

---

# **banana documentation**

***Release 0.1.14***

**banana**

**Apr 01, 2017**



---

## Contents

---

<b>1</b>	<b>Getting started with Banana</b>	<b>1</b>
1.1	Getting started with Banana	1
1.2	Let's Scaffold a Web App with Banana	3
1.3	Frequently Asked Questions	3
1.4	Support	5
<b>2</b>	<b>Writing Your Own Banana Generator</b>	<b>7</b>
2.1	Writing Your Own Generator	7
2.2	Generator Runtime Context	10
2.3	Asynchronous tasks	12
2.4	Interacting With The User	12
2.5	Composability	15
2.6	Managing Dependencies	18
2.7	Working With The File System	18
2.8	Managing Configuration	21
2.9	Testing Generators	23
2.10	Debugging Generators	25
2.11	Integrating Banana	25
<b>3</b>	<b>Let's Scaffold a Web App</b>	<b>29</b>
3.1	Step 1: Set up your dev environment	29
3.2	Step 2: Install a Banana generator	30
3.3	Step 3: Use a generator to scaffold out your app	30
3.4	Step 4: Review the Banana-generated app	32
3.5	Step 5: Preview your app in the browser	32
3.6	Step 6: Test with Karma and Jasmine	33
3.7	Step 7: Make Todos persistent with local storage	34
3.8	Step 8: Get ready for production	36
3.9	Congratulations!	37
<b>4</b>	<b>Contributing to the Banana Project</b>	<b>39</b>
4.1	Contributing	39
4.2	How to open a helpful issue	40
4.3	Pull Request Guidelines	41
4.4	Style Guide	42
4.5	Testing Guidelines	42
4.6	Issue System Overview	43



---

## Getting started with Banana

---

### Getting started with Banana

Banana 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.

Banana is language agnostic. It can generate projects in any language (Python, Nodejs, Ruby, Java, C#, etc.)

Banana by itself doesn't make any decisions. Every decision is made by *generators* which are basically plugins in the Banana environment. Its easy to create a new generator to match any workflow. Banana is always the right choice for your scaffolding needs.

Here are some common use cases:

- Rapidly create a new project
- Create new sections of a project, like a new controller with unit tests
- Create modules or packages
- Bootstrapping new services
- Enforcing standards, best practices and code styles
- Promote new projects by letting users get started with a sample app

### Getting started

`ba` is the Banana command line utility allowing the creation of projects utilizing scaffolding templates (referred to as generators). `Yo` and the generators used are installed using `pip`.

### Installing `ba` and some generators

First thing is to install `ba` using `pip`:

```
pip install banana
```

Then install the needed generator(s). Generators are pip packages named `banana-XYZ`. Search for them on our website or on PyPI. To install the Lambda generator:

```
pip install banana-lambda
```

## Basic scaffolding

We'll use `banana-lambda` in our examples below. Replace `lambda` with the name of your generator for the same result.

To scaffold a new project, run:

```
ba lambda
```

Most generators will ask a series of questions to customize its templates for your new project. To see which options are available, use the `help` command:

```
ba lambda --help
```

A lot of generators rely on specific tools and technologies. Make sure to visit the generator's site to learn about running and maintaining the new app. Easily access a generator's home page by running:

TODO

```
pip home banana-lambda
```

Generators scaffolding complex frameworks are likely to provide additional generators to scaffold smaller parts of a project. These generators are usually referred to as *sub-generators*, and are accessed as `generator:sub-generator`.

Take `generator-angular` as an example. Once the full angular app has been generated, other features can be added. To add a new controller to the project, run the controller sub-generator:

```
ba angular:controller MyNewController
```

## Other ba commands

Other than the basics covered in the previous section, `ba` is also a fully interactive tool. Simply typing `ba` in a terminal will provide a list of options to manage everything related to the generators: run, update, install, help and other utilities.

`ba` also provides the following commands.

- `ba --help` Access the full help screen
- `ba --generators` List every installed generators
- `ba --version` Get the version

## Creating a generator

See Authoring.

## Let's Scaffold a Web App with Banana

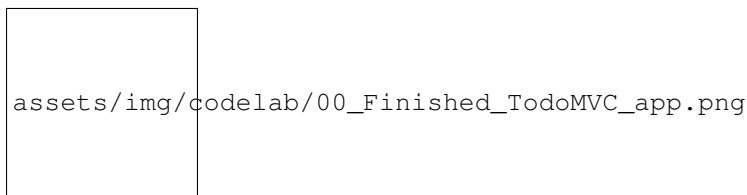
### TODO

Moin! Moin! In this 25-minutes codelab, you will build a fully functional web application from scratch with the help of Banana and [FountainJS](#). The sample app will be written in [React](#), [Angular2](#) or [Angular1](#).

Don't know any React or Angular? That's okay, we'll walk you through it. However, we do assume that you have some previous JavaScript experience.

### Build this sample app with Banana

The sample web app you'll build today will be a implementation of [TodoMVC](#). You will be able to add todos, delete todos, filter todos, and together we'll add a feature to save todos offline.



We will build the above TodoMVC app from scratch. Each step builds on the previous so go through each step one by one.

- **Step 1:** Set up your dev environment »
- **Step 2:** Install a Banana generator »
- **Step 3:** Use a generator to scaffold out your app »
- **Step 4:** Review the Banana-generated app directory structure »
- **Step 5:** Preview your app in the browser »
- **Step 6:** Test with Karma and Jasmine »
- **Step 7:** Make todos persistent with local storage »
- **Step 8:** Get ready for production »
- Like what you see? Banana can do more »

It will take approximately 25 minutes to complete this codelab. By the end, you'll have a snazzy TodoMVC app and your computer will be set up to build even more awesome web apps in the future.

## Frequently Asked Questions

*What are the goals of the project?*

*What is a command-line interface?*

*What is a package manager?*

*Will the Banana project be providing Generators for popular frameworks?*

*What license is Banana released under?*

*What should I do before submitting an issue through GitHub?*

*How can I disable Insight or Update Notifier?*

## What are the goals of the project?

The short-term goals for `Banana` are to provide developers with an improved tooling workflow so that they can spend less time on process and more time focusing on building beautiful web applications. Initially, we hope to make it as easy as possible to work with existing frameworks and tools developers are used to using.

Long-term, the project may also assist developers with creating applications using modern technologies.

## What is a command-line interface?

A command-line interface is a means for developers to interact with a system using text commands. On Linux or OSX, this is often done using the terminal. On Windows, the command shell (`cmd.exe`) or PowerShell, but we recommend you use `cmd.exe` instead for an improved experience.

## What is a package manager?

A package manager is a tool for automating the process of installing, upgrading, configuring and managing dependencies for projects. Good examples of package manager would be PyPi (Python), npm (Node.js), Bower (Web), Gem (Ruby), Composer (PHP), NuGet (.NET), etc.

## Will the Banana project be providing Generators for popular frameworks?

Our goal is to facilitate both developers and the community with the tools needed to create useful scaffolding for their projects. With that goal in mind, we'll be providing a great API (and docs) to the `Banana Generators` system with examples of how to implement new generators, but will rely on the community to create and maintain Generators for popular frameworks. This will allow us to focus on making `Banana` better without losing focus by maintaining a large number of Generators.

You can see the full [list of officially supported generators](#) on Github.

## What license is Banana released under?

`Banana` is released under a [MIT](#) license.

## What should I do before submitting an issue through GitHub?

Make sure you read the [Submitting an issue guide](#).

## How can I disable Insight or Update Notifier?

Currently the insight feature is not yet ported. As soon as we have the insight feature... You can use use a command line flag to disable them. Eg. `ba webapp --no-insight`

Insight: `--no-insight` Update Notifier: `--no-update-notifier`



# Support

## Getting Support

Banana provides an optimized **scaffolding** and workflow experience for creating compelling applications.

### Binary issues

For issues with the Banana binary, such as being unable to run Banana at all you should submit a bug ticket to the [Banana issue tracker](#) for further help.

### Scaffold issues

Our scaffolds (such as banana-lambda) are community-driven, with several of our default ones living under the [Fin-klabs organization](#) on GitHub. These are maintained by developers in the community around a particular framework. Issue trackers for some of our popular generators can be found below

- [WebApp](#)
- [add new ones here](#)

### Build issues

If you're having issues with your build tooling, you will need to open an issue in the issue tracker of your build tool e.g. the generators build tool.



---

# Writing Your Own Banana Generator

---

## Writing Your Own Generator

Generators are the building blocks of the `Banana` ecosystem. They're the plugins run by `ba` to generate files for end users.

In reading this section, you'll learn how to create and distribute your own.

### Organizing your generators

#### Setting up as a node module

A generator is, at its core, a Node.js module.

First, create a folder within which you'll write your generator. This folder must be named `generator-name` (where `name` is the name of your generator). This is important, as `Banana` relies on the file system to find available generators.

Once inside your generator folder, create a `setup.py` file. This file is ... or by entering the following manually:

```
{
  "name": "generator-name",
  "version": "0.1.0",
  "description": "",
  "files": [
    "generators"
  ],
  "keywords": ["yeoman-generator"],
  "dependencies": {
    "yeoman-generator": "^1.0.0"
  }
}
```

The `name` property must be prefixed by `generator-`. The `keywords` property must contain `"yeoman-generator"` and the repo must have a description to be indexed by our generators page.

You should make sure you set the latest version of `yeoman-generator` as a dependency. You can do this by running: `pip install --save yeoman-generator`.

The `files` property must be an array of files and directories that is used by your generator.

Add other `package.json` properties as needed.

## Folder tree

Banana is deeply linked to the file system and to how you structure your directory tree. Each sub-generator is contained within its own folder.

The default generator used when you call `ba name` is the `app` generator. This must be contained within the `app/` directory.

TODO: I do not think we support sub-generators! Sub-generators, used when you call `ba name:subcommand`, are stored in folders named exactly like the sub command.

In an example project, a directory tree could look like this:

```
--setup.py
--generators/
  --gen1.py
  --gen2.py
```

This generator will expose `ba name` and `ba name:router` commands.

Banana allows two different directory structures. It'll look in `./` and in `generators/` to register available generators.

The previous example can also be written as follows:

```
--setup.py
--gen1/
|   --generator.py
--gen2/
  --generator.py
```

If you use this second directory structure, make sure you point the `files` property in your `setup.py` at all the generator folders.

```
entry_points="""
    [banana10]
    gen1=gen1.generator
    gen2=gen2.generator
    """,
```

## Extending generator

TODO: not sure if we have anything here besides std. python extendability...

Once you have this structure in place, it's time to write the actual generator.

Banana offers a base generator which you can extend to implement your own behavior. This base generator will add most of the functionalities you'd expect to ease your task.

In the generator's `index.js` file, here's how you extend the base generator:

TODO

## Overwriting the constructor

TODO

Some generator methods can only be called inside the `constructor` function. These special methods may do things like set up important state controls and may not function outside of the constructor.

To override the generator constructor, add a constructor method like so:

```
module.exports = class extends Generator {
  // The name `constructor` is important here
  constructor(args, opts) {
    // Calling the super constructor is important so our generator is correctly set up
    super(args, opts);

    // Next, add your custom code
    this.option('babel'); // This method adds support for a `--babel` flag
  }
};
```

## Adding your own functionality

Every method added to the prototype is run once the generator is called—and usually in sequence. But, as we'll see in the next section, some special method names will trigger a specific run order.

Let's add some methods:

```
module.exports = class extends Generator {
  method1() {
    console.log('method 1 just ran');
  }

  method2() {
    console.log('method 2 just ran');
  }
};
```

When we run the generator later, you'll see these lines logged to the console.

## Running the generator

At this point, you have a working generator. The next logical step would be to run it and see if it works.

Since you're developing the generator locally, it's not yet available as a global pip module. A global module may be created and symlinked to a local one, using pip. Here's what you'll want to do:

On the command line, from the root of your generator project (in the `generator-name/` folder), type:

TODO

That will install your project dependencies and symlink a global module to your local file. After pip is done, you'll be able to call `ba name` and you should see the `console.log`, defined earlier, rendered in the terminal. Congratulations, you just built your first generator!

## Finding the project root

While running a generator, Banana will try to figure some things out based on the context of the folder it's running from.

Most importantly, Banana searches the directory tree for a `.ba-rc.json` file. If found, it considers the location of the file as the root of the project. Behind the scenes, Banana will change the current directory to the `.ba-rc.json` file location and run the requested generator there.

The Storage module creates the `.ba-rc.json` file. Calling `this.config.save()` from a generator for the first time will create the file.

So, if your generator is not running in your current working directory, make sure you don't have a `.ba-rc.json` somewhere up the directory tree.

## Where to go from here?

After reading this, you should be able to create a local generator and run it.

If this is your first time writing a generator, you should definitely read the next section on running context and the run loop. This section is vital to understanding the context in which your generator will run, and to ensure that it will compose well with other generators in the Banana ecosystem. The other sections of the documentation will present functionality available within the Banana core to help you achieve your goals.

## Generator Runtime Context

TODO One of the most important concepts to grasp when writing a Generator is how methods are running and in which context.

## Prototype methods as actions

Each method directly attached to a Generator prototype is considered to be a task. Each task is running in sequence by the Banana environment run loop.

In other words, each function on the object returned by `Object.getPrototypeOf(Generator)` will be automatically run.

## Helper and private methods

Now that you know the prototype methods are considered to be a task, you may wonder how to define helper or private methods that won't be called automatically. There are three different ways to achieve this.

1. Prefix method name by an underscore (e.g. `_private_method`).

```
class extends Generator {
  method1() {
    console.log('hey 1');
  }
}
```

```

    _private_method() {
      console.log('private hey');
    }
  }
}

```

## 2. Use instance methods:

```

class extends Generator {
  constructor(args, opts) {
    // Calling the super constructor is important so our generator is correctly_
    ↪set up
    super(args, opts)

    this.helperMethod = function () {
      console.log('won\'t be called automatically');
    };
  }
}

```

## 3. Extend a parent generator:

```

class MyBase extends Generator {
  helper() {
    console.log('methods on the parent generator won\'t be called automatically
    ↪');
  }
}

module.exports = class extends MyBase {
  exec() {
    this.helper();
  }
};

```

## The run loop

Running tasks sequentially is alright if there's a single generator. But it is not enough once you start composing generators together.

That's why Banana uses a **run loop**.

The run loop is a queue system with priority support. We use the [Grouped-queue](#) module to handle the run loop.

Priorities are defined in your code as special prototype method names. When a method name is the same as a priority name, the run loop pushes the method into this special queue. If the method name doesn't match a priority, it is pushed in the default group.

In code, it will look this way:

```

class extends Generator {
  priorityName() {}
}

```

You can also group multiple methods to be run together in a queue by using a hash instead of a single method:

```
Generator.extend({
  priorityName: {
    method() {},
    method2() {}
  }
});
```

(Note that this last technique doesn't play well with JS `class` definition)

The available priorities are (in running order):

1. `initializing` - Your initialization methods (checking current project state, getting configs, etc)
2. `prompting` - Where you prompt users for options (where you'd call `this.prompt()`)
3. `configuring` - Saving configurations and configure the project (creating `.editorconfig` files and other metadata files)
4. `default` - If the method name doesn't match a priority, it will be pushed to this group.
5. `writing` - Where you write the generator specific files (routes, controllers, etc)
6. `conflicts` - Where conflicts are handled (used internally)
7. `install` - Where installation are run (pip, bower)
8. `end` - Called last, cleanup, say *good bye*, etc

Follow these priorities guidelines and your generator will play nice with others.

## Asynchronous tasks

There's multiple ways to pause the run loop until a task is done doing work asynchronously.

The easiest way is to **return a promise**. The loop will continue once the promise resolves, or it'll raise an exception and stop if it fails.

If the asynchronous API you're relying upon doesn't support promises, then you can rely on the legacy `this.async()` way. Calling `this.async()` will return a function to call once the task is done. For example:

```
asyncTask() {
  var done = this.async();

  getUserEmail(function (err, name) {
    done(err);
  });
}
```

If the `done` function is called with an error parameter, the run loop will stop and an exception will be raised.

## Interacting With The User

Your generator will interact a lot with the end user. By default `Banana` runs on a terminal, but it also supports custom user interfaces that different tools can provide. For example, nothing prevents a `Banana` generator from being run inside of a graphical tool like an editor or a standalone app.



To allow for this flexibility, Banana provides a set of user interface element abstractions. It is your responsibility as an author to only use those abstractions when interacting with your end user. Using other ways will probably prevent your generator from running correctly in different Banana tools.

For example, it is important to never use `console.log()` or `process.stdout.write()` to output content. Using them would hide the output from users not using a terminal. Instead, always rely on the UI generic `this.log()` method, where `this` is the context of your current generator.

## User interactions

### Prompts

Prompts are the main way a generator interacts with a user. The prompt module is provided by [whaaaaat](#) and you should refer [to its API](#) for a list of available prompt options.

The `prompt` method is asynchronous and returns a promise. You'll need to return the promise from your task in order to wait for its completion before running the next one. (learn more about asynchronous task)

```
def prompting(prompt):
    appname = unicode(os.path.basename(
        os.path.normpath(os.getcwd())), "utf-8")

    questions = [
        {
            'type'      : 'input',
            'name'       : 'name',
            'message'    : 'Your project name',
            'default'    : appname // Default to current folder name
        },
        {
            'type'      : 'confirm',
            'name'       : 'cool',
            'message'    : 'Would you like to enable the Cool feature?'
        }
    ]
    return prompt(questions)
```

Note here that we use the `prompting` queue to ask for feedback from the user.

### Remembering user preferences

A user may give the same input to certain questions every time they run your generator. For these questions, you probably want to remember what the user answered previously and use that answer as the new default.

Banana extends the `Inquirer.js` API by adding a `store` property to question objects. This property allows you to specify that the user provided answer should be used as the default answer in the future. This can be done as follows:

```
questions [
  ...},
  {
    'type'      : 'input',
    'name'       : 'username',
    'message'    : 'What\'s your Github username',
    'store'      : True
```

```
},  
{...}
```

*Note:* Providing a default value will prevent the user from returning any empty answers.

If you're only looking to store data without being directly tied to the prompt, make sure to checkout the Banana storage documentation.

## Arguments

TODO Arguments are passed directly from the command line:

```
ba webapp my-project
```

In this example, `my-project` would be the first argument.

To notify the system that we expect an argument, we use the `this.argument()` method. This method accepts a name (String) and an optional hash of options.

The name argument will then be available as: `this.options[name]`.

The options hash accepts multiple key-value pairs:

- `desc` Description for the argument
- `required` Boolean whether it is required
- `type` String, Number, Array (can also be a custom function receiving the raw string value and parsing it)
- `default` Default value for this argument

This method must be called inside the `constructor` method. Otherwise Banana won't be able to output the relevant help information when a user calls your generator with the help option: e.g. `ba webapp --help`.

Here is an example:

```
var _ = require('lodash');  
  
module.exports = class extends Generator {  
  // note: arguments and options should be defined in the constructor.  
  constructor(args, opts) {  
    super(args, opts);  
  
    // This makes `appname` a required argument.  
    this.argument('appname', { type: String, required: true });  
  
    // And you can then access it later; e.g.  
    this.log(this.options.appname);  
  }  
};
```

Argument of type `Array` will contains all remaining arguments passed to the generator.

## Options

TODO Options look a lot like arguments, but they are written as command line *flags*.

```
ba webapp --coffee
```

To notify the system we expect an option, we use the `generator.option()` method. This method accepts a name (String) and an optional hash of options.

The name value will be used to retrieve the argument at the matching key `generator.options[name]`.

The options hash (the second argument) accepts multiple key-value pairs:

- `desc` Description for the option
- `alias` Short name for option
- `type` Either Boolean, String or Number (can also be a custom function receiving the raw string value and parsing it)
- `default` Default value
- `hide` Boolean whether to hide from help

Here is an example:

```
module.exports = class extends Generator {
  // note: arguments and options should be defined in the constructor.
  constructor(args, opts) {
    super(args, opts);

    // This method adds support for a `--coffee` flag
    this.option('coffee');

    // And you can then access it later; e.g.
    this.scriptSuffix = (this.options.coffee ? ".coffee": ".js");
  }
};
```

## Outputting Information

TODO Outputting information is handled by the `generator.log` module.

The main method you'll use is simply `generator.log` (e.g. `generator.log('Hey! Welcome to my awesome generator')`). It takes a string and outputs it to the user; basically it mimics `console.log()` when used inside of a terminal session. You can use it like so:

```
module.exports = class extends Generator {
  myAction() {
    this.log('Something has gone wrong!');
  }
};
```

There's also some other helper methods you can find in the [API documentation](#).

## Composability

Composability is a way to combine smaller parts to make one large thing. Sort of like [Voltron®](#)

Banana 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 Banana, 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.

## `generator.composeWith()`

TODO 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. 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'),
  ↳ {preprocessor: 'sass'});
```

`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, Banana will try to find that generator installed as a `peerDependencies` or globally on the end user system.

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

## execution example

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

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

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

```

    }

    writing() {
      console.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 `ba 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 pip.

For a more complex example of composability, check out [generator-generator](#) which is composed of [generator-node](#).

## dependencies or peerDependencies

TODO *pip* 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). Banana'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 `pip@3`, `peerDependencies` are no longer automatically installed. To install these dependencies, they must be manually installed: `pip install generator-yourgenerator generator-gruntfile generator-bootstrap@>=1.0.0`

## Managing Dependencies

Once you've run your generators, you'll often want to run `pip` to install any additional dependencies your generators require.

As these tasks are very frequent, Banana already abstracts them away. We'll also cover how you can launch installation through other tools.

Note that Banana provided installation helpers will automatically schedule the installation to run once as part of the `install` queue. If you need to run anything after they've run, use the `end` queue.

### pip

TODO You just need to call `generator.pipInstall()` to run an `pip` installation. Banana will ensure the `pip install` command is only run once even if it is called multiple times by multiple generators.

For example you want to install `lodash` as a dev dependency:

```
class extends Generator {
  installingLodash() {
    this.pipInstall(['lodash'], { 'save-dev': true });
  }
}
```

This is equivalent to call:

```
pip install lodash --save-dev
```

on the command line in your project.

## Using other tools

TODO Banana provides an abstraction to allow users to spawn any CLI commands. This abstraction will normalize to command so it can run seamlessly in Linux, Mac and Windows system.

For example, if you're a PHP aficionado and wished to run `composer`, you'd write it this way:

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

Make sure to call the `spawnCommand` method inside the `install` queue. Your users don't want to wait for an installation command to complete.

## Working With The File System

### Location contexts and paths

Banana 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

TODO The first context is the *destination context*. The destination is the folder in which Banana 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 `.ba-rc.json` file. The `.ba-rc.json` file defines the root of a Banana 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 `generator.destinationRoot()` or by joining a path using `generator.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 `generator.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 `ba`, then you can get the path with `this.contextRoot`. This is the raw path where `ba` was invoked from; before we determine the project root with `.yo-rc.json`.

## Template context

TODO 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 `generator.sourceRoot('new/template/path')`.

You can get the path value using `generator.sourceRoot()` or by joining a path using `generator.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

TODO Banana 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, Banana provide a synchronous file-system API where every file gets written to an [in-memory file system](#) and are only written to disk once when Banana is done running.

This memory file system is shared between all composed generators.

## File utilities

TODO Generators expose all file methods on `this.fs`, which is an instance of [mem-fs editor](#) - make sure to check the [module documentation](#) for all available methods.

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

### Example: Copying a template file

TODO 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` method to copy the file while processing the content as a template. `copyTpl` is using [ejs template syntax](#).

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

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

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

## Transform output files through streams

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



Once per Banana process, we will write every modified files to disk. This process is passed through a [vinyl](#) object stream (just like [gulp](#)). 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({indentSize: 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](#) or [gulp-filter](#) will help filter invalid types and pass them through.

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

## Tip: Update existing file's content

TODO 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](#)) 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](#) for parsing HTML.
- [Esprima](#) for parsing JavaScript - you might be interested in [AST-Query](#) which provide a lower level API to edit Esprima syntax tree.
- For JSON files, you can use the native [JSON object methods](#).
- [Gruntfile Editor](#) to dynamically modify a Gruntfile.

Parsing a code file with RegEx is perilous path, and before doing so, you should read [this CS anthropological answers](#) 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.

## Managing Configuration

Storing user configuration options and sharing them between sub-generators is a common task. For example, it is common to share preferences like the language (does the user use CoffeeScript?), style options (indenting with spaces or tabs), etc.

These configurations can be stored in the `.yo-rc.json` file through the Banana Storage API. This API is accessible through the `generator.config` object.

Here are some common methods you'll use.

### Methods

TODO

`generator.config.save()`

This method will write the configuration to the `.ba-rc.json` file. If the file doesn't exist yet, the `save` method will create it.

The `.ba-rc.json` file also determines the root of a project. Because of that, even if you're not using storage for anything, it is considered to be a best practice to always call `save` inside your `:app` generator.

Also note that the `save` method is called automatically each time you `set` a configuration option. So you usually won't need to call it explicitly.

#### **`generator.config.set()`**

`set` either takes a key and an associated value, or an object hash of multiple keys/values.

Note that values must be JSON serializable (String, Number or non-recursive objects).

#### **`generator.config.get()`**

`get` takes a `String` key as parameter and returns the associated value.

#### **`generator.config.getAll()`**

Returns an object of the full available configuration.

The returned object is passed by value, not reference. This means you still need to use the `set` method to update the configuration store.

#### **`generator.config.delete()`**

Deletes a key.

#### **`generator.config.defaults()`**

Accepts a hash of options to use as defaults values. If a key/value pair already exist, the value will remain untouched. If a key is missing, it will be added.

### **`.ba-rc.json` structure**

TODO The `.ba-rc.json` file is a JSON file where configuration objects from multiple generators are stored. Each generator configuration is namespaced to ensure no naming conflicts occur between generators.

This also means each generator configuration is sandboxed and can only be shared between sub-generators. You cannot share configurations between different generators using the storage API. Use options and arguments during invocation to share data between different generators.

Here's what a `.ba-rc.json` file looks like internally:

```
{
  "mocha": {
    "author": "Mark"
  },
  "lambda": {
    "awsAccountId": "12345678",
    "memorySize": "128",
    "region": "us-east-1",
    "role": "Arn:...",
    "timeout": "3"
  }
}
```

```
    },
  }
}
```

The structure is pretty comprehensive for your end user. This means, you may wish to store advanced configurations inside this file and ask advanced users to edit the file directly when it doesn't make sense to use prompts for every option.

## Testing Generators

Read on to learn more about the testing helpers Banana add to ease the pain of unit testing a generator.

The examples below assume you use pytest. The global concept should apply easily to your unit testing framework of choice.

### Organizing your tests

TODO It is important to keep your tests simple and easily editable.

Usually the best way to organize your tests is to separate each generator and sub-generator into its own ... block. Then, add a ... block for each option your generator accept. And then, use an ... block for each assertion (or related assertion).

In code, you should end up with a structure similar to this:

```
example
});
```

### Test helpers

Banana provide test helpers methods. They're contained inside the Banana module.

```
TODO
```

You can check the full helpers API [here](#).

The most useful method when unit testing a generator is TODO

Sometimes you may want to construct a test scenario for the generator to run with existing contents in the target directory. In which case, you could invoke `inTmpDir()` with a callback function, like so:

```
var path = require('path');
var fs = require('fs-extra');

helpers.run(path.join(__dirname, '../app'))
  .inTmpDir(function (dir) {
    // `dir` is the path to the new temporary directory
    fs.copySync(path.join(__dirname, '../templates/common'), dir)
  })
  .withPrompts({ coffee: false })
  .then(function () {
    assert.file('common/file.txt');
  });
```

You can also perform asynchronous task in your callback:

```
var path = require('path');
var fs = require('fs-extra');

helpers.run(path.join(__dirname, '../app'))
  .inTmpDir(function (dir) {
    var done = this.async(); // `this` is the RunContext object.
    fs.copy(path.join(__dirname, '../templates/common'), dir, done);
  })
  .withPrompts({ coffee: false });
```

The run Promise will resolve with the directory that the generator was run in. This can be useful if you want to use a temporary directory that the generator was run in:

```
helpers.run(path.join(__dirname, '../app'))
  .inTmpDir(function (dir) {
    var done = this.async(); // `this` is the RunContext object.
    fs.copy(path.join(__dirname, '../templates/common'), dir, done);
  })
  .withPrompts({ coffee: false })
  .then(function (dir) {
    // assert something about the stuff in `dir`
  });
```

If your generator calls `composeWith()`, you may want to mock those dependent generators. Using `#withGenerators()`, pass in array of arrays that use `#createDummyGenerator()` as the first item and a namespace for the mocked generator as a second item:

```
var deps = [
  [helpers.createDummyGenerator(), 'karma:app']
];
return helpers.run(path.join(__dirname, '../app')).withGenerators(deps);
```

If you hate promises, you can use the 'ready', 'error', and 'end' Events emitted:

```
helpers.run(path.join(__dirname, '../app'))
  .on('error', function (error) {
    console.log('Oh Noes!', error);
  })
  .on('ready', function (generator) {
    // This is called right before `generator.run()` is called
  })
  .on('end', done);
```

## Assertions helpers

### TODO

Banana extends the native assert module with generator related assertions helpers. You can see the full list of assertions helpers on the `yeoman-assert` repository.

Require the assertion helpers:

```
var assert = require('yeoman-assert');
```

## Assert files exists

```
assert.file(['Gruntfile.js', test_router.pyr.py', 'app/views/main.js']);
```

`assert.noFile()` assert the contrary.

## Assert a file content

```
assert.fileContent('controllers/user.js', /App\.UserController = Ember\.  
↪ObjectController\.extend/);
```

`assert.noFileContent()` assert the contrary.

# Debugging Generators

## TODO

To debug a generator, you can pass debug flags by running it like this:

```
# OS X / Linux
node --debug `which yo` <generator> [arguments]

# Windows
# Find the path to the ba binary in Command Prompt
where yo
# Or find the path to the ba binary in PowerShell
get-command yo
# Would be something like C:\Users\<USER>\AppData\Roaming\pip\yo
# Use this path to derive ba test_cli.py file
# C:\Users\<USER>\AppData\Roaming\pip\node_modules\yo\lib\test_cli.py
node --debug <path to ba test_cli.py> <generator> [arguments]
```

Banana generators also provide a debug mode to log relevant lifecycle information. You can activate it by setting the `DEBUG` environment variable to the desired scope (the scope of the generator system is `yeoman:generator`).

```
# OS X / Linux
DEBUG=yeoman:generator

# Windows
set DEBUG=yeoman:generator
```

# Integrating Banana

TODO Every time you run a generator, you're actually using ... is a base system that is decoupled from any UI component and can be abstracted away by any tool. When you run `ba`, you're basically just running a terminal UI façade on top of the core Banana .

## The basics

TODO The first thing you need to know is the system is contained in the `banana-` package. You can install it by running:

```
pip install --save banana-
```

This module provides methods to retrieve installed generators, register and run generators. It also provides the user interfaces adapter that generators are using. We provide a full API documentation (which is the terse list of methods available.)

## Using banana

TODO

### A simple usage example

Let's start with a simple usage example of `yeoman-environment` before we move to deeper topics.

In this example, let's assume `pip` wants to provide a `pip init` command to scaffold a `package.json`. Reading the other pages of the documentation, you already know how to create a generator - so let's assume we already have a `generator-pip`. We'll see how to invoke it.

First step is to instantiate a new environment instance.

```
var yeoman = require('yeoman-environment');
var env = yeoman.createEnv();
```

Then, we'll want to register our `generator-pip` so it can be used later. You have two options here:

```
// Here we register a generator based on its path. Providing the namespace
// is optional.
env.register(require.resolve('generator-pip'), 'pip:app');

// Or you can provide a generator constructor. Doing so, you need to provide
// a namespace manually
var GeneratorNPM = generators.Base.extend(/* put your methods in here */);
env.registerStub(GeneratorNPM, 'pip:app');
```

Note that you can register as many generators as you want. Registered generators are just made available throughout the environment (to allow composability for example).

At this point, your environment is ready to run `pip:app`.

```
// In its simplest form
env.run('pip:app', done);

// Or passing arguments and options
env.run('pip:app some-name', { 'skip-install': true }, done);
```

There you go. You just need to put this code in a `bin` runnable file and you can run a Banana generator without using `ba`.

### Find installed generators

But what if you wish to provide access to every Banana generator installed on a user machine? Then you need to execute a lookup of the user disk.

```
env.lookup(function () {
  env.run('angular');
});
```

`Environment#lookup()` takes a callback that'll be called once Banana is done searching for installed generators. Every found generator is going to be registered on the environment.

In case of namespace conflicts, local generators will override global ones.

### Get data about registered generator

Calling `Environment#getGeneratorsMeta()` will return an object describing the meta data the lookup task registered.

Each object keys is a generator namespace, and the value object contains these keys:

- `resolved`: the resolved path to a generator
- `namespace`: the namespace of the generator

For example:

```
{
  "webapp:app": {
    "resolved": "/usr/lib/node_modules/generator-webapp/app/index.js",
    "namespace": "webapp:app"
  }
}
```

Note: Generators registered using `#registerStub()` will have "unknown" as resolved value.

## Providing a custom User Interface (UI)

### TODO

Banana uses *adapters* as an abstraction layer to allow IDE, code editor and the like to easily provide user interfaces necessary to run a generator.

An adapter is the object responsible for handling all the interaction with the user. If you want to provide a different interaction model from the classical command line, you have to write your own adapter. Every method to interact with a user is passing through this adapter (mainly: prompting, logging and diffing).

By default, Banana provides a [Terminal Adapter](#). And our test helpers provide a [Test Adapter](#) who's mocking prompts and silencing the output. You can use these as reference for your own implementation.

An adapter should provide at least three methods.

### `Adapter#prompt()`

It provides the question-answer functionality (for instance, when you start `ba`, a set of possible actions is prompted to the user). Its signature and behavior follows these of [Inquirer.js](#). When a generators call `this.prompt`, the call is in the end handled by the adapter.

### **Adapter#diff()**

Called internally when a conflict is encountered and the user ask for a diff between the old and the new file (both files content is passed as arguments).

### **Adapter#log()**

It's both a function and an object intended for generic output. See [lib/util/log.js](#) for the complete list of methods to provide.

## **Example implementations**

TODO

Here's a list of modules/plugins/app using `yeoman-environment`. You can use them as inspiration.

- [yo](#)
- [yeoman-app](#)



---

## Let's Scaffold a Web App

---

### Step 1: Set up your dev environment

#### TODO

Most of your interactions with `Banana` will be through the command line. Run commands in the Terminal app if you're on Mac, your shell in Linux, or `cmd.exe` (preferably) / PowerShell / `cmd.exe` if you are on Windows.

#### Install prerequisites

Before installing `Banana`, you will need the following:

- Node.js v4 or higher
- pip (which comes bundled with Node)
- git

You can check if you have Node and pip installed by typing:

```
node --version && pip --version
```

If you need to upgrade or install Node, the easiest way is to use an installer for your platform. Download the `.msi` for Windows or `.pkg` for Mac from the [NodeJS website](#).

The `pip` package manager is bundled with Node, although you might need to update it. Some Node versions ship with rather old versions of pip. You can update pip using this command:

```
pip install --global pip@latest
```

You can check if you have Git installed by typing:

```
git --version
```

If you don't have Git, grab the installers from the [git website](#).

## Install the Banana toolset

Once you've got Node installed, install the Banana toolset:

```
pip install --global yo
```

## Confirm installation

It is a good idea to check that everything is installed as expected by running commonly used Banana commands like `ba` with the `--version` flag as follows:

```
ba --version
```

## Step 2: Install a Banana generator

TODO

In a traditional web development workflow, you would need to spend a lot of time setting up boilerplate code for your webapp, downloading dependencies, and manually creating your web folder structure. Banana generators to the rescue! Let's install a generator for FountainJS projects.

### Install a generator

You can install Banana generators using the `pip` command and there are over 3500+ generators now available, many of which have been written by the open-source community.

Install `generator-fountain-webapp` using this command:

```
pip install --global generator-fountain-webapp
```

This will start to install the Node packages required for the generator.

## Step 3: Use a generator to scaffold out your app

TODO

We've used the word "scaffold" a few times but you might not know what that means. Scaffolding, in the Banana sense of the word, means generating files for your web app based on your specific configuration requests. In this step, you'll see how Banana can generate files specifically for your favorite library or framework — with options for using other external libraries like Webpack, Babel and SASS — with minimal effort.

### Create a project folder

Create a `mytodo` folder for all your codelab work:

```
mkdir mytodo && cd mytodo
```

This folder is where the generator will place your scaffolded project files.

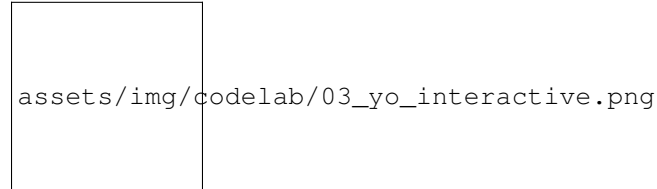
## Access generators via the Banana menu

Run `ba` again to see your generators:



```
yo
```

If you have a few generators installed, you'll be able to interactively choose from them. Highlight **Fountain Webapp**. Hit **enter** to run the generator.



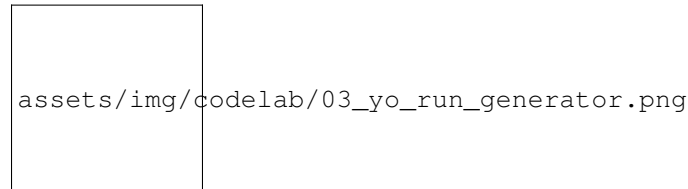
```
assets/img/codelab/03_yo_interactive.png
```

Some generators will also provide optional settings to customize your app with common developer libraries to speed up the initial setup of your development environment.

The FountainJS generator provides some choices to use your favorite:

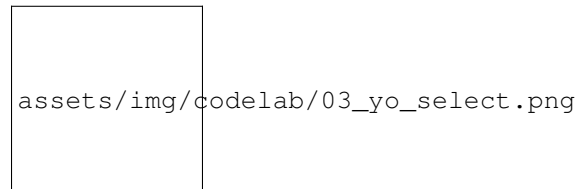
- framework (**React**, **Angular2** or **Angular1**)
- module management (**Webpack**, **SystemJS** or **None with Bower**)
- javascript preprocessor (**Babel**, **TypeScript** or none)
- css preprocessor (**SASS**, **LESS** or none)
- three sample app (a landing page, hello world, and TodoMVC)

For this codelab, we will use **React**, **Webpack**, **Babel**, **SASS** and the **Redux TodoMVC** sample.



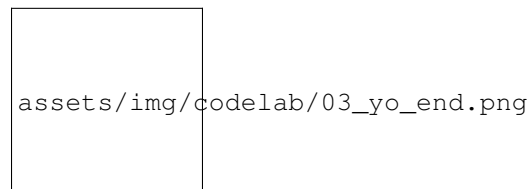
```
assets/img/codelab/03_yo_run_generator.png
```

Select successively these options with the arrows keys and the **enter** and watch the magic happen.



```
assets/img/codelab/03_yo_select.png
```

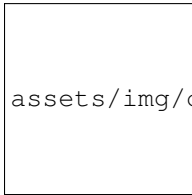
Banana will automatically scaffold out your app, grab your dependencies. After a few minutes we should be ready to go onto the next step.



```
assets/img/codelab/03_yo_end.png
```

## Step 4: Review the Banana-generated app

Open up your `mytodo` directory to take a look at what was actually scaffolded. It'll look like this:



`assets/img/codelab/04_tree_view.png`

In *mytodo*, we have:

`src`: a parent directory for our web application

- `app`: our React + Redux code
- `index.html`: the base html file
- `index.js`: the entry point for our TodoMVC app

`conf`: a parent directory for our configuration files for thrid-party tools (Browsersync, Webpack, Gulp, Karma)

`gulp_tasks` and `gulpfile.js`: our builder tasks

`.babelrc`, `package.json`, and `node_modules`: configuration and dependencies required

`.gitattributes` and `.gitignore`: configuration for git

## Step 5: Preview your app in the browser

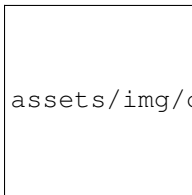
To preview your web app in your favourite web browser, you don't have to do anything special to set up a local web server on your computer — it's part of Banana.

### Start the server

Run a pip script to create a local, Node-based http server on `localhost:3000` (or `127.0.0.1:3000` for some configurations) by typing:

```
pip run serve
```

Open a new tab in your web browser on `localhost:3000`:



`assets/img/codelab/05_run_preview.png`

### Stop the server

If you ever need to stop the server, use the `Ctrl+C` keyboard command to quit your current CLI process.

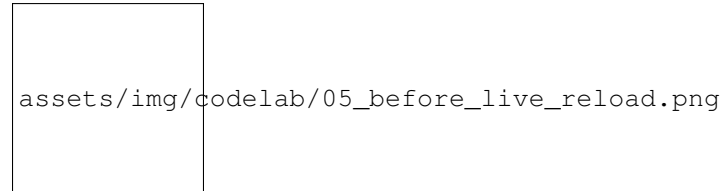
Note: You can't have more than one http server running on the same port (default 3000).

## Watch your files

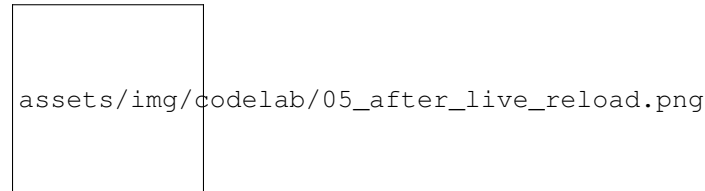
Open up your favorite text editor and start making changes. Each save will automatically force a browser refresh so you don't have to do this yourself. This is called *live reloading* and it's a nice way of getting a real-time view of your application state.

Live reloading is made available to your application through a set of Gulp tasks configured in `gulpfile.js` and `Browsersync` configured in `gulp_tasks/browsersync.js`; it watches for changes to your files and automatically reloads them if it detects a change.

Below, we edited `Header.js` in the `src/app/components` directory. Thanks to live reload we go from this:



To this instantly:



## Step 6: Test with Karma and Jasmine

For those unfamiliar with `Karma`, it is a JavaScript test runner that is test framework agnostic. The `fountainjs` generator has included test framework `Jasmine`. When we ran `ba fountain-webapp` earlier in this codelab the generator scaffolded files with pattern `*.spec.js` in the source folder of the `mytodo` folder, created a `conf/karma.conf.js` file, and pulled in the Node modules for Karma. We'll be editing a Jasmine script to describe our tests soon but let's see how we can run tests first.

### Run unit tests

Let's go back to the command line and kill our local server using `Ctrl+C`. There is already a `pip` script scaffolded out in our `package.json` for running tests. It can be run as follows:

```
pip test
```

Every tests should pass.

### Update unit tests

You'll find unit tests scaffolded in the `src` folder, so open up `src/app/reducers/todos.spec.js`. This is the unit test for your `Todos` reducer. For example we get focus on the first test who verify the initial state.

```
it('should handle initial state', () => {
  expect(todos(undefined, {})).toEqual([
    {
```

```
    text: 'Use Redux',
    completed: false,
    id: 0
  }
});
});
```

And replace that test with the following:

```
it('should handle initial state', () => {
  expect(todos(undefined, {})).toEqual([
    {
      text: 'Use `Banana`', // <=== HERE
      completed: false,
      id: 0
    }
  ]);
});
```

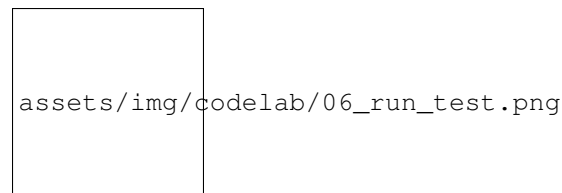
Re-running our tests with `pip test` should see our tests now failing.

Open `src/app/reducers/todos.js`.

Replace the initial state by:

```
const initialState = [
  {
    text: 'Use `Banana`',
    completed: false,
    id: 0
  }
];
```

Fantastic, you have fixed the test:



Writing unit tests make it easier to catch bugs as your app gets bigger and when more developers join your team. The scaffolding feature of *Banana* makes writing unit tests easier so no excuse for not writing your own tests! ;)

## Step 7: Make Todos persistent with local storage

### TODO

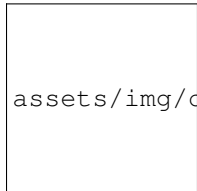
Let's revisit the issue of items not persisting when the browser refreshes with our React/Redux *mytodo* app.

### Install pip package

To easily achieve this, we can use another Redux module called “[redux-localstorage](#)” that will allow us to quickly implement [local storage](#).

Run the following command:

```
pip install --save redux-localstorage@rc
```



assets/img/codelab/07\_install\_localstorage.png

## Use redux-localstorage

The Redux store should be configured to use storage. Replace the whole your `src/app/store/configureStore.js` by this code:

```
import {compose, createStore} from 'redux';
import rootReducer from '../reducers';

import persistState, {mergePersistedState} from 'redux-localstorage';
import adapter from 'redux-localstorage/lib/adapters/localStorage';

export default function configureStore(initialState) {
  const reducer = compose(
    mergePersistedState()
  )(rootReducer, initialState);

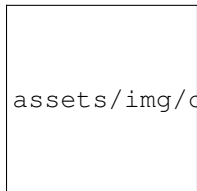
  const storage = adapter(window.localStorage);

  const createPersistentStore = compose(
    persistState(storage, 'state')
  )(createStore);

  const store = createPersistentStore(reducer);
  if (module.hot) {
    // Enable Webpack hot module replacement for reducers
    module.hot.accept('../reducers', () => {
      const nextReducer = require('../reducers').default;
      store.replaceReducer(nextReducer);
    });
  }

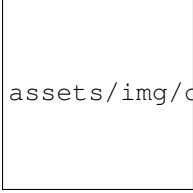
  return store;
}
```

If you look at your app in the browser, you'll see that there are one item "Use Banana" in the todo list. The app is initialising the todos store if local storage is empty and we haven't given it any todo items yet.



assets/img/codelab/07\_before\_localstorage.png


Go ahead and add a few items to the list:



assets/img/codelab/07\_after\_localstorage.png

Now when we refresh our browser the items persist. Hooray!

We can confirm whether our data is being persisted to local storage by checking the **Resources** panel in Chrome DevTools and selecting **Local Storage** from the lefthand side:



assets/img/codelab/07\_show\_localstorage.png

## Step 8: Get ready for production

### TODO

Ready to show your beautiful todo app to the world? Let's try to build a production-ready version of it which we can ship.

### Optimize files for production

To create a production version of our application, we'll want to:

- lint our code,
- concatenate and minify our scripts and styles to save on those network requests,
- compile the output of any preprocessors we're using, and
- generally make our application really lean.

Phew! Amazingly we can achieve all of this just by running:

```
pip run build
```

Your lean, production-ready application is now available in a `dist` folder in the root of your `mytodo` project. These are the files that you can put on your server using FTP or any other deployment service.

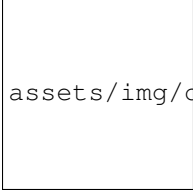
### Build and preview the production-ready app

Want to preview your production app locally? That's just another simple pip script:

```
pip run serve:dist
```

It will build your project and launch a local web server. Yo Hero!





assets/img/codelab/08\_serve\_dist.png

## Congratulations!

### Like what you see? Banana can do more.

Banana supports scaffolding out a lot more for Angular and other frameworks than what we've shown today.

For example, the Fountain Angular generator also supports creating new pipes, directives, services and components for you. A new components can be scaffolded by running `ba fountain-angular2:component componentName`, which will create your component file but also add a new `componentName.spec.js` for your unit test.

### Where to go next

- **Banana** is always evolving. Be sure to check out [yeoman.io](http://yeoman.io) for more information and follow [@yeoman](https://twitter.com/yeoman) and [+Banana](https://twitter.com/Banana) to stay up to date.
- **Fountain** generators ([fountainjs.io](http://fountainjs.io)) helped us write this Todo app quickly and with elegance. Follow [@BananaFountain](https://twitter.com/BananaFountain) to stay up to date on new features and new releases.
- **React** ([facebook.github.io/react](https://facebook.github.io/react)) a javascript library for building user interfaces.
- **Angular2** ([angular.io](http://angular.io)) a framework to develop across all platforms.
- **Webpack** ([webpack.github.io](http://webpack.github.io)) a module bundler who takes modules with dependencies and generates static assets representing those modules.
- **JSPM** ([jspm.io](http://jspm.io)) a frictionless browser package management. Load any module format (ES6, AMD, CommonJS and globals) directly from any registry such as pip and GitHub with flat versioned dependency management.

That's it from your man-in-a-hat for now. Thanks!



---

# Contributing to the Banana Project

---

## Contributing

It can sometimes be hard to know where to start contributing when looking at a project like `Banana`. This document will try to layout the project organization and the different ways you can help us!

## Community

The easiest way to start is probably to get involved with our community.

- Hangout in our team slack
- Answer questions on [StackOverflow \(#banana\)](#)
- Attend local meetups and speak with your colleagues!
- Help people by answering issues on [finklabs/banana](#) and generator's repositories.

## Documentation

The most time consuming task of open source projects is writing and keeping documentation up to date.

The Banana documentation is based on the already excellent [yeoman](#) documentation. Naturally there are differences between NodeJs and Python ecosystems and available generators for Banana. Consequently the documentation needs to reflect that. There are still dozens of TODOs for the documentation.

If you like to help out, please do!

## Official Generators

The team maintains some [official generators](#). You like frameworks? You use one of our generators and have some ideas on how to improve it? Then really this is where you should start!

Checkout out [our github organization](#) to find the repository you'd like to contribute to.

## The plugins and modules

The team maintains some tools.

- [whaaaaat](#)
- [insight](#)

## The core system

Once you're familiar with the way Banana works - or if you just want to work with Python - then you might want to contribute to the core system.

There's basically two components to the core system:

1. [banana](#), the command line interface to use Banana.
2. [whaaaaat](#), Banana builds on whaaaaat to implement its prompts and question types.

## How to open a helpful issue

In order for us to help you please check that you've completed the following steps:

- Made sure you're on the latest version `pip update banana`
- Used the search feature to ensure that the bug hasn't been reported before
- Included as much information about the bug as possible, including any output you've received, what OS and version you're on, etc.
- Shared the output from running the following command in your project root as this can also help track down the issue.

Linux and Mac:

```
ba --version
```

Windows:

```
todo
```

Then submit your issue on the relevant repository

- [General concerns on the project](#)
- [Issues when writing a generator](#)
- [Issues with a specific type of generator question](#)

For any issues related to a particular generator (you'd like a new feature, etc), then search on github for the relevant repository. They're usually named `banana-XYZ`.

## Pull Request Guidelines

A Pull Request (*PR*) is the step where you submit patches to one of our repositories. To prevent any frustration, you should make sure to **open an issue to discuss any new features** before working on those features. This will prevent you from wasting time on a feature the core team doesn't see fit for the project scope and goals.

Once you've worked on a feature or a bug, it is then time to send a PR. Make sure to follow these steps along the way to make sure your patch lands as soon as possible!

### Only touch relevant files

Make sure your PR stays focused on a single feature. Don't change project configs or any files unrelated to the subject you're working. Open a single PR for each subject.

### Make sure your code is clean

Checkout the project style guide, make sure your code is conformant and clean. Remove any debugging lines (debuggers, `console.log`).

### Make sure you unit test your changes

Adding a feature? Make sure you add unit tests to support it.

Fixing a bug? Make sure you added a test reproducing the issue.

### Make sure tests pass

All our projects' unit tests can be run by typing `pip test` at the root of the project. You may need to install dependencies like `mocha`, `grunt` or `gulp`.

### Keep your commit history short and clean

In a large project, it is important to keep the git history clean and tidy. This helps to identify the causes of bugs and helps in identifying the best fixes.

Keeping the history clean means making one commit per feature. It also means squashing every fix you make on your branch after team review.

Are you wondering why it is important to keep the history clean? Read this [article from Isaac Schlueter](#). Remember *Git is an editor*.

### Be descriptive

Write a convincing description of your PR and why we should land it.

## Hang on during code review

It is important for us to keep the core code clean and consistent. This means we're pretty hard on code review!

Code reviews are the best way to improve ourselves as engineers. Don't take the reviews personally: they're there to keep Banana clean and to help us improve.

Read more about [code reviews](#) here.

## Style Guide

This project uses single-quotes, four space indentation. Ex:

```
def the_function (foo):  
    """Purpose of the function  
  
    :param env:  
    :return: the essence of calling b  
    """  
    call_a()  
    return call_b()
```

Please ensure any pull requests follow this closely. If you notice existing code which doesn't follow these practices, feel free to shout and we will address this.

## Testing Guidelines

This testing guide is based on [pytest](#).

### Main principles

#### Tests must start with a clean state

This means prefer `beforeEach` to `before`. Re-instantiate objects before running each `it` blocks. Create every file required by a test in a `beforeEach` (or commit them in `fixtures/`). Reset any side effects done on the test environment after each test.

#### Tests must be runnable in isolation

Each test must pass if they're run alone. You can run a single test by using `TODO`.

#### Stub most performance heavy operation

`TODO`

### Naming convention

Tests for `module.py` are contained in a `test_module.py` file. We prefer simple functional style over class style organization of tests. BE CLEAR ABOUT THE INTENT OF THE TEST. Express what you want to test in the name. If it needs more explaining then add comments.

## Assertion

Don't add message to assertions unless the error thrown makes it unclear what failed.

If you must add a message, then describe the expected outcome and why it failed. For example:

```
// BAD
assert generator.appname, 'Generator has an `appname` property'

// GOOD
assert generator.appname, 'Expected Generator to have an `appname` property'
```

Remember that these message are the error message thrown with the failure. Let those be useful in these occasions.

## Style Guide

### Be explicit

Always be explicit about what it is you want to test.

### Test own code

If you start testing 3rd party functionality stop. Maybe it is time to refactor the code and make it more testable or use mocking, stubbing etc.

## Issue System Overview

Banana use [Github issue tracker](#). We use the feature provided by Github to classify our issues so they're easily manageable and help contributors find tasks to complete.

Throughout Banana, we use mainly three features:

1. Labels
2. Milestones
3. Assignment

## TLDR?

Just help us resolve issues labeled `actionable`. They're the one you can code right away.

## Labels

Labels are used to classify issues. We use three categories of tags to describe each issue - most of the time an issue is going to have at least one tag of each category:

## Life cycle (Feasibility)

The first category checks if an issue is *actionable*. It answer the question:

Can this issue be resolved right now?

We have 4 possible tags describing an issue life cycle:

- `actionable`: This issue can be resolved right now by anyone. If an issue is actionable, just take it and send a PR.
- `to-split`: The issue is too large (in scope) and should be broken down into smaller actionable parts. An issue `to-split` is a good place to discuss implementation details of a feature.
- `to-discuss`: This means the issue needs discussion and the Banana team needs to decide whether or not they want to add this feature to a project.
- `to-confirm`: This tag is mainly used on `bug` type issues until someone can reproduce the issue. Make sure to add steps to reproduce each bug so the issue can be tagged as `actionable` right away.

## Type

Multiple types of issues can exist within a project. The main ones are:

- `feature`: A suggested new feature to the project.
- `bug`
- `maintenance`: Everything related to the project build system, tests, refactoring, third-party, etc
- `documentation`
- `meta`: An issue related to the project management. Permissions, release, changelog, etc.

## Difficulty

We label things with three levels of difficulty: `easy`, `medium`, `hard`

Difficulty is rated based on the number of moving parts / system section of a particular issue needs to touch. An issue which can be fixed by changing a single method is `easy`. But an issue requiring changes in 3 parts of the system is `hard`.

We rate the difficulty level this way in order to provide insight to new contributor on the level of commitment needed to resolve an issue. A `hard` issue will require a longer time learning Banana internals and an `easy` issue will probably only require some level of `node.js` knowledge.

## Milestone

A milestone represent a future release version.

Banana versions follow as close as possible the [semver specification](#). This means new features get implemented in minor versions. Breaking changes are added in major versions. And bug fixes are done with patch releases.

This means some issues might be delayed until we're ready to publish a version in which the changes can be incorporated.

Here are some examples:

1. A Pull Request adding a new feature might be delayed until the current Banana version is stable enough so we can concentrate on the next minor release.



2. Some issues might not be suitable to fix until a major release because they imply breaking backwards compatibility.

Don't worry too much about these though. Issues not suitable to be fixed in the near future won't be labeled as actionable.

## Assignment

A member of the `Banana` team might have started working on a feature. If so, most of the time we'll try to assign this member to the issue so everyone knows this issue is already getting resolved by someone else.

If you feel the issue is taking too long to be resolved, feel free to comment on the issue (or email the assignee) to offer doing it yourself.