RESTful Web Services With Python Flask

by Saravanan Subramanian MVB · Mar. 31, 16 · Integration Zone

The idea of this post is to describe how to develop a RESTful Web Services in Python.

RESTful Web Service is an architectural style, where the data or the structural components of a system is described in the form of URI (Uniform Resource Identifier) and the behaviors are described in terms of methods. The resources can be manipulated using CRUD (Create, Read, Update and Delete) operations. The communication protocol for REST is HTTP since it suits the architecture requirement of being a stateless communication across the Client and Server.

There are many frameworks available in Python for web development, but Django (pronounced as yango) and Flask stands out of the crowd being a full stack frameworks. I prefer Flask framework since it is very small and easy to learn for beginners, whereas Django is too big for beginners.

1. The Plan

In this exercise, we will create an in-memory JSON DB to store and manipulate a simple employee database and develop RESTful APIs to perform CRUD operations using GET, POST, PUT, and DELETE methods. We will develop the below APIs

i) **GET /empdb/employee/** - Retrieve all the employees from the DB

ii) **GET /empdb/employee/** - Retrieve the details of given employee Id

ii) **POST /empdb/employee/** - Create a record in the employee DB, whereas the employee details are sent in the request as a JSON object

III) **PUT /empdb/employee/** - Update the employee DB, with the given details of employee in the data part as a JSON object

Iv) **DELETE /empdb/employee/** - Delete the employee from the DB for the employee Id.

2. Installation of Flask

To install flask framework, please refer the official website [1]. If you have pip installed in your Python environment, please follow this step.

```
$ pip install Flask
```

If you don't have pip, please download the flask from http://pypi.python.org/packages/source/F/Flask/Flask-0.10.1.tar.gz and execute the setup.py

3. Hello World - Web Server

First, we create a web server, create a dictionary to hold a JSON objects for a couple of employee records and then we add

RESTful APIs for each supported operations. Please look at the below program, which creates a web server. Save the below program into hello.py and execute it.

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

The below line of the code creates an app object from Flask.

```
app = Flask(__name__)
```

app.run() starts the web server and ready to handle the request. But at this moment, it can handle only one request. It is defined in the below line of code.

```
@app.route("/")
def hello():
    return "Hello World !"
```

Execute the above program and you will see that you web server is ready to service you.

```
* Running on http://localhost:5000/
```

Now you can open your web browser and check your web server. The server is available in the URL **http://localhost:5000/.** If you are familiar with cUrl execute the below to check the status.

```
$ curl -i http://localhost:5000/
```

4. Develop the RESTful Services

To develop the restful services for the planned objective, let's create an in-memory database in python using the dictionary data type. Please find the code snippet below: We can continue to use the ***hello.py*** and type the below code, just after the Flask app creation statement ***app = Flask(__name__)***. You can also refer the below **section 5** for the complete code.

```
empDB=[
 {
 'id':'101',
 'name':'Saravanan S',
 'title':'Technical Leader'
 },
 {
 'id':'201',
 'name':'Rajkumar P',
 'title':'Sr Software Engineer'
 }
 ]
```

4.1 GET

In the previous section, we have created two employees in the dictionary. Now let's write a code to retrieve them using web services. As per our plan, we need two implementations one is to retrieve all the employees and another one to retrieve the specific employee with the given id.

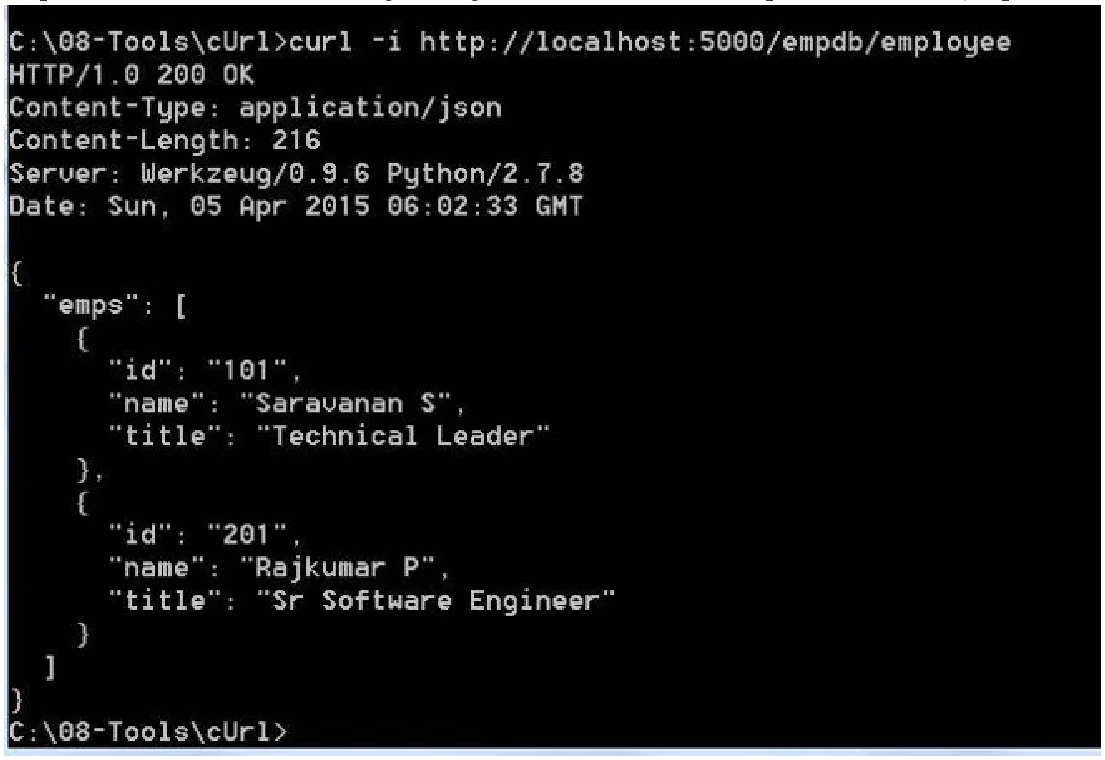
4.1.1 GET All

```
@app.route('/empdb/employee',methods=['GET'])
def getAllEmp():
    return jsonify({'emps':empDB})
```

In the above code snippet, we have created a URI named **'/empdb/employee'** and also we defined the method as **"GET"**. To service the GET call for the URI, Flask will call the function **getAllEmp()**. It will in turn simply call the **"jsonify"** method with ***employeeDB*** as the argument. The **"jsonify"** is a flask method, will set the data with the given JSON object which is passed as a Python dictionary and set the headers appropriately, in this case **"Content-type: application/json"**. We can check the above Web Service with cUrl as below:

```
cUrl> curl -i http://localhost:5000/empdb/employee
```

[caption id="attachment_91" align="alignleft" width="660"] Response for CURL[/caption]



```
C:\08-Tools\cUrl>curl -i http://localhost:5000/empdb/employee
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 216
Server: Werkzeug/0.9.6 Python/2.7.8
Date: Sun, 05 Apr 2015 06:02:33 GMT

{
  "emps": [
    {
      "id": "101",
      "name": "Saravanan S",
      "title": "Technical Leader"
    },
    {
      "id": "201",
      "name": "Rajkumar P",
      "title": "Sr Software Engineer"
    }
  ]
}
C:\08-Tools\cUrl>
```

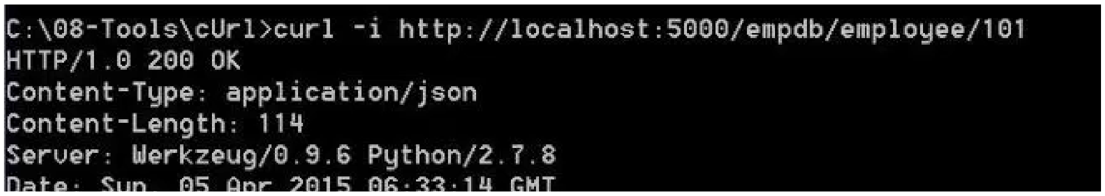
4.1.2 Get Specific

Now we develop the rest service to get an employee with a given id.

```
@app.route('/empdb/employee/<empId>',methods=['GET'])
def getEmp(empId):
    usr = [ emp for emp in empDB if (emp['id'] == empId) ]
    return jsonify({'emp':usr})
```

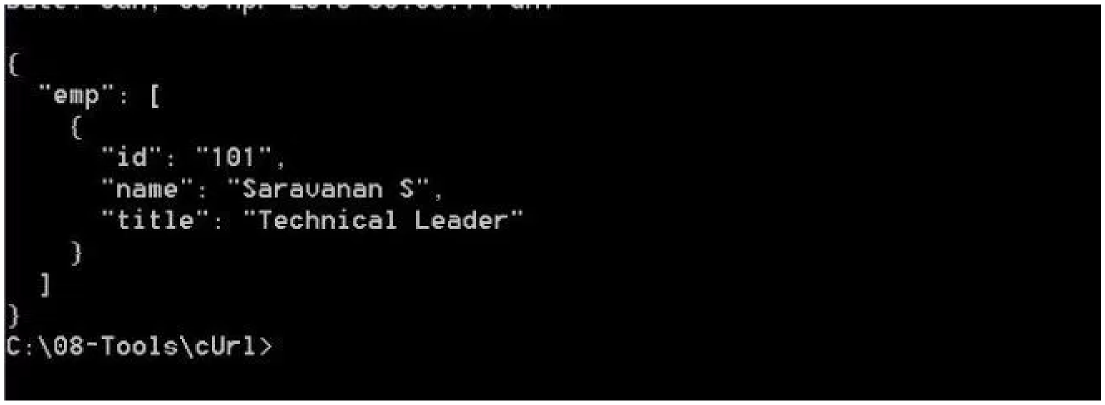
The above code will find the employee object with the given id and send the JSON object in the data. Here I have used the list comprehension technique in Python if you don't understand you can simply write in an imperative way of processing the entire dictionary using a for loop.

```
CURL > curl -i http://localhost:5000/empdb/employee/101
```



```
C:\08-Tools\cUrl>curl -i http://localhost:5000/empdb/employee/101
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 114
Server: Werkzeug/0.9.6 Python/2.7.8
Date: Sun, 05 Apr 2015 06:33:14 GMT
```

The response would be :



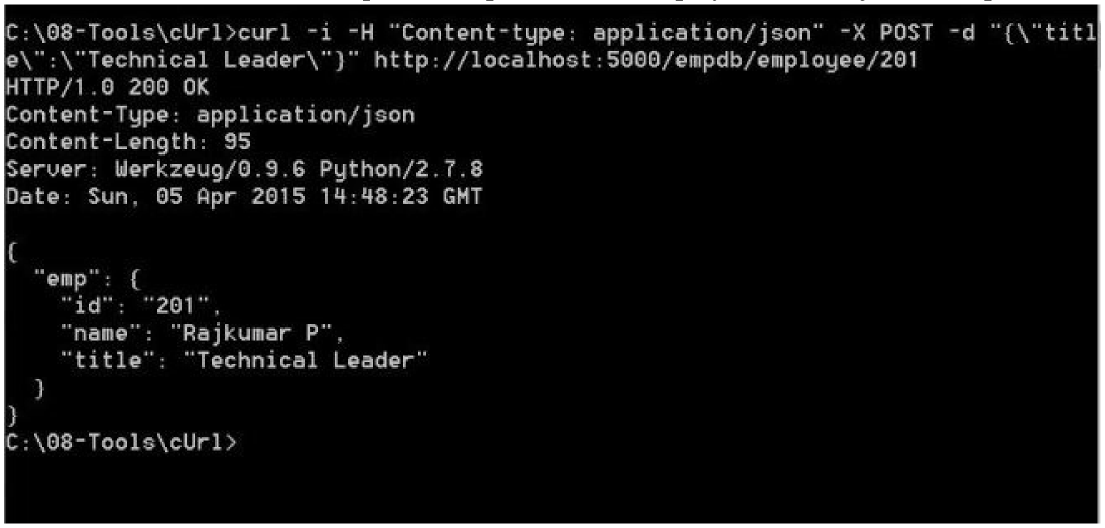
```
{
  "emp": [
    {
      "id": "101",
      "name": "Saravanan S",
      "title": "Technical Leader"
    }
  ]
}
C:\08-Tools\cUrl>
```

4.2 PUT

PUT method is used to update the existing resource. The below code gets the employee id from the URL and finds the respective object. It checks the **request.json** from the request for the new data & then it overwrites the existing. NOTE: the **request.json** will contain the JSON object set in the client request.

```
@app.route('/empdb/employee/<empId>',methods=['PUT'])
def updateEmp(empId):
    em = [ emp for emp in empDB if (emp['id'] == empId) ]
    if 'name' in request.json :
        em[0]['name'] = request.json['name']
    if 'title' in request.json:
        em[0]['title'] = request.json['title']
 return jsonify({'emp':em[0]})
```

We can also use a Postman client or cUrl to update an existing employee. The data must contain the JSON object either with a name or title.The service can be invoked as follows in cUrl. Here we update the **"title"** for employee id 201 with **"Technical Leader"**. The request is responded with employee JSON object with updated values. It also updates the employee DB.



```
C:\08-Tools\cUrl>curl -i -H "Content-type: application/json" -X POST -d "{\"titl
e\":\"Technical Leader\"}" http://localhost:5000/empdb/employee/201
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 95
Server: Werkzeug/0.9.6 Python/2.7.8
Date: Sun, 05 Apr 2015 14:48:23 GMT

{
  "emp": {
    "id": "201",
    "name": "Rajkumar P",
    "title": "Technical Leader"
  }
}
C:\08-Tools\cUrl>
```

4.3 POST

POST method is used to create a new employee inside the database. The code snippet is below:

```
@app.route('/empdb/employee',methods=['POST'])
def createEmp():
    dat = {
    'id':request.json['id'],
    'name':request.json['name'],
    'title':request.json['title']
    }
 empDB.append(dat)
 return jsonify(dat)
```

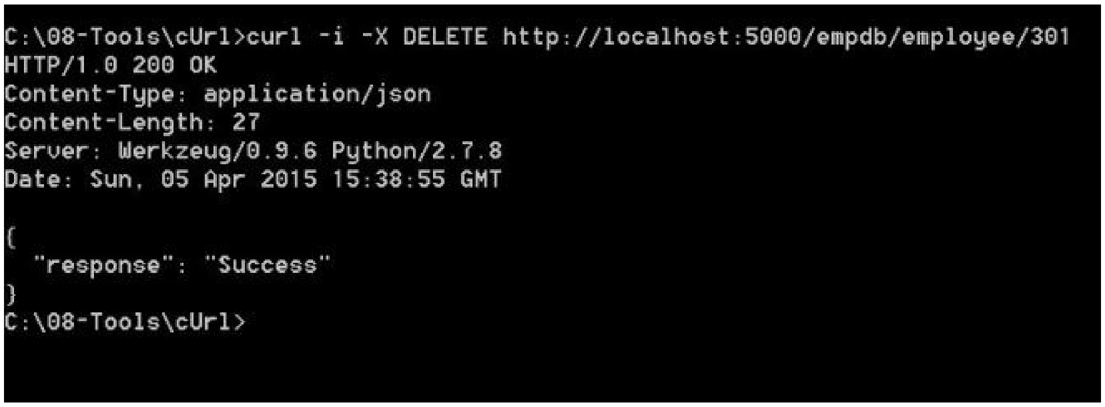
The above code simply reads the request.json for the expected values, and stores them in the local dictionary object and appends it to the employee DB dictionary. This also returns the newly added employee object as the response.

4.4 DELETE

Let's write a code to delete a given employee id.

```
@app.route('/empdb/employee/<empId>',methods=['DELETE'])
def deleteEmp(empId):
    em = [ emp for emp in empDB if (emp['id'] == empId) ]
    if len(em) == 0:
    abort(404)

    empDB.remove(em[0])
    return jsonify({'response':'Success'})
```

the above service can be used as follows:



```
C:\08-Tools\cUrl>curl -i -X DELETE http://localhost:5000/empdb/employee/301
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 27
Server: Werkzeug/0.9.6 Python/2.7.8
Date: Sun, 05 Apr 2015 15:38:55 GMT

{
  "response": "Success"
}
C:\08-Tools\cUrl>
```

5 Complete Code

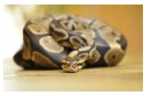
```
from flask import Flask
from flask import jsonify
from flask import request

app = Flask(__name__)

empDB=[
 {
 'id':'101',
 'name':'Saravanan S',
 'title':'Technical Leader'
 },
 {
 'id':'201',
 'name':'Rajkumar P',
 'title':'Sr Software Engineer'
 }
 ]
```

```

@app.route('/empdb/employee',methods=['GET'])
def getAllEmp():
    return jsonify({'emps':empDB})

@app.route('/empdb/employee/<empId>',methods=['GET'])
def getEmp(empId):
    usr = [ emp for emp in empDB if (emp['id'] == empId) ]
    return jsonify({'emp':usr})


@app.route('/empdb/employee/<empId>',methods=['PUT'])
def updateEmp(empId):

    em = [ emp for emp in empDB if (emp['id'] == empId) ]

    if 'name' in request.json :
        em[0]['name'] = request.json['name']

    if 'title' in request.json:
        em[0]['title'] = request.json['title']

    return jsonify({'emp':em[0]})


@app.route('/empdb/employee',methods=['POST'])
def createEmp():

    dat = {
    'id':request.json['id'],
    'name':request.json['name'],
    'title':request.json['title']
    }
    empDB.append(dat)
    return jsonify(dat)

@app.route('/empdb/employee/<empId>',methods=['DELETE'])
def deleteEmp(empId):
    em = [ emp for emp in empDB if (emp['id'] == empId) ]

    if len(em) == 0:
       abort(404)

    empDB.remove(em[0])
    return jsonify({'response':'Success'})

if __name__ == '__main__':
 app.run()
```

6 Conclusion

This is a very basic web services we have developed. I hope this helps to understand basics of RESTful Web Services development. We can make this implementation clean by proper error handling and authentication. I suggest to everyone to visit the official documentation of Flask for further learning.

Like This Article? Read More From DZone


Optimizing Flask Client Tests

Writing a Web Service Using Python Flask

Triggering API Services With Zato Scheduler: Part I

Free DZone Refcard
Getting Started With Spring Boot and Microservices

Topics: PYTHON , REST API , FLASK , INTEGRATION

Opinions expressed by DZone contributors are their own.

Integration Partner Resources

Apache Kafka + Kafka Streams + Mesos / DCOS = Scalable Microservices

by Kai Wähner MVB · Nov 06, 17 · Integration Zone

I had a talk at MesosCon 2017 Europe in Prague about building highly scalable, mission-critical microservices with Apache Kafka, Kafka Streams and Apache Mesos / DCOS. I would like to share the slides and a video recording of the live demo.

Abstract

Microservices establish many benefits like agile, flexible development and deployment of business logic. However, a

Microservices establish many benefits like agile, flexible development and deployment of business logic. However, a Microservice architecture also creates many new challenges. This includes increased communication between distributed instances, the need for orchestration, new fail-over requirements, and resiliency design patterns.

This session discusses **how to build a highly scalable, performant, mission-critical microservice infrastructure with Apache Kafka, Kafka Streams and Apache Mesos respectively DC/OS**. Apache Kafka brokers are used as powerful, scalable, distributed message backbone. Kafka's Streams API allows embedding of stream processing directly into any external microservice or business application. Without the need for a dedicated streaming cluster. Apache Mesos can be used as scalable infrastructure for both, the Apache Kafka brokers and external applications using the Kafka Streams API, to leverage the benefits of a cloud-native platforms like service discovery, health checks, or fail-over management.

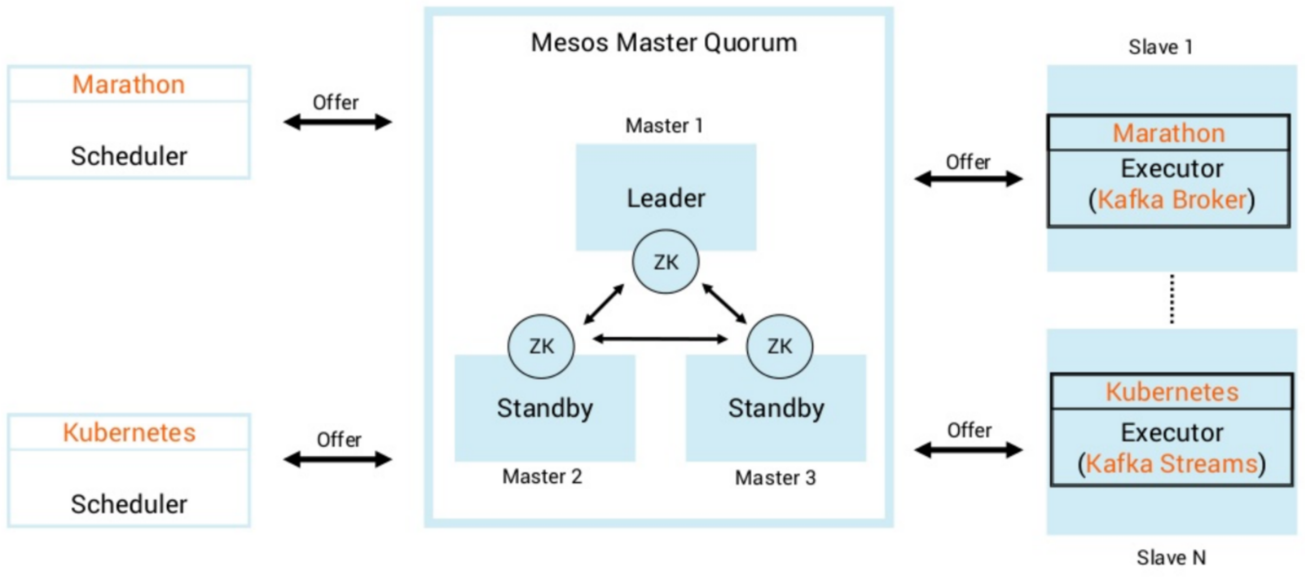
A live demo shows how to develop real-time applications for your core business with Kafka messaging brokers and Kafka Streams API. You see how to deploy/manage/scale them on a DC/OS cluster using different deployment options.

Key Takeaways

- Successful microservice architectures require a highly scalable messaging infrastructure combined with a cloud-native platform which manages distributed microservices
- Apache Kafka offers a highly scalable, mission-critical infrastructure for distributed messaging and integration
- Kafka's Streams API allows embedding of stream processing into any external application or microservice
- Mesos respectively DC/OS allow management of both, Kafka brokers and external applications using Kafka Streams API, to leverage many built-in benefits like health checks, service discovery or fail-over control of microservices
- See a live demo which combines the Apache Kafka streaming platform and DC/OS

Architecture: Kafka Brokers + Kafka Streams on Kubernetes and DC/OS

The following picture shows the architecture. You can either run Kafka Brokers and Kafka Streams microservices natively on DC/OS via Marathon or leverage Kubernetes as a Docker container orchestration tool (which is also supported by Mesosphere in the meantime).



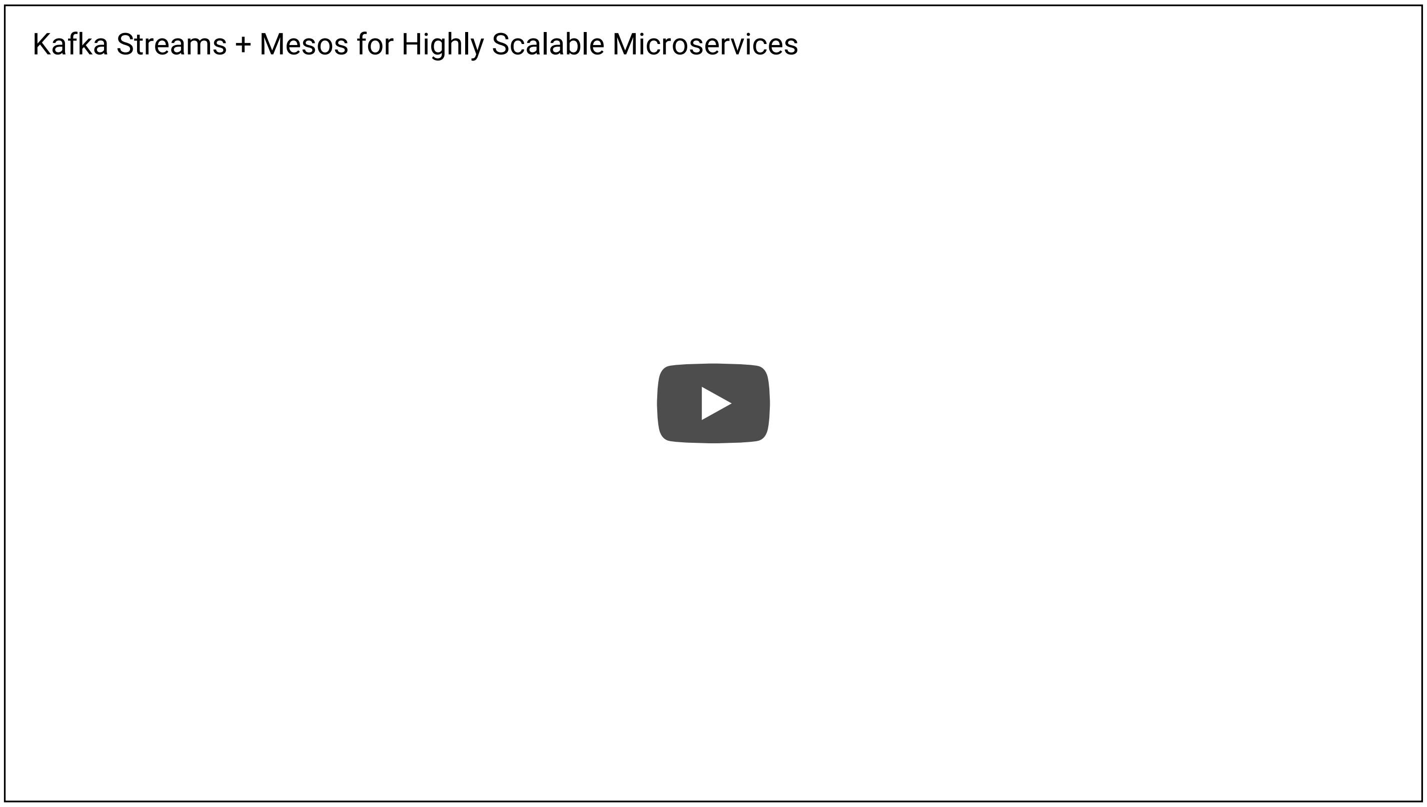
Slides

Here are the slides from my talk.

Live Demo

Live Demo

The following video shows the live demo. It is built on AWS using Mesosphere's Cloud Formation script to set up a DC/OS cluster in ten minutes.


Kafka Streams + Mesos for Highly Scalable Microservices

Here, I deployed both - Kafka brokers and Kafka Streams microservices - directly to DC/OS without leveraging Kubernetes. I expect to see many people continue to deploy Kafka brokers directly on DC/OS. For microservices many teams might move to the following stack: Microservice --> Docker --> Kubernetes --> DC/OS.

Do you also use Apache Mesos respectively DC/OS to run Kafka? Only the brokers or also Kafka clients (producers, consumers, Streams, Connect, KSQL, etc)? Or do you prefer another tool like Kubernetes (maybe on DC/OS)?

Like This Article? Read More From DZone

Transforming and Aggregating Kafka Messages With Kafka Streams

An Introduction to Kafka Streams

Streaming in Spark, Flink, and Kafka

Free DZone Refcard
Getting Started With Spring Boot and Microservices

Topics: KAFKA, KAFKA STREAMS, MESOS, DCOS, STREAM PROCESSING, KUBERNETES, CONFLUENT, MICROSERVICES, INTEGRATION