and push images
- Use node RBAC to define Builder Swarm Segment



RUNNING TEAMCITY

```
version: '3.1'
services:
  teamcity:
    restart: unless-stopped
    image: sashgorokhov/teamcity
    ports:
      - "0.0.0.0:8111:8111"
    volumes:
      - "/mnt/teamcity:/mnt/teamcity"
  teamcity_agent:
    restart: unless-stopped
    image: sashgorokhov/teamcity-agent
    environment:
      SERVER_URL: teamcity:8111
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
```

EXERCISE: BUILD SERVER

Work through the 'Build Server' exercise in the Docker for Enterprise Developers Exercises book.

DISCUSSION

- What are some features of DTR and UCP you'll want to integrate with your CI pipeline?
- Questions?

FURTHER READING

- Configure automated builds with Docker Hub: http://dockr.ly/2wjwo4Z
- Configure automated builds with Bitbucket: http://dockr.ly/2xeTX30
- Run only the image you trust: https://dockr.ly/2HIaAA5
- DockerCon Video: Delivering Ebay's CI Solution with Apache Mesos and Docker: http://dockr.ly/2vWoBdR

DOCKER FOR ENTERPRISE DEVELOPERS

Thanks for coming! Please take one of our feedback surveys:

- Docker for Enterprise Developers (standalone): http://bit.ly/2GzPQQp
- Docker Fundamentals + Enterprise Developers (combined class): https://bit.ly/2uhDfgg

Get in touch: training@docker.com

training.docker.com

YOU'RE ON YOUR WAY TO BECOMING DOCKER CERTIFIED!

- Study up with our Study Guides at http://bit.ly/2yPzAdb
- Take it online 24 hours a day
- Results delivered immediately
- Benefits include:
  - Digital certificate
  - Online verification
  - Private LinkedIn group
  - Exclusive events

SUCCESS.DOCKER.COM/CERTIFICATION