Online Documentation available at https://onejs.com/docs

Overview

OneJS lets you use JavaScript in Unity, for both runtime and editor. It focuses on UI scripting but works for anything you'd usually handle with JS.

It integrates directly with Unity's UI Toolkit, a shader-based, retained-mode UI system inspired by web tech. UI Toolkit already offers web concepts like DOM and CSS, and OneJS complements it by having a JavaScript backend (QuickJS, V8, or NodeJS via Puerts), so you can build UIs using familiar toolings like TypeScript, Preact, Tailwind, and ESBuild.

Why use OneJS

- **Lightning-fast iteration**: Change your UI code and see results instantly. No more domain reloads.
- **Familiar tooling**: Use TypeScript, (P)React, Tailwind and other web techs you already know and love.
- **Broaden the talent pool**: Studios can tap into web devs for UI work.
- **Unity-native**: Works directly with UI Toolkit - no browser, no overhead.
- **Desktop and Mobile**: Tested on Windows, Mac, iOS, and Android.
- **Powerful scripting**: Give your players the ability to mod your game with TypeScript and JSX.

JavaScript pretty much runs the show when it comes to modern UI, thanks to all the frameworks and tools that make dev work smoother across the board. It's constantly evolving, super flexible, and backed by a huge community. What we're aiming for is to bring as much of that web tech into Unity as we can, so we get the best of both worlds.

Limitations

OneJS uses Unity's UI Toolkit as its DOM layer, so keep in mind that UI Toolkit only supports a limited set of DOM and CSS features. Here's what's not supported (yet):

- Canvas (but there's a Vector API workaround)
- SVG (partially supported now, but still room to grow)
- Complex CSS selectors (i.e. Tailwind's `space-` classes won't work since `> * + *` isn't supported)
- CSS animations and keyframes (on UI Toolkit's roadmap)

👉 UI Toolkit Roadmap

WebGL support isn't in yet, but it's coming in V2. It'll follow Puerts' WebGL workflow and run on the browser's JS engine. Stay tuned.

Working with Constraints

Unity's UI Toolkit has its quirks and missing bits compared to standard web tools, but that's kind of the point. These constraints are part of game dev DNA. They push you to get creative, optimize smart, and solve problems in ways you might not have considered otherwise. Everyone's working with the same limitations, so it levels the playing field and opens the door to unique solutions.

That said, UI Toolkit is already a solid pick. It's built on proven web standards, giving your game UIs a stable, future-proof foundation. And of course, OneJS is right here to supercharge the whole experience.

Getting Started

Requirements

- Unity.Mathematics
- Unity Version 2021.3+ (for stable UI Toolkit)
- Unity Version 2022.1+ (if you need to use UI Toolkit's Vector API)

Live Reload requires `Run in Background` to be turned ON in Player settings (under Resolution and Presentation). Depending on your Unity version or platform, this may or may not be ON by default. So it never hurts to double-check.

Installation

You can use any **one** of the following methods.

- Download and import from Asset Store.
- Clone the repo directly into your Unity project's Assets folder.

```
git clone https://github.com/Singtaa/OneJS.git
```

- Clone the repo anywhere on your machine, and use `Install package from disk` from Package Manager.

Quick Start

- Drag and drop the `ScriptEngine` prefab onto a new scene.
- Enter Play mode.

If you see `[index.tsx]: OneJS is good to go` in the console, then OneJS is all set. The first time ScriptEngine runs, it'll setup the working directory with some default files. Feel free to read and tweak the various setting files (tsconfig.json, esbuild.mjs, postcss.config.js etc) as you see fit.

> By default, OneJS uses `{ProjectDir}/App` as its working directory (NOTE: `{ProjectDir}` is not your `Assets` folder; it is one level above the `Assets` folder). So, you can safely check the `App` folder into Version Control. When building for standalone, the scripts from `{ProjectDir}/App` will be automatically bundled up and be extracted to `{persistentDataPath}/App` at runtime. (The folder name "App" is also configurable on the ScriptEngine component.)

Make sure you have Typescript installed on your system (i.e. via `npm install -g typescript`)

- Open `{ProjectDir}/App` with VSCode.
- Run `npm run setup` in VSCode's terminal.
- Use `Ctrl + Shift + B` or `Cmd + Shift + B` to start up all 3 watch tasks: `esbuild`, `tailwind`, and `tsc`.
- In VSCode, save the following into the index.tsx file.

index.tsx

```
import { parseColor } from "onejs/utils"
import { Camera, Collider, CollisionDetectionMode, Color, GameObject, MeshRenderer,
PhysicMaterial, Physics, PrimitiveType, Rigidbody, Vector3 } from "UnityEngine"

// Make a plane
const plane = GameObject.CreatePrimitive(PrimitiveType.Plane)
plane.GetComp(MeshRenderer).material.color = Color.yellow

// Make a sphere
const sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere)
sphere.GetComp(MeshRenderer).material.color = parseColor("FireBrick")
sphere.transform.position = new Vector3(0, 5, 0)

// Adjust camera
Camera.main.transform.position = new Vector3(9, 4, -8)
Camera.main.transform.LookAt(new Vector3(0, 1, 0))

// Add rigidbody and physics material to make a bouncing ball
Physics.gravity = new Vector3(0, -20, 0) // -9.8 is too "floaty", -20 makes things
slightly more realistic
let rb = sphere.AddComp(Rigidbody)
rb.collisionDetectionMode = CollisionDetectionMode.Continuous
let pm = new PhysicMaterial() // use "PhysicsMaterial" in Unity 6+ (note the extra s)
pm.bounciness = 0.8
plane.GetComp(Collider).material = pm
sphere.GetComp(Collider).material = pm
```

With Unity still in Playmode, you'll see a bouncing sphere. As you can see, the code is very similar to what you'd normally write in C#. Thanks to Typescript, we get all the benefits of auto-completions, auto-imports, type-checking, etc. Feel free to play around with the code and see how OneJS live-reloads your changes in Unity.

NOTE: All .NET/C# namespaces and classes are accessible through the `cs.` global namespace in JavaScript. This approach can sometimes be more convenient than using import statements. For example:

```
import { GameObject } from "UnityEngine";

const go = new GameObject("My Obj");
```

can also be written as:

```
const go = new CS.UnityEngine.GameObject("My Obj");
```

Alrighty, next up—check out Ult Meter for a quick tutorial on using Preact in OneJS + how to get C# and JS talking to each other.

UI Workflow

In general, your UI code should depend on your core game logic. But your core game logic should not even be aware of the existence of your UI code. In other words, your JS code will be calling stuff from your C# code, but never the other way around. This one-directional dependency makes everything easy to maintain. You can modify the UI as much as you like without risking any unintended side effects in your game logic.

The best way to implement this is via C# events (or similar pub/sub mechanisms). Whenever your UI needs something, you can have your core game logic fire an event. And in your JS code, you can subscribe to C# events by appending “add_” and “remove_” to the event name.

Here’s a quick example:

TreasureChestSpawner.cs

```
// You can add this MonoBehaviour as 'spawner' in ScriptEngine's Globals list.
public class TreasureChestSpawner : MonoBehaviour {
    // Fired when a chest is spawned in the scene
    public event Action OnChestSpawned;

    ...
}
```

index.js

```
// `spawner` will now be available in your JS code as a global variable.
spawner.add_OnChestSpawned(onChestSpawned)

function onChestSpawned() {
    log("yay!")
}

// Event handler can be removed via `spawner.remove_OnChestSpawned(onChestSpawned)`
```

definitions.d.ts

```
// Optional TS Definition
declare namespace CS {
    namespace MyGame {
        export interface ChestSpawner {
            add_OnChestSpawned(handler: Function): void
            remove_OnChestSpawned(handler: Function): void
        }
    }
}

declare const spawner: CS.MyGame.ChestSpawner;
```

Reducing Boilerplates

C# events need to be properly cleaned up from the JS/Preact side. Compound that with the “add_” and “remove_” event syntax, you usually end up with a bit of verbose boilerplate. This is where you can make use of OneJS’s `useEventfulState()` function to reduce the following boilerplate:

index.tsx

```
// Assuming you've added OneJS.Samples.SampleCharacter as 'sam' to the Globals list
const App = () => {
    const [health, setHealth] = useState(sam.Health)
    const [maxHealth, setMaxHealth] = useState(sam.MaxHealth)

    useEffect(() => {
        sam.add_OnHealthChanged(onHealthChanged)
        sam.add_OnMaxHealthChanged(onMaxHealthChanged)

        onejs.subscribe("onReload", () => {
            // Cleaning up for Live Reload
            sam.remove_OnHealthChanged(onHealthChanged)
            sam.remove_OnMaxHealthChanged(onMaxHealthChanged)
        })

        return () => {
            sam.remove_OnHealthChanged(onHealthChanged)
            sam.remove_OnMaxHealthChanged(onMaxHealthChanged)
        }
    }, [])

    function onHealthChanged(v: number): void {
        setHealth(v)
    }

    function onMaxHealthChanged(v: number): void {
        setMaxHealth(v)
    }

    return <div>...</div>
}
```

To just:

index.tsx

```
const App = () => {
    const [health, setHealth] = useEventfulState(sam, "Health")
    const [maxHealth, setMaxHealth] = useEventfulState(sam, "MaxHealth")

    return <div>...</div>
}
```

`useEventfulState()` will take care of the event subscription and cleanup for you automatically.

> NOTE: `useEventfulState(obj, "Health")` assumes the C# `obj` has a property named “`Health`” and an event named “`OnHealthChanged`” (both of which can also

be auto-generated by the Source Generator below).

C# Source Generator

You may also use the EventfulProperty attribute to further reduce boilerplates on the C# side and turn this:

Character.cs

```
public class Character : MonoBehaviour {
    public float Health {
        get { return _health; }
        set {
            _health = value;
            OnHealthChanged?.Invoke(_health);
        }
    }

    public event Action<float> OnHealthChanged;

    public float MaxHealth {
        get { return _maxHealth; }
        set {
            _maxHealth = value;
            OnMaxHealthChanged?.Invoke(_maxHealth);
        }
    }

    public event Action<float> OnMaxHealthChanged;

    float _health = 200f;
    float _maxHealth = 200f;
}
```

Into just this:

Character.cs

```
public partial class Character : MonoBehaviour {
    [EventfulProperty] float _health = 200f;
    [EventfulProperty] float _maxHealth = 200f;
}
```

Note the `partial` keyword being used on the class declaration. The corresponding getters, setters, and events will be auto-created by OneJS's Source Generators .

onejs.subscribe

OneJS provides a handy `onejs.subscribe()` function that allows you to subscribe to C# events from your JS code. A key benefit of this function is that it automatically handles event cleanup during LiveReload, helping you avoid memory leaks or orphaned event handlers.

Usage is like this:

index.tsx

```
const unsubscribe = onejs.subscribe("onReload", () => {
    console.log("Engine Reloaded!")
})

const unsubscribe = onejs.subscribe(sam, "OnHealthChanged", () => {
    console.log("Sample Character's health changed!")
})
```

Deployment

OneJS has been tested on Windows, Mac, iOS, and Android. WebGL support is planned for the future. Generally, no extra steps are needed when building your standalone player with an OneJS project. Everything is set up so you can follow the standard Unity build workflow, and it will work seamlessly.

Backends

By default, OneJS uses `QuickJS` due to its small footprint. You can quickly switch the backend (`QuickJS`, `V8`, or `NodeJS`) using `npm run switch`, but make sure the Unity editor is closed before doing so.

Both `QuickJS` and `V8` are tested on mobile, but V8 is definitely the more stable and performant choice for older devices and OS's. Be sure to set the correct architecture in Player Settings before building (e.g., check both ARMv7 and ARM64).

Bundler

The Bundler component (on ScriptEngine) takes care of 3 things:

- Setting up OneJS for the first time by copying essential files into your Working Directory when the files are not found.
- Bundling your scripts at buildtime (into `bundle.tgz`).
- Extracting the bundle at runtime for your standalone Player.

The bundling and extraction process are both automatic so you don't need to worry about it too much. One thing worth mentioning is that the `outputs.tgz` file can be zero'ed out via the context menu so you don't need to keep checking it into git, for example.

PuerTS and IL2CPP

If you're running into IL2CPP build errors and you're not familiar with `xIl2cpp` (which is PuerTS's IL2CPP optimization that's turned on by default in newer versions), it's a good idea to just turn it off. You can do that by adding `PUERTS_DISABLE_IL2CPP_OPTIMIZATION` to your `Player Settings > Other Settings > Scripting Define Symbols`.

link.xml

AOT Platforms and IL2CPP builds will strip all your unused C# code. So for all the classes you'd like to call dynamically from Javascript, you'd need to preserve them. link.xml will do the job. Here's an example:

```
<linker>
    <assembly fullname="mscorlib" preserve="all" />
    <assembly fullname="OneJS" preserve="all" />
    <assembly fullname="UnityEngine.CoreModule" preserve="all" />
    <assembly fullname="UnityEngine.PhysicsModule" preserve="all" />
    <assembly fullname="UnityEngine.TextRenderingModule" preserve="all" />
    <assembly fullname="UnityEngine.UIElementsModule" preserve="all" />
    <assembly fullname="UnityEngine.IMGUIModule" preserve="all" />
    <assembly fullname="Unity.Mathematics" preserve="all" />
</linker>
```

Folks tend to run into problems when dealing with link.xml for the first time. So here are some tips.

- The file name has to be `link.xml`, not `linker.xml` or `links.xml`.
- Make sure the extension is `.xml` and not something like `.xml.txt`.
- The root xml node has to be named `<linker>`.