RX-M Cloud Native Consulting

Microservices

Lab 2 – Restful APIs

In the last lab we selected languages and tools for our Microservices platform. In this lab we will create a microservice which manages the trash can inventory for our trash can distributed application. This application will have a microservice architecture which is one of the primary prerequisites for cloud native applications.

The Cloud Native Compute Foundation (CNCF) defines "cloud native systems" as having three key properties:

- *Container packaged* - Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation. Improves overall developer experience, fosters code and component reuse and simplifies operations for cloud native applications.
- *Dynamically managed* - Actively scheduled and actively managed by a central orchestrating process. Radically improves machine efficiency and resource utilization while reducing the cost associated with maintenance and operations.
- *Micro-services oriented* - Loosely coupled with dependencies explicitly described (e.g. through service endpoints). Significantly increases the overall agility and maintainability of applications.

To begin this lab we'll add support for Trash can data storage and retrieval in our microservice skeleton from Lab 1.

1. Building the Inventory Service

To build a proper REST interface for our inventory service we should allow clients to add trash cans to the system and then retrieve them. The "resource" in this case is a trash can. Therefore it might make sense to create an IRI in our waste management application such as:

- `/waste/cans`

With `/waste` as the root of our application and `./cans` as the collection of trash cans.

The REST approach typically involves creating APIs based on resources and then using HTTP verbs to interact with those resources. In a REST interface it would be fairly common to allow new cans to be POSTed to the collection and allow users to GET existing cans from the collection, using a path parameter such as:

- `/waste/cans/{can-id}`

Using GET on the cans collection should return all of the cans (or one page of cans with semantics for retrieving the next page).

To begin let's add support for POSTing new cans to the `/waste/cans` IRI and support for GETing all of the registered cans. To start we will simply house all of the data in memory. This is not a long term solution but it will be sufficient during the early development phase as we learn about our application, its problem domain and the structure of trash can data. We'll be much better prepared to select a storage solution after some basic experience with real data.

Open the `inv.py` skeleton from lab 1 and begin by adding a Python dictionary called *wcans* to the end of the file. We'll use the *wcans* dictionary to house our can data in the prototype microservice.

```
""" 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)

wcans = {}
```

Now we can add a handler for the `/waste/cans` route. Add the following `cans()` function to the bottom of your `inv.py` file:

```
@app.route('/waste/cans', methods=['GET', 'POST'])
def cans():
    """ /cans collection allows POST of new cans and GET of all cans """
    if request.method == 'POST':
        can = request.get_json()
        wcans[can["id"]] = can
        return jsonify(can), 201
    else:  #GET
        return jsonify(list(wcans.values()))
```

The `cans()` handler function (called a view function in Flask) responds to GET and POST HTTP requests. POST requests must pass a JSON formatted Trash Can to the API. The `cans()` code gets the JSON body from the request and saves it in a variable called can. The can is then saved into the waste cans dictionary, wcans, with the can *id* as the key.

REST API clients expect new resource requests to return the object created and a 201 status for "Created". The Flask `jsonify()` function allows us to easily return the JSON representation of the accepted can along with the 201 status code.

If the user is using the GET verb on the IRI we simply convert the values in the `wcans` dictionary into a list, turn that into a JSON string and return it. The `jsonify()` function returns a 200 status (Ok) by default.

2. Testing the Inventory Service

At this point we have made the smallest testable increment to our service API. Modern development sensibilities typically suggest that we should have either already written a test for this functionality (Test Driven Development, TDD) or that we should write tests now. By writing API tests and checking them in with our code we can ensure that our code does not regress and/or violate our interface contract. Tests can also act as example code for API users.

Given the amount of material we have to cover here we will simply test the interface by hand. Take a look at the RX-M TDD and ATDD course offerings if you are interested in learning more about Test driven development and microservices.

We can use curl, a simple but powerful and agnostic tool, to test our REST APIs.

Your service should look something like this:

```
(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)

wcans = {}

@app.route('/waste/cans', methods=['GET', 'POST'])
def cans():
    """ /cans collection allows POST of new cans and GET of all cans """
    if request.method == 'POST':
        can = request.get_json()
        wcans[can["id"]] = can
        return jsonify(can), 201
    else:  #GET
        return jsonify(list(wcans.values()))
(venv) user@ubuntu:~/trash-can/inv$
```

Now run it:

```
(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)
```

Now try curling the `/waste/cans` IRI:

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans

*   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 /waste/cans 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: 3
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:31:38 GMT
<
[]
* Closing connection 0

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

Because our service starts up with an empty *wcans* dictionary we receive an empty list of cans. The status is 200 OK so things are working so far.

Next lets try POSTing a new can to the service:

```
user@ubuntu:~/trash-can/inv$ curl -sv -H "Content-Type: application/json" -d
'{"id":"42","deployed":"False","power":"grid","lat":"31.776","lon":"35.217","capacity":"500"}'
http://localhost:8080/waste/cans

*   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)
> POST /waste/cans HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 92
>
* upload completely sent off: 92 out of 92 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 123
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:32:11 GMT
<
{
  "capacity": "500",
  "deployed": "False",
  "id": "42",
  "lat": "31.776",
  "lon": "35.217",
  "power": "grid"
}
* Closing connection 0

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

The `curl` command we use to POST data requires several additional switches:

- -vv: Very Verbose (while optional, this is what produces all of the informative header and status logging)
- -H: sets headers, here we must tell the service we are POSTing JSON, using the Content-Type header with the `application/json` media type
- -X: sets the HTTP verb to POST (the default operation when setting data but being explicit doesn't hurt!)
- -d: sets the data (body) to transmit, the JSON formatted trash can data in our case

The `curl` output shows that our POST request returned a 201 created, indicating the service created a new resource in response to our request. Any time a 201 status is returned the new object is expected to be returned, as indeed it was. There are still some concerns however. First, all of our data is accepted without regard to the fields supplied or their values. Second, a 201 response should also return a location header identifying the IRI where the new resource can be found. We should address both of these issues.

Before moving on, confirm the can is there!

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans

*   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 /waste/cans 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: 143
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:32:47 GMT
<
[
  {
    "capacity": "500",
    "deployed": "False",
    "id": "42",
    "lat": "31.776",
    "lon": "35.217",
    "power": "grid"
  }
]
* Closing connection 0

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

3. API Validation and Design by Contract

APIs are contracts, a set of agreements between the service and its clients as to what is and is not legal in an interface exchange. All too often these contracts are implicit to a small or large degree.

Design By Contract (DbC) is a software correctness methodology that uses pre-conditions and post-conditions to document (or programmatically assert) the change in state caused by an interface call. This type of verification of state change can be particularly useful in free form interface systems such as REST. Mechanically, a user could pass just about any type of JSON string and request that we add it as a new trash can.

To protect against this and document our interface requirements in one fell swoop, we can create a validation method that ensures only valid trash can representations are added to the system. Add the following validation function to the bottom of your `inv.py` source file.

```
def validate_can(can):
    """ DbC checks for required can fields and settings
            returns False if can is valid
            returns a string describing the error otherwise
    """
    try:
        #Test id
        can["id"] = int(can["id"])
        if can["id"] < 0 or 999999999 < can["id"]:
            raise ValueError("can.id out of range [0..999999999]")
        #Test deployed
        if can["deployed"] == "True":
            can["deployed"] = True
        elif can["deployed"] == "False":
            can["deployed"] = False
        else:
            raise ValueError("can.deployed must be Boolean (True, False)")
        #Test capacity
        can["capacity"] = float(can["capacity"])
        if can["capacity"] <= 0.0 or 9999 < can["capacity"]:
            raise ValueError("can.capacity out of range (0.0..9999.0]")
        #Test lat
        can["lat"] = float(can["lat"])
        if can["lat"] < -90.0 or 90.0 < can["lat"]:
            raise ValueError("can.lat out of range [-90.0..90.0]")
        #Test lon
        can["lon"] = float(can["lon"])
        if can["lon"] < -180.0 or 180.0 < can["lon"]:
            raise ValueError("can.lon out of range [-180.0..180.0]")
        #Test power
        if "power" not in can:
            raise ValueError("field 'power' must be present")
    except Exception as ex:
        return str(ex)
    return "" #no errors
```

The validation function above manipulates externally supplied data. This is always a dangerous process. Well built microservices should not

crash in the face of errors, rather they should report specific details describing the problem through both the API and the logging system. The `validate_can()` function above converts JSON string data into the appropriate Python data type. The range of acceptable values for each element is also tested.

Logging and monitoring microservices is particularly challenging and important. Challenging due to the great number of services and service replicas. Important because error logging is one of the key forensic tools operations teams can use to solve production problems.

Now modify your `/waste/cans` handler to test all POSTs with the new validation function and provide detailed logging in the face of failure:

```
@app.route('/waste/cans', methods=['GET', 'POST'])
def cans():
    """ /cans collection allows POST of new cans and GET of all cans """
    if request.method == 'POST':
        can = request.get_json()
        result = validate_can(can)
        if result:
            print("ERROR: " + request.url + " : " + result)
            return jsonify(error=result), 400
        wcans[can["id"]] = can
        return jsonify(can), 201
    else:  #GET
        return jsonify(list(wcans.values()))
```

The updated `cans()` function now validates all POST data and logs error data to the user through HTTP and to the console via `print()`. As it turns out, Flask also reports all returned status codes for a given IRI.

Your program should now look something like this:

```
(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)

wcans = {}

@app.route('/waste/cans', methods=['GET', 'POST'])
def cans():
    """ /cans collection allows POST of new cans and GET of all cans """
    if request.method == 'POST':
        can = request.get_json()
        result = validate_can(can)
        if result:
            print("ERROR: " + request.url + " : " + result)
            return jsonify(error=result), 400
        wcans[can["id"]] = can
        return jsonify(can), 201
    else:  #GET
        return jsonify(list(wcans.values()))

def validate_can(can):
    """ DbC checks for required can fields and settings
            returns False if can is valid
            returns a string describing the error otherwise
    """
    try:
        #Test id
        can["id"] = int(can["id"])
        if can["id"] < 0 or 999999999 < can["id"]:
            raise ValueError("can.id out of range [0..999999999]")
        #Test deployed
        if can["deployed"] == "True":
```

```
            can["deployed"] = True
        elif can["deployed"] == "False":
            can["deployed"] = False
        else:
            raise ValueError("can.deployed must be Boolean (True, False)")
        #Test capacity
        can["capacity"] = float(can["capacity"])
        if can["capacity"] <= 0.0 or 9999 < can["capacity"]:
            raise ValueError("can.capacity out of range (0.0..9999.0]")
        #Test lat
        can["lat"] = float(can["lat"])
        if can["lat"] < -90.0 or 90.0 < can["lat"]:
            raise ValueError("can.lat out of range [-90.0..90.0]")
        #Test lon
        can["lon"] = float(can["lon"])
        if can["lon"] < -180.0 or 180.0 < can["lon"]:
            raise ValueError("can.lon out of range [-180.0..180.0]")
        #Test power
        if "power" not in can:
            raise ValueError("field 'power' must be present")
    except Exception as ex:
        return str(ex)
    return "" #no errors
```

Test the new validator by attempting to POST invalid can data.

```
(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)
```

Now try to POST a trash can with a capacity of 0:

```
user@ubuntu:~/trash-can/inv$ curl -sv -H "Content-Type: application/json" -d '{"id":"57","deployed": "True","power":
"solar","lat": "31.776","lon": "35.217","capacity": "0"}' http://localhost:8080/waste/cans

*   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)
> POST /waste/cans HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 95
>
* upload completely sent off: 95 out of 95 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 400 BAD REQUEST
< Content-Type: application/json
< Content-Length: 57
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:34:40 GMT
<
{
  "error": "can.capacity out of range (0.0..9999.0]"
}
* Closing connection 0

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

This fails with a 400 status and a helpful error message.

Examine the Flask console output:

```
(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)

ERROR: http://localhost:8080/waste/cans : can.capacity out of range (0.0..9999.0]
127.0.0.1 - - [07/Jan/2018 03:34:40] "POST /waste/cans HTTP/1.1" 400 -
```

We have a clear "ERROR" statement in the log (easy to search for with analytics tools) and the full IRI of the resource along with the application specific error.

4. Path Parameters and Individual Resource Retrieval

As a final step we want to make sure that clients can retrieve an individual trash can and also remove them. We can use a path parameter to select a specific can from the cans resource collection. For example the route `/waste/cans/42` would be used to refer to the can with the ID 42.

Add the following code to your inventory microservice to complete its API feature set:

```
@app.route('/waste/cans/<int:can_id>', methods=['GET', 'DELETE'])
def can(can_id):
    """ can id can be used as a /cans path param to GET/DELETE a single can """
    if can_id not in wcans:
        return jsonify(error="trash can id not found"), 404
    if request.method == 'GET':
        return jsonify(wcans[can_id])
    elif request.method == 'DELETE':
        del wcans[can_id]
        return ('', 204)
    else:
        return jsonify(error="bad HTTP verb, only GET and DELETE supported"), 400
```

This function accepts an integer path parameter with either the GET or DELETE verb. The first if condition ensures that a valid ID has been presented. The second if condition returns the appropriate can in response to a GET operation. The associated *elif* handles the DELETE verb, all other verbs produce an error.

Run the updated service:

```
(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)
```

To test the new functionality we can add a couple of cans and then try to display them individually, add the first can:

```
user@ubuntu:~/trash-can/inv$ curl -sv -H "Content-Type: application/json" -d '{"id":"42","deployed":
"False","power": "grid","lat": "31.776","lon": "35.217","capacity": "500"}' http://localhost:8080/waste/cans

*   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)
> POST /waste/cans HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 97
>
* upload completely sent off: 97 out of 97 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 115
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:35:53 GMT
<
{
  "capacity": 500.0,
  "deployed": false,
  "id": 42,
  "lat": 31.776,
  "lon": 35.217,
  "power": "grid"
}
* Closing connection 0

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

... and another (notice the id differs before you do it) ...

```
user@ubuntu:~/trash-can/inv$ curl -sv -H "Content-Type: application/json" -d '{"id":"57","deployed": "True","power":
```

```
"solar","lat": "31.776","lon": "35.217","capacity": "100"}' http://localhost:8080/waste/cans

*   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)
> POST /waste/cans HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 97
>
* upload completely sent off: 97 out of 97 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 115
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:37:00 GMT
<
{
  "capacity": 100.0,
  "deployed": true,
  "id": 57,
  "lat": 31.776,
  "lon": 35.217,
  "power": "solar"
}
* Closing connection 0

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

Now retrieve a specific can:

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans/42

*   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 /waste/cans/42 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: 115
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:37:53 GMT
<
{
  "capacity": 500.0,
  "deployed": false,
  "id": 42,
  "lat": 31.776,
  "lon": 35.217,
  "power": "grid"
}
* Closing connection 0

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

Try displaying all of the *cans* in the inventory:

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans

*   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 /waste/cans 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: 268
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:38:14 GMT
<
[
  {
    "capacity": 100.0,
    "deployed": true,
    "id": 57,
    "lat": 31.776,
    "lon": 35.217,
    "power": "solar"
  },
  {
    "capacity": 500.0,
    "deployed": false,
    "id": 42,
    "lat": 31.776,
    "lon": 35.217,
    "power": "grid"
  }
]
* Closing connection 0

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

Finally test the DELETE operation:

```
user@ubuntu:~/trash-can/inv$ curl -sv -X DELETE localhost:8080/waste/cans/42

*   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)
> DELETE /waste/cans/42 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 204 NO CONTENT
< Content-Type: text/html; charset=utf-8
< Content-Length: 0
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:38:42 GMT
<
* Closing connection 0

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

Now list all cans:

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans

*   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 /waste/cans 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: 135
```

```
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:39:15 GMT
<
[
  {
    "capacity": 100.0,
    "deployed": true,
    "id": 57,
    "lat": 31.776,
    "lon": 35.217,
    "power": "solar"
  }
]
* Closing connection 0

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

Our can is gone, try to retrieve it discretely:

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans/42

*   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 /waste/cans/42 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 404 NOT FOUND
< Content-Type: application/json
< Content-Length: 40
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:39:39 GMT
<
{
  "error": "trash can id not found"
}
* Closing connection 0

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

As expected the DELETE verb removes the unwanted can. Your service console log should look something like this:

```
(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 03:35:53] "POST /waste/cans HTTP/1.1" 201 -
127.0.0.1 - - [07/Jan/2018 03:37:00] "POST /waste/cans HTTP/1.1" 201 -
127.0.0.1 - - [07/Jan/2018 03:37:53] "GET /waste/cans/42 HTTP/1.1" 200 -
127.0.0.1 - - [07/Jan/2018 03:38:14] "GET /waste/cans HTTP/1.1" 200 -
127.0.0.1 - - [07/Jan/2018 03:38:42] "DELETE /waste/cans/42 HTTP/1.1" 204 -
127.0.0.1 - - [07/Jan/2018 03:39:15] "GET /waste/cans HTTP/1.1" 200 -
127.0.0.1 - - [07/Jan/2018 03:39:39] "GET /waste/cans/42 HTTP/1.1" 404 -
```

The log makes it easy to see the state changes made to our inventory.

5. HATEOS

HATEOAS (Hypermedia as the Engine of Application State) is an attribute of the Uniform Interface constraint of the REST architectural style. A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses.

At present our responses are all data. The responses that might currently benefit from HATEOS are the GET and POST operations on the `/waste/cans` IRI. In the GET data it might be nice if each resource in the list included its own IRI. You might say that the IRI is well known, `/waste/cans/{can_id}`. Well it is certainly well known to us, but what about our users? What if we decide to move the resources and change the IRI? In the simple example we have here, this may be a little far fetched but it is a common occurrence in REST APIs. Whenever possible we should avoid explicit resource paths in lieu of discovered resource paths. This gives us the freedom to change and move IRIs as our architecture evolves.

Another opportunity for HATEOS in our API is the POST operation. In our prototype, the POST object includes the can ID. What if, as we move to

production, it becomes the service's responsibility to allocate IDs. How will the caller know what ID was assigned to their object? It could remain in the JSON returned but it turns out that POST operations creating a new resource are typically obliged to return a Location header, identifying the path of the new resource.

Modify the POST code in your service to return an HTTP location header with the path of newly created resource:

```
@app.route('/waste/cans', methods=['GET', 'POST'])
def cans():
    """ /cans collection allows POST of new cans and GET of all cans """
    if request.method == 'POST':
        can = request.get_json()
        result = validate_can(can)
        if result:
            print("ERROR: " + request.url + " : " + result)
            return jsonify(error=result), 400
        wcans[can["id"]] = can
        response = jsonify(can)
        response.status_code = 201
        response.headers['Location'] = "/waste/cans/" + str(can["id"])
        response.autocorrect_location_header = False
        return response
    elif request.method == 'GET':
        return jsonify(list(wcans.values()))
    else:
        return jsonify(error="bad HTTP verb, only GET and POST supported"), 400
```

The `response.autocorrect_location_header = false` statement keeps Flask from changing the relative path set in our location header. Now rerun your service to test it:

```
(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)
```

Now try POSTing a new can:

```
user@ubuntu:~/trash-can/inv$ curl -sv -H "Content-Type: application/json" -d \
> '{"id":"42","deployed": "False","power": "grid","lat": "31.776","lon": "35.217","capacity": "500"}' \
> http://localhost:8080/waste/cans

*   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)
> POST /waste/cans HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 97
>
* upload completely sent off: 97 out of 97 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Location: /waste/cans/42
< Content-Length: 115
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:43:09 GMT
<
{
  "capacity": 500.0,
  "deployed": false,
  "id": 42,
  "lat": 31.776,
  "lon": 35.217,
  "power": "grid"
}
* Closing connection 0

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

We now can use the Location header, in this case with the value `/waste/cans/42`, to access the resource we just created.

Try it:

```
user@ubuntu:~/trash-can/inv$ curl -sv localhost:8080/waste/cans/42

*   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 /waste/cans/42 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: 115
< Server: Werkzeug/0.14.1 Python/2.7.12
< Date: Sun, 07 Jan 2018 11:43:46 GMT
<
{
  "capacity": 500.0,
  "deployed": false,
  "id": 42,
  "lat": 31.776,
  "lon": 35.217,
  "power": "grid"
}
* Closing connection 0

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

As advertised, the location header now points to the newly created resource. For more on designing effective REST APIs see the RX-M REST course offerings. In the labs ahead we'll learn how to package our services in containers, how to work with multiple services, messages and more.

Lets save our changes.

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

(venv) user@ubuntu:~/trash-can/inv$ git commit -m "adding additional CRUD and REST features"

[master d36f2b1] adding additional CRUD and REST features
 1 file changed, 71 insertions(+)

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

Congratulation you have successfully completed the lab!

Complete inv.py Listing

```
""" 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)


wcans = {}

@app.route('/waste/cans', methods=['GET', 'POST'])
def cans():
```

```
    """ /cans collection allows POST of new cans and GET of all cans """
    if request.method == 'POST':
        can = request.get_json()
        result = validate_can(can)
        if result:
            print("ERROR: " + request.url + " : " + result)
            return jsonify(error=result), 400
        wcans[can["id"]] = can
        response = jsonify(can)
        response.status_code = 201
        response.headers['Location'] = "/waste/cans/" + str(can["id"])
        response.autocorrect_location_header = False
        return response
    elif request.method == 'GET':
        return jsonify(list(wcans.values()))
    else:
        return jsonify(error="bad HTTP verb, only GET and POST supported"), 400

@app.route('/waste/cans/<int:can_id>', methods=['GET', 'DELETE'])
def can(can_id):
    """ can id can be used as a /cans path param to GET/DELETE a single can """
    if can_id not in wcans:
        return jsonify(error="trash can id not found"), 404
    if request.method == 'GET':
        return jsonify(wcans[can_id])
    elif request.method == 'DELETE':
        del wcans[can_id]
        return ('', 204)
    else:
        return jsonify(error="bad HTTP verb, only GET and DELETE supported"), 400

def validate_can(can):
    """ DbC checks for required can fields and settings
            returns False if can is valid
            returns a string describing the error otherwise
    """
    try:
        #Test id
        can["id"] = int(can["id"])
        if can["id"] < 0 or 999999999 < can["id"]:
            raise ValueError("can.id out of range [0..999999999]")
        #Test deployed
        if can["deployed"] == "True":
            can["deployed"] = True
        elif can["deployed"] == "False":
            can["deployed"] = False
        else:
            raise ValueError("can.deployed must be Boolean (True, False)")
        #Test capacity
        can["capacity"] = float(can["capacity"])
        if can["capacity"] <= 0.0 or 9999 < can["capacity"]:
            raise ValueError("can.capacity out of range (0.0..9999.0]")
        #Test lat
        can["lat"] = float(can["lat"])
        if can["lat"] < -90.0 or 90.0 < can["lat"]:
            raise ValueError("can.lat out of range [-90.0..90.0]")
        #Test lon
        can["lon"] = float(can["lon"])
        if can["lon"] < -180.0 or 180.0 < can["lon"]:
            raise ValueError("can.lon out of range [-180.0..180.0]")
        #Test power
        if "power" not in can:
            raise ValueError("field 'power' must be present")
    except Exception as ex:
        return str(ex)
    return "" #no errors
```