

Unity Game Architecture Documentation

This document provides a robust overview of the core systems implemented as part of your Unity Starter Pack architecture. It includes:

- Rationale behind each system
- Core implementation summary
- Example usage patterns (to be placed in your `Examples/` folder in Unity)

Project Structure (Suggested)

```
/Project
├── Scripts
│   ├── Runtime
│   │   ├── Events
│   │   ├── DI
│   │   ├── GameState
│   │   └── SaveSystem
│   └── Examples
```

1. Event System

Why?

- Decouples sender and receiver
- Promotes clean, modular communication
- Enables flexible editor-driven event setup via `ScriptableObject`

Core File: `EventChannelSO<T>`

A generic base class for event channels.

```
public abstract class EventChannelSO<T> : ScriptableObject
{
    public event Action<T> OnEventRaised;

    public void RaiseEvent(T value)
    {
        OnEventRaised?.Invoke(value);
    }

    public void RegisterListener(Action<T> listener) => OnEventRaised +=
listener;
    public void UnregisterListener(Action<T> listener) => OnEventRaised -=
listener;
}
```

Example Usage

IntEventChannelSO.cs

```
[CreateAssetMenu(menuName = "Events/Int Event Channel")]
public class IntEventChannelSO : EventChannelSO<int> { }
```

Example Script (Raiser)

```
[SerializeField] private IntEventChannelSO onScoreChanged;
onScoreChanged.RaiseEvent(100);
```

Example Script (Listener)

```
[SerializeField] private IntEventChannelSO onScoreChanged;
void OnEnable() => onScoreChanged.RegisterListener(HandleScore);
void OnDisable() => onScoreChanged.UnregisterListener(HandleScore);
void HandleScore(int newScore) => Debug.Log("Score: " + newScore);
```

2. Dependency Injection (DI)

Why?

- Avoids tight coupling and FindObjectOfType
- Makes testing and mocking easier
- Cleaner architecture and reusability

Core File: **ServiceLocator.cs**

```
public static class ServiceLocator
{
    private static Dictionary<Type, object> services = new();

    public static void Register<T>(T service) where T : class =>
    services[typeof(T)] = service;
    public static T Get<T>() where T : class =>
    services.TryGetValue(typeof(T), out var s) ? s as T : null;
    public static void Clear() => services.Clear();
}
```

Example Script

BootstrapInstaller.cs

```
[SerializeField] private ScoreSystem scoreSystem;
[SerializeField] private GameManager gameManager;

void Awake()
{
    ServiceLocator.Register(scoreSystem);
    ServiceLocator.Register(gameManager);
}
```

Usage in Another Script

```
var gameManager = ServiceLocator.Get<GameManager>();  
gameManager.DoSomething();
```



3. Game State Manager



Why?

- Centralizes game state logic (Menu, Play, Pause, GameOver, etc.)
- Enables UI and systems to react to game flow changes



Core File: `GameStateManager.cs`

```
public enum GameState { MainMenu, Playing, Paused, GameOver }  
  
public class GameStateManager : MonoBehaviour  
{  
    public static GameStateManager Instance { get; private set; }  
    public GameState CurrentState { get; private set; }  
    public event Action<GameState> OnStateChanged;  
  
    void Awake()  
    {  
        Instance = this;  
        DontDestroyOnLoad(gameObject);  
        ChangeState(GameState.MainMenu);  
    }  
  
    public void ChangeState(GameState newState)  
    {  
        if (newState == CurrentState) return;  
        CurrentState = newState;  
        OnStateChanged?.Invoke(newState);  
    }  
}
```



Example Script

```
void Start()  
{  
    GameStateManager.Instance.OnStateChanged += HandleStateChange;  
}  
  
void HandleStateChange(GameState state)  
{  
    if (state == GameState.Paused) ShowPauseMenu();  
}
```



4. Save & Load System



Why?

- Save game state, progress, preferences
- Uses JSON for human-readable and portable format

Core Files

SaveData.cs

```
[Serializable]
public class SaveData
{
    public int playerScore;
    public string currentLevel;
    public float[] playerPosition;
}
```

SaveSystem.cs

```
public static class SaveSystem
{
    private static string SaveFilePath =>
        Path.Combine(Application.persistentDataPath, "savegame.json");

    public static void SaveGame(SaveData data)
    {
        File.WriteAllText(SaveFilePath, JsonUtility.ToJson(data, true));
    }

    public static SaveData LoadGame()
    {
        if (!File.Exists(SaveFilePath)) return new SaveData();
        return
            JsonUtility.FromJson<SaveData>(File.ReadAllText(SaveFilePath));
    }

    public static void DeleteSave()
    {
        if (File.Exists(SaveFilePath)) File.Delete(SaveFilePath);
    }
}
```

Example Script

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.S))
        SaveSystem.SaveGame(new SaveData { playerScore = 100 });

    if (Input.GetKeyDown(KeyCode.L))
        Debug.Log(SaveSystem.LoadGame().playerScore);
}
```

Final Notes

- Keep core systems separated from usage examples
- Use your Examples/ folder to test and iterate

- Plug in these systems into gameplay gradually
- Add tooling or inspector extensions as needed

Let me know if you'd like this exported as a PDF or Markdown file too!