In the documented flow, the simple Spring Boot Web app is created. The app runs in a docker container and connects through “link” mechanism to MySql container that was created previously. The app exposes simple REST API to get and create users. Users are stored in MySql container.

1.

Under docker/boot-mysql put the JAR and Dockerfile from docker directory of this mini project.

Contents of the docker file ---

FROM java:8-jre

ADD boot-mysql-0.0.1-SNAPSHOT.jar /app/

CMD ["java", "-jar", "/app/boot-mysql-0.0.1-SNAPSHOT.jar"]

EXPOSE 8000

2.

To build the web app image, use the following from docker/boot-mysql directory ---sudo docker build -t shpboris/bootmysql .

3.

To start container, use --- sudo docker run --name bootmysql --link demo-mysql:db -p 8000:8000 -d shpboris/bootmysql

While name of the image out of which container should be created = shpboris/bootmysql , name of web app container = bootmysql, name of MySql container = demo-mysql, name of the link = db , exposed web app port = 8000.

The datasource url of web app should use the name of the link from above instruction – i.e db !!!. Like here (see in red) - spring.datasource.url=jdbc:mysql://**db**:3306/demo.

4.

To verify that container has already started, use --- sudo docker logs -f bootmysql

5.

Pay attention, that DB name, user name and password from datasource should match those of the started DB container. See docker\my-sql-setup mini project folder for details on setting up DB container.

For convenience – here is the datasource of the web app.

server.port = 8000

server.contextPath=/boot-mysql

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://db:3306/demo

spring.datasource.username=demo\_user

spring.datasource.password=demo\_pass

6.

The exposed REST API is

6.1.

Create user –

POST to <http://localhost:8000/boot-mysql/users>

Body

{

"id" : "2",

"name" : "2"

}

6.2.

Get users –

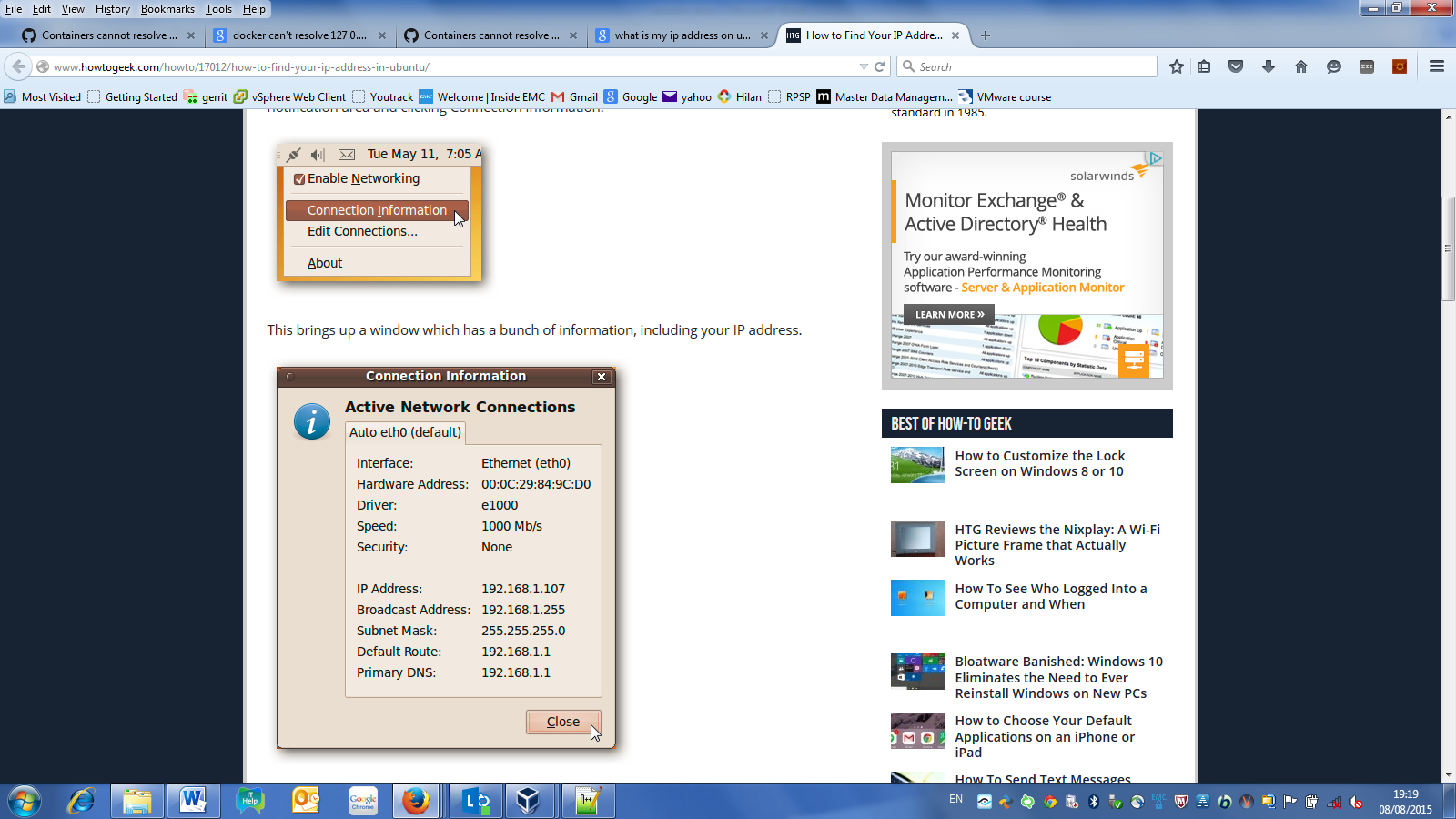
GET on <http://localhost:8000/boot-mysql/users>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* some additional useful info below \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7.

Totally different option to connect to DB, is not to use container linking, but straight IP/port address of DB. Container linking described above is tied to putting DB and App Server on the same machine so it is not optimal.

To use IP , you should use URL like this to access your DB from the application - jdbc:mysql:// 10.0.2.15:3306/demo. Pay attention to IP=10.0.2.15. It can’t be localhost or 127.0.0.1 – since this is resolved to app container IP itself !!! So access to DB will not work with localhost or 127.0.0.1 !!! To get real IP of Ubuntu within Virtualbox, follow steps on screens below and take “IP address”. <http://www.howtogeek.com/howto/17012/how-to-find-your-ip-address-in-ubuntu/>



The app container is started like this (i.e no mentioning of DB at all) ---- sudo docker run --name bootmysql -p 8000:8000 -d shpboris/bootmysql

8.

In order to connect to DB that runs at arbitrary location, you can pass the URL of DB to your application container with environment variable like this ----- sudo docker run --name rpsp-ip -e url=jdbc:mysql://10.0.2.15:3306/rpsp -p 8888:9999 -d shpboris/rpsp-ip

The part highlighted in red is the URL of DB. In order to make use of it in the Java application, you read it like normal environment variable –

1.

Somehow get access to environment

Environment env;

2.

Get your property like this -

env.getProperty("url")

3.

Here is the relevant code

@EnableTransactionManagement

**public** **class** DatabaseConfiguration **implements** EnvironmentAware {

**private** Environment env;

@Override **public** **void** setEnvironment(Environment env) {

**this**.env = env;

}

@Bean(destroyMethod = "shutdown") @Profile("!" + Constants.***SPRING\_PROFILE\_CLOUD***)

**public** DataSource dataSource() {

log.debug("Configuring Datasource");

**if** (propertyResolver.getProperty("url") == **null**

&& propertyResolver.getProperty("databaseName") == **null**) {

log.error("Your database connection pool configuration is incorrect! The application"

+ "cannot start. Please check your Spring profile, current profiles are: {}",

Arrays.*toString*(env.getActiveProfiles()));

**throw** **new** ApplicationContextException(

"Database connection pool is not configured correctly");

}

HikariConfig config = **new** HikariConfig();

config.setDataSourceClassName(propertyResolver.getProperty("dataSourceClassName"));

**if**(!org.apache.commons.lang3.StringUtils.*isEmpty*(env.getProperty("url"))){

config.addDataSourceProperty("url", env.getProperty("url"));

}

**else**{

**if** (propertyResolver.getProperty("url") == **null** || ""

.equals(propertyResolver.getProperty("url"))) {

config

.addDataSourceProperty("databaseName", propertyResolver.getProperty("databaseName"));

config.addDataSourceProperty("serverName", propertyResolver.getProperty("serverName"));

} **else** {

config.addDataSourceProperty("url", propertyResolver.getProperty("url"));

}

}

config.addDataSourceProperty("user", propertyResolver.getProperty("username"));

config.addDataSourceProperty("password", propertyResolver.getProperty("password"));

**return** **new** HikariDataSource(config);

}

@Bean **public** SpringLiquibase liquibase(DataSource dataSource) {

SpringLiquibase liquibase = **new** SpringLiquibase();

liquibase.setDataSource(dataSource);

liquibase.setChangeLog("classpath:config/liquibase/master.xml");

liquibase.setContexts("development, production");

**if** (env.acceptsProfiles(Constants.***SPRING\_PROFILE\_FAST***)) {

liquibase.setShouldRun(**false**);

} **else** {

log.debug("Configuring Liquibase");

}

**return** liquibase;

}

@Bean **public** Hibernate4Module hibernate4Module() {

**return** **new** Hibernate4Module();

}

}