Tutorial:
Getting Started with ICC HAW Hamburg

Vorbedingungen (Installationen auf dem lokalen Rechner)

- Ein 64-bit Betriebssystem
- Git installiert
- Docker installiert (nativ unter Linux, Docker for Windows oder Docker for Mac(https://docs.docker.com/docker-for-mac/install/))
    - Wenn Sie **noch macOS < 10.11 nutzen**, erwägen Sie ein Update. Docker wird anmerken, dass es keine Updates mehr für 10.10 gibt.
    - Wenn Sie **VirtualBox unter macOS nutzen**, updaten sie VirtualBox auf Version 4.8.30 oder neuer.
    - Wenn Sie **VirtualBox unter Windows nutzen**, installieren Sie **stattdessen** die Docker Toolbox: https://www.docker.com/products/docker-toolbox
- kubectl installiert (https://userdoc.informatik.haw-hamburg.de/doku.php?id=docu:informatikcomputecloud#kubectl_-_command_line_fuer_kubernetes)
- kubelogin installiert (https://userdoc.informatik.haw-hamburg.de/doku.php?id=docu:informatikcomputecloud#login)
- Einmalig bei Gitlab unter https://gitlab.informatik.haw-hamburg.de angemeldet (Bekanntmachung des Namespaces bei Kubernetes)

Hinweise

- **ALLE**: Die im folgenden Text gezeigten Beispieldateien können alle aus dem haw-world Gitlab Projekt des AI Labors heruntergeladen werden. Sie ersparen sich dadurch Fehler beim Abtippen oder durch unsichtbare Zeichen beim Copy & Paste. Das Repository finden Sie hier: https://gitlab.informatik.haw-hamburg.de/ail/haw-world

- **NUR WINDOWS**: Verwenden Sie NICHT die mingw shell, die etwa bei einigen Git Installationen dabei ist. Hier werden keine interaktiven Sessions etwa für Passworteingabe unterstützt. Nutzen Sie stattdessen das übliche CMD, die PowerShell oder Bash for Windows (falls Sie schon Windows 10 nutzen)

- **NUR WINDOWS**: Docker benötigt HyperV zur Virtualisierung. Diese Funktion ist erst ab Windows Education oder Professional oder größer verfügbar. Sollten Sie Windows Home einsetzen, können Sie kostenlos über die HAW eine entsprechende Lizenz beziehen und Ihre Installation updaten.

- **WINDOWS & macOS:** Im Nachfolgenden Sind Docker Befehle mit **sudo** geprefixt. Das betrifft nur Linux Anwender. Sie können die Docker Kommandos **ohne sudo** einsetzen.

- Für Docker muss Ihre CPU die vT Erweiterung unterstützen. Alle modernen CPUs haben diese, bei einigen Laptops lässt sich die Erweiterung im Bios aktivieren. Einige alte Geräte mit älteren Prozessoren haben leider keine vT Erweiterung. Hier müssen Sie auf die lokale Ausführung von Docker verzichten.

Erstellung und Auslieferung einer Beispiel-Webapplikation

Code-Verwaltung mit GitLab (optional)

1. Melden Sie sich bei Gitlab unter https://gitlab.informatik.haw-hamburg.de an.
2. Klicken Sie auf den grünen „New Project“ Button und erstellen Sie ein neues, privates Projekt **mit dem Namen „haw-world“**
3. Selektieren Sie in der mittigen Auswahlbox für das Protokol „HTTPS“ anstelle von „SSH“
4. Führen sie nun die Befehle, die unter „Git global setup“ stehen, in einem Terminal auf Ihrem lokalen Rechner aus.
5. Erzeugen Sie ein neues Verzeichnis oder wechseln Sie in Ihr Stammverzeichnis für Softwareprojekte auf Ihrem Rechner via Terminal
6. Führen sie nun die Befehle, die auf der Gitlab Seite unten unter „Create a new repository“ stehen, nacheinander in einem Terminal auf Ihrem Rechner aus. Damit klonen Sie das Repository auf Ihren Rechner, erzeugen eine Readme.md Datei und pushen diese wieder hoch zu Gitlab.

Bleiben Sie im aktuellen Verzeichnis als Arbeitsverzeichnis!

Erstellen der Beispielapplikation und der Docker Images

7. Starten Sie Docker auf Ihrem Rechner
8. Im Folgenden erstellen Sie die Beispiel-Webapplikation, die nach dem Start lediglich einen Text ausgibt.
   Als Arbeitsverzeichnis dient ab hier der in Schritt 5 erstellten Ordner!

   Erstellen Sie eine Datei **`index.php`** mit folgendem Inhalt:

```
<?php
echo "Hello HAW-World!";
// Zugriffsdatum protokollieren
file_put_contents("data/haw-world.log", strftime("%d.%m.%Y-%H:%M") . " ",
FILE_APPEND);
?>
```

9. Erstellen eines Images für einen Container:
   Damit Docker weiß, wie das Image aussehen soll, erstellen Sie in Ihrem Arbeitsverzeichnis eine Datei namens **`Dockerfile`** mit folgendem Inhalt:

```
# inherit from the php:7.1-apache image from docker hub
FROM nexus.informatik.haw-hamburg.de/php:7.1-apache
# create data Directory and set permissions
RUN mkdir -p /var/www/html/data

# copies your php files
COPY . /var/www/html/
```

```
RUN chown -R www-data:www-data /var/www/html/*
```

10. Bei Verwendung von GitLab:

    Fügen Sie alle neue Dateien Ihrem Git Repository hinzu und commiten sowie pushen Sie diese:

    ```
    git add .
    git commit -m "Added basic files"
    git push –u origin
    ```

11. Loggen Sie sich bei der Informatik-Docker-Registry mit Ihrem HAW Account ein:

    ```
    sudo docker login docker-hub.informatik.haw-hamburg.de
    ```

12. Erzeugen Sie nun das Image **`haw-world`** durch Ausführen des folgenden Befehls *(Punkt am Ende für das aktuelle Verzeichnis nicht vergessen!)*:

    ```
    sudo docker build -t haw-world .
    ```

    Lassen Sie sich das erzeugte Image anzeigen:

    ```
    sudo docker images
    ```

13. Erzeugen Sie einen Alias-Namen für Ihr Image, indem Sie **`docker-hub.informatik.haw-hamburg.de/`** voranstellen (taggen), damit es in die Informatik-Docker-Registry hochgeladen werden kann. Der Name des Images muss dem Pfad Ihres Gitlab Projektes entsprechen. **Wenn Sie nicht sicher sind**, schauen Sie in dem unter Schritt 2. Angelegten Projekt in Gitlab im Menüpunkt „Registry“ nach. Dort steht, wie der korrekte Pfad lauten muss. Zum Beispiel:

    ```
    sudo docker tag haw-world docker-hub.informatik.haw-
    hamburg.de/<IhreHAWKennung>/haw-world
    ```

    Lassen Sie sich erneut die lokal vorhandenen Images anzeigen:

    ```
    sudo docker images
    ```

    Beachten Sie, dass die IMAGE ID bei beiden erzeugten Images identisch ist, es sich also bei dem neu benannten Image lediglich um einen Alias handelt.

14. Laden Sie Ihr Image nun mit dem push Befehl in die Informatik-Docker-Registry:

    ```
    sudo docker push docker-hub.informatik.haw-
    hamburg.de//<IhreHAWKennung>/haw-world
    ```

Deployment (Auslieferung) der Applikation in Kubernetes (der Informatik Compute Cloud):

15. Führen Sie **`kubelogin`** aus, um sich gegen den ICC Cluster zu authentifizieren und die Zugriffskonfiguration auf Ihrem Rechner zu erhalten

    ```
    kubelogin
    ```

16. Erstellen Sie nun eine Beschreibungsdatei **`deploy.yaml`** für das Deployment des Containers in der ICC mit folgendem Inhalt *(Achtung: Einrückungen haben semantische Bedeutung und dürfen nicht verändert werden!)*:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    service: haw-world
  name: haw-world
spec:
  selector:
    matchLabels:
```

```
        service: haw-world
    template:
      metadata:
        labels:
          service: haw-world
      spec:
        containers:
        - image: docker-hub.informatik.haw-
hamburg.de/<IhreHAWKennung>/haw-world
          name: haw-world
          ports:
          - containerPort: 80
            protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  labels:
    service: haw-world
  name: haw-world
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
     service: haw-world
  type: ClusterIP
```

17. Bei Verwendung von GitLab:
    Fügen Sie auch die YAML-Datei Ihrem Git-Repository hinzu:
    ```
    git add deploy.yaml
    git commit –n -m „Added YAML Deploy file“
    git push –u origin
    ```
18. Schreiben Sie nun diese Deploymentbeschreibung via **`kubectl`** in den Cluster. Führen Sie dazu folgenden Befehl aus:
    ```
    kubectl apply –f deploy.yaml
    ```
    Der Cluster startet nun Ihre Container in den Pods und erzeugt dazu Services, so dass diese Container zugänglich sind
19. Sehen Sie sich das Ergebnis mit folgendem Befehl an:
    ```
    kubectl get deploy,pod,svc
    ```
20. Führen Sie nun den kubectl proxy aus, damit Sie auf Ihre Webseite von Ihrem Rechner aus zugreifen können:
    ```
    kubectl proxy
    ```
    Sie können nun über folgende URL auf Ihre PHP Seite zugreifen:
    http://localhost:8001/api/v1/namespaces/<IhreNutzerkennung>/services/haw-world:http/proxy/
21. Um eine interaktive Shell in Ihrem Container zu erhalten, geben Sie bitte ein:
    ```
    kubectl exec –it <Name des Pods aus Ausgabe von Schritt 19> -- bash
    ```
    Bitte beachten Sie, dass alle Änderungen beim Stoppen des Containers verloren gehen, falls Sie keinen persistenten Speicher beantragt haben.

22. Löschen Sie nun Ihr Deployment und damit den haw-world Pod, in dem Sie folgenden Befehl eingeben:
    ```
    kubectl delete deploy haw-world
    ```
    Das Deployment und damit auch der Pod werden nun gelöscht.

Automatisierung des Image Builds als Pipeline

Das von Ihnen erstellte Image beinhaltet eine konkrete Version Ihres Programmcodes. Wenn Sie Ihren Code ändern, müssen Sie also auch ein neues, aktuelles Image erzeugen. Da nur faule Informatiker, gute Informatiker sind, wollen wir diesen wiederkehrenden Prozess automatisieren und durch eine Build Pipeline ersetzen. Gehen Sie wie folgt vor:

1. Erstellen Sie eine datei mit Namen **.gitlab-ci.yml** (beachten Sie den Punkt am Anfang des Namens! Die Datei ist versteckt.) im Wurzelverzeichnis Ihrer Anwendung (also auf einer Ebene mit index.php und den anderen YAML Dateien). Gitlab liest diese Datei automatisch aus und erstellt daraus eine Buildpipeline

2. Die **.gitlab-ci.yml** wird für unseren Anwendungsfall einen Job in einer Stage definieren. Dort bauen wir das Docker Image, taggen und pushen es schließlich in die Nexus Container Registry der HAW Hamburg:

```
stages:
 - dockerize

variables:
  DOCKER_HOST: "tcp://localhost:2375"
  DOCKER_REGISTRY: "docker-hub.informatik.haw-hamburg.de"
  DOCKER_CACHE: "nexus.informatik.haw-hamburg.de"
  SERVICE_NAME: "haw-world"


createImage:
  stage: dockerize
  image: docker-hub.informatik.haw-hamburg.de/ail/docker-dind
  services:
    - docker-hub.informatik.haw-hamburg.de/ail/docker-dind
  script:
    - docker login -u $NEXUS_USER -p $NEXUS_PW $DOCKER_REGISTRY
    - docker build -t $SERVICE_NAME:latest .
  - docker tag $SERVICE_NAME:latest
$DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:$CI_PIPELINE_ID
    - docker tag $SERVICE_NAME:latest
$DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $DOCKER_REGISTRY
    - docker push
$DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:$CI_PIPELINE_ID
    - docker push
$DOCKER_REGISTRY/$CI_PROJECT_NAMESPACE/$SERVICE_NAME:latest
```

Sie werden bemerken, dass die Job Definition (alles ab „createImage“) ebenfalls ein Image definiert. Dies liegt daran, dass auch der eigentliche Build Vorgang in einem Container in der ICC ausgeführt wird. Als Basis Image verwenden wir hier das docker:stable-dind Image, da wir in unserem Build Container Docker Befehle ausführen können wollen (dind = **D**ocker **in D**ocker).

Zusätzlich spezifizieren wir einen Service („services“). Dadurch wird ein zusätzlicher Container gestartet, der unserem Build Container via **localhost** zur Verfügung steht und eine Docker Engine auf Port 2375 anbietet. Hier wählen wir auch das dind Image und spezifizieren im „variables“ Teil der gitlab Datei „`DOCKER_HOST= "tcp://localhost:2375".` Dadurch weiß Docker im Basisimage, dass es sich mit der Docker Engine im Service Image verbinden soll (localhost:2375).

Im „script“ Teil schließlich führen wir die bekannten Docker Befehle aus. Bevor wir etwas mit der Nexus Registry tun können, müssen wir uns bei der Registry einloggen. Die notwendigen Variablen für NEXUS_USER und NEXUS_PW legen sie im nächsten Schritt als Secret Variables in Gitlab an. Interessant ist weiterhin, dass wir das Image zweimal taggen, einmal mit „latest“ und einmal mit der CI_PIPELINE_ID. Dies ist eine von Gitlab für jeden Pipeline Lauf eindeutig vergebene Zahl, die als guter Versionsindikator dienen kann.

Weitere Infos zu Gitlab Pipelines finden Sie in der offiziellen Dokumentation:
https://docs.gitlab.com/ee/ci/pipelines.html

3. Zum Anlegen der NEXUS_USER und NEXUS_PW Variablen navigieren Sie in Ihrem Browser in Ihr Gitlab Projekt und dann auf **Settings** -> **CI/CD**. Klicken Sie hier auf **Expand** beim Eintrag **Secret Variables**.
   Legen Sie hier sowohl NEXUS_USER und NEXUS_PW an. Als Value tragen Sie jeweils Ihre HAW Kennung bzw. Ihr Passwort in
4. Gehen Sie auf **Settings** -> **CI/CD** -> **Runner Settings** und aktivieren sie den Shared Runner (Enable Shared Runner)
5. Commiten und Pushen Sie die **.gitlab-ci.yml** nun via Git in Ihr Gitlab Repository
6. Navigieren Sie im Browser in Gitlab in Ihre Projekt und dann auf **CI / CD** -> **Pipelines**. Sie sollten hier nun eine Darstellung Ihrer Build Pipeline sehen. Mit einem Klick auf das Symbol in der Spalte **Stages** gelangen Sie in die Details Ihres Build Jobs. Etwaige Fehler in der Konfiguration werden Sie hier finden.
   Läuft der Build wie geplant durch, wir die Pipeline grün und Sie können Ihr Image mit Docker unter Angabe der Pipeline ID als Tag abrufen:

   ```
   sudo docker pull docker-hub.informatik.haw-hamburg.de/<IhreHAWKennung>/haw-world:<PIPELINE_ID>
   ```

Verwendung von persistentem Speicherplatz in der Applikation in der ICC

Das AI-Labor bietet derzeit begrenzten persistenten Speicher als Block Storage über einen CEPH Cluster an (http://ceph.com/ceph-storage/block-storage/) . Dieser Storage kann in der ICC in From von Volumes mit festgelegter Größe genutzt werden. Um ein Persistent Volume anzufordern gehen Sie wie folgt vor:

1. Erzeugen Sie in Ihrem Projekt eine Datei **`my-pvc.yaml`** für einen PersistentVolumeClaim mit folgendem Inhalt:

   ```
   kind: PersistentVolumeClaim
   apiVersion: v1
   metadata:
     name: my-storage-claim
     labels:
       service: haw-world
   spec:
     accessModes:
     - ReadWriteOnce
   ```

```
  resources:
    requests:
      storage: 10Mi
```

Die Storage Größe können Sie variieren, in dem sie den Wert in der letzten Zeile verändern. Bedenken Sie jedoch, dass Storage nur begrenzt zur Verfügung steht und Volumes nicht vergrößer- oder verkleinerbar sind!
Der Zugriffsmodus steht auf `ReadWriteOnce`, was bedeutet, dass Lesend wie Schreibend nur von einem Pod auf dieses Volume zugreifbar ist. Alternativen finden Sie in der Kubernetes Dokumentation.

2. Legen sie den Claim im Cluster an:

```
kubectl apply -f my-pvc.yaml
```

3. Erweitern Sie das Deployment in Ihrer deploy.yaml um die Referenzierung des VolumeClaims und geben Sie einen Mountpfad an:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    service: haw-world
  name: haw-world
spec:
  selector:
    matchLabels:
      service: haw-world
  template:
    metadata:
      labels:
        service: haw-world
    spec:
      containers:
      - image: docker-hub.informatik.haw-
hamburg.de/IhreHAWKennung/haw-world
        name: haw-world
        volumeMounts:
        - mountPath: "/var/www/html/data"
          name: my-volume

        ports:
        - containerPort: 80
          protocol: TCP
      volumes:
      - name: my-volume
        persistentVolumeClaim:
          claimName: my-storage-claim
```

Beachten Sie, dass der MountPath Wert auf das „var/www/html/data“ Verzeichnis Ihres Containers gesetzt wird. Damit sind alle Daten innerhalb dieses Verzeichnisses auf dem Volume gespeichert. Ihre Applikation legt die Zugriffslogs entsprechend in diesem Verzeichnis an.

Schreiben Sie nun diese Deploymentbeschreibung via **`kubectl`** in den Cluster. Führen Sie dazu folgenden Befehl aus:

```
kubectl apply –f deploy.yaml
```

Sobald der Pod zum Deployment erzeugt wird, wird der PersistentVolumeClaim durch Kubernetes an ein PersistentVolume gebunden, welches gegen den CEPH Cluster provisioniert wird. Je nach angeforderter Größe kann die Provisionierung zwischen wenigen Sekunden (< 1GB) bis hin zu mehreren Minuten in Anspruch nehmen.

Etwaige Fehler sind einsehbar via

`kubectl get pvc` und `kubectl describe pvc <PVCName>`

WICHTIG: Das erzeugte PersistentVolume bleibt solange erhalten, wie der PersistentVolumeClaim vorhanden ist. Sie können also Ihre Deployments, Pods und Services löschen ohne Ihre Daten zu verlieren. Löschen Sie aber den PersistentVolumeClaim, so sind auch die Daten weg.

Weitere Informationen finden Sie unter

https://docs.docker.com

https://kubernetes.io/docs/concepts/

Glossar	
Begriff	**Beschreibung**
Cluster	Alle Hardware-Server, die als Knoten in die Compute Cloud einbezogen sind.
Container (Docker, K8s)	Ein Container enthält die komplette Systemumgebung für eine Applikation (Code, Laufzeitmodul, Systemwerkzeuge, Systembibliotheken) Ein Container wird in einer Image-Datei gespeichert
Deployment (K8s)	Enthält eine eingebettete Pod Definition. Ein Deployment stellt sicher, dass es immer die definierte Anzahl des definierten Pods im Cluster gibt. Gibt es zu viele, werden Pods gelöscht, gibt es zu wenige, werden neue Instanzen gestartet.
Image (Docker)	Eine Datei, die einen Container enthält
K8s	Abkürzung für „Kubernetes“ - die acht Buchstaben “ubernete” werden durch “8” ersetzt
Persistent Volume (K8s)	Ein eigenständiger, persistenter und zusammenhängender Speicherbereich fester Größe.
Persistent Volume Claim (K8s)	Eine Anfrage für eigenes Persistent Volume mit Angabe der gewünschten Größe.
Pod (K8s)	Stellt einen logischen Host dar, der eine Applikation anbietet. Enthält Container und Volumes Die Einheit des K8s Schedulings Hat eine routbare IP Adresse (kein NAT!) Ephemeral: Pods derselben Spezifikation sind funktional identisch und daher austauschbar Containers in einem Pod sind eng gekoppelt Shared Namespaces Container in einem Pod teilen IP, Port and IPC namespaces Container in einem Pod kommunizieren via localhost
Registry (Docker)	Docker – Bibliothek (für Images)
Repository (Git)	Git - Bibliothek
Service (K8s)	Der Service stellt einen logischen Endpunkt zum Zugriff auf Pods dar.

	Erlaubt Load-Balancing für eingehende Anfragen über alle passenden Pods. Auswahl des Pods ist zufällig, Session Affinität wird unterstützt (via Client IP) Stellt eine stabile IP/Port Kombination bereit sowie einen DNS Namen! Eine logische Gruppierung für Pods mit derselben Funktion (logischer Endpunkt) Gruppiert via Label Selektor