(Note: Doc and samples also available at onejs.com)

What is OneJS?

OneJS is a Javascript Engine specifically designed for Unity. It is lightweight, performant, pure C# based, has first-class Typescript support, and works everywhere. It will be available on the Unity Asset Store (currently pending review).

OneJS allows full Preact integration with UI Toolkit. Support for Tailwind and Chakra is in the works and will be coming in the next update.

Feature Highlights

- Use **Preact** for UI Development in Unity
- **Fast Iteration speed** with Live Reload
- **Awesome Performance** due to 1-to-1 interop between Preact and UI Toolkit
- **TS Definitions** for tons of UnityEngine and UIElements types. We also provide an C#-to-TS Type Converter that will make your Typing life much easier.
- **Works Everywhere** (Mac, Windows, iOS, Android, Editor, Standalone, Mono, Il2cpp)
- **Built-in Security** when you need it. (Should you choose to give your players scripting capabilities, you can set many security settings such as memory limit, call depth, script timeout, among many others, courtesy of Jint)

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)

Setting up OneJS

Setup is simple and straight-forward. 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. Refer to the included sample scene to see how Preact and UI Toolkit work. The script(s) responsible for the sample scene are under Addons/Sample (under your persistentDataPath)

ScriptEngine

`ScriptEngine` uses your project's `persistentDataPath` folder as its working directory. The first time it runs, `ScriptEngine` will set up a few things automatically in this directory. 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.
- `Addons` folder containing some sample code you can look at.

Now you can use VSCode to open up your project's `persistentDataPath` folder. You can put your own scripts anywhere really, but we recommend to keep them under the `Addons` folder.

VSCode

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

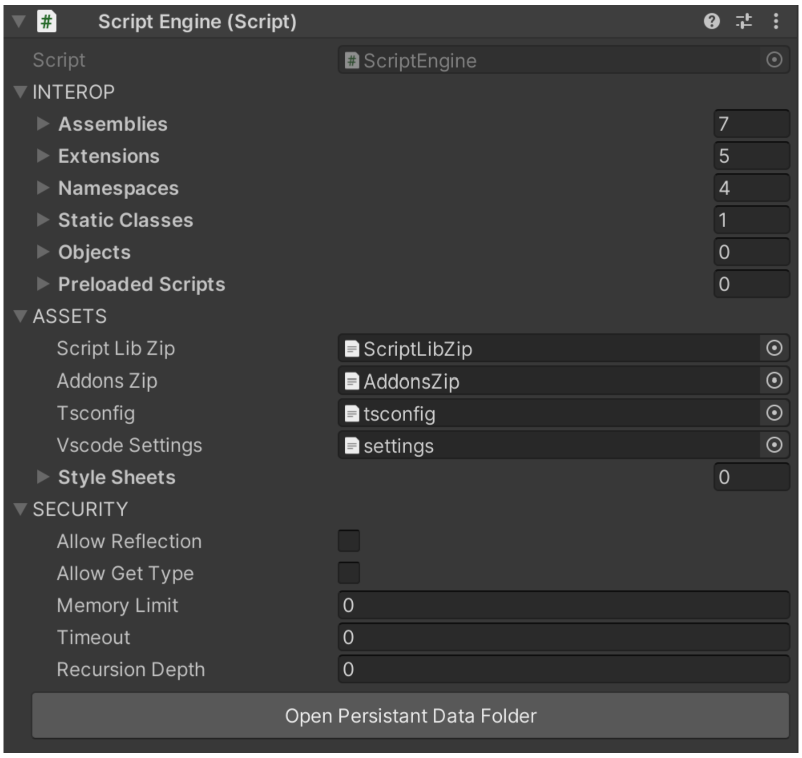
```
{ // .vscode/settings.json
    "explorer.fileNesting.enabled": true,
    "explorer.fileNesting.expand": false,
    "explorer.fileNesting.patterns": {
        "*.ts": "${capture}.js, ${capture}.d.ts",
        "*.js": "${capture}.js.map, ${capture}.min.js, ${capture}.d.ts",
        "*.jsx": "${capture}.js",
        "*.tsx": "${capture}.ts, ${capture}.js, ${capture}.d.ts",
        "tsconfig.json": "tsconfig.*.json",
        "package.json": "package-lock.json, yarn.lock"
    },
    "terminal.integrated.profiles.windows": {
        "PowerShell": {
            "source": "PowerShell",
            "icon": "terminal-powershell",
            "args": [
                "-ExecutionPolicy",
                "Bypass"
            ]
        }
    },
    "terminal.integrated.defaultProfile.windows": "PowerShell"
}
```

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`.

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.

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. Below are the settings you can set on the component, divided into 3 categories: Interop, Assets, and Security.



Script Engine (Script)	
Script	ScriptEngine
INTEROP	
Assemblies	7
Extensions	5
Namespaces	4
Static Classes	1
Objects	0
Preloaded Scripts	0
ASSETS	
Script Lib Zip	ScriptLibZip
Addons Zip	AddonsZip
Tsconfig	tsconfig
Vscode Settings	settings
Style Sheets	0
SECURITY	
Allow Reflection	☐
Allow Get Type	☐
Memory Limit	0
Timeout	0
Recursion Depth	0

Interop

Settings under Interop are generally about what features from C#/.Net you want to expose to Javascript.

- **Assemblies**: List of Assembly names you want to access from Javascript. (i.e. `"UnityEngine.CoreModule"` and `"Unity.Mathematics"`)
- **Extensions**: List of Extension names you want to access from Javascript. (i.e. `"UnityEngine.UIElements.PointerCaptureHelper"`)
- **Namespaces**: You can map C# namespaces to JS module here. (i.e. `"UnityEngine.UIElements" =>; "UnityEngine/UIElements"`)
- **Static Classes**: Map C# static classes to JS module. (i.e. `"Unity.Mathematics.math" => "math"`)
- **Objects**: Map a MonoBehaviour component to JS module. Note you can drag a Component to the slot, but you can probably only do so by locking the `ScriptEngine` Inspector and opening a new Inspector Tab for the target MonoBahaviour. (i.e. `"MaterialManager" => "matman"`)

Assets

The settings under Assets mostly contain templates or default files for OneJS to use, such as `tsconfig.json` and `settings.json`. So You don't really need to touch any of that. The only thing you may want to add are the Style Sheets (USS).

Security

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

- **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).

ScriptEngine APIs

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

Properties

```
public Engine JintEngine; // Internal Jint Engine

public Dom DocumentBody; // Dom for document.body
```

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 run a script
```

Live Reload

The `LiveReload` MonoBehaviour component watches your `persistentDataPath` directory and will reload the `ScriptEngine` (and your entry script) when code changes are detected. It has 3 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 (see FileSystemWatcher.Filter)

Multi-Device Live Reload (Coming Soon)

There will be 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.

This feature is in the works and should be available in the next version. Docs here will be updated accordingly then.

Janitor

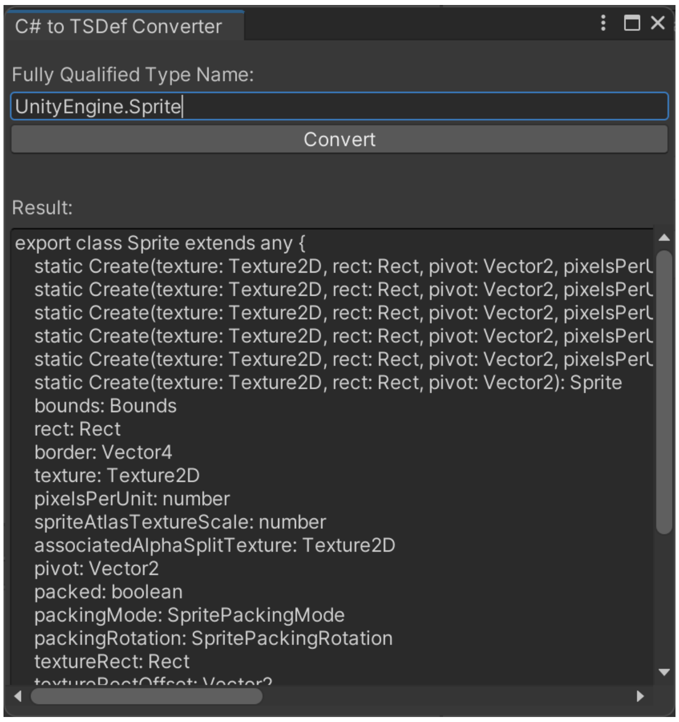
A `JanitorSpawner` should be used along side `LiveReload`. It'll spawn a Janitor GameObject that will help cleaning up previous GameObjects and Console logs upon every Reload.

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 include 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.



UnityEngine.Sprite

```
export class Sprite extends any {
    static Create(texture: Texture2D, rect: Rect, pivot: Vector2, pixelsPerU
    static Create(texture: Texture2D, rect: Rect, pivot: Vector2, pixelsPerU
    static Create(texture: Texture2D, rect: Rect, pivot: Vector2, pixelsPerU
    static Create(texture: Texture2D, rect: Rect, pivot: Vector2, pixelsPerU
    static Create(texture: Texture2D, rect: Rect, pivot: Vector2, pixelsPerU
    static Create(texture: Texture2D, rect: Rect, pivot: Vector2): Sprite
    bounds: Bounds
    rect: Rect
    border: Vector4
    texture: Texture2D
    pixelsPerUnit: number
    spriteAtlasTextureScale: number
    associatedAlphaSplitTexture: Texture2D
    pivot: Vector2
    packed: boolean
    packingMode: SpritePackingMode
    packingRotation: SpritePackingRotation
    textureRect: Rect
    textureRectOffset: Vector2
```