

Philipp Hauer's Blog

(<http://blog.philippauer.de/>)

Web Architecture, Java Ecosystem, Software Craftsmanship

Building a Dropwizard Microservice with Docker and Maven

Posted on September 20, 2015 (<http://blog.philippauer.de/building-dropwizard-microservice-docker-maven/>) by Philipp Hauer (<http://blog.philippauer.de/author/philippauer/>)

Dropwizard produces a fat jar containing every dependency your microservice needs to run. This includes a web server. This way, no web server needs to be installed and configured on the target machine. However, there is some infrastructure left (like the JRE) which still has to be installed before the deployment. That's where Docker enters the stage. With Docker we can produce an artifact containing really everything we need to run our microservice. In this post, we take a look at how we can integrate Docker into our Maven build, run our tests against the container and push the image to a repository.

Contents [hide]

- 1 Microservices and Docker
- 2 Preconditions: The Fat Jar
- 3 The Dockerized Maven Build
 - 3.1 Step 1. Building the Docker Image with the docker-maven-plugin
 - 3.2 Step 2. Running the Microservice Container
 - 3.3 Step 3. Linking with the Database Container, Exchanging Ports and Running both Containers
 - 3.4 Step 4: Run the Integration Tests against the Container
 - 3.5 Step 5: Pushing the Image to a Registry
- 4 Further Reading

Microservices and Docker

5 Related Posts

(Dropwizard)

Microservices

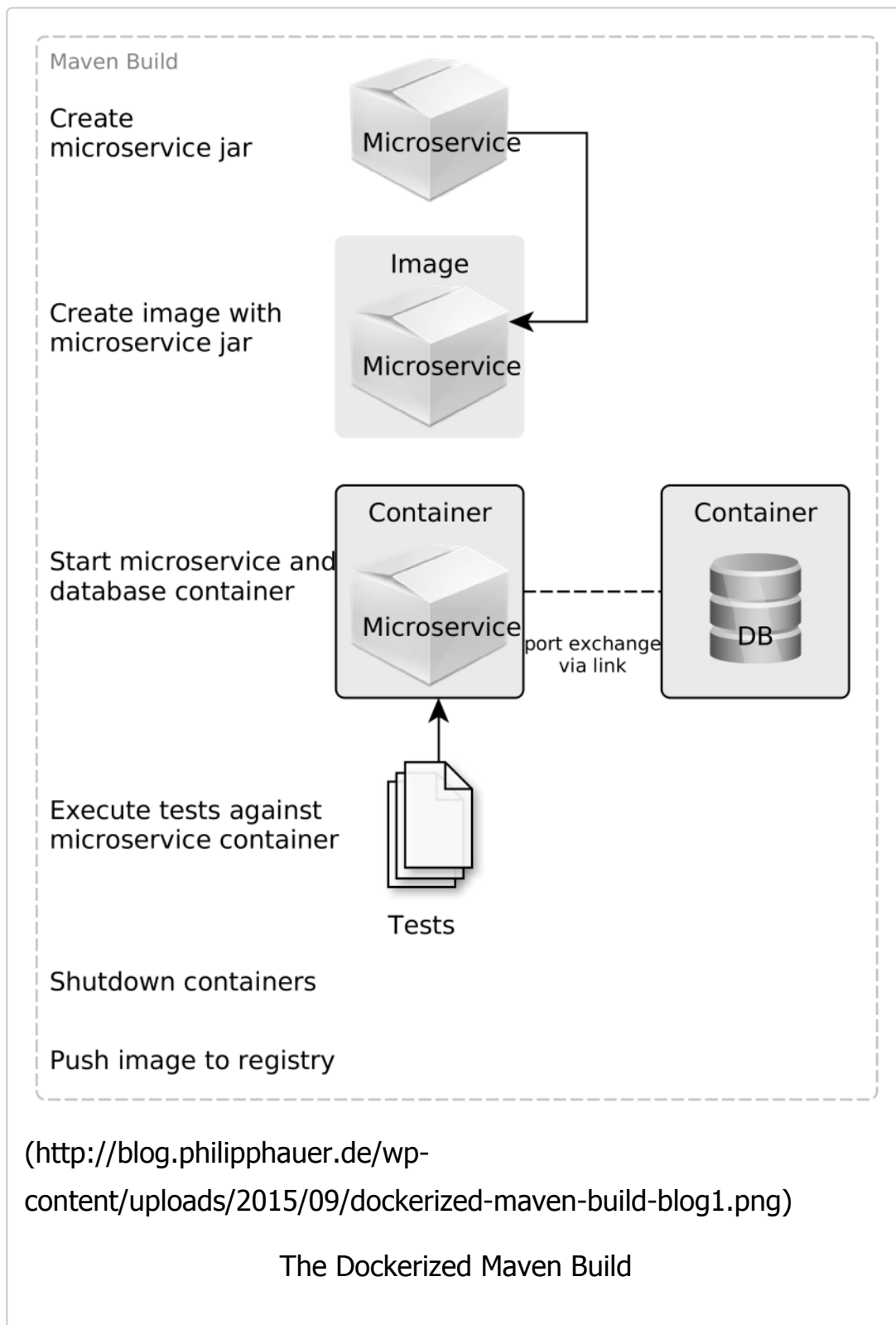
(<http://blog.philippbauer.de/microservices-nutshell-pros-cons/>) and Docker fit pretty well together. You receive the full control over the execution environment of your microservice (environment variables, runtime parameters, heap space arguments, file system, JRE). We can test with the environment that will finally be deployed into production. This increases the reliability of tests and significantly reduces the risk when deploying the application in production.

Moreover, the build is getting straightforward: One Dropwizard project produces one artifact (the runnable fat jar), which can be wrapped into one Docker image that runs independently on a target machine. Moreover, no infrastructure (JRE, web or application server) has to be installed before the deployment. The image contains everything necessary. This independence and the small deployment unit speeds up and simplifies our build and deployment. This is a huge benefit and makes the implementation of Continuous Delivery (http://blog.philippbauer.de/tutorial-continuous-delivery-with-docker-jenkins/#Continuous_Delivery_using_Docker) easy.

Preconditions: The Fat Jar

Let's assume that you have already a Dropwizard project, which produces a nice runnable fat jar. So calling `mvn package` creates `target/app-0.0.1-SNAPSHOT.jar`.

The Dockerized Maven Build



- Building a Docker image containing the microservice fat jar.
- Running the integration tests against the created container. Therefore, we have to start the container before the integration tests, run the integration tests and stop the container afterwards.
- Our microservice needs a database (in this case a MongoDB). Therefore, we also have to

run a container with MongoDB before the integration tests can run. In order to provide our microservice access to the database, we link both container together.

- Finally, we push our built and tested microservice image to a Docker registry (like Docker Hub).

Here we go.

Step 1. Building the Docker Image with the docker-maven-plugin

There are a lot of docker-maven-plugins available. I'm using the docker-maven-plugin with the groupId org.jolokia (<https://github.com/rhuss/docker-maven-plugin>), because it seems to be actively developed with a lot of commits and contributors. Let's configure the docker-maven-plugin to build the image with our microservice called "prozu". Therefore the plugin has to create a Dockerfile first. The Dockerfile is configured at two points. The first is the plugin configuration in the pom:

```

1 <properties>
2   <docker.registry.name></docker.registry.name><!-- leave empty for docker hub; use e.g
3   <docker.repository.name>${docker.registry.name}phauer/${project.artifactId}</docker.r
4 </properties>
5 ...
6 <plugin>
7   <groupId>org.jolokia</groupId>
8   <artifactId>docker-maven-plugin</artifactId>
9   <version>0.13.3</version>
10  <configuration>
11    <images>
12      <image>
13        <alias>${project.artifactId}</alias>
14        <name>${docker.repository.name}:${project.version}</name>
15        <build>
16          <from>java:8-jre</from>
17          <maintainer>phauer</maintainer>
18          <assembly>
19            <descriptor>docker-assembly.xml</descriptor>
20          </assembly>
21          <ports>
22            <port>8080</port>
23            <port>8081</port>
24          </ports>
25          <cmd>
26            <shell>java -jar \
27              /maven/${project.build.finalName}.jar server \
28              /maven/docker-config.yml</shell>
29          </cmd>
30        </build>
31      </image>
32      <!-- later more -->
33    </images>
34  </configuration>

```

```

35         </images>
36     </configuration>
37 </plugin>

```

This configuration tells the docker-maven-plugin to create a Dockerfile based on the image "java:8-jre" (so we get a JRE8), expose the ports 8080 and 8081 (the standard ports of a Dropwizard service and its admin backend) and finally provide the command to start the runnable fat jar.

But how do we tell the plugin which files should be put into the container? This is done by means of the assembly descriptor docker-assembly.xml which is placed under `src/main/docker`.

```

1 <assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2" xm
2   xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.
3   <id>${project.artifactId}</id>
4   <files>
5       <file>
6           <source>target/${project.build.finalName}.jar</source>
7           <outputDirectory></outputDirectory>
8       </file>
9       <file>
10          <source>src/main/resources/docker-config.yml</source>
11          <outputDirectory></outputDirectory>
12      </file>
13  </files>
14 </assembly>

```

This way, the plugin adds the Dropwizard jar and the configuration yml to the Dockerfile. The resulting Dockerfile can be found in `target/docker/` and looks like this:

```

1 FROM java:8-jre
2 MAINTAINER phauer
3 EXPOSE 8080 8081
4 COPY maven /maven/
5 CMD java -jar \
6     /maven/prozu-service-0.0.1-SNAPSHOT.jar server \
7     /maven/docker-config.yml

```

The folder 'maven' contains everything we specified in the assembly descriptor and is also located under `target/docker`.

Now we can build the docker container using

```
1 mvn docker:build
```

Afterwards you can check the existence of the image in the local repository using `docker images` or run it manually with `docker run -d -p 8080:8080 -p 8081:8081 phauer/prozu-service:0.0.1-SNAPSHOT` (or use `-it` instead of `-d` to run it in the foreground and to see the output).

Step 2. Running the Microservice Container

But we want to start our container with Maven. For this we have to add the following `<run>` block right under the `<build>` block above.

```

1  <properties>
2    <docker.host.address>localhost</docker.host.address><!-- this is not localhost when u
3    <prozu.port>8080</prozu.port>
4    <prozu.port.admin>8081</prozu.port.admin>
5  </properties>
6  ...
7  <run>
8    <namingStrategy>alias</namingStrategy>
9    <ports>
10     <port>${prozu.port}:8080</port>
11     <port>${prozu.port.admin}:8081</port>
12   </ports>
13   <volumes>
14     <bind>
15       <volume>${user.home}/logs:/logs</volume>
16     </bind>
17   </volumes>
18   <wait>
19     <url>http://${docker.host.address}:${prozu.port.admin}/ping</url>
20     <time>10000</time>
21   </wait>
22   <log>
23     <prefix>${project.artifactId}</prefix>
24     <color>cyan</color>
25   </log>
26 </run>

```

In the `<run>` block we define the parameters for running our microservice container. We map the ports 8080 and 8081 of our microservice within the container to the same ports on our host machine so we can access the service. Moreover, we bind the folder 'logs' within the container (this is where our microservice places its log files) to '~/.logs' so we can access the log files on our host machine. To enable logging in Dropwizard we have to add an appender in the configuration yml file:

```

1  logging:
2    level: INFO
3    appenders:
4      - type: console

```

```

5      threshold: ALL
6      timeZone: UTC
7      target: stdout
8      logFormat: "%-6level [%d{HH:mm:ss.SSS}] [%t] %logger{5} - %X{code} %msg %n"
9  - type: file
10     currentLogFilename: /logs/prozu.log
11     threshold: ALL
12     archive: true
13     archivedLogFilenamePattern: /logs/prozu-%d.log
14     archivedFileCount: 5
15     timeZone: UTC
16     logFormat: "%-6level [%d{HH:mm:ss.SSS}] [%t] %logger{5} - %X{code} %msg %n"

```

Finally, the docker-maven-plugin needs to know, how it can detect if the container has finished starting up. This is done in the configuration. We tell to poll the ping URL of the admin backend until it receives an answer. Please note that the docker host is not 'localhost' if you are using boot2docker on Windows or Mac OS X. The IP of the docker host/the VM is shown at startup of the docker daemon or by running `docker-machine ip default`.

Now we can start our container by calling `mvn docker:start`. However, our microservice needs a running database.

Step 3. Linking with the Database Container, Exchanging Ports and Running both Containers

First, we configure our database container. For that we just define another `<image>`. There is no `<build>` block necessary, because we use a predefined image from Docker Hub.

```

1  <image>
2    <alias>mongodb</alias>
3    <name>mongo:2.6.11</name>
4    <run>
5      <namingStrategy>alias</namingStrategy>
6      <cmd>--smallfiles</cmd>
7      <wait>
8        <log>waiting for connections on port</log>
9        <time>10000</time>
10     </wait>
11     <log>
12       <prefix>MongoDB</prefix>
13       <color>yellow</color>
14     </log>
15   </run>
16 </image>

```

Now we can start the MongoDB container, but it's not accessible by the microservice yet. One solution would be to bind MongoDB's port 27017 to a port on the host system. This way the microservice could access MongoDB by using this port. In this case we have to take care of

the microservice (within the container), which should always get the right IP of the host system and the port. Moreover, sometimes you don't want to expose the MongoDB on the whole host system. So let's try another approach for connecting containers: linking. Linking allows containers to exchange information (like IPs or ports) without exposing them on the host system. Fortunately, the docker-maven-plugin allows us to comfortably link containers together. The only thing we have to do is to add an element in the `<run>` block of our microservice image configuration.

```
1 <run>
2   ...
3   <links>
4     <link>mongodb:db</link>
5   </links>
6 </run>
```

Now there are additional environment variables available within the microservice container. For instance the variable `$DB_PORT_27017_TCP_ADDR` contains the IP and `$DB_PORT_27017_TCP_PORT` contains the port. The only thing left to do, is to pass this information to our microservice. We could use the configuration yml file for this or just use system properties. We are doing the latter and adjust our starting command:

```
1 <cmd>
2   <shell>java -DdbHost=$DB_PORT_27017_TCP_ADDR \
3     -DdbPort=$DB_PORT_27017_TCP_PORT -jar \
4     /maven/${project.build.finalName}.jar server \
5     /maven/docker-config.yml</shell>
6 </cmd>
```

In your Java code we can access the system properties by calling `System.getProperty("dbPort")`.

Now we can run `mvn docker:start` and the docker-maven-plugin will first start the MongoDB container and afterwards the microservice container. `mvn docker:stop` will stop both containers. That's nice!

Step 4: Run the Integration Tests against the Container

Next we have to tell maven to start both containers before the integration tests and stop them afterwards. This is easy. Just add the following executions to the docker-maven-plugin:

```
1 <execution>
2   <id>start</id>
```



```

3     <phase>pre-integration-test</phase>
4     <goals>
5         <goal>build</goal>
6         <goal>start</goal>
7     </goals>
8 </execution>
9 <execution>
10    <id>stop</id>
11    <phase>post-integration-test</phase>
12    <goals>
13        <goal>stop</goal>
14    </goals>
15 </execution>

```

Besides, we have to configure the failsafe plugin to execute the integration tests. I like to use a self-defined marker annotation (IntegrationTest) to distinguish between unit and integration tests.

```

1 package de.philippbauer.prozu.di;
2
3 import java.lang.annotation.*;
4
5 @Target(ElementType.TYPE)
6 @Retention(RetentionPolicy.RUNTIME)
7 public @interface IntegrationTest {}

```

Moreover, we pass the URL of our service to the test with the system property "service.url".

```

1 <plugin>
2     <groupId>org.apache.maven.plugins</groupId>
3     <artifactId>maven-failsafe-plugin</artifactId>
4     <version>2.18.1</version>
5     <configuration>
6         <phase>integration-test</phase>
7         <includes>
8             <include>/**/*.java</include>
9         </includes>
10        <groups>de.philippbauer.prozu.di.IntegrationTest</groups>
11        <systemPropertyVariables>
12            <service.url>http://${docker.host.address}:${prozu.port}</service.url>
13        </systemPropertyVariables>
14    </configuration>
15    <executions>
16        <execution>
17            <goals>
18                <goal>integration-test</goal>
19                <goal>verify</goal>
20            </goals>
21        </execution>
22    </executions>
23 </plugin>
24 <plugin>
25     <groupId>org.apache.maven.plugins</groupId>
26     <artifactId>maven-surefire-plugin</artifactId>
27     <version>2.18.1</version>
28     <configuration>

```

```

29         <excludedGroups>de.philippbauer.prozu.di.IntegrationTest</excludedGroups>
30     </configuration>
31 </plugin>

```

A simple integration test looks like this:

```

1  @Category(IntegrationTest.class)
2  public class EmployeeResourceTestViaDocker {
3      @Test
4      public void testConnection() throws IOException {
5          String baseUrl = System.getProperty("service.url");
6          URL serviceUrl = new URL(baseUrl + "employees");
7          HttpURLConnection connection = (HttpURLConnection) serviceUrl.openConnection();
8          int responseCode = connection.getResponseCode();
9          assertEquals(200, responseCode);
10     }
11 }

```

You can try the whole process by running `mvn verify`.

At this point I like to emphasize how great it is to use a database container for tests. This way, we don't have to set up and maintain a database for the tests manually. The plugin takes care for us and automatically starts a fresh database. That's nice for testing. Moreover, we don't have to clean an existing database because we always start with an empty one. You only have to clean the database before each test. Since creating a fresh database is that easy and fast, we don't need an in-memory-database.

Step 5: Pushing the Image to a Registry

Finally, we want to push the image to the image registry Docker Hub. The configuration needed is easy. Just add another execution to the docker-maven-plugin:

```

1  <execution>
2      <id>push-to-docker-registry</id>
3      <phase>deploy</phase>
4      <goals>
5          <goal>push</goal>
6      </goals>
7  </execution>

```

Let's assume we only want to produce an image and we don't want to deploy the fat jar to an artifact repository. Therefore, we skip the normal execution of the maven-deploy-plugin:

```

1  <plugin>
2      <groupId>org.apache.maven.plugins</groupId>
3      <artifactId>maven-deploy-plugin</artifactId>
4      <version>2.7</version>
5      <configuration>

```

```

6         <skip>true</skip>
7     </configuration>
8 </plugin>

```

Don't forget to pass your username and password of your Docker Hub account.

```
1 mvn -Ddocker.username=<username> -Ddocker.password=<password> deploy
```

This statement builds a Docker image with our microservice, tests it and finally deploys it to Docker Hub.

If you want to push your image to your own registry instead of Docker Hub just set the `docker.registry.name` property. Let's assume your docker registry runs on your local machine on port 5000:

```
1 <docker.registry.name>localhost:5000/</docker.registry.name>
```


Voila! We successfully integrated Docker into our Maven build lifecycle.

Further Reading

- Documentation of the jolokia docker-maven-plugin used in this post (<https://github.com/rhuss/docker-maven-plugin>)
- Linking Docker containers together (<https://docs.docker.com/userguide/dockerlinks/>)

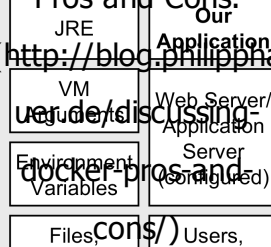
Related Posts

Delivery with Docker and Jenkins



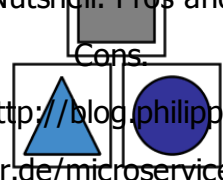
(<http://blog.philippauer.de/tutorial-continuous-delivery-with-docker-jenkins/>)

Discussing Docker. Our Docker Image Pros and Cons.




(<http://blog.philippauer.de/discussing-docker-pros-and-cons/>)

Microservices in a Nutshell. Pros and Cons.



(<http://blog.philippauer.de/microservices-nutshell-pros-cons/>)



Eclipse P2 Repository

(<http://blog.philippauer.de/offline-copy-mirror-eclipse-p2-repository/>)



This entry was posted in Build and Development Infrastructure (<http://blog.philippbauer.de/category/build-and-development-infrastructure/>), Software Architecture (<http://blog.philippbauer.de/category/software-architecture/>) and tagged Build (<http://blog.philippbauer.de/tag/build/>), Container (<http://blog.philippbauer.de/tag/container/>), Docker (<http://blog.philippbauer.de/tag/docker/>), Dropwizard (<http://blog.philippbauer.de/tag/dropwizard/>), Maven (<http://blog.philippbauer.de/tag/maven/>), Microservice (<http://blog.philippbauer.de/tag/microservice/>), Test (<http://blog.philippbauer.de/tag/test/>). Bookmark the permalink (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/>).

← Enriching RESTful Services with Swagger (<http://blog.philippbauer.de/enriching-restful-services-swagger/>)
Discussing Docker. Pros and Cons. → (<http://blog.philippbauer.de/discussing-docker-pros-and-cons/>)

2 thoughts on “Building a Dropwizard Microservice with Docker and Maven”



MulloyMorrow says:

December 28, 2015 at 9:35 pm (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/#comment-293>)

Can you clarify which dependency artifacts you are using in your test classes? Specifically:

- Category
- IntegrationTest.class

Reply (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/?replytocomment=293>)



Philipp Hauer says:

December 29, 2015 at 12:32 pm (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/#comment-294>)

Hi.

No problem. I updated my post and added the source code for the annotation IntegrationTest. Category is a JUnit class (`org.junit.experimental.categories.Category`). I hope this helps you. Thanks for your comment.

Cheers,
Philipp

Reply (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/?replytocomment=294>)

Leave a Reply

Your email address will not be published.

Type your comment here...

Name

Email

Website

Post Comment

☐ **Notify me of follow-up comments by email.**

☐ **Notify me of new posts by email.**

Search ...



I am Philipp Hauer (M.Sc.) and I work as a software engineer for Spreadshirt (<https://www.spreadshirt.com/>) in Leipzig, Germany. I focus on developing Java-based web applications and I'm enthusiastic about clean code, web architectures and continuous delivery.

Recent Posts

Don't generate (everything). Or: Framework beats Generator (<http://blog.philippbauer.de/dont-generate-everything-framework-beats-generator/>)

Analyzing the Memory Usage of a Vaadin Application (<http://blog.philippbauer.de/analyzing-memory-usage-vaadin-application/>)

Databases as a Challenge for Continuous Delivery (<http://blog.philippbauer.de/databases-challenge-continuous-delivery/>)

Tutorial: Continuous Delivery with Docker and Jenkins (<http://blog.philippbauer.de/tutorial-continuous-delivery-with-docker-jenkins/>)

Discussing Docker. Pros and Cons. (<http://blog.philippbauer.de/discussing-docker-pros-and-cons/>)

Recent Comments

Don't generate (everything). Or: Framework beats Generator (<http://blog.philippbauer.de/dont-generate-everything-framework-beats-generator/>) on Evaluating Vaadin: Strengths and Weaknesses (<http://blog.philippbauer.de/evaluating-vaadin-strengths-weaknesses/#comment-297>)

Don't generate (everything). Or: Framework beats Generator (<http://blog.philippbauer.de/dont-generate-everything-framework-beats-generator/>) on Enriching RESTful Services with Swagger (<http://blog.philippbauer.de/enriching-restful-services-swagger/#comment-296>)

Don't generate (everything). Or: Framework beats Generator (<http://blog.philippbauer.de/dont-generate-everything-framework-beats-generator/>) on Local Copy of an Eclipse P2 Repository (<http://blog.philippbauer.de/offline-copy-mirror-eclipse-p2-repository/#comment-295>)

Philipp Hauer on Building a Dropwizard Microservice with Docker and Maven (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/#comment-294>)

MulloyMorrow on Building a Dropwizard Microservice with Docker and Maven (<http://blog.philippbauer.de/building-dropwizard-microservice-docker-maven/#comment-293>)

Archives

January 2016 (<http://blog.philippbauer.de/2016/01/>)

December 2015 (<http://blog.philippbauer.de/2015/12/>)

November 2015 (<http://blog.philippbauer.de/2015/11/>)

October 2015 (<http://blog.philippbauer.de/2015/10/>)

September 2015 (<http://blog.philippbauer.de/2015/09/>)

July 2015 (<http://blog.philippbauer.de/2015/07/>)

May 2015 (<http://blog.philippbauer.de/2015/05/>)

April 2015 (<http://blog.philippbauer.de/2015/04/>)

March 2015 (<http://blog.philippbauer.de/2015/03/>)

February 2015 (<http://blog.philippbauer.de/2015/02/>)

Categories

Build and Development Infrastructure (<http://blog.philippbauer.de/category/build-and-development-infrastructure/>)

Software Architecture (<http://blog.philippbauer.de/category/software-architecture/>)

Software Craftsmanship (<http://blog.philippbauer.de/category/software-craftsmanship/>)

Tools and Environment (<http://blog.philippbauer.de/category/tools-and-environment/>)

Web Development (<http://blog.philippbauer.de/category/web-development/>)

Impressum, Privacy Policy (</legal/>)

Powered by WordPress (<http://wordpress.org/>)

Theme: Jokkmokk by Asokay (<http://asokay.com/>)