

Software Engineering and Architecture

Extending our Continuous Delivery Pipeline

Olivier Liechti
HEIG-VD
olivier.liechti@heig-vd.ch



MASTER OF SCIENCE
IN ENGINEERING

Agenda


- **Today's session is dedicated to the implementation of our pipeline, with the following next steps:**
 - **Create a new “Beers” micro-service** (and take the opportunity to see how **Spring Boot** provides mechanisms to speed up the implementation of REST APIs).
 - Use **Docker Compose** to define our “runtime topology”, which will initially consist of two containers (1 for the Clock micro-service and 1 for the Beers micro-service).
 - Update our **pipeline script** (Jenkinsfile), to build and package the new micro-service).
 - Update our **pipeline script** to create a “validation environment”, by launching the “runtime topology” during the build process.
 - Introduce the approach and the tools that we will use to **write automated tests** to validate our RESTful endpoints. Initially, we will run the tests on the host machine. Later on, we will run them in the pipeline.

Where did we leave our pipeline?


SEA Build Pipeline [Jenkins x]


Olivier


192.168.99.100:18080/job/SEA%20Build%20Pipeline/


 **Jenkins**


Jenkins > SEA Build Pipeline > [ENABLE AUTO REFRESH](#)


 [Back to Dashboard](#)


 [Status](#)


 [Changes](#)


 [Build with Parameters](#)

 [Delete Pipeline](#)

 [Configure](#)

 [Move](#)



 [Full Stage View](#)

 **Build History** [trend](#)


x


#1

May 19, 2016 11:32 AM

 [RSS for all](#)  [RSS for failures](#)

Pipeline SEA Build Pipeline

 [Recent Changes](#)

 [add description](#)

Stage View

Average stage times:
(Average full run time: ~71ms)

#1

May 19 13:32

No Changes

Setup	Commit	Validation	End
2ms	3ms	5ms	61ms
2ms	3ms	5ms	61ms
master	master	master	master

Permalinks

- [Last build \(#1\), 13 sec ago](#)
- [Last stable build \(#1\), 13 sec ago](#)
- [Last successful build \(#1\), 13 sec ago](#)
- [Last completed build \(#1\), 13 sec ago](#)

 [Help us localize this page](#)

Page generated: May 19, 2016 11:32:30 AM UTC [REST API](#) [Jenkins ver. 1.651.1](#)

What do we have in our pipeline? (1)

- **We have implemented a first micro-service:**
 - We have used Spring Boot.
 - We have implemented a very simple REST API: /clock returns the current time in a JSON payload.
 - We have packaged this micro-service in a Docker image. The executable .jar file produced by maven simply has to be copied on a custom image based on the official java:8 image.

What do we have in our pipeline? (2)

- **We have created our own “CD server image”:**
 - We have used the **official Jenkins image** as a basis.
 - We have installed a bunch of **optional plugins**
 - We have pre-installed a Jenkins “**job**”, based on the new “Pipeline” job type. This job is implemented to fetch (on our GitHub repo) a “**Jenkinsfile**” every time the pipeline is executed.
 - The Jenkinsfile, written in **Groovy**, is the script that defines all the **stages** and all the **steps** that make up our CD pipeline.
 - If we want to **change our build procedure**, we only have to update the Jenkinsfile (commit and push) and hit the “Build” button in the Jenkins Web UI.

What do we have in our pipeline? (3)

- **We have introduced Docker Compose:**
 - We have seen that Docker Compose allows us to define a set of containers, with their properties and relationships, in a .yml file.
 - This has several benefits, and for one it is more practical than running a sequence of “docker run ...” commands manually (or in a custom script).
 - We have defined a so called “topology” for our Jenkins server. Not that interesting, since there is only one container in this case.
- **We have used a “trick” to do further configuration of our Jenkins:**
 - Jenkins exposes a special REST API, which we can use to send a Groovy file. This Groovy file can use the Jenkins Java API (see JavaDocs) to do lots of interesting things. We have used this to install maven and node.js in our environment.

What do we have in our pipeline? (4)

- **Eventually, we have been able to build our first micro-service in the pipeline!**
 - We have run the maven and docker commands in the Jenkinsfile script, instead of typing them in the console.
 - When we hit the “Build” button, if everything works fine, then we end up with a fresh version of our sea/service_clock image.

Task: implement a 2nd micro-service

The Beers micro-service

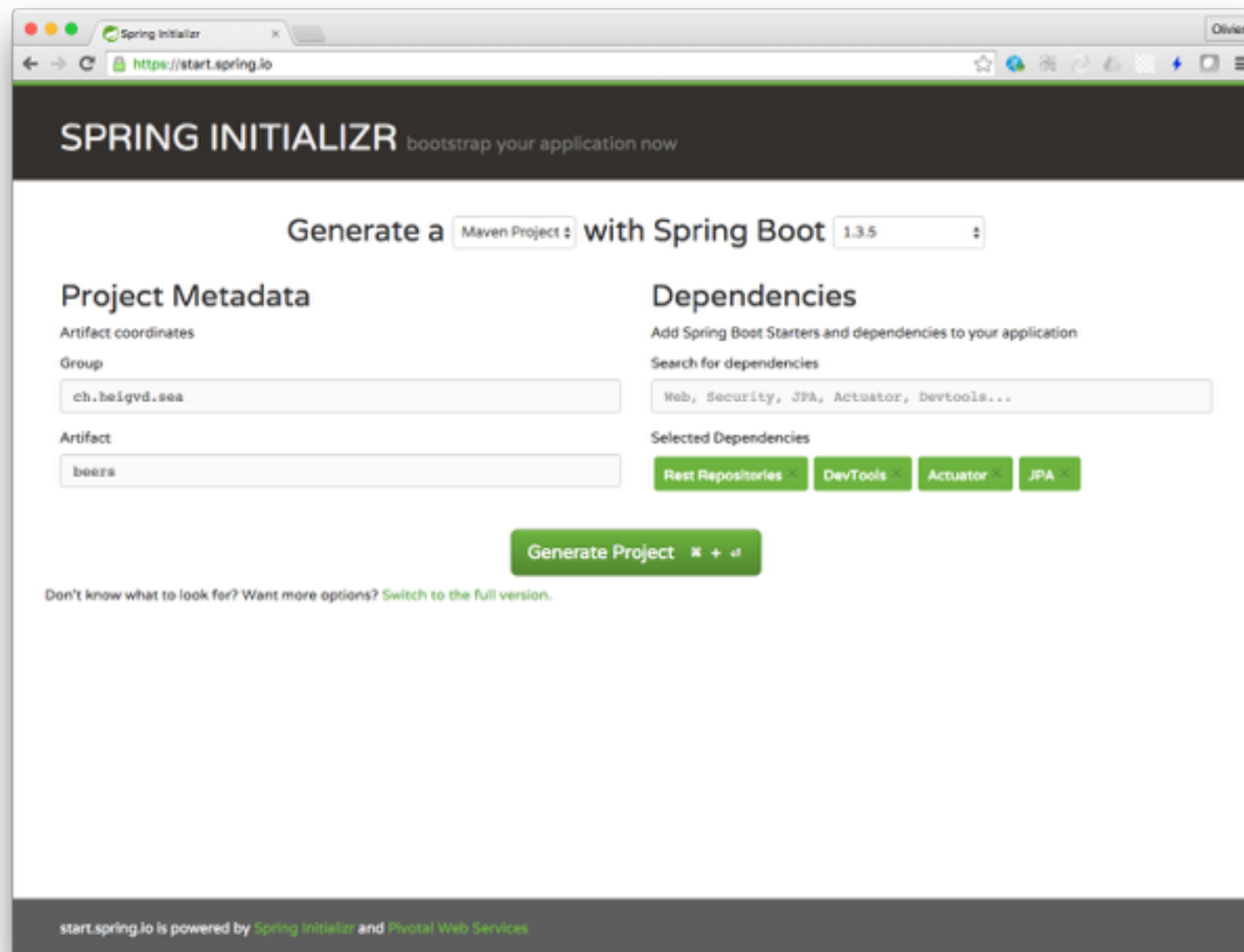
- The first micro-service allowed us to discover Spring Boot, but it is not very interesting.
- Let's build a second one, with a bit more functionality. Let's say that we want to implement a **REST API** that supports **CRUD operations** on Beers:
 - GET/POST/PUT/DELETE /beers/ HTTP/1.1
- The goal is not to spend a lot of time on the design of the domain model. We will consider a single resource (beer), with a couple of properties (name, brewery, type, description).
- We want the data to be persisted in a data store. We also want to be able to search the beers by name.
- **How much time do we need to implement that?**

Using Spring Boot & Spring Data REST

- In the whole “Spring” ecosystem, we can use a couple of components that will do most of the job for us.
- We will rely on default behavior and let the frameworks deal with the generic REST mechanics and with the data persistence.
- Here is a tutorial that gives instructions that you can follow to implement your “Beers” micro-service:
 - **<https://spring.io/guides/gs/accessing-data-rest/>**
- When the Java code works, go back to your notes to package the service in a Docker container:
 - First, do it on your machine.
 - We will integrate it in the pipeline in the next task.

Using Spring Boot & Spring Data REST

- Note: if you don't like to follow tutorials, you can also generate an “empty” skeleton via <http://start.spring.io>.



The image shows the Swagger UI interface for a REST API. The top navigation bar includes tabs for Runner, Import, Builder, and Team Library. On the right, there are icons for a sync status (SYNC OFF), a sign-in button, and a heart icon. The left sidebar contains a search bar and a list of collections: Beer API (1 request), Game Dock (8 requests), MVCDemo (10 requests), MVCDemo copy (10 requests), SEA - beers micro-... (5 requests), and telegram (3 requests). The main area displays a GET request to localhost:8080. The response body is shown in JSON format, indicating a 200 OK status and a response time of 13 ms. The JSON response is as follows:

```
{
  "_links": {
    "beers": {
      "href": "http://localhost:8080/beers?page,size,sort",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/profile"
    }
  }
}
```

RunnerImport

BuilderTeam Library

SYNC OFFSign In

Search

HistoryCollections

All Me Team

Beer API
1 request

Game Dock
8 requests

MVCDemo
10 requests

MVCDemo copy
10 requests

SEA - beers micro-...
5 requests

GET /

GET /beers

GET /profile/beers

POST /beers

GET Find Brewdog IPA

SEA - beers micro-...
5 requests

SonarQube
4 requests

telegram
3 requests

POST /beers

POST /beers

POST http://localhost:8080/beers

ParamsSendSave

AuthorizationHeaders (1)BodyPre-request ScriptTestsGenerate Code

☐ form-data☐ x-www-form-urlencoded☒ raw☐ binaryJSON (application/json)

```
1 {  
2   "name" : "Punk IPA",  
3   "brewery" : "Brewdog",  
4   "type" : "IPA"  
5 }
```

RunnerImport

BuilderTeam Library

SYNC OFFSign In

Search

HistoryCollections

All Me Team

Beer API
1 request

Game Dock
8 requests

MVCDemo
10 requests

MVCDemo copy
10 requests

SEA - beers micro-...
5 requests

GET GET /

GET GET /beers

GET GET /profile/beers

POST POST /beers

GET Find Brewdog IPA

SEA - beers micro-...
5 requests

SonarQube
4 requests

telegram
3 requests

GET /beers

GET /beers

GET http://localhost:8080/beers

Params

Send

Save

Generate Code

AuthorizationHeadersBodyPre-request ScriptTests

TypeNo Auth

BodyCookiesHeaders (5)Tests

Status: 200 OKTime: 26 ms

PrettyRawPreviewJSON

Save Response

```
1 {
2   "_embedded": {
3     "beers": [
4       {
5         "name": "Punk IPA",
6         "brewery": "Brewdog",
7         "type": "IPA",
8         "description": null,
9         "_links": {
10          "self": {
11            "href": "http://localhost:8080/beers/1"
12          },
13          "beer": {
14            "href": "http://localhost:8080/beers/1"
15          }
16        }
17      },
18      {
19        "name": "HardCore IPA",
20        "brewery": "Brewdog",
21        "type": "IPA",
22        "description": null,
23        "_links": {
```

H2 Console

localhost:8080/h2-console/login.do?jsessionid=494454a7ffbcca9100dc9d797f77d072

Auto commit

Max rows: 1000

Auto complete Off

RunRun SelectedAuto completeClearSQL statement:

SELECT * FROM BEER|

SELECT * FROM BEER;

ID	BREWERY	DESCRIPTION	NAME	TYPE
1	Brewdog	null	Punk IPA	IPA
2	Brewdog	null	HardCore IPA	IPA
3	Kronenbourg	null	1664	lager
4	Brewdog	null	Punk IPA	IPA

(4 rows, 4 ms)

Edit

jdbc:h2:mem:testdb

BEER

ID

BREWERY

DESCRIPTION

NAME

TYPE

Indexes

INFORMATION_SCHEMA

Sequences

Users

H2 1.4.191 (2016-01-21)

Task: update our pipeline

Once the micro-service is implemented...

- **One thing that is easy to do, is to adapt the existing Jenkinsfile to not only build the Clock micro-service, but also the Beers micro-service:**
 - You might have one issue, when you do a “mvn clean package” on the “Beers” micro-service. Because of the implementation of Spring JPA, this command **MUST** be run in the directory that contains the pom.xml.
 - (in my “build-image.sh” script, I was running it from another directory and using the -f flag; this raised a issue).
- **You can then implement the “validation”:**
 - Define a new Docker Compose “topology” (e.g. call it topology-runtime).
 - Test it on your machine (docker-compose up)
 - Update Jenkinsfile to do it in the pipeline!

What should you get at the end?

- **Every time you hit the “Build” button in Jenkins:**
 - Jenkins will fetch your Jenkinsfile (via git)
 - Jenkins will build the clock micro-service and package it in a new Docker image
 - Jenkins will build the beers micro-service and package it in a new Docker image
 - Jenkins will “docker-compose down/up” your runtime topology. After that, you should have 2 new containers. Each one will have a process listening on port 8080 (and you will have to map those to 2 ports of your VM).
- In other words, you will have a fresh “Test environment” that you will be able to use for manual (e.g. with Postman) and automated tests).

Task: get familiar with API tests

API Tests

- We have seen that there are many different types of tests (agile testing “quadrants”).
- We would typically write Unit Tests in the Spring Boot project (there is a test directory). This is similar to what we have done in the very first lab (where you have been given unit tests).
- What we are interested in at this stage, is to implement automated tests that will validate the behavior of our REST API.
- We should be able to write scenarios, where we will use some kind of HTTP client to issue requests, get responses. We will then make assertions on the response status and payload.
- There are of course many libraries that you can use to do that, in different programming languages.

SuperTest and supertest-as-promised

- SuperTest is a npm module, which provides you with a “fluent” API to write your API tests. When you use it, you write your assertions in callback methods (obviously testing a REST API involves a lot of async operations).
 - <https://github.com/visionmedia/supertest>
- supertest-as-promised is a wrapper, which allows you to use promises instead of callbacks. Your test code will be much clearer.
 - <https://github.com/WhoopInc/supertest-as-promised>
- When you use these test libraries, you have to use a “test runner”. In JavaScript, Jasmine and mocha are two popular examples. We will use Jasmine (and see that there is an issue to deal with, because of a naming conflict with supertest).

Preview

```
var apiPrefix = process.env.CLOCK_API_PREFIX || "http://localhost:8080/api";

var assert = require("assert");
var api = require('supertest-as-promised')(apiPrefix);
var expect = require('chai').expect;
var finish_test = require("../supertest-helper");

describe("Basic scenarios for Clock micro-service", function() {

  describe("The clock micro-service", function() {
    it("should expose a REST endpoint on /clock", function(done) {
      return api
        .get("/clock")
        .expect(200)
        .end( finish_test(done) );
    });

    it("should return the current time via the REST endpoint on /clock", function(done) {
      api
        .get("/clock")
        .expect(200)
        .then( function (response ) {
          var payload = response.body;
          expect(payload).to.be.an('object');
          expect(payload.date).to.not.be.undefined;
          done();
        })
        .catch(function(err) {
          done.fail(err);
        })
    });

    it("should reject requests to wrong URIs", function(done) {
      return api
        .get("/ThisIsAnExampleOfAURIUnknownByTheService")
        .expect(404)
        .end( finish_test(done) );
    });
  });
});
```