Location contexts and paths

Yeoman file utilities are based on the idea you always have two location contexts on disk. These contexts are folders your generator will most likely read from and write to.

Destination context

The first context is the *destination context*. The destination is the folder in which Yeoman will be scaffolding a new application. It is your user project folder, it is where you'll write most of the scaffolding.

The destination context is defined as either the current working directory or the closest parent folder containing a `.yo-rc.json` file. The `.yo-rc.json` file defines the root of a Yeoman project. This file allows your user to run commands in subdirectories and have them work on the project. This ensures a consistent behaviour for the end user.

You can **get** the *destination path* using `this.destinationRoot()` or by joining a path using `this.destinationPath('sub/path')`.

```
// Given destination root is ~/projects
class extends Generator {
  paths() {
    this.destinationRoot();
    // returns '~/projects'

    this.destinationPath('index.js');
    // returns '~/projects/index.js'
  }
}
```

And you can manually set it using `this.destinationRoot('new/path')`. But for consistency, you probably shouldn't change the default destination.

If you want to know from where the user is running `yo`, then you can get the path with `this.contextRoot`. This is the raw path where `yo` was invoked from; before we determine the project root with `.yo-rc.json`.

Template context

The template context is the folder in which you store your template files. It is usually the folder from which you'll read and copy.

The template context is defined as `./templates/` by default. You can overwrite this default by using `this.sourceRoot('new/template/path')`.

You can get the path value using `this.sourceRoot()` or by joining a path using `this.templatePath('app/index.js')`.

```
class extends Generator {
  paths() {
    this.sourceRoot();
    // returns './templates'

    this.templatePath('index.js');
    // returns './templates/index.js'
  }
};
```

An "in memory" file system

Yeoman is very careful when it comes to overwriting users files. Basically, every write happening on a pre-existing file will go through a conflict resolution process. This process requires that the user validate every file write that overwrites content to its file.

This behaviour prevents bad surprises and limits the risk of errors. On the other hand, this means every file is written asynchronously to the disk.

As asynchronous APIs are harder to use, Yeoman provide a synchronous file-system API where every file gets written to an in-memory file system (https://github.com/sboudrias/mem-fs) and are only written to disk once when Yeoman is done running.

This memory file system is shared between all composed generators (/authoring/composability.html).

File utilities

Generators expose all file methods on `this.fs`, which is an instance of mem-fs editor (https://github.com/sboudrias/mem-fs-editor) - make sure to check the module documentation (https://github.com/sboudrias/mem-fs-editor) for all available methods.

It is worth noting that although `this.fs` exposes `commit`, you should not call it in your generator. Yeoman calls this internally after the conflicts stage of the run loop.

Example: Copying a template file

Here's an example where we'd want to copy and process a template file.

Given the content of `./templates/index.html` is:

```
<html>
  <head>
    <title><%= title %></title>
  </head>
</html>
```

We'll then use the `copyTpl` (https://github.com/sboudrias/mem-fs-editor#copytplfrom-to-context-templateoptions--copyoptions) method to copy the file while processing the content as a template. `copyTpl` is using ejs template syntax (http://ejs.co).

```
class extends Generator {
  writing() {
    this.fs.copyTpl(
      this.templatePath('index.html'),
      this.destinationPath('public/index.html'),
      { title: 'Templating with Yeoman' }
    );
  }
}
```

Once the generator is done running, `public/index.html` will contain:

```
<html>
  <head>
    <title>Templating with Yeoman</title>
  </head>
</html>
```

A very common scenario is to store user answers at the prompting stage (/authoring/user-interactions.html) and use them for templating:

```
class extends Generator {
  async prompting() {
    this.answers = await this.prompt([{
      type    : 'input',
      name    : 'title',
      message : 'Your project title',
    }]);
  }

  writing() {
    this.fs.copyTpl(
      this.templatePath('index.html'),
      this.destinationPath('public/index.html'),
      { title: this.answers.title } // user answer `title` used
    );
  }
}
```

Transform output files through streams

The generator system allows you to apply custom filters on every file writes. Automatically beautifying files, normalizing whitespace, etc, is totally possible.

Once per Yeoman process, we will write every modified file to disk. This process is passed through a vinyl (https://github.com/wearefractal/vinyl) object stream (just like gulp (http://gulpjs.com/)). Any generator author can register a `transformStream` to modify the file path and/or the content.

Registering a new modifier is done through the `registerTransformStream()` method. Here's an example:

```
var beautify = require("gulp-beautify");
this.registerTransformStream(beautify({ indent_size: 2 }));
```

Note that **every file of any type will be passed through this stream**. Make sure any transform stream will passthrough the files it doesn't support. Tools like gulp-if (https://github.com/robrich/gulp-if) or gulp-filter (https://github.com/sindresorhus/gulp-filter) will help filter invalid types and pass them through.

You can basically use any *gulp* plugins with the Yeoman transform stream to process generated files during the writing phase.

Tip: Update existing file's content

Updating a pre-existing file is not always a simple task. The most reliable way to do so is to parse the file AST (abstract syntax tree (http://en.wikipedia.org/wiki/Abstract_syntax_tree)) and edit it. The main issue with this solution is that editing an AST can be verbose and a bit hard to grasp.

Some popular AST parsers are:

- Cheerio (https://github.com/cheeriojs/cheerio) for parsing HTML.
- Esprima (https://github.com/ariya/esprima) for parsing JavaScript - you might be interested in AST-Query (https://github.com/SBoudrias/ast-query) which provide a lower level API to edit Esprima syntax tree.
- For JSON files, you can use the native `JSON` object methods (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON).
- Gruntfile Editor (https://github.com/SBoudrias/gruntfile-editor) to dynamically modify a Gruntfile.

Parsing a code file with RegEx is a perilous path, and before doing so, you should read this CS anthropological answers (http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags#answer-1732454) and grasp the flaws of RegEx parsing. If you do choose to edit existing files using RegEx rather than AST tree, please be careful and provide complete unit tests. - Please please, don't break your users' code.