**2. Implementation**

**Content**

* Preliminary
* 2.1 Workflow: Update of E2E Tests with Kubernetes
  + 2.1.1 Git-Repository
  + 2.1.2 Transform Tests to Docker Image and Push into Registry
    - 2.1.3.1 Start a Cluster Yourself
      * Prepare KOPS cluster state store
      * Den Cluster starten:
      * Cluster aktualisieren
      * Namespace anlegen
      * Delete a Namespace
      * Stop the Whole Cluster
    - 2.1.3.2 Recover already started Cluster
      * Create cluster
      * Set username and password
      * Connect cluster and credentials in a context
      * Change to the created context
      * View the current context
  + 2.1.4 Start Your Container
    - Deployments abstrahieren gut
    - Jobs sind gut für Tests
  + 2.1.5 Test ausführen
  + 2.1.6 Testergebnisse ablegen
    - Aufräumen
  + 2.1.7 Tear Down a Cluster / Namespace
  + 2.1.8 Fetch the Test Results
* 2.2 Workflow by Using the JMeter Framework vwg.ngw.tqa.taas
  + 2.2.1 Clone the vwg.ngw.tqa.taas repo
  + 2.2.2 Start the Kubernetes Cluster
  + 2.2.3 Calculating the Right Number of JMeter Threads/Users
  + 2.2.4 Start JMeter-Tools
  + 2.2.5 Stop JMeter-Tools
* 2.3 Running in AWS without Kubernetes
* 2.4 Environments
  + 2.4.1 Build pipeline devstack
  + 2.4.2 Runtime Environment AWS - Kubernetes
  + 2.4.3 Runtime Environment Minikube

**Preliminary**

As mentioned in the [the Vision](https://devstack.vwgroup.com/bitbucket/projects/NGW/repos/vwg.ngw.tqa.cloudtools/browse/docs/1-Vision.md) this document describes the necessary steps for the implementation of a standard workflow, namely: the update of E2E tests. Furthermore this documents serves to show deviatations from the initial idea of putting tests into the cloud.

The prior mentioned vision gives the rough idea of what to achieve. Thus, the implemention described in this document should match the vision and should be held accountable to achieving the vision.

By now, two ways of achiving tests with our current technology stack in the AWS crystalized out.

1. The Kubernetes Way
2. The Plain Docker Way

One way is the *Kubernetes way*. The advantage of scaling turned quickly out to be also of disadvantage when it comes to tests. A Kubernetes cluster is made for *KEEPING* things running, like Webservers or REST services. Tests on the other hand, are made to be run once so a report can be produced which is supposed to be stored, possibly for a longer time than the test itself. The JMeter tooling takes advantage of this power.

The other way is the *plain Docker way* in which an AWS service (the main cloud provider used by NGW) is used. This services runs under the name Elastic Cloud Service (or ECS). The Fargate service inside of ECS provides means of simple configurations in order to run Docker containers. This could be the very same docker images that could run under Kubernetes BUT more lightweight. E2E tests could be used here without the hussle of heavy configurative settings.

By now it seems that the E2E workflow with Kubernetes is too large compared to what should be achieved.

**Software Stack etc.**

On the surface this should work best in a Virtual Machine like [VirtualBox](https://www.virtualbox.org/).

Create in VirtualBox a VM with an [Ubuntu Linux image](https://www.ubuntu.com/download/desktop). If you are behind a Proxy server you probably have to deal with the [proxy settinlgs](https://askubuntu.com/questions/175172/how-do-i-configure-proxies-without-gui) inside of the VM.

If general problems occur with the network it may be necessary to activate NAT in Ubuntu

# open the NetworkManager configuration with the Vi editor

sudo vi /etc/NetworkManager/NetworkManager.conf

# Change this line:

dns=dnsmasq

# to this (note the hash comment)

#dns=dnsmasq

# quit with <Esc>:wq and restart the network service in the VM

sudo service network-manager restart

Once you can reach the internet this technology stack

* installed [Docker client](https://docs.docker.com/install/linux/docker-ce/ubuntu/)
* installed [AWS CLI](https://docs.aws.amazon.com/cli/latest/userguide/installing.html)
* installed [kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl)
* installed [kops](https://github.com/kubernetes/kops)
* working AWS permission (compare with those of the co-workers, see also below)
* cloned Git repository (done later)
  + install the Git client with
  + sudo apt-get install git

**AWS Permissions and CLI Configuration**

In order to gain full advantages later on it is necessary that the Volkswagen AWS administrator creates new AWS users in the image of existing TQA colleagues. These advantages provide f. i. better integration of existing AWS resources in terms of access permission (see 2.1.2)

1. Log in into the AWS web console.
2. Create/use an IAM permission group and attach these policies
   * Amazon EC2FullAccess
   * IAMFullAccess
     + previously, the Volkswagen AWS administrator may have to permit the usages of this policy
   * AmazonEC2ContainerRegistryFullAccess
   * AmazonS3FullAccess
   * AmazonRoute53DomainsFullAccess
   * AmazonVPCFullAccess
   * AmazonElasticFileSystemFullAccess
   * AmazonRoute53FullAccess
   * AWSCertificatemanagerFullAccess
3. create an CLI access key and secret access key for your account with this permissions
   * you may want to put those into your password safe application
4. Configure your AWS profile
   * aws configure
   * enter your CLI access key
   * enter your secret access key
   * enter your default region
     + eu-central-1
   * you can ignore the default output format or choose a format

**2.1 Workflow: Update of E2E Tests with Kubernetes**

**2.1.1 Git-Repository**

**Why?**

Everything is code

Tests, Documentation, Architecture interpretation, Configuration... All these things and more stay in a tight relationship of a specific version of the software.

If these single parts of a software don't come with the appropriate version, bad things can happen.

A database management software may spew errors if a software requests data from columns unknown to the database just because the query was built for a different software version. The same holds true for automated UI tests which runs into errors because the test wants to push a button that only exists in a different software version.

These errors are hard to tackle if one is debugging or tracing errors in a wrong software/documentation looking at wrong configuration data.

That is the very reason why *everything is code*: Software, Database schemas, Configuration data, permissions, Documentation.

When development partners or TQA coworkers push tests to the Git repo in order to add new tests, deleting old ones, or to update existing tests it really helps to have a history to trace changes and blame/praise 😉 the responsible people or to trace down errors.

**How?**

You can gain the correct git command from Bitbucket (check if the URL does not contain an IP address). Click the "clone" button in the side panel and select "HTTPS" (SSH might not work from inside of the Volkswagen network)

| **Cloning a Git-Repository** |
| --- |
| git clone https://vwuserid@devstack.vwgroup.com/bitbucket/scm/ngw/{repo-name} |
| The correct Git repo reference is still unclear :/ |

| **First Git configuration** |
| --- |
| git config user.name "Firstname Lastname" |
| git config user.email "your-volkswagen-email-address@volkswagen.de" |

E2E tests are saved, f. i. by development partner or by TQA - Save file and committed and syncronized with the current developement state. - git commit -am "Kommentar der Änderung" - git pull --rebase - Changes are pushed to the remote Git repo - git push

**2.1.2 Transform Tests to Docker Image and Push into Registry**

**Why?**

With or without cloud, the technology choice of containerization (f. i. Docker) is manifold:

* Containerization is state of the art for run-time abstraction
* Only little overhead of container service compared to a repeatable, identitcal environment
* Decoupling from the actual host operating system
* Focusses on the actual payload application
* Quick deployment, quick removal of no longer needed container
* Decoupling of services and libraries
* Software developers can gain production-ready software
* Better bug reproduction because of direct copy of the production run-time including the OS

Since the AWS Cloud with Kubernetes was chosen as infrastructure platform the choice of containerization is almost unavoidable. Even though there are containerization alternatives (containerd, moby with sudoless docker), the Docker universe is quite easy to get into.

Docker and the Docker ecosystem offers a well explored way of diving into containerization, like a large amount of ready-to-fetch images (reminder: containers are images at run-time) with different Linux versions and most-probably already installed software packages, f. i. PostgreSQL DB, nginx HTTP server, or Selenium UI test server.

The registry is an space where container images can reside. [The Docker Hub](https://hub.docker.com) is such a place. As we write very application specific test, use configurations with passwords, public spaces are *NOT* an appropriate choice as registry. Instead, AWS offers to create private registries for our Docker images, *Amazon Elastic Container Registry (ECR)*. We aim to create our own private registry where push and pull of our images are kept safe for strangers eyes.

**How?**

| **Notice to the AWS ECR Permission** |
| --- |
| AWS accounts should be created using the AWS space that is also *tequila-revision* associated with. This way there are no additional permission actions necessary, that is: coworkers can immediately work with the ECR (after log-in of course) |

| **One-time Creation of a Docker Registry in AWS** |
| --- |
| aws ecr create-repository --repository-name ngw-tqa-E2E tests |
| Result: JSON with ECR URL --> xxxxxxxxxxxx.dkr.ecr.eu-central-1.amazonaws.com/ |
| Current: 134852546396.dkr.ecr.eu-central-1.amazonaws.com/ |

1. Transform Tests to Docker Image and push it into ECR
   * Create image
     + docker build -t 134852546396.dkr.ecr.eu-central-1.amazonaws.com/E2E tests-ihdcc:sprint63 vwa-directory/
     + see ../tools/scripts/runvwa.docker.sh
     + if the image already exists than you can re-tag it easily
       - docker tag {YOUR\_IMAGE} {ECR-registry}/{YOUR\_IMAGE]:{TAG}
   * Log-in into AWS
     + aws ecr get-login --no-include-email
     + Copy and paste the returned command and log in
     + Even shorter (Please *DO NOT* use eval parenthesis inside the CI/CD chain as the result is immediately executed):
       - $(aws ecr get-login --no-include-email)
   * Push image to AWS ECR
     + docker push 134852546396.dkr.ecr.eu-central-1.amazonaws.com/ngw-tqa-E2E tests:sprint63

**2.1.3 Make Use of a Kubernetes Cluster**

**2.1.3.1 Start a Cluster Yourself**

**Why?**

Kubernetes clusters make up a way of scaling up and down single containers combined with the resources available from Amazon. Clusters build also a way of control about how to isolate or kill those containers. With this measures it is possible to kill amok-running pods, or create new pods in order to cushion high-demand situations (f. i. during performance tests). Scaling is realized by different concepts, in the first place by *Replicas*, that is pod clones. These pods can be distributed over different worker computers, *Worker Nodes* or just *Nodes*.

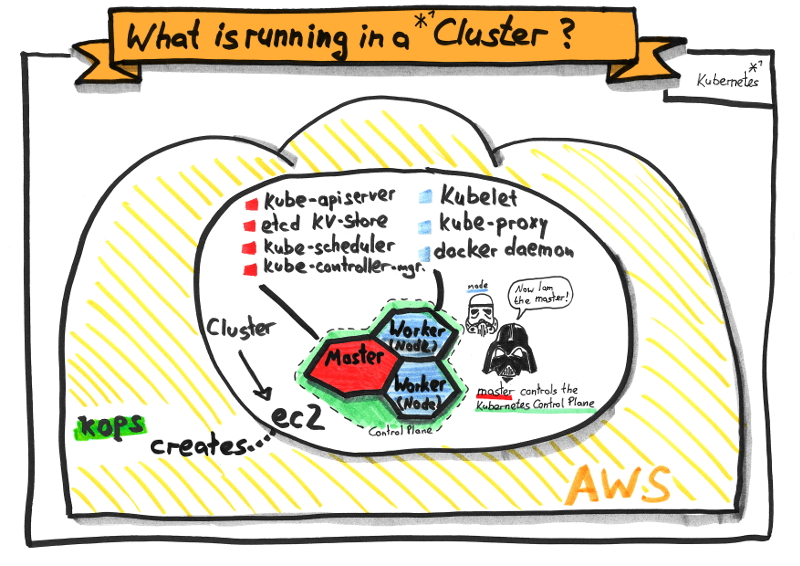
For ease of use, kops takes care for the heavy lifting of creation what we need: Kops will create an AWS EC2 instance and puts *Master* and *Nodes* into it, of which there are several Masters and Nodes possible.

The *Master* controls the Kubernetes Control Plane, which makes up the Kubernetes base. It consists of important services like:

* the kube-apiserver
* the Key-Value-Store etcd
* the kube-scheduler
* the kube-controller-manager
* all services of the Worker Nodes (everything is a Pod)

which just enables the complex interaction between all these different objects within a cluster.

In order to know the current state, each Kubernetes cluster needs a separate S3 bucket in order to store the cluster state.



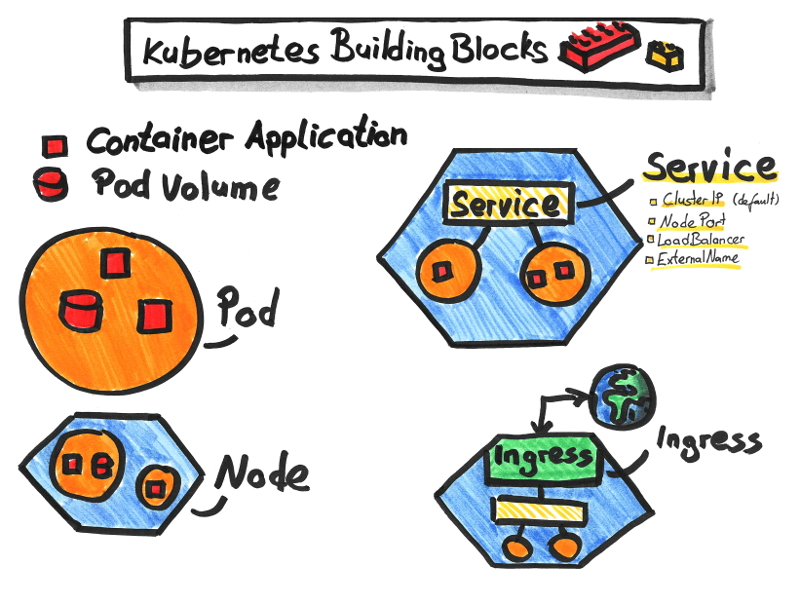
*Nodes* represent virtual worker machines, on which the work is done inside pods. Nodes are controlled by *Masters* (of which there may be more than one if you plan High Availability). In order to iterate on the work load, every node contains the same basic (though vital) programs like:

* kubelet (controls the concrete Node at hand)
* kube-proxy and
* a Container control
  + like Docker
  + (but there are more Container services besides Docker)

*Pods* represent the smallest possible and also the most important abstraction layer in Kubernetes (everything is a pod). This is the place the desired containers with the payload application do run and volumes can be mounted. *Volumes* can be used as data exchange medium between containers in the same pod. Regarding their lifecycles they might outlive running containers but this is seldomly true because when all containers cease to exist inside the pod, the pod itself is closed-down as well. Besides that there are some volume drivers where the volume inside the pod is merely mapped from the outside (f. i. special volume drivers like AWS EBS drivers). In this case the volume outlives even when the pod exited.

Inside of a pod the running containers share the same network which is isolated to the outside by default.

| **Kubenetes Object** | **Short Explaination** |
| --- | --- |
| Pod | Runtime abstraction consisting grouping one or more containers. |
| Node | Abstraction of an executing computer/machine. On this level Kubernetes binaries and pods are running. |
| Master | Like a Node. Furthermore, one or more master controls associated nodes in a high-available fashion. |
| Volume | A way of data storage inside of a pod which may be provided to all containers within that pod. |
| Service | A defined network behavior which enables internal container discovery beyond a single pods. |
| Ingress | A defined network behavior which enables external access from the Internet towards pods and thus containers (usually by means of Services). |
| Label | A label is a Key-Value pair which is certain way of addressing Kubernetes objects of all types. |
| Deployment | An organizational unit which provides high availability and a defined way of migration for pods by Replica Sets. |
| ReplicaSet | A lower organizational unit which goes great lengths to keep a defined number of Replicas running. If a pod exits, it will re-created. |
| Replica | 100 % Clone of a pod including containers and volumes which can be used for load balancing. |
| Job | Another organizational unit (in contrast to a Deployment) which does NOT re-create pods after their exit. |



Kubernetes [*Services*](https://kubernetes.io/docs/concepts/services-networking/service/) are means for pod-to-pod communication. Services are not real things, not even imaginary things. There is no pod or direct binary involved. Rather, a Kubernetes Services is a network behaviour. As a metaphor, imagine a rubber stamp which is pressed on a node. What's left is a defined network behaviour between pods.

There are no outbound communication restrictions. If the container allows it the application may communicate unrestrictedly with the internet. On the other hand, by default only inside the same pod containered applications can communicate with each other. There is now default communication way from outside inbound into the pod.

In order to tackle this situation there are [*Ingresses*](https://kubernetes.io/docs/concepts/services-networking/ingress/).

Ingresses are similar to services as Ingresses aren't real things or binaries either. An ingress is a abstracting, defined way of network behaviour, as well. In contrast to services, an ingress defines the way of how the outside world (the Internet) can communicate with pods. Ingresses can map communication ways to pods so not all pods are reachable for that configured way.

[*Proxies*](https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#so-many-proxies) help to distribute network communication from the pod outbound towards the internet.

**Overview of Commands**

| **Command** | **Way-to-short Explaination** |
| --- | --- |
| kubectl | Operates on Kubernetes objects |
| kops | Operates on Kubernetes Clusters |
| aws | Operates on AWS objects like S3, EC2, IAM, ... |

**How?**

We use Kops in order to create our clusters because it

* conveniently sets up an AWS EC2 instances
* builds the cluster nodes on top of it
* is supported by the Kubernetes team.

Kops needs an S3 bucket in which the Kops state is stored. This needs to be done only once per cluster. The cluster can be closed down and re-created with the same Kops state store.

your\_s3\_bucket\_name=$

source {YOUR\_CLUSTER\_NAME}

aws s3 mb ${your\_s3\_bucket\_name}

export KOPS\_STATE\_STORE=s3://${your\_s3\_bucket\_name\_here}

It is a good idea to save the exported environement variables inside of a source-able file, like this aws.env.

# put this into a file named aws.env

export CLUSTER\_NAME=jmeter-awsa

export AWS\_DEFAULT\_REGION=eu-central-1

export AWS\_PROFILE=default

**Start the Cluster**

This part of the document describes the manual start of a Kubernetes cluster. There is a scripted version using Makefiles described in the chapter "2.2.2 Start the Kubernetes Cluster".

kops create cluster \

--cloud aws \

--kubernetes-version ${KUBE\_VERSION}\

--zones eu-central-1a,eu-central-1b,eu-central-1c \

--name ${NAME} \

--node-count ${KUBE\_NODE\_COUNT} \

--node-size ${KUBE\_NODE\_TYPE} \

--master-count ${KUBE\_MASTER\_COUNT} \

--master-zones eu-central-1a \

--master-size ${KUBE\_MASTER\_TYPE} \

--networking weave \

--cloud-labels "InspectorScan=${AWS\_INSPECTOR},OwnerName=${TAG\_SYSTEM\_OWNER},OwnerEmail=${TAG\_SYSTEM\_OWNER\_EMAIL}" \

--image ${AWS\_AMI}

The variables have the following meaning

| **Switch or Variable** | **Explaination** |
| --- | --- |
| --kubernetes-version | Version of the used Kubernetes Binary. The client versions should match with API version of the cluster, so the client calls are executed in a predictable way. Ex. 1.8.8 |
| ${AWS\_DEFAULT\_REGION} | AWS region in which the EC2 instance should be executed which will hold the cluster. Because of data privacy reasons we MUST uset eu-central-1 because this region is located in Germany. |
| --zones | Zones (somewhat like sub-regions) in which *worker nodes* are supposed to run. |
| --name | Name of the cluster including AWS namespace URL. Helps setting up and identifying your cluster in contrast to others clusters. Ex. tqa-test-cluster.tequila-revision.de |
| --node-count | Number of *worker nodes* which are supposed to make up the cluster (don't mix this up with replicas, we aren't in the same abstraction level here) |
| --node-size | Name of an [AWS instance type](https://aws.amazon.com/de/ec2/instance-types/) for each node which makes up the computing power. Ex. t2.medium |
| --master-count | Number of *master nodes* which are supposed to make up the cluster. We don't need ultra high availability, this can be 1 master. Ex. "1" |
| --master-zones | Zones (somewhat like sub-regions) in which master nodes are supposed to run. |
| --master-size | Name of an [AWS instance type](https://aws.amazon.com/de/ec2/instance-types/) for each node which makes up the computing power. Ex. t2.medium |
| ${AWS\_INSPECTOR} | Associates a label whether an AWS inspector should check the operation system security (ex. checks installed packages). |
| ${TAG\_SYSTEM\_OWNER} | Associates a label to the contract holder of this AWS instanz group (usually Thole Gröneveld who pays the bill). |
| ${TAG\_SYSTEM\_OWNER\_EMAIL} | Associates a label to the contract holder of this AWS instanz group (his email) |
| --networking weave | Creates an [Overlay Network](https://kubernetes.io/docs/concepts/cluster-administration/networking/) which spans all pods. |
| --image | AMI: Amazon Machine Images. Reference to a Linux image which is the base for this EC2 instance. Ex. 099720109477/ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-20180306 |

Kops comes with standard internet services like DNS or an [Overlay Network](https://kubernetes.io/docs/concepts/cluster-administration/networking/) (our case: weave).

To show the currently running clusters enter the command kops get cluster. to show and edit the configuration of a running cluster enter the command kops edit cluster ${NAME}.

**Update the Cluster**

In order to finalize the start of the cluster the cluster needs to be updated one more time in order to [synchronize the cluster state](https://github.com/kubernetes/kops/blob/master/docs/cli/kops_update_cluster.md). This action confirms the current cluster configuration and writes it also into your local system.

kops update cluster ${NAME} --yes

This may take a couple of minutes up to an hour.

**Create a Namespace**

[Namespaces](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/) provide another isolation level. Even though the metaphor is not as preices as it should be, one can imagine a Kubernetes cluster namespace as a kind of a *virtual cluster*. Unlike creating the cluster (which may take up to several minutes until everything runs properly) you can easily crate a namespace in a couple of seconds' time.

It is important to note that there is a default namespace. While it is possible to work inside it, it is *considered an inappropriate practice*. The reason the default namespace for it is a kind of convenient security measure so that commands are still executed even when there is no proper namespace at hand.

Instead you should *ALWAYS* create a namespace so the pods can be handled with better care.

kubectl create namespace YOUR\_NAMESPACE

kubectl apply -f deindeployment.yaml -n YOUR\_NAMESPACE

There is no way of logging-in into a namespace. Instead kubectl accepts the general option -n YOURNAMESPACE for every command (thus *general* option).

Grasping this concept is of high importance: The person issuing kubectl commands needs to be aware of which namespaces exist and which commands should apply to which namespace.

| **/!\ Warning /!\** |
| --- |
| You *MUST NOT* temper with the internal Namespaces kube-system and kube-public |
| Seriously, *do not*. |

**Delete a Namespace**

The good thing is the ease of use once a namespace is used. Starting a namespace and deploying containers is easy. The benefit of namespaces is the level of isolation. When containers run amok, just delete them! Everything else will be still running as before. Yay!

kubectl delete namespace YOUR\_NAMESPACE

This will take just a couple of seconds.

**Stop the Whole Cluster**

Every now and then the whole cluster is not needed anymore or should be re-created in order to be a moving target against cyber-criminals and other cheeky people. Of course a stopped cluster will not induce any more costs on the AWS bill.

Stop a running cluster like this:

kops delete cluster ${NAME} --yes

This may take a couple of minutes until everything is teared down.

**2.1.3.2 Recover already started Cluster**

**Why**

The mundane task of operating with a cluster is easy for the person who started the cluster in the first place. But when the person who started the cluster is not available, say for reasons of

* personal sickness or
* going on vacation or
* what other reasons out there

then someone else has to deal with this cluster. This incorporates the whole range of checking liveness, modifying cluster properties, or shutting down the whole thing.

**How**

Result of kubectl config view --minify (Certificates are removed in favor of User/Password).

apiVersion: v1

clusters:

- cluster:

certificate-authority-data: REDACTED

server: https://api.tqa-test-cluster.tequila-revision.de

name: tqa-test-cluster.tequila-revision.de

contexts:

- context:

cluster: tqa-test-cluster.tequila-revision.de

user: tqa-test-cluster.tequila-revision.de

name: tqa-test-cluster.tequila-revision.de

current-context: tqa-test-cluster.tequila-revision.de

kind: Config

preferences: {}

users:

- name: tqa-test-cluster.tequila-revision.de

user:

password: GET FROM THE ENVIRONMENT

username: GET FROM THE ENVIRONMENT

Alternatively, this can be configured by issueing a different kubectl command:

**Create cluster**

export NAME=YOUR\_CLUSTER\_NAME

export KUBE\_API=https://api.tqa-test-cluster.tequila-revision.de

export CLUSTER\_USERNAME='yourusergoeshere' #GET FROM THE ENVIRONMENT

export CLUSTER\_PASSWORD='yourpasswordgoeshere' #GET FROM THE ENVIRONMENT

kubectl config set-cluster ${NAME} \

--server=https://${KUBE\_API} \

--insecure-skip-tls-verify=true

**Set username and password**

kubectl config set-credentials ${NAME} \

--username=${CLUSTER\_USERNAME} \

--password=${CLUSTER\_PASSWORD}

**Connect cluster and credentials in a context**

kubectl config set-context ${NAME} \

--cluster=${NAME} \

--user=${NAME} \

--namespace=default

**Change to the created context**

kubectl config use-context ${NAME}

**View the current context**

kubectl config view --minify

**2.1.4 Start Your Container**

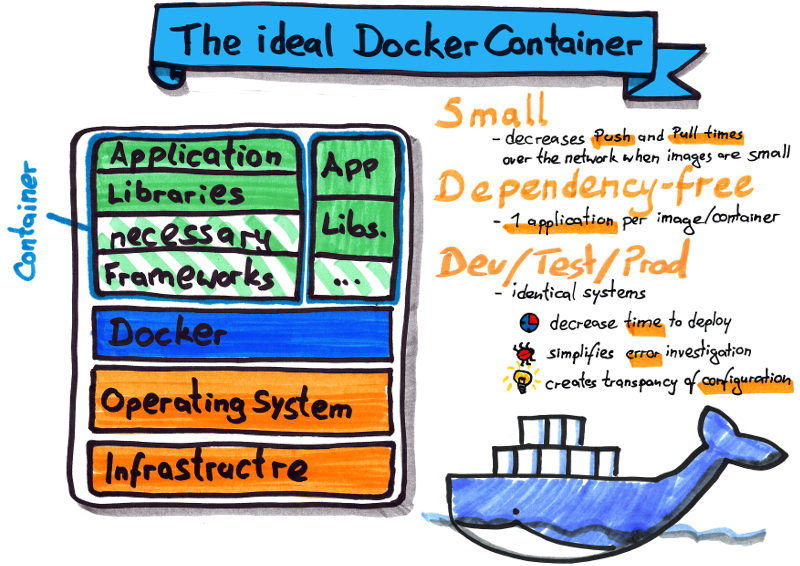
**Why?**

The reasone why we are here is that we have a payload to deploy. In the above chapters a lot of preconditions were solved in order to start (or recover) cluster. Now comes to real stuff (imaginary spoken).

To deal with one problem of computer history is to decouple applications which introduces dedicated interfaces. Containerization continues this idea. There should be exactly one application per container to reduce complexity and dependency. Each container should be testable on its own.

Here are some Best Practices for containered applications:

* as small as possible
  + large images increase push and pull durations
  + produce unnecessary load on the network
  + create your own base and sub images were senseful
    - base images (FROM image:tag) usually change slower at a slower rate than sub-images
    - such base images can be cached on infrastructure level and create less load
* as independent as possible
  + exactly 1 application (including direct dependencies)
    - libraries
    - other necessary binaries and stuff like that
  + reduce cross-dependencies
  + show up unknown dependencies
  + Unix tool philosphy: Write programs that do one thing and do it well
* Parity of Dev/Test/Production
  + one image should be used in all environments
  + reduces the time to identify defects
  + puts focus on transparency in configuration



Now that the payload is roughly described the next part is about the actual container start. That is an image is about to be run (as containers are running images).

Pods are the smallest level of abstraction in Kubernetes. But Kubernetes was designed with high availability in mind. This philosophy contradicts with the idea of tests which contain a dedicated finish (in order to create a test result).

| **Notice** |
| --- |
| Do not deploy containers on [pod level](https://kubernetes.io/docs/concepts/configuration/overview/) (kubectl apply -f container-pod.yaml). |

This is bad practice. Starting containers should be done by [Kubernetes Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) or by [Kubernetes Jobs](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/) which declare the desired state in a more proficient way (including the number of replica).



*/!\* Please refer to some further configuration [Best Practices](https://kubernetes.io/docs/concepts/configuration/overview/) aggregated by Kubernetes. Their principles are pretty excellent.

**How?**

The general idea is to let run directly dependend containers in the same pod. This is (again) done with kubectl.

As in other areas the YAML file format is used which describes the final state (which should be reached) of the target system YAML is as well used for Kubernetes. Vital part is the Docker image which is supposed to be the payload. Of course there lots of options more to fine adjust a configuration.

kubectl run your-deployment-name --image=link-to-your-docker-image:tag --port=8080

Since Kubernetes comes with a Docker daemon there is no explicit docker run command necessary because it lays in the nature of Kubernetes to run containers. Instead the Docker image reference comes into the YAML file like this:

apiVersion: v1

kind: Pod

metadata:

name: your-super-application

labels:

app: web

spec:

containers:

- name: front-end

image: nginx

ports:

- containerPort: 80

- name: your-api-application

image: your-docker-registry/your-docker-image:tag

ports:

- containerPort: 88

*Don't do pod yamls. Better do Job or Deployment yamls*

**Deployments abstract well**

[Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) are a great tool to keep software running in a scalable way. As mentioned before, creating pods by themselves (on pod level) is no good because

* this approach does not scale
* it is painstaking legwork.

It is *better* to create pods by means of *Deployments* which make implicitly use of *ReplicaSets*. The structure is only a tiny bit different to a direct pod YAML but it is worth the work.

apiVersion: apps/v1beta1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.7.9

ports:

- containerPort: 80

**Jobs are good for Tests**

The good thing about Deployments is that they restart pods in which the containers exit. This self-healing power is one of the strengths of Kubernetes:

1. Pod is created
2. Container is started
3. Test runs
4. Test ends (usually by reaching the end, or by cancelling)
5. Container ends
6. Pod ends
7. Kubernetes sees difference in number of pods running and starts Pod again. (Go to 1)

But this behaviour contradicts the exact purpose of containers running tests! When a test container stops it will just be restarted. FOREVER!

Thus tests have a different lifecycle than typical service applications in Kubernetes.

Given all this information, there is good news! There is a Kubernetes concept of [Jobs](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/).

Jobs create pods which may run by setting the key restartPolicy exactly one time. After their exit the pod stays down no matter what.

If the job contains the attribute .metadata.name then the job will not be started again. Kubernetes does not see a necessary change of its behaviour *as it exited and that's what a job is all about*. The job has to be created with a different name in order to have it executed again (but then only once again).

It is possible to start jobs any time again without getting into the Catch 22. Instead, using .metadata.generateName does the trick because for each new execution (preferably with kubectl create) a job with a *new* name is created. This job will execute exactly once and finish.

apiVersion: batch/v1

kind: Job

metadata:

generateName: test-

labels:

testType: e2e

spec:

template:

spec:

containers:

- name: your-test-application

image: your-docker-registry/your-docker-image:tag

restartPolicy: Never

backoffLimit: 4

kubectl create -f test-job.yaml

This practice has a minor disadvantage though. The list of finished jobs may grow quite large if the namespace lasts long. If the namespace is deleted every swiftly everything within will be delete along the list of finished jobs. But if the namespace is not deleted then the list of finished jobs SHOULD be pruned on a regular basis.

Senseful parameter are

* jobs age
* whether the job has run successfully

To clean up the list of finished jobs can be done like this:

kubectl delete jobs -l testType=e2e -n YOUR\_NAMESPACE

Please note that -l testType=e2e references a label that is given in the YAML above. The label is used to address Kubernetes objects.

**2.1.5 Run the Test**

**Why?**

Well, you got me there.

**How?**

With help of your Docker container. You know that.

**2.1.6 Store Test Results**

**Why?**

We need to know how the test did fail or succeed, or did not run at all. Only this way the quality of the system under test can be measured. This store is named a Test Result Repository

**How?**

The [AWS Service S3](https://docs.aws.amazon.com/us_en/AmazonS3/latest/dev/BucketRestrictions.html) is a pretty good start to have data stored over a longer period of time. While a bucket must be created with a fixed capacity, after that it is still possible to move S3 objects to different bucket with a larger capacity.

In order to find your test results in a possible large bucket quickly the need of an classification system arises:

A proposal would look like this:

* Test Category
  + LuP (Load and Performance Test)
  + E2E
  + Integration
  + Unit
  + ???
* Instant of when the test was started
  + preferably with [ISO 8601 Dates](https://en.wikipedia.org/wiki/ISO_8601)
  + modified Format: YYYY-MM-DD\_hh\_mm\_ss
    - zero padded for single digit parts
    - some file systems do not support colons ":"
  + ISO dates are sortable from Most Significant to Least Significant date part
  + Dates shall be corrected to UTC/GMT (note the "Z") timezone for comparison
  + everywhere is "Now" regardless of timezone
  + less problems with timezone math
  + Example: 2018-11-31-14:59:12Z
* Used test technology
  + only needed for necessary discrimination
    - LoadRunner (does LuP)
    - JMeter (does LuP as well)
* The tested product
* The inherent test report
  + test report summary
  + images
  + stacktraces
  + miscellaneous data and files

For instance this could be a well-populated Test Result Repository applied to a hierarchic file system

* iHDCC/
  + LuP/
    - 2018-11-31\_12\_34\_56Z/
      * index.html
      * outliers.html
      * graph1.png
      * graph2.png
    - 2018-12-05\_10\_01\_03Z/
      * index.html
      * outliers.html
      * graph1.png
      * graph2.png
  + E2E/
    - 2018-11-30-09\_52\_38/
      * TestReport.html
* DCC/
  + ...

Alternative ways is to upload zipped archives if it is too overblown to upload a whole directory structure. Keep in mind that this zip file needs to be created in the first place.

**2.1.7 Tear Down a Cluster / Namespace**

**Why?**

A running Cluster costs money because of CPU usage, even though a Kubernetes cluster will not produce a lot of load (and thus will not burn too much money).

Also with the running time, the probability of provoking a security breach is more likely.

**How?**

This statement deletes the namespace but keeps the cluster itself and other namespaces intact:

kubectl delete namespace YOUR\_NAMESPACE

This statement deletes the cluster and every thing on it.

kops delete cluster ${NAME} --yes

For context see 2.1.3 / "Stop Cluster"

**2.1.8 Fetch the Test Results**

**Why?**

Tests should carry a meaning. (If they do not, delete them).



If there is no result, you have no way of building up an idea what's going on, and the tests are even more meaningless.

**How?**

Once a report is put in a AWS S3 bucket it could be fetched either by the AWS frontend

**2.2 Workflow by Using the JMeter Framework vwg.ngw.tqa.taas**

By principle, TaaS (Testing as a Service), a workflow adapted to a JMeter framework is quite similiar to the already described in 2.1.

TaaS creates multiple pods for JMeter workers and JMeter master (do not mistake them for Kubernetes master and nodes). The JMeter master controls the workers to create the actual load.

Check also the TaaS [documentation](https://devstack.vwgroup.com/bitbucket/projects/NGW/repos/vwg.ngw.tqa.taas/) out.

**2.2.1 Clone the vwg.ngw.tqa.taas repo**

You know the drill. The [TaaS repo](https://devstack.vwgroup.com/bitbucket/projects/NGW/repos/vwg.ngw.tqa.taas/) needs to be cloned first, as seen on 2.1.1.

**2.2.2 Start the Kubernetes Cluster**

See chapter 2.1.3, but do not create a namespace. Here, this is done by a script. Of course you need a fully configured client AWS profile as well as the software stack mentioned in the start of this page.

Alternatively, you can clone the vwg.kube.aws repo, do the configuring stuff like this

git clone https://{YOUR\_USER}@devstack.vwgroup.com/bitbucket/projects/NGW/repos/vwg.kube.aws.git

# setting the configuration

export CLUSTER\_NAME=jmeter-awsa

export AWS\_DEFAULT\_REGION=eu-central-1

# default profile only if you don't have other profiles

export AWS\_PROFILE=default

Create a makefile jmeter-awsa-config.mk in vwg.kube.aws/cluster\_config/ with this content. Replace user names / emails where appropriate.

export KOPS\_STATE\_STORE ?= s3://tqa-jmeter

export DNS\_ZONE ?= tequila-revision.de

export AWS\_AMI := 099720109477/ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-20180306

export AWS\_INSPECTOR := false

# -- cluster configuration

export KUBE\_MASTER\_COUNT := 3

export KUBE\_NODE\_COUNT := 3

export KUBE\_MASTER\_TYPE := t2.medium

export KUBE\_NODE\_TYPE := t2.2xlarge

# -- to access grafana/kibana etc.

export INGRESS\_USERNAME ?= {something with common sense}

export INGRESS\_PASSWORD ?= {something with common sense}

# -- csv of namespaces

export NAMESPACE\_LIST ?= ngwsa-performance

export LEGO\_URL := https://acme-v01.api.letsencrypt.org/directory

export LEGO\_EMAIL := andre.weigelt@volkswagen.de

export TAG\_SYSTEM\_OWNER := Your name

export TAG\_SYSTEM\_OWNER\_EMAIL := Your email address

Then run it with make install or tear it down with make uninstall

The cluster will be created from this TaaS script:

./jmeter-cluster.sh

In order to have visible results Grafana will be deployed along with an Influxdb instance where the test result reside:

./dashboard.sh

**2.2.3 Calculating the Right Number of JMeter Threads/Users**

By default there are 3 worker node replicas providing the actual load, and 1 master in which the controlling takes place. If your JMX plans to simulate 50 Users/Minute it actually breaks down to:

50 Users/Min \* 3 Worker Nodes = *150 Users/Min*

Depending on the test use case, it may be necessary to cut down the *number of replicas*, or the number of users in the JMX, which is distributed among the worker nodes.

**2.2.4 Start JMeter-Tools**

Provide your ./jmeter-start.sh

**2.2.5 Stop JMeter-Tools**

./jmeter-stop.sh

**2.3 Running a Plain Docker Container in AWS**

AWS / ECS / Fargate

* Get started
* Configure Image, Memory, ENV variables
* Configure Task Definition with TaskRole
* Re-configure TD with to use S3 Role (unconfigurable before inside of Get Started)
* Go get 'em Tiger

**2.4 Environments**

There exist the following environments which are stacked on each other. Minikube makes a special case here because it is mainly used as local Kubernetes playground.

**2.4.1 Build Pipline devstack**

Any code will be hosted in private Git repositories under [devstack.vwgroup.com](https://devstack.vwgroup.com/bitbucket/projects/NGW). Furthermore, under Devstack the whole Atlassian toolchain is available which covers version control, build server, and way more tools.

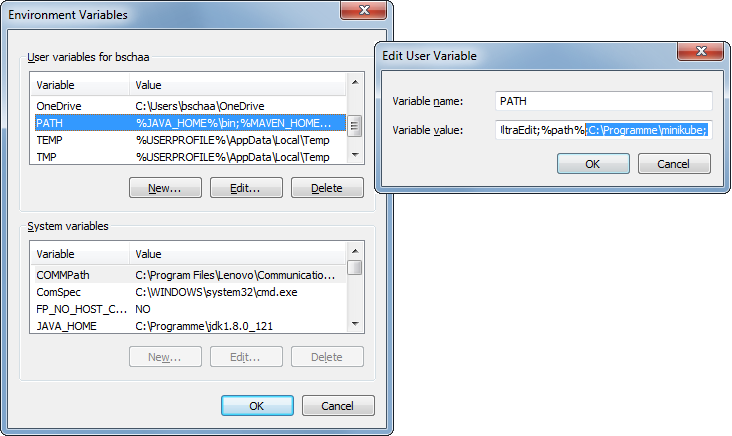
**2.4.2 Runtime Environment AWS - Kubernetes**

To date AWS does not support native Kubernetes beyond a certain US region (which is out of scope for our purpose). Besides Kubernetes on a EC2 instance, AWS supports with its ECS/Fargate product plain Docker container execution. This idea is described in the chapter *2.3 Running a Plain Docker Container in AWS*.

**2.4.3 Runtime Environment Minikube**

[Minikube](https://github.com/Kubernetes/minikube) provides a local Kubernetes environment for exactly one cluster consisting of one master and no additional worker nodes. This is possible because the Master contains all necessary binaries to play the role as worker (because even in *kube-system* everything is a pod). Often VirtualBox is the mean of virtualization.

In order to have Minikube running these things are necessary (the Minikube and Kubectl binaries are binaries which are portable and do not need to be installed with admin permissions):

* x64 Architecture computer
* Of course you need some computer memory because VirtualBox and so on...
* Activated VT-X switch in the computer's UEFI/BIOS ()
* Installed VirtualBox
  + Minikube makes use of VirtualBox. Minikube itself runs on Windows
* Installed Minikube binaries
  + this works under Windows
  + not possible inside of VirtualBox
  + [Download current binary](https://github.com/kubernetes/minikube/releases/)
  + put binary into a directory you can find easily, f.i. C:\PortableProgs\Minikube\
  + rename Binary to minikube.exe
  + create a [SHA-256](https://devstack.vwgroup.com/jira/browse/SHA-256) hash of the minikube.exe file
  + compare your hash with the given hash from the Minkube website (they must match)
  + Git-Bash provides a sha256sum binary but there are others...
    - sha256sum minikube.exe
  + Really Important: Minikube should be stored on the C: diskdrive
    - There is a kind of bug which makes it hard to run on a different drive
* Installed kubectl
  + [check the current version](https://storage.googleapis.com/kubernetes-release/release/stable.txt)
  + [download current binary](https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-binary-using-curl)
    - check the Windows tab to see the installation hints for Windows
  + put the binary into the same directory as Minikube, f.i. C:\PortableProgs\Minikube\
* Edit your Windows Environment variables, add said directory to the PATH variable
  + a **semicolon** (";") is a delimiter to have Windows search through multiple directories for executables
  + add a semicolon and put the directory path right behind it
  + another semicolon behind the last directory is optional
  + like this: 
* Run a command line shell, like cmd
* Important: Change to the C: diskdrive if you haven't yet
* Important: You need a good internet connection now
  + Minikube will download a lot of binaries in order to run properly
  + an internet connection too slow may lead to an aborted download
  + Make sure you have enough space in your Windows user directory for
    - another Virtual Machine (downloaded and added to VirtualBox by Minikube)
    - a whole lot of binaries (downloaded by Minikube)
* Start Minikube like this: minikube start
* do whatever you want to do with your mini Kubernetes cluster
  + that's why you installed kubectl
* Stop Minikube just like so: minikube stop