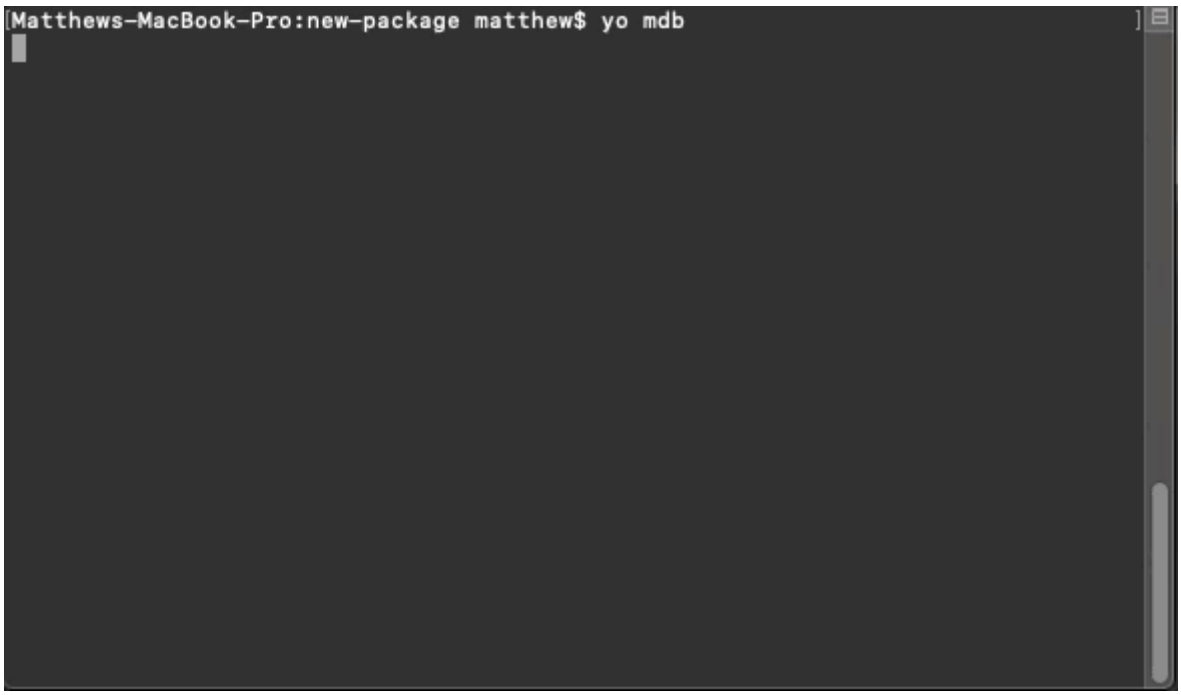
Generating code with Yeoman js

Matthew Bill
Jan 28 · 8 min read



I recently presented a talk at nor(DEV):con 2019 on generating code with Yeoman js. This article is the extended write up of that talk, with links to the corresponding resources. If you would like me to present a talk at your conference or event, then please message me through LinkedIn or Twitter.

Abstract

The abstract for the talk was the following:

> *"Fed up of writing boilerplate at work? Want to move away from that horrible mono-repo, but worried about code consistency? Find out how easy it is to use the cross-platform Yeoman js to auto-generate code and package templates in this quick 5-minute demo."*

Resources

- Generator Github Repo

- Slides

The Problem

With the continuing trend of moving away from monorepos and towards package-based solutions, we are presented with a common engineering challenge. If we are not careful, then we can end up writing the same boiler-plate code for common repository functionality time and time again, instead of writing the unique code needed for the task at hand. In short, we fail at *'DRY'*. We also have a possible issue with consistency, where we want to make sure we use the same file structure, class names, linting rules, configuration for testing, etc. This problem is further exacerbated if we move towards a microservice architecture, which I talk about in my article *'10 Practical Lessons Learn from Implementing Microservice'*.

Common Functionality

Some examples of common functionality that are required within a repository are the following:

- Version Control Files
- Package Files
- Linting
- Static Analysis
- Testing
- CI
- CD
- Documentation
- Auto Documentation
- License

This functionality must exist within the repository, even if it is just a link to a shared package of some kind. For example, our linting configuration might be within a package that we can pull down, but we still need to create a link to that package.

The Solution

One possible solution is to use a code generator, which will generate all the boiler-plate code we need so that we maintain consistency and can concentrate on writing feature-specific code. Even better yet is if we can specify certain changeable attributes, such as the name and templated files are overridden with these values. One tool that will help us do exactly this is yeoman js.

What is Yeoman

> *"Yeoman is a generic scaffolding system allowing the creation of any kind of app. It allows for rapidly getting started on new projects and streamlines the maintenance of existing projects."*
>
> *"Yeoman is language agnostic. It can generate projects in any language (Web, Java, Python, C#, etc.)"*

https://yeoman.io/learning/

Although Yeoman is essentially a Node.js package, it can be used to create code files of any language, whether they be C#, Python, Scala or plain text. It is at the end of the day, simply a tool that creates files from templates and runs the additional supporting code. Some languages have their own generators (like .NET), but you might want to consider avoiding them if you are a polyglot organisation and choose a single tool.

Why Yeoman

There are a number of different code generators on the market, so why use Yeoman over the others. Some of the main reasons, which I think are important are the following:

- Widely adopted
- Cross-platform
- Easy to use
- Great documentation
- Most people know JavaScript

Installing Yeoman

Yeoman is installed using npm. If this is not already installed on your machine, you will need to install node, which npm is bundled with. You can find out if you have npm installed by running the following command:

```
npm ––version
```

> *Note: for users more familiar with Node.js, the 'npm doctor' command is a great way to provide even more information on your installation.*

Once you have npm installed, you can install Yeoman globally and add it to your PATH by running the following command:

```
npm install yo –g
```

You are now set up to use Yeoman, but you don't have any installed generators. All Yeoman generators are also npm packages and need to be installed globally to be used. To see which generators you have installed, you can run:

```
yo
```

Many organisations that produce open source tools and libraries make use of Yeoman to create examples or get people started. You can explore different generators by visiting the Yeoman website. A great and perhaps unexpected example is the code generator by Microsoft, which is used for creating vs code extensions

To run a generator, you can either select if after running the *'yo'* command or call directly using the following command:

```
yo {name}
```

You might also like to think about using Yeoman externally for consumers of your tools as well as internally. Something else that you might find useful personally is to create

your own generator as it comes in very useful for technical tests during an interview process.

Bare Bones Generator

So let's get started creating our own generator. This is actually quite an easy process and doesn't require that many changes. We will start with something simple, but if you explore the Yeoman API further you will find it is quite rich with features.

Step #1: Create Package.Json

The way that Yeoman discovers templates is by a naming convention of the package, which is stored in the *'package.json'* file. So first of all, create your package.json file using *'npm init'* and choose a name.

```
generator-{name}
```

If using a private registry for your node packages, you can still use a scope:

```
@{scope}/generator-{name}
```

You will want your *'package.json'* file to look something like the following:

```
{
    "name": "generator-{name}",
    "version": "0.1.0",
    "description": "",
    "files": [
        "generators"
    ],
    "keywords": ["yeoman-generator"],
    "dependencies": {
        "yeoman-generator": "{version}"
    }
}
```

package.json hosted with ❤ by GitHub view raw

You will want to make sure to install *'yeoman-generator'* at the latest version when installing it:

```
npm install yeoman-generator --save
```

By using the keyword *'yeoman-generator'* and publishing your generator to the official npm registry, your generator will be automatically discoverable on the Yeoman website.

Step #2: Add Files

Next, we need some files. The files section in our package.json file instructs Yeoman where to look for generators. In our example, this is the generators folder. Our file structure wants to end up something like this (we will edit index.js in the next step).

```
├──── package.json
└──── generators/
  ├──── app/
  │  └──── index.js
  │  ├──── templates
  │  │  └──── {files to copy}
```

Yeoman by default will look for the index.js file inside of the *'app'* folder when running the generator using the *'yo'* command. The minimum we need in the index.js file is the following:

```
1  const Generator = require('yeoman-generator');
2
3  module.exports = class extends Generator {
```

```
};
```

index.js hosted with ❤ by GitHub view raw

Notice that our generator class inherits from the yeoman-generator module we installed earlier.

One thing to watch out for is that a .gitignore will not be included in an npm package when creating and uploading it. This can be a problem if you want to generate a .gitignore file, which is a common thing to do. A simple and easy way around this is just to name your .gitignore file something different (such as .gitignore-template) and when you copy it over to give it the right name.

Step #3: User Input

When a user runs your generator, you might need to get certain values to determine what files to output or to inject into the file templates. This is easily done with Yeoman and when a user runs the generator, they will be asked for this as input.

```
async prompting() {
    const self = this;
    self.answers = await self.prompt([{
        type: 'input',
        name: 'name',
        message: 'Your project name',
        default: self.appname, // Default to solution folder name
      }, {
        type: 'list',
        name: 'license',
        message: 'What license should be used?',
        choices: ['UNLICENSED', 'MIT'],
        default: 'MIT',
      }, {
        type: 'confirm',
        name: 'travis',
        message: 'Would you like to enable a travis build?',
        default: true,
      }
    }]);
}
```

index.js hosted with ❤ by GitHub view raw

The type of input can be text, but also a list of options, such as what I have used for the license input. It can also be a bool using the *'confirm'* type. You might notice that this code is within a special function called *'prompting()'*. There are a number of lifecycle methods where you can put your code and the names are fairly obvious. You will discover another lifecycle method called *'writing()'* in the next step.

Step #4: Writing Files

There are two main ways to generate files. The first is a straight copy from our templates folder, which might be useful for something like a .gitignore file that doesn't change.

```
writing() {
    const self = this;
    self.fs.copy(
        self.templatePath('.eslintignore'),
        self.destinationPath('.eslintignore'),
    );
}
```

index.js hosted with ❤ by GitHub view raw

The second is by taking a template and passing variables into it, such as user inputs or a date.

```
self.fs.copyTpl(
    self.templatePath('README.md'),
    self.destinationPath('README.md'),
    { name: self.answers.name, description: self.answers.description    },
);
```

index.js hosted with ❤ by GitHub view raw

Within your templates, you can then use the '*<%= {variable}%>*' notation to inject values into it. Here is an example of a simple README.md template:

```
# <%= name %>

<%= description %>

## Install
```

```
npm install <%= name %>
```

or

```
npm install <%= name %> - save
```

README.md hosted with ❤ by GitHub view raw

You might also want to only generate files based on user input.

```
if (self.answers.jest) {
    self.fs.copyTpl(
        self.templatePath('jest.config.js'),
        self.destinationPath('jest.config.js'),
    );
}
```

index.js hosted with ❤ by GitHub view raw

Step #5: Testing

Before uploading your new generator to the npm registry or your own private one, you are going to want to test it to make sure it behaves correctly. This can easily be done by using npm's link command at the root of your generator.

```
npm link
```

Create a new folder where you want to create a new solution using your generator and then run the following command from the root of that folder:

```
yo
```

Step #6: Publishing

That's it! All there is to do now is to publish your package to the npm registry or your own private one. Your generator is now ready to be used by anyone, by installing it globally using the following command:

```
npm install generator-{name} -g
```

Now if you run *'yo'*, it should list your generator and you are good to go. If you don't see it, then check the name of your package in the *'package.json'* file to make sure it follows the correct convention.

What Next?

There is so much more we could do with our generator. Given that it is just Node.js you can essentially create anything you can think of using Yeoman. This includes interacting with other API's, such as Github and setting up supporting DevOps tools. Just a few things that might be useful are:

- Install dependencies
- Initialize local git repo
- Creating the remote repo
- Setting up the CI/CD pipeline

This will take your tool from being just a code generator to a one that creates everything needed for a component. You can also use *'this.spawnCommand'* to run another command line tool.

```
class extends Generator {
    install() {
        this.spawnCommand('composer', ['install']);
    }
}
```

index.js hosted with ❤ by GitHub view raw

This is extremely useful for microservice architectures, where you want to be able to create new services as quickly as possible.

. . .

Please share with all your friends on social media and hit that clap button below to spread the word. Leave a response of your yeoman

generator. 👏

If you liked this post, then please follow me and check out some of my other articles.

. . .

About

Matthew Bill is a passionate technical leader and agile enthusiast from the UK. He enjoys disrupting the status quo to bring about transformational change and technical excellence. With a strong technical background (Node.js/.NET), he solves complex problems by using innovative solutions and excels in implementing strong DevOps cultures.

He is an active member of the tech community, writing articles, presenting talks, contributing to open source and co-founding the Norwich Node User Group. If you would like him to speak at one of your conferences or write a piece for your publication, then please get in touch.

Find out more about Matthew and his projects at matthewbill.gihthub.io

Thanks for reading!

. . .

Other Node.js posts you might ❤️