CPSC 491

CSU – Fullerton

VRecover

Daniel Currey

Dan Solis

Kaixiang Liu

Jonathan Mai

Jason Chiu

Goal

- Enhance the effectiveness and accessibility of exposure therapy for patients and therapists.
- Provide a cost-effective platform with home-based therapy options.
- Highly customizable scenarios for patients.
- Prioritize security and privacy of patient data, following principles from Zala et al.'s study (2022) on e-healthcare privacy as well as HIPAA compliance.

Approach

- Create a VR-based therapy platform using Unity VR, AWS services, and FishNet networking to provide secure, scalable, and synchronized therapy sessions.

Technology

Type	Software
Programming Language	C# (for backend development), TypeScript (for frontend development)
IDE	Visual Studio (for backend with C# .net core), Unity (for VR environment development)
Operating System	Windows 10/11 (32 or 64 bit) (compatible with Visual Studio and Unity)
Web Technologies	TypeScript, HTML5, CSS3 (with Tailwind CSS for styling)
Software Framework	Unity (for VR content creation and management), AWS (for cloud service)

Bootstrap

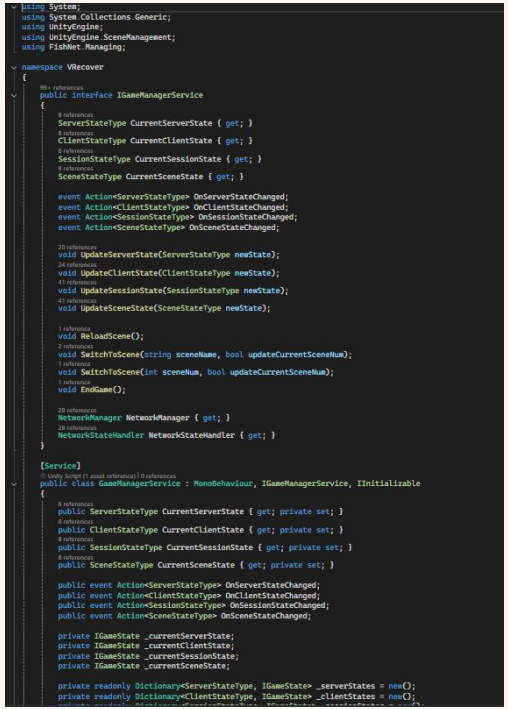
```
using System;
using UnityEngine;

namespace VRecover
{
    8 references
    public interface IInitializable
    {
        8 references
        void Initialize();
    }

    // Unity Script | 0 references
    public class Bootstrap : MonoBehaviour
    {
        public static bool isInitialized = false;
        private static GameObject servicesContainer;

        [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
        0 references
        static void OnBeforeSceneLoad()
        {
            Debug.Log("[Bootstrap] Starting initialization...");
            if (isInitialized)
            {
                Debug.LogWarning("[Bootstrap] Bootstrap is already initialized. Skipping additional initialization.");
                return;
            }

            Debug.Log("[Bootstrap] Cleaning up existing services...");
            CleanupExistingServices();

            Debug.Log("[Bootstrap] Registering services from prefabs...");
            if (!RegisterServicesFromPrefabs())
            {
                Debug.LogWarning("[Bootstrap] Some services could not be registered properly. Check the logs for details.");
            }

            isInitialized = true;
            Debug.Log("[Bootstrap] Bootstrap initialization completed successfully.");
        }

        2 references
        private static void CleanupExistingServices()
        {
            Debug.Log("[Bootstrap] Starting cleanup of existing services...");
            // Clean up any existing services container
            if (servicesContainer != null)
            {
                Debug.Log("[Bootstrap] Destroying existing services container...");
                GameObject.DestroyImmediate(servicesContainer);
            }

            var existingContainer = GameObject.Find("Services Container");
            if (existingContainer != null)
            {
                Debug.Log("[Bootstrap] Found and destroying legacy services container...");
                GameObject.DestroyImmediate(existingContainer);
            }

            // Reset ServiceLocator
            Debug.Log("[Bootstrap] Resetting ServiceLocator...");
            ServiceLocator.Reset();
            isInitialized = false;
            Debug.Log("[Bootstrap] Cleanup completed successfully.");
        }

        1 reference
        private static bool RegisterServicesFromPrefabs()
        {
            Debug.Log("[Bootstrap] Starting service registration from prefabs...");
            bool allServicesRegistered = true;

            try
            {
                if (!System.IO.Directory.Exists("Assets/Resources/Services"))
                {
                    Debug.LogError("[Bootstrap] Services directory not found at Assets/Resources/Services");
                    return false;
                }

                Debug.Log("[Bootstrap] Loading service prefabs from Resources...");
                GameObject[] servicePrefabs = Resources.LoadAll<GameObject>("Services");

                if (servicePrefabs == null || servicePrefabs.Length == 0)
                {
                    Debug.LogError("[Bootstrap] No service prefabs found in Resources/Services. Ensure prefabs are properly placed.");
                    return false;
                }

                Debug.Log($"[Bootstrap] Found {servicePrefabs.Length} service prefabs to process.");

                // Sort prefabs alphabetically by name, including the letter prefix
                Array.Sort(servicePrefabs, (a, b) => string.Compare(a.name, b.name, StringComparison.OrdinalIgnoreCase));

                servicesContainer = new GameObject("Services Container");
                DontDestroyOnLoad(servicesContainer);
                Debug.Log("[Bootstrap] Created new Services Container.");

                foreach (GameObject prefab in servicePrefabs)
                {
                    if (prefab == null)
                    {
                        Debug.LogError("[Bootstrap] Null prefab found in Services folder");
                        continue;
                    }

                    try
                    {
                        Debug.Log($"[Bootstrap] Processing service prefab: {prefab.name}");
                        GameObject serviceObject = Instantiate(prefab, servicesContainer.transform);
                        MonoBehaviour[] services = serviceObject.GetComponents<MonoBehaviour>();

                        if (services == null || services.Length == 0)
                        {
                            Debug.LogError($"[Bootstrap] Service prefab {prefab.name} does not have any MonoBehaviour components.");
                            allServicesRegistered = false;
                            continue;
                        }

                        bool serviceRegistered = false;

                        foreach (MonoBehaviour service in services)
                        {
                            if (service is IInitializable initializable)
                            {
                                try
                                {
                                    Debug.Log($"[Bootstrap] Initializing service: {service.GetType().Name}");
                                    initializable.Initialize();
                                }
                                catch (Exception ex)
                                {
```

```
1 reference
private static bool RegisterServicesFromPrefabs()
{
    Debug.Log("[Bootstrap] Starting service registration from prefabs...");
    bool allServicesRegistered = true;

    try
    {
        if (!System.IO.Directory.Exists("Assets/Resources/Services"))
        {
            Debug.LogError("[Bootstrap] Services directory not found at Assets/Resources/Services");
            return false;
        }

        Debug.Log("[Bootstrap] Loading service prefabs from Resources...");
        GameObject[] servicePrefabs = Resources.LoadAll<GameObject>("Services");

        if (servicePrefabs == null || servicePrefabs.Length == 0)
        {
            Debug.LogError("[Bootstrap] No service prefabs found in Resources/Services. Ensure prefabs are properly placed.");
            return false;
        }

        Debug.Log($"[Bootstrap] Found {servicePrefabs.Length} service prefabs to process.");

        // Sort prefabs alphabetically by name, including the letter prefix
        Array.Sort(servicePrefabs, (a, b) => string.Compare(a.name, b.name, StringComparison.OrdinalIgnoreCase));

        servicesContainer = new GameObject("Services Container");
        DontDestroyOnLoad(servicesContainer);
        Debug.Log("[Bootstrap] Created new Services Container.");

        foreach (GameObject prefab in servicePrefabs)
        {
            if (prefab == null)
            {
                Debug.LogError("[Bootstrap] Null prefab found in Services folder");
                continue;
            }

            try
            {
                Debug.Log($"[Bootstrap] Processing service prefab: {prefab.name}");
                GameObject serviceObject = Instantiate(prefab, servicesContainer.transform);
                MonoBehaviour[] services = serviceObject.GetComponents<MonoBehaviour>();

                if (services == null || services.Length == 0)
                {
                    Debug.LogError($"[Bootstrap] Service prefab {prefab.name} does not have any MonoBehaviour components.");
                    allServicesRegistered = false;
                    continue;
                }

                bool serviceRegistered = false;

                foreach (MonoBehaviour service in services)
                {
                    if (service is IInitializable initializable)
                    {
                        try
                        {
                            Debug.Log($"[Bootstrap] Initializing service: {service.GetType().Name}");
                            initializable.Initialize();
                        }
                        catch (Exception ex)
                        {
```

Service Locator

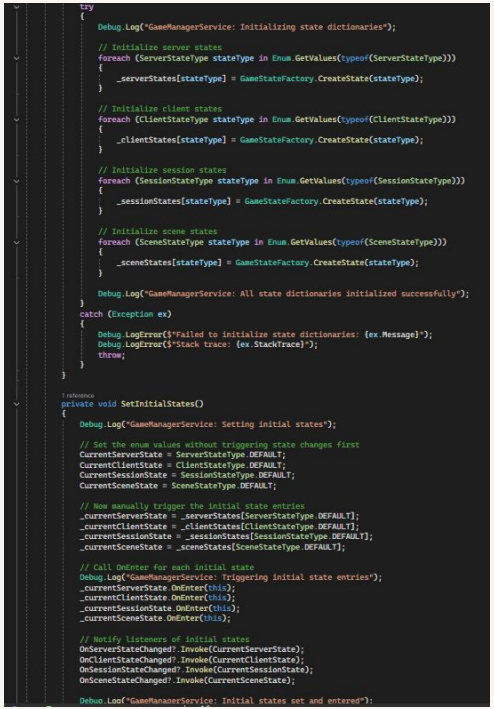
```
using System;
using System.Collections.Generic;
using UnityEngine;

namespace VRecover
{
    24 references
    public class ServiceLocator
    {
        private static ServiceLocator _instance;
        19 references
        public static ServiceLocator Instance => _instance ??= new ServiceLocator();

        private readonly Dictionary<Type, object> _services = new();

        1 reference
        private ServiceLocator() { } // Private constructor for singleton

        1 reference
        public static void Reset()
        {
            _instance = null;
        }

        0 references
        public void RegisterService<TInterface, TImplementation>(TImplementation implementation)
            where TImplementation : class, TInterface
        {
            _services[typeof(TInterface)] = implementation;
        }

        18 references
        public TInterface GetService<TInterface>() where TInterface : class
        {
            if (_services.TryGetValue(typeof(TInterface), out object service))
            {
                return (TInterface)service;
            }
            Debug.LogError($"Service of type {typeof(TInterface)} is not registered.");
            return null;
        }

        1 reference
        public void RegisterService(Type interfaceType, object implementation)
        {
            if (!interfaceType.IsInstanceOfType(implementation))
            {
                throw new ArgumentException($"Implementation must implement {interfaceType.Name}", nameof(implementation));
            }
            _services[interfaceType] = implementation;
        }
    }
}
```

Game Manager Service



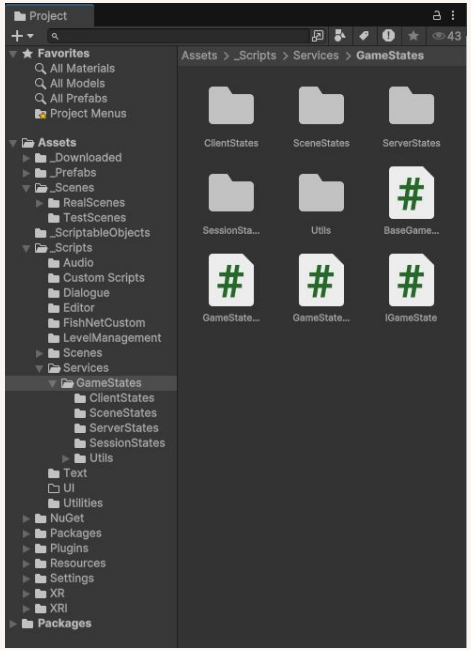
```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using FishNet.Managing;

namespace VRecover
{
    99+ references
    public interface IGameManagerService
    {
        8 references
        ServerStateType CurrentServerState { get; }
        8 references
        ClientStateType CurrentClientState { get; }
        8 references
        SessionStateType CurrentSessionState { get; }
        8 references
        SceneStateType CurrentSceneState { get; }

        event Action<ServerStateType> OnServerStateChanged;
        event Action<ClientStateType> OnClientStateChanged;
        event Action<SessionStateType> OnSessionStateChanged;
        event Action<SceneStateType> OnSceneStateChanged;

        20 references
        void UpdateServerState(ServerStateType newState);
        24 references
        void UpdateClientState(ClientStateType newState);
        41 references
        void UpdateSessionState(SessionStateType newState);
        41 references
        void UpdateSceneState(SceneStateType newState);

        1 reference
        void ReloadScene();
        2 references
        void SwitchToScene(string sceneName, bool updateCurrentSceneNum);
        1 reference
        void SwitchToScene(int sceneNum, bool updateCurrentSceneNum);
        1 reference
        void EndGame();

        28 references
        NetworkManager NetworkManager { get; }
        28 references
        NetworkStateHandler NetworkStateHandler { get; }
    }

    [Service]
    Unity Script (1 asset reference) | 0 references
    public class GameManagerService : MonoBehaviour, IGameManagerService, IInitializable
    {
        8 references
        public ServerStateType CurrentServerState { get; private set; }
        8 references
        public ClientStateType CurrentClientState { get; private set; }
        8 references
        public SessionStateType CurrentSessionState { get; private set; }
        8 references
        public SceneStateType CurrentSceneState { get; private set; }

        public event Action<ServerStateType> OnServerStateChanged;
        public event Action<ClientStateType> OnClientStateChanged;
        public event Action<SessionStateType> OnSessionStateChanged;
        public event Action<SceneStateType> OnSceneStateChanged;

        private IGameState _currentServerState;
        private IGameState _currentClientState;
        private IGameState _currentSessionState;
        private IGameState _currentSceneState;

        private readonly Dictionary<ServerStateType, IGameState> _serverStates = new();
        private readonly Dictionary<ClientStateType, IGameState> _clientStates = new();
```

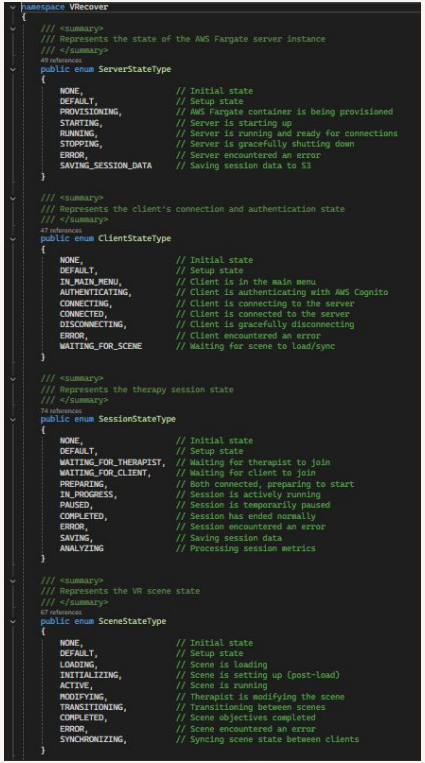
```
        private readonly Dictionary<ServerStateType, IGameState> _serverStates = new();
        private readonly Dictionary<ClientStateType, IGameState> _clientStates = new();
        private readonly Dictionary<SessionStateType, IGameState> _sessionStates = new();
        private readonly Dictionary<SceneStateType, IGameState> _sceneStates = new();

        private readonly List<string> _stateTransitionHistory = new();

        private NetworkStateSync _networkStateSync;
        private NetworkManager _networkManager;
        private NetworkStateHandler _networkStateHandler;

        28 references
        public NetworkManager NetworkManager => _networkManager;
        28 references
        public NetworkStateHandler NetworkStateHandler => _networkStateHandler;

        2 references
        public void Initialize()
        {

        }

        Unity Message | 0 references
        public void Start()
        {
            _networkManager = GameObject.FindFirstObjectByType<NetworkManager>();
            if (_networkManager == null)
            {
                Debug.LogError("GameManagerService: NetworkManager not found in scene!");
                return;
            }

            _networkStateHandler = new NetworkStateHandler(_networkManager);

            QualitySettings.vSyncCount = 0;

            _networkStateSync = FindFirstObjectByType<NetworkStateSync>();
            if (_networkStateSync == null)
            {
                Debug.Log("GameManagerService: NetworkStateSync not found, creating new instance");
                var networkObject = new GameObject("NetworkStateSync");
                _networkStateSync = networkObject.AddComponent<NetworkStateSync>();
            }

            InitializeStateDictionaries();
            SetInitialStates();
            Debug.Log("GameManagerService: Initialization complete");
        }

        1 reference
        private void InitializeStateDictionaries()
        {
            try
            {
                Debug.Log("GameManagerService: Initializing state dictionaries");

                // Initialize server states
                foreach (ServerStateType stateType in Enum.GetValues(typeof(ServerStateType)))
                {
                    _serverStates[stateType] = GameStateFactory.CreateState(stateType);
                }

                // Initialize client states
                foreach (ClientStateType stateType in Enum.GetValues(typeof(ClientStateType)))
                {
                    _clientStates[stateType] = GameStateFactory.CreateState(stateType);
```



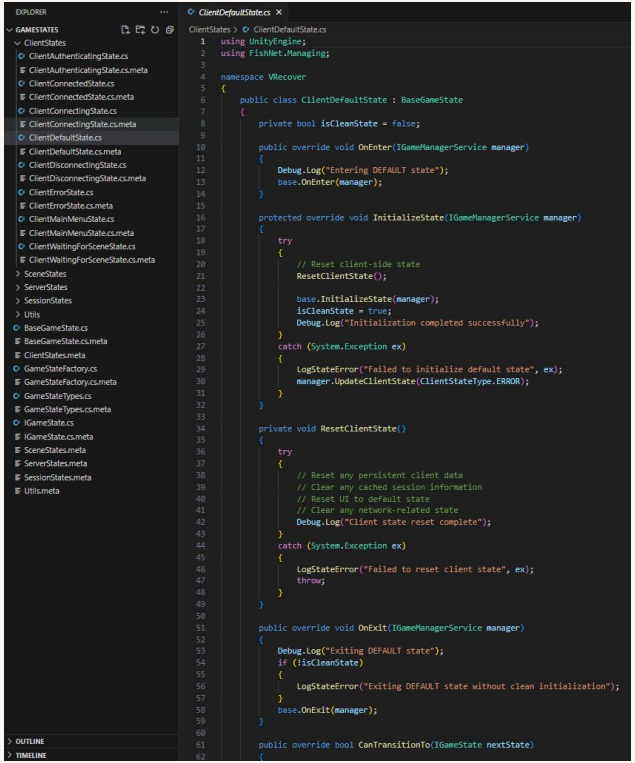
```
                try
                {
                    Debug.Log("GameManagerService: Initializing state dictionaries");

                    // Initialize server states
                    foreach (ServerStateType stateType in Enum.GetValues(typeof(ServerStateType)))
                    {
                        _serverStates[stateType] = GameStateFactory.CreateState(stateType);
                    }

                    // Initialize client states
                    foreach (ClientStateType stateType in Enum.GetValues(typeof(ClientStateType)))
                    {
                        _clientStates[stateType] = GameStateFactory.CreateState(stateType);
                    }

                    // Initialize session states
                    foreach (SessionStateType stateType in Enum.GetValues(typeof(SessionStateType)))
                    {
                        _sessionStates[stateType] = GameStateFactory.CreateState(stateType);
                    }

                    // Initialize scene states
                    foreach (SceneStateType stateType in Enum.GetValues(typeof(SceneStateType)))
                    {
                        _sceneStates[stateType] = GameStateFactory.CreateState(stateType);
                    }

                    Debug.Log("GameManagerService: All state dictionaries initialized successfully");
                }
                catch (Exception ex)
                {
                    Debug.LogError($"Failed to initialize state dictionaries: {ex.Message}");
                    Debug.LogError($"Stack trace: {ex.StackTrace}");
                    throw;
                }
            }

            1 reference
            private void SetInitialStates()
            {
                Debug.Log("GameManagerService: Setting initial states");

                // Set the enum values without triggering state changes first
                CurrentServerState = ServerStateType.DEFAULT;
                CurrentClientState = ClientStateType.DEFAULT;
                CurrentSessionState = SessionStateType.DEFAULT;
                CurrentSceneState = SceneStateType.DEFAULT;

                // Now manually trigger the initial state entries
                _currentServerState = _serverStates[ServerStateType.DEFAULT];
                _currentClientState = _clientStates[ClientStateType.DEFAULT];
                _currentSessionState = _sessionStates[SessionStateType.DEFAULT];
                _currentSceneState = _sceneStates[SceneStateType.DEFAULT];

                // Call OnEnter for each initial state
                Debug.Log("GameManagerService: Triggering initial state entries");
                _currentServerState.OnEnter(this);
                _currentClientState.OnEnter(this);
                _currentSessionState.OnEnter(this);
                _currentSceneState.OnEnter(this);

                // Notify listeners of initial states
                OnServerStateChanged?.Invoke(CurrentServerState);
                OnClientStateChanged?.Invoke(CurrentClientState);
                OnSessionStateChanged?.Invoke(CurrentSessionState);
                OnSceneStateChanged?.Invoke(CurrentSceneState);

                Debug.Log("GameManagerService: Initial states set and entered");
```

Game States



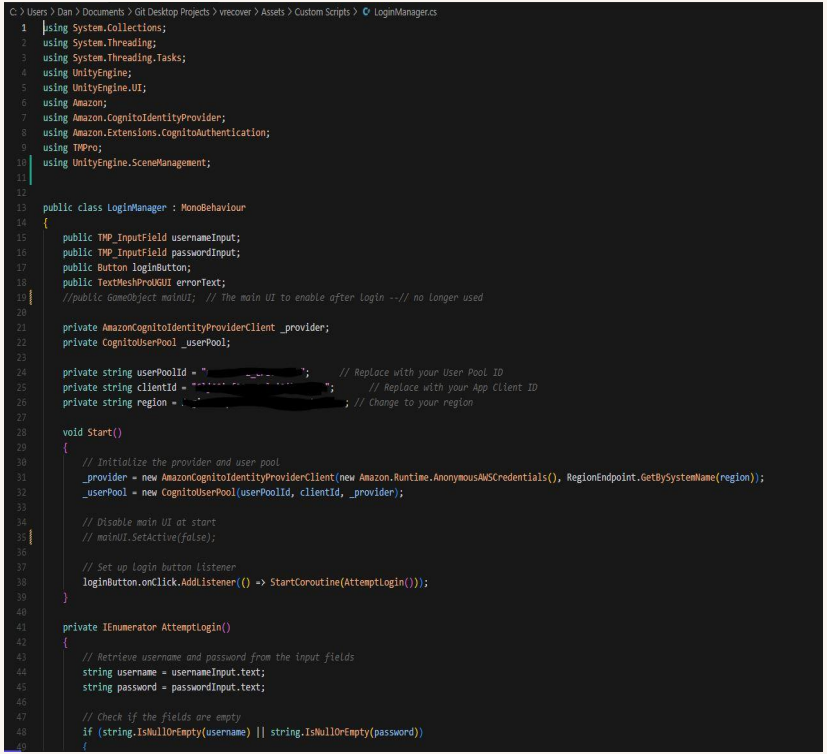


```
namespace VRecover
{
    /// <summary>
    /// Represents the state of the AWS Fargate server instance
    /// </summary>
    49 references
    public enum ServerStateType
    {
        NONE,                   // Initial state
        DEFAULT,                // Setup state
        PROVISIONING,           // AWS Fargate container is being provisioned
        STARTING,               // Server is starting up
        RUNNING,                // Server is running and ready for connections
        STOPPING,               // Server is gracefully shutting down
        ERROR,                  // Server encountered an error
        SAVING_SESSION_DATA     // Saving session data to S3
    }

    /// <summary>
    /// Represents the client's connection and authentication state
    /// </summary>
    47 references
    public enum ClientStateType
    {
        NONE,                   // Initial state
        DEFAULT,                // Setup state
        IN_MAIN_MENU,           // Client is in the main menu
        AUTHENTICATING,         // Client is authenticating with AWS Cognito
        CONNECTING,             // Client is connecting to the server
        CONNECTED,              // Client is connected to the server
        DISCONNECTING,          // Client is gracefully disconnecting
        ERROR,                  // Client encountered an error
        WAITING_FOR_SCENE       // Waiting for scene to load/sync
    }

    /// <summary>
    /// Represents the therapy session state
    /// </summary>
    74 references
    public enum SessionStateType
    {
        NONE,                   // Initial state
        DEFAULT,                // Setup state
        WAITING_FOR_THERAPIST,  // Waiting for therapist to join
        WAITING_FOR_CLIENT,     // Waiting for client to join
        PREPARING,              // Both connected, preparing to start
        IN_PROGRESS,            // Session is actively running
        PAUSED,                 // Session is temporarily paused
        COMPLETED,              // Session has ended normally
        ERROR,                  // Session encountered an error
        SAVING,                 // Saving session data
        ANALYZING               // Processing session metrics
    }

    /// <summary>
    /// Represents the VR scene state
    /// </summary>
    67 references
    public enum SceneStateType
    {
        NONE,                   // Initial state
        DEFAULT,                // Setup state
        LOADING,                // Scene is loading
        INITIALIZING,           // Scene is setting up (post-load)
        ACTIVE,                 // Scene is running
        MODIFYING,              // Therapist is modifying the scene
        TRANSITIONING,          // Transitioning between scenes
        COMPLETED,              // Scene objectives completed
        ERROR,                  // Scene encountered an error
        SYNCHRONIZING,          // Syncing scene state between clients
    }
```

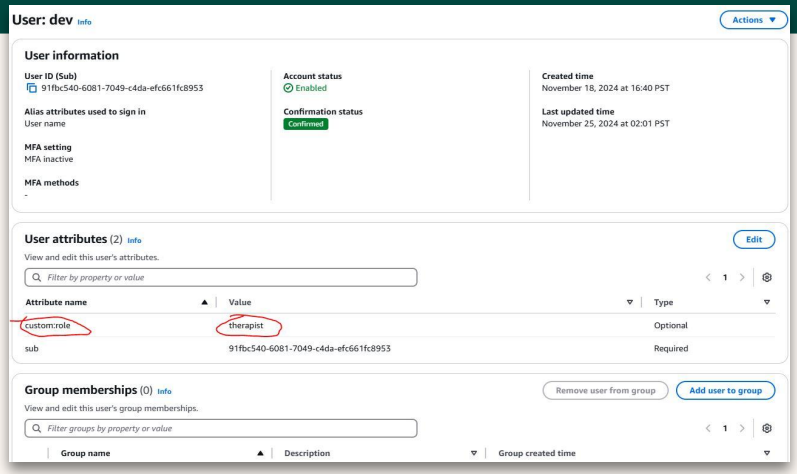


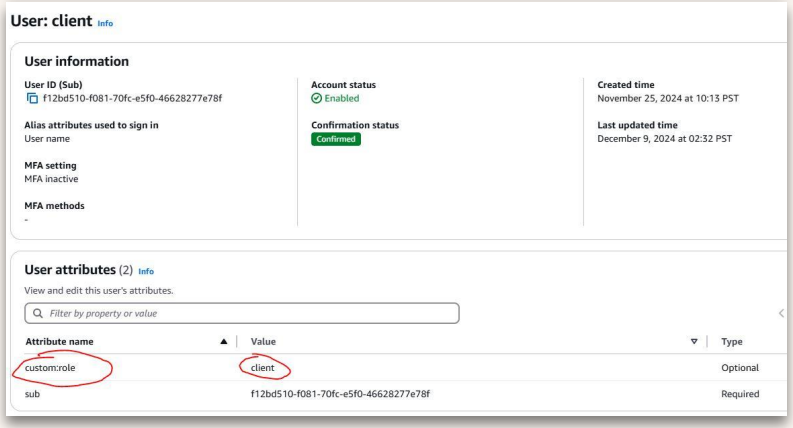
```
using UnityEngine;
using FishNet.Managing;

namespace VRecover
{
    public class ClientDefaultState : BaseGameState
    {
        private bool isCleanState = false;

        public override void OnEnter(IGameManagerService manager)
        {
            Debug.Log("Entering DEFAULT state");
            base.OnEnter(manager);
        }

        protected override void InitializeState(IGameManagerService manager)
        {
            try
            {
                // Reset client-side state
                ResetClientState();

                base.InitializeState(manager);
                isCleanState = true;
                Debug.Log("Initialization completed successfully");
            }
            catch (System.Exception ex)
            {
                LogStateError("Failed to initialize default state", ex);
                manager.UpdateClientState(ClientStateType.ERROR);
            }
        }

        private void ResetClientState()
        {
            try
            {
                // Reset any persistent client data
                // Clear any cached session information
                // Reset UI to default state
                // Clear any network-related state
                Debug.Log("Client state reset complete");
            }
            catch (System.Exception ex)
            {
                LogStateError("Failed to reset client state", ex);
                throw;
            }
        }

        public override void OnExit(IGameManagerService manager)
        {
            Debug.Log("Exiting DEFAULT state");
            if (!isCleanState)
            {
                LogStateError("Exiting DEFAULT state without clean initialization");
            }
            base.OnExit(manager);
        }

        public override bool CanTransitionTo(IGameState nextState)
        {
```

AWS Login Manager

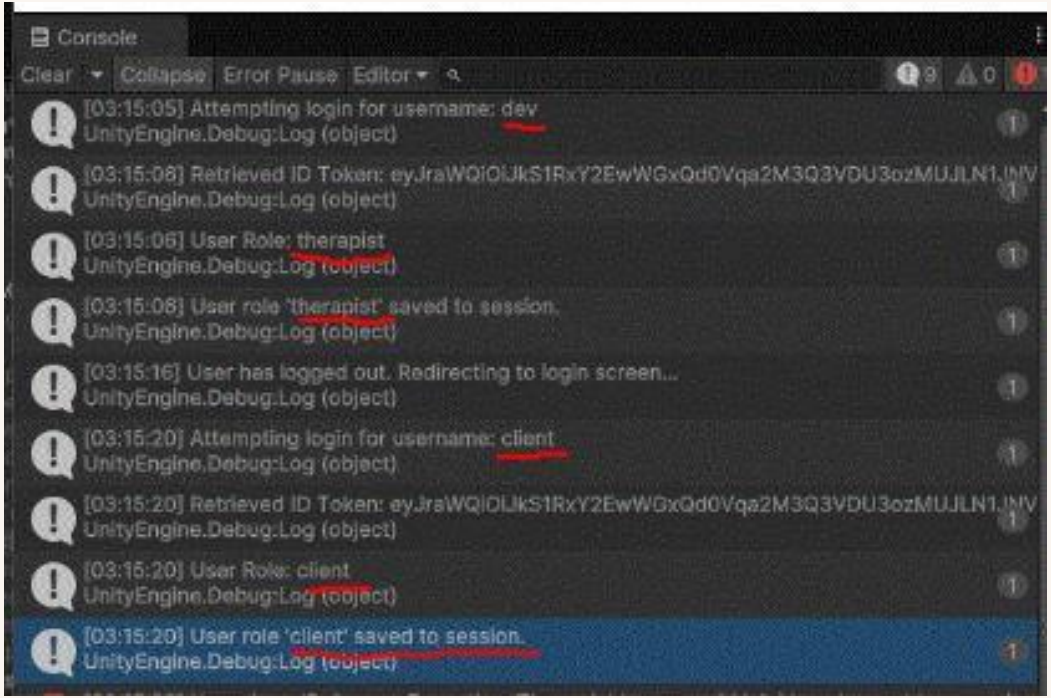


AWS User Custom Attribute

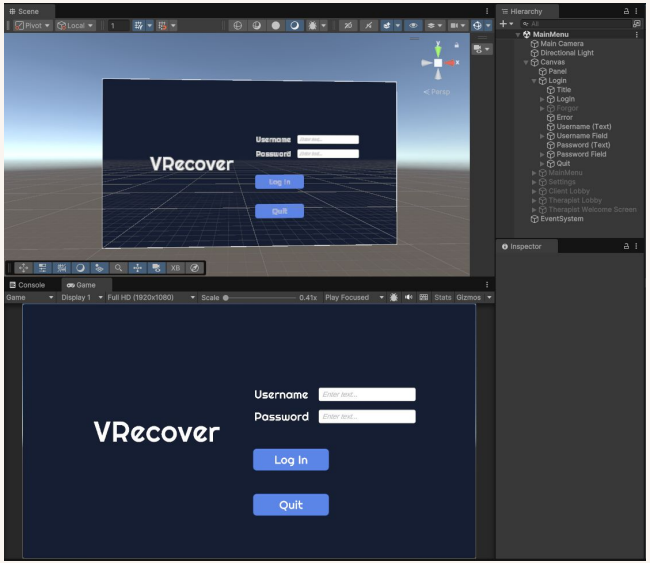


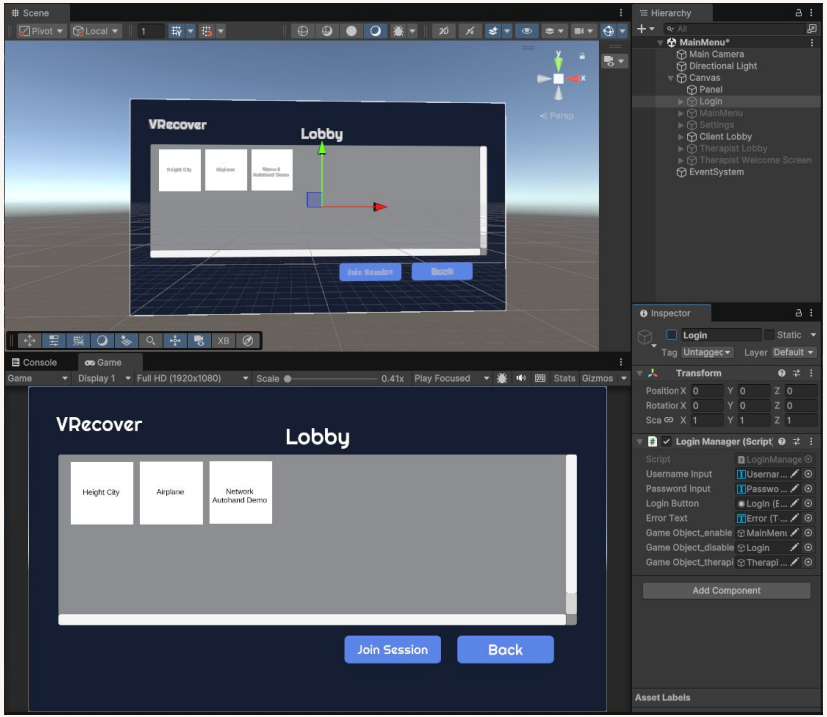


Attribute Reading

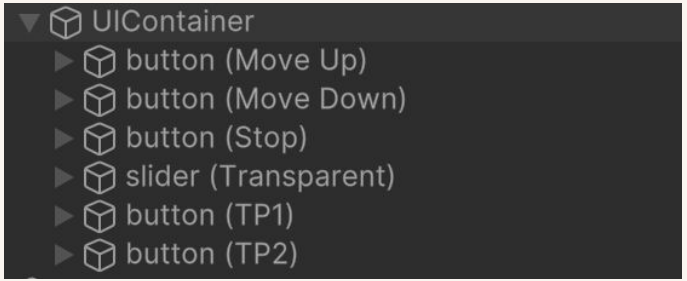


Menu


VRecover
Username
Password
Log In
Quit


VRecover
Lobby
Height City
Airplane
Network Autohand Demo
Join Session
Back

Serialized UI



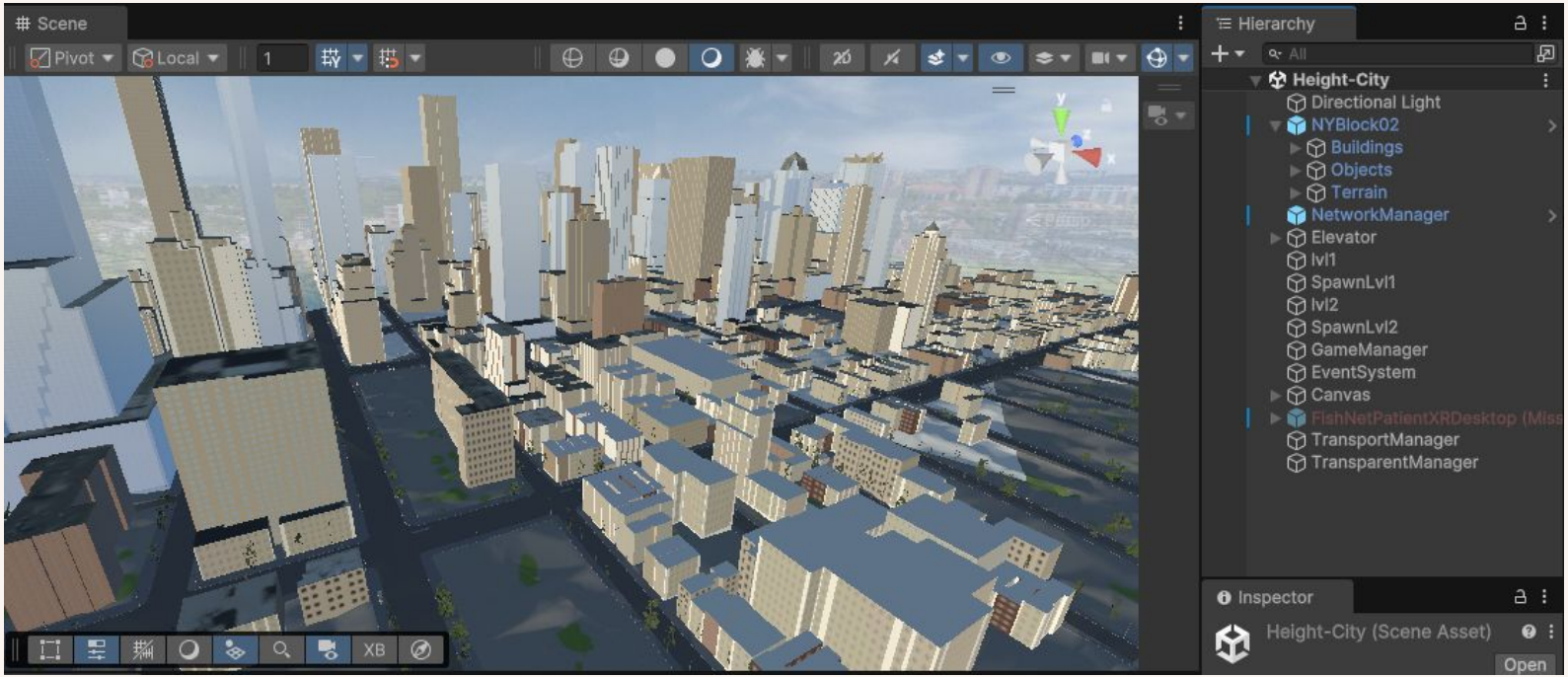

```
{
    "scene": {
        "name": "jon-test-scene",
        "elements": [
            {
                "type": "button",
                "label": "Move Up",
                "targetObject": "Elevator",
                "method": "MoveElevatorUp"
            },
            {
                "type": "button",
                "label": "Move Down",
                "targetObject": "Elevator",
                "method": "MoveElevatorDown"
            },
            {
                "type": "button",
                "label": "Stop",
                "targetObject": "Elevator",
                "method": "StopElevator"
            },
            {
                "type": "slider",
                "label": "Transparent",
                "min": 0,
                "max": 1,
                "targetObject": "Elevator",
                "property": "Renderer.material.color.a"
            },
            {
                "type": "button",
                "label": "TP1",
                "targetObject": "TransportManager",
                "method": "TeleportToSpawnPoint",
                "parameters": [ 0 ]
            },
            {
                "type": "button",
                "label": "TP2",
                "targetObject": "TransportManager",
                "method": "TeleportToSpawnPoint",
                "parameters": [ 1 ]
            }
        ]
    }
}
```

DynamicUIManager

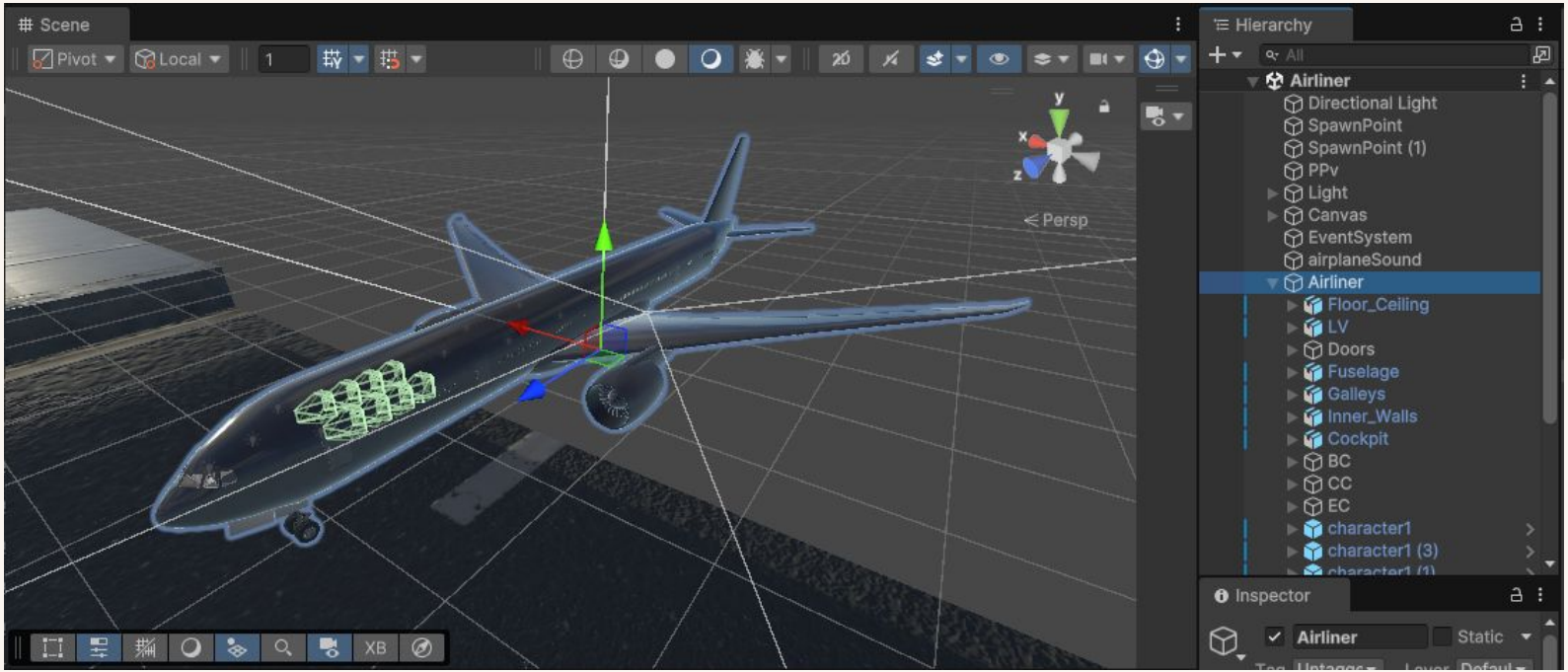
```
using UnityEngine;
using Newtonsoft.Json;
using System;
using TMPro;
using UnityEngine.UI;
using System.Reflection;

public class DynamicUIManager : MonoBehaviour
{
    [Header("Prefabs")]
    [SerializeField] private GameObject sliderPrefab;
    [SerializeField] private GameObject togglePrefab;
    [SerializeField] private GameObject buttonPrefab;

    [Header("UI Container (Assign Canvas Here)")]
    [SerializeField] private Transform canvas;

    [Header("Metadata File")]
    [SerializeField] private string metadataFileName = "metadata";

    private Transform uiContainer;

    void Start()
    {
        SetupUIContainer();

        JsonSceneMetadata sceneMetadata = LoadMetadata(metadataFileName);
        if (sceneMetadata == null)
        {
            Debug.LogError("Failed to load metadata.");
            return;
        }

        if (!string.Equals(sceneMetadata.name, UnityEngine.SceneManagement.SceneManager.GetActiveScene().name, StringComparison.OrdinalIgnoreCase))
        {
            Debug.LogWarning($"Metadata is for scene '{sceneMetadata.name}', but current scene is '{UnityEngine.SceneManagement.SceneManager.GetActiveScene().name}'.");
            return;
        }

        foreach (var element in sceneMetadata.elements)
        {
            GameObject targetObject = GameObject.Find(element.targetObject);
            if (targetObject == null)
            {
                Debug.LogWarning($"Target object '{element.targetObject}' not found in the scene.");
                continue;
            }

            CreateUIElement(element, targetObject);
        }
    }

    private void SetupUIContainer()
    {
        if (uiContainer == null)
        {
            GameObject containerObject = new GameObject("UIContainer");
            containerObject.transform.SetParent(canvas, false);

            RectTransform rectTransform = containerObject.AddComponent<RectTransform>();
            rectTransform.anchorMin = new Vector2(0, 0);
            rectTransform.anchorMax = new Vector2(1, 1);
            rectTransform.offsetMin = Vector2.zero;
            rectTransform.offsetMax = Vector2.zero;

            var layoutGroup = containerObject.AddComponent<VerticalLayoutGroup>();
            layoutGroup.spacing = 20;
            layoutGroup.padding = new RectOffset(10, 10, 10, 10);
            layoutGroup.childAlignment = TextAnchor.UpperCenter;

            var contentFitter = containerObject.AddComponent<ContentSizeFitter>();
            contentFitter.verticalFit = ContentSizeFitter.FitMode.PreferredSize;

            uiContainer = containerObject.transform;
        }
    }


    private JsonSceneMetadata LoadMetadata(string fileName)
    {
        TextAsset jsonFile = Resources.Load<TextAsset>(fileName);
        if (jsonFile == null)
        {
            Debug.LogError($"Metadata file '{fileName}' not found in Resources folder.");
            return null;
        }
```

```
    private void CreateUIElement(UIElement element, GameObject targetObject)
    {
        GameObject uiElement = null;

        if (string.Equals(element.type, "slider", StringComparison.OrdinalIgnoreCase))
        {
            uiElement = Instantiate(sliderPrefab);
            ConfigureSlider(uiElement, element, targetObject);
        }
        else if (string.Equals(element.type, "toggle", StringComparison.OrdinalIgnoreCase))
        {
            uiElement = Instantiate(togglePrefab);
            ConfigureToggle(uiElement, element, targetObject);
        }
        else if (string.Equals(element.type, "button", StringComparison.OrdinalIgnoreCase))
        {
            uiElement = Instantiate(buttonPrefab);
            ConfigureButton(uiElement, element, targetObject);
        }

        if (uiElement != null)
        {
            uiElement.transform.SetParent(uiContainer, false);
            uiElement.name = $"{element.type} ({element.label ?? element.targetObject})";
        }
    }

    private void ConfigureSlider(GameObject sliderObj, UIElement element, GameObject targetObject)
    {
        var label = sliderObj.transform.Find("Label").GetComponent<TMPro.TextMeshProUGUI>();
        label.text = element.label;

        var slider = sliderObj.GetComponentInChildren<UnityEngine.UI.Slider>();

        object currentValueObj = PropertyUtility.GetValue(targetObject, element.property);
        if (currentValueObj is float currentValue)
        {
            float range = element.max - element.min;

            slider.minValue = currentValue - range / 2f;
            slider.maxValue = currentValue + range / 2f;
            slider.value = currentValue;

            slider.onValueChanged.AddListener(value =>
            {
                PropertyUtility.SetValue(targetObject, element.property, value);
            });
        }
        else
        {
            Debug.LogWarning($"Property '{element.property}' is not a float and cannot be controlled by the slider.");
        }
    }

    private void ConfigureToggle(GameObject toggleObj, UIElement element, GameObject targetObject)
    {
        var label = toggleObj.GetComponentInChildren<TMPro.TextMeshProUGUI>();
        label.text = element.label;

        var toggle = toggleObj.GetComponentInChildren<UnityEngine.UI.Toggle>();
        toggle.isOn = Convert.ToBoolean(element.defaultValue);

        toggle.onValueChanged.AddListener(isOn =>
        {
            PropertyUtility.SetValue(targetObject, element.property, isOn);
        });
    }

    private void ConfigureButton(GameObject buttonObj, UIElement element, GameObject targetObject)
    {
        var label = buttonObj.GetComponentInChildren<TMPro.TextMeshProUGUI>();
        label.text = element.label;

        var button = buttonObj.GetComponentInChildren<UnityEngine.UI.Button>();
        button.onClick.AddListener(() =>
        {
            // Look for any MonoBehaviour scripts on the target object
            var scripts = targetObject.GetComponents<MonoBehaviour>();
            if (scripts.Length == 0)
            {
                Debug.LogError($"No MonoBehaviour scripts found on {targetObject.name}.");
                return;
            }

            // Loop through each script to find the specified method
            MethodInfo method = null;
```

Heights Scene



Airliner Scene



Dockerfile

```
# Use the available UnityCI Docker image
FROM unityci/editor:ubuntu-2020.2.1f1-linux-il2cpp-3.1.0

# Set the working directory inside the container
WORKDIR /vrecover

# Copy your project files into the container
COPY . .
 Ctrl+L to chat, Ctrl+K to generate
# Build the Unity project
RUN echo "Building Unity project..." && \
    unity-editor -projectPath /vrecover \
    -quit -batchmode -buildTarget linux64 \
    -logFile /vrecover/build.log \
    -outputPath /vrecover/Build/ServerBuild.x86_64 || \
    echo "Unity build failed. Check /vrecover/build.log for details."

# Verify the build output
RUN echo "Verifying server build output..." && \
    if [ -f "/vrecover/Build/ServerBuild.x86_64" ]; then \
    chmod +x /vrecover/Build/ServerBuild.x86_64 && \
    echo "Build output found and made executable."; \
    else \
    echo "Build output not found. Check Unity build logs."; \
    fi

# Expose the necessary ports for FishNet
EXPOSE 7770/tcp
EXPOSE 7771/udp

# Run the server in headless mode
ENTRYPOINT ["/vrecover/Build/ServerBuild.x86_64", "-batchmode", "-nographics"]
```

GitHub Actions Workflow

deploy-to-aws.yaml

.github > workflows > deploy-to-aws.yaml

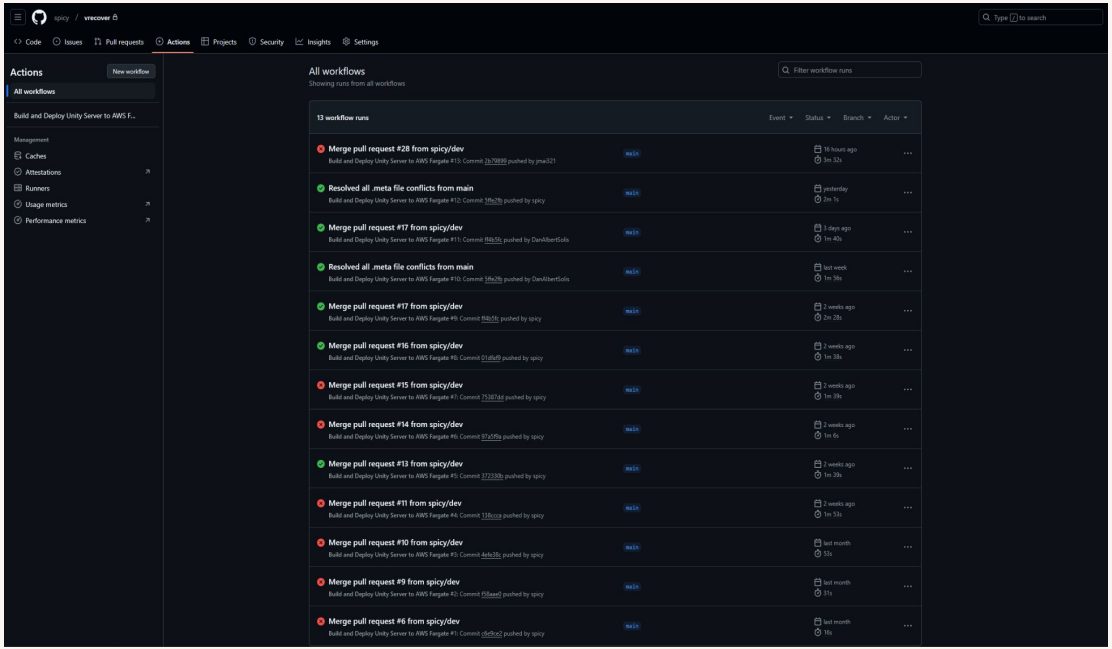
```yaml
name: Build and Deploy Unity Server to AWS Fargate

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
        uses: actions/checkout@v2

      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Log in to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build Docker Image
        run: |
          docker build -t unity-server .
          docker tag unity-server:latest ${{ secrets.ECR_REPOSITORY_URI }}:latest

      - name: Push Docker Image to ECR
        run: |
          docker push ${{ secrets.ECR_REPOSITORY_URI }}:latest

      - name: Register ECS Task Definition
        env:
          ECR_IMAGE: ${{ secrets.ECR_REPOSITORY_URI }}:latest
        run: |
          # Create a new ECS task definition based on a JSON template
          cat <<EOF > task-definition.json
          {
            "family": "VRecoverUnityServer-Task",
            "networkMode": "awsvpc",
            "containerDefinitions": [
```

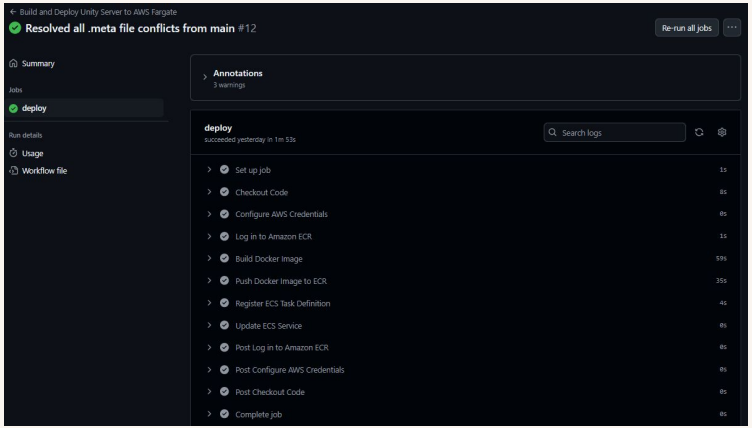
deploy-to-aws.yaml

.github > workflows > deploy-to-aws.yaml

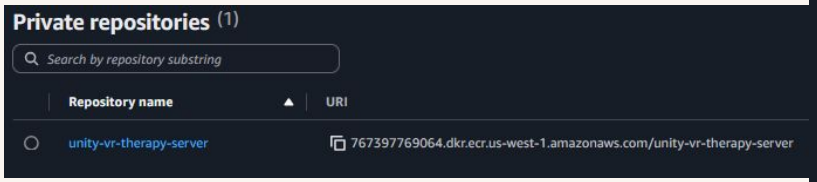
```yaml
jobs:
  deploy:
    steps:
      - name: Register ECS Task Definition
        env:
        run: |
          # Create a new ECS task definition based on a JSON template
          cat <<EOF > task-definition.json
          {
            "family": "VRecoverUnityServer-Task",
            "networkMode": "awsvpc",
            "containerDefinitions": [
              {
                "name": "UnityGameServer",
                "image": "$ECR_IMAGE",
                "essential": true,
                "memory": 512,
                "cpu": 256,
                "portMappings": [
                  {
                    "containerPort": 7770,
                    "protocol": "tcp"
                  },
                  {
                    "containerPort": 7771,
                    "protocol": "udp"
                  }
                ]
              }
            ],
            "requiresCompatibilities": [
              "FARGATE"
            ],
            "cpu": "256",
            "memory": "512",
            "executionRoleArn": "arn:aws:iam::767397769064:role/ecsTaskExecutionRole"
          }
          EOF

          aws ecs register-task-definition --cli-input-json file://task-definition.json

      - name: Update ECS Service
        run: |
          aws ecs update-service --cluster ${{ secrets.ECS_CLUSTER_NAME }} --service ${{ secrets.ECS_SERVICE_NAME }} --force-new-deployment --region ${{ secrets.AWS_REGI
```

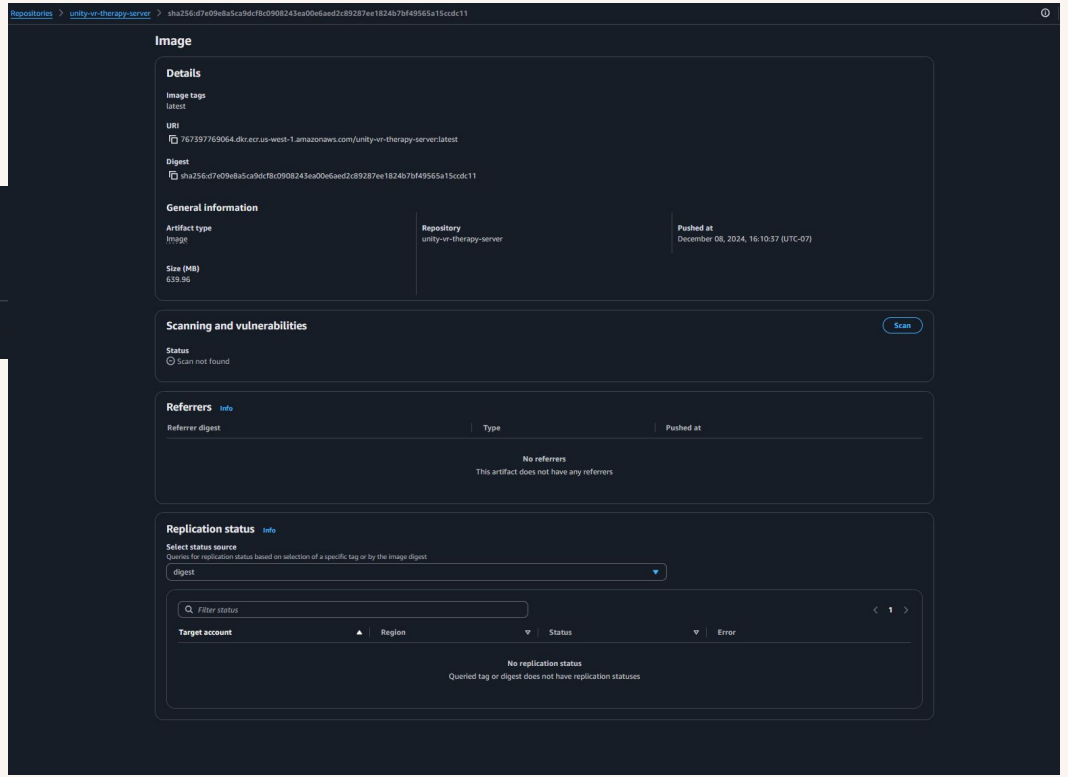
GitHub Actions Runs



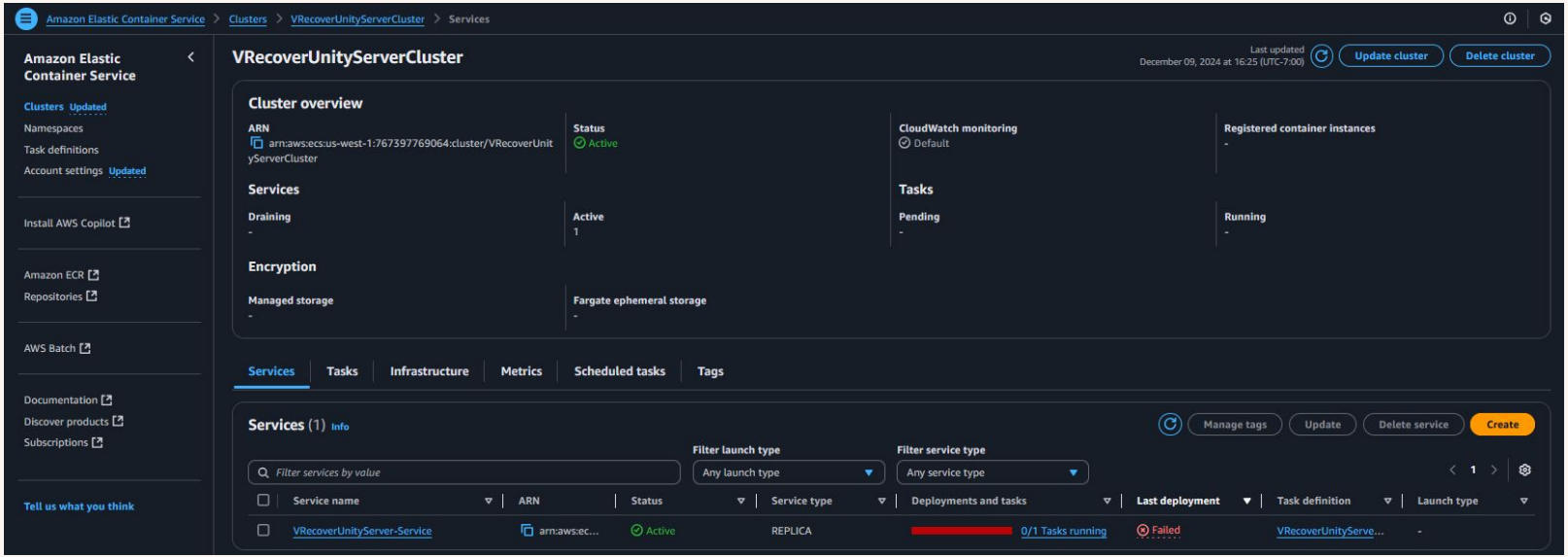


AWS Repository

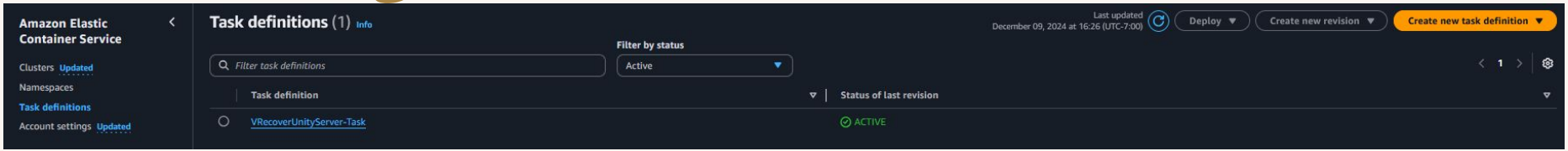


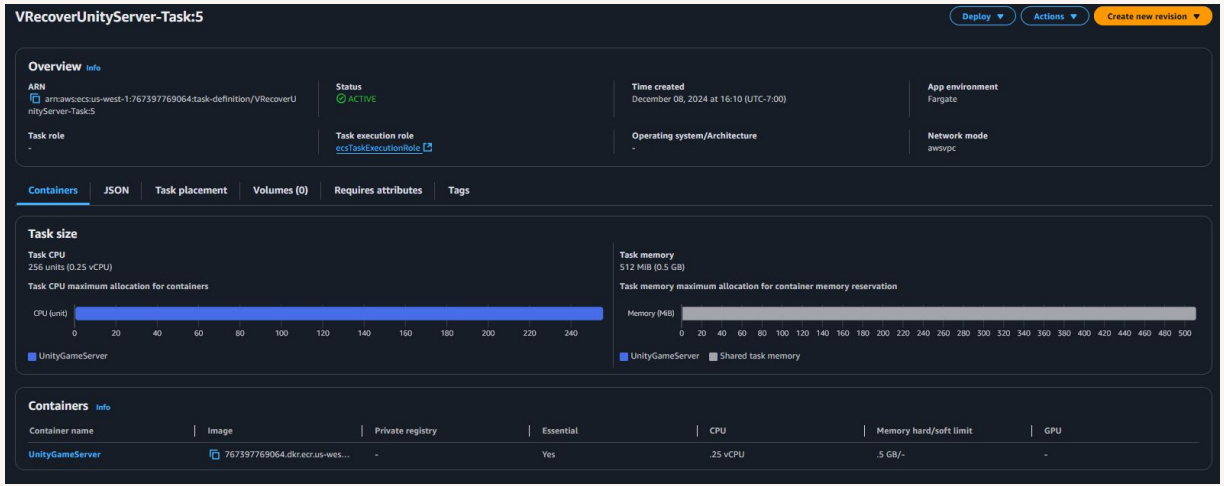


AWS Fargate, Clusters

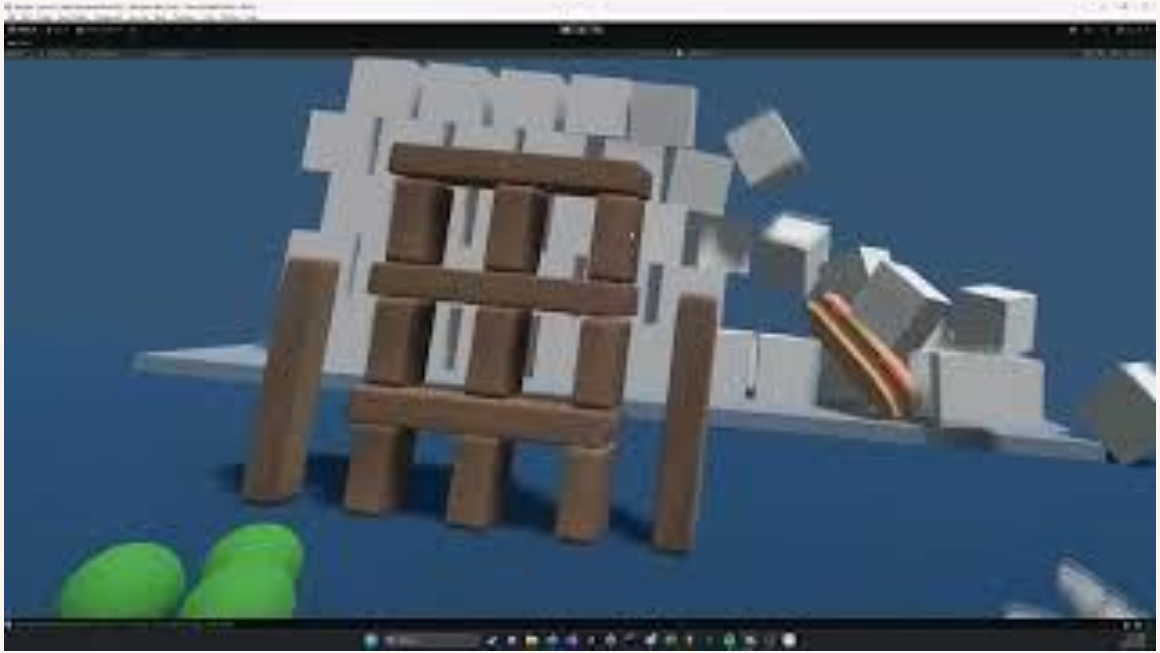


AWS Fargate Task Definitions





Demo Video



Thank You

CSU – Fullerton

CPSC 491