Labo 1: Spring boot

Dependencies toevoegen:



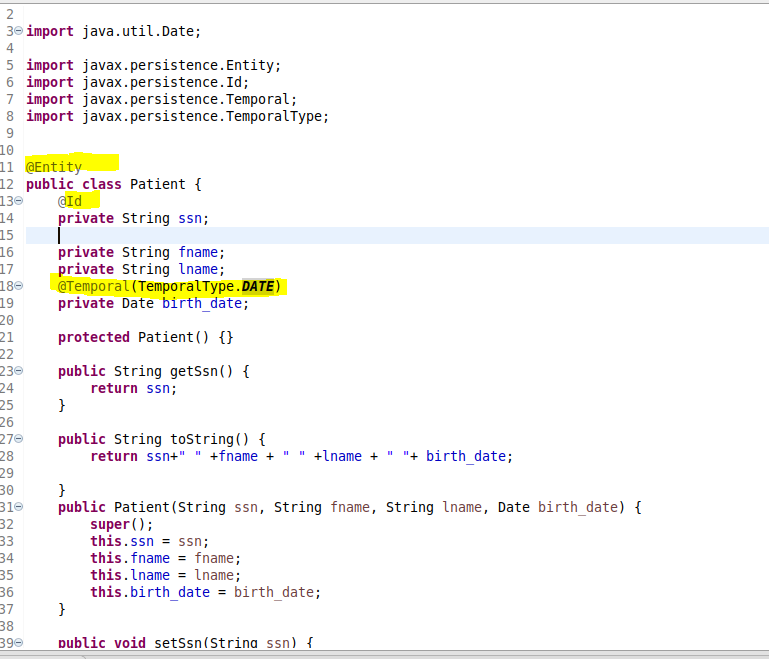


Right click project -> maven -> update project -> ok

# Entity

Klassen die moeten opgeslagen worden in de databank

@GeneratedValue(strategy=GenerationType.AUTO) indien je een auto id veld wilt



Getters/setters/constructors genereren: RightClick -> source -> kies ‘Generate ...’

Indien er meerdere objecten van iets behoren tot een klasse  
 @ManyToOne(fetch = FetchType.LAZY, optional = false)  
 @JoinColumn(name = "ward\_id", nullable = false)  
 @OnDelete(action = OnDeleteAction.CASCADE)  
Ward ward;  
Aan de andere kant:  
 @OneToMany(cascade = CascadeType.ALL, mappedBy="ward", fetch = FetchType.EAGER)  
List<Bed> beds;

# Database

Standaard voorzien CRUD methodes:

<S extends T> S save(S entity);

Optional<T> findById(ID primaryKey);

Iterable<T> findAll();

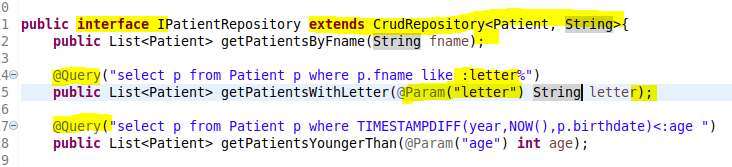
long count();

void delete(T entity);

boolean existsById(ID primaryKey);

# Repository

De basisqueries aanvullen met een paar die je zelf opstelt hier

Juiste notatie gebruiken! 

Andere mogelijkheidvoorbeelden:

Long countByX(

List<Klasse> deleteByX(

List<Klasse> findByX(  
List<Klasse> getByX(

@Nullable boven methode indien lege return value mag. Mag ook voor een optionele parameter staan.

Optional<Klasse> als return type dan geeft het Optional.empty() terug wanneer lege return value.

List<Person> find**Distinct**People**By**Lastname**Or**Firstname(String lastname, String firstname);

// Enabling ignoring case for an individual property

List<Person> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query

List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

REST

src/main/resources/application.properties

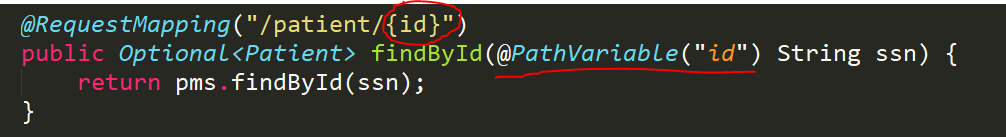


# Restcontroller

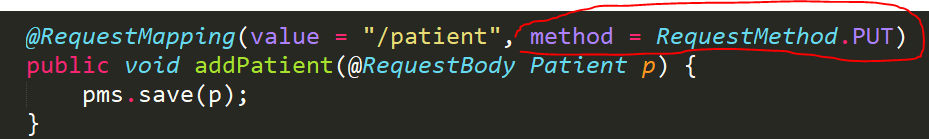
Klasse verantwoordelijk voor de rest-routing

Class annotation: @restController gebruiken

**GET methode:**



**PUT methode:**



# Autowired

@autowired boven een attribuut zetten voor het in te laden met dependancy injection

# MongoDB

***sudo service mongod start***

class annotation: @Document als het als document moet worden opgeslagen

public interface IFinanceRepository extends **MongoRepository<Invoice, String>**{}

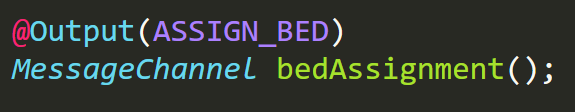
*<ClassToSave, IdType>*

# DeferredResult

Voor asyncrone data

# Output

Een bepaald bericht op de algemene channel zetten

String STRINGNAME = "assign\_bed";  
  
  
@output(STRINGNAME)  
MessageChannel methode();

De messages(links=) binden aan een bepaalde event (=rechts):

**application.properties:**

spring.cloud.stream.bindings.**assign\_bed.destination**=**assign\_bed** spring.cloud.stream.bindings.assign\_bed.contentType=application/json spring.cloud.stream.bindings.**check\_in\_result.destination**=**check\_in\_completed** spring.cloud.stream.bindings.check\_in\_result.contentType=application/json

**in main method class:**@EnableBinding(ProducerChannels.class)

# Gateway interface

de channel implementatie is zo verborgen wanneer je dat wilt gebruiken in uw code

Wordt later via @autowired gebruikt:

* Per type message een methode van wat ermee moet gebeuren

@MessagingGateway   
public interface MessageChannelGateway {   
 @Gateway(requestChannel = ProducerChannels.ASSIGN\_BED)   
 void assignBed(HospitalStay hospitalStay);   
  
 @Gateway(requestChannel = ProducerChannels.CHECK\_IN\_RESULT)   
 void emitHospitalCheckInCompleted(HospitalStay hospitalStay);  
 }

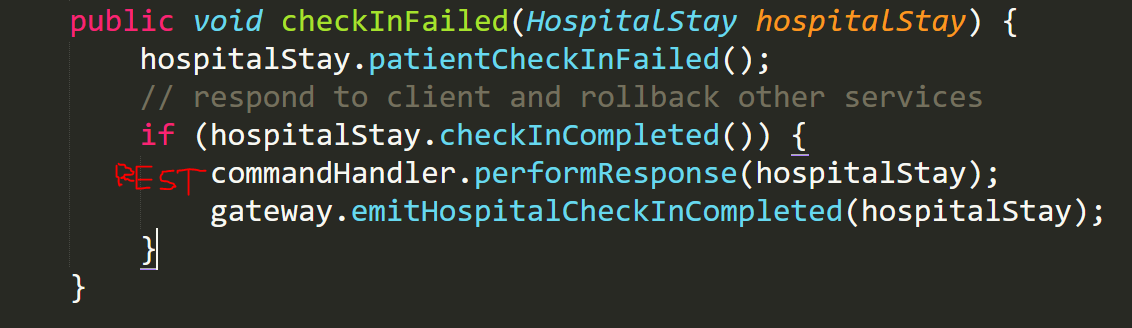
### Flow

RESTcontroller => applicatieService => Saga => functie uitvoeren met het domeinobejct

# Saga

Wordt aangeroepen door de (application)service wanneer iets binnenkomt bij de (application)service via de rest controller

* @Component boven klasse
* Via autowired de gateway(doorsluizen) en restcontroller(bericht teruggeven) opvragen
* Bevat aantal void operaties die elk een stap uit de flow afhandelen. Meestal altijd bepaalde functie van het object dat wordt meegegeven uitvoeren [en iets doorgeven aan de gateway]
* [Opt einde via de RestController een bericht terug aan de client geven]



# Application service

@component boven klasse

Bevat de saga. Vanuit de RestAPI wordt via deze klasse stukken van de saga opgeroepen.

Grootste nut van deze methodes is dat hier via de repository het juiste object wordt opgehaald op basis van een id en gecheckt wordt of het wel bestaat enzo. De uiteindelijke output gebeurd via de saga methodes zelf dus keert niet terug.

Soms wel de .save() van de repository oproepen wanneer attribuut veranderde

# Command handler

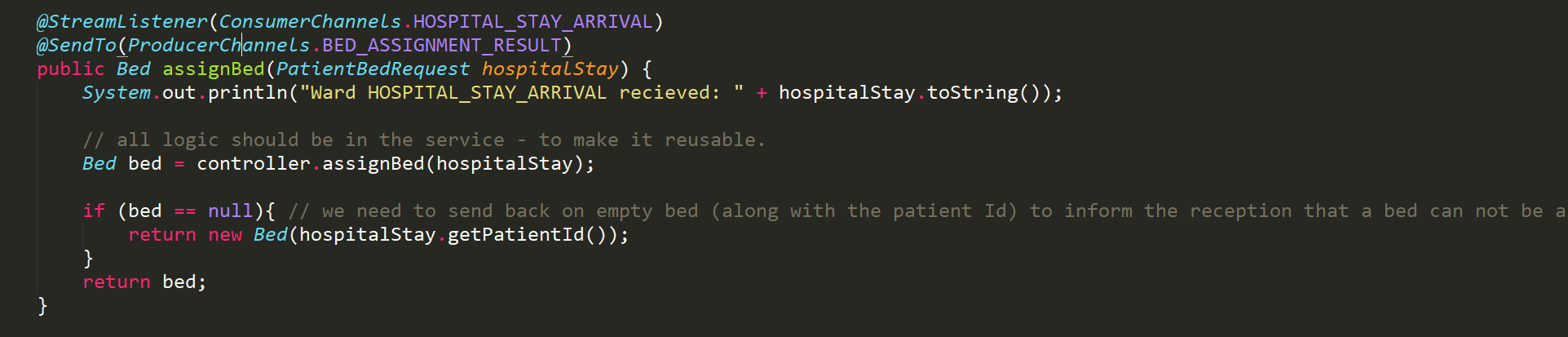
Beetje zelfde principe als de REST controller hiervoor maar nu komt request binnen in de vorm van een message. Request is een command (kan je weigeren)

Neemt een binnenkomende message en roept nodige methods op via de applicationservice

@streamlistener boven **methode** zodat het kan ontvangen van bepaalde type message

@sentTo voor het antwoord terug op de channel te zetten (zet bed id op kanaal met identificatie dat in die string staat van producerchannels. Dus ie zet wel degelijk da bed op de channel en niet enkel die string (in json))

* Ook PatientBedRequest klasse maken in het domein dat het bericht kan deserialiseren: bevat attributen (+get/set) voor alles nodig in de applicationservice

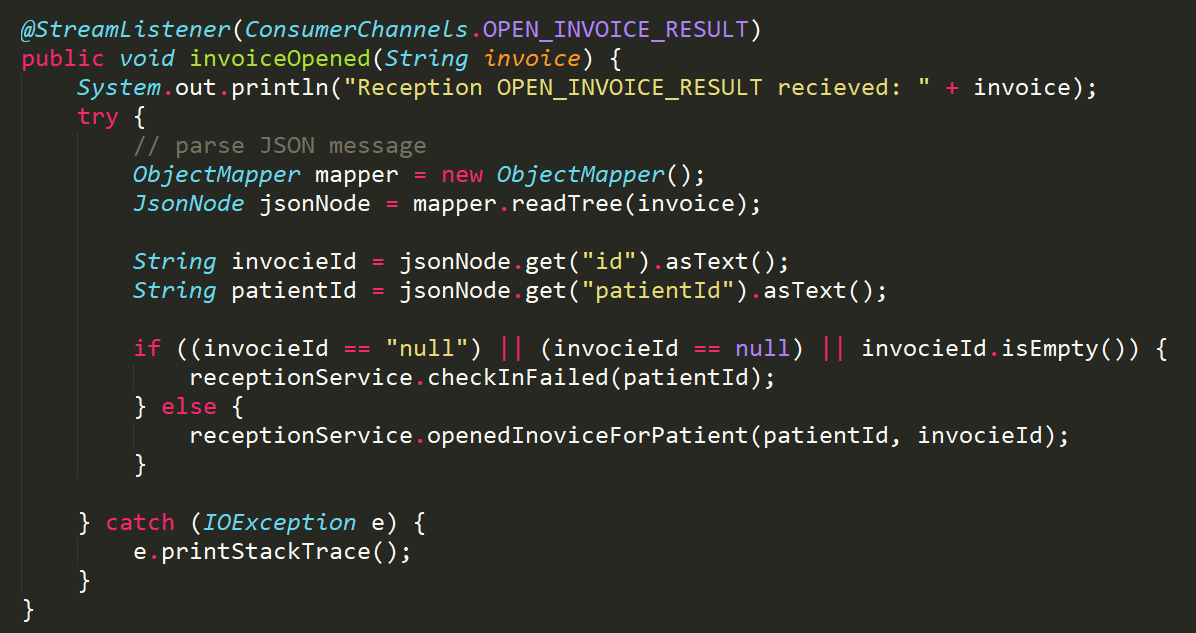


Geen logica hier, die komt in de service. Direct doorsluizen naar application service

# EventConsumer

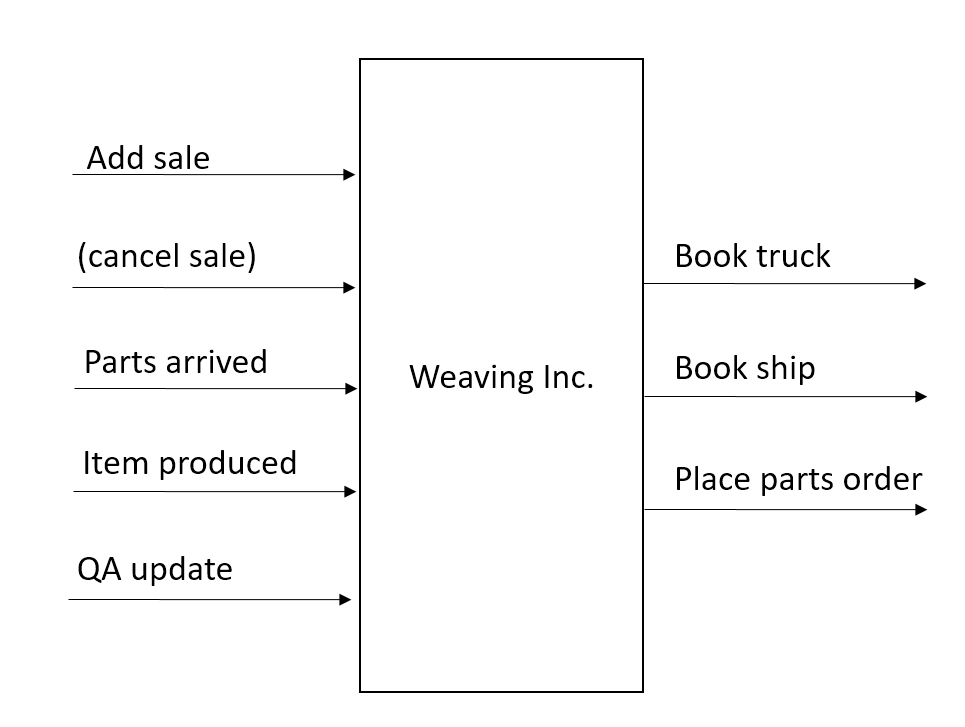
Verschil met command handler: dit is voor events (al gebeurd) op te vangen en verwerken

Bevat stream listeners die de json van de channel halen en dit doorsluizen naar de service

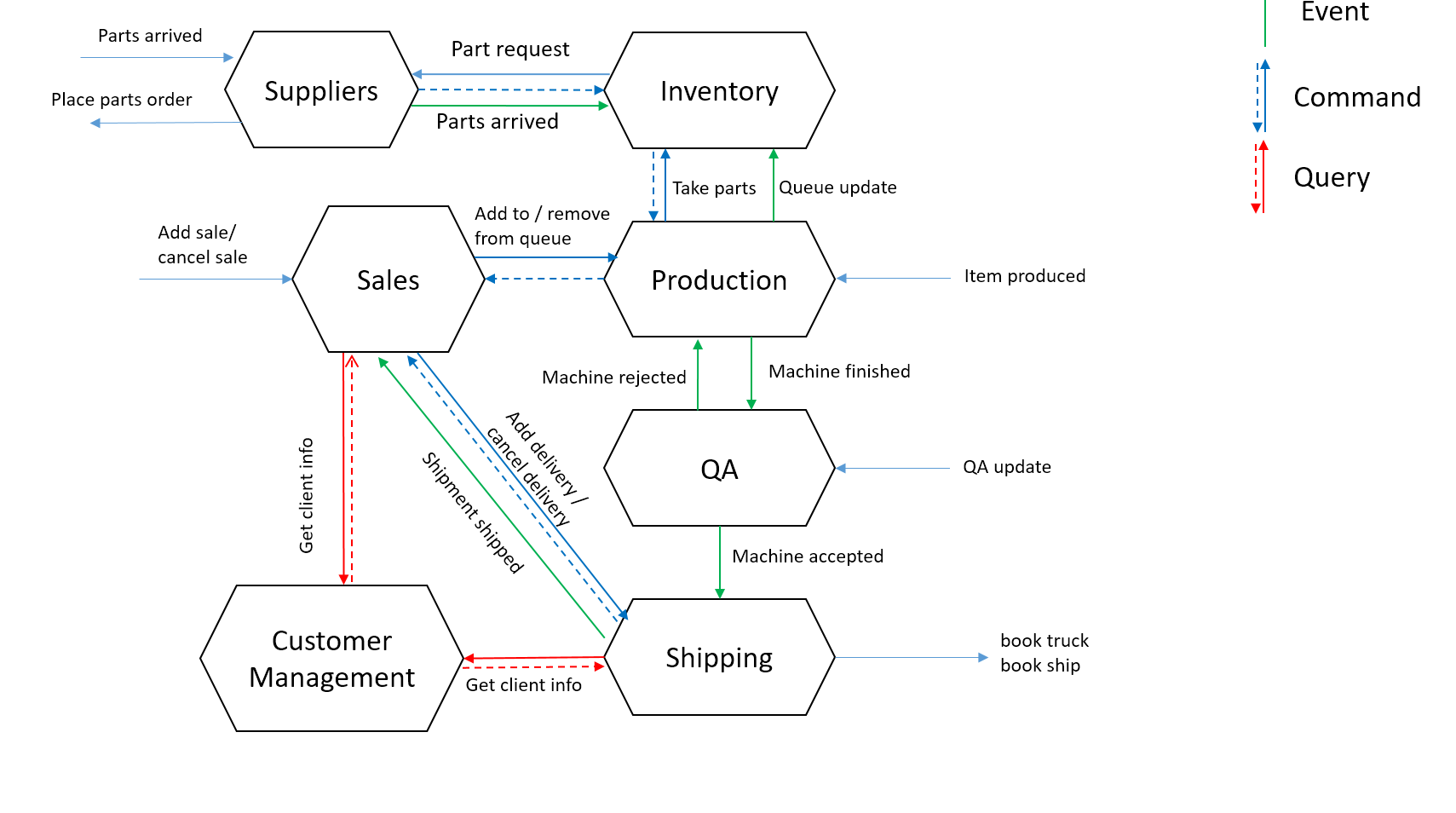


Labo 3

1. Systeem operaties bepalen. Systeem als blok zien

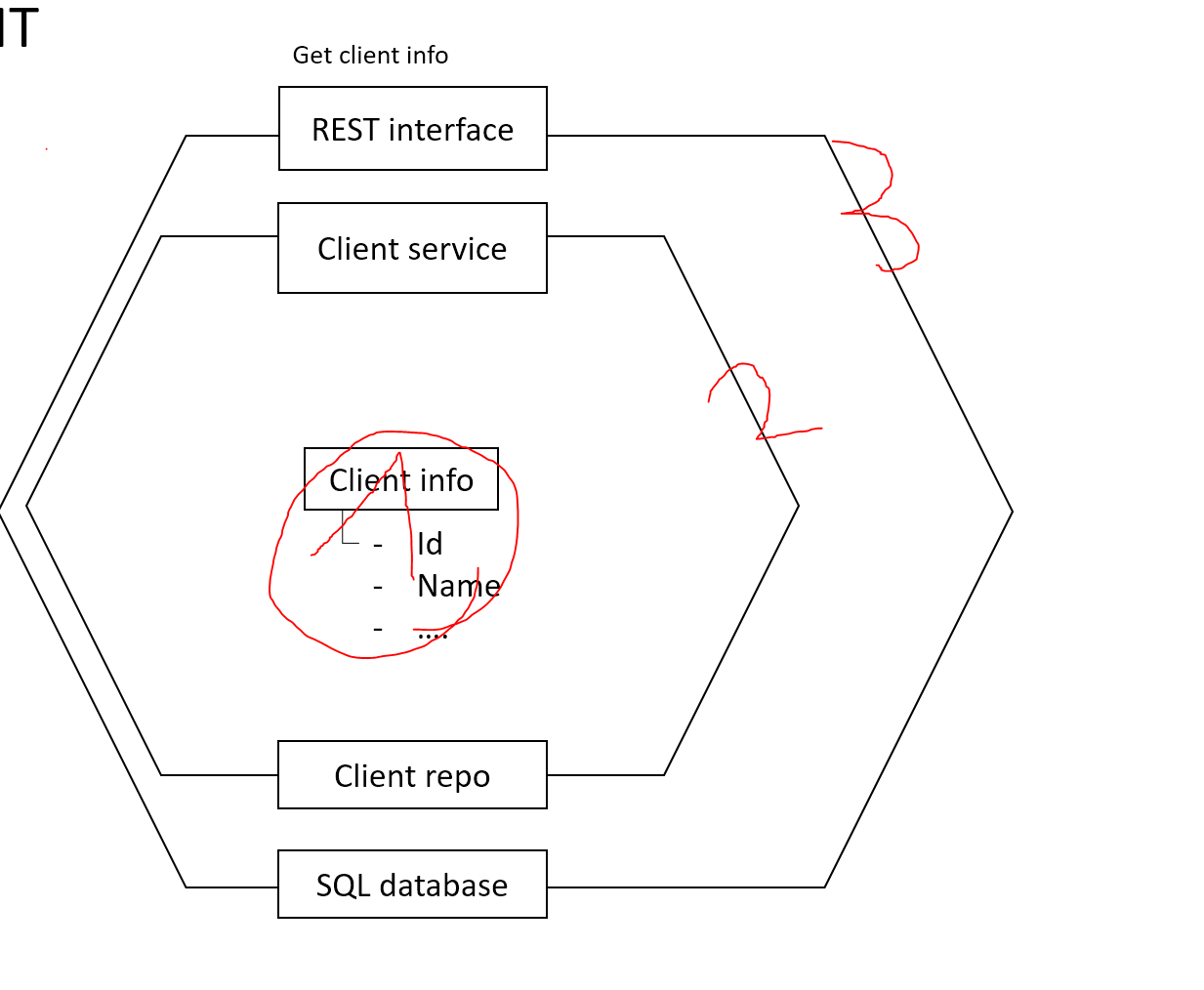


1. Services uit elkaar (in dit geval op basis fysieke departementen) halen en events/commands uitzoeken



1. Dan maak je per service een hexagonaal schema

3. contact met buitenwereld 2. hoe hij dingen intern doorsluist enzo 1. het domein model



Labo docker

# 3

***Docker ps –a*** alle containers bekijken

***docker run --name cont\_name im*** init de image img en geef het naam cont\_name  
***docker start …*** *herstarten****Docker containers prune*** alle containers verwijderen

# 4.1

***Docker pull mysql*** mysql container binnenhalen

***Docker run –d --name patient\_db mysql –e MYSQL\_ROOT\_PASSWORD=’root’ –e MYSQL\_DATABASE=’patientDB’***

**-d** voor detach (background) **-e** environment variable

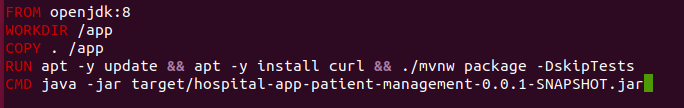
Hub.docker.com/\_/mysql environment variabeles beneden

***Docker inspect patient\_db | grep “Env” –C 10*** controle

***Docker exec –i -t patient\_db /bin/bash***   
-i en dan kunje linux commando’s runnen (exec /bin/bash in patient\_db op interactieve manier)

# 4.2

***Touch Dockerfile  
Nano Dockerfile***

Workdir: welke directory in uw container  
copy: copy huidige directory in de app  
Run: update/download curl + build applicatie met maven -y geen interactie   
CMD: de jar dat build gemaakt heeft uitvoeren  


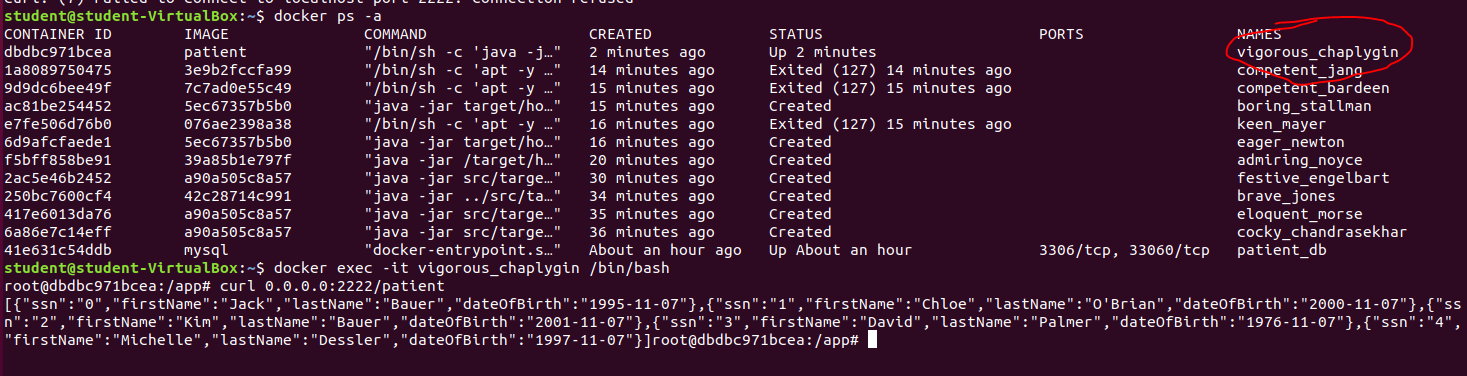
\* mag maar 1 CMD staan

\* fout in opgave, tis target/ en niet src/target

\* geen cmd [“ “] doen ma zoals het er staat

In folder waar dockerfile staat:  
***docker build –t patient .*** dien image aanmaken in docker

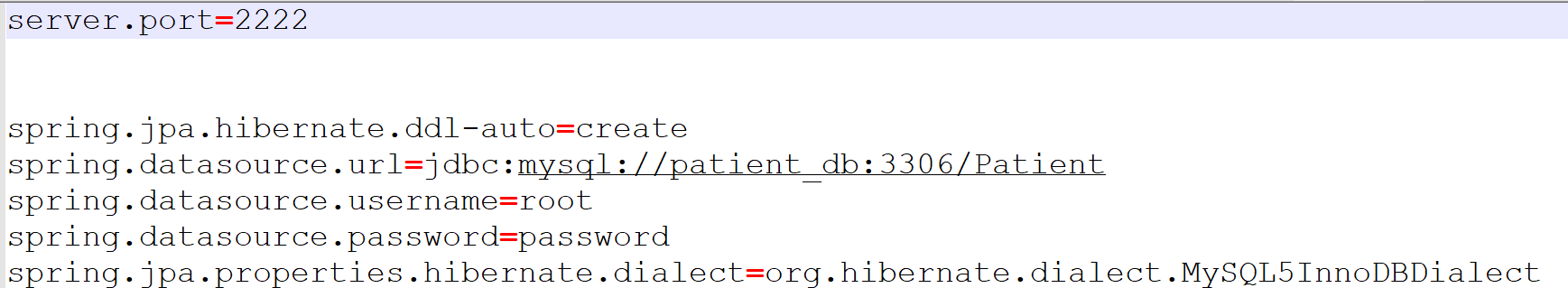
***docker run patient*** dien image runnen



***curl 0.0.0.0:2222/patient*** testen binnen de container

# 4.2.2 SQL koppelen

Application.properties voor mysql



***Mysql://CONTAINERNAME::3360/DATABANKNAME***

Docker builden en runnen dan terug: --link optie gebruiken voor mysql te linken

Kijken of de tabel gepopulate is:   
***docker exec –i mysql –u root –p –e “SELECT \* FROM Patient.patient”***

Patient toevoegen: (als ge 2 containers aant runnen zijt zou in beide beschikbaar moeten zijn)  
***curl 0.0.0.0:2222/patient -X PUT -H "ContentType:application/json" -d "{\"ssn\":\"3\",\"firstName\":\"Tom\",\"lastName\":\"Palmer\", \"dateOfBirth\":\"2001-07-25\"}"***

# 4.3

Bind-mount = bestand of map wordt gemount in een container

In de main klasse ipv localhost => <http://patient:2222>

Jar maken van Gateway API:   
***./mvnw package –DskipTests***

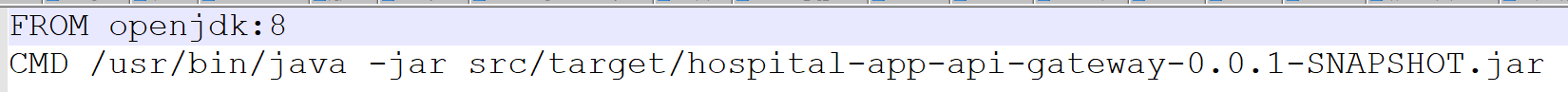
*Dockerfile maken****:***

Image builden/runnen: in commando opgeven: name container, linken patient container, volume flag om de folder met de applicatie te mounten op de container

Port instellen met –p optie of --expose –P gebruiken voor automatisch

Commando: TODO

# 4.4 compose

Meerdere containers met 1 commando starten

***Docker-compose up***

Compose file maken:

* Volgorde belangrijk
* Nodige opties instellen per container
* Hoe images bouwen wanneer de compose file wordt uitgevoerd

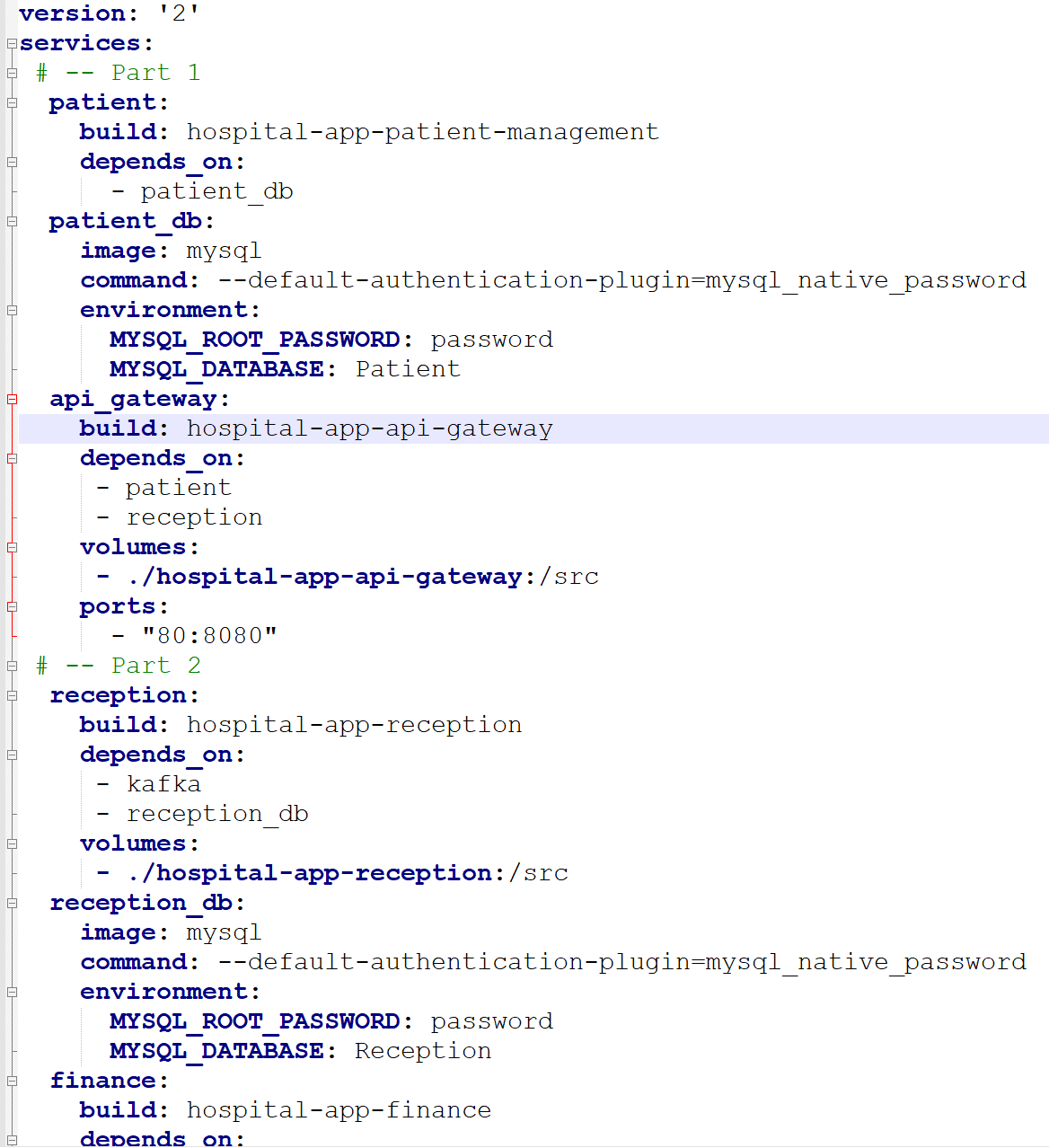
Finance heeft mongoDB nodig:

1. Mongo image downloaden en container toevoegen in de compose file
2. Finance container linken aan die container
3. In finance code hostname aanpassen: spring.data.mongodb.host=mongo

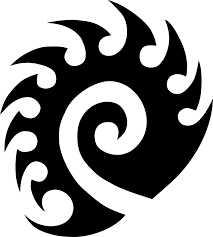
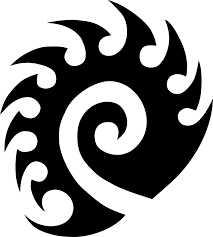
Reception en ward services hebben SQL nodig

Reception/ward/Finance communiceren via kafka/zookeeper containers. Dus die containers downloaden in composer file ook zetten.   
in properties file telkens:  
o spring.cloud.stream.kafka.binder.brokers=kafka   
o spring.cloud.stream.kafka.binder.zkNodes=zookeeper

**Composer file**





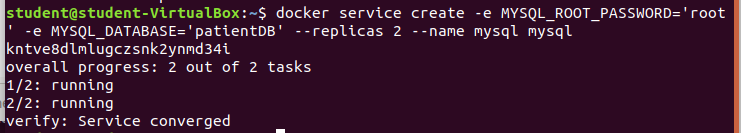
Docker Swarm labo

# 1 swarm initialiseren

Intialise master swarm: **docker swarm init**

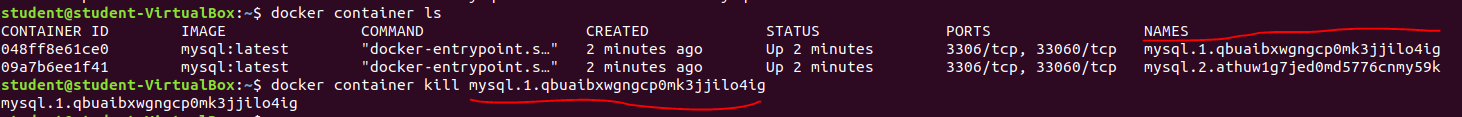
Uw token terug ophalen **docker swarm join-token worker**

Service met 2 replicas maken:



Checken of het runned: **docker service ls**   
op welke nodes draaien de tasks: **docker service ps mysql**

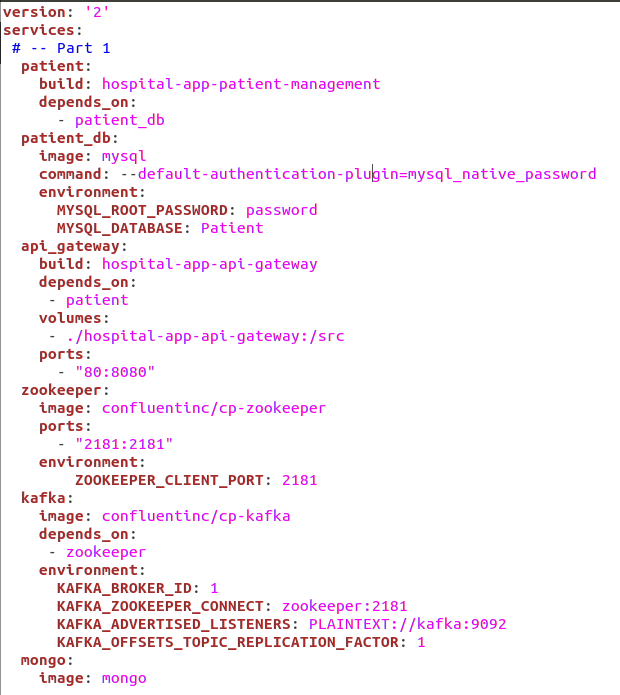
ééntje (replica) wegdoen:



Service volledig wegdoen: docker service rm mysql

# 2 vorigen brol wegdoen

Reception / ward wegdoen in composer file en dependency van reception da lijntje ook ergens. Want we gebruiken in eerste deel enkel Patient / patient DB / API gateway



Docker-compose up --build jars maken en runnen

# 2.2 compose omzetten in iets bruikbaar voor swarm

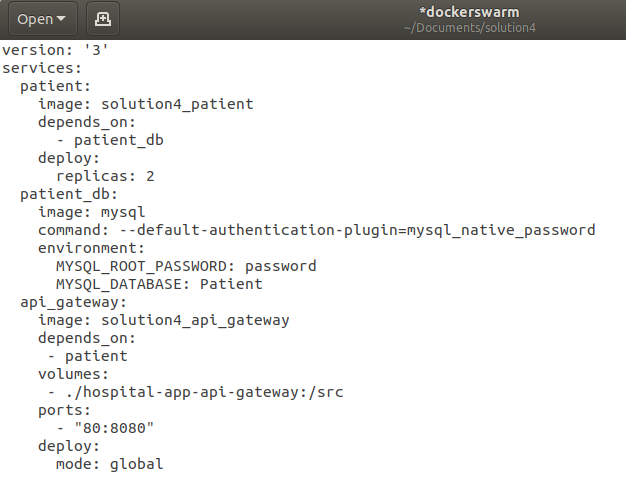
images maken **Docker-compose –f docker-compose.yml build**

checken voor hoe de gemaakt images noemen **Docker image ls**

(Build wegdoen in compose file)

Docker(compose)swarm bestand:

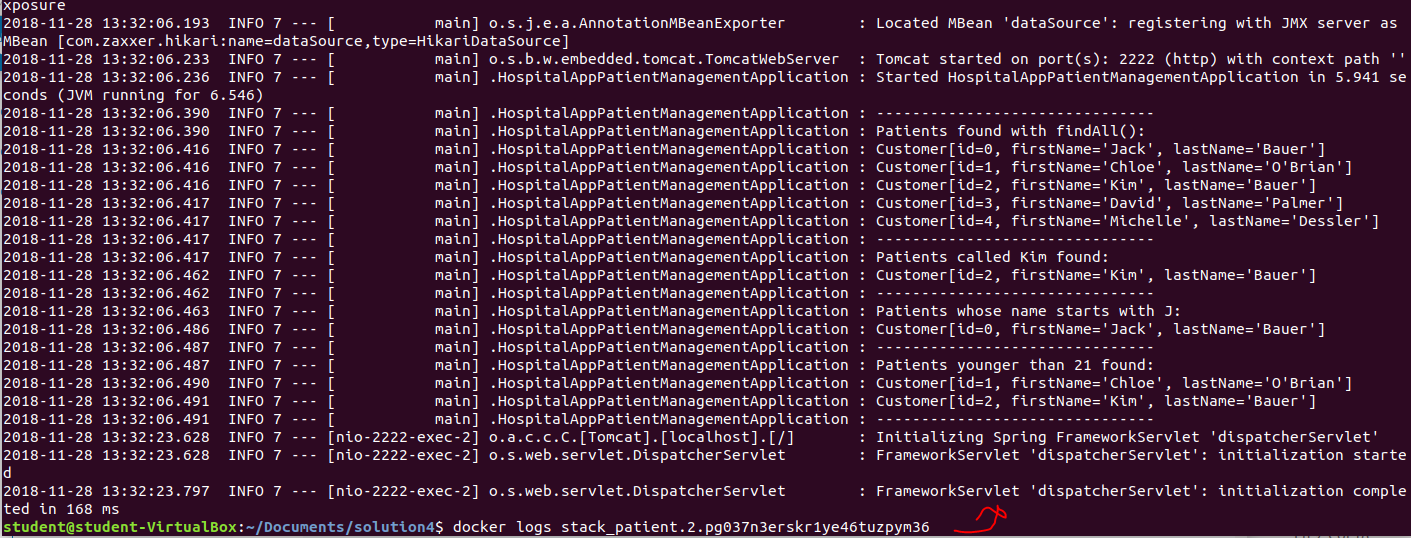
* 3 services maken altijd: patient, patient\_db, api\_gateway
* Patient: image naam (docker image ls), zeggen wat het nodig heeft (de databank), 2 replica’s maken bij ht deployen
* Patient\_db: niet vergeten de environment variabelen in te stellen
* Api\_gateway: heeft de patient service nodig, spreekt zelf natuurlijk niet onmiddellijk als gateway met de databank. Poorten ook instellen waar ie luistert.



Docker swarm opstarten op basis van het bestand:  
**Docker stack deploy –compose-file dockerswarm stack**

Testen **curl** [**http://localhost/patient**](http://localhost/patient)

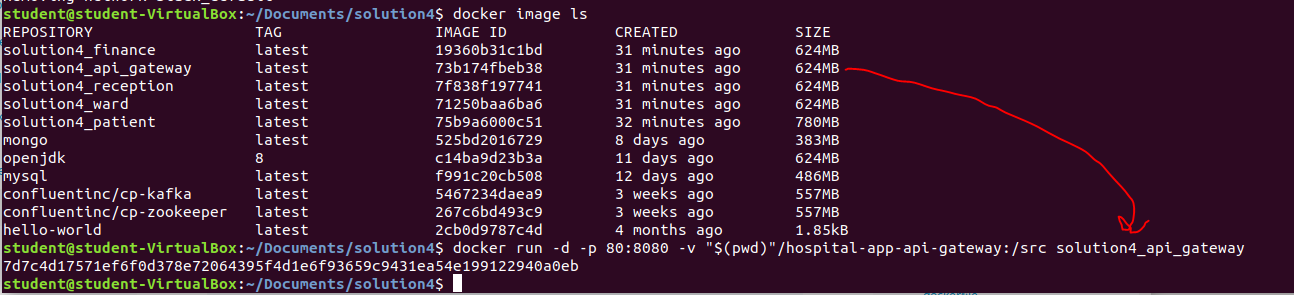
Log bestandn bekijken:



# 2.3 Circuit breaker

onze vorige stack uitzetten **Docker stack rm stack**

Image naam checken voor dien individuele application te launchen(want we willen zien wat hij zegt als patient niet beschikbaar is): docker image ls en dan docker run



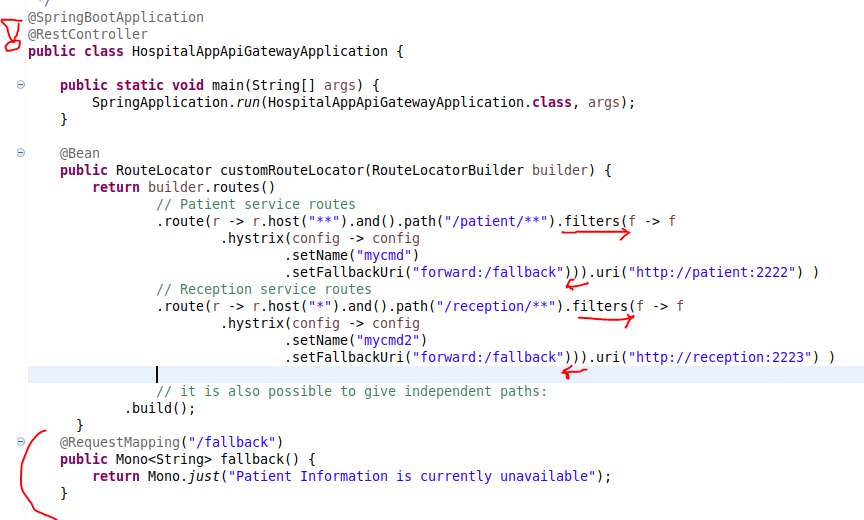
Nu zien dat hij iets deftig toont via netflix dependency (hystrix)

## Dependency toevoegen in api\_gateway:



## Fallback in api gateway main class toevoegen:

(Die pijltjes zeggen vanaf waar we het zelf moesten toevoegen)



Vorige stoppe: (docker container ls)  
docker container stop 7d7c…

Opnieuw jar hermaken:   
naar api\_gateway root folder gaan 🡺 ./mvnw package –DskipTests 🡺 cd ..

Gateway applicatie starten: (zien da je da **doet in de directory waar dockerswarm** bestand staat) (“Dat is de root directory van het project” – Bert de Saffel)





# 3 Messaging services (vanaf hier thuis afgewerkt)

Terug alles toevoegen aan de composer file en dan builden en runnen.   
in de code: >>Application.properties<<  
Group property: maar **één** van de wards bv moet de **request accepteren**. **Niet alle** ward replica’s moeten dat dan doen  
Spring.cloud.stream.bindings.open\_invoice\_cmd.**group**=open\_invoice\_group

Applicatie testen **http://localhost/reception/check\_in\_patient?patientId=1**

# 4 monitor and manage application

meten we hoe lang het duurt voor requests op te lossen. Te lang? Dan meer instances toevoegen.

Met Jmeter kunnen we requests generaten.

* Spring-boot-starter-actuator dependency toevoegen
* Micrometer-core / micrometer-registry-prometheus dependecies toevoegen (alles samenvoegen van metrics van verschillende containers)
* Aan applications.properties toevoegen:

management.security.enabled=false management.endpoints.web.exposure.include=\* management.endpoint.metrics.enabled=true management.endpoint.prometheus.enabled=true management.metrics.export.prometheus.enabled=true

services bekijken service ps stacknaam\_servicenaam

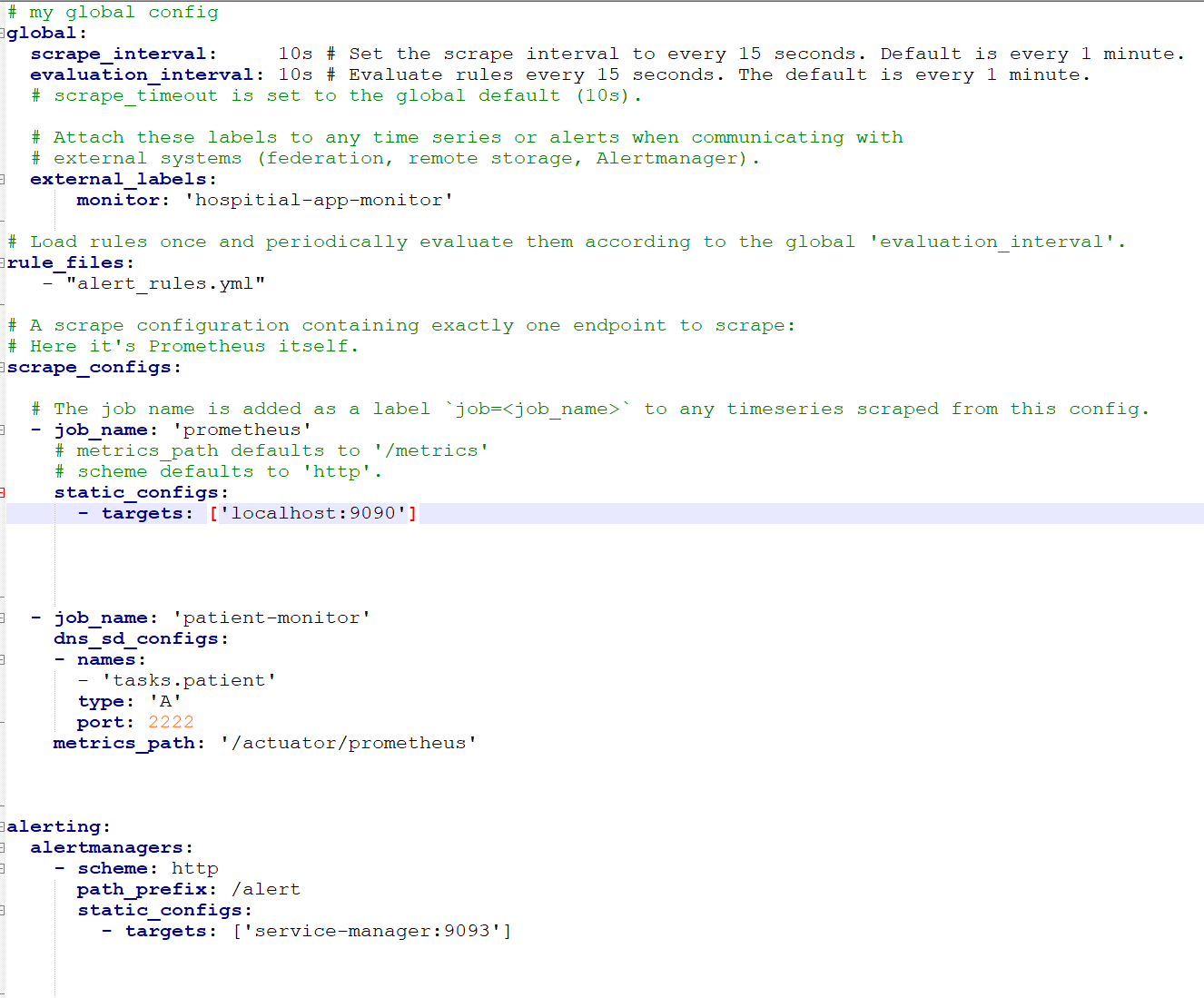
* Opnieuw specifieke image builden maar met :v2 achter de naam
* Patient service updaten zodat hij die nieuwe image gebruikt:
* docker service update [OPTIONS] SERVICE  
  opties zie: https://docs.docker.com/engine/reference/commandline/service\_update/

--update-delay 2000ms

--rolback

Testen **curl 0.0.0.0:2222/actuator**

## Prometheus.yml file maken



Dan prometheus service toevoegen aan docker compose file

Prometheus image moeje van het internet downloaden.

Testen <http://localhost:9090/targets>

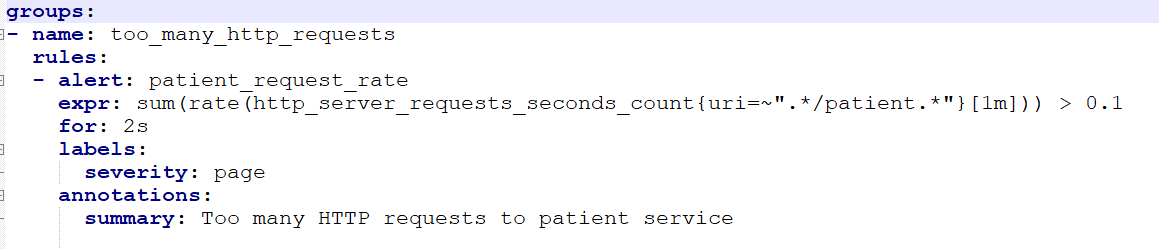
# 4.2

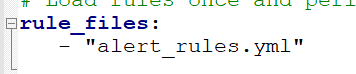
./bin/jmeter.sh (usage zie tutorial website)

…. Cirit image todo

# 4.3

Alerts.yml maken (getriggered wanneer gemiddelijke http request langer dan 0.1 seconde duurt)



Naar verwijzen in de prometheus.yml  
 

Wanneer dat getriggerd wordt dan moet het scalen

# Scaling

To do