PORTFOLIO

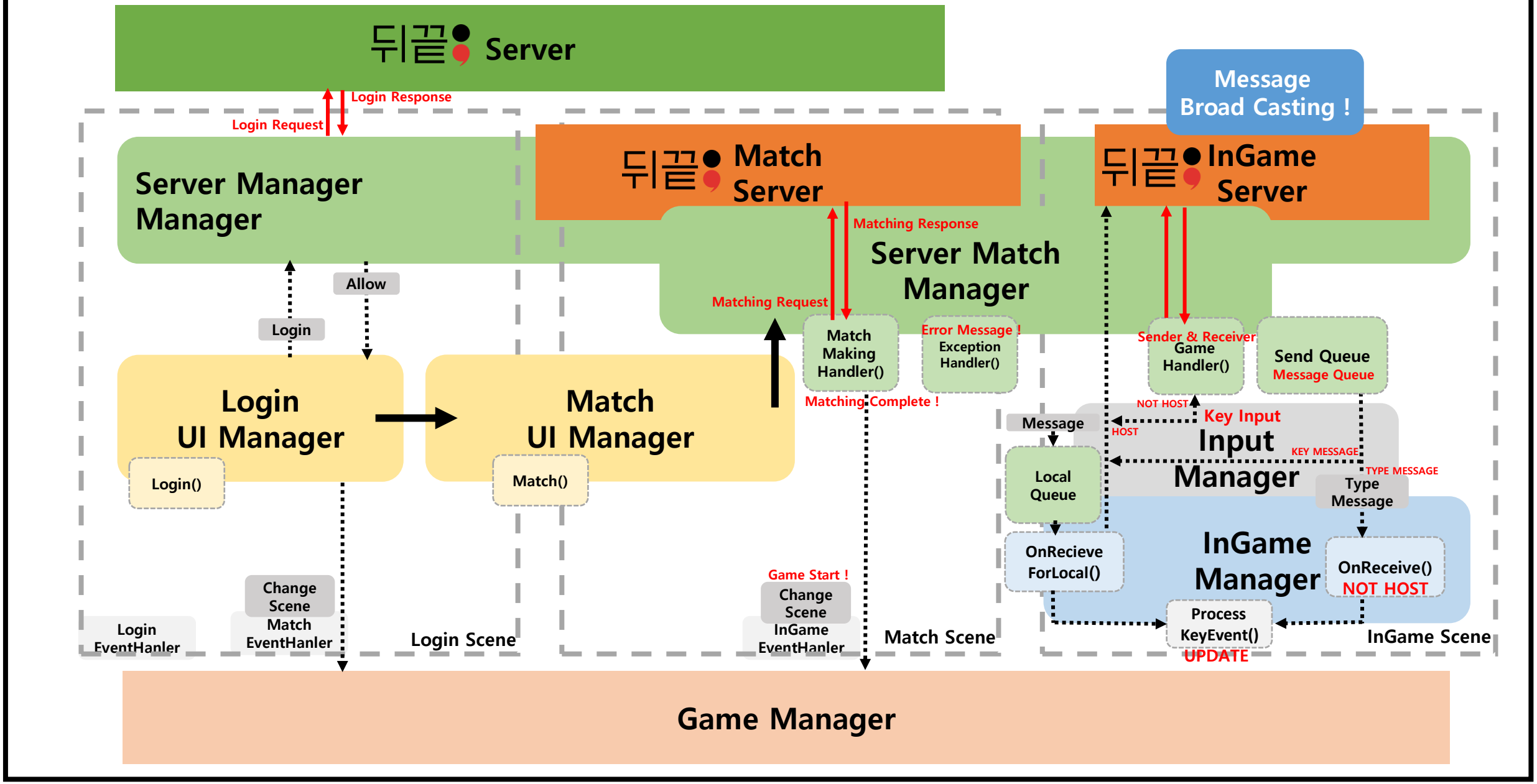
"HELLO WORLD"

KEON GI KIM

PASSIONATE DEVELOPER

CONTENTS

1. Architecture



2. Game Manager

GameManager.cs

Introduce

현재 진행중인 게임의 상태를 저장하고 처리하며, 상태 핸들러를 통해 **Scene을 전환**하는 매니저. 모든 매니저는 인스턴스화 후, 하나의 인스턴스를 가진다. (Signtone Pattern)

실행주기
어플리케이션 시작과 동시에 실행되며, 어플리케이션이 종료되기 전까지 종료되지 않는다.

동작방식
시작과 동시에 각 Scene에서 사용될 **EventHandler**들을 초기화하며, Scene전환 함수가 호출 되면 Scene을 전환하고 해당 Scene에 필요한 GameManager의 State정보를 변경한다.

EventHandler

OnGameReady, InGame, AfterInGame, OnGameOver, OnGameResult, OnGameReconnet 모두 Game Manager가 State가 변경 될 때 이벤트가 추가되며, 이벤트가 발생할 때 호출되는 이벤트 핸들러이다. 이벤트 핸들러는 게임 시작시 호출 되는 **GameStart()**와 게임 재연결시도시 호출 되는 **GameReconnect()**로 모두 초기화 된다.

IEnumerator

Update대신 사용 되는 코루틴으로, InGame Scene과 Loading Scene에서 사용 될 Update Coroutine이다.

GameState

Scene은 **GameManager**에 의해 변경이 일어나는데, GameState는 현재의 **gameState**를 표시하기 위한 **Enum**이다. 현재 gameState를 확인하여 Scene에 알맞은 EventHandler를 사용한다.

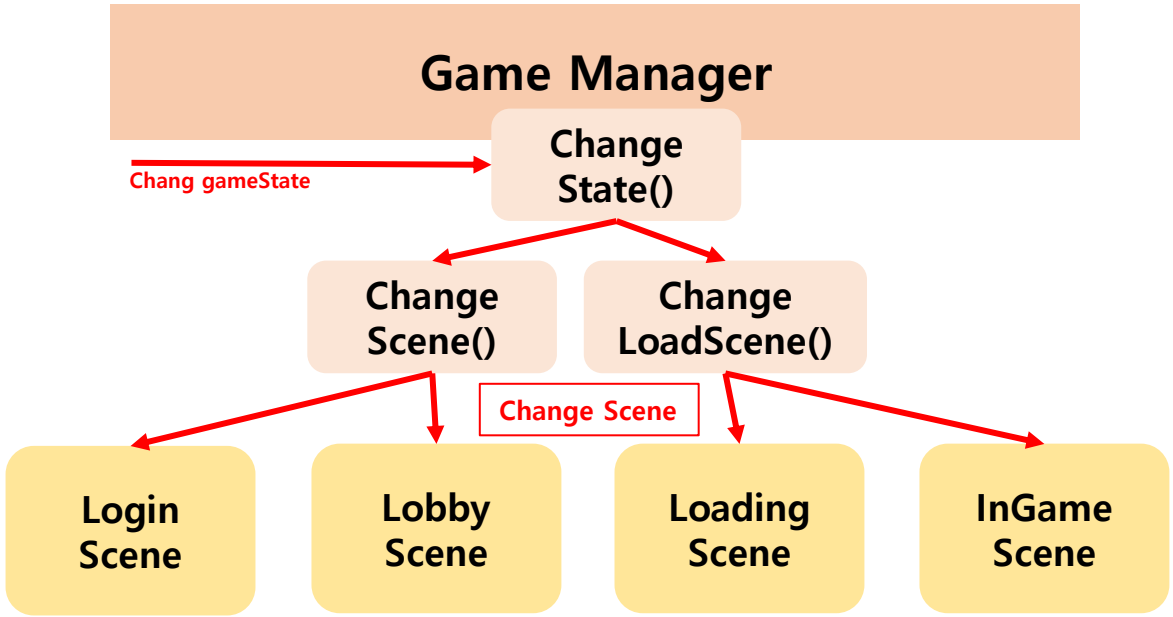
```
public class GameManager : MonoBehaviour
{
    private static GameManager instance = null;
    private static bool isCreate = false;

    #region Scene_name
    private const string LOGIN = "Login";
    private const string LOBBY = "Lobby";
    private const string READY = "Loading";
    private const string INGAME = "InGame";


    private const string PROGRESS_PERCENTAGE = "{0:N1} %";
    #endregion

    #region Actions-Events
    public static event Action OnGameReady = delegate { };
    public static event Action InGame = delegate { }; =
    public static event Action LateInGame = delegate { };
    public static event Action OnGameOver = delegate { };
    public static event Action OnGameResult = delegate { };
    public static event Action OnGameReconnect = delegate { };

    private string asyncSceneName = string.Empty;
    private IEnumerator InGameUpdateCoroutine;
    private IEnumerator LoadUpdateCoroutine;

    [SerializeField]
    private CanvasGroup canvasGroup;
    참조 24개
    public enum GameState { Login, MatchLobby, Ready, Start, Over, Result, InGame, Reconnect };
    private GameState gameState;
    #endregion
    참조 11개
    public static GameManager GetInstance()
    {
        if (instance == null)
        {
            Debug.LogError("GameManager instance does not exist.");
            return null;
        }

        return instance;
    }
```

2. Game Manager

GameManager.cs

ChangeState()

Public 함수로 외부 클래스에서 **Scene의 이동이 필요할 때** 호출된다. 변경을 원하는 state를 통해 현재 gameState를 업데이트하고, 변경 될 gameState에서 일어날 **Event**를 찾아 호출한다. InGame과 Loading Scene의 경우 앞서 말한 것처럼, 각 **Ienumerator**를 **StartCoroutine**를 통해 호출 시켜 **Update()**를 대신한다.




```
public void ChangeState(GameState state, Action<bool> func = null)
{
    gameState = state;
    switch (gameState)
    {
        case GameState.Login:
            break;
        case GameState.MatchLobby:
            MatchLobby(func); // 각 스테이트 함수 이후 canvas 교체
            break;
        case GameState.Ready:
            StartCoroutine(LoadUpdateCoroutine); // 페이크 로딩 코루틴
            GameReady();
            break;
        case GameState.Start:
            GameStart();
            break;
        case GameState.Over:
            GameOver();
            break;

        case GameState.Result:
             GameResult();
             break;

        case GameState.InGame:
            StartCoroutine(InGameUpdateCoroutine);
            break;
        default:
            Debug.Log("Unknown State, Please Confirm current state");
            break;
    }
}
```

3. Server Manager

ServerManager.cs

Introduce

서버와 클라이언트간 로그인 접속 및 회원가입을 위한 서버 **API**로 InputField에 입력 된 ID/PW를 통해 서버에게 **로그인 / 회원가입 요청을** 진행한다.

실행 주기
어플리케이션 시작과 동시에 실행되며, 어플리케이션이 종료되기 전까지 종료 되지 않는다.

동작방식
Login 버튼 클릭시 사용자의 ID와 PW가 서버에 전달되어 로그인 요청이 이루어지고, 로그인이 활성화 되었을 경우 **토큰**을 반환하여 어플리케이션이 종료되더라도 일정 시간 동안 재로그인 없이 **로그인 활성화**가 가능하다. 회원 가입의 경우도 Sign Up 버튼 클릭 시 사용자의 ID와 PW 그리고 닉네임이 서버에 전달되어 회원 가입 요청이 이루어지며 자동으로 로그인 요청이 이루어진다.

CustomSignIn

Backend.Bmember.CoustomSignUp() 함수의 호출로 서버에게 회원가입 요청을 보내며 함수의 리턴 값을 통해 성공 여부를 파악한다. 뒤끝 서버 회원가입에는 닉네임 설정이 자동으로 설정되지 않아 **Backend.Bmember.UpdateNickname()** 호출을 통해 **초기 닉네임 값을 id 값으로** 초기화한다.

성공적으로 회원가입이 되고, 닉네임을 업데이트하면 **OnPrevServerAuthorized()**를 통해 현재 플레이어의 로그인 여부를 업데이트하고 로그인을 자동 호출한다.

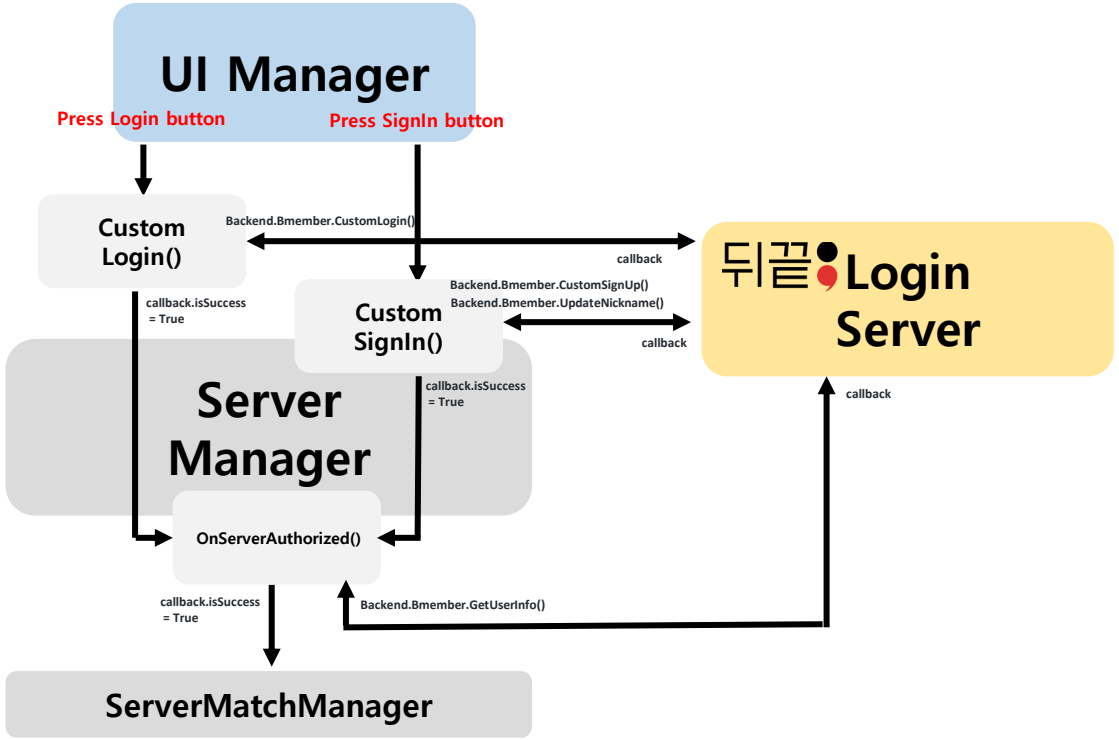
```
public void CustomSignIn(string id, string pw, Action<bool, string> func)
{
    string tempNickName = id; // 기본 닉네임은 ID로 함
    Enqueue(Backend.BMember.CustomSignUp, id, pw, callback =>
    {
        if (callback.IsSuccess())
        {
            Debug.Log("회원 가입 성공 !");
            loginSuccessFunc = func;
            return;
        }

        Debug.LogError(id + pw);
        Debug.LogError("Failed Custom Sign Up₩n" + callback.ToString());
        func(false, string.Format(BackendError,
            callback.GetStatusCode(), callback.GetErrorCode(), callback.GetMessage())); // 에러 토큰 발행
    });

    //닉네임 세팅
    Enqueue(Backend.BMember.UpdateNickname, id, bro =>
    {
        // 닉네임이 없으면 매치서버 접속이 안됨
        if (!bro.IsSuccess())
        {
            Debug.LogError("Failed Create User Nickname₩n" + bro.ToString());
            func(false, string.Format(BackendError,
                bro.GetStatusCode(), bro.GetErrorCode(), bro.GetMessage()));
            return;
        }

        loginSuccessFunc = func;
        OnPrevServerAuthorized(); // Load UserInfo
    });
}
```

3. Server Manager

Server Manager.cs

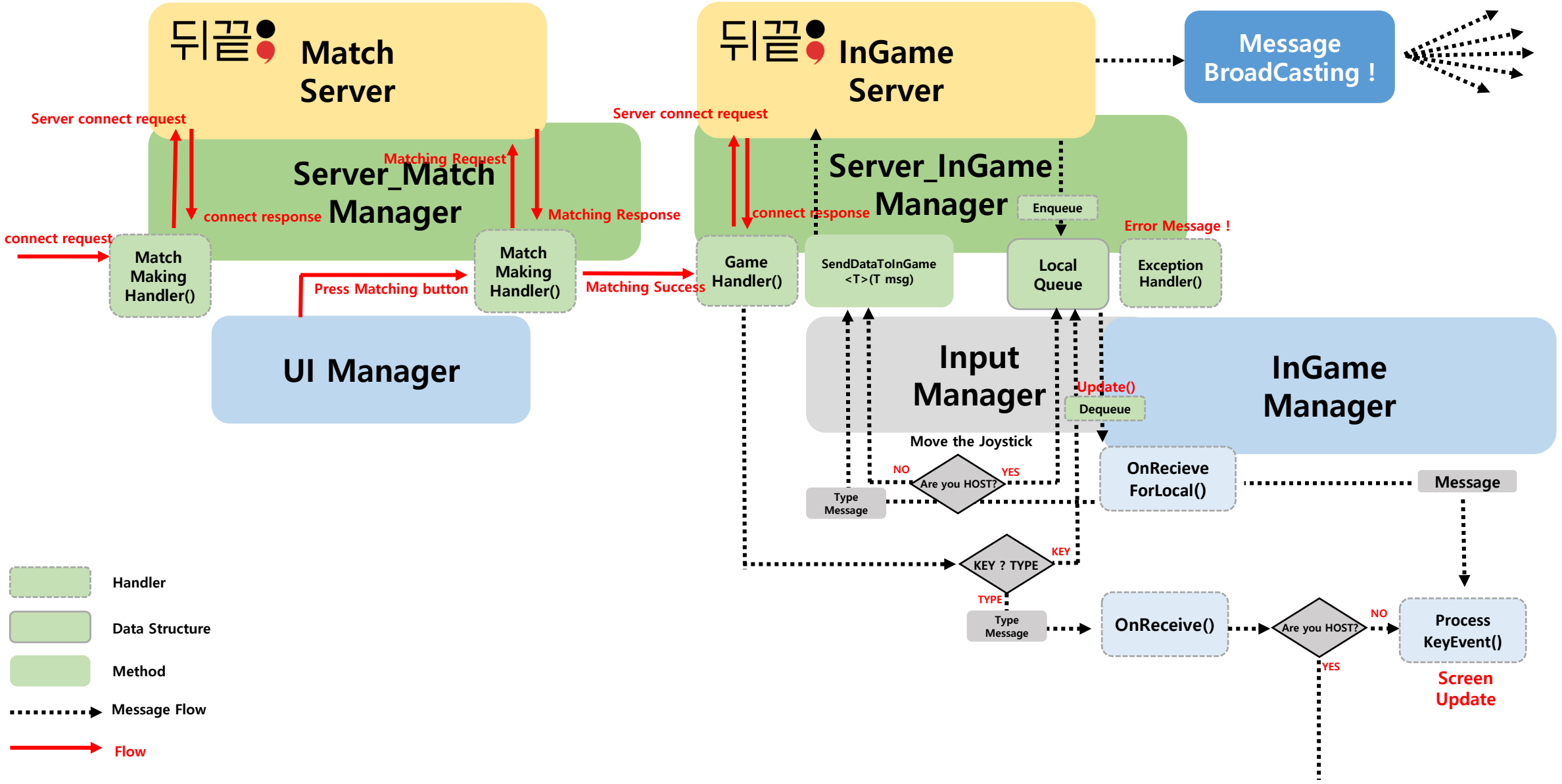
OnServerAuthorized()

OnPrevServerAuthorized() 이후 호출 되는 메소드로, **Backend.Bmember.GetUserInfo()**를 통해 서버에 저장 되어있는 유저의 정보를 호출한다. 만약 해당 유저의 정보 중 닉네임의 정보가 null 값으로 존재한다면 에러 메시지를 표시한다. 접속한 로그인 callback 값이 존재한다면 **ServerManager.GetMatchList()**를 통해 리스트 받아온다.




```
private void OnServerAuthorized() // 뒤끝 서버 회원 인증
{
    Enqueue(Backend.BMember.GetUserInfo, callback =>
    {
        if (!callback.IsSuccess())
        {
            Debug.LogError("Failed to get the user information.₩n" + callback);
            loginSuccessFunc(false, string.Format(BackendError,
                callback.GetStatusCode(), callback.GetErrorCode(), callback.GetMessage()));
            return;
        }

        Debug.Log("UserInfo₩n" + callback);
        var info = callback.GetReturnValuetoJSON()["row"];
        if (info["nickname"] == null) // 닉네임 정보가 null일 경우 닉네임 업데이트 창 표시
        {
            Debug.Log("닉네임 정보를 찾을 수 없습니다.");
            return;
        }

        userNickName = info["nickname"].ToString();
        userIndate = info["inDate"].ToString();
        if (loginSuccessFunc != null)
        {
            Debug.Log("진행 중인 게임을 확인합니다.");
            ServerMatchManager.GetInstance().GetMatchList(loginSuccessFunc);
            return;
        }
    });
}
```

4. ServerMatchManager



4. ServerMatchManager

ServerMatchManager.cs
Partial - ServerMatchManager

Introduce

매칭 서버와 연결과 매칭 요청을 위한 매니저, **MatchMakingHandler**에 의해 매칭 서버와 연결을 진행하고 서버에 접속해 있는 사용자들에게 매칭 요청을 진행한다. 만약, 오류가 발생할 경우 **Exception Handler**에 의해 Error Message 표출을 각 Scene의 UIManager에게 요청한다.

실행주기

ServerManager와 동일하게 어플리케이션 시작과 동시에 실행되며 어플리케이션 종료 전까지 종료되지 않으며 주로 Lobby Scene에서 동작된다.

동작방식

Lobby Scene 에서 부터 본격적으로 실행된다. Matching 버튼 클릭 시 사용자는 매칭 서버와 접속을 요청하게 된다. 접속이 정상 처리 될 경우, 매칭 서버는 만들어진 매칭 룸끼리 매칭을 잡아준다. 매칭이 정상 처리될 경우 매칭 서버와의 연결을 종료하며, Ingame Scene으로 전환되고 유저들이 보내는 메세지를 받아 처리한다. 만약, 게임이 종료되어 종료 요청이 들어오면 게임 결과 창을 UI에 나타내며 동시에 인게임 서버의 접속을 종료 한다.

Update()

ServerMatchManager는 InGame에서 HOST가 처리할 메시지를 담을 **LocalQueue** 를 가지고 있다. 만약 InGameServer 혹은 MatchServer에 접속 되어 있다면 메시지를 송수신하기 위해 **BackEnd.Match.Poll()** 매 프레임마다 호출하고 **SetHostSession()** 을 통해 localQueue가 존재한다면 로컬 큐의 메시지를 확인하여 InGame 화면을 업데이트한다.

```
void Update()
{
    if (isConnectInGameServer || isConnectMatchServer)
    {
        Backend.Match.Poll();

        // 호스트의 경우 로컬 큐가 존재 '
        // 큐에 있는 패킷을 로컬에서 처리
        if (localQueue != null)
        {
            while (localQueue.Count > 0)
            {
                Debug.Log("로컬 큐의 메세지를 확인합니다.");
                var msg = localQueue.Dequeue();
                InGameManager.instance.OnRecieveForLocal(msg);
            }
        }
    }
}
```

4. ServerMatchManager

ServerMatchManager.cs
Partial - ServerMatchManager

MatchMakingHandler()

매칭 서버와 상호작용을 핸들링하는 핸들러로 다음과 같은 경우에 각 이벤트가 발생할 경우 이벤트에 알맞게 핸들러에 추가 되는 작업들을 진행한다.

- **Backend.Match.OnJoinMatchMakingServer**
매칭 서버에 접속하게 되면 호출 되는 이벤트로 ProcessAccessMatchMakingServer() 를 호출하여 현재 접속 상태를 최신화 시킨다.

- **Backend.Match.OnMatchMakingResponse**
매칭 신청 시에 호출 되는 이벤트로 ProcessMatchMakingResponse() 를 호출하여 callback 되는 args를 통해 각 State값에 알맞게 LobbyUIManager를 최신화 시킨다.

- **Backend.Match.OnLeaveMatchMakingServer**
매칭 서버에서 **접속을 종료**할 때 호출 되는 이벤트로 **IsConnectMatchServer** 의 State값을 최신화한다.

- **Backend.Match.OnMatchMakingRoomCreate**
룸 생성 시에 호출 되는 이벤트로 **CreateRoomResult()** 를 호출하여 저장 되어있는 유저의 정보를 토대로 새로운 유저 카드를 생성시키고 **ReqestMatch()** 를 호출하여 매칭을 시작한다.

```
private void MatchMakingHandler() // 대기방 핸들러
{
    Debug.Log("Successfully Connencted MatchMakingHandler on MatchManager");
    // 매칭 서버에 접속하면 호출
    Backend.Match.OnJoinMatchMakingServer += (args) =>...
    // 매치 신청 관련 작업 호출
    Backend.Match.OnMatchMakingResponse += (args) =>...

    // 매칭 서버에서 접속 종료할 때 호출
    Backend.Match.OnLeaveMatchMakingServer += (args) =>...

    // 대기 방 생성/실패 여부
    Backend.Match.OnMatchMakingRoomCreate += (args) =>...
}
```

4. ServerMatchManager

ServerInGameManager.cs
Partial - ServerMatchManager

Introduce

인게임 서버와의 연결과 게임에서의 메시지 송수신을 위한 매니저, 같은 인게임 서버에 접속된 사용자가 송신하는 메시지를 **BackEnd.Match.OnMatchRelay**로 수신받아 클라이언트의 **InGameManager**에서 처리하도록 한다.

Backend.Match.OnMatchRelay()

인게임 서버와 상호작용을 핸들링하는 핸들러로 다음과 같은 경우에 각 이벤트가 발생할 경우 이벤트에 알맞게 핸들러에 추가 되는 작업들을 진행한다

- **Backend.Match.OnSessionJoinInServer**
  인게임 서버에 접속하게 되면 호출 되는 **이벤트로 AcessInGameRoom() 을** 호출하고 ServerMatchManager.**ProcessMatchSucess()** 를 통해 얻은**inGameRoomToken** 을 이용하여 **Backend.Match.JoinGameRoom()** 을 통해 게임 룸에 접속한다.

- **Backend.Match.OnMatchInGameStart**
  게임 룸에 성공적으로 접속 되고 로딩 씬 이전에 호출 되는 이벤트로 **GameSetup()** 를 호출하고 **SendSetUserCharacterIndex()** 를 통해 게임 룸에 접속 되어 있는 유저의 케릭터 정보를 송신한다. 그리고, **OnGameReady()** 를 호출하여 **SetHostSession()**을 통해 **HOST**를 결정하고 **샌드박스** 모드일 경우 **SetAIPlayer()**를 통해 AI 정보를 최신화한다.

- **Backend.Match.OnMatchResult**
  인게임 서버에 매치 결과가 전송 될 때 호출 되는 이벤트로, 정상적으로 기록 되었는지 확인하고, **GameManager.ChangeState()** 를 통해 **gameState** 를 **Result**로 변경한다. 그리고 모든 **세션 리스트**를 초기화한다.

- **Backend.Match.OnLeaveInGameServer**
  인게임 서버에서 접속을 종료할 때 호출 되는 이벤트로 isConnectInGameServer 의 State값을 최신화한다..

```
private void GameHandler()
{
    Debug.Log("Successfully Connencted GameHandler on MatchManager");
    Backend.Match.OnSessionJoinInServer += (args) =>...

    Backend.Match.OnSessionListInServer += (args) =>...

    Backend.Match.OnMatchInGameAccess += (args) =>...

    //여기 체크 포인트 check point
    Backend.Match.OnMatchInGameStart += () =>...

    Backend.Match.OnMatchResult += (args) =>...

    Backend.Match.OnMatchRelay += (args) =>...

    Backend.Match.OnMatchChat += (args) =>...

    Backend.Match.OnLeaveInGameServer += (args) =>...

    Backend.Match.OnSessionOnline += (args) =>...

    Backend.Match.OnSessionOffline += (args) =>...

    Backend.Match.OnChangeSuperGamer += (args) =>...
}
```

4. ServerMatchManager

ServerInGameManager.cs
Partial - ServerMatchManager

GameHandler() (cont.)

- **Backend.Match.OnMatchRelay**
  인게임 서버를 통해 받은 패킷이 존재할 때 **InGameManager.OnReceive()**를 통해 처리하도록 한다. **InGameManager**는 **args**를 알맞은 **Message**로 **Parsing**하여 처리한다.

AddMsgToLocalQueue()

클라이언트가 HOST일 경우 **InputManager** 을 통해 입력 받는 (**화면에 업데이트 해야 하는 정보**) 모든 키(**Key != Type**) 메세지를을 **LocalQueue**에 저장한다.

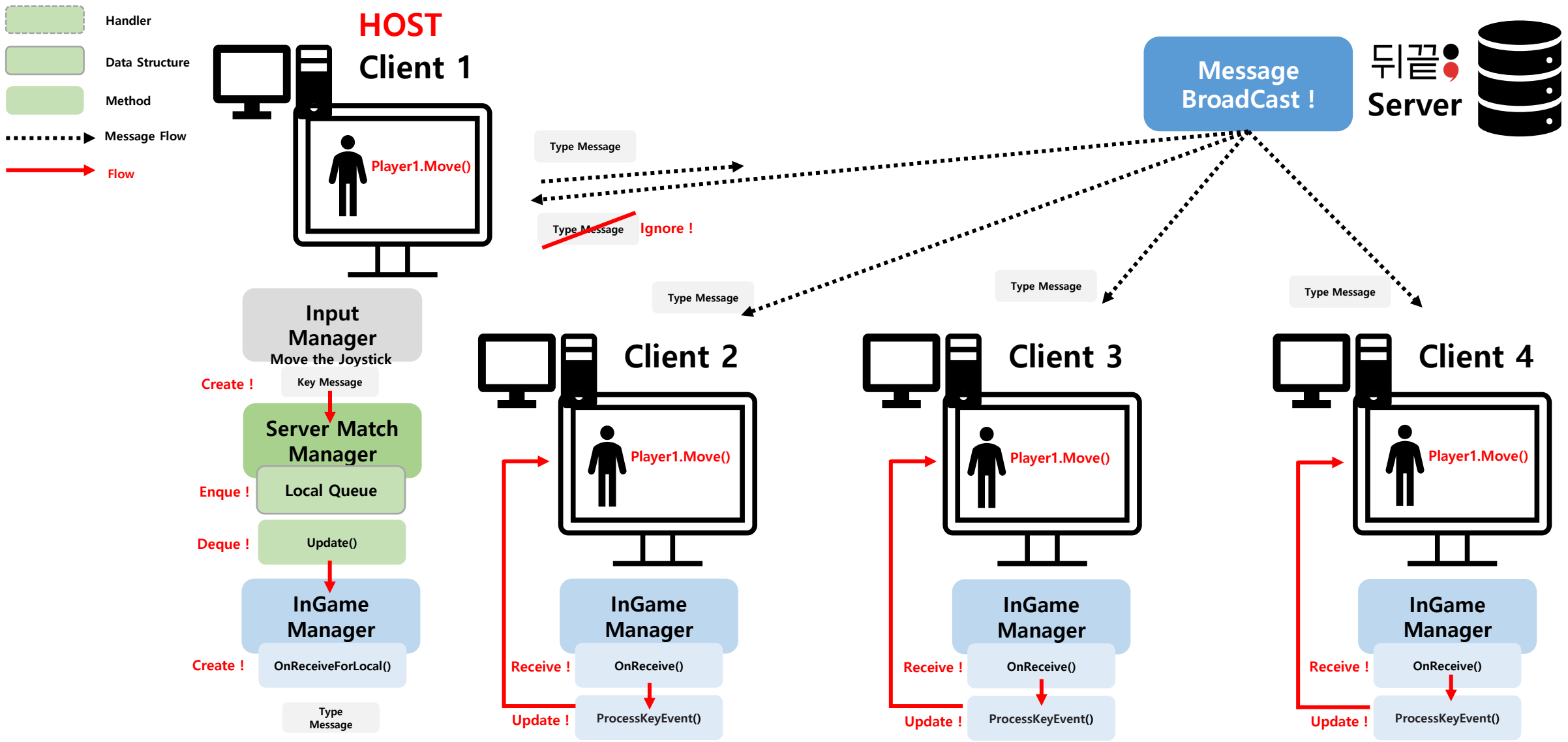
SendDataToInGame()

제네릭 타입의 메시지를 **DataParser.DataToJsonData()**를 통해 **전송 가능한 byte형식의 Message**로 변환한다. 그리고 **BackEnd.Match.SendToInGameRoom()**을 통해 인게임 서버에 전송하고 인게임 서버는 이를 수신하여 **브로드캐스팅**한다.

```
Backend.Match.OnMatchRelay += (args) =>
{
    // 각 클라이언트들이 서버를 통해 주고받은 패킷들
    // 서버는 단순 브로드캐스팅만 지원 (서버에서 어떠한 연산도 수행하지 않음)

    // 게임 사전 설정 ( AI 생성 요청 )
    if (PrevGameMessage(args.BinaryUserData) == true)
    {
        // 게임 사전 설정을 진행하였으면 바로 리턴
        return;
    }

    if (InGameManager.instance == null)
    {
        // 월드 매니저가 존재하지 않으면 바로 리턴
        return;
    }
    // 만약 사전 검사 및 월드매니저가 존재한다면 OnRecieve를 통해 브로드 캐스트 된 메세지를 읽는다.
    InGameManager.instance.OnRecieve(args);
};
```

```
public void AddMsgToLocalQueue(KeyMessage message)
{
    if (isHost == false || localQueue == null)
    {
        return;
    }

    localQueue.Enqueue(message);
}
public void SendDataToInGame<T>(T msg)
{
    var byteArray = DataParser.DataToJsonData<T>(msg);
    Backend.Match.SendDataToInGameRoom(byteArray);
}
```

5. InGameManager



5. InGameManager

InGameManager.cs

Introduce

사용자의 업데이트 정보를 API를 통해 서버로 **송신**하거나, 수신된 메세지를 통해 **인게임 정보**를 수시로 **업데이트**하는 매니저.

실행주기

Loading Scene에서 InGame Scene으로 전환되면서 실행되고, 인게임이 종료됨과 동시에 종료 된다.

동작방식

생성과 동시에 **Initalization()**을 통해 플레이어 정보(케릭터, 위치, 닉네임), **StartCount**와 같은 정보를 초기화하며. **GameManager**의 **InGame.EventHandler**를 최신화한다. 게임이 진행되면, **MatchManager**에 의해 정해진 **HOST**는 타 클라이언트가 불러오는 메세지를 **OnReceive()**를 통해 처리하지 않고 **LocalQue**에 담아놓고 **OnReceiveForLocal()**를 통해서 처리한다. **LocalQueue**에 담겨 있는 **타 사용자**의 **KeyMessage**는 **Key Type**에 따라 **ProcessPlayerData()**를 통해 처리하고 타 클라이언트들에게 재 송신한다. **HOST**가 아닌 클라이언트는 HOST에게 **재가공** 되는 메시지들을 **OnReceive()**를 통해 **송신받아** **ProcessPlayerData()**를 통해 화면을 **최신화**한다.

InitializeGame()

InGameManager가 실행되면 **GameManager.OnGameOver** 이벤트에 인게임 서버에 게임의 종료를 알리는 **OnGameOver()**를 추가하고 **GameManager.OnGameResult** 이벤트에 로비 씬으로 변경을 요청하는 **OnGameResult**를 추가한다.
그리고, **dieEvent** 이벤트에 캐릭터가 사망하였을 때 호출 되는 **PlayerDieEvent** 를 추가한다. 마지막으로, **SetPlayer()**를 통해 인게임의 각 클라이언트의 **Player** 오브젝트를 세팅한다.

```
public void InitializeGame()
{
    //owner_player = 오너 플레이어 객체 (반드시 월드에 인스턴스된 Player프리팹일것)
    GameManager.OnGameOver += OnGameOver; // 게임 종료 이벤트
    GameManager.OnGameResult += OnGameResult; // 게임 종료 결과 이벤트
    dieEvent += PlayerDieEvent;
    SetPlayerInfo();
}

private void PlayerDieEvent(SessionId diePlayer, SessionId killPlayer)
{
    Player killer = players[killPlayer];
    survivedPlayerCount -= 1;
    killer.killCnt += 1;

    if (killer.isOwner)
    {
        killer.OnPlayerKillCountUpdate(killer.killCnt); // 죽인사람 killCount증가업데이트
    }
    if (players[userPlayerIndex].isOwner)
    {
        players[userPlayerIndex].OnAlivePlayerCountUpdate(survivedPlayerCount); // 게임 주인 HUD 업데이트
    }

    if(ServerMatchManager.GetInstance().IsHost() == false)
    {
        return;
    }

    gameRecord.Push(diePlayer);
    Debug.Log(string.Format("살아있는 유저 수 : {0} / 저장된 레코드 수 : {1}",survivedPlayerCount, gameRecord.Count));
    if (survivedPlayerCount <= 1)
    {
        Debug.Log("WHAT THE FUCK");
        SendGameEndOrder();
    }
}
```

5. InGameManager

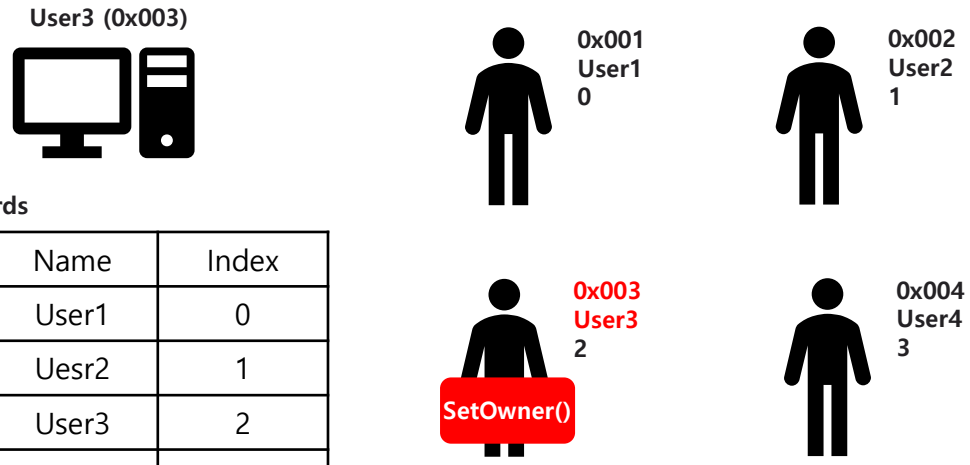
InGameManager.cs

SetPlayerInfo()

클라이언트들의 Player 오브젝트와 **샌드박스 모드**일 경우 AI의 Player 오브젝트를 생성하는 함수이다. 각 클라이언트의 유저 정보와 케릭터 정보가 포함되어 있는 **gameUserRecords**를 통해 생성한다.

SessionId로 정렬된 세션리스트(gameUserRecords)를 Foreach문을 통해 각 클라이언트의 정보를 파악하고 세션리스트에 저장 되어 있는 **케릭터 인덱스** 값을 추출하여 playerPrefabs 리스트에 알맞은 인덱스 값에 해당하는 Prefab을 찾아 **Instantiate()**를 통해 Player 오브젝트를 생성한다.

자신과 동일한 세션 정보를 가지는 즉, 자신의 케릭터를 생성할 때는 **InGameUIManager.Init()**을 통해 화면의 **HUD**를 설정하고, **Player.SetOwner()**를 통해 인게임에서 플레이할 케릭터를 설정한다.



gameUserRecords

SessionId	Name	Index
0x001	User1	0
0x002	Uesr2	1
0x003	User3	2
0x004	user4	3

```
public void SetPlayerInfo()
{
    if (ServerMatchManager.GetInstance().sessionIdList == null)
    {
        // 현재 세션ID리스트가 존재하지 않으면, 0.5초 후 다시 실행
        Invoke("SetPlayerInfo", 0.5f);
        return;
    }

    var gamers = ServerMatchManager.GetInstance().gameUserRecords;

    int size = gamers.Count;
    if (size <= 0)
    {
        Debug.Log("No Player Exist!");
        return;
    }

    if (size > MAXPLAYER)
    {
        Debug.Log("Player Pool Exceed!");
        return;
    }

    int index = 0;
    players = new Dictionary<SessionId, Player>();
    //playersTest = new List<Player>();
    foreach (var record in gamers.OrderByDescending(x => x.Key))
    {
        survivedPlayerCount += 1;
        GameObject player = Instantiate(playerPrefabs[record.Value.m_characterIdx],
                    new Vector3(statringPoints[index].transform.position.x, statringPoints[index].transform.position.y, statringPoints[index].transform.position.z),
                    Quaternion.identity, playerPool.transform);

        Debug.Log("USER SESSION ID INFO : " + record.Value.m_sessionId);
        players.Add(record.Value.m_sessionId, player.GetComponent<Player>());
        player.GetComponent<Player>().mySessionId = record.Value.m_sessionId;
        //playersTest.Add(player.GetComponent<Player>());
        if (ServerMatchManager.GetInstance().IsMySessionId(record.Value.m_sessionId)) // 유저 세팅
        {
            owner_player = player;
            InGameUIManager.GetInstance().Init();
            InGameUIManager.GetInstance().setCharacterStatusBar(player);
            players[record.Value.m_sessionId].SetAsOwner();
            userPlayerIndex = record.Value.m_sessionId;
        }
        else // 타 유저 세팅
```

5. InGameManager

InGameManager.cs

OnReceive()

브로드캐스팅 된 메시지를 **수신**하였을 **때 ServerInGameManager.GameHandler**에 의해 **Backend.Match.OnMatchRelay**를 통해 호출 된다. **args**를 **DataParser.ReadJsonData()** 를 통해 Message로 변환 후 각 Message 의 **각 Type** 에 알맞게 처리한다. 만약, 자신이 **HOST**라면 해당 **수신된 Message**는 무시한다.

- **Protocol.Type.StartCount**
HOST 클라이언트가 인게임의 로딩을 끝내고 보내온 메시지로 해당 메시지를 수신 받으면 InGameManager.SetStartCount()를 설정한다.

- **Protocol.Type.GameStart**
HOST 클라이언트가 SetStartCount()를 종료하고 보내온 메시지로 해당 메시지를 수신 받으면 GameManager.gameState 가 InGame 으로 변경되고 InputManager의 입력을 허용한다.

- **Protocol.Type.GameEnd**
HOST 클라이언트가 게임이 종료되었을 때 보내온 메시지로 해당 메시지를 수신 받으면 SetGameRecod()를 호출하여 Record UI를 출력하고 GameManager.gameState를 Over 로 변경한다.

- **Protocol.Type.Key**
모든 클라이언트의 InputManager 에 의해 수신되는 메시지로 서버의 **HOST**만 처리하여 **TypeMessage**로 재가공 후 **브로드캐스팅**한다.

```
#region handler
참조 1개
public void OnRecieve(MatchRelayEventArgs args)
{

    if (args.BinaryUserData == null)...
    Message msg = DataParser.ReadJsonData<Message>(args.BinaryUserData);
    if (msg == null)...

    // Host가 아닐 때 내가 보내는 패킷은 받지 않는다.
    if (!ServerMatchManager.GetInstance().IsHost() && args.From.SessionId == userPlayerIndex)...

    if (players == null)...
    switch (msg.type)
    {
        case Protocol.Type.StartCount:
            StartCountMessage startCount = DataParser.ReadJsonData<StartCountMessage>(args.BinaryUserData);
            Debug.Log("wait second : " + (startCount.time));
            InGameUIManager.GetInstance().SetStartCount(startCount.time);
            break;

        case Protocol.Type.GameStart:
            InGameUIManager.GetInstance().SetStartCount(0, false); // SetStartCount 하자마자 게임 시간
            GameManager.GetInstance().ChangeState(GameManager.GameState.InGame);
            break;

        case Protocol.Type.GameEnd:
            GameEndMessage endMessage = DataParser.ReadJsonData<GameEndMessage>(args.BinaryUserData);
            Debug.Log("게임 시간 소요 혹은 게임 목표 달성, 게임 매치를 종료 합니다." + endMessage.count);

            foreach(var data in endMessage.sessionList)
            {
                Debug.Log(string.Format("Switch: Session id = {0} / Kill = {1}", data, players[(SessionId)data]));
            }
            SetGameRecord(endMessage.count, endMessage.sessionList);
            GameManager.GetInstance().ChangeState(GameManager.GameState.Over);
            break;

        // 모든 유저가 처음 보내는 비설정 메세지
        case Protocol.Type.Key:
            KeyMessage keyMessage = DataParser.ReadJsonData<KeyMessage>(args.BinaryUserData);
            ProcessKeyEvent(args.From.SessionId, keyMessage); // 호스트만 초기화 후 재 브로드캐스팅
            break;
```

5. InGameManager

InGameManager.cs

OnReceive() (Cont.)

- **Protocol.Type.PlayerMove**
HOST 클라이언트에서 업데이트 이후 케릭터의 **이동 Input** 정보를 알리는 메시지로, ProcessPlayerData() 에 의해 처리된다.

- **Protocol.Type.PlayerNoMove**
HOST 클라이언트에서 업데이트 이후 케릭터의 **이동 종료 Input** 정보를 알리는 메시지로, ProcessPlayerData() 에 의해 처리된다.

- **Protocol.Type.PlayerAttack**
HOST 클라이언트에서 업데이트 이후 케릭터의 **공격 Input** 정보를 알리는 메시지로, ProcessPlayerData() 에 의해 처리된다.

- **Protocol.Type.PlayerStopAttack**
HOST 클라이언트에서 업데이트 이후 케릭터의 공격 **종료 Input** 정보를 알리는 메시지로, ProcessPlayerData() 에 의해 처리된다.

- **Protocol.Type.PlayerAcquireItem**
HOST 클라이언트에서 업데이트 이후 케릭터의 **피격 정보**를 알리는 메시지로, ProcessPlayerData() 에 의해 처리된다.

- **Protocol.Type.PlayerAcquireItem**
HOST 클라이언트에서 업데이트 이후 케릭터의 장비 **획득 정보**를 알리는 메시지로, ProcessPlayerData() 에 의해 처리된다.

```
        // 이하 호스트가 보내고 비호스트만이 받을 설정 메세지
        case Protocol.Type.PlayerMove:
            PlayerMoveMessage moveMessage = DataParser.ReadJsonData<PlayerMoveMessage>(args.BinaryUserData);
            ProcessPlayerData(moveMessage);
            break;

        case Protocol.Type.PlayerNoMove:
            PlayerNoMoveMessage noMoveMessage = DataParser.ReadJsonData<PlayerNoMoveMessage>(args.BinaryUserData);
            ProcessPlayerData(noMoveMessage);
            break;

        case Protocol.Type.PlayerAttack: // 공격 GunController
            PlayerAttackMessage attackMessage = DataParser.ReadJsonData<PlayerAttackMessage>(args.BinaryUserData);
            ProcessPlayerData(attackMessage);
            break;

        case Protocol.Type.PlayerStopAttack: // 공격 GunController
            PlayerStopAttackMessage stopAttackMessage = DataParser.ReadJsonData<PlayerStopAttackMessage>(args.BinaryUserData);
            ProcessPlayerData(stopAttackMessage);
            break;

        case Protocol.Type.PlayerReload: // 공격 GunController
            PlayerReloadMessage reloadMessage = DataParser.ReadJsonData<PlayerReloadMessage>(args.BinaryUserData);
            ProcessPlayerData(reloadMessage);
            break;

        case Protocol.Type.PlayerSwitchWeapon: // 무기 변경 Player
            PlayerSwitchMessage switchMessage = DataParser.ReadJsonData<PlayerSwitchMessage>(args.BinaryUserData);
            ProcessPlayerData(switchMessage);
            break;

        case Protocol.Type.PlayerDamaged: // 피격 Player
            PlayerDamegedMessage damegedMessage = DataParser.ReadJsonData<PlayerDamegedMessage>(args.BinaryUserData);
            ProcessPlayerData(damegedMessage);
            break;

        case Protocol.Type.PlayerAcquireItem: // 장비 획득 Player
            PlayerAcquireMessage acquiredMessage = DataParser.ReadJsonData<PlayerAcquireMessage>(args.BinaryUserData);
            ProcessPlayerData(acquiredMessage);
            break;


        default:
            Debug.Log("Unknown protocol type");
            return;
    }
}
```

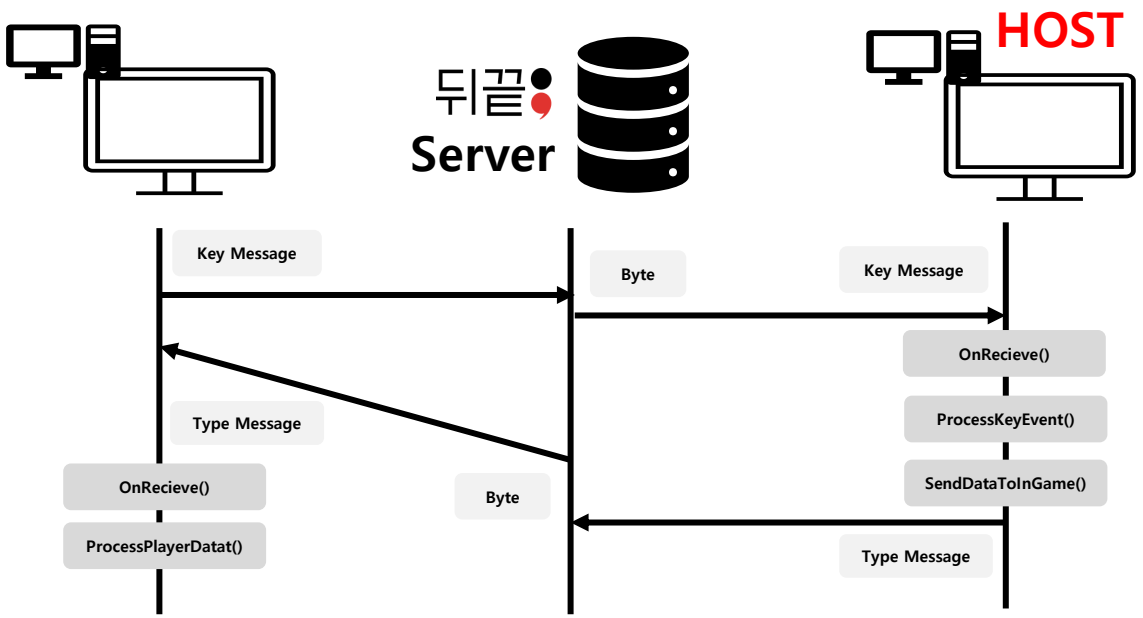
5. InGameManager

InGameManager.cs

ProcessKeyEvent()

KeyMessage가 수신 되었을 때 **OnReceive()** 에 의해 **또는 ServerMatchManager.Update()**에서 실행 되는 **OnReceiveForLocal**에 의해 호출 되며, 클라이언트가 **HOST**일 경우에만 실행 된다.

각, KeyMessage는 송신할 때, 고유의 KeyEventCode를 넣어 송신하는데 ProcessKeyEvent()는 이를 읽어 화면을 업데이트 하며, **HOST가 아닌 클라이언트** 화면 업데이트를 위해 **TypeMessage**로 재가공하여 브로드 캐스팅한다.




```
private void ProcessKeyEvent(SessionId index, KeyMessage keyMessage)
{
    Debug.Log("호스트의 메세지를 읽고 최신화 합니다.");
    if (ServerMatchManager.GetInstance().IsHost() == false)
    {
        return;
    }
    bool isMove = false;
    bool isNoMove = false;
    bool isAttack = false;
    bool isStopAttack = false;
    bool isReload = false;
    bool isSwitch = false;

    int keyData = keyMessage.keyData;

    Vector3 moveVector = Vector3.zero;
    Vector3 targetPos = Vector3.zero;
    Vector3 playerPos = Vector3.zero;

    if (keyData == KeyEventCode.MOVE)
    {
        moveVector = new Vector3(keyMessage.x, keyMessage.y, keyMessage.z);
        moveVector = Vector3.Normalize(moveVector);
        isMove = true;
    }
    else if (keyData == KeyEventCode.NO_MOVE)
    {
        playerPos = new Vector3(keyMessage.x, keyMessage.y, keyMessage.z);
        isNoMove = true;
    }

    else if (keyData == KeyEventCode.ATTACK)
    {
        isAttack = true;
        targetPos = new Vector3(keyMessage.x, keyMessage.y, keyMessage.z);
    }
    else if (keyData == KeyEventCode.STOP_ATTACK)
    {
        isStopAttack = true;
    }
    else if (keyData == KeyEventCode.RELOAD)
    {
        isReload = true;
    }
    else if (keyData == KeyEventCode.SWITCH)
    {
        isSwitch = true;
    }
```

5. InGameManager

InGameManager.cs

ProcessKeyEvent() (Cont.)

- **isMove**
  HOST 클라이언트는 해당 메시지를 받고 즉시, **Player.controller.SetMoveVector()**를 통해 화면을 업데이트 하고, **ServerMatchManager.SendDataToInGame()**을 통해 **PlayerMoveMessage**를 생성하고 송신한다.

- **isNoMove**
  HOST 클라이언트는 해당 메시지를 받고 즉시, **Player.controller.SetMoveVector(Vector3.zero)**를 통해 케릭터를 정지시키고 해당 **Player.SetPosition()**을 통해 현재 위치를 동기화하여 화면을 업데이트 하고, **ServerMatchManager.SendDataToInGame()**을 통해 **PlayerNoMoveMessage**를 생성하고 송신한다.

- **isAttack**
  HOST 클라이언트는 해당 메시지를 받고 즉시, **Player.guntroller.FireAction()** 을 통해 케릭터의 공격을 실행시켜 화면을 업데이트 하고, **ServerMatchManager.SendDataToInGame()**을 통해 **PlayerAttackMessage**를 생성하고 송신한다.

- **isStopAttack**
  HOST 클라이언트는 해당 메시지를 받고 즉시, **Player.guntroller.FireAction()** 을 통해 케릭터의 공격을 정지시켜 화면을 업데이트 하고, **ServerMatchManager.SendDataToInGame()**을 통해 **PlayerStopAttackMessage**를 생성하고 송신한다.

- **isReload**
  HOST 클라이언트는 해당 메시지를 받고 즉시, **Player.guntroller.tryReload()** 을 통해 케릭터의 장전을 실행시켜 화면을 업데이트하고, **ServerMatchManager.SendDataToInGame()**을 통해 **PlayerReloadMessage**를 생성하고 송신한다.

```
if (isMove) // 호스트 : 타 유저 플레이어의 움직임 처리
{
    Debug.Log("플레이어는 움직여라!" + index);
    //Debug.Log("player info : " + players[index]+moveVector);
    players[index].controller.SetMoveVector(moveVector);
    PlayerMoveMessage msg = new PlayerMoveMessage(index, playerPos, moveVector);
    ServerMatchManager.GetInstance().SendDataToInGame<PlayerMoveMessage>(msg);

}

else if (isNoMove) // 호스트 : 타 유저 플레이어의 멈춤
{
    players[index].SetPosition(playerPos);
    players[index].controller.SetMoveVector(Vector3.zero); // Vector zero 값만 들어옴 -> zero로 만드는 건 클라에서 하고 Position 가져와서 동기화 처리
    PlayerNoMoveMessage msg = new PlayerNoMoveMessage(index, playerPos);
    ServerMatchManager.GetInstance().SendDataToInGame<PlayerNoMoveMessage>(msg);
}

else if (isAttack)
{
    Debug.Log("플레이어는 공격해버려!");
    players[index].guntroller.FireAction(targetPos);
    PlayerAttackMessage msg = new PlayerAttackMessage(index, targetPos);
    ServerMatchManager.GetInstance().SendDataToInGame<PlayerAttackMessage>(msg);
}

else if (isStopAttack)
{
    Debug.Log("플레이어는 공격을 멈춰라!" + index);
    players[index].guntroller.FireStopAction();
    PlayerStopAttackMessage msg = new PlayerStopAttackMessage(index);
    ServerMatchManager.GetInstance().SendDataToInGame<PlayerStopAttackMessage>(msg);
}

else if (isReload)
{
    Debug.Log("플레이어는 장전 해라!");
    players[index].guntroller.tryReload();
    PlayerReloadMessage msg = new PlayerReloadMessage(index);
    ServerMatchManager.GetInstance().SendDataToInGame<PlayerReloadMessage>(msg);
}

else if (isSwitch)
{
    players[index].guntroller.ChangeGun();
    Debug.Log("플레이어는 무기를 변경해라!!! "+index);
    PlayerSwitchMessage msg = new PlayerSwitchMessage(index);
    ServerMatchManager.GetInstance().SendDataToInGame<PlayerSwitchMessage>(msg);
}
```

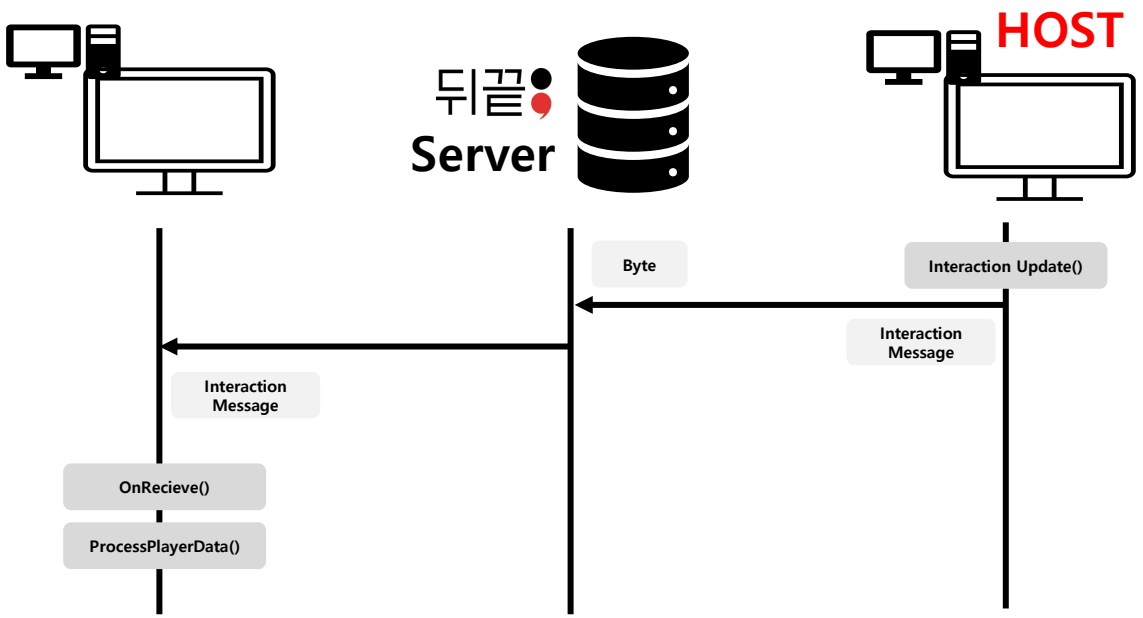
5. InGameManager

InGameManager.cs

ProcessPlayerData()

ProcessKeyEvent()와 **동일**하게 **화면을 업데이트** 하기 위한 메소드로서, **재가공** 된 메세지들을 **HOST가 아닌 클라이언트**에서만 업데이트하며 **KeyEvent**가 아닌 **피격, 습득** 처럼 **상호작용**에 의해 생성 된 메세지들을 수신하였을 때 **모든 클라이언트**가 업데이트를 진행한다.

오버로딩 된 각 메소드 중 KeyEvent와 동일한 메소드의 내부 동작은 **ProcessKeyEvent()**와 동일하다.




```
참조 1개
private void ProcessPlayerData(PlayerMoveMessage msg)...
참조 1개
private void ProcessPlayerData(PlayerNoMoveMessage msg)...
참조 1개
private void ProcessPlayerData(PlayerAttackMessage msg)...
참조 1개
private void ProcessPlayerData(PlayerStopAttackMessage msg)...

참조 1개
private void ProcessPlayerData(PlayerReloadMessage msg)...
참조 1개
private void ProcessPlayerData(PlayerSwitchMessage msg)...
참조 1개
private void ProcessPlayerData(PlayerDamegedMessage msg)...

참조 1개
private void ProcessPlayerData(PlayerAcquireMessage msg)
{
    ItemCategory Item = msg.Item;
    int grade = msg.grade;
    int effectAmount = msg.effectAmount;
    Vector3 playerPos = new Vector3(msg.pos_x, msg.pos_y, msg.pos_z);
    switch (Item)
    {
        case ItemCategory.HealPack:
            players[msg.playerSession].GetItem(ItemCategory.HealPack, effectAmount);
            Debug.Log("healpack");
            break;

        case ItemCategory.ShieldPack:
            players[msg.playerSession].GetItem(ItemCategory.ShieldPack, effectAmount);
            Debug.Log("ShieldPack");
            break;

        case ItemCategory.MainAmmo:
            players[msg.playerSession].GetItem(ItemCategory.MainAmmo, effectAmount);
            Debug.Log("MainAmmo");
            break;

        case ItemCategory.SubAmmo:
            players[msg.playerSession].GetItem(ItemCategory.SubAmmo, effectAmount);
            Debug.Log("SubAmmo");
            break;

        case ItemCategory.Pistol:
            players[msg.playerSession].GetWeapon(ItemCategory.Pistol, grade, effectAmount);
            Debug.Log("Pistol");
            break;

        case ItemCategory.Rifle:
            players[msg.playerSession].GetWeapon(ItemCategory.Rifle, grade, effectAmount);
            Debug.Log("Rifle");
            break;
    }
    players[msg.playerSession].SetPosition(playerPos);
}
```

REFERENCE

@Reference BACKEND SDK Tutorial https://developer.thebackend.io/unity3d/realtime/matchMake/tutorial/

※ 해당 프로젝트는 뒤끝 튜토리얼을 참고하였으며 수정을 거쳐 완성 된 프로젝트임을 명시합니다.