COMPOSABILITY

> “Composability is a way to combine smaller parts to make one large thing. Sort of like Voltron® (http://25.media.tumblr.com/tumblr_m1zllfCJV21r8gq9go11_250.gif)

Yeoman offers multiple ways for generators to build upon common ground. There’s no sense in rewriting the same functionality, so an API is provided to use generators inside other generators.

In Yeoman, composability can be initiated in two ways:

- A generator can decide to compose itself with another generator (e.g., `generator-backbone` uses `generator-mocha`).
- An end user may also initiate the composition (e.g., Simon wants to generate a Backbone project with SASS and Rails). Note: end user initiated composition is a planned feature and currently not available.

`this.composeWith()`

The `composeWith` method allows the generator to run side-by-side with another generator (or subgenerator). That way it can use features from the other generator instead of having to do it all by itself.

When composing, don’t forget about the running context and the run loop (/authoring/running-context.html). On a given priority group execution, all composed generators will execute functions in that group. Afterwards, this will repeat for the next group. Execution between the generators is the same order as `composeWith` was called, see execution example.

API

`composeWith` takes two parameters.

1. `generatorPath` - A full path pointing to the generator you want to compose with (usually using `require.resolve()`).
2. `options` - An Object containing options to pass to the composed generator once it runs.

When composing with a `peerDependencies` generator:

```
this.composeWith(require.resolve('generator-bootstrap/generators/app'), {prep
```

`require.resolve()` returns the path from where Node.js would load the provided module.

Note: If you need to pass `arguments` to a Generator based on a version of `yeoman-generator` older than 1.0, you can do that by providing an `Array` as the `options.arguments` key.

Even though it is not an encouraged practice, you can also pass a generator namespace to `composeWith`. In that case, Yeoman will try to find that generator installed as a `peerDependencies` or globally on the end user system.

```
this.composeWith('backbone:route', {rjs: true});
```

composing with a Generator class

`composeWith` can also take an object as its first argument. The object should have the following properties defined:

- `Generator` - The generator class to compose with
- `path` - The path to the generator files

This will let you compose with generator classes defined in your project or imported from other modules. Passing `options` as the second argument to `composeWith` works as expected.

```
// Import generator-node's main generator
const NodeGenerator = require('generator-node/generators/app/index.js');

// Compose with it
this.composeWith({
  Generator: NodeGenerator,
  path: require.resolve('generator-node/generators/app')
});
```

execution example

```
// In my-generator/generators/turbo/index.js
module.exports = class extends Generator {
  prompting() {
    this.log('prompting - turbo');
  }

  writing() {
    this.log('writing - turbo');
  }
};

// In my-generator/generators/electric/index.js
module.exports = class extends Generator {
  prompting() {
    this.log('prompting - zap');
  }

  writing() {
    this.log('writing - zap');
  }
};

// In my-generator/generators/app/index.js
module.exports = class extends Generator {
  initializing() {
    this.composeWith(require.resolve('../turbo'));
    this.composeWith(require.resolve('../electric'));
  }
};
```

Upon running `yo my-generator`, this will result in:

```
prompting - turbo
prompting - zap
writing - turbo
writing - zap
```

You can alter the function call order by reversing the calls for `composeWith`.

Keep in mind you can compose with other public generators available on npm.

For a more complex example of composability, check out generator-generator (https://github.com/yeoman/generator-generator/blob/master/app/index.js) which is composed of generator-node (https://github.com/yeoman/generator-node).

dependencies or peerDependencies

npm allows three types of dependencies:

- `dependencies` get installed local to the generator. It is the best option to control the version of the dependency used. This is the preferred option.
- `peerDependencies` get installed alongside the generator, as a sibling. For example, if `generator-backbone` declared `generator-gruntfile` as a peer dependency, the folder tree would look this way:

```
├───generator-backbone/
└───generator-gruntfile/
```

- `devDependencies` for testing and development utility. This is not needed here.

When using `peerDependencies`, be aware other modules may also need the requested module. Take care not to create version conflicts by requesting a specific version (or a narrow range of versions). Yeoman's recommendation with `peerDependencies` is to always request *higher or equal to (>=)* or *any (*)* available versions. For example:

```
{
  "peerDependencies": {
    "generator-gruntfile": "*",
    "generator-bootstrap": ">=1.0.0"
  }
}
```

Note: as of npm@3, `peerDependencies` are no longer automatically installed. To install these dependencies, they must be manually installed: `npm install generator-yourgenerator generator-gruntfile generator-bootstrap@">=1.0.0"`