(/)

Configuring tasks

This guide explains how to configure tasks for your project using a `Gruntfile`. If you don't know what a `Gruntfile` is, please read the Getting Started (/getting-started/) guide and check out a Sample Gruntfile (/sample-gruntfile/).

Grunt Configuration

Task configuration is specified in your `Gruntfile` via the `grunt.initConfig` method. This configuration will mostly be under task-named properties, but may contain any arbitrary data. As long as properties don't conflict with properties your tasks require, they will be otherwise ignored.

Also, because this is JavaScript, you're not limited to JSON; you may use any valid JavaScript here. You may even programmatically generate the configuration if necessary.

```
grunt.initConfig({
  concat: {
    // concat task configuration goes here.
  },
  uglify: {
    // uglify task configuration goes here.
  },
  // Arbitrary non-task-specific properties.
  my_property: 'whatever',
  my_src_files: ['foo/*.js', 'bar/*.js'],
});
```

Task Configuration and Targets

When a task is run, Grunt looks for its configuration under a property of the same name. Multi-tasks can have multiple configurations, defined using arbitrarily named "targets." In the example below, the `concat` task has `foo` and `bar` targets, while the `uglify` task only has a `bar` target.

```
grunt.initConfig({
  concat: {
    foo: {
      // concat task "foo" target options and files go here.
    },
    bar: {
      // concat task "bar" target options and files go here.
    },
  },
  uglify: {
    bar: {
      // uglify task "bar" target options and files go here.
    },
  },
});
```

Specifying both a task and target like `grunt concat:foo` or `grunt concat:bar` will process just the specified target's configuration, while running `grunt concat` will iterate over *all* targets, processing each in turn. Note that if a task has been renamed with grunt.task.renameTask (/grunt.task#grunt.task.renametask), Grunt will look for a property with the *new* task name in the config object.

Options

Inside a task configuration, an `options` property may be specified to override built-in defaults. In addition, each target may have an `options` property which is specific to that target. Target-level options will override task-level options.

The `options` object is optional and may be omitted if not needed.

```
grunt.initConfig({
  concat: {
    options: {
      // Task-level options may go here, overriding task defaults.
    },
    foo: {
      options: {
        // "foo" target options may go here, overriding task-level options.
      },
    },
    bar: {
      // No options specified; this target will use task-level options.
    },
  },
});
```

Files

Because most tasks perform file operations, Grunt has powerful abstractions for declaring on which files the task should operate. There are several ways to define **src-dest** (source-destination) file mappings, offering varying degrees of verbosity and control. Any multi task will understand all the following formats, so choose whichever format best meets your needs.

All files formats support `src` and `dest` but the "Compact" and "Files Array" formats support a few additional properties:

- `filter` Either a valid fs.Stats method name (http://nodejs.org/docs/latest/api/fs.html#fs_class_fs_stats) or a function that is passed the matched `src` filepath and returns `true` or `false`.
- `nonull` If set to `true` then the operation will include non-matching patterns. Combined with grunt's `--verbose` flag, this option can help debug file path issues.
- `dot` Allow patterns to match filenames starting with a period, even if the pattern does not explicitly have a period in that spot.
- `matchBase` If set, patterns without slashes will be matched against the basename of the path if it contains slashes. For example, a?b would match the path `/xyz/123/acb`, but not `/xyz/acb/123`.
- `expand` Process a dynamic src-dest file mapping, see "Building the files object dynamically" (/configuring-tasks#building-the-files-object-dynamically) for more information.
- Other properties will be passed into the underlying libs as matching options. See the node-glob (https://github.com/isaacs/node-glob) and minimatch (https://github.com/isaacs/minimatch) documentation for more options.

Compact Format

This form allows a single **src-dest** (source-destination) file mapping per-target. It is most commonly used for read-only tasks, like grunt-contrib-jshint (https://github.com/gruntjs/grunt-contrib-jshint), where a single `src` property is needed, and no `dest` key is relevant. This format also supports additional properties per src-dest file mapping.

```
grunt.initConfig({
  jshint: {
    foo: {
      src: ['src/aa.js', 'src/aaa.js']
    },
  },
  concat: {
    bar: {
      src: ['src/bb.js', 'src/bbb.js'],
      dest: 'dest/b.js',
    },
  },
});
```

Files Object Format

This form supports multiple src-dest mappings per-target, where the property name is the destination file, and its value is the source file(s). Any number of src-dest file mappings may be specified in this way, but additional properties may not be specified per mapping.

```
grunt.initConfig({
  concat: {
    foo: {
      files: {
        'dest/a.js': ['src/aa.js', 'src/aaa.js'],
        'dest/a1.js': ['src/aa1.js', 'src/aaa1.js'],
      },
    },
    bar: {
      files: {
        'dest/b.js': ['src/bb.js', 'src/bbb.js'],
        'dest/b1.js': ['src/bb1.js', 'src/bbb1.js'],
      },
    },
  },
});
```

Files Array Format

This form supports multiple src-dest file mappings per-target, while also allowing additional properties per mapping.

```
grunt.initConfig({
  concat: {
    foo: {
      files: [
        {src: ['src/aa.js', 'src/aaa.js'], dest: 'dest/a.js'},
        {src: ['src/aa1.js', 'src/aaa1.js'], dest: 'dest/a1.js'},
      ],
    },
    bar: {
      files: [
        {src: ['src/bb.js', 'src/bbb.js'], dest: 'dest/b/', nonull: true},
        {src: ['src/bb1.js', 'src/bbb1.js'], dest: 'dest/b1/', filter: 'isFile'},
      ],
    },
  },
});
```

Older Formats

The **dest-as-target** file format is a holdover from before multi tasks and targets existed, where the destination filepath is actually the target name. Unfortunately, because target names are filepaths, running `grunt task:target` can be awkward. Also, you can't specify target-level options or additional properties per src-dest file mapping.

Consider this format deprecated, and avoid it where possible.

```
grunt.initConfig({
  concat: {
    'dest/a.js': ['src/aa.js', 'src/aaa.js'],
    'dest/b.js': ['src/bb.js', 'src/bbb.js'],
  },
});
```

Custom Filter Function

The `filter` property can help you target files with a greater level of detail. Simply use a valid fs.Stats method name (http://nodejs.org/docs/latest/api/fs.html#fs_class_fs_stats). The following will clean only if the pattern matches an actual file:

```
grunt.initConfig({
  clean: {
    foo: {
      src: ['tmp/**/*'],
      filter: 'isFile',
    },
  },
});
```

Or create your own `filter` function and return `true` or `false` whether the file should be matched. For example the following will only clean folders that are empty:

```
grunt.initConfig({
  clean: {
    foo: {
      src: ['tmp/**/*'],
      filter: function(filepath) {
        return (grunt.file.isDir(filepath) && require('fs').readdirSync(filepath).length ===
0);
      },
    },
  },
});
```

Globbing patterns

It is often impractical to specify all source filepaths individually, so Grunt supports filename expansion (also know as globbing) via the built-in node-glob (https://github.com/isaacs/node-glob) and minimatch (https://github.com/isaacs/minimatch) libraries.

While this isn't a comprehensive tutorial on globbing patterns, know that in a filepath:

- `*` matches any number of characters, but not `/`
- `?` matches a single character, but not `/`
- `**` matches any number of characters, including `/`, as long as it's the only thing in a path part
- `{}` allows for a comma-separated list of "or" expressions
- `!` at the beginning of a pattern will negate the match

All most people need to know is that `foo/*.js` will match all files ending with `.js` in the `foo/` subdirectory, but `foo/**/*.js` will match all files ending with `.js` in the `foo/` subdirectory *and all of its subdirectories*.

Also, in order to simplify otherwise complicated globbing patterns, Grunt allows arrays of file paths or globbing patterns to be specified. Patterns are processed in-order, with `!`-prefixed matches excluding matched files from the result set. The result set is uniqued.

For example:

```
// You can specify single files:
{src: 'foo/this.js', dest: ...}
// Or arrays of files:
{src: ['foo/this.js', 'foo/that.js', 'foo/the-other.js'], dest: ...}
// Or you can generalize with a glob pattern:
{src: 'foo/th*.js', dest: ...}

// This single node-glob pattern:
{src: 'foo/{a,b}*.js', dest: ...}
// Could also be written like this:
{src: ['foo/a*.js', 'foo/b*.js'], dest: ...}

// All .js files, in foo/, in alpha order:
{src: ['foo/*.js'], dest: ...}
// Here, bar.js is first, followed by the remaining files, in alpha order:
{src: ['foo/bar.js', 'foo/*.js'], dest: ...}

// All files except for bar.js, in alpha order:
{src: ['foo/*.js', '!foo/bar.js'], dest: ...}
// All files in alpha order, but with bar.js at the end.
{src: ['foo/*.js', '!foo/bar.js', 'foo/bar.js'], dest: ...}

// Templates may be used in filepaths or glob patterns:
{src: ['src/<%= basename %>.js'], dest: 'build/<%= basename %>.min.js'}
// But they may also reference file lists defined elsewhere in the config:
{src: ['foo/*.js', '<%= jshint.all.src %>'], dest: ...}
```

For more on glob pattern syntax, see the node-glob (https://github.com/isaacs/node-glob) and minimatch (https://github.com/isaacs/minimatch) documentation.

Building the files object dynamically

When you want to process many individual files, a few additional properties may be used to build a files list dynamically. These properties may be specified in both "Compact" and "Files Array" mapping formats.

`expand` Set to `true` to enable the following options:

- `cwd` All `src` matches are relative to (but don't include) this path.
- `src` Pattern(s) to match, relative to the `cwd`.
- `dest` Destination path prefix.
- `ext` Replace any existing extension with this value in generated `dest` paths.
- `extDot` Used to indicate where the period indicating the extension is located. Can take either `'first'` (extension begins after the first period in the file name) or `'last'` (extension begins after the last period), and is set by default to `'first'` *[Added in 0.4.3]*
- `flatten` Remove all path parts from generated `dest` paths.
- `rename` This function is called for each matched `src` file, (after extension renaming and flattening). The `dest` and matched `src` path are passed in, and this function must return a new `dest` value. If the same `dest` is returned more than once, each `src` which used it will be added to an array of sources for it.

In the following example, the `uglify` task will see the same list of src-dest file mappings for both the `static_mappings` and `dynamic_mappings` targets, because Grunt will automatically expand the `dynamic_mappings` files object into 4 individual static src-dest file mappings—assuming 4 files are found—when the task runs.

Any combination of static src-dest and dynamic src-dest file mappings may be specified.

```
grunt.initConfig({
  uglify: {
    static_mappings: {
      // Because these src-dest file mappings are manually specified, every
      // time a new file is added or removed, the Gruntfile has to be updated.
      files: [
        {src: 'lib/a.js', dest: 'build/a.min.js'},
        {src: 'lib/b.js', dest: 'build/b.min.js'},
        {src: 'lib/subdir/c.js', dest: 'build/subdir/c.min.js'},
        {src: 'lib/subdir/d.js', dest: 'build/subdir/d.min.js'},
      ],
    },
    dynamic_mappings: {
      // Grunt will search for "**/*.js" under "lib/" when the "uglify" task
      // runs and build the appropriate src-dest file mappings then, so you
      // don't need to update the Gruntfile when files are added or removed.
      files: [
        {
          expand: true,     // Enable dynamic expansion.
          cwd: 'lib/',      // Src matches are relative to this path.
          src: ['**/*.js'], // Actual pattern(s) to match.
          dest: 'build/',   // Destination path prefix.
          ext: '.min.js',   // Dest filepaths will have this extension.
          extDot: 'first'   // Extensions in filenames begin after the first dot
        },
      ],
    },
  },
});
```

Templates

Templates specified using `<% %>` delimiters will be automatically expanded when tasks read them from the config. Templates are expanded recursively until no more remain.

The entire config object is the context in which properties are resolved. Additionally, `grunt` and its methods are available inside templates, eg. `<%= grunt.template.today('yyyy-mm-dd') %>`.

- `<%= prop.subprop %>` Expand to the value of `prop.subprop` in the config, regardless of type. Templates like this can be used to reference not only string values, but also arrays or other objects.
- `<% %>` Execute arbitrary inline JavaScript code. This is useful with control flow or looping.

Given the sample `concat` task configuration below, running `grunt concat:sample` will generate a file named `build/abcde.js` by concatenating the banner `/* abcde */` with all files matching `foo/*.js` + `bar/*.js` + `baz/*.js`.

```
grunt.initConfig({
  concat: {
    sample: {
      options: {
        banner: '/* <%= baz %> */\n',   // '/* abcde */\n'
      },
      src: ['<%= qux %>', 'baz/*.js'],  // [['foo/*.js', 'bar/*.js'], 'baz/*.js']
      dest: 'build/<%= baz %>.js',      // 'build/abcde.js'
    },
  },
  // Arbitrary properties used in task configuration templates.
  foo: 'c',
  bar: 'b<%= foo %>d', // 'bcd'
  baz: 'a<%= bar %>e', // 'abcde'
  qux: ['foo/*.js', 'bar/*.js'],
});
```

Importing External Data

In the following Gruntfile, project metadata is imported into the Grunt config from a `package.json` file, and the grunt-contrib-uglify plugin (http://github.com/gruntjs/grunt-contrib-uglify) `uglify` task is configured to minify a source file and generate a banner comment dynamically using that metadata.

Grunt has `grunt.file.readJSON` and `grunt.file.readYAML` methods for importing JSON and YAML data.

```
grunt.initConfig({
  pkg: grunt.file.readJSON('package.json'),
  uglify: {
    options: {
      banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
    },
    dist: {
      src: 'src/<%= pkg.name %>.js',
      dest: 'dist/<%= pkg.name %>.min.js'
    }
  }
});
```

Found an error in the documentation? File an issue (https://github.com/gruntjs/grunt-docs/issues).