ZeroVM Documentation

Release latest

The ZeroVM Team

October 20, 2015

Contents

ZeroVM is a lightweight virtualization technology based on Google Native Client (NaCl). It is a sandbox which isolates and runs a single process at a time, unlike other virtualization and container technology which provide an entire virtualized operating system and execution environment capable of running multiple processes.

ZeroCloud is a platform based on ZeroVM and Openstack Swift which can be used to run all sorts of applications, including REST services, MapReduce, and batch processing of any kind. Applications which deal with a lot of data or require significant parallel processing are a good fit for ZeroCloud.

CHAPTER 1

Where should I start?

If you're interested in learning in depth about the core ZeroVM sandbox technology, check out the *ZeroVM core documentation*.

If you're interested in developing web applications, MapReduce applications, or just to need handle large amounts of data, you don't need to know too many details about the core ZeroVM sandbox technology; you can skip straight to the *ZeroCloud section*.

CHAPTER 2

ZeroVM: Lightweight, single process sandbox

2.1 ZeroVM Overview

The primary function of ZeroVM is to isolate a single application and provide a secure sandbox which cannot be broken out of. In other words, "untrusted" code can run safely inside ZeroVM without breaking out and compromising the host operating system. One can easily appreciate the utility of this by looking at applications like the Python.org shell and the Go Playground.

This comes with some necessary limitations. Run time, memory usage and I/O must be carefully controlled to prevent abuse and resource contention among processes.

2.2 Isolation and Security

ZeroVM has two key security components: static binary validation and a limited system call API.

Static binary validation works by ensuring that untrusted code does not execute any unsafe instructions. All jumps must target "the start of 32-byte-aligned blocks, and instructions are not allowed to straddle these blocks". (See http://en.wikipedia.org/wiki/Google_Native_Client and http://research.google.com/pubs/pub34913.html.) The big advantage of this is that validation can be performed just once before executing the untrusted program, and no further validation or interpretation is required. All of this is provided by Native Client and is not unique to ZeroVM.

The second security component–a limited syscall API–is a major differentiator between plain NaCl and ZeroVM. In ZeroVM, only six system calls are available:

- `pread`
- `pwrite`
- `jail`
- `unjail`
- `fork`
- `exit`

This minimizes potential attack surfaces and facilitates security audits of the core isolation mechanisms. Compare this to the standard NaCl system calls, of which there are more than 50.

2.3 In-memory File System

A virtual in-memory file system is made available to ZeroVM by the ZeroVM RunTime environment (ZRT). Writes which occur at runtime are completely thrown away. The only way to persistently write data is to map a file in the virtual file system to a file or other device on the host operating system. This is accomplished by the use of *channels*.

Arbitrary file hierarchies can be loaded into a ZeroVM instance by mounting tar archives as disk images. This is necessary particularly in the case of the ZeroVM Python interpreter port, which requires not only a cross-compiled Python interpreter executable but also the Python standard library packages and modules; these files must accessible for Python programs to run inside ZeroVM. Any tarball can be mounted to the virtual file system, and multiple images can be mounted simultaneously. Each tarball is mounted by defining a channel, just as with any other file. (Note that there is a *limit <zerovm-manifest-channel-max>* to the number of channels per instance.)

2.4 Channels and I/O

All I/O in ZeroVM is modeled through an abstraction called "channels". Channels act as the communication medium between the host operating system and a ZeroVM instance. On the host side, the channel can be pretty much anything: a file, a pipe, a character device, or a TCP socket. Inside a ZeroVM instance, the all channels look like files.

2.4.1 Channels Restrictions

The most important thing to know about channels is that they must be declared prior to starting a ZeroVM instance. This is no accident, and it bears important security implications. For example: It would be impossible for user code (which is considered to be "untrusted") to open and write to a socket, *unless* the socket is declared beforehand. The same goes for files stored on the host; there is no way to read from or write to host files unless the file channels are declared in advance.

Channels also have several attributes to further control I/O behavior. Each channel defintion must declare:

- number of read operations
- number of write operations
- total bytes limit for reads
- total bytes limit for writes

If channel limits are exceeded at runtime, the ZeroVM trusted code will raise an `EDQUOT` (Quota exceeded) error.

2.4.2 Channels Definitions

In addition to read/write limits, channel definitions consist of several other attributes. Here is a complete list of channel attributes, including I/O limits:

- `uri`: Definition of a device on the host operating system. This can be a normal file, a TCP socket, a pipe, or a character device.

  For files, pipes, and character devices, the value of the `uri` is simply a file system path, e.g., `/home/me/foo.txt`.

  TCP socket definitions have the following format: `tcp:<host>:<port>`, where `<host>` is an IP address or hostname and `<port>` is the TCP port.

- `alias`: A file alias inside ZeroVM which maps to the device specified on the host by `uri`. Regardless of the host type of the device, everything looks like a file inside a ZeroVM instance. That is, even a socket will appear as a file in the virtual in-memory file system, e.g, `/dev/mysocket`. Aliases can have arbitrary definitions.
- **`type`: Choose from the following enumeration:**
  - 0 (sequential read / sequential write)
  - 1 (random read / sequential write)
  - 2 (sequential read / random write)
  - 3 (random read / random write)
- `etag`: Typically disabled (0). When enabled (1), record and report a checksum of all of the data which passed through the channel (both read and written.
- `gets`: Limit on the number of read operations for this channel.
- `get_size`: Limit on the total number of bytes which can be read from this channel.
- `puts`: Limit on the number of write operations for this channel.
- `put_size`: Limit on the total number of bytes which can be read from this channel.

Channels limits must be an integer value from 1 to 4294967296 (2^32).

If a *ZeroVM manifest file* (a plain-text file), channels are defined using the following format:

```
Channel = <uri>,<alias>,<type>,<etag>,<gets>,<get_size>,<puts>,<put_size>
```

Here are some examples:

```
Channel = /home/me/python.tar,/dev/1.python.tar,3,0,4096,4096,4096,4096
Channel = /dev/stdout,/dev/stdout,0,0,0,0,1024,1024
Channel = /dev/stdin,/dev/stdin,0,0,1024,1024,0,0
Channel = tcp:192.168.0.10:27175,/dev/myserver,3,0,65536,65536,65536,65536
```

2.5 ZeroVM Manifest

A manifest file is the most primitive piece of input which must be provided to ZeroVM. Here is an example manifest:

```
Version = 20130611
Timeout = 50
Memory = 4294967296,0
Program = /home/me/myapp.nexe
Channel = /dev/stdin,/dev/stdin,0,0,8192,8192,0,0
Channel = /dev/stdout,/dev/stdout,0,0,0,0,8192,8192
Channel = /dev/stderr,/dev/stderr,0,0,0,0,8192,8192
Channel = /home/me/python.tar,/dev/1.python.tar,3,0,8192,8192,8192,8192
Channel = /home/me/nvram.1,/dev/nvram,3,0,8192,8192,8192,8192
```

The file consists of basic ZeroVM runtime configurations and one or more *channels*.

Required attributes:

- `Version`: The manifest format version.
- `Timeout`: Maximum life time for a ZeroVM instance (in seconds). If a user program (untrusted code) exceeds the time limit, the ZeroVM executable will return an error.

- `Memory`: Contains two 32-bit integer values, separated by a comma. The first value specifies the amount of memory (in bytes) available for the user program, with a maximum of 4294967296 bytes (4 GiB). The second value (0 for disable, 1 for enable) sets memory entity tagging. FIXME: This etag feature is undocumented.
- `Program`: Path to the untrusted executable (cross-compiled to NaCl) on the host file system.

2.5.1 Channel Definition Limit

A manifest can define a maximum of 10915 channels.

2.6 Embeddability

The lightweightness and security that ZeroVM provides makes it ideal for embedding computation in virtually any existing system, particularly data storage systems. You can embed "untrusted" computation in any system simply by invoking ZeroVM as a subprocess, for example. Think "stored procedures", but much more powerful.

A prime example of this embeddability is *ZeroCloud*, which facilitates data-local computation inside of OpenStack Swift.

CHAPTER 3

ZeroCloud: Cloud storage & compute platform

3.1 ZeroCloud Overview

ZeroCloud is a converged cloud storage and compute platform, powered by OpenStack Swift and ZeroVM. It combines the massively scalable object storage capabilities of Swift with the application isolation of ZeroVM to create a platform for developing cloud applications.

The key concept of ZeroCloud is data-local computation: instead of pushing and pull data between the data storage nodes and computation nodes, we send the application to the data and do the work the in-place on the storage system. Another way to think about it: it's like stored procedures (for a relational database), but *much* more powerful and scalable.

3.1.1 Use cases

Not all types of applications are suitable for the ZeroCloud platform. The only real data persistence mechanism available is OpenStack Swift object storage, which is eventually consistent. This makes ZeroCloud most suitable for highly-available and partition-tolerant applications which process a lot of data.

Here are some types of applications which are a good match for ZeroCloud:

- MapReduce (searching, sorting, analytics, etc.)
- Batch processing (logs, images, etc.)
- File server/repository applications

ZeroCloud also provides tools for easily building REST services for your applications.

3.1.2 Language support

Currently, you can only write applications for ZeroCloud using Python and C/C++. This limitation comes from the ZeroVM execution environment, which requires applications to be cross-compiled to NaCl for validation and secure execution.

The recommended development language for ZeroCloud is Python. Python 2.7.3 has been ported to NaCl and is available to use on ZeroVM and ZeroCloud. See https://github.com/zerovm/zpython2.

3.2 Setting up a development environment

The easiest way to get up and running is to install and run ZeroCloud on DevStack inside VirtualBox. We provide some Vagrant scripts to make the setup require little effort.

With this environment, you can not only write and run applications on ZeroCloud, but you can also hack on ZeroCloud on itself.

Note to Linux users: There should be packages available (using the favored package manager of your distro) for just about everything you need to install.

3.2.1 Install VirtualBox

Download and install VirtualBox: https://www.virtualbox.org/wiki/Downloads.

3.2.2 Install Vagrant

Download and install Vagrant: https://www.vagrantup.com/downloads.html.

3.2.3 Clone the ZeroCloud source code

First, you'll need to download and install Git: http://git-scm.com/downloads

From a command line, clone the soure code. In this example, we just checkout the code into a `zerocloud` folder in our home directory.

```
$ git clone https://github.com/zerovm/zerocloud.git $HOME/zerocloud
```

3.2.4 vagrant up

Now we can actually start OpenStack Swift and ZeroCloud, inside a VM:

```
$ cd $HOME/zerocloud/contrib/vagrant
$ vagrant up
```

`vagrant up` will download and install DevStack and configure it to run Swift with the ZeroCloud middleware. This usually takes about 10-15 minutes.

3.2.5 Install and configure command line clients

To interact with and test your ZeroCloud deployment, you'll need to install a handful of tools. You can install all of these tools from PyPI using `pip` on your Vagrant/Virtual Box VM.

```
$ sudo pip install python-swiftclient python-keystoneclient zpm
```

Note: `zpm` (ZeroVM Package Manager) is a tool which make it easier to develop, package, and deploy applications for ZeroCloud.

To authenticate with your ZeroCloud installation, you'll need to set up your credentials in some environment variables. A configuration file is provided for convenience in `$HOME/zerocloud/contrib/vagrant`.

```
$ source /vagrant/adminrc
```

You can test your client configuration by running `zpm auth`:

```
$ zpm auth
Auth token: PKIZ_Zrz_Qa5NJm44FWeF7Wp...
Storage URL: http://127.0.0.1:8080/v1/AUTH_7fbcd8784f8843a180cf187bbb12e49c
```

If you see output that looks like this, everything should be in order.

Setting a couple of environment variables with these values will make commands more concise and convenient to execute:

```
$ export OS_AUTH_TOKEN=PKIZ_Zrz_Qa5NJm44FWeF7Wp...
$ export OS_STORAGE_URL=http://127.0.0.1:8080/v1/AUTH_7fbcd8784f8843a180cf187bbb12e49c
$ export OS_STORAGE_ACCOUNT=AUTH_7fbcd8784f8843a180cf187bbb12e49c
```

3.2.6 Restarting DevStack

If you ever need to kill/restart DevStack (as is the case if you modify ZeroCloud code or change Swift configuration settings), first log in to the vagrant box:

```
$ vagrant ssh
```

Next, we need to terminate all of the DevStack processes. The first time you do this, you need to use a little brute force. First, run rejoin-stack.sh:

```
$ cd $HOME/devstack
$ ./rejoin-stack.sh
```

This will put you into a screen session. To terminate DevStack, press 'ctrl+a backslash', then 'y' to confirm. NOTE: The first time you restart DevStack after provisioning the machine, not all of the Swift processes will be killed. A little brute force is needed:

```
$ ps ax | grep [s]wift | awk '{print $1}' | xargs kill
```

Now restart DevStack:

```
$ cd $HOME/devstack
$ ./rejoin-stack.sh
```

Subsequent restarts are easier. Run `./rejoin-stack.sh` as above, press 'ctrl+a backslash', 'y' to confirm, then run `./rejoin-stack.sh` again.

To log out of the vagrant box and keep everything running, press 'ctrl+a d' to detach from the screen session. You can now log out of the box ('ctrl+d').

3.3 Running code on ZeroCloud

These examples below include executing code using just plain old `curl` commands on the command line, as well as scripting using Python and the requests module.

Jump to a section:

- *POST a ZeroVM image*
- *POST a job description to a ZeroVM application*
- *Run a ZeroVM application with an object GET*
- *MapReduce application*

3.3.1 Setup: Getting an auth token

The first thing you need to do is get an auth token and find the storage URL for your account in Swift. For convenience, you can get this information simply by running `zpm auth`:

```
$ zpm auth
Auth token: PKIZ_Zrz_Qa5NJm44FWeF7Wp...
Storage URL: http://127.0.0.1:8080/v1/AUTH_7fbcd8784f8843a180cf187bbb12e49c
```

Setting a couple of environment variables with these values will make commands more concise and convenient to execute:

```
$ export OS_AUTH_TOKEN=PKIZ_Zrz_Qa5NJm44FWeF7Wp...
$ export OS_STORAGE_URL=http://127.0.0.1:8080/v1/AUTH_7fbcd8784f8843a180cf187bbb12e49c
```

3.3.2 POST a Python script

This is the simplest and easiest way to execute code on ZeroCloud.

First, write the following the code into a file called `example`.

```
#!file://python2.7:python
import sys
print("Hello from ZeroVM!")
print("sys.platform is '%s'" % sys.platform)
```

Execute it using `curl`:

```
$ curl -i -X POST -H "X-Auth-Token: $OS_AUTH_TOKEN" \
  -H "X-Zerovm-Execute: 1.0" -H "Content-Type: application/python" \
  --data-binary @example $OS_STORAGE_URL
```

Using a Python script:

```
import os
import requests

storage_url = os.environ.get('OS_STORAGE_URL')
headers = {
    'X-Zerovm-Execute': 1.0,
    'X-Auth-Token': os.environ.get('OS_AUTH_TOKEN'),
    'Content-Type': 'application/python',
}

with open('example') as fp:
    response = requests.post(storage_url,
                             data=fp.read(),
                             headers=headers)
    print(response.content)
```

You can write and execute any Python code in this way, using any of the modules in the standard library.

3.3.3 POST a ZeroVM image

Another way to execute code on ZeroCloud is to create a specially constructed tarball (a "ZeroVM image") and `POST` it directly to ZeroCloud. A "ZeroVM image" is a tarball with at minimum a `boot/system.map` file. The `boot/system.map`, or job description, contains runtime execution information which tells ZeroCloud what to execute.

This is useful if your code consists of multiple source files (not just a single script). You can pack everything into a single file and execute it. This method is also useful if you want to just execute something once, meaning that once ZeroCloud executes the application, the app is thrown away.

In this example, we'll do just that. Create the following files:

`mymath.py`:

```
def add(a, b):
    return a + b
```

`main.py`:

```
import mymath
a = 5
b = 6
the_sum = mymath.add(a, b)
print("%s + %s = %s" % (a, b, the_sum))
```

Create a `boot` directory, then `boot/system.map` file:

```
[{
    "name": "example",
    "exec": {
        "path": "file://python2.7:python",
        "args": "main.py"
    },
    "devices": [
        {"name": "python2.7"},
        {"name": "stdout"}
    ]
}]
```

Create the ZeroVM image:

```
$ tar cf example.tar boot/system.map main.py mymath.py
```

Execute the ZeroVM image directly on ZeroCloud using `curl`:

```
$ curl -i -X POST -H "Content-Type: application/x-tar" \
  -H "X-Auth-Token: $OS_AUTH_TOKEN" -H "X-Zerovm-Execute: 1.0" \
  --data-binary @example.tar $OS_STORAGE_URL
```

Using a Python script:

```
import os
import requests

storage_url = os.environ.get('OS_STORAGE_URL')
headers = {
    'X-Zerovm-Execute': 1.0,
    'X-Auth-Token': os.environ.get('OS_AUTH_TOKEN'),
    'Content-Type': 'application/x-tar',
}
```

```
with open('example.tar') as fp:
    response = requests.post(storage_url,
                             data=fp.read(),
                             headers=headers)
    print(response.content)
```

3.3.4 POST a job description to a ZeroVM application

This method is useful if you want to execute the same application multiple times, for example, to run an application to process multiple different files.

In this example, we will upload a packaged application into Swift and then subsequently POST job descriptions to execute the application. This can be done multiple times, and with different arguments. We'll use this to build a small application. Create a directory `sampleapp` and in it, create the following files:

`main.py`:

```
import csv
with open('/dev/input') as fp:
    reader = csv.reader(fp)

    for id, name, email, balance in reader:
        print('%(name)s: %(balance)s' % dict(name=name, balance=balance))
```

Create an `example.tar` containing the Python script:

```
$ tar cf example.tar main.py
```

Create a container for the application:

```
$ swift post example
```

Upload the image into Swift:

```
$ swift upload example example.tar
```

Now we need to create a couple of files for the application to read and process.

`data1.csv`:

```
id,name,email,balance
1,Alice,alice@example.com,1000
2,Bob,bob@example.com,-500
```

`data2.csv`:

```
id,name,email,balance
3,David,david@example.com,15000
4,Erin,erin@example.com,25000
```

Upload the data files into Swift:

```
$ swift upload example data1.csv data2.csv
```

`job.json`:

```
[{
    "name": "example",
    "exec": {
```

```
        "path": "file://python2.7:python",
        "args": "main.py"
    },
    "devices": [
        {"name": "python2.7"},
        {"name": "stdout"},
        {"name": "input", "path": "swift://~/example/data1.csv"},
        {"name": "image", "path": "swift://~/example/example.tar"}
    ]
}]
```

Execute it using `curl`:

```
$ curl -i -X POST -H "Content-Type: application/json" \
  -H "X-Auth-Token: $OS_AUTH_TOKEN" -H "X-Zerovm-Execute: 1.0" \
  --data-binary @job.json $OS_STORAGE_URL
```

Execute it using a Python script:

```
import os
import requests

storage_url = os.environ.get('OS_STORAGE_URL')
headers = {
    'X-Zerovm-Execute': 1.0,
    'X-Auth-Token': os.environ.get('OS_AUTH_TOKEN'),
    'Content-Type': 'application/json',
}

with open('job.json') as fp:
    response = requests.post(storage_url,
                             data=fp.read(),
                             headers=headers)
    print(response.content)
```

You can process a different input file by simply changing the `job.json` and re-running the application (using `curl` or the Python script above). For example, change this line

```
{"name": "input", "path": "swift://~/example/data1.csv"},
```

to this:

```
{"name": "input", "path": "swift://~/example/data2.csv"},
```

Your `job.json` file should now look like this:

```
[{
    "name": "example",
    "exec": {
        "path": "file://python2.7:python",
        "args": "main.py"
    },
    "devices": [
        {"name": "python2.7"},
        {"name": "stdout"},
        {"name": "input", "path": "swift://~/example/data2.csv"},
        {"name": "image", "path": "swift://~/example/example.tar"}
    ]
}]
```

Try running that and see the difference in the output:

```
$ curl -i -X POST -H "Content-Type: application/json" \
  -H "X-Auth-Token: $OS_AUTH_TOKEN" -H "X-Zerovm-Execute: 1.0" \
  --data-binary @job.json $OS_STORAGE_URL
```

3.3.5 Run a ZeroVM application with an object GET

It is possible to attach applications to particular types of objects and run that application when the object is retrieved (using a GET request) from Swift.

In this example, we'll write an application which processes JSON file objects and returns a pretty-printed version of the contents. The idea here is that we take some raw JSON data and make it more human-readable.

Create the following files in a new directory `sampleapp2`:

`data.json`:

```
{"type": "GeometryCollection", "geometries": [{ "type": "Point", "coordinates": [100.0, 0.0]}, {"type
```

`prettyprint.py`:

```
import json
import pprint

with open('/dev/input') as fp:
    data = json.load(fp)
    print(pprint.pformat(data))
```

`config`:

```
[{
    "name": "prettyprint",
    "exec": {
        "path": "file://python2.7:python",
        "args": "prettyprint.py"
    },
    "devices": [
        {"name": "python2.7"},
        {"name": "stdout"},
        {"name": "input", "path": "{.object_path}"},
        {"name": "image", "path": "swift://~/example/prettyprint.tar"}
    ]
}]
```

Upload the test data:

```
$ swift post example  # creates the container, if it doesn't exist already
$ swift upload example data.json
```

Bundle and upload the application:

```
$ tar cf prettyprint.tar prettyprint.py
$ swift upload example prettyprint.tar
```

Upload the configuration to a `.zvm` container:

```
$ swift post .zvm  # creates the container, if it doesn't exist already
$ swift upload .zvm config --object-name=application/json/config
```

Now submit a GET request to the file, and it will be processed by the `prettyprint` application. Setting the `X-Zerovm-Execute` header to `open/1.0` is required to make this work. (Without this header you'll just get the raw file, unprocessed.)

Using `curl`:

```
$ curl -i -X GET $OS_STORAGE_URL/example/data.json \
  -H "X-Zerovm-Execute: open/1.0" -H "X-Auth-Token: $OS_AUTH_TOKEN"
```

Using a Python script:

```
import os
import requests

storage_url = os.environ.get('OS_STORAGE_URL')
headers = {
    'X-Zerovm-Execute': 'open/1.0',
    'X-Auth-Token': os.environ.get('OS_AUTH_TOKEN'),
}

response = requests.get(storage_url + '/example/data.json',
                        headers=headers)
print(response.content)
```

3.3.6 MapReduce application

This example is a parallel wordcount application, constructed to utilize the MapReduce features of ZeroCloud.

Create the project directory

Create a directory for the project files. For example:

```
$ mkdir ~/mapreduce
```

Then change into that directory:

```
$ cd ~/mapreduce
```

Create Swift containers

We need to create two containers in Swift: one to hold our application data, and one to hold the application itself.

```
$ swift post mapreduce-data
$ swift post mapreduce-app
```

Upload sample data

Create a couple of text files and upload them into the `mapreduce-data` container. You can use the samples below, or any text you like.

`mrdata1.txt`:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut diam sapien,
dictum eleifend erat in, luctus pellentesque est. Aliquam diam est,
tincidunt ac bibendum non, vehicula ut enim. Sed vitae mi orci. Nam
scelerisque diam ut orci iaculis dictum. Fusce consectetur consectetur
risus ut porttitor. In accumsan mi ut velit venenatis tincidunt. Duis id
dapibus velit, nec semper odio.  Quisque auctor massa vitae vulputate
venenatis. Pellentesque velit eros, pretium in hendrerit a, viverra vitae
neque. Vivamus mattis vehicula lectus vel fringilla. Curabitur sem urna,
condimentum nec lectus non, tristique elementum sapien. Quisque luctus
ultrices ante sed dignissim. Integer non commodo enim, quis semper diam.
```

`mrdata2.txt`:

```
Curabitur pulvinar diam eros, eget varius justo hendrerit sed. Maecenas
hendrerit aliquam libero id mollis. Donec semper sapien tellus, sed
elementum dolor ornare eu. Vestibulum lacinia mauris quis ipsum porta, ut
lobortis sapien consectetur. Sed quis pretium justo, mattis aliquet nisl.
Donec vitae elementum lectus. Morbi fringilla augue non elit pulvinar, non
fermentum quam eleifend. Integer ac sodales lorem, a iaculis sapien.
Phasellus vel sodales lorem. Integer consequat varius mi in pretium.
Aliquam iaculis viverra vestibulum. Ut ut arcu sed orci malesuada pulvinar
sit amet sed felis. Nullam eget laoreet urna. Sed eu dapibus quam. Nulla
facilisi. Aenean non ornare lorem.
```

`mrdata3.txt`:

```
Vivamus lacinia tempor massa at molestie. Aenean non erat leo. Curabitur
magna diam, ultrices quis eros quis, ornare vehicula turpis. Donec
imperdiet et mi id vestibulum. Nullam tincidunt interdum tincidunt. Nullam
eleifend vel mauris in bibendum. Maecenas molestie est ac rhoncus
elementum. Duis imperdiet hendrerit congue. Quisque facilisis neque a
semper egestas. Vestibulum nec lacus diam.  Nam vitae volutpat lacus.
Donec sodales dui est, ac malesuada arcu sodales vitae.
```

Upload the files:

```
$ swift upload mapreduce-data mrdata1.txt
$ swift upload mapreduce-data mrdata2.txt
$ swift upload mapreduce-data mrdata3.txt
```

Add `zapp.yaml`

Add a ZeroVM application configuration template:

```
$ zpm new
```

This will create a `zapp.yaml` file in the current directory. Open `zapp.yaml` in your favorite text editor.

First, give the application a name, by changing the

Change the `execution` section

```
execution:
  groups:
    - name: ""
      path: file://python2.7:python
      args: ""
      devices:
```

```
      - name: python2.7
      - name: stdout
```

to look like this:

```
execution:
  groups:
    - name: "map"
      path: file://python2.7:python
      args: "mapper.py"
      devices:
      - name: python2.7
      - name: stdout
      - name: input
        path: "swift://~/mapreduce-data/*.txt"
      connect: ["reduce"]
    - name: "reduce"
      path: file://python2.7:python
      args: "reducer.py"
      devices:
      - name: python2.7
      - name: stdout
```

Note: The `connect` directive enables communication from the first execution group to the second. The creates a data pipeline where the results from the `map` execution, run on each text file in the `mapreduce-data` container, can be piped to the `reduce` part and combined into a single result.

We also need to update the `bundling` section

```
bundling: []
```

to include two Python source code files (which we will create below):

```
bundling: ["mapper.py", "reducer.py"]
```

The code

Now let's write the code that will do our MapReduce wordcount.

`mapper.py`:

```
import os

# Word count:
with open('/dev/input') as fp:
    data = fp.read()

with open('/dev/out/reduce', 'a') as fp:
    path_info = os.environ['LOCAL_PATH_INFO']

    # Split off the swift prefix
    # Just show the container/file
    shorter = '/'.join(path_info.split('/')[2:])
    # Pipe the output to the reducer:
    print >>fp, '%d %s' % (len(data.split()), shorter)
```

`reducer.py`:

```
import os
import math

inp_dir = '/dev/in'

total = 0
max_count = 0

data = []

for inp_file in os.listdir(inp_dir):
    with open(os.path.join(inp_dir, inp_file)) as fp:
        for line in fp:
            count, filename = line.split()
            count = int(count)
            if count > max_count:
                max_count = count
            data.append((count, filename))
            total += count

fmt = '%%%sd %%s' % (int(math.log10(max_count)) + 2)

for count, filename in data:
    print fmt % (count, filename)
print fmt % (total, 'total')
```

Bundle, deploy, and execute

Bundle:

```
$ zpm bundle
created mapreduce.zapp
```

Deploy:

```
$ zpm deploy mapreduce-app mapreduce.zapp
```

Execute:

```
$ zpm execute mapreduce.zapp --container mapreduce-app
 104 mapreduce-data/mrdata1.txt
 101 mapreduce-data/mrdata2.txt
  69 mapreduce-data/mrdata3.txt
 274 total
```

3.4 Developing Python applications

In the section *Running code on ZeroCloud* we saw some examples of running basic Python programs. This tutorial will cover Python application development in more depth, including more detailed descriptions of the directives in the application template (`zapp.yaml`). It will also cover how to include third party Python libraries in your zapp.

3.4.1 Python application template

The most covenient way to build Python applications on ZeroCloud is use to utilities build into the `zpm` tool.

To create a new application, simply run:

```
$ zpm new --template python
Created './zapp.yaml'
Created './.zapp'
Created './.zapp/tox.ini'
```

Notice that this creates a couple of files and a directory. `zapp.yaml` is this application template, and most of time you'll be modifying this file to make changes to your application config. `.zapp/` is "hidden" directory which contains extra artifacts to assist with bundling your application, including `tox.ini` which is used to download and cache third-party (pure) Python dependencies. For the most part, you will not need to directly change anything in the `.zapp/` directory, although some corner cases <link> require this.

The default `zapp.yaml` should look something like this:

```
# This describes the type of application. Bundling and deployment
# behavior can vary between application types.
project_type: python

# This section describes the runtime behavior of your zapp: which
# groups of nodes to create and which nexe to invoke for each.
execution:

  # Your application can consist of multiple groups. This is typically
  # used for map-reduce style jobs. This is a list of groups, so
  # remember to add "-" infront of each group name.
  groups:

      # Name of this group. This is used if you need to connect groups
      # with each other.
    - name: ""

      # The NaCl executable (nexe) to run on the nodes in this group.
      path: file://python2.7:python

      # Command line arguments for the nexe.
      args: ""

      # Input and output devices for this group.
      devices:
      - name: python2.7
      - name: stdout

# Meta-information about your zapp.
meta:
  Version: ""
  name: "myapp"
  Author-email: ""
  Summary: ""

help:
  # Short description of your zapp. This is used for auto-generated
  # help.
  description: ""

  # Help for the command line arguments. Each entry is a two-tuple
  # with an option name and an option help text.
  args: []
```

```
# Files to include in your zapp. Your can use glob patterns here, they
# will be resolved relative to the location of this file.
bundling: []
```

Let's look at each section in a bit more detail:

```
# This describes the type of application. Bundling and deployment
# behavior can vary between application types.
project_type: python
```

The `project_type` directive simply indicates that this is a Python project. This is so `zpm` understands what to do exactly for things like application bundling, which are specific to the project type.

```
# This section describes the runtime behavior of your zapp: which
# groups of nodes to create and which nexe to invoke for each.
execution:

  # Your application can consist of multiple groups. This is typically
  # used for map-reduce style jobs. This is a list of groups, so
  # remember to add "-" infront of each group name.
  groups:

      # Name of this group. This is used if you need to connect groups
      # with each other.
    - name: ""

      # The NaCl executable (nexe) to run on the nodes in this group.
      path: file://python2.7:python

      # Command line arguments for the nexe.
      args: ""

      # Input and output devices for this group.
      devices:
      - name: python2.7
      - name: stdout
```

The `execution` section can contain one or more `groups`. We call them "groups" because certain configurations can result in the creation of many ZeroVM instances, as is the case with *MapReduce applications on ZeroCloud*. Each group must define a `name` which is unique among all of the groups.

`path` defines the base image to use for execution. The format of this field is as follows: `file://<image-name>:<exe-name>` In this example, we indicate that the `python2.7` base image shall be used, and from that, we execute the `python` binary contained within that image.

`args` is used to supply additional arguments to the `python` executable. In most cases, we will simply invoke a Python script by settings args to something like `foo.py`, but you can supply additional positional arguments as well, just as if you were running a Python script from a command line (`python foo.py arg1 arg1 etc.`).

`devices` defines which I/O devices are to be made available to the ZeroVM instances in this group, and how the devices should be configured. See I/O Devices for more detail.

```
# Meta-information about your zapp.
meta:
  Version: ""
  name: "myapp"
  Author-email: ""
  Summary: ""
```

The `meta` section simply contains metadata about the application, and should be pretty self-explanatory. The only

required property here is `name`, which is used to contstruct the `.zapp` file when `zpm bundle` is called. For example, if the `name` is `foo`, then `zpm bundle` will bundle the application as `foo.zapp`.

```
help:
  # Short description of your zapp. This is used for auto-generated
  # help.
  description: ""

  # Help for the command line arguments. Each entry is a two-tuple
  # with an option name and an option help text.
  args: []
```

The `help` section is deprecated. You can ignore it for now.

```
# Files to include in your zapp. Your can use glob patterns here, they
# will be resolved relative to the location of this file.
bundling: []
```

The `bundling` section defines which files/directories within the project directory should be included in the zapp at bundle time. You can include individual files in this way, or entire directories.

3.4.2 Including third party (pure) Python dependencies

`zapp.yaml` has an optional directive called `dependencies`. In this section you can list third party Python dependencies, which will be fetched from PyPI. Note that third party Python code must be *pure* Python. Here are a few examples:

```
dependencies: [
    "pngcanvas",
]
```

In this example, we declare the pngcanvas library as a dependency. This is the simplest and most typical example.

Here is a more complicated example:

```
dependencies: [
    "pngcanvas",
    ["glibc", "glibc", "pyglibc"],
    ["purepng", "png"],
]
```

In this example, we declare pngcanvas, glibc, and purepng as dependencies.

Because `glibc` installs both a `glibc.py` module and a `pyglibc` package, we specify both of those in the tail of the list.

Similarly, we also want to include purepng. The difference here is that while the package name on PyPI is `purepng`, the only Python module installed is simply called `png`.

If you don't know which modules/packages to include from a given Python package, you can either look at the `setup.py` (a `glibc` example: https://github.com/zyga/python-glibc/blob/1097a1e5d1e243f08a4872fdb0f088c3c019bc12/setup.py#L35-36) or have a look at the `.zapp/.zapp/venv/lib/python2.7/site-packages` directory after you run `zpm bundle` (which will install and cache the dependencies you specify). This may take some trial and error and multiple `zpm bundle --refresh-deps` commands (see *below*). Fortunately, you won't need to do this often.

Note: The dependency management feature of `zpm` could be the target of future improvement. The initial implementation works for a lot of cases, but may be inefficient for more complex corner cases and varied Python packaging

configurations. If you run into a case which doesn't work, or otherwise have problems with or questions about this feature, please file a bug report.

Refreshing dependencies

Dependencies are cached in the `.zapp/` directory so that `zpm` doesn't redundantly re-fetch dependencies each time you call `zpm bundle`. However, at times you will need to add/remove dependencies, and therefore refresh the cached Python packages. To do this, you can simply run:

```
$ zpm bundle --refresh-deps
```

This will clear the cache, re-fetch all dependencies per the `dependencies` directive in the `zapp.yaml`, and bundle your zapp as usual.

Tip: To double-check if a change in dependencies is reflected correctly in your zapp, you can use `tar tf <myapp>.zapp` to check the contents of the archive.

Exceptional cases

Some Python packages on PyPI may specify an external download location. The BitVector library is one such example. This can cause problems when `zpm bundle` is called. Here is an excerpt of one such error:

```
Downloading/unpacking BitVector (from -r /home/user1/projects/foo/.zapp/deps.txt (line 1))
  Could not find any downloads that satisfy the requirement BitVector (from -r /home/user1/projects/
  Some externally hosted files were ignored (use --allow-external BitVector to allow).
```

To workaround this, modify your `.zapp/tox.ini` and add a custom `install_command` to the `[testenv:venv]` section:

```
[tox]
toxworkdir={toxinidir}/.zapp
envlist = venv
skipsdist = true

[testenv:venv]
deps = -r{toxinidir}/deps.txt
install_command = pip install
                  --allow-external BitVector
                  {opts} {packages}
```

Similar errors can occur if the external source is unverified, which can result in errors like the following. (Indeed, adding the `--allow-external BitVector` is not enough to successfully install this specific dependency.)

```
Downloading/unpacking BitVector (from -r /home/user1/projects/foo/.zapp/deps.txt (line 1))
  Could not find any downloads that satisfy the requirement BitVector (from -r /home/user1/projects/
  Some insecure and unverifiable files were ignored (use --allow-unverified BitVector to allow).
```

In this case, the final step for this workaround is to further edit the `.zapp/tox.ini` and adding the suggested `--allow-unverified` option to the `pip install` command:

```
[tox]
toxworkdir={toxinidir}/.zapp
envlist = venv
skipsdist = true
```

```
[testenv:venv]
deps = -r{toxinidir}/deps.txt
install_command = pip install
                  --allow-external BitVector
                  --allow-unverified BitVector
                  {opts} {packages}
```

After making these changes, run `zpm bundle -r` and everything should work correctly.

3.5 Job Chaining

Applications executing on ZeroCloud have the potential to dynamically start new jobs, allowing for arbitrary sequencing or "chaining" of programs. Because each execution instance can only attach to a certain number of devices (due to the data-local semantics of ZeroCloud computation), this allows your program to read/write any number of various objects.

This tutorial will illustrate the basics of this feature and how to use it in your application. In this example, we'll build an application with two scripts: `script1.py` and `script2.py`. `script1.py` will be the entry point of the application, which will chain-call a second job execution to run `script2.py`.

You can chain-call as many jobs as you want in this manner, until the `chain_timeout` is reached. See ZeroCloud job chaining configuration for more information.

For a more complex application which uses this feature, have a look at *Example Application Tutorial: Snakebin*.

Jump to a section:

3.5.1 Environment Setup

See *Setting up a development environment* before continuing. If you've already done this, feel free to jump ahead to the next section.

3.5.2 Create an application template

To start building our application, we first need to create a `zapp.yaml` application template. `zpm` can do this for us:

```
$ zpm new
Created './zapp.yaml'
```

Open `zapp.yaml` in your favorite text editor and modify the `execution` section, which looks something like this:

```
execution:
  groups:
    - name: ""
      path: file://python2.7:python
```

```
      args: ""
      devices:
      - name: python2.7
      - name: stdout
```

Edit the `execution` section and define an execution group name and arguments. We also need to modify the configuration of the `stdout` device to enable job chaining. For example:

```
execution:
  groups:
    - name: "job-chain-test"
      path: file://python2.7:python
      args: "script1.py"
      devices:
      - name: python2.7
      - name: stdout
        content_type: message/http
```

The execution group name is just an arbitrary name. `args` needs to be at least the name of a Python script to execute and can also include any positional arguments. For the `stdout` device, we must add the content type to enable special behavior for any content which is written to it. `message/http` indicates to the ZeroCloud middleware that the content can either be interpreted as a new job request, or it can simply be a response to the client. More on that later.

You will also need to define an application name in the `meta` section. For simplicity, let's give the application the same name as the execution group:

```
meta:
  Version: ""
  name: "job-chain-test"
  Author-email: ""
  Summary: ""
```

Finally, we'll need to include some code in the application. We'll add *the code* later, but for now we just need to tell our `zapp.yaml` application config to include those source files when bundling. Simply modify the `bundling` section to include our script file names:

```
bundling: ["script1.py", "script2.py"]
```

3.5.3 The code

Now that we've got our basic app configuration done, let's dig into the code.

Create a file called `script1.py` in the same directory as `zapp.yaml` and add the following code:

```
import json
import sys

job = json.dumps([{
    "name": "script2",
    "exec": {
        "path": "file://python2.7:python",
        "args": "script2.py"
    },
    "devices": [
        {"name": "python2.7"},
        {"name": "stdout"},
        {"name": "image",
```

```
            "path": "swift://~/chain/job-chain-test.zapp"},
    ],
}])

http_response = """\
HTTP/1.1 200 OK\r
Content-Type: application/json\r
Content-Length: %(content_len)s\r
X-Zerovm-Execute: 1.0
\r
%(content)s"""

sys.stdout.write(http_response % dict(content=job, content_len=len(job)))
```

There are a couple of important things to highlight here. In order for ZeroCloud to interpret the `sys.stdout.write` call as a job request:

- The status code and status reason don't too matter too much here. `200 OK` is a good default, but the behavior is no different if you specify, for example, `404 Not Found`.
- `Content-Type` *must* be `application/json`
- `X-Zerovm-Execute` *must* be set to `1.0`; this indicates to ZeroCloud that this is not just a normal HTTP response, but a special ZeroVM execution request.

Note: The HTTP specification requires status line and header fields to end with a carriage return + line feed (\r\n). The \n newline characters are implicit in multi-line string above, but the \r carriage must be explicitly added. If you omit the \r most clients probably won't complain, but it's best to follow the specification.

If `X-Zerovm-Execute` is omitted, this HTTP response would simply be sent back to the client. This is the kind of response we'll be sending in `script2.py`:

```
import json
import sys

resp = json.dumps({"reply": "This is from script2.py"})

http_response = """\
HTTP/1.1 200 OK\r
Content-Type: application/json\r
Content-Length: %(content_len)s\r
\r
%(content)s"""

sys.stdout.write(http_response % dict(content=resp, content_len=len(resp)))
```

A couple of things to highlight here:

- When writing a response intended for the client, you can use any `Content-Type` you like; it doesn't have to be `application/json`. It can be `text/plain`, `text/html`, `image/png`, etc.
- In fact, it doesn't even need to be properly structured HTTP text. For simple cases, you can simply just `print` text and it will get wrapped up in a proper HTTP response by ZeroCloud before sending it to the client. (It's just that writing proper HTTP yourself means your can return different statuses in different cases, like `404 Not Found`, `500 Internal Server Error`, etc.

3.5.4 Deploying the application

Time to bundle an deploy the application. First, bundle:

```
$ zpm bundle
created job-chain-test.zapp
```

For this example, we'll deploy the application to a container called `chain`. You can create this container first if you like (using `swift post chain`), or you can just let `zpm deploy` do it for you automatically.

```
$ zpm deploy chain job-chain-test.zapp
```

3.5.5 Running the application

The easiest way to run the application is to send an HTTP request to ZeroCloud using `curl`:

```
$ curl -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/chain/job-chain-test.zapp" -
```

The output should look like this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 36

{"reply": "This is from script2.py"}
```

Let's take a look at what's going on in this request.

- `curl -X POST`: ZeroVM application execution requests are expected to use the `POST` method.
- `-H "X-Zerovm-Execute: 1.0"`: This indicates to ZeroCloud that the `POST` request should be interpreted as ZeroVM execution.
- `-H "X-Zerovm-Source: swift://~/chain/job-chain-test.zapp"`: This indicates the application for ZeroCloud to execute. The `job-chain-test.zapp` contains all the other information and code necessary to execute. The `~` in the `swift://` path is an alias for your account ID. (The account ID is the `$OS_STORAGE_ACCOUNT` environment variable. See *Setting up a development environment*.)
- `-H "X-Auth-Token: $OS_AUTH_TOKEN"`: This is simply an auth token which Swift/ZeroCloud requires for us to access services. If you omit this, Swift will respond with a `403 Unauthorized`.
- `$OS_STORAGE_URL`: This is simply the destination for the `POST` request.

3.5.6 Passing data through the call chain

If you want to pass data directly from one job to the next job in the call chain, you can set environment variables in the job description. To illustrate this, let's modify `script1.py` and `script2.py`.

In `script1.py`, we want to define some environment variables (`myvar` and `FOO`) to be set when `script2.py` executes:

```
import json
import sys

job = json.dumps([{
    "name": "script2",
    "exec": {
        "path": "file://python2.7:python",
```

```
        "args": "script2.py",
        "env": {
            "FOO": "bar",
            "myvar": "12345",
        },
    },
    "devices": [
        {"name": "python2.7"},
        {"name": "stdout"},
        {"name": "image",
         "path": "swift://~/chain/job-chain-test.zapp"},
    ],
}])

http_response = """\
HTTP/1.1 200 OK\r
Content-Type: application/json\r
Content-Length: %(content_len)s\r
X-Zerovm-Execute: 1.0
\r
%(content)s"""

sys.stdout.write(http_response % dict(content=job, content_len=len(job)))
```

In `script2.py`, let's read those variables from the environment and include them in the client response:

```
import json
import os
import sys

resp_dict = {"reply": "This is from script2.py"}
resp_dict["myvar"] = os.environ.get("myvar")
resp_dict["FOO"] = os.environ.get("FOO")
resp = json.dumps(resp_dict)

http_response = """\
HTTP/1.1 200 OK\r
Content-Type: application/json\r
Content-Length: %(content_len)s\r
\r
%(content)s"""

sys.stdout.write(http_response % dict(content=resp, content_len=len(resp)))
```

To test this, first we need to re-bundle:

```
$ zpm bundle
```

Then re-deploy:

```
$ zpm deploy chain job-chain-test.zapp --force
```

Note: We need to specify `--force` here since we're overwriting the previously deployed object.

To test the application, we can use the same `curl` command as before:

```
$ curl -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/chain/job-chain-test.zapp" -
```

The output should look something like this:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 68

{"myvar": "12345", "reply": "This is from script2.py", "FOO": "bar"}
```

3.6 Logging

Most often, the types of problems you will need to troubleshoot have to do specifically with your application code, and not the execution environment itself. The ZeroCloud execution environment tends to be less verbose about errors, in order to not leak potentially sensitive information to a client. Showing stacktraces, for example, is not always a good idea.

The lack of verbosity is there for a reason, but this of course can make troubleshooting and debugging difficult. This tutorial aims to illustrate what types of errors are often encountered when developing applications for ZeroCloud and what you can do catch and log these errors to aid you in debugging.

Jump to a section:

3.6.1 Environment Setup

See *Setting up a development environment* before continuing. If you've already done this, feel free to jump ahead to the next section.

3.6.2 The `stderr` device

The best way to log information or errors in your application is to use the `stderr` device. We'll see how to do that by building a very simple "hello, world"zapp. Just by adding `stderr` to the application config, any crashes in the application code due to uncaught exceptions, or syntax errors will be logged. We can also explicitly log information to `stderr` by writing to the `sys.stderr` device or to the file `/dev/stderr`. In this guide, we'll go through some examples to cover all of these cases.

To start with, create an application template with `zpm`:

```
$ zpm new
Created './zapp.yaml'
```

Open `zapp.yaml` in your favorite text editor. Modify the `meta` section, and set `logtest` for the name:

```
meta:
  Version: ""
  name: "logtest"
  Author-email: ""
  Summary: ""
```

Also modify the `execution` section, which by default looks something like this:

```
execution:
  groups:
    - name: ""
      path: file://python2.7:python
      args: ""
      devices:
      - name: python2.7
      - name: stdout
```

Edit the `execution` section and give the execution a name, args, and a `stderr` device. For example:

```
 execution:
   groups:
     - name: "logtest"
       path: file://python2.7:python
       args: "logtest.py"
       devices:
       - name: python2.7
       - name: stdout
       - name: stderr
         path: swift://~/logs/logtest.log
```

The group name can be anything you like. For this example, we just call it "logtest". The args should at least be the name of a Python script. We will write that script below. For this example, we will just call it `logtest.py`.

The `stderr` device is one of the special built-in devices that any ZeroCloud application can use. We need to specify a Swift object path to write to in order to capture error output. In this example, we've chosen `logtest.log` in the `logs` container.

Finally, modify the `bundling` the section to include `logtest.py`:

```
bundling: ["logtest.py"]
```

3.6.3 Logging syntax errors

Let's make our script into a basic "hello, world" application which has some syntax errors. Create the file `logtest.py` and add the following code:

```
print "hello, world"
foo = [1, 2, 3
print foo
```

The syntax error is the missing closing bracket on line 2.

Before we deploy, we'll need to create the `log` container:

```
$ swift post logs
```

Now let's bundle, deploy, and run the application to see what happens:

```
 $ zpm bundle
 created logtest.zapp
 $ zpm deploy logtest logtest.zapp
 $ curl -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/logtest/logtest.zapp" -H "X
 (no output)
```

Notice that there is no output. Adding the `-i/--include` flag in the `curl` command will show us the HTTP response headers and give us a clue about what went wrong.

```
$ curl -i -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/logtest/logtest.zapp" -H
```

The output should look something like this:

```
HTTP/1.1 200 OK
X-Nexe-Retcode: 1
X-Nexe-System: logtest
X-Nexe-Cdr-Line: 2.289, 2.260, 0.04 2.00 1021 66644156 34 88 0 0 0 0
X-Zerovm-Device: stdout
X-Nexe-Policy: Policy-0
X-Nexe-Validation: 2
Content-Length: 0
X-Nexe-Etag: /dev/stdout d41d8cd98f00b204e9800998ecf8427e
Connection: close
Etag: 433f8ee1c8da2c16bb625e87a1e89e7f
X-Timestamp: 1415976829.81199
X-Nexe-Status: ok
Date: Fri, 14 Nov 2014 14:53:49 GMT
Content-Type: text/html
X-Chain-Total-Time: 2.289
X-Trans-Id: txac6be961d72141ba9dbdf-005466177b
```

The most important line here is highlighted: `X-Nexe-Retcode: 1`. This tells us that our *application* (not the execution environment; this is an important distinction) exited with a status code of 1. In other words, there was probably a crash.

Let's download the log and see what it says:

```
$ swift download logs logtest.log
logtest.log [auth 0.030s, headers 0.038s, total 0.038s, 0.011 MB/s]
$ cat logtest.log
  File "logtest.py", line 3
    print foo
        ^
SyntaxError: invalid syntax
```

This is the same error output we would expect if we were to run `logtest.py` on local host:

```
$ python logtest.py
  File "logtest.py", line 3
    print foo
        ^
SyntaxError: invalid syntax
```

Fix the syntax error in `logtest.py`:

```
print "hello, world"
foo = [1, 2, 3]
print foo
```

Then re-bundle, re-deploy, and run:

```
$ zpm bundle
Created logtest.zapp
$ zpm deploy logtest logtest.zapp --force
app deployed to
  http://127.0.0.1:8080/v1/AUTH_123def/logtest/
$ curl -i -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/logtest/logtest.zapp" -H
HTTP/1.1 200 OK
X-Nexe-Retcode: 0
```

```
X-Nexe-System: logtest
X-Nexe-Cdr-Line: 2.246, 2.229, 0.03 1.98 1021 66644156 11 23 0 0 0 0
X-Zerovm-Device: stdout
X-Nexe-Policy: Policy-0
X-Nexe-Validation: 2
Content-Length: 23
X-Nexe-Etag: /dev/stdout c1def45af975364b4ee99d550d1d98da
Connection: close
Etag: d4c7b38bf909b5bc74eb81e73a5da81a
X-Timestamp: 1415977445.97419
X-Nexe-Status: ok
Date: Fri, 14 Nov 2014 15:04:05 GMT
Content-Type: text/html
X-Chain-Total-Time: 2.246
X-Trans-Id: tx40647fee812944ef80a99-00546619e3

hello, world
[1, 2, 3]
```

Note that the `X-Nexe-Retcode` is now `0`, and we get the output we expect.

3.6.4 Logging exceptions

Let's try raising an exception to see what happens. Add one more line of code to `logtest.py`:

```
print "hello, world"
foo = [1, 2, 3]
print foo
raise Exception("test exception")
```

Re-bundle, re-deploy, and run:

```
$ zpm bundle
Created logtest.zapp
$ zpm deploy logtest logtest.zapp --force
app deployed to
  http://127.0.0.1:8080/v1/AUTH_123def/logtest/
$ curl -i -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/logtest/logtest.zapp" -H
HTTP/1.1 200 OK
X-Nexe-Retcode: 1
X-Nexe-System: logtest
X-Nexe-Cdr-Line: 2.230, 2.211, 0.05 1.95 1021 66644156 19 163 0 0 0 0
X-Zerovm-Device: stdout
X-Nexe-Policy: Policy-0
X-Nexe-Validation: 2
Content-Length: 23
X-Nexe-Etag: /dev/stdout c1def45af975364b4ee99d550d1d98da
Connection: close
Etag: a3afd778d9eafbf4d4429208d85aa35a
X-Timestamp: 1415977837.42819
X-Nexe-Status: ok
Date: Fri, 14 Nov 2014 15:10:37 GMT
Content-Type: text/html
X-Chain-Total-Time: 2.230
X-Trans-Id: txac87f5e842f04439b540c-0054661b6b

hello, world
[1, 2, 3]
```

This time we get the same output as above, but the `X-Nexe-Retcode` is `1`. Let's grab the log again to see what it says:

```
$ swift download logs logtest.log
logtest.log [auth 0.043s, headers 0.059s, total 0.060s, 0.009 MB/s]
$ cat logtest.log
Traceback (most recent call last):
  File "logtest.py", line 4, in <module>
    raise Exception("test exception")
Exception: test exception
```

Similar to syntax errors, any uncaught/unhandled exceptions will be logged to `stderr`.

3.6.5 Explicit logging

Let's look at a case where we want to explicitly log something to `stderr`. We'll keep building on the code we have already in `logtest.py`:

```
import sys

try:
    print "hello, world"
    foo = [1, 2, 3]
    print foo
    raise Exception("test exception")
except Exception:
    sys.stderr.write("Something bad happened\n")
```

Here we are catching the exception and explictly logging a message instead of letting the exception get logged (as we experienced above).

Let's test this to see what happens:

```
$ zpm bundle
Created logtest.zapp
$ zpm deploy logtest logtest.zapp --force
app deployed to
  http://127.0.0.1:8080/v1/AUTH_123def/logtest/
$ curl -i -X POST -H "X-Zerovm-Execute: 1.0" -H "X-Zerovm-Source: swift://~/logtest/logtest.zapp" -H
HTTP/1.1 200 OK
X-Nexe-Retcode: 0
X-Nexe-System: logtest
X-Nexe-Cdr-Line: 2.258, 2.239, 0.07 1.94 1021 66644156 12 45 0 0 0 0
X-Zerovm-Device: stdout
X-Nexe-Policy: Policy-0
X-Nexe-Validation: 2
Content-Length: 23
X-Nexe-Etag: /dev/stdout c1def45af975364b4ee99d550d1d98da
Connection: close
Etag: 83c1b9378a58e72a87f3cc82af081b67
X-Timestamp: 1415980070.92201
X-Nexe-Status: ok
Date: Fri, 14 Nov 2014 15:47:50 GMT
Content-Type: text/html
X-Chain-Total-Time: 2.258
X-Trans-Id: tx89ceb1b12459438ab9e8c-0054662424
```

```
hello, world
[1, 2, 3]
```

Note that the `X-Nexe-Retcode` is `0`, meaning the application exited gracefully.

Let's have a log at the log:

```
$ swift download logs logtest.log
logtest.log [auth 0.047s, headers 0.063s, total 0.065s, 0.001 MB/s]
$ cat logtest.log
Something bad happened
```

As an alternative to using the `sys.stderr` device, you can write logging output to the file `/dev/stderr`. The following code will behave the exact same way as above:

```
try:
    print "hello, world"
    foo = [1, 2, 3]
    print foo
    raise Exception("test exception")
except Exception:
    with open("/dev/stderr", "a") as stderr:
        stderr.write("Something bad happened\n")
```

If you do it this way, note that you need to open the file with "append" mode (*"a"*) specified.

3.7 Example Application Tutorial: Snakebin

Snakebin is a combination of Pastebin and JSFiddle, for Python. It allows a user to create and store Python scripts in ZeroCloud, retrieve them using a unique URL, and execute them through a web interface. Essentially this is a copy of the Go Playground, but for Python.

In this tutorial, we will be building the entire application from scratch and deploying it to ZeroCloud. The result will be a web application, complete with a REST API and a basic UI written in HTML and JavaScript. The entire backend for the REST API will be implemented in Python.

3.7.1 Overview

Jump to a section:

- *Part 1: Upload/Download Scripts*
- *Part 2: Execute Scripts*
- *Part 3: Search Scripts*

We will build the application in three parts. In the *first part*, we will implement a REST API for uploading and downloading Python scripts to/from ZeroCloud. We will also implment a basic UI to interact with the REST interface in HTML and JavaScript.

In the *second part*, we will add execution functionality to the API, as well as a "Run" button to the UI to execute code. The secure isolation of ZeroVM will ensure that any arbitrary code can run safely.

In the *third and final part*, we will implement a parallelized MapReduce-style search function for searching all existing documents in Snakebin. The search function will be driven by yet another addition to the API and will include a "Search" field in the UI.

3.7.2 Setup

The first thing you'll need to do is set up a *development environment*, including the `python-swiftclient` and `zpm` command line tools.

Next, you should create a working directory on your local machine. In this tutorial, we will put all project files in a directory called `snakebin` inside the home directory. Change to this directory as well.

```
$ mkdir $HOME/snakebin
$ cd $HOME/snakebin
```

3.7.3 Swift Container Setup

To deploy and run the application, we'll need three containers:

- `snakebin-api`: This will serve as the base URL for REST API requests. This container will only contain the HTML / JavaScript UI files.
- `snakebin-app`: This will contain all of the application files, except for the UI files.
- `snakebin-store`: This will serve as our document storage location. No direct access will be allowed; all documents must be accessed through the REST API.

Go ahead and create these containers now. You can do this using the `swift` command line tool:

```
$ swift post snakebin-api
$ swift post snakebin-app
$ swift post snakebin-store
```

Double-check that the containers were created:

```
$ swift list
snakebin-api
snakebin-app
snakebin-store
```

3.7.4 Add `zapp.yaml`

The next thing we need to do is add the basic configuration file which defines a ZeroVM application (or "zapp"). `zpm` can do this for us:

```
$ zpm new --template python
```

This will create a `zapp.yaml` file in the current directory. Open `zapp.yaml` in your favorite text editor. Change the `execution` section

```
execution:
  groups:
    - name: ""
      path: file://python2.7:python
      args: ""
      devices:
      - name: python2.7
      - name: stdout
```

to look like this:

```
execution:
  groups:
    - name: "snakebin"
      path: file://python2.7:python
      args: "snakebin.py"
      devices:
      - name: python2.7
      - name: stdout
        content_type: message/http
      - name: stdin
      - name: input
        path: swift://~/snakebin-store
```

Edit the `bundling` section

```
bundling: []
```

to include the source files for our application (which we will be creating below):

```
bundling: ["snakebin.py", "save_file.py", "get_file.py", "index.html"]
```

Finally, we need to specify third-party dependencies so that `zpm` knows how to bundle our application:

```
dependencies: [
    "falcon",
    "six",
    "mimeparse",
]
```

The final result should look like this:

```
project_type: python

execution:
  groups:
    - name: "snakebin"
      path: file://python2.7:python
      args: "snakebin.py"
      devices:
      - name: python2.7
      - name: stdout
        content_type: message/http
      - name: stdin
      - name: input
        path: swift://~/snakebin-store

meta:
  Version: ""
  name: "snakebin"
  Author-email: ""
  Summary: ""

help:
  description: ""
  args: []

bundling: ["snakebin.py", "save_file.py", "get_file.py", "index.html"]

dependencies: [
```

```
    "falcon",
    "six",
    "mimeparse",
]
```

3.7.5 Part 1: Upload/Download Scripts

First, we need to build an application for uploading and retrieving scripts, complete with a basic HTML user interface.

Before we dig into the application code, let's think about our API design.

REST API

For the time being, we only need to support a few different types of requests:

`GET /snakebin-api:` Get an empty HTML form for uploading a script.

`POST /snakebin-api:` Post file contents, get a `/snakebin-api/:script` URL back.

`GET /snakebin-api/:script:` Retrieve uploaded file contents.

> If a request specifies the header `Accept: text/html` (as is the case with a web browser), load the HTML UI page with the script textarea populated. For any other `Accept` value, just return the raw script contents.

The Code

ZeroCloud provides a CGI-like environment for handling HTTP requests. A lot of what follows involves setting and reading environment variables and generating HTTP responses from scratch.

http_resp

Since generating HTTP responses is the most crucial part of this application, let's first define utility function for creating these responses. In your `snakebin` working directory, create a file called `snakebin.py`. Then add the following code to it:

```
def http_resp(code, reason, content_type='message/http', msg='',
              extra_headers=None):
    if extra_headers is None:
        extra_header_text = ''
    else:
        extra_header_text = '\r\n'.join(
            ['%s: %s' % (k, v) for k, v in extra_headers.items()]
        )
        extra_header_text += '\r\n'

    resp = """\
HTTP/1.1 %(code)s %(reason)s\r
%(extra_headers)sContent-Type: %(content_type)s\r
Content-Length: %(msg_len)s\r
\r
%(msg)s"""
    resp %= dict(code=code, reason=reason, content_type=content_type,
                 msg_len=len(msg), msg=msg, extra_headers=extra_header_text)
    sys.stdout.write(resp)
```

Notice the last line, which is highlighted: `sys.stdout.write(resp)`.

The ZeroCloud execution environment handles most communication between parts of an application through `/dev/stdout`, by convention. To your application code (which is running inside the ZeroVM virtual execution environment), `/dev/stdout` looks just like the character device you would expect in a Linux-like execution environment, but to ZeroCloud, you can write to this device to either communicate to a client or start a new "*job*", all using HTTP. (In this tutorial, we'll be doing both.)

For `http_resp`, we'll need to import `sys` from the standard library. Add an `import` statement to the top of the file:

```
import sys
```

Job

A "*job*" is defined by a collection of JSON objects which specify commands to execute, environment variables to set (for the execution environment), and device mappings. ZeroCloud consumes job descriptions to start new jobs, which can consist of one or more program execution groups. For the moment, we'll only be dealing with single program jobs. (In *part three*, we'll need to define some multi-group jobs to implement the MapReduce search function. But don't worry about that for now.)

Tip: For complete details on structure and options for a job description, check out the full documentation.

Let's create a class which will help us generate these jobs. Add the class below to `snakebin.py`. For simplicity, some Swift object/container names are hard-coded.

```
class Job(object):

    def __init__(self, name, args):
        self.name = name
        self.args = args
        self.devices = [
            {'name': 'python2.7'},
            {'name': 'stdout', 'content_type': 'message/http'},
            {'name': 'image', 'path': 'swift://~/snakebin-app/snakebin.zapp'},
        ]
        self.env = {}

    def add_device(self, name, content_type=None, path=None):
        dev = {'name': name}
        if content_type is not None:
            dev['content_type'] = content_type
        if path is not None:
            dev['path'] = path
        self.devices.append(dev)

    def set_envvar(self, key, value):
        self.env[key] = value

    def to_json(self):
        return json.dumps([self.to_dict()])

    def to_dict(self):
        return {
            'name': self.name,
            'exec': {
                'path': 'file://python2.7:python',
```

```
                'args': self.args,
                'env': self.env,
            },
            'devices': self.devices,
        }
```

This class makes use of the `json` module, so lets import that as well:

```
import json
```

GET and POST handling

Now we're getting into the core functionality of our application. It's time to add code to handle the `POST` and `GET` requests in the manner that we've defined in our *API definition* above.

To make things easy for us, we can write this functionality as a WSGI application and use light-weight API framework like Falcon to implement the various endpoint handlers.

We'll need to add a handful of new things to `snakebin.py`:

- a utility function to query container databases to check if an object with a given name already exists
- a utility function to generate a random "short name", using script upload contents as the random seed
- a couple of "handler" classes and some helper functions for dealing with the various types of requests
- a `main` block which sets up the WSGI application and registers the endpoint handlers

Here's what that looks like:

```
def _object_exists(name):
    """Check the local container (mapped to `/dev/input`) to see if it contains
    an object with the given ``name``. /dev/input is expected to be a sqlite
    database.
    """
    conn = sqlite3.connect('/dev/input')
    try:
        cur = conn.cursor()
        sql = 'SELECT ROWID FROM object WHERE name=? AND deleted=0'
        cur.execute(sql, (name, ))
        result = cur.fetchall()
        return len(result) > 0
    finally:
        conn.close()


def random_short_name(seed, length=10):
    rand = random.Random()
    rand.seed(seed)
    name = ''.join(rand.sample(string.ascii_lowercase
                               + string.ascii_uppercase
                               + string.digits, length))
    return name


def _handle_script(req, resp, account, container, script):
    # Go get the requested script, or 404 if it doesn't exist.
    if _object_exists(script):
        private_file_path = 'swift://~/snakebin-store/%s' % script
```

```
        job = Job('snakebin-get-file', 'get_file.py')
        job.add_device('input', path=private_file_path)
        job.set_envvar('HTTP_ACCEPT', os.environ.get('HTTP_ACCEPT'))
        # Setting this header and content_type will make ZeroCloud
        # intercept the request and spawn a new job, instead of responding
        # directly to the client.
        resp.set_header('X-Zerovm-Execute', '1.0')
        resp.content_type = 'application/json'
        resp.status = falcon.HTTP_200
        resp.body = job.to_json()
    else:
        resp.status = falcon.HTTP_404


def _handle_script_upload(req, resp, account, container, script=None):
    file_data = req.stream.read()
    file_hash = hashlib.sha1(file_data)
    short_name = random_short_name(file_hash.hexdigest())

    snakebin_file_path = 'swift://~/snakebin-store/%s' % short_name
    public_file_path = 'swift://~/snakebin-api/%s' % short_name

    if _object_exists(short_name):
        # This means the file already exists. No problem!
        # Since the short url is derived from the hash of the contents,
        # just return a URL to the file.
        path = '/api/%s/%s/%s' % (account, container, short_name)

        file_url = urlparse.urlunparse((
            'http',
            os.environ.get('HTTP_HOST'),
            path,
            None,
            None,
            None
        )) + '\n'
        resp.status = falcon.HTTP_200
        resp.body = file_url
    else:
        # Go and save the file.
        # We need to spawn another ZeroVM job to write this file.
        job = Job('snakebin-save-file', 'save_file.py')
        job.set_envvar('SNAKEBIN_POST_CONTENTS',
                       base64.b64encode(file_data))
        job.set_envvar('SNAKEBIN_PUBLIC_FILE_PATH', public_file_path)
        job.add_device('output', path=snakebin_file_path,
                       content_type='text/plain')
        # Setting this header and content_type will make ZeroCloud
        # intercept the request and spawn a new job, instead of responding
        # directly to the client.
        resp.set_header('X-Zerovm-Execute', '1.0')
        resp.content_type = 'application/json'
        resp.status = falcon.HTTP_200
        resp.body = job.to_json()


class RootHandler(object):
```

```
    def on_get(self, req, resp, account, container):
        """Serve a blank index.html page."""
        with open('index.html') as fp:
            resp.body = fp.read()
        resp.content_type = 'text/html; charset=utf-8'
        resp.status = falcon.HTTP_200

    def on_post(self, req, resp, account, container):
        """Handle the form post/script upload."""
        _handle_script_upload(req, resp, account, container)


class ScriptHandler(object):

    def on_get(self, req, resp, account, container, script):
        _handle_script(req, resp, account, container, script)

    def on_post(self, req, resp, account, container, script):
        # Also allow new/modified scripts to be uploaded when the client is on
        # a page like `/snakebin-api/Wg4re8mXbV`.
        _handle_script_upload(req, resp, account, container, script=script)


if __name__ == '__main__':
    app = falcon.API()
    app.add_route('/{account}/{container}', RootHandler())
    app.add_route('/{account}/{container}/{script}', ScriptHandler())

    handler = wsgiref.handlers.SimpleHandler(
        sys.stdin,
        sys.stdout,
        sys.stderr,
        environ=dict(os.environ),
        multithread=False,
    )
    handler.run(app)
```

This codes makes use of more standard library modules, so we need to add import statements for those, as well as falcon, a third party library.

```
import base64
import hashlib
import json
import os
import random
import sqlite3
import string
import sys
import urlparse
import wsgiref.handlers

import falcon
```

Your `snakebin.py` file should now look something like this:

```
import base64
import hashlib
import json
import os
```

```
import random
import sqlite3
import string
import sys
import urlparse
import wsgiref.handlers

import falcon


def http_resp(code, reason, content_type='message/http', msg='',
              extra_headers=None):
    if extra_headers is None:
        extra_header_text = ''
    else:
        extra_header_text = '\r\n'.join(
            ['%s: %s' % (k, v) for k, v in extra_headers.items()]
        )
        extra_header_text += '\r\n'

    resp = """\
HTTP/1.1 %(code)s %(reason)s\r
%(extra_headers)sContent-Type: %(content_type)s\r
Content-Length: %(msg_len)s\r
\r
%(msg)s"""
    resp %= dict(code=code, reason=reason, content_type=content_type,
                 msg_len=len(msg), msg=msg, extra_headers=extra_header_text)
    sys.stdout.write(resp)


class Job(object):

    def __init__(self, name, args):
        self.name = name
        self.args = args
        self.devices = [
            {'name': 'python2.7'},
            {'name': 'stdout', 'content_type': 'message/http'},
            {'name': 'image', 'path': 'swift://~/snakebin-app/snakebin.zapp'},
        ]
        self.env = {}

    def add_device(self, name, content_type=None, path=None):
        dev = {'name': name}
        if content_type is not None:
            dev['content_type'] = content_type
        if path is not None:
            dev['path'] = path
        self.devices.append(dev)

    def set_envvar(self, key, value):
        self.env[key] = value

    def to_json(self):
        return json.dumps([self.to_dict()])

    def to_dict(self):
```

```
        return {
            'name': self.name,
            'exec': {
                'path': 'file://python2.7:python',
                'args': self.args,
                'env': self.env,
            },
            'devices': self.devices,
        }


def _object_exists(name):
    """Check the local container (mapped to `/dev/input`) to see if it contains
    an object with the given ``name``. /dev/input is expected to be a sqlite
    database.
    """
    conn = sqlite3.connect('/dev/input')
    try:
        cur = conn.cursor()
        sql = 'SELECT ROWID FROM object WHERE name=? AND deleted=0'
        cur.execute(sql, (name, ))
        result = cur.fetchall()
        return len(result) > 0
    finally:
        conn.close()


def random_short_name(seed, length=10):
    rand = random.Random()
    rand.seed(seed)
    name = ''.join(rand.sample(string.ascii_lowercase
                               + string.ascii_uppercase
                               + string.digits, length))
    return name


def _handle_script(req, resp, account, container, script):
    # Go get the requested script, or 404 if it doesn't exist.
    if _object_exists(script):
        private_file_path = 'swift://~/snakebin-store/%s' % script

        job = Job('snakebin-get-file', 'get_file.py')
        job.add_device('input', path=private_file_path)
        job.set_envvar('HTTP_ACCEPT', os.environ.get('HTTP_ACCEPT'))
        # Setting this header and content_type will make ZeroCloud
        # intercept the request and spawn a new job, instead of responding
        # directly to the client.
        resp.set_header('X-Zerovm-Execute', '1.0')
        resp.content_type = 'application/json'
        resp.status = falcon.HTTP_200
        resp.body = job.to_json()
    else:
        resp.status = falcon.HTTP_404


def _handle_script_upload(req, resp, account, container, script=None):
    file_data = req.stream.read()
    file_hash = hashlib.sha1(file_data)
```

```
    short_name = random_short_name(file_hash.hexdigest())

    snakebin_file_path = 'swift://~/snakebin-store/%s' % short_name
    public_file_path = 'swift://~/snakebin-api/%s' % short_name

    if _object_exists(short_name):
        # This means the file already exists. No problem!
        # Since the short url is derived from the hash of the contents,
        # just return a URL to the file.
        path = '/api/%s/%s/%s' % (account, container, short_name)

        file_url = urlparse.urlunparse((
            'http',
            os.environ.get('HTTP_HOST'),
            path,
            None,
            None,
            None
        )) + '\n'
        resp.status = falcon.HTTP_200
        resp.body = file_url
    else:
        # Go and save the file.
        # We need to spawn another ZeroVM job to write this file.
        job = Job('snakebin-save-file', 'save_file.py')
        job.set_envvar('SNAKEBIN_POST_CONTENTS',
                       base64.b64encode(file_data))
        job.set_envvar('SNAKEBIN_PUBLIC_FILE_PATH', public_file_path)
        job.add_device('output', path=snakebin_file_path,
                       content_type='text/plain')
        # Setting this header and content_type will make ZeroCloud
        # intercept the request and spawn a new job, instead of responding
        # directly to the client.
        resp.set_header('X-Zerovm-Execute', '1.0')
        resp.content_type = 'application/json'
        resp.status = falcon.HTTP_200
        resp.body = job.to_json()


class RootHandler(object):

    def on_get(self, req, resp, account, container):
        """Serve a blank index.html page."""
        with open('index.html') as fp:
            resp.body = fp.read()
        resp.content_type = 'text/html; charset=utf-8'
        resp.status = falcon.HTTP_200

    def on_post(self, req, resp, account, container):
        """Handle the form post/script upload."""
        _handle_script_upload(req, resp, account, container)


class ScriptHandler(object):

    def on_get(self, req, resp, account, container, script):
        _handle_script(req, resp, account, container, script)
```

```
    def on_post(self, req, resp, account, container, script):
        # Also allow new/modified scripts to be uploaded when the client is on
        # a page like `/snakebin-api/Wg4re8mXbV`.
        _handle_script_upload(req, resp, account, container, script=script)


if __name__ == '__main__':
    app = falcon.API()
    app.add_route('/{account}/{container}', RootHandler())
    app.add_route('/{account}/{container}/{script}', ScriptHandler())

    handler = wsgiref.handlers.SimpleHandler(
        sys.stdin,
        sys.stdout,
        sys.stderr,
        environ=dict(os.environ),
        multithread=False,
    )
    handler.run(app)
```

get_file.py and save_file.py

In snakebin.py, there are some references to additional source files to handle saving and retrieval of uploaded documents. Let's create those now.

get_file.py:

```
import os
from xml.sax.saxutils import escape

import snakebin


if __name__ == '__main__':
    with open('/dev/input') as fp:
        contents = fp.read()

    http_accept = os.environ.get('HTTP_ACCEPT', '')
    if 'text/html' in http_accept:
        # Something that looks like a browser is requesting the document:
        with open('/index.html') as fp:
            html_page_template = fp.read()
            html_page = html_page_template.replace('{code}', escape(contents))
        snakebin.http_resp(200, 'OK', content_type='text/html; charset=utf-8',
                           msg=html_page)
    else:
        # Some other type of client is requesting the document:
        snakebin.http_resp(200, 'OK', content_type='text/plain', msg=contents)
```

save_file.py:

```
import base64
import os

import snakebin
```

```
def save_file(post_contents, public_file_path):
    script_contents = base64.b64decode(post_contents)

    with open('/dev/output', 'a') as fp:
        fp.write(script_contents)

    _rest, container, short_name = public_file_path.rsplit('/', 2)
    file_url = 'http://%(host)s/api/%(acct)s/%(cont)s/%(short_name)s\n'
    file_url %= dict(host=os.environ.get('HTTP_HOST'), cont=container,
                     acct=os.environ.get('PATH_INFO').strip('/'),
                     short_name=short_name)

    snakebin.http_resp(201, 'Created', msg=file_url)


if __name__ == '__main__':
    post_contents = os.environ.get('SNAKEBIN_POST_CONTENTS')
    public_file_path = os.environ.get('SNAKEBIN_PUBLIC_FILE_PATH')

    save_file(post_contents, public_file_path)
```

User Interface

To complete the first iteration of the Snakebin application, let's create a user interface. Create a file called `index.html` and add the following code to it:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Snakebin</title>
    <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/codemirror.min.css
    <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/codemirror.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/mode/python/python.min.js"></scrip
    <script>
        $(document).ready(function() {

            // Add syntax highlighting for Python code:
            var code = $('#code')[0];
            var editor = CodeMirror.fromTextArea(code, {
                mode: "text/x-python",
                lineNumbers: true
            });

            // Called when a script posting is successful.
            var saveSuccess = function(data, textStatus, jqXHR) {
                var url = jqXHR.responseText;
                $('#save-status').html(
                    'Saved to <a id="save-url" href="' + url
                    + '">' + url + '</a>'
                );
            };

            // Attach save functionality to the "Save" button
            $('#save').click(function() {
                var request = {
                    'url': window.location.href,
```

```
                    'type': 'post',
                    'data': editor.getValue(),
                    'headers': {
                        'X-Zerovm-Execute': 'api/1.0'
                    },
                    'success': saveSuccess
                };
                $.ajax(request);
            });
        });
    </script>
  </head>
  <body>
    <textarea id="code" rows="15" cols="80" wrap="off"
              autocorrect="off" autocomplete="off"
              autocapitalize="off" spellcheck="false">{code}</textarea>
    <p>
      <input id="save" type="submit" value="Save" />
      <div id="save-status"></div>
    </p>
  </body>
</html>
```

Bundle and deploy

Bundle:

```
$ zpm bundle
created snakebin.zapp
```

Deploy:

```
$ zpm deploy snakebin-app snakebin.zapp
app deployed to http://127.0.0.1:8080/v1/AUTH_123def/snakebin-app/
```

Setting an environment variable for the storage account ID will make commands more concise and convenient to execute:

```
$ export OS_STORAGE_ACCOUNT=AUTH_123def...
```

Configure the endpoint handler zapp for `snakebin-api`, `snakebin-app`, and `snakebin-store`:

```
$ swift post --header "X-Container-Meta-Rest-Endpoint: swift://$OS_STORAGE_ACCOUNT/snakebin-app/snake
$ swift post --header "X-Container-Meta-Rest-Endpoint: swift://$OS_STORAGE_ACCOUNT/snakebin-app/snake
$ swift post --header "X-Container-Meta-Rest-Endpoint: swift://$OS_STORAGE_ACCOUNT/snakebin-app/snake
```

We'll also need to set execution permissions for unauthenticated (anonymous) users on the same three containers:

```
$ swift post --header "X-Container-Meta-Zerovm-Suid: .r:*,.rlistings" snakebin-api
$ swift post --header "X-Container-Meta-Zerovm-Suid: .r:*,.rlistings" snakebin-app
$ swift post --header "X-Container-Meta-Zerovm-Suid: .r:*,.rlistings" snakebin-store
```

Test

Now that the first working part of our application is deployed, let's test uploading and retrieving some text.

First, create a file called `example.py`, and add any text to it. For example:

```
print "hello world!"
```

Now upload it:

```
$ curl -X POST -H "X-Zerovm-Execute: api/1.0" $OS_STORAGE_URL/snakebin-api --data-binary @example.py
http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/GDHh7vR3Zb
```

The URL returned from the `POST` can be used to retrieve the document:

```
$ curl http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/GDHh7vR3Zb
print "hello world!"
```

Note: Note that in the `POST` we have to supply the `X-Zerovm-Execute: api/1.0` header because this tells ZeroCloud how to interpret the request. Alternatively, you can change the `/v1/` part of the URL to `/api/` to make requests simpler, and also to accomodate simpler `GET` requests, using `curl` (as is shown above) or a web browser.

We can also try this through the web interface. Open a web browser and go to `http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api`. You should get a page that looks something like this:

Save

Type some text into the box and play around with saving documents. You can also try to browse the the document we created above on the command line (`http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/GDHh7vR3Zb`).

3.7.6 Part 2: Execute Scripts

In this part, we'll add on to what we've built so far and allow Python scripts to be executed by Snakebin.

API updates

To support script execution via HTTP (either from the command line or browser), we will need to add a couple more endpoints to our API:

GET /snakebin-api/:script/execute: Execute the specified `:script` and return the output as text. The script must already exist and be available at `/snakebin-api/:script`.

POST /snakebin-api/execute: Execute the contents of the request as a Python script and return the output as text.

The following changes will implement these two endpoints.

The Code

We need to add a couple of things to support script execution. First, we need to add a utility function to just execute code, and second, we need to update the endpoint handlers to support execution.

First, we need to tweak `_handle_script` to support execution:

```
def _handle_script(req, resp, account, container, script, execute=False):
    # Go get the requested script, or 404 if it doesn't exist.
    if _object_exists(script):
        private_file_path = 'swift://~/snakebin-store/%s' % script

        job = Job('snakebin-get-file', 'get_file.py')
        job.add_device('input', path=private_file_path)
        job.set_envvar('HTTP_ACCEPT', os.environ.get('HTTP_ACCEPT'))
        if execute:
            job.set_envvar('SNAKEBIN_EXECUTE', 'True')
        # Setting this header and content_type will make ZeroCloud
        # intercept the request and spawn a new job, instead of responding
        # directly to the client.
        resp.set_header('X-Zerovm-Execute', '1.0')
        resp.content_type = 'application/json'
        resp.status = falcon.HTTP_200
        resp.body = job.to_json()
    else:
        resp.status = falcon.HTTP_404
```

Next, add an `execute_code` utility function, which actually do the execution:

```
def execute_code(code):
    # Patch stdout, so we can capture output from the submitted code
    old_stdout = sys.stdout
    new_stdout = StringIO.StringIO()
    sys.stdout = new_stdout

    # Create a module with the code
    module = imp.new_module('dontcare')
    module.__name__ = "__main__"

    # Execute the submitted code
    exec code in module.__dict__

    # Read the response from the code
    new_stdout.seek(0)
    output = new_stdout.read()
```

```
    # Unpatch stdout
    sys.stdout = old_stdout

    return output
```

`execute_code` requires the `imp` and `StringIO` standard library modules, so we need to import those:

```
import base64
import hashlib
import imp
import json
import os
import random
import sqlite3
import string
import StringIO
import sys
import urlparse
import wsgiref.handlers

import falcon
```

Next, update the `ScriptHandler` class (to support direct POSTing of scripts for execution):

```
class ScriptHandler(object):

    def on_get(self, req, resp, account, container, script):
        _handle_script(req, resp, account, container, script)

    def on_post(self, req, resp, account, container, script):
        if script == 'execute':
            file_data = req.stream.read()
            resp.content_type = 'text/plain'
            resp.status = falcon.HTTP_200
            resp.body = execute_code(file_data)
        else:
            # Also allow new/modified scripts to be uploaded when the client is
            # on a page like `/snakebin-api/Wg4re8mXbV`.
            _handle_script_upload(req, resp, account, container, script=script)
```

and add a new `ScriptExecuteHandler` class:

```
class ScriptExecuteHandler(object):

    def on_get(self, req, resp, account, container, script):
        _handle_script(req, resp, account, container, script, execute=True)
```

Finally, we need to register the new handler (and add a comment to explain some new behavior for `ScriptHandler`):

```
if __name__ == '__main__':
    app = falcon.API()
    app.add_route('/{account}/{container}', RootHandler())
    # Handles `POST /{account}/{container}/execute` as well
    app.add_route('/{account}/{container}/{script}', ScriptHandler())
    app.add_route('/{account}/{container}/{script}/execute',
                  ScriptExecuteHandler())

    handler = wsgiref.handlers.SimpleHandler(
```

```
        sys.stdin,
        sys.stdout,
        sys.stderr,
        environ=dict(os.environ),
        multithread=False,
    )
    handler.run(app)
```

Now we need to make some modifications to `get_file.py` to allow execution of a script. We need to read the `SNAKEBIN_EXECUTE` environment variable and execute a script if it is present. Update `get_file.py` to this:

```
import os
from xml.sax.saxutils import escape

import snakebin


if __name__ == '__main__':
    with open('/dev/input') as fp:
        contents = fp.read()

    http_accept = os.environ.get('HTTP_ACCEPT', '')
    execute = os.environ.get('SNAKEBIN_EXECUTE', None)
    if 'text/html' in http_accept:
        # Something that looks like a browser is requesting the document:
        if execute is not None:
            output = snakebin.execute_code(contents)
            snakebin.http_resp(200, 'OK',
                               content_type='text/html; charset=utf-8',
                               msg=output)
        else:
            with open('/index.html') as fp:
                html_page_template = fp.read()
                html_page = html_page_template.replace('{code}',
                                                       escape(contents))
            snakebin.http_resp(200, 'OK',
                               content_type='text/html; charset=utf-8',
                               msg=html_page)
    else:
        # Some other type of client is requesting the document:
        output = contents
        if execute is not None:
            output = snakebin.execute_code(contents)
        snakebin.http_resp(200, 'OK', content_type='text/plain', msg=output)
```

We now need to update the UI with a "Run" button to hook in the execution functionality. Update your `index.html` to look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Snakebin</title>
    <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/codemirror.min.cs
    <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/codemirror.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/mode/python/python.min.js"></scrip
    <script>
        $(document).ready(function() {
```

```
        // Add syntax highlighting for Python code:
        var code = $('#code')[0];
        var editor = CodeMirror.fromTextArea(code, {
            mode: "text/x-python",
            lineNumbers: true
        });

        // Called when a script posting is successful.
        var saveSuccess = function(data, textStatus, jqXHR) {
            var url = jqXHR.responseText;
            $('#save-status').html(
                'Saved to <a id="save-url" href="' + url
                + '">' + url + '</a>'
            );
        };

        // Attach save functionality to the "Save" button
        $('#save').click(function() {
            var request = {
                'url': window.location.href,
                'type': 'post',
                'data': editor.getValue(),
                'headers': {
                    'X-Zerovm-Execute': 'api/1.0'
                },
                'success': saveSuccess
            };
            $.ajax(request);
        });

        // Called when a script execution is successful.
        var runSuccess = function(data, textStatus, jqXHR) {
            var statusText = '';
            statusText += textStatus;
            statusText += ', X-Nexe-Retcode: ' + jqXHR.getResponseHeader('X-Nexe-Retcode');
            statusText += ', X-Nexe-Status: ' + jqXHR.getResponseHeader('X-Nexe-Status');
            $('#run-status').text(statusText);
            // Convert newlines to br tags and display execution output
            $('#run-output').html(jqXHR.responseText.replace(/\n/g, '<br />'));
        };

        // Attach run functionality to the "Run" button
        $('#run').click(function() {
            var execUrl = (window.location.href.split('snakebin-api')[0]
                           + 'snakebin-api/execute');
            var request = {
                'url': execUrl,
                'type': 'post',
                'data': editor.getValue(),
                'headers': {
                    'X-Zerovm-Execute': 'api/1.0'
                },
                'success': runSuccess
            };
            $.ajax(request);
        });
    });
</script>
```

```
  </head>
  <body>
    <textarea id="code" rows="15" cols="80" wrap="off"
              autocorrect="off" autocomplete="off"
              autocapitalize="off" spellcheck="false">{code}</textarea>
    <p>
      <input id="save" type="submit" value="Save" />
      <input id="run" type="submit" value="Run" />
      <div id="save-status"></div>
    </p>
    <hr />
    <p>Status:</p>
    <div id="run-status"></div>
    <hr />
    <p>Output:</p>
    <div id="run-output"></div>
  </body>
</html>
```

Redeploy the application

First, rebundle your application files:

```
$ zpm bundle
```

To redeploy, we'll use the same `zpm` command as before, but we'll need to specify the `--force` flag, since we're deploying to an un-empty container:

```
$ zpm deploy snakebin-app snakebin.zapp --force
```

Test

First, let's try executing one of the scripts we already uploaded. This can be done simply by `curl``ing the URL of the script and appending ``/execute`:

```
$ curl http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/GDHh7vR3Zb/execute
hello world!
```

Next, let's trying posting the `example.py` script directly to the `/snakebin-api/execute` endpoint:

```
$ curl -X POST http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/execute --data-binary @exar
hello world!
```

Let's also test the functionality in the web browser. If you nagivate to `http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api`, the new page should look something like this:

```
1 {code}
```

Save Run

Status:

Output:

Try writing some code into the text box and click `Run` to execute them.

Try also accessing the `/snakebin-api/:script/execute` endpoint directly in the browser using the same the URL in the POST example above:

`http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/GDHh7vR3Zb/execute`

3.7.7 Part 3: Search Scripts

The final piece of Snakebin is a simple search search mechanism, which will find document which contain a given search term. All documents in `snakebin-store` will be searched in a parallelized fashion using the MapReduce semantics of ZeroCloud.

API updates

The final endpoint we'll add to our API is `search`:

`GET /snakebin-api/search?q=:term`: Return a JSON list of URLs to the documents (in `snakebin-store`) which contain `:term`. When this endpoint is hit, a MapReduce job of multiple nodes will be spawned to perform the search.

The Code

For the MapReduce job, we need to add two new Python modules.

search_mapper.py

```
import os

with open('/dev/input') as fp:
    contents = fp.read()

search_term = os.environ.get('SNAKEBIN_SEARCH')

if search_term in contents:
    document_name = os.environ.get('LOCAL_PATH_INFO').split('/')[-1]
    doc_url = 'http://%(host)s/api/%(acct)s/%(cont)s/%(short_name)s\n'
    doc_url %= dict(host=os.environ.get('HTTP_HOST'), cont='snakebin-api',
                    acct=os.environ.get('PATH_INFO').strip('/'),
                    short_name=document_name)

    with open('/dev/out/search-reducer', 'a') as fp:
        fp.write(doc_url)
```

search_reducer.py

```
import json
import os

import snakebin

inp_dir = '/dev/in'

results = []
for inp_file in os.listdir(inp_dir):
    with open(os.path.join(inp_dir, inp_file)) as fp:
        result = fp.read().strip()
        if result:
            results.append(result)

snakebin.http_resp(200, 'OK', content_type='application/json',
                   msg=json.dumps(results))
```

These two files will handle the bulk of the search operation.

To kick off the search, we need to make some more changes to `snakebin.py`. First, add a `_handle_search` utility function:

```
def _handle_search(req, resp, account, container):
    query = urllib.unquote(req.params.get('q'))
    mapper_job = Job('search-mapper', 'search_mapper.py')
    mapper_job.add_device('input', path='swift://~/snakebin-store/*')
    mapper_job.set_envvar('SNAKEBIN_SEARCH', query)
    mapper_job.set_envvar('HTTP_ACCEPT',
                          os.environ.get('HTTP_ACCEPT', ''))
    mapper_dict = mapper_job.to_dict()
    mapper_dict['connect'] = ['search-reducer']

    reducer_job = Job('search-reducer', 'search_reducer.py')
    reducer_dict = reducer_job.to_dict()

    map_reduce_job = json.dumps([mapper_dict, reducer_dict])
    resp.body = map_reduce_job
    resp.set_header('X-Zerovm-Execute', '1.0')
    resp.content_type = 'application/json'
```

```
    resp.status = falcon.HTTP_200
    sys.stderr.write('submitting search job')
```

`_handle_search` needs the `urllib` module from the standard library, so we must import it:

```
import base64
import hashlib
import imp
import json
import os
import random
import sqlite3
import string
import StringIO
import sys
import urllib
import urlparse
import wsgiref.handlers

import falcon
```

Finally, we need to make one small tweak to `ScriptHandler` to hook in the search function:

```
class ScriptHandler(object):

    def on_get(self, req, resp, account, container, script):
        if script == 'search':
            _handle_search(req, resp, account, container)
        else:
            _handle_script(req, resp, account, container, script)

    def on_post(self, req, resp, account, container, script):
        if script == 'execute':
            file_data = req.stream.read()
            resp.content_type = 'text/plain'
            resp.status = falcon.HTTP_200
            resp.body = execute_code(file_data)
        else:
            # Also allow new/modified scripts to be uploaded when the client is
            # on a page like `/snakebin-api/Wg4re8mXbV`.
            _handle_script_upload(req, resp, account, container, script=script)
```

Now for the final changes to the user interface:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Snakebin</title>
    <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/codemirror.min.css
    <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/codemirror.min.js"></script>
    <script src="//cdnjs.cloudflare.com/ajax/libs/codemirror/4.6.0/mode/python/python.min.js"></scrip
    <script>
        $(document).ready(function() {

            // Add syntax highlighting for Python code:
            var code = $('#code')[0];
            var editor = CodeMirror.fromTextArea(code, {
                mode: "text/x-python",
```

```
        lineNumbers: true
    });

    // Called when a script posting is successful.
    var saveSuccess = function(data, textStatus, jqXHR) {
        var url = jqXHR.responseText;
        $('#save-status').html(
            'Saved to <a id="save-url" href="' + url
            + '">' + url + '</a>'
        );
    };

    // Attach save functionality to the "Save" button
    $('#save').click(function() {
        var request = {
            'url': window.location.href,
            'type': 'post',
            'data': editor.getValue(),
            'headers': {
                'X-Zerovm-Execute': 'api/1.0'
            },
            'success': saveSuccess
        };
        $.ajax(request);
    });

    // Called when a script execution is successful.
    var runSuccess = function(data, textStatus, jqXHR) {
        var statusText = '';
        statusText += textStatus;
        statusText += ', X-Nexe-Retcode: ' + jqXHR.getResponseHeader('X-Nexe-Retcode');
        statusText += ', X-Nexe-Status: ' + jqXHR.getResponseHeader('X-Nexe-Status');
        $('#run-status').text(statusText);
        // Convert newlines to br tags and display execution output
        $('#run-output').html(jqXHR.responseText.replace(/\n/g, '<br />'));
    };

    // Attach run functionality to the "Run" button
    $('#run').click(function() {
        var execUrl = (window.location.href.split('snakebin-api')[0]
                       + 'snakebin-api/execute');
        var request = {
            'url': execUrl,
            'type': 'post',
            'data': editor.getValue(),
            'headers': {
                'X-Zerovm-Execute': 'api/1.0'
            },
            'success': runSuccess
        };
        $.ajax(request);
    });

    // Call when search is successful.
    var searchSuccess = function(data, textStatus, jqXHR) {
        var urls = JSON.parse(jqXHR.responseText);
        var results = '';
        for (var i = 0; i < urls.length; i++) {
```

```
                    var url = urls[i];
                    results += '<a href="' + url + '">' + url + '</a><br />';
                }
                $('#search-results').html(results);
                $('#search-results').css('visibility', 'visible');
            };

            // Attach search funcionality to the "Search" button
            $('#search').click(function() {
                var searchTerm = encodeURIComponent($('#search-text').val());
                var searchUrl = (window.location.href.split('snakebin-api')[0]
                                 + 'snakebin-api/search?q=' + searchTerm);
                var request = {
                    'url': searchUrl,
                    'type': 'get',
                    'success': searchSuccess
                };
                $.ajax(request);
            });
        });
    </script>
  </head>
  <body>
    <p>
      <input id="search-text" type="input">
      <input id="search" type="submit" value="Search">
      <div id="search-results" style="visibility: hidden;"></div>
    </p>
    <hr />
    <textarea id="code" rows="15" cols="80" wrap="off"
              autocorrect="off" autocomplete="off"
              autocapitalize="off" spellcheck="false">{code}</textarea>
    <p>
      <input id="save" type="submit" value="Save" />
      <input id="run" type="submit" value="Run" />
      <div id="save-status"></div>
    </p>
    <hr />
    <p>Status:</p>
    <div id="run-status"></div>
    <hr />
    <p>Output:</p>
    <div id="run-output"></div>
  </body>
</html>
```

We also need to update the `zapp.yaml` to include the new Python files. Update the bundling section:

```
bundling: ["snakebin.py", "save_file.py", "get_file.py", "index.html",
           "search_mapper.py", "search_reducer.py"]
```

Redeploy the application

Just as we did before in *part 2*, we need to redeploy the application, using `zpm`:

```
$ zpm bundle
$ zpm deploy snakebin-app snakebin.zapp --force
```

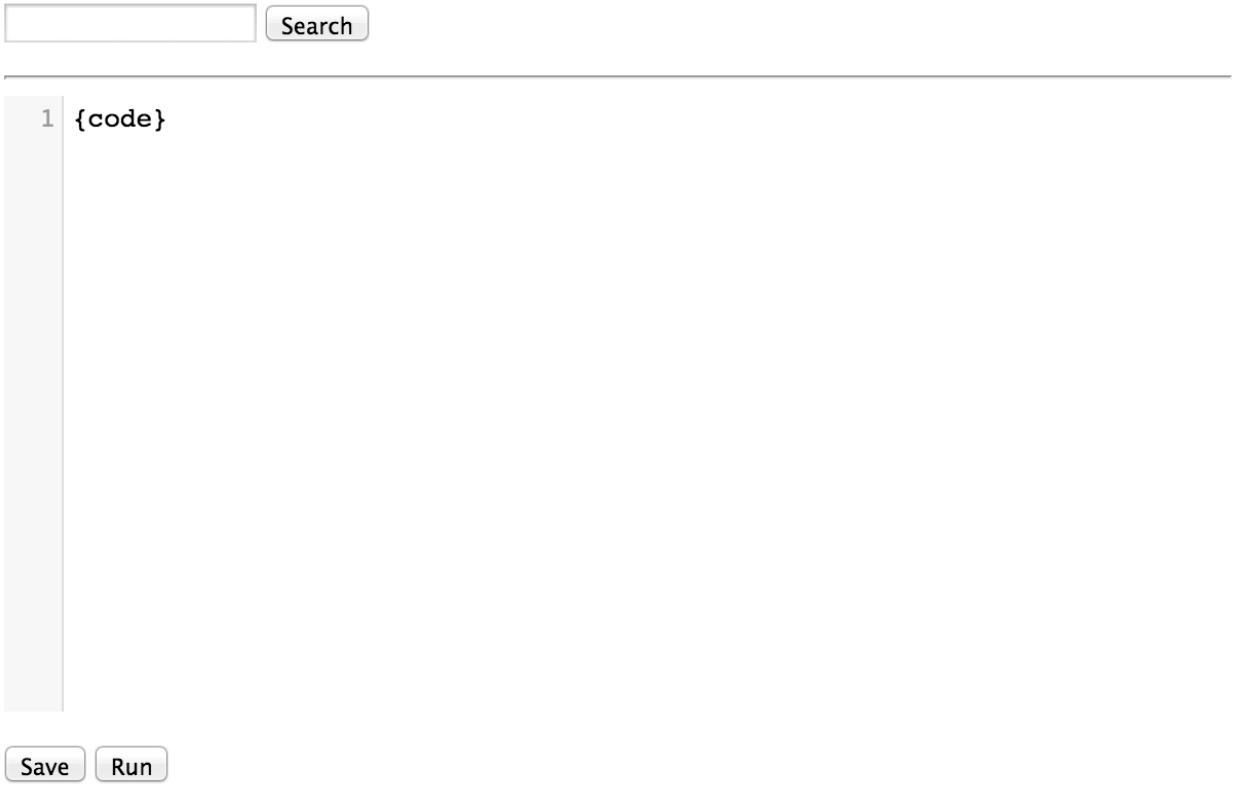
Test

First, let's try executing the search on the command line. (You should post a couple of a scripts to Snakebin first, otherwise your search won't return anything, obviously.)

```
$ curl http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/search?q=foo
["http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api/IOFW0Z8UYR", "http://127.0.0.1:8080/api/
```

Let's also test the functionality in the web browser. If you navigate to `http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api`, the new page should look something like this:

`http://127.0.0.1:8080/api/$OS_STORAGE_ACCOUNT/snakebin-api`, the new page should look something like this:



Try typing in a search term and clicking "Search".

Try also accessing the `/snakebin-api/search?q=:term` endpoint directly in the browser.

CHAPTER 4

ZeroVM Command Line Tools

Here are some tools which help with developing, testing, bundling, and deploying ZeroVM/ZeroCloud applications:

4.1 ZeroVM Command Line Tools

The ZeroVM project includes a number of command line tools that make it easier to interact with ZeroVM and ZeroCloud deployments.

4.1.1 zpm

The ZeroVM Package Manager (ZPM) is the tool that you use to create and deploy ZeroVM applications.

View the full zpm documentation here

4.1.2 zvsh and zvapp

zvsh is a simple shell program that makes it easy to run a single zerovm instance on a local machine. zvapp is a tool to run a cluser of zerovm instances on a single machine.

View to full zvsh/zvapp documentation here

CHAPTER 5

Contributing to ZeroVM

5.1 Contributing to ZeroVM

We love contributors! We want to make ZeroVM a successful platform with a great community. For this to happen, we want to hear how you use ZeroVM and work with you to make it better for your use cases!

To make contributing easy, we've documented the process we follow below. Please read through it so you know how we like to work.

tl;dr

We work like most other open source projects on GitHub. Make your changes as small commits and submit them as a pull request. We'll be in touch and work with you to get the changes into the main tree as fast as possible!

5.1.1 Pull Requests

Please use GitHub pull requests when contributing code. We will try to quickly review and merge your code.

When you submit a pull request, a review cycle is started. The typical process looks like this:

- Jenkins makes a checkout of your branch and runs tests. This will tell you if something simple broke (so you avoid breaking the build).
- Someone will take a look at the code and give you feedback. The feedback can be about the code or the design.
- Based on the feedback the pull request is either merged directly or some changes may be needed.
- If changes are needed, you should update the commits and push again. You will need to use `git push -f` this time. The pull request will automatically update and this starts a new review cycle.

Feel free to remind us on *IRC or the mailinglist* if it takes more than a couple of days for us to respond to a pull request. The IRC channel is also a great place to get some real-time feedback on a pull request.

Rebasing

You should make sure to rebase your code to the head of `master` before submitting the pull request. That ensures that there will be no merge conflicts initially.

If the pull request is not merged quickly, it can become outdated. This happens when other pull requests are merge in front of it and cause merge conflicts with the code you changed. You should then rebase the code again. See *Rebasing Stale Pull Requests* for detailed instructions.

It will generally be appreciated if you keep your code up to date in this way – even when there are no conflicts. Rebasing regularly simplifies the history.

Ordering Commits

Your commits should tell a story to the reviewer. You should therefore make sure to order them so they can be read like that. This means that you should:

- Begin with the easy and non-controversial changes first. Consider putting these changes into their own pull request so you can get them out of the way.
- Make sure each commit depends on the past only, not the future. It is very confusing for a reviewer to read commit A if it calls a function introduced later in commit B or C.

See *Reordering Commits* for details.

Fixing Your Own Mistakes

Nobody writes perfect code in the first try, so it often happens that you discover a mistake in an earlier commit. Please pay attention to this situation. When a reviewer sees a bug or some bad design in commit A, they will likely stop and begin write a comment about it. Even if you fix the problem yourself in commit B or C, it would have been much more helpful if you had avoided introducing it at all.

Please see *Amending Commits* for details on how to fix such mistakes with Git.

5.1.2 Writing Good Commits

We have collected some guidelines on how to create good commits – commits that are easy to review and understand later.

Further reading

- What's in a Good Commit?

One Change Per Commit

Please take care to ensure that each commit only deals with one logical change. Make your commits small – it is much easier to review five small commits than one massive commit. So try to err on the side of making too many commits. If you decide to split a commit, then see *Splitting Commits* for help.

Hint: It is much easier to combine commits than to split them: use `git rebase -i` and change `pick` into `squash` for the commits that should be combined into one. Done!

The question is then what one "logical change" is. Here you will have to use your best judgment. Some examples of what *not* to do:

- Do not fix a bug *and* move a function to a more logical location.
- Do not fix a bug *and* fix an unrelated typo.
- Do not fix a bug *and* reformat the code for readability.

While each change might be good, please use two commits in these cases. To put it in another way, different classes of changes should not be mixed in the same commit:

- Bug fixes.
- New features.
- Moving code around (refactoring).
- Whitespace and formatting changes.
- Style changes.
- Unrelated typo fixes.

Your goal should always be to make sure that trivial commits stay trivial: a typo fix is trivial to review so you should make it trivial for the reviewer to accept the commit.

In general, you should *stop* when you see yourself include the word "and" in a commit message. If you feel the need to make a bullet list, then you are likely including too much in the commit. In any case, you should work with your reviewer. Try to follow their advice or explain to them why the changes really belong together.

Commit Messages

Writing good commit messages is an art. You want the message to be concise and to clearly explain the proposed change. Please follow this format:

```
topic: short summary line (less than 50 characters)

After a blank line, you can include a bigger description of the
changes. Wrap the text at about 72 characters -- this makes it
nicely centered when viewed in "git log".
```

Include relevant keywords for the GitHub bug tracker. Adding "fixes #123" to the commit message will make GitHub close issue #123 when the commit is merged into the main repository.

When explaining the change remember to focus on two things:

- Explain *what* the change is. The diff technically shows this, so you should describe the change at a more high level. An excellent way to do this is to show the output before and after the change. Reviewers often have limited context so this is very helpful.
- Explain *why* you make the change. This is extremely important and the part most often left out. The commit message is often all that is left of the intent and reasoning behind a change when someone looks at it again a year later because they found a bug that seems to have been introduced by your change.

  Knowing what you changed is good, but what is really helpful in that situation is to know *why* you changed things the way you did. So please explain why this solution is the good solution. Explain what other solutions you investigated and why they won't work. Doing so will save time for the poor programmer who is debugging your code in the future.

The second point is the more important point, so please try to put emphasis on that.

Coding Style

For Python-based projects, we enforce PEP8 and Pyflakes standards. Checks are run automatically on each pull request to signal if there is a style violation.

5.1.3 Why make small commits?

Easy to Review

As mentioned several times above, small commits are easier to review than large commits. Trying to understand what a patch that touches 100 lines does is often more than twice as hard as understanding two changes that each touch 50 lines.

Makes the History Useful

When each commit is an atomic step that takes the code from one working state to another, the project history becomes a very useful tool when looking for bugs. The `git bisect` command helps you here. The command runs a binary search on the commit history to help you find the commit that introduced a particular bug.

When `git bisect` finds the first commits that triggers the bug, the real debugging can begin. If the commit it finds is small and does just one thing, then it is normally easy to understand why the problem.

Enables Revert

Sometimes it is decided that the change in the commit should be undone – maybe the commit was found using `git bisect` and it is determined that it introduced a regression or a bug.

The `git revert` command can then be used to undo the commit. It will apply an inverse patch to the repository. Since the inverse patch is based on a commit, this only works if the entire commit should be undone. If there are good, but unrelated changes in the commit, it becomes more work to revert it.

Selected Independently

A reviewer might tell you that some part of your pull request is great, while another part is not so great. If you have already split your work into small logical units, then it is easy for you to drop a commit that is not needed. Alternatively, the reviewer can easily use `git cherrypick` to select the commits he like and ignore the rest.

Further reading

- Debugging with Git: Binary Search
- 5 Reasons for Keeping Your Git Commits as Small as You Can
- git help bisect
- git help revert
- git help cherrypick

5.1.4 Branches

We follow a workflow similar to Git where we maintain a branch called `stable` for bugfix releases. This branch is continuously merged into `master` during normal development – this ensures that bugfixes are incorporated with the newest features.

As ASCII art it looks like this:

```
master: ----- o --- o --- o --- o
             /           /
stable: --- o --------- o
```

All releases are made from the `stable` branch. We release bugfixes once per month by tagging and releasing whatever code we have in the `stable` branch. We make a feature release every three months. These are also made from `stable`, but the are preceded by a merge of `master` into `stable`. It looks like this:

```
master: ----- o --- o --- o --- o
             /           /       \
stable: --- o --------- o ------- o
```

This merge brings all the new features developed since the last release onto the `stable` branch.

5.1.5 Git Tips and Tricks

We have collected some tips and tricks for solving common problems when using Git.

Further reading

- Pro Git

Rebasing Stale Pull Requests

When other pull requests are merged infront of your pull request, conflicts can occur. You as a contributor, is often the one who can solve these conflicts best – rebasing the code to the head of `master` will ensure this.

You rebase your pull request with:

```
$ git checkout master
$ git pull upstream master
$ git rebase master your-branch
```

This will first make sure that your `master` is up to date with regards to the upstream repository. The upstream should be the repository you forked on GitHub (the repository living under `github.com/zerovm/`).

Now push the branch to GitHub again with `git push -f origin your-branch`. The pull request will automatically update.

Further reading

- Branching - Rebasing
- git help rebase

Reordering Commits

Modern distributed version control systems like Git gives you the tools to reorder commits. Using the interactive mode of `git rebase`, you can easily reorder commits. While having your feature branch checked out, you run:

```
$ git rebase -i master
```

This will open your editor with a file that shown an "execution plan" for the interactive rebase. Each line represents a commit and by reordering the lines you instruct Git to reorder the corresponding commits.

After you save the file and close the editor, Git will begin reordering commits. If conflicts occur, you should use `git mergetool` to solve them. This starts your three-way merge tool which should let you figure out how to best solve the conflicts.

Further reading

- Rewriting History: Reordering Commits
- git help rebase

Amending Commits

When you want to change a commit to fix a bug, you *amend* it in the Git terminology. If the fix concerns the last commit you made, then simply use `git commit --amend` to redo the commit. You can use `git commit --amend` as many times you want to fine-tune a commit.

If you want to fix something that committed further in the past, you should instead follow this procedure:

1. Commit the fix by itself. Use `git add -p` to stage just the fix by itself if there are other changes in the same file.
2. Use `git rebase -i` to reorder the commits so that the bugfix is right after the commit that introduced the bug. In addition to reordering the commits, change the action from `pick` to `fixup`.

This will do the same as if you had used `git commit --amend` to fix the bug. With these steps, you can easily fix past mistakes.

Further reading

- Interactive Staging: Staging Patches
- git help commit
- git help add

Splitting Commits

The general advice is to make *small commits that do one thing*. Even when you try to make small commits at commit-time, you will inevitably end up with some commits that you later decide that you want to split.

We will distinguish between two cases: if the commit you want to split is the previous commit or a commit further back in the history.

- If you want to split the last commit, you run:

```
$ git reset HEAD^
$ git add foo.c
$ git commit -m 'foo: fixed #123'
$ git add bar.c
$ git commit -m 'bar: fixed typo'
```

  The important command is `git reset`, which will undo the commit. The working tree is not touched (so your modifications are still present), but the branch is rewinded and the index is reset. This means that your modifications show up again in `git diff`, for example.

  As shown, you can now commit the changes in as many commits as you like. Use `git add -p` to interactively add part of a file to be committed, for example. You will find the previous commit message as `.git/COMMIT_EDITMSG`, so you can refer to it when making new commits.

- If you want to split an earlier commit `X`, you run:

```
$ git rebase -i X^
```

In the line for `X`, change `pick` to `edit` (or just `e`), save the file, and close the editor. Git will then update to `X` to allow you to edit the commit. To actually split the commit, you will now use the procedure described above for splitting the last commit. That is, you run:

```
$ git reset HEAD^
```

to undo the commit. Then commit the files in as many small commits as you like and finally run:

```
$ git rebase --continue
```

to finish the rebase operation.

Further reading

- Rewriting History: Splitting a Commit
- git help reset
- git help rebase

5.2 Contact Us

There are several way to get support for ZeroVM and to talk to the developers.

5.2.1 Mailing Lists

We host our mailing lists at Google Groups. We have two mailing lists:

- A list for users: zerovm@googlegroups.com. This is a list for general discussion about ZeroVM: what is it, how to install it, etc, Please sign up on Google Groups.
- A list for development: zerovm-devel@googlegroups.com. This list is used for discussion about the ongoing development of ZeroVM. Please sign up on Google Groups.

5.2.2 IRC Channel

Users and developers hang out in `#zerovm` on irc.freenode.net. You can use their webchat client if you like.

CHAPTER 6

Further Reading

6.1 Glossary of Terms

Channel Channels are the key component of the ZeroVM I/O subsystem. On the host system side, channels are backed by the file system using regular files, pipes, character devices, or TCP sockets. On the guest side, each channel is a device file, which can either be used as a character or block device.

Channels are the only way for ZeroVM to communicate with the "outside world", i.e., the host file system, other ZeroVM nodes, etc. Before a ZeroVM instance starts, channels must be declared in the *manifest*.

Daemon mode ZeroVM can be started in daemon mode to reduce startup of instances/nodes in a multi-node job. Although ZeroVM startup time is ~5ms, user programs, for example, running on *zpython*, will incur an additional startup penalty. Daemon mode allows a multi-node computation to "pre-warm" an instance and fork additional copies for each unit of computation, thereby paying the additional startup time penalty only once.

Manifest A text file which must be provided to ZeroVM in order to run a *NaCL* application. Manifest files must include the following *mandatory* fields:

- `Version`: Manifest format version.
- `Program`: Full path to a *NaCL* application to be validated and run.
- `Timeout`: Timeout, in seconds. ZeroVM will stop the user program and exit after the specified time has elapsed. Valid values are 1..2147483647.
- `Memory`: Memory space in bytes available for the user program.
- `Channel`: Mapping for I/O between a ZeroVM instance and the host system. Multiple channels be specified in a manifest.

Manifests can also contain the following *optional* fields:

- `Nameserver`: Address of a nameserver which resolves network channel definitions.
- `Node`: Node ID number of a given ZeroVM instance in a cluster of VMs. Manditory if `NameServer` is specified.
- `Job`: Path to a Unix socket. Used for receiving commands/manifests and to send reports in *daemon mode*.

Native Client, NaCl See http://en.wikipedia.org/wiki/Google_Native_Client.

NVRAM (configuration file) INI-style configuration file used by ZeroVM and the *ZRT*. Includes the following:

- `[fstab]` section: Channel definitions for `tar` images to be mounted as directory/file hierarchies in the in-memory file system.
- `[env]` section: Environment variable definitions.

- `[mapping]` section: Channel definitions for stdin, stdout, stderr, and other devices not included in `[fstab]`.
- `[debug]` section: Optional. Debug verbosity level configuration.
- `[time]` section: Optional. Defines the starting time for the ZeroVM clock. With this you can define the number of seconds since Jan 1, 1970. (See Unix time.)

System Map The system map is the JSON configuration file that is passed to the ZeroCloud middleware when executing a job within ZeroCloud.

Trusted ZeroVM and the *ZRT*. Provides a secure sandbox for running *untrusted* code.

Untrusted User code run inside the ZeroVM *NaCL*-based sandbox. Untrusted code is validated before it is run.

Zebra Custom-configured deployment of *ZeroCloud*, hosted by Rackspace. *Zebra* is an alpha-testing service and playground for *ZeroCloud*.

ZeroCloud Middleware for OpenStack Swift which provides the capability to run ZeroVM applications on object storage nodes. Can be used to initiate map/reduce-style jobs on collections of Swift objects.

See https://github.com/zerovm/zerocloud.

ZeroVM Application, zapp An archive file (typically created by *zpm*) containing a `zapp.yml` configuration file and user application code.

ZeroVM Package Manager, zpm Command-line utility which helps to create, bundle, deploy (to *ZeroCloud*), and execute (on *ZeroCloud*) ZeroVM user applications.

See https://github.com/zerovm/zpm.

ZeroVM Runtime, ZRT Provides a POSIX-like environment and in-memory file system for use by *untrusted* user programs.

ZeroVM Shell, zvsh Utility program which makes ZeroVM easy to use by providing rich command-line options for running and debugging ZeroVM instances. Also includes manifest/NVRAM configuration file generation functionality (so you don't have to write all of your configuration files by hand).

See https://github.com/zerovm/zerovm-cli.

zpython ZeroVM ports of CPython interpreters. There are ongoing efforts to port both Python 2.7.3 and Python 3.2.2 to run inside ZeroVM.

Zwift Deprecated synonym for *ZeroCloud*.