האוניברסיטה הפתוחה
The Open University of Israel
الجامعة المفتوحة

המחלקה למתמטיקה ומדעי המחשב

20375 סמינר בנושא מיוחד במדעי המחשב

זמינות מערכות תוכנה בארכיטקטורת Microservices, במיכלי Docker ועל Kubernetes - סקירה ודוגמאות

מוגש על ידי גיא כרמי

בהנחיית פרופסור דוד לורנץ

אוגוסט 2020

מבוא

ארכיטקטורת Microservices היא ארכיטקטורה מודרנית לבניית מערכות תוכנה.
לארכיטקטורה זו מספר יתרונות, טכנולוגיים ועסקיים, אשר הופכים אותה לבחירה פוטנציאלית עבור חברות רבות.
בעבודה זו אסקור את השימוש במיכלי Docker ובמערכת Kubernetes על מנת להריץ מערכת Microservices, תוך שימת דגש על זמינות המערכת.
אחד החסרונות הבולטים של ארכיטקטורה זו הוא הסיבוכיות הגדלה ככל שיש יותר חלקים במערכת, כאשר כל חלק מתנהג באופן שונה ויש לו צרכים אחרים.
את בעיות אלו פותרים באמצעות מיכלים ומערכות לניהול מיכלים.
טרם בשלותם של פתרונות למיכלים ולמערכות לניהול מיכלים, ארכיטקטורת Microservices לא הייתה נפוצה, ויש שמתייחסים למיכלים בתור הטכנולוגיה שאפשרה את התפתחותה של ארכיטקטורת תוכנה זו [6].
מיכלים הינם פתרון ל"עטיפה" וירטואלית של יישום, כך שניתן יהיה להשתמש בו ללא תלות במכונה עליו הוא רץ.
מערכות לניהול מיכלים מאפשרות להריץ מערכות מורכבות, המבוססות על מיכלים, בפשטות יחסית. מערכות אלו אחראיות על התקשורת בין המיכלים, מנטרות את בריאות המיכלים, העומס עליהם ועוד.
לאורך העבודה, אדגים כל פרק באמצעות בניית מערכת בארכיטקטורת Microservices, עטיפת כל חלק במיכלים מסוג Docker ופריסתה במערכת לניהול מיכלים מסוג Kubernetes, כמו כן אדגים שימוש בתכונות (features) הרלוונטיות לשמירה על זמינות המערכת.
עבור ההדגמה, אבנה מערכת שתעזור לי בבעיה אישית ויום-יומית שאני נתקל בה רבות, הצורך לזכור את השם שלי! המערכת תורכב משני מיקרו-סרוויסים, האחד יהיה אחראי על שמי הפרטי, והשני על שם המשפחה.
כדי לעקוב אחר הדוגמאות לאורך עבודה זו, יש צורך בהתקנת Go, Python, Docker, Kubectl ובגישה לאשכול קוברנטיס (ניתן להריץ אשכול מקומי באמצעות minikube). לא אגע בתהליכי ההתקנה של תוכנות אלו. כמו כן, פקודות ההרצה מותאמות למערכת ההפעלה Linux.
הדוגמה נבנית לאורך העבודה בחלקים, על מנת להבין טוב יותר איך כל החלקים משתלבים יחדיו, ניתן לראות את הדוגמה השלמה במאגר הקוד של הפרויקט ב- GitHub, אותו אפשר למצוא כאן
https://github.com/GuyCarmy/microservices-docker-kubernetes-seminar-20375

ארכיטקטורת מיקרו-סרוויסים - Microservices

ארכיטקטורת מיקרו סרוויסים נחשבת לארכיטקטורת תוכנה מודרנית, אשר מתאימה לבניית אפליקציות אשר נפרסות (deployed) בענן [6].
אפליקציות הבנויות בארכיטקטורה זו מורכבות ממספר יחידות קטנות (micro) שכל אחת מהן אחראית לתת שירות מסוים (service) ויכולה להיפרס בצורה עצמאית.
ההשוואה המתבקשת היא למערכות הבנויות כמונוליט (monolith) - יישום (Application) הכתוב במאגר קוד יחיד אשר רץ כיחידה אחת ואחראי לכל תחומי המערכת. לעיתים קרובות, היתרונות (והחסרונות) של ארכיטקטורת המיקרו סרוויסים נמדדים בהשוואה למונוליט.

בארכיטקטורת מיקרו סרוויסים, היחידות מתקשרות באמצעות ממשקים ישירים (REST APIs למשל), מתווכי הודעות (message brokers) או הזרמת אירועים [5], אמצעי תקשורת אשר אינם תלויים בשפה או ספרייה ספציפית [6]. כך ניתן לכתוב כל מיקרו סרוויס בשפה אחרת ולהשתמש בכלים שונים ובמבני נתונים שונים, בהתאם לצורך של אותו סרוויס, וללא שיקולים הכוללים את האפליקציה כולה, אשר לעיתים אינם אופטימליים במקרה הספציפי.

את היתרונות של ארכיטקטורה זו אפשר להסביר גם בערך עסקי ולא רק בערך הנדסי [5],

- גודל קטן - יש פחות תלויות וכל מאגר קוד של סרוויס הוא קטן יותר ונגיש יותר, ולכן קל יותר להכניס קוד חדש לסרוויס קיים, לתקן באגים, להוסיף יכולות ולבצע ניסויים. כמו כן קל יותר להכניס מפתח חדש לעבודה על סרוויס קיים.
- גמישות טכנולוגית - צוותים יכולים להשתמש בטכנולוגיות שונות בהתאם ליכולות ולצורך, לכן קל יותר להרכיב צוות עבור משימה מסוימת, ישנה גמישות בגיוס אנשי פיתוח עם ניסיון בכלים שונים. טכנולוגיות חדשות מתפתחות כל הזמן, וכך ניתן לבחור את הכלי המתאים ביותר למשימה וצורך ספציפיים.
- הפרדה תשתיתית - סרוויסים שונים יכולים לגדול בכמות (Scale up) וכך להתמודד עם עומס גדל בצורה יעילה ומדויקת, ולכן חסכונית. מכאן שהיכולת של היישום כולו לעמוד בעומס טובה יותר, שכן אם יש עומס על שירות שניתן על ידי אחד המיקרו סרוויסים, ניתן להגדיל את הכמות שלו, ללא צורך להגדיל את שאר הסרוויסים (או את המונוליט כולו), כך ה"סקיילביליות" של היישום כולו משתפרת באופן יעיל [6].

כל סרוויס יכול להיות באחריות צוות אחר, ניתן לפתח במקביל וללא תלות בין הצוותים. כאמור, כל צוות יכול לעבוד בשפה אחרת ובכלים שונים, ודבר זה לא משפיע על התוצאה הסופית (האפליקציה כולה).

השפעות הבחירה בארכיטקטורה זו, גדולות יותר מבניית תוכנה בלבד. ישנן השפעות על מבנה קבוצת הפיתוח בארגון, ניתן לבנות צוותים על פי דרישה ולצוות יש יותר חופש בחירה בטכנולוגיות בהן הוא ישתמש למימוש הדרישה [6], כלומר הארכיטקטורה משפיעה על פרמטרים עסקיים שיש לקחת בחשבון.

לרוב, מיקרו סרוויסים עטופים במיכלים, ומיכלים נחשבים הטכנולוגיה שאיפשרה את התפתחות הארכיטקטורה. כך ניתן לפרוס אותם בקלות ובמהירות על מגוון תשתיות (מכונות פיזיות, וירטואליות, ופלטפורמות ענן שונות) [6].
החיסרון העיקרי של ארכיטקטורה זו, הוא שסיבוכיות הפריסה של המערכת גדלה.
מערכת הכתובה כיישום בודד ניתנת להרצה בצורה קלה וברורה. מערכת המחולקת ליישומים רבים וקטנים, מצריכה הרצה של כל יחידה בנפרד, כאשר לכל יחידה תצורה שונה וצרכים שונים.
לפתרון בעיה זו קיימות "מערכות לניהול מיכלים" (Container Orchestration Frameworks) אותן אסקור בהמשך.
ראוי לציין כי אין זה הכרח להשתמש במיכלים או במערכות לניהולם, ניתן לבנות אפליקציה עם סרוויסים רבים אשר כל אחד מהם רץ ישירות על המכונה עליו הוא פרוס או לחילופין בתוך מכונות וירטואליות, אך השימוש במיכלים מקטין משמעותית את התקורה של פריסת הסרוויסים, והשימוש במערכות לניהול מיכלים מקטין משמעותית את התקורה של תחזוקתם, התיאום ביניהם והתאמתם למצב בכל רגע. בכך, כלים אלו מעצימים את היתרונות של ארכיטקטורה זו.

על מנת להדגים את הגמישות הטכנולוגית אשר ארכיטקטורת מיקרו-סרוויסים נותנת, בחרתי לכתוב שני מיקרו-סרוויסים בשפות שונות. כפי שציינתי בפרק המבוא, מטרת המערכת תהיה להזכיר לי את שמי.
מובן מאליו, ולכן אין צורך להרחיב על הסיבה לבחירה זו, שהשפה המתאימה ביותר לניהול שמות פרטיים היא Go, ואילו לשפה Python קיים יתרון יחסי עבור ניהול שמות משפחה.
כאמור, ישנם מספר דרכים להעביר מידע בין מיקרו-סרוויסים, בחרתי בתקשורת ישירה על ידי API-ים למען הפשטות.

דוגמה

מיקרו-סרוויס-שם-פרטי בשפת Go

```
package main

import (
	"fmt"
	"net/http"
)

const FIRST_NAME = "Guy"

func main() {
	http.HandleFunc("/get_first_name", firstNameHandler)
	fmt.Println("http server listening on port: 8090")
	http.ListenAndServe(":8090", nil)
}

func firstNameHandler(w http.ResponseWriter, req *http.Request) {
	fmt.Println("Handles firstName request")
	fmt.Fprint(w, FIRST_NAME)
	return
}
```

בקוד זה נעשה שימוש בספריית net/http המובנת בשפה [15] לכתיבת http APIs.

את הקוד אשמור בקובץ main.go. כעת, נריץ את מיקרו-סרוויס זה על ידי הפקודה

```
go run main.go
```

ולאחר מכן נבדוק שהסרוויס אכן מחזיר את שמי הפרטי על ידי הפקודה

```
curl "http://localhost:8090/get_first_name"
```

מיקרו-סרוויס-שם-משפחה בשפת Python

```
from flask import Flask

api = Flask(__name__)

LAST_NAME = "Carmy"

@api.route('/get_last_name')
def last_name():
    return LAST_NAME


if __name__ == '__main__':
    api.run(port=8091, host='0.0.0.0')
```

בקוד זה נעשה שימוש בספרייה Flask שאינה מובנת בשפה [16], על מנת להתקינה נריץ את הפקודה

```
pip install flask
```

את הקוד אשמור בקובץ main.py. כעת, נריץ את מיקרו-סרוויס זה על ידי הפקודה

```
python main.py
```

ולאחר מכן נבדוק שהסרוויס אכן מחזיר את שם משפחתי על ידי הפקודה

```
curl "http://localhost:8091/get_last_name"
```

הדפסת-שם-מלא בשפת Python

אמנם לא מיקרו-סרוויס, אך על מנת לבחון את שני המיקרו-סרוויסים הנ"ל ולהדגים אלמנטים נוספים בהמשך העבודה, בחרתי לכתוב סקריפט Python קצר אשר מושך את שתי השמות ומדפיס את השם המלא. הסקריפט מקבל את כתובות המיקרו-סרוויסים בתוך משתני סביבה אותם נעביר בהמשך.

```
import requests
import os

first_name = requests.get(
    '{}/get_first_name'.format(
        os.getenv("MICROSERVICE_FIRST_NAME_URL")
    )
)

last_name = requests.get(
    '{}/get_last_name'.format(
        os.getenv("MICROSERVICE_LAST_NAME_URL")
    )
)

print('{} {}'.format(first_name.text, last_name.text))
```

הפעם ישנו שימוש בספרייה requests שגם היא ספרייה חיצונית [17], לכן נריץ את הפקודה הבאה על מנת להתקין את החבילה

```
pip install requests
```

לאחר מכן נגדיר את משתני הסביבה ונריץ את הסקריפט (כמובן שעל המיקרו סרוויסים שלנו לרוץ ברקע)

```
export MICROSERVICE_FIRST_NAME_URL="http://localhost:8090"
export MICROSERVICE_LAST_NAME_URL="http://localhost:8091"
python main.py
```

נצפה לראות את השם המלא מודפס.

מיכלים - Containers

מיכלים הם פתרון קליל (lightweight) לוירטואליזציה של תוכנה, יצירת תמונה של מיכל כוללת עטיפה של קוד המקור עם כל התלויות שלו, סוג מערכת ההפעלה וחבילות הקוד הנדרשות, לקובץ אחד, כך שניתן יהיה להריץ אותה על כל תשתית [1].

בניגוד למכונות וירטואליות, מיכלים קלים וגמישים יותר ועונים על צורך ספציפי יותר.
מכונות וירטואליות "מסתירות" את החומרה מהמשתמש, אם יש תקלה בחומרה ספציפית, אפשר להרים את אותה מכונה וירטואלית על חומרה אחרת.
מיכלים לעומת זאת, "מסתירים" את התוכנה, כך בהינתן תקלה במיכל ספציפי (הגעה למקרה קצה כלשהו בקוד), אפשר לאתחל את אותו מיכל (כלומר, אותה תוכנה). בנוסף ובדומה לעיל, במקרה של תקלה במכונה ספציפית (וירטואלית או פיזית), ניתן להריץ את אותו מיכל על מכונה אחרת.
המיכלים חולקים את משאבי מערכת ההפעלה של המכונה המארחת, וכך שומרים על אותה "קלילות", זאת לעומת מכונות וירטואליות רגילות, אשר מכילות את כל מערכת ההפעלה בכל תמונה [2].

היתרונות המרכזיים של מיכלים:

1. יצירת קובץ הניתן להרצה וניוד בין מכונות, אשר אינו תלוי במערכת ההפעלה של המכונה המארחת.
2. היכולת לפתח, לבדוק ולפרוס (to deploy) באותם תנאים על מספר רב של מכונות ומערכות הפעלה שונות.
3. היכולת לתקשר בין מיכלים.
4. היכולת לבודד בין מיכלים, כך שטעויות (באגים) או קוד זדוני במיכל מסוים, לא יוכלו להשפיע על מיכלים אחרים או על מערכת ההפעלה.

כמו כן, ישנם מספר יתרונות נוספים, אשר באים לידי ביטוי בהשוואה למכונות וירטואליות מסורתיות:

5. מהירות האתחול ביחס למכונות וירטואליות.
6. יעילות בניצול משאבי מערכת ההפעלה, פחות תקורה (overhead) ביחס למכונות וירטואליות.

7. קלות הניהול, ישנן מערכות רבות לניהול מיכלים (Container orchestration frameworks) אשר מקלות על התהליך. מערכות כאלה לא נפוצות עבור מכונות וירטואליות.

תנאי הכרחי לשימוש במיכלים הוא היכולת להריץ אותם בצורה מבודדת תוך שימוש במשאבי המכונה. לכן, בין היתר, ישנו שימוש ביכולות ממערכת ההפעלה לינוקס כגון control groups המאפשרים מתן משאבים לקבוצת תהליכים מסוימת ו- namespacing המאפשרים הסתרת והפרדת המשאבים בין הקבוצות.

כאמור, מיכלים אמורים לאפשר הרצה דומה בכל מערכת ההפעלה, אך במקור תוכננו עבור מערכת ההפעלה לינוקס ולכן מתבססות על יכולות שלה. על מנת לרוץ על מערכת הפעלה אחרות מתבצעות התאמות שונות, אשר לעיתים פוגעות בביצועים.

החיסרון שבבידוד כזה לכל מיכל, הוא חוסר היכולת של המיכל לשמור זיכרון לטווח ארוך. לכן מיכל כשלעצמו הוא חסר "מצב" (stateless), ומכל הרצה של המיכל נצפה לקבל תוצאה זהה.

ישנן תוכנות שכן צריכות לזכור דברים לטווח ארוך (כמו בסיסי נתונים), לשם כך קיימת היכולת להצמיד אמצעי אחסון (mount a volume) למיכל, כך שהתוכנה תוכל לגשת למקום מסוים ומוגדר בדיסק (מחוץ למיכל).

על מנת לנצל את יכולות אלו ממערכת ההפעלה המארחת, מותקן עליה רכיב ההרצה אשר מקשר בין משאבי המכונה לצרכי המיכלים.

רכיב הרצה זה, אמור להיות תואם לסוג המיכל שנמצא בשימוש, אך בזכות סטנדרט קבוע שאומץ על ידי כל השחקנים בתעשייה, Open Container Initiative, או בקיצור OCI, מרבית היכולות של המיכלים חוצות תשתיות וכלים [2].

ישנם מספר מימושים לקונספט של מיכלים, כאשר הנפוץ שבהם הוא Docker.

דוקר - Docker

דוקר הינו המימוש הנפוץ ביותר של רעיון המיכלים.

דוקר התחיל את חייו בתור פרויקט קוד פתוח בתחילת 2013, וכתוב בשפת התכנות Go.

בתחילה, הפרויקט היה חלק ממוצר אחר שאותה שיווקה החברה, אך לאחר פרק זמן של כחצי שנה, החברה שינתה את שמה ל- Docker Inc ודוקר הפך להיות מוצר עצמאי, והעיקרי שלה [3].

דוקר משתמש בארכיטקטורת שרת-לקוח, כאשר לרוב שניהם נמצאים על אותה מכונה, אך ניתן גם לעבוד עם שרת מרוחק [4].

השרת, אשר נקרא גם ה- docker daemon הוא זה שבונה ומריץ את המיכלים.

צד הלקוח הינו בדרך כלל ה- CLI (קיצור ל- Command Line Interface) אשר נקרא docker, אך ישנם גם GUI-ים (Graphical User Interface) למערכות הפעלה מסוימות.

האובייקטים הבסיסיים ביותר במימוש של דוקר הם ה- "תמונה" וה- "מיכל".

תמונה מגדירה את המפרט של מערכת ההפעלה והתלויות שצריכות להיות בו. מיכל הוא הרצה של תמונה.

הרצת מיכל נעשית באופן דומה ואחיד, ללא קשר לתמונה הרצה בו. המשמעות היא, שניתן באופן אחיד יחסית להריץ מיכלים עם טכנולוגיות שונות המתבססים על מערכות הפעלה שונות.

כפי שתיארתי בהסבר על מיכלים, גם דוקר משתמש ביכולות של מערכת ההפעלה.

דוקר מייצר מספר namespaces לניהול כל מיכל, ו- cgroup יחיד לכל מיכל, זאת על מנת להגביל את המשאבים הניתנים לו ולמנוע הגעה למגבלות החומרה. כך ניתן לאפשר הרצת מספר מיכלים על אותה חומרה ללא הפרעה. לבסוף, דוקר מייצר UnionFS, מערכת קבצים קלה המאפשרת ניהול קבצים בתוך המיכל [4].

אחד המרכיבים באקו סיסטם שדוקר יצר, הוא ה- Registry (אשר נקרא DockerHub), בו נמצאות תמונות רבות ושימושיות, למשל, תמונה המכילה את כל התלויות הנפוצות עבר שפת קוד מסוימת, או תמונה המכילה כבר בתוכה תוכנה מסוימת.

ניתן להשתמש בתמונות מוכנות מראש ולהריץ מיכלים בעזרתן, אך ניתן ונפוץ גם להגדיר תמונה ספציפית לדרישות עבור קוד מסוים (אשר לרוב מתבססת על תמונה מוכנה מראש), בקובץ בשם Dockerfile.

בנוסף, ניתן לפתוח חשבון ב- DockerHub ולהוסיף ל- Registry תמונות שיצרנו (באופן פרטי או פומבי).

דוגמה

בשלב הקודם, בנינו שני מיקרו-סרוויסים בשפות שונות. כאשר הרצנו אותם, התבססנו על כך שהשפות מותקנת על המכונה שלנו, כל החבילות הנוספות שבשימוש מותקנות גם כן, ושכל הגרסאות מתאימות. אלו הנחות לא טריוויאליות, אם על המכונה עלייה נריץ את הסרוויסים יהיו חסרות חבילות מסוימות, או שהיא תכיל גרסאות שונות, לא נוכל להיות בטוחים בהצלחת ההרצה.

כעת נראה איך שימוש במיכלי דוקר מאפשר לנו להריץ את אותו הקוד באותם תנאים על כל מכונה.

נגדיר תמונה עבור כל אחד מהמיקרו סרוויסים שבנינו. התמונה תתבסס על תמונה מוכנה מראש של השפה המתאימה, תכיל את הקוד שכתבנו ותוגדר להריץ את הסרוויס מיד עם הרצת מיכל.

את התמונות נעלה ל- Registry ב- DockerHub על מנת שנוכל להשתמש בהן מאוחר יותר, התמונות האלה הן בעצם קובץ הניתן לניוד בין מכונות, אשר הרצתו אינה תלויה בתצורת המכונה המארחת.

תמונה עבור מיקרו-סרוויס-שם-פרטי

```
FROM golang:1.14-alpine
ADD . /app/
WORKDIR /app
RUN go build .
ENTRYPOINT [ "/app/app" ]
```

(את התצורה נשמור קובץ בשם Dockerfile).

ניתן לראות כי התמונה נעולה בגרסה 1.14 של שפת Go (המילה alpine מייצגת את מערכת ההפעלה עליה מבוססת תמונה זו, אין לכך חשיבות בדוגמה שלנו), זוהי הגרסה עלייה פיתחתי את הסרוויס ולכן אני בטוח שהיא תואמת את ריצתו.

את התמונה נבנה באמצעות הפקודה

```
docker build -t "guycarmy/microservice-first-name" .
```

השם guycarmy הוא שם ה- Registry האישי שלי ב- DockerHub, ניתן לשנות את שם זה בהתאם לצורך. הנקודה בסוף השורה חשובה, ומסמנת איפה נמצא קובץ התצורה (ששמו Dockerfile).

לאחר בנייתה, נוכל להריץ מיכל המבוסס על תמונה זו באמצעות הפקודה

```
docker run -p 8090:8090 "guycarmy/microservice-first-name:latest"
```

התג latest הוא תג אוטומטי כיוון שלא ציינו תג במפורש בשלב הבנייה. כמו כן אנו מגדירים את הפורט (port) שהסרוויס עובד בו, כך הבקשות לפורט זה ברשת הפנימית במחשב שלנו יעברו למיכל זה.

לאחר הרצת המיכל, נוכל לבדוק שהמיקרו-סרוויס עובד ותקין באותה הצורה בה בדקנו אותו כאשר הרצנו אותו מקומית

```
curl "http://localhost:8090/get_first_name"
```

נדחוף את התמונה ל- Registry שלנו באמצעות הפקודה

```
docker push "guycarmy/microservice-first-name:latest"
```

מכיוון ש- DockerHub הוא ברירת המחדל עבור ה- Registry, לא נדרש לציין דבר מלבד השם.

תמונה עבור מיקרו-סרוויס-שם-משפחה

```
FROM python:3.7-slim
RUN pip install flask==1.1.2
ADD . /app
WORKDIR /app
ENTRYPOINT [ "python", "main.py" ]
```

(את התצורה נשמור קובץ בשם Dockerfile).

גם הפעם ניתן לראות כי התמונה מבוססת על תמונה של גרסה ספציפית של שפת Python, שוב, זוהי הגרסה עליה פיתחתי את הסרוויס (הפעם המילה slim אינה מייצגת את מערכת ההפעלה, אלא שזוהי גרסה רזה של תמונה עבור השפה. במבט חטוף, לא ניתן לדעת על איזו מערכת הפעלה אנו מתבססים, ושוב, אין לכך חשיבות בדוגמה זו).

בנוסף, נראה שההתקנה של החבילה החיצונית בה נעשה שימוש, גם כן נעולה עבור גרסה ספציפית שאנו יודעים כי היא מתאימה לקוד ולצרכים שלנו.

את התמונה נבנה באמצעות הפקודה

```
docker build -t "guycarmy/microservice-last-name" .
```

ונריץ מיכל באמצעות הפקודה

```
docker run -p 8091:8091 "guycarmy/microservice-last-name:latest"
```

לאחר הרצת המיכל, נוכל לבדוק שהמיקרו-סרוויס עובד ותקין באותה הצורה בה בדקנו אותו כאשר הרצנו אותו מקומית

```
curl "http://localhost:8090/get_last_name"
```

שוב, נדחוף את התמונה לשימוש מאוחר יותר, נשתמש בפקודה

```
docker push "guycarmy/microservice-last-name:latest"
```

תמונה עבור הדפסת-שם-מלא

```
FROM python:3.7-slim
RUN pip install requests==2.24.0
ADD . /job
WORKDIR /job
ENTRYPOINT [ "python", "main.py" ]
```

(את התצורה נשמור קובץ בשם Dockerfile).

כאן אין על מה להרחיב, אנחנו משתמשים באותה תמונת בסיס ומתקינים את החבילה הרלוונטית בגרסה הרלוונטית.

את התמונה נבנה באמצעות הפקודה

```
docker build -t "guycarmy/print-full-name" .
```

ונריץ מיכל באמצעות הפקודה

```
docker run --network host \
       -e MICROSERVICE_FIRST_NAME_URL="http://localhost:8090" \
       -e MICROSERVICE_LAST_NAME_URL="http://localhost:8091" \
       "guycarmy/print-full-name:latest"
```

הפעם נראה שני הבדלים בפקודת הרצת המיכל. הראשון הוא תצורת הרשת, אשר מוגדרת לרשת המחשב המארח, כלומר הרשת המקומית. השני הוא העברת שתי משתני סביבה עם הכתובות של המיקרו-סרוויסים.

תוצאת הרצה זו, אמורה להיות הדפסת השם המלא.

על מנת לדחוף את התמונה לשימוש מאוחר יותר, נשתמש בפקודה

```
docker push "guycarmy/print-full-name"
```

תקשורת בין מיכלים

ראינו כי אנו יודעים לנייד את התמונות בין מכונות (באמצעות העלאתן ל- DockerHub) ולהריץ אותן בכל מכונה באותם התנאים (אותם הגדרנו בקובץ התצורה לדוקר).

על מנת להשלים את ההתייחסות לארבעת היתרונות שמניתי בתחילת הפרק, אדגים את היכולות לתקשר בין מיכלים ולבודד מיכלים.

בדוגמאות עד כה, ראינו כיצד המיכלים מתקשרים ביניהם על בסיס הרשת המקומית, המיקרו-סרוויסים חשפו את הפורטים שלהם, וסקריפט ההדפסה הוגדר להשתמש ברשת המקומית.

לעומת זאת, כאשר נריץ מיכל, למשל על בסיס התמונה של הדפסת-שם-מלא, מבלי שנגדיר את הרשת

```
docker run \
       -e MICROSERVICE_FIRST_NAME_URL="http://localhost:8090" \
       -e MICROSERVICE_LAST_NAME_URL="http://localhost:8091" \
       "guycarmy/print-full-name:latest"
```

נקבל שגיאה, שכן ברשת המבודדת של מיכל זה, אין שום סרוויס שיענה לקריאות בפורטים 8090 ו- 8091.

בנוסף, אם היינו מריצים את כל אחד מהמיקרו סרוויסים ללא הגדרת הפורט, למשל

```
docker run "guycarmy/microservice-last-name:latest"
```

ואז היינו משתמשים בבדיקה שלנו

```
curl "http://localhost:8090/get_last_name"
```

היינו מקבלים את השגיאה הבאה

```
curl: (7) Failed to connect to localhost port 8090: Connection
refused
```

זוהי דוגמה לכך שתצורת ברירת המחדל, מבודדת בין המיכלים, ולא ניתן לגשת ממיכל למיכל או ממיכל למערכת ההפעלה.

כעת נראה כיצד ניתן ליצור תקשורת בין המיכלים שלא על בסיס הרשת המקומית, אלה על בסיס רשת הייעודית למערכת שלנו. ניצור רשת

```
docker network create "names-network"
```

נריץ את המיקרו-סרוויסים ברשת, ניתן להם שמות כך שלאחר מכן נוכל לפנות אליהם, ונגדיר להם לרוץ ברקע

```
docker run --network "names-network" --name "ms-last-name" -d
"guycarmy/microservice-last-name:latest"
docker run --network "names-network" --name "ms-last-name" -d
"guycarmy/microservice-last-name:latest"
```

אפשר לראות שלא חשפנו במפורש את הפורטים, בתוך הרשת הייעודית המיכלים יכולים לחשוף את הפורטים שלהם על פי הצורך.

נריץ את סקריפט הדפסת השם המלא ברשת, נעביר לו במשתני הסביבה את הכתובות על פי השמות שנתנו למיכלים

```
docker run --network "names-network" \
    -e MICROSERVICE_FIRST_NAME_URL="http://ms-first-name:8090" \
    -e MICROSERVICE_LAST_NAME_URL="http://ms-last-name:8091" \
    "guycarmy/print-full-name:latest"
```

שוב, נצפה לראות את השם המלא מודפס.

מערכות לניהול מיכלים - Container Orchestration Frameworks

בשנים האחרונות היינו עדים לאימוץ נרחב של רעיון המיכלים בדגש על Docker containers, זאת לאור הקלות בה ניתן לפרוס שירותי תוכנה (services) ולנצל את משאבי השרתים, כאשר אלו עטופים במיכלים [7], ולאור יתרונות נוספים אשר סקרתי בפרקים הקודמים.

כאשר משלבים את ארכיטקטורת המיקרו-סרוויסים עם הפרקטיקה של עטיפת כל סרוויס כמיכל, נוצר קושי לא מבוטל לנהל מספר רב של מיכלים כשלכל אחד התצורה הייחודית שלו.

מערכות ניהול (orchestration) מספקות תמיכה לניהול של מערכות תוכנה מבוזרות ומורכבות, אשר מבוססות על מיכלים, ובאות לפתור בעיה זו.

בפרק הבא, אסקור מערכת לניהול מיכלים ספציפית אשר בולטת במיוחד בשוק - Kubernetes.

ברמה הפשוטה ביותר ניתן להריץ את המיכלים ישירות על מכונות (וירטואליות או לא) באמצעות דוקר, ולקבל פריסה של השירותים שלנו (deployed services). הרי כפי שהוצג בפרק על דוקר, אחד החלקים הוא השרת (Docker daemon) אשר תפקידו לתאם בין מערכת ההפעלה למיכל, ולאפשר את הרצת המיכל.

כאמור, מיכל הוא יחידה מבודדת ומספר מיכלים שרצים במקביל לא יודעים אחד על השני, ולא יכולים לסנכרן בינם לבין עצמם. מכאן עולה הצורך במערכת לניהול מיכלים שתנהל את הפריסה שלהם [8].

מצד אחד מערכות ניהול מיכלים (Container Orchestration Frameworks) הן מוצר משלים ליכולת הבסיסית להרצת מיכלים, כמו זאת המגיעה כחלק מתוכנת דוקר. המערכות מתבססות על יכולות אלו ומרחיבות אותן.

מהצד השני, מערכות הניהול הללו עוטפות עבורנו גם את השרתים (או, המכונות) עליהן ירוצו המיכלים, לאשכולות (Clusters), ומונעות מאיתנו את הצורך לנהל מספר רב של מכונות באופן עצמאי.

אשכול הוא מקבץ של מכונות אשר עליהן ירוצו היישומים שלנו.

ישנן מספר מערכות לניהול מיכלים, הנפוצה שבהן היא Kubernetes.

מערכת Kubernetes

מערכת Kubernetes היא המערכת לניהול מיכלים הנפוצה ביותר, קוברנטיס נבנתה בחברת גוגל כהמשך למערכת ניהול מיכלים אחרת שהייתה בשימוש פנימי בלבד בשם Borg, וכיום היא מערכת קוד פתוח [9].

גרסה 1.0 של Kubernetes הופצה ב-21 ביולי 2015 ובמהרה הפכה ל"סטנדרט" עבור מערכות הבנויות בארכיטקטורת מיקרו סרוויסים.

קוברנטיס עונה על כל היכולות המצופות ממערכת לניהול מיכלים, ומצטיינת ביכולות "הבראת" (healing capabilities) היישומים הרצים בה, זאת על ידי אתחול מיכלים שנכשלו, או, העברתם למכונה תקינה במידה והמכונה המארחת אותם נכשלה.

על יכולותיה לתת מענה לבעיית הזמינות ארחיב בהמשך עבודה זו.

בפרק זה אסקור את חלקי המערכת.

בדומה למרבית המערכות לניהול מיכלים, קוברנטיס מבוססת על ארכיטקטורת מנהל-פועלים.

בתוך מכונת המנהל ישנם ארבעה רכיבים עיקריים.

הרכיב החמישי שעל מכונת המנהל, עליו לא ארחיב בעבודה זו, מוקדש לתיאום מכונות וירטואליות מספקי הענן הנפוצים (cloud-controller-manager), והמערכת יכולה לעבוד גם בלעדיו (אך צריך יהיה לחבר את הפועלים באופן ידני יותר).

בנוסף לאילו, ישנם שלושה רכיבים על כל מכונת "פועל".

חלקי המערכת

אשכול הוא מקבץ של מכונות. מערכות ניהול מיכלים שונות עובדות בארכיטקטורת מנהל - פועלים. אחת המכונות באשכול היא המנהלת (Master) והאחרות הן פועלות (Nodes).

קביעת תצורת האשכול נעשית בצורה הצהרתית (Declarative), לעומת ציווי-ית (Imperative), כלומר אם ברצוני לקבל אשכול עם 5 עותקים של יישום מסוים וכרגע יש לי 4, עלי להצהיר בקובץ התצורה (בדרך כלל קובץ מסוג YAML או JSON) שמספר העותקים הרצוי הוא 5, ולא לצוות על האשכול "הוסף עותק".

על המכונה המנהלת רצים שירותים אשר משווים את קבצי התצורה לתצורה בפועל, ובמידת הצורך מבצעים התאמות. בדוגמה הנ"ל, כאשר יזוהה הפער בין ההצהרה על צורך ב- 5 עותקים, לבין המצב בפועל של 4 עותקים, המערכת תפעל באופן אוטומטי להוסיף עוד עותק של יישום זה לאשכול.

Master

על מכונת המנהל - המאסטר - רצים ארבעה רכיבים עיקריים.

בפועל, ההפרדה בין מכונת "מנהל" לבין מכונת "פועל" היא יותר לוגית, ואמנם גם נהוגה בפועל, אך אינה הכרחית. ניתן להתקין רכיבים אלו על כל מכונה באשכול.

בנוסף, להשגת זמינות גבוהה, ניתן להתקין עותקים (replicas) של רכיבים אלו על מספר מכונות שונות וכך ליצור יתירות (redundancy) ולשפר את יציבות האשכול כולו [12.1].

שם אחר לרכיבים אלו, הוא מרכז השליטה (Control Plane).

kube-apiserver

חלק זה הוא ממשק המשתמש של קוברנטיס, והדרך שלנו לדבר עם מרכז השליטה.

כפי שראינו, קביעת התצורה של האשכול, נעשית באופן הצהרתי (declarative), כך ניתן לפנות לממשק (API) ולהצהיר על תצורה עבור האשכול או היישומים הרצים בו [12.1].

etcd

חלק זה הוא בסיס הנתונים של המערכת. זהו בסיס נתונים אמין (reliable) ועקבי (consistent) מבוסס זוגות של מפתח-ערך (key-value) המיועד למידע הקריטי של מערכות מבוזרות [13].

בתוך בסיס הנתונים מאוחסנות ההצהרות עבור תצורת המערכת המבוקשת [12.1].

kube-scheduler

המושג שיבוץ (Scheduling) בהקשר של קוברנטיס מתייחס להבטחה שלכל Pod ייבחר Node אשר יהיה אחראי להרצתו [12.5].

המשבץ מחפש Pod-ים חדשים שנוצרו ובוחר Node עבורם [12.1]. עוד על השיקולים הנלקחים בחשבון בבחירת ה- Node בפרק על Scheduling.

kube-controller-manager

רכיב זה מריץ את לולאות השליטה עליהן מבוססת המערכת.

קוברנטיס מבוססת על הצהרת התצורה הרצויה, תפקיד לולאות השליטה הוא להשוות בין ההצהרה, כפי שהועברה מהמשתמש אל ה- kube-apiserver ונשמרה בבסיס הנתונים (etcd), לבין המצב בפועל (כפי שמדווח על ידי ה- kubelet, המוסבר בהמשך).

למרות שכל לולאת שליטה ממומשת כתהליך (process) נפרד, למען הפשטות הן כולן אסופות יחדיו תחת ה- kube-controller-manager ורצות כתהליך יחיד [12.1].

Node

המכונות באשכול שתפקידן להריץ את המיכלים, ה- "פועלים" כפי שהוצגו קודם.
על כל מכונה כזו רצים שלושה רכיבים.

kubelet

ה- kubelet הוא בעצם היישום המייצג את ה- Node, הוא הוא האחראי להריץ את המיכלים (בעזרת ה- Container runtime), ולדווח את מצבם [12.1].
רכיב זה מקבל מה- kube-scheduler תצורות של Pod-ים אשר עליו להריץ, ומוודא כי המיכלים המוגדרים ב- Pod-ים אלו רצים על ה- Node.

kube-proxy

מערכות לניהול מיכלים אחראיות לנהל גם את התקשורת שמגיעה אל המיכלים הפרוסים באשכול. בהגעת בקשה (request) לשירות, למבקש (בין אם חיצוני לאשכול ובין אם מתוך מיכל שרץ בתוך האשכול) אין ידע על מבנה האשכול ואין יכולת לדעת היכן רץ היישום המבוקש, לכן על המערכת לתת מענה להעברת הבקשה למיכל אשר מכיל את היישום הנכון.
מכאן, שעל המערכת לנהל מיפוי בין היישומים למכונות עליהן הם רצים, ובין המיכלים לפורטים (ports) שהם חושפים. כמובן נדרשת גם מניעת התנגשות בין מיכלים החושפים את אותם פורטים ונמצאים על אותן מכונות [7].
כמו כן, במידה וקיים יותר ממיכל אחד של אותו יישום, תפקיד המערכת לבחור את הפנוי ביותר לענות על בקשת השירות (Load Balancing).
רכיב זה רץ על כל מכונה באשכול ואחראי על הגדרות התקשורת, כך שכל תקשורת נכנסת המגיעה למכונה תגיע אל המיכל אליה היא מיועדת.
כתלות במכונה, במידת האפשר הרכיב יפנה למערכת ההפעלה ויגדיר את חוקי התקשורת בעזרתה. אחרת, רכיב זה יבצע את ההעברות התקשורת בעצמו [12.1].

Container runtime

רכיב ההרצה של המיכלים, כפי שהוצג בפרק על מיכלים, חייב להיות נוכח על כל מכונה באשכול, אחרת כמובן שלא תוכל לעמוד בייעודה, לארח את המיכלים.
מערכת קוברנטיס תומכת במספר תוכנות שונות להרצת מיכלים, ביניהן דוקר כמובן [12.1].
למרות זאת, השאיפה היא לנתק את התלות במימוש מסוים, לכן ישנו ממשק אחיד לרכיבי הרצת מיכלים, שכל מימוש שיעמוד בו יהיה תואם לקוברנטיס [14].

Pod

ה- Pod הוא האובייקט הקטן ביותר הניתן לפריסה והרצה על מערכת קוברנטיס.

ניתן להסתכל על Pod כמעטפת למיכל, המכילה בנוסף לסוג המיכל שצריך לרוץ את הפקודות שצריך להריץ עליו, האחסון שצריך לחבר לו והפורט שהוא צריך לחשוף.

ניתן לעטוף באמצעות Pod גם מספר מיכלים אשר חייבים לעבוד ביחד כיחידה לוגית אחת, אך לרוב נגדיר Pod לכל מיכל [12.2].

ה- Pod הוא יחידת הבסיס, והוא מופע בודד של היישום העטוף במיכל(ים) שמוגדר(ים) כחלק מאותה יחידה. ניתן להגדיר את ה- Pod ישירות, אך כשלעצמו, ה- Pod אינו אחראי על זמינות המיכלים. במידה שמכונה אליה שובץ Pod קורסת למשל, חלק אחר במערכת יצטרך לשים לב לכך ולשבץ את ה- Pod מחדש. לכן לרוב הוא יווצר כחלק מאובייקט "גדול" יותר (controller) אותו נגדיר [12.2].

Controllers

לולאות השליטה הוזכרו כבר כרצות ברכיב kube-controller-manager, אלו לולאות אינסופיות אשר בתוכן מתבצעות פעולות (בדרך כל אל מול kube-apiserver אך לעיתים ישירות) על מנת להתאים את המצב בפועל לתצורה הרצויה [12.6].

כל לולאת שליטה עושה זאת עבור אובייקט אחד מאלו שיוצגו להלן.

אובייקטים אלו מוגדרים בקבצי תצורה (Configuration files) מסוג yaml (לרוב) או json ומכילים שדות מוגדרים המשמשים את לולאות השליטה.

זהו חלק מהותי מאוד במערכות לניהול מיכלים, שהרי מטרת מערכות אלו היא להריץ יישומים. ישנן מספר אפשרויות לפריסת יישומים באמצעות קוברנטיס, כמובן שכל האפשרויות מבוססות על מיכלים, אך בתוך מיכל ניתן לכתוב יישומים בעלי אופי שונה [7].

ניתן להריץ מיכל על ידי הגדרתו כיחידה קטנה ופשוטה, אותה כבר פגשנו, Pod, המתאימה בדרך כלל לסקריפטים פשוטים, אך עבור יישומים מורכבים יותר ישנן הגדרות אשר מתאימות ליישום המבצע חישוב רצוף וסופי (Job) והגדרות אשר מתאימות ליישום החושף ממשק וצריך להיות זמין (Deployment), כאשר ההבדל הוא אופי הריצה. באופן צפוי, הראשון יתחיל את ריצתו ויקבל שיבוץ (will be scheduled) עד שתסתיים, ואילו השני ישובץ לרוץ עד להסרתו.

אחריות המערכת לדאוג שמיכלים של יישום הנותן שירות יהיו משובצים על מכונות מתוך האשכול עד אשר היישום יוסר. במידה ומכונה מסוימת באשכול מאבדת כשירות, המערכת תשבץ את המיכלים מחדש במכונה אחרת (automatic rescheduling).

כזכור, מיכלים כשלעצמם הם חסרי "מצב" (stateless), כלומר חסרי זכרון אחסון ארוך טווח, והפתרון לכך הוא האפשרות להצמיד אמצעי אחסון ארוך טווח (persistent volume) למיכל ספציפי [7]. קוברנטיס תומכת באפשרות זאת ומאפשרות הצמדה של דיסק למיכל.

בנוסף, יישומים רבים יכולים לעבוד בתצורות שונות עבור מצבים שונים, ונזקקים למשתני סביבה (environment variables) על מנת לדעת באיזה תצורה הם צריכים לרוץ. גם כאן קוברנטיס מאפשרת הגדרת משתני סביבה אלו ו- "הזרקתם" אל תוך המיכלים הרצים בהתאם לתצורה שהגדיר המשתמש.

ReplicaSet

מטרתו של ReplicaSet הוא להבטיח ריצה של מספר עותקים (Replicas) של Pod-ים.

תצורת ה- Pod מוטמעת כחלק מהשדות המוגדרים בתצורת אובייקט זה, כלומר, כשם ש- Pod עוטף Container, ה- ReplicaSet עוטף Pod. כמו כן, בדומה ל- Pod, גם ReplicaSet הוא יחידת בסיס, ולכן לרוב לא נגדיר אותו בעצמנו [12.3].

Deployment

זהו לראשונה אובייקט שבו מומלץ להשתמש ישירות. זוהי התצורה העיקרית בה נשתמש על מנת לבקש מקוברנטיס להריץ מיכלים עבורנו, כלומר להריץ את היישום שבנינו.

אובייקט של Deployment ידאג להרים ReplicaSet שבתורו ידאג להרים Pod-ים שבתורם יריצו מיכלים [12.3].

משימה נוספת איתה מערכת לניהול מיכלים צריכה להתמודד, היא היכולת לבצע שדרוג ליישום מסוים מבלי לפגוע ביכולתו לתת שירות. בכל המערכות לניהול מיכלים ישנן אפשרויות לביצוע שדרוגים מתגלגלים (rolling upgrades). הדוגמה הפשוטה ביותר היא העלאת מיכל של היישום עם גרסה חדשה והעברת בקשות חדשות אליו, לפני הורדת המיכל הישן [7].

במערכת קוברנטיס,יכולות זו קיימת עבור משאב ה- Deployment. אם נבצע שינוי בתצורה, למשל החלפה למיכל שבו גרסה חדשה של היישום שלנו, ה- Pod-ים ייפרסו מחדש באופן מתגלגל (rollout), כך שזמינות היישום לא תפגע [12.3], זה נעשה על ידי יצירת ReplicaSet חדש שמריץ את Pod-ים מהסוג החדש, והגדלתו במקביל להקטנת ה- ReplicaSet הקודם.

נניח והמערכת צריכה 10 עותקים של יישום מסוים על מנת לעמוד בעומס, בצורה זו יהיו לנו 1 עדכני ו- 9 ישנים, לאחר מכן 2 ו- 8 וכן הלאה עד שיהיו 10 חדשים ו- 0 ישנים [12.3].

הבטחה נוספת של ה- Deployment היא המשך ישיר, היכולת לגלגל לאחור (rollback). במידה ועדכון שעשינו פגום, ניתן להחזיר למצב התקין האחרון, לכן אותו ReplicaSet ישן שכרגע יש לו 0 עותקים (אשר התצורה שלו שמורה בבסיס הנתונים אך הוא אינו צורך משאבי מחשוב מהאשכול) יתחיל לייצר עותקים (Pod-ים) עד שיגיע לכמות הנדרשת, וה- ReplicaSet החדש יוקטן ל- 0.

DaemonSet

אובייקט זה מבטיח שכל Pod שהוגדר באמצעותו, ירוץ על כל המכונות (Nodes) באשכול.
ה- DaemonSet שימושי פחות עבור יישומים המשרתים צד לקוח או עונים לבקשות (requests), אותם נגדיר באמצעות Deployment, אלא עבור יישומים שעוזרים לנו לנהל את האשכול או את היישומים בו.
דוגמה נפוצה היא יישום שתפקידו לאסוף את כל הדפסות התיעוד (logs) של היישומים האחרים שלנו, דוגמה נוספת היא ניטור מצב המכונות באשכול [12.3].

Job

כפי שכבר הוצג בפרק על מערכות לניהול מיכלים, שני סוגי הפריסות העיקריות הן של יישומים אשר צריכים להיות זמינים תמיד ועונים על בקשות על פי דרישה, את אלו נפרוס באמצעות אובייקט Deployment, ויישומים אשר צריכים לבצע מספר מוגדר של פעולות ולסיים את פעולתן.
הסוג השני ייפרס באמצעות אובייקט מסוג Job, אשר יריץ Pod אחד (או יותר) עד לסיומו.
בהתאם לרצוי, ניתן לקבל מה- Job מספר הבטחות, למשל שריצת ה- Pod תסתיים בהצלחה.
במידה והרצה מסוימת תיכשל, ה- Job ירים Pod נוסף עד לסיום מוצלח (ניתן להגדיר את כמות ניסיונות עד שה- Job יוותר וידווח כי נכשל, או לחילופין להגדיר שירוץ רק פעם אחת ולא ינסה כלל הרצה נוספת במידה ונכשל) [12.3].

CronJob

לא ארחיב, אך אובייקט זה עוטף Job ומייצר אותו במרווחי זמן מוגדרים מראש [12.3].
אובייקט זה שימושי עבור דברים שצריכים לקרות באופן תקופתי וחוזר, למשל ביצוע גיבוי כל לילה בחצות.

StatefulSet

אובייקט זה מתאים לפריסה של יישומים אשר צריכים אחסון קבוע (persistent volume). בדומה ל- Deployment, גם כאן ניתן להגדיר מספר עותקים של היישום, אך גמישות השיבוץ של StatefulSet שונה, לכל Pod מוצמד מזהה ייחודי וכאשר יש צורך להחליף Pod, הוא בהכרח יעלה בתוך מכונה באשכול ממנה יש גישה לאותו שטח אחסון [12.3].

Service

חלק זה במערכת קוברנטיס הוא חשוב במיוחד, Service הוא אבסטרקציה לחשיפת היישום הרץ על קבוצה של Pod-ים בתור שירות ברשת (החיצונית או הפנימית של האשכול) [12.4].

בקוברנטיס לכל Pod יש כתובת IP ייחודית משלו, לכן על מנת שיישום יוכל לתת שירות קבוע, הלקוחות שלו יצטרכו לעקוב בכל רגע נתון אחרי כתובות ה- IP של כל ה- Pod-ים של אותו יישום, לבחור אחד ולשלוח אליו בקשה.

זהו מצב לא רצוי, שכן Pod-ים באים והולכים מסיבות שונות, וקוברנטיס מבטיח לנו רק מהו מספר העותקים של כל יישום, ולא איפה הם יהיו או באיזה כתובת יהיו זמינים.

לכן נדרשים ה- Service-ים, לכל יישום (שנפרס באמצעות אובייקט Deployment למשל) נייצר גם Service כך שתיווצר לנו גם "כתובת" קבועה שבאמצעותה ניתן לקבל את השירות אותו מציע היישום.

ישנם שלושה סוגים עיקריים של Service-ים, הראשון לשימוש בתוך האשכול (כאשר יישום א' מקבל שירות מיישום ב' ושניהם פרוסים על מכונות בתוך האשכול), זהו ה- ClusterIP והוא נותן כתובת DNS פנימית לשימוש בתוך האשכול.

הסוג השני, NodePort, מייצר אובייקט מהסוג הראשון, אך בנוסף חושף port לשימוש מחוץ לאשכול. חשיפת פורט אינה מספיקה כדי לאתר בנוחות את היישום מבחוץ (נדרשות הכתובות של המכונות באשכול), והיא בדרך כלל צעד משלים לשימוש בחיבור של היישום לכתובת DNS חיצונית.

הסוג השלישי, LoadBalancer ניתן לשימוש רק אם מערכת הקוברנטיס שלנו מאוחסנת בספק ענן התומך בה, השם עלול לבלבל, שכן גם ClusterIP עושה Load Balancing בין עותקי היישום בתוך האשכול.

אובייקט LoadBalancer מייצר עבורנו אובייקטים מהסוג הראשון והשני, אך בנוסף שולח בקשה לספק הענן ליצור עבורנו External Load Balancer (או Internal Load Balancer, כזה החשוף רק בתוך הרשת הפנימית של ספק הענן, שמכילה ישויות נוספות על פני הרשת הפנימית של האשכול), וכך לקבל כתובת IP קבועה וחיצונית מספק הענן שלנו, אליה אפשר לגשת מחוץ לאשכול.

דוגמה

תצורת Deployment עבור המיקרו-סרוויסים

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-first-name-deployment
  labels:
    app.kubernetes.io/name: microservice-first-name
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: microservice-first-name
  template:
    metadata:
      labels:
        app.kubernetes.io/name: microservice-first-name
    spec:
      containers:
      - name: microservice-first-name
        image: guycarmy/microservice-first-name:latest
        ports:
        - containerPort: 8090
```

(את תצורה זו נשמור בקובץ deployment.yaml, בדוגמה שלי, תחת תיקייה בשם kubernetes).

נפרק את התצורה:

בשורה הראשונה נראה את גרסת הממשק של קוברנטיס איתה אנו רוצים לעבוד. יש לבדוק בתיעוד של קוברנטיס בהתאם לגרסה, באיזה גרסת ממשק נמצא המשאב הרצוי.

לאחר מכן, ברור כי סוג התצורה היא Deployment, כלומר זוהי תצורה עבור משאב זה. כפי שהסברתי קודם, זהו משאב אשר עוטף ReplicaSet שבתורה עוטפת Pod (אחד או יותר).

בשלב הבא נגדיר מידע נוסף על המשאב שלנו, שמו כמובן, ותוויות נוספות [12.9], תוויות אלו מאפשרות קישור בין כל החלקים הנעים המרכיבים את משאב זה, ועל כך אפרט מיד.

בשלב ה- spec הראשון, אנו בעצם מגדירים את פרטי התצורה עבור ה- ReplicaSet שיווצר, נגדיר את מספר העותקים שאנחנו רוצים ואת התוויות של ה- Pod-ים אשר ReplicaSet זה אמור לנהל.

לאחר מכן נעבור ל- template, בו נגדיר את תצורת ה- Pod. גם עבורו נוסיף את התוויות.

כך, לאחר שהגדרנו את כל התוויות, המערכת תוכל להשתמש בהן כדי לקשר בין שלושת החלקים, ה- Deployment שמנהל עבורנו ReplicaSet-ים שמנהלים עבורנו Pod-ים.

בשלב ה- spec השני, נגדיר פרטים נוספים עבור תצורת ה- Pod, המיכל (או מיכלים, לא במקרה שלנו) שהוא אמור לעטוף, ובאיזה פורט לחשוף את המיכל.

התצורה עבור המיקרו-סרוויס השני זהה עד כדי שינויי השמות והפורט (זמינה לעיון במאגר הקוד של הפרויקט).

כעת, נוכל לפרוס את המיקרו-סרוויסים באמצעות הפקודה

```
kubectl apply -f "kubernetes/deployment.yaml"
```

לאחר מכן נוכל לראות כי נוצרו לנו גם Deployment, גם ReplicaSet וגם- Pod, נבדוק זאת באמצעות הפקודות

```
kubectl get deployments
kubectl get replicasets
kubectl get pods
```

נקבל

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
microservice-first-name-deployment    0/1     1            0           3s
NAME                                             DESIRED   CURRENT   READY   AGE
microservice-first-name-deployment-799565cbdb    1         1         0       3s
NAME                                                     READY   STATUS              RESTARTS   AGE
microservice-first-name-deployment-799565cbdb-7ldqv      0/1     ContainerCreating   0          3s
```

נראה כי הראשון (deployment) קיבל את השם שנתנו לו, השני (replicaset) קיבל שם המבוסס על הראשון, והשלישי (Pod) קיבל שם המבוסס על השני.

כמו כן, נראה כי מיד לאחר הרצת פקודת הפריסה, המשאבים אינם מוכנים והמיכל עדיין בתהליך היצירה. אם נריץ את אותן פקודות שוב לאחר כמה רגעים (בנוסף הרצתי את פקודת הפריסה עבור המיקרו-סרוויס השני), נראה כי כעת כל המשאבים מוכנים.

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
microservice-first-name-deployment    1/1     1            1           5m29s
microservice-last-name-deployment     1/1     1            1           55s
NAME                                             DESIRED   CURRENT   READY   AGE
microservice-first-name-deployment-799565cbdb    1         1         1       5m29s
microservice-last-name-deployment-576566b475     1         1         1       55s
NAME                                                     READY   STATUS    RESTARTS   AGE
microservice-first-name-deployment-799565cbdb-7ldqv      1/1     Running   0          5m29s
microservice-last-name-deployment-576566b475-wflh8       1/1     Running   0          55s
```

תצורת Service עבור המיקרו-סרוויסים

```yaml
apiVersion: v1
kind: Service
metadata:
  name: microservice-first-name-service
spec:
  selector:
    app.kubernetes.io/name: microservice-first-name
  ports:
    - port: 80
      targetPort: 8090
```

(את תצורה זו נשמור בקובץ service.yaml, בתוך התיקייה kubernetes).

בשלב זה, נגדיר את משאב ה- Service עבור המיקרו-סרוויסים.

זהו משאב דיי פשוט. נגדיר עבורו את שמו, ובסעיף ה- spec נגדיר את ה- selector ואת הפורטים.

ה- Service חושף את היישום לרשת. ברירת המחדל לסוג ה- Service היא ClusterIP ולכן במקרה הזה היישום יהיה חשוף ברשת הפנימית של קוברנטיס.

כל קריאה שתגיע ל- Service תועבר לאחד ה- Pod-ים המתאימים לתוויות שהוגדרה ב- selector, הפורט של ה- Service יהיה הפורט שהוגדר, ואילו הקריאות יועברו לפורט היעד ב- Pod.

גם הפעם התצורה עבור המיקרו-סרוויס השני זהה עד כדי שינוי השמות ופורט היעד.

כעת נוסיף את ה- Service לאשכול

```
kubectl apply -f "kubernetes/service.yaml"
```

נוכל לבדוק שהוא נוצר על ידי הפקודה

```
kubectl get services
```

נקבל

```
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
microservice-first-name-service   ClusterIP   10.99.250.134   <none>        80/TCP    46s
```

בהמשך, נשתמש ב- Service-ים שחשפנו על מנת לפנות אל המיקרו-סרוויסים שלנו

תצורת CronJob עבור הדפסת-שם-מלא

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: print-full-name-cronjob
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: print-full-name
            image: guycarmy/print-full-name:latest
            env:
            - name: MICROSERVICE_FIRST_NAME_URL
              value: http://microservice-first-name-service
            - name: MICROSERVICE_LAST_NAME_URL
              value: http://microservice-last-name-service
          restartPolicy: Never
```

(את תצורה זו נשמור בקובץ cronjob.yaml, בתוך תיקיית kubernetes).

זוהי תצורה עבור משאב CronJob, כאמור, משאב זה מייצר תחתיו Job, וזה בתורו מייצר Pod אשר מריץ את המיכל שתמונתו מוגדרת.

לאחר הגדרת שם המשאב, בדומה לתצורת Deployment, נראה כמה שורות spec.

הראשון עבור ה- CronJob, לוח זמני יצירת ה- Job ב- cron syntax, משמעות הסימון `"* * * * *"` היא כל דקה עגולה, לא אכנס לתחביר של כלי זה, רק אציין כי ניתן להגדיר באמצעותו ברמת דיוק גבוהה לוחות זמנים בסייקל יומי, שבועי, חודשי או אפילו שנתי.

ה- spec השני הוא עבור ה- Job, כאן אין דבר מיוחד להגדיר, מלבד את ה- spec השלישי, כלומר את ה- Pod (בשלב זה כבר נוכל לשים לב כי ה- template הוא תמיד הגדרת ה- Pod שהוא אבן הבסיס [12.2]).

הגדרת Pod כפי שכבר ראינו, לרוב מתחילה בהגדרת המיכל המיעוד לרוץ בו.

לאחר מכן נגדיר משתני סביבה אשר יוזרקו למיכל. ראינו כי הסקריפט להדפסת שם מלא פונה למיקרו-סרוויסים שלנו, ובכל סוג הרצה (מקומית, בתוך רשת דוקר ייעודית, ועכשיו בתוך אשכול קוברנטיס) המיקרו-סרוויסים מקבלים כתובת שונה.

כצפוי, בתוך האשכול, נפנה למיקרו-סרוויסים דרך משאב ה- Service אשר חושף אותם, לכן אלו ערכיהם של משתני הסביבה.

לבסוף, יש להגדיר את מדיניות האתחול של ה- Job (במידה ונכשל), במקרה זה אין פה פעולה מורכבת אלה סקריפט פשוט ולכן אגדיר שלא יתבצע אתחול למען הפשטות.

כעת נוכל לפרוס את ה- CronJob באמצעות הפקודה

```
kubectl apply -f "kubernetes/cronjob.yaml"
```

נראה כי נוצר ה- CronJob, נריץ

```
kubectl get cronjobs
```

נקבל

```
NAME                      SCHEDULE     SUSPEND   ACTIVE   LAST SCHEDULE   AGE
print-full-name-cronjob   * * * * *    False     0        <none>          11s
```

בהגעה לדקה עגולה, יווצר לנו ה- Job הראשון, נריץ

```
kubectl get jobs
```

נקבל

```
NAME                                  COMPLETIONS   DURATION   AGE
print-full-name-cronjob-1598246580    1/1           5s         5s
```

ועבורו יווצר גם Pod, נריץ

```
kubectl get pods
```

נקבל

```
NAME                                                  READY   STATUS      RESTARTS   AGE
microservice-first-name-deployment-799565cbdb-7ldqv   1/1     Running     0          15h
microservice-last-name-deployment-576566b475-wflh8    1/1     Running     0          15h
print-full-name-cronjob-1598246580-xpk5q              0/1     Completed   0          13s
```

(כמובן שנראה גם את ה- Pod-ים של המיקרו-סרוויסים שלנו).

עבור Pod נוכל לראות את ההדפסה שלו, נריץ

```
kubectl logs "print-full-name-cronjob-1598246580-xpk5q"
```

ונקבל

```
$ k logs print-full-name-cronjob-1598246580-xpk5q
Guy Carmy
```

בשלב זה יש לנו מערכת עובדת, עטופה במיכלים ופרוסה (deployed) בקוברנטיס, אך לא נעצור כאן. בפרק הבא אמנה כמה בעיות שיכולות לפגוע בזמינות המערכת שלנו, ונראה איך מערכת קוברנטיס פותרת בעיות אלו. את הפתרונות נוסיף למערכת שלנו להשלמת הדוגמה.

זמינות מערכות - Availability

נהוג להגדיר את הזמינות (availability) של מערכות בכמות הזמן, בדרך כלל באחוזים בחודש או בשנה, בה יש הפסקה (outage) בשירות אותו המערכת אמורה לספק, הגדרת הזמינות כוללת גם זמני תחזוקה מתוכננים וגם הפסקות שירות שאינן מתוכננות [10].

הסתכלות אחרת על מספר זה תהיה "הסבירות שבה מערכת זמינה כאשר צריך אותה".

כל חברות הענן הגדולות (אמזון, גוגל, מיקרוסופט) מגדירות את זמינות המערכות שלהם ברמה המשפטית בהסכמי רמת השירות (Service Level Agreement - SLA), והסכמים אלו מחריגים מקרים ספציפיים אשר לא נספרים כנגדם בחישוב הזמינות.

מחשוב הענן של גוגל (Google Cloud Compute Engine) למשל, החריג בעבר (עד נובמבר 2016) כל הפסקה בשירות של עד 5 דקות, כלומר הפסקות בשירות לצורך שדרוג גרסה אשר נמשכו 3 דקות, לא הצטברו לכדי פגיעה בהסכם (כיום הסכם השירות מחריג הפסקות של עד דקה אחת) [11].

זמינות של 99.9% מהזמן, משמעותה שהשירות אינו זמין במשך כ- 525 דקות בשנה (במצטבר).

ישנם גורמים רבים להפסקה בשירות של מערכת, בחלק זה מניתי בקצרה את חלקן.

גורמים לאי זמינות מערכות

בעיות תוכנה

בעיות תוכנה יכולות להיגרם מסיבות רבות, החל מטעויות אדם פשוטות כמו שגיאת הקלדה, טעויות לוגיות, שימוש לא נכון בכלים או חבילות וכדומה.

בעיות בניהול זיכרון למשל, יכולות לגרום לגלישה מכמות הזיכרון המותרת, ולכן לשגיאות בתפקודה של המערכת.

ישנן גם בעיות הנגרמות כתוצאה ממצב לא צפוי, למשל קלטים שאינם מטופלים כראוי ולכן גורמים לשגיאה ולכישלון של המערכת, ומכאן לחוסר זמינות שלה.

בעיות חומרה

נתחיל מהנדיר והמוזר ביותר, פגיעה פיזית, למשל הצפה של חדר שרתים, יכולה לגרום לבעיות זמינות. מפגעי מזג אוויר יכולים לגרום לנזקים תשתיתיים רבים שיובילו לחוסר זמינות.

לפעמים הוירטואליזציה והשימוש במילה "ענן" עבור שירותי מחשוב מודרניים עלול לבלבל, בסופו של דבר ברור לכולנו שמדובר במחשבים פיזיים הנמצאים במקום כלשהו.

ישנן כמובן בעיות חומרה פחות איזוטריות, תקלה במערכת מיזוג האוויר (קירור) בחוות השרתים, ככל הנראה תוביל להתחממות יתר של רכיבי חומרה עד לכשלונם.

בעיות בזרם החשמל יכולות לגרום נזק כמובן, מתח נמוך מידי או גבוה מידי יובילו לעבודה לא תקינה של רכיבי חומרה. הפסקת חשמל פשוטה גם כן תוביל לבעית זמינות.

חלק מרכיבי החומרה מבצעים פעולות מכניות, למשל דיסק קשיח מסתובב הוא רכיב חומרה רגיש יחסית ותקלה בו עלולה להוביל לתקלה בשירות.

בנוסף לאלו, בעיות הנגרמות כתוצאה ממצב לא צפוי (באגים) קיימות גם ברכיבי חומרה, ניסיון לבצע פעולה שרכיב החומרה לא תומך בה עלול להוביל לכישלונו.

בעיות חיצוניות

ישנן בעיות הנובעות מסיבה חיצונית למערכת, עומס רב (יום הנחות מיוחד דוגמת Black Friday על אתרי קניות למשל) יכול לגרום למערכת להיכשל בהתמודדות עם העומס ומכאן לאבד מזמינותה.

שימוש ב- Kubernetes לפתרון בעיות זמינות של מערכות

בפרק הקודם הראיתי מספר אתגרים המשפיעים על הזמינות של מערכת.

בחלק זה של העבודה אסקור איך מערכת קוברנטיס ניגשת לאתגרים אלו ומגשרת עליהם, על מנת לשפר את זמינות המערכות הפרוסות בה.

Auto Healing

על מנת לפתור בעיות זמינות מהסוג שהגדרנו כבעיות תוכנה, מערכת קוברנטיס מכילה יכולות ל"ריפוי אוטומטי" של מיכלים.

כפי שראינו, על כל מכונה באשכול יש את רכיב ה- kubelet, שכאמור, אחד מתפקידיו הוא לדאוג שכל המיכלים השייכים ל- Pod-ים שהוקצו לאותה מכונה, רצים.

על מנת לאפשר ל- kubelet לבדוק את זמינותו של המיכל, ישנן שתי בדיקות הניתנות להגדרה עבור כל מיכל, חיות (liveness) ומוכנות (readiness).

את הבדיקות מגדירים בהצהרת התצורה של ה- Pod (בפועל, בקובץ ה- Deployment), ישנן שתי דרכים עיקריות לבצע בדיקות אלו, על ידי קריאת API למיכל, או על ידי הרצת פקודות בתוך המיכל. אם ה- API או הרצת הבדיקה מחזירים שגיאה, אז הבדיקה נכשלת [12.10].

שתי הבדיקות מתבצעות כל פרק זמן נתון (ברירת המחדל היא כל 10 שניות), כאשר אם בדיקת החיות נכשלת, אז המיכל מאותחל, ואילו אם בדיקת המוכנות נכשלת, אז אל אותו המיכל לא יועברו בקשות (כלומר, אל אותו ה- Pod לא יועברו קריאות מה- Service).

דוגמה

נוסיף את בדיקות החיות והמוכנות למיקרו-סרוויסים שלנו, כך שיהיו זמינים יותר.

תחילה, נוסיף פונקציות ו- API routes שיאפשרו לקוברנטיס לפנות אל המיקרו-סרוויסים ולקבל מהם תשובה.

הוספת פונקציות חיות ומוכנות למיקרו-סרוויסים

עבור מיקרו-סרוויס-שם-פרטי, נוסיף את הפונקציות הבאות לקובץ main.go

```
func readinessHandler(w http.ResponseWriter, req *http.Request) {
      fmt.Println("Handles readiness request")
      fmt.Fprint(w, "im ready")
      return
}

func livenessHandler(w http.ResponseWriter, req *http.Request) {
      fmt.Println("Handles liveness request")
      fmt.Fprint(w, "im alive")
      return
}
```

במקרה זה הפונקציות לא עושות דבר מלבד לוודא שהסרוויס אכן רץ. במקרים "אמיתיים", רצוי לבדוק שכל התנאים לתפקוד תקין של הסרוויס מתקיימים, למשל שיש גישה לבסיס הנתונים שלו.

כמו כן נוסיף את השורות הבאות לפונקצית ה- main באותו קובץ על מנת להגדיר את ה- routes

```
      http.HandleFunc("/_ready", readinessHandler)
      http.HandleFunc("/_alive", livenessHandler)
```

עבור מיקרו-סרוויס-שם-משפחה, נוסיף את הפונקציות הבאות לקובץ main.py

```
@api.route('/_ready')
def readiness():
    return "im ready"

@api.route('/_alive')
def liveness():
    return "im alive"
```

כעת, נוסיף את השימוש בפונקציות אלו על ידי קוברנטיס

הוספת בדיקות חיות ומוכנות למיקרו-סרוויסים

```
        livenessProbe:
          httpGet:
            path: /_alive
            port: 8090
        readinessProbe:
          httpGet:
            path: /_ready
            port: 8090
```

(את שורות אלו נוסיף בקובץ deployment.yaml תחת הגדרת ה- container).

אלו השורות עבור מיקרו-סרוויס-שם-פרטי, התוספת עבור המיקרו-סרוויס השני זהה עד כדי שינוי הפורט שבשימוש.

Scheduling

מערכות לניהול מיכלים נדרשות לאזן בין הצורך למצות את המשאבים הקיימים במכונות המרכיבות את האשכול (resource utilization), לבין הצורך להבטיח רמת שירות (QoS - Quality of Service) נאותה למיכלים שרצים בו [7].

שיבוץ בקוברנטיס הוא הפעולה בו מתאימים בין Node לבין Pod, כלומר בוחרים למיכל את המכונה עלייה הוא ירוץ [12.5].

פעולה זו מתבצעת על ידי kube-scheduler אשר נגענו בו כאחד מרכיבי ה- Master.

הפעולה מתבצעת בשני שלבים, שלב הסינון (Filtering) ולאחריו שלב הניקוד (Scoring).

בשלב הסינון יפסלו כל המכונות אשר לא יכולות להתאים ל- Pod הנדרש, למשל מכונות שאין להן מספיק משאבים (CPU, Memory), מכונות שאין גישה מהן לאחסון הנדרש (Persistent Volume), מכונות שבהן ה- Port הנדרש כבר תפוס ועוד.

בשלב השני ניתן ניקוד לכל מכונה שעברה את השלב הראשון, ולבסוף נבחרת המכונה בעלת הניקוד הגבוה ביותר, זהו השלב לו הפוטנציאל להשפיע על זמינות המערכת [12.8].

בברירת המחדל, המערכת תעדיף פיזור על פני אזורי זמינות (regions / zones) שונים, עבור Pod-ים השייכים ל- StatefulSet או ReplicaSet. לדוגמה, אם ליישום מסוים כבר יש שני Pod-ים פרוסים, אחד על מכונה באזור זמינות A ואחד על מכונה באזור זמינות B, אז מכונות מאזור זמינות C ו-D יקבלו ניקוד גבוה יותר בשלב זה.

כך מערכת קוברנטיס מגשרת על בעיות זמינות שהגדרנו בתור בעיות חומרה, בחירת מכונה מאזור זמינות אחר מעלה את שרידות המערכת הפרוסה בקוברנטיס, שכן הסבירות לכשל (הצפה או נזק תשתיתי) באזורים שונים במקביל נמוכה יותר.

אלמנט נוסף המשפיע על איכות השירות (QoS), הוא דירוג ה- Pod-ים לפינוי [12.11].

במידה ויישום משתמש בכמות שונה של משאבים (זיכרון וכוח עיבוד) בזמנים שונים, נוצר מתח בין שיבוץ מיכלים על פי הצריכה באותו רגע (כך אולי ניתן לשבץ יותר מיכלים ולנצל את כל משאבי האשכול), לבין היכולת לאפשר למיכל להעלות את כמות המשאבים שלו במידת הצורך (על מנת לספק את השירות שלשמו הוא נוצר) [7].

על מנת להקל על ההתמודדות הזו, קוברנטיס מאפשרת להצהיר על מינימום משאבים עבור מיכלים של יישום מסוים, ומבטיחה את הכמות הזו, בין אם תנוצל בפועל ובין אם לאו.

בנוסף, היכולת להצהיר על מקסימום משאבים עבור מיכלים של יישום מסוים, מאפשרת למערכת לתת למיכל משאבים נוספים (מעבר למינימום המוצהר) במידה ואלו פנויים, וכך לנצל משאבים שלא בשימוש. מהצד השני, המערכת יכולה למנוע משירות מסוים להשתלט על כל

המשאבים ובכך לפגוע בזמינות של מיכלים אחרים. כאשר מיכל מסוים דורש יותר מדי משאבים (כלומר יותר מהמקסימום שהוצהר), המערכת תדע לאתחל אותו.

כדי לעודד שימוש בהצהרת מינימום ומקסימום משאבים, המערכת מתעדפת Pod-ים אשר הוגדרו עבורם דרישות משאבים.

ישנן שלוש דרגות על פיהן קוברנטיס מחליט איזה Pod יפונה במידת הצורך.

הדרגות הן Guaranteed, Burstable, ו- BestEffort.

כחלק מהגדרות Pod, ניתן להצהיר על כמות המשאבים הדרושה (מינימום) ועל הרף העליון עבור כמות המשאבים (מקסימום).

על מנת לקבל דרגת Guaranteed, צריך להצהיר מינימום ומקסימום, ועליהם להיות שווים.

כך המערכת יכולה לשבץ בביטחון את ה- Pod-ים.

אם ישנו רק מינימום או רק מקסימום, תתקבל דרגת Burstable.

אם אין הגדרות משאבים כלל, תינתן דרגת BestEffort.

דוגמה

ראינו בסעיף זה שתי אלמנטים לשיפור הזמינות.

הגדלת כמות העותקים ופיזורם לאזורי זמינות

תחילה, על מנת למצות את יכולות הפיזור לאזורי זמינות שונים, צריך יותר מעותק אחד של כל יישום. נגדיל את מספר העותקים ל- 2 (או יותר).

```
  replicas: 2
```

(שורה זו כבר קיימת בקבצי ה- deployment.yaml של שני המיקרו-סרוויסים. נשנה את המספר)

כעת, ברירת המחדל היא שכל עותק ישובץ לאזור זמינות אחר. זאת במידת האפשר כמובן.

כאשר האשכול בנוי ממכונה יחידה (כמו אשכול minikube המשמש להדגמה), מן הסתם לא נקבל פיזור בין מכונות.

הגדרת משאבים נדרשים ומקסימליים לקבלת דירוג QoS גבוה

שנית, על מנת לקבל דרגת שירות גבוהה, נגדיר דרישות מינימום ומקסימום זהות למשאבים

```
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
          limits:
            cpu: 100m
            memory: 100Mi
```

(את שורות אלו נוסיף בקובץ deployment.yaml תחת הגדרת ה- container).

כעת נצפה לראות את השורה

```
QoS Class:       Guaranteed
```

עבור Pod-ים של היישום, למשל על ידי הרצת

```
kubectl describe pod
"microservice-first-name-deployment-799565cbdb-7ldqv"
```

Autoscaling

במערכת קוברנטיס ניתן להגדיר סקיילרים אוטומטיים על פי מאפיינים שונים.

תפקיד האוטו סקיילר הוא שתהיה כמות מספיקה של עותקים מהיישום על מנת שהשירות הניתן על ידי היישום יהיה זמין מצד אחד, ולשמור על יעילות בשימוש במשאבים מצד שני [6].

כך מערכת קוברנטיס מונעת מצבים בהם עומס רב גורם לאי זמינות של יישומים, ומגשרת על בעיית הזמינות שהוגדרה כבעיה חיצונית.

ישנם שני סוגים של אוטו-סקיילרים, רוחבי ואורכי.

הראשון והנפוץ הוא Horizontal Pod Autoscaler, והוא משנה את מספר העותקים (replicas) של יישום מסוים בהתאם לצורך.

המימוש הבסיסי ביותר הוא לפי כמות המשאבים (CPU, Memory), כך למשל ניתן להגדיר כי אם ממוצע ה- CPU אשר בשימוש על פני ארבעת העותקים של Pod מסוים היא מעל 70%, יעלה Pod חמישי והעומס יחולק בין חמישה Pod-ים [12.7]

ישנם מימושים נוספים ומתוחכמים יותר עבור סקיילרים אוטומטיים, למשל על פי אורך של תור מסוים, או על פי מדדי ביצועים בהתאמה אישית (Custom Metrics).

הסוג השני והפחות נפוץ הוא Vertical Pod Autoscaler, והוא משנה את כמות המשאבים המינימלית והמקסימלית המבוקשת על ידי כל עותק של היישום.

סוג זה הוא חדש יותר ועדיין יש לו מגבלות, לכן לא ארחיב עליו בעבודה זו.

דוגמה

הוספת Horizontal Pod Autoscaler

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: microservice-first-name-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: microservice-first-name-deployment
  minReplicas: 2
  maxReplicas: 4
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

(את התצורה נשמור בקובץ hpa.yaml בתיקיית kubernetes של המיקרו-סרוויס המתאים)

נוכל לראות שזהו HorizontalPodAutoscaler המתייחס ל- Deployment ששמו microservice-first-name-deployment. כאשר כמות העותקים משתנה דינמית בין 2 ל-4 על פי העומס המוגדר על ידי כמות ה- CPU הממוצעת שבשימוש בכל עותקי היישום.

האוטו-סקיילר ישאף שה- CPU הממוצע בין כל עותקי היישום יהיה 50%. כך, אם ישנם 4 עותקים וכל אחד מהם משתמש רק ב- 25% ממשאבי העיבוד שלו, אפשר להוריד 2 עותקים וכך לצמצם את העלויות. לעומת זאת אם ישנם 2 עותקים ב- 75%, האוטו-סקיילר יעלה עותק נוסף על מנת לעמוד בעומס ולהימנע ממצב של אי זמינות.

הקובץ המתאים עבור המיקרו-סרוויס השני זהה לחלוטין בשינוי השם של ה- Deployment אליו האוטו-סקיילר מתייחס.

סיכום

בעבודה זו ראינו כיצד כלים כמו מיכלים ומערכות לניהול מיכלים, ובפרט Docker ו- Kubernetes, מגשרים על החסרונות של מערכות תוכנה בארכיטקטורת Microservices ומאפשרים למצות את היתרונות של מערכות אלו, תוך השגת זמינות ויציבות עבור המערכת.

לארכיטקטורת מיקרו-סרוויסים יתרונות רבים גם בפן העסקי וגם בפן הטכנולוגי. ראינו כי ניתן לפרוס את המערכת ביעילות רבה יותר ולהתאימה לעומס משתנה, ישנה גמישות טכנולוגית רבה יותר המאפשרת לנצל יתרונות של טכנולוגיות לצרכים ספציפיים, מה שמאפשר גם גמישות בגיוס הצוות.

בהדגמה, ראינו כיצד אפשר לבנות מערכת הבנויה ממיקרו-סרוויסים הכתובים בשפות שונות על מנת לנצל את היתרון היחסי של כל שפה.

מיכלים הם פתרון לעטיפה של אותם מיקרו-סרוויסים יחד עם התלויות שלהם, כך ניתן לפתח אותם ולאחר מכן להריץ אותם על כל מכונה באותם התנאים.

בהדגמה ראינו עבור כל מיקרו-סרוויס תמונה המייצגת אותו יחד עם כל התלויות שלו. כמו כן, ראינו כי לאחר יצירת התמונות ולמרות שהמיקרו-סרוויסים כתובים בשפות שונות, הרצתם במיכל נעשית באופן מאוד דומה.

על מנת להתמודד עם מספר רב של מיכלים אשר להם צרכים שונים (משתני סביבה, הגדרות תקשורת, צפי לעומס שונה ועד), פגשנו במערכות לניהול מיכלים.

ראינו את חלקי מערכת קוברנטיס, המורכבת ממכונת "מנהל", אשר מכילה את רכיבי ה-Control Plane, וממכונות "פועלים" עליהן רצים המיכלים.

בהדגמה, ראינו כיצד נשתמש במשאבים Deployment ו- CronJob על מנת להגדיר תצורה עבור היישומים השונים שלנו, בהתאם לצרכים שלהם. ראינו כי בסופו של דבר היישום רץ בתוך Pod העוטף את המיכל ותצורתו. כמו כן הגדרנו Service על מנת לאפשר את הגישה אל ה- Pod-ים שנוצרו.

מנינו מספר בעיות פוטנציאליות שיכולות לגרום לאי זמינות המערכת, וראינו איך קוברנטיס ניגשת לפתור אותן.

איך אפשר לפתור בעיות תוכנה פשוטות באמצעות בדיקות חיות וזמינות, ושימוש ב"ריפוי אוטומטי". איך אפשר לגשר על בעיות חומרה באמצעות שיבוץ נכון של ה- Pod-ים, ואיך אפשר להתמודד עם עומס משתנה באמצעות סקיילרים אוטומטיים.

בהדגמה, ראינו איך נגדיר כל אחד מאלו עבור המערכת שלנו.

באופן אישי, בחרתי לסקור את ארכיטקטורה זו וכלים אלו מכיוון שיש לי מעט ניסיון מקצועי בהם ורציתי להעמיק את הידע סביבם. במהלך כתיבת העבודה למדתי המון על הכלים בהם השתמשתי.

דוקר הוא כלי נוח מאוד לעבודה ולפריסה. הוא שם סוף לטיעון "אצלי במחשב זה עבד".

כמו כן, מערכת קוברנטיס נפוצה מאוד ובכל העננים המקובלים (AWS, GCP, Azure ועוד) ניתן להקים מערכות קוברנטיס מנוהלות באופן פשוט ומיידי.

אני חושב שזוהי ארכיטקטורה טובה, אשר מצד אחד ובאמצעות הכלים שמניתי ניתן להגיע איתה למוצר מתפקד באופן מהיר, כאשר בהתחלה נשמור על מספר מיקרו-סרוויסים קטן על מנת להימנע מסיבוכים.

מהצד השני, בחירת ארכיטקטורה זו היא הכנה טובה לעתיד, כאשר הצוות והעומס על המערכת גדלים, הארכיטקטורה מתאפיינת ב"סקיילאביליות" גבוהה, הן על ידי הגדלת מספר המיקרו-סרוויסים והן על ידי הגדלת הכמות של כל מיקרו-סרוויס, בהתאם לצרכים המשתנים.

מקורות

- [1] Pahl, C., & Lee, B. (2015). Containers and Clusters for Edge Cloud Architectures - A Technology Review. 2015 3rd International Conference on Future Internet of Things and Cloud. doi:10.1109/ficloud.2015.35
- [2] Containerization. (n.d.). Retrieved from https://www.ibm.com/cloud/learn/containerization
- [3] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. Linux J. 2014, 239, Article 2 (March 2014), 1 pages.
- [4] Docker overview. (2020, August 26). Retrieved from https://docs.docker.com/get-started/overview/#docker-architecture
- [5] Microservices. (n.d.). Retrieved from https://www.ibm.com/cloud/learn/microservices
- [6] Taherizadeh, S., & Grobelnik, M. (2020). Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. Advances in Engineering Software, 140, 102734. doi:10.1016/j.advengsoft.2019.102734
- [7] Truyen, E., Landuyt, D. V., Preuveneers, D., Lagaisse, B., & Joosen, W. (2019). A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks. Applied Sciences, 9(5), 931. doi:10.3390/app9050931
- [8] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, & Ferhat Khendek. (2019). Kubernetes as an Availability Manager for Microservice Applications.
- [9] Kubernetes. (n.d.). Kubernetes/kubernetes. Retrieved from https://github.com/kubernetes/kubernetes
- [10] Mina Nabi, Maria Toeroe, & Ferhat Khendek (2016). Availability in the cloud: State of the artJournal of Network and Computer Applications, 60, 54 - 67.
- [11] Compute Engine Service Level Agreement (SLA). (n.d.). Retrieved from https://cloud.google.com/compute/sla
- [12] Production-Grade Container Orchestration. (n.d.). Retrieved from https://kubernetes.io/

- [12.1] Kubernetes Components. (2020, August 28). Retrieved from https://kubernetes.io/docs/concepts/overview/components/
- [12.2] Pods. (n.d.). Retrieved from https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/
- [12.3] Controllers. (n.d.). Retrieved from https://kubernetes.io/docs/concepts/workloads/controllers/
- [12.4] Service. (2020, August 04). Retrieved from https://kubernetes.io/docs/concepts/services-networking/service/
- [12.5] Kubernetes Scheduler. (2020, July 17). Retrieved from https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
- [12.6] Controllers. (2020, August 21). Retrieved from https://kubernetes.io/docs/concepts/architecture/controller/
- [12.7] Horizontal Pod Autoscaler. (2020, June 20). Retrieved from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
- [12.8] Scheduling Policies. (2020, June 02). Retrieved from https://kubernetes.io/docs/reference/scheduling/policies/
- [12.9] Recommended Labels. (2020, July 10). Retrieved from https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/
- [12.10] Configure Liveness, Readiness and Startup Probes. (2020, August 04). Retrieved from https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- [12.11] Configure Quality of Service for Pods. (2020, August 11). Retrieved from https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/

- [13] Etcd. (n.d.). Retrieved from https://etcd.io/
- [14] Kubernetes. (2019, May 05). Kubernetes/container-runtime-interface. Retrieved from https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md

- [15] Package http. (n.d.). Retrieved from https://golang.org/pkg/net/http/
- [16] Flask. (n.d.). Retrieved from https://pypi.org/project/Flask/