VERSION v1.6.x

Why OneJS?

OneJS integrates widely adopted techs such as Typescript and (P)React into Unity, providing a familiar and powerful toolset for UI design. The use of React ensures maintainable and scalable UI code, while also enabling fast iteration speed for a streamlined development process.

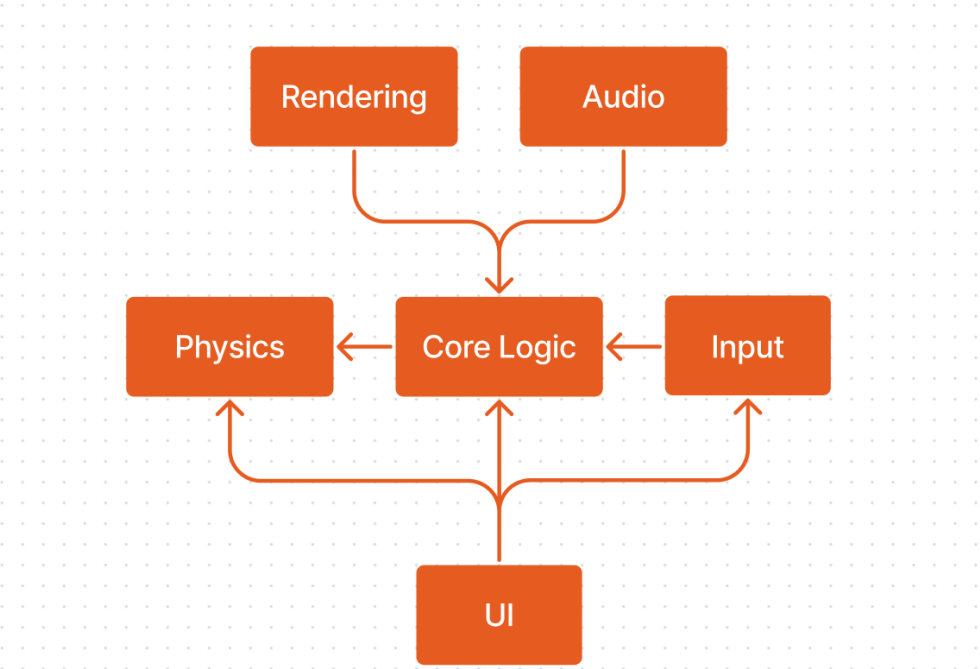
> **Fast Iteration**: The interpreted JavaScript allows for instant reloading of code changes, significantly speeding up your UI designing process. (Web devs take this for granted, but C# compilation and Domain reloading speed haven't been kind to us Unity developers).
>
> **Browser-less**: OneJS directly integrates with Unity's UI Toolkit, avoiding the need for a browser and simplifying access to UI Toolkit's DOM features.
>
> **Web-centric Techs**: OneJS supports popular web techs like Preact, Tailwind, Styled Components, and Emotion, enabling developers to work with familiar tools and improve productivity in Unity.
>
> **A Script Engine**: Besides being an UI solution, OneJS is also a Script Engine for Unity. You can access any .Net objects/classes/namespaces from Typescript. This means you can easily give scripting capabilities to your players if you choose to. (Think WoW Addons, but instead of Lua+XML, you've got Typescript+JSX).

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

It's important to have clear boundaries between different systems, such as Core Logic, Rendering, Audio, Physics, Input, and UI. This allows for better management of dependencies and reduces 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 will not have any effect on the rest of the systems, allowing 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)

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)

Also 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.

Quick Start

After downloading and importing OneJS from the Asset Store. You can just

Drag and drop the `ScriptEngine` prefab onto a new scene.

Enter Play mode.

In the console, if you see `[index.js]: OneJS is good to go.`, then OneJS is all set. You can now move on to Tutorial 101 for a step-by-step guide on writing your first script.

If you are already experienced with Unity, Typescript, and (P)React, you can also just skip to the included sample scenes to see how Preact and UI Toolkit work together. (*The script(s) responsible for the sample scenes are located under `{ProjectDir}/OneJS/Samples`.*)

More Info

OneJS uses `{ProjectDir}/OneJS` 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 `OneJS` folder into Version Control. When building for standalone, the scripts from `{ProjectDir}/OneJS` will be automatically bundled up and be extracted to `{persistentDataPath}/OneJS` at runtime. Refer to the Deployment page for more details on that.

The first time `ScriptEngine` runs, it will set up a few things automatically under `{ProjectDir}/OneJS`. These are:

- A default `tsconfig.json`
- A default `.vscode/settings.json`
- A default `index.js` script (that just logs something to the console)
- `ScriptLib` folder containing all the Javascript library files (and TS definitions) that are used by OneJS.
- `Samples` folder containing some sample code you can look at.

These folders and files will be auto-generated if deleted. Sometimes it maybe helpful to manually delete them when upgrading major OneJS versions.

VSCode

`{ProjectDir}/OneJS` is the folder to open when using VSCode. Typescript (.ts and .tsx) is the recommended language to use with OneJS. To have VSCode continuously transpile TS to JS in watch mode, use `Ctrl + Shift + B` or `Cmd + Shift + B` and choose `tsc: watch - tsconfig.json`.

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

You can, of course, just use plain .js and .jsx files as well. But do note that by default OneJS only support CommonJS modules (i.e. `require()` and `module.exports`). So if you want to use ES

modules (i.e. `import / export` statements), Typescript is the way to go.

The default `.vscode/settings.json` will enable Explorer File Nesting for you, as well as some PowerShell settings for better usage on Windows.

C#-JS Workflow

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

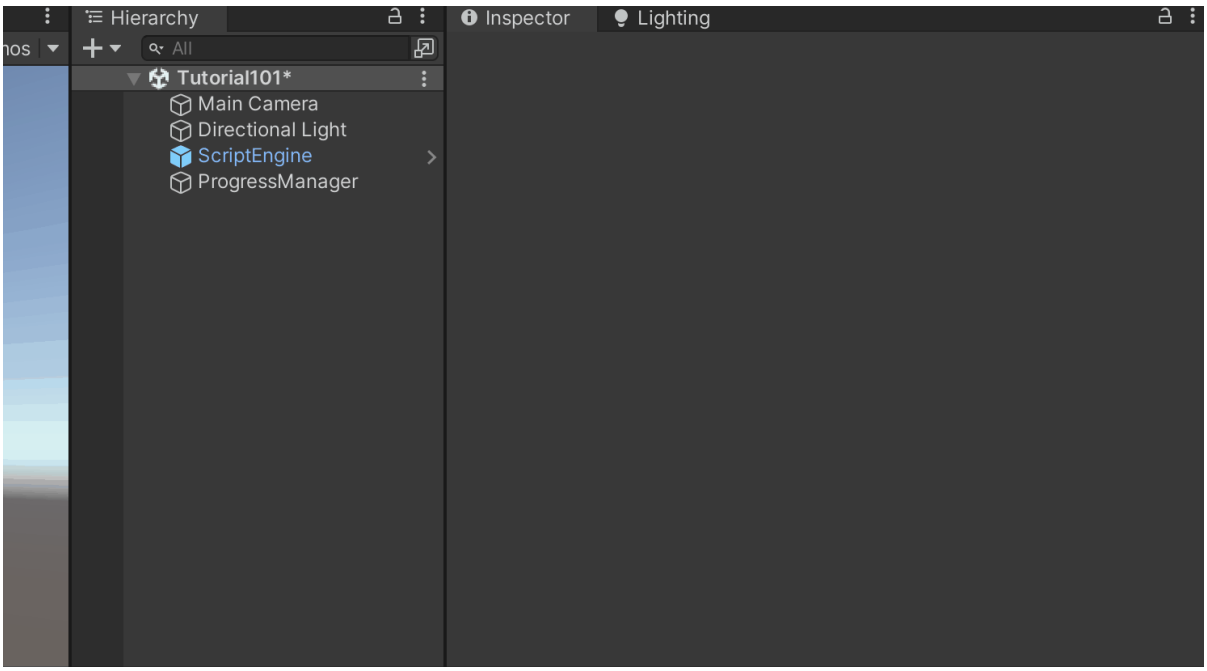
```
var { GameObject, MeshRenderer, Color } = importNamespace("UnityEngine")
var sphere = GameObject.CreatePrimitive(0)
sphere.GetComponent(MeshRenderer).material.color = Color.red
```

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

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

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

To pass .Net objects to TS, you can use the ScriptEngine/INTEROP/Objects list:



ProgressManager can now be accessible from TS via `require("pman")`

The INTEROP/Objects list accepts any UnityEngine.Object. This includes things like GameObject, MonoBehaviour, ScriptableObject, RenderTexture, Sprite, etc.

UI Dataflow

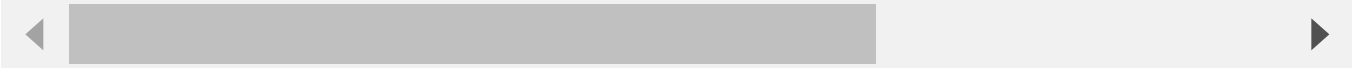
In general, your UI code should depend on your core game logic. But your core game logic should not even be aware of the existence of your UI code. In other words, your TS code will be calling stuff from your C# code, but never the other way around. This one-directional dependency makes everything easy 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:

```
// C#
public class TreasureChestSpawner : MonoBehaviour {  // Map this object to be 'chest-spa
    public event Action OnChestSpawned; // Fired when a chest is spawned in the scene

    ...
}
```



```
// TS
var spawner = require("chest-spawner") as MyGame.ChestSpawner
spawner.add_OnChestSpawned(onChestSpawned)

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

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


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

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:

```
const App = () => {
    var charman = require("charman")

    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:

```
const App = () => {
    var charman = require("charman")

    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. This is demonstrated in the Overwatch UI sample.

> NOTE: `useEventfulState(obj, "Health")` assumes the C# `obj` has a property named "`Health`" and an event named "`OnHealthChanged`".

C# Source Generator

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

```
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:

```
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.

Deployment

The Bundler component (on ScriptEngine) takes care of bundling your scripts at buildtime and extracting the bundle during runtime.

You can configure where your scripts should reside under ScriptEngine's MISC section (via the Editor & Player WorkingDir properties).

The bundling and extraction process is fairly automatic so you don't need to worry about it too much.

Live Reload, on or off

For production deployment where you don't need Live Reload, remember to turn it off by unchecking the "Turn On For Standalone" option on the Live Reload component. By default, this option is on because during development you may still find Live Reload useful for Standalone builds/apps.

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:

```
<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>
```

OneJS Limitations

3rd Party JS Modules

One of the goals of OneJS is to cut dependencies on browser and NodeJS. When you are working with OneJS, you are working restrictly within the confines of the Unity environment. Therefore, browser features not supported by UI Toolkit are most likely not supported by OneJS. And here we are talking about hard browser features like Canvas, SVG, Web Audio, complex CSS selectors, CSS animation keyframes, etc that are currently not supported by UI Toolkit.

(You can use UI Toolkit's Vector API as a direct replacement for Canvas. And SVG is on UI Toolkit's roadmap.)

So you need to keep this limitation in mind when using 3rd party JS modules (via npm for example). Libraries like Preact, Tween.js, and Lodash that have zero or minimal dependency on browser features besides some DOM manipulations, they will work with OneJS out-of-the-box.

But things like three.js will obviously not work. Same goes for libraries that make heavy use of CSS animations. The time you save on using these libraries may not be worth the effort you need to put into polyfilling the missing browser features.

(In light of CSS Animations, UI Toolkit does support UI Transitions which are limited in scope but fairly easy to work with and performant!)

In general, if your goal is to be able to use all 3rd-party browser and node modules out-of-the-box with Unity, you will have to go for an embedded-browser solution. OneJS is for folks who want to do Preact & JSX with UI Toolkit and don't want to embed full-blown webviews in their apps.

Custom Editor UI

Currently, OneJS is for runtime only. However, custom Editor Window support is on our roadmap.

WebGL

Currently you can actually use OneJS with WebGL. But keep in mind that UI Toolkit itself can be buggy in WebGL, and WASM linking can be (really) slow if you preserve whole assemblies like in the example link.xml. As time goes, UI Toolkit should become more stable with WebGL builds.

Upgrading OneJS

You can upgrade OneJS as usual from Unity's Package Manager or through the OneJS private repo. If you encounter any issues, try deleting the `ScriptLib` folder and `tsconfig.json` file under OneJS's working directory (i.e. `{ProjectDir}/OneJS`). They'll be auto-regenerated by the new ScriptEngine.

If you are using Tailwind, make sure to run the tailwind compiler at least once to flush out any stale styling.

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

ScriptEngine

The `ScriptEngine` component is the core of OneJS. It manages interop between C# and JS (via Jint) and provides the DOM implementations needed by Preact.

INTEROP

Settings under INTEROP are generally about what features from C#/.Net you want to expose to Javascript. *Whenever you start having trouble accessing .Net stuff from JS, here is the place you should check first.*

> **Objects**: Maps an Unity Object to a js module name (any string that you choose). Objects declared here will be accessible from Javascript via `require("objname")`. The Objects list accepts any UnityEngine.Object, not just MonoBehaviours. To pick a specific MonoBehaviour component, you can right-click on the Inspector Tab of the selected

GameObject and pick Properties. A standalone window will pop up for you to drag the specifc MonoBehavior from.

Assemblies: List of Assembly names you want to access from Javascript. (i.e. `"UnityEngine.CoreModule"` and `"Unity.Mathematics"`)

Namespaces: You can map C# namespaces to JS module here. (i.e. `"UnityEngine.UIElements" => "UnityEngine/UIElements"`)

Extensions: List of Extension names you want to access from Javascript. (i.e. `"UnityEngine.UIElements.PointerCaptureHelper"`)

Static Classes: Map C# static classes to JS module. (i.e. `"Unity.Mathematics.math" => "math"`)

SECURITY

`ScriptEngine` provides the following security settings for you to set in the Inspector.

Catch .Net Exceptions

Allow Reflection

Allow `GetType()`

Memory Limit

Timeout

Recursion Depth

These are some of the security settings exposed directly from Jint. To set more granular security measures such as Member Accessor & TypeResolver, you can do so during the `OnPostInit` event (refer to the event API below).

STYLING

You can add additional USS Style Sheets here and also tweak screen breakpoints.

MISC

Here, you can set the OneJS WorkingDir paths for Editor and runtime. The Bundler will use this info for automatic bundling and extraction. The only thing you need to make sure is that the 2 paths are not the same.

Also under MISC, you can set what scripts you want to load before and after every engine reload.

ScriptEngine APIs

`ScriptEngine` exposes some public APIs for you to use from code.

Properties

```
public string WorkingDir; // The OneJS Working Directory

public Engine JintEngine; // Internal Jint Engine

public Document Document; // Document object (`document` in js)

public Dom DocumentBody; // document.body

public DateTime StartTime // Engine Start time. Reset on every reload.
```

Events

```
public event Action OnPostInit; // Happens after every ScriptEngine reload

public event Action OnReload; // Happens when ScriptEngine is just about to reload
```

Methods

```
public void RunScript(string scriptPath); // Run a script as is

public void ReloadAndRunScript(string scriptPath); // Reloads the ScriptEngine and then

public void RegisterReloadHandler(Action handler); // Sub to the OnReload event and will
```

Live Reload

Live Reload requires `Run in Background` to be turned ON in Player settings (under Resolution and Presentation).

The `LiveReload` MonoBehaviour component watches your `{ProjectDir}/OneJS` directory and will reload the `ScriptEngine` (and your entry script) when code changes are detected. Settings:

Run On Start (default On):

Entry Script (default "index.js"): Which file to run on `ScriptEngine` reload. Note this should always be a .js file.

Watch Filter (default "*.js"): What type of files to watch

Multi-Device Live Reload

There is a 4th setting (Net Sync) that you can set to enable Live Reload across different devices. For example, you can make code changes in VSCode on your Desktop and have the change live reloaded on your deployed mobile app. Devices will self discover as long as they are on the same network (i.e. same wifi or lan).

After toggling on Net Sync, you can pick 3 modes:

Auto defaults to Server for Desktop, and Client for Mobile

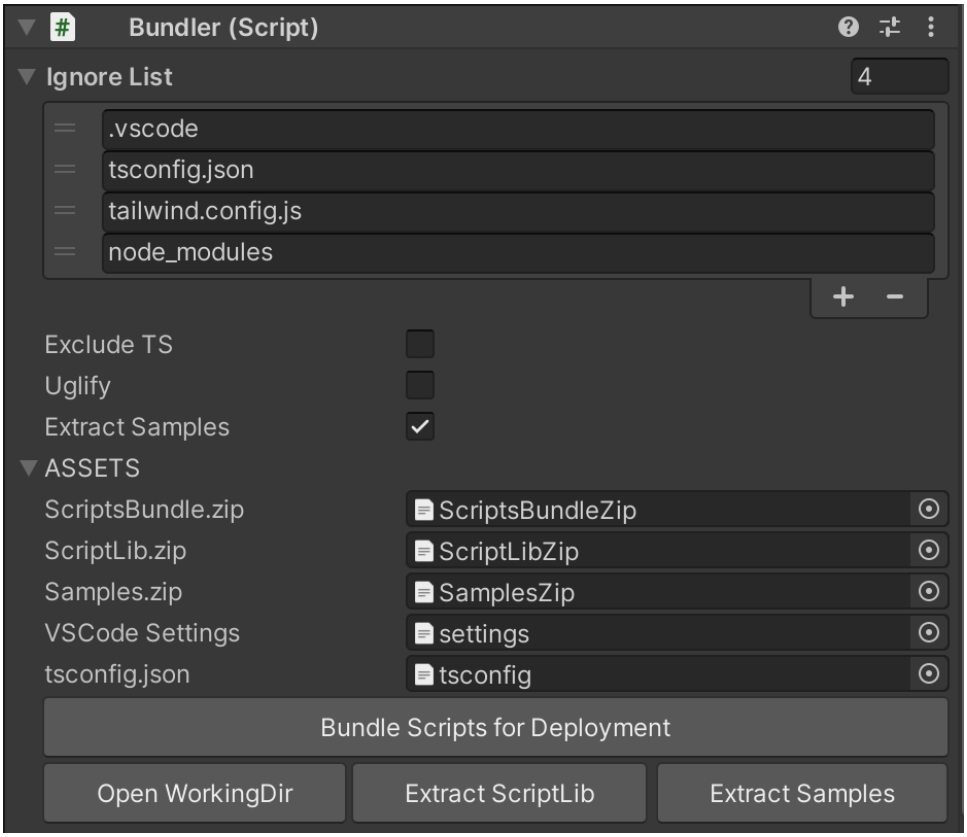
Server will be responsible for broadcasting code changes to clients

Client will be listening for code changes from Server

Janitor

A `JanitorSpawner` should be used along side `LiveReload`. It'll spawn a Janitor GameObject that will help cleaning up previous GameObjects and Console logs upon every Reload.

Bundler



The Bundler component takes care of bundling your scripts at buildtime and extracting the bundle during runtime. More specifically, it turns everything under your Editor WorkingDir into a zip at buildtime. And at runtime, the Bundler will extract the zip into your Player WorkingDir. In most cases, you don't need to worry about the bundling and extraction process as they are automatic.

It's worth noting that it's okay to have multiple ScriptEngines in your scene(s). Just be mindful that during build, every instance of the Bundler component (found in all scenes in your Build Settings) will run the bundling process. So if you do have multiple Bundlers and Working Directories, you may want to have separate sets of assets (i.e. ScriptsBundle.zip, ScriptLib.zip, etc) for each of the Bundler. To do that, you can just duplicate the existing ones and drag the new ones to a different Bundler accordingly.

Dom

OneJS's `Dom` is a thin wrapper over the underlying VisualElement. `Dom` basically implements the mininum amount of features that's required from Preact.

The `Dom` object has 2 properties that are worth mentioning. One is `dom.ve`, the other is `dom.style`.

`dom.ve` is useful for grabbing the actual VisualElement. But because VisualElement is from C#/.Net, sometimes working with it can be a bit verbose. For example, to set the `rotate` style

for a VisualElement, you'd need to do:

```
ref.current.ve.style.rotate = new StyleRotate(new Rotate(new Angle(-36)))
```

Which is fairly tedious to write. This is where `dom.style` comes in. It's a convenience wrapper over the `.ve.style` counterpart. So you can just do this:

```
ref.current.style.rotate = -36
```

Full code example:

```
import { h, render } from "preact"
import { Dom } from "OneJS/Dom"
import { useRef, useEffect } from "preact/hooks"
import { Angle, Rotate, StyleRotate } from "UnityEngine/UIElements"

const App = () => {
    const ref = useRef<Dom>()

    useEffect(() => {
        // ref.current returns the Dom object
        // ref.current.ve returns the underlying VisualElement
        log(ref.current.ve.resolvedStyle.width)

        // Both the following lines do the same thing
        ref.current.ve.style.rotate = new StyleRotate(new Rotate(new Angle(-36)))
        ref.current.style.rotate = -36
    }, [])

    return <div ref={ref}>Hello</div>
}

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

Tailwind

Tailwind support went through a major rework in v1.6. This is currently available via the private OneJS repo. Please report any issues you may find.

Before this update, Tailwind support in OneJS primarily depended on static utility classes in USS. But now you can actually use the Tailwind compiler (CLI) and the tailwind.config.js file for any theming and customization needs.

Usage

Enter Playmode with ScriptEngine in the scene.

This will create a new `tailwind.config.js` file under the working directory (`{ProjectDir}/OneJS`) if one is not found. Please append to the `content` field accordingly for the js files you'd like to monitor.

Install Tailwind via npm: (under directory `{ProjectDir}/OneJS`)

```
npm install -D tailwindcss
```

Run the tailwindcss compiler in watch mode

```
npx tailwindcss -i ./input.css -o ./output.css --watch
```

To make things easier, a default VSCode Task (tasks.json) is also provided so you can quickly start the tailwind compiler without opening a separate Shell/Terminal. You can use `Ctrl + Shift + B` to access the task `tailwindcss: watch` in VSCode.

Caveats

When a USS file is modified (directly or indirectly by tailwind compiler) at runtime/playmode, The editor will need focus in order for the change to be processed.

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 easily make workarounds via `tailwind.config.js` . Refer to `ScriptLib/onejs/onejs-tw-config.js` for examples and the Tailwind Customization docs for more info.

Runtime USS

One of the current limitations of UI Toolkit is its inability to load USS files at runtime (from strings). So, we decided to fill in the gap.

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

```
import { h, render } from "preact"
import { useEffect } from "preact/hooks"
import { 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={t
        <div><label text="Lorem Ipsum" /></div>
        <button onClick={setUSS} text="Set USS"></button>
    </div>
}

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

Limitations

Runtime USS was made to work by separating/decoupling Asset path management (editor mode feature) from the StyleSheet importer. So any path-based values such as image and font urls are currently not supported in USS strings. To use these, please either go for inline styles or static uss files.

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 `onejs/styled`:

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

```
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 }>

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

`attr` also works as you'd expect:

```
import styled from "onejs/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`:

```
import { emo } from "onejs/styled"
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.

Async Await Support

```
// C#
public class Foo : MonoBehaviour {
    public async Task<string> GetAsyncString() {
        await Task.Delay(1000);
        return "GetAsyncString";
    }

    public async Task DoAsyncMethod() {
        await Task.Delay(1000);
        print("DoAsyncMethod");
    }
}
```

```
// TS
var foo = require("foo")

async function test() {
    const text = await foo.GetAsyncString()
    log(text)
    await foo.DoAsyncMethod()
}

test()
```

Custom Controls

OneJS comes with a couple custom controls by default. More will come in the future.

GradientRect

You can see the use of `GradientRect` in the Fortnite UI demo. They were used for the health and shield bars. General usage is like this:

```
import { render, h } from "preact"
import { parseColor as c } from "onejs/utils/color-parser"

const App = () => {
    return <div>
        <gradientrect colors={[c("#02BC23"), c("#48E025")]} style={{ width: 100, height:
        <gradientrect colors={[c("red"), c("#00A0E6"), c("#00A0E6"), c("red")]} style={{
    </div>
}

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

Refer to the `GradientRect` C# code for more info.

Flipbook

`Flipbook` was used in the Overwatch UI demo for the lightning animation. Example Usage:

```
import { render, h } from "preact"
import { ImageLoader } from "OneJS/Utils"

const lightningTexture = ImageLoader.Load(__dirname + "/resources/lightning.png")

const App = () => {
    return <div>
        <flipbook src={lightningTexture} num-per-row={8} count={32} interval={0.025} ran
    </div>
}

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

Refer to the `Flipbook` C# code for more info.

C# to TS Definition

Out of the box, OneJS provides tons of TS definitions for the Unity ecosystem. `UnityEngine`, `UIElements`, `Mathematics`, to name a few. But there will be times that you'll want to make your

own TS definitions for things we haven't covered.

So to make things easier for you, we included an auto converter that can extract TS definitions out of any C# type. You can access it from Unity menu (OneJS -> C# to TSDef Converter). The name you use should be the fully qualified type name. Remember you can use syntax like `Foo`1` for generics and `Foo+Bar` for nested types.

Please note that no such converter is perfect. Ours works better than all the other ones we've tried. It'll get 90% of the work done for you, but you'll still need to make adjustments here and there.