bolt Documentation

Release 0.2.10

Isaac Rodriguez

Nov 03, 2018

Contents

Bolt is a task runner inspired by grunt and written in python that helps you automate any task in your project whether it is executed in your development environment or in your CI/CD pipeline. Bolt gives you the power to specify how tasks should be executed, and it takes care of the rest. And it is as simple as describing and configuring your tasks in the `boltfile.py`.

```
# boltfile.py

import bolt

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': './requirements.txt'
        }
    },
    'delete-pyc': {
        'sourcedir': './src',
        'recursive': True
    },
    'nose': {
        'directory': './tests',
        'options': {
            'with-xunit': True,
            'xunit-file': './logs/unit_test_log.xml'
        }
    }
}

bolt.register_task('run-tests', ['pip', 'delete-pyc', 'nose'])
```

```
# in your favorite shell

bolt pip
# to install requirements

bolt nose
# executes unit tests

bolt run-tests
# installs requirements, deletes .pyc files, and runs unit tests
```

CHAPTER 1

Why Use Bolt?

Let's face it, you want to automate everything, but doing so becomes a burden; especially, if you are working on a cross-platform application. You may find your-self switching CI/CD systems and going through the pain of rewriting your pipelines to the specific domain languages they use. Python is cross-platform and any pipline will allow you to execute a command. This makes Bolt ideal to create reusable tasks that can execute in any environment indpendently of tools. And, It's fun!

CHAPTER 2

How Can I Get Started?

You can start by installing bolt and following the examples in the Getting Started guide. Once you become familiar with Bolt, you can look at other topics in Using Bolt, to learn about the different features it provides.

CHAPTER 3

This is Great! I want to Help!

Help is highly appreciated! If you want to contribute to the project, make sure to read our guidelines. If you are a tool developer, and you want to provide Bolt support in your library or application don't hesitate asking for help. We want to build a great community around Bolt, and we will help you in any way we can.

Make sure you read the bolt documentation.

3.1 Using Bolt

You want to try Bolt? That's great! You can start by installing Bolt and creating your first *boltfile.py*. You can find how to do it in the *Getting Started* guide. Then, you can check the other topics in this section to learn more about Bolt, its features, and how to create your own tasks.

3.1.1 Getting Started

In this section, we will take a look at some of the basics by installing Bolt and creating an initial `boltfile.py` to learn the main concepts.

Installing Bolt

Bolt can be installed directly from PyPi using pip.

```
pip install bolt-ta
```

Your First Bolt File

A Bolt file (`boltfile.py`) is just a Python script that defines the tasks available and the configuration for each of those tasks. Once we have a Bolt file for our project, we can run the available tasks at any time. In this section, we will learn the basics of the Bolt file by creating one that defines a very common scenario: First, it will run a task to install any missing requirements for the application (basically a `pip install`). Then, it will clean all the `.pyc` files out

of the source tree to insure a clean state. After that, it will execute all the unit tests. And finally, we will add a greeting message at the begining of our tasks just to demonstrate how to create a simple custom task.

Let's begin by looking at the structure for a our sample python project. The following shows the contents of the root directory (simplified for clarity):

```
- project-root
    |- source
    |- tests
    |- requirements.txt
    |- boltfile.py
```

Installing Requirements

We will start by creating a `boltfile.py` file at the root of our project as shown in the structure above. Once the file is created, we will create a configuration object inside the file to point to our requirements file, which contains the requirements of our application. The initial contents of `boltfile.py` look like the following:

```
import bolt

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
    }
}
```

You can now run `bolt pip` from the command line and see how the specified requirements are installed. In my example, the `requirements.txt` file only lists the `requests` module:

```
(btsample) D:\Projects\Playground\bolt-sample> bolt pip
INFO - Executing Python Package Installer
Collecting requests (from -r requirements.txt (line 1))
Using cached requests-2.13.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.13.0
(btsample) D:\Projects\Playground\bolt-sample>
```

Let's go over the example to understand what's going on. When bolt is executed in a directory containing a `boltfile.py`, the file is loaded as any other python module. Bolt requires the `boltfile.py` to define a `config` variable that is set to a configuration object, which is nothing but a Python dictionary. The root keys in the dictionary (in our case `pip`) are the id of the tasks we want to configure. Turns out Bolt provides a set of out-of-the-box tasks that can be used without any further process, and one of them is `pip`.

The `pip` task requires to specify a command to execute. In our sample we use the `install` command, but you can use any command supported by the actual `pip` package Installer. The `install` allows to specify a requirements file. In our example, we set the `r` option to the file containing the requirements (`requirements.txt`). If you think about it, all we are doing is invoking `pip install -r requirements.txt`, which is what you will normally use from the command line, but Bolt is taking care of invoking the command for us (see *pip task documentation* for more information about how to use the task).

Because the `pip` task is provided out-of-the-box, we do not need to register it with Bolt, so we can just execute it from the command line by invoking `bolt pip`.

Cleaining PYCs and Executing Unit Tests

Before we run our unit tests, we want to clear any `.pyc` files that have been generated from a previous run. Bolt provies a task (`delete-pyc`) to do just that and it can be configured as follows:

```
import bolt

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
    },
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    }
}
```

As you can see, the configuration of the `delete-pyc` task is self-explanatory. The task will search the `sourcedir` specified for `.pyc` files and it will delete them. Because we specified the `recursive` option, it will also search the entire directory tree under `source` and delete all the matches (for more information see the *delete-pyc task documentation*).

Let's not stop there! We don't want to just delete the `.pyc` files, we also want to execute the unit tests. In my example, I use `nose` as the test runner since Bolt already provides a task for that. Let's take a look at the updated `boltfile.py`:

```
import bolt

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
    },
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    },
    'nose': {
        'directory': 'tests'
    }
}

bolt.register_task('run-tests', ['pip', 'delete-pyc', 'nose'])
```

We added nose to the configuration, which just uses a `directory` parameter that points to the location of the tests (see the *nose task documentation* for more information). But, we also added the following line at the end: `bolt.register_task('run-tests', ['pip', 'delete-pyc', 'nose'])`. Let's take a look at what that does.

The `run-tests` task, which we are defining, is composed of the three other tasks that we have configured. These tasks will be executed sequentially when the `run-tests` task is invoked by invoking Bolt as `bolt run-tests`. We can additionally run `bolt delete-pyc` to just delete the `.pyc` files, run `bolt nose` to just run the unit tests, and of course `bolt pip` as we saw before.

Bolt will take care of executing the task you provide and insuring the correct configuration is passed to the task. It will also handle and report errors and stop execution if there are any problems, so the problems can be fixed.

Display a Greeting When Bolt Runs

Bolt provides a set of tasks that can be used as soon as you install it, but it also allows you to add other tasks that are specific to your project. Furthermore, tool makers can provide their own tasks to integrate Bolt with their applications and libraries. To demonstrate how easy is to create a Bolt task, we will provide one that displays a greeting at the begining of the `run-tests` task. Let's take a look at the implementation, and then, we'll discuss it.

```
import bolt

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
    },
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    },
    'nose': {
        'directory': 'tests'
    },
    'greet': {
        'message': 'Welcome to Bolt!'
    }
}


def greet_task(**kwargs):
    config = kwargs.get('config')
    message = config.get('message')
    print(message)


bolt.register_task('greet', greet_task)
bolt.register_task('run-tests', ['greet', 'pip', 'delete-pyc', 'nose'])
```

We first added a configuration key `greet` for our task. This is the id we chose for the task, and we will also use it to register it with Bolt. The configuration takes a `message` option, which value is the message we want to display.

Then we added a new function `greet_task`, which is the callable object that Bolt will call when the task is invoked. The funtion receives a keyword arguments object, which contains a `config`, which value is the configuration we defined. The function retrieves the configuration and reads the message from it in order to display it. Notice that the value of the `config` keyword argument is not the entire configuration; it just contains the parameters related to our task, in other words the value is:

```
{
    'message': 'Welcome to Bolt!'
}
```

Once we have the function and its configuration, we register it by calling `bolt.register_task('greet', greet_task)` where the first parameter is the id of our task, which we also used for the configuration, and the

second parameter is the callable we want to execute, in our case the function `greet_task`. Finally, we put our greet task at the beginning of `run-tests` and we will see the message when we execute it.

That's it! You can run `bolt greet` to just see the message, or you can execute `bolt run-tests` and see the message followed by the other tasks.

3.1.2 Disecting the Bolt File

The `boltfile.py` is the main script used by Bolt to execute the tasks which with it is invoked. The file contains the task definitions, as well as, the configuration parameters for the tasks. If you haven't done so yet, review the *Getting Started* guide to familiarize your-self with a very basic example of a `boltfile.py`. In this section, we will look at more advanced examples to learn the different features of Bolt.

In essence, a Bolt File is just a Python script. Within it, you can use the Bolt API to define and configure the tasks you want to execute. You can name the script whatever you want and place it in any location, but Bolt, by default, will look in the current working directory for a file named `boltfile.py` and use it if no other file is specified. This is the recommended way to work with Bolt.

```
# Assumes boltfile.py in the current working directory.
bolt task-to-execute

# Uses specified file.
bolt task-to-execute --bolt-file myboltfile.py
```

Tip: You can run `bolt --help` to see Bolt's usage and supported arguments.

The Structure of a Bolt File

There are three distinct sections in a `boltfile.py`. The first section, like in any other Python module is the imports, and your can bring in any module you want.

The second part is the registration of tasks. This involves registering custom task modules you want to use, as well as, defining custom tasks to create more complex execution workflows.

The third section of the `boltfile.py` is the configuration. Every Bolt File must declare a `config` variable set to a dictionary where the configuration parameters are defined. The dictionary can be empty, but it is required to define the variable. Of course, an empty dictionary will not help us to do much.

The following shows the contents of the `boltfile.py` that we created in the *Getting Started* guide and illustrates the three sections.

```
# Imports section
import bolt

# Task registration section
bolt.register_task('run-tests', ['pip', 'delete-pyc', 'nose'])

# Configuration section
config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
```

(continues on next page)

(continued from previous page)

```
    },
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    },
    'nose': {
        'directory': 'tests'
    }
}
```

Tip: It doesn't make any difference if you include your configuration before the registration of tasks. As a matter of fact, I started doing it that way myself because of the experience I had with Grunt. But my experience has been that it makes the `boltfile.py` more readable if you register your tasks right after the imports, and users can see the tasks available immediately after opening the file. Overtime, your configuration will grow with the usage of Bolt and users will have to scroll all the way down to find the available tasks, which is, usually, the most important part of the file.

The Import Section

The import section of the `boltfile.py` is no different than the imports in any other python script or module. You will always need to import `bolt` to gain access to its API, which is used, among other things, to register tasks. You will import and register other modules containing custom tasks. Finally, you can import any other libraries you might need.

The Task Registration Section

There are different ways to define and register tasks with Bolt. In this section, we will take a look at the different options and when we should use each one of them.

Bolt Provided Tasks

Bolt provides a set of tasks that are always available when executing Bolt. You don't need to register them because Bolt does that for you, and they can be configured without prior registration. This is the case for the tasks shown in the following example, which we will use as starting point.

```
import bolt

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
    },
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    },
    'nose': {
        'directory': 'tests'
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The tasks in the example (`pip`, `delete-pyc`, and `nose`) are provided by Bolt; therefore, we don't need to register them to use them. With this simple example you can still run each task independently to execute them.

```
# Install requirements
bolt pip

# Delete existing .pyc files
bolt delete-pyc

# Execute unit tests
bolt nose
```

As you can see, it is very easy to leverage the existing functionality in Bolt, but the true power comes from the ability to define and create your own tasks or use other tasks provided by tool and library implementers. Let's take a look at other ways to define tasks.

Composing Tasks From Existing Ones

In the example above, we can use any of the three tasks provided by Bolt, but most of the time I will want to run all those tasks together. I want to make sure that when anyone working on my project gets source changes they can have the correct environment setup; therefore, I want them to install any required packages, and execute the tests with a clean run. For that I can define a composite task that will execute all three. The following shows the full contents of the `boltfile.py` after adding the composite tasks.

```
import bolt

bolt.register_task('run-tests', ['pip', 'delete-pyc', 'nose'])
bolt.register_task('default', ['run-tests'])

config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        }
    },
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    },
    'nose': {
        'directory': 'tests'
    }
}
```

We added two additional lines to our bolt file. The first one defines a composite task `run-tests` that execute the previous three. The second line registers a `default` task that executes the previously defined `run-tests`. Both of this tasks will execute the same set of steps.

Now, I can execute `bolt run-tests` from the command line to execute all tasks, or I can simply call `bolt`.

Tip: The `default` task is a special task that gets executed when calling Bolt without specifying a task to execute.

You should always provide a `default` task in your `boltfile.py`.

Tip: You want your `default` task to be compose of the steps you will execute more often. I like to define `default` as the task that I will always execute when I pull new changes from my central repo and before publishing those changes, so I usually include steps to install new required packages, clean the project tree, and execute the unit tests.

Registering Additional Modules

As you start using Bolt more, you will find your-self implementing your own custom tasks or using modules provided by third-party libraries you use (see *Creating Custom Tasks*). In order to use those tasks, you need to import the module containing them and register the module. The following example shows how can your register the tasks in a custom or third-party provided module.

```
# Removed contents for simplicity.
import my_custom_tasks

bolt.register_module_tasks(my_custom_tasks)
```

Now, all the tasks registered by `my_custom_tasks` become available for use and configure (see *Creating Custom Tasks* for more information about how to create your own).

The Power of Configuration

Bolt provides a very powerful configuration mechanism that abstracts what the user wants to do from task implementers that expose configuration settings. This means Bolt gives users the power to describe the configuration parameters of a task, and it takes care of resolving the configuration before it is sent to the task implementation, so that developers implementing tasks get a consistent set of configuration options.

To illustrate how Bolt processes configuration options, I will describe a scenario that I recently run into in one of my projects.

In a recent project, I found myself using the `awscli` and `boto3` libraries available for Python. Without going too much into the details of what I was doing, let's just say that I usually work on a Windows machine, but many of my applications and scripts are executed in Linux; therefore, cross-platform it is very important for my projects (and one of the reasons why I choose Python).

Turns out that when you use `awscli` and/or `boto3` in Windows, you need to install an additional dependency called `pypiwin32`. This dependency is not installed nor can be installed on Linux, so that simple fact threw me out for a few seconds on how I was going to manage the requirements for my project. Thankfully, I had Bolt at my disposal and I was able to fix the problem in a very simple, elegant way.

The first step was to add `awscli` and `boto3` to my `requirements.txt` file.

```
# In requirements.txt
awscli>=1.11
boto3>=1.4
```

Then, I created a second requirements file called `requirements_win.txt` and added the Windows specific library.

```
# In requirements_win.txt
pypiwin32>=219
```

I still want all the people collaborating in my code to have the correct set of requirements, but I don't want them to have to worry about what they need to install because we use bolt for that. So, this is what I did in my bolt file:

```
# Many lines removed for simplicity.

import bolt
import sys

# Define a task to install the requirements.
if sys.platform.startswith('win'):
    bolt.register_task('requirements', ['pip', 'pip.win']) # More on this below.
else:
    bolt.register_task('requirements', ['pip'])


bolt.register_task('run-tests', ['requirements', 'delete-pyc', 'nose'])
bolt.register_task('default', ['run-tests'])


config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': 'requirements.txt'
        },
        'win': {
            'options': {
                'r': 'requirements_win.txt'
            }
        }
    },
}
```

This may seem more complicated than it really is once you understand how Bolt processes configurations, so let's take a look at it step by step.

The first change I made was to check for the OS in which we are running and register a `requirements` task to install the requirements accordingly. Since, the `boltfile.py` is just a Python script, I can import `sys` and create conditional code if I want to.

Now, let's take a look at what I do on Windows because it is something we haven't seen yet `bolt.register_task('requirements', ['pip', 'pip.win'])`. What is this ``pip.win`` thing?

There might be times when I want to configure a task differently depending on the environment I'm running (I will show another example later, but this is so cool that we will expain it first). In those circumstances, instead of providing a completely different `boltfile.py` with a different configuration, Bolt allows me to nest configuration options that I name my self.

The `pip` task knows nothing about the `win` option specified, and it doesn't have to worry about it, but when the `pip` task is invoked as `pip.win`, Bolt takes the configuration options for `pip` and then adds or overwrites any options defined in the nested `win` configuration. Therefore, the configuration passed to the `pip` task when called as `pip.win` will look like the following:

```
config = {
    'commmand': 'install',  # Taken from parent
    'options': {
        'r': 'requirements_win.txt'
    }
}
```

When the task is invoked as `pip`, the configuration passed is:

```
config = {
    'commmand': 'install',  # Taken from parent
    'options': {
        'r': 'requirements.txt'
    }
}
```

In the registration of the `requirements` task for Windows, we execute both, where if we run on Linux we just execute `pip`.

Tip: You can nest configurations as deep as you want, so it will be possible to define tasks as `pip.win.32` and `pip.win.64` if needed. In my experience, one level of nesting is what you will need for most practical cases, and it keeps the configuration readable.

A More Common Configuration Example

The previous example is pretty cool, and it solve a very real problem, but most of the time you will not need or want to have a lot of conditional code in your `boltfile.py`. The following scenario illustrates a more common approach to define and configure tasks differently for different environments.

Many times I find my self wanting to execute a task differently when I run it in my local development environment than when that task is running in the CI/CD pipeline for my project. A very common scenario for all my projects is that when I run the unit tests locally, which I do all the time, I run them with bare options, so I configure the task in the same way as the examples above.

During the build process, however, I want to get more information about the execution of the tests, and I want to produce some reports and post them to my CI/CD system. Usually, I want a tests results report, and a code coverage report. The following shows the tasks I normally register and configure to execute the unit tests in the different environments.

```
# Lines omitted for simplicity.

# Developer's tasks. I like to keep the names short, to type less when
# I run them.
#
bolt.register_task('ut', ['pip', 'delete-pyc', 'nose'])

# Ci/CD Tasks
#
bolt.register_task('run-tests', ['pip', 'nose.ci'])

config = {
    # Again, lines omitted for simplicity.

    'nose': {
        'directory': './tests',
        'ci': {
            'options': {
                'with-xunit': True,
                'xunit-file': os.path.join('output', 'unit_tests_log.xml'),
                'with-coverage': True,
                'cover-erase': True,
```

(continues on next page)

(continued from previous page)

```
                'cover-package': './source',
                'cover-branches': True,
                'cover-html': True,
                'cover-html-dir': os.path.join('output', 'coverage')
            }
        }
    }
}
```

When I'm working on the project, I execute `bolt ut`, which does all the operations I want in my local development environment. In CI/CD, I execute `bolt run-tests`, which runs different tasks, but I want you to focus on the different options that I use with `nose`.

Without using any conditional code in the `boltfile.py` itself, I can run ``nose` in different ways by specifying a nested configuration `ci`.

Tip: If you look at the options set for `nose.ci`, you can see that I use `os.path.join()` to resolve the location where reports will be generated. This illustrates the power of configuration as code.

3.1.3 Bolt Provided Tasks

Bolt provides the implementation of a few common tasks that most Python projects should be able to leverage their functionality. This tasks are registered when Bolt is executed, and users only need to configure and invoke them in their `boltfile.py`.

The following documents the included tasks and how they work.

delete-files

This task deletes files matching a specified `pattern` found in a specified `sourcedir`. A `recursive` flag can be specified to search also in sub-directories of `sourcedir`.

The `pattern` specified follows the matching rules of the Python standard library `glob.glob()` function.

The following example configures the `delete-file` task to delete all the files in a `tmp` directory located at the project root, and all its sub-directories:

```
config = {
    'delete-files': {
        'sourcedir': './tmp',
        'pattern': '*.*',
        'recursive': True
    }
}
```

The `sourcedir` configuration option indicates the directory to search for file matches. This option is required.

The `pattern` option specifies the matching pattern to find files. This option is required.

The `recursive` option indicates if sub-directories should be searched for matches. This option is optional and has a value of `False` by default.

delete-pyc

Searches for `.pyc` in the specified directory and deletes them. The task allows to recursively search sub-directories for `.pyc` files.

The following example shows how to configure the task to recursively delete files from a source directory and its sub-directories:

```
config = {
    'delete-pyc': {
        'sourcedir': './source',
        'recursive': True
    }
}
```

The `sourcedir` option specifies the directory to search for `.pyc` files. This option is required.

The `recursive` option indicates if sub-directories should be searched for matches. This option is optional and has a value of `False` by default.

class bolt.tasks.bolt_delete_files.**DeleteFilesTask**

class bolt.tasks.bolt_delete_files.**DeletePycTask**

mkdir

Creates the directory specified, including intermediate directories, if they do not exist:

```
config = {
    'mkdir': {
        'directory': 'several/intermediate/directories'
    }
}
```

class bolt.tasks.bolt_mkdir.**ExecuteMKDir**

shell

The `shell` task allows executing a shell command with specified arguments inside the bolt execution context. This task comes handy when no bolt specific implementation has been provided for a particular task or to invoke an existing script that should be included as part of the process.

The trade-off of using this task is that commands are system specific and it makes it harder to implement a cross-platform `boltfile.py`.

The task takes a `command` parameter specifying the command to be executed, and an `arguments` option that must be set to a list of string for each of the command line argument tokens to be passed to the tool.

The following example shows how to invoke an existing Python script that takes a few parameters:

```
config = {
    'shell': {
        'command': 'python',
        'arguments': ['existing_script.py', '--with-argument', '-f', '--arg-with', 'a_
↪value']
    }
}
```

Todo: Find a better example.

exception bolt.tasks.bolt_shell.**ShellError**(*shell_code*)

class bolt.tasks.bolt_shell.**ShellExecuteTask**

pip

The `pip` task provides an automation hook to execute `pip` inside of Bolt. In its simplest form, the task does not require any configuration, and it just assumes a `requirements.txt` file is provided at the current working directory, which will be used to execute a `pip install`.

The task also provides a simple form where a `command` and `package` are specified to allow install a single package.

```
config = {
    'pip': {
        'command': 'install',
        'package': 'package_name'
    }
}
```

The supported `pip` functionality can be configured by setting the `command` option to a valid `pip` command, and providing a set of arguments to `pip` as an `options` dictionary where the keys are valid `pip` arguments in short or long form without leading dashes and the values are the respective argument values, or `True` in the case of flags. The following shows a more advance use of this task.

```
config = {
    'pip': {
        'command': 'install',
        'options': {
            'r': './data/project_requirements.txt',
            'target': './requirements',
            'upgrade': True,
            'force-reinstall': True
        }
    }
}
```

class bolt.tasks.bolt_pip.**ExecutePipTask**

exception bolt.tasks.bolt_pip.**PipError**(*pip_code*)

set-vars

Sets environment variables for all specified variable:value pairs. The following shows how the task is configured:

```
config = {
    'set-vars': {
        'vars': {
            'STRING_VAR': 'string_value',
            'INT_VAR': 10
        }
    }
}
```

Numeric vars will be converted to their integer representation.

class bolt.tasks.bolt_set_vars.**SetVarsTask**

setup

The `setup` task provides an automation hook to execute `setup.py` commands and options inside of Bolt. The task, in its simplest form, assumes a default `setup.py` in the current working directory and uses a `build` command as a default if no configuration is provided.

The task configuration allows spcifying a setup script, which by default will be set to `setup.py` if no script is specified, a valid command, and it command arguments. The following example shows how to configure the task.

```
config = {
            'setup':{
                    'script': 'special_setup.py',
                    'command': 'install',
                    'options': {
                            'verbose': True,
                            'dry-run': True
                    }
            }
}
```

exception bolt.tasks.bolt_setup.**BuildSetupError**(*code=4*)

class bolt.tasks.bolt_setup.**ExecuteSetupTask**

nose

Executes unit tests using nose and nosetests as the unit test runner. The task allows to specify the directory where the tests are located through the `directory` parameter and supports all the arguments available in the installed version of nosetests:

```
config = {
    'nose': {
        'directory': 'test/unit',
        'options': {
            'xunit-file': 'output/unit_tests.xml'
            'with-coverage': True,
            'cover-erase': True,
            'cover-package': 'mypackage',
            'cover-html': True,
            'cover-html-dir': 'output/coverage',
        }
}
```

class bolt.tasks.bolt_nose.**ExecuteNoseTask**

exception bolt.tasks.bolt_nose.**NoseError**(*nose_code*)

conttest

This task uses `conttest` to monitor a directory for changes and executes the specified task everytime a change is made. The following configuration is supported:

```
config = {
    'conttest': {
        'task': 'registered_task',
        'directory': './directory/to/monitor/'
    }
}
```

The `task` parameter is the task to be executed and must be registered in `boltfile.py`. The `directory` parameter is the directory (including sub-directories) to monitor for changes.

To use this task, you need to have `conttest` installed, which you can do by calling:

```
pip install conttest
```

class bolt.tasks.bolt_conttest.**ExecuteConttest**

3.1.4 Creating Custom Tasks

The real power of Bolt is the ability to extend its functionality through new tasks that can be registered, configured, and executed through the `boltfile.py`. This section explains the different ways in which you can provide new tasks to bolt and some best practices that you should consider when creating your own tasks.

What is a Bolt Task

A Bolt task is nothing but a Python callable object that will be invoked when its id is scheduled for execution. This means that to provide a new custom task you need to implement the callable and register it with Bolt giving it a unique id, which can be used to invoke the task.

The *Getting Started* guide demonstrated this with a very simple example. Let's revisit it on its own for clarity:

```
import bolt

def greeting(**kwargs):
    config = kwargs.get('config')
    message = config.get('message')
    print(message)


bolt.register_task('greet', greeting)

config = {
    'greet': {
        'message': 'Hello from Bolt!'
    }
}
```

In this example, we implement our task right in the `boltfile.py`. As you will see later, the recommended way is to create your tasks in their own package or modules so you can re-use them, but this example will help us understand how things work.

Right after the `import` statement, you have our task callable, which is a function that takes a set of keyword arguments (`**kwargs`). The function reads the `config` argument and from it, it extracts the `message` we want to display.

The next step is to register the callable with Bolt, which is done by the call to `bolt.register_task()`. We pass a unique identifier to our task and the callable that will be invoked.

Finally, we use the unique identifier for the task in our `config` to configure the message that we want to display.

As you can see, it is very simple to add a new custom task, but you will want to implement your tasks in a way that you can re-use them in different projects. The best way to do that is by creating your own modules or packages containing the Bolt tasks and then install them as requirements to the projects that use them. Let's take a look at that.

Implementing Custom Bolt Task Modules/Packages

Like any other python tool or application, you want to implement your Bolt tasks in their own modules or packages, so that you can install them as requirements in the projects you use them. The process to implement the tasks in a module is the same as above. The only difference is that you will have to provide a mechanism to register those tasks with Bolt, so they become available. Let's take a look at an example:

```
# in my_bolt_module.py

def greeting(**kwargs):
    config = kwargs.get('config')
    message = config.get('message')
    print(message)


def register_tasks(registry):
    registry.register_task('greet', greeting)
```

As we discussed, the implementation stays the same, but we added a `register_tasks()` function that takes a `registry` parameter, which allows us to make available the task to clients.

Now if someone wants to use our task, they can install the module and add it to the `boltfile.py`:

```
import bolt
import my_bolt_module

bolt.register_module_tasks(my_bolt_module)

config = {
    'greet': {
        'message': 'Hello from Bolt!'
    }
}
```

In this example, we first import the module containing the tasks, and then we register them by calling `bolt.register_module_tasks()`. Bolt will create the `registry` instance and pass it to the registration function in the module, which will make the task available.

Note: The `bolt.register_task()` function grabs the instance of the `registry` and delegates to its method to register the task. Even-thought the result is the same, you should always use `bolt.register_task()` in your `boltfile.py` and `registry.register_task()` in the `register_task()` function of your custom modules.

Using a Callable Class to Implement Bolt Tasks

Bolt tasks are callable objects; therefore, you can implement your task in a callable class. The following example shows how to implement the `greet` task in a callable class:

```
# in my_bolt_module.py

class GreetingTask(object):

    def __call__(**kwargs):
        config = kwargs.get('config')
        message = config.get('message')
        print(message)


def register_tasks(registry):
    registry.register_task('greet', GreetingTask())
```

This example implements the same task, but it uses a callable class (a class that implements a `__call__()` method) to implement the functionality. When the task is registered, we use a class instance as opposed to the function name as the callable registration. Other than that, the code is the same.

You may ask your-self why use a class when implementing a function is simpler. For very simple tasks, a function will work fine. When I started working on Bolt, most of the standard tasks were implemented as functions. Overtime, I reealize that classes will suit me better for the following reasons:

Classes are more suitable for testing. I write all my code using a TDD (Test Driven Development) process, and you should too. Unit testing functions that return a result is very simple, but testing functions with side-effects, it is a little bit more complicated. It didn't take long to see that most task were accessing external resources or code that will perform operations but will not return any useful values. In these cases, unit testing a function becomes very difficut, because it is hard to mock a specific state. Using a class makes unit testing simpler because you can always set the class to a desired state.

Classes simplify passing parameters. In our examples, we are dealing with just one option in our configuration. As soon as you start supporting more configuration options, you have to deal with validation of those options and conditional code that depends of values of those parameters. Classes work a lot better because you can have internal implementation methods that can access those options as data members, as opposed to having to pass them as parameters to other functions.

Classes can keep alive resources after execution. Imagine a task that needs to start a web-server to make a service available, while subsequent tasks run tests against the server. This task will have to start the service in a separate process and keep it running until the tests are done, but it will be nice to shut down the server once the test is complete. As we will see below, Bolt supports a `tear_down()` method that gets invoked at the end, and where resources can be freed. This can only be done with classes and not with functions.

The Execution Context

We have seen how to create new tasks and how support configuration options for them. But once in a while, you will run into a situation where it will be nice to share some data or state among different tasks. In those situations, you can use the execution context object.

The execution context is a Python dictionary like object where you can store key/value pairs to share them with subsequent tasks.

Tip: The context object is a plain Python dictionary that is passed to every task being executed, but this might change in the future, so you should assume that the only available interface for this object is that of a dictionary.

I am not a big fan of sharing data between tasks because it can create unwanted dependencies among otherwise independent tasks, but I also recognize that it is a concept that may come handy in certain situations. In general, try to avoid task implementations that rely on certain properties available in the context object and always provide suitable

defaults in case the properties are missing. Let's take a look at a scenario where the execution context might come handy.

Assume we are writing a task that requires a job name from a service and doing so, it is an expensive operation. Furthermore, there is group of tasks that will use that job name, so you want to retrieve it once and use it in all other tasks.

In a situation like this, we will write a task that retrieves the job name and stores it in the context object, so subsequent tasks can use it (I'm using functions for simplicity, but I prefer classes).

```
# In my_job_tasks.py

def retrieve_job_name(**kwargs):
    config = kwargs.get('config')
    job_id = config.get('job-id')
    manager = JobManager()
    job = manager.get_job(job_id)    # Very expensive operation.
    context = kwargs.get('context')
    context['job-name'] = job.name


def notify_job_name(**kwargs):
    config = kwargs.get('config')
    context = kwargs.get('context')
    job_name = config.get('job-name') or context.get('job-name')
    notifier = Notifier()
    notifier.notify_job_name(job_name)


def register_tasks(registry):
    registry.register_task('retrieve-job', retrieve_job_name)
    registry.register_task('notify-job-name', notify_job_name)
```

As you can see, the `retrieve_job_name` task retrieves the job name and stores it in the context object. Then, the value is used by the `notify_job_name` task. Notice how we still try to retrieve the job name from the task `config`. This allows to override that value in the `boltfile.py` which might come handy during testing.

Tip: When implementing a task that relies on some information stored in the context object, think about whether there is a suitable default or might be convenient to override the value through the configuration.

Tip: Avoid creating dependencies between tasks by over-using the context object. However, you'll find that some times it is a very handy feature.

3.2 Contributing to Bolt

Todo: Write contribute topic.

3.3 Documentation ToDos

The following is a list of ToDos to improve the documentation. Ideally, we won't have any entries here.

Todo: Write contribute topic.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/bolt-task-automation/checkouts/latest/docs/source/contribute.rst, line 5.)

Todo: Find a better example.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/bolt-task-automation/checkouts/latest/bolt/tasks/bolt_shell.py:docstring of bolt.tasks.bolt_shell, line 26.)

Python Module Index

b

Index