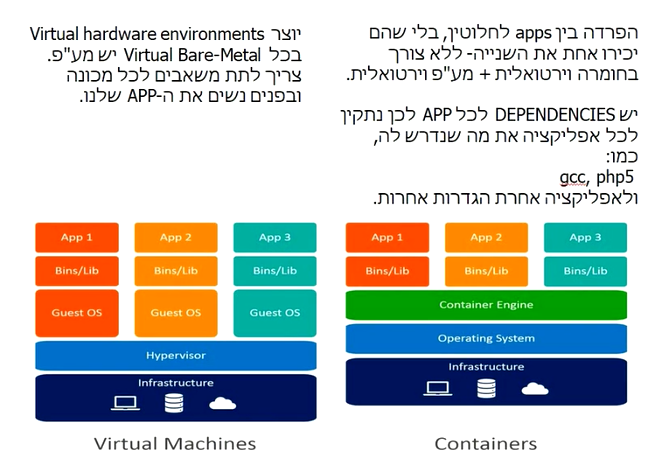
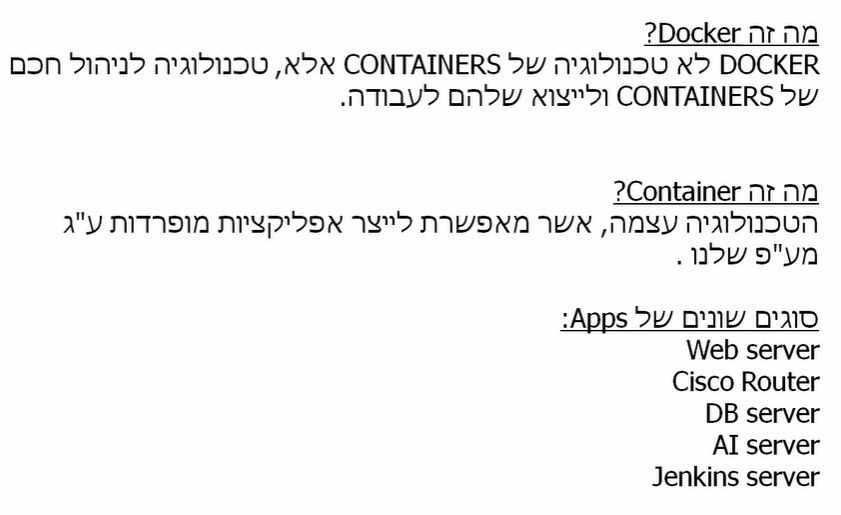
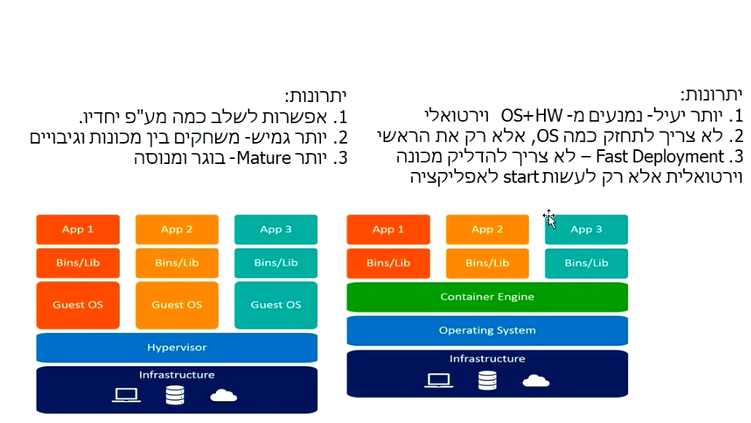
DOCKER-CONTAINER VIA AWS

1.לאחר שייצרנו מכונה בענן ןקיבלנו את המפתח נתחבר למכונה בעזרת

נכניס את הכתובת החיצונית ונטען את המפתח MOBAXTREN

מה זה בעצם ?DOCKER-CONTAINER\*



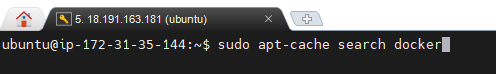


תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי







ןנבחר



ואז נריץ



השיטה הזאת פחות מומלצת כי אנחנו מסתמכים על הספריות של אובונטו ןלכן אנחנו ךא תמיד נקבל עדכונים בזמן

לעומת זאת אפשר לבצע בדרך יותר מומלצת !!!

ניגש לאתר <https://get.docker.com/>

ונריץ את שורת הפקודה הבאה :

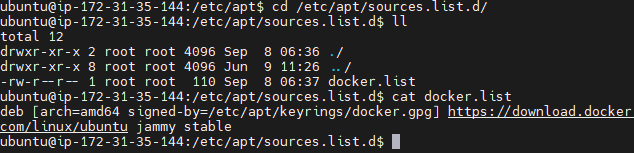
curl -fsSL https://get.docker.com -o get-docker.sh

שהיא בעצם מושכת את כל מה שרשום באותו אתר לתוך קובץ get-docker.sh

נריץ את הקובץ בפקודה

sh get-docker.sh

וכעת יש לנו דוקר מותקן על המכונה בענן והוא מתעדכן ישירות מול חברת דוקר ולא דרך אובונטו

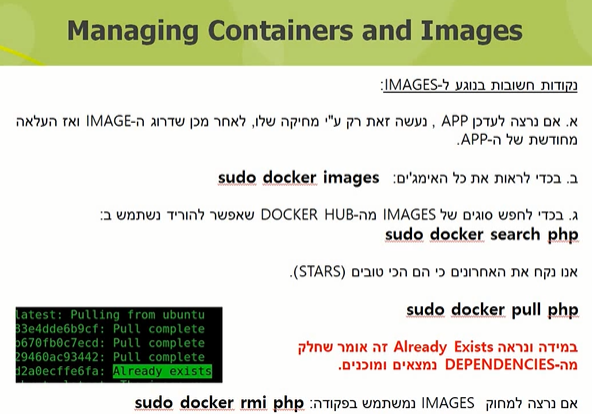


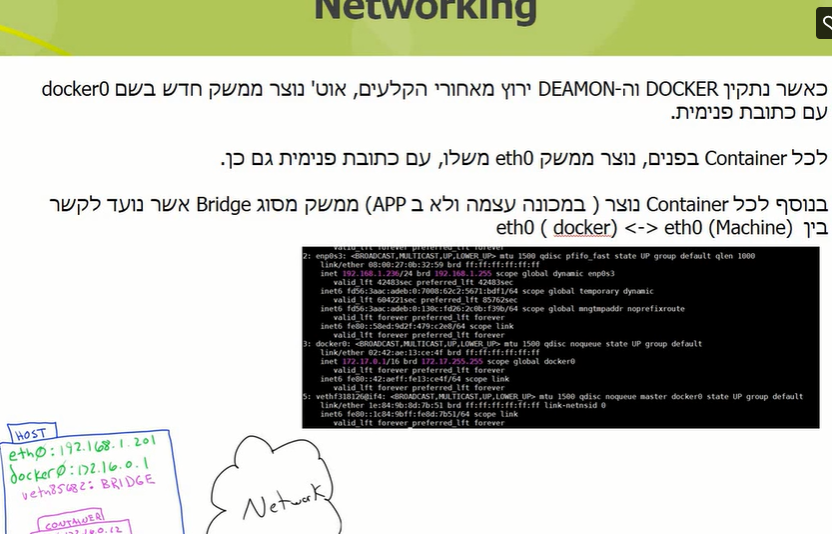


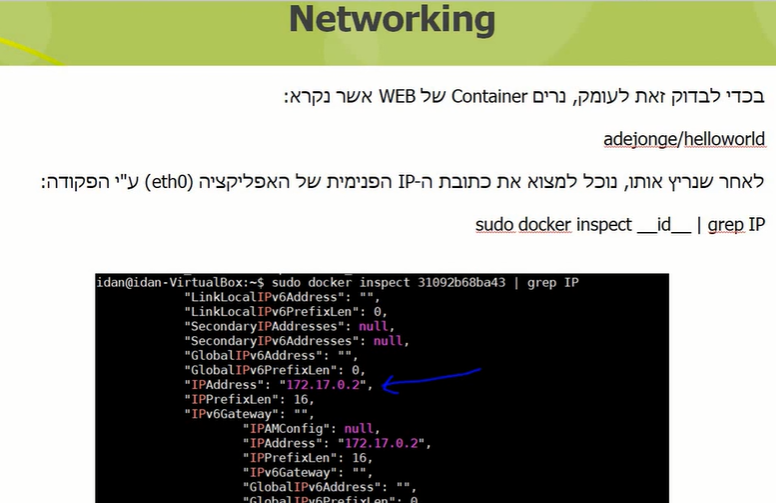


תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי







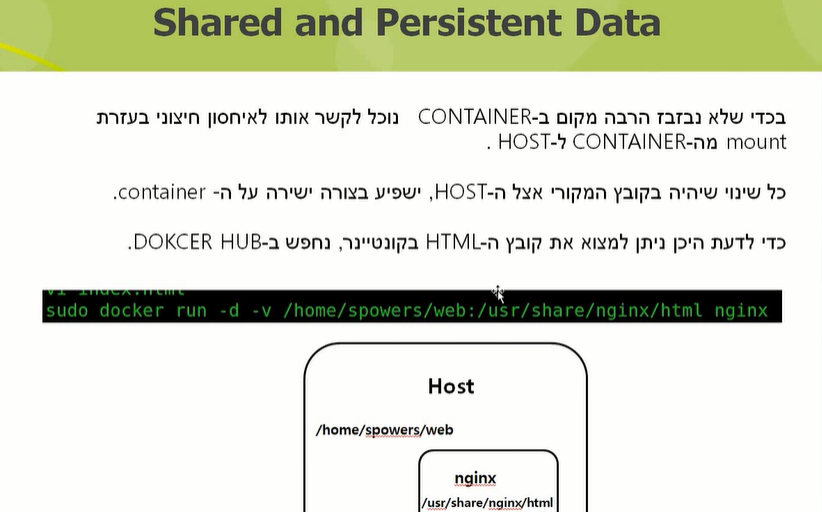
Dockerui

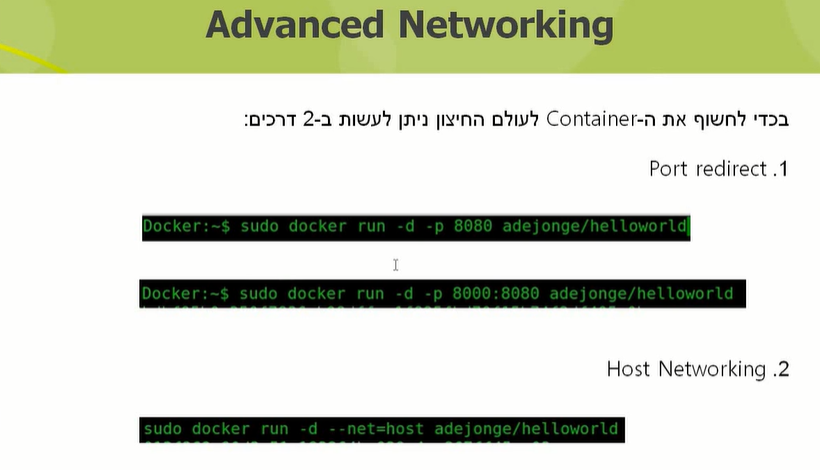
docker pull abh1nav/dockerui:latest

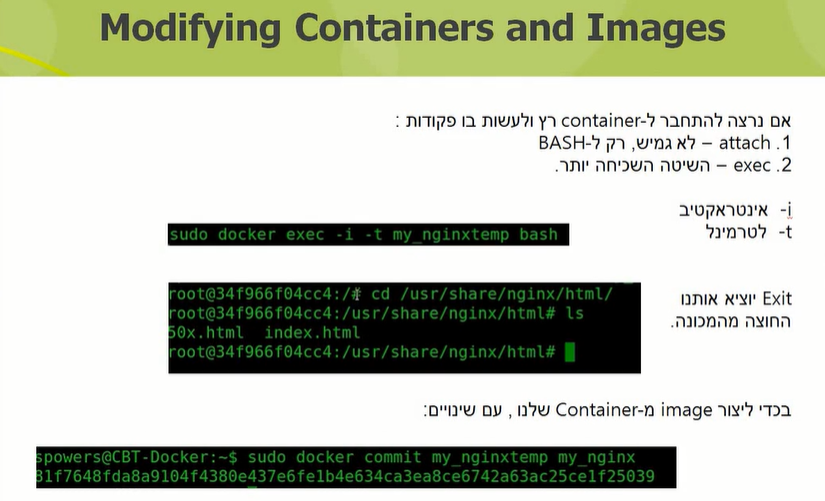
+ הפקודה הבאה

docker run -d -p 9000:9000 --name dockerui \

abh1nav/dockerui:latest -e="http://<dockerd host ip>:4243"







נרשום

Cat > index.html

ונוכל לערוך את הדף לדוגמא נרשום

+ ctrl+D<h1>shay dgani web image </h1>

נרשום שוב Cat > index.html

ונקבל :

תמונה שמכילה טקסט, כתום

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

לסיום נרשום EXIT

כעת ניצור image

sudo docker commit shay\_nginx shay\_image\_nginx

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

\*\*

על מנת למחוק את כל האימאגים צריך לרשום לולאה :

for i in `sudo docker ps` ;do `sudo docker stop $i` ; done

+

for i in `sudo docker ps -a` ;do `sudo docker rm $i` ; done

תמונה שמכילה טקסט

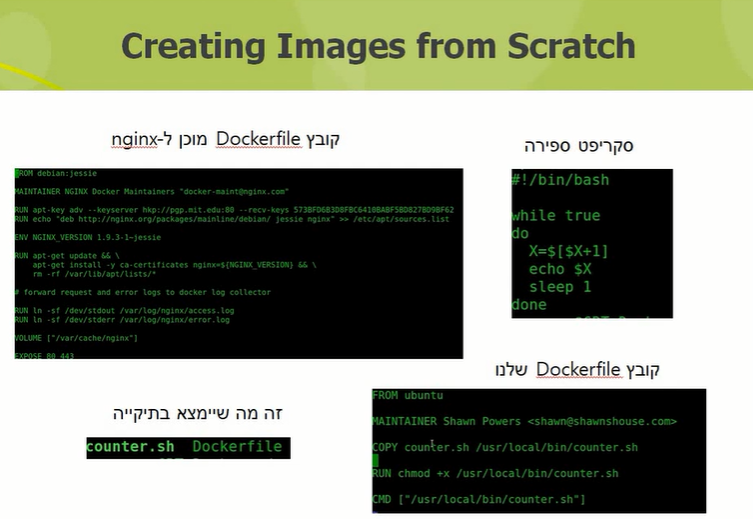
התיאור נוצר באופן אוטומטי

עכשיו ניצור מחדש את הקונטיינר מהאימאג שיצרנו

sudo docker run -d -p 80 --name shay\_nginx shay\_image\_nginx

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטימעולה!!



דוגמא ל DOCKER FILE

From ubintu

MAINTAINER shay.dgani <shayd@digital.gov.il>

COPY counter.sh /usr/local/bin/counter.sh

RUN chmod +x /usr/local/bin/counter.sh

RUN set -x \

&& apt-get update && apt-get upgrade -y

# Forwrd request logs to Docker log collector #

RUN ln -sf /dev/stdout /var/log/access.log \

&& ln -sf /dev/stderr /var/log/error.log

CMD ["/usr/local/bin/counter.sh"]

נבנה עכשיו את האימאג

sudo docker build -t shay\_counter2 .

כולל המקודה בסוף!!

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

משימה:

תמונה שמכילה טקסט

התיאור נוצר באופן אוטומטי

**הרחבה:**

מה זה דוקר?

דוקר הוא פלטפורמת תוכנה שמאפשרת לבנות, לשתף ולהריץ יישומים בתוך מיכלים (Containers).

בשביל להבין טוב יותר למה צריך פלטפורמה כזאת בואו נניח שיש לכם מערכת שמורכבת ממספר רכיבים: בסיס נתונים, שרת ווב, שרת יישום, סרביסים קטנים שעוזרים ליישום לרוץ ועוד.

בלי דוקר אתם עלולים להיתקל במספר בעיות בעבודה עם מערכת כזאת. קודם כל בסביבת הפיתוח:

1. יכול להיות שתרצו לנסות לפתח גירסא חדשה של המערכת מול גירסא חדשה יותר של בסיס הנתונים, אבל עדיין לשמור על בסיס הנתונים הישן מותקן על המחשב כדי לתקן באגים בגירסת הפרודקשן.
2. ובעצם אותה הבעיה קיימת עם כל רכיב תשתית שבחרתם - שידרוג של השרת, או של גירסת שפת התכנות או ספריות מסביב. כל אלה יכולים ליצור התנגשויות כשמנסים לשלב מספר גירסאות על אותה מכונת פיתוח.
3. כשנכנס מפתח חדש לצוות הוא עלול למצוא את עצמו מבלה יום שלם בהתקנות. התקנת כל החלקים והתשתיות של המערכת על מכונה חדשה נהיית מורכבת יותר ככל שיש יותר רכיבים.

ובאותו האופן בשלב המעבר לייצור אתם עשויים להיתקל במספר בעיות:

1. קושי להוסיף או להוריד "שרתים" מהרשת - מאחר וכל שרת צריך להחזיק את כל רכיבי התשתית כדי להריץ את המערכת.
2. בעיית ניצול של השרתים הקיימים בגלל התקנות גלובליות - יכול להיות שהשרת שלכם מספיק "פנוי" רוב הזמן כדי להריץ עוד רכיב במערכת, אבל הרכיב הזה דורש גירסא אחרת של בסיס הנתונים ואתם לא רוצים ליצור התנגשות.

דוקר יודע לפתור חלק גדול מהבעיות האלה באמצעות שימוש במיכלים. כל מיכל מחזיק בתוכו אפליקציה אחת או יותר ודוקר עצמו יודע "להפעיל" מיכלים כאלה בתוך סביבה חצי מבודדת. זה לא מכונה וירטואלית נפרדת אלא משהו הרבה יותר קליל. התהליכים והקבצים נשמרים ישירות על המכונה הנוכחית, אבל בנפרד מהמערכת הראשית ובעצם אנחנו נותנים ליישום הרגשה שהוא רץ לבד, למרות שהוא בעצם רץ עם עוד הרבה יישומים אחרים.

חשוב לציין שבגלל שאין פה הפרדה מוחלטת בין המיכלים אם מיכל מסוים ירצה לשבור לכם את השרת או להאזין למיכלים אחרים הוא יוכל לעשות את זה. לכן חשוב לזכור שדוקר אינו מכונה וירטואלית וחשוב להריץ רק מיכלים שאתם סומכים עליהם.

.

ההבדל בין אימג'ים לקונטיינרים?

לפני שנתחיל לדבר על קונטיינרים צריך לדבר על Image-ים. אימג' היא חבילה שכוללת את כל מה שיישום צריך כדי לרוץ: הקוד, ספריות בהן הוא תלוי, הגדרות, משתני סביבה וכן הלאה. האימג' נשמר לכם על המחשב ואתם יכולים ליצור ממנו קונטיינר.

קונטיינר הוא הפעלה של אימג' מסוים.

אפשר לחשוב על הקשר בין השניים כמו הקשר בין "קובץ" ל"תהליך". האימג' מייצג קובץ על הדיסק, וכשמפעילים את האימג' מקבלים "תהליך" בזיכרון. בגלל זה הפקודה שמראה את כל הקונטיינרים שכרגע רצים לכם על המחשב היא:

$ docker ps

בדיוק כמו שהפקודה ps מראה את כל התהליכים.

העבודה עם דוקר מחולקת לשני תהליכים מרכזיים:

1. תהליך אחד הוא כל הטיפול באימג'ים - אימג' הוא ממש "מחשב" קפוא ומוכן לפעולה, ואנחנו יכולים להעביר אותו ממקום למקום כמו שמעבירים קבצים.
2. תהליך שני הוא הפעלת קונטיינר מתוך האימג' - אחרי שהשגנו אימג' (מחבר, מהרשת, או כי בנינו אחד לבד) אנחנו יכולים "להפשיר" אותו כלומר להריץ אותו בתור תוכנית על המחשב.

למרות שקונטיינר מרגיש לפעמים כמו מכונה וירטואלית, במציאות הקונטיינר הוא פשוט תוכנית שרצה לכם על המחשב. מערכת ההפעלה לינוקס יודעת להפעיל תוכנית בתוך סביבה סגורה ולכן מתוך הקונטיינר כשמסתכלים על הקבצים, התהליכים או מערכת ההפעלה - זה נראה שאנחנו רצים לבד על המחשב.

סיכום קצר על דוקר :

דוקר הוא פלטפורמת תוכנה שמאפשרת לבנות, לשתף ולהריץ יישומים בתוך מיכלים (Containers). מיכל הוא סביבה פנימית בתוך המכונה שמבחינת היישום שרץ בתוכה היא נראית כמו מערכת הפעלה מלאה.

בפיתוח בסביבת מונולית היה לנו בסיס נתונים אחד ענק לכל המידע של המערכת, והיתה לנו מכונה אחת שמחזיקה את הפרויקט. התקנות של מכונות חדשות היו נדירות וקוד חדש תמיד השתמש באותו בסיס נתונים.

בעבודה בסביבת Micro Services המערכת שלנו מורכבת מהמון המון מערכות קטנות ובדרך כלל כל מערכת קטנה מגיעה עם רכיבי התשתית שלה. בסביבה כזאת אנחנו רוצים שכל רכיב קטן יהיה כמה שיותר חסכוני במשאבים ושנוכל להרים מהר מאוד עוד עותקים של כל רכיב (או למחוק אותם). אנחנו לא מתקינים פעם אחת ושוכחים מזה, אלא אנחנו מעלים ומורידים רכיבים כל הזמן.

הרצת כל רכיב בתוך Container מאפשרת לנו לעשות בדיוק את זה:

1. קונטיינר נוצר מתוך תבנית שנקראת Image ויצירת קונטיינר היא תהליך מאוד מהיר.
2. קונטיינר מקבל מבחוץ את כל המידע שהוא צריך בשביל לעבוד וכך קונטיינרים יכולים לשתף מידע ביניהם בצורה מבוקרת.
3. קונטיינר איננו מערכת הפעלה מלאה ודורש רק את המשאבים שצריך בשביל היישום שהוא מריץ. הקונטיינר משתמש בכל משאבי המכונה בדיוק כמו אפליקציה מקומית שרצה על השרת.

חשוב לציין שאין פה הפרדה מוחלטת בין המיכלים: אם מיכל מסוים ירצה לשבור לכם את השרת או להאזין למיכלים אחרים הוא יוכל לעשות את זה. דוקר אינו מכונה וירטואלית וחשוב להריץ רק מיכלים שאתם סומכים עליהם.

התקנה והפעלת קונטיינרים:

לפני שנתחיל לדבר על קונטיינרים צריך לדבר על Image-ים. אימג' היא תבנית שכוללת את כל מה שיישום צריך כדי לרוץ: הקוד, הספריות בהן הוא תלוי, ההגדרות, משתני הסביבה וכן הלאה. האימג' נשמר לכם על המחשב ואתם יכולים ליצור ממנו קונטיינר. אפשר לחשוב על אימג' בתור מכונה שמוכנה להרצה - ורק צריך להפעיל פקודה אחת כדי שהמכונה הזאת תתחיל לעשות את מה שאנחנו מצפים ממנה.

וקונטיינר הוא הפעלה של אימג' מסוים.

אפשר לחשוב על הקשר בין השניים כמו הקשר בין "קובץ" ל"תהליך". האימג' מייצג קובץ על הדיסק, וכשמפעילים את האימג' מקבלים "תהליך" בזיכרון. בגלל זה הפקודה שמראה את כל הקונטיינרים שכרגע רצים לכם על המחשב היא:

$ docker ps

בדיוק כמו שהפקודה ps מראה את כל התהליכים.

התקנת דוקר:

בשביל שנוכל להתחיל לייצר קונטיינרים צריך תחילה להתקין את דוקר או ליתר דיוק את המנוע Docker Engine. מנוע זה הוא שמריץ את הקונטיינרים. אפשר לבחור התקנה למערכת ההפעלה שלכם מהדף הזה:

<https://hub.docker.com/search/?type=edition&offering=community>

אם אתם עובדים כמוני על מכונת Windows, תרצו להתקין תוכנה שנקראת Docker Desktop. תוכנה זו כוללת את המנוע של דוקר ועוד כלי עזר שנשתמש בהם בפיתוח. לדוקר יש גם אינגרציה טובה עם WSL, אז אחרי התקנת Docker Desktop מומלץ להיכנס לממשק ההגדרות שלו, ללחוץ שם על אייקון גלגל השיניים ושם לסמן Use the WSL 2 based engine.

אחרי ההתקנה תרצו לוודא שדוקר עובד לכם כמו שצריך. כנסו ל CMD או ל Terminal וכתבו:

$ docker --version

Docker version 20.10.6, build 370c289

במהלך הקורס אני אעבוד עם Docker Desktop שמותקן אצלי על מערכת ההפעלה חלונות. בגלל שאני אוהב לעבוד מתוך WSL אני נכנס אחרי התקנת Docker Desktop למסך ההגדרות ומוודא שהאינטגרציה עם WSL מופעלת. אחרי כל ההתקנות אתם צריכים להגיע למצב שגם מתוך CMD וגם מתוך WSL אפשר לכתוב docker version ולקבל את הגירסה הנכונה.

הפקודה docker run מחפשת אימג' ומריצה אותה. אני יודע שרק הרגע התקנתם את דוקר אז אין לכם עדיין אף אימג' על המחשב, אבל תופתעו לגלות שדוקר יודע לחפש ולהוריד אימג'ים באופן אוטומטי מתוך מאגר עצום של אימג'ים קהילתיים ברשת. אנחנו גם יכולים לייצר מאגרים פנימיים עבור אימג'ים פרטיים של המערכת שלנו.

נתחיל בהתקנה של האימג' מתוך הריפוזיטורי הזה:

<https://github.com/docker-library/hello-world>

האימג' עצמו נבנה וזמין במאגר של דוקר שנקרא Dockerhub וזה הקישור אליו שם:

<https://hub.docker.com/_/hello-world>

יש בו תוכנית קטנה בשם hello שמדפיסה הודעת ברכה על המסך. הפקודה הבאה מורידה את האימג' מדוקר האב, יוצרת ממנו קונטיינר ומפעילה את הקונטיינר:

$ docker run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

1b930d010525: Pull complete

Digest: sha256:2557e3c07ed1e38f26e389462d03ed943586f744621577a99efb77324b0fe535

Status: Downloaded newer image for hello-world:latest

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/get-started/

בואו נראה עכשיו איזה קונטיינרים ואיזה אימג'ים יש לנו. הפקודה הבאה מציגה את כל האימג'ים שהתקנתם על המחשב:

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

hello-world latest fce289e99eb9 7 weeks ago 1.84kB

והפקודה הבאה מציגה את כל הקונטיינרים שיש לכם על המחשב:

$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

4cb8496372d3 hello-world "/hello" 7 minutes ago Exited (0) 3 minutes ago reverent\_sanderson

73177b43ff6d hello-world "/hello" 7 minutes ago Exited (0) 7 minutes ago focused\_bose

ה -a אומר לפקודה להוסיף גם את הקונטיינרים שעכשיו סגורים. שימו לב שאצלי יש שני קונטיינרים לאותו אימג' בגלל שהפעלתי את docker run פעמיים.

אנחנו יכולים להפעיל מחדש קונטיינר שנסגר עם הפקודה:

$ docker start -a 4cb8496372d3

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.

2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

(amd64)

3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it

to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/get-started/

שימו לב להחליף את 4cb8496372d3 במזהה הקונטיינר שמופיע אצלכם. הפעם ה -a אומר לפקודה שאנחנו רוצים שהפלט של הקונטיינר יופיע אצלנו על המסך.

הצגה חוזרת של רשימת הקונטיינרים מראה שפעולה זאת לא עשתה שום שינוי ולא יצרה קונטיינר חדש. בסך הכל הפעילה מחדש קונטיינר שנסגר:

$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

4cb8496372d3 hello-world "/hello" 9 minutes ago Exited (0) 18 seconds ago reverent\_sanderson

73177b43ff6d hello-world "/hello" 9 minutes ago Exited (0) 9 minutes ago focused\_bose

לעומת זאת אם תפעילו שוב את docker run תיצרו קונטיינר חדש עבור האימג':

$ docker run hello-world

...

$ docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ff88c7e43750 hello-world "/hello" 7 seconds ago Exited (0) 6 seconds ago zealous\_khayyam

4cb8496372d3 hello-world "/hello" 11 minutes ago Exited (0) 2 minutes ago reverent\_sanderson

73177b43ff6d hello-world "/hello" 11 minutes ago Exited (0) 11 minutes ago focused\_bose

לכן אפשר לחשוב על קונטיינר בתור משהו בין מכונה וירטואלית לתהליך - מצד אחד הוא רץ על המחשב שלכם כמו תהליך רגיל בלי תיווך המכונה הוירטואלית, אבל מצד שני הוא מנותק מהמחשב שלכם ויכול להכיל תוכנות ולעבוד על קבצים שיישמרו כולם בתוך הקונטיינר.

העבודה עם דוקר מציעה לנו את היתרונות הבאים:

1. אפשר להחזיק כל מספר של גירסאות או תוכנות על המכונה ולא תהיה התנגשות ביניהן כי כל תשתית רצה בתוך קונטיינר משלה. זה מעולה בארכיטקטורת Micro Service כי כך כל סרביס יכול לעבוד עם התוכנות שהוא צריך.
2. קל להרים עוד מכונה או להוסיף מתכנת חדש לצוות - פשוט תנו להם את הקונטיינרים המתאימים ושם הם כבר ימצאו את כל מה שצריך.
3. קל לנתק את הקשר בין התוכנות שרצות לבין מספר וסוג השרתים שיש לנו, וכך לקבל ניצול טוב יותר של התשתית.

מה אפשר ללמוד מדף האימג' ב Docker Hub?

בסיס הנתונים PostgreSQL הוא סוג של Service. הוא רץ כל הזמן ברקע ומאזין לבקשות ומחזיר תשובות. אנחנו מתקשרים עם פוסטגרס או דרך קובץ תקשורת מיוחד (שנקרא Unix Socket) או דרך פרוטוקול TCP בפורט 5432. בעבודה עם דוקר יהיה לנו יותר נוח לעבוד בחיבור TCP ולדמיין שבסיס הנתונים רץ על מכונה אחרת מהאפליקציה שצריכה אותו.

בעולם של דוקר אנחנו לא "מתקינים" תוכנה אלא "מושכים" את האימג' שלה ויוצרים ממנו קונטיינר.

נמשיך לדף האימג' בדוקר האב ונראה מה אפשר ללמוד ממנו לפני הרצת הקונטיינר הראשון: <https://hub.docker.com/_/postgres>.

הדבר הראשון לשים לב אליו הוא הטקסט Docker Official Images. אלה אימג'ים רשמיים של דוקר ולכן יש יותר סיכוי שהם יהיו בטוחים לשימוש ולא ינסו להשתמש במחשב שלנו כדי לכרות ביטקוינים.

אני ממשיך לרשימת ה Supported tags. כל אימג' מכיל בנוסף לשם שלו גם תגית שמציינת משהו לגבי אופן הבניה הספציפי. לדוגמה ברשימה שלנו של Postgresql אני רואה שהתגית מורכבת ממספר הגירסה ובחלק מהגירסאות מופיע -alpine. אלפין היא הפצת לינוקס קטנטנה ובדוקר נותנים לנו לבחור בין מכונה שמריצה לינוקס מלא (מבוסס Debian) ובתוכו שרת PostgreSQL, לבין מכונה שמריצה הפצת לינוקס מצומצמת ובתוכה שרת PostgreSQL. האימג' הרגיל בטח יהיה יותר גדול אבל כשנתחבר אליו יהיה יותר קל להתמצא כי מותקנים עליו יותר כלים.

החלק החשוב הבא הוא ה Environment Variables. ב Docker משתני סביבה הם הדרך הנפוצה ביותר להעביר מידע לקונטיינר. בדוגמה של בסיס הנתונים נשים לב למשתני הסביבה הבאים:

1. המשתנה POSTGRES\_PASSWORD שקובע את סיסמת המנהל לבסיס הנתונים
2. המשתנה POSTGRES\_USER שקובע את שם המשתמש שמנהל את בסיס הנתונים
3. המשתנה POSTGRES\_DB שקובע את שם בסיס הנתונים שייווצר

כשנתחבר לבסיס הנתונים נצטרך לדעת את שלושת הערכים האלה כדי שנוכל לעבוד איתו. פריטי מידע כמו שמות משתמשים וסיסמאות כמעט אף פעם לא נשמרים בתוך האימג'.

עד כאן אנחנו יודעים מספיק כדי להפעיל בסיס נתונים PostgreSQL דרך דוקר. עכשיו ניגש לעבודה ונפעיל אחד כזה.

יצירת הקונטיינר:

בואו נפעיל מחלון cmd או wsl את הפקודה הבאה:

$ docker run postgres

כדי לנסות לפתוח בסיס נתונים. אבל זה לא יעבוד. תראו את הפלט:

Error: Database is uninitialized and superuser password is not specified.

You must specify POSTGRES\_PASSWORD to a non-empty value for the

superuser. For example, "-e POSTGRES\_PASSWORD=password" on "docker run".

You may also use "POSTGRES\_HOST\_AUTH\_METHOD=trust" to allow all

connections without a password. This is \*not\* recommended.

See PostgreSQL documentation about "trust":

https://www.postgresql.org/docs/current/auth-trust.html

למרות השגיאה הפקודה כן עשתה משהו - היא משכה את האימג' ממאגר האימג'ים המרכזי של דוקר אליי למחשב. אני יכול לראות את כל האימג'ים שיש לי על המחשב עם הפקודה:

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

postgres latest 293e4ed402ba 6 days ago 315MB

שימו לב שבגלל שלא ציינתי tag באופן אוטומטי התגית שנבחרה היא latest. אני יכול לנסות להריץ תגית אחרת עם סימן נקודותיים ואחריו התגית:

$ docker run postgres:12-alpine

Unable to find image 'postgres:12-alpine' locally

12-alpine: Pulling from library/postgres

540db60ca938: Pull complete

a3cb73039552: Pull complete

39855706e49a: Pull complete

3f0f1a7f7d33: Pull complete

2f65e3924365: Pull complete

44817ec3b5bb: Pull complete

310cea271e89: Pull complete

717b5444b286: Pull complete

Digest: sha256:002161f3f4472b37e6014e7d5136a81076ebec5add084a2cd433cf84a416bb11

Status: Downloaded newer image for postgres:12-alpine

Error: Database is uninitialized and superuser password is not specified.

You must specify POSTGRES\_PASSWORD to a non-empty value for the

superuser. For example, "-e POSTGRES\_PASSWORD=password" on "docker run".

You may also use "POSTGRES\_HOST\_AUTH\_METHOD=trust" to allow all

connections without a password. This is \*not\* recommended.

See PostgreSQL documentation about "trust":

https://www.postgresql.org/docs/current/auth-trust.html

וגם זה לא עבד מאותה סיבה, אבל כן שווה כבר לראות את הגודל השונה של שתי האימג'ים:

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

postgres 12-alpine a58cf5527d36 6 days ago 158MB

postgres latest 293e4ed402ba 6 days ago 315MB

עכשיו בואו נתקן את הבעיה ונפעיל את בסיס הנתונים. דוקר התלונן שלא בחרנו סיסמה ולכן הוא לא יכל ליצור את הקונטיינר. אנחנו זוכרים שהסיסמה, כמו גם שם המשתמש ושם בסיס הנתונים, עוברים בתור משתני סביבה. אנחנו יוצרים אותם בשורת הפקודה עם המתג -e.

סך הכל הפקודה הבאה מריצה את בסיס הנתונים עם בחירת שם משתמש, שם בסיס נתונים וסיסמה:

$ docker run -e POSTGRES\_PASSWORD=ninja -e POSTGRES\_USER=shay -e POSTGRES\_DB=helloworld postgres

אני יכול לראות בחלון את הפלט של הפקודה וכל הפרטים על בסיס הנתונים. עכשיו שיש לי בסיס נתונים המשימה הבאה תהיה להתחבר אליו ורצוי מקונטיינר אחר. בשביל זה אני צריך לגלות מה כתובת ה IP של הקונטיינר שיצרתי. זיכרו שלמרות שקונטיינר רץ אצלי על המחשב, הוא מתנהג בתור מכונה נפרדת - עם כתובת IP ופורטים משלו.

בשביל לקבל מידע מפורט על הקונטיינר אני מפעיל מחלון אחר את הפקודה docker inspect ומעביר לה את מזהה הקונטיינר. כלומר תחילה אפעיל docker ps, אסתכל מה ה Container ID ואז אריץ docker inspect עם ערך זה. במקרה שלי:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

2e4d7cd70573 postgres "docker-entrypoint.s…" 59 minutes ago Up 59 minutes 5432/tcp db

$ docker inspect 2e4d7cd70573

דוקר אינספקט מדפיס המון פלט לגבי הקונטיינר ומה שמעניין אותי הוא כתובת ה IP כלומר השדה IPAddress. תיאורטית קונטיינר יכול להחזיק מספר כרטיסי רשת (וירטואליים) וכך יהיו לו מספר כתובות IP. במקרה שלנו תהיה רק אחת כי זו ברירת המחדל.

אצלי כתובת ה IP היא 172.17.0.2. בעזרתה נוכל להפעיל קונטיינר אחר וממנו להתחבר לקונטיינר בסיס הנתונים שלנו. אני משתמש באותו אימג' בשם postgres שכבר הורדנו ובונה ממנו קונטיינר חדש בחלון אחר:

$ docker run --rm -it postgres /bin/bash

שימו לב לפרמטרים:

1. הפעם אני לא מעביר משתני סביבה. אני לא רוצה ליצור בסיס נתונים אלא להתחבר לבסיס נתונים קיים.
2. אני מעביר את הפרמטר --rm כדי למחוק את הקונטיינר אוטומטית מיד אחרי שהוא ייסגר.
3. אני מעביר את הפרמטרים -it כדי לקבל גישה אינטרקטיבית לקונטיינר.
4. אחרי שם האימג' אני מעביר את שם התוכנית /bin/bash. בדרך כלל אימג' פוסטגרס יוצר קונטיינר שמפעיל בסיס נתונים, אבל כשאני מעביר פקודה בסוף שורת ה run אני מבקש מדוקר להפעיל את הפקודה שלי ולא את הפקודה שמוגדרת באימג'.

התוצאה היא כניסה לתוך שורת פקודה לעבודה אינטרקטיבית בקונטיינר. בתוך המסוף שנפתח אני יכול לכתוב hostname כדי להיווכח שאני בתוך קונטיינר ולגלות את ה Container ID, ואחר כך להפעיל:

$ psql -h 172.17.0.2 -U shay mydb

כדי להפעיל לקוח psql אינטרקטיבי שיתחבר לבסיס הנתונים על הקונטיינר השני. הפקודה מבקשת ממני סיסמה ואני מקליד את הסיסמה שבחרתי לבסיס הנתונים ואז פותחת לי גישה אינטרקטיבית ל DB. שימו לב רק להחליף את כתובת ה IP בכתובת ה IP של הקונטיינר שלכם.

בתוך החלון האינטרקטיבי אני יכול להפעיל את הפקודות הבאות כדי ליצור טבלה ולהכניס לתוכה נתונים:

# create table test(x integer, y integer);

# insert into test values(10, 20);

# select \* from test;

בסוף אני יכול לצאת מ psql וזה יוציא אותי גם מהקונטיינר וימחק אותו.

נזכור שבסיס הנתונים עדיין רץ ולכן אם אני מריץ מחדש קונטיינר של לקוח psql אני יכול להתחבר לאותו בסיס נתונים ולראות את הנתונים שה psql הקודם יצר.

סיכום:

ראינו איך להפעיל קונטיינר מתוך אימג' של בסיס נתונים ואיך להתחבר אליו דרך CLI Client מקונטיינר אחר. למדנו לקרוא דף תיעוד של Image ב Dockerhub והבנו איך להעביר פרמטרים לקונטיינר.

אלה הפקודות המרכזיות שראינו בפרק::

# create a container from an image

$ docker run <image>

# list all images on disk

$ docker image ls

# delete a container

$ docker rm <container\_id>

# list all running containers

$ docker ps

# list all containers (active and inactive)

$ docker ps -a

# delete a container

$ docker rm <container\_id>

# Run a postgres container with a database

$ docker run -v pgdata:/var/lib/postgresql/data -e POSTGRES\_PASSWORD=ninja -e POSTGRES\_USER=shay -e POSTGRES\_DB=helloworld postgres

# Create a new volume

$ docker volume create <volume\_name>

# List all volumes

$ docker volume ls

# Create a psql client container and connect to a DB

$ docker run -it --rm postgres psql -h localhost -U shay helloworld

אנחנו כבר רואים שאנחנו יכולים להשתמש בדוקר כדי לשפר את חווית הפיתוח שלנו, להתקין בקלות תוכנות ולקבל מספר גירסאות של תוכנה על המחשב.

עכשיו אתם- חפשו ב Docker Hub אימג' שמעניין אתכם ונסו להפעיל אותו. נסו לגלות כמה שיותר מידע דרך קריאת דף האימג' ואם בחרתם אימג' של סרביס נסו להתחבר אליו ממכונה אחרת.

שמירת המידע של בסיס הנתונים ב Volume

כל כתיבה על מערכת הקבצים של קונטיינר תימחק כשהקונטיינר ייסגר. בשביל לשמור מידע גם להפעלות הבאות של הקונטיינר אנחנו משתמשים ב Volumes.

**איפה שומרים את המידע:**

נחזור לחלון הראשון שמריץ את שרת בסיס הנתונים ונסגור את הקונטיינר עם לחיצה על Ctrl+C. אחרי זה נריץ אותו מחדש וננסה שוב להתחבר לבסיס הנתונים ולשלוף את המידע שהיה עליו. התוצאה עשויה להפתיע:

helloworld=# select \* from test;

ERROR: relation "test" does not exist

כל המידע שיצרתי בקונטיינר בסיס הנתונים נמחק ברגע שסגרתי את הקונטיינר. ובאמת קונטיינר לא נועד כדי לשמור מידע ואנחנו חושבים על קונטיינרים בתור משהו שיכול כל הזמן להימחק ולהיווצר מחדש מתוך האימג'.

כדי להתגבר על מגבלה זו וכן לאפשר לבסיס נתונים לשמור את המידע בצורה קבועה אנחנו נצטרך למפות "תיקיית עבודה" מהמחשב החיצוני לקונטיינר, כך שהקונטיינר יכתוב את המידע שלו לאותה תיקיה והתיקיה תישאר ושתמור את הקבצים גם כשהקונטיינר ייסגר. דוקר מאפשר לנו שני מנגנוני מיפוי, באחד דוקר שומר אצלו את כל הקבצים ובשני הוא פשוט ממפה תיקיה קיימת על הדיסק. בהמשך הקורס יהיה לנו שיעור ספציפי בנושא זה ובינתיים בואו נראה איך להיעזר במנגנון המובנה של דוקר כדי לשמור את המידע גם אחרי שהקונטיינר נסגר.

תחילה אני יוצר Volume שישמור את המידע וישרוד אחרי הקונטיינר עם:

$ docker volume create pgdata

השם pgdata הוא שם "אזור האחסון" ואני יכול לבחור שם כל שם שארצה. אני יכול גם לראות את כל אזורי האחסון שדוקר כבר יצר עם הפקודה:

$ docker volume ls

ולמחוק אחד עם docker volume rm.

לאחר היצירה אני מעביר את ה volume לקונטיינר וממפה אותו לתיקיה ספציפית ש Postgres יודע לכתוב אליה:

$ docker run -v pgdata:/var/lib/postgresql/data -e POSTGRES\_PASSWORD=ninja -e POSTGRES\_USER=shay -e POSTGRES\_DB=helloworld postgres

את התיקיה /var/lib/postgresql/data לקחתי מהתיעוד של האימג'. רק האימג' יודע לאיזה תיקיות הוא כותב ואיזה תיקיות צריכות להישמר בין הפעלות.

נסו עכשיו שוב את אותו המשחק:

1. כניסה מחלון אחר.
2. יצירת טבלה והוספת ערכים.
3. סגירת הקונטיינר עם בסיס הנתונים והפעלה מחדש.
4. כניסה נוספת מהקונטיינר השני כדי לראות שהטבלה והנתונים נשמרו.

## הפעלת nginx בתוך קונטיינר

## שרת Nginx הוא שרת ווב שיודע לקבל בקשות HTTP ולשלוח בתגובה קבצים או להעביר את הבקשות לסרביסים שונים ולהחזיר את התשובות מהם. בשיעור זה נריץ אותו בתוך קונטיינר כדי לראות איך אנחנו מתחברים מדפדפן לקונטיינר

דף האימג' של nginx נמצא בקישור: <https://hub.docker.com/_/nginx>.

הדבר הראשון שאנחנו רואים הוא הכיתוב Docker Official Images שאומר שמדובר באימג' רשמי ולכן יש פחות סיכוי לבעיות אבטחה או בעיות אחרות איתו. ברשימת התגיות אנחנו רואים שהגירסה המרכזית להתקנה נקראת mainline שזו באמת הגירסה היציבה של nginx, ושהאימג' זמין בגירסת דביאן מלאה או בגירסת alpine מצומצמת.

בחלק How to use this image אני יכול למצוא כמה דוגמאות להפעלה של האימג' כדי להגיש תוכן סטטי כלומר קבצי HTML, CSS ו JavaScript. הדוגמה הראשונה המעניינת היא:

$ docker run --name some-nginx -v /some/content:/usr/share/nginx/html:ro -d nginx

והיא מראה לנו ש nginx מגיש את הקבצים מתיקיית /var/share/nginx/html של הקונטיינר. בשביל להגיש קבצים מהמחשב המקומי אני צריך למפות תיקיה שיש בה את הקבצים כדי שתהיה זמינה בתוך הקונטיינר. הסיומת :ro אומרת שהקונטיינר לא יוכל לשנות קבצים אלה, והמתג -d גורם לקונטיינר לא להשתלט על המסוף ומאפשר לי להמשיך להריץ פקודות נוספות. בהמשך השיעור נכתוב בעצמנו דוגמה דוגמה כדי להגיש קובץ HTML שנכתוב לדפדפן.

## מיפוי פורטים:

דוגמא מעניינת נוספת ממחישה את מיפוי הפורטים:

$ docker run -d -p 8080:80 nginx

בתוך הקונטיינר שרת הווב nginx יקשיב על פורט 80. זה נתון, כך עובד האימג'. אם יש לי את כתובת ה IP של הקונטיינר שמריץ nginx, אני אוכל להתחבר לפורט 80 שלו כדי לקבל את עמוד ה HTML.

אפשרות נוספת להתחבר לקונטיינר היא לפתוח צינור בין המחשב שלי לפורט 80 על הקונטיינר. על מכונת הפיתוח לא נהוג להריץ תוכניות שמקשיבות לפורטים נמוכים ולכן אני בוחר את הפורט 8080. הפקודה בעצם "מחברת" בין פורט 8080 על מכונת הפיתוח שלי לפורט 80 בתוך הקונטיינר, כך שאם דרך הדפדפן אגלוש למכונה localhost בפורט 8080, הבקשה תעבור אוטומטית לקונטיינר לפורט 80 והתשובה מהקונטיינר תישלח חזרה לדפדפן. מנגנון מיפוי פורטים עוזר לנו במצב פיתוח לבדוק את הקוד שלנו שרץ בקונטיינר, כי הרבה יותר נוח לעבוד על המכונה המקומית מאשר לבדוק כל פעם מה כתובת ה IP של הקונטיינר.

מיפוי Volumes:

בחלק Complex Configuration אני רואה שאם יש לי קובץ הגדרות nginx.conf שאני רוצה להעביר לקונטיינר אני יכול לעשות את זה באמצעות מיפוי Volumes.

ראינו שאנחנו יכולים ליצור Volume כדי לשמור בו מידע שיישמר גם אחרי שהקונטיינר ייסגר. זה הזמן לגלות שאפשר להשתמש בכל תיקיה על המחשב שלכם בתור Volume, ובאמצעות תיקיות על המחשב אני יכול להכין מראש מידע שהקונטיינר יוכל לראות. בדוגמה של nginx, אני יכול למפות קובץ קונפיגורציה מסוים בתור Volume כך שהקונטיינר יראה את הקובץ הזה בתוך מערכת הקבצים שלו. זה בדיוק התפקיד של השורה:

$ docker run --name my-custom-nginx-container -v /host/path/nginx.conf:/etc/nginx/nginx.conf:ro -d nginx

השורה מחברת את הקובץ שנמצא על המחשב המקומי שלי בנתיב /host/path/nginx.conf לתוך הקונטיינר ושם נותנת לו את השם /etc/nginx/nginx.conf. בתוך הקונטיינר זה שם שמור ש nginx יודע לחפש שם את קובץ ההגדרות שלו. הסיומת :ro אומרת שהקונטיינר לא יוכל לשנות את הקובץ הזה.

דוגמה נוספת לשימוש במיפוי תהיה למפות תיקיה עם קבצי HTML למקום שהקונטיינר מכיר ואז nginx שבתוך הקונטיינר יוכל לשלוח ממנה את הקבצים. תחילה אכין קובץ index.html שאותו ארצה להגיש עם nginx ובו אכתוב את התוכן הבא:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

</head>

<body>

<h1>Hello From Docker</h1>

</body>

</html>

אני שומר את הקובץ בתיקיית html בתוך תיקיית הפרויקט שלי. אצלי הנתיב המלא לקובץ הוא /home/shay/work/courses/docker-microservices/demos/06-nginx-container/after/html, אבל אצלכם הוא כמובן יכול להיות אחר. עכשיו אני כותב:

$ docker run -v /home/shay/work/courses/docker-microservices/demos/06-nginx-container/after/html:/usr/share/nginx/html:ro -p 8080:80 nginx

כדי להפעיל קונטיינר nginx שיגיש את הקובץ מתוך תיקיית html. טריק פשוט לקצר את השורה הוא להשתמש במשתנה הסביבה PWD שמכיל את הנתיב המלא לתיקיה הנוכחית. אם אני עכשיו בתוך תיקיית html שמכילה את הקובץ index.html שיצרתי אני יכול לכתוב:

$ docker run -v $PWD:/usr/share/nginx/html:ro -p 8080:80 nginx

אחרי הפעלת הקונטיינר אני יכול להיכנס לדפדפן ולגלוש ל localhost:8080 כדי לקבל את דף הפתיחה של nginx.

הפעלת פקודות בתוך הקונטיינר:

הקונטיינר מתנהג כמו מכונת לינוקס לכל דבר, ולמרות שהוא מריץ פקודה מסוימת (התוכנית nginx), אפשר להריץ עליו פקודות נוספות ואפילו לתקשר איתן. הפקודה הבאה מאפשרת להריץ shell בתוך הקונטיינר כדי לקבל ממשק עבודה אינטרקטיבי שם:

$ docker exec -it 503919e7eacc /bin/bash

אתם תצטרכו להחליף את 503919e7eacc במזהה הקונטיינר שלכם. בתוך הקונטיינר אפשר להציץ על קובץ הקונפיגורציה של nginx:

root@503919e7eacc:/# cat /etc/nginx/nginx.conf

user nginx;

worker\_processes 1;

error\_log /var/log/nginx/error.log warn;

pid /var/run/nginx.pid;

events {

worker\_connections 1024;

}

http {

include /etc/nginx/mime.types;

default\_type application/octet-stream;

log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

'$status $body\_bytes\_sent "$http\_referer" '

'"$http\_user\_agent" "$http\_x\_forwarded\_for"';

access\_log /var/log/nginx/access.log main;

sendfile on;

#tcp\_nopush on;

keepalive\_timeout 65;

#gzip on;

include /etc/nginx/conf.d/\*.conf;

זיכרו ששינויים בקונטיינר לא נשמרים כשהקונטיינר נסגר ולכן אל תנסו "לתקן" בעיות באמצעות ביצוע פעולות בתוך קונטיינרים. אנחנו נכנסים לקונטיינר כדי ללמוד על המצב ולהבין איך הוא נוצר. את הבעיות (אם יהיו) נתקן באמצעות יצירה טובה יותר של קונטיינרים או תיקון של האימג'.

הפקודה exit יוצאת מהמסוף של הקונטיינר ועכשיו אני יכול לסגור ולמחוק אותו עם:

$ docker rm -f 503919e7eacc

תקציר הפקודות שראינו:

# Create a new container and run it on the background

$ docker run -d nginx

# List all active containers

$ docker ps

# Stop and delete a container

$ docker rm -f <container\_id>

# Execute a shell inside a container

$ docker exec -it <container\_id> /bin/bash

# Or if bash is not installed (for example in alpine)

$ docker exec -it <container\_id> /bin/sh

# See the logs from a container

$ docker logs <container\_id>

$ docker run -v <local\_path>:<path\_inside\_container> ...

## תרגול שימוש ב Docker:

## קונטיינר Hello World:

מתוך דוקר על המחשב שלכם או מתוך Play With Docker הריצו קונטיינר מתוך האימג': <https://hub.docker.com/_/hello-world>

הציגו את כל הקונטיינרים שרצים על המכונה עם docker ps. האם הקונטיינר שיצרתם מופיע ברשימה? למה הוא לא מופיע? איך תציגו אותו בכל זאת?

הציגו את כל ה image-ים שמותקנים אצלכם על המחשב. האם האימג' של hello-world מופיע שם? כמה מקום הוא תופס לכם על הדיסק?

מצאו את ה Container ID של הקונטיינר החדש שיצרתם. אצלי ה Container ID הוא 0671ce82b412. הפעילו את הפקודות הבאות תוך החלפת ה Container ID בזה שלכם:

$ docker start -a 0671ce82b412

$ docker run hello-world

שתי הפקודות הדפיסו את אותו פלט על המסך. השתמשו בפקודות שלמדתם ונסו להבין מה ההבדל בין הפקודות?

הפעלת תוכנית בכל שפת תכנות:

אחד השימושים של דוקר הוא להפעלת כלים שלא מותקנים לנו על המחשב בלי שנצטרך להתקין אותם ואחר כך להסתבך עם התנגשות עם כלים אחרים שאולי מותקנים לנו על המחשב. בואו ניקח לדוגמה את שפת התכנות אליקסיר, שאני מניח שלא מותקנת אצלכם על המכונה.

אצלכם על המכונה או במכונת Play With Docker צרו קובץ בשם helloworld.exs עם התוכן הבא:

defmodule Main do

def twice(x) do

x \* 2

end

def main do

IO.puts "Hello World. 2 \* 2 = #{twice(2)}"

end

end

Main.main

והפעילו את הפקודה הבאה, מתוך אותה תיקיה בה נמצא הקובץ, כדי להפעיל אותו בשפת התכנות אליקסיר:

$ docker run -it --rm -v "$PWD":/usr/src/myapp -w /usr/src/myapp elixir elixir helloworld.exs

אם הכל הלך כמו שצריך תקבלו על המסך את ההדפסה:

Hello World. 2 \* 2 = 4

הסבירו מה אומרים המתגים -it, --rm, -v ו -w ? אם אתם לא זוכרים או לא בטוחים בדקו בתיעוד על docker run בקישור: <https://docs.docker.com/engine/reference/run/>.

עכשיו מצאו שפת תכנות שאתם מכירים ולא מותקנת לכם על המחשב, כיתבו תוכנית קצרה בשפה זו והריצו אותה באמצעות Docker Container.

הפעלת קונטיינר ברקע ותקשורת איתו:

הפקודה הבאה מפעילה קונטיינר מתוך האימג' ubuntu:

$ docker run -d ubuntu sleep infinity

הפעילו את הפקודה ובדקו שהקונטיינר נדלק ופעיל, ומצאו את ה Container ID שלו.

בתוך מכונת לינוקס הפקודה hostname -I מדפיסה את כתובת ה IP של המכונה. הפעילו אותה בתוך הקונטיינר כדי לגלות את הכתובת.

סגרו את הקונטיינר ומחקו אותו.

## שילוב מספר קונטיינרים עם

## Docker Compose

## אחרי שראינו איך להפעיל קונטיינר אנחנו ממשיכים לצעד הבא בפיתוח והוא הפעלת מספר קונטיינרים וחיבור ביניהם - כלומר יצירת מיקרו סרביס. בשיעור זה ניצור את הסרביס הראשון שלנו שמורכב מקונטיינר Node.JS וקונטיינר redis, ונראה איך להריץ את שניהם יחד עם כלי בשם docker-compose.

## קונטיינרים זה לא מספיק:

אנחנו יודעים איך להריץ קונטיינרים, אבל קונטיינרים לבד זה לא מספיק:

1. רוב הסרביסים שנכתוב יצטרכו יותר מקונטיינר אחד: יהיה לנו קונטיינר אחד לבסיס הנתונים וקונטיינר אחר לאפליקציה, והרבה פעמים יהיו קונטיינרים נוספים עבור משימות שרצות ברקע או תהליכי עזר אחרים.
2. קונטיינרים שונים באותו סרביס צריכים דרך קלה לתקשר ביניהם. הם לא צריכים לבנות על זה שכולם רצים על אותו מחשב או לדעת כתובות IP אחד של השני.
3. נוח לנו לנהל מספר קונטיינרים במקום מרכזי שם אנחנו קובעים את הערכים למשתני הסביבה, ל Volumes ושאר דברים שקונטיינרים צריכים.

דוקר קומפוז הוא בדיוק הכלי המרכזי שיפתור לנו את כל בעיות התיאום:

1. בקובץ בשם docker-compose.yml אנחנו כותבים את כל הטופולוגיה של הקונטיינרים: איזה קונטיינרים צריך, מה האימג' לכל קונטיינר, מהם משתני הסביבה שמתאימים לו ומה ה Volumes שהוא יחובר אליהם.
2. דוקר קומפוז יודע לבנות רשת תקשורת פנימית לכל הקונטיינרים באותו סרביס, כלומר כל אלה שמוגדרים בתוך אותו docker-compose.yml. הוא גם יודע להשתמש ברשתות חיצוניות אם צריך.
3. דוקר קומפוז ייתן לנו אפשרות בפקודה אחת להפעיל או לכבות את כל הקונטיינרים.

דוקר קומפוז יהיה כלי העבודה המרכזי שלנו בקורס זה בעבודה עם קונטיינרים וביצירת Micro Services במצב פיתוח. במעבר לפרודקשן אנחנו נראה איך להעביר את העבודה שלנו מ docker compose לכלי שנקרא Kubernetes שמתאים יותר להרצת קונטיינרים בסביבת פרודקשן.

בניית הסרביס הראשון שלנו עם docker compose:

דוקר קומפוז משתמש בקובץ הגדרות שנקרא docker-compose.yml (וכן אפשר לבחור גם כל שם אחר, אבל docker-compose.yml הוא ברירת המחדל). הקובץ כולל בלוק עבור כל קונטיינר שנצטרך לנהל - ושם נמצאים כל הפרמטרים שהיינו מעבירים לפקודת docker run, כלומר הגדרות ה volume, מיפוי פורטים, שם האימג' והגדרות רבות נוספות.

בשביל הדוגמה הראשונה אני רוצה לבנות תוכנית Node.JS שמנהלת מונה. כל פעם שנשלח לה בקשת POST היא תגדיל את ערך המונה ב-1 וכל פעם שנשלח בקשת GET היא תחזיר את ערך המונה הנוכחי. קוד ראשוני של התוכנית יכול להיראות כך:

const express = require('express')

const app = express()

const port = 3000

let counter = 0;

app.post('/', (req, res) => {

counter += 1;

res.send({ counter })

})

app.get('/', (req, res) => {

res.send({ counter })

})

app.listen(port, () => {

console.log(`Example app listening at http://localhost:${port}`)

})

הקוד תלוי ב express ולכן אני צריך לכתוב קובץ package.json עם התלות. מאחר ויש לי Node.JS מותקן כבר על המחשב אני יכול להשתמש בו כדי ליצור את הקובץ ולהתקין את התלות:

$ npm init -y

$ npm install --save express

השלב הבא הוא להריץ את הקוד שכתבתי כדי לראות אם זה עובד. כאן יש מספר גישות:

1. מאחר ומותקן לי כבר Node.JS על המכונה, אני יכול להשתמש בו ולהריץ את הקוד לוקאלית אצלי. הקוד לא ניגש עדיין לבסיס נתונים או לקונטיינרים אחרים ולכן יעבוד, אבל לאורך זמן כשארצה להוסיף לו בסיס נתונים זה כבר יהיה מסורבל (כי אני לא רוצה להתקין אצלי על המחשב גם בסיס נתונים ואת כל שאר התוכנות שהוא צריך).
2. מאחר ויש לי דוקר על המכונה אני יכול לכתוב משורת הפקודה את פקודת ה docker run עם כל הפרמטרים המתאימים (ואולי זה תרגיל טוב אז שווה לכם עכשיו לנסות) כדי להריץ את התוכנית. בהמשך כשאצטרך להוסיף בסיס נתונים אני אצטרך בחלון חדש להפעיל עוד docker run ולאורך זמן אני אצטרך לזכור את כל הפרמטרים שכל קונטיינר צריך. גם פיתרון זה לא מתאים ככל שהתוכנית גדלה.
3. אני יכול להשתמש ב docker compose כדי לבנות קונטיינר שיריץ את התוכנית שלי, ואחר כך כשהתוכנית תצטרך גם קונטיינרים נוספים אוסיף אותם לקובץ ההגדרות של docker compose וכך הוא יריץ גם אותם.

כמובן שאנחנו נבחר באפשרות השלישית. נכתוב קובץ בשם docker-compose.yml עם התוכן הבא:

version: "3.9"

services:

counter:

image: node:16

volumes:

- ".:/app"

command: bash -c "npm install -g nodemon && npm install && nodemon counter.js"

working\_dir: /app

ports:

- "3000:3000"

הקובץ בפורמט שנקרא [yaml](https://he.wikipedia.org/wiki/YAML). פורמט זה קצת דומה ל JSON או XML מבחינה זו שהוא מגדיר "אוביקטים", אבל הוא הרבה יותר קריא ולכן אנשים אוהבים להשתמש בו בבניית קבצי קונפיגורציה. כל שורה בקובץ מגדירה שדה באוביקט או תא במערך, האינדנטציה בקובץ קובעת מה יהיה בתוך מה. שורות שמתחילות ב - מגדירות תאים במערך והאחרות מפתחות באוביקט. בקובץ ה yaml שהדבקתי כאן אנחנו רואים שבאוביקט הראשי יש שני מפתחות, אחד נקרא version והשני נקרא services. במפתח version הערך הוא מחרוזת, ובמפתח services הערך הוא אוביקט פנימי. בתוך האוביקט הפנימי של services יש מפתח בשם counter, ובתוכו אוסף גדול של מפתחות:

1. המפתח image קובע איזה אימג' יריץ את ה counter. אני בחרתי את node בגירסה 16, והאימג' נלקח מ Dockerhub.
2. המפתח volumes קובע את מיפוי התיקיות. אני רוצה למפות את התיקיה הנוכחית מהמחשב שלי (זאת עם הקוד) לתיקיית /app בתוך הקונטיינר. אנחנו רואים שהערך של volumes הוא מערך ויש בו רק תא אחד. באופן כללי אפשר להגדיר כמה מיפויים שרוצים פשוט מוסיפים עוד שורות שמתחילות בסימן -.
3. המפתח command קובע איזו פקודה להריץ כשהקונטיינר עולה. ברירת המחדל היא מה שמוגדר באימג' אבל אנחנו נדרוס את זה כדי להפעיל את התוכנית שלנו.
4. בנוגע לתוכן הפקודה עצמו - הכתיב bash -c ואז מרכאות מאפשר לי להפעיל מספר פקודות אחת אחרי השניה. אני משתמש ב npm install -g כדי להתקין בתוך הקונטיינר את הכלי nodemon (שזה כלי שמריץ תוכנית node וכל פעם שהקוד משתנה טוען אותה מחדש), אחרי זה אני מפעיל npm install, כי אני לא רוצה לסמוך על זה שתיקיית node\_modules קיימת או מכילה את הדברים הנכונים ובסוף מריץ את nodemon עם התוכנית שלי. בצורה כזאת הקונטיינר יכול להישאר באוויר וכל פעם שאני משנה קוד הקוד המעודכן ייטען בצורה אוטומטית.
5. המפתח working\_dir קובע מהי תיקיית העבודה הנוכחית בתוך הקונטיינר, כלומר מה התיקיה שבתוכה נפעיל את פקודת ה command. אני בוחר את התיקיה אליה מיפיתי את הקוד.
6. המפתח ports קובע מיפוי פורטים. גם הוא מקבל מערך ואנחנו ממפים את פורט 3000 ממכונת הפיתוח לפורט 3000 בקונטיינר, כמו שעשינו עם -p בהפעלת docker משורת הפקודה.

אחרי שכל הקוד במקום נוכל להריץ את הקונטיינר מתוך שורת הפקודה עם הפקודה:

$ docker compose up

או אם אנחנו רוצים שלא ישתלט על החלון נוכל להוסיף את האופציה d:

$ docker compose up -d

אחרי זה בשביל לסגור את הקונטיינר אפשר ללחוץ Ctrl+C מהחלון שמריץ את docker-compose, או אם הפעלתם אותו ברקע אז לכתוב באותה תיקיה:

$ docker compose stop

ובשביל למחוק את הקונטיינרים שנוצרו אנחנו כותבים:

$ docker compose down

התוצאה היא הרצת קונטיינר עם הקוד שלי שמקשיב על פורט 3000. אני יכול להתחבר מדפדפן כדי לראות את ה JSON, או לשלוח הודעת POST עם curl כדי להעלות את ערך המונה:

$ curl -X POST localhost:3000

## הוספת שרת רדיס לדוגמת הקומפוז:

## הקוד שכתבנו משתמש במשתנה בזיכרון כדי לאחסן את המונה. זה נחמד למשחקים אבל ברור שלא יעיל כי ברגע שאני אסגור את הקונטיינר או אפילו אטען מחדש את השרת הכל יימחק. בגלל שאנחנו בדוקר מאוד קל לי להוסיף בסיס נתונים מסודר שישמור את הערך - ואני אשתמש בבסיס נתונים שנקרא redis.

## איך להוסיף רדיס ל docker compose

נעדכן את הקובץ docker-compose.yml כך שיכיל את התוכן הבא:

version: "3.9"

services:

counter:

image: node:16

working\_dir: /app

command: bash -c "npm install -g nodemon && npm install && nodemon counter.js"

volumes:

- ".:/app"

ports:

- "3000:3000"

redis:

image: redis

השינוי היחיד הוא להוסיף service חדש בשם redis שמשתמש באימג' redis.

השם שאני בוחר לסרביס הוא בעצמו בעל משמעות. בתוך הרשת הפנימית ש docker compose יצר, הקונטיינרים יכולים לפנות אחד לשני דרך שם זה. אם counter ינסה עכשיו להתחבר למכונה בשם redis הוא יגיע בדיוק לשרת ה redis שיצרתי. הרבה פעמים, כמו שנראה בהמשך הקורס, לא נרצה להשתמש בשם הזה redis בתוך קוד האפליקציה כדי שיהיה יותר קל להעביר את האפליקציה ממקום למקום - ואז נעביר את שם השרת בתור משתנה סביבה. לדוגמה בשיעור זה השם redis מספיק טוב.

נלך לעדכן את Counter כדי שיתחבר לשרת ה redis שיצרנו. תחילה אני מתקין את החבילה redis עם הפקודה:

$ npm install --save redis

ואחר כך משתמש בה בקוד באופן הבא:

const express = require('express')

const app = express()

const port = 3000

let

redis = require('redis'),

/\* Values are hard-coded for this example, it's usually best to bring these in via file or environment variable for production \*/

client = redis.createClient({

port : 6379,

host : 'redis', // replace with your hostanme or IP address

});

app.get('/', (req, res) => {

client.get('count', (err, count) => {

if (err) return next(err);

res.send({ count });

});

});

app.post('/', (req, res, next) => {

client.incr('count', (err, count) => {

if (err) return next(err);

res.send({ count });

});

});

app.listen(port, () => {

console.log(`Example app listening at http://localhost:${port}`)

})

רדיס הוא בסיס נתונים ששומר ערכים מסודרים לפי מפתחות. הפקודה client.get מחזירה ערך ממפתח מסוים והפקודה client.incr מעלה את הערך במפתח מסוים ומחזירה את הערך החדש (אגב הפקודה client.set משנה ערך של מפתח, אבל לא היינו צריכים אותה בתוכנית כאן).

העדכון של הקוד מבצע:

1. מתחבר לשרת redis בשם "redis". בדרך כלל נרצה לקחת את השם הזה ממשתנה סביבה, אבל פה אנחנו רק בתחילת הדרך אז לא נורא להשתמש בשם Hard Coded.
2. בתוך קוד הנתיבים אני משתמש ב Redis Client שיצרתי כדי לקבל או לעדכן את הערך עבור המפתח count.

באמצעות אותו docker compose up נוכל להרים את שני הקונטיינרים ולראות איך counter מתחבר לשרת ה redis. נפעיל curl כדי לראות את ערך המונה ולשנות אותו.

דוקר קומפוז מאפשר לנו ליצור קונטיינרים בכל מבנה שנצטרך וגם מאפשר העברה של פרמטרים לאותם קונטיינרים בדיוק כמו שהעברנו בהפעלה משורת הפקודה. בשיעור זה נראה את הפרמטרים המרכזיים אותם נרצה להעביר בדוקר קומפוז.

מיפוי פורטים:

בהפעלת דוקר קונטיינר הקונטיינר רץ בתוך סביבה מבודדת. כשתוכנית בתוך הקונטיינר מתחילה להאזין לחיבורי רשת היא בוחרת על איזה פורט להקשיב בתוך אותה סביבה מבודדת, ודוקר ממפה את הפורט הזה לפורט על המכונה האמיתית שמריצה את התוכנית. אנחנו בוחרים את מיפויי הפורטים בהפעלת דוקר משורת הפקודה עם -p ובתוך קובץ docker-compose.yml אפשר להוסיף לבלוק של סרביס בלוק בשם ports שקובע את המיפוי.

לדוגמה הקובץ docker-compose.yml הבא מפעיל שרת nginx שמקשיב בתוך הקונטיינר לפורט 80, אבל על המחשב המקומי הוא מחובר לפורט 8080. בשביל להיכנס לשרת מהדפדפן אני מקליד את הכתובת: http://localhost:8080, כמובן אחרי שאפעיל את הקונטיינר עם docker compose up:

version: "3.9"

services:

web:

image: "nginx"

ports:

- "8080:80"

מיפוי Volumes:

בהפעלת דוקר קונטיינר משורת הפקודה ראינו שמיפינו Volumes באמצעות המתג -v. בדיוק באותו אופן אפשר למפות volumes דרך הגדרה בקובץ docker-compose.yml שלנו.

כשאנחנו מגדירים volume דוקר קומפוז יכול ליצור אותו באופן אוטומטי או להשתמש ב volume קיים או בתיקיה קיימת.

קובץ ה docker-compose.yml הבא יפעיל את הפקודה cat /etc/shells אבל קודם ימפה קובץ מהמחשב שלי לתוך /etc/shells כך שהפקודה בעצם תציג את תוכן הקובץ מהמחשב שלי:

version: "3.9"

services:

web:

image: "ubuntu"

command: ["cat", "/etc/shells"]

volumes:

- "/home/shay/a.txt:/etc/shells"

בדרך כלל נשתמש במנגנון זה כדי ליצור Volume לבסיס נתונים כך שהמידע יישמר גם אחרי שהקונטיינר יימחק. לדוגמה קובץ ה docker-compose.yml הבא פותח שרת redis שישמור את הידע שלו גם אחרי שהקונטיינר ייסגר:

version: "3.9"

services:

redis:

image: "redis:alpine"

volumes:

- data:/data

volumes:

data:

בגלל שה Volume מוגדר במקום מרכזי אני יכול להשתמש באותו Volume לשני קונטיינרים שונים. לדוגמה קובץ ה docker-compose.yml הבא יפעיל שרת nginx וכל שניה ישנה את קובץ ה HTML שאותו שרת מציג באמצעות עדכון הקובץ מקונטיינר אחר:

version: "3.9"

services:

writer:

image: "ubuntu"

command: ["bash", "-c", "while true; do echo hello $$(date +%s) ; sleep 1; done > /html/index.html"]

volumes:

- "data:/html:rw"

server:

image: "nginx"

ports:

- "8080:80"

volumes:

- "data:/usr/share/nginx/html:ro"

volumes:

data:

שימו לב שבנוסף להגדרת ה Volumes ראינו כאן שאפשר להשתמש בפקודה command כדי לקבוע שהקונטיינר יריץ פקודה אחרת מתוך האימג'.

הגדרת משתני סביבה:

הגדרה אחרונה שנראה לשיעור זה מדוקר קומפוז היא הגדרת משתני סביבה לקונטיינר. המפתח נקרא environment והוא עובד בדיוק כמו מתג -e בהרצת קונטיינר משורת הפקודה. הדוגמה הבאה מתוך התיעוד של אימג' wordpress ב dockerhub יוצרת מכונת וורדפרס ובסיס נתונים, ומעבירה למכונת הוורדפרס את פרטי הגישה לבסיס הנתונים באמצעות משתני סביבה:

version: '3.9'

services:

wordpress:

image: wordpress

restart: always

ports:

- 8080:80

environment:

WORDPRESS\_DB\_HOST: db

WORDPRESS\_DB\_USER: exampleuser

WORDPRESS\_DB\_PASSWORD: examplepass

WORDPRESS\_DB\_NAME: exampledb

volumes:

- wordpress:/var/www/html

db:

image: mysql:5.7

restart: always

environment:

MYSQL\_DATABASE: exampledb

MYSQL\_USER: exampleuser

MYSQL\_PASSWORD: examplepass

MYSQL\_RANDOM\_ROOT\_PASSWORD: '1'

volumes:

- db:/var/lib/mysql

volumes:

wordpress:

db:

שיטת העברת פרטי התחברות באמצעות משתני סביבה לא מומלצת מטעמי אבטחה. בהמשך הקורס נדבר על ניהול Secrets מסודר ב docker-compose וזו תהיה השיטה שנרצה להשתמש בה בעולם האמיתי.

שווה לשים לב למפתח החדש restart שמופיע בדוגמה. הפקודה restart: always אומרת שכל פעם שהקונטיינר ייסגר מאיזושהי סיבה באופן אוטומטי קומפוז ירים אותו מחדש.

סיכום:

בשיעור זה ראינו מספר דוגמאות לכתיבת docker-compose.yml. ראינו איך למפות volumes, ports ומשתני סביבה ואיך לשנות את פקודת ברירת המחדל שהקונטיינר מריץ. הדוגמה האחרונה מדף התיעוד של וורדפרס מסכמת את כל היכולות שראינו ואני מדביק אותה מחדש כאן:

version: '3.9'

services:

wordpress:

image: wordpress

restart: always

ports:

- 8080:80

environment:

WORDPRESS\_DB\_HOST: db

WORDPRESS\_DB\_USER: exampleuser

WORDPRESS\_DB\_PASSWORD: examplepass

WORDPRESS\_DB\_NAME: exampledb

volumes:

- wordpress:/var/www/html

db:

image: mysql:5.7

restart: always

environment:

MYSQL\_DATABASE: exampledb

MYSQL\_USER: exampleuser

MYSQL\_PASSWORD: examplepass

MYSQL\_RANDOM\_ROOT\_PASSWORD: '1'

volumes:

- db:/var/lib/mysql

volumes:

wordpress:

db:

למידע נוסף על כל הפקודות שתוכלו לרשום בקובץ docker-compose.yml אפשר להסתכל בתיעוד בקישור: <https://docs.docker.com/compose/compose-file/compose-file-v3/#volume-configuration-reference>

התמודדות עם אי זמינות של סרביסים:

אחד האתגרים במערכות מבוזרות הוא שסרביסים מסוימים במערכת יכולים להיות לא זמינים, לפעמים לכמה רגעים ולפעמים אפילו לכמה שעות. זה יכול לקרות בגלל עומס, מתקפת האקרים על סרביס מסוים, באג בקוד או אפילו בעיית חומרה. ככל שבמערכת שלנו יש יותר סרביסים ותקשורת ביניהם כך ניתקל לעתים יותר קרובות בבעיות של אי זמינות חלקים במערכת.

אין דרך אחת שתתאים לכל התרחישים להתמודד עם אי זמינות סרביס מסוים. לדוגמה אם הסרביס שמוסיף משתמש שרכש מוצר לרשימת דיוור לא עובד לא יקרה כלום אם המערכת תתעלם מהבעיה וכשהסרביס יעלה חזרה הוא יוסיף לרשימת הדיוור את כל האנשים שנכנסו בזמן שהוא לא היה זמין. אבל, אם הסרביס שקולט את התשלום שנכנס לא עובד אני בבעיה כי אין דרך אוטומטית לשחזר את כל הקניות שנעשו כשלא הייתי זמין ומהר מאוד יגיעו לקוחות כועסים שלא יקבלו את המוצרים ששילמו עליהם.

חלק מהעבודה בכתיבת קוד למערכת מבוזרת הוא המחשבה על כל אי זמינות והבחירה המודעת מה עושים כשרכיב מסוים שאני מתבסס עליו כרגע לא זמין. האם לנסות שוב עוד כמה דקות? האם להפיל גם את עצמי? האם להציג הודעת שגיאה למשתמשים? בפיתוח מערכת מבוזרת אנחנו חושבים על כל מקרה בפני עצמו ובונים את הקוד בהתאמה.

בגלל שמאוד קל להרים קונטיינר ובגלל שדוקר קומפוז (או כל כלי ניהול קונטיינרים אחר שנבחר) יודע לזהות שקונטיינר נפל ולהרים אותו מחדש, לפעמים במקרה של בעיה בקוד או אי זמינות סרביס מסוים נבחר לרסק את המערכת ולתת לקומפוז להפעיל מחדש את הקונטיינר, בתקווה שעד הפעם הבאה דברים יסתדרו.

לדוגמה סרביס A מספק API לגישה למידע מסוים ותלוי בבסיס הנתונים שלו. סרביס B פונה לקבלת המידע ואז A מנסה להוציא שליפה ומגלה שבסיס הנתונים כרגע לא זמין. אז הקוד בסרביס A יכול להחליט לזרוק Exception ולרסק את הקונטיינר, דוקר קומפוז יעלה אותו מחדש ואולי אחרי העליה מחדש ה DB כבר חזר לעבוד. בזמן הזה סרביס B קיבל כישלון בתקשורת כי סרביס A התרסק ולכן הוא יצטרך להחליט מה לעשות - אם לפנות מחדש, אם לחכות קצת, אם להציג למשתמש הודעת שגיאה או מה בדיוק.

מקרה מיוחד מסוג זה קורה בעליה של המערכת. כשהסרביס שלי מורכב מאפליקציה ובסיס נתונים קשה לדעת מי יצליח לעלות יותר מהר. אם האפליקציה עולה לפני שבסיס הנתונים מוכן היא לא תוכל לעבוד, ואז תסגור את עצמה ודוקר קומפוז יצטרך להפעיל אותה שוב ושוב, עד שבסיס הנתונים יהיה מוכן. הפעלות חוזרות מסוג זה, גם בעליית המערכת וגם בהתמודדות עם נפילות, יכולות ליצור עומס ולגרום לתקלות במקומות אחרים.

הפעלת אפליקציה שתלויה בסדר עליה בלי טיפול:

בשביל הדוגמה אני בונה אפליקציה שמתחברת לבסיס נתונים MongoDB. למה דווקא מונגו? כי הדרייבר של מונגו ממש גרוע בלחכות לבסיס הנתונים שיעלה, וכך יהיה לנו קל לראות מה קורה כשלא מטפלים בבעיות בסדר העליה. תוכנות אחרות (לדוגמה רדיס) לפעמים יהיו יותר חכמות והלוגיקה של טיפול בסדר עליה שונה כבר כתובה בתוך הדרייבר.

שרת ה Node.JS שלי ינסה פשוט להתחבר למונגו, ואחרי שהוא מצליח להתחבר הוא יקשיב ללקוחות שמתחברים לשרת ולכל אחד יענה את המחרוזת Connected To Mongo. זה כל הקוד:

const express = require('express')

const app = express()

const port = 3000

const { MongoClient } = require("mongodb");

const { MONGO\_USER, MONGO\_PASSWORD, MONGO\_HOST } = process.env;

const uri = `mongodb://${MONGO\_USER}:${MONGO\_PASSWORD}@${MONGO\_HOST}:27017/`;

const client = new MongoClient(uri);

(async () => {

await client.connect();

console.log(`connected`);

app.get('/', async (req, res) => {

res.send("Connected To Mongo");

})

app.listen(port, () => {

console.log(`Example app listening at http://localhost:${port}`)

})

})();

מבחינת תחביר אתם יכולים לשים לב שהקוד עטוף בפונקציה אסינכרונית שמפעילה את עצמה, וכל זה כדי שאוכל לכתוב await בתוך הקוד הראשי. בנוסף חשוב לראות שהכתובת של שרת המונגו מגיעה מבחוץ דרך משתנה סביבה.

קובץ ה docker-compose.yml הבא מפעיל בסיס נתונים מונגו וגם את השרת:

version: "3.9"

services:

web:

image: node:17

working\_dir: /app

command: bash -c "npm install && node server-starter.js"

volumes:

- ".:/app"

ports:

- "3000:3000"

environment:

MONGO\_USER: root

MONGO\_PASSWORD: example

MONGO\_HOST: mongo

mongo:

image: mongo:5

command: bash -c "sleep 60 && /usr/local/bin/docker-entrypoint.sh mongod"

environment:

MONGO\_INITDB\_ROOT\_USERNAME: root

MONGO\_INITDB\_ROOT\_PASSWORD: example

ב Dockerfile הגדרתי שני סרביסים: הראשון אמור להיות מוכר לכם והוא פשוט מפעיל שרת Node. הוא כולל את מיפוי ה Volume כדי לקבל את קוד המערכת בתוך הקונטיינר ומיפוי פורטים כדי שנוכל להתחבר לשרת מהמחשב המקומי. בסרביס השני כבר יש רעיון חדש. המפתח command בסרביס השני דורס את פקודת ברירת המחדל של האימג' מונגו, ובמקומה מפעיל את:

bash -c "sleep 60 && /usr/local/bin/docker-entrypoint.sh mongod"

אני רוצה לגרום למונגו לחכות כמה שניות לפני שהוא עולה, כדי לתת ליישום ה Node.JS שלי זמן לנסות להתחבר אליו ושמונגו לא יהיה זמין. בשביל זה הוספתי sleep לפני הפקודה "האמיתית" של האימג'. אבל, איך אפשר לדעת שהפקודה האמיתית של האימג' היא דוקא mongod? ומה עושים במקרה הכללי כשלא בטוחים מה הפקודה האמיתית של האימג'?

בשביל זה אנחנו צריכים להיכנס לקובץ ההוראות שיצר את האימג', קובץ שנקרא Dockerfile. לכל אימג' בדוקר יש קובץ כזה וגם אנחנו בהמשך הקורס נלמד לכתוב קבצי Dockerfile. באימג'ים הרשמיים של דוקר, קבצי ה Dockerfile הם ציבוריים ואתם יכולים למצוא אותם בגיטהאב. מתוך dockerhub תמיד יהיה קישור לקובץ ה Dockerfile שיצר כל אימג' מדף האימג'.

במקרה של מונגו מעקב אחרי הקישורים מביא אותי לקובץ ה Dockerfile שנמצא בקישור הזה: <https://github.com/docker-library/mongo/blob/fd0435ccfb9bb82c788370152deec6d07081f1f4/5.0/Dockerfile>

בשורה 119 שלו אני מוצא את השורה:

CMD ["mongod"]

וזה אומר ש mongod היא פקודת ברירת המחדל של האימג'. ה Dockerfile מגדיר גם סקריפט שנקרא Entrypoint בשורה 116 עם השורה:

ENTRYPOINT ["docker-entrypoint.sh"]

סך הכל שתי הפקודות האלה מגדירות מה תהיה תוכנית ברירת המחדל שרצה כשהאימג' עולה. תוכנית זו תהיה:

/usr/local/bin/entrypoint.sh mongod

כלומר קודם כל ה Entrypoint Script ואחריו בתור פרמטר ראשון עובר הערך של CMD.

הפקודה Command בתוך קובץ docker-compose.yml משנה את הערך של CMD מבחוץ. הסקריפט entrypoint.sh בדרך כלל מפעיל את הפרמטר הראשון שהוא קיבל אם פרמטר כזה עבר. שילוב הדברים אומר שאני יכול לשנות את פקודת ברירת המחדל או להשתמש בקיימת או לשלב. אני רוצה לחכות כמה שניות לפני שמפעיל את בסיס הנתונים לכן אני כותב:

bash -c "sleep 60 && /usr/local/bin/docker-entrypoint.sh mongod"

כשאני מפעיל docker compose up עם קבצי הדוגמה אני מקבל את הפלט:

Attaching to text-mongo-1, text-web-1

text-web-1 |

text-web-1 | up to date, audited 89 packages in 3s

text-web-1 |

text-web-1 | 5 packages are looking for funding

text-web-1 | run `npm fund` for details

text-web-1 |

text-web-1 | 1 critical severity vulnerability

text-web-1 |

text-web-1 | To address all issues, run:

text-web-1 | npm audit fix

text-web-1 |

text-web-1 | Run `npm audit` for details.

text-web-1 | npm notice

text-web-1 | npm notice New minor version of npm available! 8.5.1 -> 8.7.0

text-web-1 | npm notice Changelog: <https://github.com/npm/cli/releases/tag/v8.7.0>

text-web-1 | npm notice Run `npm install -g npm@8.7.0` to update!

text-web-1 | npm notice

text-web-1 | /app/node\_modules/mongodb/lib/sdam/topology.js:312

text-web-1 | const timeoutError = new error\_1.MongoServerSelectionError(`Server selection timed out after ${serverSelectionTimeoutMS} ms`, this.description);

text-web-1 | ^

text-web-1 |

text-web-1 | MongoServerSelectionError: connect ECONNREFUSED 172.21.0.2:27017

text-web-1 | at Timeout.\_onTimeout (/app/node\_modules/mongodb/lib/sdam/topology.js:312:38)

text-web-1 | at listOnTimeout (node:internal/timers:559:17)

text-web-1 | at processTimers (node:internal/timers:502:7) {

text-web-1 | reason: TopologyDescription {

text-web-1 | type: 'Unknown',

text-web-1 | servers: Map(1) {

text-web-1 | 'mongo:27017' => ServerDescription {

text-web-1 | \_hostAddress: HostAddress { isIPv6: false, host: 'mongo', port: 27017 },

text-web-1 | address: 'mongo:27017',

text-web-1 | type: 'Unknown',

text-web-1 | hosts: [],

text-web-1 | passives: [],

text-web-1 | arbiters: [],

text-web-1 | tags: {},

text-web-1 | minWireVersion: 0,

text-web-1 | maxWireVersion: 0,

text-web-1 | roundTripTime: -1,

text-web-1 | lastUpdateTime: 219650767,

text-web-1 | lastWriteDate: 0,

text-web-1 | error: MongoNetworkError: connect ECONNREFUSED 172.21.0.2:27017

text-web-1 | at connectionFailureError (/app/node\_modules/mongodb/lib/cmap/connect.js:375:20)

text-web-1 | at Socket.<anonymous> (/app/node\_modules/mongodb/lib/cmap/connect.js:295:22)

text-web-1 | at Object.onceWrapper (node:events:642:26)

text-web-1 | at Socket.emit (node:events:527:28)

text-web-1 | at emitErrorNT (node:internal/streams/destroy:164:8)

text-web-1 | at emitErrorCloseNT (node:internal/streams/destroy:129:3)

text-web-1 | at processTicksAndRejections (node:internal/process/task\_queues:83:21)

text-web-1 | }

text-web-1 | },

text-web-1 | stale: false,

text-web-1 | compatible: true,

text-web-1 | heartbeatFrequencyMS: 10000,

text-web-1 | localThresholdMS: 15,

text-web-1 | logicalSessionTimeoutMinutes: undefined

text-web-1 | }

text-web-1 | }

text-web-1 |

text-web-1 | Node.js v17.6.0

text-web-1 exited with code 1

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.943+00:00"},"s":"I", "c":"NETWORK", "id":4915701, "ctx":"thread1","msg":"Initialized wire specification","attr":{"spec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":13},"incomingInternalClient":{"minWireVersion":0,"maxWireVersion":13},"outgoing":{"minWireVersion":0,"maxWireVersion":13},"isInternalClient":true}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.943+00:00"},"s":"I", "c":"CONTROL", "id":23285, "ctx":"thread1","msg":"Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.944+00:00"},"s":"W", "c":"ASIO", "id":22601, "ctx":"thread1","msg":"No TransportLayer configured during NetworkInterface startup"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.944+00:00"},"s":"I", "c":"NETWORK", "id":4648601, "ctx":"thread1","msg":"Implicit TCP FastOpen unavailable. If TCP FastOpen is required, set tcpFastOpenServer, tcpFastOpenClient, and tcpFastOpenQueueSize."}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.944+00:00"},"s":"W", "c":"ASIO", "id":22601, "ctx":"thread1","msg":"No TransportLayer configured during NetworkInterface startup"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.944+00:00"},"s":"I", "c":"REPL", "id":5123008, "ctx":"thread1","msg":"Successfully registered PrimaryOnlyService","attr":{"service":"TenantMigrationDonorService","ns":"config.tenantMigrationDonors"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.944+00:00"},"s":"I", "c":"REPL", "id":5123008, "ctx":"thread1","msg":"Successfully registered PrimaryOnlyService","attr":{"service":"TenantMigrationRecipientService","ns":"config.tenantMigrationRecipients"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.944+00:00"},"s":"I", "c":"CONTROL", "id":5945603, "ctx":"thread1","msg":"Multi threading initialized"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.945+00:00"},"s":"I", "c":"CONTROL", "id":4615611, "ctx":"initandlisten","msg":"MongoDB starting","attr":{"pid":1,"port":27017,"dbPath":"/data/db","architecture":"64-bit","host":"25c620521d3e"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.945+00:00"},"s":"I", "c":"CONTROL", "id":23403, "ctx":"initandlisten","msg":"Build Info","attr":{"buildInfo":{"version":"5.0.6","gitVersion":"212a8dbb47f07427dae194a9c75baec1d81d9259","openSSLVersion":"OpenSSL 1.1.1f 31 Mar 2020","modules":[],"allocator":"tcmalloc","environment":{"distmod":"ubuntu2004","distarch":"aarch64","target\_arch":"aarch64"}}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.945+00:00"},"s":"I", "c":"CONTROL", "id":51765, "ctx":"initandlisten","msg":"Operating System","attr":{"os":{"name":"Ubuntu","version":"20.04"}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.945+00:00"},"s":"I", "c":"CONTROL", "id":21951, "ctx":"initandlisten","msg":"Options set by command line","attr":{"options":{}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.945+00:00"},"s":"I", "c":"STORAGE", "id":22297, "ctx":"initandlisten","msg":"Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. See http://dochub.mongodb.org/core/prodnotes-filesystem","tags":["startupWarnings"]}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:28.945+00:00"},"s":"I", "c":"STORAGE", "id":22315, "ctx":"initandlisten","msg":"Opening WiredTiger","attr":{"config":"create,cache\_size=1447M,session\_max=33000,eviction=(threads\_min=4,threads\_max=4),config\_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),builtin\_extension\_config=(zstd=(compression\_level=6)),file\_manager=(close\_idle\_time=600,close\_scan\_interval=10,close\_handle\_minimum=250),statistics\_log=(wait=0),verbose=[recovery\_progress,checkpoint\_progress,compact\_progress],"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.611+00:00"},"s":"I", "c":"STORAGE", "id":22430, "ctx":"initandlisten","msg":"WiredTiger message","attr":{"message":"[1650887009:611714][1:0xffffbad27980], txn-recover: [WT\_VERB\_RECOVERY\_ALL] Set global recovery timestamp: (0, 0)"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.611+00:00"},"s":"I", "c":"STORAGE", "id":22430, "ctx":"initandlisten","msg":"WiredTiger message","attr":{"message":"[1650887009:611753][1:0xffffbad27980], txn-recover: [WT\_VERB\_RECOVERY\_ALL] Set global oldest timestamp: (0, 0)"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.615+00:00"},"s":"I", "c":"STORAGE", "id":4795906, "ctx":"initandlisten","msg":"WiredTiger opened","attr":{"durationMillis":670}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.615+00:00"},"s":"I", "c":"RECOVERY", "id":23987, "ctx":"initandlisten","msg":"WiredTiger recoveryTimestamp","attr":{"recoveryTimestamp":{"$timestamp":{"t":0,"i":0}}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.620+00:00"},"s":"I", "c":"STORAGE", "id":4366408, "ctx":"initandlisten","msg":"No table logging settings modifications are required for existing WiredTiger tables","attr":{"loggingEnabled":true}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.620+00:00"},"s":"I", "c":"STORAGE", "id":22262, "ctx":"initandlisten","msg":"Timestamp monitor starting"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.622+00:00"},"s":"W", "c":"CONTROL", "id":22120, "ctx":"initandlisten","msg":"Access control is not enabled for the database. Read and write access to data and configuration is unrestricted","tags":["startupWarnings"]}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.622+00:00"},"s":"W", "c":"CONTROL", "id":22138, "ctx":"initandlisten","msg":"You are running this process as the root user, which is not recommended","tags":["startupWarnings"]}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.622+00:00"},"s":"W", "c":"CONTROL", "id":22140, "ctx":"initandlisten","msg":"This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind\_ip <address> to specify which IP addresses it should serve responses from, or with --bind\_ip\_all to bind to all interfaces. If this behavior is desired, start the server with --bind\_ip 127.0.0.1 to disable this warning","tags":["startupWarnings"]}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.623+00:00"},"s":"I", "c":"STORAGE", "id":20320, "ctx":"initandlisten","msg":"createCollection","attr":{"namespace":"admin.system.version","uuidDisposition":"provided","uuid":{"uuid":{"$uuid":"1127e085-7f8d-4196-8505-1b98962c3ac8"}},"options":{"uuid":{"$uuid":"1127e085-7f8d-4196-8505-1b98962c3ac8"}}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.627+00:00"},"s":"I", "c":"INDEX", "id":20345, "ctx":"initandlisten","msg":"Index build: done building","attr":{"buildUUID":null,"namespace":"admin.system.version","index":"\_id\_","commitTimestamp":null}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.627+00:00"},"s":"I", "c":"REPL", "id":20459, "ctx":"initandlisten","msg":"Setting featureCompatibilityVersion","attr":{"newVersion":"5.0"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.627+00:00"},"s":"I", "c":"NETWORK", "id":4915702, "ctx":"initandlisten","msg":"Updated wire specification","attr":{"oldSpec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":13},"incomingInternalClient":{"minWireVersion":0,"maxWireVersion":13},"outgoing":{"minWireVersion":0,"maxWireVersion":13},"isInternalClient":true},"newSpec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":13},"incomingInternalClient":{"minWireVersion":13,"maxWireVersion":13},"outgoing":{"minWireVersion":13,"maxWireVersion":13},"isInternalClient":true}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.627+00:00"},"s":"I", "c":"NETWORK", "id":4915702, "ctx":"initandlisten","msg":"Updated wire specification","attr":{"oldSpec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":13},"incomingInternalClient":{"minWireVersion":13,"maxWireVersion":13},"outgoing":{"minWireVersion":13,"maxWireVersion":13},"isInternalClient":true},"newSpec":{"incomingExternalClient":{"minWireVersion":0,"maxWireVersion":13},"incomingInternalClient":{"minWireVersion":13,"maxWireVersion":13},"outgoing":{"minWireVersion":13,"maxWireVersion":13},"isInternalClient":true}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.627+00:00"},"s":"I", "c":"STORAGE", "id":5071100, "ctx":"initandlisten","msg":"Clearing temp directory"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.627+00:00"},"s":"I", "c":"CONTROL", "id":20536, "ctx":"initandlisten","msg":"Flow Control is enabled on this deployment"}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.628+00:00"},"s":"I", "c":"FTDC", "id":20625, "ctx":"initandlisten","msg":"Initializing full-time diagnostic data capture","attr":{"dataDirectory":"/data/db/diagnostic.data"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.628+00:00"},"s":"I", "c":"STORAGE", "id":20320, "ctx":"initandlisten","msg":"createCollection","attr":{"namespace":"local.startup\_log","uuidDisposition":"generated","uuid":{"uuid":{"$uuid":"aedca484-f70f-4ec2-9532-1f46fed2c792"}},"options":{"capped":true,"size":10485760}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.634+00:00"},"s":"I", "c":"INDEX", "id":20345, "ctx":"initandlisten","msg":"Index build: done building","attr":{"buildUUID":null,"namespace":"local.startup\_log","index":"\_id\_","commitTimestamp":null}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.634+00:00"},"s":"I", "c":"REPL", "id":6015317, "ctx":"initandlisten","msg":"Setting new configuration state","attr":{"newState":"ConfigReplicationDisabled","oldState":"ConfigPreStart"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.635+00:00"},"s":"I", "c":"CONTROL", "id":20712, "ctx":"LogicalSessionCacheReap","msg":"Sessions collection is not set up; waiting until next sessions reap interval","attr":{"error":"NamespaceNotFound: config.system.sessions does not exist"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.635+00:00"},"s":"I", "c":"STORAGE", "id":20320, "ctx":"LogicalSessionCacheRefresh","msg":"createCollection","attr":{"namespace":"config.system.sessions","uuidDisposition":"generated","uuid":{"uuid":{"$uuid":"aa2e93ad-1352-4e75-b9bc-14318732496f"}},"options":{}}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.635+00:00"},"s":"I", "c":"NETWORK", "id":23015, "ctx":"listener","msg":"Listening on","attr":{"address":"/tmp/mongodb-27017.sock"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.636+00:00"},"s":"I", "c":"NETWORK", "id":23015, "ctx":"listener","msg":"Listening on","attr":{"address":"127.0.0.1"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.636+00:00"},"s":"I", "c":"NETWORK", "id":23016, "ctx":"listener","msg":"Waiting for connections","attr":{"port":27017,"ssl":"off"}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.643+00:00"},"s":"I", "c":"INDEX", "id":20345, "ctx":"LogicalSessionCacheRefresh","msg":"Index build: done building","attr":{"buildUUID":null,"namespace":"config.system.sessions","index":"\_id\_","commitTimestamp":null}}

text-mongo-1 | {"t":{"$date":"2022-04-25T11:43:29.643+00:00"},"s":"I", "c":"INDEX", "id":20345, "ctx":"LogicalSessionCacheRefresh","msg":"Index build: done building","attr":{"buildUUID":null,"namespace":"config.system.sessions","index":"lsidTTLIndex","commitTimestamp":null}}

הדברים שקרו לפי הסדר:

1. האימג' של נוד התקין את כל התלויות שלו וניסה להעלות את השרת.
2. השרת זרק הודעת שגיאה שהוא לא הצליח להתחבר למונגו והתרסק.
3. מונגו עלה אחרי 60 שניות של המתנה. אבל אף שרת כבר לא מנסה להתחבר אליו.

המודול wait-on של node.js

דרך טובה להתמודד עם בעיות מסוג זה היא ליצור מנגנון שמחכה שבסיס הנתונים יעלה בתוך תוכנית ה Node.JS שלי. ב node יש מודול בשם wait-on שמציע מנגנון מאוד מתוחכם להמתנה למשאבים חיצוניים באמצעות הגדרת מערך של אותם משאבים והפעלת הפונקציה האסינכרונית waitOn שחוזרת רק אחרי שכל המשאבים במערך מוכנים. הקוד נראה כך:

const express = require('express');

const app = express();

const port = 3000;

const { MongoClient } = require("mongodb");

const { MONGO\_USER, MONGO\_PASSWORD, MONGO\_HOST } = process.env;

const uri = `mongodb://${MONGO\_USER}:${MONGO\_PASSWORD}@${MONGO\_HOST}:27017/`;

const waitOn = require('wait-on');

const opts = {

resources: [

`tcp:${MONGO\_HOST}:27017`,

],

};

// Create a new MongoClient

const client = new MongoClient(uri);

async function run() {

try {

await waitOn(opts);

// Connect the client to the server

await client.connect();

// Establish and verify connection

await client.db("admin").command({ ping: 1 });

app.get('/', (req, res) => {

res.send("Connected To Mongo :)");

});

app.listen(port);

} finally {

// Ensures that the client will close when you finish/error

await client.close();

}

}

run().catch(console.dir);

המודול wait-on חושף פקודה בשם waitOn שמקבלת אוביקט אופציות וממתינה למשאבים שמפורטים באוביקט זה במפתח resources שלו. את המשאבים אנחנו כותבים בתור מחרוזות כשסימן נקודותיים מפריד בין החלקים השונים של המשאב, לדוגמה:

tcp:foo.com:8000

או:

http://foo.com:8000/bar

רק כשהמשאבים "מוכנים", כלומר wait-on מצליח לשלוח בקשה ולהתחבר לנתיב שמפורט שם, אז wait-on חוזר והתוכנית ממשיכה. הפעם כשננסה להפעיל את השרת נראה ש node לא מתקדם אחרי התקנת התלויות ובמקום הוא מחכה למונגו שיעלה. רק אחרי שמונגו עולה גם שרת ה node מוכן לעלות ולנסות להתחבר.

נ.ב. מה קורה אם מונגו מתרסק באמצע:

נניח שעברנו בשלום את העליה אבל אחרי שעתיים של ריצה מונגו מחליט להתרסק. בשביל לדמות את זה אני יכול להפעיל את הפקודה:

$ docker compose stop mongo

כרגע שרת האפליקציה לא שם לב שבסיס הנתונים שלו ירד, כי אין לנו שליפות מבסיס הנתונים. אבל אם היינו מנסים לשלוף מידע מבסיס הנתונים אחרי שבסיס הנתונים ירד אז גם שרת האפליקציה היה מתרסק, כי לא היה לו בסיס נתונים לשלוף ממנו.

בסביבת פרודקשן נוכל לטפל במצב כזה באמצעות הוספת מנגנון הפעלה מחדש של הקונטיינר אחרי כל התרסקות. כשנלמד על מעבר לפרודקשן וקוברנטיס נראה איך להגדיר את זה.

בסביבת פיתוח מספיק לי לפתור את נושא העליה המדורגת כי ממילא תמיד אפשר לסגור את הכל ולהפעיל מחדש.

בחיים האמיתיים לפעמים תרצו לממש פיתרונות מורכבים יותר, לדוגמה לבנות מנגנון שמזהה כשסרביס מסוים שהוא מחכה לו נפל ואז מחכה כמה שניות ומנסה להתחבר שוב. אין אפשרות אחת נכונה לכל המצבים והאתגר שלנו כמתכנתים שבונים מערכת מבוזרת הוא להבין מה הפיתרון שמתאים למקרה הספציפי שלנו.

## דוגמה לפיתוח מיקרו סרביס מלא - חלק 1 בסיס הנתונים

כל הדוגמאות שראינו עד עכשיו הציגו את קוד הסרביסים מוכן והדגש היה על בניית קובץ ה docker-compose. בחיים קוד מערכת מוכן לא נוחת עלינו מהשמים. בשיעורים הקרובים נראה איך להשתמש ב docker-compose במצב פיתוח ונבנה יחד מיקרו סרביס מלא הכולל בסיס נתונים, סקריפט שרץ ברקע פעם ביום ו API Server כדי להבין את תהליך הפיתוח ואת הקשר בין החלקים השונים בקומפוז.

הדרישות שלי למצב פיתוח סרביס הן די פשוטות:

1. אני רוצה לערוך את הקוד בתוך עורך טקסט על המכונה שלי.
2. אני רוצה ששינוי בקוד ייקלט מיד בתוך הקונטיינר.
3. אני רוצה תהליך של בדיקות אוטומטיות שירוץ כל הזמן ברקע.
4. אני רוצה להיות מסוגל להתחבר בקלות לבסיס הנתונים או למערכת הקבצים במכונות בתוך הקונטיינר.

דוגמה לסרביס שכולל בדיקות אוטומטיות ב Node.JS

נקים פרויקט Node.JS ו Express פשוט שכולל עבודה עם בסיס נתונים לשמירת נתוני מזג אוויר והצגתם באמצעות API. המערכת תריץ כל יום בשמונה בבוקר סקריפט שימשוך את נתוני מזג האוויר הנוכחיים בתל אביב מאתר weatherapi.com וישמור את התוצאה בטבלה בבסיס הנתונים. בנוסף יהיה שרת API שכתוב ב Express שיוכל להחזיר את כל הנתונים מאותה טבלה.

יצירת בסיס הנתונים:

נתחיל עם בסיס הנתונים כי כולם תלויים בו. בבסיס הנתונים תהיה טבלה אחת עם נתוני מזג אוויר היסטוריים בתל אביב, ובה שלוש עמודות: שתי העמודות הראשונות לתאריך ושעת הדגימה, עמודה שלישית לטמפרטורה נוכחית במעלות צלזיוס ועמודה רביעית לתיאור מזג האוויר, לדוגמה Sunny.

ב SQL זו הפקודה שיוצרת את הטבלה:

CREATE TABLE weather(

queried\_at\_day date,

queried\_at\_time time,

temprature float,

description text

);

CREATE UNIQUE INDEX weather\_day\_index ON weather(queried\_at\_day, queried\_at\_time);

אני רוצה ליצור קובץ docker-compose.yml שיוצר בסיס נתונים חדש עם הטבלה ריקה כדי שהסקריפט שלי יוכל להוסיף אליה נתונים. לאימג' של Postgres יש כבר מנגנון לאיתחול בסיס נתונים באמצעות קבצי SQL או קבצי Shell Scripts, ובמקרה הזה נוח להשתמש בו. במקרים אחרים אם אתם עובדים עם Image-ים אחרים או אם מנגנון האיתחול שלכם שונה, תצטרכו לבנות בעצמכם אימג' של בסיס הנתונים שתואם למנגנון האיתחול שלכם, או לשנות סיגנון עבודה ולעבור להשתמש במנגנון הקיים.

המנגנון הקיים באימג' של Postgres מבוסס על הרצת קבצים מתיקיה מסוימת וקבועה מראש. אני שם בתיקיה את כל קבצי ה SQL שאני צריך בשביל לאתחל את בסיס הנתונים ובאופן אוטומטי כשנוצר הקונטיינר אם תיקיית ה Data שלו ריקה, כלומר אם אין לו כרגע בסיס נתונים, הוא ייקח את קבצי ה SQL מאותה תיקיית איתחול ויריץ אותם לפי הסדר. במצב פיתוח אני אשתמש בתיקיה מקומית לקבצי ה SQL האלה ובמצב ייצור אני אבנה אימג' ואשים אותם כחלק מהאימג'.

אני יוצר תיקיה בשם db ובתוכה שומר קובץ בשם 01-init-weather-table.sql. בתוך הקובץ אני כותב את פקודת ה SQL שיוצרת את הטבלה. לאחר מכן אני ממשיך לכתוב את קובץ ה docker-compose.yml הבא:

version: "3.9"

services:

db:

image: "postgres"

environment:

POSTGRES\_USER: postgres

POSTGRES\_DB: weather

POSTGRES\_PASSWORD: example

volumes:

- ./db/:/docker-entrypoint-initdb.d/

ports:

- "5432:5432"

חוץ ממיפוי תיקיית האיתחול יצרתי גם מיפוי פורטים כדי שאוכל להתחבר לבסיס הנתונים מתוכנת ניהול בסיס נתונים שיש לי על המחשב (אני משתמש ב dbeaver) ולראות שהכל נוצר כמו שצריך.

בואו ננסה את זה:

1. הפעילו docker compose up וראו בלוג את ההרצה של קובץ ה SQL שלכם.
2. סגרו את המכונות (עם docker compose down או Ctrl+C) והריצו מחדש. שימו לב איך קבצי ה SQL רצים שוב בגלל שאין Volume קבוע שישמור את בסיס הנתונים.

בשביל ליצור בהמשך טבלה חדשה או להריץ פקודת SQL לעדכון בסיס הנתונים אני יוצר קובץ SQL חדש בתיקיית db עם הפקודה החדשה - אני קראתי לקובץ 02-create-test-table.sql - ואז מוריד ומפעיל מחדש את הקונטיינר:

$ docker compose down

$ docker compose up

העברת הסודות ל Secrets

בסיס הנתונים אומנם עובד אבל קובץ ה docker-compose.yml שלנו לא נראה טוב. משתני הסביבה POSTGRES\_USER ו POSTGRES\_DB, שקובעים מה יהיה בסיס הנתונים לפיתוח, מרגישים כמו משהו שנרצה לשנות אותו בנפרד משינוי הגדרות השרתים. בנוסף הסיסמה לבסיס הנתונים מרגישה כמו משהו סודי יותר שלא כדאי לשמור אותה בקובץ שלכולם תהיה גישה אליו וכולם ישנו אותו כל הזמן.

נטפל תחילה בסיסמה.

לדוקר קומפוז מנגנון פשוט שמאפשר לשמור סיסמה בקובץ נפרד ולהעביר אותה פנימה לקונטיינר בזמן ריצה. הסיסמה לא תישמר בצורה מאובטחת או משהו, אבל אפשר לחשוב על זה בתור הכנה לעתיד - כשנעבור לסביבת ניהול קונטיינרים יותר רצינית כמו קוברנטיס אותו מנגנון ימשיך איתנו וכן יאפשר לנו לשמור את הסודות בצורה מאובטחת. בינתיים המנגנון יעזור לנו לעשות סדר בקוד ולשמור כל דבר במקום שלו.

קודם כל אני יוצר קובץ חדש בתיקיה חדשה עבור הסיסמה. אני קורא לתיקיה secrets ובתוכה אני יוצר קובץ בשם postgres-passwd ובתוכו אני כותב את הסיסמה שבחרתי (במקרה שלנו example).

אחר כך אני חוזר לקובץ docker-compose.yml ומוסיף לו את הבלוק הבא בסוף הקובץ:

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

ובתוך הבלוק של db אני מוסיף מערך בשם secrets ובו ערך אחד שנקרא postgres-passwd, בדיוק אותו שם שבחרתי ל secret בבלוק הראשי של הקובץ:

db:

secrets:

- postgres-passwd

שינוי זה יגרום לדוקר קומפוז לקחת את הקובץ secrets/postgres-passwd ולמפות אותו פנימה לתוך הקונטיינר בתוך התיקיה /run/secrets. האימג' של פוסטגרס יודעת לקרוא את הסיסמה לבסיס הנתונים גם מתוך קובץ על הקונטיינר. בשביל זה אני צריך להעביר לה משתנה סביבה בשם POSTGRES\_PASSWORD\_FILE והתוכן שלו הוא הנתיב לקובץ בתוך הקונטיינר בו רשומה הסיסמה. סך הכל קובץ ה docker-compose.yml שלי אחרי הוספת משתנה הסביבה וכל הסודות נראה כך:

version: "3.9"

services:

db:

image: "postgres:14.1"

environment:

POSTGRES\_USER: postgres

POSTGRES\_DB: weather

POSTGRES\_PASSWORD\_FILE: /run/secrets/postgres-passwd

secrets:

- postgres-passwd

volumes:

- ./db/:/docker-entrypoint-initdb.d/

ports:

- "5432:5432"

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

העברת משתני הסביבה לקובץ env

האתגר הבא והאחרון לשיעור זה הוא העברת משתני הסביבה לקובץ סביבה. קובץ סביבה הוא פשוט קובץ שמגדיר את כל מה שכתוב בתוך בלוק environment, והיתרון שלו שאנחנו יכולים לשתף את המשתנים האלה בין מספר בלוקים ב docker compose, כך שכשנרצה להוסיף סרביס שיתחבר לבסיס הנתונים לא נצטרך לשכפל את שם בסיס הנתונים, המשתמש או הנתיב לקובץ הסיסמה.

אני יוצר תיקיה בשם env ובתוכה קובץ בשם dev.env עם התוכן הבא:

POSTGRES\_USER=postgres

POSTGRES\_DB=weather

POSTGRES\_PASSWORD\_FILE=/run/secrets/postgres-passwd

ואז מעדכן את קובץ ה docker-compose.yml שלי ומחליף את בלוק ה environment בשורה:

env\_file: ./env/dev.env

מפעיל שוב לוודא שהכל עדיין עובד ואם כן אנחנו מוכנים להמשיך לשלב הבא.

## דוגמה לפיתוח מיקרו סרביס מלא - חלק 2 תהליך שרץ ברקע

אחרי שיש לנו בסיס נתונים מוכן נמשיך לבניית סקריפט שמריץ ברקע תהליך מסוים פעם ביום. במצב פיתוח נכתוב את הקוד ונריץ את הסקריפט ידנית, ובמצב ייצור נגדיר אותו שירוץ אוטומטית.

 יצירת סקריפט שכותב לבסיס הנתונים:

החלק הבא במערכת הוא סקריפט שמתחבר ל API ולבסיס הנתונים, שולף נתונים מה API וכותב אותם לבסיס הנתונים. הסקריפט בעיקרון ירוץ פעם ביום אבל אם במקרה הוא ירוץ יותר פעמים זה לא נורא. האינדקס הייחודי בבסיס הנתונים ידאג שבכל מקרה לא תהיה לי יותר מדגימה אחת לכל נקודת זמן.

בגלל שהסקריפט מתחבר לבסיס הנתונים אני צריך להעביר לו את הגדרות החיבור. יש לי כבר קובץ env וקובץ סוד עם הסיסמה לבסיס הנתונים ולכן אני יכול להעביר אותם דרך קובץ docker-compose.yml. הסקריפט גם יצטרך למשוך מידע משירות מזג האוויר weatherapi.com. אתר זה דורש רישום קצר (וחינמי) אחריו אתם מקבלים API Key בו צריך להשתמש בכל שאילתה. אני שומר את ה API Key הזה גם בקובץ סוד ומוסיף בלוק מתאים לסודות של docker compose.

שלב אחרון בתבנית של הסקריפט הוא החלטה על האימג' והפקודה. אני כותב את הסקריפט ב node.js אז אני אשתמש באימג' הרגיל של נוד. בגדול אני רוצה שהסקריפט ירוץ פעם ביום, אבל במצב פיתוח נוח לי יותר להריץ את הסקריפט ידנית כשאני צריך אותו ולכן אני מפעיל את הפקודה sleep infinity. כשארצה להריץ את הסקריפט פשוט אתחבר לקונטיינר עם docker compose exec ואריץ ידנית.

סך הכל ה docker-compose.yml שלי נראה כך:

version: "3.9"

services:

query-script:

image: "node:17"

working\_dir: /app

command: sleep infinity

volumes:

- ./query-script:/app

env\_file: ./env/dev.env

environment:

WEATHER\_API\_KEY\_FILE: /run/secrets/weather-apikey

secrets:

- postgres-passwd

- weather-apikey

db:

image: "postgres:14.1"

env\_file: ./env/dev.env

secrets:

- postgres-passwd

volumes:

- ./db/:/docker-entrypoint-initdb.d/

ports:

- "5432:5432"

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

weather-apikey:

file: ./secrets/weather-apikey

נמשיך לכתיבת הסקריפט. צרו תיקיה חדשה עבור הסקריפט, הפעילו בתוכה npm init כדי ליצור פרויקט Node חדש, הדביקו את הקוד הבא בקובץ query-weather.js באותה תיקיה:

const axios = require('axios');

const fs = require('fs');

const API\_KEY = fs.readFileSync(process.env.WEATHER\_API\_KEY\_FILE, { encoding: 'utf8' });

axios.get('http://api.weatherapi.com/v1/current.json', {

params: {

q: 'Tel Aviv',

key: API\_KEY,

}

})

.then(function (response) {

// handle success

console.log(response.data);

})

.catch(function (error) {

// handle error

console.log(error);

})

הגדרות docker-compose למצב פיתוח ב Node.JS

האתגר הבא שלנו הוא לבנות שיטת עבודה עם docker-compose שתתאים לנו למצב פיתוח. במצב פיתוח אני רוצה להיות מסוגל כל הזמן לשנות את הקוד ולראות את התוצאה מיד בלי שאצטרך לבנות מחדש את האימג' והקונטיינר.

במצב פיתוח אני לא צריך שהמשימה שלי תרוץ פעם ביום. בפרק על מעבר לפרודקשן נלמד איך ליצור משימה מתוזמנת אמיתית. בינתיים מספיק לי להיות מסוגל להריץ את הסקריפט שלי בצורה ידנית עם docker compose exec.

דרך אחת ליישם מנגנון כזה היא למפות את תיקיית הפרויקט לתוך תיקיית העבודה של האימג' מתוך docker-compose.yml. צריך לשים לב שבמצב פרודקשן אנחנו חושבים שכל תיקיית node\_modules כבר תהיה על האימג' ולכן קונטיינרים לא מפעילים npm install. זה אומר שאתם צריכים להפעיל npm install מהמכונה שלכם לפני שמפעילים docker compose up. אם יש לכם node מותקן בדיוק בגירסה שצריכה להיות באימג' אז אפשר להפעיל את ה node המקומי שלכם. אפשרות יותר בטוחה היא להוסיף פקודה ב docker-compose.yml שתתקין את התלויות. ה docker-compose.yml הוא מקום מעולה לעשות בו התאמות ספציפיות של הריצה לסביבה מסוימת, כיוון שאנחנו נייצר עותק של הקובץ לכל סביבה.

לכן עדכנו את הבלוק של ה query-script ב docker compose כך שיריץ בעצמו npm install בעליה של הקונטיינר:

query-script:

image: "node:17"

working\_dir: /app

command: sleep infinity

volumes:

- ./query-script:/app

secrets:

- postgres-passwd

- weather-apikey

env\_file: ./env/dev.env

environment:

WEATHER\_API\_KEY\_FILE: /run/secrets/weather-apikey

ונסביר מה יש בו:

1. השורה command משנה את הפקודה שהקונטיינר מריץ כשהוא עולה. במקום להריץ את מה שכתוב ב Dockerfile הוא יריץ את הערך של command. בשביל להריץ מספר פקודות אחת אחרי השניה אני מפעיל bash -c ומעביר את שורת הפקודה המלאה שהייתי כותב ב bash.
2. בלוק ה Environment קובע את משתני הסביבה שיעברו לתוכנית. שימו לב שבגלל השימוש בסודות אני לא כותב כאן סיסמאות בצורה ישירה אלא רק את הנתיבים לקבצי הסיסמאות.
3. במפתח env\_file אני מעביר את קובץ משתני הסביבה שלי שיתווסף להגדרות שאני רושם ב environment.
4. במפתח secrets אני רושם את הסודות בהם הקונטיינר משתמש כדי שדוקר קומפוז ימפה לתוכו בתור קבצים.

והפעילו את שני הסרביסים מהקובץ עם:

$ docker compose up

מחלון אחר ומאותה תיקיה הריצו את הסקריפט על המכונה שדוקר-קומפוז הרים באמצעות:

$ docker compose exec query-script node /app/query-weather.js

ואם הכל עבד כמו שצריך תקבלו במסוף את מזג האוויר העדכני בתל אביב:

{

location: {

name: 'Tel Aviv-Yafo',

region: 'Tel Aviv',

country: 'Israel',

lat: 32.07,

lon: 34.76,

tz\_id: 'Asia/Jerusalem',

localtime\_epoch: 1622701173,

localtime: '2021-06-03 9:19'

},

current: {

last\_updated\_epoch: 1622700900,

last\_updated: '2021-06-03 09:15',

temp\_c: 23.3,

temp\_f: 73.9,

is\_day: 1,

condition: {

text: 'Sunny',

icon: '//cdn.weatherapi.com/weather/64x64/day/113.png',

code: 1000

},

wind\_mph: 4.3,

wind\_kph: 6.8,

wind\_degree: 133,

wind\_dir: 'SE',

pressure\_mb: 1016,

pressure\_in: 30.5,

precip\_mm: 0,

precip\_in: 0,

humidity: 57,

cloud: 2,

feelslike\_c: 25,

feelslike\_f: 77,

vis\_km: 10,

vis\_miles: 6,

uv: 6,

gust\_mph: 4.9,

gust\_kph: 7.9

}

}

עדכון הסקריפט וכתיבה לבסיס הנתונים:

הגיע הזמן להמשיך בפיתוח הסקריפט כדי שגם יכתוב את התוצאות לבסיס הנתונים. אני אשתמש בספריה knex.js כדי לכתוב לבסיס הנתונים. נתקין את הספריה עם:

$ npm install --save knex pg

מתוך תיקיית query-script. שימו לב שלמרות שהפקודה הזאת מתקינה דברים לתוך node\_modules בהרצה האמיתית של הסקריפט אני לא אשתמש בקבצים אלה ואצטרך לבנות מחדש את האימג' כדי להתקין מחדש את התלויות. המטרה היחידה שלי כאן בהרצת הפקודה היא להוסיף את הספריות ל package.json ולוודא שאפשר להתקין אותן ושאין לי איזו שגיאת כתיב.

אני רוצה לנסות להתחבר לבסיס הנתונים ולכן צריך לבנות מחדש את האימג' כדי שבתוך האימג' גם יהיו לי התלויות החדשות. אני מאתחל את הסרביס:

$ docker compose down

$ docker compose up

נעדכן את הסקריפט ונוסיף בתחילתו את הקטע הבא כדי לבדוק התחברות לבסיס הנתונים:

const fs = require('fs');

const API\_KEY = fs.readFileSync(process.env.WEATHER\_API\_KEY\_FILE, { encoding: 'utf8' });

const DB\_PASSWORD = fs.readFileSync(process.env.POSTGRES\_PASSWORD\_FILE, { encoding: 'utf8' }).trim();

const knex = require('knex')({

client: 'pg',

connection: {

host : process.env.DB\_HOST,

user : process.env.POSTGRES\_USER,

password : DB\_PASSWORD,

database : process.env.POSTGRES\_DB,

}

});

knex.select().from('weather').then((result) => {

console.log(result);

knex.destroy();

});

שימו לב שהסקריפט עכשיו משתמש בשם של שרת בסיס הנתונים כדי להתחבר אליו. לא יצרתי עדיין את משתנה הסביבה DB\_HOST ב docker-compose.yml, אז עכשיו לפני ההפעלה הבאה אני צריך לעדכן את בלוק משתני הסביבה ולהוסיף אותו:

query-script:

environment:

DB\_HOST: db

הפעלה מחדש של הסקריפט תדפיס מעל מזג האוויר את הסימן [] המייצג מערך ריק. הסיבה היא שבאמת כרגע אין שורות בטבלה weather, וזה סימן טוב כי זה אומר שהסקריפט הצליח להתחבר לבסיס הנתונים.

נשים לב לנקודה חשובה באופן כתיבת הקוד: בשום מקום בקוד אני לא שומר פרטי גישה לבסיס הנתונים או שמות מפורשים של שרתים. אני משתמש בשם db שהוגדר ב docker-compose.yml כדי לקבל קישור לשרת בסיס הנתונים, במשתני סביבה מ docker-compose.yml שמחזיקים את פרטי הגישה וניגש לקבצים ש docker-compose.yml הגדיר את המיפוי שלהם. זה חשוב כי התנהלות כזאת תאפשר לי בקלות להפעיל את הסקריפט גם ממקומות נוספים למשל לצורך העלאה לשרת או לענן.

החלק האחרון בקידוד הוא עדכון הסקריפט כך שיכתוב את נתוני מזג האוויר לבסיס הנתונים, תוך התעלמות משגיאות בהכנסה כתוצאה מהאינדקס, כי אנחנו יודעים שאם מפעילים את הסקריפט כמה פעמים באותו זמן הוא לא יכניס מידע כפול.

נעדכן את הסקריפט כך שיכיל את הקוד הבא:

const axios = require('axios');

const fs = require('fs');

const API\_KEY = fs.readFileSync(process.env.WEATHER\_API\_KEY\_FILE, { encoding: 'utf8' });

const DB\_PASSWORD = fs.readFileSync(process.env.POSTGRES\_PASSWORD\_FILE, { encoding: 'utf8' }).trim();

const knex = require('knex')({

client: 'pg',

connection: {

host : 'db',

user : process.env.POSTGRES\_USER,

password : DB\_PASSWORD,

database : process.env.POSTGRES\_DB,

}

});

axios.get('http://api.weatherapi.com/v1/current.json', {

params: {

q: 'Tel Aviv',

key: API\_KEY,

}

})

.then(async function (response) {

// handle success

const now = Date.now();

const { data } = response;

try {

await knex('weather').insert({

queried\_at\_day: knex.raw('NOW()'),

queried\_at\_time: knex.raw('NOW()'),

temprature: data.current.temp\_c,

description: data.current.condition.text,

});

} catch(err) {

// DB Error, no big deal it's probably

// because of the index

}

knex.destroy();

console.log(response.data);

})

.catch(function (error) {

// handle error

console.log(error);

})

ונריץ.

התוצאה על המסך היא שוב נתוני מזג האוויר, ועכשיו אפשר למצוא גם נתונים היסטוריים בבסיס הנתונים.

התחברות לבסיס הנתונים ובדיקה שהנתונים שם:

אם יש לכם dbeaver מותקן תוכלו להשתמש בו כדי להתחבר לבסיס הנתונים. אם לא אפשר להריץ משורת הפקודה:

$ docker compose exec db psql -U postgres weather

בתוך ממשק העבודה עם בסיס הנתונים אפעיל פקודת select כדי לראות את המידע ששמרתי:

weather=# select \* from weather;

queried\_at\_day | queried\_at\_time | temprature | description

----------------+-----------------+------------+---------------

2021-06-03 | 07:21:30.650691 | 23.8 | Partly cloudy

(1 row)

הסקריפט עבד! הוא פנה לשרת, משך את נתוני מזג האוויר וכתב אותם לבסיס נתונים מקומי אצלי.

## דוגמה לפיתוח מיקרו סרביס מלא - חלק 3 ה API

בשביל שהסרביס יתקשר עם העולם החיצון אנחנו נצטרך לחשוף משהו החוצה. הרבה פעמים זה יהיה שרת API או שרת ווב שמאזין על פורט מסוים ומחזיר מידע. בשיעור זה נכתוב שרת ב Node.JS שמאזין על פורט 80 ומחזיר את נתוני מזג האוויר. אחר כך נחשוף החוצה את פורט 80 שלו באמצעות חיבורו לפורט לבחירתנו על המחשב המקומי.

פיתוח שרת Express:

ניצור תיקיה חדשה עבור שרת ה API שלנו, אני קורא לה weather-api-server ובתוכה אני מפעיל את הפקודה express-generator כדי ליצור פרויקט express חדש:

$ mkdir weather-api-server

$ cd weather-api-server

$ npx express-generator --no-view

גם פרויקט זה יצטרך לגשת לבסיס הנתונים ולכן אני כבר מעדכן את קובץ ה docker-compose.yml שלי כדי לכלול את הסרביס החדש ולהעביר אליו את פרטי ההתחברות. אני עדיין עובד במצב פיתוח ולכן יכול לדלג על יצירת אימג'ים ולהשתמש באימג' הרגיל של node עם מיפוי קבצים:

server:

image: node:17

working\_dir: /app

command: bash -c "npm install && npm install -g nodemon && nodemon bin/www"

volumes:

- ./server:/app

env\_file: ./env/dev.env

environment:

DB\_HOST: db

secrets:

- postgres-passwd

ports:

- "3000:3000"

נסביר את הסעיפים בקובץ:

1. המפתח image מקבל את האימג' node:17 ואיתו המפתח working\_dir קובע את תיקיית העבודה, כלומר התיקיה בה מותקן הפרויקט.
2. המפתח command כולל את הפקודה להרצה. פה אנחנו צריכים להריץ שתי פקודות: תחילה התקנת התלויות ואחר כך הפעלת השרת.
3. אני מחבר את פורט 3000 של המחשב המקומי לפורט 3000 של הקונטיינר, כדי לגשת מבחוץ ל API.
4. משתנה הסביבה DEBUG גורם לשרת להציג יותר אינפורמציה למסוף כדי שיהיה קל למצוא שגיאות.
5. בתוך קובץ הסביבה dev.env יש את כל פרטי הגישה לבסיס הנתונים. אני שומר אותם בקובץ נפרד כדי שיהיה קל לשתף אותם בין services שונים בקובץ ה docker-compose.yml.
6. הסיסמה לבסיס הנתונים ממופה לתוך קובץ כדי שיהיה קל בעתיד לעבור להשתמש במנגנון Secrets של Docker.
7. קוד המקור ממופה לתוך תיקיית הפרויקט כדי שיהיה קל לשנות קבצים ולראות מיד את התוצאה.

אפשר כבר להריץ בדיקה ראשונה עם docker compose up, להתחבר בדפדפן לפורט 3000 ולראות את מסך הפתיחה של אקספרס.

מעבר ל nodemon כדי שיהיה קל לשנות את הקוד:

כרגע כל שינוי קוד שאכניס יחייב אותי לסגור את שרת ה node ולהפעיל אותו מחדש, שזה אומר לסגור את הקונטיינר ולהפעיל אותו מחדש. במצב פיתוח זה פשוט בזבוז זמן.

הכלי nodemon באופן אוטומטי מזהה שקוד השתנה ויודע להוריד ולהעלות שרת נוד כשזה קורה. כל מה שאני צריך בשביל להשתמש בו במצב פיתוח הוא לעדכן את ה docker-compose.yml כדי שיתקין אותו ואז יפעיל אותו. אני יכול להוסיף את nodemon ל package.json בתור devDependency אבל יותר קל להתקין אותו גלובאלית בתוך הקונטיינר בעליה.

נעדכן את פקודת ההפעלה של הקונטיינר לשורה:

command: sh -c "npm install && npm install -g nodemon && nodemon bin/www"

ונפעיל מחדש. עכשיו אפשר לשנות את קוד השרת ולקבל תמיד את הגירסה העדכנית בלי לסגור ולפתוח קונטיינרים.

עדכון הקוד ושליפת המידע מבסיס הנתונים:

בתוך הפרויקט אצור תיקיה חדשה בשם lib ובתוכה קובץ בשם db עם התוכן הבא:

const fs = require('fs');

const { POSTGRES\_PASSWORD\_FILE } = process.env;

const dbPassword = fs.readFileSync(POSTGRES\_PASSWORD\_FILE, { encoding: 'utf8' }).trim();

const knex = require('knex')({

client: 'pg',

connection: {

host : process.env.DB\_HOST,

user : process.env.POSTGRES\_USER,

password : dbPassword,

database : process.env.POSTGRES\_DB,

}

});

async function getLatestReports() {

return await knex.from('weather').select('\*').limit(10);

}

module.exports = {

getLatestReports,

};

ובקובץ routes/index.js אכתוב את הקוד הבא:

var express = require('express');

var router = express.Router();

const { getLatestReports } = require('../lib/db');

/\* GET home page. \*/

router.get('/', async function(req, res, next) {

const reports = await getLatestReports();

res.send(reports);

});

module.exports = router;

בשביל שתוכלו לראות את התוצאה יש למחוק את הקובץ index.html מתיקיית public.

ואפשר לרענן את העמוד בדפדפן כדי לקבל 10 שורות מבסיס הנתונים (לא האחרונות, אין שם עדיין Order. נחזור לזה בשיעור הבא).

## דוגמה לפיתוח מיקרו סרביס מלא - חלק 4 מעבר לסביבת בדיקות

סביבות פיתוח רבות מאפשרות כתיבת בדיקות אוטומטיות והרצה שלהן תוך כדי לתהליך הפיתוח וגם ב Node.JS יש הרבה אפשרויות לסביבות בדיקות. בשיעור זה אראה איך להוסיף לפרויקט סביבת הרצת בדיקות עם ספריית Jest ולהריץ את הבדיקות במקביל לכתיבת הקוד.

דרישות מסביבת בדיקות:

בהרצת בדיקות אוטומטיות אנחנו רוצים מספר דברים שונים מעבודת פיתוח:

1. אין צורך לחשוף פורטים החוצה, כי כל הבדיקות רצות בתוך קונטיינר פנימי.
2. נרצה לעבוד על בסיס נתונים נפרד כדי לא לקלקל נתונים שאולי צריך בשביל הפיתוח.
3. נרצה למחוק את המידע בבסיס הנתונים בין בדיקות כדי שכל בדיקה תוכל ליצור לעצמה את המידע שהיא צריכה.
4. נרצה לכתוב קוד בדיקות ולהקים מנגנון שמריץ את הבדיקות כל פעם שיש שינוי בקוד.

יש המון סוגים של Micro Services שאתם הולכים לכתוב. המטרה שלי בשיעור זה היא לא להציע את הדרך הכי טובה לשלב קוד בדיקות, אלא להראות איך החלקים השונים של המנגנון עובדים יחד. יכול להיות שבמערכת שלכם תשתמשו בספריית בדיקות אחרת או בדרך אחרת להריץ את הבדיקות, וזה בסדר גמור. קחו מהשיעור הזה את העקרונות הכלליים ואת שיטת העבודה ונסו ליישם אותם ב Micro Service שאתם כותבים.

התקנת בסיס נתונים נפרד לבדיקות:

נקודה ראשונה שבטוח תצטרכו בכתיבת בדיקות לסרביס שלכם היא הקמת בסיס נתונים נפרד לבדיקות. זה חשוב כי לפעמים בבדיקות אנחנו רוצים לשבור דברים או להכניס מידע בעייתי לבסיס הנתונים, ואין סיבה שנקלקל משהו בסביבת הפיתוח.

מאחר וכבר כתבנו את המערכת כך שכל הגדרות בסיס הנתונים נשמרות בקובץ env חיצוני, צריך רק לשכפל את הקובץ, לעדכן את הערכים ולעדכן את docker-compose.yml להצביע לקובץ החדש.

צרו קובץ בשם env/test.env עם התוכן הבא:

POSTGRES\_USER=postgres

POSTGRES\_DB=weather\_test

POSTGRES\_PASSWORD\_FILE=/run/secrets/postgres-passwd

WEATHER\_API\_KEY\_FILE=/run/secrets/weather-apikey

אני לא רוצה שסביבת הבדיקות תקלקל שום דבר מסביבת הריצה האמיתית שכבר בניתי, ולכן אני מעדיף ליצור קובץ docker-compose.yml חדש מאפס עבור סביבת הבדיקות. צרו קובץ חדש בשם docker-compose-test.yml שהוא פשוט עותק של docker-compose.yml ושנו בתוכו את הנתיב ל env\_file כך שיצביע לקובץ הסביבה של הבדיקות ולא של הפיתוח.

וכאן אנחנו חייבים להבין נקודה נוספת לפני שנוכל להתחיל להריץ. לא סיפרתי לכם עד עכשיו, אבל כל פעם ש docker compose עולה ובונה קונטיינרים הוא משתמש בשם התיקיה הנוכחית בתור "שם הפרויקט". שם הפרויקט הוא חלק מהשם של כל קונטיינר בפרויקט וכך דוקר קומפוז יודע להפריד בין פרויקטים בתיקיות שונות.

עכשיו בגלל שאנחנו עוברים לאחסן מספר קבצי הגדרות באותה תיקיה, אי אפשר יותר להשתמש בשם התיקיה בתור "שם הפרויקט", כי תהיה התנגשות.

הפרמטר -p קובע לדוקר את שם הפרויקט ולכן בשביל להריץ את פרויקט הבדיקות אני מפעיל:

$ docker compose -f docker-compose-test.yml -p weather-test up

ובשביל לסגור את פרויקט הבדיקות אני אריץ:

$ docker compose -f docker-compose-test.yml -p weather-test down

באופן דומה אם אני רוצה במקביל להריץ את פרויקט הפיתוח מהשיעורים הקודמים אני יכול לעשות את זה מאותה תיקיה בתנאי שאני נותן לו שם, למשל:

$ docker compose -f docker-compose-test.yml -p weather-dev up

וכמובן סוגר אותו עם אותו השם:

$ docker compose -f docker-compose-test.yml -p weather-dev down

כתיבת קוד בדיקות:

בתוך קוד הפרויקט אני רוצה להוסיף קוד בדיקות שיוודא שהמערכת אכן מחזירה את הנתונים הנכונים. תחילה אני מתקין את ספריית jest עצמה ואיתה את ספריית supertest. כנסו לתיקיית שרת ה API ושם כתבו:

$ npm install --save jest supertest

לאחר מכן בבלוק scripts הוסיפו את האפשרויות להרצת בדיקות פעם אחת או באופן מחזורי:

"scripts": {

"start": "node ./bin/www",

"test": "jest",

"test:watch": "jest --watchAll"

},

צרו תיקיה חדשה בשם tests ובתוכה הקובץ starter.test.js עם התוכן הבא:

describe('Sample Test', () => {

it('should test that true === true', () => {

expect(true).toBe(true)

})

})

חיזרו למסוף והפעילו:

$ npm test

> weather-api-server@0.0.0 test /home/shay/work/courses/fullstack/microservices/video-course/demos/21-testing/weather-api-server

> jest

PASS tests/starter.test.js

Sample Test

✓ should test that true === true (1 ms)

Test Suites: 1 passed, 1 total

Tests: 1 passed, 1 total

Snapshots: 0 total

Time: 0.225 s

Ran all test suites.

הפלט שהופיע אצלי מראה שבדיקה אחת הופעלה בהצלחה.

עכשיו אני רוצה להעביר את הרצת הבדיקות לתוך קונטיינר כך שלקוד הבדיקות תהיה גישה לבסיס הנתונים ולאפליקציה. אני מעדכן את ה docker-compose-test.yml לתוכן הבא:

version: "3.9"

services:

weather-api-server:

working\_dir: "/app"

image: "node:16"

command: bash -c "npm install && npm run test:watch"

env\_file:

- "env/test.env"

environment:

DEBUG: weather-api-server:\*

volumes:

- ./weather-api-server/:/app

secrets:

- postgres-passwd

db:

image: "postgres"

env\_file:

- "env/test.env"

volumes:

- ./db/:/docker-entrypoint-initdb.d/

secrets:

- postgres-passwd

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

נפעיל את הפרויקט ונראה את הבדיקה שלנו באוויר:

$ docker compose -f docker-compose-test.yml -p weather-test up

כתיבת בדיקת API מלאה עם חיבור לבסיס נתונים:

נמשיך לכתיבת בדיקה אמיתית לשרת שלנו. ניצור קובץ חדש בתיקיית tests בשם server.test.js ובו התוכן הבא:

const request = require('supertest');

const app = require('../app');

const knex = require('../lib/db');

const meteo = {

queried\_at\_day: '1/1/2021',

queried\_at\_time: '00:00:00',

temprature: 18,

description: 'Test',

};

describe('Weather Endpoint', () => {

beforeEach(async () => {

await knex('weather').del();

await knex('weather').insert(meteo);

});

it('should return current data', async () => {

const res = await request(app)

.get('/');

expect(res.statusCode).toEqual(200)

expect(res.body).toEqual([meteo]);

})

})

שימו לב שהקוד מוחק את כל הנתונים מבסיס הנתונים לפני כל הרצת בדיקה. במקרה שלי זה קל כי יש רק טבלה אחת. במקרים יותר מורכבים נרצה לבנות מנגנון איפוס בסיס נתונים יותר מתוחכם.

הבדיקה נכשלה. מסתבר שהקוד המקורי שכתבתי בשרת מוסיף את השעה 00:00:00 לשדה התאריך. זו התנהגות מבלבלת ולכן עדיף לחזור ולתקן את הקוד. בקובץ routes/index.js עדכנו את הקוד לגירסה הבאה התקינה:

const express = require('express');

const knex = require('../lib/db');

const router = express.Router();

/\* GET home page. \*/

router.get('/', async function(req, res, next) {

const result = await knex('weather').limit(10).select();

res.send(result.map((data) => ({

queried\_at\_day: new Intl.DateTimeFormat('en-US').format(data.queried\_at\_day),

queried\_at\_time: data.queried\_at\_time,

temprature: data.temprature,

description: data.description,

})));

});

module.exports = router;

וכך אפשר להמשיך את התהליך: כותבים קוד, כותבים לו בדיקה ומתקנים את הקוד. צעד אחרי צעד עד שמקבלים מיקרו סרביס שלם עובד.

## בניית אימג'ים עם

## Dockerfile

## ארכיטקטורה של פיתוח אימג'

אימג' דוקר הוא תבנית ממנה אנחנו יוצרים קונטיינרים. את התבנית יוצרת תוכנה שנקראת Docker Daemon שרצה אצלנו על המחשב מתוך קובץ הוראות שנקרא Dockerfile ומתוך תיקיה במערכת הקבצים שנקראת Context Directory.

בעת בניית האימג' אותו Docker Daemon מקבל את ה Context ואת ה Dockerfile ומתחיל לבצע את ההוראות מה Dockerfile אחת אחרי השניה. את התוצאה הוא שומר וזו האימג' ממנה נוכל ליצור קונטיינרים.

בחירת אימג' בסיס:

המשימה הראשונה בעת יצירת אימג' חדש היא בחירת Image בסיס. בעיקרון אפשרי ליצור אימג' בלי אימג' בסיס (נקרא From Scratch), אבל ברוב המוחלט של הפעמים כן נרצה להתחיל ממשהו קיים כי זה פשוט הרבה יותר קל, ויש כבר אימג'ים טובים לכל סוג של סרביס שנרצה. בעתיד כשהמערכת תהיה באוויר שווה לשקול כחלק מסבב אופטימיזציה כן לבנות אימג'ים יותר קטנים או ספציפיים למערכת שלכם, אבל אין טעם להתעסק עם זה כשרק מתחילים לקודד.

בשיעור זה נבנה אימג' פשוט שמדפיס על המסך Hello World באמצעות הפעלת תוכנית Node.JS.

בשביל להפעיל תוכנית Node.JS נרצה להתחיל מאימג' בסיס של node.js, להעתיק אליו את הקוד שאנחנו רוצים להריץ ולהתקין את התלויות.

נפתח תיקיה חדשה על המכונה שתהיה ה Context. אתם יכולים לקרוא לה בכל שם שתרצו ואני אבחר בשביל הדוגמה את השם hello-node-world. בתוך התיקיה אני יוצר קובץ בשם Dockerfile והשורה הראשונה בתוכו בוחרת את אימג' הבסיס ונראית כך:

FROM node:17

הפקודה FROM אומרת לדוקר מה שם אימג' הבסיס ממנו נתחיל ובמקרה שלנו אני בוחר את האימג' node:16. כדאי ביצירת אימג' להשתמש בגירסה מפורשת ולא לקחת את הגירסה העדכנית ביותר כדי שהאימג' יהיה יציב, כלומר שתמיד תדעו איזה גירסה אתם מקבלים.

### **העתקת קבצים מתיקיית ה Context לאימג'**

בגלל שאנחנו בונים אימג' אנחנו רוצים שיהיו עליו כל הקבצים הדרושים להפעלה שלו. בדוגמאות בשיעורים קודמים ראינו איך אנחנו מריצים אימג'ים קיימים ואיך אנחנו מעבירים להם את הקוד באמצעות מיפוי Volumes - למשל ראינו איך לשלוח קובץ HTML ל nginx באמצעות מיפוי תיקיה בתור Volume.

הפעם אנחנו רוצים לבנות אימג' שאפשר יהיה להעלות אותו ל Dockerhub ולשתף אותו גם עם אנשים אחרים שאולי אין להם את הקוד. לכן אני רוצה להעתיק את הקוד שאני כותב לתוך האימג', יחד עם כל שאר הדברים שהקוד צריך בשביל לרוץ.

בגלל שיש לי Node.JS מותקן על המחשב שלי, אני יכול לפתח את הקוד אצלי על המחשב ובסוף להעתיק אותו לאימג' באמצעות ה Dockerfile. בדרך כלל בזמן הפיתוח אני ארצה להפעיל את הקוד מתיקייה אצלי על המחשב, ובכל Deploy להעביר את הכל לאימג' מסודר.

אני מפעיל את הפקודות הבאות כדי ליצור תוכנית Node.JS ולהתקין חבילה לדוגמה אצלי על המכונה:

$ npm init -y

$ npm install --save colors

נכתוב את הקוד הבא בקובץ בשם main.js:

var colors = require('colors');

console.log('hello'.green);

console.log('i like cake and pies'.underline.red);

console.log('inverse the color'.inverse);

console.log('OMG Rainbows!'.rainbow);

console.log('Run the trap'.trap);

ושווה להריץ רק בשביל לראות שהכל עובד עם:

$ node main.js

בזמן בניית האימג' אני רוצה להעתיק אליה את package.json ו package-lock.json, אבל אני לא רוצה להעתיק את node\_modules מהמכונה שלי ומעדיף לתת ל Docker Daemon להתקין אותו. אפשרות אחת שיש לי היא למחוק את תיקיית node\_modules מתיקיית הפרויקט לפני שאני בונה את האימג'. או אפשרות קצת יותר נוחה היא ליצור קובץ בשם .dockerignore ולרשום בו:

node\_modules/

עכשיו אפשר לחזור ל Dockerfile ולהוסיף אליו את השורות שיעתיקו את הקוד ואת קבצי ה package.json ו package-lock.json פנימה לתוך האימג':

FROM node:16

WORKDIR /app

COPY . .

הפקודה האחרונה פשוט תעתיק את כל התוכן של תיקיית ה Context פנימה לתוך תיקיית העבודה בתוך הקונטיינר, בלי הנתיבים שכתובים ב dockerignore.

הפעלת פקודות בזמן בניית אימג'

אחרי שהעתקנו את הקבצים צריך להפעיל npm install כדי להתקין את התלויות. הפקודה RUN מפעילה תוכנית בתוך האימג' והפקודה ENV מגדירה משתנה סביבה - ואותו משתנה סביבה גם ישמור על הערך שלו כשהקונטיינר יעלה. בגלל שאימג'ים צריכים להתאים לסביבת פרודקשן אני מגדיר את NODE\_ENV ואז מתקין באופן הבא:

FROM node:16

WORKDIR /app

COPY . .

ENV NODE\_ENV=production

RUN npm install

הגדרת פקודת הרצה:

הרצת קונטיינר מהאימג' שיצרנו

השתמשו ב docker image ls כדי לראות את האימג' החדש שיצרתם ואחר כך הפעילו ממנו קונטיינר עם:

$ docker run -t --rm hello-node-world

אבל רגע - מה המשמעות של ה -t שם? ביום רגיל כל הפלט של תוכניות בתוך קונטיינר מנותב לדוקר עצמו ומשם חזרה אלינו למסוף. המעבר דרך דוקר מקלקל את הצבעים וגורם להדפסה להופיע רק בצבע אחד. באופן כללי כשאנחנו רוצים לקבל פלט מדוקר אלינו למסוף כדאי להוסיף -t וכשרוצים להעביר קלט מהמסוף שלנו לתוך דוקר (למשל בשביל לעבוד עם קונטיינר אינטרקטיבי) צריך להעביר גם -i.

שווה לציין שבדיוק כמו עם האימג'ים מ Docker Hub, גם אנחנו יכולים להפעיל קונטיינר חדש ולבחור להפעיל בו פקודה אחרת מפקודת ברירת המחדל שכתבנו באימג' באמצעות כתיבת הפקודה אחרי שם האימג'. לדוגמה כדי להפעיל פקודת echo נוכל לכתוב:

docker run -t --rm hello-node-world echo hello world

דוקר עדיין יבנה קונטיינר מתוך האימג' שיצרנו ופשוט לא יפעיל את הפקודה שרשמנו ב CMD.

## העברת מידע בזמן ריצה באמצעות Volume

אימג' דוקר הוא תבנית ליצירת קונטיינרים והקונטיינר הוא זה שמריץ את הקוד ועושה את כל העבודה. לפעמים בזמן יצירת האימג' אנחנו עדיין לא יודעים את כל מה שהקונטיינר ידע כשיריצו אותו או שאנחנו רוצים להשאיר חלק מההחלטות לזמן ריצת הקונטיינר. לצורך כך דוקר מציע מספר מנגנונים להעביר מידע לקונטיינר שנוצר מתוך האימג'. המנגנון הראשון נקרא Volumes ועליו נדבר בשיעור זה. בשיעור הבא נדבר על מנגנון נוסף שנקרא Environment Variables.

מתי נשתמש ב Volume:

העברת מידע לקונטיינר בזמן ריצה באמצעות Volume מאפשרת לקונטיינר לכתוב מידע שיישמר גם אחרי שהקונטיינר יסגר, או לקרוא מידע שלא היה זמין בזמן יצירת האימג'. דוגמאות למקרים אמיתיים בהם נשתמש במנגנונים אלה כוללות:

1. באימג' של שרת ווב, נרצה לאפשר למי שמפעיל קונטיינר לבחור מהם דפי האינטרנט שהשרת יגיש.
2. באימג' של תוכנה מסוימת, נרצה לאפשר למי שמפעיל קונטיינר לשנות חלק מההגדרות באמצעות העברת קובץ קונפיגורציה.
3. באימג' של בסיס נתונים, נרצה לאפשר לבסיס הנתונים לכתוב מידע שיישמר גם אחרי שהקונטיינר ייסגר.

השימוש ב Volume לא דורש כלום מבחינת כתיבת האימג' מלבד תיעוד והבנה של המנגנון. בזמן כתיבת האימג' אנחנו מניחים שתיקיה מסוימת תועבר לנו בזמן ריצה באמצעות מיפוי Volume. אנחנו נכתוב לתיקיה זו ערכי ברירת מחדל של דברים שאפשר לדרוס אותם ואם בזמן הפעלת הקונטיינר מישהו יחליט להעביר לנו Volume בצורה אוטומטית ה Volume ימופה לאותה תיקיה והקוד שלנו ישתמש ב Volume במקום במערכת הקבצים הפנימית של הקונטיינר.

### **דוגמה: קונטיינר שמציג תוכן של קובץ:**

נכתוב לדוגמה אימג' עבור קונטיינר שמציג תוכן של קובץ. צרו תיקיה חדשה ובתוכה קובץ טקסט בשם helloworld.txt ובו תוכלו לרשום איזה תוכן שאתם רוצים, ובנוסף קובץ בשם Dockerfile עם התוכן הבא:

FROM alpine:3

COPY helloworld.txt /var/data/

CMD ["cat", "/var/data/helloworld.txt"]

ובנו את האימג' עם:

$ docker build . -t 10-volumes

תחילה נריץ קונטיינר עם קובץ ברירת המחדל ונקבל את תוכן הקובץ helloworld.txt שיצרנו:

$ docker run --rm 10-volumes

This is just a text file

ובגלל שאנחנו יודעים איזה קובץ האימג' קורא נוכל למפות קובץ אחר או תיקיה אחרת באמצעות Volume ולקבל פלט אחר:

$ docker run --rm -v /etc/shells:/var/data/helloworld.txt 10-volumes

# /etc/shells: valid login shells

/bin/sh

/bin/bash

/usr/bin/bash

/bin/rbash

/usr/bin/rbash

/bin/dash

/usr/bin/dash

/bin/zsh

/usr/bin/zsh

/usr/bin/tmux

/usr/bin/screen

מי מנהל את ה Volume:

דוקר יכול למפות כל תיקיה בתור Volume ולהעביר אותה לאימג', ובנוסף דוקר תומך במנגנון ניהול Volumes עצמאי, לא מתוך תיקיות. ניהול כזה יעזור לנו כשנרצה להעלות את האימג'ים שיצרנו לענן (למשל ל AWS) שם אין לנו מערכת קבצים.

אנחנו יכולים לבקש מדוקר ליצור לנו Volume בניהול עצמאי שלו באמצעות הפקודה:

$ docker volume create <volume\_name>

לדוגמה בשביל ליצור Volume בשם volume-demo-datadir אני מפעיל את הפקודה:

$ docker volume create volume-demo-datadir

אחרי שיצרתי את ה Volume אני יכול לראות אותו באמצעות הפקודה:

$ docker volume ls

local volume-demo-datadir

אם אני אעביר עכשיו את ה Volume שיצרתי בתור קלט לקונטיינר חדש שאני יוצר מתוך אימג' הדוגמה שלנו התוצאה עשויה להפתיע. דוקר מזהה שמדובר ב Volume חדש, ומעתיק אליו את התוכן שנמצא באימג' לפני יצירת הקונטיינר. בגלל זה ההפעלה הבאה תציג לי בדיוק את תוכן הקובץ helloworld.txt ששמור באימג':

$ docker run --rm -v volume-demo-datadir:/var/data/ 10-volumes

This is just a text file

בשביל שה Volume יכיל מידע אחר אני צריך קודם לאפס אותו ולכתוב לתוכו קובץ אחר. דרך אחת לעשות את זה היא למפות אותו לתוך קונטיינר אינטרקטיבי, למשל קונטיינר של ubuntu, משם לכתוב את הקובץ helloworld.txt עם התוכן שאנחנו רוצים ואז בחיבור ה Volume לקונטיינר הדוגמה נקבל את תוכן הקובץ ששמור עליו.

סיכום:

שימוש ב Volumes מאפשר לנו להעביר קבצים או תיקיות לתוך קונטיינרים כדי שהקונטיינר יוכל לכתוב מידע שיישאר אחרי שהקונטיינר ייסגר, או כדי להשאיר גמישות באימג' ומקום להעביר מידע בזמן ריצה.

ראינו שאנחנו לא צריכים לציין כלום ב Dockerfile כדי להשתמש ב Volume וזה היה ה Dockerfile שכתבנו:

FROM alpine:3

COPY helloworld.txt /var/data/

CMD ["cat", "/var/data/helloworld.txt"]

ובזמן הריצה ראינו שתי דרכים להעביר Volume לקונטיינר. אפשרות אחת היא למפות תיקיה או קובץ קיימים מהמחשב בתור ה Volume:

$ docker run --rm -v /etc/shells:/var/data/helloworld.txt 10-volumes

ואפשרות שניה היא ליצור Volume באמצעות docker volume create ואז למפות אותו:

$ docker run --rm -v volume-demo-datadir:/var/data/ 10-volumes

בדרך כלל בזמן פיתוח נשתמש במיפוי תיקיות מהמכונה המקומית ובמעבר ל Production נמפה Volumes באמצעות המנגנונים של docker, ובצורה כזאת ה Volume יוכל להישמר בענן יחד עם הסרביס.

## פיתוח אימג' לבסיס נתונים עם קבצי איתחול

קריאה ב Dockerfile של postgres:

בסידרת השיעורים על docker compose ראינו איך להתאים את בסיס הנתונים שנוצר מאימג' של postgres לצרכים שלנו. בשביל זה יצרנו קבצי איתחול בפורמט sql ושמנו אותם בתיקיה מיוחדת. קונטיינר שנוצר מתוך האימג' ידע להסתכל באותה התיקיה בדיוק אחרי שהוא יצר בסיס נתונים חדש ולהפעיל את כל קבצי ה sql שבה.

בשיעור זה נראה איך עובד אותו מנגנון, ואיך להרחיב אותו - תחילה על ידי יצירת אימג' שמגיע כבר עם קבצי האיתחול, אחר כך באמצעות יצירת אימג' שמגיע כבר עם בסיס נתונים מוכן ובסוף ניצור אימג' שגם מריץ סקריפט מסוים בעליה של הקונטיינר לפני שמפעיל את בסיס הנתונים.

התחנה הראשונה בהבנה איך עובד המנגנון של האימג' postgres היא הקובץ Dockerfile של האימג׳. אנחנו יכולים להגיע לקובץ ה Dockerfile שיצר את האימג' בגלל שהאימג' מנוהל כקוד פתוח. הקישור לקובץ מפורסם בדף האימג' על dockerhub - פשוט לחצו על גירסה כלשהי מרשימת התגיות ותגיעו ל Dockerfile שיצר אותה.

אני לוחץ על הגירסה 14.2 ומגיע ל Dockerfile בקישור:

<https://github.com/docker-library/postgres/blob/36abfddd6f7235770d00f8546b199936b0ca77aa/14/bullseye/Dockerfile>.

אני מיד שם לב שהאמג' מתחיל מאימג' הבסיס debian:bullseye-slim, ומתחיל בסידרת פקודות RUN שמאתחלות את המכונה ומתקינות את התלויות ובסוף גם את בסיס הנתונים עצמו. אחריהן אני מוצא מספר פקודות מעניינות.

הפקודה:

ENV PGDATA /var/lib/postgresql/data

מגדירה את משתנה הסביבה PGDATA שיצביע על התיקיה בה פוסטגרס שומר את קבצי בסיס הנתונים שלו. ראינו שאנחנו יכולים בהרצה משורת הפקודה או מתוך docker compose לשנות את הערך של משתנה זה וכך להשפיע על קונטיינרים שרצים מתוך אימג' זה. שווה לשים לב ששאר משתני הסביבה כמו POSTGRES\_USER לא מופיעים בקובץ, פשוט כי הם לא נחוצים לצורך יצירת האימג'.

שתי שורות למטה אנחנו מוצאים את:

VOLUME /var/lib/postgresql/data

פקודת Volume בתוך Dockerfile אומרת לדוקר שהתיקיה הזאת חייבת להיות ממופה מתוך Volume חיצוני - ואם בזמן ריצה לא מעבירים Volume, יש ליצור Volume חדש אנונימי ולמפות אותו לתיקיה זאת.

מאחר וכבר הרצנו בעבר קונטיינרים של פוסטגרס, אנחנו יכולים להריץ:

$ docker volume ls

ונוכל לראות את ה Volumes האנונימיים שנוצרו מפקודות Volume בתוך קבצי Dockerfile. הם נראים בערך כך:

7c3b8a3a657558ab012fbf9823a6e7d186e0e6da599dde1847507c7d9e5c5f72

הסיבה ליצירת Volume לתיקיה מסוימת ב Dockerfile היא שיפור ביצועים. כתיבה וקריאה מ Volume היא קצת יותר מהירה מכתיבה וקריאה ממערכת הקבצים המובנית של קונטיינרים, ולכן אם אנחנו יודעים שקונטיינר הולך לכתוב הרבה קבצים לתיקיה מסוימת אז כתיבה כזאת משפרת ביצועים. רוב ה Dockerfile-ים שאנחנו נכתוב יהיו מיועדים להעברת קוד שלנו לפרודקשן ואנחנו נהיה אלה שנריץ אותם - ולכן באימג'ים שלנו כמעט ולא נראה פקודת Volume ב Dockerfile.

המשחק המעניין הבא, והוא גם זה שגורם לתיקיית האיתחול לעבוד, הוא הגדרת סקריפט איתחול. שימו לב לשורות:

ENTRYPOINT ["docker-entrypoint.sh"]

CMD ["postgres"]

ראינו כבר באימג'ים קודמים שבנינו ש CMD מגדירה איזה פקודה להריץ. עכשיו אנחנו רואים שיש הוראה נוספת שקשורה לאותו נושא - ההוראה ENTRYPOINT. בדרך כלל אימג'ים גנריים כמו פוסטגרס רוצים לאפשר למשתמשים מידה מסוימת של התאמה אישית אבל גם לשמור חלק מהשליטה על פקודת ההפעלה. הדרך שלהם לעשות את זה היא השילוב בין entrypoint ל cmd. הסקריפט שמוגדר ב entrypoint, אם מוגדר, יהיה זה שירוץ כשהקונטיינר עולה, והמערך שמוגדר ב cmd יועבר לסקריפט זה בתור פרמטרים של שורת הפקודה. במקרה של האימג' פוסטגרס, כשקונטיינר יעלה הוא יריץ את הסקריפט docker-entrypoint.sh ויעביר לו את הפרמטר postgres. בתוך הסקריפט אנחנו מצפים למצוא קוד שמטפל בקריאת הפרמטרים של שורת הפקודה וטיפול מתאים בהם.

באותה תיקיית מקור בגיט אני מוצא גם את הקובץ docker-entrypoint.sh. הקובץ הזה הוא סקריפט שרץ בעליית כל קונטיינר מהאימג' פוסטגרס והוא דואג ליצירת בסיס הנתונים, לבדיקה אם הסיסמה מוגדרת, להפעלת כל הסקריפטים מתיקיית האיתחול וכל התנהגות נוספת של הקונטיינר. לא נעבור על כל הקוד אבל כן שווה לשים לב לתנאי בשורה 300 שלו:

if [ "$1" = 'postgres' ] && ! \_pg\_want\_help "$@"; then

ולסיום שלו בשורה 340:

exec "$@"

הסקריפט מפעיל את הפקודה שהוא קיבל בתור פרמטר שורת פקודה (התוכן של ה CMD מה Dockerfile), וכולל טיפול מיוחד במקרה שאותו תוכן הוא הפקודה postgres.

איך לבנות אימג' שמוסיף קבצי איתחול:

עכשיו שאנחנו מבינים איך האימג' של פוסטגרס בנוי אנחנו מוכנים להתחיל להרחיב אותו. הרחבה ראשונה תהיה בניית אימג' שכולל קובץ איתחול לטבלה Built In בתוך האימג'. אימג' כזה יעזור לנו כשנרצה להעביר מערכת לפרודקשן בלי שנצטרך למפות Volumes בזמן ריצת הקונטיינר.

אני יוצר בתיקיה קובץ בשם 01-create-table.sql עם התוכן הבא:

CREATE TABLE test (x integer, y integer);

INSERT INTO test(x, y) VALUES(10, 20);

INSERT INTO test(x, y) VALUES(10, 20);

INSERT INTO test(x, y) VALUES(10, 20);

ויוצר Dockerfile עם התוכן הבא:

FROM postgres:14.2

COPY 01-create-table.sql /docker-entrypoint-initdb.d/

שתי השורות האלה מספיקות כדי לקבל אימג' שכבר מגיע עם קובץ האיתחול שיוצר את הטבלה. נבנה את האימג' וניצור ממנו קונטיינר ונוכל לראות שבסיס הנתונים נוצר עם טבלת הבדיקה ונתוני הבדיקה:

$ docker build . -t postgres-demo1

$ docker run --rm -e POSTGRES\_PASSWORD=ninja postgres-demo1

# a lot of output text, and then

/usr/local/bin/docker-entrypoint.sh: running /docker-entrypoint-initdb.d/01-create-table.sql

CREATE TABLE

INSERT 0 1

INSERT 0 1

INSERT 0 1

איך להפעיל סקריפט בעליה של הקונטיינר:

התאמה אישית יותר מעניינת שאפשר להוסיף תהיה הרצת סקריפט שלי בעליה של הקונטיינר. אותו סקריפט יוכל למשל להוריד קבצי sql מנתיב מסוים ידוע מראש מהרשת, ואז ליצור את בסיס הנתונים לפי ההוראות בנתיב זה. אימג' כזה יכול להיות יותר נוח לסביבת בדיקות, ואז נוכל לעדכן את קבצי האיתחול של בסיס הנתונים בלי ליצור אימג' מחדש.

מאחר שאני יודע איזו פקודה מפעיל האימג' של פוסטגרס, אני יכול לכתוב סקריפט entrypoint שלי שיוריד את הקבצים, ישים אותם במקום הנכון ואז יפעיל את ה docker-entrypoint.sh הראשי של פוסטגרס.

נכתוב קובץ my-docker-entrypoint.sh עם התוכן הבא:

#!/usr/bin/env bash

set -Eeo pipefail

curl https://gist.githubusercontent.com/shay/e5ac316e6e6b50edcd5fe3277c9db580/raw/e8468239b4410c16899f91113104967c3ae860a8/demo.sql -o /docker-entrypoint-initdb.d/01-create-table.sql

echo "Running original entrypoint with $@"

exec /usr/local/bin/docker-entrypoint.sh "$@"

הקובץ מוריד הוראות ליצירת sql מנתיב ידוע מראש שיצרתי, ואחר כך מפעיל את ה docker-entrypoint.sh עם כל הפרמטרים שקיבל, כלומר עם התוכן של מערך cmd.

ה Dockerfile שמתאים לו הוא:

FROM postgres:14.2

RUN apt-get update && apt-get install -y curl

COPY my-docker-entrypoint.sh /usr/local/bin/

RUN chmod +x /usr/local/bin/my-docker-entrypoint.sh

ENTRYPOINT ["my-docker-entrypoint.sh"]

CMD ["postgres"]

הפקודות ENTRYPOINT ו CMD דורסות את ההגדרות של האימג' postgres:14.2. שתי השורות שבאמצע מעתיקות את הסקריפט ונותנות לו הרשאת הרצה, ושורת ה RUN מתקינה את התלויות.

נבנה ונריץ קונטיינר מהאימג' ונראה ששוב נוצרה לנו הטבלה, אבל הפעם לפי ההוראות שבקישור.

## שימוש ב Multistage Builds

בבניית Docker Image יהיו לנו מצבים שאנחנו צריכים אימג' בסיס מסוים כדי לבצע עבודת "הכנה", אבל אחרי שההכנה הסתיימה אפשר לקחת את התוצרים ולהגיש אותם מאימג' אחר. דוגמה פשוטה היא אפליקציית next.js שיוצרת אתר HTML סטטי. אני צריך את node.js וכל מה שקשור אליו בשביל לבנות את האתר הסטטי, אבל אחרי שבניתי אותו אני יכול להגיש את התוצאה מאימג' של nginx.

בדוקר יש מנגנון שנקרא multi stage builds שנועד לתת מענה למצבים כאלה ולאפשר בניה מאימג' אחד ואריזה באימג' אחר. בואו נראה איך זה עובד.

אפליקציית next רגילה באימג' של node:

נתחיל עם כתיבת Dockerfile לאפליקציית next רגילה באימג' של node.js. בשביל לבנות את האפליקציה אני צריך להריץ:

$ npm run build

$ npx next export

בהנחה שיש לי בתיקיה תת-תיקיה בשם next-demo-app, בואו נבנה את ה Dockerfile הבא:

FROM node:17

WORKDIR /app

COPY next-demo-app .

RUN npm install

RUN npm run build

RUN npx next export

CMD ["npm", "run", "start"]

אחרי זה אני בונה את האימג' עם:

$ docker build . -t multistage-demo

אני יכול להריץ את האימג' לראות שהכל עובד:

$ docker run -p 8080:3000 --rm multistage-demo

אבל מה שבאמת מעניין זה לראות את הגודל של האימג':

$ docker images | grep multistage-demo

multistage-demo latest e67684b5f7d1 2 minutes ago 1.24GB

כן ראיתם נכון - זה 1.24 ג'יגה.

מעבר ל Multistage Builds:

אבל אז אני נזכר שאני לא באמת צריך להגיש את האפליקציה משרת של next - כי הפעלתי export וכל האפליקציה שלי זמינה בתור קבצי html סטטיים. לכן מספיק לי להגיש אותה משרת ווב רגיל ופשוט, למשל nginx. נו, אם רק הייתי יכול להעביר את כל הקבצים שבניתי לאימג' nginx ...

ופה נכנס הרעיון של multistage builds. נעדכן את ה Dockerfile לקוד הבא:

FROM node:17 AS builder

WORKDIR /app

COPY next-demo-app .

RUN npm install

RUN npm run build

RUN npx next export

FROM nginx:1.21-alpine

COPY --from=builder /app/out /usr/share/nginx/html/

הוספת שורת FROM נוספת ל Dockerfile מאפשרת לשים בצד את כל מה שיצרנו, ולהתחיל לבנות את האימג' מחדש מתוך אימג' בסיס חדש. בפקודות COPY אני יכול להוסיף --from כדי להעתיק תוצרי בניה מאימג'ים ששמתי בצד. לכן ה Dockerfile המעודכן לוקח את תיקיית out שמכילה את קבצי ה html הסטטיים (אלה שנוצרו ב export), ומעתיק אותה לתיקיה ממנה nginx יגיש את הקבצים הסטטיים שלו.

אפשר לבנות מחדש ולהריץ כמעט עם אותן פקודות (צריך לשנות את מיפוי הפורטים כי nginx מאזין על פורט 80) ואתם תראו שה Dockerfile עדיין עובד, אבל יותר מעניין להסתכל על גודל האימג' עכשיו:

$ docker image ls| grep multistage

multistage-demo latest 76259b691276 About a minute ago 23.8MB

וכן ראיתם נכון - גודל האימג' הוא עכשיו 23 מגה בלבד, למרות שהוא עושה בדיוק את אותו דבר שעשה האימג' של הג'יגה. הסוד הוא פשוט להשתמש בדיוק במה שאנחנו צריכים.

## איך להעלות אימג' שיצרנו ל Docker Hub

מאגרי דוקר (Registries):

מאגר דוקר (Docker Regitry) הוא פשוט מקום ששומר אימג'ים. הפקודה docker pull מושכת אליי למחשב אימג' מהמאגר והפקודה docker push שולחת למאגר אימג' מהמחשב שלי. הפקודה docker run בה השתמשנו כדי ליצור קונטיינרים הפעילה מאחורי הקלעים את docker pull אם היא היתה צריכה להריץ קונטיינר מאימג' שלא היה קיים אצלי על המחשב.

ברירת המחדל של docker pull היא למשוך אימג' מדוקר האב, אבל אנחנו יכולים לבקש ממנו לעבוד עם כל מאגר אחר כולל מאגרים פרטיים.

ננסה את זה יחד. הפעילו docker run --rm hello-world כדי להריץ קונטיינר מהאימג' hello-world, לאחר מכן וודאו באמצעות docker image ls שהאימג' נמצא אצלכם על המכונה. הפעילו docker rm hello-world כדי למחוק את האימג' ובסוף:

$ docker pull hello-world

Using default tag: latest

latest: Pulling from library/hello-world

b8dfde127a29: Pull complete

Digest: sha256:5122f6204b6a3596e048758cabba3c46b1c937a46b5be6225b835d091b90e46c

Status: Downloaded newer image for hello-world:latest

docker.io/library/hello-world:latest

כדי למשוך מחדש את האימג', ולאחר מכן וודאו שוב באמצעות docker image ls שהאימג' שוב על המכונה.

כל אימג' במאגר שמור ב Repository ואיתו יחד שמורים כל הגירסאות של אותו אימג'. פקודת docker pull תשלוף את גירסת latest כברירת מחדל ואפשר לציין גירסה אחרת עם סימן נקודותיים ושם הגירסה אחרי שם האימג'.

העלאת אימג' משלנו לדוקר האב

הפקודה:

$ docker login

מחברת את כלי שורת הפקודה לחשבון דוקר. פרטי ההתחברות (שם וסיסמה) נשמרים בקובץ ~/.docker/config.json. הפקודה:

$ docker logout

מנתקת את החיבור.

אחרי שהתחברנו נוכל להעלות אימג' שלנו ל Docker Hub. אני אקח את האימג' של שרת Node.JS שבנינו באחד השיעורים הקודמים. אני מזכיר שה Dockerfile שלו נראה כך:

FROM node:16

COPY . /app

WORKDIR /app

RUN npm install

ENV DEBUG=""

ENV PORT=3000

CMD ["npm", "start"]

ובתוך תיקיית app היה שם פרויקט שרת ווב ב Express.

תחילה אני נכנס לתיקיית הפרויקט ובונה את האימג', אבל הפעם בבנייה אני נותן לו שם שמורכב משם המשתמש שלי ב Docker Hub, אחריו / ואחריו שם האימג' כלומר:

$ docker build app -f Dockerfile -t shay/node-webapp-demo

אחרי שהבניה הסתיימה אני יכול להשתמש בפקודת push כדי לדחוף את האימג' ל Docker Hub:

$ docker push shay/node-webapp-demo

העלאה לריפוזיטורי פרטי:

חשבון דוקר החינמי שקיבלתם מאפשר לכם לפתוח ריפוזיטורי פרטי אחד, או בתשלום של $5 לחודש תוכלו לקבל כמה ריפוזיטורים פרטיים שתרצו. זיכרו שכל ריפוזיטורי מחזיק את כל הגירסאות של אימג' מסוים.

בואו ניצור ריפוזיטורי פרטי ונעלה גם אליו את האימג' שיצרנו.

אני נכנס לאתר Docker Hub ושם לוחץ על הכפתור Create Repository. אני בוחר את השם my-node-webapp ובהגדרות הגישה בוחר שהריפוזיטורי יהיה פרטי. אני שומר עם כפתור Create ובמסך הבא מקבל את ההוראות איך להעלות את האימג'.

עכשיו בחזרה לשורת הפקודה אני בונה מחדש את האימג' עם השם שמתאים לריפוזיטורי שפתחתי:

$ docker build app -f Dockerfile -t shay/my-node-webapp:latest

ומעלה אותה עם:

$ docker push shay/my-node-webapp:latest

ריענון של דף הריפוזיטורי בדפדפן יראה לי את האימג' עם התגית latest. עכשיו כשאני מתנתק מ docker אני כבר לא יכול למשוך את האימג' בגלל שהוא פרטי:

$ docker logout

$ docker pull shay/my-node-webapp:latest

## מושגים בסיסיים בקוברנטיס

## קוברנטיס היא מערכת הפעלה לקלאסטרים:

אתם יודעים כבר שלמחשבים יש מערכות הפעלה: יש Windows, Linux ו OS/X; לטלפונים יש את Android ו iOS ועוד כמה וגם מכשירי IOT הרבה פעמים מגיעים עם איזושהי מערכת הפעלה מבוססת לינוקס.

מה שלא כל כך אינטואיטיבי הוא המחשבה שאפשר לייצר מערכת הפעלה גם למשהו שהוא לא מחשב. למשהו גדול יותר ממחשב. במקרה שלנו לאוסף של שרתים. אוסף של שרתים נקרא Cluster וקוברנטיס הוא (היא?) מערכת הפעלה לקלאסטרים.

מערכת הפעלה רגילה של מחשב אחראית על הפעלת תוכניות וניהול וחלוקת משאבי המערכת לתוכניות שרצות. היא גם דואגת שתוכניות לא ידרסו אחת את השניה, ומספקת לתוכניות גישה קלה ואחידה להתקני הקלט פלט של המחשב. כלומר אם יש לי תוכנית שכתובה למערכת הפעלה Windows, אז זה לא משנה לה איזה סוג מקלדת מחוברת למחשב או אפילו אם זו מקלדת אלחוטית. התוכנית מקבלת את הקלט באמצעות התיווך של מערכת ההפעלה ויכולה לעבוד באותה צורה עם כל המקלדות.

מערכת הפעלה של קלאסטר לוקחת קונספט דומה, ומאפשרת לנו לכתוב "תוכניות" שרצות על הקלאסטר. תוכנית שרצה על קלאסטר תהיה בדרך כלל סוג של סרביס שמקבל בקשות ושולח תשובות - כמו Web Application או בסיס נתונים. קוברנטיס תפעיל את היישומים האלה, תדאג לחלק להם את משאבי הקלאסטר - כלומר תחליט מי ירוץ על איזה שרת ואיזה חלק מהשרת הוא יקבל - ותדאג להשאיר אותם בחיים ולהפעיל אותם מחדש אם אחד מהם מתרסק.

וכמו שתוכנית רגילה לא צריכה להכיר את החומרה עצמה עליה היא רצה, כי היא תמיד עובדת בתיווך מערכת ההפעלה, כך תוכנית שרצה בתוך קוברנטיס לא חייבת להכיר את השרתים עצמם עליהם היא רצה, ויכולה להניח שהם מתנהגים בצורה מסוימת - לפי הממשק שהתוכנית מקבלת מקוברנטיס, כלומר ממערכת ההפעלה.

תוכניות הן אימג'ים, תהליכים הם קונטיינרים:

כמו שמערכת הפעלה של מחשבים יודעת לקחת קבצי הפעלה - לדוגמה ב Windows קבצי EXE - ולהפעיל אותם, כלומר ליצור מהם תהליכים על המחשב, כך קוברנטיס יודעת לקחת קבצי הפעלה שהם OCI Images וליצור מהם תהליכים שהם Containers.

אימג' הוא כל מה שאתם מכירים מעבודה עם Docker או Podman. זה מידע בינארי שמתאר אפליקציה מסוימת ואת כל התלויות של אותה אפליקציה. אימג'ים מאוחסנים ב Registries וגם לפני קוברנטיס ידענו להפעיל קונטיינר מתוך אימג' עם דוקר לדוגמה באמצעות הפעלת:

$ docker run hello-world

קוברנטיס בתור מערכת הפעלה יודע (יודעת?) לקחת אימג'ים כמו ה [hello-world](https://hub.docker.com/_/hello-world) שלנו ולהריץ אותם על הקלאסטר, כלומר לבנות מהם קונטיינר ולתת לאחת המכונות בקלאסטר להריץ את הקונטיינר הזה. קוברנטיס גם תשים לב אם הקונטיינר מתרסק ותדע להפעיל קונטיינר חדש מאותו אימג'.

במערכת הפעלה רגילה אנחנו משתמשים בתוכנת Installer כדי להתקין את התוכניות שלנו. זה לא נדיר להוריד מהרשת קובץ בסיומת msi, להריץ אותו וכך לקבל תוכנה כזאת או אחרת זמינה לנו על המחשב. בקוברנטיס ההתקנה מתבצעת באמצעות קבצי טקסט בפורמט YAML שמתארים לקלאסטר מה האימג' שאנחנו רוצים להתקין ומה דרישות המערכת שהוא צריך.

לדוגמה הקובץ הבא הוא YAML שמגדיר Deployment (שזו התקנה) של אימג' של שרת הווב nginx:

apiVersion: apps/v1

kind: Deployment

metadata:

annotations:

kompose.cmd: kompose convert

kompose.version: 1.25.0 (a70f80cc)

creationTimestamp: null

labels:

io.kompose.service: nginx

name: nginx

spec:

replicas: 1

selector:

matchLabels:

io.kompose.service: nginx

strategy: {}

template:

metadata:

annotations:

kompose.cmd: kompose convert

kompose.version: 1.25.0 (a70f80cc)

creationTimestamp: null

labels:

io.kompose.service: nginx

spec:

containers:

- image: nginx

name: nginx

ports:

- containerPort: 80

resources: {}

restartPolicy: Always

status: {}

פודים, דיפלומנטס וסרביסים

שלושה מושגים שכדאי להכיר כבר מתחילת העבודה שלכם עם קוברנטס הם Pod, Deployment ו Service (יש עוד הרבה, אבל את שלושת אלה נצטרך ממש מהשניה הראשונה):

1. פוד הוא קונטיינר. או לפחות לתקופה הקרובה יהיה לנו נוח מאוד לחשוב עליו בתור קונטיינר. פוד זה משהו שרץ על שרת מתוך אימג', וקוברנטיס יוודא שאותו פוד ימשיך לרוץ כל עוד צריך אותו ואם הוא מתרסק קוברנטס ירים אחד חדש במקומו.
2. דיפלוימנט זה מה שיוצר את הפודים, כלומר זאת התקנה של תוכנה. מרגע ששלחתם Deployment לקוברנטיס הוא (היא?) ינסה ליצור קונטיינרים (פודים) מתוך התיאור שמופיע ב Deployment.
3. סרביס זאת הדרך שלנו להתחבר לפוד. תזכרו שלקוברנטיס יש קלאסטר שלם של שרתים עליו הוא יכול להריץ את הפוד שלכם, וגם אם ישעמם לו או אם אחד השרתים יהיה עמוס קוברנטיס יוכל להזיז את הפוד משרת לשרת. סרביס זה סוג של מיפוי פורטים שאומר שכשמתחברים לפורט מסוים ב IP הציבורי של הקלאסטר אז קוברנטיס צריך לחבר אתכם לפוד המתאים, איפה שהוא לא רץ כרגע.

כל אחד מהשלושה יכול להיות מיוצג על ידי קובץ YAML שמתאר את המאפיינים שלו. ראינו כבר קובץ YAML עבור Deployment, ואם אתם סקרנים אז הנה קובץ YAML פשוט עבור סרביס:

apiVersion: v1

kind: Service

metadata:

annotations:

kompose.cmd: kompose convert

kompose.version: 1.25.0 (a70f80cc)

creationTimestamp: null

labels:

io.kompose.service: nginx

name: nginx

spec:

type: NodePort

ports:

- name: "80"

port: 80

targetPort: 80

selector:

io.kompose.service: nginx

status:

loadBalancer: {}

הקובץ מתאר סרביס שמתחבר ל Deployment של ה nginx שיצרנו קודם. המפתח להבין את הקובץ הוא הבלוק שמתחיל במילה selector, שקובע לאיזה תוכנית הסרביס הזה צריך לחבר אותנו, ו port שקובע לאיזה פורט צריך להתחבר שם.

# קוברנטיס והענן: התקני קלט ופלט

עד לפה ראינו שקוברנטיס היא (הוא?) מערכת הפעלה לקלאסטרים. ראינו שהתפקיד שלה הוא להפעיל תוכניות, כלומר אימג'ים, ולייצר מהם קונטיינרים ולוודא שאותם קונטיינרים ממשיכים לרוץ ומקבלים את משאבי השרת שהם צריכים.

היבט נוסף בו אנלוגיית מערכת ההפעלה תעזור לנו להבין את קוברנטיס הוא הגישה להתקני קלט ופלט. מערכת הפעלה כזכור לכם מחברת את התוכניות שרצות על המחשב עם החומרה של המחשב באמצעות דרייברים. תוכנית לא כותבת ישירות לדיסק אלא בתיווך מערכת ההפעלה. תוכנית לא קוראת ישירות מהמקלדת אלא בתיווך מערכת ההפעלה.

באותו האופן קוברנטיס מתווך את הגישה של הקונטיינרים למשאבים חיצוניים שנשמרים בענן. לדוגמה:

1. מערכת הקבצים של קונטיינר לא שורדת כשהקונטיינר נסגר. בתיווך קוברנטיס הקונטיינר יכול לקבל Volume שזה מקום על הדיסק שאפשר לכתוב אליו והמידע יישמר גם כשהקונטיינר ייסגר.
2. קונטיינר יכול לקבל מידע סודי באמצעות מנגנון Secrets של קוברנטיס.
3. קונטיינר יכול לקבל משתני סביבה שמועברים אליו מתוך קוברנטיס.

קלאסטרים שונים יכולים לממש את ההתקנים החיצוניים בדרכים שונות: לדוגמה קלאסטר קוברנטיס שרץ על ענן של Azure יכול לחבר Volume לכונן אחסון בענן של אזור, וקלאסטר אחר שרץ על אמזון יחבר את ה Volume לכונן אחסון בענן של אמזון. מבחינת התוכנית שלנו שרצה בתוך הקלאסטר הגישה היא זהה בשני המקרים. אנחנו מקבלים Volume וקוברנטיס אחראי על מיפוי אותו Volume להתקן האיחסון החיצוני.

ואם ההקדמה הזאת עשתה לכם חשק להמשיך וללמוד קוברנטיס תשמחו לשמוע שיש אינסוף מדריכים טובים ברשת כולל כאלה שיתנו לכם לתרגל בצורה אינטרקטיבית. אני ממליץ להתחיל מהקישור הזה: <https://kubernetes.io/docs/tutorials/kubernetes-basics/> שכולל סידרה של 6 טוטוריאלס אינטרקטיביים לעבודה עם קוברנטיס והתקנת יישומים באמצעותו.

עמוד התיעוד הראשי של קוברנטס הוא גם מקום לימוד מצוין אם אתם אוהבים לקרוא מגילות של טקסט (נו אל תדאגו יש גם תמונות). קישור: <https://kubernetes.io/docs/concepts/>

## הקמת סביבת קוברנטיס מקומית

## הדרך הכי קלה לקבל קלאסטר קוברנטיס היא להפעיל סביבה מקומית באמצעות Docker Desktop שכבר התקנו. אחרי שנתאים את היישומים שלנו לרוץ על הקלאסטר המקומי נוכל להמשיך ולהעלות אותם לקלאסטר בענן כדי שיהיו זמינים לכל האינטרנט

## איך להפעיל את קוברנטיס מקומית:

ב Docker Desktop נכנס ללוח הבקרה ונלחץ על אייקון גלגל השיניים. בתפריט בצד שמאל אני בוחר Kubernetes ומסמן את תיבת הבחירה Enable Kubernetes. לאחר מכן לוחץ Apply & Restart.

עכשיו אני יכול לפתוח מסוף של Windows Terminal, להיכנס ל WSL ולוודא שהכל הותקן כמו שצריך. הפעילו:

$ kubectl config get-contexts

ותראו שאתם מקבלים את הפלט:

CURRENT NAME CLUSTER AUTHINFO NAMESPACE

\* docker-desktop docker-desktop docker-desktop

לאחר מכן הפעילו:

$ kubectl cluster-info

ותראו שהפלט יראה דומה ל:

Kubernetes control plane is running at https://kubernetes.docker.internal:6443

CoreDNS is running at https://kubernetes.docker.internal:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

זה אומר שקוברנטיס רץ ואתם מחוברים לקלאסטר המקומי שנמצא אצלכם על המחשב.

## התקנת: Kompose.

בשביל להפוך את קבצי ה docker-compose.yml שאנחנו רגילים לעבוד איתם לקבצי הגדרות של קוברנטיס נשתמש בכלי שנקרא kompose. התקינו אותו מהקישור הזה:

[https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/](https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/%20)

בחרו את מערכת ההפעלה שלכם ולכו לפי ההוראות. אחרי שסיימתם תוודאו שהכל עובד ושאתם יכולים להפעיל את הפקודות הבאות (הפלט אצלכם אולי יהיה קצת שונה כי הגירסה של מיניקוב אולי התקדמה מאז שהקלטתי את הוידאו. הדבר החשוב הוא לראות שאתם מקבלים בפלט מספרי גירסאות):

$ kompose version

1.26.1 (a9d05d509)

## העלאת סרביס ראשון ל Kubernetes - חלק 1

## תזכורת: מה היה בסרביס

אנחנו נעבוד על סרביס המונה שהיה הסרביס הראשון שכתבנו בקורס. בסרביס היה קובץ שרת node.js אחד שהתחבר למכונת Redis ושמר שם כמה מבקרים היו באתר. כל פעם שהגיעה בקשת POST לנתיב הראשי הוא העלה את המונה באחד, ובבקשת GET החזיר את הערך העדכני. אתם יכולים לדפדף אחורה לשיעור "המיקרו סרביס הראשון שלי" כדי להיזכר איך בנינו אותו או להמשיך איתי לתזכורת זריזה.

הסרביס היה בנוי מקובץ server.js הבא:

const express = require('express')

const app = express()

const port = 3000

const { createClient } = require('redis');

const { REDIS\_HOST } = process.env;

(async () => {

const client = createClient({ url: `redis://${REDIS\_HOST}` });

client.on('error', (err) => console.log('Redis Client Error', err));

await client.connect();

app.post('/', async (req, res) => {

const newValue = await client.incr('counter');

res.send({ counter: newValue })

})

app.get('/', async (req, res) => {

const counter = await client.get('counter');

res.send({ counter })

})

})();

app.listen(port, () => {

console.log(`Example app listening at http://localhost:${port}`)

})

ומקובץ docker-compose.yml שנראה כמעט כך:

version: "3.9"

services:

web:

image: node:17

working\_dir: /app

command: bash -c "npm install -g nodemon && npm install && nodemon counter.js"

volumes:

- ".:/app"

ports:

- "3000:3000"

environment:

REDIS\_HOST: redis

redis:

image: redis

volumes:

- "data:/data"

volumes:

data:

אני אומר כמעט כי פעם קודמת שעבדנו על דוגמת ה Counter לא היה לנו Volume עבור ה data של רדיס, והמידע נמחק כל פעם שאיפסנו את הקונטיינר. הפעם רציתי לתת לקוברנטיס להתמודד גם עם Volumes ולכן אני יוצר אחד.

יצירת אימג':

קובץ docker-compose.yml שלנו מתאים למצב פיתוח שם אנחנו רוצים שכל שינוי בקוד מיד יגרום לעדכון בתוכנית שרצה, ואנחנו רוצים כמה שפחות שלבים עד העלאת המערכת. בגלל זה כל ההתקנות קורות כשהקונטיינר עולה והקבצים ממופים מהתיקיה המקומית על המחשב שלי.

במעבר לפרודקשן הדרישות שלנו שונות:

1. אנחנו צריכים קונטיינר שיעלה הכי מהר שרק אפשר, כי אולי הוא צריך להחליף קונטיינר אחר שהתרסק.
2. אנחנו צריכים שהקוד יהיה צפוי ומתואם עם "גירסה" שיצרנו בצורה מסודרת.
3. אנחנו בשום פנים לא רוצים לעקוב אחר שינויים בקוד ולהעלות אוטומטית מחדש את המערכת אם הקוד משתנה. קוד פרודקשן צריך לרוץ כמה שיותר יציב ובלי הפרעות.

לכן לפני שאני בכלל יכול לדבר על העלאת המערכת לקוברנטיס אני צריך לדבר על סגירת גירסה ומעבר לפרודקשן. היה לנו שיעור בקורס על מעבר לפרודקשן כשהעלינו את אפליקציית מזג האוויר לשרת אמיתי ואנחנו הולכים לעשות מהלך דומה עם הסרביס מהשיעור הראשון. אתם מוזמנים לעצור כאן את הקריאה ולנסות לבד לבנות גירסת פרודקשן של המערכת לפי מה שלמדנו באותו שיעור או להמשיך ולראות איך אני עושה את המעבר.

קודם כל נבנה Dockerfile עבור כל סרביס שיצרנו. את רדיס אפשר להשאיר כמו שהוא כיוון שאין שם התאמה אישית של המערכת או קוד שלנו, ולכן מספיק לי לבנות Dockerfile לסרביס ה web. זה יהיה התוכן שלו:

FROM node:17

# Create app directory

WORKDIR /app

# Install app dependencies

# A wildcard is used to ensure both package.json AND package-lock.json are copied

# where available (npm@5+)

COPY package\*.json ./

ENV NODE\_ENV=production

RUN npm ci --only=production

# Bundle app source

COPY . .

EXPOSE 3000

CMD [ "node", "server.js" ]

הפעילו (תוך החלפת שם המשתמש shay בשם משתמש dockerhub שלכם):

$ docker build . -t shay/deploy-counter-service:1.0

כדי לוודא שהכל נבנה כמו שצריך.

לאחר מכן בקובץ docker-compose.yml אנחנו רוצים לבדוק את האימג' שיצרנו ולראות שאפשר להפעיל ממנו קונטיינר והסרביס עדיין עובד. אני לא רוצה לקלקל את קובץ ה docker-compose.yml שכבר יצרתי למצב פיתוח ולכן אני יוצר קובץ חדש בשם docker-compose-k8s.yml שיעזור לי בהעלאה לקוברנטיס. בינתיים נרשום בו את התוכן הבא:

version: "3.9"

services:

web:

image: "shay/deploy-counter-service:1.0"

ports:

- "3000:3000"

environment:

REDIS\_HOST: redis

redis:

image: "redis:alpine"

volumes:

- "data:/data"

volumes:

data:

ומריץ את כל המערכת עם:

$ docker compose -f docker-compose-k8s.yml up

כדי לבדוק שהכל עדיין עובד.

העברת האימג' לרג'יסטרי חיצוני:

בדרך כלל בסביבות פרודקשן הקלאסטר ירוץ על איזשהו ספק איחסון ענן וימשוך אימג'ים מ Image Registry של אותו ספק איחסון. אם זה יהיה AWS אז האימג'ים יישמרו ברג'יסטרי פרטי שם; אם על Digital Ocean או Linode אז האימג'ים יישמרו שם וכך הלאה. כמובן שתמיד אפשר לערבב ולשמור אימג'ים ב Registry אחד ואת הקלאסטר אצל ספק אחר, או לשמור את האימג'ים במאגר ציבורי.

בשביל הדוגמאות בקורס אני מעדיף לא להיכנס לקונפיגורציות של קלאסטר ובמקום זה להתמקד בתהליכי העבודה איתו. מנקודת מבט של מתכנתים שצריכים לבנות מערכת שתרוץ על קוברנטס אנחנו לא הולכים לקנפג איך הקלאסטר ימשוך את האימג'ים - אבל כן נצטרך לייצר את התהליך שיוצר את האימג', כלומר את ה Dockerfile. לכן לצורך הדוגמאות בשיעור אני אעלה את האימג'ים למאגר הציבורי של Dockerhub.

נתחיל בבניית האימג'. החליפו את שם המשתמש shay בשם המשתמש שלכם ב dockerhub והריצו:

$ docker push shay/deploy-counter-service:1.0

בחלק הבא נמשיך להעלות את האימג' הזה יחד עם רדיס לקוברנטיס המקומי שרץ אצלי על המחשב.

## העלאת סרביס ראשון ל Kubernetes - חלק 2

יצירת מניפסטים לקוברנטיס:

הכלי kompose שהתקנו בשיעור הקודם יודע לקחת קובץ docker-compose.yml ולהפוך אותו לקבצי Manifest של קוברנטיס. אלה קבצים בפורמט yaml שמתארים דברים שאנחנו שולחים לקלאסטר כגון Deployment ו Service. הגירסה העדכנית ביותר נכון למועד הקלטת הקורס יודעת לעבוד עם קבצי docker-compose.yml בגירסה 3.8 בלבד, ולכן לפני שנוכל להשתמש בקובץ שיצרנו כדי ליצור ממנו קבצי הגדרות של קוברנטיס נצטרך לשנות את מספר הגירסה בקובץ docker-compose-k8s.yml ל 3.8. בכל מקרה אם לא תעשו את זה תקבלו פשוט הודעת שגיאה מ kompose ואז תוכלו לתקן.

צרו תיקיה בשם k8s, כנסו אליה ומתוכה הפעילו את הפקודה:

$ kompose convert -f ../docker-compose-k8s.yml

והפלט שנקבל:

WARN Service "redis" won't be created because 'ports' is not specified

INFO Kubernetes file "web-service.yaml" created

INFO Kubernetes file "redis-deployment.yaml" created

INFO Kubernetes file "data-persistentvolumeclaim.yaml" created

INFO Kubernetes file "web-deployment.yaml" created

נראה מה קיבלנו-

הקובץ redis-deployment.yaml אחראי על התקנת הפוד לשרת רדיס. בתוך הקובץ יש מידע מנהלתי שאומר שיצרנו את הקובץ עם קומפוז ושהוא נקרא redis, והחלק המעניין מתחיל במילה spec. הערך replicas קובע שיהיה לנו שרת רדיס אחד, אסטרטגיית Recreate אומרת שכשאנחנו רוצים להעלות גירסה חדשה אז קוברנטס ימחק את הפוד הישן ויצור פוד חדש. בהמשך אנחנו רואים את הגדרת הקונטיינר שיותקן על הפוד, אנחנו רואים את האימג' שלו שנלקח מהבלוק המתאים בקובץ ה docker-compose-k8s.yml ואת ה restartPolicy שאומר שכל פעם שהקונטיינר מתרסק צריך להפעיל אותו מחדש. כמו דוקר, גם קוברנטס מנהל לעצמו את ה Volumes ואנחנו יכולים להשתמש בכוננים מקומיים או בכונני רשת, או כשיש לכם קלאסטר בענן אז תוכלו להשתמש בשירות האיחסון של אותו ענן בתור Volume.

הקובץ web-deployment.yaml מגדיר את הפוד עבור מכונת ה web שלנו. הוא מאוד דומה לקובץ ה deployment של רדיס.

הקובץ data-persistentvolumeclaim.yaml מגדיר ווליום חיצוני - זה הווליום data שהגדרנו בקובץ docker-compose שלנו. האופן שבו קוברנטיס מנהל Volume-ים חיצוניים קצת שונה מזה של דוקר קומפוז: בקוברנטיס אנחנו יוצרים "בקשה לקבל גישה ל Volume חיצוני", והקלאסטר יודע להקצות איחסונים חיצוניים מתוך האיחסונים החיצוניים שזמינים לו. זה אומר שאם הקלאסטר שלכם רץ על AWS אז הדיסקים החיצוניים שתקבלו ישבו גם הם בענן של AWS. שימו לב שאנחנו גם מבקשים גודל, בדוגמה שלנו 100 מגה.

הקובץ האחרון web-service.yml מגדיר שהפודים של Web יהיו נגישים לפודים אחרים בקלאסטר ויקשיבו לפורט 3000.

הוספת מיפוי פורטים ל redis:

הבדל אחד שאנחנו כבר רואים בין web ל redis ובין קוברנטס ל docker-compose הוא ש web קיבל קובץ Service אבל רדיס לא קיבל אחד. ב docker-compose כל הסרביסים היו נגישים לכל הסרביסים האחרים, וחשיפת הפורטים היתה רק בשביל מכונות מחוץ לקלאסטר שצריכות לגשת לסרביסים. לכן גם בלי לחשוף את הפורטים של רדיס עדיין השרת שלי יכל היה להתחבר אליו.

במעבר לקוברנטס ההתנהגות משתנה. קוברנטס דורש שכל פוד שצריך לתקשר עם פודים אחרים יצהיר על איזה פורטים הוא מאזין ושיהיה לו סרביס מתאים להגדרת חיבורי הפורטים. הגישה מהעולם החיצון תנוהל בצורה אחרת שנקראת Ingress Controller עליה נדבר ממש עוד מעט. מה שבטוח שכמו שדברים מוגדרים כרגע שרת ה Web לא יוכל לגשת למכונת הרדיס ולכן המערכת לא תעבוד.

אפשר לתקן את זה בלי בעיה באמצעות עדכון הקובץ docker-compose-k8s.yml והוספת מיפוי הפורטים שם. זה יהיה הקובץ המעודכן:

version: "3.8"

services:

web:

image: "shay/deploy-counter-to-k8s:1.0"

ports:

- "3000:3000"

environment:

REDIS\_HOST: redis

redis:

image: "redis:alpine"

volumes:

- "data:/data"

ports:

- "6379:6379"

volumes:

data:

ניצור מחדש את קבצי ההגדרות של קוברנטס:

$ kompose convert -f ../docker-compose-k8s.yml

ועכשיו קיבלתי גם קובץ בשם redis-service.yaml שמגדיר את הפורט שרדיס מתקשר בו עם העולם.

התקנת המערכת על הקלאסטר וחיבור פנימה:

אנחנו מוכנים לקחת את הניסיון הראשון שלנו בהתקנת המערכת על קלאסטר Minikube. בתוך תיקיית k8s שמכילה את קבצי הקונפיגורציה הריצו פשוט:

$ kubectl apply -f .

אופן העבודה של קוברנטס דורש מילה או שתיים. אנחנו רגילים שהתקנה היא משהו שאנחנו "עושים", שאנחנו מפעילים docker compose ואז המערכת עולה. ב Kubernetes הנחת העבודה היא שהמערכת כל הזמן באוויר. הפקודה kubectl apply שולחת לקוברנטס קבצי הגדרות חדשים, ועכשיו קוברנטס יצטרך לשבור את הראש איך להתאים את המערכת שכבר יש לו באוויר למצב שביקשנו בקבצי ההגדרות. כמעט תמיד הוא עושה עבודה מצוינת ולכן Deployments עם קוברנטס הם מאוד פשוטים.

איך יודעים אם זה עבד? הכי קל להפעיל docker ps:

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

b68e63d81de8 3900abf41552 "docker-entrypoint.s…" 3 minutes ago Up 3 minutes k8s\_redis\_redis-85b767bd44-h7txc\_default\_354fd672-74f5-4285-bdef-2c9ce3b805aa\_0

6db502f2d5a3 6e6cbb566240 "docker-entrypoint.s…" 3 minutes ago Up 3 minutes k8s\_web\_web-65494c8888-hpb79\_default\_3ca8a8b5-3ee2-4469-bb83-0d5e7c65d1de\_0

ואני רואה שנוצרו לי שני קונטיינרים של קוברנטיס שמתאימים לשני הסרביסים בקובץ הדוקר קומפוז.

אני ממשיך להפעיל:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

redis-85b767bd44-h7txc 1/1 Running 0 5m11s

web-65494c8888-hpb79 1/1 Running 0 5m11s

שמראה לי ששני הפודים שלי באוויר. אני מזכיר שפוד זה קונטיינר בשפה של קוברנטיס (נו, כמעט. פוד יכול להכיל גם מספר קונטיינרים, אבל בשבילנו נוח יותר לחשוב על כל פוד בתור קונטיינר, וממילא רובם הם כאלה).

והפקודה:

$ kubectl get services

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 7d5h

redis ClusterIP 10.106.223.21 <none> 6379/TCP 6m26s

web ClusterIP 10.97.82.198 <none> 3000/TCP 6m26s

שמראה לי ששני הפודים מחוברים לקלאסטר באמצעות שני סרביסים בשמות redis ו web.

### כניסה לקונטיינר ושימוש במערכת

אני עדיין לא יכול להתחבר למערכת מהמחשב הרגיל שלי, אבל כן יכול לקבל Shell על הקונטיינרים הפנימיים ושם לשחק עם הדברים. נפעיל משורת הפקודה:

$ kubectl exec -it deployment/web -- /bin/bash

ונקבל מסוף על הקונטיינר שמריץ את שרת ה Node.JS. כדי לשחק עם השרת אני משתמש ב curl מתוך הקונטיינר:

$ curl localhost:3000

ורואה שאני מקבל את הערך הריק של המונה. אני יכול גם לשלוח בקשת POST ולראות שהערך עולה.

את הלוגים של הקונטיינרים אני יכול לראות באמצעות:

$ kubectl logs deploy/web

$ kubectl logs deploy/redis

## העלאת סרביס ראשון ל Kubernetes - חלק 3

## בחלק זה ניצור Ingress Controller ונגדיר Ingress Rules כדי שנוכל להתחבר לקלאסטר מבחוץ

## הוספת: Ingress

אינגרס הוא הדרך של קוברנטיס לקבל חיבורים מהעולם החיצון. אנחנו מתקינים דבר שנקרא Ingress Controller, שזה בעצם שרת nginx, בתוך הקלאסטר, ואז כל חיבור נכנס מגיע לאותו nginx שיודע לנתב אותו פנימה לתוך הסרביסים של הקלאסטר ומהם לפוד המתאים.

אני מתקין את ה nginx לפי ההוראות כאן: <https://kubernetes.github.io/ingress-nginx/deploy/#docker-desktop>

שזה אומר שאני מפעיל את הפקודה:

$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.1/deploy/static/provider/cloud/deploy.yaml

אחרי התקנת nginx אני צריך להגדיר חוקי כניסה שיגרמו ל nginx להעביר את החיבורים אליי לתוך הפודים. כתבו קובץ חדש בתיקיית k8s בשם ingress.yaml עם התוכן הבא:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: ingress

annotations:

kubernetes.io/ingress.class: "nginx"

spec:

rules:

- http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: web

port:

number: 3000

הקובץ מגדיר שכל גישה לנתיב הראשי תועבר פנימה לסרביס web בפורט 3000. בעתיד כשיהיו לנו יותר סרביסים נוכל לבנות מערכת ניתוב יותר מתוחכמת אבל זה כבר נושא לשיעורים הבאים.

הפעילו שוב:

$ kubectl apply -f k8s

כניסה דרך הדפדפן או עם curl לפורט 80 תחזיר אוביקט עם counter ריק, ושליחת הודעת POST תעלה ב-1 את המונה.

## איפוס הקלאסטר:

בסיום העבודה אתם יכולים להיכנס לממשק הניהול של Docker Desktop, ללחוץ שם על אייקון החיפושית ולבחור מהתפריט "Reset Kubernetes Cluster" כדי לאפס את כל העבודה שעשיתם ולחזור לקלאסטר נקי לקראת משחקים וניסויים נוספים.

מומלץ גם לצאת מ Docker Desktop ולהיכנס חזרה כי לפעמים איפוס קלאסטר תוקע חלק מהיכולות שלו.

## חיבור kubectl לענן של okteto

חברות רבות מציעות לנהל בשבילכם קלאסטר קוברנטיס בענן שלהן. כמעט כולן לוקחות כסף על שימוש אמיתי בפרודקשן אבל כוללות מסלול חינמי להתנסות ולימוד. רוב המסלולים החינמיים מוגבלים בזמן או דורשים תהליך רישום מסורבל, אבל יש יוצאים מהכלל. אוקטטו מציעים מסלול חינמי למפתחים שרוצים להתנסות עם קוברנטס בענן שלהם. בשיעור זה נראה איך אני פותח שם חשבון ומעלה את הסרביס שכתבתי דרך קוברנטס לענן של אוקטטו.

מה זה קוברנטס בענן:

בשיעור הקודם הפעלנו קלאסטר קוברנטיס מקומית על המחשב וראינו איך להפעיל עליו סרביס ראשון. אבל בסביבת הפיתוח, למרות שהיא נראית כמו קלאסטר אמיתי, אנחנו עדיין מדברים על קלאסטר של מכונה אחת שמתאים רק לבדיקות ופיתוח.

חברות כמו אמזון, IBM, גוגל, מייקרוסופט, אורקל, עליבאבא וחברות ענק רבות נוספות מוכרות כבר שנים שירותי ענן, שזה בעצם אפשרות להריץ קוד ולגשת לאיחסון על רשת ציבורית גדולה שהם מתחזקים. גם חברות הוסטינג קטנות יותר כמו Digital Ocean ו Linode כבר מפעילות קלאסטרים של קוברנטס להשכרה.

קוברנטיס בענן הוא בעצם קלאסטר שמנוהל ומתוחזק על ידי חברת הוסטינג. בדיוק כמו שפעם היינו לוקחים מחברה כזאת מכונה (מה שנקרא VPS), היום אותן חברות משכירות לנו גישה לקלאסטר מלא של מכונות ובממשק שאנחנו מכירים - דרך kubectl.

חיבור הכלי kubectl לקלאסטר קוברנטיס בענן שאנחנו לוקחים מחברת איחסון שונה בין חברת איחסון אחת לאחרת אבל תמיד יתבסס על איזשהו קובץ קונפיגורציה שנקבל מחברת האיחסון ונעביר ל kubectl. אחרי הגדרת הקונפיגורציה כל פקודת kubectl שנריץ, ובפרט הפקודה apply ש"מתקינה" את המערכת, תרוץ על הקלאסטר המרוחק.

בשיעור זה אראה לכם איך לפתוח חשבון בשירות הענן של okteto. אני יודע, זאת לא חברה גדולה או מוכרת כמו אמזון או גוגל, אבל שירות הענן שלהם ידידותי וכולל מסלול חינמי שאינו מוגבל בזמן וכמעט לא דורש רישום. זה מאוד נוח למי שרוצה להתחיל לעבוד על מערכות או שרוצה ללמוד איך קוברנטיס עובד בלי להתחייב לשירות איחסון ענן גדול.

חיבור לאוקטטו והגדרת kubectl:

הכניסה לאוקטטו היא דרך הקישור: <https://okteto.com/>

אין צורך להירשם ומספיק ללחוץ על כפתור Login ולבחור Login with Github. לאחר הכניסה תקבלו מסך עם כיתוב בגדול שאומר Your namespace is empty. במסלול החינמי יש לנו 5 מרחבי שמות לעבודה, כל מרחב שמות הוא פרויקט.

נכנס בתפריט הצד ל Settings ושם אני בוחר Download Config File. זה נותן לכם קובץ בשם okteto-kube.config בתוכו נמצאים פרטי הקלאסטר שלכם ופרטי הגישה אליו. שימרו טוב על הקובץ כי מי שיש לו את הקובץ יכול לשנות הגדרות בקלאסטר ולהעלות אליו מערכות חדשות.

במסך ההורדה אוקטטו מציגים לכם את השורה הבאה שעליכם להפעיל מהמסוף כדי להתחיל להשתמש בקובץ הקונפיגורציה שלהם:

$ export KUBECONFIG=$HOME/Downloads/okteto-kube.config:${KUBECONFIG:-$HOME/.kube/config}

הפקודה kubectl קוראת את קובץ הקונפיגורציה שלה ממשתנה סביבה בשם KUBECONFIG, ואם הוא לא מוגדר היא תחפש את קובץ הקונפיגורציה בנתיב הסטנדרטי ~/.kube/config. קוברנטיס יודע לעבוד עם מספר קבצי קונפיגורציה וייתן לנו להחליף ביניהם. כל קובץ קונפיגורציה כולל את פרטי ההתחברות לקלאסטר מסוים.

פקודת ה export שאנחנו רואים מגדירה את הקובץ okteto-kube.config בתור קובץ הקונפיגורציה המרכזי ל kubectl ועדיין שומרת על קובץ הקונפיגורציה "הרגיל" בתור קובץ שני.

שימו לב לפני שאתם מפעילים את ה export לוודא שהנתיב לקובץ okteto-kube.config תואם למה שקיים אצלכם על המכונה ולאחר מכן הפעילו את הפקודה. אחרי הפעלה ומאותו מסוף אני יכול לכתוב פקודת kubectl כמו:

$ kubectl get services

ולקבל את ההודעה:

No resources found in shay namespace.

אומנם הוא התלונן שאין עדיין שום דבר בענן שלי, אבל לפחות הוא הצליח לאיית את שם המשתמש שלי נכון - מה שאומר שהחיבור לאוקטטו עבר בהצלחה.

עכשיו רגע, מה קרה לקלאסטר המקומי שלי? הוא עדיין שם כמובן ואפשר להשתמש ב kubectl כדי לתקשר גם איתו כי רשמנו את שני קבצי הקונפיגורציה בשורת ה export. הפעילו:

$ kubectl config get-contexts

ונקבל:

CURRENT NAME CLUSTER AUTHINFO NAMESPACE

\* cloud\_okteto\_com cloud\_okteto\_com 6b5b39bc-9c83-4d7b-a345-b03922c7e979 shay

docker-desktop docker-desktop docker-desktop

כלומר יש לנו שני קונטקסטים: אחד נקרא cloud\_okteto\_com והוא כרגע הפעיל כי הוא מסומן בכוכבית, והשני נקרא docker-desktop והוא כרגע לא פעיל. ה namespace שלי על okteto הוא shay כי זה שם החשבון, ולכן אני יכול להפעיל:

$ kubectl cluster-info -n shay

Kubernetes control plane is running at https://146.148.56.200:443

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

בשביל להחליף קונטקסט אני משתמש בפקודה:

$ kubectl config use-context docker-desktop

Switched to context "docker-desktop".

עכשיו אם אני אפעיל את אותה פקודת cluster-info אני כבר לא צריך לציין namespace ופשוט מקבל:

$ kubectl cluster-info

Kubernetes control plane is running at https://kubernetes.docker.internal:6443

CoreDNS is running at https://kubernetes.docker.internal:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

מאחר ואנחנו רוצים להתקין את ה counter על אוקטטו, נחזור לקונטקסט שלהם להמשך עבודה:

$ kubectl config use-context cloud\_okteto\_com

התקנת מערכת ה Counter על אוקטטו:

ניכנס לאותה תיקיית k8s שיצרנו בשיעור הקודם וננסה להתקין את המערכת שבנינו לענן של אוקטטו. אני מפעיל:

$ kubectl apply -f .

וזה כמעט עבד. רוב הקבצים הצליחו לעבור בהצלחה אבל אני מקבל הודעת שגיאה על הקובץ ingress.yaml:

Error from server: error when applying patch:

{"metadata":{"annotations":{"kubectl.kubernetes.io/last-applied-configuration":"{\"apiVersion\":\"networking.k8s.io/v1\",\"kind\":\"Ingress\",\"metadata\":{\"annotations\":{},\"name\":\"ingress\",\"namespace\":\"shay\"},\"spec\":{\"rules\":[{\"http\":{\"paths\":[{\"backend\":{\"service\":{\"name\":\"web\",\"port\":{\"number\":3000}}},\"path\":\"/\",\"pathType\":\"Prefix\"}]}}]}}\n"}},"spec":{"rules":[{"http":{"paths":[{"backend":{"service":{"name":"web","port":{"number":3000}}},"path":"/","pathType":"Prefix"}]}}]}}

to:

Resource: "networking.k8s.io/v1, Resource=ingresses", GroupVersionKind: "networking.k8s.io/v1, Kind=Ingress"

Name: "ingress", Namespace: "shay"

for: "ingress.yaml": admission webhook "ingress.webhook.okteto.com" denied the request: host must match the `\*shay.cloud.okteto.net` pattern. Interested in custom hosts support? Upgrade to the Pro plan

ההודעה די מסבירה את עצמה: היא אומרת שהגדרתי מסלול כניסה לקלאסטר שלא דרך ה Hostname שקיבלתי מהם, והסיבה היא שהגדרות ה Ingress שבחרתי לייצר, לא כוללות הגדרה עבור שדה host. הם גם אומרים שהדומיין שהם נתנו לי הוא shay.cloud.okteto.net ולכן ה host חייב להיות דומיין זה או איזשהו סאבדומיין שלו.

אני מוסיף את ההגדרה המתאימה לקובץ ingress.yaml ומקבל:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: ingress

annotations:

kubernetes.io/ingress.class: "nginx"

spec:

rules:

- host: shay.cloud.okteto.net

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: web

port:

number: 3000

ובניסיון הבא כבר מצליח לשלוח את כל המערכת שלי לענן של אוקטטו.

הפעלה חדשה של kubectl get services מחזירה לי את הפלט:

redis ClusterIP 10.153.238.130 <none> 6379/TCP 4m41s

web ClusterIP 10.154.131.3 <none> 3000/TCP 4m39s

והפעלת kubectl get ingress מחזירה את הפלט:

NAME CLASS HOSTS ADDRESS PORTS AGE

ingress <none> shay.cloud.okteto.net 35.225.69.73 80 8h

נלך לראות אם זה עובד: בדפדפן אני נכנס לכתובת שמופיעה בשדה hosts, כלומר ל shay.cloud.okteto.net. ומקבל אוביקט JSON עם מונה מכוון ל null. מתוך המסוף אני שולח בקשת POST לאותה כתובת עם:

$ curl -X POST shay.cloud.okteto.net

ומרענן את העמוד בדפדפן כדי לקבל את הערך החדש של המונה.

מעניין לראות שלא הייתי צריך להתקין את ה Ingress Controller כשקוברנטיס רץ בענן של אוקטטו - הם כבר התקינו אותו בשבילי, ואני הייתי צריך רק להעלות את קובץ ההגדרות.

בסך הכל המעבר מהתקנה מקומית להתקנה בענן של אוקטטו עברה ממש חלק. אתם יכולים להיכנס למסך הניהול של אוקטטו ושם לראות את הלוגים של כל שלושת הסרביסים.

בחרו שירות ענן שנראה לכם מעניין - זה יכול להיות גוגל, אמזון, IBM, אורקל או אפילו Linode או Digital Ocean. נסו להירשם אליו, לקחת שם קלאסטר קוברנטס ולחבר את ה kubectl שלכם לקלאסטר שלקחתם. לאחר מכן השתמשו ב kubectl apply כדי להעלות את מערכת ה Counter לאותו שירות ענן שבחרתם.

אחד הדברים שמתכנתים הכי אוהבים בעבודה עם קוברנטס זה קלות המעבר בין קלאסטרים ובין סביבות. העובדה שאני רק צריך לבחור קובץ קונפיגורציה ולהפעיל kubectl apply אומרת שמאוד קל לי להעלות גירסה לסביבת טסט ואחרי זה לסביבת פרודקשן, או לעבור לעבוד בסביבת פיתוח אישית לכל מתכנת בקלאסטר משלו בענן. העובדה שאני מתקשר באמצעות אותו kubectl ובאמצעות אותן פקודות גם עם minikube שמותקן אצלי על המחשב, גם עם אמזון, גם עם גוגל וגם עם אוקטטו יוצרת תחרות בריאה בין חברות הוסטינג וחווית פיתוח טובה עבורנו.

## העלאת יישום מזג אוויר לקוברנטיס - הכנה לפרודקשן

בניית אימג'ים:

במעבר למצב פרודקשן המטרה שלנו היא לארגן את הקוד כך שלא נצטרך שום דבר שנמצא מקומית על המחשב כדי להריץ את הסרביס. אם נסתכל על ה docker-compose.yml שלנו הבעיות הן בסך הכל:

1. מיפוי תיקיות מקומיות כ Volumes אומר שהקוד נטען מאותן תיקיות במקום ממה ששמור על האימג'.
2. מיפוי הסיסמאות בתור קבצים מקומיים אומר שאנחנו צריכים לשמור מידע סודי ב Plain Text ועל המחשב שלנו.
3. מיפוי קבצי אתחול בסיס הנתונים כ Volume מתיקיה מקומית אומר שאנחנו צריכים את התיקיה הזאת גם על השרת.

את כל הבעיות האלה אפשר לפתור אם נבנה אימג' מסודר לכל חלק במערכת ונשמור את האימג' הזה בריפוזיטורי למשל ב Dockerhub. ראינו כבר איך בונים Dockerfile ואימג'ים ובשיעור זה נשתמש בידע שיש לנו כדי להכין את המערכת לפרודקשן.

יצירת אימג'ים מסודרים והחלפת ה Build ב Image

המערכת שבנינו בשיעורים האחרונים כוללת שני Services ובסיס נתונים אחד. כרגע אנחנו משתמשים באימג' קיים של node.js למצב פיתוח של האפליקציה ואימג' קיים של פוסטגרס בשביל בסיס הנתונים. לשניהם אנחנו ממפים קבצים מספריה מקומית. במצב פיתוח מיפוי כזה הוא מצוין כי הוא חוסך לנו לבנות מחדש את האימג'ים כל פעם שמשנים משהו בקבצים. במצב פרודקשן אנחנו דווקא כן רוצים לסגור את האימג' ושכל העדכונים יהיו יזומים.

לכן אני בונה אימג', כלומר Dockerfile, לכל סרביס. סך הכל יש לי ב docker-compose.yml שלושה סרביסים מוגדרים ולכן נצטרך לבנות 3 קבצי Dockerfile.

נתחיל עם בסיס הנתונים: ה Dockerfile שלו יתחיל מהאימג' postgres וכדאי גם לציין את הגירסה שעליה בדקתם. נכון להקלטת הסרט הגירסה העדכנית ביותר היא 14.0 ולכן אשתמש בה. חוץ מזה אנחנו צריכים להעתיק את הקבצים לאימג' במקום למפות אותם. סך הכל נוסיף לתיקיית db את הקובץ Dockerfile עם התוכן הבא:

FROM postgres:14.2

COPY . /docker-entrypoint-initdb.d/

אני בונה את האימג' עם:

$ docker build . -t shay/deploy-weather-to-k8s-db:1.0

ואז מעלה אותו ל Dockerhub עם:

$ docker push shay/deploy-weather-to-k8s-db:1.0

הסרביס השני הוא השרת weather-api-server ובשבילו אני צריך לבנות Dockerfile שמתאים ל Node.JS. הפעם אנחנו מתחילים מהאימג' node בגירסה 17, מעתיקים את הקבצים ונרצה גם להגדיר את הפקודה שהקונטיינר יפעיל בהעלאה שלו.

בקונטיינר מבוסס node.js צריך לחשוב על התקנת התלויות. בניגוד למצב פיתוח, במצב פרודקשן מקובל להתקין את התלויות בזמן בניית האימג' כלומר שהאימג' יבנה לעצמו את תיקיית node\_modules. בצורה כזאת קונטיינרים יעלו יותר מהר ובצורה יותר אמינה. בשביל שתיקיית node\_modules ממצב הפיתוח לא תועתק בטעות לאימג', אני יוצר קודם כל קובץ .dockerignore עם התוכן הבא:

node\_modules/

ולאחר מכן אני ממשיך ליצור את ה Dockerfile עם התוכן הבא:

FROM node:17

WORKDIR /app

COPY . .

ENV NODE\_ENV=production

RUN npm install

CMD ["node", "bin/www"]

שימו לב שוויתרתי על nodemon ואני פשוט מריץ ישירות את התוכנית שלי. במצב פרודקשן אנחנו רוצים להיות הכי יעילים שרק אפשר. נבנה ונעלה גם את האימג' הזה:

$ docker build . -t shay/deploy-weather-to-k8s-weather-api-server

$ docker push shay/deploy-weather-to-k8s-weather-api-server

לא שכחתי שנשאר עדיין הסקריפט query-script שצריך לרוץ פעם ביום. בהמשך השיעוד נחזור אליו.

דבר נוסף שאנחנו צריכים במעבר לפרודקשן הוא לשנות את הערכים של משתני הסביבה. לא את כולם וכמובן לא חייבים, אבל אנחנו כן רוצים תמיכה בערכים שונים בין הסביבות. לכן נעתיק את הקובץ dev.env לעותק חדש בשם production.env. בשלב זה אני לא משנה את הערכים של אף משתנה סביבה ומשאיר את העדכון רק בתור הכנה לעתיד.

בשביל לבדוק את השינוי ניצור קובץ docker-compose-k8s.yml עם התוכן הבא:

version: "3.9"

services:

weather-api-server:

image: "shay/deploy-weather-to-k8s-weather-api-server:1.0"

ports:

- "3000:3000"

environment:

DB\_HOST: db

env\_file:

- "env/k8s.env"

secrets:

- postgres-passwd

db:

image: "shay/deploy-weather-to-k8s-db:1.0"

env\_file:

- "env/k8s.env"

ports:

- "5432:5432"

secrets:

- postgres-passwd

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

weather-apikey:

file: ./secrets/weather-apikey

ונריץ בעזרת:

$ docker compose -f docker-compose-k8s.yml -p weather-prod up

אם הכל עבד לפי התוכנית אתם אמורים להיות מסוגלים לגלוש ל localhost פורט 3000 ולקבל רשימה ריקה של קריאות מזג אוויר.

מה קורה עם query-script:

זוכרים את הסקריפט שכתבנו שמושך נתוני מזג אוויר ומעדכן את בסיס הנתונים? במצב פיתוח הפעלנו אותו בצורה ידנית, אבל במעבר למצב Production נרצה להריץ אותו בצורה אוטומטית פעם ביום.

מאחר והמערכת שלנו תרוץ על קוברנטיס אני אבנה מנגנון בניה לסקריפט כדי שיתאים להיות משימה מתוזמנת שם. משימה מתוזמנת על קוברנטיס זה קונטיינר שעושה את ה"משימה" שלו בעליה פעם אחת ומיד יוצא. בהתקנה של הקונטיינר הזה אנחנו נגדיר אותו בקוברנטיס בתור Job ונגיד בדיוק מתי צריך להפעיל אותו, וקוברנטיס ידאג להפעלה לפי שעון פעם ביום.

לכן בשלב הזה מספיק לי להוסיף לתיקיית query-script קובץ .dockerignore עם התוכן:

node\_modules/

וקובץ Dockerfile עם התוכן:

FROM node:17

WORKDIR /app

COPY . .

ENV NODE\_ENV=production

RUN npm install

CMD ["node", "query-weather"]

הקובץ מאוד דומה ל Dockerfile של שרת האקספרס, רק שהפעם ב CMD שלו הוא מריץ את הסקריפט ומיד מסיים. נבנה את האימג' עם:

$ docker build . -t shay/deploy-weather-to-k8s-query-script:1.0

ואז נעדכן את ה docker-compose-prod.yml כך שיכיל את התוכן הבא:

version: "3.9"

services:

weather-api-server:

image: "shay/deploy-weather-to-k8s-weather-api-server:1.0"

ports:

- "3000:3000"

environment:

DB\_HOST: db

env\_file:

- "env/k8s.env"

secrets:

- postgres-passwd

query-script:

image: "shay/deploy-weather-to-k8s-query-script:1.0"

env\_file:

- "env/k8s.env"

environment:

DB\_HOST: db

secrets:

- postgres-passwd

- weather-apikey

db:

image: "shay/deploy-weather-to-k8s-db:1.0"

env\_file:

- "env/k8s.env"

ports:

- "5432:5432"

secrets:

- postgres-passwd

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

weather-apikey:

file: ./secrets/weather-apikey

## העלאת מערכת מזג האוויר לקוברנטס - יצירת המניפסטים לקוברנטיס

יצירת קבצי ה yaml והעלאה לאוקטטו:

אני מתחיל בדיוק באותה צורה בה עבדתי על יישום ה counter, והפעם החיים שלי אפילו יותר פשוטים - כבר עשינו הרבה עבודה הכנה ובנינו גירסת פרודקשן של קובץ ההגדרות שלנו docker-compose-prod.yml. נמשיך ממנו:

1. אני מעתיק את הקובץ docker-compose-prod.yml לקובץ הגדרות חדש עבור קוברנטיס שיקרא docker-compose-k8s.yml.
2. אני מעדכן את הגירסה ל 3.8 כדי ש kompose יעבוד.
3. אני יוצר תיקיה חדשה בשם k8s, נכנס אליה ובתוכה מריץ:

$ kompose convert -f ../docker-compose-k8s.yml

WARN Service "query-script" won't be created because 'ports' is not specified

INFO Kubernetes file "db-service.yaml" created

INFO Kubernetes file "weather-api-server-service.yaml" created

INFO Kubernetes file "weather-apikey-secret.yaml" created

INFO Kubernetes file "postgres-passwd-secret.yaml" created

INFO Kubernetes file "db-deployment.yaml" created

INFO Kubernetes file "env-production-env-configmap.yaml" created

INFO Kubernetes file "postgres-data-persistentvolumeclaim.yaml" created

INFO Kubernetes file "query-script-deployment.yaml" created

INFO Kubernetes file "weather-api-server-deployment.yaml" created

והפלט נראה ממש בסדר! כל הסרביסים נוצרו, הסודות נוצרו ורק query-script לא קיבל קובץ service בגלל שלא היה לו מיפוי פורטים. האמת שגם זה הגיוני, כי query-script צריך לעבוד בתור משימה מתוזמנת ולא להקשיב לחיבורים נכנסים.

איך נראים קבצי ההגדרות:

כניסה לתיקיית k8s תראה לנו שקיבלנו קבצי deployment, service ו persistentvolumeclaim כמו בדוגמאות הקודמות, ובנוסף אליהם קיבלנו גם שני סוגים נוספים של קבצים.

הקובץ env-production-env-configmap.yaml הוא קובץ הגדרות סביבה. שימו לב לתוכן שלו:

apiVersion: v1

data:

POSTGRES\_DB: weather

POSTGRES\_PASSWORD\_FILE: /run/secrets/postgres-passwd

POSTGRES\_USER: postgres

WEATHER\_API\_KEY\_FILE: /run/secrets/weather-apikey

kind: ConfigMap

metadata:

creationTimestamp: null

labels:

io.kompose.service: db-env-production-env

name: env-production-env

הקובץ מתאים בדיוק לקובץ ה env.production שיצרתי ומכיל את כל משתני הסביבה שהיישום שלי צריך. בקוברנטיס, משתני סביבה מאוחסנים במבנה שנקרא ConfigMap.

בכניסה לקובץ deployment שמשתמש במשתני סביבה, לדוגמה לקובץ db-deployment.yaml נוכל למצוא שם בלוק כזה:

- env:

- name: POSTGRES\_DB

valueFrom:

configMapKeyRef:

key: POSTGRES\_DB

name: env-production-env

שאומר שהקונטיינר לוקח את הערך של משתנה הסביבה POSTGRES\_DB מתוך ConfigMap שנקרא env-production-env.

סוג שני של קובץ חדש שקיבלנו הוא קובץ secret. ראינו בעבודה עם docker compose שאנחנו שומרים דברים כמו סיסמאות או מפתחות API בקבצים וטוענים אותם עם בלוק secrets בקובץ docker-compose.yml. הכלי kompose הפך כל סוד לקובץ הגדרות משלו שנראה בערך כך:

apiVersion: v1

data:

postgres-passwd: bmluamEK

kind: Secret

metadata:

creationTimestamp: null

labels:

io.kompose.service: postgres-passwd

name: postgres-passwd

type: Opaque

למרות שהסיסמה נראית מוצפנת, למעשה זה בסך הכל קידוד base64. משורת הפקודה אני יכול לכתוב את הפקודה הבאה כדי לפענח אותה:

$ echo -n bmluamEK | base64 -d

ninja

בדרך כלל את קבצי ה secret לא נרצה לשמור ב git. אנחנו ניצור אותם ונעלה אותם ישירות לקלאסטר.

יצירת קובץ ingress.yaml:

אחרי שסיימתי את ההמרה אני עדיין צריך ליצור קובץ ingress כדי להגדיר איזה חיבורים נכנסים הקלאסטר שלי מעוניין לקבל. בתוך אותה תיקיית k8s צרו קובץ בשם ingress.yaml עם התוכן הבא:

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: ingress

annotations:

kubernetes.io/ingress.class: "nginx"

spec:

rules:

- host: shay.cloud.okteto.net

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: weather-api-server

port:

number: 3000

הקובץ דומה מאוד למה שיצרנו בשיעור הקודם והשינוי היחיד הוא שם הסרביס שאליו אנחנו רוצים להתחבר.

העלאה לאוקטטו:

נסיים עם העלאה לקלאסטר בענן. בתוך תיקיית k8s וודאו שאתם מחוברים להגדרות הקלאסטר עם:

$ kubectl config get-contexts

ואז הפעילו:

$ kubectl apply -f .

ניכנס למסך הניהול של הקאלסטר כדי לראות מה קרה ו ... יש בעיה. אנחנו רואים שהקונטיינרים ממשיכים למשוך מחדש את האימג' אבל לא מצליחים לעלות.

אתם מוזמנים לנסות לחפש לבד את הבעיה, או לחכות לוידאו הבא בו נגלה מה הבעיה ואיך לתקן אותה.

## העלאת מערכת מזג האוויר לקוברנטס - תיקון הקונטיינרים

תקלה 1 - קבצי הסודות לא במקום:

כדי להבין מה קורה בקונטיינרים הדבר הראשון שאנחנו רוצים לעשות הוא להוריד את האש - כלומר לקבל קונטיינר אחד מתפקד שאפשר יהיה להתחבר אליו ולראות את הודעת השגיאה.

אני בוחר לעבוד על weather-api-server ולכן אני נכנס לקובץ weather-api-server-deployment.yaml ושם יש לי את כל ההגדרות של הקונטיינר, ואני יכול לשנות חלק מההתנהגות שלו, בדיוק כמו שעשיתי עם docker compose. אני רוצה לשנות את הפקודה הראשונה שהקונטיינר מפעיל להיות sleep infinity, כדי שהקונטיינר יעלה ומיד ייכנס לשינה. אחרי זה אני אוכל להתחבר פנימה לקונטיינר ולחפש מה הבעיה ולמה השרת שלי לא נדלק.

אני מחפש בקובץ את הבלוק:

image: shay/21-weather-api-server:1.0

name: weather-api-server

ports:

- containerPort: 3000

ואחרי המפתח name מוסיף מפתח בשם command:

image: shay/21-weather-api-server:1.0

name: weather-api-server

command: ["sleep", "infinity"]

ports:

- containerPort: 3000

אני מעביר את ההגדרה החדשה לקלאסטר עם:

$ kubectl apply -f weather-api-server-deployment.yaml

ומחכה. אחרי כמה שניות אני יכול להפעיל:

$ kubectl get pods

ולראות שהפוד שמתאים ל weather-api-server פעיל, ואז להתחבר אליו עם:

$ kubectl exec -it deploy/weather-api-server -- /bin/bash

בפנים אני מנסה להפעיל את השרת בעצמי:

$ node bin/www

ומקבל את השגיאה:

root@weather-api-server-886f7cf5f-cls72:/app# node bin/www

node:internal/fs/utils:344

throw err;

^

Error: EISDIR: illegal operation on a directory, read

at Object.readSync (node:fs:723:3)

at tryReadSync (node:fs:433:20)

at Object.readFileSync (node:fs:479:19)

at Object.<anonymous> (/app/lib/db.js:3:24)

at Module.\_compile (node:internal/modules/cjs/loader:1097:14)

at Object.Module.\_extensions..js (node:internal/modules/cjs/loader:1149:10)

at Module.load (node:internal/modules/cjs/loader:975:32)

at Function.Module.\_load (node:internal/modules/cjs/loader:822:12)

at Module.require (node:internal/modules/cjs/loader:999:19)

at require (node:internal/modules/cjs/helpers:102:18) {

errno: -21,

syscall: 'read',

code: 'EISDIR'

}

Node.js v17.4.0

בדיקה בקוד מראה לי שהשגיאה הגיעה מהשורה:

const DB\_PASSWORD = fs.readFileSync(process.env.POSTGRES\_PASSWORD\_FILE, { encoding: 'utf8' }).trim();

מעניין! נלך לראות אם אנחנו מצליחים למצוא את הקובץ. תחילה אני מציג את הערך של משתנה הסביבה:

$ env | grep PASSWORD\_F

POSTGRES\_PASSWORD\_FILE=/run/secrets/postgres-passwd

ולאחר מכן מנסה להציג את תוכן הקובץ:

$ cat /run/secrets/postgres-passwd

וזה נותן לי את השגיאה:

cat: /run/secrets/postgres-passwd: Is a directory

וכאן המקום להגיד מילה על איך קוברנטיס מנהל סודות, ולמה postgres-passwd היא תיקיה ולא קובץ. נחזור לקובץ הסוד שיצרנו שנקרא k8s/postgres-passwd-secret.yaml. אני מזכיר שזה היה התוכן שלו:

apiVersion: v1

data:

postgres-passwd: bmluamEK

kind: Secret

metadata:

creationTimestamp: null

labels:

io.kompose.service: postgres-passwd

name: postgres-passwd

type: Opaque

בקוברנטיס, קובץ סוד יכול להחזיק מספר סודות. במפתח data נמצא אוביקט שמכיל כרגע את המפתח היחיד postgres-passwd, אבל תיאורטית אפשר היה לכתוב מפתחות וערכים נוספים אחריו. בשביל לתמוך בזה קובץ הסוד ממופה בקונטיינר לתיקיה, וכל סוד ספציפי יהיה קובץ. לכן בקונטיינר שלנו הסוד האמיתי שמור בקובץ /run/secrets/postgres-passwd/postgres-passwd. בואו נראה את זה באמצעות הרצת:

$ cat /run/secrets/postgres-passwd/postgres-passwd

ובאמת קיבלתי את הפלט ninja שזו הסיסמה לבסיס הנתונים.

עכשיו שאנחנו מבינים את הבעיה גם התיקון יהיה פשוט. אני צריך לעדכן את הגדרות הסביבה בקובץ ה env שלי. בשביל לא לקלקל אני מעתיק את הקובץ production.env לקובץ חדש בשם k8s.env ובתוכו מעדכן את הנתיבים לקבצים:

POSTGRES\_USER=postgres

POSTGRES\_DB=weather

POSTGRES\_PASSWORD\_FILE=/run/secrets/postgres-passwd/postgres-passwd

WEATHER\_API\_KEY\_FILE=/run/secrets/weather-apikey/weather-apikey

עכשיו אני ממשיך לעדכון הקובץ docker-compose-k8s.yml עם קובץ הסביבה החדש:

version: "3.8"

services:

weather-api-server:

image: "shay/21-weather-api-server:1.0"

ports:

- "3000:3000"

environment:

DB\_HOST: db

env\_file:

- "env/k8s.env"

secrets:

- postgres-passwd

query-script:

image: shay/21-query-script:1.0

env\_file:

- "env/k8s.env"

environment:

DB\_HOST: db

secrets:

- postgres-passwd

- weather-apikey

db:

image: "shay/21-db:1.0"

volumes:

- postgres-data:/var/lib/postgresql/data

env\_file:

- "env/k8s.env"

ports:

- "5432:5432"

secrets:

- postgres-passwd

secrets:

postgres-passwd:

file: ./secrets/postgres-passwd

weather-apikey:

file: ./secrets/weather-apikey

volumes:

postgres-data:

מייצר מחדש את קבצי ההגדרות של קוברנטיס ומעלה מחדש לקלאסטר:

$ kompose convert -f ../docker-compose-k8s.yml

$ kubectl apply -f .

המתנה קצרה והצגה מחדש של הפודים מראה שיש שיפור אבל אנחנו עדיין לא מתפקדים. זה הפלט אצלי:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

db-79cb565dfd-r2g5w 0/1 ContainerCreating 0 27s

query-script-f4f7db95f-s5d4g 0/1 Completed 2 31s

weather-api-server-5b6bc997c-87xjw 1/1 Running 0 31s

weather-api-server-886f7cf5f-cls72 1/1 Terminating 0 10m

שרת ה weather-api-server הצליח לעלות, אבל בסיס הנתונים עדיין מתרסק.

תקלה 2 - תיקיית ה data של פוסטגרס לא ריקה:

ההתרסקות השניה קלה יותר לפיתרון מאחר והפעם כבר יש לנו הודעת שגיאה בלוגים. אפשר לראות את הלוג מתוך מסך הניהול של אוקטטו או להפעיל מהמסוף:

$ kubectl logs deploy/db

ואני מקבל את השגיאה:

The files belonging to this database system will be owned by user "postgres".

This user must also own the server process.

The database cluster will be initialized with locale "en\_US.utf8".

The default database encoding has accordingly been set to "UTF8".

The default text search configuration will be set to "english".

Data page checksums are disabled.

initdb: error: directory "/var/lib/postgresql/data" exists but is not empty

It contains a lost+found directory, perhaps due to it being a mount point.

Using a mount point directly as the data directory is not recommended.

Create a subdirectory under the mount point.

אומר לי קוברנטיס, או יותר נכון אומר לי פוסטגרס - "תשמע חביבי התיקיה שאתה רוצה שאני אשמור בה את בסיס הנתונים אינה ריקה. יש בתוכה תיקיה בשם lost+found. זה בדרך כלל קורה כשאתה מנסה להשתמש בתיקיה ראשית של מערכת קבצים בתור תיקיית מידע. בבקשה תשתמש בתת-תיקיה ריקה כי אני צריך שליטה מלאה במה שמופיע שם".

והוא צודק.

בעוד שדוקר קומפוז יצר תיקיות ריקות בתור Volumes, קוברנטיס יוצר ממש מערכת קבצים שלמה ב Volume ולכן התיקיה הראשית אינה ריקה - היא כוללת מידע מנהלי של מערכת הקבצים.

ואחרי שאני מבין את הבעיה אני גם יודע איך לפתור אותה - פשוט אבקש מקוברנטיס להעביר בתור Volume איזושהי תת-תיקיה של מערכת הקבצים הראשית ולא את התיקיה הראשית. בשביל זה אני נכנס לקובץ k8s/db-deployment.yaml ושם מוצא את הבלוק:

volumeMounts:

- mountPath: /run/secrets/postgres-passwd

name: postgres-passwd

- mountPath: /var/lib/postgresql/data

name: postgres-data

ומשנה את ה mountPath השני כך שישתמש בתת-תיקיה של ה Volume באמצעות הוספת המפתח subPath. התוצאה נראית כך:

volumeMounts:

- mountPath: /run/secrets/postgres-passwd

name: postgres-passwd

- mountPath: /var/lib/postgresql/data

name: postgres-data

subPath: postgres

אני שולח את ההגדרה החדשה לקלאסטר עם:

$ kubectl apply -f .

ואחרי המתנה קצרה אני רואה שהמצב השתפר מאוד - גם בסיס הנתונים וגם השרת רצים:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

db-74644897c-qqprc 1/1 Running 0 31s

query-script-f4f7db95f-s5d4g 0/1 CrashLoopBackOff 10 30m

weather-api-server-5b6bc997c-87xjw 1/1 Running 0 30m

בנוסף אני יכול לגלוש לדף הראשי של המערכת ולקבל JSON ריק, שאומר שהשרת מצליח להתחבר לבסיס הנתונים וכרגע אין נתוני מזג אוויר שם.

תקלה 3 - ה Query Script התייאש:

האתגר האחרון שלנו לשיעור הזה הוא להעיר את ה query script כדי שיהיו לנו נתוני מזג אוויר בבסיס הנתונים. הפוד של הסקריפט הוא query-script-f4f7db95f-s5d4g, ואני יכול לראות בלוגים או במסך הניהול של אוקטטו שהוא רץ פעם אחת והסתיים, ולא מצא סיבה לרוץ שוב.

הדרך הכי קלה להריץ שוב את הסקריפט היא למחוק את הפוד. קוברנטיס ייצור את הפוד מחדש כי כבר שלחנו אליו את ה Deployment, ותזכרו שה Deployment מגדיר כמה קונטיינרים מהסקריפט הזה צריכים להיות פעילים בכל רגע נתון. ברגע שמחקתי חסר קונטיינר אחד ולכן הוא נוצר. הפקודה הבאה מוחקת את הפוד ותגרום להרצה מחדש שלו:

$ kubectl delete pod query-script-f4f7db95f-s5d4g

אחרי שנתתם לסקריפט זמן לרוץ תוכלו להיכנס לכתובת של השרת דרך הדפדפן ולראות את התוצאה ב JSON.

בשיעור הבא נלמד על משימות מתוזמנות ונראה איך לגרום לסקריפט לרוץ פעם ביום בבוקר.

## העלאת מערכת מזג האוויר לקוברנטס - הגדרת משימה מתוזמנת

משימה מתוזמנת בקוברנטיס זה שירות של הקלאסטר שבזמנים שאתם מגדירים מעלה קונטיינר, מריץ את הפקודה שלו וסוגר אותו. את הגדרת הזמנים אנחנו כותבים באותו פורמט של cron,

בשביל להפוך את ה query-script שלנו מ Deployment ל CronJob אנחנו צריכים בסך הכל לשנות כמה הגדרות ב yaml. בתור התחלה אני מוחק את ה Deployment כדי לא לבלבל:

$ kubectl delete deployment/query-script

ועכשיו אני מעדכן את הקובץ query-script-deployment.yaml באופן הבא:

1. אני מעדכן את המפתח apiVersion להחזיק את הערך batch/v1beta1. בגירסאות חדשות יותר של קלאסטר קוברנטיס (השרת) אפשר לוותר על ה beta ולכתוב פשוט batch/v1. על אוקטטו נכון למועד הקלטת הסרט גירסת קוברנטיס היא 1.20 ולכן עדיין צריכה את ה beta.
2. אני מעדכן את המפתח kind לערך CronJob.
3. אני מוחק את המפתח strategy.
4. אני משנה את הערך של restartPolicy ל Never. אין צורך להתחיל מחדש סקריפט שנכשל, ממילא בתור משימה מתוזמנת הוא ירוץ שוב עוד מעט.
5. אני מוחק את המפתח selector.
6. והשינוי הגדול ביותר - בתוך המפתח spec הראשי אני יוצר מפתח schedule שיגדיר מתי להריץ את הסקריפט, ומפתח jobTemplate שבתוכו יהיו כל ההגדרות שקודם היו ב spec הראשי.

סך הכל הקובץ אחרי כל העדכונים נראה כך:

apiVersion: batch/v1beta1

kind: CronJob

metadata:

annotations:

kompose.cmd: kompose convert -f ../docker-compose-k8s.yml

kompose.version: 1.26.1 (a9d05d509)

creationTimestamp: null

labels:

io.kompose.service: query-script

name: query-script

spec:

schedule: "\* \* \* \* \*"

jobTemplate:

spec:

template:

metadata:

annotations:

kompose.cmd: kompose convert -f ../docker-compose-k8s.yml

kompose.version: 1.26.1 (a9d05d509)

creationTimestamp: null

labels:

io.kompose.service: query-script

spec:

containers:

- env:

- name: DB\_HOST

value: db

- name: POSTGRES\_DB

valueFrom:

configMapKeyRef:

key: POSTGRES\_DB

name: env-k8s-env

- name: POSTGRES\_PASSWORD\_FILE

valueFrom:

configMapKeyRef:

key: POSTGRES\_PASSWORD\_FILE

name: env-k8s-env

- name: POSTGRES\_USER

valueFrom:

configMapKeyRef:

key: POSTGRES\_USER

name: env-k8s-env

- name: WEATHER\_API\_KEY\_FILE

valueFrom:

configMapKeyRef:

key: WEATHER\_API\_KEY\_FILE

name: env-k8s-env

image: shay/21-query-script:1.0

name: query-script

resources: {}

volumeMounts:

- mountPath: /run/secrets/postgres-passwd

name: postgres-passwd

- mountPath: /run/secrets/weather-apikey

name: weather-apikey

restartPolicy: Never

volumes:

- name: postgres-passwd

secret:

items:

- key: postgres-passwd

path: postgres-passwd

secretName: postgres-passwd

- name: weather-apikey

secret:

items:

- key: weather-apikey

path: weather-apikey

secretName: weather-apikey

status: {}

אני שולח את הקובץ לקלאסטר עם:

$ kubectl apply -f query-script-deployment.yaml

עכשיו במסך הניהול של אוקטטו אני יכול לראות את כל הג'ובים שהופעלו ואת הלוגים שלהם. כרגע ההגדרה ב schedule מריצה את הסקריפט כל דקה כדי שנוכל לראות אותו רץ. בשביל להריץ אותו פעם ביום נצטרך לכתוב:

schedule: "0 0 \* \* \*"

## משימות מתוזמנות

## בשעור זה נלמד על crontab שהוא שירות המערכת המנהל משימות מתוזמנות. משימות אלו מאפשרות הגדרת גיבוי יומי או שבועי, בדיקות שוטפות של המערכת או כל משימה אחרת שתרצו לבצע במרווחי זמן קבועים.

## הגדרת משימה מתוזמנת:

התהליך cron הוא שירות מערכת שתפקידו להריץ משימות מתוזמנות. כמו כל שירות מערכת תוכלו לוודא שפעיל ולראות יומן פעולות אחרונות מהשירות באמצעות הפקודה:

$ systemctl status cron

או לצפות ביומן הפעולות בלבד (ללא סטטוס) באמצעות הפקודה:

$ journalctl --unit=cron

לאחר שווידאתם שהשירות פעיל תוכלו להמשיך להגדרת המשימות עצמן. המשימות מוגדרות בקבצים בתוך התיקיה /var/spool/cron/crontabs/ אך למשתמשים רגילים אין גישה לתיקיה זו. כדי לצפות או לערוך את קובץ המשימות המתוזמנות שלכם תצטרכו להיעזר בתוכנית crontab. המתג e גורם לתוכנית לפתוח את קובץ המשימות המתוזמנות שלכם לעריכה:

$ crontab -e

## מבנה קובץ crontab:

מבנה הקובץ מוסבר בהערות בגוף הקובץ (או בעמוד התיעוד man 5 crontab). כל שורה שאינה מתחילה ב-# מייצגת משימה מתוזמנת ומורכבת מ-6 שדות מופרדים ברווחים:

* שעה לביצוע
* דקה לביצוע
* יום בחודש
* חודש
* יום בשבוע
* הפקודה עצמה

בכל אחד מהשדות ניתן להשתמש ב-\* כדי לציין "כל אחד" (לדוגמא כל דקה, כל שעה, כל יום) ולהוסיף סימן / כדי לציין אחד מתוך, לדוגמא כל 5 דקות מיוצג על ידי \*/5 בעמודת הדקות.

השורה הבאה תוסיף את הטקסט "Good Morning" בכל תחילת יום בשעה 00:00 של כל יום בחודש בכל חודש:

0 0 \* \* \* echo "Good Morning: " $(date) >> /home/shay/cronlog

בשדה הפקודה ניתן לכתוב כל פקודת יוניקס שהייתם כותבים משורת הפקודה ומומלץ להשתמש בנתיב אבסולוטי לפקודה. הפלט של הפקודה יישלח אליכם במייל באמצעות התיבה שהגדרתם בשעור הקודם.

להלן עוד מספר דוגמאות לשורות שניתן לרשום בקובץ crontab ומשמעותן:

# crontab format

# m h dom mon dow cmd

# Backup the server daily

0 0 \* \* \* /bin/backup.sh

# Backup the database every hour

0 \* \* \* \* /bin/backup\_db.sh

# Send email once a week

0 0 \* \* 4 /bin/send\_report.sh

# Check data every 5 minutes

0-59/5 \* \* \* \* /bin/check\_data.sh

# update modification date of file alive every 20 minutes of every even hour

0,20,40 0-23/2 \* \* \* touch alive

# run /bin/greet.sh at 00:00 on January 1st

0 0 1 1 \* /bin/greet.sh

# run /bin/play.sh at 13:13 on the 13th of every month but only if it's Friday

13 13 13 \* 5 /bin/play.sh

## .