מתכנת פשוט

קטגוריות

- כמה מילים על ... (7)
- תכנות - כללי (1)
- Angular 2 (2)
- Big Data (9)
- DevOps (1)
- docker (1)
- elastic (6)
- Firebase (1)
- Ionic (1)
- JavaScript (1)
- Machine Learning (11)
- Node (1)
- RxJS (1)
- TypeScript (2)

ארכיון הבלוג

- ▼ 2021 (2)
  - ◄ ספטמבר (1)
  - ▼ מאי (1)
    - מדריך מקיף על Docker
- ◄ 2020 (17)
- ◄ 2019 (2)
- ◄ 2018 (1)
- ◄ 2017 (9)

פרטים עלי

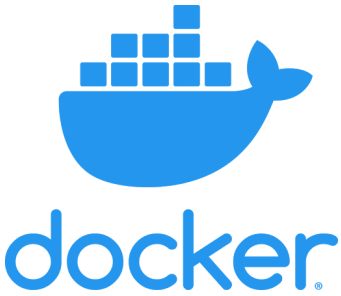
רפאל ג'אן

למה מתכנת פשוט? 1. כי אני מתכנת פשוט (לא מנהל וכו') 2. כי אני מתכנת פשוט (למה לסבך אם אפשר פשוט)

הצג את הפרופיל המלא שלי

- התחברו אליי בלינקדאין
- אודות
- ראשי

מדריך מקיף על Docker



תוכן עניינים

הקדמה

אני יודע שמקובל לכתוב פוסטים קצרים-בינוניים. העדפתי לכתוב פוסט ארוך ומקיף הכל בדף אחד - all in one.
למה זה טוב?
כי כשהכל במקום אחד זה יותר קל. אחרי שתילמדו את החומר ותירצו לחזור להיזכר בנושא מסוים, תצטרכו לחפש במקום אחד ולא לעבור בין מספר דפים ולחפש אחד אחד.
אם אתם חושבים אחרת, מוזמנים לכתוב בתגובות.

מה זה בעצם Docker?

Docker היא תוכנה ליצירת containers. היא לא התוכנה היחידה, יש עוד, אבל היא הכי פופלרית.

מה זה container?

container היא תוכנה שמאגדת בתוכה קוד יחד עם כל מה שהקוד תלוי בו, כך שהוא יוכל לרוץ מהר ובצורה אמינה בכל סביבה שבה נשים את ה-container.

בהתחלה, יש שלא מבינים מה ההבדל בין container ל-virtual machine. כשעובדים עם containers זה נהיה ברור לגמרי, אבל בכל זאת נסביר. אז באמת לשניהם יש הפרדה של משאבים משאר המערכת שבה הם נמצאים, ולשניהם יש הקצאה של זיכרון. אבל הם שונים בתכלית. container הוא דבר מאוד נייד. אני יכול לשלוח container ואני יכול לאחסן אותו במקום שמאחסן containers. ואז ממש בשניות להריץ אותו במערכת אחרת. הוא גם בד"כ הרבה יותר קטן ויעיל. לעומתו, VM שאותו בד"כ מתקינים בסביבה מסויימת ואז משתמשים בו באותה הסביבה והוא לא נועד לשליחה ואחסון.

להלן תיאור ההבדלים מהאתר הרישמי של docker:

דפים

- מדיניות פרטיות לאפליקציית ישראקאר
- מדיניות פרטיות לאפליקציית "שיעורי ישיבת הר-ברכה"
- מדיניות פרטיות לאפליקציית "פניני הלכה השלם"
- מדיניות פרטיות "JoinRide"

קצרים. הסיבה הפשוטה היא שתוכלו לחזור למאמר מדי פעם ופשוט לעשות חיפוש במאמר בלי לעבור בין דפים שונים (מדיי פעם יש חריגות ותיראו גם מאמרים נפרדים באותו תחום).
בנוסף, אני כותב מדי פעם מאמרים פחות טכניים (כמו המאמר "למה כל מתכנת צריך לכתוב בלוג?") אבל גם מאמרים אלו יהיו קשורים לתכנות.
מוזמנים לקרוא ולהגיב,
רפאל

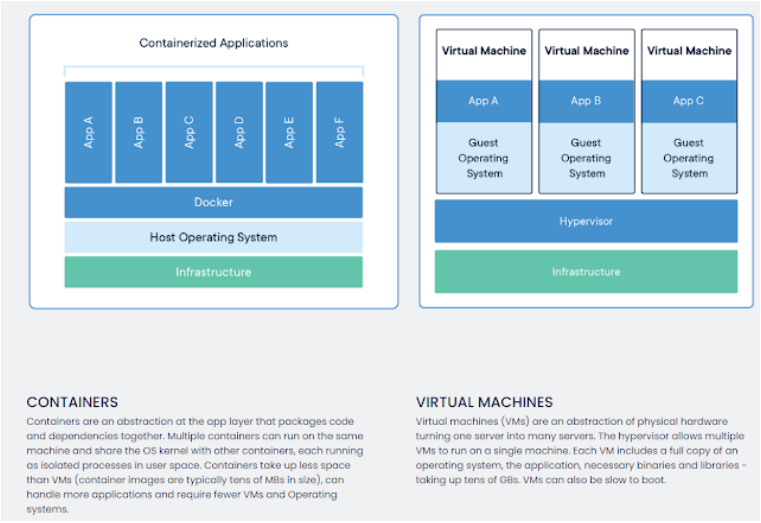
רשומות פופולריות

- מדריך מקיף על Docker
- מדריך מקיף על Airflow
- Firebase
- Angular 2 - ארכיטקטורה
- RxJS - The ReactiveX library for JavaScript
- מדריך מקיף על Elastic Stack - חלק 3 - Elasticsearch
- Angular2 - ראוטינג
- ספריית pandas - חלק 1 - מבני נתונים - Series
- ספריית NumPy - חלק 1
- ספריית pandas - חלק 3 - מבני נתונים - DataFrame - חלק ב

תוויות

- כמה מילים על ...
- תכנות - כללי
- Angular 2
- Big Data
- DevOps
- docker
- elastic
- Firebase
- Ionic
- JavaScript
- Machine Learning
- Node
- RxJS
- TypeScript

דיווח על שימוש לרעה



אחת הבעיות ש-containers באו לפתור זו בעיית ה-"אצלי זה עובד" הידועה. כיון שה-container מאגד בתוכו את כל מה שהוא צריך, אם אצלי זה עובד, זה יעבוד גם אצלך, אין פה מה להסתבך. לא צריך לשאול את המפתח של הקוד באיזה גירסה של JAVA השתמשת, ועם איזה גירסה של תוכנה מסויימת עבדת. הכל מגיע ארוז ב-container אחד ומוכן לשימוש. כאילו קיבלתם את המחשב של מי ששלח לכם את התוכנה והכל מוכן לעבודה בלי להסתבך ובלי לבזבז זמן.

התחלת עבודה

נצלול ישר פנימה וזה יעזור לנו להבין את הדברים יותר מהר.

את כל העבודה שלי עם docker אני עושה על גירסה 19.03.13 ועובד במחשב עם מערכת הפעלה לינוקס (ubuntu 18.04) אבל ברוב הדברים לא אמור להיות הבדל גם אם עובדים על windows כיון שבסופו של דבר הפקודות הם פקודות של docker.

אז קודם כל צריך להתקין docker. אני לא אסביר את זה כי זה די בסיסי וגם עלול להשתנות במשך הזמן, אז כדאי פשוט להיעזר באינטרנט לשם כך.

Image לעומת Container

כשעוסקים ב-docker יש שני מושגים בסיסיים, image ו-container
כשאנחנו רוצים לבנות container חדש אנחנו כותבים קובץ שנקרא Dockerfile (בכוונה עם D גדולה) ובו מתואר איך ה-container שלנו ייבנה. מה-Dockerfile הזה אנחנו יוצרים image ע"י הפקודה docker build. את ה-image אפשר לשנע ממקום למקום. לשלוח אותו למישהו או להעלות אותו ל-DB של docker images.
אפשר לומר שה-image הוא כמו כונן קשיח שמותקנות עליו התוכנות שאנחנו צריכים, והוא מוכן לשימוש. אבל כל עוד הוא לא דלוק אי אפשר להשתמש בו.

כשאנחנו רוצים להשתמש ב-image אנחנו משתמשים בפקודה docker run שמייצרת container ע"י image. אפשר לומר שזה מקביל להדלקה של הכונן הקשיח. עכשיו שהוא דלוק אפשר לעבוד איתו ולהיכנס אליו.
זה ההבדל בין image ל-container וחשוב להבין את זה. זה עוזר בהמשך להבנה של כל מיני נושאים.

הפעולה הארוכה היא בד"כ פעולת ה-docker build שמייצרת את ה-image ואילו פעולת ה-docker run מהירה מאוד יחסית.

המשל שכתבתי לגבי הדיסק הקשיח, מסביר בצורה טובה את העניין אבל הוא לא מדוייק (בכל זאת זה רק משל) כי מ-image אחד אני יכול להרים כמה containers שאני רוצה במקביל, ולכל container יהיה את הסביבה של עצמו. זה לא כמו כונן קשיח משותף.

בנוסף ל-image יש עוד כמה שימושים. למשל image יכול לשמש כתשתית ל-image אחר. כל Dockerfile מתחיל במילה FROM ולאחריה שם ה-image שעליו הוא מבוסס. ז"א שעל פי רוב, image לא יתחיל מאפס (למרות שזו גם אפשרות שנדבר עליה בהמשך) אלא יהיה מבוסס על image אחר, למשל על image של מערכת הפעלה כמו הפקודה הזו:

```
FROM ubuntu:18.04
```

עכשיו נעסוק ב-Dockerfile שהוא הבסיס ליצירת image.

Dockerfile - המתכון ליצירת image

אפשר להתסכל על Dockerfile כעל מתכון ליצירת image שבו כתובות כל ההוראות כדי ליצור את ה-image.
נתחיל עם דוגמה פשוטה:

```
FROM ubuntu:18.04
RUN apt-get update && apt-get -y install gdb
ENTRYPOINT ["/bin/bash"]
```

בשורה הראשונה ביצירת ה-image הזה אנחנו מתבססים על image שנקרא ubuntu עם תג 18.04. בעצם כל image מבוסס על image אחר. מקובל לקרוא לו ה-base image. במקרה שלנו אנחנו מתחילים מ-base image שמכיל את מערכת ההפעלה ubuntu בגירסה 18.04.
בשורה השניה, אנו משתמשים בפקודת RUN, שמריצה פקודות בתוך ה-container. במקרה הזה בהתחלה מעדכנים את apt-get (למי שלא מכיר, מדובר על כלי של לינוקס שנועד להתקנת תוכנות). ולאחר מכן מתקינים gdb (תוכנת debugger). היינו יכולים לכתוב את 2 הפקודות האלו בשתי שורות נפרדות כל אחת עם RUN אחר, אבל זה היה פחות יעיל. בהמשך נסביר למה.
בשורה השלישית, משתמשים בפקודת ENTRYPOINT שמריצה פקודות ב-command line של ה-container. במקרה הזה היא

מריצה /bin/bash שזה אומר שנקבל טרמינל מסוג bash.

איך משתמשים ב-Dockerfile?

כמו שכתבתי לעיל, ה-Dockerfile הוא רק המתכון ליצירת image. כדי להשתמש בו נשתמש בפקודה של docker. כל הפקודות של docker מתחילות במילה docker ולאחר מכן שם הפקודה. כדי לבנות image נשתמש בפקודה docker build. לצורך הדוגמה הנוכחית, צריך להיות באותו path שבו נמצא ה-Dockerfile שלנו. ואז נכתוב את הפקודה הבאה:

```
docker build -t my-app:1.0 .
```

שימו לב לנקודה בסוף הפקודה.
הצורה הכללית של הפקודה הזו היא:

```
docker build -t image_name:tag_name path_to_Dockerfile
```

עכשיו נסביר את הפקודה שאנחנו משתמשים בה. בעצם אחרי הדגל t- אנחנו נותנים ל-image שם ואז נקודותיים ואז שם של תג. במקרה שלנו ל-image יקראו my-app והתג יהיה 1.0. התג עוזר לנו ליצור גרסאות שונות לאותו image. לא חייב לתת תג, אפשר גם לתת רק שם ל-image בלי תג ואז docker בצורה אוטומטית ישתמש בתג שנקרא latest.
בסוף הפקודה צריך לתת את ה-path של ה-Dockerfile. במקרה שלנו אנחנו נמצאים באותה ספריה שבה נמצא ה-Dockerfile ולכן כתבתי רק נקודה, שמשמעותה היא המיקום הנוכחי.
docker יחפש תמיד קובץ בשם Dockerfile. אם אנחנו רוצים להשתמש בקובץ אחר למשל Dockerfile.debug אז נשתמש בדגל f- בצורה הבאה:

```
docker build -f Dockerfile.debug -t my-app .
```

בואו נראה מה התוצאה של הפקודה הזו:

```
(base) rafael@myubuntu:~/learning$ docker build -t my-app:1.0 .
[+] Building 29.0s (6/6) FINISHED
 => [internal] load build definition from Dockerfile                    0.2s
 => => transferring dockerfile:      131B                               0.0s
 => [internal] load .dockerignore                                       0.2s
 => => transferring context: 2B                                         0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04         1.6s
 => CACHED [1/2] FROM
docker.io/library/ubuntu:18.04@sha256:538529c9d229fb55f50e6746b119e899775205d62c0fc1b7e679b30d
02ecb6e8                                                               0.0s
 => [2/2] RUN apt-get update && apt-get -y install  gdb                25.2s
 => exporting to image                                                  1.9s
 => => exporting layers                                                 1.9s
 => => writing image sha256:c7f332fd3daf7a105f7089ee8601d889591cc0f2467d738966f12f29882df8c3
 0.0s
 => => naming to docker.io/library/my-app:1.0                           0.0s
```

לאחר ש-docker סיים לבנות לנו את ה-image נבדוק מה קיבלנו. כדי לראות את כל ה-images שלנו נשתמש בפקודה:

```
docker images
```

ונקבל את הפלט הזה:

```
REPOSITORY     TAG    IMAGE ID       CREATED          SIZE
my-app         1.0    c7f332fd3daf   8 minutes ago    221MB
```

ניתן לראות פה את שם ה-image, ואת התג שלו. בנוסף docker נותן לו id ייחודי. אפשר גם לראות מתי הוא נוצר ומה הגודל שלו.

מה קרה אם אני יוצר image עם אותו שם?
אם אני משנה את התג, אז אין שום בעיה. למשל אם אני אשתמש בפקודה הזו:

```
docker build -t my-app .
```

כיון שעכשיו לא נתתי תג, אז כמו שהסברתי לעיל docker ייתן את התג latest. ואז נקבל את הרשימה הבאה:

```
REPOSITORY     TAG    IMAGE ID       CREATED          SIZE
my-app         1.0    c7f332fd3daf   11 minutes ago   221MB
my-app         latest c7f332fd3daf   11 minutes ago   221MB
```

אמנם ברשימה יש לנו שני images אבל לפי ה-id image ניתן לראות שמדובר על אותו image. כי docker מזהה שבאמת זה אותו image אז הוא אמנם מוסיף ברשימה עוד image כי נתנו תג חדש אבל לפי ה-ID אנחנו מבינים שזה בדיוק אותו image.

ואם עכשיו אריץ שוב את הפקודה:

```
docker build -t my-app .
```

יווצר עוד image עם בדיוק אותו שם my-app ואותו תג latest, שימו לב מה docker עושה עכשיו.
אם לא שיניתי את ה-Dockerfile הוא חכם מספיק לדעת שהוא בונה את אותו ה-image. ואפילו ברשימת ה-images אנחנו נראה שהוא לא נוצר עכשיו אלא נראה את זמן יצירת ה-image המקורי כי זה בעצם אותו image.

Dangling images

עכשיו נמחק את ה-image שנקרא my-app:1.0 עם הפקודה הבאה:

```
docker rmi my-app:1.0
```

ולכן עכשיו ברשימה יש לנו רק image אחד:

```
REPOSITORY     TAG    IMAGE ID       CREATED          SIZE
my-app         latest c7f332fd3daf   11 minutes ago   221MB
```

נשנה את ה-Dockerfile שיראה כך:

```
FROM ubuntu:18.04
RUN apt-get update && apt-get -y install gdb && apt-get -y install nano
ENTRYPOINT ["/bin/bash"]
```

הוספתי פקודה להתקנת תוכנה שנקראת nano שזה editor פופולארי בלינוקס.
ועכשיו כשנשתמש בפקודה:

```
docker build -t my-app .
```

ונבדוק איזה images יש לנו, נקבל:

```
REPOSITORY     TAG    IMAGE ID        CREATED           SIZE
my-app         latest 69d1063130ca    2 days ago        222MB
<none>         <none> c7f332fd3daf    2 days ago        221MB
```

כשיוצרים image עם אותו שם ואותו תג, ה-image הישן נקרא dangling image, בעברית dangling זה מתנדנד. ההיגיון אומר שאם יצרת image עם אותו שם ואותו תג כנראה שאתה כבר לא צריך את ה-image הישן ולכן הוא "מתנדנד" בין חיים למוות או משהו כזה. כבר אין לו שם ותג אבל יש לו ID.

אם רוצים למחוק את כל ה-dangling images אפשר להשתמש בפקודה הבאה:

```
docker image prune
```

יצירת container מ-image או במילים אחרות הרצת image

לאחר שיצרנו image אפשר להשתמש בו כדי ליצור ממנו container. כדי לעשות את זה נשתמש בפקודה הבאה:

```
docker run -it image_name:tag
```

גם במקרה הזה אם לא נשתמש ב-tag אז docker יניח שה-tag הוא latest. הדגל it- הוא קיצור ל-interactive terminal וזה אומר שיהיה לנו טרמינל פעיל ל-container שנוכל להריץ בו פקודות בתוך ה-container.
במקרה שלנו נשתמש בפקודה הבאה:

```
docker run -it my-app
```

ברגע שנריץ את ה-container אז docker יצור איתו ויתן לו גם שם (בד"כ שם מוזר) וגם ID.
כדי לראות את כל ה-containers שרצים נשתמש בפקודה:

```
docker ps
```

ונירא משהו כזה:

```
CONTAINER ID   IMAGE   COMMAND      CREATED         STATUS         PORTS  NAMES
93fbdbcbaa79   my-app  "/bin/bash"  18 seconds ago Up 17 seconds         awesome_satoshi
```

ה-container שיצרנו קיבל את השם היפהפה awesome_satoshi וה-ID שלו הוא 93fbdbcbaa79.
כדי לעצור את ה-container אפשר להשתמש בפקודה הבאה:

```
docker stop container_id
```

במקרה שלנו הפקודה תהיה:

```
docker stop 93fbdbcbaa79
```

אם יש לנו טרמינל פעיל ל-container (כי כשהרצנו אותו השתמשנו ב-it-) אפשר פשוט לרשום בטרמינל exit.
צריך לדעת שהפקודה הזו לא מוחקת את ה-container אלא רק עוצרת אותו. לא נראה אותו עכשיו ברשימה של docker ps כי הוא מראה רק containers שרצים, אבל אם נוסיף את הדגל a- ניראה אותו:

```
docker ps -a
```

כדי להריץ container שעצרנו אפשר להשתמש בפקודה:

```
docker start container_id
```

בכל הפקודות שהשתמשתי ב-container_id אפשר גם להשתמש ב-name של ה-container.
בנוסף, כשיוצרים container אפשר גם לקבוע לו את השם כך שיהיה נוח לעבוד איתו. לצורך כך נשתמש בדגל name בצורה הבאה:

```
docker run -it --name rafael_app my-app
```

עכשיו שם ה-container יהיה rafael_app. ניתן לראות זאת ע"י docker ps.

הורדת image מ-docker repository

עד כה, ראינו בקצרה איך יוצרים image ע"י docker build ולאחר מכן איך משתמשים בו ליצירת container ע"י docker run.
עכשיו נראה איך אפשר להוריד container מוכן מתוך מחסן של containers.
המחסן הרישמי של containers נקרא docker hub והוא נמצא בכתובת https://hub.docker.com. פה ניתן לחפש ולמצוא את רוב ה-containers הרישמיים של תוכנות פופולאריות. למשל ניתן למצוא שם את ubuntu ואת postgres ועוד אינספור containers של תוכנות. ניתן גם להעלות לשם container שלנו.

כדי להוריד container נשתמש בפקודה docker pull. אם לא נציין מאיפה להוריד, אז הדיפולט יהיה מ-docker hub.

למשל אם נירצה להוריד container של mongodb נחפש את mongodb ב-docker hub ונגיע ל-https://hub.docker.com/_/mongo נלחץ על הטאב של Tags ושם ניראה את כל ה-images הזמינים להורדה. צריך לבחור image שמתאים למערכת ההפעלה של המחשב שבו אנחנו רוצים להשתמש ב-container הזה. בד"כ התג של כל image מתאר את סוג ה-container. למשל אם נשתמש בפקודה הבאה:

```
docker pull mongo:windowsservercore-ltsc2016
```

אנחנו נוריד את ה-image של mongo שמיועד ל-windows server. במקרה הזה לא מצוין מאיפה להוריד את ה-image ולכן docker ינסה להוריד מ-docker hub.

מלבד docker hub, יש עוד הרבה חברות שמציעות שירות של docker repository. למשל לאמזון יש את ECR ולגוגל יש את GCR ויש עוד רבים.

דוגמה: אם למשל, אנחנו רוצים להריץ סקריפט של python ויש לנו מחשב ללא התקנה של python ואנחנו לא רוצים להתקין עליו python. אפשר בקלות להוריד image של python ולהשתמש בו להריץ את הסקריפט. כדי להוריד את ה-image נשתמש בפקודה:

```
docker pull python
```

כיון שלא ציינו איזה תג להוריד, הדיפולט יהיה latest. הוא כמובן ירד מ-docker hub.

אם למשל הסקריפט הפשוט שלנו נראה כך:

```
for x in range(6):
  print(x)
```

ואז ניתן להריץ את הסקריפט בצורה הבאה:

א. פקודת run עם it- עבור interactive terminal

ב. חיבור volume - חיבור של זיכרון מהמחשב המארח (host) אל ה-container. נרחיב על כך בהמשך. במקרה הזה חיברנו את המיקום הנוכחי (pwd) ב-host אל הספריה /usr/src/myapp/ ב-container.

ג. קביעת ה-working directory ב-container. זו הספריה שנגיע אליה בעליה של ה-container.

ד. שם ה-image שבו אנו משתמשים ליצירת ה-container

ה. הפקודה שאנו רוצים להריץ ב-container

הפלט יראה כך:

```
$ docker run -it -v "$PWD":/usr/src/myapp -w /usr/src/myapp python python my-script.py
0
1
2
3
4
5
```

יצירת image מ-container

לעיל הסברתי איך ליצור container מ-image וזה מה שנעשה ברוב הפעמים. אבל לפעמים יש צורך הפוך, ליצור image מ-container.

למשל, אם הורדתם image מסוים מ-docker hub ואתם רוצים להוסיף משהו ל-image הזה, למשל להתקין משהו בתוכו. הדרך הנכונה היא לשנות את ה-Dockerfile שלו, ושם להוסיף פקודת התקנה של מה שאנו צריכים.

אבל לא תמיד יש לנו את ה-Dockerfile של ה-image. אם אנחנו מורידים image מ-docker hub יהיה לנו את ה-image ונוכל לראות אותו ברשימה כשנריץ את הפקודה docker images, אבל לא יהיה לנו את ה-Dockerfile שלו (אגב, ב-docker hub אפשר גם לראות את ה-Dockerfile אבל יש מקרים אחרים שבהם הקובץ הזה לא יהיה זמין לנו).

לצורך זה יש את הפקודה `docker commit`. אז איך עושים את זה?

קודם נריץ את ה-container, וניכנס לתוכו בעזרת הפקודה `docker exec` (אם יש צורך להיכנס אליו כ-root אפשר להשתמש ב-`docker exec -u root`).

בתוך ה-container נתקין את מה שאנחנו צריכים. לאחר מכן, נצא מה-container (לא נסגור אותו, רק נצא ממנו כשהוא עדיין פעיל) ואז נבדוק מה ה-id שלו, ע"י הפקודה `docker ps`. ועכשיו נוכל ליצור image חדש מה-container הזה שיהיה מבוסס על ה-image המקורי בתוספת מה שהתקנו בתוכו, ע"י הפקודה:

```
$ docker commit container_id new_image_name
```

לדוגמה, אם השתמשנו ב-image של הכלי שנקרא airflow, והתקנו בתוכו את ה-CLI של AWS, וה-id של ה-container הוא a123456bc7d8 אז נכתוב את הפקודה הבאה:

```
$ docker commit a123456bc7d8 apache/airflow:2.1.2.awscli
```

השם שנתנו ל-image הוא שם ה-image המקורי בתוספת של awscli כך שיהיה קל לזכור מה יש ב-image הזה. נוכל לראות את ה-image החדש ע"י הפקודה `docker images` ולהשתמש בו, ומעכשיו הוא מכיל בתוכו גם התקנה של awscli.

הרצת container ברקע והתחברות אליו

לפעמים יש לנו צורך להריץ container ברקע ואנחנו לא מעוניינים להיכנס אליו. במקרה כזה נוכל להריץ אותו עם הדגל d- עבור detached.

```
docker run -d image_name
```

ונוכל לבדוק אם הוא רץ ע"י הפקודה docker ps. הפקודה הזו גם תראה לנו מהו ה-id container שלו.

אם בכל זאת לאחר שה-container רץ אנו רוצים להיכנס אליו, נוכל להשתמש בפקודה:

```
docker exec -it docker_id /bin/bash
```

וכך נקבל טרמינל בתוך ה-container.

פקודות Dockerfile נפוצות

ה-Dockerfile הוא המתכון ליצירת ה-image. יש לו סט פקודות רחב ואת כולם ניתן לראות באתר הרישמי. בפיסקה הזו נסביר על הפקודות שלדעתי יותר נפוצות ושימושיות.

FROM <image:tag>

כל Dockerfile מתחיל ב-FROM. הפקודה הזו בעצם מבססת את ה-image שלנו על גבי image אחר (שנקרא base image). כל מה שיש ב-image שהתבססנו עליו יהיה עכשיו גם ב-image שלנו. בד"כ מתבססים על image של מערכת הפעלה מסויימת או תוכנה אחרת ידועה. סביר מאוד להניח שאם ניגד ל-Dockerfile של אותו image שאנחנו מתבססים עליו גם הוא מתבסס על image אחר.
ניתן גם להתחיל ממש מאפס אם נתבסס על FROM scratch ואז נוכל להעתיק פנימה ל-container מה שאנחנו רוצים בלי שיהיה כלום בהתחלה.

WORKDIR </path/to/workdir>

הפקודה הזו קובעת את ה-working directory לפקודות RUN, CMD, ENTRYPOINT, COPY, ADD שבאים אחריה עד סוף ה-Dockerfile או עד ששוב נשתמש ב-WORKDIR. אם ה-WORKDIR הבא הוא יחסי, הוא יתייחס ל-WORKDIR הקודם.
דוגמה:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

ההדפסה של pwd תהיה /a/b/c/
אם ה-path שציינו לא קיים, הוא יווצר.

LABEL <key>=<value> <key>=<value> ...

מוסיף metadata ל-image בצורה של key=value. פקודה זו נועדה בשביל לעשות לנו סדר. אנחנו יכולים בעזרתה להוסיף מידע ל-image בשביל להדפיס אותו בהמשך או בשביל להבין מה יש ב-image בלי לחקור את ה-Dockerfile שלו. כדי לראות מה ה-labels שיש ב-image מסוים ניתן להשתמש בפקודה

```
docker inspect image_id
```

היא מדפיסה הרבה מידע וגם את ה-labels.

ARG <name>[=<default value>]

הפקודה ARG מאפשרת לנו להגדיר ארגומנט ולשלוח אליו ערך בפקודת docker build בצורה הבאה:

```
docker build --build-arg <varname>=<value>
```

למשל אם הגדרנו ב-Dockerfile ארגומנט בצורה הבאה:

```
ARG private_name
```

ובנינו את ה-image כך:

```
docker build --build-arg private_name=Rafael
```

אז הערך "Rafael" יכנס לארגומנט private_name ואפשר להשתמש בו אחרי השורה של ההגדרה ARG. ניתן גם לתת לו ערך דיפולטיבי, למשל:

```
ARG private_name=Yosi
```

ואז אם לא נקבע ערך ב-docker build הערך הדיפולטיבי יכנס לארגומנט.
שימו לב: הארגומנט מוכר רק כחלק מתהליך ה-build הוא לא מוכר בתוך ה-image שנוצר. אם רוצים להעביר מידע לתוך ה-image אפשר להשתמש ב-ENV.

ENV <key>=<value> ...

הפקודה הזו מגדירה משתנה סביבה (environment variable). המשתנה יהיה מוגדר גם בתוך ה-container.
השימוש הוא בצורה הבאה:

```
ENV file_name=my_file.txt
```

ניתן גם להגדיר כמה משתנים באותה פקודה:

```
ENV file_name=my_file.txt my_folder=src
```

אם רוצים להשתמש ברווחים צריך להשתמש בגרשיים או בלוכסן:

```
ENV my_name="Bil Gates" my_friend=Steve\ Jobes
```

COPY <src>... <dest>

העתקה של קבצים/ספריות מהמחשב לתוך ה-container. למשל בפקודה הבאה:

```
COPY myfile.txt /home
```

הוא יעתיק את הקובץ myfile.txt מהספריה הנוכחית במחשב לתוך ספריית home ב-container. ניתן גם להעתיק קבצים מספריות פנימיות במיקום הנוכחי שלי. למשל בצורה הזו:

```
COPY tmp/myfile.txt /home
```

אבל (מסיבות של security) לא ניתן להעתיק ממיקום שהוא מחוץ לספריה הנוכחית שלי, למשל כך:

```
COPY ../myfile.txt /home
```

במקרה הזה נקבל שגיאה כזו:

```
COPY failed: Forbidden path outside the build context
```

ניתן גם להשתמש ב-* עבור העתקה של קבצים רבים עם אותה התחלה:

```
COPY myfile* /home
```

במקרה הזו הוא יעתיק את כל הקבצים שמתחילים ב-myfile.
או ב-? עבור החלפה של אות אחת:

```
COPY my?txt
```

במקרה הזה הוא יעתיק את כל הקבצים שמתחילים ב-my לאחר מכן כל סימן ואז txt.
ניתן גם להעתיק כמה קבצים בפקודה אחת, אבל כולם יועתקו למיקום אחד:

```
COPY file1 file2 /home/
```

אם מעתיקים כמה קבצים (בצורה מפורשת או ע"י *) חובה שהיעד יהיה ספריה ויש לכתוב לוכסן לאחריו. בדיוק כמו בדוגמה האחרונה, בשונה מהדוגמאות הקודמות.

ADD <src>... <dest>

דומה ל-COPY אבל הוא תומך בעוד שתי אפשרויות. דבר ראשון הוא מאפשר העתקה מ-URL ולא רק קבצים וספריות מקומיים.
ודבר שני, הוא מאפשר לחלץ קובץ tar מהמחשב ישירות ל-container.

RUN

הרצה של פקודה בתוך ה-container. לדוגמה, הפקודה הבאה תיצור ספריה בשם my_folder בתוך ה-container:

```
RUN mkdir my_folder
```

אם יש לנו צורך להריץ כמה פקודות, עדיף לשרשר אותם ב-RUN אחד ולא לעשות RUN עבור כל פקודה. הסיבה לכך היא שעבור כל RUN ה-docker מייצר שיכבה נוספת ב-image. כל שיכבה תופסת מקום ולוקחת זמן. ניראה דוגמה כדי להדגיש את ההבדל.
אם אני צריך:

- ליצור ספריה
- להעתיק לתוכה קבצים (בדוגמה שלנו נעתיק את ספריית lib שגודלה הוא 12MB)
- (לעשות עליהם איזושהי עבודה)
- למחוק את הקבצים

אני יכול לעשות את זה בשתי דרכים.
דרך ראשונה, פחות יעילה, בכמה פקודות:

```
RUN mkdir my_folder
RUN cp -r /lib/* /my_folder


# do some work
RUN rm -rf my_folder
```

דרך שניה, יעילה, בפקודה אחת:

```
RUN mkdir my_folder && cp -r /lib/* /my_folder && rm -rf my_folder
```

נבדוק עכשיו את גודל ה-images שנוצרו:

```
REPOSITORY              TAG      IMAGE ID       CREATED              SIZE
learn_docker_one_run    latest   ededf53f059d   3 seconds ago        63.1MB
learn_docker_multi_run  latest   f28018162ff5   About a minute ago   75.2MB
```

כפי שרואים, קיבלנו הבדל של 12MB בגודל ה-images. למרות שבשניהם בסופו של דבר קיבלנו את אותה תוצאה סופית. לכן עדיף לשרשר ככל שניתן.

CMD

הפקודה הזו בדרך כלל תהיה אחרונה ב-Dockerfile. היא קובעת מה תהיה הפקודה הדיפולטיבית שה-container יריץ ברגע שמעלים אותו.
לדוגמה:

```
CMD ["my_executable","param1","param2"]
```

ברגע שנרים את ה-container הוא יריץ את my_executable עם הפרמטרים param1 param2.
הרבה פעמים אנו בונים image עבור הרצה של service מסוים. ע"י שימוש ב-CMD נוכל לבנות image שיריץ אתה מה שאנחנו רוצים ישר בעלייה.
מותר להשתמש רק ב-CMD פעם אחת. ואם כותבים יותר מאחד אז רק האחרון יצא לפועל.
ה-CMD יכול לעבוד בשיתוף פעולה עם ENTRYPOINT, כדי להבין את כל האפשרויות כדאי לקרוא את הפירוט שיש באתר הרישמי.

ENTRYPOINT

דומה ל-CMD. לא ניכנס פה להבדלים ביניהם, ועל כך מומלץ ללכת שוב לאתר הרישמי. נציין רק כמה נקודות ביחסים בין הפקודות האלו:

- ב-Dockerfile צריך שיהיה לפחות CMD או ENTRYPOINT אפשר גם שניהם.
- נשתמש ב-ENTRYPOINT כשאנחנו רוצים שה-container ישמש כ-executable.
- נשתמש ב-CMD כדי להעביר ארגומנטים דיפולטיבים עבור ENTRYPOINT. או עבור פקודה שאנו רוצים שתצא לפועל ברגע שה-container שאנו עובדים איתו עולה.

הטבלה הבאה מהאתר הרישמי מראה את התוצאה של קומבינציות שונות של CMD / ENTRYPOINT:

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

אפשרויות פופולאריות ב-docker build

כרגיל, כדי לדעת הכל תצטרכו ללכת לאתר הרישמי. כדי לדעת חלק קטן אבל מאוד מעשי הישארו עימנו.
אז הפקודה הכי בסיסית היא:

```
docker build -t image_name .
```

הפקודה הזו תחפש להשתמש באופן דיפולטיבי בקובץ שנקרא Dockerfile. אבל מה עושים אם אצלנו יש שני קבצים, אחד נקרא Dockerfile.debug והשני Dockerfile.release?
לצורך הזה נשתמש באפשרות f:

```
docker build -f Dockerfile.release -t image_name .
```

לפעמים תהליך ה-build לא מדפיס הכל, במיוחד אם אנו משתמשים ב-buildkit (ראה על כך בהמשך פוסט זה). כדי לראות את כל הפלט של תהליך ה-build נשתמש ב-progress:

```
docker build --progress=plain -t image_name .
```

בתהליך יצירת ה-image לפעמים נוצרים container-ים נוספים שלא נצרכים בסוף. כדי למחוק אותם בסיום התהליך ולחסוך זיכרון

במחשב נשתמש ב-rm:

```
docker build --rm -t image_name .
```

כדי לקבוע ערך של ARG נשתמש ב-build-arg:

```
docker build --build-arg MY_ARG=some_value  -t image_name .
```

אפשרויות פופולאריות ב-docker run

כהרגלנו בקודש נזכיר שכדי לדעת הכל תצטרכו ללכת לאתר הרישמי. ונעבור לחלק המעשיו.
הפקודה הבסיסית היא:

```
docker run -it image_name
```

הדגל it נותן לנו interactive terminal. זה אומר שה-container יתחבר לנו ל-terminal ונוכל לראות אותו מריץ משהו או לכתוב בתוכו פקודות, תלוי איך בנינו את ה-container. זה נקרא ריצה ב-foreground.
ואם אנחנו רוצים שה-container ירוץ במנותק מהטרמינל שלנו מה שנקרא detached mode, אז נשתמש ב-d.

```
docker run -d image_name
```

ואז הוא ירוץ ברקע.
לאחר שאנחנו יוצאים מ-container, עדיין מערכת הקבצים שלו נשמרת. זה דבר טוב לטובת debugging. אבל אם אנחנו מריצים container ב-foreground ואנחנו יודעים שאין לנו צורך במערכת הקבצים שלו לאחר שסגרנו אותו, אז אפשר מראש לקבוע שמערכת הקבצים שלו לא תישמר ע"י שימו ב-rm:

```
docker run --rm -it image_name
```

כדי לתת ערך עבור ENV שהגדרנו ב-Dockerfile נשתמש ב-env:

```
docker run --env my_file="/home/config.txt" -it image_name
```

יש מקרים שבהם אנחנו צריכים לשנות את ה-ENTRYPOINT שנקבע ב-Dockerfile. למשל אם ה-container מריץ אוטומטית service מסוים ועכשיו אנחנו רוצים טרמינל (למשל bash) בתוך ה-container לפני שהוא מריץ את ה-service. לצורך כך נשתמש ב-entrypoint:

```
docker run -it --entrypoint /bin/bash image_name
```

בצורה דומה, ניתן פשוט להוסיף בסיום הפקודה את מה שאנו רוצים להריץ למשל כך:

```
docker run -it image_name /bin/bash
```

בדרך כלל כל container רץ עם PID namespace נפרד. מה שמאפשר לו להתנתק מה-PID (process ID) שהמחשב שלנו כבר הקצה לצרכים אחרים וכך ה-container יכול לקבוע לעצמו כל מספר PID שהוא רוצה אפילו 1. במקרה שאנו רוצים לשתף PID namespace בין ה-container למחשב שלנו נשתמש ב-pid:

```
docker run --pid=host -it image_name
```

ואם צריך לשתף PID namespace בין שני containers אז נכתוב זאת כך:

```
docker run --pid=container:container_name_id -it image_name
```

כאשר container_name_id הוא ה-id או השם של ה-container שאנו רוצים להיות איתו באותו PID namespace.

איך מוחקים image/container ואיך מוחקים הכל

ה-image בד"כ שוקל לא מעט ולכן הרבה פעמים אנחנו רוצים למחוק אותו מהמחשב שלנו כשאין לנו בו צורך.
כדי למחוק image נשתמש בפקודה:

```
docker rmi image_name/id
```

כשיוצרים container מ-image גם הוא תופס מקום. ואפילו אם אנחנו עוצרים את ה-container הוא עדיין תופס מקום. אפשר לראות את כל ה-containers גם אלו שכבר נעצרו ע"י הפקודה:

```
docker ps -a
```

כדי למחוק container נשתמש בפקודה:

```
docker rm container_id
```

גם הפקודה rm וגם rmi אמורות לפנות מקום במחשב שלנו, ובכל זאת נתקלתי הרבה פעמים במצב שהם אמנם מחקו את ה-image/container מהרשימה אבל לא התפנה באמת מקום במחשב. כדי לפנות מקום לגמרי אפשר להשתמש בזהירות בפקודה:

```
docker image prune
```

כשתשתמשו בה תקבלו את האזהרה הבאה:

```
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N]
```

כפי שלמדנו לעיל, dangling images הם images שכבר לא ממש בשימוש. למשל שיצרנו image עם אותו שם ל-image שקיים אז ה-image הישן לא נמחק אבל אין לו כבר שם אלא רק ID. אם נשיב y לאזהרה אז ימחקו כל ה-dangling images ואז באמת נראה שהתפנה מקום.

פקודה יותר קטלנית היא:

```
docker system prune
```

הפקודה הזו כבר תיתן לנו את האזהרה הזו:

```
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - all dangling build cache

Are you sure you want to continue? [y/N]
```

אז היא מוחקת גם containers וגם images. בנוסף היא מוחקת cache שתופס לא מעט זיכרון. וגם networks שלא בשימוש. אז היא אמנם מוחקת הרבה אבל יחסית בעדינות. רק מה שנראה לא ממש בשימוש. הפקודה הזו מאוד שימושית כשלומדים docker

ואם אתה ממש רוצה ללכת עד הסוף ולא ממש אכפת לך שהכל יימחק אז יש הנשק הקטלני ביותר שמוחק ללא רחמים:

```
docker system prune -a
```

פקודה זו כבר לא מוחקת רק מה שלא בשימוש, אלא הכל. את כל ה-images ואת כל ה-containers ובאמת מנקה באופן יסודי.
כשנשתמש בה נקבל את האזהרה הזו:

```
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all images without at least one container associated to them
  - all build cache

Are you sure you want to continue? [y/N]
```

אז הפקודה הזו מוחקת את כל ה-images שלא בשימוש, לא רק dangling images. ובנוסף את כל ה-build cache. שני אלא בד"כ מהווים את המסה העיקרית של הזיכרון שנתפס ע"י docker.
להשלמת הנושא אפשר לגשת לאתר הרישמי.

כמה מילים על docker networks

ה-docker יוצר networks שבהם רצים ה-containers. כדי לראות את כל ה-networks נשתמש ב:

```
docker network ls
```

כדי ליצור network נכתוב:

```
docker network create network_name
```

כדי להריץ container על network מסוים נשתמש באפשרות של net:

```
docker run --net network_name -it image_name
```

ואם אנחנו רוצים להריץ container על רשת של container אחר ולא אכפת לי מה שם הרשת, אז פשוט נכתוב:

```
docker run --net container:container_name_id -it image_name
```

docker compose

מדובר על כלי שמאפשר לנו לכתוב פקודות בצורה מובנת וכך לא נצטרך לכתוב פקודות ארוכות. הפקודות נכתבות בקובץ yaml (שימו לב שה-indentation או הזחה בעברית, חשובה מאוד בפורמט yaml).

דוגמה (לקוחה מהקורס המצוין של נענע. בכלל נענע מסבירה מעולה, מומלץ לחפש תכנים בערוץ שלה):
אנחנו רוצים להריץ container של mongodb יחד עם container של mongo-express באותה רשת. בשיטה הרגילה נעשה זאת כך:

```
docker network create mongo-network

docker run -d -p 27017:27017 -e MONGO_INITDB_ROOT_USERNAME=admin -e
MONGO_INITDB_ROOT_PASSWORD=password --net mongo-network --name mongodb mongo

docker run -d -p 8081:8081 -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin -e
ME_CONFIG_MONGODB_ADMINPASSWORD=password -e ME_CONFIG_MONGODB_SERVER=mongodb --net mongo-
network --name mongo-express mongo-express
```

כפי שרואים מדובר על פקודות ארוכות ומורכבות.
לעומת זאת ב-compose נכתוב קובץ בשם mongo.yaml כזה:

```
version: '3'
services:
  mongodb:
    image: mongo
    ports:
      - 27017:27017
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
  mongo-express:
    image: mongo-express
    ports:
      - 8080:8081
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password
      - ME_CONFIG_MONGODB_SERVER=mongodb
```

בדוגמה זו רואים בצורה ברורה כמה ה-compose יכול לסדר את העניינים. זה עדיין לא קצר, אבל זה מאוד מסודר וקריא.
בנוסף, כשמריצים מספר images ב-compose אחד docker ייצור network משותף עבורם.

אז כתבנו קובץ yaml, איך משתמשים בו?
כדי להריץ את ה-images שבקובץ yaml משתמשים בפקודה הבאה:

```
docker-compose -f mongo.yaml up
```

ואם רוצים לרוץ במצב של detached נוסיף דגל d- בסוף הפקודה.

כדי לעצור את כל ה-containers ולמחוק את הרשת שנוצרה נכתוב:

```
docker-compose -f mongo.yaml down
```

כרגיל, נגענו רק בקצה הקרחון. אם נושא זה מעניין אותך האתר הרישמי מחכה לך.

כמה מילים על docker volumes

ל-docker יש מערכת קבצים משלו שמופרדת מהמחשב שלנו. ברגע שיצאנו מה-container בגדול כל מה שהיה שם נמחק (לא בדיוק, כמו שראינו לעיל לגבי docker run -rm-- אבל לא נהוג לחפש קבצים של docker שנסגר). אם אנחנו לא רק מריצים containers אלא גם רוצים לעבוד בתוך container, אנחנו מעוניינים שכל העבודה שלנו תישמר. לצורך כך יש את האפשרות לקשר בין מערכת הקבצים של ה-container לזו של המחשב. הדבר הזה נקרא volume.

הפקודה הבאה:

```
docker run -v /path/in/host:/path/in/container -it image_name
```

תריץ container שהספריה path/in/container אצלו מחוברת לספריית /path/in/host במחשב שלנו. וכך גם לאחר שסגרנו את ה-container נוכל לראות במחשב את הקבצים שהיו במיקום הזה ב-container.
סוג נוסף הוא ה-named volume. יש לו שם אבל לא מציינים במפורש את המיקום שלו:

```
docker run -v name:/path/in/container -it image_name
```

השם של ה-volume צריך להיות ייחודי באותו מחשב. docker יקבע את המיקום של ה-volume על המחשב.
סוג נוסף של volume הוא ה-anonymous volume וניתן להשתמש בו בצורה הבאה:

```
docker run -v /path/in/container -it image_name
```

הסוג הזה דומה ל-named volume בכל שאר הפרטים. docker יקבע לו את השם ואת המיקום. הסוג הזה כנראה היה שמיש לפני ש-docker הכניסו את ה-named volume. במצב הנוכחי, לא מצאתי לו שימוש.

כדי לקבל מידע על ה-volume נשתמש ב-inspect:

```
docker volume inspect volume_name
```

וכדי למחוק volume נשתמש ב-rm:

```
docker volume rm volume_name
```

אפשר גם ליצור volume בלי קשר להרצה של container ע"י הפקודה:

```
docker volume create volume_name
```

העתקה מה-container אל המחשב ולהפך

לפעמים יש לנו צורך להעתיק קובץ או ספריה מהמחשב אל ה-container. ראינו שאפשר לעשות את זה כחלק מה-Dockerfile ע"י פעול COPY או ADD. אבל כשמעתיקים ב-Dockerfile אנחנו בעצם מעתיקים לתוך image ולא לתוך container.
כדי להעתיק לתוך container נשתמש בפקודה הבאה:

```
docker cp <src_path> <container>:<dest_path>
```

כאשר:
src_path הוא הקובץ או הספריה (כולל המיקום שלה) במחשב שלנו שרוצים להעתיק לתוך ה-container.
container הוא ה-ID או ה-name של ה-container.
dest_path הוא המיקום ב-container שאליו רוצים להעתיק.

באופן דומה, כדי להעתיק מה-container למחשב נשתמש בפקודה הבאה:

```
docker cp <container>:<src_path> <dest_path>
```

כאשר:
src_path הוא הקובץ או הספריה (כולל המיקום שלה) ב-container שמשם רוצים להעתיק למחשב.
container הוא ה-ID או ה-name של ה-container.
dest_path הוא המיקום במחשב שאליו רוצים להעתיק.
אגב, אפשר לעשות פעולות אלו גם על container כשהוא רץ וגם כשהוא לא רץ.

תקשורת עם ה-container

ניתן לעבוד בתוך container וניתן גם להריץ container ולתקשר איתו מבחוץ. למשל אם אנחנו רוצים להריץ שירות מסוים כמו מסד נתונים אפשר שהוא ירוץ ב-container ונתקשר איתו מהמחשב שלנו. לצורך כך נגדיר את הפורטים הפנימי שה-container מאזין עליו, ואת הפורט החיצוני במחשב שלנו שדרכו נדבר עם ה-container. נעשה את זה ע"י שימוש באפשרות p בצורה הבאה:

```
docker run -p 5432:80 image_name
```

כאן הגדרנו שפורט 5432 במחשב שלנו מחובר (bind) לפורט 80 בתוך ה-container.

בניית image בשיטת multi-stage

במקרים רבים אנו מעוניינים ליצור סוגים שונים של container עבור אותו פרוייקט. למשל אחד עבור release ואחד עבור debug.
אפשרות אחת היא להשתמש במספר קבצי Dockerfile ולתת לכל אחד סיומת שונה. למשל Dockerfile.release ו-Dockerfile.debug.
אפשרות שניה היא לכתוב Dockerfile בשיטת multi-stage.
בשיטה הזו בקובץ Dockerfile יחיד אפשר להגדיר מספר images ולציין בפקודה docker build איזה מהסוגים אנו רוצים לבנות.
הכי קל להסביר את זה ע" דוגמה:

```
# Stage 0
FROM ubuntu:18.04 AS builder
RUN mkdir my_project

COPY my_project my_project

# Stage 1
FROM builder AS debug
RUN cd my_project && cmake -DCMAKE_BUILD_TYPE=Debug && make my_cool_project
ENTRYPOINT ["/bin/bash"]

# Stage 2
FROM alpine:latest AS release
COPY --from=builder my_project my_project
RUN cd my_project && cmake -DCMAKE_BUILD_TYPE=Release && make my_cool_project
CMD ["/my_project/my_cool_project"]
```

מה שיפה בזה, שהתחביר פה מאוד ברור. אפילו לפני שנסביר לגבי זה, כבר אפשר לראות את הכוונה מאחורי זה.
כל stage מתחיל במילה FROM. בדוגמה הזו יש שלוש stages:
בראשון אנחנו מכינים את התשתית.
בשני אנחנו בונים את הפרוייקט עבור debug.
בשלישי אנחנו בונים את הפרוייקט עבור release.

הסבר מפורט:
בשלב הראשון התבססנו על image של ubuntu, יצרנו ב-container ספרייה בשם my_project והעתקנו לתוכה את התוכן של

ספריית my_project מהמחשב שלנו.
בשלב השני התבססנו על השלב הראשון (זה אומר שגם השלב הזה מבוסס על ubuntu), נכנסנו לספריית my_project, ובנינו שם את הפרוייקט שלנו עבור debug. ואז קבענו שכשמריצים את הפרוייקט שלנו הוא נכנס לטרמינל מסוג bash.
בשלב השלישי התבססנו על image של alpine שזה סוג של images מאוד קטנים שנועדו עבור release/production. העתקנו מהשלב של ה-builder את הספרייה my_project. נכנסנו לספריה my_project ובנינו אותה עבור release. ואז קבענו שכשמריצים את הפרוייקט שלנו הוא פשוט מריץ את התוכנה שבנינו.

בשלב הראשון, אין בכלל פקודת CMD/ENTRYPOINT זה אומר שהשלב הזה לא אמור להיות שלב סופי שלנו אלא רק שלב ביניים בדרך לשלב אחר. שימו לב לתוספת ה-AS שבכל פקודת FROM. ה-AS מאפשר לנו לתת שם לשלב הזה וכך בשלבים הבאים להתייחס לשלב הזה ע"י השם שנתנו לו. לא חובה לתת שם. אם לא ניתן שם נוכל להתייחס אליו לפי המספר שלו. הלשבים ממוספרים החל מאפס. השלב הראשון הוא שלב 0 השני 1 וכן הלאה.
אם לא הייתי נותן שם לשלב ה-builder הייתי כותב ב-COPY של שלב ה-release את הדבר הבא:

```
COPY --from=0 my_project my_project
```

אז לא חובה לתת שמות לשלבים, אבל זה מאוד מומלץ.
קודם כל זה הרבה יותר קריא.
ודבר שני, אם אנחנו משנים את סדר ה-stages זה לא שובר לנו את ה-Dockerfile, אבל אם התייחסנו לשלבים ע"י מספרים, זה יכול להרוס את מה שתכננו לבנות.

כדי לבנות שלב מסוים נשתמש באפשרות target בצורה הבאה:

```
docker build --target=release -t container_name .
```

אם לא נציין איזה target אנחנו רוצים לבנות אז הדיפולט יהיה ה-stage האחרון שב-Dockerfile. בדוגמה לעיל, ה-release.

אז ראינו שע"י multi-stage ניתן להשתמש ב-Dockerfile אחד עבור כל צורות הבנייה שהפרוייקט צריך. יתרון נוסף שיש ל--multi stage הוא לאפשר לנו להקטין את גודל ה-image. בשלבים ראשונים נבנה את הפרוייקט, ובשלב הסופי נעתיק רק את מה שצריך עבור הריצה (runtime). ואז ב-image הסופי לא יהיה לנו את כל התוצרי ביניים שנוצרו במהלך בניית הפרוייקט ולא יהיה לנו את קבצי המקור. עוד על נושא זה ראה להלן בפיסקה על "הקטנת גודל ה-image".

buildkit לעומת legacy build

לאחרונה (החל מגירסה 18.09), docker הוציאו מנגנון build חדש שנקרא buildkit. מאז שהוא יצא קוראים למגנון הישן legacy build. נכון לעכשיו הוא נתמך רק בלינוקס.
למנגנון החדש יש הרבה יתרונות, מוזמנים לחפש על כך ברשת. וכאן יש פוסט טוב על הנושא. היתרונות החשובים שאני ראיתי הם:

- מנגנון cache משופר
- בנייה בצורה מקבילית של פקודות בלתי תלויות
- בנייה חכמה של multi-stage
- פלט נוח יותר

שלושת הנקודות הראשונות גורמות לבנייה מהירה יותר של images. זה משתנה מאוד לפי המקרה הספציפי אבל במקרים רבים מדובר על קיצור זמן משמעותי.
במקרים של multi-stage יש הבדל משמעותי בין buildkit ל-legacy build. ה-build-kit בונה רק את השלבים שצריך עבור ה-target שאנו רוצים לבנות. למשל אם יש לנו עשרה stages ואנחנו מעוניינים לבנות את 7 stage. הוא ילך ל-7 stage ויראה באילו stages הוא תלוי (גם מבחינת FROM וגם מבחינת COPY --from). ואז הוא יילך לאותם stages ויראה במי הם תלויים וכן הלאה. וכך הוא יבנה רק את ה-stages הנצרכים עבור ה-target שבחרנו. לעומת זאת ב-legacy build הוא יבנה את כל ה-stages עד ל-target שבחרנו. אם בחרנו את האחרון הוא יבנה פשוט הכל. ודבר כזה יכול לקחת הרבה יותר זמן.

ה-buildkit מתוכנן להיכנס בגרסאות הבאות כמנגנון דיפולטיבי עבור בנייה של images. בינתיים אפשר לקנפג את docker להשתמש בו בצורה הבאה:

```
sudo vim /etc/docker/daemon.json
```

בתוך הקובץ הזה נוסיף לתוך הסוגריים הראשיים את הטקסט הבא:

```
"features": { "buildkit": true }
```

ואז צריך לעשות restart ל-daemon של docker:

```
sudo systemctl restart docker
```

אפשר כמובן לחזור שוב ל-legacy build אם רוצים.

הקטנת גודל ה-image

במקרים רבים אנחנו מגיעים ל-image גדול מאוד. אני אישית ראיתי images שעברו את ה-30GB. גם אם המצב שלכם טוב יותר, עדיין אנחנו תמיד נעדיף image כמה שיותר קטן מהסיבות הבאות:

- ככל שה-image יותר קטן כך קל יותר לנייד אותו
- הוא תופס לנו פחות מקום במחשב
- אם אנחנו מאחסנים אותו באיזשהו repository בתשלום, ככל שהוא קטן יותר הוא חוסך לנו כסף
- מבחינת security כשה-image מכיל דברים שאין לנו בהם צורך זה מעלה את הסיכון שלנו (ה-surface attack גדל)

אז איך ניתן להקטין את ה-image?
יש על זה הרבה מאמרים ברשת. להלן ההמלצות שלי:

- תשתמשו ב-multi-stage כך שב-image עבור הפיתוח יש כל מה שצריך לפיתוח אבל ב-image שהולך ל-production יש רק מה שצריך ל-production.
- במקום שה-production יהיה מבוסס על ubuntu כדאי שהוא יהיה מבוסס על alpine או על distroless.
- השתמשו בפחות שכבות ב-image. כמו שראינו לעיל, במקרים רבים אפשר להשתמש ב-RUN יחיד במקום במספר RUNs.

לגבי הנקודה השניה, כמו שראינו בפיסקה של multi-stage יש images שנקראים alpine והם מאוד מאוד קטנים וכדאי להשתמש בהם ל-production. בנוסף יש images שנקראים distroless שמיוצרים ע"י גוגל. ראיתי הסברים שהם עדיפים ל-production מבחינת security. הם בד"כ קצת יותר גדולים מ-alpine אבל עדיין מאוד קטנים. שימו לב, ב-distroless אין אפילו טרמינל, כך שהוא נועד ממש לריצה ולא לעבודה בתוכו.

שמירת image כקובץ tar/gz וטעינת image

לפעמים יש צורך לשלוח image במייל או בדרך אחרת. לצורך כך נוח לשמור את ה-image כקובץ tar או אפילו כקובץ gz מוקטן.
כדי לשמור image כקובץ tar נשתמש בפקודה הבאה:

```
docker save image_name > pick_name.tar
```

כדי לשמור image כקובץ tar.gz נשתמש בפקודה:

```
docker save image_name | gzip > pick_name.tar.gz
```

מהצד השני, כדי לטעון image שנשמשר כ-tar או gz נשתמש בפקודה:

```
docker load image_name < pick_name.tar
```

אני מעריך שפקודות אלו יצטרכו קצת שינוי כדי לעבוד על windows.

המלצות security

להלן רשימת המלצות מבחינת security. לא ניכנס לעומק ההסברים, רק נתאר בקצרה ומי שמעוניין להרחיב יגגל את הנושא המעניין אותו:

1. אל תריצו container כ-root. תמיד השתמשו בפקודה USER ב-Dockerfile כדי לבטל הרשאות שמשתמש הקצה לא צריך והשאירו רק הרשאות שאתם צריכים. אם ה-container צריך הרשאות root לפעולות מסוימות השתמשו ב-gosu במקום sudo. כאן יש דוגמה שמתחילה ב-USER של root ב-Dockerfile ומשנה למשתמש אחר (במקרה הזה ל-jenkins) בסוף הקובץ entrypoint.sh ע"י הפקודה gosu.
2. השתמשו רק בהרשאות הנצרכות, ולכן אל תשתמשו ב-

```
docker run --privileged ...
```

במקום זאת השתמשו ב:

```
docker run --cap-drop=ALL --cap-add=CAP_<needed capability> ...
```

3. השתמשו ב-base image המינימאלי הדרוש.כמו שדיברנו לעיל, במקום להשתמש ב-ubuntu השתמשו ב-distroless.
4. שים לב שאין ssh ב-container שלך.
5. אל תכניס מידע סודי ל-image אפילו אם הוא מאוכסן ב-repository מאובטח.
6. תחתום את ה-image שאתה בונה. למשל, אפשר להשתמש ב-Docker Content Trust (DCT).
7. סרוק את ה-images שלך מבעיות נפוצות (common vulnerabilities and exposures (CVEs)). ישנם כלים שמיועדים לכך.
8. שים באותה רשת רק את האפליקציות שצריכות לתקשר ביניהן ולא יותר מזה. מה שנקרא zero trust network. כל container יכול להיות במספר רשתות. כך שאם A צריך לדבר עם B ו-C אבל B לא מדבר עם C אתה צריך 2 רשתות. אחת ל-A עם B ושניה ל-A עם C.
9. השתמש ב-read-only containers - כדי למנוע מהאקרים להכניס דברים לא רצויים לתוך ה-container שלך. השימוש פשוט מאוד:

```
docker run --read-only ...
```

אם ה-container שלך צריך את היכולת שיכתבו לתוכו יש לך שתי אפשרויות:

א. השתמש בדגל tmpfs-- ב-docker run לאפשר כתיבה רק במקום ספציפי ב-container. אפשר להשתמש בדגל הזה כמה פעמים כדי לאפשר כתיבה במספר מקומות
ב. השתמש ב-volume שמחובר למיקום מסוים ב-container שרק אליו אתה רוצה לכתוב.

DIVE - כלי לחקירת images

כלי open-source מצוין שנתקלתי בו ומאפשר לחקור docker image.

בעזרתו ניתן לראות בכל שלב מה-Dockerfile אילו קבצים נוספו/נמחקו/שונו ומה הגודל של כל קובץ. דרך מצויינת לחקור image כדי להבין מה קרה בכל שלב.

מנווטים בו ע"י החיצים ועוברים בין צד שמאל לימין עם כפתור Tab. אפשר גם לעשות חיפוש ע"י ctrl+f.

הפרוייקט הזה נמצא כאן. יש שם את כל ההסברים איך להתקין אותו ולהשתמש בו אז לכו נא לשם.

סיום

אני חושב שזה המדריך הארוך ביותר שכתבתי. רק המדריך של elastic-search יכול להתחרות בו אבל הוא מפוצל לכמה חלקים אז המדריך הזה מנצח.

אם הגעתם עד לכאן ותירגלתם את הדברים, לדעתי אתם כבר חצי מומחים ל-docker.

תותחים, נתראה בפוסט הבא

אהה ואם אהבתם, תכתבו משהו למטה... שאני אדע שמישהו בכלל קורא.

at מאי 27, 2021

Labels: docker

62 תגובות:

אנונימי 27 במאי 2021 בשעה 15:07

איזו השקעה מדהימה!
עדיין לא עברתי על הרוב, אך ניכר שהשקעת המון!

השב

תשובות

רפאל ג'אן 28 במאי 2021 בשעה 0:38

אכן לקח המון זמן לכתוב הכל. תודה!

השב

naftali10027 במאי 2021 בשעה 16:06

ואאו

ניסיתי הרבה פעמים להתחיל ללמוד דוקר וכל פעם הפסקתי רק בגלל הכמות אפשרויות שסיחררה אותי

קראתי ועכשיו מרגיש טיפה יותר מתמצא בתוך ים האפשרויות

תודה. מדריך מעולה.

השב

תשובות

רפאל ג'אן 28 במאי 2021 בשעה 0:38

תודה נפתלי

השב

Unknown27 במאי 2021 בשעה 16:29

אחד המדריכים הטובים שקראתי בעברית בנושא! מקצועי ומעמיק בדיוק ברמה שצריך לפני שניגשים לדוקומנטציה הכבדה של דוקר.

השב

תשובות

רפאל ג'אן 28 במאי 2021 בשעה 0:37

תודה רבה. משמח לשמוע

השב

Yonii27 במאי 2021 בשעה 20:27

וואו, מעולה.

בא לי בדיוק בזמן😉

השב

תשובות

רפאל ג'אן 28 במאי 2021 בשעה 0:37

מעולה! שמח לשמוע

השב

Kobiba28 במאי 2021 בשעה 1:54

אהבתי את המדריך מאוד!! מושקע ביותר וממש for dummies כמו שאני אוהב.

אפשר קישור למדריך לElastic-search?

השב

תשובות

רפאל ג'אן 28 במאי 2021 בשעה 3:01

תודה אחי. הינה הקישור http://meta-pa.blogspot.com/2020/09/elastic-stack-1.html

זה החלק הראשון מתוך 4. בחלק העליון של האתר בצד ימין יש רשימה של כל המאמרים. אתה יכול למצוא שם את כל החלקים.

השב

Asaf28 במאי 2021 בשעה 4:28

כתבתי מסמך עבור המפתחים שלנו בין השאר, בנושא Docker. אולם, לאחר שראיתי את מה שכתבת, ראיתי לנכון להוסיף קישור לכאן. עבודה יפה ומקיפה!

השב

תשובות

רפאל ג'אן 28 במאי 2021 בשעה 4:31

לכבוד הוא לי

השב

Senan Zedan28 במאי 2021 בשעה 13:59
כתבת מעולה, תודה רבה
השב

תשובות


רפאל ג'אן 29 במאי 2021 בשעה 11:39
תודה :-)

השב

Unknown29 במאי 2021 בשעה 9:35
תודה רבה על ההשקעה
השב

תשובות


רפאל ג'אן 29 במאי 2021 בשעה 11:39
בשמחה :-)

השב

דוד 29 במאי 2021 בשעה 17:04
תודה !
השב

תשובות


רפאל ג'אן 29 במאי 2021 בשעה 23:20
בשמחה

השב

Unknown30 במאי 2021 בשעה 0:57
מלך! תודה רבה לך, איזה מדריך מצוין פשוט סגרת פינה בנושא שאין בו הרבה מדריכים כאלו פשוטים להבנה
השב

תשובות


רפאל ג'אן 30 במאי 2021 בשעה 1:39
תודה רבה אחי. ממש משמח לשמוע שזה עוזר

השב

Unknown30 במאי 2021 בשעה 2:40
רפאל תודה רבה לך!
המדריך הזה פשוט מדהים!
השב

תשובות


רפאל ג'אן 30 במאי 2021 בשעה 5:00
וואוו תודה רבה. ממש משמח לשמוע

השב

אנונימי 30 במאי 2021 בשעה 8:53
כתיבה מצויינת!!!
תודה על ההשקעה
השב

תשובות


רפאל ג'אן 30 במאי 2021 בשעה 11:45
תודה לך אנונימי יקר

השב

Ben8830 במאי 2021 בשעה 20:20

מדריך נפלא!
המון אנשים חייבים לך הרבה שעות :)

השב

תשובות

רפאל ג'אן 30 במאי 2021 בשעה 22:50

תודה רבה בן. זה שזה עוזר זה מספיק לי :-)

השב

אנונימי 1 ביוני 2021 בשעה 0:30

שאפו... אחלה מדריך. עזר לי מאוד

השב

תשובות

רפאל ג'אן 1 ביוני 2021 בשעה 0:46

שמח לשמוע

השב

Dvir1 ביוני 2021 בשעה 0:32

מעולה, תמשיך כך !

השב

תשובות

רפאל ג'אן 1 ביוני 2021 בשעה 0:46

ב"ה נמשיך. תודה

השב

אנונימי 2 ביוני 2021 בשעה 5:32

נא תקן את הכותרת: מדריך מקיף על Docker --> מדריך מקיף על Docker
לדעתי יש מקום לציין באותה נשימה גם את PODMAN ואולי להציג מה זה CRI - container runtime interface

השב

תשובות

רפאל ג'אן 2 ביוני 2021 בשעה 5:38

תודה אנונימי יקר, תיקנתי. ותודה על ההצעה

השב

Unknown9 ביוני 2021 בשעה 3:02

עבודה מרשימה רפאל. מאוד נהנתי לקרא את הסיכומים שלך בשפה ברורה ונהירה וכמובן...בעברית. בדיוק חיפשתי משהו בנושא KIBANA אך לא מצאתי. אולי זה יהיה הנושא הבא שלך?

השב

תשובות

רפאל ג'אן 9 ביוני 2021 בשעה 3:05

תודה לך. שמח מאוד לשמוע. האמת שכתבתי על KIBANA, תוכל למצוא זאת כאן https://meta-pa.blogspot.com/2020/09/elastic-stack-4-kibana.html

השב

אנונימי 9 ביוני 2021 בשעה 4:36

תודה רבה!
אתה תיכתב בדפי ההיסטוריה כמי שמנגיש את העולמות האלה לדוברי השפה העברית...!
תודה!

השב

תשובות

רפאל ג'אן 9 ביוני 2021 בשעה 4:48

וואו, תודה רבה. כמה שיותר יעזור לעם ישראל יותר טוב ב"ה

השב

אנונימי 11 ביוני 2021 בשעה 4:48

מדהים! תודה רבה

השב

תשובות

רפאל ג'אן 11 ביוני 2021 בשעה 6:20

תודה לך אנונימי יקר

השב

אנונימי 20 ביוני 2021 בשעה 20:57

.Nice article. Thanks for sharing
Online Training

השב

תשובות

רפאל ג'אן 20 ביוני 2021 בשעה 23:42

Thanks Ram

השב

Unknown24 ביוני 2021 בשעה 6:33

רפאל מדריך מעולה! ממש תודה רבה,
בן

השב

תשובות

רפאל ג'אן 24 ביוני 2021 בשעה 6:57

תודה בן. שמח לשמוע

השב

אנונימי 27 ביוני 2021 בשעה 12:42

אלוף תודה רבה על מדריך מפורט יישר כוח

השב

תשובות

רפאל ג'אן 27 ביוני 2021 בשעה 14:21

תודה אחי/אחותי שמח מאוד לעזור

השב

ארז 4 ביולי 2021 בשעה 22:44

תודה רבה!
המדריך הזה עזר לי מאוד

השב

תשובות

רפאל ג'אן 10 ביולי 2021 בשעה 11:35

תודה ארז, שמח שעזרתי

השב

אנונימי 5 בספטמבר 2021 בשעה 12:30

תודה רבה רבה.
ניסיתי להבין קצת דוקר והרגשתי קצת אבוד בהתחלה מרוב כל התסבוכת שכתובה בדוק הרשמי.
פישטת את הכל בצורה נפלאה. אלוף!!!!

השב

תשובות

רפאל ג'אן 18 באוקטובר 2021 בשעה 1:32

תודה אנונימי יקר. שמח לשמוע שעזרתי :-)

השב

ערן 18 בספטמבר 2021 בשעה 22:49

תודה רבה על מדריך מעולה
מגיע עם ידע בdocker ובהחלט המדריך חידד לי דברים והרחיב את הידע
תשקול לפתוח ערוץ ביוטיוב :)

השב

תשובות

רפאל ג'אן 18 באוקטובר 2021 בשעה 1:33

תודה ערן, אם זה עוזר גם למביני עניין בתחום זה עוד יותר מושלם

השב

אנונימי 2 באוקטובר 2021 בשעה 15:17

עזר לי ברמותתתתתת
ברור מובן ומקיף
תודה ענקית
ה יברך אותך

השב

תשובות

רפאל ג'אן 18 באוקטובר 2021 בשעה 1:33

תודה אנונימי יקר. שמח לשמוע שעזרתי, ה' יברך גם אותך ב"ה :-)

השב

Unknown12 באוקטובר 2021 בשעה 5:03

עושה סדר, הרבה דוגמאות בהירות וטובות. תודה על ההשקעה!

השב

תשובות

רפאל ג'אן 18 באוקטובר 2021 בשעה 1:33

תודה unknown12 יקר. שמח לשמוע שעזרתי :-)

השב

משה 10 בדצמבר 2021 בשעה 13:41

תודה רבה, מדריך מעולה!

השב

תשובות

רפאל ג'אן 9 במרץ 2022 בשעה 5:16

תודה לך משה

השב

אנונימי 19 בדצמבר 2021 בשעה 10:07

אלופפפ! הסבר מושלם!!!
מצפה לבלוג יותר ארוך ומפורט בנושא :) אתה מסביר כל כך טוב

השב

תשובות

רפאל ג'אן 9 במרץ 2022 בשעה 5:16


שמח לשמוע. תודה לך אנונימי

השב

אנונימי 11 בפברואר 2022 בשעה 5:22

תודה רבה, מדריך מטורף!!!

השב

תשובות


רפאל ג'אן 9 במרץ 2022 בשעה 5:17

תודה! בזכותך ובזכות שאר האנשים הטובים פה, הוא הגיע למקום הראשון בגוגל כשמחפשים מדריך דוקר

השב

Unknown25 במרץ 2022 בשעה 6:19

מדריך מצויין! תודה רבה!

השב

תשובות


רפאל ג'אן 6 באפריל 2022 בשעה 7:43

תודה לך Unknown25. שמח שעזרתי. וואו יש פה קהילה של מה של unknowns.

השב

הזן תגובה

רשומה חדשה יותר דף הבית רשומה ישנה יותר

הירשם ל- תגובות לפרסום (Atom)

רשומות פופולריות

- מדריך מקיף על Docker
  תוכן עניינים הקדמה מה זה בעצם Docker? מה זה container? התחלת עבודה Image לעומת Container Dockerfile - המתכון ליצירת image Dangling images יצ...
- מדריך מקיף על Airflow
  תוכן העניינים מה זה Airflow? קצת היסטוריה יש כלים מתחרים? אז למה דווקא Airflow? מה זה DAG? הארכיטקטורה של Airflow התחלת עבודה הרצת משימות ב...
- Firebase
  הקדמה Firebase הוא סטאטראפ (שבהתחלה נקרא Envolve) שהוקם בשנת 2011 ע"י Andrew Lee ו- James Tamplin וגם רכש ע"י גוגל ב...
- Angular 2 - ארכיטקטורה
  אנגולר זה Framework לבניית אפליקציות תוך שימוש ב- HTML ו- JavaScript או TypeScript . אנגולר זה פרויקט קוד פתוח שמפותח ומתוחזק ע"י...
- RxJS - The ReactiveX library for JavaScript
- מדריך מקיף על Elastic Stack - חלק 3 - Elasticsearch
  זה החלק השלישי במדריך על Elastic Stack. בחלק הראשון עסקנו ב-filebeat. בשני ב-logstash ובחלק הזה נעסוק בכלי השלישי בשרשרת - El: elasticsearch...

- Angular2 - ראוטינג
  המקור להסברים כאן הוא מהאתר הרישמי: https://angular.io/docs/ts/latest/guide/router.html ברוב המימושים של routing הדבר הראשון שצריך...

- ספריית pandas - חלק 1 - מבני נתונים - Series
  מבוא ספריית pandas היא python package שמאפשרת עבודה מהירה, גמישה ואינטואיטיבית עם מבני נתונים רלציונים. יוצרי הספריה התכוונו ליצור ס...

- ספריית NumPy - חלק 1
  מבוא ספריית NumPy היא ספריית python מאוד נפוצה. ספריה זו נועדה לסייע בעבודה עם מערכים רב ממדיים. אובייקט NumPy הוא טבלה של אלמנטים, ב...

- ספריית pandas - חלק 3 - מבני נתונים - DataFrame - חלק ב
  הוספת עמודות על בסיס עמודות אחרות - פונקציית assign פונקציית assign מאפשרת הוספת עמודות על בסיס עמודות אחרות. הפונקצייה לא משנה את ה-df...