Getting Started With Schematic Options

This sample project builds upon the sample `getting-started` project. We will extend the schematics to provide `inputs` for the schematic to use.

Create a new project in the `schematics` folder.

```
schematics schematic --name=getting-started-with-options
```

Anatomy of a Schematic

Project Parts

package.json

- The `name` and `version` properties are required to publish to npm.
  - Use semantic versioning to update the version number.
- The `schematics` property points to the `collection.json` file.
  - This is a manifest of the schematic items in the collection.
  - You should have at least one schematic defined.

```
{
  "name": "@angularlicious/getting-started",
  "version": "0.0.0",
  "description": "A schematics",
  "keywords": [
    "schematics",
    "angularlicious",
    "matt vaughn"
  ],
  "author": "Matt Vaughn",
  "license": "MIT",
```

```
  "schematics": "./collection.json",
  "peerDependencies": {
    "@angular-devkit/core": "^7.0.5",
    "@angular-devkit/schematics": "^7.0.5",
    "@types/jasmine": "^2.6.0",
    "@types/node": "^8.0.31",
    "jasmine": "^2.8.0",
    "typescript": "~3.1.6"
  }
}
```

tsconfig.json

This configuration is used by the `tsc` compiler. The only addition to the default configuration is the `outDir`. Configure and use the `outDir` when you are ready to create a `publishable` package for your

- Use `outDir` to define the location of compiled output.
  - The compiler will only output the `*.js` files.
  - The build script will need to copy other file types:
    - package.json
    - collection.json
    - schema.json
    - schema.d.ts
    - README.md

```
{
  "compilerOptions": {
    "baseUrl": "tsconfig",
    "lib": [
      "es2018",
      "dom"
    ],
    "module": "commonjs",
    "moduleResolution": "node",
    "noEmitOnError": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitAny": true,
    "noImplicitThis": true,
    "noUnusedParameters": true,
    "noUnusedLocals": true,
    "outDir": "./../../dist/schematics/getting-started",
    "rootDir": "src/",
    "skipDefaultLibCheck": true,
    "skipLibCheck": true,
    "sourceMap": true,
    "strictNullChecks": true,
    "target": "es6",
    "types": [
      "jasmine",
      "node"
```

```
      ]
    },
    "include": [
      "src/**/*",
    ],
    "exclude": [
      "src/*/files/**/*"
    ]
  }
```

collection.json

This is the most important file for your schematic. Your schematic cannot be used without it. Think of it as the manifest for the capabilities of the schematic collection. It is collection-based - you must either have a collection of one schematic or you can add as many as you require.

Most schematic collections are related. For example, Angular has a an `angular` schematic collection that contains all of the individual schematic items, for example:

- new
- class
- enum
- component
- module

A good principle to follow is that a schematic should do only one thing and do that one thing well. The `SR` or `Single Responsibility` will allow for more complex schematic operations to be composed by smaller schematics. Therefore, you will need to put some thought into the organization and implementation of your schematics.

Use the same principles as if you were developing a library.

- The top-level attribute is the name of the collection. The name of the sample below is `schematics` - which is an unfortunate name. Because we are using Angular schematics to create a schematics collection of schematics using the schematic called schematics.

  ```
  Take time to select a good name for your schematic collection. Avoid
  using the word `schematic` or `schematics`.
  ```

- Each section contains the definition of the specific schematic item in the collection.
  - description
  - factory
  - schema
  - extends

The `factory` configuration is important. It points to the entry point of the specified schematic. At a minimum provide the value of the folder that contains the default `index.ts` file - by convention the loader will use the

`index` as the entry point.

```
// By default, collection.json is a Loose-format JSON5 format, which means it's
loaded using a
// special loader and you can use comments, as well as single quotes or no-quotes
for standard
// JavaScript identifiers.
// Note that this is only true for collection.json and it depends on the tooling
itself.
// We read package.json using a require() call, which is standard JSON.
{
  // This is just to indicate to your IDE that there is a schema for
collection.json.
  "$schema": "./node_modules/@angular-devkit/schematics/collection-schema.json",

  // Schematics are listed as a map of schematicName => schematicDescription.
  // Each description contains a description field which is required, a factory
reference,
  // an extends field and a schema reference.
  // The extends field points to another schematic (either in the same collection
or a
  // separate collection using the format collectionName:schematicName).
  // The factory is required, except when using the extends field. Then the
factory can
  // overwrite the extended schematic factory.
  "schematics": {
    "my-schematic": {
      "description": "An example schematic",
      "factory": "./my-schematic/index#mySchematic"
    },
    "my-other-schematic": {
      "description": "A schematic that uses another schematics.",
      "factory": "./my-other-schematic"
    },
    "my-full-schematic": {
      "description": "A schematic using a source and a schema to validate
options.",
      "factory": "./my-full-schematic",
      "schema": "./my-full-schematic/schema.json"
    },
    "my-extend-schematic": {
      "description": "A schematic that extends another schematic.",
      "extends": "my-full-schematic"
    }
  }
}
```

Schematic Parts

index.ts

The `index.ts` file is the entry point for the specified schematic. All schematics will require an entry point. By convention, always use `index` as the name for this file.

- The schematic will require additional tooling provided by `@angular-devkit/schematics` helpers.
  - Use the `import` to add any required helpers.
- The implementation of the `loader` is a `default` function that returns a `Rule`.
- The function allows for inputs to be passed in using the `options` value.

Notice that the type for `options` by default is `any`. This may work for simple cases or examples. However, for production or published schematics the `options` should be strongly typed using the definition of the `schema.json` file. You will need to use a tool/utility to convert the `schema` to a strongly-typed definition file (*.d.ts) so it can be used as an interface for specified options.

```
import {
  Rule,
  SchematicContext,
  SchematicsException,
  Tree,
  apply,
  filter,
  mergeWith,
  move,
  noop,
  template,
  url,
  branchAndMerge,
} from '@angular-devkit/schematics';
import { strings } from '@angular-devkit/core';
import { parseName } from '@schematics/angular/utility/parse-name'
import { getWorkspace } from '@schematics/angular/utility/config'
import { buildDefaultPath } from '@schematics/angular/utility/project'
import { WorkspaceProject } from '@schematics/angular/utility/workspace-models';

/**
 * Use to setup the target path using the specified options [project].
 * @param host the current [Tree]
 * @param options the current [options]
 * @param context the [SchematicContext]
 */
export function setupOptions(host: Tree, options: any, context: SchematicContext)
{
  const workspace = getWorkspace(host);
  if (!options.project) {
    context.logger.error(`The [project] option is missing.`);
    throw new SchematicsException('Option (project) is required.');
  }
  context.logger.info(`Preparing to retrieve the project using:
${options.project}`);
  const project = <WorkspaceProject>workspace.projects[options.project];

  if (options.path === undefined) {
    context.logger.info(`Preparing to determine the target path.`);
```

```
    options.path = buildDefaultPath(project);
    context.logger.info(`The target path: ${options.path}`);
  }

  options.type = !!options.type ? `.${options.type}` : '';

  const parsedPath = parseName(options.path, options.name);
  options.name = parsedPath.name;
  options.path = parsedPath.path;

  context.logger.info(`Finished options setup.`);
  return host;
}

export default function (options: any): Rule {
  return (host: Tree, context: SchematicContext) => {

    setupOptions(host, options, context);

    const templateSource = apply(url('./files'), [
      options.spec ? noop() : filter(path => !path.endsWith('.spec.ts')),
      template({
        ...strings,
        ...options,
      }),
      move(options.path),
    ]);

    return branchAndMerge(mergeWith(templateSource));
  };
}
```

schema.json

Notice that the signature of the schematic's entry point (`index.ts`) has an `options` parameter of type `any`. This allows the schematic developer to define properties as input options for the specified schematic.

The following options `schema` contains:

- $schema: indicates the location of the schema for the JSON file.
- id: the unique identifier for the specified schematic
- title: a displayable title for the speciified schematic
- type: defines type as `object`
- properties: a collection of properties to be used as `options` for the specified schematic
- required: a collection of property names that are required for the specified schematic
    - the schematic developer will need to provide validation for `required` options

```
{
  "$schema": "http://json-schema.org/schema",
```

```
  "id": "MyFullSchematicsSchema",
  "title": "My Full Schematics Schema",
  "type": "object",
  "properties": {
    "index": {
      "type": "number",
      "default": 1
    },
    "name": {
      "type": "string"
    }
  },
  "required": [
    "name"
  ]
}
```

Templates

In most code generation scenarios, templates are used to generate files and folders based on options and inputs. The schematics engine will need to know the location of the templates. By convention, most schematic developers put the templates in a folder called `files` - maybe `templates` would have been a better name, right?

You can name this folder anything, because you will make a reference to it by name to load the `template source` in the schematic default function. The sample code below demonstrates the retrieval of the `template` source tree using `apply` and `url` functions on the folder path of `files`. It is also filtering out unwanted files with the extension `.spec.ts` if the current `spec` flag in the options is set to `false`.

```
const templateSource = apply(url('./files'), [
      options.spec ? noop() : filter(path => !path.endsWith('.spec.ts')),
      template({
        ...strings,
        ...options,
      }),
      move(options.path),
    ]);
```

options: any[one]?

The `options` available for a specific schematic depends on the configuration of the `schema.json` file. This file contains information about the specific schematic as well as a `properties` collection that defines the types available for the options - this also includes a `required` collection, which is just a list of property names that are required for the schematic.

```
{
 "$schema": "http://json-schema.org/schema",
```

```
  "id": "MyFullSchematicsSchema",
  "title": "My Full Schematics Schema",
  "type": "object",
  "properties": {
    "index": {
      "type": "number",
      "default": 1
    },
    "name": {
      "type": "string"
    }
  },
  "required": [
    "name"
  ]
}
```

The current `schema` for the `my-full-schematic` has minimal information. Running the same schematic in the `getting-started` project will output (2) files using the value supplied for the `name` option. Following the path:

- schema has a `name` property defined as `required`
- pass in the value for the `name` option using `--name` switch with a value
- the schematic engine will use the value along with the templates
- the output will be a file with the template structure and option values

```
ng generate @angularlicious/getting-started:my-full-schematic --name="test"
```

The template (`test2`) has syntax to allow for template `text` to be side-by-side with special markup to add the option values.

```
<% if (name) { %>
  Hello <%= name %>, I'm a schematic.
<% } else { %>
  Why don't you give me your name with --name?
<% } %>
```

The output of the executed schematic says hello to the `name` value supplied in the command line `--name` switch.

```
Hello test, I'm a schematic.
```

More options are better, right?

So, how do we learn about all these options. Need some examples? Checkout the Angular Schematics used by the Angular Cll.

> See the `angular-cli` project on GitHub. The resource links are below.

The code snippets below was borrowed from the `class` schematic. See the class schematic with the index.ts and schema.json for a great example on how to create `project` and `path` options for the schematic..

> A co-worker told me that good programmers never write code that
> they can borrow or steal. I think he stole some stuff from me...
> hmmmm!

Update the `schema` to include a `path` and a `project` option. Update the `required` list to indicate which properties are *not* `optional`.

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "MyFullSchematicsSchema",
  "title": "My Full Schematics Schema",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "The name of the class.",
      "$default": {
        "$source": "argv",
        "index": 0
      }
    },
    "path": {
      "type": "string",
      "format": "path",
      "description": "The path to create the class.",
      "visible": false
    },
    "project": {
      "type": "string",
      "description": "The name of the project.",
      "$default": {
        "$source": "projectName"
      }
    },
  },
  "required": [
    "name",
    "project"
  ]
}
```

Now, we just need to do (3) things:

1. build
2. link
3. use

```
npm run build:schematics
```

```
npm link ./dist/schematics/getting-started-with-options
```

```
ng generate @angularlicious/getting-started-with-options:my-full-schematic --
name="not-optional"
```

Oh, no...there's an error! It is good to see errors sometimes, it means that things are working - there are validations and they are trying to tell us something important. Remember, that we updated the `schema` to `require` some things. And when things are required and they are not there, bad things happen, and good software tells you about them. Here's the error message:

```
ng generate @angularlicious/getting-started-with-options:my-full-schematic --
name="not-optional"
Schematic input does not validate against the Schema: {"name":"not-
optional","project":null}
Errors:

Data path ".project" should be string.
```

We can fix this by updating our command with better inputs.

- name
- project

  ```
    Note: we will need to create an application:
    `ng generate @angularlicious/getting-started-with-options:my-full-
  schematic --name="not-optional"`
  ```

```
ng generate @angularlicious/getting-started-with-options:my-full-schematic --
name="not-optional" --project=web-app-with-options
```

So, it is not enough to just pass in more options. The schematic is required to handle and/or use them. This is where things go a little dark. As of this writing there a sets of utilities that simplify many operations performed by schematics. We will need (more like require) these utilities.

The current location of these [utilities](Angular Schematic Utilities) can be found here:

- https://github.com/angular/angular-cli/tree/master/packages/schematics/angular/utility

Let's install some utilities...and get the lights back on!

```
npm install -D @schematics/angular@latest
```

To use the utility functions, you will need to add `imports` from the specified utility item. You may need to peek into the utility items to determine where a specific function lives. For example:

```
import { parseName } from '@schematics/angular/utility/parse-name'
import { getWorkspace } from '@schematics/angular/utility/config'
import { buildDefaultPath } from '@schematics/angular/utility/project'
import { WorkspaceProject } from '@schematics/angular/utility/workspace-models';
```

The `index.ts` needs to be updated to use the options for the schematic. But there are some new and useful functions

- `getWorkspace(host)`: use to retrieve the `workspace` using the current `Tree` (a.k.a. host).
- `buildDefaultPath(project)`: use to build the `path` to the specified `project`.
- `parseName`: returns a [Location].

```
const workspace = getWorkspace(host);
if (!options.project) {
throw new SchematicsException('Option (project) is required.');
}
const project = workspace.projects[options.project];

if (options.path === undefined) {
options.path = buildDefaultPath(project);
}
```

Once again, the code implementation is `borrowed` from the `class` schematic. If this work as intended, we should get the `output` of the schematic in the specified `project`. Make sure to build and link the updated schematic before executing the command.

```
npm run build-schematic:getting-started-with-options
npm link ./dist/schematics/getting-started-with-options
ng generate @angularlicious/getting-started-with-options:my-full-schematic --
name="not-optional" --project=web-app-with-options
```

```
import {
  Rule,
  SchematicContext,
  SchematicsException,
  Tree,
  apply,
  filter,
  mergeWith,
  move,
  noop,
  template,
  url,
  branchAndMerge,
} from '@angular-devkit/schematics';
import { strings } from '@angular-devkit/core';
import { parseName } from '@schematics/angular/utility/parse-name'
import { getWorkspace } from '@schematics/angular/utility/config'
import { buildDefaultPath } from '@schematics/angular/utility/project'
import { WorkspaceProject } from '@schematics/angular/utility/workspace-models';

/**
 * Use to setup the target path using the specified options [project].
 * @param host the current [Tree]
 * @param options the current [options]
 * @param context the [SchematicContext]
 */
export function setupOptions(host: Tree, options: any, context: SchematicContext)
{
  const workspace = getWorkspace(host);
  if (!options.project) {
    context.logger.error(`The [project] option is missing.`);
    throw new SchematicsException('Option (project) is required.');
  }
  context.logger.info(`Preparing to retrieve the project using:
${options.project}`);
  const project = <WorkspaceProject>workspace.projects[options.project];

  if (options.path === undefined) {
    context.logger.info(`Preparing to determine the target path.`);
    options.path = buildDefaultPath(project);
    context.logger.info(`The target path: ${options.path}`);
  }

  options.type = !!options.type ? `.${options.type}` : '';

  const parsedPath = parseName(options.path, options.name);
  options.name = parsedPath.name;
  options.path = parsedPath.path;

  context.logger.info(`Finished options setup.`);
  return host;
```

```
}

export default function (options: any): Rule {
  return (host: Tree, context: SchematicContext) => {

    setupOptions(host, options, context);

    const templateSource = apply(url('./files'), [
      options.spec ? noop() : filter(path => !path.endsWith('.spec.ts')),
      template({
        ...strings,
        ...options,
      }),
      move(options.path),
    ]);

    return branchAndMerge(mergeWith(templateSource));
  };
}
```

Resources

- Hans Larsen: Schematics - An Introduction
- @angular-devkit/schematics Documentation
- @angular-devkit/schematics GitHub
- https://github.com/angular/angular-cli
- https://github.com/angular/angular-cli/blob/master/CONTRIBUTING.md
- Fork the `angular-cli` repo.
    - https://github.com/angular/angular-cli/fork
- Angular Utility Items
- Custom Schematics in an Nx Workspace
- Simple Schematic