rolling your own cli tools

Most modern libraries and frameworks have a command line interface that will make it easier to build your own applications using the particular framework. Look at Angular CLI or Polymer CLI for an idea of what I'm talking about.

The idea is that we can create a tool like a Yeoman generator without having to learn the way Yeoman works.

Getting started

To start create a `package.json` file.

```
npm init
```

We'll install the application's dependencies next. Note that as of NPM 5.0 we don't need to indicate we're saving them as dependecies (`--save`) as this is assumed when you install a package.

```
npm install caporal colors prompt shelljs
```

Edit the package.json file so that it looks like the exmaple below.

Pay special attention to the `bin` section of the package. we've created a scaffold script that will run `index.js`.

```
{
  "name": "scaffold""scaffold""version".0",
  "main": "ind": "s",
  "bin": {
 "bin""bin"ld": "in": ""scaffold": "index.js"
  },
  "dependencies""^0.3.0""^0.3.0""caporal": "^0.3.0",
    "colors": "^1.1.2",
    "prompt""^0.7.7"
```

```
  }
}
"^0.7.7""shelljs": "^0.7.7"
  }
}
```

Buildng the app

At a minimum Caporal requires:

- a command
- one or more arguments
- an action to execute when we use the command

The example below will take a template and, optionally, a variant of the template we want to create. It will log the arguments and options to the console.

```
#!/usr/bin/env node

const prog = require('caporal');

prog
  .version('1.0.0')
  .command('create', 'Create a new application')
  .argument('<template>', 'Template to use')
  .option('--variant <variant>', 'Which <variant> of the template we\'ll c
  .action((args, options, logger) => {
    console.log({
      args: args,
      options: options
    });
  });

prog.parse(process.argv);
```

To test the program use NPM's link to add the project to our global npm space without having to publish it to the NPM registry and download it back to our

development workstation.

```
npm link
```

This will put your scripts in your path and allow you to run your script without prepending the full path.

```
scaffold --help
```

If we run the full command:

```
scaffold create node --variant mvc
```

We should get this in response:

```
{ args: { template: 'node' }, options: { variant: 'mvc' } }
```

I know, we don't really need to log the data we've just entered so we can change the action to point to a separate file (discussed below)

```
#!/usr/bin/env node

const prog = require('caporal');
const createCmd = require('./lib/create');

prog
  .version('0.0.1')
  .command('create', 'Create a new element')
  .argument('<element>', 'Element template to use')
  .defaultValue('publish-element')
  .option('--element-name <name>', 'What element we want to create')
  .defaultValue('my-element')
  .action(createCommand);

prog.parse(process.argv);
```

The `create.js` file host all the application logic for the create command and is isolated from other commands and the Caporal logic itself. We're using Shelljs to run Unix commands in our project and prompt to get user input.

The commands will prompt the user, replace placeholder elements, and copy content from our template to the elements we create.

```
const prompt = require('prompt');
const shell = require('shelljs');
const fs = require('shelljs'nst colors = require("colors/safe");

"colors/safe"// Set prompt as bluege = colors.blue("Replace");

// Command function

module.exports = (args, options, logger) => {
  const variant = options.variant || 'default';
  const elementPath = `${__dirname}/../e"Replace"{args.element}/${variant}
  const localPath = process.cwd();
  // File variables
  const variables = require(`${elementPath}/_variables`);
`${__dirname}/../elements/${args.element}/${variant}`xist, bail
  if !(fs.existsSync(elementPath)) {
    logger.error(`The requested element for ${args.element} wasn't found.
    process.exit(1);
  } else {
    // otherwise copy the files
    logger.info('Copying files…');
    shell.cp('-R', `${elementPath'Copying files…'`The requested element fo
    logger.info('✓ The files have been copied!');
  }

  // Remove variables file from the current directory
  // since we only need it on the template directory
  if (fs.existsSync(`${localPath}/_variables.js`)) {
    shell.rm(`${localPath}/_variables.js`);
  }

  logger.info('Please fill the following values…');
```

```
  // Ask for variable values
  prompt.start().get(variables, (err, result) => {

    // Remove MIT License file if another is selected
    if (result.license !== 'MIT') {
      shell.rm(`${localPath}/LICENSE`);
    }

    // Replace variable values in all files
    shell.ls('-Rl', '.').forEach(entry => {
      if (entry.isFile()) {
        // Replace '[VARIABLE]` with the variable value
        // from the prompt
        variables.forEach(variable => {
          shell.sed('-i', `\\[${variable.toUpperCase()}\\]`, \
          result[variable], entry.name);
        });

        // Insert current year in files
        shell.sed('-i', '\\[YEAR\\]', new Date().getFullYear(), \
        entry.name);
      }
    });

    logger.info('✓ Success!');
  });
}
```

Building templates

Now that we have the logic for creating elements, let's look at creating the templates we'll use as the original for each element.

Each element has its own template and we may have more than one set of templates for our application. The structure may look like this tree.

```
.
└─ elements
  └─ my-element
  └── publish-video
  └── publish-content
  └── publish-grid
└─ application
```

We'll look at the `my-element` default that will act as our default template. Each element will mirror the structure of this default element. The directory has the following structure:

```
.
├── LICENSE
├── _variables.js
├── lib
├── package.json
├── myElement.js
```

- **LICENSE** is our element's license. MIT by default
- **_varaiables.js** contains the items
- **llib** is a directory to hold any additional files we need
- **package.json**
- **myElement.js**

`_variables.js` contains the variables we want to replace in our elements. The replacement and processing is handled in the create script so we'll just present the file as is.

```
//  Variables to replace
//
//  They are asked to the user as they appear here.
//  User input will replace the placeholder values
//  in the template files

module.exports = [
  'name',
```

```
  'version',
  'description',
  'author',
  'license'
];
'name'
```

The **`package.json`** files is an example of what the template looks like. Variables to be replaced match the names in `_variables.js` in square brackets `[]`.

```
 {
  "name": "[NAME]",
 "[NAME]""version": "[VERSION]",
  "description"CRIPTION]",
  "main": "serv": "s",
  "scripts": "scripts""scripts"o \"Error: no test specified\" && exit 1",
    "start": "node serve"Error: no test specified\" && exit 1""start" },
  "author": "[AUTHOR": ""author""start:dev": "nodemon server.js"
  },
  "author"v": "^2.0.0",
  ": ""license": "[LICENSE]",
  "dependencies": {
    "dotenv": "^2.0.0",
    "hapi"": "^1.11.0"
  }
": ""hoek": "^4.1.0"
  },
  "devDependencies": {
    "nodemon": "^1.11.0"
  }
}
```

`myElement.js` is the core of the template. We create a V1 Custom Element and take advantage of features in the specification, working with observed attributes and reflecting changes in the attributes to the code and vice versa.

> Custom Elements V1 use only ES2015 exclusively.

```
class myElement extends HTMLElement {

  static get observedAttributes() {
    return ['disabled'];
  }
  // A getter/setter for a disabled property.
  get disabled() {
    return this.hasAttribute('disabled');
  }

  set disabled(val) {
    if (val) {
      this.setAttribute('disabled', '');
    } else {
      this.removeAttribute('disabled');
'disabled'
  // Can define constructor arguments if you wish.
  constructor() {
    // If you define a constructor, always call super() first!
    // This is specific to CE and required by the spec.
    super();

  }

  attributeChangedCallback(name, oldValue, newValue) {
    // When the drawer is disabled, update keyboard/screen reader behavior
    if (this.disabled) {
      this.setAttribute('tabindex', '-1');
      this.setAttribute('aria-disabled', 'true');
    } else {
      this.setAttribute('tabindex', '0');
      this.setAttribute('aria-disabled', 'false');
    }
  }
}

'tabindex'// Associate the custom element with the class we just created
customElements.define('my-element', myElement);
```

So now that we have it all together we can publish it to NPM if se so choose, because we linked the CLI tool using NPM we can run it without having to worry about publishing it.