# **03.디플로이먼트(Deployment), 서비스(Service)를 활용해 서버 띄워보기**

1. **디플로이먼트(Deployment)란?**

### **✅ 디플로이먼트(Deployment)란?**

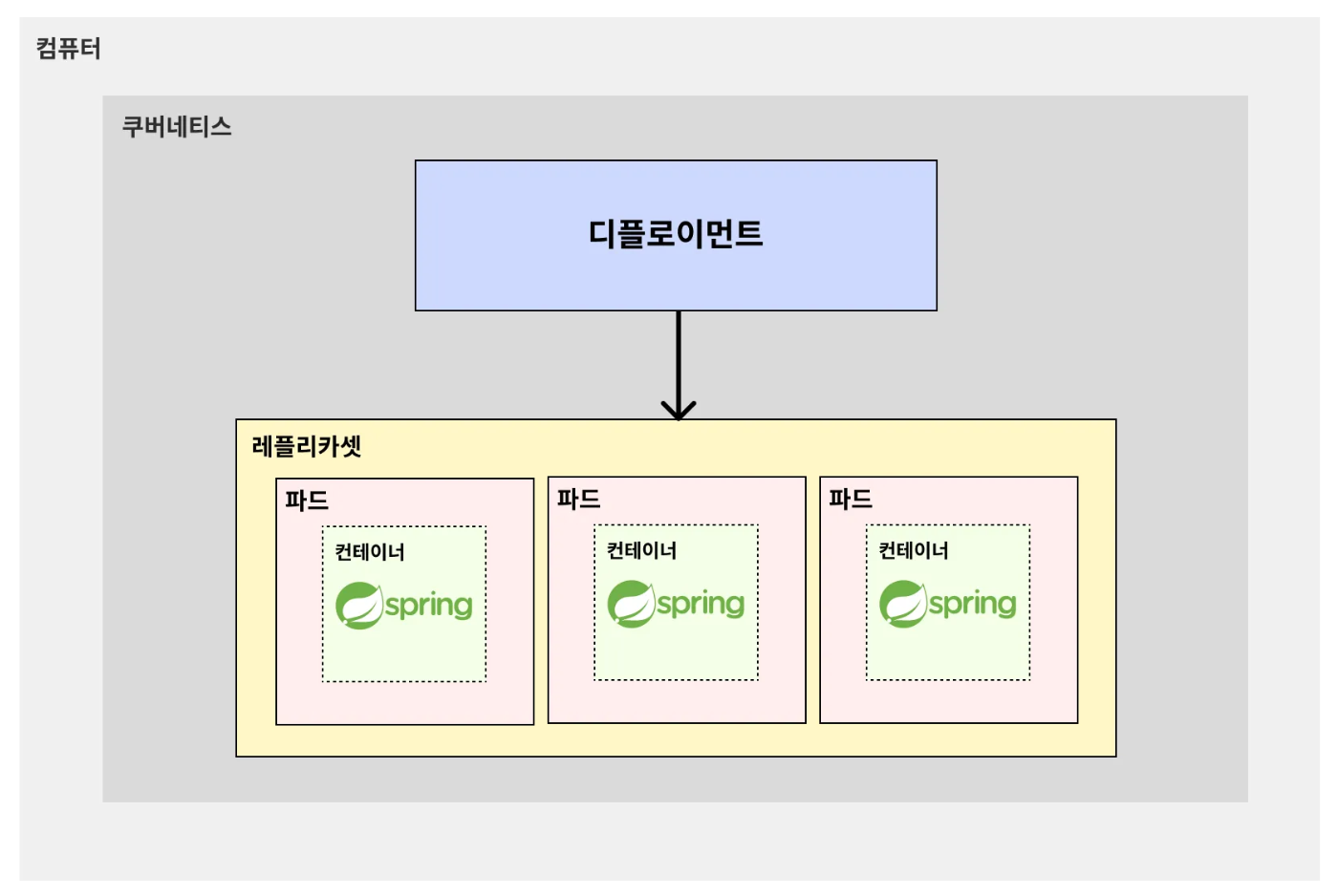
| 💡[First Word 법칙]  **디플로이먼트(Deployment)** : 파드를 묶음으로 쉽게 관리할 수 있는 기능 |
| --- |

현업에서는 일반적으로 서버를 작동시킬 때 파드(Pod)를 수동으로 배포하진 않는다. **디플로이먼트(Deployment)**라는 걸 활용해서 파드(Pod)를 자동으로 배포한다.

### **✅ 디플로이먼트(Deployment)의 장점**

* 파드의 수를 지정하는 대로 여러 개의 파드를 쉽게 생성할 수 있음.
  + ex) 파드를 100개를 생성하라고 시키면 디플로이먼트가 알아서 파드를 100개 생성해준다.
* 파드가 비정상적으로 종료된 경우, 알아서 새로 파드를 생성해 파드 수를 유지한다.
* 동일한 구성의 여러 파드를 일괄적으로 일시 중지, 삭제, 업데이트를 하기가 쉽다.
  + ex) 디플로이먼트를 활용하면 ‘100개의 파드로 띄워져있는 결제 서버’를 한 번에 일시 중지/삭제/업데이트하는 게 굉장히 쉽다.

### ✅ 디플로이먼트(Deployment)의 구조



* **디플로이먼트(Deployment)가 레플리카셋(ReplicaSet)을 관리하고, 레플리카셋(ReplicaSet)이 여러 파드(Pod)를 관리하는 구조다.**
  + 레플리카(Replica) : 복제본
  + 레플리카셋(ReplicaSet) : 복제본끼리의 묶음

1. **[예제] 디플로이먼트를 활용해 백엔드(Spring Boot) 서버 3개 띄워보기**

### **✅ 디플로이먼트를 활용해 백엔드(Spring Boot) 서버 3개 띄워보기**

기존의 파드 다 삭제하고 시작 할 것.



🧑🏻‍🏫실제 서비스를 운영하다보면 트래픽이 증가해서 서버가 버벅거리는 경우가 생긴다. 이 때는 서버를 수평적 확장(서버의 개수를 늘리는 방식)을 통해 해결한다. 이런 상황을 가정해 백엔드 서버인 Spring Boot 서버를 3대로 늘려보자.

1. **매니페스트 파일 수정하기**

기존 매니페스트 파일 (spring-pod.yaml)

| apiVersion: v1  kind: Pod  metadata:  name: spring-pod**-1**  spec:  containers:  - name: spring-container  image: spring-server  imagePullPolicy: IfNotPresent  ports:  - containerPort: 8080    **---**  apiVersion: v1  kind: Pod  metadata:  name: spring-pod**-2**  spec:  containers:  - name: spring-container  image: spring-server  imagePullPolicy: IfNotPresent  ports:  - containerPort: 8080    **---**  apiVersion: v1  kind: Pod  metadata:  name: spring-pod**-3**  spec:  containers:  - name: spring-container  image: spring-server  imagePullPolicy: IfNotPresent  ports:  - containerPort: 8080 |
| --- |

새로운 매니페스트 파일 (spring-deployment.yaml)

| apiVersion: apps/v1  kind: Deployment  # Deployment 기본 정보  metadata:  name: spring-deployment # Deployment 이름  # Deployment 세부 정보  spec:  replicas: 3 # 생성할 파드의 복제본 개수  selector:  matchLabels:  app: backend-app # 아래에서 정의한 Pod 중 'app: backend-app'이라는 값을 가진 파드를 선택  # 배포할 Pod 정의  template:  metadata:  labels: # 레이블 (= 카테고리)  app: backend-app  spec:  containers:  - name: spring-container # 컨테이너 이름  image: spring-server # 컨테이너를 생성할 때 사용할 이미지  imagePullPolicy: IfNotPresent # 로컬에서 이미지를 먼저 가져온다. 없으면 레지스트리에서 가져온다.  ports:  - containerPort: 8080 # 컨테이너에서 사용하는 포트를 명시적으로 표현 |
| --- |

1. **기존 파드 삭제하기**

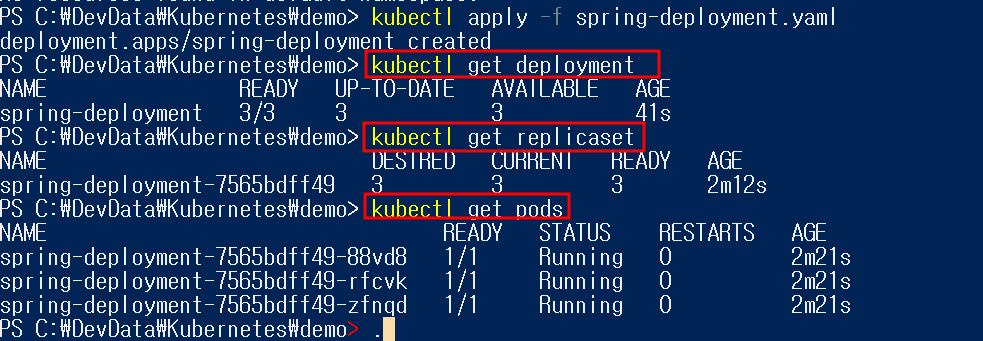
| $ kubectl delete pod spring-pod-1 spring-pod-2 spring-pod-3  $ kubectl get pods # 잘 삭제됐는 지 확인하기 |
| --- |

1. **매니페스트 파일을 기반으로 디플로이먼트(Deployment) 생성하기**

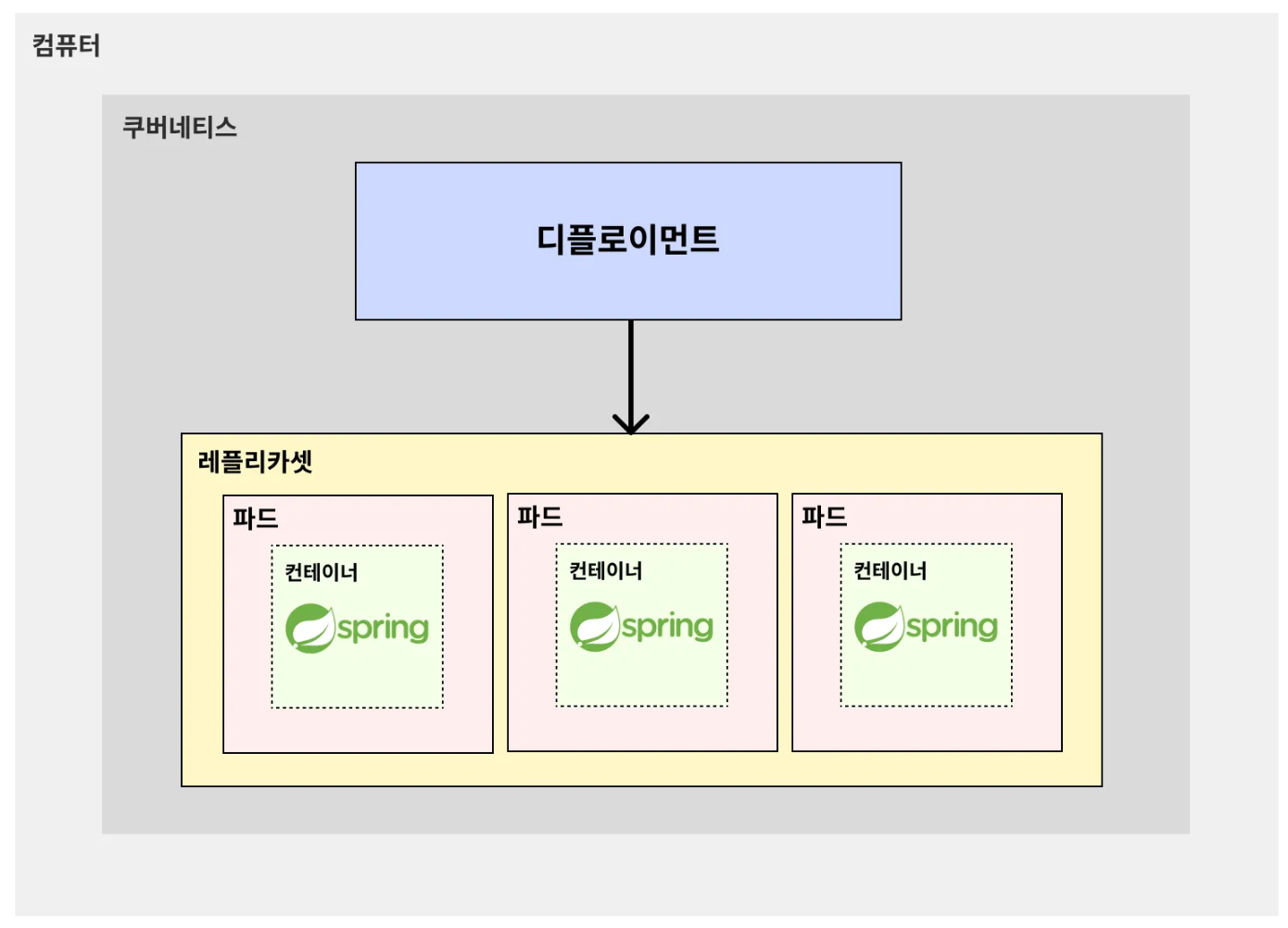
| $ kubectl apply -f spring-deployment.yaml |
| --- |

1. **디플로이먼트, 리플리카셋, 파드가 잘 생성 됐는지 확인**

| $ kubectl get deployment  $ kubectl get replicaset  $ kubectl get pods |
| --- |



### **✅ 전체 구조**



백엔드 서버 3개를 각각의 파드에 띄웠다. 실제 요청을 보낼 때는 각 서버에 균등하게 트래픽이 분배되어야 한다. 그런데 사용자보고 여러 백엔드 서버에 알아서 균등하게 요청을 하라고 시킬 수는 없다. 따라서 파드 앞단에 알아서 여러 파드에 균등하게 요청을 분배해줄 무언가가 필요하다. 쿠버네티스에서는 **서비스(Service)**가 여러 파드에 균등하게 요청을 분배해주는 역할을 한다. 다음 강의에서 서비스(Service)에 대해 자세히 알아보자.

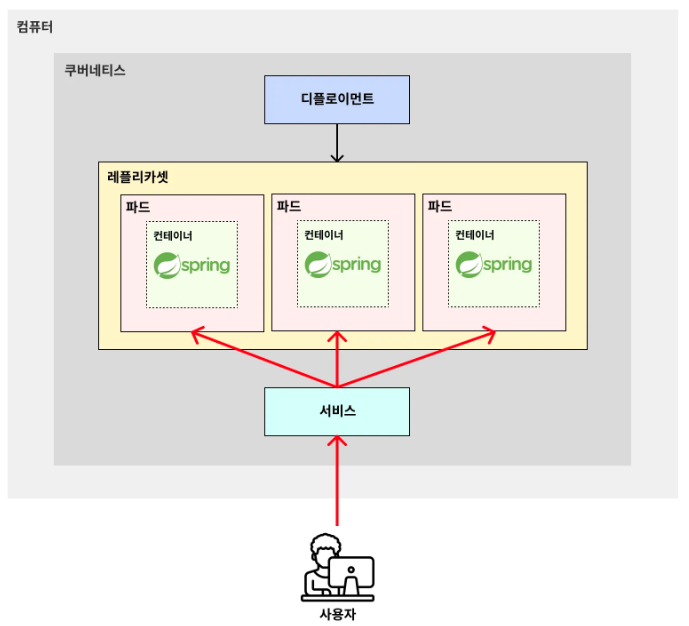
1. **서비스(Service)란?**

### **✅ 서비스(Service)란?**

| 💡[First Word 법칙]  **서비스(Service)** : 외부로부터 들어오는 트래픽을 받아, 파드에 균등하게 분배해주는 로드밸런서 역할을 하는 기능 |
| --- |

실제 서비스에서 파드(Pod)에 요청을 보낼 때, 포트 포워딩(**port-forward**)이나 파드 내로 직접 접근(**kubectl exec …**)해서 요청을 보내진 않는다. 서비스(Service)를 통해 요청을 보내는 게 일반적이다.

### **✅ 서비스(Service)의 구조**

****

1. **[예제] 서비스(Service)를 활용해 백엔드(Spring Boot) 서버와 통신해보기**

### **✅ 서비스(Service)를 활용해 백엔드(Spring Boot)와 통신해보기**

| 🧑이전 강의에서 디플로이먼트를 활용해 백엔드 서버(Spring Boot) 3개를 띄웠었다. 하지만 디플로이먼트에 포함되어 있는 모든 파드에 골고루 요청을 분배하기 위해 서비스(Service)를 생성해야 한다.  **([예제] 디플로이먼트를 활용해 백엔드(Spring Boot) 서버 3개 띄워보기 )-> 위의 예제 참조하기(demo 예제)** |
| --- |

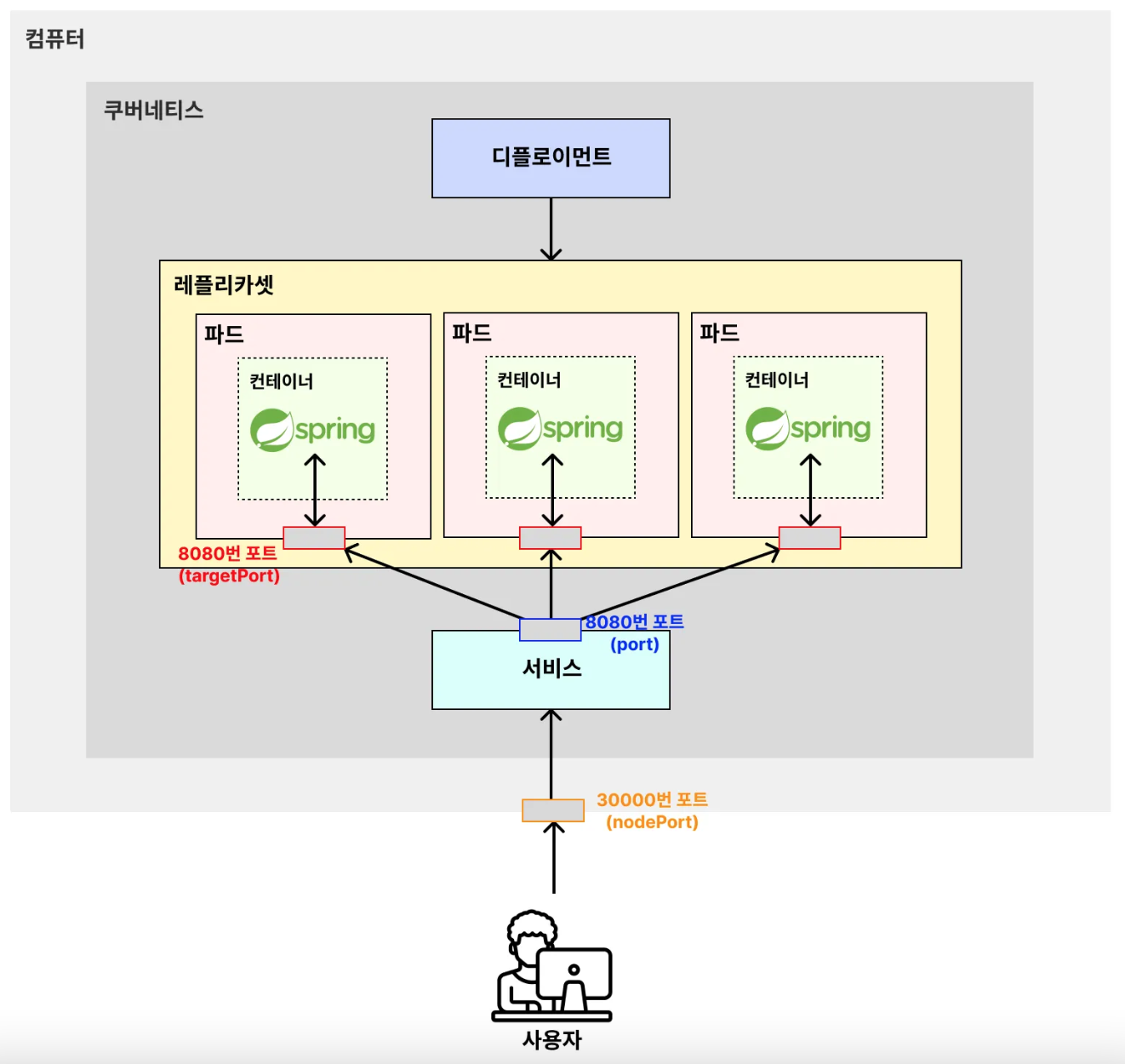
1. **매니페스트 파일 추가하기**

spring-service.yaml

| apiVersion: v1  kind: Service  # Service 기본 정보  metadata:  name: spring-service # Service 이름    # Service 세부 정보  spec:  type: NodePort # Service의 종류  selector:  app: backend-app # 실행되고 있는 파드 중 'app: backend-app'이라는 값을 가진 파드와 서비스를 연결  ports:  - protocol: TCP # 서비스에 접속하기 위한 프로토콜  port: 8080 # 쿠버네티스 내부에서 Service에 접속하기 위한 포트 번호  targetPort: 8080 # 매핑하기 위한 파드의 포트 번호  nodePort: 30000 # 외부에서 사용자들이 접근하게 될 포트 번호 |
| --- |

**backend-app=> spring-deployment.yaml 에있는 앱과 연결한다는 뜻**

* + **Service** 종류에 대해 한 번 짚고 넘어가자. 우선 아래 3가지 개념에 대해서만 이해하고 넘어가자.
    - **NodePort** : 쿠버네티스 내부에서 해당 서비스에 접속하기 위한 포트를 열고 외부에서 접속 가능하도록 한다.
    - **ClusterIP** : 쿠버네티스 내부에서만 통신할 수 있는 IP 주소를 부여. 외부에서는 요청할 수 없다.
    - **LoadBalancer** : 외부의 로드밸런서(AWS의 로드밸런서 등)를 활용해 외부에서 접속할 수 있도록 연결한다.

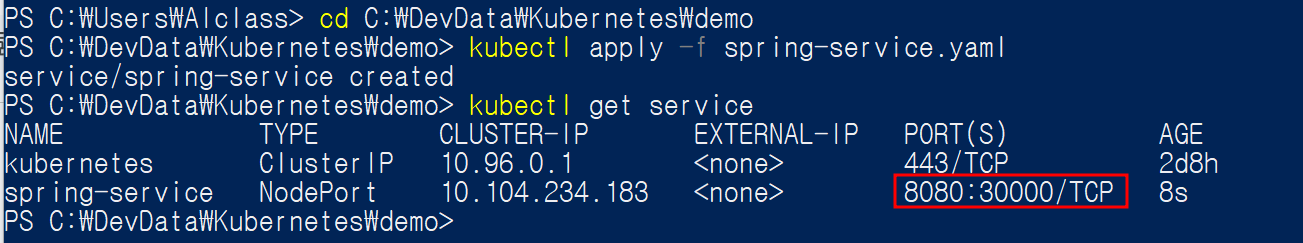


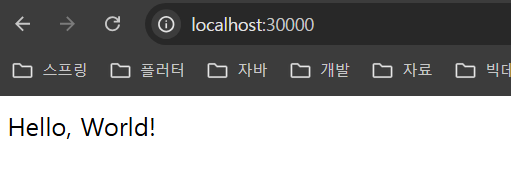
1. **매니페스트 파일을 기반으로 서비스(Service) 생성하기**

| $ kubectl apply -f spring-service.yaml |
| --- |

1. **서비스가 잘 생성 됐는지 확인**

| $ kubectl get service |
| --- |



1. **잘 접속되는 지 확인**  
   
2. **디플로이먼트를 활용한 서버 개수 조절 방법**

### **✅ 트래픽이 늘어나서 서버를 5개로 늘리고 싶다면?**

디플로이먼트(Deployment)를 활용하면 쉽게 서버의 개수를 늘릴 수 있다.

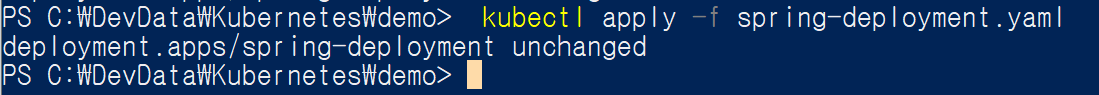
1. **매니페스트 파일 수정**

spring-deployment.yaml

| apiVersion: apps/v1  kind: Deployment  metadata:  name: spring-deployment  spec:  replicas: 5  selector:  matchLabels:  app: backend-app  template:  metadata:  labels:  app: backend-app  spec:  containers:  - name: spring-container  image: spring-server  imagePullPolicy: IfNotPresent  ports:  - containerPort: 8080 |
| --- |

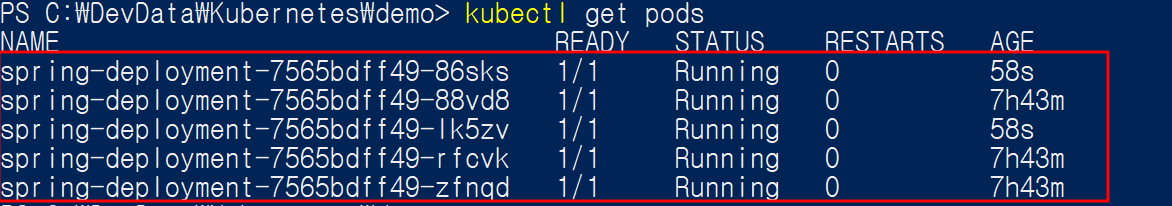
1. 변경사항 적용

| $ kubectl apply -f spring-deployment.yaml |
| --- |

**kubectl apply** 명령어는 새롭게 오브젝트(디플로이먼트, 파드 등)를 생성할 때도 사용하고, 변경 사항을 적용시킬 때도 사용할 수 있는 편리한 명령어이다.  


1. **잘 적용 됐는지 확인하기**

| $ kubectl get pods |
| --- |

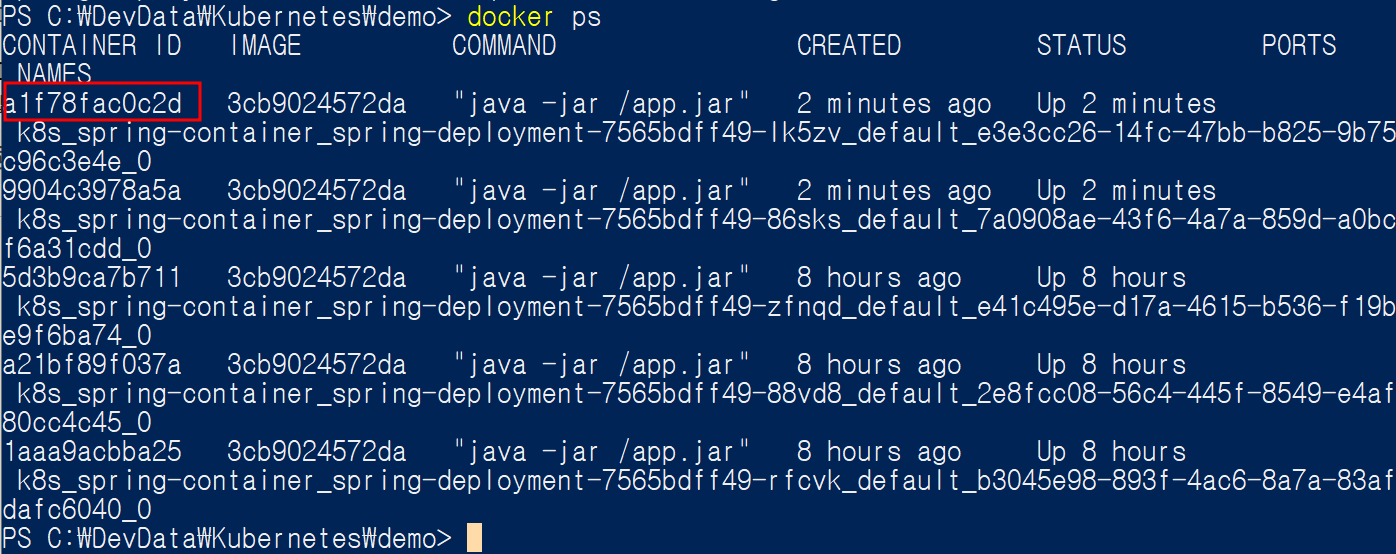


1. **서버가 죽었을 때 자동으로 복구하는 기능 (Self-Healing)**

### **✅ 실행되고 있는 파드 내 서버가 비정상적으로 종료된다면?**

1. **특정 파드의 컨테이너 종료시키기**
2. 실행 중인 컨테이너 조회하기

| $ docker ps |
| --- |



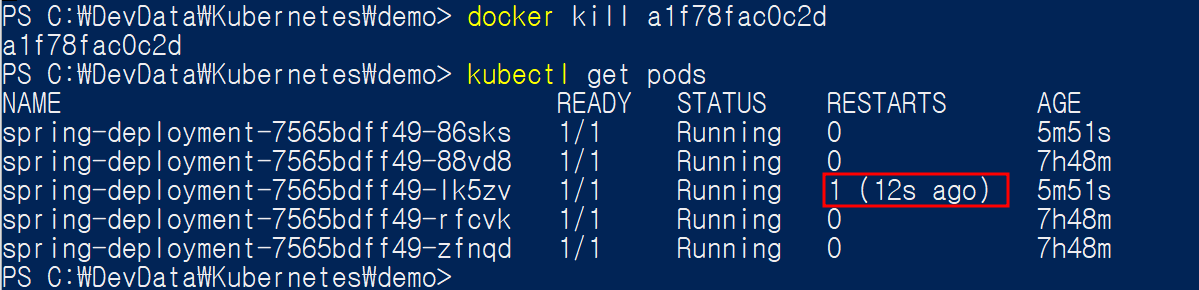
**빨간색 으로 선택한 컨테이너 죽이기**

1. 컨테이너 종료하기

| # docker kill [컨테이너 ID]  $ docker kill 8c085c887430 |
| --- |

1. **파드 조회하기**

| $ kubectl get pods |
| --- |

  
 파드를 조회해보니 여전히 5개의 파드가 작동하고 있는 걸 알 수 있다. 그런데 제일 첫 번째 파드를 보니 RESTARTS에 1이라고 기록되어 있다. 즉, 파드 내에 컨테이너가 작동하지 않음을 인식하고 컨테이너를 새로 만들어 서버를 재시작 시킨 것이다.

### **✅ 요약**

쿠버네티스는 파드 내의 컨테이너가 종료되면 자동으로 컨테이너를 재시작시킨다. 이 기능을 보고 쿠버네티스에서는 **셀프 힐링(Self-Healing)**이라고 한다. 즉, **자동 복구 기능**을 가지고 있다.

1. **새로운 버전의 서버로 업데이트 시키기**

🧑실제 서버를 운영하다보면 기능을 업데이트를 할 일이 많이 발생한다. 그럼 쿠버네티스에서는 새로운 버전의 백엔드 서버로 어떻게 업데이트 시키는 지 알아보자.

### **✅ 새로운 버전의 서버로 업데이트 시키기**

1. **코드 수정하기**

| AppController   package com.example.demo;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;  @RestController  public class AppController {  @GetMapping("/")  public String home() {  **return "Version 1.0";**  }  } |
| --- |

1. **Spring Boot 프로젝트 다시 빌드하기**

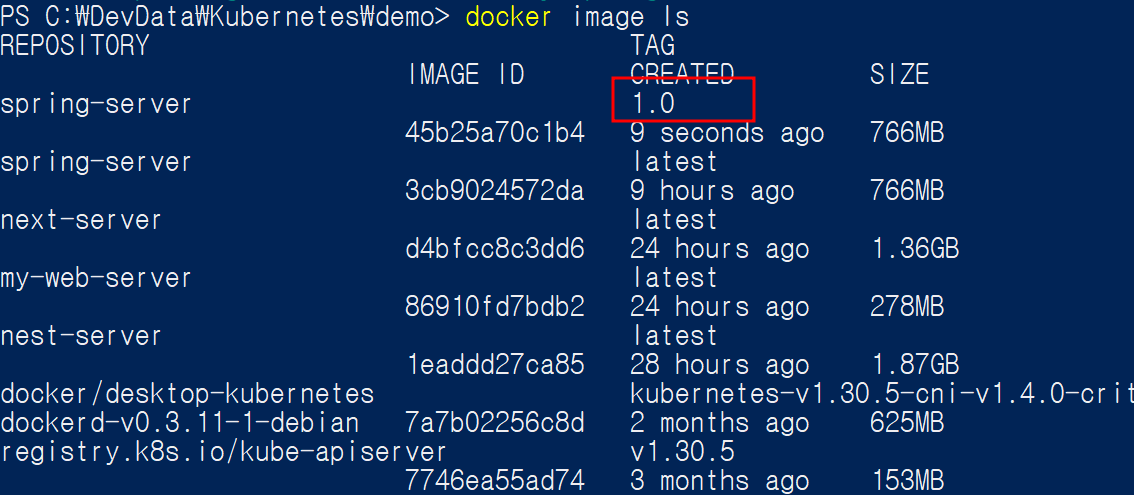
| $ ./gradlew clean build |
| --- |

1. **빌드된 jar 파일을 기반으로 새로 이미지 빌드하기**

| $ docker build -t spring-server:**1.0** . |
| --- |

1. **이미지가 잘 생성 됐는지 확인하기**

| $ docker image ls |
| --- |



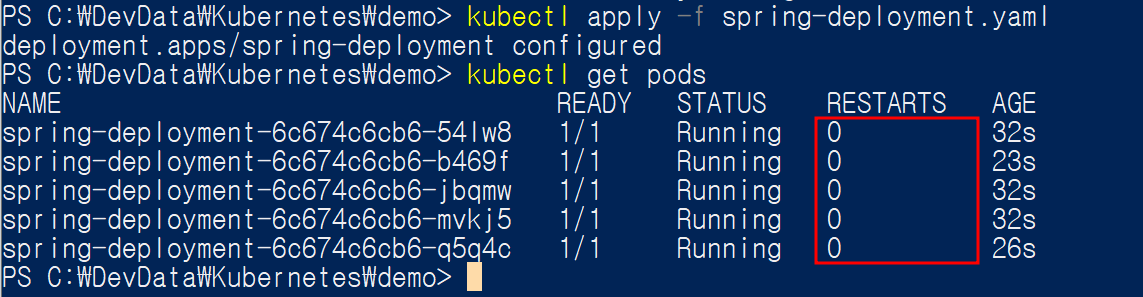
1. **기존 매니페스트 파일 수정하기**

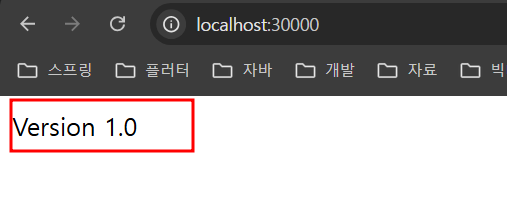
spring-deployment.yaml

| apiVersion: apps/v1  kind: Deployment  metadata:  name: spring-deployment  spec:  replicas: 5  selector:  matchLabels:  app: backend-app  template:  metadata:  labels:  app: backend-app  spec:  containers:  - name: spring-container  image: **spring-server:1.0**  imagePullPolicy: IfNotPresent  ports:  - containerPort: 8080 |
| --- |

1. **수정된 매니페스트 파일을 기반으로 업데이트하기**

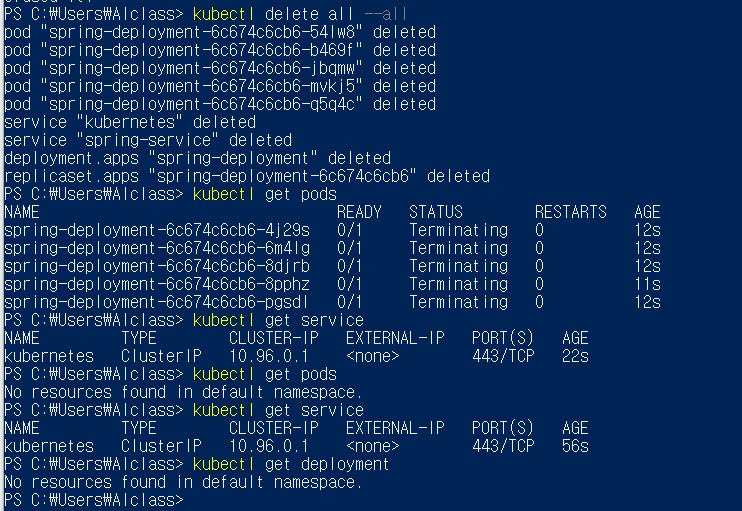
| $ kubectl apply -f spring-deployment.yaml |
| --- |



1. **업데이트 됐는 지 확인하기  
     
   한번더 해볼 것.**
2. **[예제] 디플로이먼트, 서비스를 활용해 백엔드(Nest.js) 서버 띄워보기**

### **✅ 디플로이먼트, 서비스를 활용해 백엔드(Nest.js) 서버 띄워보기**

| # 전부 삭제  $ kubectl delete all --all |
| --- |



이상태에서 진행 합니다.

[요구 사항]

* 파드 4개 띄우기
* 서비스(Service)를 활용해 http://localhost:31000에서 통신할 수 있도록 만들기

1. **Nest.js 프로젝트 만들기**

| # nest new {프로젝트명}  $ nest new nest-server |
| --- |

1. **프로젝트 실행시켜보기**

| $ npm i  $ npm run start |
| --- |

1. **Dockerfile 작성하기**

Dockerfile

| FROM node  WORKDIR /app  COPY . .  RUN npm install  RUN npm run build  EXPOSE 3000  ENTRYPOINT [ "node", "dist/main.js" ] |
| --- |

1. **.dockerignore 작성하기**

.dockerignore

| node\_modules |
| --- |

1. **Dockerfile을 바탕으로 이미지 빌드하기**

| $ docker build -t nest-server:1.0 . |
| --- |

1. **이미지가 잘 생성 됐는지 확인하기**

| $ docker image ls |
| --- |



1. **매니페스트 파일 생성하기**

nest-deployment.yaml

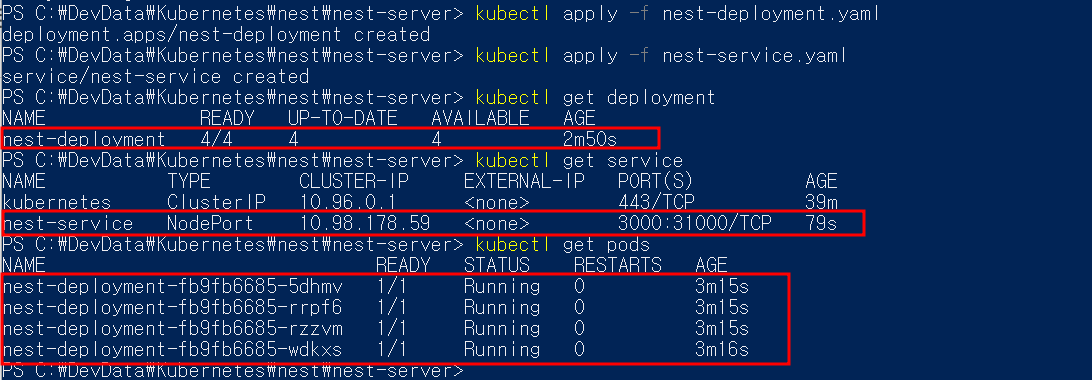
| apiVersion: apps/v1  kind: Deployment  # Deployment 기본 정보  metadata:  name: nest-deployment # Deployment 이름  # Deployment 세부 정보  spec:  replicas: 4 # 생성할 파드의 복제본 개수  selector:  matchLabels:  app: backend-app # 아래에서 정의한 Pod 중 'app: backend-app'이라는 값을 가진 파드를 선택  # 배포할 Pod 정의  template:  metadata:  labels: # 레이블 (= 카테고리)  app: backend-app  spec:  containers:  - name: nest-container # 컨테이너 이름  image: nest-server:1.0 # 컨테이너를 생성할 때 사용할 이미지  imagePullPolicy: IfNotPresent # 로컬에서 이미지를 먼저 가져온다. 없으면 레지스트리에서 가져온다.  ports:  - containerPort: 3000 # 컨테이너에서 사용하는 포트를 명시적으로 표현 |
| --- |

nest-service.yaml

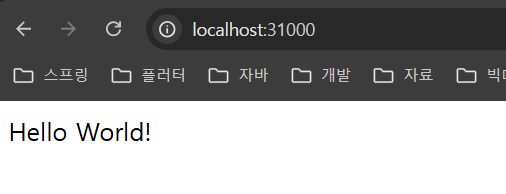
| apiVersion: v1  kind: Service  # Service 기본 정보  metadata:  name: nest-service    # Service 세부 정보  spec:  type: NodePort # Service의 종류  selector:  app: backend-app # 실행되고 있는 파드 중 'app: backend-app'이라는 값을 가진 파드와 서비스를 연결  ports:  - protocol: TCP # 서비스에 접속하기 위한 프로토콜  nodePort: 31000 # 외부에서 사용자들이 접근하게 될 포트 번호  port: 3000 # 쿠버네티스 내부에서 Service에 접속하기 위한 포트 번호  targetPort: 3000 # 매핑하기 위한 파드의 포트 번호 |
| --- |

1. **매니페스트 파일 기반으로 오브젝트 생성**

| $ kubectl apply -f nest-deployment.yaml  $ kubectl apply -f nest-service.yaml |
| --- |

****

1. 정상적으로 실행 됐는지 확인하기

****

### **✅ 업데이트 하기**

[요구 사항]

* 백엔드 서버 띄운 이후에 **Hello World!**라고 응답하는 서버에서 **Hi World!**라고 응답하는 서버로 업데이트 하기

1. **Nest.js 코드 수정하기**

app.service.ts (src 폴더 안에)

| import { Injectable } from '@nestjs/common';  @Injectable()  export class AppService {  getHello(): string {  return **'Hi World!'**;  }  } |
| --- |

1. **이미지 새로 빌드하기**

| $ docker build -t nest-server:**1.1** . |
| --- |



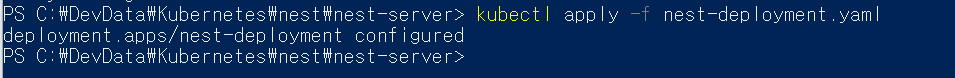
1. **매니페스트 파일 수정하기**

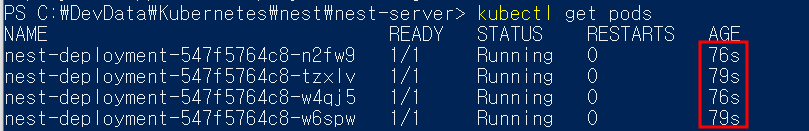
nest-deployment.yaml

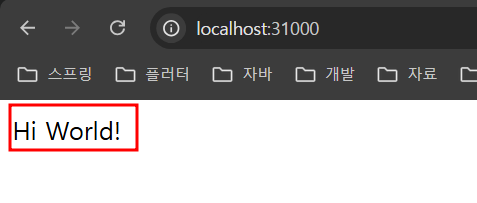
| apiVersion: apps/v1  kind: Deployment  # Deployment 기본 정보  metadata:  name: nest-deployment # Deployment 이름  # Deployment 세부 정보  spec:  replicas: 4 # 생성할 파드의 복제본 개수  selector:  matchLabels:  app: backend-app # 아래에서 정의한 Pod 중 'app: backend-app'이라는 값을 가진 파드를 선택  # 배포할 Pod 정의  template:  metadata:  labels: # 레이블 (= 카테고리)  app: backend-app  spec:  containers:  - name: nest-container # 컨테이너 이름  image: nest-server:**1.1** # 컨테이너를 생성할 때 사용할 이미지  imagePullPolicy: IfNotPresent # 로컬에서 이미지를 먼저 가져온다. 없으면 레지스트리에서 가져온다.  ports:  - containerPort: 3000 # 컨테이너에서 사용하는 포트를 명시적으로 표현 |
| --- |

1. **수정된 매니페스트 파일 적용시키기**

| $ kubectl apply -f nest-deployment.yaml  $ kubectl get pods |
| --- |

****

****

1. **업데이트 됐는 지 확인하기  
   **
2. **[요약] 지금까지 나온 명령어 정리**

**파드(Pod) 관련 명령어**

**✅ 파드 조회**

| $ kubectl get pods |
| --- |

#### **✅ 파드 내부로 접속**

| # kubectl exec -it [파드명] -- bash  $ kubectl **exec** -it nginx-pod -- **bash** |
| --- |

#### **✅ 파드 포트 포워딩**

| # kubectl port-forward pod/[파드명] [로컬에서의 포트]/[파드에서의 포트]  $ kubectl port-forward pod/nginx-pod **80**:80 |
| --- |

#### **✅ 파드 삭제**

| # kubectl delete pod [파드명]  $ kubectl delete pod nginx-pod # nginx-pod라는 파드 삭제 |
| --- |

**디플로이먼트(Deployment) 관련 명령어**

#### **✅ 디플로이먼트 조회**

| $ kubectl get deployment |
| --- |

#### **✅ 디플로이먼트 삭제**

| # kubectl delete deployment [디플로이먼트명]  $ kubectl delete deployment spring-deployment # spring-deployment라는 디플로이먼트 삭제 |
| --- |

**서비스(Service) 관련 명령어**

**✅ 서비스 조회**

| $ kubectl get **service** |
| --- |

#### **✅ 서비스 삭제**

| # kubectl delete service [서비스명]  $ kubectl delete **service** spring-service # spring-service라는 서비스 삭제 |
| --- |

**공통 명령어**

**✅ 매니페스트 파일에 적혀져있는 리소스(파드 등) 생성**

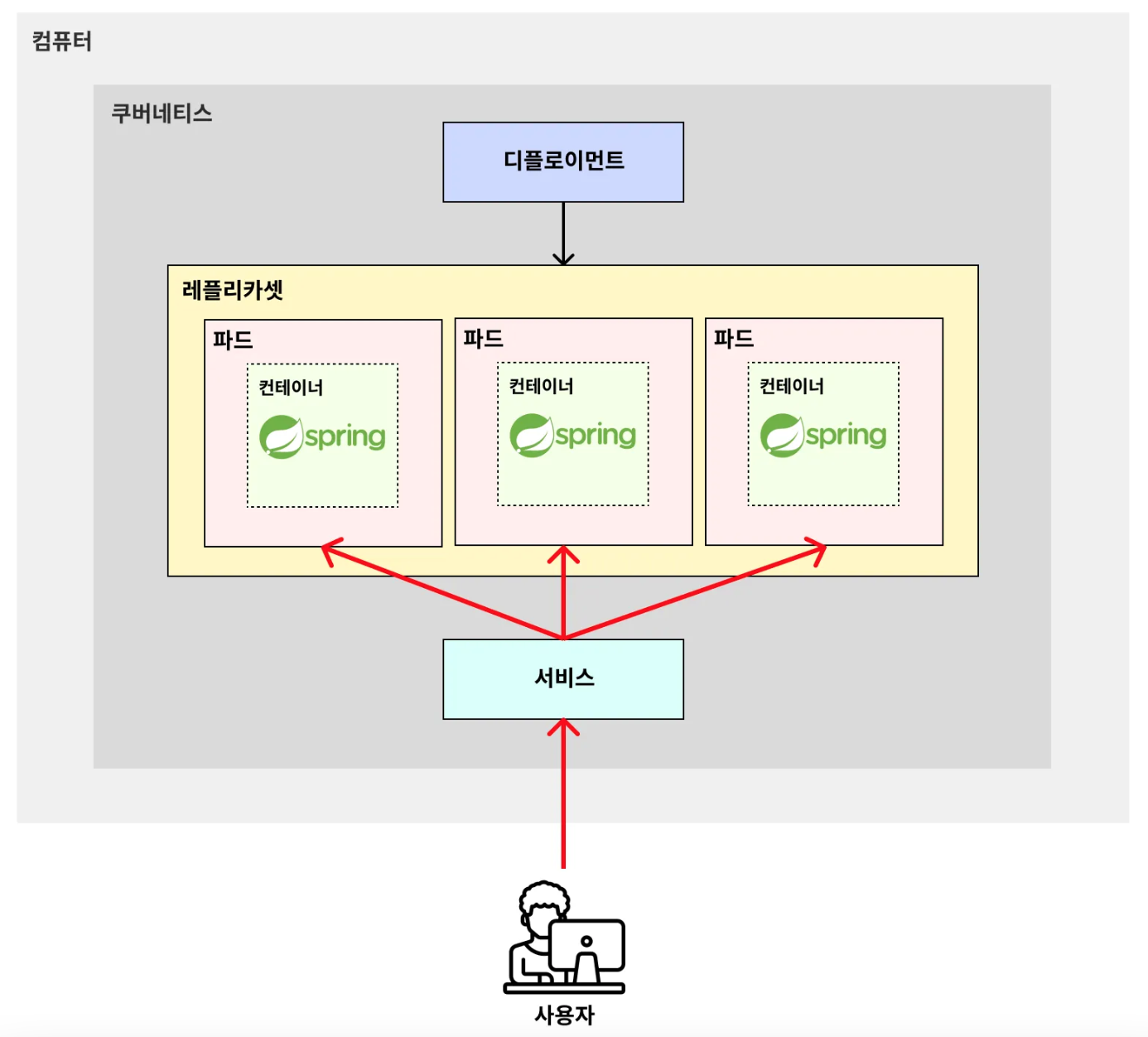
| # kubectl apply -f [파일명]  $ kubectl apply -f nginx-pod.yaml |
| --- |

#### **✅ 모든 리소스 삭제**

| $ kubectl delete all --all |
| --- |

1. **[요약] 파드(Pod), 디플로이먼트(Deployment), 서비스(Service) 개념 정리**

### **✅ 쿠버네티스에서의 핵심 개념**

****

* 파드(Pod) : 일반적으로 쿠버네티스에서 하나의 프로그램을 실행시키는 단위 (쿠버네티스에서 가장 작은 단위)
* 디플로이먼트(Deployment) : 파드를 묶음으로 쉽게 관리할 수 있는 기능
* 서비스(Service) : 외부로부터 들어오는 트래픽을 받아, 파드에 균등하게 분배해주는 로드밸런서 역할을 하는 기능

| **💡**[First Word 법칙]  쿠버네티스에서는 **서비스(Service), 디플로이먼트(Deployment), 파드(Pod)와 같은 리소스**를 보고 **오브젝트(Object)**라고 부른다. |
| --- |

### **✅ 학습 Tip) 쿠버네티스의 핵심 개념에만 우선 집중하자.**

쿠버네티스에서 위 개념 말고도 **스테이트풀셋(StatefulSet), 잡(Job)과** 같은 다양한 개념이 존재한다. OT에서 얘기한 파레토의 법칙에 따르면 이 모든 개념을 처음부터 다 흡수하려고 할 필요 없다. 현업에서 가장 많이 사용되고 중요한 개념 위주로 먼저 배우고 익숙해지는 게 중요하다.

쿠버네티스를 학습할 때 발목을 붙잡는 건 모든 개념을 한 번에 다 익히려고 하는 욕심 때문에 발생한다. 자주 쓰이는 개념 가지고 이것저것 만들어보면서 익숙해진 다음에 새로운 개념들을 하나씩 하나씩 추가적으로 학습해나가야 한다. 그래야 빨리 배울 수 있다.