# 렌트카계약(CarRental)

# Table of contents

* [렌트카](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#---)
  + [서비스 시나리오](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EC%84%9C%EB%B9%84%EC%8A%A4-%EC%8B%9C%EB%82%98%EB%A6%AC%EC%98%A4)
  + [체크포인트](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EC%B2%B4%ED%81%AC%ED%8F%AC%EC%9D%B8%ED%8A%B8)
  + [분석/설계](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EB%B6%84%EC%84%9D%EC%84%A4%EA%B3%84)
  + [구현:](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EA%B5%AC%ED%98%84-)
    - [DDD 의 적용](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#ddd-%EC%9D%98-%EC%A0%81%EC%9A%A9)
    - [GateWay 적용](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#GateWay-%EC%A0%81%EC%9A%A9)
    - [CQRS/saga/correlation](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%ED%8F%B4%EB%A6%AC%EA%B8%80%EB%9E%8F-%ED%8D%BC%EC%8B%9C%EC%8A%A4%ED%84%B4%EC%8A%A4)
    - [동기식 호출 과 Fallback 처리](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EB%8F%99%EA%B8%B0%EC%8B%9D-%ED%98%B8%EC%B6%9C-%EA%B3%BC-Fallback-%EC%B2%98%EB%A6%AC)
    - [비동기식 호출 과 Eventual Consistency](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EB%B9%84%EB%8F%99%EA%B8%B0%EC%8B%9D-%ED%98%B8%EC%B6%9C-%EA%B3%BC-Eventual-Consistency)
    - [폴리글랏 퍼시스턴스](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%ED%8F%B4%EB%A6%AC%EA%B8%80%EB%9E%8F-%ED%8D%BC%EC%8B%9C%EC%8A%A4%ED%84%B4%EC%8A%A4)
  + [운영](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EC%9A%B4%EC%98%81)
    - [CI/CD 설정](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#cicd%EC%84%A4%EC%A0%95)
    - [Deploy / Pipeline](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#Deploy-Pipeline)
    - [ConfigMap](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#ConfigMap)
    - [동기식 호출 / 서킷 브레이킹 / 장애격리](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EB%8F%99%EA%B8%B0%EC%8B%9D-%ED%98%B8%EC%B6%9C-%EC%84%9C%ED%82%B7-%EB%B8%8C%EB%A0%88%EC%9D%B4%ED%82%B9-%EC%9E%A5%EC%95%A0%EA%B2%A9%EB%A6%AC)
    - [오토스케일 아웃](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EC%98%A4%ED%86%A0%EC%8A%A4%EC%BC%80%EC%9D%BC-%EC%95%84%EC%9B%83)
    - [무정지 재배포](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#%EB%AC%B4%EC%A0%95%EC%A7%80-%EC%9E%AC%EB%B0%B0%ED%8F%AC)
    - [Liveness Probe](https://github.com/leo99k/carrental/blob/main/README_%EC%9D%B4%EB%8F%99%ED%9D%AC.md#Liveness-Probe)

# 서비스 시나리오

기능적 요구사항

1. 고객이 차량을 선택하여 계약(Contract)한다.
2. 고객이 결제(Pay)한다.
3. 계약이 되면 계약 내역이 조회 페이지(MyPage)로 전달된다.
4. 결제가 완료되면 차량 예약(Reservation)이 완료된다.
5. 고객이 계약을 취소할 수 있다.
6. 계약이 취소되면 차량 예약이 취소된다.
7. 고객이 결제 및 예약 상태를 한 화면에서 확인 할 수 있다.
8. **계약이 완료 또는 취소한 정보를 고객관리로 전달한다.**

비기능적 요구사항

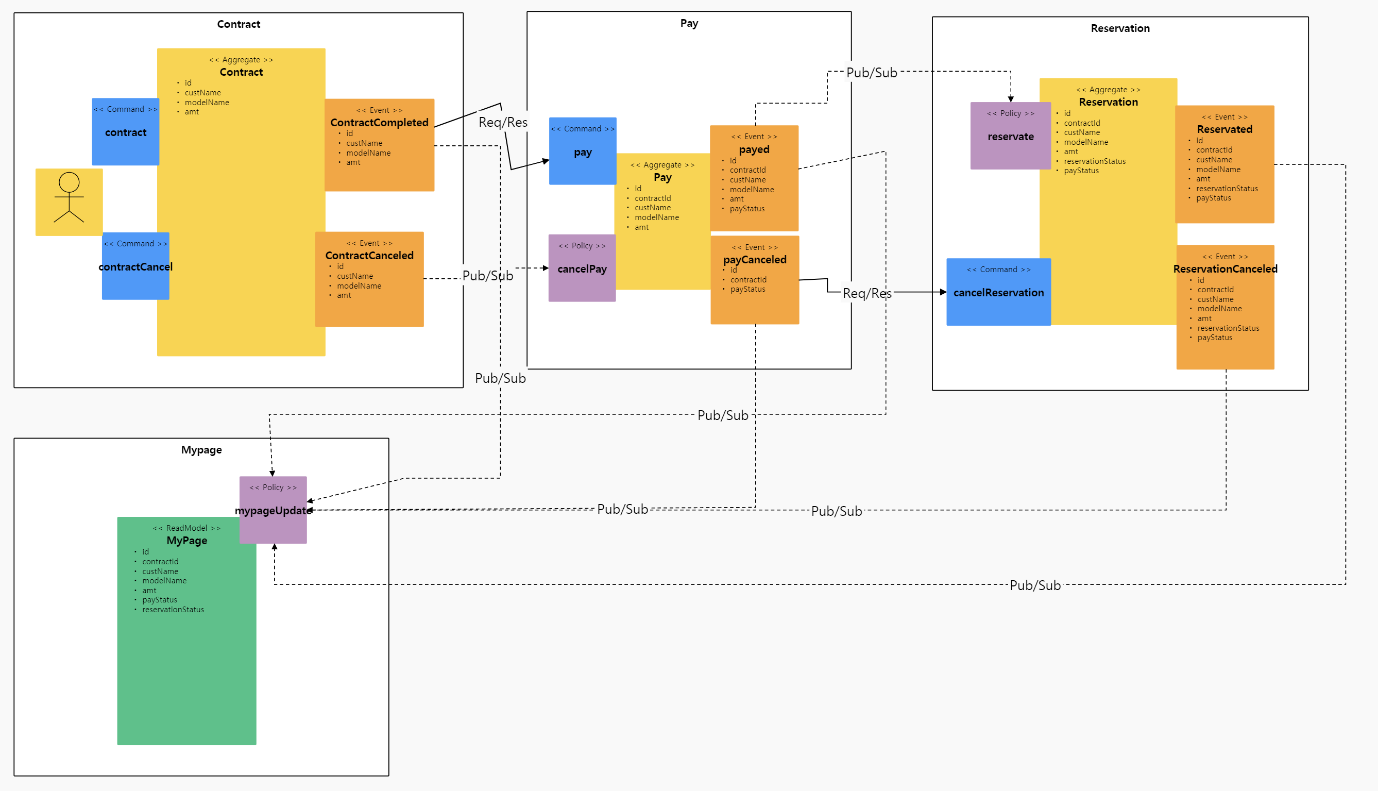
* 트랜잭션
  + 계약이 되지 않은 결제는 아예 성립되지 않아야 한다 Sync 호출
* 장애격리
  + 예약 기능이 수행되지 않더라도 계약은 365일 24시간 받을 수 있어야 한다 Async (event-driven), Eventual Consistency
  + 결제시스템이 과중되면 사용자를 잠시동안 받지 않고 결제를 잠시후에 하도록 유도한다 Circuit breaker, fallback
* 성능
  + 계약자가 일련의 차량 계약에 대한 상태를 조회 화면(MyPage)에서 확인할 수 있어야 한다 CQRS

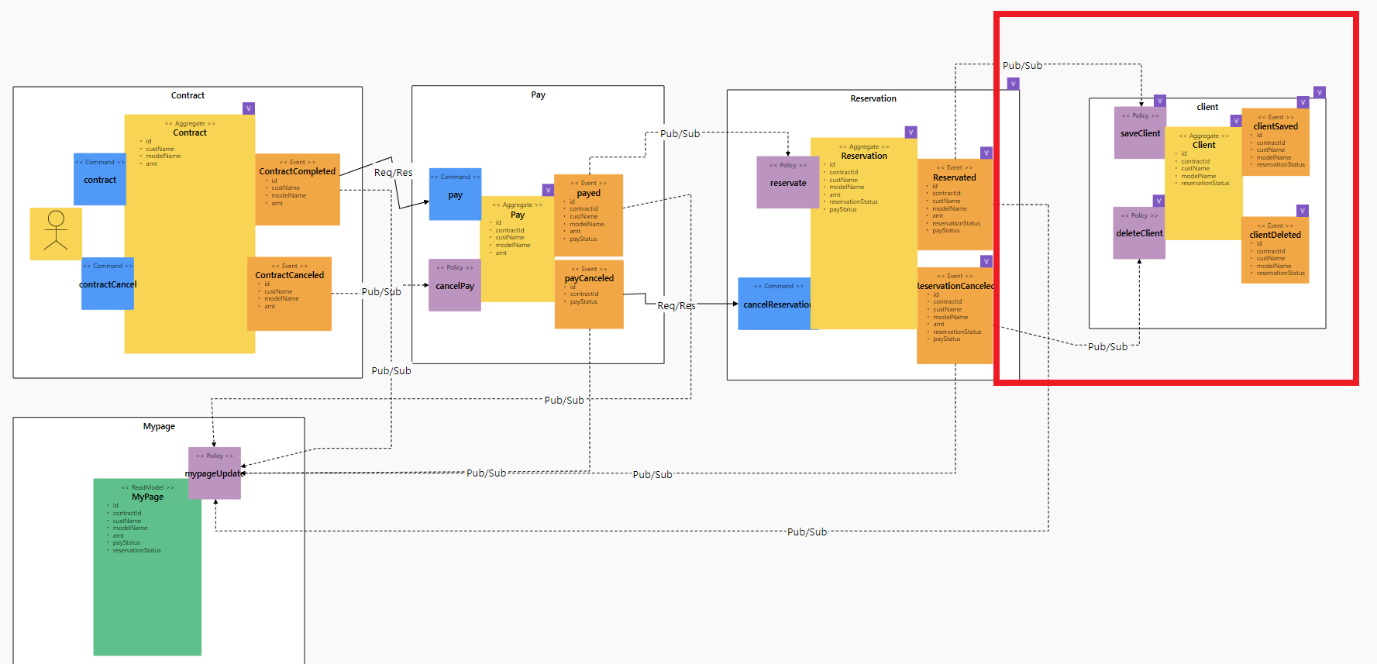
# 체크포인트

* 체크포인트 : <https://workflowy.com/s/assessment-check-po/T5YrzcMewfo4J6LW>
* 분석 설계
  + 이벤트스토밍:
    - 스티커 색상별 객체의 의미를 제대로 이해하여 헥사고날 아키텍처와의 연계 설계에 적절히 반영하고 있는가?
    - 각 도메인 이벤트가 의미있는 수준으로 정의되었는가?
    - 어그리게잇: Command와 Event 들을 ACID 트랜잭션 단위의 Aggregate 로 제대로 묶었는가?
    - 기능적 요구사항과 비기능적 요구사항을 누락 없이 반영하였는가?
  + 서브 도메인, 바운디드 컨텍스트 분리
    - 팀별 KPI 와 관심사, 상이한 배포주기 등에 따른  Sub-domain 이나 Bounded Context 를 적절히 분리하였고 그 분리 기준의 합리성이 충분히 설명되는가?
      * 적어도 3개 이상 서비스 분리
    - 폴리글랏 설계: 각 마이크로 서비스들의 구현 목표와 기능 특성에 따른 각자의 기술 Stack 과 저장소 구조를 다양하게 채택하여 설계하였는가?
    - 서비스 시나리오 중 ACID 트랜잭션이 크리티컬한 Use 케이스에 대하여 무리하게 서비스가 과다하게 조밀히 분리되지 않았는가?
  + 컨텍스트 매핑 / 이벤트 드리븐 아키텍처
    - 업무 중요성과  도메인간 서열을 구분할 수 있는가? (Core, Supporting, General Domain)
    - Request-Response 방식과 이벤트 드리븐 방식을 구분하여 설계할 수 있는가?
    - 장애격리: 서포팅 서비스를 제거 하여도 기존 서비스에 영향이 없도록 설계하였는가?
    - 신규 서비스를 추가 하였을때 기존 서비스의 데이터베이스에 영향이 없도록 설계(열려있는 아키택처)할 수 있는가?
    - 이벤트와 폴리시를 연결하기 위한 Correlation-key 연결을 제대로 설계하였는가?
  + 헥사고날 아키텍처
    - 설계 결과에 따른 헥사고날 아키텍처 다이어그램을 제대로 그렸는가?
* 구현
  + [DDD] 분석단계에서의 스티커별 색상과 헥사고날 아키텍처에 따라 구현체가 매핑되게 개발되었는가?
    - Entity Pattern 과 Repository Pattern 을 적용하여 JPA 를 통하여 데이터 접근 어댑터를 개발하였는가
    - [헥사고날 아키텍처] REST Inbound adaptor 이외에 gRPC 등의 Inbound Adaptor 를 추가함에 있어서 도메인 모델의 손상을 주지 않고 새로운 프로토콜에 기존 구현체를 적응시킬 수 있는가?
    - 분석단계에서의 유비쿼터스 랭귀지 (업무현장에서 쓰는 용어) 를 사용하여 소스코드가 서술되었는가?
  + Request-Response 방식의 서비스 중심 아키텍처 구현
    - 마이크로 서비스간 Request-Response 호출에 있어 대상 서비스를 어떠한 방식으로 찾아서 호출 하였는가? (Service Discovery, REST, FeignClient)
    - 서킷브레이커를 통하여  장애를 격리시킬 수 있는가?
  + 이벤트 드리븐 아키텍처의 구현
    - 카프카를 이용하여 PubSub 으로 하나 이상의 서비스가 연동되었는가?
    - Correlation-key: 각 이벤트 건 (메시지)가 어떠한 폴리시를 처리할때 어떤 건에 연결된 처리건인지를 구별하기 위한 Correlation-key 연결을 제대로 구현 하였는가?
    - Message Consumer 마이크로서비스가 장애상황에서 수신받지 못했던 기존 이벤트들을 다시 수신받아 처리하는가?
    - Scaling-out: Message Consumer 마이크로서비스의 Replica 를 추가했을때 중복없이 이벤트를 수신할 수 있는가
    - CQRS: Materialized View 를 구현하여, 타 마이크로서비스의 데이터 원본에 접근없이(Composite 서비스나 조인SQL 등 없이) 도 내 서비스의 화면 구성과 잦은 조회가 가능한가?
  + 폴리글랏 플로그래밍
    - 각 마이크로 서비스들이 하나이상의 각자의 기술 Stack 으로 구성되었는가?
    - 각 마이크로 서비스들이 각자의 저장소 구조를 자율적으로 채택하고 각자의 저장소 유형 (RDB, NoSQL, File System 등)을 선택하여 구현하였는가?
  + API 게이트웨이
    - API GW를 통하여 마이크로 서비스들의 집입점을 통일할 수 있는가?
    - 게이트웨이와 인증서버(OAuth), JWT 토큰 인증을 통하여 마이크로서비스들을 보호할 수 있는가?
* 운영
  + SLA 준수
    - 셀프힐링: Liveness Probe 를 통하여 어떠한 서비스의 health 상태가 지속적으로 저하됨에 따라 어떠한 임계치에서 pod 가 재생되는 것을 증명할 수 있는가?
    - 서킷브레이커, 레이트리밋 등을 통한 장애격리와 성능효율을 높힐 수 있는가?
    - 오토스케일러 (HPA) 를 설정하여 확장적 운영이 가능한가?
    - 모니터링, 앨럿팅:
  + 무정지 운영 CI/CD (10)
    - Readiness Probe 의 설정과 Rolling update을 통하여 신규 버전이 완전히 서비스를 받을 수 있는 상태일때 신규버전의 서비스로 전환됨을 siege 등으로 증명
    - Contract Test : 자동화된 경계 테스트를 통하여 구현 오류나 API 계약위반를 미리 차단 가능한가?

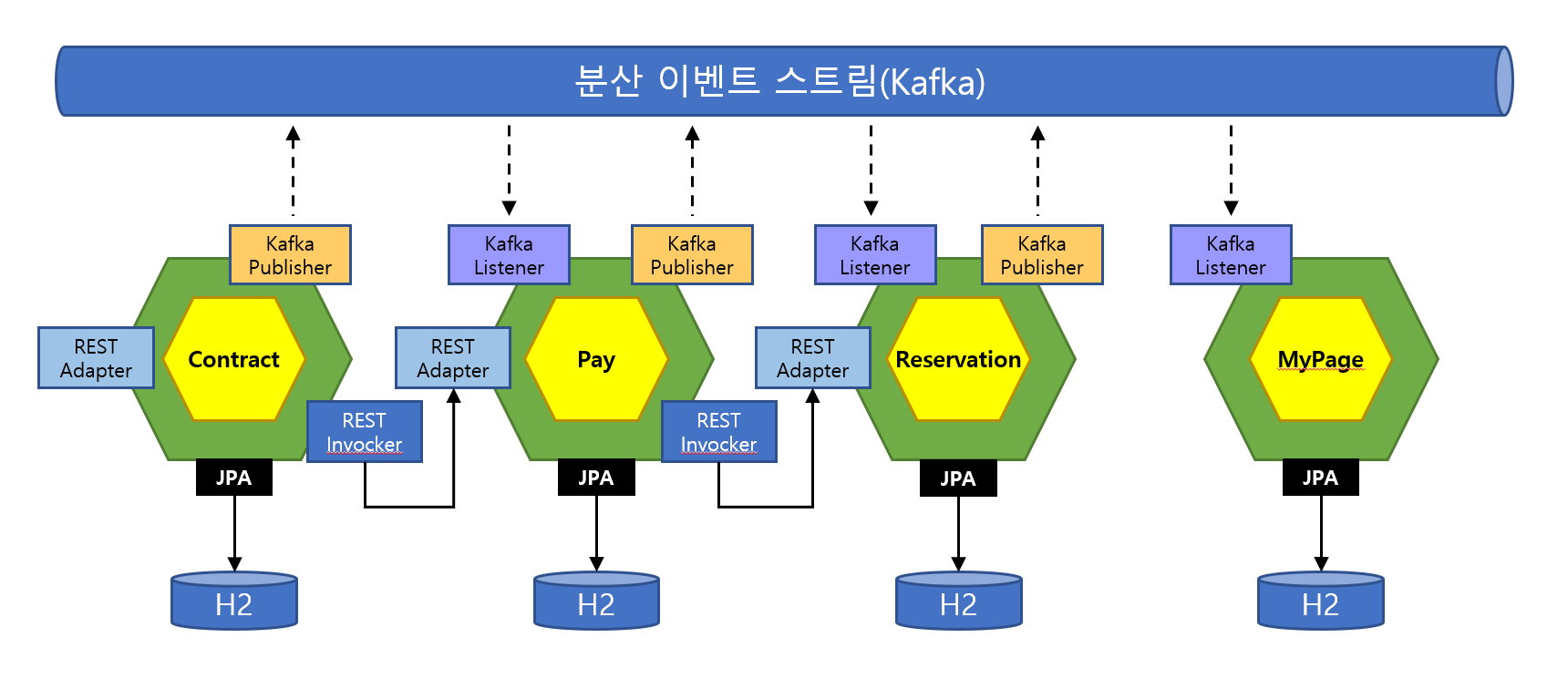
# 분석/설계

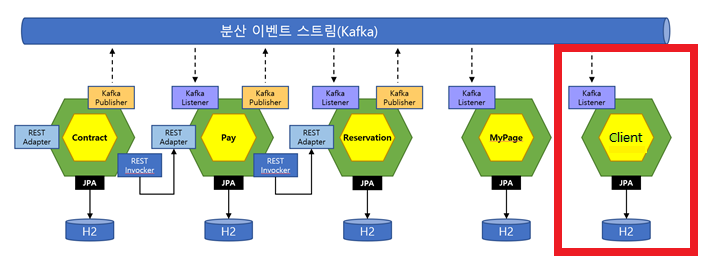
## Event Storming 결과





## 헥사고날 아키텍처 다이어그램 도출





- Chris Richardson, MSA Patterns 참고하여 Inbound adaptor와 Outbound adaptor를 구분함

- 호출관계에서 PubSub 과 Req/Resp 를 구분함

- 서브 도메인과 바운디드 컨텍스트의 분리: 각 팀의 KPI 별로 아래와 같이 관심 구현 스토리를 나눠가짐

# 구현:

분석/설계 단계에서 도출된 헥사고날 아키텍처에 따라, 각 BC별로 대변되는 마이크로 서비스들을 스프링부트로 구현하였다. 구현한 각 서비스를 로컬에서 실행하는 방법은 아래와 같다 (각자의 포트넘버는 8081 ~ 808n 이다)

cd C:\projects\carrental\Contract

mvn spring-boot:run

cd C:\projects\carrental\Pay

mvn spring-boot:run

cd C:\projects\carrental\Reservation

mvn spring-boot:run

cd C:\projects\carrental\Mypage

mvn spring-boot:run

cd C:\projects\carrental\gateway

mvn spring-boot:run

**cd C:\projects\carrental\Client**

**mvn spring-boot:run**

## DDD 의 적용

* 각 서비스내에 도출된 핵심 Aggregate Root 객체를 Entity 로 선언하였다: (예시는 Contract 마이크로 서비스).

package carrental;

import javax.persistence.\*;

import org.springframework.beans.BeanUtils;

import carrental.external.Pay;

import carrental.external.PayService;

@Entity

@Table(name="Contract\_table")

public class Contract {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

private Long id;

private String custName;

private String modelName;

private Integer amt;

@PostPersist

public void onPostPersist(){

System.out.println("################### Contract >> 계약 생성 ##############################");

ContractCompleted contractCompleted = new ContractCompleted();

BeanUtils.copyProperties(this, contractCompleted);

contractCompleted.publishAfterCommit();

//Following code causes dependency to external APIs

// it is NOT A GOOD PRACTICE. instead, Event-Policy mapping is recommended.

System.out.println("################### Contract >> PAY REST 호출 ##############################");

Pay pay = new Pay();

// mappings goes here

BeanUtils.copyProperties(this, pay);

pay.setContractId(this.getId());

ContractApplication.applicationContext.getBean(PayService.class).pay(pay);

}

@PostUpdate

public void onPostUpdate(){

System.out.println("###################계약 수정(취소)##############################");

ContractCanceled contractCanceled = new ContractCanceled();

BeanUtils.copyProperties(this, contractCanceled);

contractCanceled.publishAfterCommit();

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getCustName() {

return custName;

}

public void setCustName(String custName) {

this.custName = custName;

}

public String getModelName() {

return modelName;

}

public void setModelName(String modelName) {

this.modelName = modelName;

}

public Integer getAmt() {

return amt;

}

public void setAmt(Integer amt) {

this.amt = amt;

}

}

* Entity Pattern 과 Repository Pattern 을 적용하여 JPA 를 통하여 다양한 데이터소스 유형 (RDB or NoSQL) 에 대한 별도의 처리가 없도록 데이터 접근 어댑터를 자동 생성하기 위하여 Spring Data REST 의 RestRepository 를 적용하였다

package carrental;

import org.springframework.data.repository.PagingAndSortingRepository;

import org.springframework.data.rest.core.annotation.RepositoryRestResource;

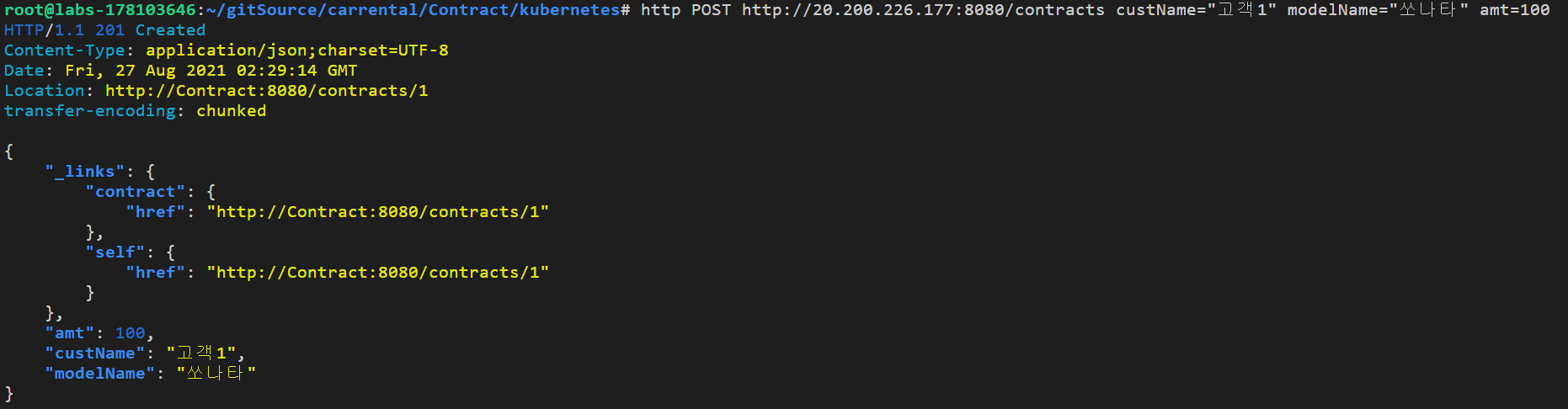
@RepositoryRestResource(collectionResourceRel="contracts", path="contracts")

public interface ContractRepository extends PagingAndSortingRepository<Contract, Long>{

}

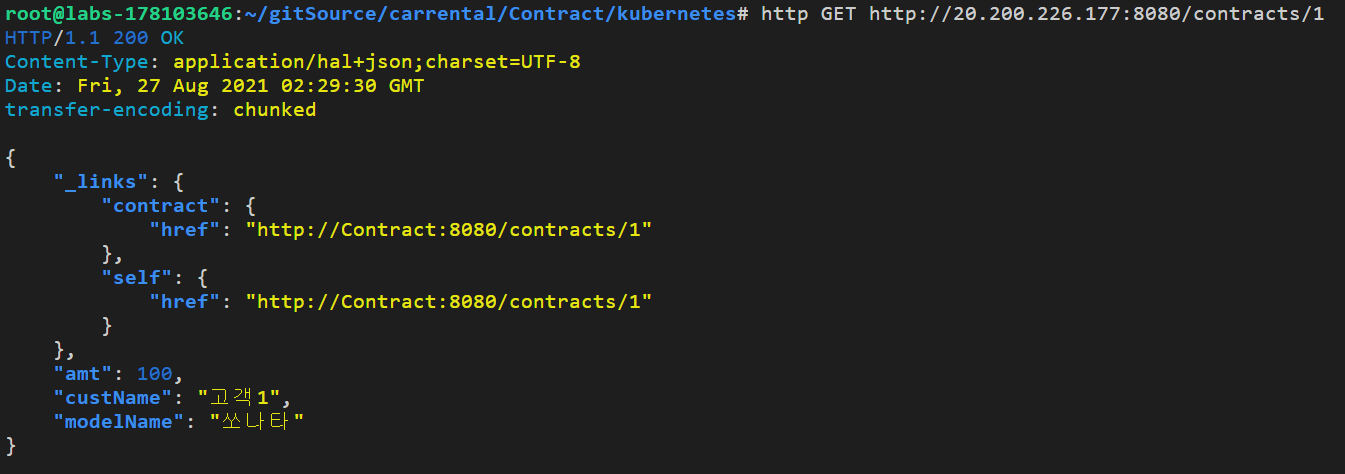
* 적용 후 REST API의 테스트

# Contract 서비스의 렌탈계약처리

http POST http://20.200.226.177:8080/contracts custName="고객1" modelName="쏘나타" amt=100

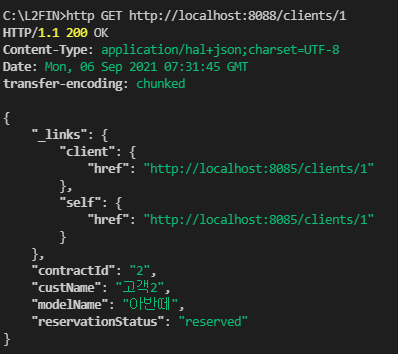
# 계약 상태 확인

http GET http://20.200.226.177:8080/contracts/1



#계약 완료 이후 고객 정보 확인

http GET http://localhost:8088/clients/1



# GateWay 적용

API GateWay를 통하여 마이크로 서비스들의 집입점을 통일할 수 있다. 다음과 같이 GateWay를 적용하였다.

server:

port: 8088

---

spring:

profiles: default

cloud:

gateway:

routes:

- id: Contract

uri: http://localhost:8081

predicates:

- Path=/contracts/\*\*

- id: Pay

uri: http://localhost:8082

predicates:

- Path=/pays/\*\*

- id: Reservation

uri: http://localhost:8083

predicates:

- Path=/reservations/\*\*

- id: Mypage

uri: http://localhost:8084

predicates:

- Path= /myPages/\*\*

globalcors:

corsConfigurations:

'[/\*\*]':

allowedOrigins:

- "\*"

allowedMethods:

- "\*"

allowedHeaders:

- "\*"

allowCredentials: true

---

spring:

profiles: docker

cloud:

gateway:

routes:

- id: Contract

uri: http://Contract:8080

predicates:

- Path=/contracts/\*\*

- id: Pay

uri: http://Pay:8080

predicates:

- Path=/pays/\*\*

- id: Reservation

uri: http://Reservation:8080

predicates:

- Path=/reservations/\*\*

- id: Mypage

uri: http://Mypage:8080

predicates:

- Path= /myPages/\*\*

globalcors:

corsConfigurations:

'[/\*\*]':

allowedOrigins:

- "\*"

allowedMethods:

- "\*"

allowedHeaders:

- "\*"

allowCredentials: true

server:

port: 8080

로컬의 경우는 8088 / 서버의 경우 8080 port로 Contract서비스 정상 호출



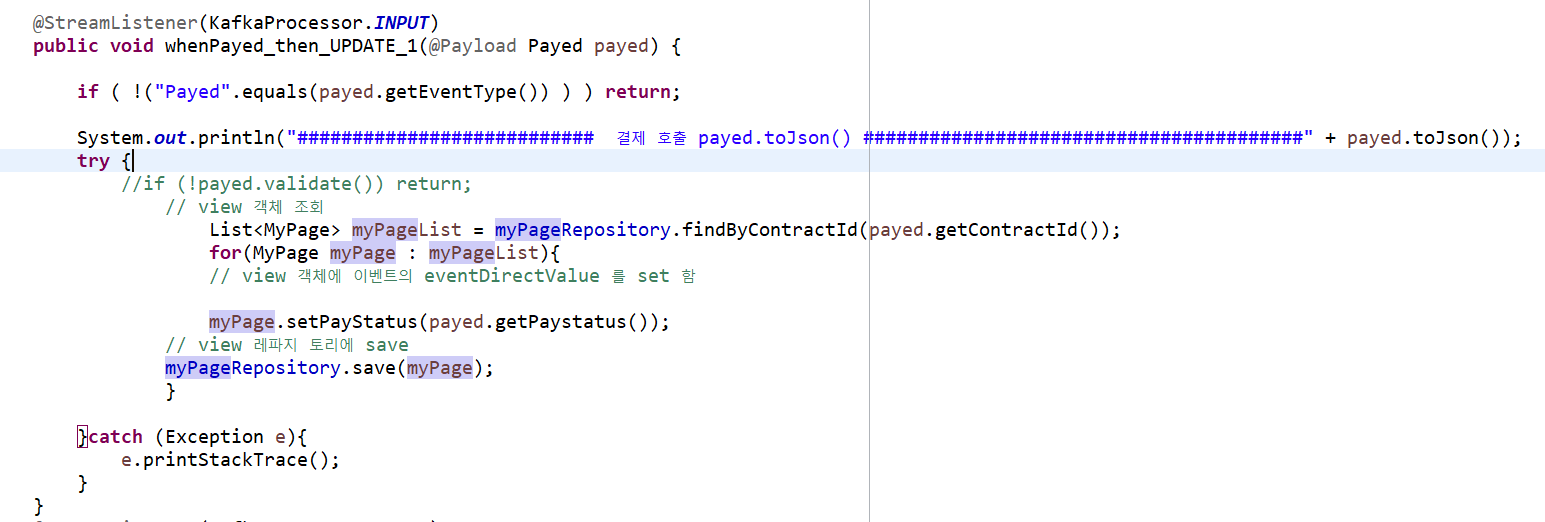


# CQRS/saga/correlation

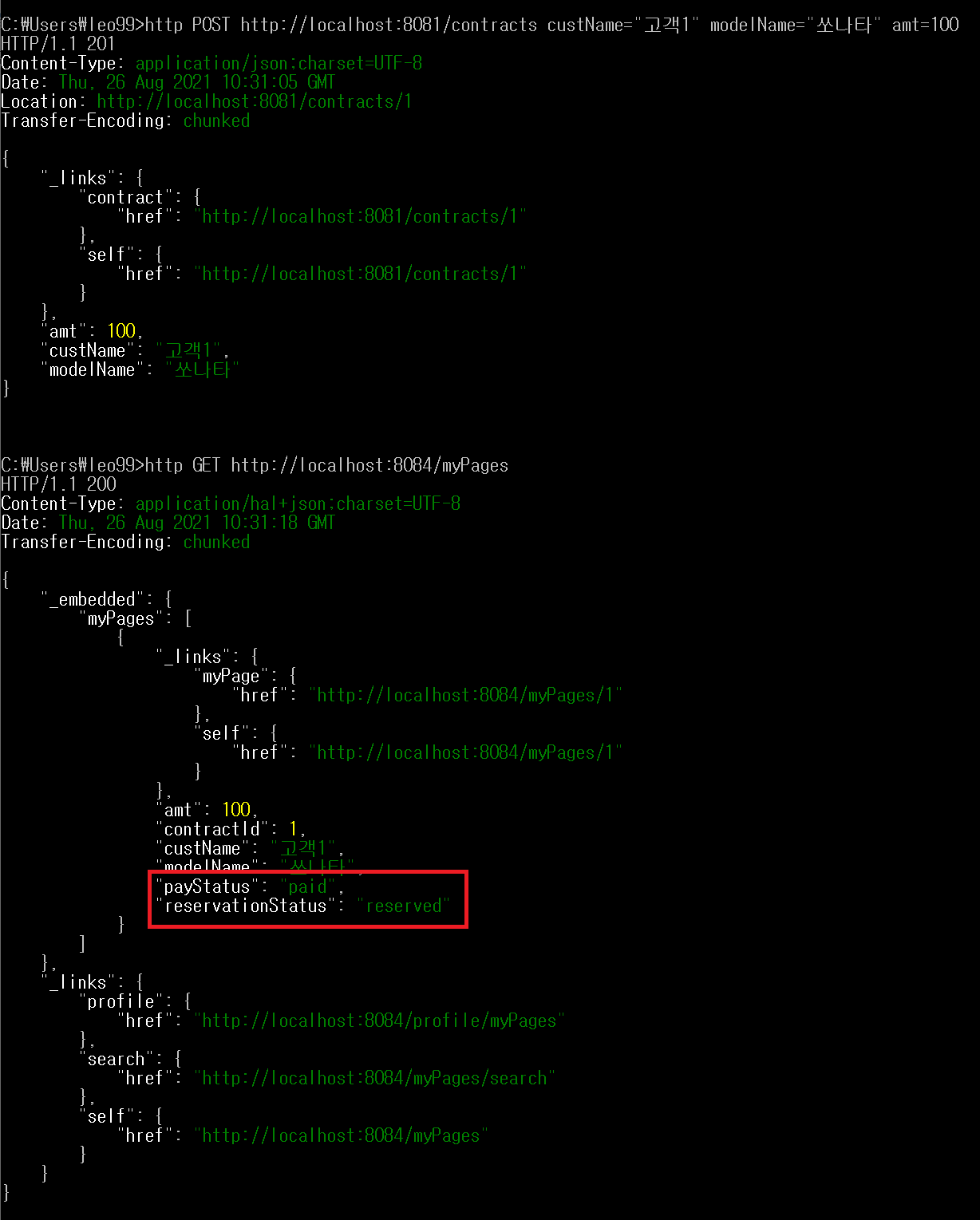
Materialized View를 구현하여, 타 마이크로서비스의 데이터 원본에 접근없이(Composite 서비스나 조인SQL 등 없이)도 내 서비스의 화면 구성과 잦은 조회가 가능하게 구현해 두었다. 본 프로젝트에서 View 역할은 MyPages 서비스가 수행한다.

* 비동기식으로 처리되어 발행된 이벤트 기반 Kafka 를 통해 수신/처리 되어 별도 Table 에 관리한다
* viewpage MSA ViewHandler 를 통해 구현



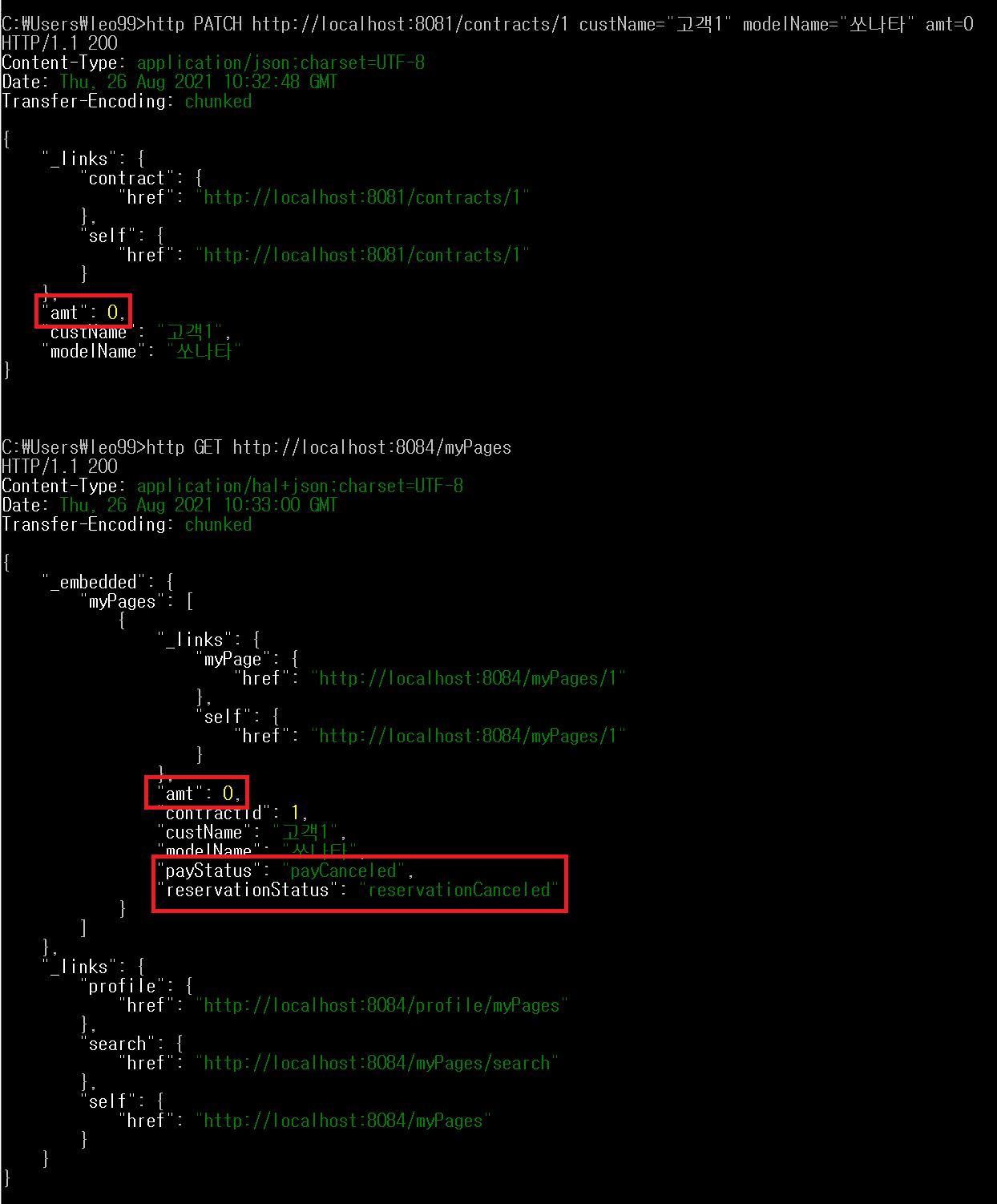


계약 실행 후 MyPages 화면



계약취소 취소 후 MyPages 화면

\*\* 의미상으로 결제금액이 0으로 수정이 발생하면 취소로 판단하였음.



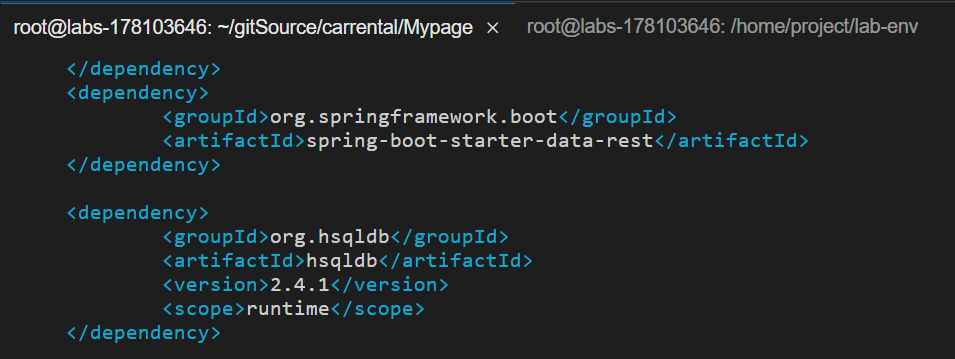
위와 같이 계약을 하게 되면 Contract > Pay > Reservation > MyPage로 계약이 생성되고 상태가 reserved 상태로 되고, 계약 취소가 되면 상태가 reservationCancelled로 변경되는 것을 볼 수 있다.

또한 Correlation을 Key를 활용하여 Id를 Key값을 하고 원하는 주문하고 서비스간의 공유가 이루어 졌다.

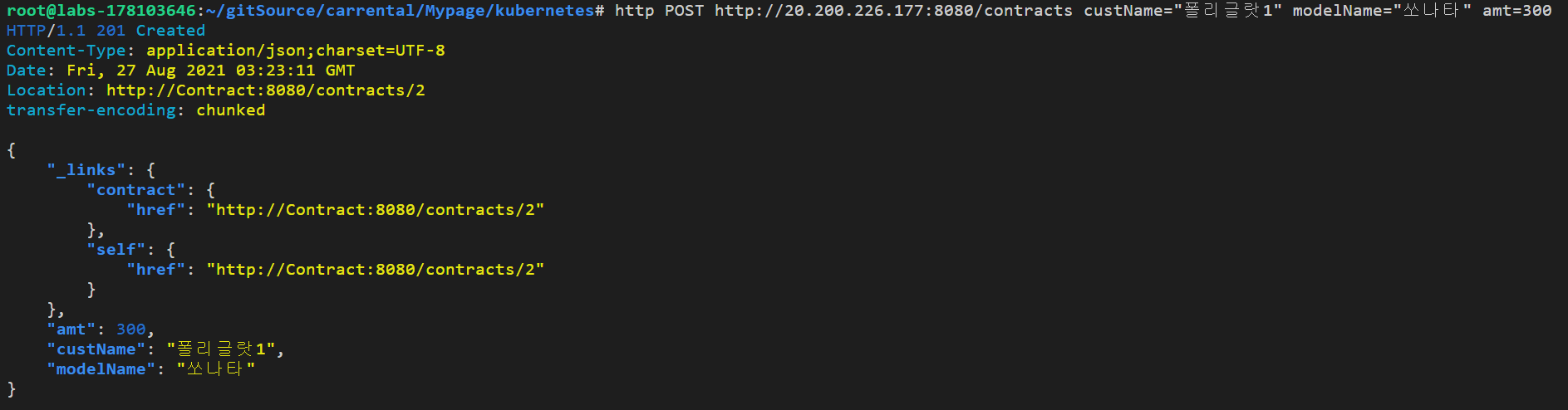
위 결과로 서로 다른 마이크로 서비스 간에 트랜잭션이 묶여 있음을 알 수 있다.

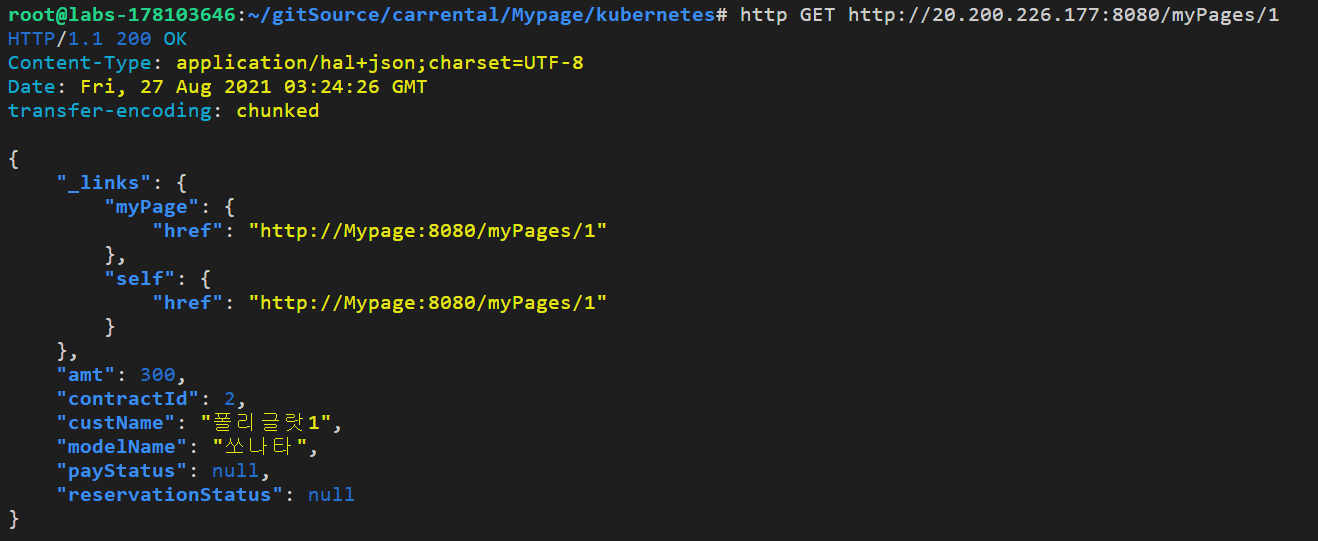
**폴리글랏 퍼시스턴스**

- 최초 MyPage 서비스에서 H2 DB를 적용하였으나, hsqldb 로 변경해보기 위해서 pom.xml 파일에 설정을 변경하였다.



변경 후 계약 생성을 하였을 경우 MyPage 에 정상적으로 데이터가 생성됨을 확인할 수 있다.





\*\* 캡쳐 시 결제/예약서비스가 죽어 있어서 상태는 업데이트되지 않았음.

## 동기식 호출 과 Fallback 처리

분석단계에서의 조건 중 하나로 계약(Contract)->결제(Pay) 간의 호출은 동기식 일관성을 유지하는 트랜잭션으로 처리하기로 하였다. 호출 프로토콜은 이미 앞서 Rest Repository 에 의해 노출되어있는 REST 서비스를 FeignClient 를 이용하여 호출하도록 한다.

* 결제서비스를 호출하기 위하여 Stub과 (FeignClient) 를 이용하여 Service 대행 인터페이스 (Proxy) 를 구현

package carrental.external;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

// feignclient 기술 적용

// url: http://localhost:8082 - application.yaml에 정의함

@FeignClient(name = "Pay", url="${api.url.pay}")

public interface PayService {

@RequestMapping(method = RequestMethod.POST, path = "/pays", consumes = "application/json")

public void pay(@RequestBody Pay pay);

}

주문을 받은 직후(@PostPersist) 결제를 요청하도록 처리

# Contract.java (Entity)

@PostPersist

**public** **void** onPostPersist(){

---중략---

System.***out***.println("################### Contract >> PAY REST 호출 ##############################");

Pay pay = **new** Pay();

// mappings goes here

BeanUtils.*copyProperties*(**this**, pay);

pay.setContractId(**this**.getId());

ContractApplication.*applicationContext*.getBean(PayService.**class**).pay(pay);

}

* 동기식 호출에서는 호출 시간에 따른 타임 커플링이 발생하며, 결제 시스템이 장애가 나면 주문도 못받는다는 것을 확인:

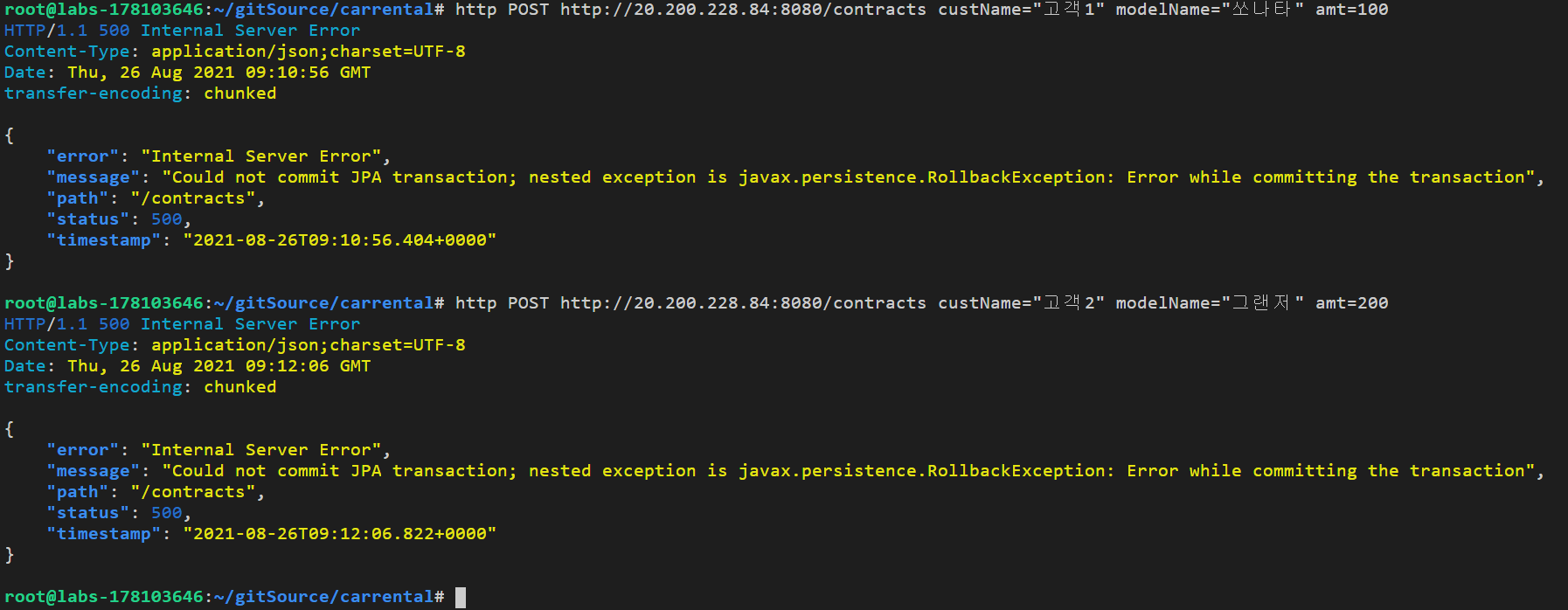
# 결제 (pay) 서비스를 잠시 Pay 서비스 중단함.

kubectl delete deploy,service pay

#주문처리

http POST http://20.200.228.84:8080/contracts custName="고객1" modelName="쏘나타" amt=100 >> #실패

http POST http://20.200.228.84:8080/contracts custName="고객2" modelName="그랜저" amt=200 >> #실패



#결제서비스 재기동

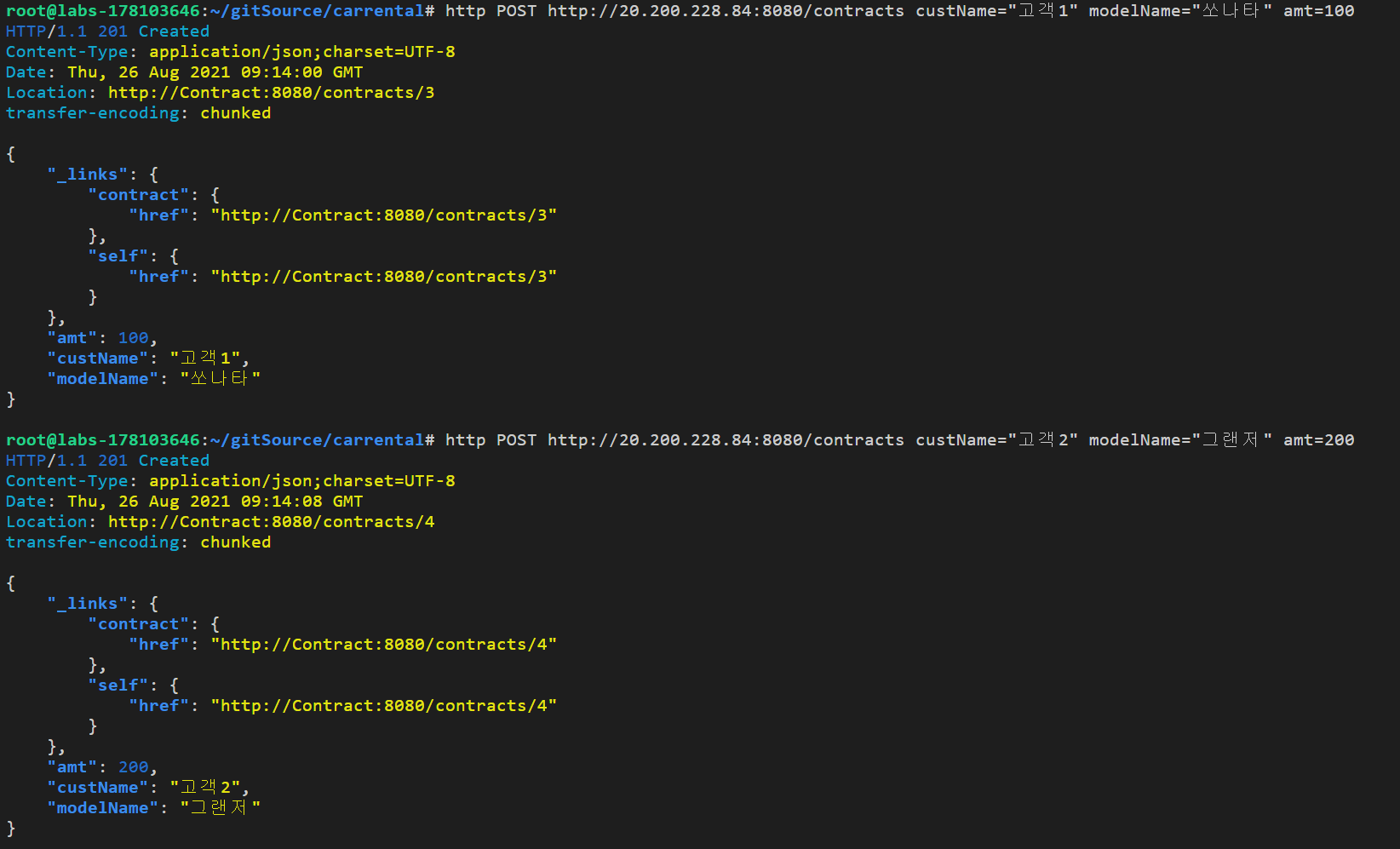
kubectl apply -f Pay/kubernetes/deployment.yml

kubectl apply -f Pay/kubernetes/service.yaml

#재주문처리

http POST http://20.200.228.84:8080/contracts custName="고객1" modelName="쏘나타" amt=100 >> #성공

http POST http://20.200.228.84:8080/contracts custName="고객2" modelName="그랜저" amt=200 >> #성공



* 또한 과도한 요청시에 서비스 장애가 도미노 처럼 벌어질 수 있다. (서킷브레이커, 폴백 처리는 운영단계에서 설명한다.)

## 비동기식 호출 / 시간적 디커플링 / 장애격리 / 최종 (Eventual) 일관성 테스트

계약을 요청한 후 결제가 완료되고(동기식호출) 후 예약 처리시 비동기식으로 처리하여 예약 시스템의 문제로 인해 계약/결제 처리가 블로킹 되지 않도록 처리한다.

* 이를 위하여 결제 처리 후 곧바로 예약 처리를 호출하는 도메인 이벤트를 카프카로 송출한다(Publish)

package carrental;

import javax.persistence.\*;

import org.springframework.beans.BeanUtils;

import carrental.external.Reservation;

import carrental.external.ReservationService;

@Entity

@Table(name="Pay\_table")

public class Pay {

--중략--

@PostPersist

public void onPostPersist(){

Payed payed = new Payed();

BeanUtils.copyProperties(this, payed);

payed.setPaystatus("paid");

payed.publishAfterCommit();

}

* 예약 서비스에서는 결제 승인 이벤트에 대해서 이를 수신하여 자신의 정책을 처리하도록 PolicyHandler 를 구현한다:

package carrental;

import carrental.config.kafka.KafkaProcessor;

import com.fasterxml.jackson.databind.DeserializationFeature;

import com.fasterxml.jackson.databind.ObjectMapper;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.cloud.stream.annotation.StreamListener;

import org.springframework.messaging.handler.annotation.Payload;

import org.springframework.stereotype.Service;

@Service

public class PolicyHandler{

@Autowired ReservationRepository reservationRepository;

@StreamListener(KafkaProcessor.INPUT)

public void wheneverPayed\_Reservate(@Payload Payed payed){

if ("Payed".equals(payed.getEventType())) {

if(!payed.validate()) return;

// Sample Logic //

Reservation reservation = new Reservation();

reservation.setContractId(payed.getContractId());

reservation.setCustName(payed.getCustName());

reservation.setModelName(payed.getModelName());

reservation.setAmt(payed.getAmt());

reservation.setPayStatus(payed.getPaystatus());

reservation.setReservationStatus("reserved");

reservationRepository.save(reservation);

} else {

--

}

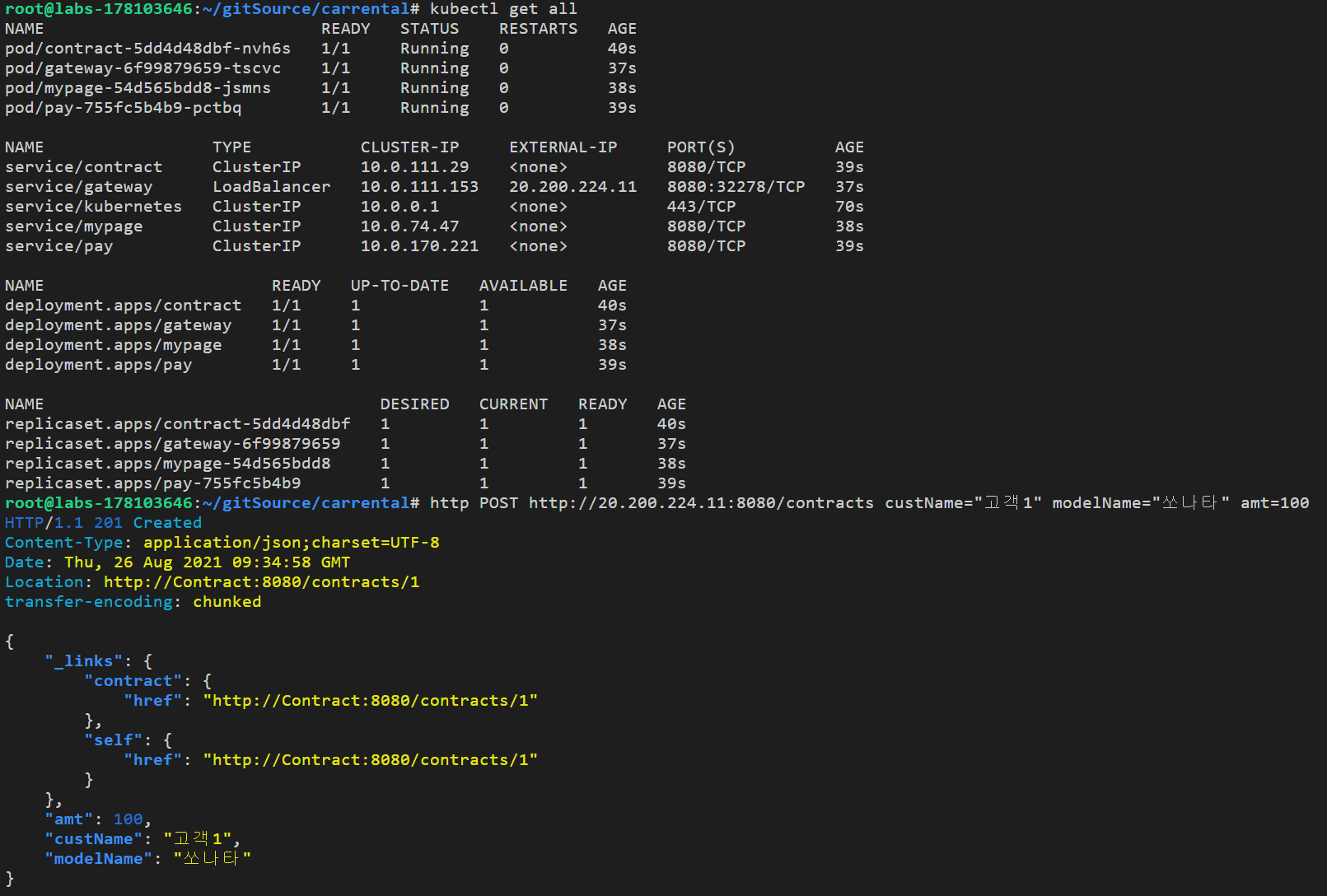
예약 시스템은 계약/결제와 완전히 분리 되어있으며, 이벤트 수신에 따라 처리되기 때문에, 상점시스템이 유지보수로 인해 잠시 내려간 상태라도 계약을 받는데 문제가 없다:

# 예약 서비스 (Reservation) 를 잠시 내려놓음

kubectl delete deploy,service reservation

#주문처리

http POST http://20.200.224.11:8080/contracts custName="고객1" modelName="쏘나타" amt=100 >> 성공

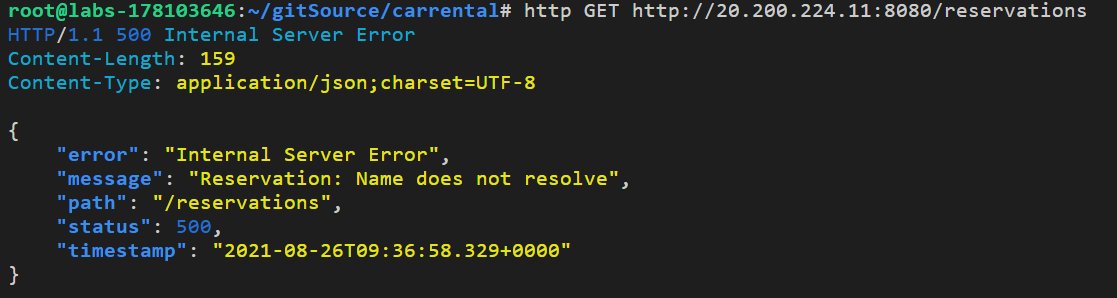


http POST http://20.200.224.50:8080/contracts custName="고객2" modelName="그랜저" amt=200 >> 성공



#예약 시스템 상태 조회

http GET <http://20.200.224.11:8080/reservations> >> 서버 에러



#상점 서비스 기동

kubectl apply -f Reservation/kubernetes/deployment.yml

kubectl apply -f Reservation/kubernetes/service.yaml

#예약 상태 확인

http GET <http://20.200.224.11:8080/reservations> >> 장애 동안 발생한 계약이 정상적으로 처리되어 있음.



# 운영

## CI/CD 설정

각 구현체들은 각자의 source repository 에 구성되었고, 각 소스 경로에서

mvn package 및 컨테이너 레지스트리 의 리파지토리로 이미지를 빌드한 후

미리 작성한 yml 파일을 통해 배포하였다.

소스 가져오기

git clone <https://github.com/leo99k/carrental.git>

## Deploy / Pipeline

* build 하기

계약 경로로 접속

cd Contract

mvn package

az acr build --registry user05 --image user05.azurecr.io/contract:v1 .

미리 생성한 yml 파일을 이용하여 배포

kubectl apply -f deployment.yml

kubectl apply -f service.yaml //또는 kubectl expose deploy mypage --type=ClusterIP --port=8080 (gateway 의 경우는 LoadBalancer) 로 적용한다.

\* docker로 할 경우

-- 도커 빌드

docker build -t user05.azurecr.io/contract:v1 .

-- 도커 푸시

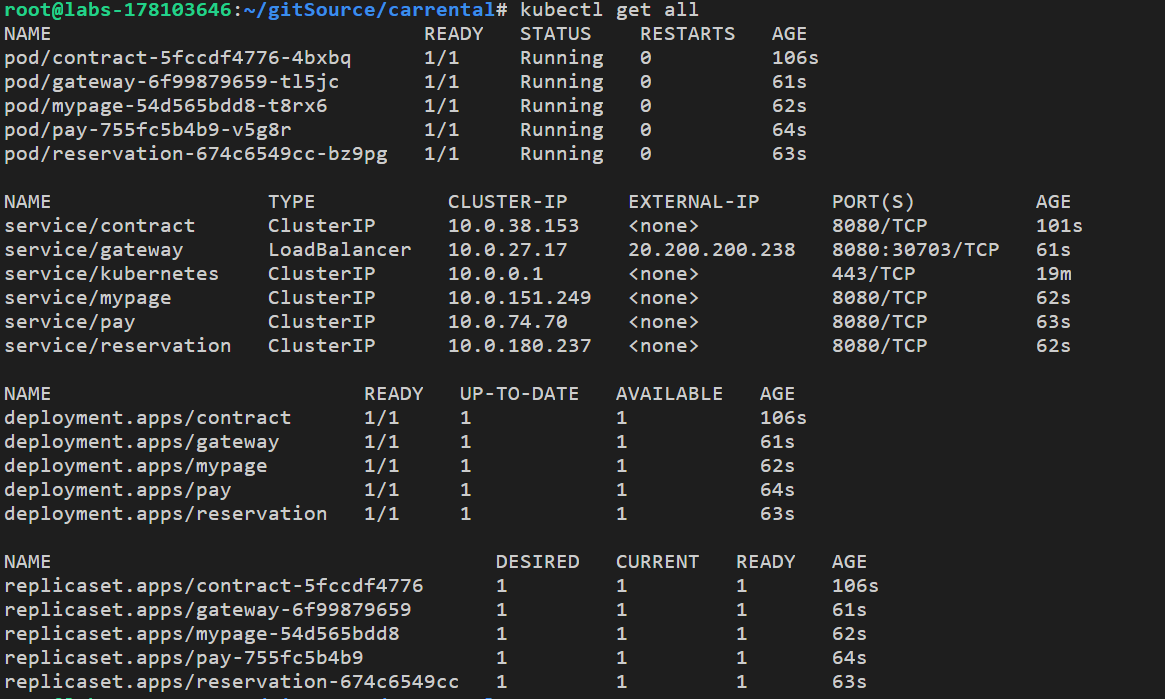
docker push user05.azurecr.io/contract:v1

--권한 오류시

az acr login --name user05

각 Pay / Reservation / MyPage / gateway 도 동일하게 처리한다.

Service, Pod, Deploy 상태 확인



## ConfigMap

* deployment.yml 파일에 설정

## Config map Set start

env:

- name: SYSTEM\_MODE

valueFrom:

configMapKeyRef:

name: systemmode

key: sysmodeval

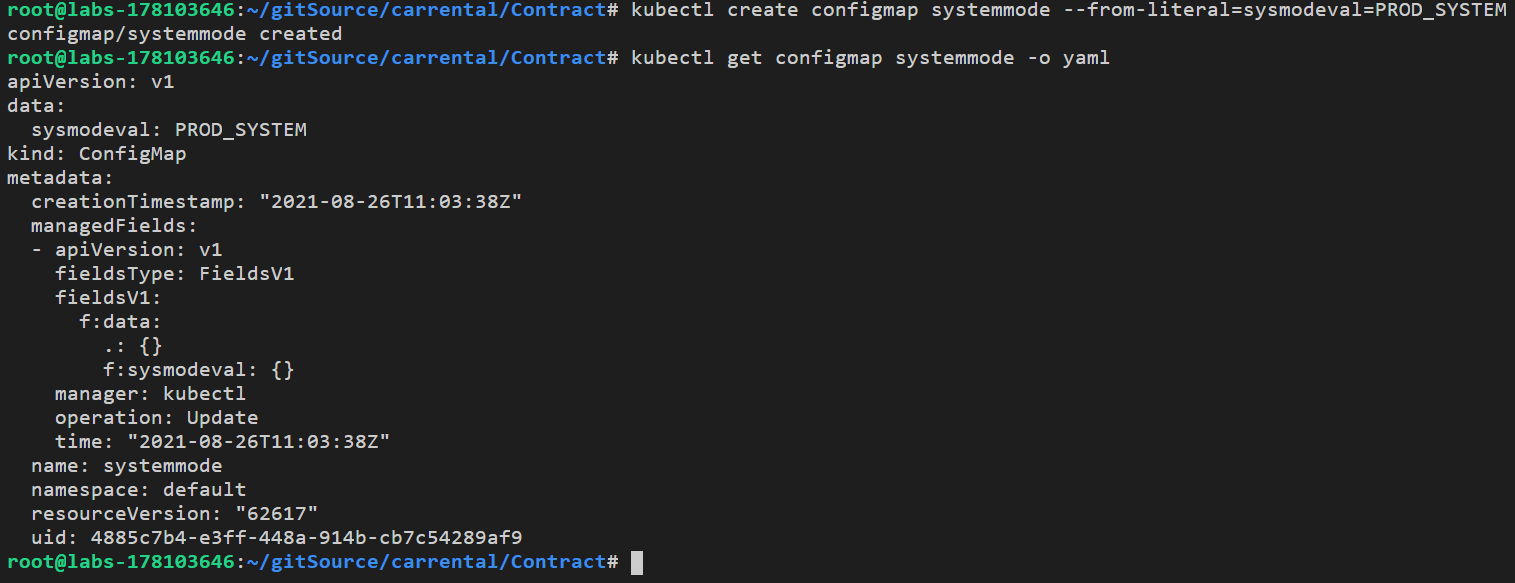
## Config map Set end

ConfigMap 생성

kubectl create configmap systemmode --from-literal=sysmodeval=PROD\_SYSTEM

Configmap 생성, 정보 확인

kubectl get configmap systemmode -o yaml



Contract 호출 시 해당 시스템이 로컬/개발/운영 인지 여부를 ConfigMap 을 이용해서 확인하도록 구현하였음

@PostPersist

**public** **void** onPostPersist(){

// configMap 테스트

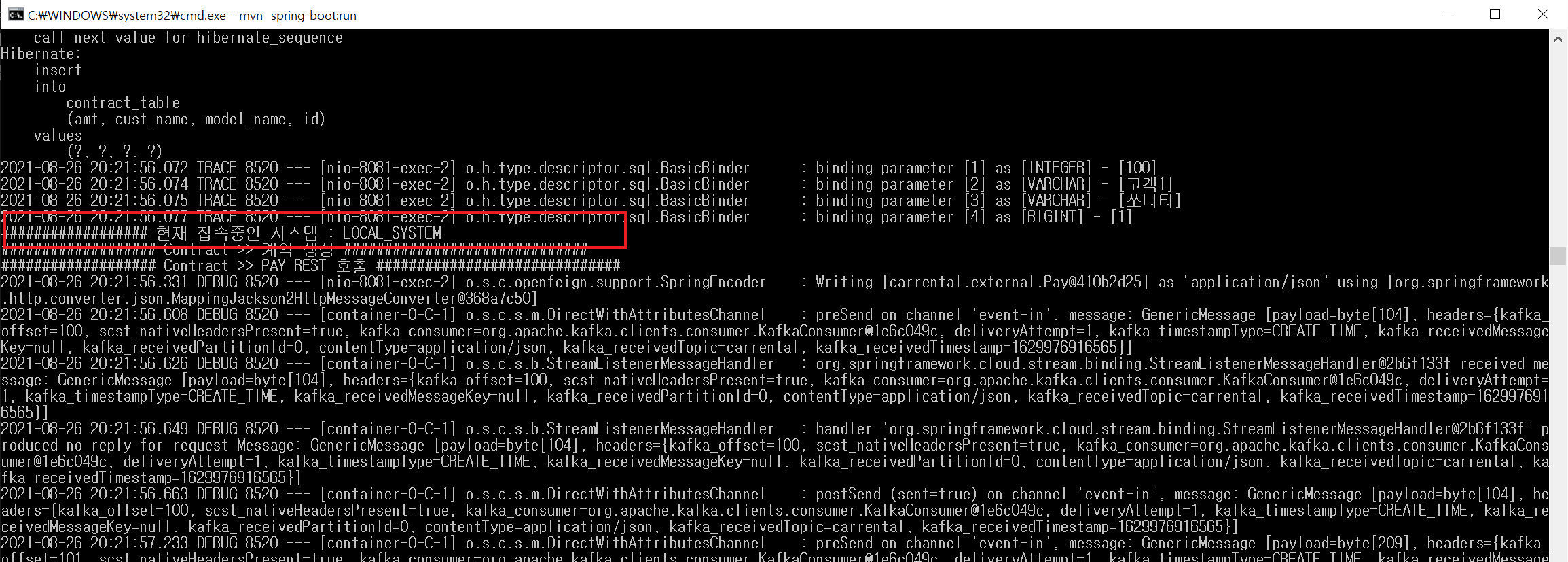
String sysMode = System.*getenv*("SYSTEM\_MODE");

**if**(sysMode == **null**) sysMode = "LOCAL\_SYSTEM";

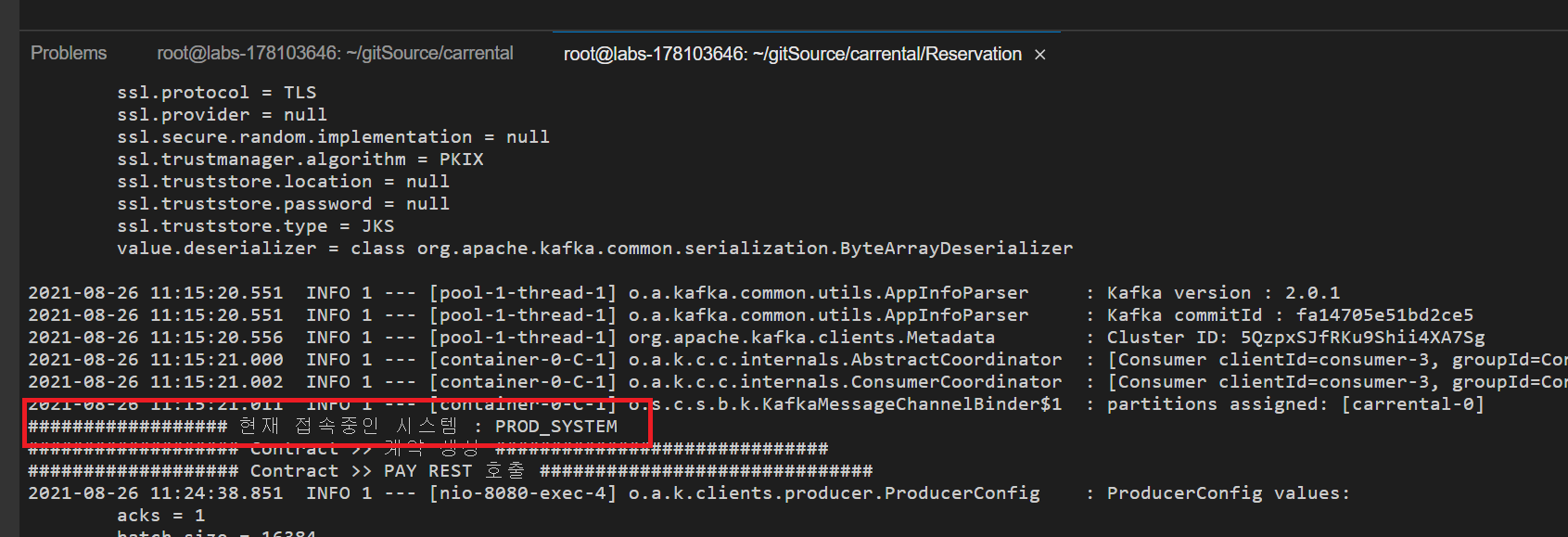
System.***out***.println("################## 현재 접속중인 시스템 : " + sysMode);

System.***out***.println("################### Contract >> 계약 생성 ##############################");

\*로컬의 경우



\*서버의 경우



## 동기식 호출 / 서킷 브레이킹 / 장애격리

* 서킷 브레이킹 프레임워크의 선택: Spring FeignClient + Hystrix 옵션을 사용하여 구현함

시나리오는 계약(Contract)-->결제(Pay) 시의 연결을 RESTful Request/Response 로 연동하여 구현이 되어있고, 결제 요청이 과도할 경우 CB 를 통하여 장애격리.

* Hystrix 를 설정: 요청처리 쓰레드에서 처리시간이 610 밀리가 넘어서기 시작하여 어느정도 유지되면 CB 회로가 닫히도록 (요청을 빠르게 실패처리, 차단) 설정

Contract 소스 application.yml 에 설정

# circuit breaker 설정 start

feign:

hystrix:

enabled: true

hystrix:

command:

default:

execution.isolation.thread.timeoutInMilliseconds: 610

# circuit breaker 설정 end

* 피호출 서비스(결제:Pay) 의 임의로 처리를 지연시킨다.

Pay.java

@PostPersist

public void onPostPersist(){

--중략--

// delay 처리

try {

Thread.currentThread().sleep((long) (400 + Math.random() \* 220));

} catch (InterruptedException e) {

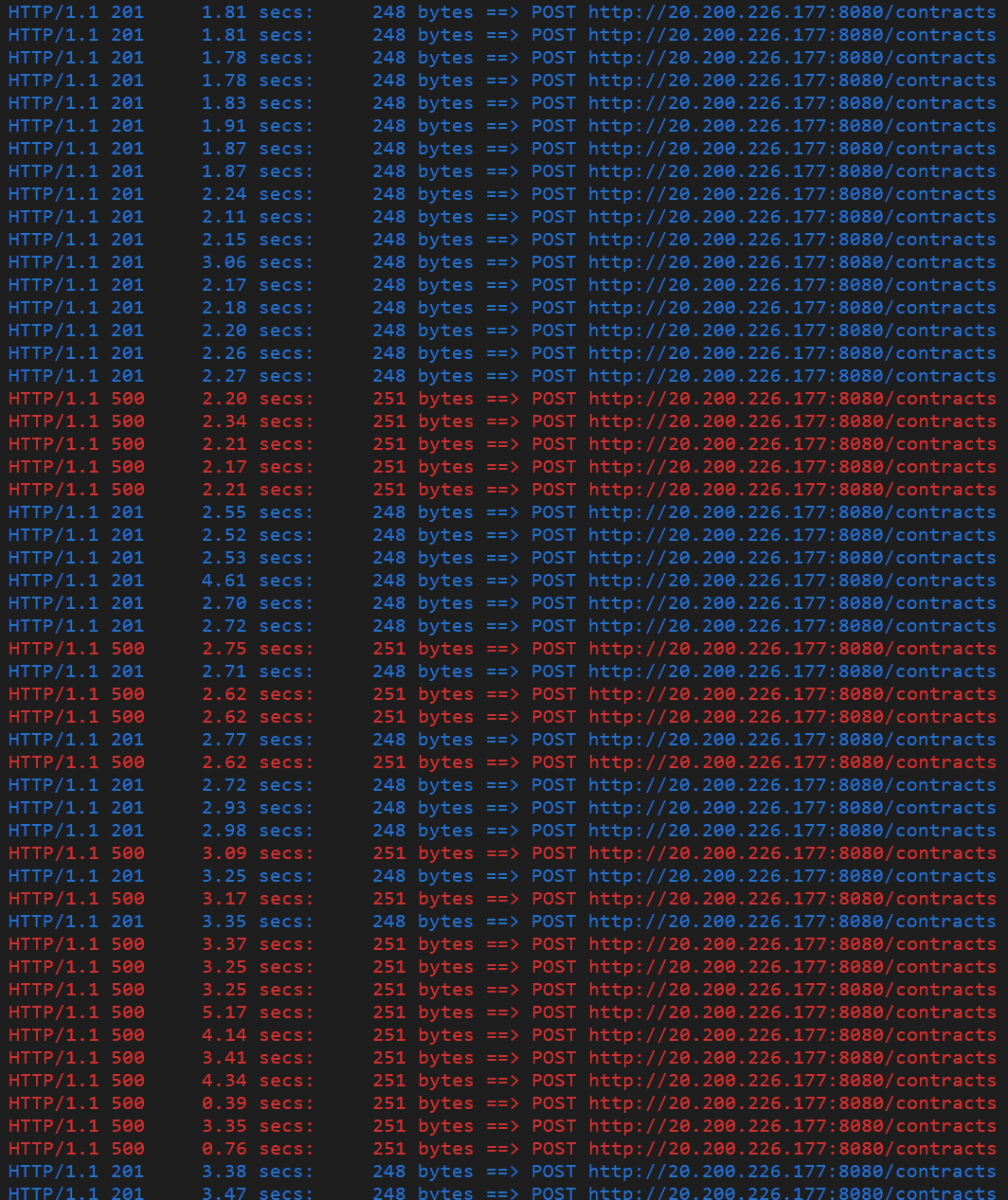
e.printStackTrace();

}

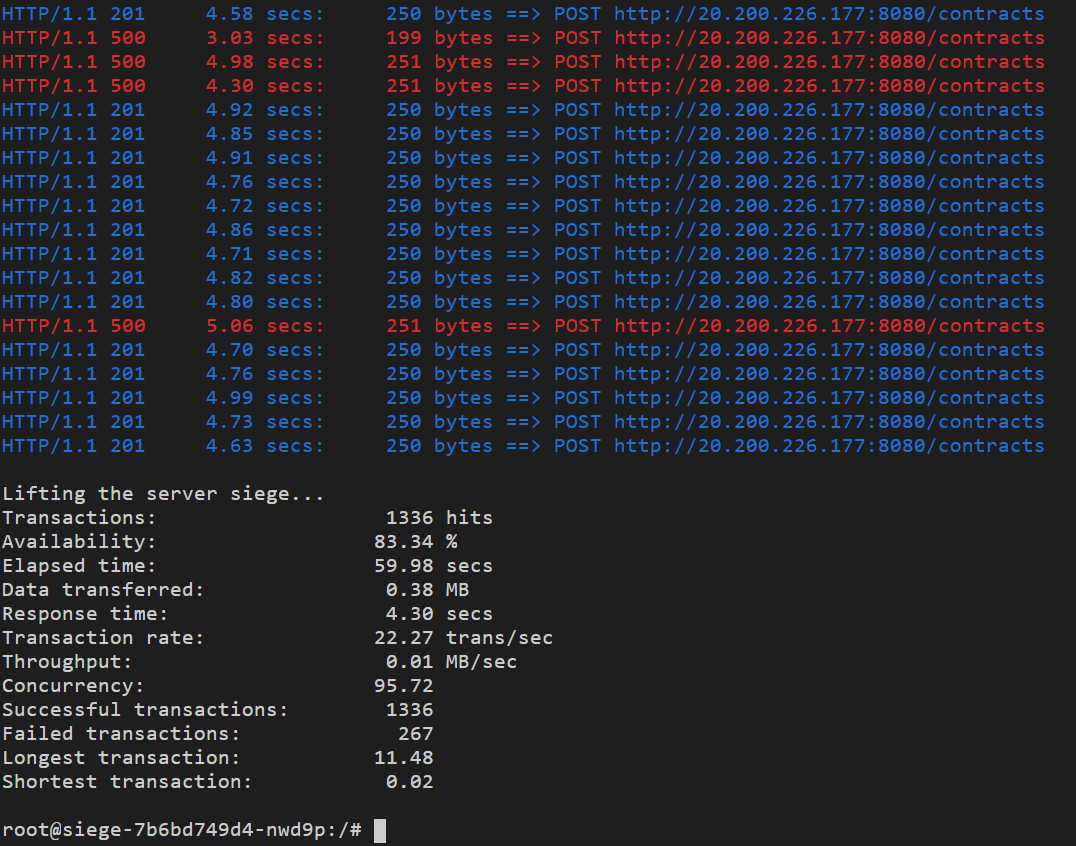
}

* 부하테스터 siege 툴을 통한 서킷 브레이커 동작 확인:
* 동시사용자 100명
* 60초 동안 실시

siege -c100 -t60S -v --content-type "application/json" 'http://20.200.226.177:8080/contracts POST {"custName": "siege1", "modelName": "부하테스트", "amt": 150}'



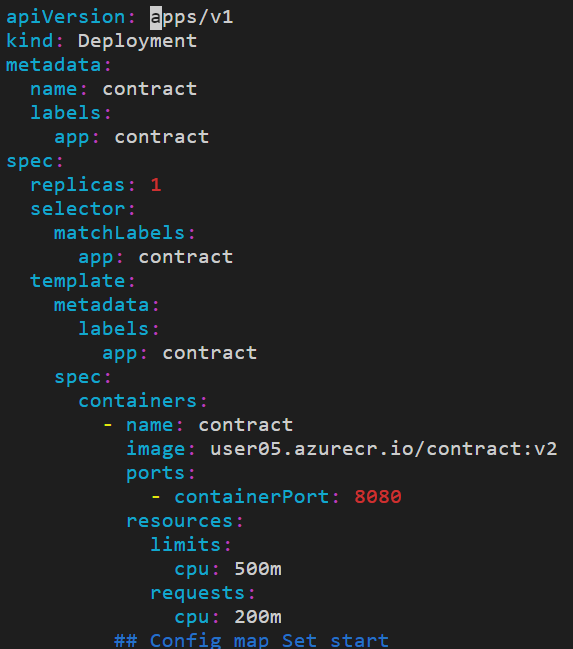
--중략—



* 운영시스템은 죽지 않고 지속적으로 CB 에 의하여 적절히 회로가 열림과 닫힘이 벌어지면서 자원을 보호하고 있음을 보여줌. 하지만, 83.34% 가 성공하였고, 17%가 실패했다는 것은 고객 사용성에 있어 좋지 않기 때문에 Retry 설정과 동적 Scale out (replica의 자동적 추가,HPA) 을 통하여 시스템을 확장 해주는 후속처리가 필요.

## 오토스케일 아웃

* 앞서 서킷 브레이커(CB) 는 시스템을 안정되게 운영할 수 있게 해줬지만 사용자의 요청을 100% 받아들여주지 못했기 때문에 이에 대한 보완책으로 자동화된 확장 기능을 적용하고자 한다.
* Contract서비스 deployment.yml 에서 시스템 성능을 낮춘다.



* 다시 배포해준다.
* Order 서비스에 대한 replica 를 동적으로 늘려주도록 HPA 를 설정한다. 설정은 CPU 사용량이 15프로를 넘어서면 replica 를 10개까지 늘려준다

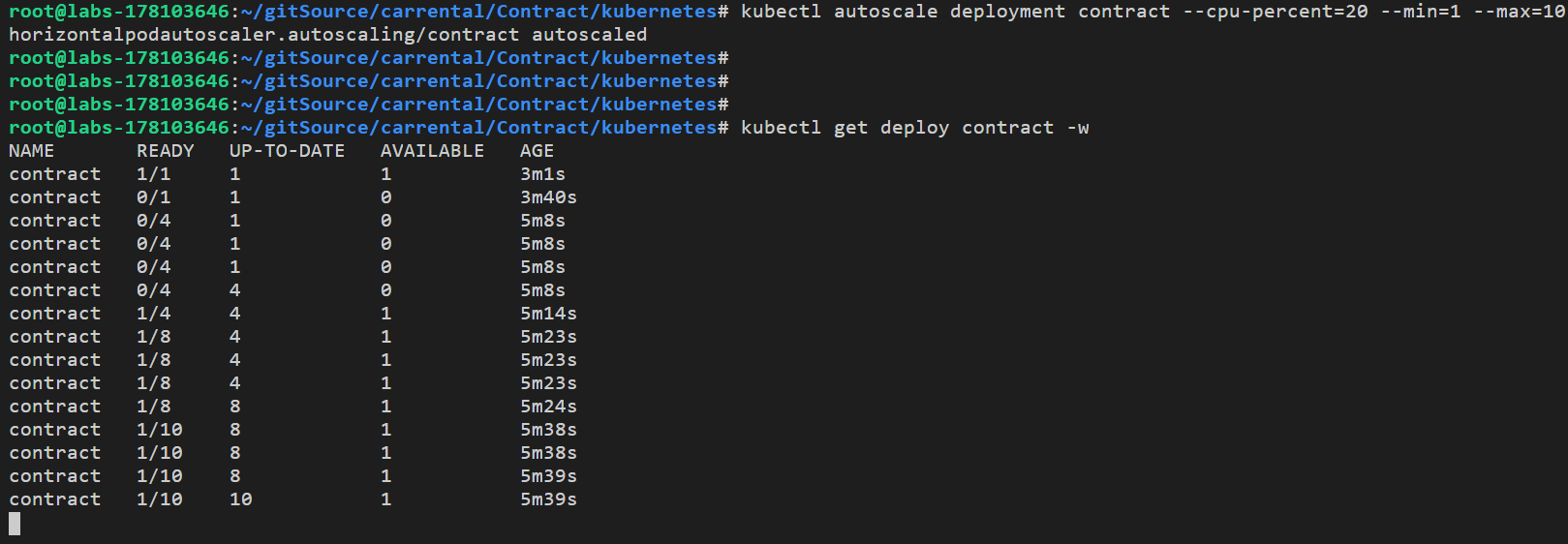
kubectl autoscale deployment contract --cpu-percent=15 --min=1 --max=10

* siege를 활용해서 워크로드를 250명, 60초간 걸어준다.

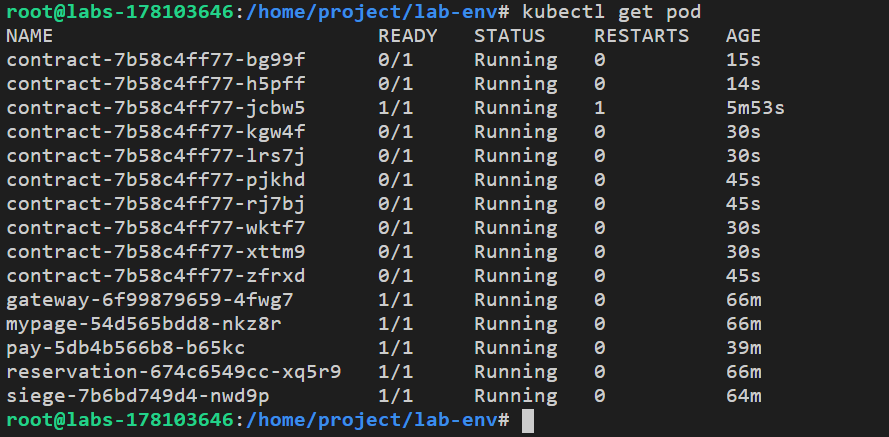
siege -c250 -t60S -v --content-type "application/json" 'http://20.200.226.177:8080/contracts POST {"custName": "siege1", "modelName": "부하테스트", "amt": 150}'

오토스케일이 어떻게 되고 있는지 모니터링을 걸어둔다

kubectl get deploy contract -w



kubectl get pod



## 무정지 재배포 (Readiness Probe)

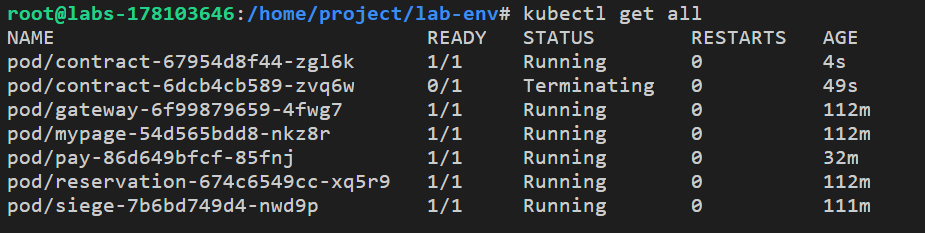
Readiness 설정 적용 전

서비스가 정상적으로 수행되고 있는지 확인을 위해 부하를 걸어준다.

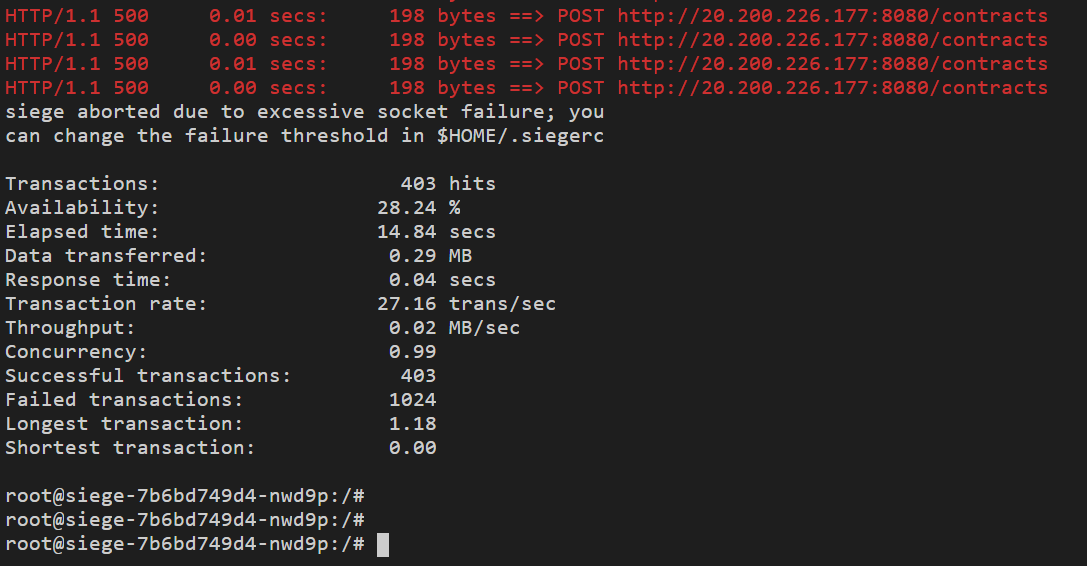
siege -c1 -t120S -v --content-type "application/json" 'http://20.200.226.177:8080/contracts POST {"custName": "siege1", "modelName": "부하테스트", "amt": 150}'

Contrant 서비스의 deployment.yml 파일을 deployment\_v2.yml 버전으로 생성한 후

kubectl apply -f deployment\_v2.yml 로 버전을 변경하여 배포를 수행한다.



배포중 서비스 중단으로 인해 Availability 가 28% 임을 확인할 수 있다.



Readiness 설정 적용 후

배포 파일에 설정을 적용한다.

readinessProbe:

httpGet:

path: '/actuator/health'

port: 8080

initialDelaySeconds: 10

timeoutSeconds: 2

periodSeconds: 5

failureThreshold: 10

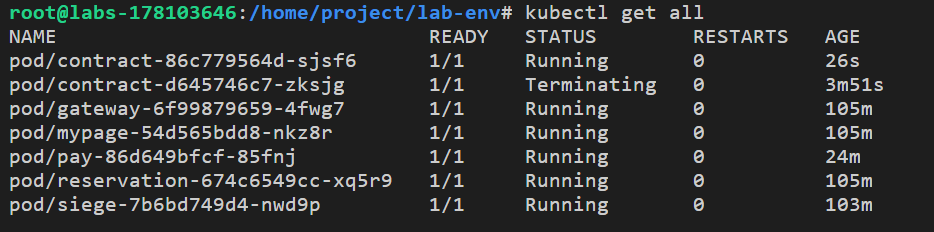
서비스가 정상적으로 수행되고 있는지 확인을 위해 부하를 걸어준다.

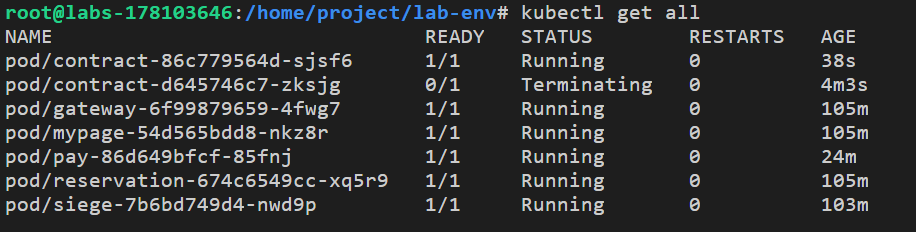
siege -c1 -t120S -v --content-type "application/json" 'http://20.200.226.177:8080/contracts POST {"custName": "siege1", "modelName": "부하테스트", "amt": 150}'

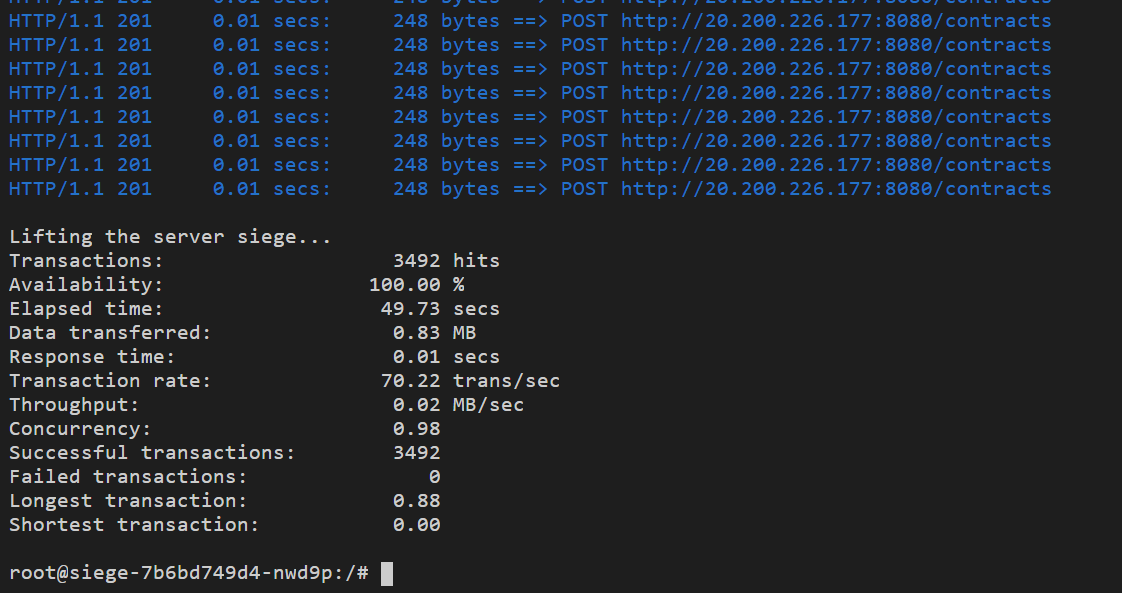
Contrant 서비스의 deployment.yml 파일을 deployment\_v2.yml 버전으로 생성한 후

kubectl apply -f deployment\_v2.yml 로 버전을 변경하여 배포를 수행한다.

배포중 서비스 중단이 없음을 확인할 수 있다.







## Liveness Probe

* order 서비스 deployment.yml livenessProbe 설정을 사용하지 않는 포트인 8089로 변경하여 서비스가 비정상 상태로 인식하도록 배포 하여 liveness probe 가 동작함을 확인

livenessProbe:

httpGet:

path: '/actuator/health'

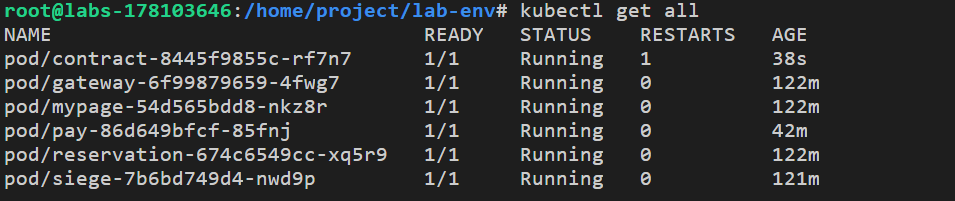
port: 8089

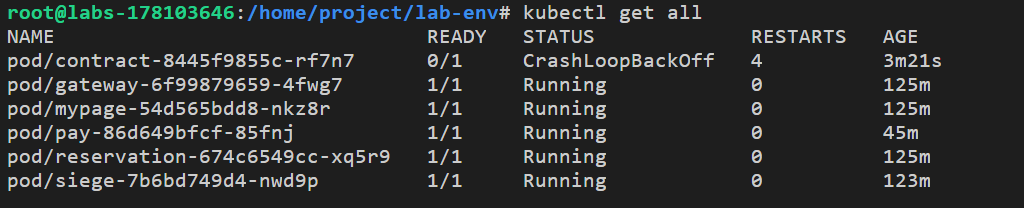
initialDelaySeconds: 15

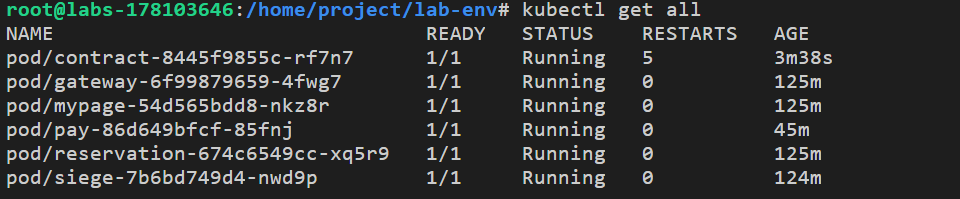
timeoutSeconds: 2

periodSeconds: 5

failureThreshold: 5







개인 과제

고객 관리 MSA 추가

