VERSION v2.0.x

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 three methods.

Download and import from Asset Store.

Get private repo access (`/collab` cmd in Discord), and clone the repo directly into your Unity project's Assets folder.

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.

> 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. Refer to the Deployment page for more details on that. (The 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.

TSX

```
import { parseColor } from "onejs/utils"
import { Camera, Collider, CollisionDetectionMode, Color, GameObject, MeshRenderer, Physi

// 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 slight
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.

Move on to Ult Meter for a tutorial on interop between C# and JS.

Why OneJS?

> **Lightning-fast iteration**: Change your UI code and see results instantly. No more waiting for compilation or domain reloads.
>
> **Familiar tooling**: Use TypeScript, (P)React, Tailwind and other web technologies you already know and love.
>
> **Unity-native**: Works directly with UI Toolkit - no browser, no overhead.
>
> **Desktop and Mobile**: Tested on Windows, Mac, iOS, and Android. WebGL support also in the works.
>
> **Powerful scripting**: Give your players the ability to mod your game with TypeScript and JSX.

Ultimately, our goal is to integrate as many web technologies into Unity as possible, without relying on a browser.

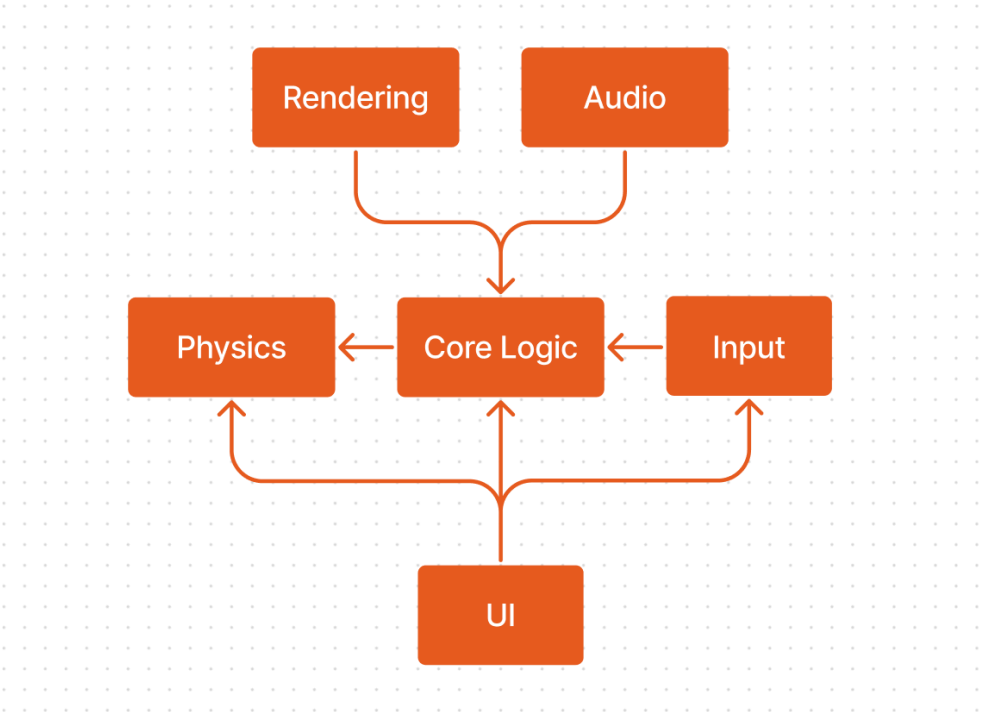
Fast Iteration Speed

Having fast iteration speed is a special trait to UI Development. And here's why.

Clear boundaries between different systems, such as Core Logic, Rendering, Audio, Physics, Input, and UI, are essential. This separation allows for better management of dependencies and minimizes the risk of changes in one system affecting others.

UI's goal is to access and reflect the state of the game and also to provide ways for players to alter the game state. Thus, your UI code can depend on basically all the other systems in your game, but not vice-versa (no other systems should depend on UI).

Changes to your UI code should not have any dependency effect on any other system. This allows for rapid iteration speed without consequences!



On the other hand, any changes made to the public APIs of Physics, for example, will have a cascading effect on Core Logic, Input, and UI.

(Good use of Dependency Inversion and Events (or any Pub/Sub) mechanisms will make sure your other systems don't need to know anything about UI)

Working with Constraints

Unity's UI Toolkit has its rough edges and lacks features compared to standard web environments. However, these constraints echo a fundamental aspect of game development. They force developers to think creatively, find clever solutions, and optimize their code and designs. These limitations level the playing field, requiring every developer to navigate them and find unique opportunities.

Compared to alternatives, UI Toolkit is already a superior choice. Based on proven web standards, it'll ensure a solid foundation for future game UIs. And of course, OneJS is here to supercharge your dev experience.

C#-JS Workflow

OneJS uses Puerts to glue C# and TS together. You can access any .Net Assemblies/Namespace/Classes from Javascript:

TS

```
var { GameObject, MeshRenderer, Color } = CS.UnityEngine
var sphere = GameObject.CreatePrimitive(0)
sphere.GetComp(MeshRenderer).material.color = Color.red
```

With Type Definitions, the same code can be written in Typescript as:

TS

```
import { GameObject, PrimitiveType, MeshRenderer, Color } from "UnityEngine"

const sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere)
sphere.GetComp(MeshRenderer).material.color = Color.red
```

To pass serialized Unity objects to TS, you can use the `Globals` list on ScriptEngine. Please see the Ult Meter tutorial for using it in practice.

UI Dataflow

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 TS code will be calling stuff from your C# code, but never the other way around. This one-directional dependency makes everything easy to maintain.

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 TS code, you can subscribe to C# events by appending "add_" and "remove_" to the event name.

Here's a quick example:

CS

```
// Example MonoBehaviour component
public class TreasureChestSpawner : MonoBehaviour {  // You should add this object as 'sp
    public event Action OnChestSpawned; // Fired when a chest is spawned in the scene, fo

    ...
}
```

TS

```
spawner.add_OnChestSpawned(onChestSpawned)

function onChestSpawned() {
```

```
    log("yay!")
}

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

TS

```
// Example 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;
```

You can see this workflow in more detail from the Fortnite UI sample.

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:

TS

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

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

        onEngineReload(() => {  // Cleaning up for Live Reload
            charman.remove_OnHealthChanged(onHealthChanged)
            charman.remove_OnMaxHealthChanged(onMaxHealthChanged)
        })

        return () => {  // Cleaning up for side effect
            charman.remove_OnHealthChanged(onHealthChanged)
            charman.remove_OnMaxHealthChanged(onMaxHealthChanged)
```

```
        }
    }, [])

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

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

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

To just:

TS

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

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:

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 Source Generators.

New in V2

Puerts

OneJS V2 has changed from using Jint to Puerts. Jint is a pure .Net JS interpreter so it has great interop ergonomics (i.e. it's one of the few interpreters that supports operator overloading) and a small build size (~8MB). Downside of Jint is the slow interop performance, especially when it comes to dealing with deep recursive functions (like the `diff` functions in Preact).

Puerts provides 3 native backends (QuickJS, V8, and NodeJS) and is much more performant than Jint (roughly 20x to 100x). Its static wrappers and blittable optimizations can also eliminate GC allocations during interop. Having 3 backends to choose from is also nice. Use QuickJS for its small build size (~20MB). Use V8 for raw speed. Use NodeJS for stuff like node modules and `WebAssembly`.

(Use `npm run switch` to quickly change the backend. Just make sure the Unity editor is closed before doing so.)

ESBuild

OneJS V2 now uses an ESBuild workflow, bundling everything into a single file for easier deployment. It automatically handles node modules and has experimental Custom Editor/Inspector support.

ScriptEngine

Due to the switching to Puerts, some of the interop settings for ScriptEngine has changed. You can directly expose Unity/.Net objects as global variables. It's similar to the old way except you don't need to `require()` anymore on the JS side. You can just access the global name directly.

Type definition conversion has been removed because Puerts does `.d.ts` generation automatically. OneJS provides some helper methods for this via the `DTSGen` class.

OneJS Limitations

OneJS uses Unity's UI Toolkit as its DOM layer. And UI Toolkit only contains a subset of Dom and CSS features. The following are currently not supported.

- Canvas (but there's the Vector API)
- SVG (on UI Toolkit's roadmap)
- Complex CSS selectors
- CSS animation and keyframes (on UI Toolkit's roadmap)

UI Toolkit Roadmap

Note on Past Limitations

For OneJS V1, node modules and custom Editor UI were not officially supported. For V2, they are. With the NodeJS backend, you can install and use packages as usual with `npm install`. Custom Editor UI support is currently experimental in V2. You can take a look at `esbuild.mjs` for more details.

WebGL

We haven't implement this yet, but WebGL will be supported for V2 at some point. It will use Puerts' WebGL workflow and piggy-back on browser's own JS engine.

Upgrading OneJS

You can upgrade OneJS as usual from Unity's Package Manager or through the OneJS private repo. For the latter, you can either just `git clone` into your project's Assets folder, or put the repo anywhere on your machine and use Package Manager's `Install from disk` option.

If any problem persists, please come to our Discord Server for assistance.

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 (v2) 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. V8 is more performant and has a much higher recursion limit than QuickJS.

Both QuickJS and V8 are tested on mobile devices. 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 `outputs.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.

Live Reload, on or off

For production deployment where you don't need Live Reload, remember to turn it off by unchecking the "Enable For Standalone" option on the Runner component (if you had it on

before). This will save some CPU cycles for you since, in most cases, Live Reload will be unnecessary for Standalone.

link.xml for AOT Platforms and IL2CPP

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:

XML

```xml
<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>`.

Runtime USS

OneJS allows you to load USS strings dynamically at runtime via `document.addRuntimeUSS()`.

TS

```ts
import { h, render } from "preact"
import { useEffect, useState } from "preact/hooks"

const App = () => {
    const [text, setText] = useState("#App {\n    padding: 100;\n}")

    const setUSS = () => {
        document.clearRuntimeStyleSheets()
        document.addRuntimeUSS(text)
```

```
    }

    useEffect(setUSS, [])

    return <div name="App">
        <textfield multiline={true} onValueChanged={(e) => setText(e.newValue)} value={te
        <div><label text="Lorem Ipsum" /></div>
        <button onClick={setUSS} text="Set USS"></button>
    </div>
}
```

Caveat

Runtime USS was made to work by using and modifying some internal Unity C# code. At the time, it was meant as a quick hack because Unity had said they were working on their own runtime-friendly Stylesheets. But that never came to be. So, we will be implementing our own custom stylesheets soon from scratch.

If you work for a company and are worried about potential license issue, you have 3 options currently:

> Wait for us to fully re-implement custom stylesheets, or
>
> Don't use `runtime uss` and `emo`, and strip `OneJS.CustomStyleSheets` from your runtime, or
>
> Get Full-access source code from Unity.

Styled Components and Emotion

OneJS provides APIs that are equivalent to the `styled`, `css`, and `attr` APIs from Styled Components, and also the `css` API from Emotion.

These are available from `preact/styled` (technically, it's provided by the `onejs-preact` package, but the shortcut mappings in tsconfig.json allows you to just use `preact/styled`).

TSX

```
import { h, render } from "preact"
import styled, { CompType, uss } from "preact/styled"

const Button = styled.button`
    border-width: 0;
    border-radius: 3px;
```

```
        background-color: green;
        color: white;
        &:hover {
            background-color: red;
        }

        ${props => props.primary && uss`
            background-color: white;
            color: black;
        `}
    ` as CompType<{ primary: boolean }, JSX.Button>

    render(<Button text="Hello!" primary />, document.body)
```

`attr` also works as you'd expect:

TSX

```
    import styled from "preact/styled"
    import { h, render } from "preact"

    const Jin = styled.button.attrs(props => ({
        text: props.foo
    }))`
        color: red;
    `

    const Cup = styled(Jin).attrs(props => ({
        text: "Fooooooo",
        bar: props.bar || 50
    }))`
        font-size: ${props => props.bar}px;
    `

    render(<Cup foo="My Text" />, document.body)
```

Emotion's `css` (which is different from Styled Components' `css`) is available as `emo` from `onejs/styled`:

TSX

```
    import { emo } from "onejs"
    import { h, render } from "preact"

    const Foo = (props) => {
        return <div class={emo`
            font-size: ${props.size ?? 10}px;
            color: red;
        `}>Hello</div>
```

```
}

render(<Foo size={30} />, document.body)
```

Please get in our Discord to share what other APIs you'd like to see supported.

Tailwind

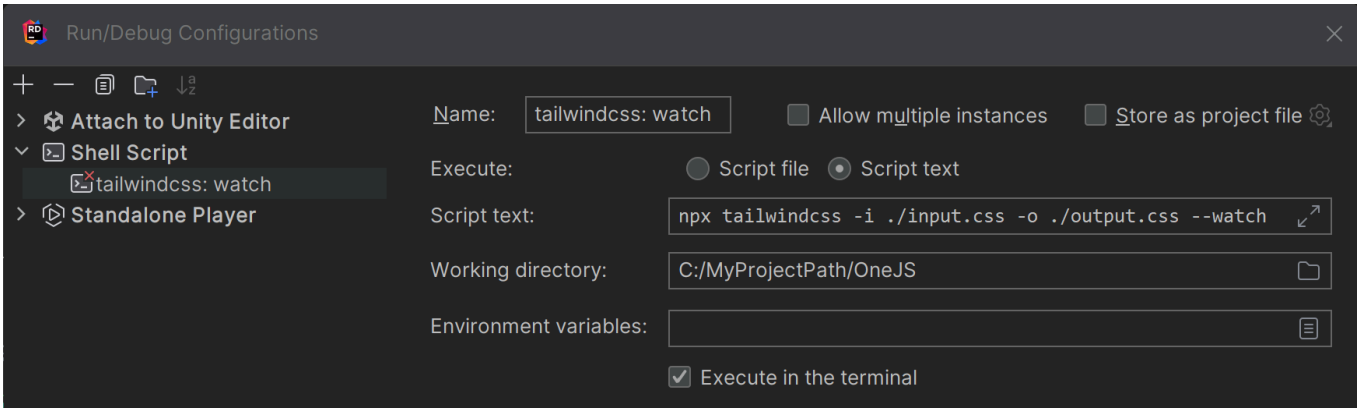
OneJS has built-in support for Tailwind. A default VSCode task is provided for you to start a Tailwind watcher. It's basically running the following command.

BASH

```
npx postcss input.css -o ../Assets/tailwind.uss --watch
```

All you have to do to use Tailwind is start the task and drag tailwind.uss onto ScriptEngine's Stylesheets list in Unity.

For IntelliJ IDEA, you can set up a Shell Script in Run/Debug Configurations, like so:



Caveats

USS only supports a subset of CSS features. Some utility classes will not work because of the following USS limitations.

> Complex selectors are not supported. So things like `space-y-0.5` that uses advanced selectors (`> * + *`) won't work.
>
> `var()` usage inside of another function like `rgb()` is not supported.

But since we now have the full power of the Tailwind compiler, we can make various workarounds via `tailwind.config.js`. Refer to `ScriptLib/onejs/onejs-tw-config.js` for examples and the Tailwind Customization docs for more info.

VSCode Extension

Remember, you can use the official Tailwind VSCode extension for better dev experience in VSCode.

Quick Example

JSX

```jsx
import { render, h } from "preact"

const App = () => {
    return (
        <div class="w-full h-full flex justify-center p-3">
            <div class="max-w-md mx-auto bg-white rounded-xl overflow-hidden md:max-w-2x
                <div class="md:flex md:flex-row md:h-full">
                    <div class="md:shrink-0 md:h-full">
                        <div class="h-60 w-full bg-crop md:h-full md:w-56" style={{ backg
                    </div>
                    <div class="p-8">
                        <div class="text-sm text-indigo-500 bold">Quote of the Day</div>
                        <a href="#" class="mt-1 text-lg text-black">I used to be an adver
                        <p class="mt-2 text-slate-500">- Town Guard, Elder Scrolls 5: Sky
                    </div>
                </div>
            </div>
        </div>
    )
}

render(<App />, document.body)
```

This is almost the exact little demo shown on this page.