Technology Conversations

Microservices Development with Scala, Spray, MongoDB, Docker and Ansible

This article tries to provide one possible approach to building microservices. We'll use Scala as programming language. API will be RESTful JSON provided by Spray and Akka. MongoDB will be used as database. Once everything is done we'll pack it all into a Docker container. Vagrant with Ansible will take care of our environment and configuration management needs.

We'll do the books service. It should be able to do following:

- List all books
- Retrieve all the information related to a book
- Update an existing book
- Delete an existing book

This article will not try to teach everything one should know about Scala, Spray, Akka, MongoDB, Docker, Vagrant, Ansible, TDD, etc. There is no single article that can do that. The goal is to show the flow and the setup that one might use when developing services. Actually, most of this article is equally relevant for other types of developments. Docker has much broader usage than microservices, Ansible and CM in general can be used for any type of provisioning and Vagrant is very useful for quick creation of virtual machines.

Environment

We'll use Ubuntu as a development server. Easiest way to set up a server is with Vagrant. If you don't have it already, please download and install it. You'll also need Git to clone the repository with the source code. The rest of the article will not require any additional manual installations.

Let's start by cloning the repo.

```
git clone https://github.com/vfarcic/books-service.git
cd books-service
```

Next we'll create an Ubuntu server using Vagrant. The definition is following:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.synced_folder ".", "/vagrant"
  config.vm.provision "shell", path: "bootstrap.sh"
  config.vm.provider "virtualbox" do |v|
    v.memory = 2048
  end
  config.vm.define :dev do |dev|
    dev.vm.provision :shell, inline: 'ansible-playbook /vagrant/ansible/dev.ym
  end
  config.vm.define :prod do |prod|
    prod.vm.provision :shell, inline: 'ansible-playbook /vagrant/ansible/prod.
  end
end
```

We defined the box (OS) to be Ubuntu. Sync folder is /vagrant meaning that everything inside the current directory on the host will be available as the /vagrant directory inside the VM. The rest of things we'll need will be installed using Ansible so we're provisioning our VM with it through the bootstrap.sh script. Finally, this Vagrantfile has two VMs defined: **dev** and **prod**. Each of them will run Ansible that will make sure that everything is installed properly.

Preferable way to work with Ansible is to divide configurations into roles. In our case, there are four roles located in **ansible/roles** directory. One will make sure that Scala and SBT are installed, the other that Docker is up and running, and another one will run the MongoDB container. The last role (books) will be used later to deploy the service we're building to the production VM.

As example, definition of the **mongodb** role is following.

```
- name: Directory is present
  file:
    path=/data/db
    state=directory
  tags: [mongodb]

- name: Container is running
  docker:
    name=mongodb
    image=mongo
    ports=27017:27017
    volumes=/data/db:/data/db
  tags: [mongodb]
```

This should be self-explanatory for those used to work with Docker. The role makes sure that the directory is present and that the mongodb container is running. Playbook ansible/dev.yml is where we tie it all together.

```
- hosts: localhost
  remote_user: vagrant
  sudo: yes
  roles:
    - scala
    - docker
    - mongodb
```

As the previous example, this one should also be self-explanatory. Every time we run this playbook, all tasks from roles scala, docker and mongodb will be executed.

Nice thing about Ansible and Configuration Management in general is that they don't blindly run scripts but are acting only when needed. If you run the provisioning the second time, Ansible will detect that everything is in order and do nothing. On the other hand if, for example, you delete the directory **/data/db**, Ansible will detect that it is absent and create it again.

Let's bring the **dev VM** up! First time it might take a bit of time since Vagrant will need to download the whole Ubuntu distribution, install few packages and download Docker images for MongoDB. Each next run will be much faster.

```
vagrant up dev
vagrant ssh dev
ll /vagrant
```

vagrant up creates a new VM or brings the existing one to life. With **vagrant ssh** we can enter the newly created box. Finally, **ll /vagrant** lists all files within that directory as a proof that all our local files are available inside the VM.

That's it. Our development environment with Scala, SBT and MongoDB container is ready. Now it's time to develop our books service.

Books Service

I love Scala and Akka. Scala is a very powerful language and Akka is my favourite framework for building message driven JVM applications. While it was born from Scala, Akka can be used with Java as well.

Spray is simple yet very powerful toolkit for building REST/HTTP based applications. It's asynchronous, uses Akka actors and has a great (if weird at the beginning) DSL for defining HTTP routes.

In the TDD fashion, we do tests before implementation. Here's an example of tests for the route that retrieves the list of all books.

```
"GET /api/v1/books" should {

  "return OK" in {
    Get("/api/v1/books") ~> route ~> check {
      response.status must equalTo(OK)
    }
  }

  "return all books" in {
    val expected = insertBooks(3).map { book =>
      BookReduced(book._id, book.title, book.author)
    }
    Get("/api/v1/books") ~> route ~> check {
      response.entity must not equalTo None
      val books = responseAs[List[BookReduced]]
```

```
        books must haveSize(expected.size)
        books must equalTo(expected)
      }
    }

  }
```

These are very basic tests that hopefully show the direction one should take to test Spray based APIs. First we're making sure that our route returns the code 200 (OK). The second spec, after inserting few example books to the DB, validates that they are correctly retrieved. Full source code with all tests can be found in ServiceSpec.scala.

How would we implement those tests? Here's the code that provides implementation based on the tests above.

```
val route = pathPrefix("api" / "v1" / "books") {
  get {
    complete(
      collection.find().toList.map(grater[BookReduced].asObject(_))
    )
  }
}
```

That was easy. We define the route **/api/v1/books**, **GET** method and the response inside the **complete** statement. In this particular case, we're retrieving all the books from the DB and transforming them to the **BookReduced** case class. Full source code with all methods (GET, PUT, DELETE) can be found in the ServiceActor.scala.

Both tests and implementation presented here are simplified and in real world scenarios there would be more to do. Actually, complex routes and scenarios are where Spray truly shines.

While developing you can run tests in a quick mode.

[Inside the VM]

```
cd /vagrant
sbt "~test-quick *Spec"
```

Whenever source code changes, all affected tests that end with **Spec** will be re-run automatically. The reason to run only **Spec** tests is that there will be others that should be run after the deployment (more about this in the next article).

I tend to have terminal window with test results displayed at all times and get continuous feedback of the quality of the code I'm working on.

Testing, Building and Deploying

As any other application, this one should be tested, built and deployed.

Let's create a Docker container with the service. Definition needed for the creation of the container can be found in the Dockerfile.

[Inside the VM]

```
cd /vagrant
sbt "testOnly *Spec"
sbt assembly
sudo docker build -t vfarcic/books-service .
sudo docker push vfarcic/books-service
```

We run **Spec** tests, assemble the JAR, build docker container and push it to the Hub. If you're planning to reproduce those steps, please create the account in hub.docker.com and change **vfarcic** to your username.

The container that we built contains everything we need to run this service. It is based on Ubuntu, has JDK7, contains an instance of MongoDB and has the JAR that we assembled. From now on this container can be run on any machine that has Docker installed. There is no need for JDK, MongoDB or any other dependency to be installed on the server. Container is self-sufficient and can run anywhere.

Let's deploy (run) the container we just created in a different VM. That way we'll simulate deployment to production.

To create the production VM with books service deployed run following.

[from source directory]

```
vagrant halt dev
vagrant up prod
```

First command stops the development VM. Each requires 2GB. If you have plenty of RAM you might not need to stop it and can skip this command. The second brings up the production VM with the books service deployed.

After a bit of waiting, new VM is created, Ansible is installed and the playbook **prod.yml** is run. It installs Docker and runs **vfarcic/books-service** that was previously built and pushed to the Docker Hub. While running, it will have the port **8080** exposed and share the directory **/data/db** with the host.

Let's try it out. First we should send PUT requests to insert some test data.

```
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 1, "title": "My Fir
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 2, "title": "My Seco
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 3, "title": "My Thi
```

Let's check whether the service returns correct data.

```
curl -H 'Content-Type: application/json' http://localhost:8080/api/v1/books
```

We can delete a book.

```
curl -H 'Content-Type: application/json' -X DELETE http://localhost:8080/api/v1
```

We can check that the deleted book is not present any more.

```
curl -H 'Content-Type: application/json' http://localhost:8080/api/v1/books
```

Finally, we can request a specific book.

```
curl -H 'Content-Type: application/json' http://localhost:8080/api/v1/books/_id
```

That was a very quick way to develop, build and deploy a microservice. One of the advantages of Docker is that it simplifies deployments by reducing needed dependencies to none. Even though the service we built requires JDK and MongoDB, neither needs to be installed on the destination server. Everything is part of the container that will be run as a Docker process.

Summary

Microservices have existed for a long time but until recently they did not get enough attention due to problems that arise when trying to provision environments capable of running hundreds if not thousands microservices. Benefits that were gained with microservices (separation, faster development, scalability, etc) were not as big as problems that were created with increased efforts that needed to be put intro deployment and provisioning. Docker and CM tools like Ansible can reduce this effort is almost negligible. With deployment and provisioning problems out-of-the-way, microservices are getting back in fashion due to benefits they provide. Development, build and deployment times are faster when compared to monolithic applications.

Spray is a very good choice for microservices. Docker containers shine when they contain everything the application needs but not more. Using big Web servers like JBoss and WebSphere would be an overkill for a single (small) service. Even Web servers with smaller footprint like Tomcat are not needed. Play! is great for building RESTful APIs. However, it still contains a lot of things we don't need. Spray, on the other hand, does only one thing and does it well. It provides asynchronous routing capabilities for RESTful APIs.

We could continue adding more features to this service. For example, we could add registration and authentication module. However, that would bring us one step closer to monolithic applications. In microservices world, new services would be new applications and in case of Docker new containers, each of them listening on a different port and happily responding to our HTTP requests.

When building microservices try to create them in a way that they do one or very few things. Complexity is solved by combining them together, not building one big monolithic application.

This entry was posted in Architecture, Docker, Scala and tagged Docker, Microservices, MongoDB, Monolithic Server, Scala, Spray on January 26, 2015 [http://technologyconversations.com/2015/01/26/microservices-development-with-scala-spray-mongodb-docker-and-ansible/] by Viktor Farcic.

---

16 thoughts on “Microservices Development with Scala, Spray, MongoDB, Docker and Ansible”

Andrew
January 29, 2015 at 12:58 am

You have a whole application (including database) packaged as one Docker container. What do microservices have to do with all this?

Viktor Farcic Post author
January 29, 2015 at 7:37 am

The difference is that this is not the whole application. If, for example, we add authentication, users omanagement, shopping cart, etc. they would not be added to the same service/application but created as new completely separated and independent services.

---

jeroenrosenberg
February 5, 2015 at 1:26 pm

Very nice write up One remark: in the Curl commands you refer to port 8080, while in the Vagrantfile port 8000 is used at the host system for port forwarding. So in order to access the app from outside the VM I believe you should use port 8000.

Viktor Farcic Post author
February 5, 2015 at 10:49 pm

You're right. I tried curl commands from inside the VM. I changed the VM port to 8080 so that it works from the host as well. Thank you for spotting it.

alexglue
February 23, 2015 at 3:06 pm

Why don't u use just [Ansible + Docker_Ubuntu] container instead of [Vagrant_Ubuntu + Ansible + Docker]?

Viktor Farcic Post author
February 23, 2015 at 3:40 pm

I used Vagrant in order to avoid more dependencies on readers laptops. Windows is the major problem. Ansible does not work (without some hacks) and Docker requires boot2docker on Windows and IOS (which is VM in itself). Personally, I use Ubuntu on my Laptop and launch VMs with Vagrant only when I need to simulate multiple servers.

Same can be done without Vagrant by running Ansible and Docker directly. It's just that with Vagrant we know that everyone is on the same OS.

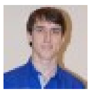
P.S. Docker is using Ubuntu but that is unrelated with the Vagrant OS (or laptops if used without Vagrant).

igorpelevanyuk
March 10, 2015 at 10:49 pm

Thanks Victor for such a nice use case!
I have several questions regarding this example. Would you like to answer, please.
I have made several changes in Vagrant file: used box with 32bit ubuntu, and reduced amount of RAM for VM to 1024. Now, when I try to "up dev" vagrant blocked on during provisioning (when I comment the provision line for dev it finish) and I end up with this lines and it hungs until I push

```
Ctrl+C:
==> dev: Running provisioner: shell…
dev: Running: inline script
==> dev: stdin: is not a tty
```

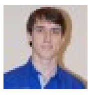
Would you like to comment on this, please.


igorpelevanyuk
March 10, 2015 at 11:35 pm

Sorry for the previous post question. It just required some time to finish the provisioning of vagrant box. But now I see another problem: in the vagrant machine the docker is installed, but when I check running containers there is no any of them (mongodb secifically), the status is "exited(1)" and the quick tests do not pass:

```
root@books-service-dev:/vagrant# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f651d6ee0f54 dockerfile/mongodb:latest mongod 45 minutes ago Exited (1) 45 minutes ago
mongodb
```

Could you explain this, please.


igorpelevanyuk
March 10, 2015 at 11:44 pm

I also did docker log:

```
root@books-service-dev:/vagrant# docker logs f651d6ee0f54
2015/03/10 21:48:31 exec format error
2015/03/10 22:36:34 exec format error
```

May be it is due to the running the whole thing on the 32bit operating system? or just because I have used 32 bit version of linux?

Viktor Farcic Post author

March 11, 2015 at 12:46 pm

It might be caused by 32 bit OS. I tried it only with 64 bits.

If I'd have to guess I'd say that the problem is that Dockerfile FROM instruction is based on the 64 bits Ubuntu. You can change it to 32 bits, build it again and see whether it works. I'd like to help more but I don't have any machine running on 32 bits. If you run into problems that I might help without actually trying it out, please contact me on email or Hangouts (info is in the About page).

Pingback: Microservices : Reading List | Digital Software Development

massenz
June 3, 2015 at 8:49 pm

Reblogged this on Code Trips & Tips and commented:

Very useful post; I'm trying to do something similar with Sentinel (http://github.com/massenz/sentinel) but running it in three containers: one Ngnix with static files, a mid-tier one with Scala Play and the backend with MongoDb container.

A post to follow, soon as I have it all working!

Pingback: Microservices Top 20 Links – 2015 | /dev

bunkertor
July 15, 2015 at 2:38 pm

Reblogged this on Agile Mobile Developer

---

xjxy
August 8, 2015 at 8:00 am

Tremendously helpful post with almost perfect quality. Thanks!

Pingback: Microservices Development with Scala, Spray, MongoDB, Docker and Ansible | Dinesh Ram Kali.