Bookshelf - An event-driven microservice showcase Documentation

Release 0.2

Sylvain Hellegouarch

Oct 02, 2017

Contents

1 Get Started 5
1.1 Local deployment 5
1.2 Kubernetes deployment 6
1.3 Mesos/Marathon deployment 6

2 Usage 9
2.1 Service Discovery 9
2.2 REST API 10
2.3 Testing 12

HTTP Routing Table 13

Author Sylvain Hellegouarch

Release 0.2

License BSD

Source code https://github.com/Lawouach/event-driven-microservice

Build status https://travis-ci.org/Lawouach/event-driven-microservice

The event-driven microservice showcase demonstrates an implementation of ideas developed by Russ Miles in his Antifragile Software book.

Note: The code presented here is not a framework nor a library to be re-used as-is.

Russ describes an architecture that is best supported by microservices. The objective is to design your application so that it embraces change instead of ignoring or fighting it.

To achieve this, the book introduces the following kinds of microservices:

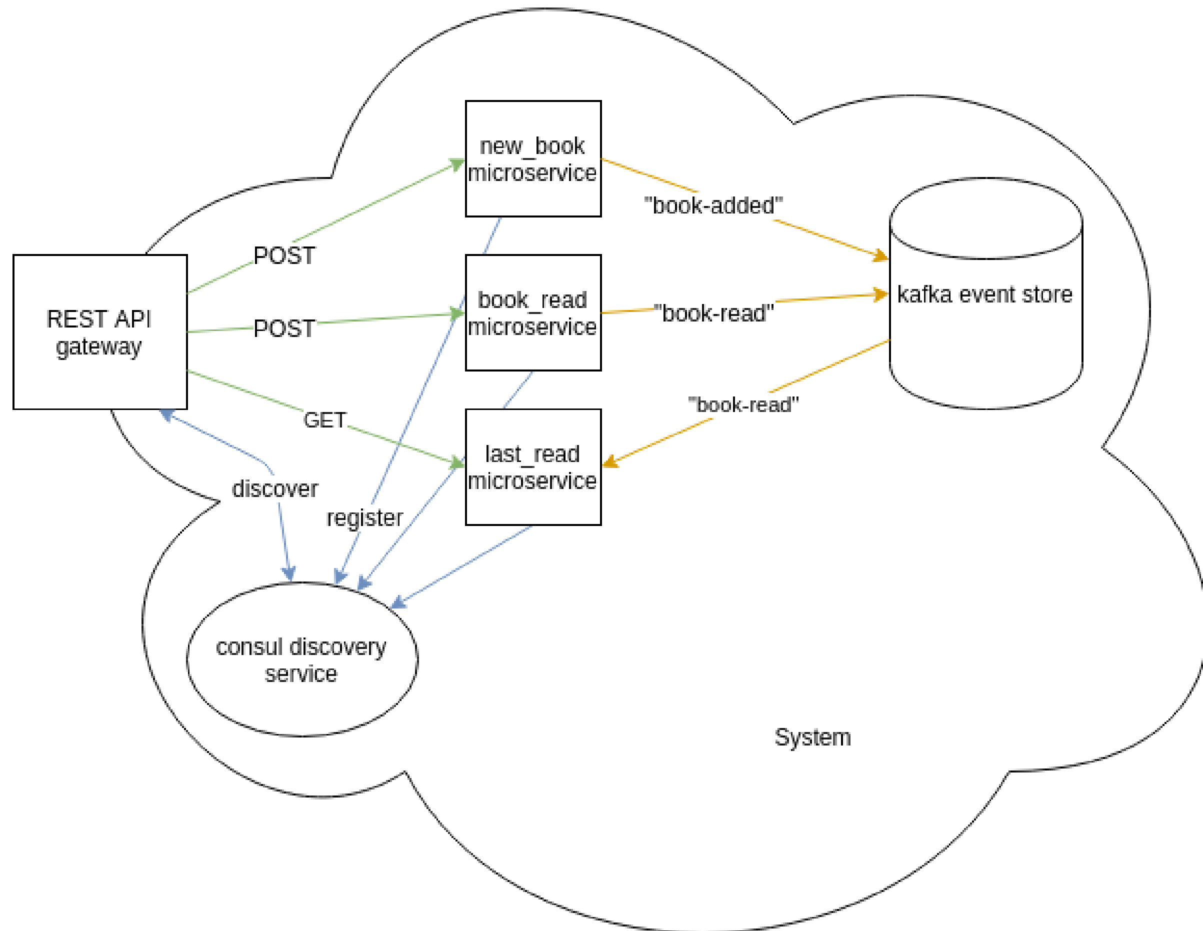
- views: a view is meant to answer read-only queries
- aggregates: an aggregate handles update operations

By making a difference between the two (the rough idea behind CQRS), we can support a different set of requirements for query and command operations.

In this repository, we have a set of aggregates and views. They expose a HTTP interface but also consume or generate events carried by the Kafka broker.

From an external point of view however, it's best if we expose a simple REST HTTP interface. The code therefore provides what Russ calls a gateway. They are services that permit communication with a response from the system. This means, that external clients should go through the gateway to interact with the system. In our implementation, the gateway is a simple reverse proxy that forwards calls to the appropriate internal aggregate or view.

The following diagram represents the general architecture:



The nice aspect of event-driven architecture is it supports a clean decoupling between microservices. To enforce the dynamic nature of a microservice architecture design for change, the service discovery also supports that decoupling idea, since microservices don't have to know each other's location. They know how to ask the discovery service location based on a set of rich criteria.

Why Python 3.5?

These examples rely on Python 3.5 because the new async features brought by Python 3.4 and consolidated in 3.5.1 fit extraordinaly well a function-driven code design.

The language also supports type hints that these examples don't yet benefit from but will in the future to discover services.

Why Kafka?

Kafka is a brilliant platform to store events. It's fast, scalable and flexible. There are plenty of clients out there for it too.

There are alternatives to flow events across the board.

Why Consul?

Consul is a nifty tool that has an extensive featureset while being easy to setup and a small footprint. It supports service

discovery via DNS and HTTP which makes it very powerful for various kinds of service discovery. Indeed, a DNS record may be present when a microservice has been started, but it doesn't mean the service is ready per-se. Using the HTTP interface to register said service only when ready, means other services can be sure they can start using it.

CHAPTER 1

Get Started

Local deployment

To run the system on your local machine you will need the following properly installed:

- Docker 1.10+
- Docker Compose 1.6+

Once they are installed, you can get the code:

```
$ git clone https://github.com/Lawouach/event-driven-microservice.git
```

First, build the appropriate image:

```
$ docker build -t bookshelf:0.2 .
```

You can now run the cluster with a simple call to:

```
$ docker-compose up
```

After a few seconds this will have started:

- consul (1 node)
- zookeeper (1 node)
- kafka (1 node)
- the bookshelf microservices

To stop, simply hit *Ctrl-C* in the same console or run *docker-compose down* from a different terminal.

When all the services are running, you can access:

- the gateway on http://<host>:8000
- the *last_read* microservice on port http://<host>:8080

- the *new_book* microservice on port http://<host>:8081
- the *book_read* microservice on port http://<host>:8082
- the service discovery service is available on <host>:8500 via HTTP and localhost:8600 via DNS
- the kafka broker is available via <host>:9092

Replace *<host>* with the hostname or address where your containers are running. On a Linux box, this is likely *localhost*. If you are using a virtual machine to run your docker containers (like through boot2docker, you may retrieve the IP through *boot2docker ip*.

Kubernetes deployment

A different kind of deployment and management of the cluster is through Kubernetes. Kubernetes is a service orchestration and management toolkit that makes it simple to run and scale microservices on premises or on public cloud.

This assumes you have installed Kubernetes according to your needs.

Make sure you also have the *kubectl* command in your *PATH* on your local machine. Simply download Kubernetes, unpack it and set your *PATH* pointing to the *kubernetes/cluster* directory.

Then run the following command:

```
$ kubctl create -f kube.yaml
```

This will create four different replication controllers:

- one to manage Zookeeper with an internal kube service called *zoo*
- one to manage Kafka with an internal kube service called *events*
- one to manage Consul with an internal kube service called *disco*
- one to manage the Bookshelf showcase with a public kube service called *bookshelf*

If you are running this on AWS, this will result into a AWS load-balancer to be created with port *8000* exposed. This will be used to access the bookshelf REST API exposed by the gateway.

Mesos/Marathon deployment

Marathon is a service orchestration and management tool, much like Kubernetes, that allows you to run and scale your microservices accross datacenters.

To make it to try it out, the repository provides a simple set of provisioning scripts that will deploy a single Mesos/Marathon node locally in a VirtualBox virtual machine.

You will need:

- VirtualBox
- Vagrant

Once installed, run the following command:

```
$ vagrant up
```

This will create a single virtual machine with 1 CPU, 3Gb RAM and 40Gb disk usage.

Once the process is finished, you will be able to access:

- the mesos dashboard
- the marathon dashboard
- the consul dashboard

To execute your microservices, run the following commands:

```
$ curl -X POST -H "Content-Type: application/json" --data @marathon/mesos-consul.json␣
↪http://localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/zookeeper.json␣
↪http://localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/kafka.json http://
↪localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/newbook-
↪microservice.json http://localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/readbook-
↪microservice.json http://localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/lastread-
↪microservice.json http://localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/listbooks-
↪microservice.json http://localhost:8079/v2/apps
$ curl -X POST -H "Content-Type: application/json" --data @marathon/api-gateway.json␣
↪http://localhost:8079/v2/apps
```

You may want to give 10 seconds between each call so that each service had the time to properly start up.

Once all services are running you will see them in the marathon and consul dashboards. You will be able to call the bookshelf API on http://localhost:8080/bookshelf.

Note: In order to declare the services, we rely on Consul with the Mesos-Consul bridge that listen to Mesos events to automatically register microservices managed by marathon to the consul service discovery. Funnily, the mesos-consul service is itself managed by marathon.

The downside is that we can't benefit from the groups features of marathon because the task name will be derived from the complete task identification. So if your app is defined in a group and has the identifier `/microservice/bookshelf/newbook`, it will be registered to the consul service as `microservice-bookshelf-newbook`.

If you can adapt your microservices to support this naming then you should rely on marathon grouping feature.

CHAPTER 2

Usage

Service Discovery

Once the cluster is up and running, all microservices register automatically to Consul. The consul UI can be reached at http://<host>:8500/ui.

You can discover them as follows:

```
$ curl http://<host>:8500/v1/catalog/services
{"consul":[],"lastreadbooks":["books","last"],"newbook":["books","new"]}
```

You can then obviously query Consul to discover more about a specific service:

```
$ curl http://<host>:8500/v1/catalog/service/newbook
[{"Node":"disco","Address":"172.19.0.2","ServiceID":"newbook1","ServiceName":"newbook
↪","ServiceTags":["books","new"],"ServiceAddress":"172.19.0.5","ServicePort":8080,
↪"ServiceEnableTagOverride":false,"CreateIndex":5,"ModifyIndex":5}]
```

Or seen via DNS:

```
$ dig @0.0.0.0 -p 8600 newbook.service.consul SRV

; <<>> DiG 9.9.5-11ubuntu1.2-Ubuntu <<>> @127.0.0.1 -p 8600 newbook.service.consul SRV
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 13670
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;newbook.service.consul.            IN      SRV

;; ANSWER SECTION:
newbook.service.consul.     0       IN      SRV     1 1 8080 disco.node.dc1.consul.
```

```
;; ADDITIONAL SECTION:
disco.node.dc1.consul.      0       IN      A       172.19.0.5

;; Query time: 3 msec
;; SERVER: 127.0.0.1#8600(127.0.0.1)
;; WHEN: Mon Feb 29 14:34:20 CET 2016
;; MSG SIZE  rcvd: 140
```

When terminating the service, the service will automatically deregisters itself.

REST API

All the calls of the REST API are executed against the gateway running on port 8000.

The API is self documented through a Swagger spec and available at http://<host>:8000/apidocs/index.html.

> **Warning:** The API is far from being exhaustive. It will be augmented as new services are implemented. Let's recall this is a learning exercise.

POST /bookshelf/books

Push a new book onto the bookshelf. The book is returned with its internal id set.

Example request:

```
POST /bookshelf/books HTTP/1.1
Host: example.com
Content-Type: application/json


{
   "title": "1984",
   "author": "George Orwell",
   "published": "1949"
}
```

Example response:

```
HTTP/1.1 201 Created
Content-Type: application/json


{
   "id": "45b9fc30-5dfb-4c55-b762-2fc9560305c2",
   "title": "1984",
   "author": "George Orwell",
   "published": "1949"
}
```

GET /bookshelf/books

Retrieve the list of all your books.

Example request:

```
GET /bookshelf/books HTTP/1.1
Host: example.com
```

Example response:

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


[{
   "id": "45b9fc30-5dfb-4c55-b762-2fc9560305c2",
   "title": "1984",
   "author": "George Orwell",
   "published": "1949"
}]
```

POST /bookshelf/books/(id)/finished

Set the book identified by *id* as finished.

Example request:

```
POST /bookshelf/books/45b9fc30-5dfb-4c55-b762-2fc9560305c2/finished HTTP/1.1
Host: example.com
Content-Type: application/json


{
   "id": "45b9fc30-5dfb-4c55-b762-2fc9560305c2",
   "title": "1984",
   "author": "George Orwell",
   "published": "1949"
}
```

Example response:

```
HTTP/1.1 204 No Content
```

GET /bookshelf/books/last/read

Retrieve the list of five last read books.

Example request:

```
GET /bookshelf/books/last/read HTTP/1.1
Host: example.com
```

Example response:

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


[{
   "id": "45b9fc30-5dfb-4c55-b762-2fc9560305c2",
   "title": "1984",
   "author": "George Orwell",
   "published": "1949"
}]
```

Testing

This repository comes with a set of unit tests that exercise the code:

```
$ export PYTHONPATH=$PYTHONPATH:`pwd`
$ py.test --cov=bookshelf --cov-report=html test/
```

You will need to install first:

- pytest
- pytest-asyncio
- pytest-cov
- asynctest

HTTP Routing Table

/bookshelf

GET /bookshelf/books, 10
GET /bookshelf/books/last/read, 11
POST /bookshelf/books, 10
POST /bookshelf/books/(id)/finished, 11