2024.06.12

캡스톤 디자인 I
최종 발표

백엔드 서비스 프로바이더
: 하향식 기반 방법론을 적용한 경량형 컨테이너 서비스 개발

팀명 : 악동개발자
20211939 허유정
20217140 김창인

프로젝트 개요

1) 프로젝트 배경
2) 프로젝트 목표
3) 주요 기능 목록

프로젝트 결과

1) 메타버스 게임 서버 구축
2) 이미지 빌드
3) 데모 영상

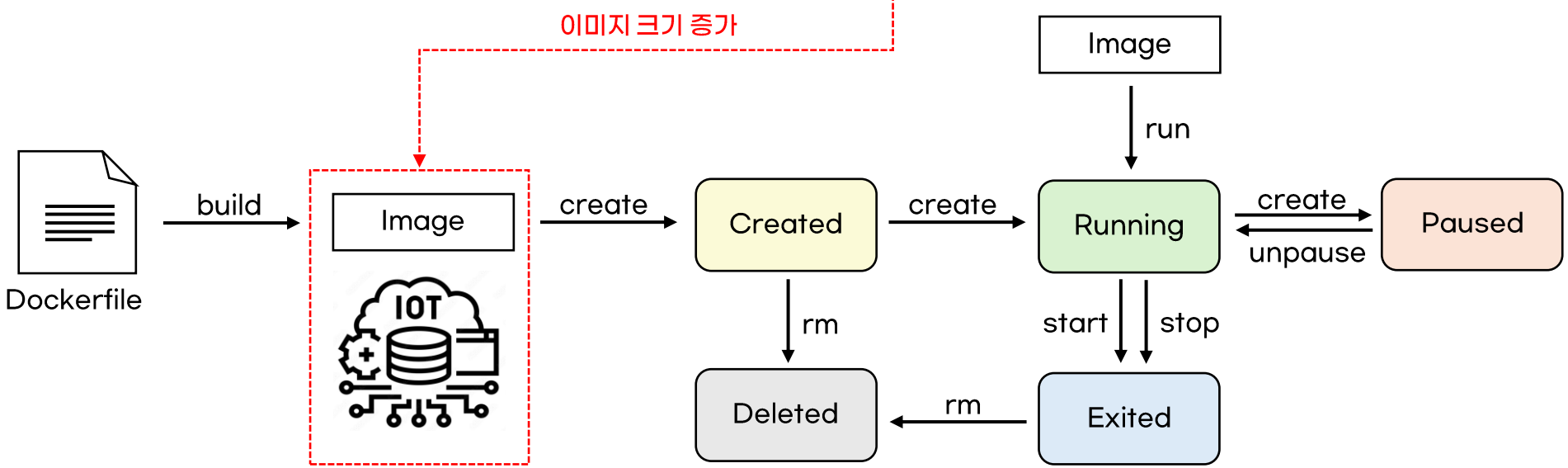
결론

1) 진행 계획
2) 팀원의 역할
3) 기대효과
4) 우수성 입증 자료

프로젝트 개요 - 프로젝트 배경

[프로젝트 배경]

- 컨테이너 기술의 등장으로 애플리케이션의 빠른 배포와 확장, 일관된 환경 실행으로 **서버 관리의 효율성** 증대
- IoT 서비스 개발에 따라 제한된 자원을 가지고 있으므로 **효율적인 메모리**를 필요로 하는 컨테이너 개발 및 배포 환경이 필요함
- 기존 컨테이너 기술의 경우 대규모 이미지를 저장하고 실행하는데 **제약사항**이 생기며 전체적인 성능에 영향을 줌

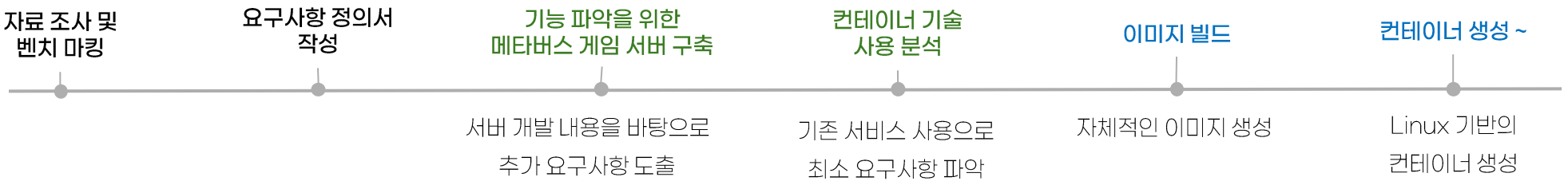


< 이미지 기반의 컨테이너 생성 과정 >

프로젝트 개요 - 프로젝트 목표

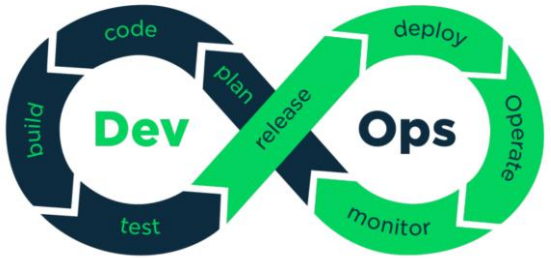
[프로젝트 목표]

- 소규모 프로젝트나 조직에 부담되지 않으며 메모리의 효율적 사용을 위한 **경량화 된 서비스 개발**을 목표로 함
- 자원의 효율성, 자동화된 관리, 지속적인 모니터링을 위해 **DevOps 방법론**을 사용하여 더 나은 서비스 제공을 위함
- 하향식 개발 방식으로 서비스 제공을 위한 **최소 요구사항을 파악**하고 균일한 조건에서의 테스트를 필요로 함



경량형 컨테이너 서비스

- 요구사항 분석 및 설계
- 이미지 빌드
- 컨테이너 런타임 관리
- 명령어 인터페이스



메타버스 게임 서버 개발

- 컨테이너 최소요건 충족을 위한 게임 서버 구축 (컨테이너 환경에서의 일관성 유지 위함)
- 지속적인 기능 테스트(안정성 유지)
- 지속적인 모니터링 및 로깅

프로젝트 개요 - 주요 기능 목록

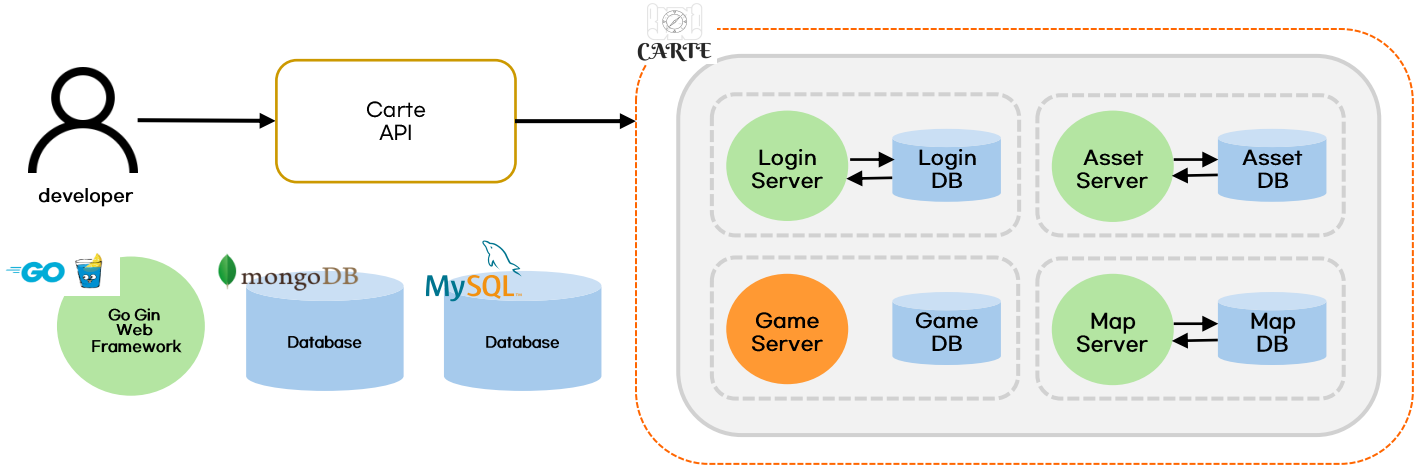
[요구사항 정의서]

요구사항 명	요구사항 설명	요구사항 ID
경량화 된 이미지 빌드	애플리케이션을 컨테이너 이미지로 빌드하는 기능	REQ-001
컨테이너 런타임 엔진	실행 중인 컨테이너에 대한 기본적인 관리 기능	REQ-002
스케일링	확장성을 위한 오케스트레이션 기능으로 컨테이너 인스턴스 조정 자동화	REQ-003
CLI	사용자가 명령줄에서 컨테이너 관리 기능을 사용할 수 있는 CLI 제공	REQ-004
운영 테스트(Ops)	메타버스 게임 서버 구축과 지속적인 테스트를 통한 추가 요구사항 도출	REQ-005

프로젝트 결과 - 메타버스 게임 서버 구축

[메타버스 게임 서버 구축]

- 컨테이너 기술 기능 파악
  - 상용화된 컨테이너 런타임 도구를 활용하여 컨테이너 기술 개발의 **최소 요구사항을 파악**하기 위함
  - 메타버스 게임의 직관적인 UI로 실시간 **모니터링 및 협업 촉진 가능**
- 사용자 테스트
  - 다수의 학생들을 대상으로 한 동시접속자 테스트를 진행하고 진행 로그 파일을 통해 운영 과정을 확인 및 예외 상황을 식별 가능

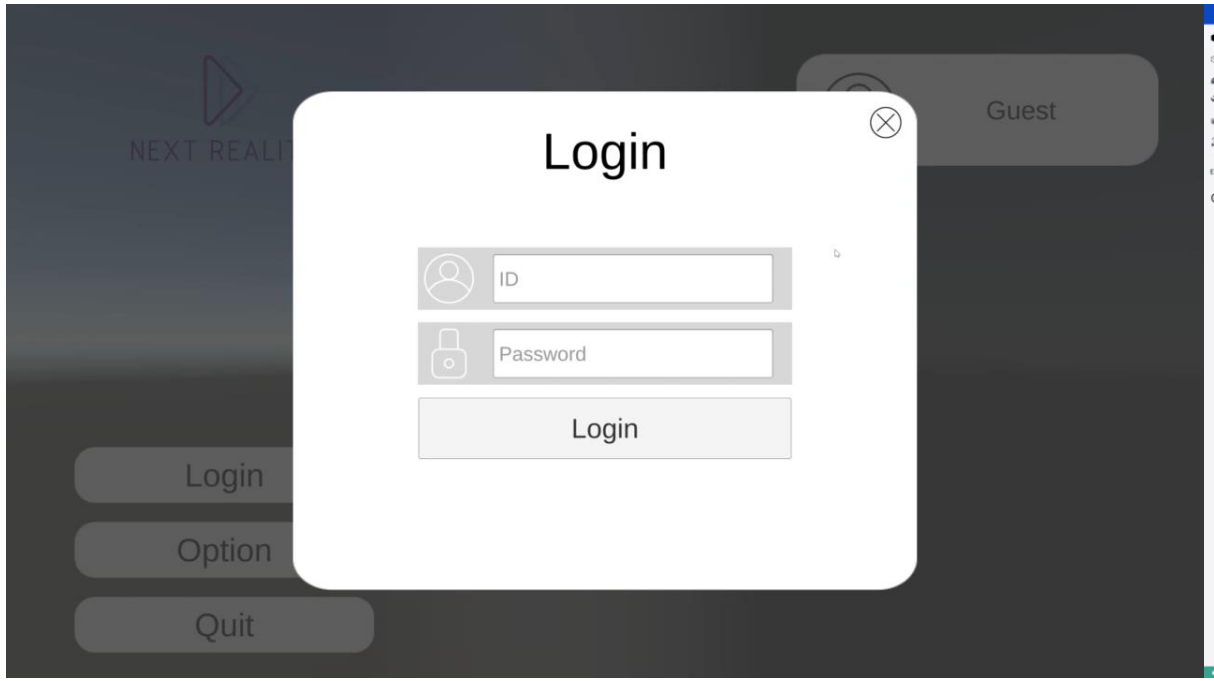


< 메타버스 게임 서버 구조도 >

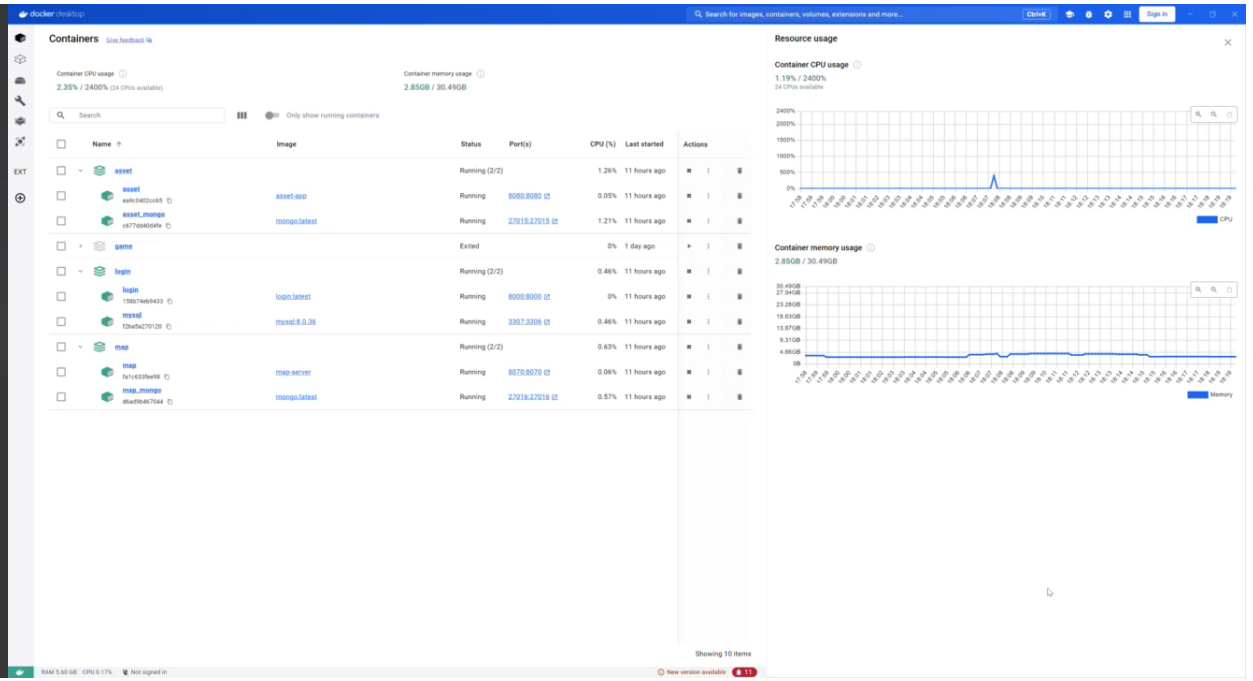
프로젝트 결과 - 메타버스 게임 서버 구축

- 메타버스 게임 서버의 사용성을 검증하기 위해 약 80명의 학생을 대상으로 테스트 진행
- 사용자 관점에서 경험 가능한 상황을 확인하고 개선 포인트 도출을 위함
- 다수의 사용자가 동시에 Login 및 여러 개의 Asset down 을 통한 사용성 테스트 진행

part2 프로젝트 결과 - 데모 영상



Game UI



Server Log & Monitoring

프로젝트 결과 - 이미지 빌드

[Cartefile 구성요소]

	Cartefile	Dockerfile
빌드 명령어	Carte build	docker build -t {이미지명:태그명}{dockerfile의 경로}
기본 명령	FROM, NAME, PATH, exe_PATH	FROM, RUN, CMD, LABEL, EXPOSE, ENV, ADD, ENTRYPOINT, VOLUME, USER, WORKDIR, , ARG, ONBUILD, STOPSIGNAL, HEALTHCHECK, SHELL

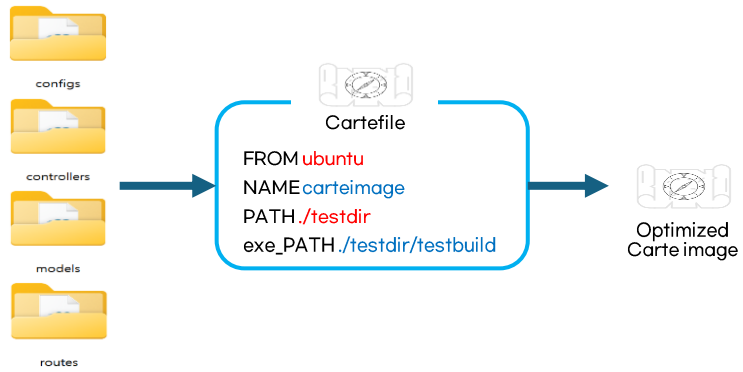
메타버스 게임 서버를 기존 컨테이너 기술에 접목시켜 서비스 개발에 필요한 최소 요건 파악

여러 레이어로 구성된 이미지 생성을 통해 수정되는 요구사항에 신속한 대응이 가능하도록 함

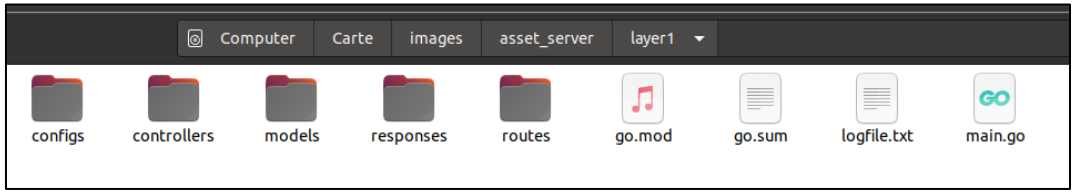
명령어 단순화와 불필요한 파일 제거를 통한 이미지 크기 축소를 목표로 함

프로젝트 결과 - 이미지 빌드

[Layer 구성]

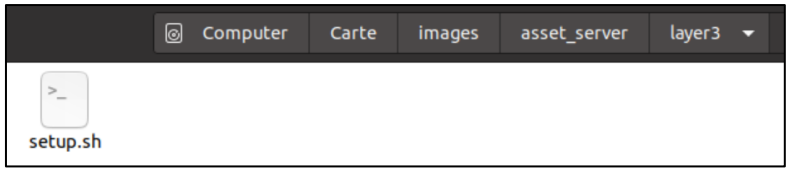


메타 데이터



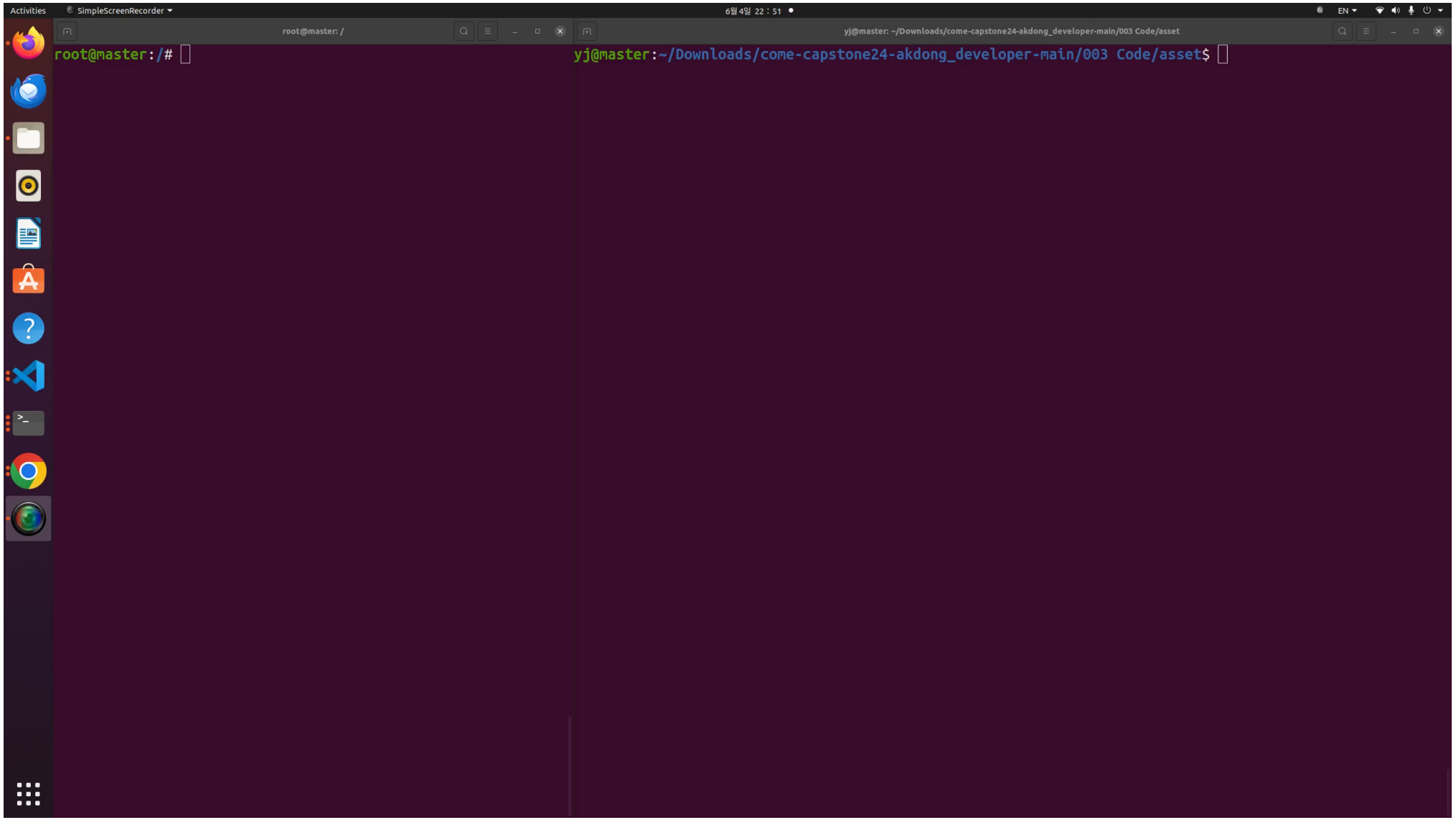
Layer 1 : 빌드 파일을 위한 디렉토리 파일 구성

Layer 2 : 빌드 파일

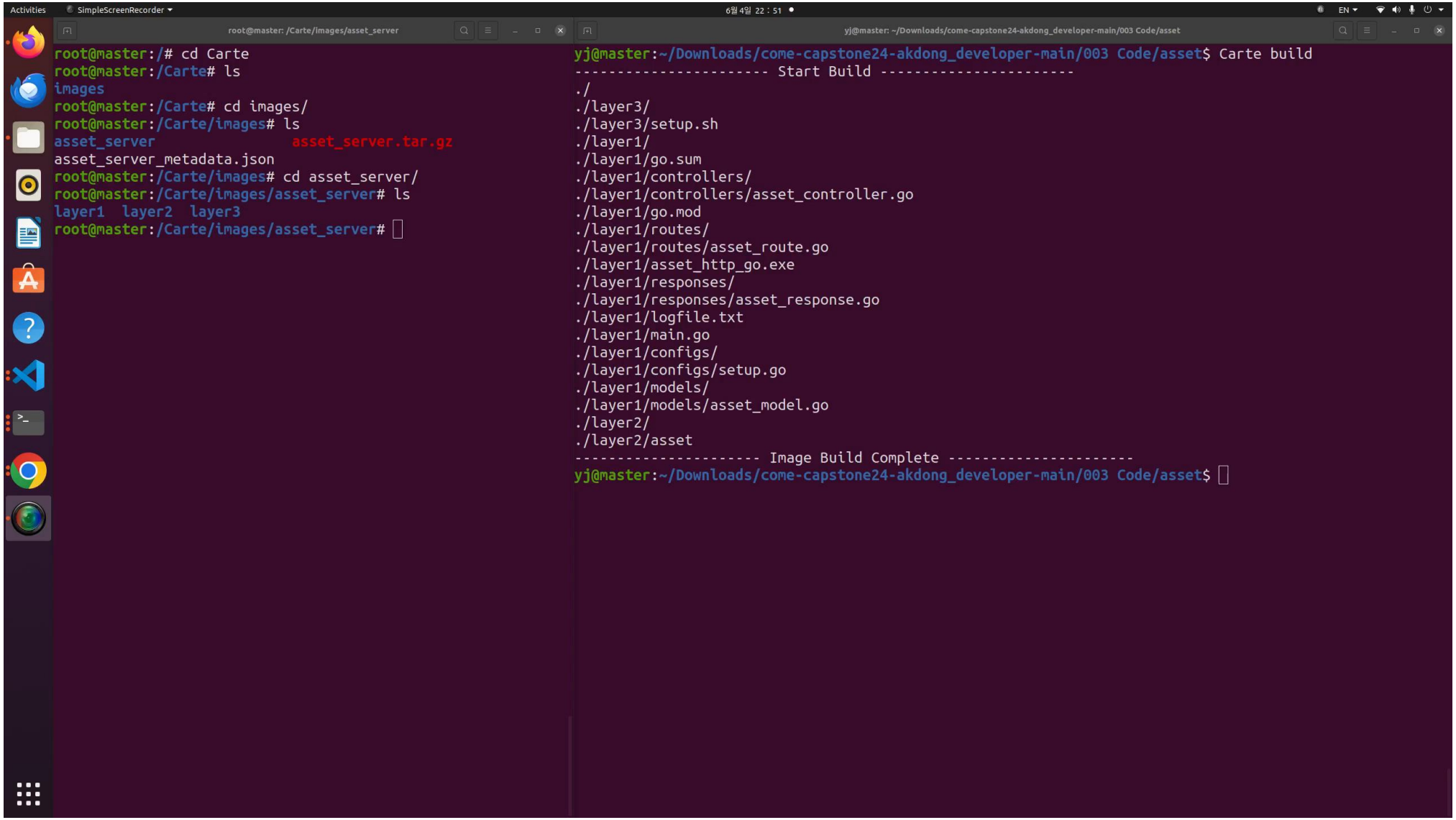


Layer 3 : 실행 명령어

part2 프로젝트 결과 - 데모 영상



part2 프로젝트 결과 - 데모 영상



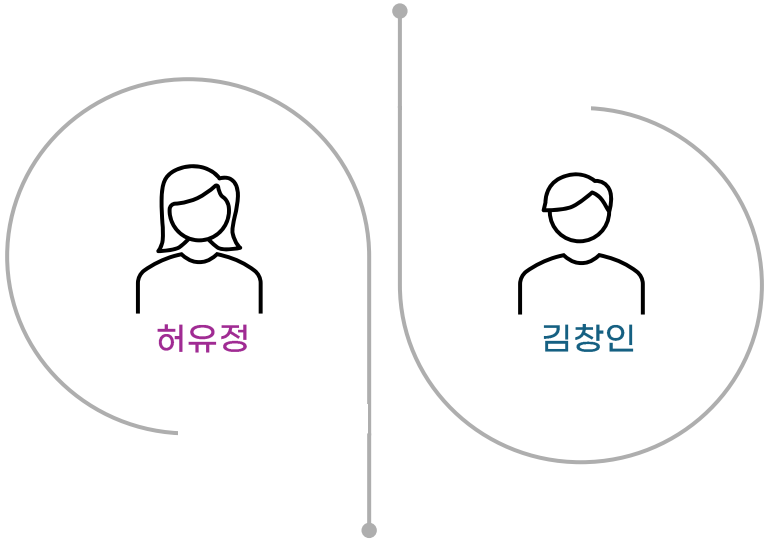
결론 - 진행 계획

구분	1월	2월	3월	4월	5월	6월	7월	8월	9월	10월	11월
계획 및 서비스 벤치마킹											
컨테이너 기술 관련 학습 및 언어 학습											
계획 요구사항 정의											
메타버스 게임 서버 구축											
추가 요구사항 추가											
컨테이너 이미지 빌드											
컨테이너 런타임											
CLI											
최종 테스트											

결론 - 팀원의 역할

[팀원의 역할]

- **메타버스 게임서버**
  - Login, Map 서버 구축
  - 이미지 빌드 요구사항 분석
- **컨테이너 이미지 빌드**
  - 레이어 계층 분류
  - 명령줄 인터페이스(CLI) 구성
  - 이미지 파일 생성 및 빌드



- **메타버스 게임서버**
  - Asset 서버 구축
  - 이미지 빌드 요구사항 분석
- **컨테이너 이미지 빌드**
  - 레이어 계층 분류
  - 이미지 파일 생성 및 빌드
  - 컨테이너 실행 구조 확립

결론 - 기대효과

[기대효과]

- 소규모 프로젝트나 조직에 부담되지 않는 **경량화 된 서비스 개발로 효율적인 메모리 관리와, 생산성을 향상**시킬 것으로 기대함
- 특허 및 실용신안 출원을 통해 개발된 기술의 **지적 재산권을 확보**함으로써, 기술적 리더십을 확립하고 장기적인 사업화 기반 마련 가능
- 컨테이너 **기술 교육 자료로 활용**하여 실습 중심의 학습의 학생들에게 인프라 관리 기술을 가르치는데 유용한 도구가 될 수 있을 것으로 기대함

결론 - 우수성 입증 자료

[춘계공동 학술대회 참여]

AI 통합 지능형 컨테이너 오케스트레이션 방법론: IoT 서비스 최적화 설계

본 연구는 IoT 서비스의 성능 및 확장성을 극대화하기 위해 AI 기반 알고리즘과 엣지 컴퓨팅을 통합한 지능형 컨테이너 오케스트레이션 방법론을 제안한다. IoT 디바이스의 실시간 데이터 처리 요구와 자원 사용 패턴을 분석하여, 엣지 컴퓨팅 환경에서 컨테이너 리소스의 동적 할당 및 관리를 최적화함으로써, IoT 애플리케이션의 효율성을 증대시키고 네트워크 부하를 줄인다. 또한, 전체 시스템의 에너지 효율성을 개선하고, 지속 가능한 디지털 전환을 위한 실질적인 IoT 서비스 솔루션을 제공하는 것을 목표로 한다

이산사건 시스템 형식론 기반 컨테이너 분산 예측 방법론

컨테이너 기술의 중요성이 증가함에 따라, 이를 효율적으로 관리하기 위한 컨테이너 오케스트레이션 플랫폼의 중요성이 대두되고 있다. 본 논문에서는 자원의 효율적인 분산 요구에 주목하여 이산사건 시스템 형식론에 기반한 컨테이너 분산 방법론을 제시한다. 컨테이너 환경에서 발생하는 이벤트인 컨테이너 생성, 시작, 삭제, 오류 구분을 위해 리소스 사용량 데이터를 기반으로 하여 성능을 예측하고 시스템 특성 파악이 가능하도록 설계한다. 컨테이너 상태 변화에 따른 시스템 전체 자원 업데이트로 향후 자원 요구 사항을 예측하고, 필요 자원을 동적으로 할당함으로써 시스템의 효율성을 높일 수 있다.

2024.06.12

감사합니다

부록

[벤치마킹 세부사항_컨테이너 관리 플랫폼(1)]

	Amazon Elastic Container Service(Amazon ECS)	Amazon Elastic Kubernetes Service(Amazon EKS)	Azure Kubernetes Service(AKS)
정의	- Docker 컨테이너 지원 - Amazon EC2 인스턴스 클러스터에서 응용 프로그램 실행 가능한 컨테이너 관리 서비스	쿠버네티스 응용 프로그램 및 관리 인프라를 실행하기 위한 컨테이너 오케스트레이션	- 쿠버네티스 서비스를 사용하여 컨테이너화된 어플리케이션 - 쉬운 배포/관리 가능
장점	- k8s보다 간단한 개념 - AWS서비스에서의 통합 간편	- 계획적으로 애플리케이션을 격리함으로써 보안 성능을 향상 - 쿠버네티스 Control Plane설치 및 운영 없이 AWS에서 관리 가능	- 완성된 클러스터 제공으로 편리한 유지보수 - Azure의 Master관리로 개발자의 부담 낮춤 - Node Scaling의 자동화
단점	- 사용량에 따라 비용 발생 - AWS 제품에 종속	- 사용량에 따라 비용 발생 - AWS 제품에 종속	- 사용량에 따라 비용 발생 - AWS 제품에 종속

부록

[벤치마킹 세부사항_컨테이너 관리 플랫폼(2)]

	Rancher	Portainer CE	Harshicorp nomad
정의	- 쿠버네티스 업그레이드, 백업 및 배포 가능 - kubernetes 환경을 지원할 수 있는 멀티 클러스터 컨테이너 관리 플랫폼	쉽게 도커를 배포하고 구성할 때 사용하는 웹 UI 기반 관리 툴	클라우드 플랫폼에서 컨테이너 대규모 배포 및 관리 가능한 오케스트레이터
장점	- 자체적인 일반 계정 시스템 보유 - 앱 카달로그 지원 - 지표 기반의 모든 클러스터 관리 가능	- 도커 환경을 GUI로 보여줌으로써 간편한 사용 가능	- 작업스케쥴링에 중점 - 구조적 단순함
단점	- 많은 서버 리소스 요구 - 쿠버네티스 관리 중점	대규모 컨테이너 배포에 부적합	- 특정 기능이나 확장성 측면에서 제한적 - 유료 구독 필요

부록

[요구사항 정의서 세부사항]

컨테이너 런타임 엔진(REQ-002)

요구사항 명	요구사항 설명	요구사항 ID
컨테이너 시작	사용자가 직접 컨테이너를 시작할 수 있는 기능 추가 제공	REQ-002-001
컨테이너 정지	실행 중인 컨테이너를 정지할 수 있는 기능	REQ-002-002
컨테이너 재시작	정지된 컨테이너를 재시작하여 가동할 수 있는 기능	REQ-002-003
컨테이너 삭제	더 이상 필요하지 않은 컨테이너를 사용자가 직접 삭제할 수 있는 기능	REQ-002-004

스케일링(REQ-003)

요구사항 명	요구사항 설명	요구사항 ID
수동 스케일링	사용자가 수동으로 컨테이너의 인스턴스 수를 조정할 수 있는 기능	REQ-003-001
자동 스케일링	시스템이 자동으로 한정된 리소스에 맞추어 인스턴스 수 조정할 수 있는 기능	REQ-003-002

부록

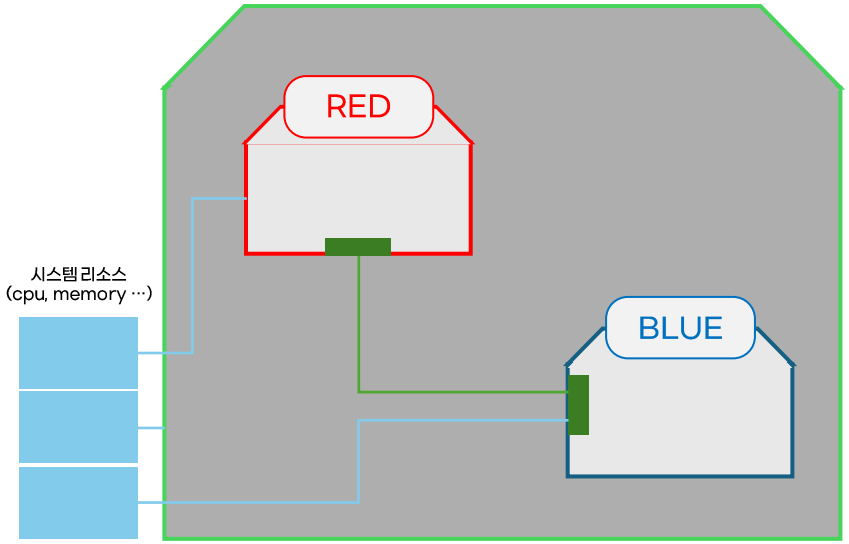
[요구사항 정의서 세부사항]

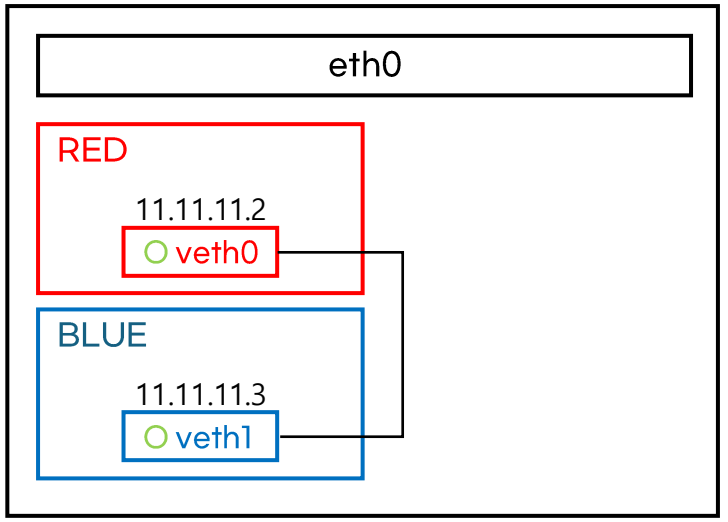
운영 테스트 Ops (REQ-005)

요구사항 명	요구사항 설명	요구사항 ID
Login Server	- 사용자 접속 정보 관리 기능 - 입력된 사용자 정보 토대로 일치 여부 확인 기능	REQ-005-001
Asset Server	- Asset 업로드 및 다운로드 관리 - Asset 정보 관리 기능 - 카테고리별 에셋 조회 관리	REQ-005-002
Map Server	- 다중 사용자 접속 가능 Map - 제작자와 Map 데이터를 기준으로 정보 저장 기능	REQ-005-003
Game Server	- 다중 사용자 로그인 확인 - 여러 사용자와의 소통 기능 및 실시간 정보 업데이트 가능	REQ-005-004

부록

[컨테이너 생성 환경 구축]

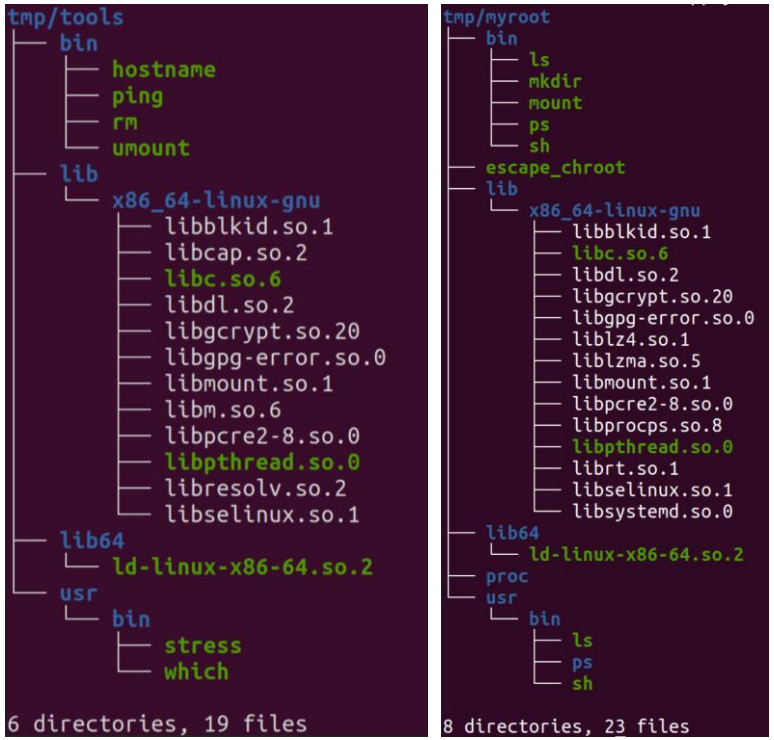




cgroup으로 자원 할당된 RED, BLUE 컨테이너 생성과 1:1 통신을 통한 컨테이너 생성 기능을 확보하고자 함

부록

[컨테이너 제작]



(lower layer) tools, myroot 이미지 사용

Cgroup 생성 및 설정

```
root@worker2:~# mkdir /sys/fs/cgroup/cpu/blue
root@worker2:~# mkdir /sys/fs/cgroup/memory/blue
root@worker2:~# echo 40000 > /sys/fs/cgroup/cpu/blue/cpu.cfs_quota_us;
root@worker2:~# echo 209715200 > /sys/fs/cgroup/memory/blue/memory.limit_in_bytes
root@worker2:~# echo 0 > /sys/fs/cgroup/memory/blue/memory.swappiness
```

- CPU : 40% 제한
- Memory : 200MB 제한 swap off (메모리 부족 시, 프로세스 종료)

mount / uts / ipc / fork pid namespace

```
root@worker2:~# unshare -m -u -i -fp nsenter --net=/var/run/netns/BLUE /bin/sh;
# echo "1" > /sys/fs/cgroup/cpu/blue/cgroup.procs;
# echo "1" > /sys/fs/cgroup/memory/blue/cgroup.procs;
```

- 네트워크 : nsenter을 사용하여 var/run안의 BLUE 사용
- shell 실행

부록

[컨테이너 제작]

Overlay mount

```
# mkdir /bluefs
# mkdir /bluefs/container
# mkdir /bluefs/work
# mkdir /bluefs/merge
# mount -t overlay overlay -o lowerdir=tmp/tools:tmp/myroot,upperdir=/bluefs/container,workdir=/bluefs/work /bluefs/merge
# tree /bluefs/merge
/bluefs/merge
├── bin
│   ├── hostname
│   ├── ls
│   ├── mkdir
│   ├── mount
│   ├── ping
│   ├── ps
│   ├── rm
│   ├── sh
│   └── umount
├── escape_chroot
├── lib
│   └── x86_64-linux-gnu
│       ├── libblkid.so.1
│       ├── libcap.so.2
│       ├── libc.so.6
│       ├── libdl.so.2
│       ├── libgcrypt.so.20
│       ├── libgpg-error.so.0
│       ├── liblz4.so.1
│       ├── liblzma.so.5
│       ├── libmount.so.1
│       ├── libm.so.6
│       ├── libpcre2-8.so.0
│       ├── libprocps.so.8
│       ├── libpthread.so.0
│       ├── libresolv.so.2
│       ├── librt.so.1
│       ├── libselinux.so.1
│       └── libsystemd.so.0
├── lib64
│   └── ld-linux-x86-64.so.2
├── proc
└── usr
    └── bin
        ├── ls
        ├── ps
        ├── sh
        ├── stress
        └── which
```

overlay mount를 위한 디렉토리

merge

container

tools

myroot

pivot_root

```
# mkdir -p /bluefs/merge/put_old
# cd /bluefs/merge
# pivot_root . put_old;
# cd /
# ls put_old
bin     boot   dev  home  lib32  libx32      media  opt   redfs  run   snap  swapfile  tmp  var
bluefs  cdrom  etc  lib   lib64  lost+found  mnt    proc  root   sbin  srv   sys       usr
# mount -t proc proc /proc;
# umount -l put_old
# rm -rf put_old
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
0              1       0  0 06:31 ?        00:00:00 /bin/sh
0             18       1  0 06:37 ?        00:00:00 ps -ef
```

- 새로운 루트 파일 시스템 : overlay mount에서 생성함(merge: mount point)
- Mount point 진입 후, 현재 경로를 새로운 루트파일 시스템으로 놓고 기존 것을 put_old에 넣은 후 제거
  [put_old에는 현재 컨테이너가 속한 host의 root filesystem이 부착되어 있으므로 보안을 위해 제거함 : umount진행]

부록

[컨테이너 제작]

RED-BLUE 통신

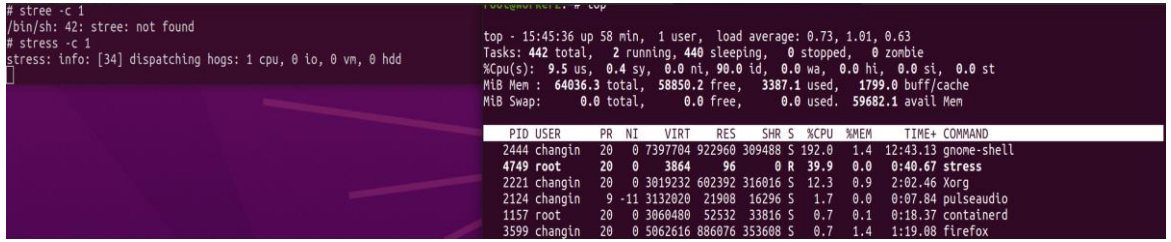
```
# ping 11.11.11.3
PING 11.11.11.3 (11.11.11.3) 56(84) bytes of data.
64 bytes from 11.11.11.3: icmp_seq=1 ttl=64 time=0.068 ms
64 bytes from 11.11.11.3: icmp_seq=2 ttl=64 time=0.047 ms
64 bytes from 11.11.11.3: icmp_seq=3 ttl=64 time=0.043 ms
64 bytes from 11.11.11.3: icmp_seq=4 ttl=64 time=0.025 ms
64 bytes from 11.11.11.3: icmp_seq=5 ttl=64 time=0.051 ms
64 bytes from 11.11.11.3: icmp_seq=6 ttl=64 time=0.064 ms
64 bytes from 11.11.11.3: icmp_seq=7 ttl=64 time=0.056 ms
64 bytes from 11.11.11.3: icmp_seq=8 ttl=64 time=0.051 ms
64 bytes from 11.11.11.3: icmp_seq=9 ttl=64 time=0.046 ms
```

- RED -> BLUE

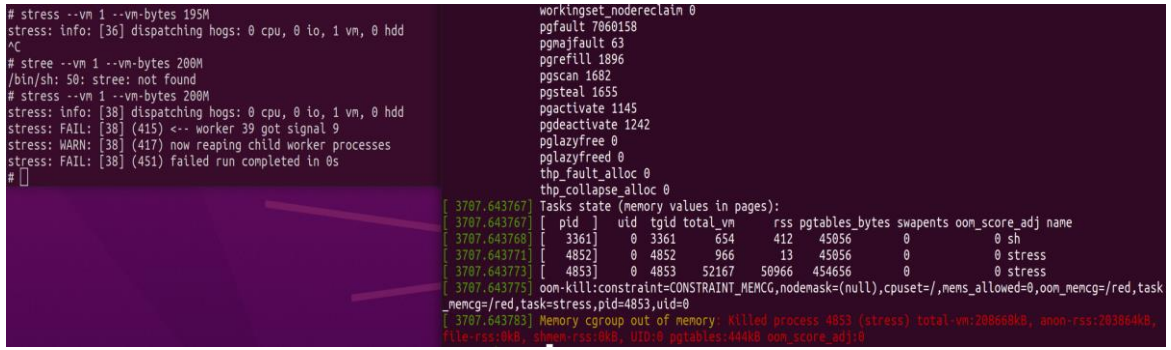
```
# ping 11.11.11.2
PING 11.11.11.2 (11.11.11.2) 56(84) bytes of data.
64 bytes from 11.11.11.2: icmp_seq=1 ttl=64 time=0.043 ms
64 bytes from 11.11.11.2: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 11.11.11.2: icmp_seq=3 ttl=64 time=0.056 ms
64 bytes from 11.11.11.2: icmp_seq=4 ttl=64 time=0.024 ms
64 bytes from 11.11.11.2: icmp_seq=5 ttl=64 time=0.046 ms
64 bytes from 11.11.11.2: icmp_seq=6 ttl=64 time=0.054 ms
64 bytes from 11.11.11.2: icmp_seq=7 ttl=64 time=0.059 ms
64 bytes from 11.11.11.2: icmp_seq=8 ttl=64 time=0.058 ms
64 bytes from 11.11.11.2: icmp_seq=9 ttl=64 time=0.047 ms
```

- BLUE -> RED

CPU 리소스, 메모리 리소스 확인



- CPU 리소스 확인



- 메모리 리소스 확인(200M 초과시 프로세스 종료)