
buildjs-guide

Release 0.1.0

Sep 27, 2017

Contents

1	Getting Started	3
1.1	Installation	3
1.2	A Simple Build	3
1.3	Tutorials and Screencasts	4
2	Advanced Topics	5
2.1	Packaging	5
2.2	Resolving Dependencies	7
3	Design Goals	9
3.1	Reusability	9
3.2	Portability	9
3.3	Speed	9
4	Modules	11
4.1	Interleave	11
4.2	Rigger	11
4.3	FindMe	12
4.4	ResolveMe	12
5	Partner Tools	13
5.1	Jake	13
5.2	Volo	13
6	BuildJS Roadmap	15
6.1	Use BuildJS Online	15
7	Indices and tables	17

The BuildJS Tool Suite has been created to provide a best-of-breed solution for building JS-centric web applications and libraries. While a large number of build tools already exist in the JS/web space, the BuildJS strive to have a point of difference through the *speed* of the tools.

Additionally, whereas most of the other tools are designed as larger applications, the BuildJS suite are designed with reusability in mind. There are a number of smaller *modules* that can be used individually, but come together in a single, convenient tool, *Interleave*.

BuildJS is a suite of [Node.js](#) libraries and command-line tools that can be used together to create a sophisticated and efficient build process for your JS application or library.

CHAPTER 1

Getting Started

To get started with BuildJS, you have two choices. You can either install the tools on your local machine and start writing builds now, or [try it online](#). This getting started guide focuses on working on your local machine as that is likely to be the most common use case.

Installation

To install the BuildJS tools locally, you will need to have [Node.js](#) running on your machine. If you don't already have node running, then head over to the [Node.js](#) site and work through the installation process.

If you do have node installed, then let's start by installing [Interleave](#):

```
npm install -g interleave
```

NOTE: On *nix/macOS systems you may need to run the above with `sudo`.

To check that interleave is in fact installed, let's just ask it for its version:

```
interleave --version
```

If everything is working fine, you will see the current version of Interleave installed, if not feel free to head over to the [mailing list](#) and ask for some help.

A Simple Build

While it's difficult to illustrate how useful Interleave using a simple example, it will help to understand some of the fundamental approaches of it and the BuildJS tool suite.

For our simple example, we will look at some code that makes use of the [String.trim](#) function. Now while this function is available in most browsers, it's not available in all so we should probably include a shim to ensure our code works as expected.

Our very simple code would look something like:

```
// shim String.trim for browsers that don't support it
if (! String.prototype.trim) {
  String.prototype.trim = function () {
    return this.replace(/^\s+|\s+$/g, '');
  };
}

function sayHello(name) {
  console.log('Hello ' + name.trim() + '!');
}
```

Now, let's move the shim into a file of it's own to keep our file as “clean” as possible:

```
//= shims/trim

function sayHello(name) {
  console.log('Hello ' + name.trim() + '!');
}
```

In the code above, the significant line is the `//= shims/trim` comment. In the BuildJS stack, single-line comments that have an equals character right after the single-line comment start characters is a *Rigger* directive.

While we focus primarily on JS code in our guide, the directives can also be applied to CSS (`/*= foo */`) and precompiled languages like CoffeeScript (`#= foo`).

To be completed

- include information on using remote includes
 - include information on using Rigger aliases
-

Tutorials and Screenscasts

While the BuildJS tool suite doesn't have a great deal of online resources yet available online, the following are definitely worth a look if you are looking to get started using parts of the BuildJS suite.

AMD Build With Grunt And Rigger

Derick Bailey has produced an excellent tutorial and screencast which covers using *Rigger*, *grunt* the *grunt-rigger* plugin to create a Backbone plugin.

<http://www.watchmecode.net/amd-builds-with-grunt>

CHAPTER 2

Advanced Topics

Interleave is a very powerful tool and is designed to provide both web application and library developers functionality that will make their life easier. This section of the documentation provides guides on how to use aspects of Interleave and the BuildJS tool suite to do this.

Packaging

One of the awesome features of Interleave is that it can be used to generate different module specific ([CommonJS](#), [AMD](#), etc) versions of your library from a single source file. This is not the default functionality of Interleave, but can be enabled by specifying a `--wrap` option when running your build.

One of the primary reasons to consider using Interleave's packaging is to enable to reuse other library code without having to get stressed about implementing module specific `require`, `define`, etc calls to include the required libraries. This is something that Interleave will take care of for you in the packaged files.

To specify a dependency, you can simply use the *FindMe* module requirement syntax:

```
// req: underscore as _
```

or to request a particular version:

```
// req: underscore 1.3.x as _
```

A Packaging Example

Let's work through a small, somewhat contrived example of a library that is designed for 'fonging' (a term from the movie *A Knight's Tale*). Now, `fong.js` is going to need `underscore` to operate correctly, so we'll need to include a `// req: comment`:

```
// req: underscore as _  
function fong(crowd) {  
    // find valid, fongable targets
```

```
var targets = _.filter(crowd, function(target) {
    return (target.name === 'Chaucer' || target.annoying) &&
           typeof target.kick == 'function';
});

// kick each of the valid targets
_.invoke(targets, 'kick');

// return the kicked targets
return targets;
}
```

While this is an extremely trivial and contrived example it helps to demonstrate a few key points:

- There are no CommonJS *require* statements in the library to include underscore.
- There are no export mechanisms (globbing, module.exports or AMD define) to expose the functionality of this module.
- The code is not wrapped in a closure, thus you might think the generated code would horribly pollute the global scope (in a more complicated example).

Building our Packages

Building our appropriately packaged, distribution files is very simple:

```
interleave build src/*.js --wrap
```

Using the `--wrap` option with no arguments instructs Interleave that you want to generate packages for the common package types (amd, commonjs and glob). If you would like to generate only some of these package types you can specify `--wrap=amd,commonjs` or something similar.

AMD

```
define('fong', ['underscore'], function(_) {
    function fong(crowd) {
        // find valid, fongable targets
        var targets = _.filter(crowd, function(target) {
            return (target.name === 'Chaucer' || target.annoying) &&
                   typeof target.kick == 'function';
        });

        // kick each of the valid targets
        _.invoke(targets, 'kick');

        // return the kicked targets
        return targets;
    }

    return typeof fong != 'undefined' ? fong : undefined;
});
```

CommonJS (Node)

```
var _ = require('underscore');

function fong(crowd) {
  // find valid, fongable targets
  var targets = _.filter(crowd, function(target) {
    return (target.name === 'Chaucer' || target.annoying) &&
      typeof target.kick == 'function';
  });

  // kick each of the valid targets
  _.invoke(targets, 'kick');

  // return the kicked targets
  return targets;
}

if (typeof fong != 'undefined') {
  module.exports = fong;
}
```

Globbering

```
// req: underscore as _
(function(glob) {
  function fong(crowd) {
    // find valid, fongable targets
    var targets = _.filter(crowd, function(target) {
      return (target.name === 'Chaucer' || target.annoying) &&
        typeof target.kick == 'function';
    });

    // kick each of the valid targets
    _.invoke(targets, 'kick');

    // return the kicked targets
    return targets;
  }

  if (typeof fong != 'undefined') {
    glob.fong = fong;
  }
})(this);
```

Resolving Dependencies

To be completed.

CHAPTER 3

Design Goals

The BuildJS suite has been built with the following goals:

Reusability

To be completed.

Portability

To be completed.

Speed

To be completed.

CHAPTER 4

Modules

There are a number of modules that make up the BuildJS Tool Suite. Each of these modules has a number of configuration options and APIs for working with that particular tool. This section of the guide is designed to provide the information required to “dive deep” into any of these tools.

Interleave

If BuildJS was a band, then Interleave would be the frontman. It’s likely to be the only tool that you actually need to install, as other parts of the suite will wired in as needed.

Installation

If you are working with Interleave for standalone command line usage, then it is recommended that you install it globally using the `-g` option:

```
npm install -g interleave
```

In cases where you are working with Interleave for a more complicated build and are bringing in friends such as *Jake* to help out, then you should probably include it in a `package.json devDependencies` section instead.

Source Code

<https://github.com/buildjs/interleave>

Rigger

To be completed.

Installation

```
npm install rigger
```

Source Code

<https://github.com/buildjs/rigger>

FindMe

To be completed.

Installation

```
npm install findme
```

Source Code

<https://github.com/buildjs/findme>

ResolveMe

To be completed.

Installation

```
npm install resolveme
```

Source Code

<https://github.com/buildjs/resolveme>

CHAPTER 5

Partner Tools

Jake

To be completed.

Volo

To be completed.

BuildJS Roadmap

Use BuildJS Online

This feature is coming soon, but suffice to say because the BuildJS suite has been built following Node's async recommended approach this is possible :)

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`