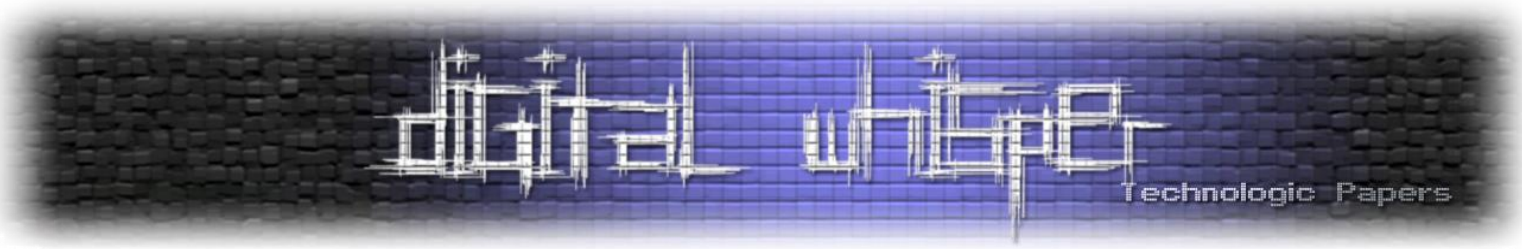


לקבל מושג ירוק על קוברנטיס (Kubernetes)

מאת ליאור בר-און

הקדמה

המאמר הזה נכתב לאנשי-תוכנה מנוסים, המעוניינים להבין את Kubernetes - בהשקעת זמן קצרה. הבאזז מסביב ל-Docker ו-Docker Orchestration הוא כרגע רב מאוד. דברי שבח רבים מסופרים על הטכנולוגיות הללו, מבלתי להתייחס לפרטים ועם מעט מאוד ראיה עניינית וביקורתית. כאנשי-תוכנה ותיקים אתם בוודאי מבינים שהעולם הטכנולוגיה מלא Trade-offs, וכדאי לגשת לטכנולוגיות חדשות עם מעט פחות התלהבות עיוורת - וקצת יותר הבנה.

המאמר פורסם במקור כ[פוסט](#) בבלוג "[Software Archiblog](#) - בלוג ארכיטקטורת תוכנה" מאת ליאור בר-און.

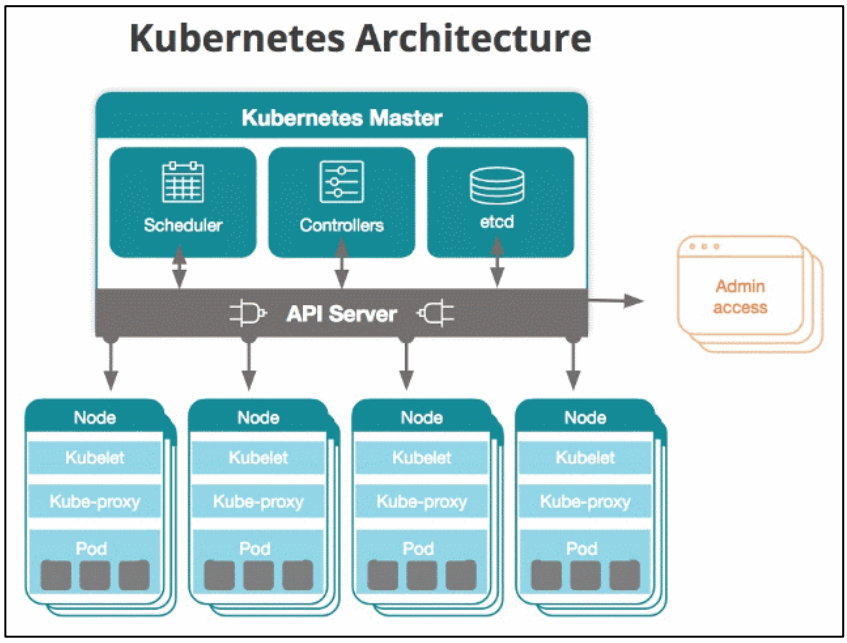
אני הולך לספק את הידע במאמר לא בפורמט ה- "From A to Z" כפי שדיי מקובל - אבל ע"פ סדר שנראה לי יותר נכון והגיוני. אני רוצה לדלג על דיון ארוך על ההתקנה, ועוד דיון ארוך על הארכיטקטורה - עוד לפני שאנחנו מבינים מהי קוברנטיס. אני הולך לדלג על כמה פרטים - היכן שנראה לי שהדבר יתרום יותר להבנה הכללית.

אז מהי בעצם קוברנטיס? פורמלית קוברנטיס היא Container Orchestration Framework (הנה פוסט שלי על הנושא) שהפכה בשנתיים האחרונות לסטנדרט דה-פאקטו (הנה פוסט אחר שלי שמנתח את העניין).

בכדי לפשט את הדברים, ניתן פשוט לומר שקוברנטיס היא סוג של סביבת ענן:

- אנו אומרים לה מה להריץ - והיא מריצה. אנו "מזינים" אותה ב-Containers (מסוג Docker או rkt) והגדרות - והיא "דואגת לשאר".
- אנו מקצים לקוברנטיס כמה שרתים (להלן "worker nodes" או פשוט "nodes") שהם השרתים שעליהם ירוצו הקונטיינרים שלנו. בנוסף יש להקצות עוד כמה שרתים (בד"כ - קטנים יותר) בכדי להריץ את ה-master nodes - ה"מוח" מאחורי ה-cluster.

- קוברנטיס תדאג ל-containers שלנו:
  - היא תדאג להרים כמה containers בכדי לאפשר high availability - ולהציב אותם על worker nodes שונים.
  - אם יש עומס עבודה, היא תדאג להריץ עוד עותקים של ה-containers, וכשהעומס יחלוף - לצמצם את המספר. מה שנקרא auto-scaling.
  - אם container קורס, קוברנטיס תדאג להחליף אותו ב-container תקין, מה שנקרא גם auto-healing.
  - קוברנטיס מספקת כלים נוחים לעדכון ה-containers לגרסה חדשה יותר, בצורה שתצמצם למינימום את הפגיעה בעבודה השוטפת - מה שנקרא deployment.
    - כפי שראינו בפוסט על Docker - פעולת restart של Container תהיה מהירה משמעותית מ-VM, שזה גם אומר לרוב deployments מהירים יותר.
  - לשימוש בקוברנטיס יש יתרון בצמצום משמעותי של ה-Lock-In ל-Cloud Vendor[1], והיכולת להריץ את אותה תצורת "הענן" גם On-Premises.
    - הסתמכות על קוד פתוח, ולא קוד של ספק ספציפי - הוא גם יתרון, לאורך זמן, וכאשר הספק עשוי להיקלע לקשיים או לשנות מדיניות כלפי הלקוחות.
  - קוברנטיס גם מספקת לנו מידה רבה של Infrastructure as Code, היכולת להגדיר תצורה רצויה לתשתיות רשת, אבטחה ועוד - מה שמייתר כלי ניהול תצורה (Provisioning) כגון Chef, Puppet או Ansible.



[1] אל דאגה! לספקי הענן יש אינטרס עליון לגרום לנו ל-Lock-In גם על סביבת קוברנטיס. לאחר שהניסיונות להציע חלופות "מקומיות" לקוברנטיס כשלו - רק טבעי שהם יתמקדו בלהציע יכולות שיפשטו את השימוש בקוברנטיס, אך גם יוסיפו סוגים חדשים של Lock-In. בכל מקרה, ברוב הפעמים אנו כבר תלויים בתשתיות כמו RDS, Athena, S3 ועוד - ולכן כבר יש Lock-In מסוים גם בלי קשר לקוברנטיס. "Cloud Agnostic Architecture" הוא בסה"כ מיתוס, השאלה היא רק מידת התלות.

יעילות

עם כל היעילות המוגברת שהתרגלנו אליה מריצה בענן בעזרת שירותים כמו AWS EC2 או Azure Virtual Machines (להלן "ענן של מכונות וירטואליות") - קוברנטיס מאפשרת רמה חדשה וגבוהה יותר של יעילות בניצול משאבי-חומרה.

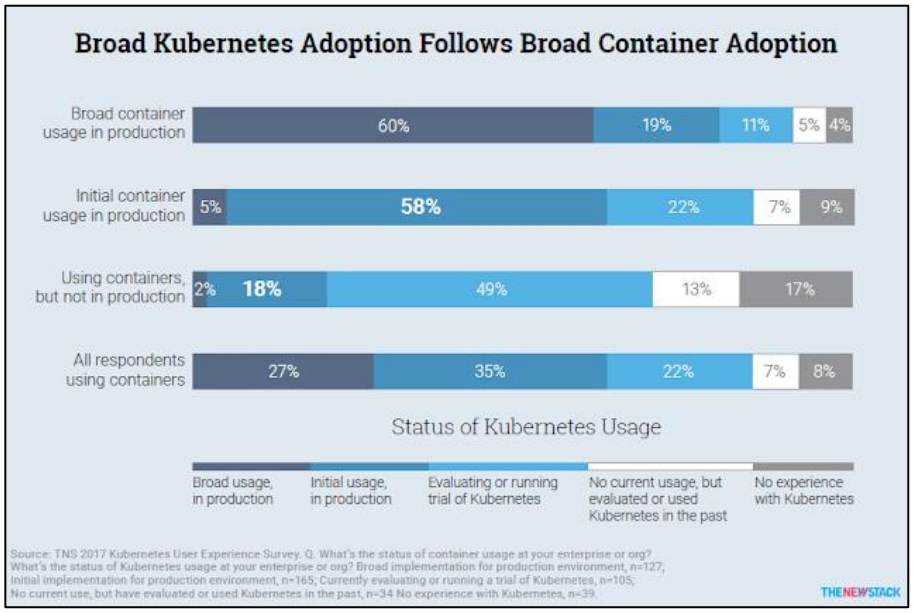
בתצורה קלאסית של מיקרו-שירותים, הפופולרית כיום - ייתכן ומדובר בניצולת חומרה טובה בכמה מונים. הכל תלוי בתצורה, אבל דיי טיפוסי להריץ את אותו ה-workload של מיקרו-שירותים בעזרת קוברנטיס על גבי 20%-50% בלבד מהחומרה שהייתה נדרשת על גבי "ענן ציבורי של מכונות וירטואליות".

איך זה קורה? למכונה וירטואלית יש overhead גבוה של זיכרון (הרצת מערכת ההפעלה + hypervisor) על כל VM שאנו מריצים. זה לא כ"כ משמעותי כשמריצים שרת גדול (כיום נקרא בבוז: Monolith) - אך זה מאוד משמעותי כאשר מריצים שרתים קטנים (להלן: מיקרו-שירותים).

מעבר לתקורה הישירה שעתה ציינו, יש תקורה עקיפה וגדולה יותר: כאשר אני מריץ על שרת 4 מיקרו-שירותים בעזרת VMs ומקצה לכל אחד מהמיקרו-שירותים 25% מהזיכרון וה CPU ההגבלה היא קשיחה. אם בזמן נתון שלושה מיקרו-שירותים משתמשים ב-10% ממשאבי המכונה כ"א, אבל המיקרו-שירות הרביעי זקוק ל-50% ממשאבי המכונה - הוא לא יכול לקבל אותם. ההקצאה של 25% היא קשיחה ואינה ניתנת להתגמשות, אפילו זמנית[2].

בסביבת קוברנטיס ההגבלה היא לא קשיחה: ניתן לקבוע גבולות מינימום / מקסימום ולאפשר מצב בו 3 מיקרו-שירותים משתמשים ב-10% CPU ו/או זיכרון כ"א, והרביעי משתמש ב-50%. אפשר שגם 10 דקות אח"כ המיקרו-שירות הרביעי יהיה idle - ומיקרו-שירות אחר ישתמש ב-50% מהמשאבים.

[2] שווה לציין שזה המצב בענן ציבורי. כששכן שלנו למכונה ב-AWS רוצה יותר CPU - למה שנסכים לתת לו? אנחנו משלמים על ה-"slice" שלנו במכונה - שהוגדר בתנאי השירות. בפתרונות של ענן פרטי (כמו VMWare) ישנן יכולות "ללמוד" על brusts של שימוש בקרב VM ולהתאים את המשאבים בצורה יעילה יותר. כלומר: המערכת רואה ש-VM מספר 4 דורש יותר CPU אז היא משנה, בצורה מנוהלת, את ההגדרות הקשיחות כך שלזמן מסוים - הוא יקבל יותר CPU מה VMs האחרים הרצים על אותה המכונה. טכנולוגית ה-VM עדיין מקצה משאבים בצורה קשיחה - אך תכנון דינמי יכול להגביר יעילות השימוש בהם. זה יכול לעבוד רק כאשר כל ה-VMs על המכונה שייכים לאותו הארגון / יש ביניהם הסכמה. T3/T2 instances ב-EC2 הם VMs שעובדים על עיקרון דומה: ב"חוזה" שלנו רשום ש-instance יכול לעבוד ב-burst ולקבל יותר משאבים - אך עדיין יש פה עבודה לפי חוזה, ולא אופטימיזציה גלובלית של המשאבים על המכונה (מכיוון ש-VMs שייכים לארגונים שונים).



הכרה חברתית

יהיה לא נכון להתעלם מהתשואה החברתית של השימוש בקוברנטיס. מי שמשתמש היום בקוברנטיס, ובמיוחד בפרודקשן - נתפס כמתקדם טכנולוגית, וכחדשן. ההכרה החברתית הזו - איננה מבוטלת.

כמובן שזו הכרה זמנית, שרלוונטית רק שנים בודדות קדימה. ביום לא רחוק, שימוש בקוברנטיס ייחשב מפגר ומיושן, ועל מנת להיתפס כמתקדמים / חדשנים - יהיה עלינו לאמץ טכנולוגיה אחרת.

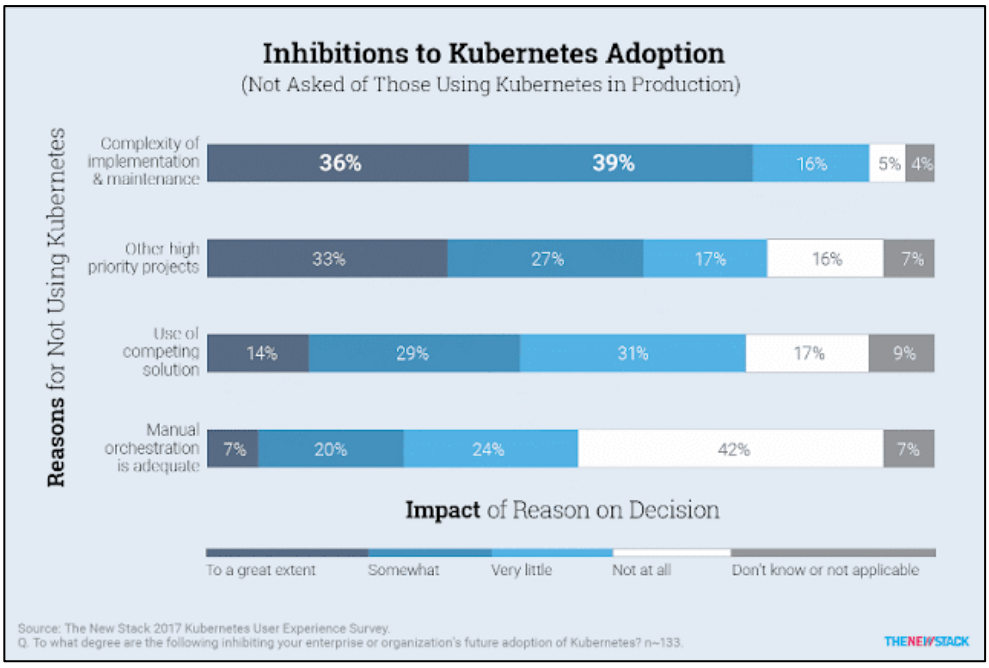
יש לי תחושה שפעמים רבות, ההכרה החברתית היא שיקול חזק לא פחות משיקולי היעילות - בבחירה בקוברנטיס. טבע האדם.

מחירים

כמובן שיש לכל הטוב הזה גם מחירים:

- קוברנטיס היא טכנולוגיה חדשה שיש ללמוד - וכמות / מאמץ הלמידה הנדרש הוא לרוב גבוה ממה שאנשים מצפים לו.
- התסריטים הפשוטים על גבי קוברנטיס נראים דיי פשוטים ואוטומטים. כאשר נכנסת לתמונה גם אבטחה, הגדרות רשת, ותעדוף בין מיקרו-שירות אחד על האחר - הדברים הופכים למורכבים יותר! Troubleshooting - עשוי להיות גם דבר לא פשוט, מכיוון ש"מתחת למכסה המנוע" של קוברנטיס - יש מנגנונים רבים.

- ברוב המקרים נרצה להריץ את קוברנטיס על שירות ענן, ולכן נידרש עדיין לשמר מידה של מומחיות כפולה בשני השירותים: לשירות הענן ולקוברנטיס יש שירותים חופפים כמו Auto-Scaling, הרשאות ו-Service Discovery (בד"כ: DNS).
- הטכנולוגיה אמנם לא ממש "צעירה", והיא בהחלט מוכחת ב-Production - אך עדיין בסיסי הידע והקהילה התומכת עדיין לא גדולה כמו פתרונות ענן מסחריים אחרים. יש הרבה מאוד אינטגרציות, אך מעט פחות תיעוד איכותי וקל להבנה.
- כמו פעמים רבות בשימוש ב-Open Source - אין תמיכה מוסדרת. יש קהילה משמעותית ופעילה - אבל עדיין הדרך לפתרון בעיות עשויה להיות קשה יותר מהתבססות על פתרון מסחרי.
- גם בשימוש ב"קוברנטיס מנוהל" (EKS, AKS, ו-GKE), החלק המנוהל הוא החלק הקטן, והשאר - באחריותנו.
- האם החיסכון הצפוי מניהול משאבים יעיל יותר, יצדיק במקרה שלכם שימוש בסביבה שדורשת מכם יותר תפעול והבנה?
- במקרה של ניהול מאות או אלפי שרתים - קרוב לוודאי שזה ישתלם.
- שימוש בקוברנטיס עשוי לפשט את סביבת התפעול, וה Deployment Pipeline. ההשקעה הנדרשת היא מיידית - בעוד התשואה עשויה להגיע רק לאחר זמן ניכר, כאשר היישום הספציפי באמת הגיע לבגרות.
- במקרים לא מעטים, ארגונים נקלעים לשרשרת של החלטות שנגזרות מצו האופנה ובניגוד לאינטרס הישיר שלהם: עוברים למיקרו-שירותים כי "כך כולם עושים" / "סיפורי הצלחה" שנשמעים, משם נגררים לקוברנטיס - כי יש להם הרבה מאוד שרתים לנהל, שכבר נהיה דיי יקר. לו היינו עושים שיקולי עלות/תועלת מול המצב הסופי - כנראה שהרבה פעמים היה נכון לחלק את המערכת למודולים פשוטים, או להשתמש במיקרו-שירותים גדולים ("midi-services") - וכך לשלוט טוב יותר בעלויות והמורכבויות האחרות.

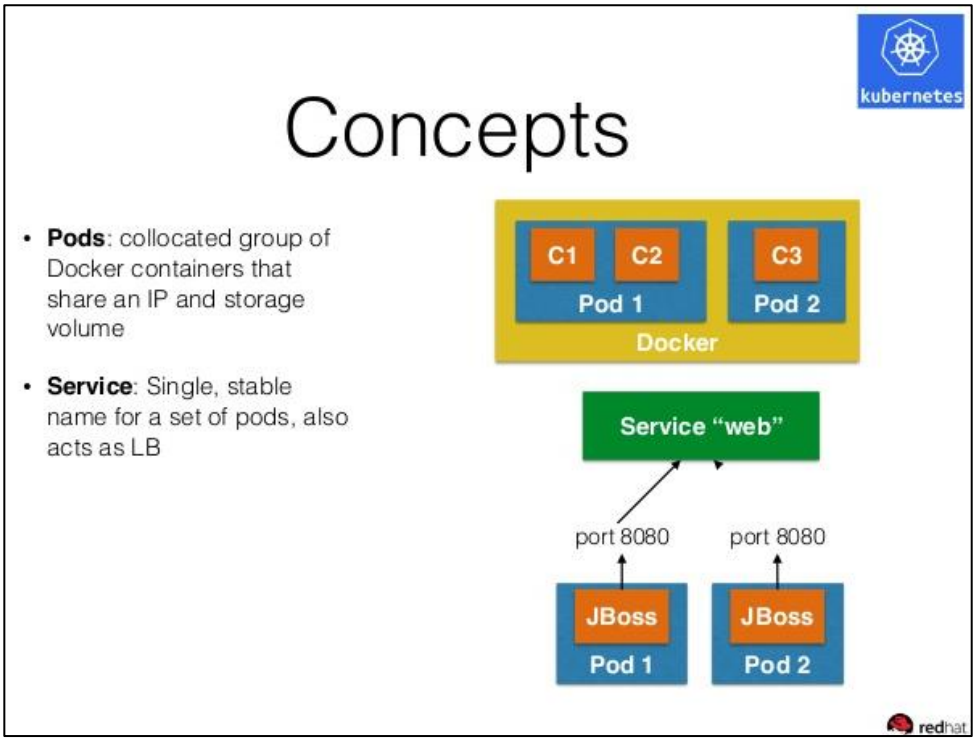


קוברנטיס בפועל

דיברנו עד עכשיו על עקרונות ברמה הפשטה גבוה. בואו ניגש לרמה טכנית קצת יותר מוחשית.

לצורך הדיון, נניח שהעברנו כבר את כל השירותים שלנו לעבוד על-גבי Docker וגם התקנו כבר Cluster של קוברנטיס. זה תהליך לא פשוט, שכולל מעבר על כמה משוכות טכניות לא קלות - אך נניח שהוא נגמר. בכדי להבין קוברנטיס חשוב יותר להבין מה יקרה אחרי ההתקנה, מאשר להבין את ההתקנה עצמה.

לצורך הדיון, נניח שאנו עובדים על AWS ו-EKS ואמזון מנהלים עבורנו את ה-Masters nodes. ה-Worker nodes שלנו נמצאים ב-Auto-Scaling Group - מה שאומר שאמזון תנהל עבורנו את ה-nodes מבחינת עומס (תוסיף ותוריד מכונות ע"פ הצורך) והחלפת שרתים שכשלו. זה חשוב! אנחנו גם משתמשים ב-ECR (קרי Container Registry מנוהל) , ואנו משתמשים בכל האינטגרציות האפשריות של קוברנטיס לענן של אמזון (VPC, IAM, ELB, וכו'). התצורה מקונפגת ועובדת היטב - ונותר רק להשתמש בה. אנחנו רק רוצים "לזרוק" קונטיינרים של השירותים שלנו - ולתת ל"קסם" לפעול מעצמו. רק לומר בפשטות מה אנחנו רוצים - ולתת לקוברנטיס לדאוג לכל השאר!



יצירת Pod

הפעולה הבסיסית ביותר היא הרצה של Container. בקוברנטיס היחידה האטומית הקטנה ביותר שניתן להריץ נקראת Pod והיא מכילה Container אחד או יותר. כרגע - נתמקד ב-Pod עם Container יחיד, זה יהיה המצב ברבים מהמקרים.

אני מניח שאנחנו מבינים מהו Container (אם לא - שווה לחזור צעד אחורה, ולהבין. למשל: הפוסט שלי בנושא), ויש לנו כבר Image שאנו רוצים להריץ ב-ECR. בכדי להריץ Container, עלינו לעדכן את קוברנטיס ב-manifest file המתאר Pod חדש מצביע ל-container image. קוברנטיס ירשום את ה-Pod ויתזמן אותו לרוץ על אחד מה nodes שזמינים לו. כאשר ה-node מקבל הוראה להריץ Pod עליו להוריד את ה-container image - אם אין לו אותו כבר. כל node מחזיק עותקים עצמאיים של ה- container images משיקולים של high availability.

הנה קובץ ה-manifest שהרכבנו:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    env: dev
    version: v1
spec:
  containers:
  - name: hello-world-ctr
    image: hello-world:latest
    ports:
    - containerPort: 8080
      protocol: TCP
```

קובץ ה-manifest בקוברנטיס מורכב מ-4 חלקים סטנדרטיים:

1. **גרסת ה-API**

1. הפורמט הוא לרוב <api group>/<version> אבל כמה הפקודות הבסיסיות ביותר בקוברנטיס נמצאות תחת API Group שנקרא core - ולא צריך לציין אותו.

2. ה-API של קוברנטיס נמצא (בעת כתיבת המאמר) בגרסה 1.13 - אז למה גרסה 1? ניהול הגרסאות בקוברנטיס הוא ברזולוציה של משאב. הקריאה לייצור pod היא עדיין בגרסה 1 (כמו כמעט כל ה-APIs. בעת כתיבת המאמר אין עדיין גרסת v2 לשום API, מלבד v2alpha או v2beta - כלומר גרסאות v2 שעדיין אינן GA).

2. **סוג (kind)** - הצהרה על סוג האובייקט המדובר. במקרה שלנו: Pod.

3. **metadata** - הכולל שם ו-labels שיעזרו לנו לזהות את ה-pod שיצרנו.

1. ה-labels הם פשוט זוגות key/value שאנחנו בוחרים. הם חשובים מאוד לצורך ניהול Cattle של אובייקטים, והם בלב העבודה בקוברנטיס.

4. **spec** - החלק המכיל הגדרות ספציפיות של המשאב שהוגדר כ-"Type".

1. name - השם שניתן ל-container בתוך ה-Pod, וצריך להיות ייחודי. במקרה של container יחיד בתוך ה-pod - אין בעיה כזו.

2. image - כמו בפקודת docker run...

3. ports - ה-port שיהיה published. TCP הוא ערך ברירת-המחדל, אך הוספתי אותו בכדי לעשות את ה-Yaml לקריא יותר.

תזכורת קצרה על Yaml:

פה ייתכן וצריך לעצור שנייה ולחדד כמה מחוקי-הפורמט של Yaml. ייתכן ונדמה לכם ש-Yaml הוא פורמט פשוט יותר מ-JSON - אבל זה ממש לא נכון. זה פורמט "נקי לעין" - אבל מעט מורכב. רשימה ב-Yaml נראית כך:

```
mylist:
   - 100
   - 200
```

"mylist" הוא ה-key, והערך שלו הוא רשימה של הערכים 100 ו-200. כל האיברים שלפניהם הסימן " - " ובעלי עימוד זהה - הם חברים ברשימה. סימן ה-Tab ברוב ה-Editors מתתרגם ל-2 או 4 רווחים. ב-Yaml הוא שקול לרווח אחד, ולכן שימוש בו הוא מקור לבעיות ובלבולים. הימנעו משימוש ב-Tab בתוך קבצי Yaml!

המבנה הבא, שגם מופיע ב-manifest (ועשוי לבלבל) הוא בעצם רשימה של Maps:

```
channels:
  - name: '#mychannel'
    password: ''
  - name: '#myprivatechannel'
    password: 'mypassword'
```

"channel" הוא המפתח הראשי, הכולל רשימה. כאשר יש "מפתח: ערך" מתחת ל"מפתח: ערך" באותו העימוד - משמע שמדובר ב-Map. כלומר, המפתח "channel" מחזיק ברשימה של שני איברים, כל אחד מהם הוא מפה הכוללת שני מפתחות "name" ו-"password".

אם נחזור לדוגמה של הגדרת ה-container ב-manifest למעלה, בעצם מדובר במפתח "containers" המכיל רשימה של איבר אחד. בתוך הרשימה יש מפה עם 3 מפתחות ("image", "name" ו-"ports") כאשר המפתח האחרון "ports" מכיל רשימה עם ערך יחיד, ובה מפה בעלת 2 entries.

הנה מדריך המתאר את ה-artifacts הבסיסיים ב-Yaml ממנו לקחתי את הדוגמאות. חשוב להזכיר שיש עוד כמה וכמה artifacts ב-Yaml - אם כי כנראה שלא נזדקק להם בזמן הקרוב.

עכשיו כשיש לנו manifest, אנחנו יכולים להריץ את ה-Pod:

```
$ kubectl apply -f my-manifest-file.yml
```

kubectl הוא כלי ה-command line של קוברנטיס. פקודות מסוימות בו יזכירו לכם את ה-command line של docker. במקרה הזה אנו במקרה הזה אנו מורים לקוברנטיס להחיל קונפיגורציה. הפרמטר f- מציין שאנו מספקים שם של קובץ.

תוך כמה עשרות שניות, לכל היות, ה-pod שהגדרנו אמור כבר לרוץ על אחד ה-nodes של ה-cluster של קוברנטיס.

אנו יכולים לבדוק אלו Pods רצים בעזרת הפקודה הבאה:

```
$ kubectl get pods
```

עמודה חשובה שמוצגת כתוצאה, היא עמודת הסטטוס - המציגה את הסטטוס הנוכחי של ה-pod. אמנם יש רשימה סגורה של מצבים בו עשוי להיות pod, אולי עדיין הסטטוס המדווח יכול להיות שונה. למשל: הסטטוס ContainerCreating יופיע בזמן שה-docker image יורד ל-node. זה מצב נפוץ - אך לא מתועד היטב. את הסטטוס ניתן למצוא בעיקר... בקוד המקור של קוברנטיס.

הפקודה הבאה (וריאציות), בדומה לפקודת ה-Docker המקבילה - תציג את הלוגים של ה-Container ב-Pod:

```
$ kubectl logs my-pod
```

אם ב-Pod יש יותר מ-2 containers (מצב שלא אכסה במאמר), הפקודה תציג לוגים של ה-container הראשון שהוגדר ב-manifest. אפשר לציין את שם ה-container כפי שצוין ב-manifest - וכך להגיע ל-container נתון בתוך Pod-מרובה containers.

עבור תקלות יותר בסיסיות (למשל: ה-pod תקוע על מצב ContainerCreating וכנראה שה-node לא מצליח להוריד את container image) - כדאי להשתמש בפקודה:

```
$ kubectl describe pods my-pod
```

התוצאה תהיה ססטוס מפורט שיכיל את הפרטים העיקריים מתוך ה-manifest, רשימה של conditions של ה-pod, ורשימת כל אירועי-המערכת שעברו על ה-pod מרגע שהורנו על יצירתו. הנה דוגמה להפעלה הפקודה (מקור):

```
Name:  nginx-deployment-1006230814-6winp
Node:  kubernetes-node-wul5/10.240.0.9
Start Time: Thu, 24 Mar 2016 01:39:49 +0000
...
Status:  Running
IP:  10.244.0.6
Controllers: ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID: docker://90315cc9f513c750f244a355eb1149
    Image:  nginx
    Image ID:  docker://6f623fa05180298c351cce53963707
    Port:  80/TCP
    Limits:
      cpu: 500m
      memory: 128Mi
    State:  Running
      Started:  Thu, 24 Mar 2016 01:39:51 +0000
    Ready:  True
    Restart Count: 0
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5kdvl (ro)
Conditions:
  Type  Status
  Initialized  True
  Ready  True
  PodScheduled  True
Volumes:
  default-token-4bcbi:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
    Optional:  false
...
Events:
  FirstSeen LastSeen Count From  SubobjectPath  Type  Reason  Message
  --------- -------- ----- ----  -------------  --------  ------  -------
  54s  54s  1 {default-scheduler }  Normal  Scheduled Successfully assigned nginx-deployment-1006230814-6winp to kubernetes-node-wul5
  54s  54s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Pulling  pulling image "nginx"
  53s  53s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Pulled  Successfully pulled image "nginx"
  53s  53s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Created  Created container with docker id 90315cc9f513
  53s  53s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Started  Started container with docker id 90315cc9f513
```

ניתן גם, בדומה ל-Docker, לגשת ישירות ל-console של ה-container שרץ ב-pod שלנו בעזרת הפקודה:

```
$ kubectl exec my-pod -c hello-world-ctr -it -- bash
```

במקרה הזה ציינתי את שם ה-container, אם כי לא הייתי חייב. נראה לי שזה מספיק, לבינתיים.

בואו נסגור את העניינים:

```
$ kubectl delete -f my-manifest-file.yml
```

הפעולה הזו עלולה להיראות מוזרה ברגע ראשון. הסרנו את הקונפיגורציה - ולכן גם ה-Pod ייסגר? לשימוש בקוברנטיס יש שתי גישות עיקריות:

- גישה אימפרטיבית - בה מורים לקוברנטיס איזה שינוי לבצע: להוסיף משאב, לשנות משאב, או להוריד משאב.
- פקודות כגון kube ctl create או kubectl replace הן בבסיס הגישה האימפרטיבית.
- גישה דקלרטיבית - בה מורים לקוברנטיס מה המצב הרצוי - והוא יגיע עליו בעצמו.
- פקודת kubectl apply - היא בבסיס הגישה הדקלרטיבית. אפשר להגדיר כמעט הכל, רק באמצעותה.
- החלת patch על גבי קונפיגורציה קיימת הוא משהו באמצע: זו פקודה דקלרטיבית, אך מעט חורגת מה lifecycle המסודר של הגישה הדקלרטיבית הקלאסית. סוג של תרגיל נינג'ה.

כמובן שהגישה הדקלרטיבית נחשבת קלה יותר לשימוש ולתחזוקה לאורך זמן - והיא הגישה הנפוצה והשלטת.

מכיוון שאין לנו הגדרה אלו Pods קיימים סה"כ במערכת - לא יכולנו לעדכן את המערכת בהגדרה הכוללת - ולכן הסרנו את המניפסט.

סיכום

ניסינו להגדיר בקצרה, ובצורה עניינית מהם היתרונות הצפויים לנו משימוש בקוברנטיס - מול הצפה של החסרונות והעלויות. לא פעם קראתי מאמרים שמתארים את קוברנטיס כמעבר מהובלת מסע על גבי פרד - לנסיעה ברכב שטח יעיל! אני חושב שההנחה הסמויה ברוב התיאורים הללו היא שהקוראים עדיין עובדים על גבי מערכות On-Premises ללא טכנולוגיות ענן. מעבר משם לקוברנטיס - היא באמת התקדמות אדירה.

המעבר משימוש ב"ענן של מכונות וירטואליות" לקוברנטיס - הוא פחות דרמטי. קוברנטיס תספק בעיקר יעילות גבוהה יותר בהרצת מיקרו-שירותים, וסביבה אחידה וקוהרנטית יותר למגוון פעולות ה-deployment pipeline. עדיין, נראה שקוברנטיס היא סביבה מתקדמת יותר - ואין סיבה שלא תהפוך לאופציה הנפוצה והמקובלת, בתוך מספר שנים. אחרי ההקדמה, עברנו לתהליך יצירה של Pod בסיסי, ובדרך עצרנו בכמה נקודות בכדי להדגים כיצד התהליך "מרגיש" בפועל.

חשוב לציין שה-Pod שהגדרנו הוא עצמאי ו"חסר גיבוי" - מה שנקרא "naked pod". אם naked pod כשל מסיבה כלשהי (הקוד שהוא מריץ קרס, או ה-node שעליו הוא רץ קרס/נסגר) - הוא לא יתוזמן לרוץ מחדש. מנגנון ה-auto-healing של קוברנטיס שייך לאובייקט / אבסטרקציה גבוהה יותר בשם ReplicaSet. אבסטרקציה מעט יותר גבוהה, שבה בדרך כלל משתמשים - נקראת Deployment. כיסינו דיי הרבה למאמר אחד. הנושאים הללו מצדיקים מאמר משלהם.

שיהיה בהצלחה!