RX-M Cloud Native Consulting

Microservices

Lab 1 – Microservice Hello World

Microservices represent a specialization of service-oriented architecture (SOA) used to build cloud native distributed applications. Services in a microservice architecture (MSA) are processes that communicate with each other over technology-agnostic network interfaces. The microservices architectural style has been shaped by, and enables, the DevOps movement, integrating particularly well with Agile teams and continuously deployed systems. Because of its focus on incremental and localized changes, Microservices enable rapid innovation in large scale systems, where gridlock might otherwise ensue.

In this lab and the labs ahead we will build some simple services to explore and demonstrate some of the key properties of microservices using the classroom supplied virtual lab machine. The class room VM is an Ubuntu 16.04 server with a basic desktop installed. Versions of the lab system are available for Virtual Box and VMWare desktop hypervisors. If you do not have a copy of the lab VM, instructions for downloading it can be found here:

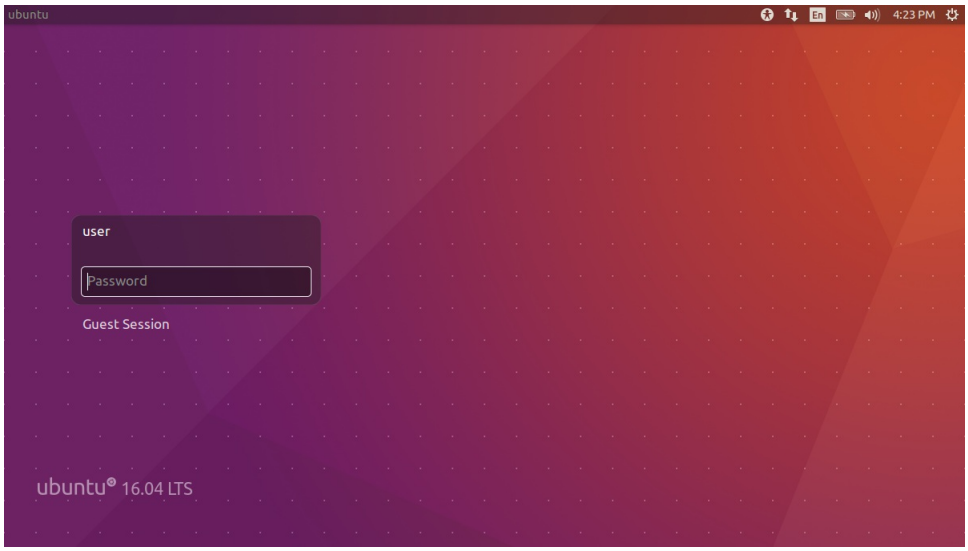
- https://github.com/RX-M/classfiles/blob/master/lab-setup.md

N.B. If you cannot access the lab VM, any base Ubuntu 16.04 system will work.

The Lab system user account is "user" and the password is "user" (with full sudo permissions).

1. Login to the Lab System

Start your classroom virtual machine and log in as "user" with the password "user".



2. Conceptualizing a Microservice-based System

Microservice based applications are known for being highly scalable. This makes them a good fit for systems that interact with the growing population of connected things in the real world, like cell phones, cars and even trash cans. The microservice architectural style is also a key attribute of cloud native applications, typically described as:

- Microservice oriented
- Container packaged
- Dynamically orchestrated

We will take a look at each of these aspects in this lab and the labs ahead.

2.a The Trash Can App

Cities can save a large amount of money by simply knowing when their trash cans are full. This would avoid trash overflowing on the streets and also keep the city from sending maintenance personal out to collect trash from trash cans that are not full. In this lab and the labs ahead we will incrementally build a microservice-based application to create a cloud native solution to the trash problem in cities around the world.

Problem Statement: Cities currently schedule trash pickups by calendar time not by necessity. Some trash cans overflow, polluting, and some are checked by staff even though they are empty, wasting time and fuel.

Our goal is to solve this problem.

Our initial system requirements include the following:

- Manage trash can inventory
- Track trash can locations
- Track trash levels in cans
- Produce optimized trash pick up routes for trash trucks

Optimally all of the trash cans deployed in our city will be "smart", and able to tell us their trash levels via Low-Power Wide-Area Network

(LPWAN). We can use an algorithmic approach for "dumb" trash cans by having staff report trash levels at various time intervals, allowing us to use machine learning to predict trash levels with better than random accuracy in the future.

2.b System Design

Most software architects and designers working with modern technology prefer to avoid Big Design Up Front (BDUF). However spending some time understanding the problem domain and establishing the basic subsystems involved is almost always an essential first step. System design is the process of establishing the basic subsystems of the overall solution and defining their relationships. These subsystems should map directly to the bounded contexts identified within the problem domain. The relationships between subsystems will be defined by service contracts implemented by interfaces.

> Bounded Context is a central pattern in Domain-Driven Design [DDD] (term coined by Eric Evans in his 2003 book). DDD decomposes complex software systems into bounded contexts based on natural partitions in the real world problem domain and the unique perspectives that those contexts embody.

One thing we need not concern ourselves with is getting everything exactly right up front. The microservices architectural style is focused on learning about the system being built incrementally and supporting evolution in the system architecture. We can change the structure of the system as we go incrementally so that the final solution is arrived at, not envisioned prior to practical experience.

Also in this early design phase we may want to select some initial communications schemes, as well as programming languages and tools. In a microservice architecture we generally want to let each development team build their services the way they think best, after all, they are (or will be) the experts on implementing the services they create. This is also a particularly Agile approach. However, for the entirety of the application to work well together we'll need to think about the ways in which the services communicate, the types of data stores we want to use, which messaging platforms we will work with and other similar things that give the overall application interoperability and control the scope of systems our platform team will have to support and monitor.

Selecting an interface scheme

Let's assume that the City already has several software systems and front ends. In order to ensure that our trash app can easily integrate with existing parts of the City's platform we will provide API based access to the trash app. In essence the trash app is itself is a bounded context. Imagine that after discussion we decide to make interfaces within our application as open as possible and that back end performance is not a primary concern. Choosing *REST* as an API approach might make sense in this situation.

Selecting programming languages

While letting our developers work in programming languages they like and are familiar with is very important, languages are complex and we don't want to have too many in a single system if we can avoid it. One or two is great. Three or four could also be justified but the justification should be clear. Five or six could begin to produce diminishing returns, making it hard for developers to change teams or debug/augment other services they need to use. Mastering a programming language is no small task and lack of experience can produce bugs and leave us with no experienced coders to do the debugging.

Let's assume our team is familiar with *Python* and that this is a suitable language for the requirements we have established. With this as a given, we'll focus on building our first service in Python.

2.c Microservice or Monolith

To begin our project we may start by building several individual microservices or we may start by creating a single monolithic service that solves the entire trash problem. Wait! Isn't this a class on microservices? Why on earth would we build a monolith?

Often it is easier to create a basic solution for a given problem component as a single program. It may not scale cleanly or be easy to maintain but it can teach us many things we do not yet know about the application and the problem domain. Once completed we can separate out the microservices, decomposing the monolith into its essential components. We will start right out with microservices to serve the goals of the course.

2.d The First Service

Ward Cunningham (inventor of the Wiki) suggests that you should always build the simplest thing that could possibly work first. While this advice is directed at the process of constructing software, it mirrors the microservice/monolith thought process. The goal is to gain experience before making decisions that will be expensive to countermand. It is easy to change the implementation of a service behind an interface, it is more expensive to change the responsibilities (therefore interface) of services.

Perhaps the simplest thing we could build to get started with our trash app is a service that inventories all of the trash cans in the city (meeting requirement 1). If we were to flesh out the requirement, we might unearth a user story something like this:

- As a city engineer
- I want to keep track of all of the city's trash cans and their locations
- So that I can efficiently deploy and service trash cans throughout the city

Following a brief interview, we discover that the city engineers are interested in tracking the following trash can attributes:

- ID - The identifying number of the trash can itself
- Deployed - True if the trash can is deployed for use in the city or False if the trash can is in the warehouse
- Power Source - How the trash can's embedded trash level detector is powered (grid/solar/wind/battery/none)
- Latitude - The trash can's position to 1000s of a degree (e.g. 31.776, south latitudes are negative)
- Longitude - The trash can's position to 1000s of a degree (e.g. 35.217, west longitudes are negative)
- Capacity - How many liters of trash the container can hold

We can describe such a representation in one of the many REST API description languages. For example, in OAI (the Open API Initiative) these TrashCan object features could be described like this:

```
openapi: 3.0.0
info:
  title: Trash Can App
  description: Next generation trash management is here!
  version: 0.1.0

components:
  schemas:
    TrashCan:
      type: object
      properties:
        id:
          type: integer
        deployed:
          type: bool
        power_source:
          type: string
        latitude:
          type: float
        longitude:
          type: float
        capacity:
          type: integer
      required:
        - id
```

Defining a contract that users can build stub code from or reference as documentation is a critical part of the interface design process.

This trash can "model" is a conceptual representation. It is used within the bounded context of our application but also consumed and produced by our users. In other words, users will want to retrieve trash can information and add new trash cans to the system from time to time. This makes the trash can model part of our interface. In some API terms, the trash can would be a resource, in others, a type, in still others, a class/object or a message. Nomenclature aside, we will need to exchange these "TrashCan" things with the outside world. This means that the representation will be part of our interface contract. Even so the representation should also be extensible so that we can evolve the interface over time if needed.

2.e Languages and Tools

We have decided to initially code in *Python* and expose a *REST* API and our first service will inventory the city's trash cans. While it is probably clear that the trash can data should be persisted we'll begin by simply storing the trash can data in memory. This will allow us to grow confident about the size and shape of the trash can data before committing to a storage platform. Also, and critically, it will keep us from letting the storage solution drive our design decisions. Ultimately the business will evolve around its natural boundaries (bounded contexts), not around our database choices.

Creating a REST API from scratch would be reinventing the wheel. There are many good Python frameworks available for this. For example *Flask*. Flask is a BSD licensed microframework for Python based on Werkzeug and Jinja 2. Werkzeug is a toolkit for WSGI, the standard Python interface for web applications, and Jinja2 renders templates. Sounds like a fit! In the real world you might survey the landscape a bit more, replace Werkzeug with uWSGI/Nginx etc., but remember the framework/stack we use is an implementation detail hidden behind the API and we can change it at will without impacting the users of the service.

Selecting a service isolation strategy

Given that multiple teams may be building Python services, we may run into library version conflicts. What if team A wants (needs) to use Python 2.7 and team B uses Python 3.4? What if team C needs Python 3.3. What about conflicting library versions? These are common problems. One popular way to solve these problems is with Python Virtual environments. What about containers you say? Try this first and you will be happy to switch to containers when we cover them in a later lab!

Selecting a serialization scheme

Exchanging data over a REST API involves some form of data serialization. For REST APIs this typically involves XML or JSON. While XML is mature and has wide spread tool support, it is not as easy to read by humans, takes longer to parse and consumes more bytes than the equivalent JSON. Given the groundswell of support for JSON in the industry we will use *JSON* exclusively.

Selecting a character encoding scheme

When dealing with text formats like JSON it is also often important to select a character encoding scheme. Here again the industry has shown wide spread support for UTF-8 on the wire. We'll use UTF-8 for all string data in our API interfaces to make string exchange easy, compact, and consistent.

Automating the CI/CD pipeline

We should also choose a source code management system, testing approaches and tools, CI/CD solutions, etc. Due to time limitations, we'll restrict our additional tooling to a source code version control system. Git is the most popular tool in this space so lets choose it for simplicity.

We have now selected our key languages, tools, and technologies for our first microservice development push:

- Microservice architecture
- Python Language for (at least some) service development
- REST API (until we have a reason to use something else)
- Python Virtualenv (to support multiple isolated Python/Library version installs on a single box)
- Flask Python REST library
- JSON data serialization
- UTF-8 character encoding
- Git version control

3. Setting Up a Development Environment

Before we begin coding we will need to prepare a development environment, but which Python version will we use and which Flask version? What if we need to change versions? What if other programs running on the system install incompatible versions of Python or Flask?

Hmmm. Running lots of independent programs on the same computer could be a challenge. Yet creating a VM for each microservice would be pretty inefficient.

One possible solution is to use a language specific isolation scheme. For example, Ruby offers the Ruby Version Manager (RVM), NodeJS uses NPM to install libraries on a per project basis and Python offers Virtualenv. Virtualenv enables multiple side-by-side installations of Python and its libs. We'll configure our initial development environment with Virtualenv.

Open a terminal on your lab system and create a working directory for your Trash Can system.

```
user@ubuntu:~$ mkdir trash-can

user@ubuntu:~$ cd trash-can/
```

Now create a subdirectory for the inventory service:

```
user@ubuntu:~/trash-can$ mkdir inv

user@ubuntu:~/trash-can$ cd inv/
```

Before we install any packages we should update the package indexes to ensure we have the latest package lists:

```
user@ubuntu:~/trash-can/inv$ sudo apt-get update

Hit:1 http://us.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://us.archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:3 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:4 http://us.archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:5 http://us.archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [709 kB]
Get:6 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [435 kB]
Get:7 http://us.archive.ubuntu.com/ubuntu xenial-updates/main i386 Packages [660 kB]
Get:8 http://security.ubuntu.com/ubuntu xenial-security/main i386 Packages [391 kB]
Get:9 http://us.archive.ubuntu.com/ubuntu xenial-updates/main Translation-en [294 kB]
Get:10 http://us.archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [580 kB]
Get:11 http://security.ubuntu.com/ubuntu xenial-security/main Translation-en [189 kB]
Get:12 http://us.archive.ubuntu.com/ubuntu xenial-updates/universe i386 Packages [537 kB]
Get:13 http://us.archive.ubuntu.com/ubuntu xenial-updates/universe Translation-en [234 kB]
Fetched 4,336 kB in 4s (943 kB/s)
Reading package lists... Done

user@ubuntu:~/trash-can/inv$
```

Now we can install the Python PIP package manager, the standard tool for installing Python libraries (this will also install Python 2.7):

```
user@ubuntu:~/trash-can/inv$ sudo apt-get install python-pip

...
```

We may not have the latest version of PIP when installed from Ubuntu packages. Ask PIP to upgrade itself in case a newer version is available.

```
user@ubuntu:~/trash-can/inv$ pip -V

pip 8.1.1 from /usr/lib/python2.7/dist-packages (python 2.7)

user@ubuntu:~/trash-can/inv$ sudo -H pip install --upgrade pip

Collecting pip
  Downloading pip-9.0.1-py2.py3-none-any.whl (1.3MB)
```

```
    100% |████████████████████████████████| 1.3MB 911kB/s
Installing collected packages: pip
  Found existing installation: pip 8.1.1
    Not uninstalling pip at /usr/lib/python2.7/dist-packages, outside
    environment /usr
Successfully installed pip-9.0.1

user@ubuntu:~/trash-can/inv$
```

Next install Python Virtual Environments using PIP:

```
user@ubuntu:~/trash-can/inv$ sudo -H pip install virtualenv

Collecting virtualenv
  Downloading virtualenv-15.1.0-py2.py3-none-any.whl (1.8MB)
    100% |████████████████████████████████| 1.8MB 733kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.1.0

user@ubuntu:~/trash-can/inv$
```

Now we can create a virtual environment for our inventory project:

```
user@ubuntu:~/trash-can/inv$ virtualenv venv
New python executable in /home/user/trash-can/inv/venv/bin/python
Installing setuptools, pip, wheel...done.

user@ubuntu:~/trash-can/inv$ ls -laF
total 12
drwxrwxr-x 3 user user 4096 Jan  7 03:08 ./
drwxrwxr-x 3 user user 4096 Jan  7 03:04 ../
drwxrwxr-x 6 user user 4096 Jan  7 03:08 venv/

user@ubuntu:~/trash-can/inv$ ls -lF venv/
total 20
drwxrwxr-x 2 user user 4096 Jan  7 03:08 bin/
drwxrwxr-x 2 user user 4096 Jan  7 03:08 include/
drwxrwxr-x 3 user user 4096 Jan  7 03:08 lib/
drwxrwxr-x 2 user user 4096 Jan  7 03:08 local/
-rw-rw-r-- 1 user user   60 Jan  7 03:08 pip-selfcheck.json
user@ubuntu:~/trash-can/inv$
```

Creating a virtual environment in a given directory just creates a venv subdirectory containing several shell scripts. The shell scripts set the Python executable and library path to the venv directory, allowing each project directory to have its own unique set of Python interpreters and libraries. To activate the virtual environment you will need to run the activate script (every time you want to run, debug or test your Python program). Enable the Python virtual environment:

```
user@ubuntu:~/trash-can/inv$ source venv/bin/activate

(venv) user@ubuntu:~/trash-can/inv$ python --version

Python 2.7.12

(venv) user@ubuntu:~/trash-can/inv$ which python

/home/user/trash-can/inv/venv/bin/python

(venv) user@ubuntu:~/trash-can/inv$
```

Our Python virtual environment ensures that our project will always have a stable version of Python and dependent packages. Virtual environments are nice and widely used but they will not protect us from all changes to the host system that indirectly impact Python or its libraries. It is also not easily deployable. If someone needs to run our Python application we will have to tar/zip the virtual environment and the app together before sharing it to guarantee consistency. We'll look at better more flexible means for microservice packaging later (containers!).

As you work with the lab system remember that you can enable the virtual environment and disable it with the following commands from within the `~/trash-can/inv` directory:

- `source venv/bin/activate`
- `deactivate`

For example:

```
(venv) user@ubuntu:~/trash-can/inv$ echo $PATH
```

```
/home/user/trash-can/inv/venv/bin:...

(venv) user@ubuntu:~/trash-can/inv$ deactivate

user@ubuntu:~/trash-can/inv$ echo $PATH

/home/user/bin:...

user@ubuntu:~/trash-can/inv$ which python

/usr/bin/python

user@ubuntu:~/trash-can/inv$ . venv/bin/activate

(venv) user@ubuntu:~/trash-can/inv$ echo $PATH

/home/user/trash-can/inv/venv/bin:...

(venv) user@ubuntu:~/trash-can/inv$
```

You can tell when you have the virtual environment enabled because the prompt is prefixed with `(venv)`.

Now that we have PIP installed and a virtual Python environment active, we can install the Flask framework into our virtual environment (make sure your virtual environment is currently active, otherwise Flask will be installed on the global host path):

```
(venv) user@ubuntu:~/trash-can/inv$ pip install Flask

Collecting Flask
  Downloading Flask-0.12.1-py2.py3-none-any.whl (82kB)
    100% |████████████████████████████████| 92kB 1.4MB/s
Collecting itsdangerous>=0.21 (from Flask)
  Downloading itsdangerous-0.24.tar.gz (46kB)
    100% |████████████████████████████████| 51kB 4.9MB/s
Collecting click>=2.0 (from Flask)
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
    100% |████████████████████████████████| 71kB 3.8MB/s
Collecting Werkzeug>=0.7 (from Flask)
  Downloading Werkzeug-0.12.1-py2.py3-none-any.whl (312kB)
    100% |████████████████████████████████| 317kB 2.1MB/s
Collecting Jinja2>=2.4 (from Flask)
  Downloading Jinja2-2.9.6-py2.py3-none-any.whl (340kB)
    100% |████████████████████████████████| 348kB 2.7MB/s
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->Flask)
  Downloading MarkupSafe-1.0.tar.gz
Building wheels for collected packages: itsdangerous, MarkupSafe
  Running setup.py bdist_wheel for itsdangerous ... done
  Stored in directory: /home/user/.cache/pip/wheels/fc/a8/66/24d655233c757e178d45dea2de22a04c6d92766abfb741129a
  Running setup.py bdist_wheel for MarkupSafe ... done
  Stored in directory: /home/user/.cache/pip/wheels/88/a7/30/e39a54a87bcbe25308fa3ca64e8ddc75d9b3e5afa21ee32d57
Successfully built itsdangerous MarkupSafe
Installing collected packages: itsdangerous, click, Werkzeug, MarkupSafe, Jinja2, Flask
Successfully installed Flask-0.12.1 Jinja2-2.9.6 MarkupSafe-1.0 Werkzeug-0.12.1 click-6.7 itsdangerous-0.24

(venv) user@ubuntu:~/trash-can/inv$
```

This installs Flask and all of its dependencies into our local venv directory.

```
(venv) user@ubuntu:~/trash-can/inv$ find . -name "*Flask*"
./venv/lib/python2.7/site-packages/Flask-0.12.1.dist-info
(venv) user@ubuntu:~/trash-can/inv$
```

4. Building a Test Service

Let's code a very simple Flask service to test our installation. Once we have this hello service working we can set about building a proper Trash Inventory service (in the next lab).

Using your favorite editor (`vim` is good for the CLI folks, nano may be easier for CLI users not familiar with VI and `gedit` should work for the GUI inclined (you can always install something else if you prefer) create the following Python program:

```
(venv) user@ubuntu:~/trash-can/inv$ vim inv.py

(venv) user@ubuntu:~/trash-can/inv$ cat inv.py
```

```
""" Trash Can Inventory Service
"""

from flask import Flask
from flask import jsonify
from flask import request

app = Flask(__name__)

VERSION = "0.1"

@app.route('/')
def version():
    """ Root IRI returns the API version """
    return jsonify(version=VERSION)
```

```
(venv) user@ubuntu:~/trash-can/inv$
```

This program begins by importing the "Flask" class from the Python flask package installed in our virtual environment. An instance of this class will support our Web Service. Next we import the "jsonify" function which we will use to return JSON representations of our API responses. Finally the "request" import will allow us to inspect the request data sent by clients.

After the import statements we create an instance of the Flask class with the `__name__` argument, which in Python resolves to the name of the module (`inv` for the the `inv.py` file in our case). We also set a constant API version of 0.1.

We then use the route() decorator to tell Flask what URL should trigger our function. The function simply returns the versions string in this case. The jsonify function allows us to convert Python key value pairs into legal JSON and it also sets the content type header to `application/json` in the response object.

To run the service we need to export the *FLASK_APP* environment variable (setting it to the file name of our Python program).

Then we can run Flask:

```
(venv) user@ubuntu:~/trash-can/inv$ export FLASK_APP=inv.py

(venv) user@ubuntu:~/trash-can/inv$ python -m flask run -p 8080

 * Serving Flask app "inv"
 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
```

In the example above we ask Python to run the installed Flask module (-m flask) and then ask flask to run our service on port 8080 (run -p 8080). Flask locates our inv.py module through the FLASK_APP environment variable.

Flask reports that it is running the "inv" app and that it is listening on the local loopback interface (127.0.0.1), port 8080. You can choose a host interface (other than the default loopback) using the `-h` switch.

Now open another terminal window and test your service with `curl` :

```
user@ubuntu:~/trash-can/inv$ curl -s localhost:8080

{
  "version": "0.1"
}

user@ubuntu:~$
```

We can use the `-v` verbose, or -vv very verbose switches to see the HTTP headers and other data exchanged between `curl` and our service:

```
user@ubuntu:~/trash-can/inv$ curl -vs localhost:8080

* Rebuilt URL to: localhost:8080/
*   Trying ::1...
* connect to ::1 port 8080 failed: Connection refused
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
```

```
< Content-Length: 23
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:26:54 GMT
<
{
  "version": "0.1"
}
* Closing connection 0

user@ubuntu:~/trash-can/inv$
```

Back in the application terminal you should see the server logging the inbound requests.

```
(venv) user@ubuntu:~/trash-can/inv$ python -m flask run -p 8080

 * Serving Flask app "inv"
 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
127.0.0.1 - - [07/Jan/2018 03:26:32] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [07/Jan/2018 03:26:54] "GET / HTTP/1.1" 200 -
```

So far so good. Type CTRL+C to quit the server.

```
127.0.0.1 - - [07/Jan/2018 03:26:54] "GET / HTTP/1.1" 200 -
^C

(venv) user@ubuntu:~/trash-can/inv$
```

5. Checking the Code Into Source Control

At this point normal programmers will start becoming nervous. We have written several lines of code and yet we have not checked this code into a source code control system to protect it.

For this project we'll use the Git source code control system to ensure that we do not accidentally delete or overwrite our work. To begin we'll use the `git init` command to create a repository for our code:

```
(venv) user@ubuntu:~/trash-can/inv$ git init
Initialized empty Git repository in /home/user/trash-can/inv/.git/
(venv) user@ubuntu:~/trash-can/inv$
```

The *init* subcommand creates a hidden `.git` folder in your project directory. We can use the `git status` subcommand to see the state of our project:

```
(venv) user@ubuntu:~/trash-can/inv$ git status

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        inv.py
        inv.pyc
        venv/

nothing added to commit but untracked files present (use "git add" to track)

(venv) user@ubuntu:~/trash-can/inv$
```

The status subcommand displays all of the files that are not being tracked for version control by Git. In our case the `.py` file is our source code and the only thing we would like to track. The `.pyc` file is a temporary compiled version of our source and the `venv/` directory is the Python virtual environment. We can tell git to ignore `pyc` files and the `venv` directory by adding them to a `.gitignore` file. Try it:

```
(venv) user@ubuntu:~/trash-can/inv$ echo "*.pyc" > .gitignore

(venv) user@ubuntu:~/trash-can/inv$ git status

On branch master

Initial commit

Untracked files:
```

```
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        inv.py
        venv/

nothing added to commit but untracked files present (use "git add" to track)

(venv) user@ubuntu:~/trash-can/inv$
```

Adding `*.pyc` to the `.gitignore` caused git to no longer show `.pyc` files in its statistics. This did add the `.gitignore` file itself to the list of unchecked in files. Because we don't want to lose this file we should commit it.

Now add the `venv/` directory to the ignore list:

```
(venv) user@ubuntu:~/trash-can/inv$ echo "venv/" >> .gitignore

(venv) user@ubuntu:~/trash-can/inv$
```

```
(venv) user@ubuntu:~/trash-can/inv$ git status

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        inv.py

nothing added to commit but untracked files present (use "git add" to track)

(venv) user@ubuntu:~/trash-can/inv$
```

Now `git status` shows us only the files that we want to commit to revision control, the Python source code and our `.gitignore`.

Git allows you to stage a set of files to be committed at once. Stage the `.gitignore` and the `.py` file:

```
(venv) user@ubuntu:~/trash-can/inv$ git add .gitignore
```

```
(venv) user@ubuntu:~/trash-can/inv$ git status

On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        inv.py

(venv) user@ubuntu:~/trash-can/inv$
```

```
(venv) user@ubuntu:~/trash-can/inv$ git add inv.py

(venv) user@ubuntu:~/trash-can/inv$
```

```
(venv) user@ubuntu:~/trash-can/inv$ git status

On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

```
        new file:   .gitignore
        new file:   inv.py

(venv) user@ubuntu:~/trash-can/inv$
```

Now both of our changed files are staged for commit. To commit them use the *commit* subcommand:

```
(venv) user@ubuntu:~/trash-can/inv$ git commit -m "initial trash can inventory commit"

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'user@ubuntu.(none)')

(venv) user@ubuntu:~/trash-can/inv$
```

As you can see, Git is not interested in committing code from anonymous parties. As suggested in the output, add your name and email address to the Git configuration (you can use phony info if you like).

```
(venv) user@ubuntu:~/trash-can/inv$ git config user.email "me@example.com"

(venv) user@ubuntu:~/trash-can/inv$ git config user.name "Bat Man"
```

Now retry your commit:

```
(venv) user@ubuntu:~/trash-can/inv$ git commit -m "initial trash can inventory commit"

[master (root-commit) 1bba49d] initial trash can inventory commit
 2 files changed, 14 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 inv.py

(venv) user@ubuntu:~/trash-can/inv$
```

You can use the `git log` subcommand to view the commits in your repository:

```
(venv) user@ubuntu:~/trash-can/inv$ git log

commit 50e353ae65d4d8bdadd521c83ff4f9b908db0b2c
Author: Bat Man <me@example.com>
Date:   Sun Jan 7 03:29:34 2018 -0800

    initial trash can inventory commit

(venv) user@ubuntu:~/trash-can/inv$
```

If you know Git already these steps were probably second nature. If not you will get a chance to pick up the basics as we progress through the labs in this course. However learning Git properly is another class...

We now have a proper development environment and a tested microservice skeleton exposing a REST API with JSON formatted data exchange. All checked into a Git repository. In the next lab we'll implement the Trash Can Inventory functionality using the tools we have configured here.

Congratulation you have completed the lab!