Integration testing with TestContainers and JUnit 5

Nikolay Kuznetsov

Zalando

@nikolayk812

Helsinki JUG

9 December 2019

About me

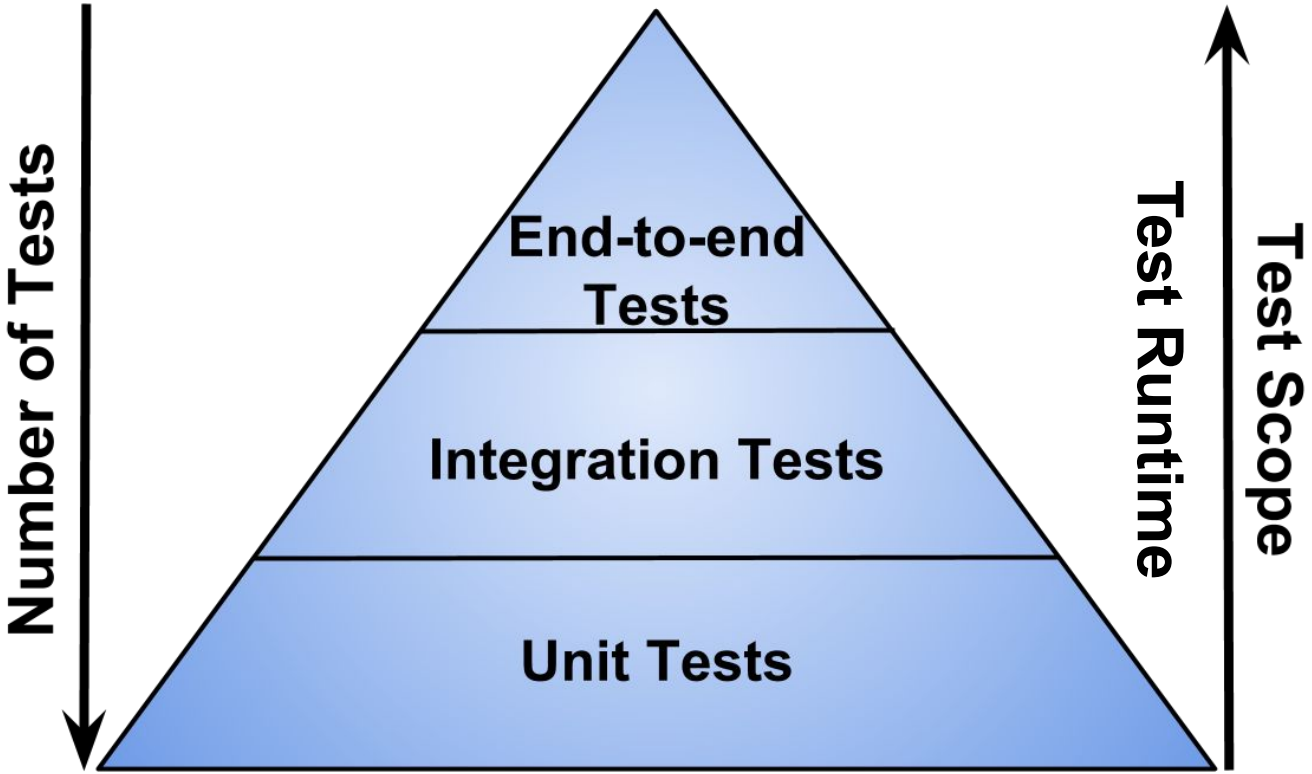
- *Go* developer at **Zalando Wardrobe**
- 6+ years of *Java* experience
- Conference speaker:
  - Voxxed Days Cluj, Container Days Hamburg
- TestContainers-Go contributor

Why integration testing?

2 unit tests, 0 integration tests

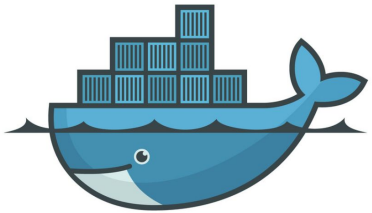
Basic integration test

Trade-offs


Number of Tests
End-to-end Tests
Integration Tests
Unit Tests
Test Runtime
Test Scope

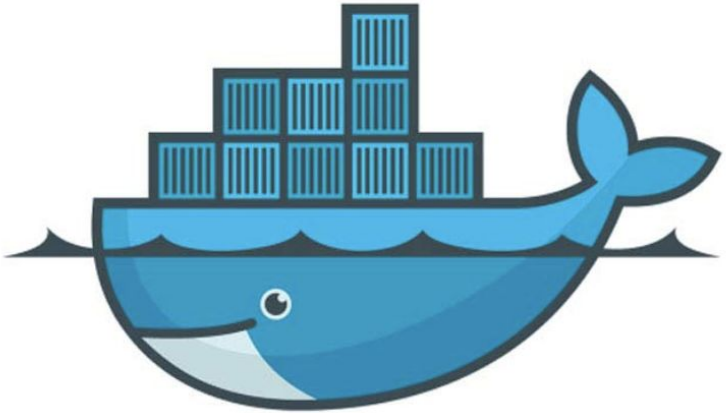
Integration testing evolution

- In-memory mocking
- Local DBs
- Vagrant
- Docker / Docker Compose
- Docker API

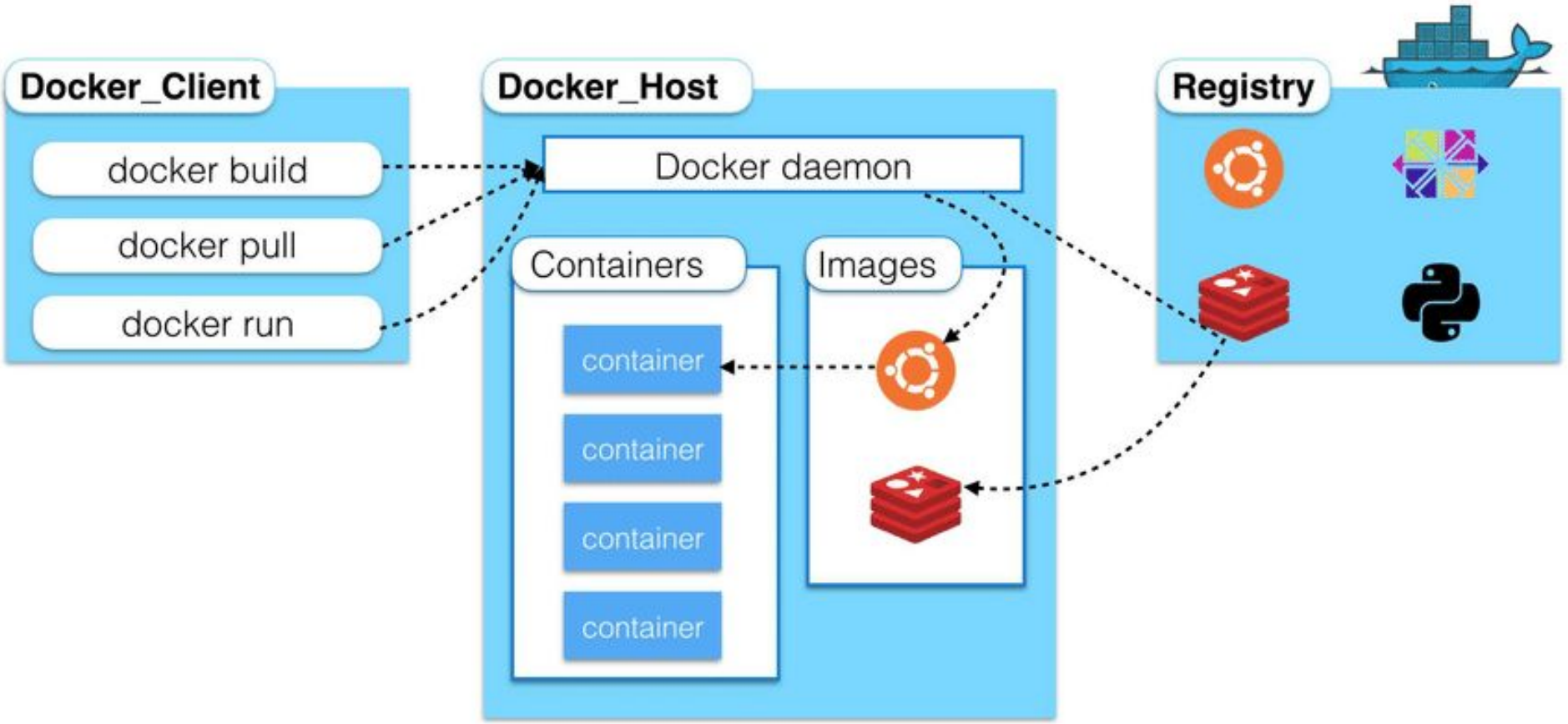


Docker advantages

- 100% compatible database
- Same version as production
- Empty or known state

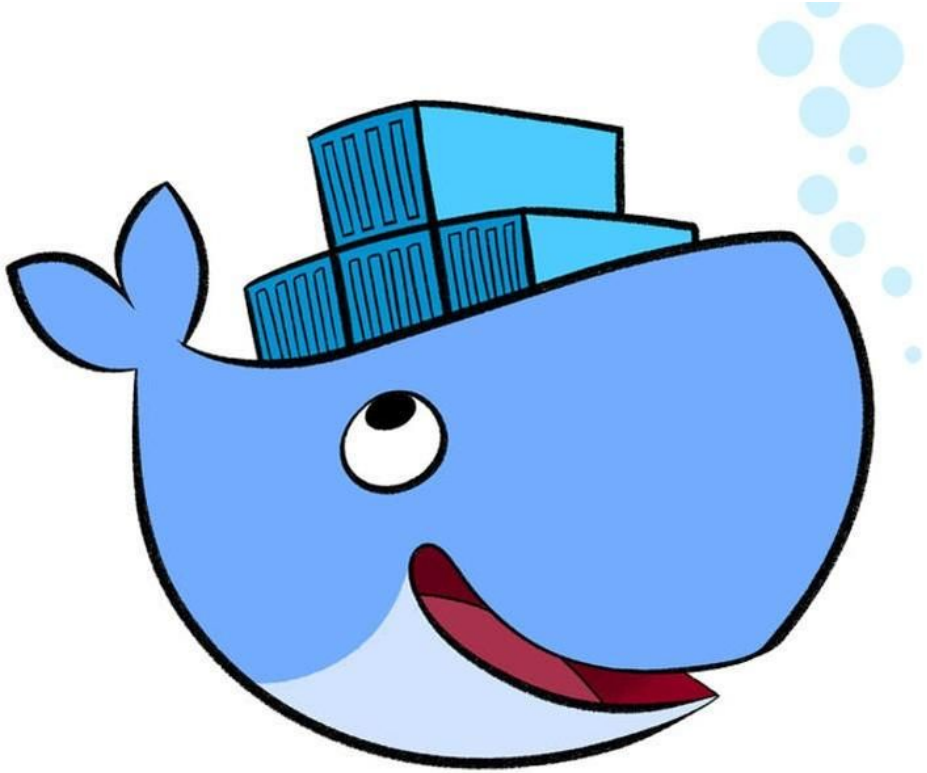


Docker architecture



How to start a container for test?

- Shell scripts
- Maven plugin
- Docker Compose
- **Docker API**
- MiniKube, Kubernetes



Shell scripts

```
#!/bin/sh

DB_NAME=users
TABLE_NAME=users

if [[ ! $(docker ps -q -f name=user-postgres) ]]; then
    if [[ $(docker ps -aq -f status=exited -f name=user-postgres) ]]; then
        # cleanup
        docker rm user-postgres
    fi
    docker run -d --name user-postgres -p 5432:5432 postgres:9.6.13
    sleep 5
    docker exec user-postgres psql --user=postgres -p 5432 -c "CREATE DATABASE $DB_NAME;"
else
    echo "it's already running"
fi
```

Maven plugins

github.com/fabric8io/docker-maven-plugin

docker-maven-plugin

This is a Maven plugin for building Docker images and managing containers for integration tests. It works with Maven 3.0.5 and Docker 1.6.0 or later.

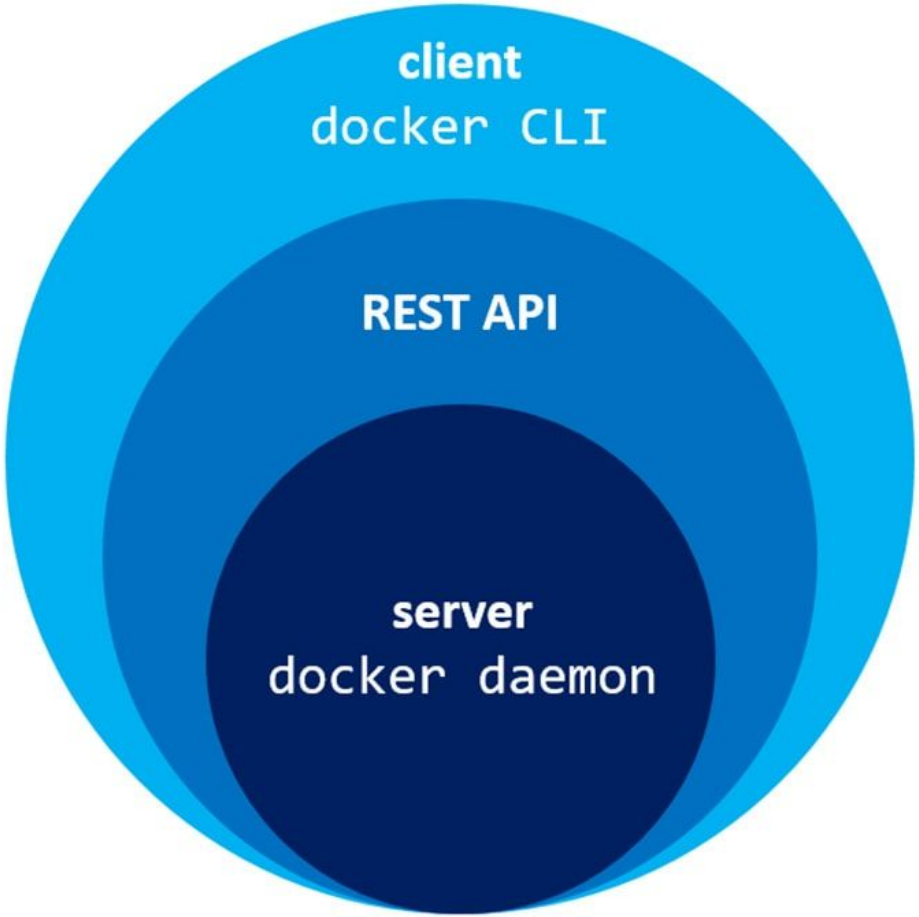
Goals

Goal	Description	Default Lifecycle Phase
`docker:start`	Create and start containers	pre-integration-test
`docker:stop`	Stop and destroy containers	post-integration-test

Docker Compose

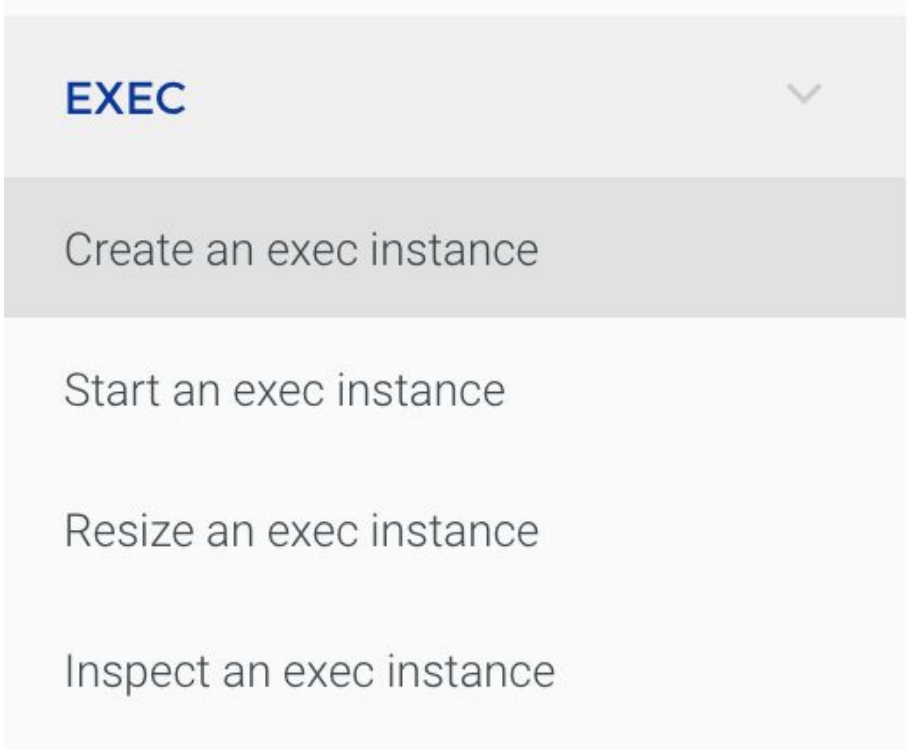
```
version: "2"
services:
  eureka:
    image: tc-demo/eureka
    ports:
      - "8761:8761"
  user:
    image: tc-demo/user
    ports:
      - "8083:8083"
    environment:
      EUREKASERVER_URI: "http://eureka:8761/eureka/"
      EUREKASERVER_PORT: "8761"
    restart: on-failure
```

Docker API



docs.docker.com/engine/api/latest

Exec example



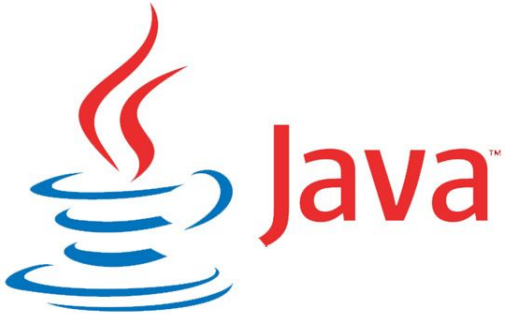
POST /containers/{id}/exec

REQUEST SAMPLES

```
{
    "AttachStdin": false,
    "AttachStdout": true,
    "AttachStderr": true,
    "DetachKeys": "ctrl-p,ctrl-q",
    "Tty": false,
  - "Cmd": [
        "date"
    ],
  - "Env": [
        "FOO=bar",
        "BAZ=quux"
    ]
}
```


TESTCONTAINERS

TestContainers flavors



TestContainers Java

- github.com/testcontainers/testcontainers-java
- Wraps **docker-java** library
- Docker environment discovery
- Host port randomization
- Containers clean up on JVM shutdown
- Readiness waiting strategies

As simple as

```
var redis = new GenericContainer("redis:5.0.6")
        .withExposedPorts(6379);


var postgres = new PostgreSQLContainer();
```

Docker environment discovery

```
Found Docker environment with local Unix socket (unix:///var/run/docker.sock)
```

```
[main] INFO  o.testcontainers.DockerClientFactory - Connected to docker:
  Server Version: 19.03.4
  API Version: 1.40
  Operating System: Docker Desktop
  Total Memory: 1998 MB
```

Talking to Docker via UDS

- curl --unix-socket */var/run/docker.sock* http:/localhost/**containers**/json

- curl --unix-socket */var/run/docker.sock* http:/localhost/**networks**

Host port randomization

- To prevent port conflicts
- Enables parallel builds
- API to get a host port

```
✗ docker ps

a950b04b6847        postgres:9.6.12
      0.0.0.0:32823->5432/tcp   mystifying_
```

Containers cleanup

```
[main] INFO  o.testcontainers.DockerClientFactory - Ryuk started
- will monitor and terminate Testcontainers containers on JVM exit


✗ docker ps

94b4792180c5        quay.io/testcontainers/ryuk:0.2.3   "/app"
minutes             0.0.0.0:32812->8080/tcp             testcontainers-ryuk-0
```

https://github.com/testcontainers/moby-ryuk

Waiting strategies

- Host port
- HTTP
- Log message
- Docker healthcheck
- Combined / Custom

Host port waiting strategy

- Default: at first exposed port with timeout of 60s

```
GenericContainer userContainer = new GenericContainer("tc-demo/user:latest");

userContainer.waitingFor(new HostPortWaitStrategy());
```

- Both from outside and **inside** container

Internal port check

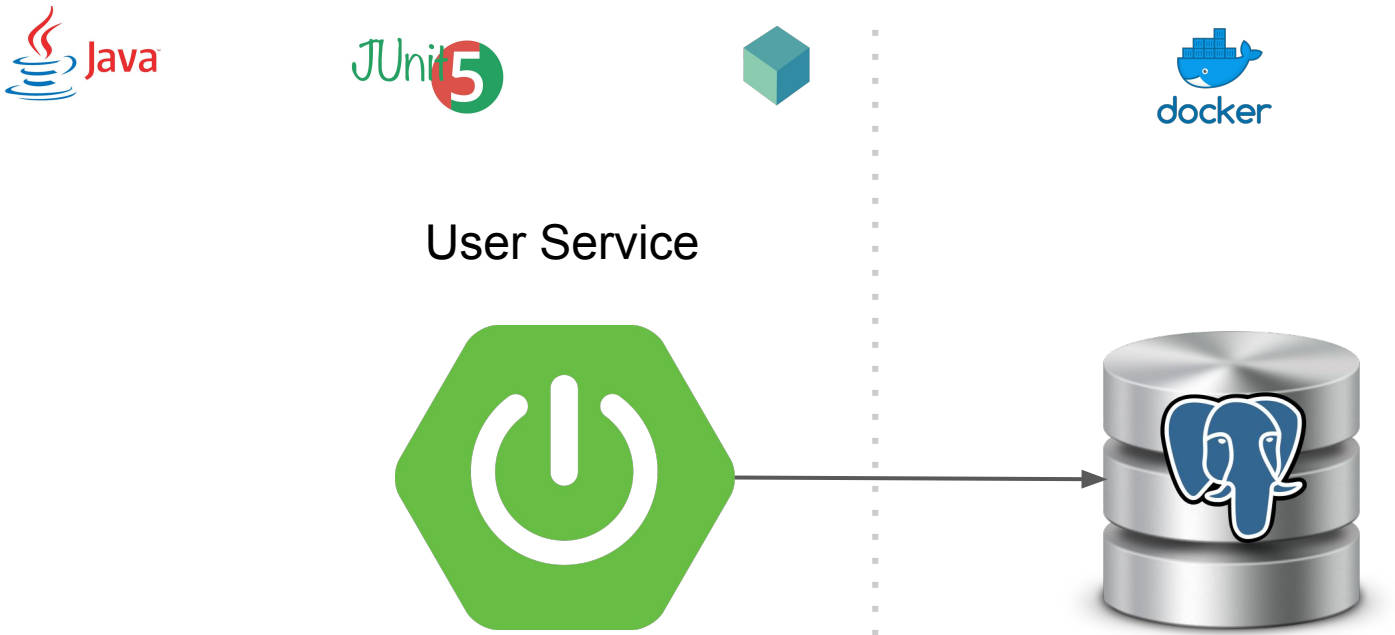
```
String command = "true";
for (int port : internalPorts) {
    command += " && ";
    command += " (";
    command += format("cat /proc/net/tcp{,6} | awk '{print $2}' | grep -i :%x", port);
    command += " || ";
    command += format("nc -vz -w 1 localhost %d", port);
    command += " || ";
    command += format("/bin/bash -c '</dev/tcp/localhost/%d'", port);
    command += ")";
}
```

HTTP waiting strategy

- Status & response body predicate

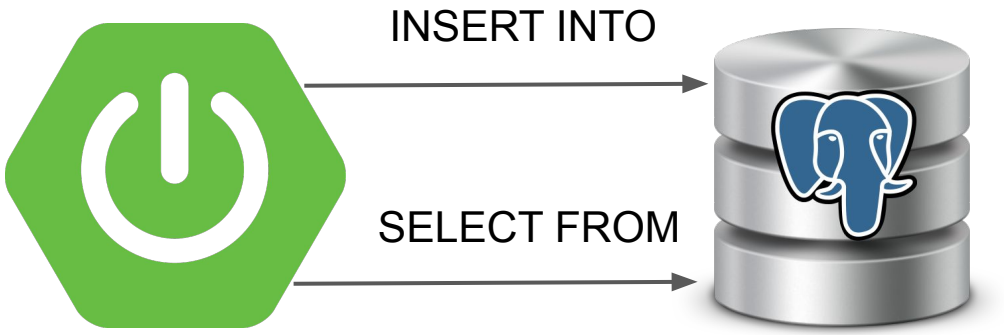
```
GenericContainer userContainer = new GenericContainer("tc-demo/user:latest")

HttpWaitStrategy httpWaitStrategy = new HttpWaitStrategy();
httpWaitStrategy.withStartupTimeout(Duration.ofMinutes(2));
userContainer.waitingFor(httpWaitStrategy.forPath("/actuator/health")
        .forStatusCode(200));
```

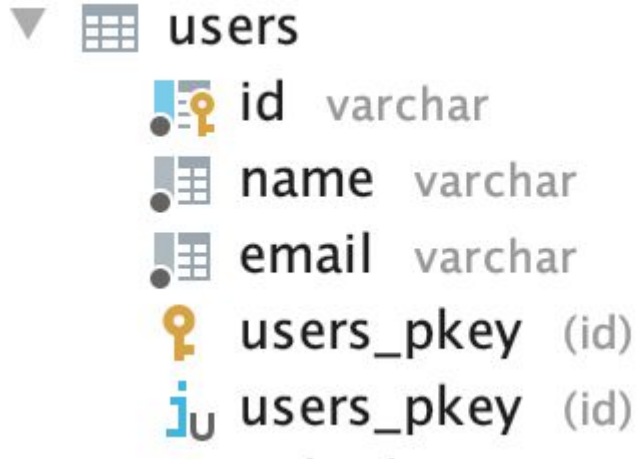
Demo setup



Demo scenario

User Service





Demo

github.com/nikolayk812/hjug-tc-demo

Demo recap

- JUnit 5 Extension API

- TestContainer modules

JUnit 5 extension points

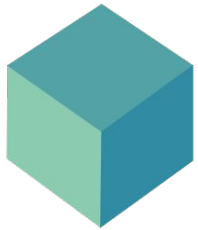
- Life-cycle callbacks
- Conditional execution
- Parameter resolution
- Exception handling

JUnit 5 extension logic

- Implement interface(s) from *o.j.j.api.extension* package
  - i.e. *BeforeEachCallback*, *ExecutionCondition*
- Register with *@ExtendsWith annotation*
- See *@Testcontainers* for reference

TestContainers modules

- Preconfigured, optimized for testing
- Wrappers on top of *GenericContainer* class
- 14 databases
- MockServer, LocalStack, Kafka, ToxiProxy

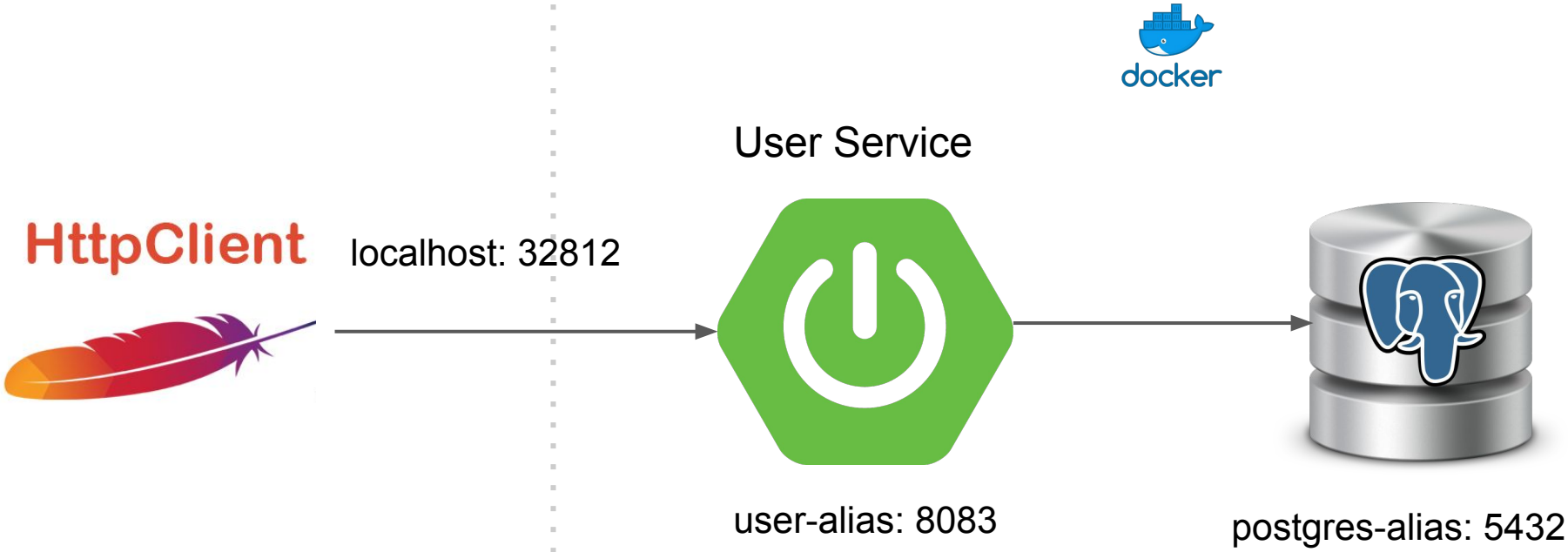


Demo-2: setup

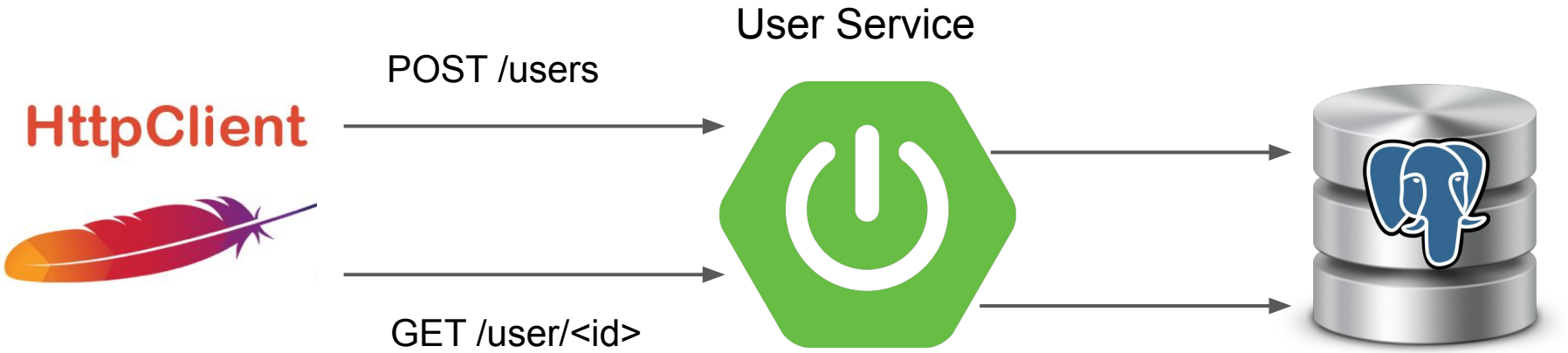
User Service

HttpClient

Demo-2: Docker network



Demo-2: scenario



Demo-2

github.com/nikolayk812/hjug-tc-demo

Demo-2: recap

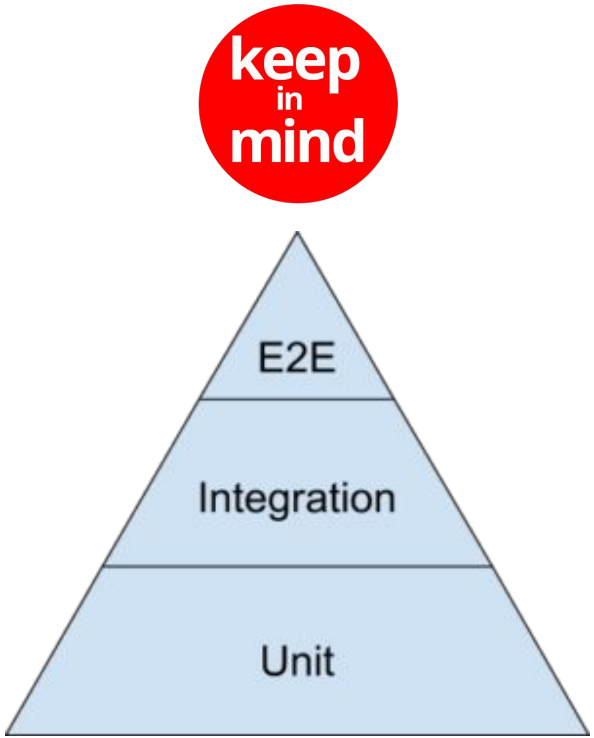
- Docker network and alias

```
new PostgreSQLContainer().withExposedPorts(5432)
        .withNetwork(Network.newNetwork())
        .withNetworkAliases("postgres-alias");
```

Why end-to-end testing?

- Business flows across **multiple** services

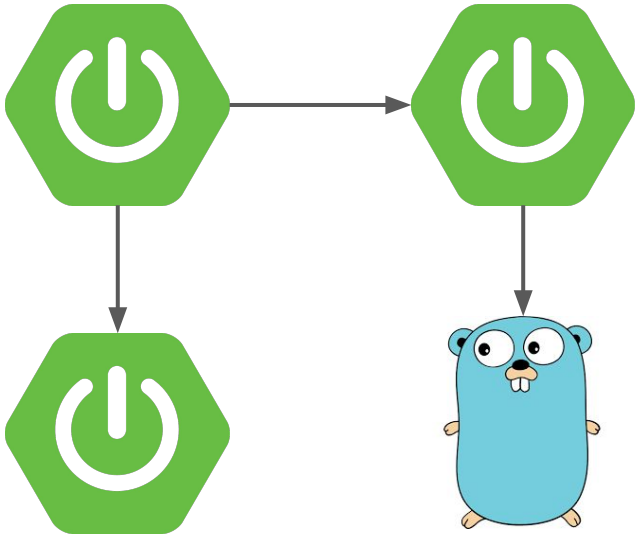
- Regression, when
  - \+ new service
  - \- legacy service



Some cluster

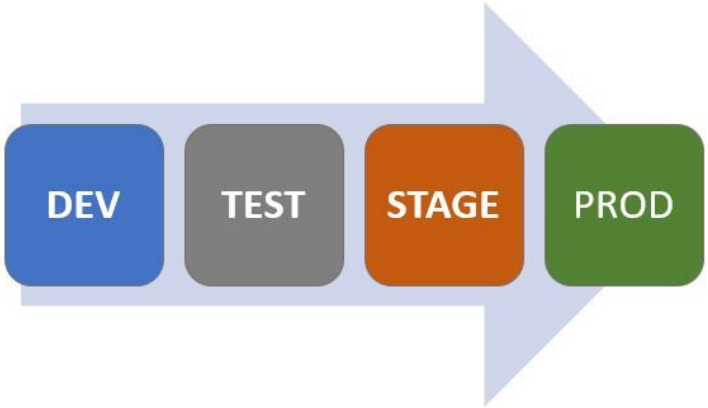
Kubernetes

Spring Cloud



E2E strategies

- Against a deployed cluster



- Against on-demand in memory cluster

Deployed cluster cons

- Replace a service with a newer version => **instability**
- Temporary service name => **non-discoverable**
- Unexpected databases states
  - Care to clear data after the test?

On-demand cluster cons

- Time to start all containers
- Memory + CPU
- How actually to create it?

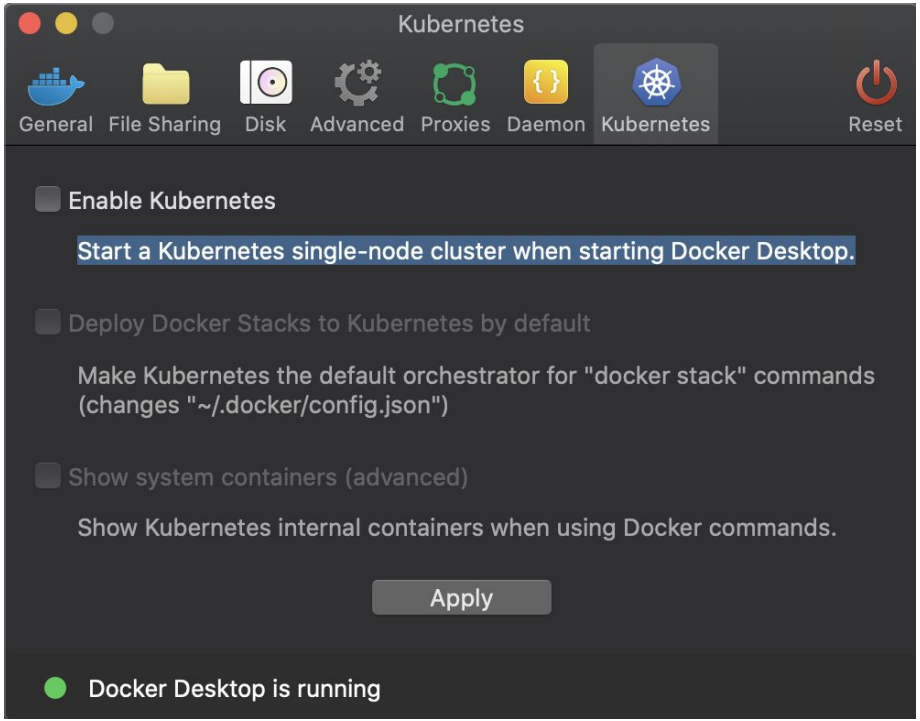
On-demand Kubernetes for E2E?

minikube

kind

K3S

MicroK8s

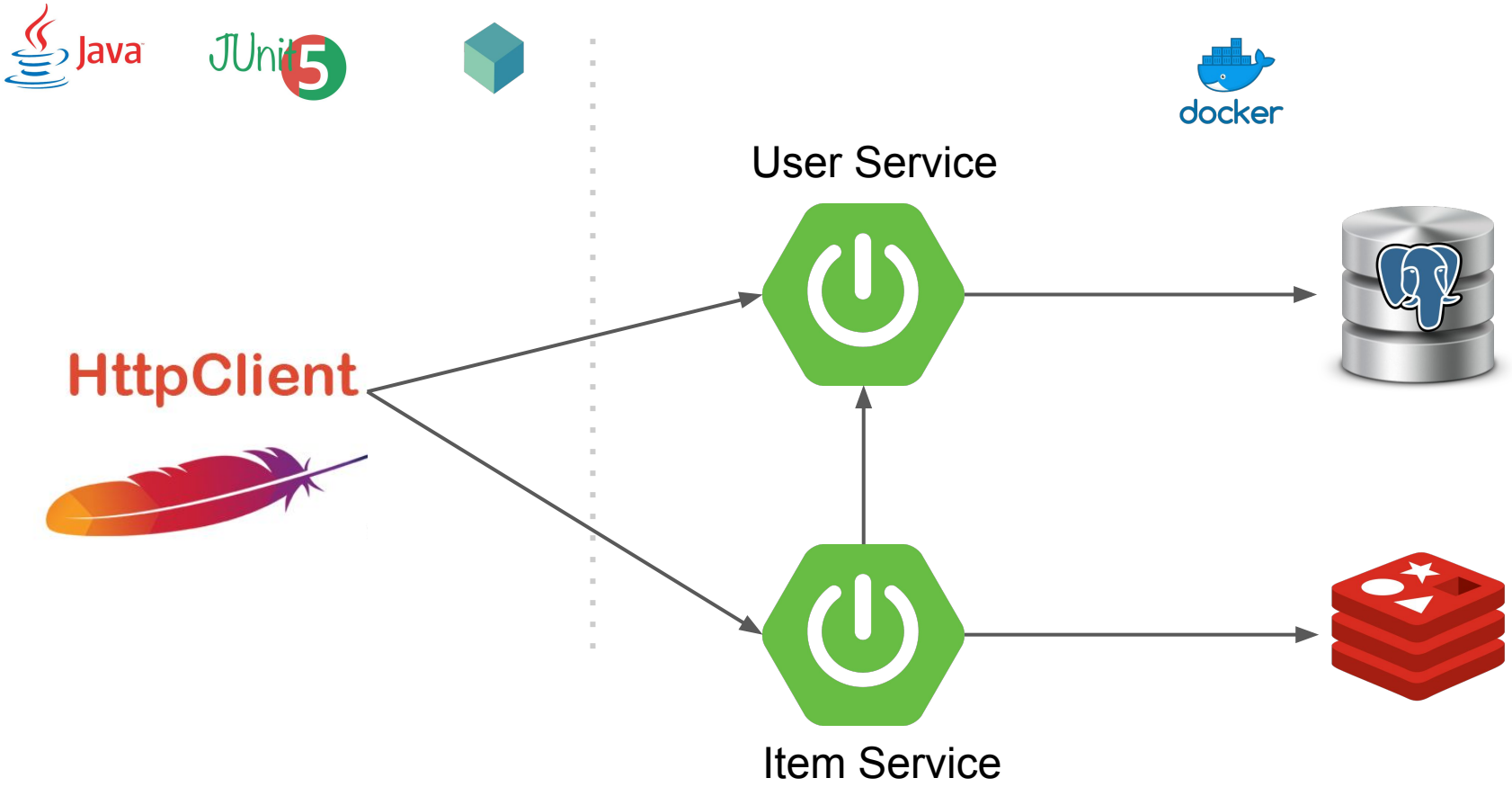


YAGNI

On-demand cluster TC approach

- Each service started by TestContainers
- Shared Docker network
- Functional tests
- Unless testing Kubernetes manifests

E2E setup



Hints

- Host port forwarding

  Testcontainers.exposeHostPorts()

- Fixed host port (for remote debugging)

  GenericContainer.addFixedExposedPort()

- Reusable containers

  github.com/testcontainers/testcontainers-java/issues/781

Takeaways

- https://testcontainers.org
- Balance between flexibility, speed and features
- Works on Mac, Linux, Windows
- Great for integration tests!
- Possible to use for end-to-end tests

Thank you!

@nikolayk812

nikolayk812

nikolayk812