İlk adım olarak developerler için bir server ı manuel olarak ayağa kaldırıyoruz.

Bu instance a ssh ile bağlanıyoruz.

Microservice project dosyalarını vs code a kopyaladık.

sudo dnf update -y

sudo hostnamectl set-hostname petclinic-dev-server

sudo dnf install docker -y

sudo systemctl start docker

sudo systemctl enable docker

sudo usermod -a -G docker ec2-user

sudo curl -SL <https://github.com/docker/compose/releases/download/v2.17.3/docker-compose-linux-x86_64> -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

sudo dnf install git -y

sudo dnf install java-11-amazon-corretto -y

newgrp docker

Komutlarını sırasıyla giriyoruz.

Gerçek hayatta biz bu instance ı manuel kaldırmayız. Terraform ile hazırlarız ve developer terraform apply ile ayağa kaldırır.

İkinci adım olarak kendimize bir repo hazırlayacaz. Bunun için önce uygulamayı lokal repoya klonluyoruz.

git clone <https://github.com/clarusway/petclinic-microservices-with-db.git>

Biz fork yaparak da projeyi kendi repomuza alabiliriz ancak, forkladığımızda commit de yapsak github da yeşillik olmaz 😉

petclinic-microservice-with-db isimli bir repo oluşturduk GitHub ımızda.

Lokalimizdeki repoya girip burada .git doyasını siliyoruz.

cd petclinic-microservices-with-db

rm -rf .git

Daha sonra şu komutları adım adım giriyoruz.

git init

git add .

git config --global user.email "you@example.com"

git config --global user.name "Your Name"

git commit -m "first commit"

git branch -M main

Son satırdaki komut branchı main branch yapıyor.

git remote add origin [https://[github](about:blank) username]:[your-token]@github.com/[your-git-account]/[your-repo-name-petclinic-microservices-with-db.git]

komutuyla repomuza remote repoyu tanımlıyoruz ve lokali bu origine push ediyoruz:

git push origin main

burada repo ismini yanlış girerek hata alanalar için "git remote set-url origin [https://token@url](https://token@url/)" komutu ile origin repo değiştirlebilir.

"git remote -v" komutu da uzaktaki repoyu gösterir.

şimdi repomuzda bir dev branch ı oluşturuyoruz.

git checkout main

git branch dev

git checkout dev

git push --set-upstream origin dev

git branch dev branch ını olutşururken git checkout dev de dev branchına geçiyor. ilk deva push ederken de --set-upstream ile diyoruz. bunun da kısayolu -u dir. bu sayede lokalde oluşturduğumuz branch remote ta da oluşur.

daha sonra dev branchında iken bir de release branchı oluşturuyoruz.

git checkout dev

git branch release

git checkout release

git push --set-upstream origin release

.bashrc ye

parse\_git\_branch() {   
 git branch 2> /dev/null | sed -e ' /^[^\*]/d' -e 's/\*\(.\*\)/(\1)/'   
}   
export PS1="\[\e[1;31m\]\u\[\e[35m\]@\h# \W:\[\e[33m\]\[\e[1;36m\]\$(parse\_git\_branch)$\[\033[00m\] "

komutunu yapıştırırsak bize çalıştığımız branchı da gösterecek.

**şimdi 3.adımdayız** ve dev branchına geçiyoruz tekrar.

mvn test komutu hem compile hem unit testleri yapıyordu

mvvn package hem compiler hem built hem de edest yapıyordu.

mvn clean install komutu da hem compile, hem build hem test hem de lokal repoya kopyalıyor.

mvn deploy da repoya gönderiyor.

clean da target dosyaları varsa onları siliyor. mvn lifecycle ı hatırlayalım:

A Build Lifecycle is Made Up of Phases 
Each ot these build lifecycles is defined by a ditterent list of build phases, wherein a build phase represents a stage in 
the lifecycle. 
For example, the default lifecycle comprises of the following phases (for a complete list ot the lifecycle phases, refer to 
the Litecycle Reterence): 
validate - validate the project is correct and all necessary information is available 
compile - compile the source code of the project 
test -test the compiled source code using a suitable unit testing framework. These tests should not require the code 
be packaged or deployed 
- take the compiled code and package it in its distributable tormat, such as a JAR. 
verify - run any checks on results ot integration tests to ensure quality criteria are met 
- install the package into the local repository, for use as a dependency in other projects locally 
install 
deploy - done in the build environment, copies the final package to the remote repository tor snaring witn other 
developers and projects. 
These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the 
default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first 
validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run 
integration tests against that package, verity the integration tests, install the verified package to the local repository, then 
deploy the installed package to a remote repository. 

mvn da bu komutları bilmemiz yeterli.



ancak programla birlikte gelen bir araç, maven lite olarak düşünebiliriz.

eğer windows da çlışıyorsak bu script çalıştırılır:



biz instance a mvn yüklemedik bu nedenle şu anda mvn komutları çalışmaz.

biz her instance a sürekli maven yüklersek farklı versionlar yüklemek zorunda kalabiliriz ve versiyon kontrolünü kaybederiz.

bu nedenle mvnw yani maven wrapper kullanılır. bu komut ilgili mvn wraper ını yüklüyor ve işlemlerini onun üzerinden yapıyor.

v wrapper/ dists/apache-maven-... 
> apache-maven-3.5.4 
apache-maven-3.5.4-bin.zip 

apache-maven-3.5.4 versioyununun wraper ını yükledi, bütün developerlar mvnw sayesinde bu verisoynu aynı versiyonu kullanacak.

biz ./mvnw clean install

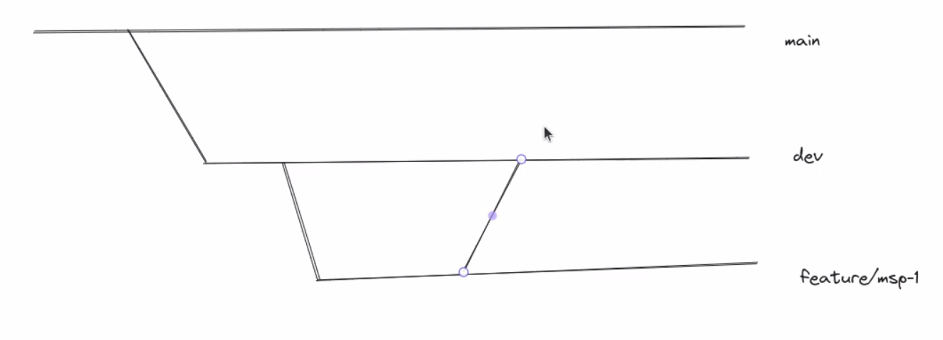
komutunu girdiğimizde bütün serverlar içinde .jar file ları oluşturdu. ve file ları lokal repoya (.m2) ye atıyor ancak bunu nereden biliyordu? pom.xml deki gav dan:

<parent> 
<artifactld>spring-boot-starter-parent</ar 
2.1.1. 
</parent> 

kısa bir mvn tekrarı yaptık.

**Projenin 4. adımına geldik:**

bu adımda bir developerın proje üstünde çalışırken feature branchı oluşturup burada çalışıp daha sonra dev branchına push etmesini senarize edeceğiz:



git checkout dev

git branch feature/msp-4

git checkout feature/msp-4

feature/msdp-r branchındayız. şimdi ana klasör altında b,r .sh dosyası oluşturup içine ./mvnw clean package

komutunu yazıyoruz.

burada komutları oluşturup dosyalama yönünde verilen bir talimat verildi ve bi de bunu yaptık.

0 README-.md 
s 
package-with-mvn-wrapper.sh X 
petclinic-microservices-with-db > S package-with-mvn-wrapper.sh 
. /mvnw clean package 
1 

görevimizi bitirdik. şimdi bu değişikliği push edecez:

git add .

git commit -m 'added packaging script'

git push --set-upstream origin feature/msp-4

şimdi dev branchına geçip feature/msp-4 branchını dev branchına merge ediyoruz:

git checkout dev

git merge feature/msp-4

git push origin dev

Şimdi 5. adımdayız:

Bu adımda da yeni bir görevimiz var:   
Prepare development server script with [terraform files](infrastructure/msp-5-dev-server-of-petclinic/dev-server.tf, dev-variable.tf, dev.auto.tfvars, petclinicserver-userdata.sh) for developers, enabled with `Docker`, `Docker-Compose`, `Java 11`, `Git` and save them under `infrastructure` folder.

Developerlar için içinde docker, docker compose, java 11 ve git yüklü bir instance ayağa kaldıracak bir terraform dosyası oluşturun dediler. Az önce oluşturduğumuz dev server ı bu kez terraform file ile oluşturlması için terraform file yazacaz.

hemen yeni bri branch olutşuruyoruz.

git checkout -b feature/msp-5

komutu branchı hem olutşurur hem o brancha geçer.

ana klasör içerisinde infrastructure isimli bir klasör oluşturuyoruz.

burada "kendimizin oluşturduğu" (proje klasöründe hazır olan) terraform fileları infrastructure içerisine yapıştırdık.

v infrastructure 
dev-server.tf 
dev-variable.tf 
dev.auto.tfvars 
S petclinicserver-userdata.sh 

görevi tamamladık ve branchı push edip dev branchına merge ediyoruz:

git add .

git commit -m 'added terraform files for dev server'

git push --set-upstream origin feature/msp-5

git checkout dev

git merge feature/msp-5

git push origin dev

**Şimdi 6.adımdayız.**

burada görevimiz bir dockerfile oluşturmak.

öncelikle admin-server microservice için bir dockerfile oluşturacaz.

developerler bunun openjdk:11-jre ile çalışacağını söyledi.

neden openjdk:11 kullanmıyoruz? jdk jjava kod yazılması için gerekliydi. oysa bizim zaten .jar file larımız hazır code geliştirilmeyecek. bu nedenle o image gereksiz büyük biz de yeterli olduğu için jre yi kullanıyoruz.

Dockerfile ımızı en basit hali ile yazarsak:

**FROM openjdk:11-jre**

**COPY ./target/\*.jar /app.jar**

**ENTRYPOINT [ "java", "-jar", "/app.jar" ]**

önce çalıştığını görecez sonra eklemeler yapacaz.

diğer serverlarda da .jar file lar var bu Dockerfile onlar için de geçerli olur.

Biz bir microservice oluşturuyoruz, bu mimaride sıralı şekilde ayağa kalkaması gerekiyor microservice lerin:

Architecture diagram of the Spring Petclinic Microservices 
YAML 
configuration 
Spring Cloud 
Config 
Git 
static resources 
REST requests 
fetch configuratim 
fetch configuratibn 
Eureka 
Senrice 
Dixovery 
retrieve microservices IP 
bad•balanced 
HTTP requests 
customers-service 
REST API 
crandom pcrb 
UI + API Gateway 
(Wt 080> 
vets-service 
REST API 
crandom port> 
Spring 300t 
Admin Server 
Zipkin 
Distributed 
Logging 
Server 
distributed 
visits-service 
REST APE 
Crandom port* 

önce Config server ayağa kalkacak, o kalktıktan sonra

Discovery server ayağa kalakacak.

Daha sonra diğer microserviceler config serverdan verileri alıp ayağa kalkıp discovery server a bilgi verecek.

Bu işlem için başka bir programa ihtiyacımız var. Dockerize

Dockerize yüklendiği zaman microservice in ayapa kalkamadan önce başka bir microservice in ayakta olduğundan emin olur.

Bu nedenle ilk server olan config server a yüklenmesine gerek yok.

**FROM openjdk:11-jre**

**ADD** [**https://github.com/jwilder/dockerize/releases/download/v0.6.1/dockerize-alpine-linux-amd64-v0.6.1.tar.gz**](https://github.com/jwilder/dockerize/releases/download/v0.6.1/dockerize-alpine-linux-amd64-v0.6.1.tar.gz) **dockerize.tar.gz RUN tar -xzf dockerize.tar.gz**

**RUN chmod +x dockerize**

**COPY ./target/\*.jar /app.jar**

**ENTRYPOINT [ "java", "-jar", "/app.jar" ]**

ADD komutuyla dockerize ı yükledik ve yüklenen dosyanın ismini dockerize.tar.gz olarak değiştirdik.

Run tar -xzf ile unzip ettik.

RUN chmod +x ile de executable hale getirdik programı.

şimdi EXPOSE ile hangi porttan çalışacağını belirtecez. bunu bize developer söyler:

Discovery Server - http://localhost:8761 
• Config Server - http://locaIhost:8888 
AngularJS frontend (API Gateway) - http://localhost:8080 
• Customers, Vets and Visits Services - random port, check Eureka Dashboard 
Tracing Server (Zipkin) - http://locaIhost:9411/zipkin/ (we use openzipkin) 
Admin Server (Spring Boot Admin) - http://localhost:9090 
• Grafana Dashboards - http://localhost:3000 
• Prometheus - http://Iocalhost:9091 
Hystrix Dashboard for Circuit Breaker pattern - http://localhost:7979 - On the home page is a form where you can 
enter the URL for an event stream to monitor, for example the api-gateway service running locally: 
http://10ca1host:8888/actuator/hystrix.stream or running into docker: http://api- 
gateway : 8888/ actuator/hystrix. stream 
You can tell Config Server to use your local Git repository by using local Spring profile and setting GIT REPO 
environment variable, for example: -Dspring.profiles.active=local -DGIT REPO=/projects/spring-petc1inic- 
microservi ces-config 

admin-server ın 9090 portundan çalışacağını belirtmişler.

**FROM openjdk:11-jre**

**ADD** [**https://github.com/jwilder/dockerize/releases/download/v0.6.1/dockerize-alpine-linux-amd64-v0.6.1.tar.gz**](https://github.com/jwilder/dockerize/releases/download/v0.6.1/dockerize-alpine-linux-amd64-v0.6.1.tar.gz) **dockerize.tar.gz**   
**RUN tar -xzf dockerize.tar.gz**

**RUN chmod +x dockerize**

**COPY ./target/\*.jar /app.jar**   
**EXPOSE 9090**

**ENTRYPOINT [ "java", "-jar", "/app.jar" ]**

bu uygulama default olarak in-memory database (HSQLDB) ile çalışıyor.

ancak developlerlar ile konuştuk ve mysql database inde çalışmasına karar verildi.

eğer microservice leri docker ile çalıştırırken mysql kullanıyorsak mysql profile ini Dockerfile a şu şekilde eklememiz gerekiyor:

**ENV SPRING\_PROFILES\_ACTIVE docker,mysql**

Dockerfile da ARG yi daha sonra değiştirmek isteyebileceğimiz data lar için kullanıyoruz. ARG ile variable olarak veriyi girip daha sonra değiştirebiliyoruz.

**FROM openjdk:11-jre**

**ARG DOCKERIZE\_VERSION=v0.6.1**

**ARG EXPOSED\_PORT=9090**

**ENV SPRING\_PROFILES\_ACTIVE docker,mysql**

**ADD** [**https://github.com/jwilder/dockerize/releases/download/${DOCKERIZE\_VERSION}/dockerize-alpine-linux-amd64-${DOCKERIZE\_VERSION}.tar.gz**](https://github.com/jwilder/dockerize/releases/download/$%7BDOCKERIZE_VERSION%7D/dockerize-alpine-linux-amd64-$%7BDOCKERIZE_VERSION%7D.tar.gz) **dockerize.tar.gz**

**RUN tar -xzf dockerize.tar.gz**

**RUN chmod +x dockerize**

**ADD ./target/\*.jar /app.jar**

**EXPOSE ${EXPOSED\_PORT}**

**ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]**

Dockerfile ımızın son hali bu oldu.

dev branchına gidip feaute/msp-6 branchı oluşturalım.

admin-server altında Dockerfile ı oluşturalım.

api-gateway server için ve sırasıyla diğer serverlar için de aynı işlemleri yapıyoruz.

Dockerfile içeriğinde sadece portlar değişiyor. Portları da daha önce belirttiğimiz gibi developerlar veriyor.

git add .

git commit -m 'added Dockerfiles for microservices'

git push -u origin feature/msp-6

git checkout dev

git merge feature/msp-6

git push origin dev

komutlarını tek tek girelim ve msp-6 yı bitirelim.

Bu günlük bu kadar.

Dockerfile üzerinde biraz daha konuşalım.

**FROM openjdk:11-jre**

**ARG DOCKERIZE\_VERSION=v0.6.1**

**ARG EXPOSED\_PORT=9090**

**ENV SPRING\_PROFILES\_ACTIVE docker,mysql**

**ADD** [**https://github.com/jwilder/dockerize/releases/download/${DOCKERIZE\_VERSION}/dockerize-alpine-linux-amd64-${DOCKERIZE\_VERSION}.tar.gz**](https://github.com/jwilder/dockerize/releases/download/$%7BDOCKERIZE_VERSION%7D/dockerize-alpine-linux-amd64-$%7BDOCKERIZE_VERSION%7D.tar.gz) **dockerize.tar.gz**

**RUN tar -xzf dockerize.tar.gz**

**RUN chmod +x dockerize**

**ADD ./target/\*.jar /app.jar**

**EXPOSE ${EXPOSED\_PORT}**

**ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]**

ADD ve COPY her ikisi de local den file kopyalar ancak best practice COPY kullanılır. ADD komutunu dışarıdan URL almak için kullanıyoruz ancak güvenli değil. Birisi gidip bu adresteki linke kötü amaçlı yazılım eklerse konteynırmızı ele geçirebilir. Şirketler de buna engel olmak için ADD kullanıldığında uyarı verecek scriptler yazarlar.

**ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]**

Burada araya **-Djava.security.egd=file:/dev/./urandom** girdi. Burada -D delimeter demek. bu komutu developerlar verir normalde ancak projede olduğu için inceleyelim.

bunun defaultunda son kısımdaki urandom değil random dur. random daha yavaş ve sitemin kilitlenmesine neden olabiliyor. urandom daha hızlıdır ve sistemin kilitlenmemesi için kullanılır. Güvenliği daha düşüktür. Bunun nedenine ise ders sonunda optional olarak bakacaz.

ARG neden kullanılıyor?

ARG values available 
Dockerfile 
build image 
Image 
run conbainer 
Container 
ENV values available 

ARG build time da değiştirebileceğimiz bir değişkeni Dockerfile da belirtmek için kullanılır. Bu Dockerfile daki örnekte Dockerize ın verisonu değişebilir bunu sonradan tek tek Dockerfile larda değiştirmemek için ARG ile belirttik.



build ederken bu şekilde versionu güncelleyebiliriz.

ancak ENV ile belirtilen environment variable lar konteynır ayağa kalktıktan sonra kullanılacak variable lar için kullanılır. ARG detanımlanan variable lar ise konteynır ayağa kalktıktan sonra kullanılmaz.

TDİ-2162 Elif 
7:26 PM 
ARG ile tanimladigim bu degiskenler, bu imajdan 
yaratilan conteynir icinden erişilemez.BunIar 
environment variable degildir, sistemden erişilen 
degiskenler tanimlamak istersek sadece ENV talimatini 
kullaniyorum.Arg kullanmiyoruz, ARG sadece imaj 
oluşturma işlemlerinde degisken kullanmak istedigimiz 
zaman kullaniyoruz.yani konteynir icinden bu versiyona 
erişemem. 

TOI-2162 Elif PM 
Buildtime = imajin olusturuldugu zaman . imaj build 
ederken degisken kullanmak istiyorsak Arg kullaniriz. 
Runtime = konteynirin icinde gecerli . burada degisken 
kullanmak istersek ENV kullaniriz. 

Developerlar bize dedi ki eğer uygulamayı konteynır içerisinde kullanıp mysql ile çalıştıracaksak

**ENV SPRING\_PROFILES\_ACTIVE docker,mysql**

Dockerfile a eklenecek.

If you are running the microservices with Docker, you have to add the mysqt profile into the (Dockerfile) 
[docker/Dockerfile]: 
ntTIVF 

Eğer biz uygulamayı microservice lerde çalıştırmayacaksak

../mvnw spring-boot:run komutunu her bir klasörün içinde terminalde yazarak default olarak çalıştırırız.

Every microservice is a Spring Boot application and can be started locally using IDE or 
./mvnw spring-boot: run 
command. Please note that supporting services (Config and Discovery Server) must be started before any other 
application (Customers, Vets, Visits and API). Startup of Tracing server, Admin server, Grafana and Prometheus is optional. 
If everything goes well, you can access the following services at given location: 
Discovery Server - http://localhost:8761 
• Config Server - http://locaIhost:8888 
AngularJS frontend (API Gateway) - http://localhost:8080 
• Customers, Vets and Visits Services - random port, check Eureka Dashboard 
Tracing Server (Zipkin) - http://locaIhost:9411/zipkin/ (we use openzipkin) 
Admin Server (Spring Boot Admin) - http://localhost:9090 
• Grafana Dashboards - http://localhost:3000 
• Prometheus - http://Iocalhost:9091 
Hystrix Dashboard for Circuit Breaker pattern - http://localhost:7979 - On the home page is a form where you can 
enter the URL for an event stream to monitor, for example the api-gateway service running locally: 
http://10ca1host:8888/actuator/hystrix.stream or running into docker: http://api- 
gateway : 8888/ actuator/hystrix. stream 

bu şekilde çalıştırırsak uygulama instance ta çalışır.

(Interviewlerde böyle default verip Kubernetes te nasıl çalıştırırsınız diye sorabilirler)

v spring-petclinic-customers-service 
> java.org/springframework.. 
v resource S 
> db 
O 
application.properties 
bootstrap .yml 
logback-spring.xml 

bootstrap.yml x 
petclinic-microservices-with-db ) spring-petclinic-discoveryserver 
6 
8 
9 
spring: 
cloud : 
config: 
uri: http://localhost:8888 
appli cation : 
name: discovery-server 
spring: 
profiles: docker 
cloud : 
conf ig : 
uri: http://COnfid•server:8888 

defaultu kullanırsan localhosta ( üstteki ) iken spring\_profile dokcer ise cpnfig-server a git. çünkü konteynırda çalışırken localhost kalırsa konteynıra ulaşamaz çünkü buradaki localhost konteynır ın local i değil, instance ın lokalidir.

Bu profiller maven da , pom.xml de yer alıyor. Eğer profile olmasaydı developerlar cloud ve konteynır için ayrı ayrı uygulama yazmak zorunda kalırdı. Ya da HSQLDB veya mysql dbsinde çalışması için her ikisi için ayrı uygulama yazacaklardı. Profile işte tek uygulamaya gömülerek farklı varyasyonlarda uygulamanını kullanılabilmesini sağlar.

**ENV SPRING\_PROFILES\_ACTIVE docker,mysql**

biz de burada spring\_profiles\_active değişeninde profilleri docker ve mysql olarak seçiyoruz.