

오목 온라인 개발 문서
표진원
2025년 9월 2일

프로젝트 시작 계기

저는 머신러닝과 인공지능 시스템에 관심을 가지면서, 단순히 모델을 학습하는 수준을 넘어 실제 서비스 환경에서 동작하는 **AI** 시스템을 구축해보고 싶었습니다.

특히 사용자와 상호작용하는 서비스 속에서, 데이터 처리, 서버 아키텍처, 실시간 통신, **AI** 로직 서빙, 성능 검증까지 아우르는 엔드투엔드(**End-to-End**) 흐름을 직접 경험해보고자 했습니다.

이 과정에서 비록 규칙 기반 **AI**였지만, 오목 온라인 게임에 **AI** 봇을 결합하여 실제 유저가 접속해 플레이할 수 있는 시스템을 설계·구현하며, **AI**를 서비스화(**Serving**)하는 전 과정을 경험할 수 있었습니다

여러 게임 장르를 고민하던 중 오목(**Gomoku**)을 선택한 이유는 다음과 같습니다:

- UI 및 그래픽이 단순하여 게임 개발 초기에 복잡한 아트 리소스에 시간을 뺏기지 않고 시스템 설계에 집중 가능
- 턴제 게임으로서 비즈니스 로직을 상대적으로 명확하게 설계할 수 있음
- 누구나 아는 룰을 기반으로 하기 때문에 데모 프로젝트로 적합
- 실시간 **PvP**뿐 아니라 오목 **AI** 봇을 직접 설계·구현해볼 수 있는 기회 제공

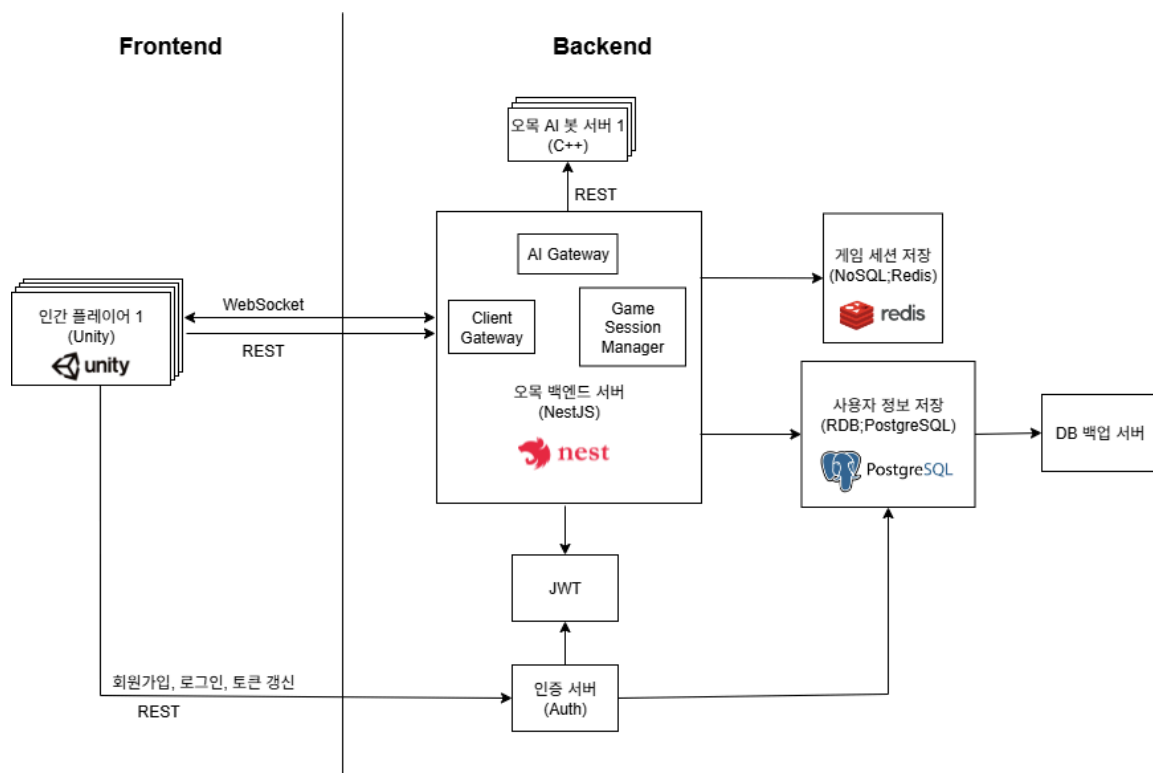
또한, 이번 프로젝트는 단순한 오목 게임을 넘어서 현업에서 사용하는 핵심 백엔드 기술 스택을 학습하기 위해 기획되었습니다.

예를 들어:

- 회원가입, 로그인, **JWT** 인증을 통한 사용자 관리 경험
- **PostgreSQL**을 활용한 유저 정보 및 매치 히스토리 관리 및 저장
- **DB** 백업 컨테이너를 두어 주기적으로 **DB** 백업 (최대 3일치 저장)
- **Redis** 기반 실시간 게임 세션 관리 및 상태 동기화
- **Docker Compose**를 통한 멀티 컨테이너 환경 구성 및 마이크로서비스 아키텍처(**MSA**) 실습
- **Unity**를 활용한 게임 클라이언트 개발 경험
- **REST API + WebSocket**을 통한 클라이언트-서버 통신 프로토콜 설계 경험
- **C++** 기반 **AI** 봇 서버 설계 및 **NestJS** 서버와의 비동기 통신 구현

즉, 이 프로젝트는 단순한 게임을 만드는 것을 넘어 게임 클라이언트부터 서버, 데이터베이스, 실시간 통신, AI, 컨테이너 기반 배포까지 전체 시스템을 직접 설계하고 구현하는 경험을 목표로 시작했습니다.

아키텍처 구성도



1. 프론트엔드 (Unity)

- 역할
 - 플레이어 UI 및 게임 보드 렌더링
 - 서버와 실시간 통신 및 이벤트 처리
- 주요 특징
 - **WebSocket:** 실시간 게임 상태 동기화
 - **REST API:** 로그인, 회원가입, 토큰 갱신
 - **JWT 기반 세션 관리**

2. 백엔드 (NestJS 기반)

2.1. 주요 컴포넌트

- **Client Gateway**
 - Unity 클라이언트와 WebSocket 통신 담당
 - 실시간 이벤트 처리 및 응답
- **Game Session Manager**
 - 게임 세션 생성, 상태 관리, 종료 처리
 - Redis를 활용한 보드 상태 및 세션 동기화
- **AI Gateway**
 - AI 봇 서버(C++)와 REST API 통신
 - AI 응답을 받아 게임 세션에 반영

2.2. 데이터 관리

- **PostgreSQL (RDB)**
 - 사용자 정보, 게임 결과, 매치 히스토리 영구 저장
- **Redis (NoSQL)**
 - 실시간 게임 보드 상태 캐싱

2.3. 인증 및 보안

- JWT 기반 인증 및 사용자 세션 관리
- 인증 서버(Auth)에서 토큰 발급 및 검증
- 토큰을 활용해 REST 및 WebSocket 접근 제어

3. AI 봇 서버 (C++)

- 역할
 - 오목 AI 알고리즘 실행 및 최적 수 계산
 - NestJS 서버와 REST API 기반 통신
- 핵심 알고리즘
 - Minimax + Alpha-Beta Pruning (불필요한 후보 탐색 제외)
 - 휴리스틱 기반 후보군 축소 (고득점 가능성이 높은 후보순으로 탐색)
- 고성능 & 비동기 처리 & 확장성
 - C++ 기반으로 고성능
 - Thread Pool 기반 다중 비동기 요청 처리
 - 다수의 AI 요청을 병렬 처리 가능
 - 요청량 증가 시 AI 봇 서버 인스턴스를 수평 확장(horizontal scaling) 가능 → Docker 컨테이너 기반으로 손쉽게 스케일링 지원

4. 서버 간 통신 구조

- Unity ↔ NestJS
 - WebSocket: 실시간 턴 알림 및 보드 상태 전송
 - REST API: 회원가입, 로그인, 토큰 갱신 (Auth 경유)
- NestJS ↔ AI 서버
 - REST API로 비동기 요청 → Thread Pool로 빠른 응답 처리

- **NestJS ↔ Redis**

- 게임 세션 및 보드 상태 캐싱 및 조회

- **NestJS ↔ PostgreSQL**

- 사용자 정보 및 매치 기록 저장

5. 배포 및 운영

- **Docker Compose** 기반 컨테이너 환경

- Unity 빌드, NestJS 서버, AI 서버, PostgreSQL, Redis, 인증 서버를 각각 독립 컨테이너로 실행
- 서비스 간 의존성을 최소화하고, ****마이크로서비스 아키텍처(MSA)****를 적용

- 확장성

- NestJS 서버, AI 봇 서버, Redis, DB를 개별 컨테이너 단위로 수평 확장 가능
- 추후 대규모 동시 접속 상황에서도 유연하게 대응 가능한 방향으로 설계