ChatGPT

Unity Developer Interview Flashcards

This document contains **100 interview questions and answers** for Unity developers along with references to the official Unity documentation. The second half of the document provides a **project specification** and **prompt** you can feed to a coding agent (for example, Codex running in VS Code) to build a local web application that displays these flashcards and lets you add more.

Flashcards

Each question is numbered and paired with a concise answer. Citations point to relevant Unity documentation or guides. If you are writing your own cards later, keep the answers brief and cite official sources.

MonoBehaviour & Script Lifecycle

1. **What is the purpose of `MonoBehaviour` in Unity?**
   Answer: `MonoBehaviour` is the base class from which every Unity script derives. It allows you to attach scripts to GameObjects so that they participate in Unity's event loop (functions like `Awake`, `Start`, `Update`, etc.). Without inheriting from `MonoBehaviour`, a class cannot be attached to a GameObject and will not receive engine callbacks [1] [2].

2. **How do `Awake` and `Start` differ?**
   Answer: `Awake` is called when the script instance is being loaded and is used to initialize variables; it is called once and always before any `Start` methods [3]. `Start` is called on the frame when a script is enabled just before any `Update` methods; it is also called once during the lifetime of the script and can be defined as a coroutine [4] [5].

3. **When should you use `FixedUpdate` instead of `Update`?**
   Answer: `FixedUpdate` is called every fixed-framerate frame and should be used for physics calculations and applying forces to Rigidbodies [1]. `Update` is called every frame and is appropriate for non-physics game logic and input handling [2].

4. **What is `LateUpdate` and when is it useful?**
   Answer: `LateUpdate` is called every frame after all `Update` functions have executed. Use it for follow-up logic that depends on other objects finishing their `Update`, such as camera follow scripts [6].

5. **Can `Start` be a coroutine?**
   Answer: Yes. You can declare `Start` with the `IEnumerator` return type and use `yield` statements inside it. The documentation notes that `Start` can be defined as a coroutine to perform asynchronous initialization [5].

6. **What is the correct way to destroy objects at runtime?**
   Answer: Use `Object.Destroy(obj)` to mark an object for destruction at the end of the current frame. `DestroyImmediate` is intended for editor scripts and destroys objects

immediately; using it at runtime can cause errors or unintentionally remove assets. To deactivate an object without destroying it, call `GameObject.SetActive(false)` [7].

7. **When is `OnCollisionEnter` called, and what is required for it to fire?**
   Answer: `OnCollisionEnter` is invoked when the collider/rigidbody attached to a `MonoBehaviour` begins touching another collider/rigidbody. Collision events are only sent if at least one of the colliders has a non-kinematic Rigidbody attached [8].

8. **Why should you avoid modifying a Rigidbody's `Transform` directly?**
   Answer: Unity's physics engine expects you to move Rigidbodies using physics methods. Directly changing the `Transform` of a Rigidbody can cause unrealistic behaviour; instead, use methods like `AddForce` or `AddTorque` on the Rigidbody [9].

9. **What is `WaitForSeconds`, and how does `Time.timeScale` affect it?**
   Answer: `WaitForSeconds` is a yield instruction used in coroutines to suspend execution for a given number of seconds. The actual wait time is multiplied by `Time.timeScale`, so if you change the time scale (e.g., set it to 0 when pausing), the wait will be scaled accordingly [10].

10. **What is a coroutine in Unity, and how is it different from multi-threading?**
    Answer: A coroutine is a method that returns `IEnumerator` and can suspend execution via `yield` statements. Coroutines run on Unity's main thread and allow you to spread tasks across multiple frames; they are **not** multi-threading. For true parallelism, use the Job System or C# async/await [11].

GameObjects & Components

1. **What is a GameObject in Unity?**
   Answer: A GameObject is the fundamental object in Unity scenes. It can represent characters, props, scenery, cameras, waypoints, etc. A GameObject's behaviour comes from the Components attached to it [12]. Even an empty GameObject contains a `Transform` component by default [12].

2. **What is a Component, and how do Components relate to GameObjects?**
   Answer: Components are functional pieces of a GameObject that define its behaviour. You can attach many Components to a single GameObject to build up its functionality. Every GameObject must have exactly one `Transform` component [13].

3. **Why does every GameObject have a `Transform`?**
   Answer: The `Transform` determines a GameObject's position, rotation and scale. A GameObject cannot exist without a `Transform`; without it, the object would have no location in the scene [12] [14].

4. **How do Tags and Layers differ, and what are they used for?**
   Answer: Tags are marker values used to identify objects in scripts (for example, tagging certain objects as "Enemy" or "Pickup"), while Layers group GameObjects to include or exclude them from operations such as rendering, lighting and physics. The Tags and Layers settings let you define custom tags and layers [15].

5. **What is the difference between disabling a GameObject and disabling a Component?**
   Answer: Disabling a GameObject using `SetActive(false)` deactivates the GameObject and

all its Components and children. Disabling a Component by setting its `enabled` property to `false` stops only that Component's behaviour while leaving the GameObject and other Components active. To temporarily hide an object without destroying it, use `SetActive(false)` [7].

6. **How do you add a Component to a GameObject in a script?**
   Answer: Call `gameObject.AddComponent<YourComponentType>()` on a GameObject. This creates and attaches a new Component of type `YourComponentType` and returns a reference to it. Components define the behaviour of GameObjects [13].

7. **What are Prefabs, and why are they useful?**
   Answer: A Prefab is an asset that stores a fully configured GameObject and all of its Components and child objects. Prefabs act as templates from which you can create new instances. Editing the Prefab asset propagates changes to all of its instances, and you can nest Prefabs and override properties on individual instances [16].

8. **What is a nested Prefab?**
   Answer: Unity's Prefab system allows one Prefab to be a child of another. Nested Prefabs help organize complex objects: you can build small Prefabs and assemble them into larger structures. Properties can be overridden on instances while maintaining a link to the parent Prefabs [16].

9. **How do you instantiate a Prefab at runtime?**
   Answer: Use `Instantiate(prefab, position, rotation)` in a script. This creates a copy of the Prefab at the given position and rotation. Prefabs are commonly used to spawn power-ups, projectiles, NPCs and other objects during gameplay [16].

10. **What is the difference between a GameObject and a Prefab?**
    Answer: A GameObject exists in a scene and can be manipulated at runtime. A Prefab is an asset that stores a GameObject's configuration and can be instantiated multiple times across scenes. Editing the Prefab asset updates all its instances [16].

11. **How can you organize GameObjects hierarchically?**
    Answer: GameObjects can be parented to create hierarchies. The `Transform` stores parent/child relationships; a child's local position, rotation and scale are relative to its parent. If a Transform has no parent, its values are in world space [14].

12. **What is the difference between world space and local space?**
    Answer: World space coordinates are measured relative to the scene origin, while local space coordinates are measured relative to a GameObject's parent. The Transform's position, rotation and scale values are relative to its parent; if there is no parent, they are in world space [14].

Scenes & Asset Management

1. **What is a Scene in Unity?**
   Answer: A Scene is an asset containing part or all of a game. Each Scene file can represent a level, menu or environment. When you create a new project, Unity opens a sample Scene containing only a Camera and a Light [17].

2. **Can you have multiple Scenes in a single project?**
   Answer: Yes. A project can contain multiple Scenes (for example, one per level). Unity also supports multi-scene editing, allowing you to load and edit several Scenes at once [17] [18].

3. **What is the New Scene dialog and what can you do there?**
   Answer: The New Scene dialog lists Scene templates and allows you to create new Scenes from templates, pin frequently used templates and edit them. It supports multi-scene editing by letting you load multiple Scenes in the editor [18].

4. **How do you manage Scenes in the Build Settings?**
   Answer: Open **File > Build Settings**. The Scenes In Build list shows all Scenes that will be included in your build. You can add open Scenes, drag Scenes from the Project window, exclude a Scene by unchecking it, remove a Scene or reorder Scenes; Unity builds the Scenes in the listed order [19].

5. **What types of builds does Unity produce?**
   Answer: Unity produces a **release build**, which contains only what is necessary to run the application, and a **development build**, which includes scripting debug symbols and the Profiler to enable debugging and profiling during development [20].

6. **What is the incremental build pipeline?**
   Answer: Unity uses an incremental build pipeline that rebuilds only the artifacts that have changed since the previous build. This speeds up iteration during development [21].

7. **What is an AssetBundle?**
   Answer: An AssetBundle is an archive file used to group together assets (such as models, textures, Prefabs, audio clips or entire Scenes). AssetBundles are platform-specific because Unity builds asset data based on the selected `BuildTarget` [22].

8. **Why would you use AssetBundles?**
   Answer: AssetBundles allow dynamic content delivery by loading assets on demand. They help reduce the initial build size, provide platform-specific assets and improve memory usage by loading only the assets your application needs [23].

9. **How do AssetBundles differ from the Resources folder?**
   Answer: The **Resources** folder bundles assets directly into the main build, while AssetBundles are separate archive files that you load on demand. AssetBundles provide low-level asset management and allow you to control when assets are loaded into memory; they are recommended for larger projects and dynamic content [24].

10. **How do you create platform-specific builds?**
    Answer: In the Build Settings window, select the desired platform in the Platform list. If the platform module is not installed, click **Install**. Then choose **Switch Platforms**; Unity may re-import assets in formats that match the target platform [25].

11. **Why might you compress AssetBundles?**
    Answer: Compressing AssetBundles using LZMA or LZ4 reduces their download size and makes distribution more efficient [26]. Choose the compression method based on whether you need fast streaming (LZ4) or maximum compression (LZMA).

Physics & Collisions

1. **What is a Rigidbody and why is it necessary for physics?**
   Answer: A Rigidbody component allows a GameObject to be affected by simulated gravity and other forces. To simulate physics such as movement, gravity, collisions and joints, you must assign a Rigidbody to your GameObject [9].

2. **How does Unity simulate physics for Rigidbodies?**
   Answer: Unity's physics engine calculates forces, velocity and collisions based on a Rigidbody's properties (mass, drag, gravity, etc.). To move a Rigidbody via physics, use `AddForce` or `AddTorque`. Directly modifying the Transform bypasses the physics simulation and can cause erratic behaviour [9].

3. **What is the difference between kinematic and non-kinematic Rigidbodies?**
   Answer: A non-kinematic (dynamic) Rigidbody is controlled by the physics engine and responds to forces and collisions. A kinematic Rigidbody ignores physics and is moved via scripts by modifying its Transform. Collision events such as `OnCollisionEnter` are only sent if at least one collider has a non-kinematic Rigidbody [8].

4. **When should you use `FixedUpdate` for physics calculations?**
   Answer: Perform physics calculations and apply forces in `FixedUpdate` because it is called at a fixed frame rate, ensuring consistent simulation regardless of frame rate [1].

5. **How do colliders and triggers differ?**
   Answer: A collider gives a GameObject a physical shape for collision detection. If you enable its `isTrigger` property, the collider no longer blocks other objects; instead, it sends trigger events (`OnTriggerEnter`, `OnTriggerExit`) when another collider enters or exits. Use triggers for detection zones without physical interaction.

6. **How do you detect collision events in a script?**
   Answer: Implement `OnCollisionEnter(Collision collision)` in a `MonoBehaviour`. Unity calls this method when the attached collider/rigidbody begins touching another collider/rigidbody; a `Collision` parameter provides contact points and impact velocity. These events require at least one non-kinematic Rigidbody [8].

7. **How do you apply force to a Rigidbody?**
   Answer: Call `rigidbody.AddForce(Vector3 force, ForceMode mode)` to apply a force, or `AddTorque` for rotation. Use `ForceMode.Force` for continuous acceleration and `ForceMode.Impulse` for instant changes. Apply forces in `FixedUpdate` for consistent results [9] [1].

8. **Why should you avoid scaling Rigidbodies at runtime?**
   Answer: Scaling a Rigidbody's Transform can distort its collider and lead to unpredictable collisions. Instead, design your objects at the correct size or adjust mesh scaling before adding a Rigidbody.

9. **How do you create a trigger zone that doesn't impede physics?**
   Answer: Add a collider to a GameObject and set its `isTrigger` property to true. Triggers send `OnTriggerEnter` and `OnTriggerExit` callbacks without blocking objects. At least one of the colliders involved must have a Rigidbody to receive events [8].

Input & Control

1. **How does `Input.GetAxis` work?**
   Answer: `Input.GetAxis` returns the value of a virtual axis (e.g., “Horizontal” or “Vertical”) as a float between −1 and 1. The value is smoothed over time and is frame-rate independent [27].

2. **Why should you multiply `Input.GetAxis` by `Time.deltaTime`?**
   Answer: Multiplying the axis value by `Time.deltaTime` converts a per-frame value into a per-second value. The `GetAxis` documentation shows rotating a player by multiplying `Input.GetAxis("Horizontal")` by `Time.deltaTime` and a speed to ensure frame-rate-independent movement [27].

3. **What are the two input handling systems in Unity?**
   Answer: Unity supports the legacy **Input Manager** and the newer **Input System** package. The Input Manager is the default built-in system; the Input System package provides greater flexibility and broader device support [28].

4. **How does the UI Toolkit handle input?**
   Answer: When you enter Play mode, UI Toolkit automatically creates a default event system that processes input events and sends them to active UI panels. This default event system is used if there is no other active event system and works with both input handling systems [29] [30].

5. **How do you select the active input handling system?**
   Answer: Choose **Edit > Project Settings > Player**, then under Other Settings > Configuration set **Active Input Handling** to **Input Manager**, **Input System Package** or **Both** [31]. If the Input System package is active, you configure input actions in **Project Settings > Input System Package** [32].

6. **What is the EventSystem component used for in Unity UI?**
   Answer: An EventSystem reads the Scene and executes events based on input. UI Toolkit creates a default event system to handle input; in Unity UI (uGUI), you typically add an `EventSystem` GameObject with input modules to process pointer, keyboard and gamepad events [29].

7. **What is the difference between `Input.GetAxis` and `Input.GetAxisRaw`?**
   Answer: `Input.GetAxis` returns a smoothed value between −1 and 1, gradually ramping up and down. `Input.GetAxisRaw` returns the immediate value of the virtual axis (−1, 0 or 1) without smoothing. Use the raw version when you need instantaneous input (e.g., digital input).

8. **How can you handle input for multiple players?**
   Answer: With the Input System package, add a `PlayerInput` component to each player GameObject and assign unique input actions or control schemes. Each `PlayerInput` can be configured to respond to a specific device or player slot.

9. **What is `Time.deltaTime`, and why is it important in input processing?**
   Answer: `Time.deltaTime` is the time in seconds since the last frame. When moving or rotating objects in response to input, multiplying by `deltaTime` ensures a consistent speed regardless of frame rate. The `GetAxis` example multiplies by `Time.deltaTime` to achieve this [27].

UI & Canvas

1. **What is the Canvas in Unity UI?**
   Answer: The Canvas is a special GameObject that defines the area in which all UI elements must reside. If you create a UI element and there is no Canvas in the Scene, Unity automatically creates one and makes your UI element its child [33].

2. **What happens if you create a UI element without a Canvas?**
   Answer: Unity automatically creates a Canvas GameObject with the necessary components and places your UI element as a child of it [33].

3. **What are the Canvas render modes?**
   Answer: Canvas render modes are **Screen Space – Overlay**, **Screen Space – Camera** and **World Space**. Screen Space – Overlay renders the UI in 2D on top of the scene; Screen Space – Camera renders the UI through a camera and allows perspective effects; World Space renders the UI as a 3D object within the scene [34].

4. **How is the draw order of UI elements determined?**
   Answer: Within a Canvas, UI elements are drawn in the order they appear in the hierarchy. Reorder them in the hierarchy to change which elements appear on top [34].

5. **Why do UI elements use `RectTransform` instead of `Transform`?**
   Answer: `RectTransform` extends `Transform` with properties for width, height, pivot and anchors. These properties let UI elements define their position and size in 2D space relative to their parent and to screen dimensions.

6. **What is the difference between UI Toolkit and Unity UI (uGUI)?**
   Answer: **UI Toolkit** is a newer UI framework that uses a default event system and supports modern UI paradigms. It works with the new Input System and uses UXML (markup) and USS (stylesheets). **Unity UI** (uGUI) uses GameObjects, a Canvas and components like `Button` and `Image`, and relies on the `EventSystem` component for input [35].

7. **How can you make a UI element responsive to different resolutions?**
   Answer: Use anchors and pivots in the `RectTransform` to specify how the element scales and positions relative to its parent. Layout components such as `VerticalLayoutGroup`, `HorizontalLayoutGroup` and `ContentSizeFitter` help build flexible layouts that adapt to different screen sizes.

8. **What is a Raycast Target and why is it important?**
   Answer: In Unity UI, graphics have a **Raycast Target** property. When enabled (default), the element can receive pointer events and blocks them from passing through. Disable this property when an image is purely decorative so that it does not block clicks on underlying controls.

9. **How do you handle input events on UI elements?**
   Answer: In uGUI, implement event interfaces such as `IPointerClickHandler`, `IPointerEnterHandler`, etc., or use components like `Button` and assign callback methods to their events (e.g., `Button.onClick`). In UI Toolkit, register callbacks on visual elements (for example, `button.clicked += () => { ... }`).

ScriptableObjects & Data Management

1. **What is a `ScriptableObject`?**
   Answer: `ScriptableObject` is a class that lets you create objects which live independently of GameObjects. You instantiate a ScriptableObject via `CreateInstance`, save it to an asset file and reference it from multiple scenes. ScriptableObjects centralize data accessible across scenes [36].

2. **When should you use ScriptableObjects instead of MonoBehaviours?**
   Answer: Use ScriptableObjects when you need to store data or configuration that is independent of scene objects. Because ScriptableObjects exist as assets, multiple objects can share the same data, reducing duplication and making it easy to adjust game settings in one place [36].

3. **How do you create a ScriptableObject asset?**
   Answer: Define a class that derives from `ScriptableObject` and optionally annotate it with `[CreateAssetMenu]` to make it accessible from the Unity menu. In an editor script or via the context menu, call `CreateInstance<YourScriptableType>()`, then save the instance to an asset file. When referenced by a MonoBehaviour field, Unity will automatically load the asset [36].

4. **How can ScriptableObjects improve memory usage compared to MonoBehaviours?**
   Answer: Because ScriptableObjects live as separate assets, you can store data once and reference it from many places instead of duplicating values across multiple MonoBehaviour instances. This centralization reduces memory usage and simplifies updates [36].

5. **Can ScriptableObjects contain behaviour?**
   Answer: ScriptableObjects can include methods like any C# class, but they should not contain per-frame behaviour. They are designed for data; logic that depends on frame updates should remain in MonoBehaviours.

6. **How do you edit ScriptableObject data in the Inspector?**
   Answer: Use the `[CreateAssetMenu]` attribute on your ScriptableObject class to add a menu item for creating an asset. Expose public fields or use `[SerializeField]` on private fields to make them editable in the Inspector. Select the asset in the Project window to edit its data.

Coroutines & Asynchronous Programming

1. **How do you start a coroutine?**
   Answer: In a `MonoBehaviour`, call `StartCoroutine(IEnumeratorMethod())`. The method must return `IEnumerator` and use `yield` statements to suspend execution [11].

2. **How do you stop a running coroutine?**
   Answer: Use `StopCoroutine(yourCoroutine)` to stop a specific coroutine or `StopAllCoroutines()` to stop all coroutines on the MonoBehaviour. You can store the coroutine reference returned by `StartCoroutine` to stop it later.

3. **Why should you avoid long blocking tasks in coroutines?**
   Answer: Coroutines run on Unity's main thread and do not provide true multi-threading. Performing long-running tasks inside a coroutine will block the main thread and cause frame rate drops. For heavy tasks, use Unity's Job System or C# async/await [11].

4. **What does `yield return null` do in a coroutine?**
   Answer: `yield return null` pauses the coroutine until the next frame, allowing you to spread logic over multiple frames without blocking other scripts.

5. **How can you execute code every fixed timestep inside a coroutine?**
   Answer: Use `yield return new WaitForFixedUpdate()` to suspend execution until the next physics update. This synchronizes the coroutine with `FixedUpdate` for physics-related tasks.

6. **Why should you cache `WaitForSeconds` instances?**
   Answer: Creating a new `WaitForSeconds` inside a coroutine allocates memory. The programming best practices guide recommends caching and reusing yield instructions like `WaitForSeconds` to avoid per-frame allocations and reduce garbage collector overhead [37].

Navigation & Pathfinding

1. **What is a NavMesh?**
   Answer: A NavMesh is a simplified representation of your world geometry used for navigation. It is baked in the editor and allows agents to compute paths around obstacles [38].

2. **How do you generate a NavMesh in Unity?**
   Answer: Mark static geometry as navigation static and bake the NavMesh using Unity's navigation window. Baking analyzes the environment and creates traversable surfaces for agents [38].

3. **How do agents use the NavMesh for pathfinding?**
   Answer: Attach a `NavMeshAgent` component to your character. At runtime, assign a destination (e.g., `agent.destination = targetPosition`), and the agent will compute and follow a path along the NavMesh [39].

4. **What are NavMesh layers and off-mesh links?**
   Answer: NavMesh layers let you classify parts of the NavMesh (e.g., ground vs. water) so that agents can be configured to walk only on specific layers. Off-mesh links are manually placed connections that let agents traverse gaps or special moves (like jumping across a chasm or using a door) [39].

5. **How do you handle moving obstacles with NavMesh?**
   Answer: For moving obstacles, use `NavMeshObstacle` components with carving enabled to carve holes in the NavMesh at runtime. Alternatively, rebake a dynamic NavMesh or use local avoidance on agents.

Animation & Animator

1. **What is an Animation State Machine?**
   Answer: An animation state machine is a flowchart-like system that defines the possible animation states (idle, walk, run, jump, etc.) and the transitions between them. Unity's Mecanim visual layout shows these states and transitions; it allows you to control which animation clips play for your character or GameObject [40].

2. **What is an Animator Controller?**
   Answer: An Animator Controller contains the state machine for a character's animations. It holds states, transitions and parameters, and can include sub-state machines, animation layers and blend trees to manage complex animations [41].

3. **What are animation parameters in an Animator?**
   Answer: Animation parameters are variables (float, int, bool or trigger) that you set from scripts or animation clips. The Animator Controller uses these parameters to decide when transitions occur and to drive blend trees [42].

4. **What are animation transitions?**
   Answer: Animation transitions define how the Animator switches or blends from one state to another. They specify blend durations and conditions (such as parameter thresholds) that activate the transition [43].

5. **What is an animation blend tree?**
   Answer: A blend tree blends between multiple similar animation clips based on a float parameter. For example, it can smoothly blend between walk and run animations as the character's speed parameter changes [44].

6. **What is an animation layer?**
   Answer: An animation layer contains its own state machine controlling part of a model (for example, upper-body motions). Higher layers override lower layers for the body parts they control, allowing you to combine different animation sets [45].

Programming Best Practices & Performance

1. **Why is comparing Unity objects using `== null` different from `ReferenceEquals`?**
   Answer: Unity overrides the equality operators for types derived from `UnityEngine.Object`. As a result, `obj == null` can return `true` even if the underlying C# reference is valid (for example, after the object has been destroyed). Use `ReferenceEquals` or cast to `System.Object` to check for actual null references [46].

2. **Why should you avoid C# finalizers in Unity?**
   Answer: Finalizers run on a separate finalizer thread, while most Unity APIs require the main thread. Finalizers run non-deterministically, may never run and increase garbage collector overhead; throwing exceptions from finalizers can halt your application [47].

3. **What performance risks are associated with frequent memory allocations in Unity?**
   Answer: Allocating memory in per-frame methods (like `Update`) increases garbage collector overhead and can cause frame rate spikes. Cache and reuse lists, use non-allocating versions of methods and cache yield instructions to reduce allocations [37].

4. **Why should you cache components such as `GetComponent` results?**
   Answer: `GetComponent` is relatively expensive and allocates memory. Call it in `Awake` or `Start`, store the returned reference and reuse it rather than calling it repeatedly in `Update` [48].

5. **When should you use `Destroy` versus `DestroyImmediate`?**
   Answer: Use `Destroy` to mark an object for destruction at the end of the current frame

(runtime). `DestroyImmediate` destroys objects instantly and is intended for editor scripts. Using it in gameplay code can cause errors or permanently remove assets [7].

6. **Why should you avoid using LINQ in performance-critical code?**
   Answer: LINQ queries allocate memory and can be slow. The programming best practices guide advises avoiding LINQ in runtime code, especially in `Update` or other hot paths [37].

Additional Topics

1. **How should you move a Rigidbody smoothly without using `Transform.Translate`?**
   Answer: Use `Rigidbody.MovePosition` or apply forces rather than changing the Transform. The physics manual notes that you should use Rigidbody methods (like `AddForce` and `AddTorque`) instead of modifying the Transform directly to avoid unwanted behaviour [9].

2. **What is the difference between `Awake` and `OnEnable`?**
   Answer: `Awake` is called when the script instance is being loaded; it runs once per object lifetime and is used for initialization. `OnEnable` is called every time the object becomes enabled. If a GameObject starts enabled, Unity calls `Awake` and then `OnEnable` before `Start`. `Awake` always runs before `Start` [3].

3. **What is the advantage of using `Addressables` over manually managing AssetBundles?**
   Answer: The AssetBundle documentation recommends using the **Addressables** package to define and build AssetBundles. Addressables provide a higher-level API for managing AssetBundles, automate asset loading and handle dependencies, whereas using the low-level AssetBundle API requires more manual bookkeeping [49].

4. **What are the benefits of using a development build during testing?**
   Answer: A development build includes debug symbols and the Profiler, allowing you to attach debuggers, inspect variables and perform deep profiling. These options are not included in release builds [50].

5. **How can you reduce garbage collector overhead related to coroutines?**
   Answer: Cache yield instructions like `WaitForSeconds` and reuse them instead of instantiating new ones each time. The best practices guide explicitly suggests caching `WaitForSeconds` to avoid per-frame allocations and reduce GC overhead [37].

6. **How do you centralize configuration data that is accessible across Scenes?**
   Answer: Use ScriptableObjects to store configuration data. Because ScriptableObjects live independently of Scenes and can be loaded automatically when referenced by a MonoBehaviour field, they provide a central place for settings and reduce duplication [36].

7. **How does `Time.timeScale` influence coroutine waits like `WaitForSeconds`?**
   Answer: `WaitForSeconds` multiplies the specified wait time by `Time.timeScale`; if `Time.timeScale` is 0 (for example, when the game is paused), the wait time becomes infinite. Changing `Time.timeScale` can therefore speed up or slow down coroutine waits [10].

8. **What is `Time.deltaTime`, and why is it used with `Input.GetAxis`?**
   Answer: `Time.deltaTime` is the duration of the last frame in seconds. Multiplying input values (from `Input.GetAxis`) by `Time.deltaTime` converts per-frame values into

per-second values, ensuring movement and rotation remain consistent across different frame rates [27].

9. **Why should you avoid performing expensive operations in `Update`?**
Answer: Code inside `Update` runs every frame. Performing expensive operations there (like repeated `GetComponent` calls or allocating memory) can cause slowdowns. The best practices guide advises doing such work in `Awake` or `Start` and caching results [37].

10. **What is the purpose of layering UI in a Canvas?**
Answer: UI elements within a Canvas are drawn in hierarchy order. Organizing them in layers (e.g., background, content, overlay) makes it easy to manage their draw order by reordering them in the hierarchy [34].

11. **How do you handle dynamic loading of Scenes or assets at runtime?**
Answer: Use AssetBundles (or the Addressables system) to load Scenes or assets on demand. AssetBundles allow you to keep assets out of memory until they are needed and provide dynamic content delivery [23] [49].

Project Prompt: Build a Unity Interview Flashcard Website

Below is a comprehensive prompt you can give to a coding agent (for example, Codex or Copilot in Visual Studio Code). The prompt specifies the tech stack, product requirements, user stories and design guidelines needed to create a local web application that displays the flashcards defined above and allows you to add more.

Goal

Create a responsive web application that serves as a study tool for Unity developer interviews. The application displays flashcards containing questions and answers (initially the 100 cards defined above) and allows users to add, edit and delete cards. It should run on a local server so that you can develop and test it offline.

Recommended Tech Stack

- **Backend:** Node.js with Express.js for the server API.
- **Frontend:** React with TypeScript for the UI. Use functional components and hooks. Feel free to use a component library like **Material-UI** or **Tailwind CSS** for styling if it speeds development, but plain CSS modules are acceptable.
- **Database/Storage:** Persist the flashcards in a local JSON file (`cards.json`) for simplicity. When the server starts, load the JSON file into memory; when cards are added/edited/deleted, update the file. Alternatively, implement an in-memory store and a simple file-based persistence layer.
- **Build Tools:** Use **Vite** or **Create React App** for the React build. Use **nodemon** for auto-reloading the server during development.

Product Requirements (PRD)

1. **Card Data Model**
2. Each card has an `id` (UUID), `question` (string), `answer` (string), `sources` (array of citation strings) and `createdAt` timestamp.

3. Store the initial 100 cards by loading them from the provided Markdown or JSON file.

4. **API Endpoints**

5. `GET /api/cards` – return a list of all cards.
6. `GET /api/cards/:id` – return a single card by id.
7. `POST /api/cards` – add a new card (provide JSON body with `question`, `answer` and optional `sources`). Return the created card.
8. `PUT /api/cards/:id` – update an existing card.
9. `DELETE /api/cards/:id` – remove a card by id.

10. Use Express middleware to parse JSON and handle errors. Persist changes to the `cards.json` file.

11. **Frontend Features**

12. Display all cards in a grid of **flip cards**. The front side shows the question; clicking flips the card to reveal the answer and citations. Use CSS transitions for the flip animation.
13. Provide a search bar to filter cards by keywords in the question or answer. Search should update results in real time.
14. Allow adding a new card via a form (fields for question, answer and sources). Validate that question and answer are not empty. After adding, the new card should appear in the grid.
15. Allow editing and deleting existing cards. Editing opens a modal or inline form. Deleting prompts for confirmation.
16. Persist user modifications by calling the appropriate API endpoints.
17. Responsive design: on large screens, display cards in multiple columns; on small screens, stack them vertically.

18. Provide a dark/light theme toggle using CSS variables or a UI library's theming system. Remember the user's preference using local storage.

19. **User Stories & Tickets**

Ticket ID	User Story	Acceptance Criteria
T1	As a developer, I want to set up the project repository with a Node/Express backend and a React frontend.	The repository contains a server folder with Express code and a client folder with a React app. Running `npm install` installs dependencies. Running `npm run dev` concurrently starts the server and the React development server.
T2	As a user, I want to see a list of flashcards so that I can study questions.	Navigating to `/` shows a grid of cards with questions visible. Clicking a card flips it to show the answer and sources.
T3	As a user, I want to search cards by keyword so that I can find specific topics.	Typing in the search bar filters the card grid in real time by matching keywords in questions or answers. Clearing the search shows all cards again.

Ticket ID	User Story	Acceptance Criteria
T4	As a user, I want to add a new flashcard.	There is a button to open an “Add Card” form. Submitting the form with valid data adds the card via `POST /api/cards`, updates the `cards.json` file and shows the new card in the grid. Validation errors display under the appropriate fields.
T5	As a user, I want to edit an existing flashcard.	Each card has an edit button. Clicking it opens a form populated with the card’s data. Submitting sends a `PUT /api/cards/:id` request, updates the file and updates the card in the grid.
T6	As a user, I want to delete a flashcard.	Each card has a delete button. Clicking it prompts for confirmation. Confirming sends `DELETE /api/cards/:id`, removes the card from `cards.json` and updates the UI.
T7	As a user, I want the application to remember my theme preference.	Toggling the dark/light theme stores the preference in `localStorage`. Reloading the page retains the selected theme.

1. **Design Guidelines**
2. **Layout:** Use a responsive grid. On desktop, cards can be displayed in three or four columns; on mobile, they stack in a single column.
3. **Card Design:** Each card should have a front (question) and back (answer + citations). Use a subtle shadow and border radius. The flip animation should be smooth and accessible (consider providing an alternative like a “Show Answer” button for screen readers).
4. **Typography:** Use a clean, readable font (e.g., Roboto or Inter). Headings should stand out; answers can be slightly smaller.
5. **Colours:** Choose a neutral light theme with a contrasting primary colour for buttons and highlights (e.g., blue). For dark mode, invert the background and text colours while maintaining sufficient contrast.
6. **Icons:** Use icons to represent editing and deleting actions (e.g., pencil and trash bin) and a plus icon for adding cards.
7. **Accessibility:** Ensure all interactive elements are keyboard navigable. Provide `aria-labels` on buttons. Ensure colour contrast meets WCAG guidelines.
8. **Non-Functional Requirements**
9. **Performance:** The application should load the initial 100 cards quickly. Lazy-load images or heavy assets if added later.
10. **Extensibility:** The code should be organized into reusable components and modules to facilitate adding features (for example, user accounts or flashcard categories).
11. **Testing:** Provide unit tests for the API endpoints using Jest or a similar testing framework.
12. **Documentation:** Include a `README.md` with setup instructions, project description and future improvement ideas.

Developer Notes for the Agent

1. **Setup:** Initialize a new Node.js project in the server directory and a separate React project in the client directory. Use TypeScript for type safety. Use `concurrently` to run both projects with a single command (`npm run dev`).
2. **Data File:** Save the initial 100 cards from this document into a `cards.json` file. Each card should include the `id`, `question`, `answer` and `sources` (an array of citation strings). Load this file on server startup. When modifying cards, write the updated array back to the file.
3. **Server Code:** Implement Express routes as described above. Use the `fs` module to read/write the JSON file. Handle errors gracefully (e.g., return 404 if a card is not found). Apply CORS middleware so that the React frontend can communicate with the API.
4. **Client Code:** Fetch data from the API using `fetch` or Axios. Maintain cards in React state. Use React Router to manage routes if necessary (for example, a dedicated page for editing). Implement form validation and display error messages. Use `useEffect` to load cards on page load. Use `useState` and `useReducer` to manage complex state if needed. Implement the flip animation with CSS (e.g., transform: rotateY(180deg)).
5. **Testing:** Write Jest tests for server routes (e.g., `GET` returns proper data, `POST` adds a card). Optionally write React testing library tests for UI components (e.g., adding a card updates the list).
6. **Future Enhancements:** Suggest features such as categorizing cards, importing/exporting decks, syncing data with a cloud backend, or implementing spaced repetition algorithms.

How to Use This Prompt

Provide the coding agent with this entire document (the 100 flashcards and the PRD prompt). Instruct the agent to:

1. Parse the flashcard data and convert it into a JSON file.
2. Scaffold the backend and frontend using the specified stack.
3. Implement the tickets and design guidelines described above.
4. Run the project locally and ensure that the cards load correctly and that you can add/edit/delete cards.

After following these steps, the agent should deliver a functioning local web application that meets the requirements. You can then add more questions by updating the JSON file or using the “Add Card” feature.

---

[1] Unity - Scripting API: MonoBehaviour.FixedUpdate()
https://docs.unity3d.com/530/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html

[2] Unity - Scripting API: MonoBehaviour.Update()
https://docs.unity3d.com/6000.3/Documentation/ScriptReference/MonoBehaviour.Update.html

[3] Unity - Scripting API: MonoBehaviour.Awake()
https://docs.unity3d.com/530/Documentation/ScriptReference/MonoBehaviour.Awake.html

[4] [5] Unity - Scripting API: MonoBehaviour.Start()
https://docs.unity3d.com/6000.3/Documentation/ScriptReference/MonoBehaviour.Start.html

[6] Unity - Scripting API: MonoBehaviour.LateUpdate()
https://docs.unity3d.com/560/Documentation/ScriptReference/MonoBehaviour.LateUpdate.html

[7] Unity - Scripting API: Object.DestroyImmediate
https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Object.DestroyImmediate.html

[8] Unity Script Reference – MonoBehaviour.OnCollisionEnter
https://docs.unity3d.com/352/Documentation/ScriptReference/MonoBehaviour.OnCollisionEnter.html

[9] Unity - Manual: Introduction to rigid body physics
https://docs.unity3d.com/6000.3/Documentation/Manual/RigidbodiesOverview.html

[10] Unity - Scripting API: WaitForSeconds
https://docs.unity3d.com/2017.2/Documentation/ScriptReference/WaitForSeconds.html

[11] Unity - Manual: Write and run coroutines
https://docs.unity3d.com/6000.3/Documentation/Manual/Coroutines.html

[12] Unity - Manual: The GameObject-Component Relationship
https://docs.unity3d.com/510/Documentation/Manual/TheGameObject-ComponentRelationship.html

[13] Unity - Manual: Introduction to components
https://docs.unity3d.com/6000.3/Documentation/Manual/Components.html

[14] Unity - Manual: Transforms
https://docs.unity3d.com/2022.3/Documentation/Manual/class-Transform.html

[15] Unity - Manual: Tags and Layers
https://docs.unity3d.com/2021.3/Documentation/Manual/class-TagManager.html

[16] Unity - Manual: Prefabs
https://docs.unity3d.com/2020.1/Documentation/Manual/Prefabs.html

[17] [18] Unity - Manual: Introduction to scenes
https://docs.unity3d.com/6000.3/Documentation/Manual/CreatingScenes.html

[19] [20] [21] [25] [50] Unity - Manual: Build Settings
https://docs.unity3d.com/2023.2/Documentation/Manual/BuildSettings.html

[22] [23] [24] [26] [49] Unity - Manual: Introduction to AssetBundles
https://docs.unity3d.com/6000.3/Documentation/Manual/AssetBundlesIntro.html

[27] Unity - Scripting API: Input.GetAxis
https://docs.unity3d.com/560/Documentation/ScriptReference/Input.GetAxis.html

[28] [29] [30] [31] [32] [35] Unity - Manual: Runtime UI event system and input handling
https://docs.unity3d.com/6000.3/Documentation/Manual/UIE-Runtime-Event-System.html

[33] [34] Unity - Manual: Canvas
https://docs.unity3d.com/2020.1/Documentation/Manual/UICanvas.html

[36] Unity - Scripting API: ScriptableObject
https://docs.unity3d.com/6000.3/Documentation/ScriptReference/ScriptableObject.html

[37] [46] [47] [48] Unity - Manual: Unity programming best practices
https://docs.unity3d.com/Manual/programming-best-practices.html

[38] [39] Unity - Navmesh and Pathfinding (Pro only)
https://docs.unity3d.com/400/Documentation/Manual/NavmeshandPathfinding.html

[40] [41] [42] [43] [44] [45] Unity - Manual: Animation State Machine
https://docs.unity3d.com/6000.3/Documentation/Manual/AnimationStateMachines.html