UE5 Dedicated Server 기반 포트폴리오 개요 (Project Overview)

포트폴리오 시연 영상 : **포트폴리오 시연 유튜브 영상 링크 (클릭)**

깃 허브 : **깃 허브 프로젝트 링크 (클릭)**

본 프로젝트는 **UE5 Dedicated Server 환경에서 동작**하는 **개인전(Free-for-All) PvPvE 기반 Flag Capture 게임**입니다.

플레이어는 동일한 목표를 두고 경쟁하지만, 전장에는 **다른 플레이어(PvP)** 뿐만 아니라 **좀비 AI(PvE)** 가 동시에 존재하도록 설계되어 있습니다.

이 프로젝트의 목적은 단순히 “멀티플레이 게임을 구현해 보았다” 는 결과를 보여주는 것이 아니라,

멀티플레이 게임에서 필수적으로 요구되는 구조적 설계와 권한 분리 원칙을 실제 코드와 동작으로 증명하는 데 있습니다.

특히 다음과 같은 설계 목표를 중심으로 프로젝트를 구성하였습니다.

- 서버 권위(Server Authority)를 끝까지 유지하는 구조
- 모든 플레이어 상태를 PlayerState 단일 진실(SSOT) 기준으로 관리
- Lyra 스타일 아키텍처의 실전 적용
  - Experience 기반 게임 흐름 구성
  - GameFeature 단위 기능 분리
  - GAS(Gameplay Ability System)를 활용한 전투 및 상태 처리
- 매치 시작부터 종료 이후까지 이어지는
- 완결된 게임 세션 라이프사이클
  - 시작 → 경쟁 → 종료 → 결과 집계 → 기록 저장 → 재접속

위의 모든 흐름은 코드 구조, 서버 로그, 실제 플레이 동작 영상을 통해 검증 가능하도록 설계되었습니다.

핵심 게임 목표 (Core Gameplay Goal)

본 게임의 승리 조건은 다음과 같습니다.

정해진 시간 동안 더 많은 깃발을 캡처한 플레이어가 승리합니다.

다만, 본 게임은 단순한 킬 수 중심의 데스매치 구조를 지양하고 있습니다.

- PvP 전투는

  → 상대를 처치하기 위한 목적이 아니라

  → 상대의 목표 달성을 방해하거나 유리한 상황을 만들기 위한 수단입니다.
- 좀비 AI는

  → 단순한 적대 유닛이 아니라

  → 전장의 긴장도를 높이고 판단을 어렵게 만드는 환경적 압박 요소로 작용합니다.
- 플레이어는 매 순간

  → 깃발 캡처를 시도할지

  → 상대 플레이어를 견제할지

  → 위험을 감수할지 혹은 회피할지를 판단해야 합니다.

이로 인해 본 게임의 플레이는 항상

- 리스크 관리
- 위치 선정
- 타이밍 판단 을 요구하도록 설계되었습니다.

네트워크 & 권한 설계

(Server Authority / SSOT)

본 프로젝트는 **Server Authority 100%를 전제로 설계**되었습니다. **모든 게임 판정과 상태 변경은 서버에서만 수행**되며, **클라이언트는 서버가 확정한 결과를 표현하는 역할만 담당**합니다. 이를 통해 네트워크 환경에서도 일관된 게임 상태와 공정한 플레이를 보장하도록 구성하였습니다.

서버와 클라이언트의 역할 분리 본 프로젝트에서는 서버와 클라이언트의 책임을 명확하게 분리하였습니다.

클라이언트의 역할

- **플레이어 입력을 의도(Intent) 형태로 서버에 전달**
- 서버가 승인한 결과를 기반으로
- 애니메이션, 이펙트, 사운드 등 시각적·청각적 표현 수행

서버의 역할

- 모든 전투 및 상호작용에 대한 **판정 수행**
- 게임 상태 변경의 **최종 확정**
- 점수 계산 및 승패 판정
- 매치 결과 및 기록 데이터 저장

이와 같은 구조를 통해 클라이언트 단에서의 임의 조작이나 결과 왜곡 가능성을 최소화하였습니다.

네트워크 & 권한 설계

단일 진실 (Single Source of Truth, SSOT) 본 프로젝트에서 **모든 핵심 게임 상태는 PlayerState를 단일 진실(Single Source of Truth) 로 관리**합니다.
PlayerState가 관리하는 주요 데이터는 다음과 같습니다.

- HP / Armor
- Weapon / Ammo
- Captures
- PvP Kills
- Zombie Kills
- Headshots

반면, Pawn / Character는 다음과 같은 역할만 수행합니다.

- 애니메이션 재생
- 이펙트 표현
- 사운드 출력

즉, **Pawn과 Character는 코스메틱 표현 전용 객체로 한정**되어 있으며, 게임의 실제 상태나 판정 로직을 보유하지 않습니다.
이와 같은 구조를 통해 다음과 같은 효과를 얻을 수 있었습니다. **Late Join 상황에서도** PlayerState 복제를 통해 플레이어 상태가 정확하게 복원되고
서버 권위 기반 구조로 인해 **클라이언트 조작 및 치트 가능성 최소화**. 상태 관리 책임이 명확해져 네트워크 디버깅 및 유지보수 난이도가 감소합니다.
본 프로젝트는 이러한 구조를 단순한 이론이 아닌 **실제 구현과 동작 증거로 검증하는 것을 목표**로 하고 있습니다.

Experience (Lyra 샘플 프로젝트 구조)

로비에서 게임 시작이 확정되면, 서버는 다음 단계에서 사용할 **Experience를 선택**합니다. 제 포트폴리오에서는 Experience를 통해 게임의 단계 흐름을 관리합니다. **Experience는 "이 단계에서 어떤 기능을 켤지" 를 정의한 데이터 묶음**입니다. 로비, 전투, 결과 단계는 필요한 UI와 입력, 능력이 전부 다르기 때문에 if 문 분기가 아니라 **Experience 단위로 분리**했습니다.

Experience는 항상 서버가 결정합니다. 서버가 하나를 확정하면, 해당 Experience ID가 클라이언트로 복제되고, 서버와 클라이언트가
동일한 Experience를 로드합니다. 이후 Experience에 정의된 **GameFeature들이 모두 로딩되고 활성화되면** Experience는 Ready 상태가 됩니다.
저는 이 Ready 이전에는 **스폰을 막아두는 구조를 사용**했습니다.

그래서 UI나 입력이 적용되기 전에 캐릭터가 먼저 스폰되는 문제는 **구조적으로 발생하지 않습니다.** Experience는 게임 규칙을 직접 알지 않으며, 전투 판정이나 점수 계산 같은 규칙은 **모두 서버 시스템이 담당**합니다. 이 구조를 통해 서버가 타이밍과 권한을 책임지고, Experience로 기능을 안전하게 조립합니다.

XP_Lobby (로비 전용 Experience : 방 생성 / 입장 / 채팅 / 캐릭터 선택 / Ready 상태 관리 / 매치 시작 대기)
XP_Combat (실제 플레이가 이루어지는 Experience : 전투 입력 활성화 / 무기 / 전투 / GAS 기능 활성화 / 점수 계산 및 승패 판정 진행)
XP_Result (매치 종료 Experience : 결과 UI 표시 / 점수 및 통계 정리 / 다음 세션 전환 대기)

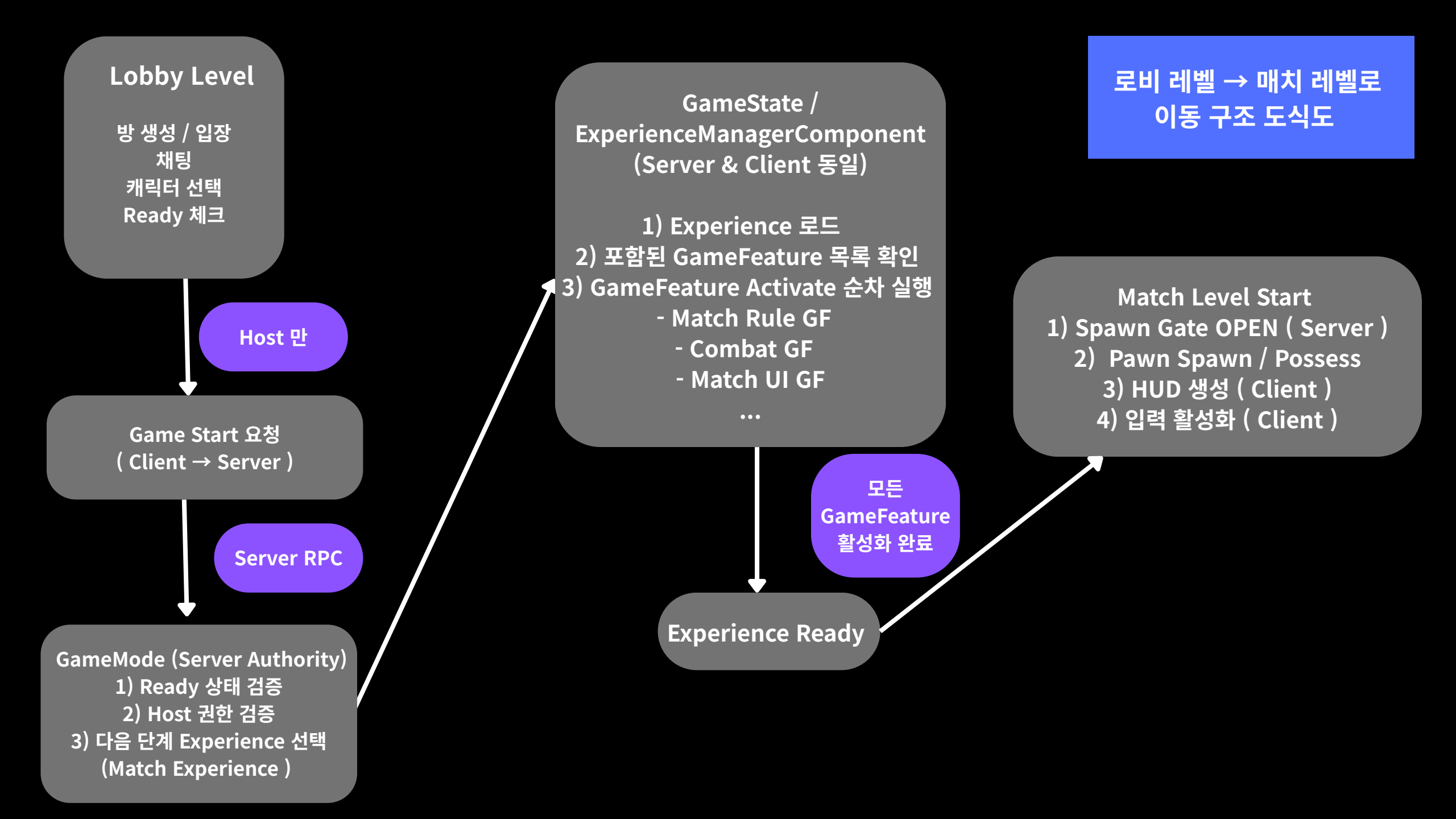
GameFeature(Lyra 샘플 프로젝트 구조)

Experience에서 어떤 기능을 사용할지가 정해지면, 그 기능들은 GameFeature 단위로 활성화됩니다. 먼저 플러그인은 코드와 에셋을 하나로 묶는 단위로, 동적 라이브러리와 유사한 개념입니다. 다만 플러그인은 프로젝트 설정 단계에서 로드 여부가 미리 결정되며, 게임 실행 중에 로드 여부를 변경하는 구조는 아닙니다.

반면 GameFeature는 플러그인 형태이긴 하지만, 이미 로드 가능한 상태의 기능을 런타임에 Activate 또는 Deactivate 할 수 있는 생명주기 구조를 제공합니다.

따라서 핵심 차이는 '로드'가 아니라 '활성화 여부가 런타임에 결정된다'는 점입니다. 이 구조 덕분에 매치 단계에 따라 필요한 기능만 안전하게 켜고 끌 수 있습니다.

GF_Lobby (로비 UI/ 방 생성, 입장 / 채팅 / Ready 상태 관리 / 캐릭터 선택
GF_Match_UI (전투 HUD / 점수판 / 알림 UI / 결과 화면 UI) 등등.


로비 레벨 → 매치 레벨로
이동 구조 도식도
Lobby Level
방 생성 / 입장
채팅
캐릭터 선택
Ready 체크
Host 만
Game Start 요청
(Client → Server)
Server RPC
GameMode (Server Authority)
1) Ready 상태 검증
2) Host 권한 검증
3) 다음 단계 Experience 선택
(Match Experience)
GameState /
ExperienceManagerComponent
(Server & Client 동일)
1) Experience 로드
2) 포함된 GameFeature 목록 확인
3) GameFeature Activate 순차 실행
- Match Rule GF
- Combat GF
- Match UI GF
…
모든
GameFeature
활성화 완료
Experience Ready
Match Level Start
1) Spawn Gate OPEN (Server)
2) Pawn Spawn / Possess
3) HUD 생성 (Client)
4) 입력 활성화 (Client)

전투 시스템 설계 (GAS : Gameplay Ability System)

프로젝트의 전투 시스템은 **서버 권위(Server Authority) 100% 구조로 설계**했습니다. **클라이언트는 "쐈다"라는 입력 의도만 서버에 전달**하고, 실제 발사 가능 여부, 히트 판정, 데미지 계산, 사망 확정은 **오직 서버만 수행**합니다. 멀티플레이에서 가장 위험한 지점은 **클라이언트가 판정하거나 값을 생성하는 순간** 입니다. 그 순간 치팅, 동기화 불일치, 롤백 문제가 발생하기 때문입니다. 그래서 **전투의 핵심 로직을 전부 서버로 올리고**, **클라이언트는 연출과 UI 갱신만 담당하도록 역할을 명확히 분리**했습니다.

1) 전투 흐름 개요

- 클라이언트가 Fire 입력 → **GA_Fire가 활성화 요청 (데미지 계산 x)**
- 서버가 발사 요청을 검증합니다. → **Ammo / State / Cooldown 검증**
- 서버가 라인트레이스로 히트 판정을 수행 → **Trace / Headshot / Falloff 계산**
- **서버가 최종 DamageValue 확정. 즉, "누가 맞았는지 / 얼마의 데미지인지"는 서버가 100% 결정합니다.**

2) ASC 위치 차이 해결 – Resolve 단계

플레이어와 좀비는 **ASC 위치가 다릅니다.**

- 플레이어 → **PlayerState에 ASC (SSOT)**
- 좀비 → **Pawn에 ASC 직접 보유**

이 차이를 전투 코드에 그대로 두면 Player/AI 분기가 곳곳에 퍼지게 됩니다. 그래서 **서버에 Resolve 단계**를 두었습니다. Resolve의 역할은 단 하나입니다: **"대상이 누구든 TargetASC 하나만 반환해라."** 이 과정을 통해 ASC 위치 차이를 전투 로직 전체가 아니라 Resolve 한 지점에서만 처리하도록 흡수했습니다. 이후 전투 파이프라인은 대상이 누구든 동일한 GAS 흐름으로 처리됩니다.

3) GAS를 사용한 이유 - 역할 분리

전투는 GAS 파이프라인으로 표준화했습니다.

- 행위는 **Ability 단위로 통일**
- 값 전달은 **GameplayEffect**
- 규칙 처리는 **AttributeSet 단일 책임**

특히 데미지는 바로 Health를 깎지 않고 Meta Attribute(IncomingDamage) 로 전달한 뒤 PostGameplayEffectExecute에서

- Shield 우선 감소
- Health 감소
- Health ≤ 0 → 서버 Death 확정을 처리합니다. 즉, 규칙은 AttributeSet 한 곳에만 존재합니다.

Client

1) Input(Fire Press/Release)

2) GA_Fire Activate 요청 (Intent Only)

데미지 계산 x
히트 판정 x
사망 판단 x

DEDICATED SERVER (Authority 100% 계산 구간)

[1] Validate Fire
- Ammo 존재?
- Cooldown OK?
- Death / Stun 상태?
- 슬롯 유효성?

[2] Trace & Hit 판정
- Hit Actor 확정
- Bone 기반 Headshot 판정
- 거리 기반 Falloff 계산
- 팀/무적/룰 적용

[3] 최종 DamageValue 확정
→ 서버가 데미지 값 완전 결정

[4] Resolve Target ASC
"이 대상의 체력 계산기는 어디?"

▸ Zombie
Pawn —▸ ASC 직접 보유
▸ Player
Pawn —▸ PlayerState —▸ ASC (SSOT)

✓ 이후부터는 TargetASC 하나만 사용

GAS 파이프라인 (전투 전달 표준화 레이어)

1) Apply GE_AmmoCost → Shooter ASC

2) Apply GE_Damage
(SetByCaller DamageValue)
→ TargetASC

▼

3) Meta Attribute : IncomingDamage

▼

4) AttributeSet::
PostGameplayEffectExecute

▸ Shield 먼저 감소
▸ 남은 값 Health 감소
▸ Health <= 0 → 서버 Death 확정

✓ 규칙은 오직 AttributeSet에서만 처리
✓ 무기/스킬 늘어나도 구조 동일

COSMETIC & REPLICATION

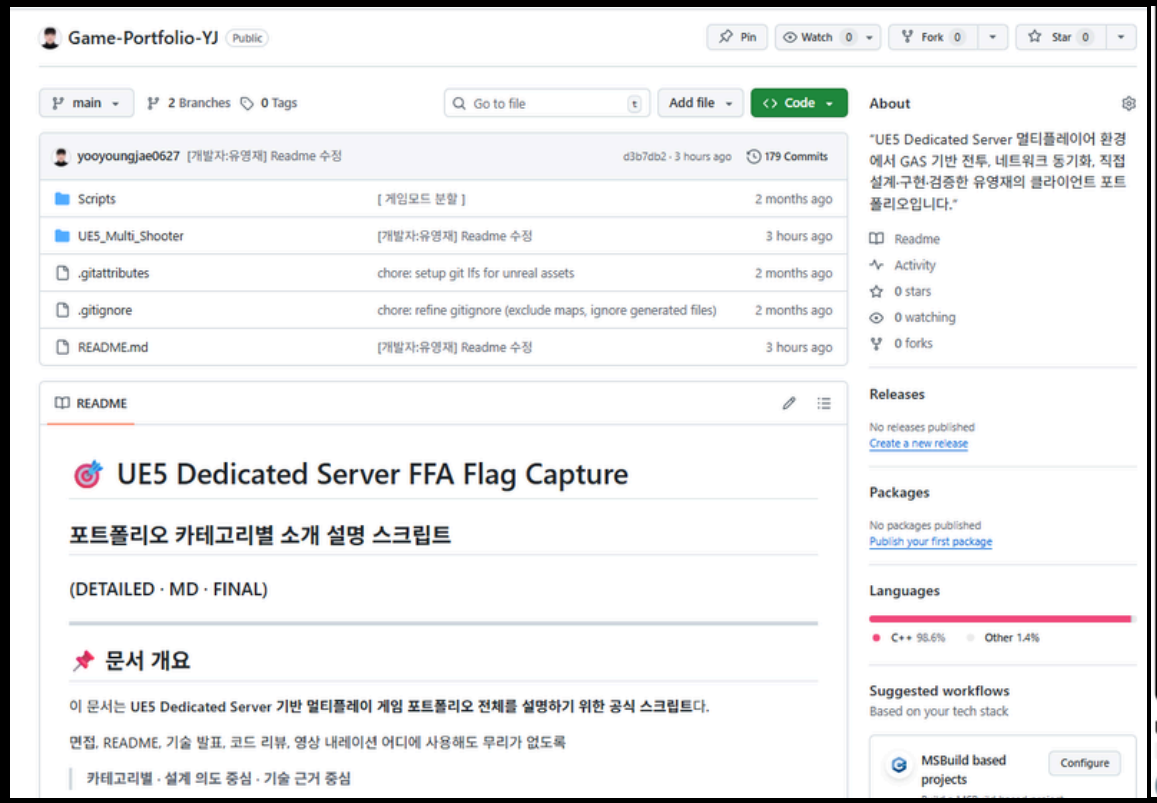
1) GameplayCue (연출 전용)
- MuzzleFlash
- HitImpact
- HitMarker

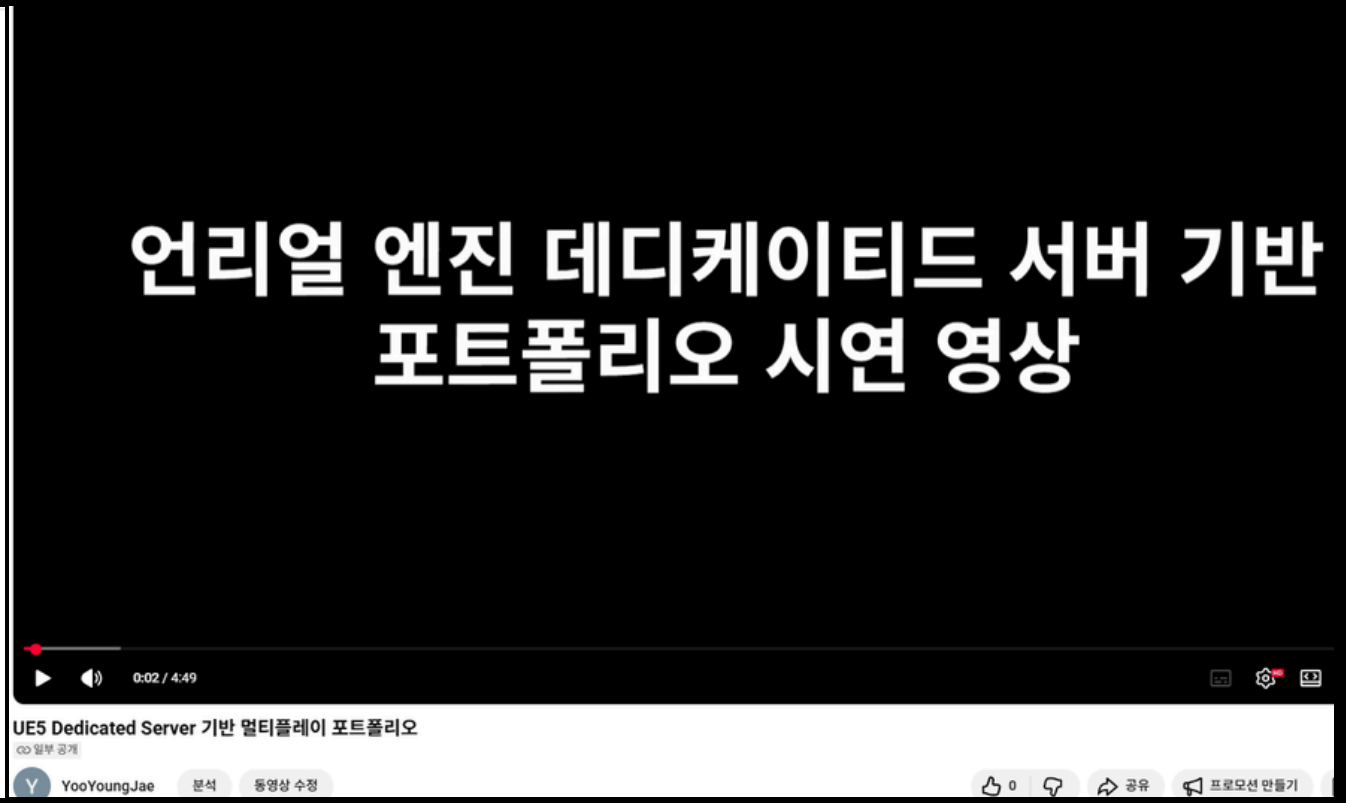
2) Replication (SSOT 기반)
- Health
- Ammo
- Death

3) Client UI 갱신
(No Tick / Delegate 기반)

GAS 데미지 시스템 도식도

1) 클라는 의도만 보낸다
2) 서버가 판정과 값 확정을 전부 담당한다
3) Resolve가 ASC 위치 차이를 흡수한다
4) GAS는 전달 파이프라인 표준
5) AttributeSet은 규칙 단일 처리
6) GameplayCue는 연출 전담
7) UI는 SSOT 복제값만 표시





포트폴리오 GitHub 링크

https://github.com/yooyoungjae0627/Game-Portfolio-YJ

포트폴리오 유튜브 영상 링크

https://www.youtube.com/watch?v=JLWoK7s5ayU