Є монолітні додатки (Трьохтирові, виконуються в одному процесі. Бідь-які зміни потребують перезборки моноліта. Охоплюють багато контекстів додатку.), а їх протилежність - мікросервіси.

**Мікросервіси** - стиль архітектури розробки додатків в вигляді набору незалежно розгортуваних сервісів кожен з яких прцює в власному процесі, а комунікація між ними відбувається по HTTP чи RPC. Самі по собі ці сервіси можуть бути написані на різних мовах і мати різну структуру збереження даних.

Є популярний вираз, що окремий мікросервіс може бути настільки великим, щоб міг повністю поміщатися в голові одного розробника.

Коли відбуваються зміни в якійсь частині моноліта, потрібно перезбирати моноліт. Коли якісь зміни відбуваютьсяв певному сервісі, просто перезбирається окремий сервіс.

Мінус мікросервісів в комунікації через, наприклад, RPC. Такий обмін інформацією є повільніший ніж в моноліті, який виповняється в одному процесі. При використанні мікросервісів приходиться пересікати межі процесів мікросервісів.

Сервіс може мати декілька процесів. Наприклад, процес додатка і процес БД який використовує тільки цей додаток. Тільки першим процесом має запускатися якийсь менеджер процесів.

Закон Конвея - коли команди розміщають логіку на тих слоях, до яких мають доступ (UI, MiddlewareDevelopers, DBAs).

Мікросервісний підхід призводить до утоворення кросс-технологічних команд, оскільки сервіси розбиваються відповідно до їхнього бізнес функціоналу. Такі сервіси можуть містити повний стек технологій для реалізації цього бізнес функціоналу, в тому числі інтерфейс, сховище даних і будь-які зовнішні взаємодії.

**Bounded Context** - термін для опису сутностей, які тісно взяємодіють між собою, тобто виконують якусь бізнес-функцію і можуть існувати автономно. Поняття походить з DDD (Domain Driven Design) який пропонує ділити складну структуру на декілька контекстів і мапити відносини між ними.

**Polyglot Persistence** - підхід, коли для мікросервісів можна вибирати різні технології для вирішення одинакової задачі. Наприклад, можна використовувати різні види БД для окремих сервісів.

Сервіси одного додатка можуть знаходитися на різних машинах і доменних іменах. Використовується Cross Origin Resource Sharing таким чином можна робити запити на різні машини і потім об’єднувати їх.

**X-Forwarded-For** заголовок потрібен для того щоб передати на сервіс ІР-адрес клієнта який робить запит. Бо стандартний метод визначення ІР не буде спрацьовувати оскільки робота може йти через проксі, тоді в ІР буде просто ІР проксі, а не клієнта.

В мікросервісах для ідентифікації запиту клієнта не використовують сесії як в моноліті, а використовують Identity Service який приймає якийсь credentional (логін, пароль) з UI, і повертає на UI певний JSON identity токен. В JSON токені вказується алгоритм шифрування, тип токена, підпис тощо. Зміст JSON токена закодовано.

**Способи інтеграції сервісів між собою:**

- shared database

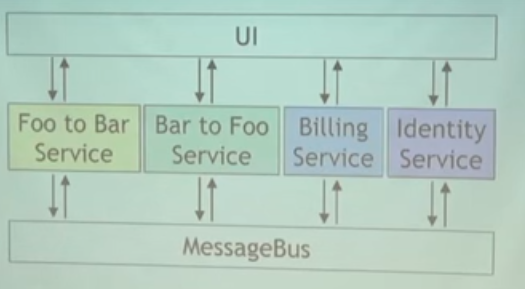
- REST / RPC

- asynchronous messages

**shared database** - найпростіший спосіб інтеграції сервісів між собою. Це коли у нас є декілька сервісів але по факту вони працюють з одним й тим самим додатком. Такий підхід порушує інкапсуляцію сервісів і на практиці це означає, що в майбутньому буде важко такі сервіси деплоїти окремо, тощо. Тобто між ними утворються залежності і можна втратити саму перевагу розробки через сервіси.

**REST / RPC** - виглядає найочевиднішим і найлогічнішим, оскільки легко інтегрувати сервіси з UI. Правда такий підхід утворює зв’язок між сервісами. Таким чином може утворювати складну систему в майбутньому.

**asynchronous messages -** дозволяє підтримувати слабку залежність між сервісами. В такому випадку ми не комунікуємо з жодним сервісом напряму, а працюєму з посередником в рамках абстракції. Добавляється шина повідомленнь, яка комунікує між собою сервіси. Правда такй підхід може трохи ускладнити рішення в загальному.



**Сутності шини повідомлень:**

- **Повідомлення** (в вигляді JSON). Типи: команда, подія, запит, відповідь.

- **Канал** (абстракція яка поєднує в собі однорідні повідомлення). Приклад іменування "scope.under\_score" - через точку групуються повідомлення.

- **Виробник** **повідомлень** (сервіс який утворює повідомлення). Наприклад, зберегли Ентіті в базу, добавили повідомлення в шину повідомлень щоб інші сервіси про це дізналися.

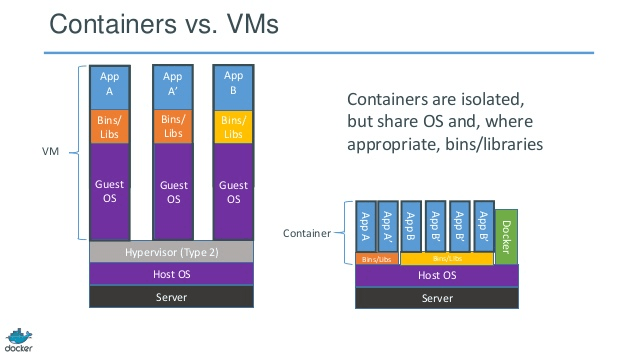
- **Споживач** **повідомлень** (сервіс, який приймає повідомлення). Споживачі повинні бути готовими до того, що одне повідомлення може прийти до них 2 раза, тобто вони повинні гарантувати ідемпотентність. Також бестпрактіс є споживання з підтвердженням. Тобто, коли нам відправили повідомлення, а ми зламалися - підтвердження не відправиться і таким чином не видалиться з шини повідомлень; Коли у нас все буде ок, ми витягнемо повідомлення, виконаємо потрібні дії і відправимо підтвердження. Споживач має 2 режима. **Push** - коли повідомлення отримується одразу після розміщення в шину. **Pool** - коли сервіс споживач сам звертається в шину і перевіряє наявність повідомлень для нього.

**Идемпотентність** - властивість об’єкта, коли над ним виконуються якісь повторні дії, а його стан після цього не міняється.

**SOA** - Service Oriented Archutecture. Трішки відрізняється від мікросервісів через великий фокус на ESB для інтеграції монолітів. (Enterprise Service Bus - представляє і викликає сервіси інтегрованих систем.)

Централізоване управління - SOA. Децентралізоване - Bounded Context (DDD).

--------------------------------------------------------------------------------------------------------------------------------------



- Образи контейнерів мають менший розмір в порівнянні з образами для віртуальних машин.

- Швидший запуск.

- Запускаються тільки ті процеси, які ми вказали. (ВМ, наприклад, запускає ОС зі всіма своїми процесами.)

- Контейнери вирішують конфлікт, коли різним сервісам потрібні бібліотеки різних версій.

- Контейнери легко переносити між машинами чи хостингами\облаками.

**Контейнеризація** дозволяє нам запускати наші додатки в будь-якому енвайрменті без необхідності налаштування кожного окремого енвайрменту і самого додатка. Робиться це шляхом запуску контейнера з додатком.

**Docker** - платфорама яка автоматизує упаковку і розгортання додатку і забезпечує ізоляцію від енвайрменту на рівні додатку. Він упаковує додаток з всіма його залежностями і потім його можна запустити на будь якому енвайрменту де є Docker. Тобто, наприклад, той самий контейнер, який девелопер розробляє і тестує на своєму ноутбуці можна легко мігрувати на сервер в облако тощо.

**docker run** -p - відкрити всі контейнери.

**Докеризація** - приведення додатку до вигляду готовому для запуску в Докер-контейнері.

Контейнери схожі чимось на Git. Контейнери маюь ревізії, комміти, історію, pull, push, повторне використання, шарінг etc.

**DockerHub** - колекція базових образів.

docker **info -** інфо про встановлений докер

docker **history** image\_name покаже всі слої образа в хронологічному порядку.

docker **pull** ubuntu

docker **push** betterUbuntu - запушити образ в **DockerHub.**

docker push trukhinyuri/nginx

На докерхабі самими виробниками залиті їхні контейнери, тому ми просто можемо вказувати, наприклад, що ми хочемо контейнер де предвстановлено Ubuntu такої-то версії де вже установлено PostgreSQL такої-то то версії.

**boot2docker (DockerToolbox)** - віртуальна машина (з nix’овим ядром) і сам Докер. З неї можна спілкуватися з контейнерами - запускати їх, зупитняти тощо. Якщо використовувати його на Windows чи MacOS, то коли ми захочемо прокинути порт з контейнера на верх (docker run -p 80:80 some\_container), то він прокинаться в віртульній машині. 80:80 - порт всередині контейнера:порт який буде видно зовні. На Unix-like системах порт з контейнера прокинеться нормально. Щоб дізнатися IP на віртуальній машині - boot2docker ip.

Докер не застосовує емуляцію заліза, а тільки ізолює процеси і файлову систему, тому працювати з ним продуктивніше на відміну від віртуалізації (швидше запуск, можна запустити в 5-6 раз більше контейнерів, маленький образ).

root@ubuntu:~$ # ми всередині віртуалки ubuntu.

root@55f676jgh67:/ # ми в середині якогось контейнера.

**Контейнер** - запущений образ + слой файлової системи + метадані. Контейнери складаються з леєрів які наслідуються одне від одного. Кожен леєр містить інформацію про всі файли які в ньому були змінені.

**Образ** (image) - read only слой. З образів запускаються контейнери.

docker **images** - показує всі образи.

**Слой** (layer) - зліпок файлової системи.

Дістати образ з репозиторія і відправити назад після змін:

docker **pull**, docker **commit**

Створити новий образ на основі існуючого:

docker **build**

Створити новий образ з нуля:

docker **create**, docker **import**

Подивитися всі контейнери включаючи зупинені і їх айдішки:

docker **ps -a**

До контейнера можна звертатися як по ID так і по імені:

docker **start** some\_container

Закомітити зміни в образ по айдішці:

docker **commit <ID>** image\_name

Зберегти на диск:

docker **save** imge\_name > /flash/image.tar

Завантажити з диска:

docker **load** < /flash/image.tar

**Droplet** - віртуальна машина на Digital Ocean.

**CoreOS** - спеціально розроблена ОС для запуску докер образів. Основана на Лінуксі. Має встановлений Докер.

**IaaS -** infrastructure as a servece - надання облачної інфраструктури для керування ресурсами, обробкою і збереженням даних, серверами, програмним забезпеченням і операційними системами для запуску клієнтами свого ПО тощо.

**PaaS** - platform as a service - платформа по потребы. Коли заказчик маэ доступ до використання облачних платформ типу операцыйних систем, СУБД, засобыв розробки та тестування тощо і відповідно не затрачується на налаштування та підтримку додаткового ПО і заліза.

**SaaS** - sowtware as a service - програмне забезпечення по потребі. Коли заказчик має доступ до додатку через інтернет і не витрачається на установку, налаштування і підтримку додатку та устаткування для нього.

**AuFS** - допоміжна файлова система, яка какскадно об’єднує і монтує файлові системи Лінукс.

Стоворити контейнер з Убунту:

sudo docker run -i -t ubuntu /bin/bash

**-і** залишає відкритим STDIN (поток вводу з клавіатури). Тобто ми запускаємо контейнер і прєднуємо консоль до цього контейнера. (interactive)

**-t** назначає контейнеру псевдо-tty.

**tty** - teletypewriter - керуючий термінал для процесу.

До прикладу, установка **nano** в контейнер:

apt-get update

apt-get install -y nano

Для підключення до контейнеру:

docker **attach** some\_container

Подивитися деталі контейнера:

docker **inspect** some\_container

Видалити контейнер:

docker **rm** some\_container

Скопіювати дані в контейнер чи витягнути дані з контейнера:

docker cp <путь к данным на хосте> <имя контейнера>:<путь>

**nginx** - НТТР сервер, проксі сервер, почтовий сервер для високонавантажених систем.

Скрипт для запуску образу при старті хоста (наприклад, ми змонтували образ з nginx):

description "Nginx"

author "Me"

start on filesystem and started docker

stop on runlevel [!2345]

respawn

script

/usr/bin/docker start -a nginx

end script

**Dockerfile -** міститьв собі набір інструкцій з аргументами і визначає, як буде будуватися образ. Кожна інструкція добавляє новий слой в образ і коммітить зміни. Тобто:

- запускається контейнер з образу

- виконується інструкція і вносить зміни в образ

- запуск еквівалента docker commit для внесення змін в образ

- запуск нового контейнера з нового образу

- виконання наступної інструкції і повторення кроків вище.

Приклад створення Dockerfile і подальшому монтуванні образу командою docker **build**:

mkdir static\_web

cd static\_web

touch Dockerfile #створили Dockerfile

Сам **Dockerfile** з інформацією по зборці образа:

FROM ubuntu:14.04 #завжди має бути першим в файлі і вказує з якого базового образа робити наш. Тобто ми повинні наслідуватися від якогось контейнера.

MAINTAINER Petro Andriets <web.it.www@gmail.com>

RUN apt-get update #отримати оновлений список пакетів

RUN apt-get install -y nginx #установити\обновити пакет

RUN echo 'Hello from container' > /usr/share/nginx/html/index.html #створюємо index.html

EXPOSE 80 #вказуємо порт додатку в контейнері чи прокидаємо порти між контейнерами

Створюємо образ з нашого Dockerfile:

docker **build** -t andriets/nginx ~/static\_web #andriets - ім’я репозиторія

#ngibx - ім’я образу (по дефолту latest)

#~/static\_web - шлях до папки з Dockerfile

На ДокерХаб компанії часто викладають Докерфайл по якому білдиться їх контейнер, тому їх можна також вважати trusted як контейнери від Ubuntu, PostgreSQL, CentOS, MySQL etc.

Файл **.dockerignore** - список паттернів виключень.

По замовчуванню Docker кешує кожен крок і формує кеш зборок. Щоб виключити:

docker build **--no-cache** -t andriets/nginx ~/static\_web

**CMD** вказує, що запустити, коли контейнер запуститься:

CMD ["/bin/bash", "-1"] #запускаємо баш і передаємо йому параметр в виді масива

**ENTRYPOINT** робить теж саме, тільки її не можна перегружати як CMD. (хіба використати флаг --entrypoint)

**WORKDIR** покаже робочу директорію з якої запускатимуться CMD i ENTRYPOINT.

**VOLUME** добавляє розділи в образ які можна розшарювати і повторно використовувати між контейнерами. #VOLUME ["/opt/project"]

**ADD** добавляє файли чи папки в образ. можна вказувати і ссилки:

ADD software.lic /opt/application/software.lic

ADD http://wordpress.org/latest.zip /root/wordpress.zip

**ONBUILD** добавляє трігери в образ. Трігер виконується, наприклад, коли образ використовується як базовий для іншого образу.

**Комунікація контейнерів між собою**: зв’язок через **проброс портів** або **лінковку контейнерів**.

**Проброс портів:**

docker run -p 127.0.0.1:80:80 #прив’язка до локалхоста

docker run -p 80:80/udp #можна прив’язати UDP порти

**Лінковака контейнерів:**

docker -**ps** показує зв’язані контейнери

Наприклад, аби створити зв’язок між контейнерами web i db треба використати **--link name:alias**

docker run -d -P --name web --link db:db trukhinyuri/webapp python app.py

**sudo usermod -a -G docker vagrant** - щоб не набирати кожного разу sudo перед docker. This thing add user vagrant into docker group. Need to relog-in after.

vagrant дозволяє налаштовувати ВМ. Наприклад, можна написати скріпт, який підніме ВМ з потрібним конфігом.

По hands-on (youtube):

- docker version

- docker pull ubuntu:14.04

- docker images — показує наявні образи.

- docker run -it ubuntu:14.04 — запускає цей образ. Фактично ми опиняємося всередині убунти.

- ps aux — показує процеси. В даному випадку ніяких процесів всередини образа не відбуватиметься.

- ps aux | grep skype — виведе процеси skype із всього списку процесів які виконуються.

- sudo touch Dockerfile

FROM ubuntu:14.04

CMD echo “Hello” #має бути хоча б один процес при запуску контейнера, якщо не вказати нічого, то буде запущено баш.

- docker build -t peter/test\_dockerfile . #білдимо образ. -t задає ім”я (як на докерхабі/test\_dockerfile) образу. Точка (.) вказує що ми працюємо в поточному каталозі.

Контейнер буде складатися з декількох леєрів (в нашому випадку 1. убунта, 2. CMD echo процес). Цим леєрам будуть даватися окремі айдішки в вигляді хеш. Леєр який буде останній в контейнері буде фінальним леєром для контейнера.

- docker run peter/test\_dockerfile — запустить контейнер який ми збілдили.

В Докер хаб при push аплоадяться тільки леєри, яких ще нема в репозиторії. Відбувається перевірка по хешу.

- docker pull redis — запулити редіс.

- docker run -d redis — запустити редіс. -d вказує, що процес буде бігти в бекграунді. (детач)

- docker ps — покаже які процеви зараз виконуються.

- docker ps -a — покаже всі процеси які були запущені і зупинені.

- docker kill <Id> - вбити процес по айді.

Так як контейнера наслідуються одне від одного, рекомендують розбивати задачі на різні контейнери. Наприклад, в одному контейнері налаштувати джаву, в іншому томкат і т.д. Тоді кожен контейнер займатиметься своєю задачею і не треба буде ребілдити кучу всього, якщо десь відбудуться якісь зміни. Тобто потрібно дотримуватися ієрархії контейнерів.

Якшо поміняти “середній” леєр, то потрібно перебілдити всі леєри, які йдуть після нього. Тому, зазвичай контейнери які часто змінюють роблять останніми.

Важливо пам”ятати, що кожна команда в докер файлі створює новий леєр, тому, якщо ми в контейнері викачуємо якісь ісходніки, конфігуруємо їх, білдимо, а потім стартуємо — то всі ці кроки будуть окремими леєрами і в результаті ми отримаємо громіздкий контейнер. Навіть якщо ми вкажемо, що треба виалити ці всі попередні дані, то ми отримаємо просто новий леєр де це буде вказано. До прикладу Докерфайл:

FROM ubuntu 14:04

RUN apt-get uodate -q

RUN sudo apt-get install software-properties-common -y

RUN sudo add-apt-repository ppa:chris-lea/node.js -y

RUN apt-get update -q

RUN sudo apt-get install nodejs -y

Тому такі операції краще виконувати в одній строці RUN, тоді на виході в контейнері залишиться тільки один леєр без зайвих даних, які ви видалили.

**CMD** / **RUN**:

CMD каже, що виконувати при старті контейнера. Якщо вказати декілька CMD то виконається тільки остання.

RUN каже, що виконувати на момент компіляції контейнера.

При повторному білді докерфайла будуть виконуватися тільки змінені строки, так як докер кешує вже виконані і якщо строка не змінилася, то він візьме закешований результат. Тут важливо пам”ятати, що команди, які при виконані можуть повертати різний результат (наприклад, apt-get update) то їх не треба кешувати. Для цього можна запустити команду з ключом “не використовувати кеш”.

**Приклад** білда і запуску контейнера з **nodejs**:

Dockerfile:

FROM ubuntu 14:04

RUN apt-get uodate -q

RUN sudo apt-get install software-properties-common -y

RUN sudo add-apt-repository ppa:chris-lea/node.js -y

RUN apt-get update -q

RUN sudo apt-get install nodejs -y

- docker build -t forzamilan-nodejs

- docker run -it forzamilan-nodejs

- nodejs - -version #ми всередині контейнера дивимось версію nodejs

- exit

Також важливо пам”ятати, що всі зміни після виходу з контейнера втрачаються. Тобто, якщо ми видалимо якусь директорію всередині контейнера і перезапустимо цей контейнер, то папка знову з”явиться.

По замовчуванню порт на якому висить контейнер не видно зовні контейнера. Потрібно в ручну прокидати порт на зовні. Всередині докерфайла вказуємо **EXPOSE** 6001, робимо ребілд контейнера і тепер контейнер доступний зовні по 6001 порту. Далі при запуску вказуємо, що ми прокинули порт і мапимо його відповідно:

docker run -it **-p 6001:6001** forzamilan-nodejs

Коли нам потрібно якось перманентно зберігати дані, то

1) можна замапити якусь папку з лінукса в папку всередині контейнера в яку це все зберігати.

2) можна створити окремий контейнер в який писатимуться дані (рекомендовано)

Fiddler — прокси, который работает с трафиком между Вашим компьютером и удаленным сервером, и позволяет инспектировать и менять его

**Ендпоінт** - клас, який обробляє закінечення урла запиту. Приклад, аннотація над класом RequestMapping(“health”) і методи в ньому помаркані також, наприклад RequestMapping(“heartBeat”)