Imgae

עבור תוכנה שמריצים, יש לרוב צורך בעוד run-time environment (פענוח השפה של התוכנה) ו-dependencies (תוכנות נוספות שצריך עבור ההרצה). הבעיה היא איך ניתן לשנע את שלושת המרכיבים ולהתקין אותם בקלות. Image הוא קובץ tar מקובץ שמכיל את כל הנ"ל ונבנה ע"י סט הוראות (dockerfile).

Containers:

פתרון של docker להרצה של תהליכים בצורה מבודדת ללא גישה למידע הנמצא מחוץ לקופסת ה-container.

Kubernetes

פלטפורמה להרצה של container-ים, המספקת יכולת ניהול יעילה.

היסטוריה:

VMware יצרו בשנות ה-2000 את ה-VM המסחרי, ובזאת הפרידו את ה-OS מהברזלים. ב-2010 דוקר יצאה לשוק, ואיפשרה את השימוש ב-isolated process לשימוש ע"י כלל המתכנתים. גוגל לקחה את borg (מוצר לפריסה וכו'), והחלה לשלב החל מ-2012 את הקובקנטיס, כפלטפורמה שתנהל את הדוקר. ב-2017 יצאה גרסת קוברנטיס ראשונה של production, אך הפתרון לא מתכלל את הכל, לודג' UI, דאגה לשרתים ואבטחת מידע. במקום יצאו סביבות מסחריות (openshift לדוג') שנותנים את שאר המעטפת הנצרכת.

רכיבי cluster Kubernetes

CONTROLS

הדרך של k8s לעבוד היא בעזרת controller-ים, רכיבים שכל הזמן מחפשים להגיע ל-desired state מסוים. 'כל רכיב שמחכה לי – צריך לעבור טיפול שלא יחכה לי'

רכיבים ב-master:

* ETCD מסד נתונים קל משקל, מצוין בשמירת קונפיגורציות. אם הרכיב תקול – כלום לא יעבוד.
* APISERVER שכבת api שעוטפת את מסד הנתונים. אף רכיב של קוברנטיס לא מתקשר ישירות עם ה-ETCD, אלא רק דרך ה-API. למשתמש יש פקודת kubectl, או Oc ב-openshift כדי לתקשר עם ה-API ולתת הוראות ל-master. ההוראות נשמרות ב-ETCD. אם הרכיב תקול – כלום לא יעבוד.
* SCHEDULER – מחכה ל-event של שינוי ב-ETCD שמחכה עבורו, ומבצע את השינוי הנדרש. למשל – הקמה של pod. הרכיב משנה את הסטאטוס מ-pending ל-scheduling, עד שיעביר את המשימה ל-worker. ההקצאה ל-worker מתבצעת ב-ETCD, על ה-worker יש agent בשם kubelet שסורק את ה-ETCD ומחפש משימות עבור ה-worker. אם הרכיב תקול – הסטאטוס יישאר בpending או scheduling.

רכיבים ב-worker:

* Container engine/runtime – תוכנה המתקשרת עם ה-kernel, ויוצרת isolated process ומנהלת אותם עבור ה-8s. (קרנל – שכבה המקשרת בין התוכנה לחומרה. ממומשת ע"י ה-driver-ים).
* kubelet - סורק את ה-ETCD ומחפש משימות עבור ה-worker. ה-kubelet הוא עובד בתצורת controller pattern, כלומר כל עוד מתרחש אירוע מסויים, תגיע למצב רצוי (desire state). ה-kubelet מריץ container engine ושולט דרכו ב-container-ים שהוא מריץ.
* Kube-proxy – שולט על התקשורת של ה-proces-ים, פותח וסוגר פורטים (כלומר אחראי על ה-tableroute).
* Nginx – ingress controller, כלומר רכיב שמנתב את התקשורת לתוך ה-service-ים המתאימים. הרכיב יושב ב-namespace משלו, קורא את קבצי ה-ingress, ומקבל את התקשורת ומנתב אותה לתוך ה-service-ים המתאימים.

CNI/SDN – LAN לוגי בין הפודים. התקשורת צריכה לצאת ל-NODE ולצאת דרך ה-NIC שלו, אבל לוגית מימשו לנו LAN שיחבר בין כלל הפודים

אובייקטים בקוברנטיס

* POD – תא. מכיל container-ים. הסיבה לשים container-ים ביחד היא ששניהם חולקים את אותו ip address, ומכירים אחד את השני. משמעות לדוג', שהם לא יכולים להיות באותו פורט.
* Namespace – הפרדה לוגית בין סביבות שונות, משאפר פודים עם שמות זהים ב-namespace-ים שונים, מה שהופך אותו לאידיאלי עבור הפרדת סביבות prod וכד'.
* Replica set (או RS)– נועד לעטוף את הפודים ולאפשר הקמה של מספר פודים זהים. מכיל שני רכיבים: pod naming- דואג לשמות לפודים שבאחריותו, pod config – קובץ הקונפיגורציה של הפודים. בתוך ה-spec של ה-RS, יש template, שהוא בעצם המידע של ה-pod. אם צריך למחוק פוד – יימחק הצעיר ביותר.  
  הדרך שבה ה-RS מזהה את הפודים היא ע"י שדה בשם matchLabels שנמצא ב-selector, שם מוגדרים labels בתצורת and (כלומר הפודים חייבים להכיל את כלל ה-labels שמוגדרים ב-matchLabels). ה-RS למעשה מבצע תמיד kubectl get pos -l [[labels]].  
  במחיקת RS יש מעין finalizer בלתי נראה שמוחק את הפודים.
* Deployment – האובייקט המיועד להנגשה למשתמשים, הוא עוטף את replicaset, ומכיל מידע זהה ל-replicaset עבוד הפודים. ה-deployment יוצר RS עם ID שנוצר מה-hash של הקונפיגורציה של ה-deployment. כשמשנים ב-deployment את המידע של ה-replicaset, נוצר replicaset חדש, אבל ה-replicaset לא נמחק, אלא רק נפטר מהפודים שלו. המטרה היא לאפשר תחקור קל ויעיל של ההבדל בין גרסות deployment, כשלפעמים השינוי של ה-deployment  
  כברירת מחדל מספר ה-replicaset שנשמרים אחורה הוא 10, וזה מוגדר ב-  
  להשלים REPLICA
* Service – מאפשר גישה לפוד. כתובות ה-IP של פודים הם דינמיים, ובכל פעם פוד עולה עם IP שונה. ה-service מאפשר service discovery, כלומר גילוי של הפודים ע"י שימוש ב-DNS (הוא מגדיר את השם ב-DNS). ל-service יש כתובת IP מופץ ב-DNS בכל ה-nodes, והוא מכיר את הפודים הזמינים כרגע. הוא מעביר את התקשורת ל-IP של פוד.  
  הקמה: kubectl expose deployment [[deplymentName]]. ניתן לעשות port redirection ג"כ.  
  סוגי service:
  + clusterIP: סרוויס שמאפשר גישה רק פנימית בתוך ה-cluster
  + NodePort: סורוויס שמנגיש את הגישה גם למחוץ ל-cluster, ע"י פתיחת פורט מסוג high port על כלל ה-node-ים. זה שימושי עבור backing service, כלומר שירות שאינו HTTP (כי צריך לציין פורט מוזר). ב-cluster-ים גדולים ניתן להגדיר שה-port ייפתח רק ב-worker שבו יש את ה-POD.
  + LoadBalancer: תוספת על NodePort, מאפשר לחבר LoadBalancer לפודים, במקום להיות תלוי בכתובת IP יחידה של ה-POD. שירות שניתן ע"י ספקים שונים ומבצע loadbalancer בצורה אוטומטית. ה-loadbalancer עצמו הוא לא של k8s. הסיבה לצורך ב-loadBalancer ב-service הוא מימוש שלל health Chack שבודק אם ה-NODE זמין והאם ה-port פתוח עליו. השירות החיצוני מבצע port-forwarding ל-node-port של 0078nginx.
* Ingress(/route)– קובץ הנקרא ע"י ה-nginx, וממפה נתיב לפוד. הנתיב יכול להיות שם הדומיין או החלק שאחרי ה-'/'.
* DeamonSet – אובייקט מקביל ל-deployment, ההבדל הוא שאצלו יש פוד אחד(או יותר) על כל NODE, באופן קבוע. המטרה היא לשמור יכולת ניטור על כלל ה-NODE-ים.
* statefulSet – מיועד ל-statefull application, כלומר אפליקציות שאכפת להם מהסביבה, בעיקר DB, שהם חייבים לדעת מה שם המכונה וה-IP וכד'. הפי'צרים שניתנים פה הם:
  + סדר הדלקה וכיבוי
  + Naming convention
  + IP ספציפים
* PVC – מיפוי של volume ממקור אכסון חיצוני לתוך ה-POD. ה-POD משתמש בנתיב כתקייה מקומית, אך בפועל ישנו אכסון, המקושר ל-k8s ע"י PV, ומיפוי של ה-PV ע"י PVC, שאותו מקשרים בתוך ה-deployment. בפועל, כאשר יוצרים PVC, ה-k8s יוצר באכסון מקום ומקשר אליו PV אוטומטית.

סכמות אובייקטים:

* Version: הסכמה של האובייקט, שלפיו הk8s מפענח את האובייקט
* Kind: סוג האובייקט
* Metadata: מידע גנרי על האובייקט
* Spec: הקונפיגורציה של האובייקט

פקודות:

* לכל האובייקטים יש get, create, update, delete. Describe כדי לקבל בצורה קריאה את כל הפרטים.
* ניתן להגדיר -o wide לכל פקודה, על מנת לקבל ב-output פרטים נוספים.
* ניתן להוסיף -w לפקודות השליפה עבור watch, שיעדכן אותי בכל פעם שמתבצע שינוי.
* אם מוסיפים -l – מקבלים את הפודים עם ה-lable המסויים הזה.
* Logs [[objectname]] – מביא את הלוגים על אובייקט מסויים

volumes:

Volume הוא שטח אכסון עבור ה-container-ים, עבור כלל השימושים. ה-volume תמיד יישב יחד עם הפוד על אותו ה-NODE. ניתן לשתף את ה-volumes בין ה-containet-ים, ואף בין ה-pod-ים, אבל זה לא מומלץ.

סוגים:

* emptyDir – תקייה ריקה, אם הפוד נופל – המידע יימחק.
* hostPath – שטח אכסון שיושב על ה-node עצמו, ומוקצה לשימוש ה-container, לא מומלץ לשימוש (עדיף להשתמש ב-PVC)
* configMap – קובץ הגדרות שנשאב מתוך ה-k8s api.
* Secret – מקרה פרטי של configMap עבור קבצים חסויים (לודג' סיסמאות, PKI). להבדיל ב-CM שנשמר כקובץ, ה-secret נשמר ב-RAM, הוא שמור כ-Base64, ויש לו הגבלת גודל לכ-1500 bytes. חשוב לשים לב לא לשים אותו כ-enviroment variable, ולא לשרשר אותו בשום צורה גלויה.

Helm chrat

על מנת לארוז את כלל חלקי הפרויקט יחד, יצרו פרויקט שנקרא , שיודע לארוז ולדלוור את כל הפרויקט יחד. ה-helm זקוק ל-kubectl (או oc), כדי לרוץ, והוא שואב את הקונפיגורציה מתוך \.kube\config שזה הנתיב של הגדרות ה-k8s. יש באינטרנט מלאי ש helm שזמיני להתקנה במיידית.

פקודות ה-helm

* Helm create – יצירת מבנה התיקיות עבור פרויקט helm חדש (builer plate).
* Helm template . – בונה את התיקייה הנוכחית לקבצים בצורתם הסופית
* Helm package. - בונה את התיקייה הנוכחית לקובץ tgz, תיקייה מקובצת שמכיל את כלל המידע
* Helm install [[instace name]] [[file]] – מקבל tgz ובונה אותו ב-k8s עם השם שהוזרק לתוך ה-k8s.
* Helm upgrade [[instace name]] [[file]] – מקבל tgz ומשרדרג את ה-helm עם אותו עם השם שקיים ב-k8s. ניתן להשתמש ב-'-f' כדי להוסיף קובץ שידרוס את ה-value.yaml

קבצי ה-heml

* Values.yaml – הקובץ שמכיל את המידע עבור הבנייה
* Chart.yaml – מכיל את המידע על הפרויקט (מהו הפרוייקט וכו')
* Test – תקייה המכילה unitest, שירוצו מיד לאחר ה-helm install
* Template: מבנה ה-yaml-ים עצמם. קבצי ה-YAML מלאים ב-{{setting}}, ששואבים את מידע ממקור אחר. האידיאל הוא לא לשנות כלום במבנה הקבצים, מלבד ב-values.
  + {{.Values.something}} – שולף את הערכים עבור something מתוך ה-values.
  + {{if …..}} – תנאי if
  + {{with …..}} - loop על הערך המתקבל, עד לתג {{end}}
  + {{…|…}} – העברה של הלק שמאל לחלק ימין
  + {{ ….. |nindent X}} – את הערך המתקבל תזיח X רווחים ימינה
  + {{ ….. |default ….}} – קח את צד ימין אם שמאל לא קיים

Environment specific

הרציונל הוא הפרדה בין הגדרות אפליקציה/מערכת, כגון image,emptydir וכד', למול הגדרות סביבה, כמו גודל זיכרון, מס' פודים, כתובת DB.

Health-check

בין כל pod ל-service יש רכיב שנקרא probe, שמבצע בדיקות liveness/readiness ה-yaml של ה-probe הוא בעצם ה-yaml של ה-deployment. אם הבדיקה של ה-readiness נפלה – ה-service לא ינגיש את הפורט, אם הבדיקה של ה-liveness נפלה – ה-k8s יפיל את הפוד וינסה להרים אותו.

יששנם סוגי בדיקות שונות כמו HTTP שמחכה לסטאטוס 200, או tcpsocket שרק רוצה תקשורת TCP בפורט מסויים. הבדיקות יכולות להמתין זמן מסוים מההתחלה (initialDelay), וזמן לבדיקה מחזורית (periodSeconds)

לרוב liveness יבדוק פורט מרכזי, וייקח initialDelay ארוך יחסית, ע"מ לאפשר לתחקר אם משהו כל הזמן קורס. Readiness יבדוק מכלול של זמינות (thread-ים, זיכרון וכוד'), וינגיש את הבדיקות בפורט אחר, אותו לא אחשוף החוצה כלל.

Custom value

הנגשה של עריכה של קובץ ה-value. מתבצע ע"י קובץ value חיצוני (לא קשור למבנה ה-helm), והצמדה לקובץ ה-tgz באמצעות הפקודה install/upgrade.

Operator

ה-k8s מאפשר ליצור סוגי ישויות חדשים, עם controller שיטפל בסוג הישות החדשה. בשביל ליצור דבר כזה יוצרים יישות מסוג custom resource, שמגדירה את הסכמה של הסוג ישות. ישנם SDK (סביבת פיתוח לפרוייקט מסוג ספציפי), כשהמוביל בהם הוא של GO. ה-SDK הופכים ל-image שמהווה controller עבור ה-CRD.

ArgoCD

תצורת עבודה בה ה-source of truth הוא ה-git. ה-git מעדכן באמצעות pull (ידני או מחזורי) את ה-argocd, שמעדכן בצורה אוטומטית את ה-k8s, מקרה פרטי של אופרטור מוכר. ה-ArgoCD מגיע כ-helm (כברירת מחדל הסרויס מוגדר כ-clusterIP, אם רוצים לפתוח צריך לשנות ל-LoadBalancer), עם סיסמה ראשונית ששמורה כ-secret (כמובן ב-base64).

ע"מ לפרוס אפליקציה ב-k8s שאנחנו יושבים עליו, משתמשים ב-kubernetes.default.svc. כלומר ל-svc שנמצא ב-default ונקרא Kubernetes, שהוא השער של ה-k8s.

ניתן לציין קובץ value מסויים כדי להפריד בין סביבות: מייצרים קבצי value שונים עבור כל סביבה, וב—Argo משתמשים בקובץ ה-value המתאים לסביבה.

מהו platform:

אנחנו ב-appdev, כלומר אנחנו רוצים לפתח אפליקציה, אבל אנחנו תלויים ב-3 דברים:

* Beacking services: - אפליקציות שאנחנו משתמשים בהם במוצר שלנו, לדוג' DB, S3 ועוד תוכנות שיושבות מאחורי הקלעים
* Tools – כלים שסביבת הפיתוח תלויה בהם, לדוג' source control management (git), automation (Jenkins), knowlagebase(confluence).
* Infra: הסביבה שעליו יושב המוצר, לדוג': מערכת הפעלה, k8s, סביבה וירטואלית..

שלושת אלה נקראים platform. כשמנסים ליצור מוצר – צריך לדאוג גם לכל זה. אם המוצר מגיע ישר ללקוח (כמו אתר אינטרנט) אפשר לרכוש אופציות זולות במקום לפתח ולטפח את כל הנ"ל. אך אם המוצר מגיע לארגונים, אני צריך לדאוג לכל הנ"ל כחלק מהמוצר עצמו. ככל שאנחנו מוצר יותר קטן – נשתדל לייצא את כל הנ"ל למקורות חינמיים פשוטים.