OUR BLOG

Wonderland Engine 1.0.0 JavaScript Migration

JONATHAN HALE | FEBRUARY 9, 2023

New JavaScript architecture in Wonderland Engine and how to migrate.

Wonderland Engine has been undergoing massive changes in the way it handles, interacts, and distributes JavaScript code.

This blog post will guide you through those changes. In addition, the Migrations section will go over each new feature to detail migration steps.

Motivation

Until now, WonderlandEngine relied on local scripts. Users would create scripts, and the editor would pick them up and bundle them to create the final application.

It was required to pre-bundle any third-party libraries, and place them in the project's folder structure.

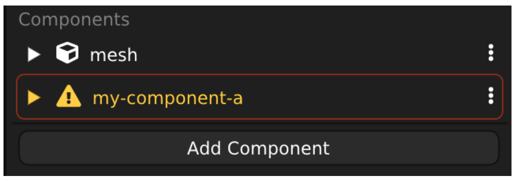
The old system had several limitations:

- No **npm** dependencies
- Potential namespace clash with other **WebAssembly** libraries
- Our custom bundler / transform didn't support advanced use cases, such as **TypeScript**

We have been working on a new JavaScript ecosystem to help you work seamlessly with your favorite tools.

Editor Components

If you were previously a npm user, chance is that you might have encountered this:



You might have seen his warning, especially if you are a npm user. It happens when the we forget to register components in the entry point (`index.js` file).

Starting **1.0.0**, the components "seen" by the editor will not be picked from the application bundle. Besides fixing the above error, it will be possible to register **more** components in the editor than in the final application. This will allow advanced users to setup highly complex projects with streamable components **in the future**.

With this change, the editor will now require some inputs to discover components:

- Listing components or folders in `Views > Project Settings > JavaScript > sourcePaths`
- Adding **dependencies** in the root `package.json` file

Example of `package.json` with a library exposing components:

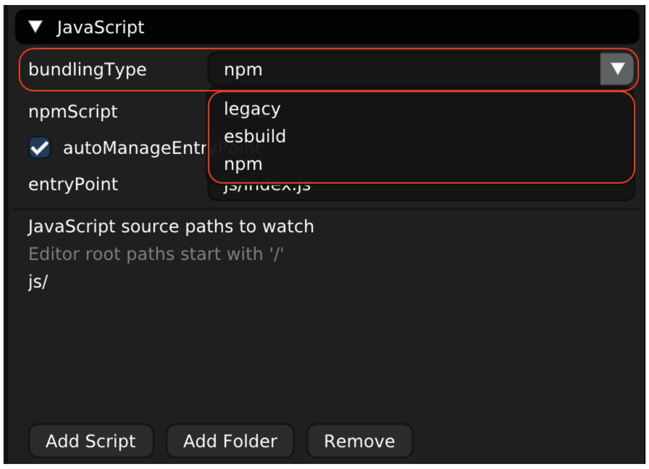
```json
{
  "name": "MyWonderfulProject",
  "version": "1.0.0",
  "description": "My Wonderland project",
  "dependencies": {
    "@wonderlandengine/components": "^0.9.2"
  }
}
```

The editor is thus capable of finding components by reading your `package.json` to **speed up development time** and improve **shareability**. For more information, please have a look at the Writing JavaScript Libraries tutorial.

Bundling

There is now a new setting that allows you to modify the bundling type:

`Views > Project Settings > JavaScript > bundlingType`



Let's have a look at each of these options.

Esbuild

Your scripts are going to be bundled using the Esbuild bundler.

This is the **default** choice. You are advised to stick to this setting whenever you can for performance reasons.

Npm

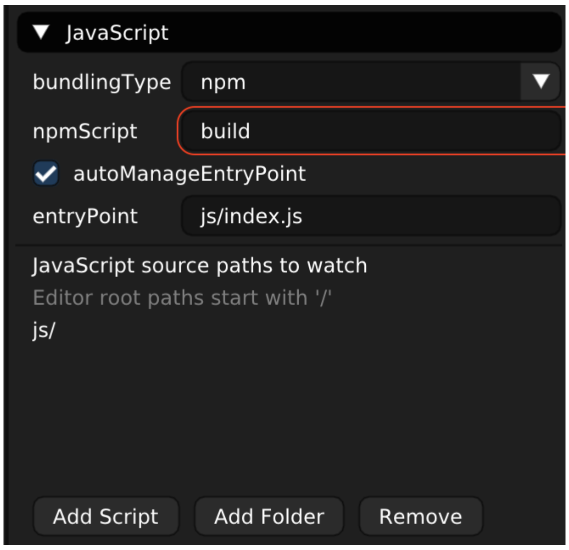
Your scripts are going to be bundled using your own npm script.

Example of `package.json` with a custom `build` script:

```json
{
  "name": "MyWonderfulProject",
  "version": "1.0.0",
  "description": "My Wonderland project",
  "type": "module",
  "module": "js/index.js",
  "scripts": {
    "build": "esbuild ./js/index.js --bundle --outfile=\"deploy/MyWonderland-bundle.js\""
  },
  "devDependencies": {
    "esbuild": "^0.15.18"
  }
}
```

The script **npm** script name can be set in the editor settings:

`Views > Project Settings > JavaScript > npmScript`



▼ JavaScript

bundlingType: npm

npmScript: build

✓ autoManageEntryPoint

entryPoint: js/index.js

JavaScript source paths to watch

Editor root paths start with '/'

js/

This script can run any command, as long as it generates your final application bundle.

You can use your favorite bundler, such as Webpack or Rollup. However, we advise users to use tools like Esbuild to reduce iteration time.

Application Entry Point

We went through how the editor retrieve components. However, we haven't talked about how those components are registered at runtime, i.e., when running in a **browser**.

The editor supports a new option that will automatically generate the entry point of your application, i.e., an `index.js` file.

The generated index would look like that:

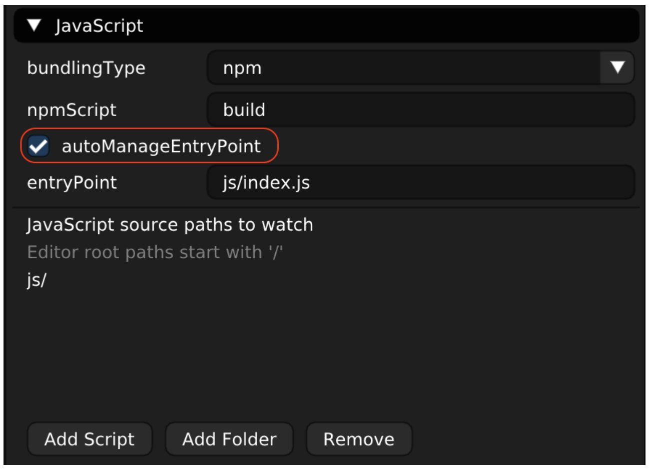
```
import '@wonderlandengine/components';

import {MyComponentA} from './js/my-component-a.js';
WL.registerComponent(MyComponentA);

import {MyComponentB} from './js/my-component-b.js';
WL.registerComponent(MyComponentB);
```

This feature can be turned on at:

`Views > Project Settings > JavaScript > autoManageEntryPoint`



▼ JavaScript

bundlingType: npm

npmScript: build

✓ autoManageEntryPoint

entryPoint: js/index.js

JavaScript source paths to watch

Editor root paths start with '/'

js/

With this option, the file will be auto-generated with your own scripts as well as the ones listed in the `package.json` dependencies.

For simple applications, you don't need to manage the entry point yourself and you can have this option enabled. For more complex uses cases, you are free to create and manage your own `index.js` file.

Managing manually the index allows to create complex scenarios will multiple entry points.

JavaScript Classes

WonderlandEngine **1.0.0** comes with a new way to declare components: **ES6 Class**.

```
import {Component, Type} from '@wonderlandengine/api';

class Forward extends Component {
    /* Registration name of the component. */
    static TypeName = 'forward';
    /* Properties exposed in the editor. */
    static Properties = {
        speed: {type: Type.Float, default: 1.5}
    };
    constructor() {
        this._forward = new Float32Array([0, 0, 0]);
    }
    update(dt) {
        this.object.getForward(this._forward);
        this._forward[0] *= this.speed;
        this._forward[1] *= this.speed;
        this._forward[2] *= this.speed;
        this.object.translate(this._forward);
    }
}
```

There are a couple of things to note:

- We use the api from `@wonderlandengine/api` and not from the global symbol `WL`
- We create a class that inherits from the api `Component` class
- The registration name of the component is now a static property
- The properties are set on the class

JavaScript Isolation

For users using the **internal bundler**, you may have seen code such as:

component-a.js

```
var componentAGlobal = {};

WL.registerComponent('component-a', {}, {
    init: function() {
        comonentAGlobal.init = true;
    },
});
```

component-b.js

```
WL.registerComponent('component-b', {}, {
    init: function() {
        if(componentAGlobal.init) {
              console.log('Component A has been initializaed before B!');
        }
    },
});
```

The above code is making some assumptions about the `componentAGlobal` variable. It expects `component-a` to be registered first and prepended in the bundle.

This used to work because the WonderlandEngine internal bundler didn't perform **isolatation**.

With **1.0.0**, whether you use **esbuild** or **npm**, this will not work anymore. Major bundlers will not be able to make the link between `componentAGlobal` used in `component-a` and the one used in `component-b`;

As a rule of thumb: Think about each file as **isolated** when using a bundler.

Migrations

Some manual migrations steps are required depending on whether your project was using **npm** or not.

Each section will describe the appropriate steps required based on your previous setup.

Editor Components

Internal Bundler

For users previously using the **internal bundler**, i.e., with the `useInternalBundler` checkbox **activated**:

```
Views > Project Settings > JavaScript > useInternalBundler
```

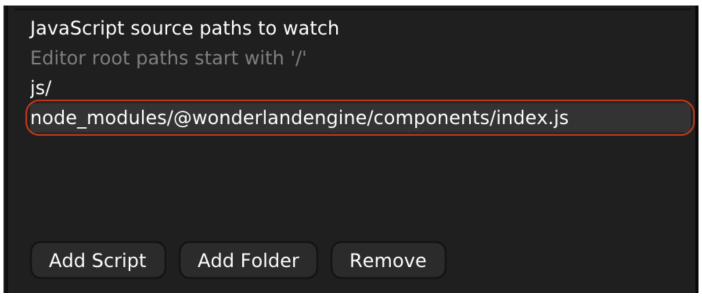
No further step is **required**.

Npm

For **npm** users, you will need to ensure that your own scripts are listed in the `sourcePaths` setting.

If you are using a library, please ensure that it's been migrated to WonderlandEngine **1.0.0** following the Writing JavaScript Libraries tutorial.

In case one of your dependency isn't up-to-date, you can add local the path to the `node_modules` folder in the `sourcePaths` settings. Example:



JavaScript source paths to watch

Editor root paths start with '/'

js/

node_modules/@wonderlandengine/components/index.js

Always keep in mind that the generated bundle via **npm** or **esbuild** will not be used anymore to find the components in the editor. It will only be used when **running your application**.

Bundling

Internal Bundler

For users previously using the **internal bundler**. **No further step** is **required**.

The project should have auto-migrated to the `esbuild` setup.

Npm

No further step is **required**.

Application Entry Point

Internal Bundler

For users previously using the **internal bundler**, **No further step** is **required**.

The `autoManageEntryPoint` setting will be enabled by default.

Npm

For npm users, the `autoManageEntryPoint` setting will be **disabled** by default to avoiding erasing any existing `index.js` file.

You can then enable or disable the option. Please note that enabling this option will **overwrite** the content of any previous file existing at the same location.

JavaScript Classes

This section is the same for every users, whether you used to have `useInternalBundler` enabled or not.

Let's have a look at some code comparing the old way versus the new way:

Before 1.0.0

```
WL.registerComponent('player-height', {
    height: {type: WL.Type.Float, default: 1.75}
}, {
    init: function() {
        WL.onXRSessionStart.push(this.onXRSessionStart.bind(this));
        WL.onXRSessionEnd.push(this.onXRSessionEnd.bind(this));
    },
    start: function() {
        this.object.resetTranslationRotation();
        this.object.translate([0.0, this.height, 0.0]);
    },
    onXRSessionStart: function() {
        if(!['local', 'viewer'].includes(WebXR.refSpace)) {
            this.object.resetTranslationRotation();
        }
    },
    onXRSessionEnd: function() {
        if(!['local', 'viewer'].includes(WebXR.refSpace)) {
            this.object.resetTranslationRotation();
            this.object.translate([0.0, this.height, 0.0]);
        }
    }
});
```

After 1.0.0

```
/* Don't forget that we now use npm dependencies */
import {Component, Type} from '@wonderlandengine/api';

export class PlayerHeight extends Component {
    static TypeName = 'player-height';
    static Properties = {
        height: {type: Type.Float, default: 1.75}
    };

    init() {
        /* WonderlandEngine 1.0.0 is moving away from a global
         * instance. You can now access the current engine instance
         * via `this._engine`. */
        this._engine.onXRSessionStart.push(this.onXRSessionStart.bind(this));
        this._engine.onXRSessionEnd.push(this.onXRSessionEnd.bind(this));
    }
    start() {
        this.object.resetTranslationRotation();
        this.object.translate([0.0, this.height, 0.0]);
    }
    onXRSessionStart() {
        if(!['local', 'viewer'].includes(WebXR.refSpace)) {
            this.object.resetTranslationRotation();
        }
    }
    onXRSessionEnd() {
        if(!['local', 'viewer'].includes(WebXR.refSpace)) {
            this.object.resetTranslationRotation();
            this.object.translate([0.0, this.height, 0.0]);
        }
    }
}
```

Those two examples are equivalent and will lead to the same result.

You will note a difference for the line **5**:

```
WL.onXRSessionStart.push(this.onXRSessionStart.bind(this));
```

versus

```
this._engine.onXRSessionStart.push(this.onXRSessionStart.bind(this));
```

Because we now migrated to **npm dependencies** and standard bundling, there is no need for a global `WL` variable.

Having a globally exposed engine came with two limitations:

- Harder to share components
- Impossible to have multiple engine instances running

While the second bullet point is not a common use case, we don't want to limit any users in terms of scalability.

JavaScript Isolation

This section is the same for every users, whether you used to have `useInternalBundler` enabled or not.

There are multiple ways to share data between components, and it's up to the developer of the application to choose the most appropriate one.

We will give a few examples about sharing data that do not rely on global variables.

State in Components

Components are bags of data that are accessed via other components.

You can thus create components to hold the state of your application. As an example, if you are creating a game with three states:

- ***Running***
- ***Won***
- ***Lost***

You can create a singleton component with the following shape:

game-state.js

```
import {Component, Type} from '@wonderlandengine/api';

export class GameState extends Component {
  static TypeName = 'game-state';
  static Properties = {
    state: {
      type: Type.Enum,
      values: ['running', 'won', 'lost'],
      default: 0
    }
  };
}
```

THe `GameState` component can then added to a manager object. This object should then be referenced by components that will alter the state of the game.

Let's create a component to change the state of the game when a players dies:

player-health.js

```
import {Component, Type} from '@wonderlandengine/api';

import {GameState} from './game-state.js';

export class PlayerHealth extends Component {
  static TypeName = 'player-health';
  static Properties = {
    manager: {type: Type.Object},
    health: {type: Type.Float, default: 100.0}
  };
  update(dt) {
    /* The player died, change the state. */
    if(this.health <== 0.0) {
      const gameState = this.manager.getComponent(GameState);
      gameState.state = 2; // We set the state to `lost`.
    }
  }
}
```

This is one way demonstrating how to replace globals in your application **pre-1.0.0**.

Exports

It's also possible to share variable via **import** and **export**. However, remember that the object is going to be identical in the entire bundle.

We can revisit the above example with exports:

game-state.js

```
export const GameState = {
  state: 'running'
};
```

player-health.js

```
import {Component, Type} from '@wonderlandengine/api';

import {GameState} from './game-state.js';

export class PlayerHealth extends Component {
  static TypeName = 'player-health';
  static Properties = {
    manager: {type: Type.Object},
    health: {type: Type.Float, default: 100.0}
  };
  update(dt) {
    if(this.health <== 0.0) {
      GameState.state = 'lost';
    }
  }
}
```

This solution works, but isn't bulletproof.

Let's have a look at an example where this doesn't work. Let's image that this code is in a library called `gamestate`.

- Your application depends on `gamestate` version **1.0.0**
- Your application depends on library A
- Library A depends on `gamestate` version **2.0.0**

Your application will end up with two copies of the `gamestate` library, because both aren't compatible in terms of version.

When library A updates the `GameState` object, it is in fact changing its own instances of this export. This happens because both versions aren't compatible, making your application bundle two different instances of the library.

Last Update: February 9, 2023