API Platform - Training: Light4j Side Car Proxy 201

Introduction

NOTE: This tutorial is in point form currently, and will be expanded to a full "script" once the initial review is done.

In this training session, we will cover the complete steps on how to develop APIs using the Light4j Side Car Proxy. We will be covering topics like: creating a simple Spring Boot application, how to prepare your API for a containerized environment, how to integrate Light4J Sidecar Proxy with your API, and how to test your API.

Each section will cover a different part of the development process, and by the end of the training, you will have a containerized business API running with Light4j Sidecar Proxy.

1. Creating A Simple API Using Spring Boot

Goal	• To create a simple "hello world" application using Spring Boot
Prerequisites	• **An IDE that supports JAVA** • **JDK** (JDK 11 is used in this tutorial) • **Gradle 4+** or **Maven 3.2+** (Gradle is used in this tutorial) • **Git for Windows** or **Sourcetree**

In this section we will cover a basic way to develop a simple Spring Boot service. There are infinite ways to approach doing this, for this tutorial we will be using **spring-microservice-archetype** as a starting point for our API. This project makes the preparing your API for a containerized environment easier since the required files will be generated when it is instantiated.

First thing we want to do is we want to clone the repo, which can be found here. If you are using **Sourcetree**, it would be easiest to use **SSH**. If you are using **git**, the command would be:

```
git clone https://bitbucket.sunlifecorp.com/scm/eiin/springboot-
microservice-archetype.git <servicename>
```

Note: <servicename> is the name of your service.

Once the project is finished cloning, the first thing we are going edit is the **gradle-wrapper.properties** file, located in the **/gradle/wrapper/** folder found in the projects root directory. We need to update this file to use a more up to date version of **Gradle**. Change the file to look like this (This tutorial uses **Gradle 6.3**, but you can use any version after 4):

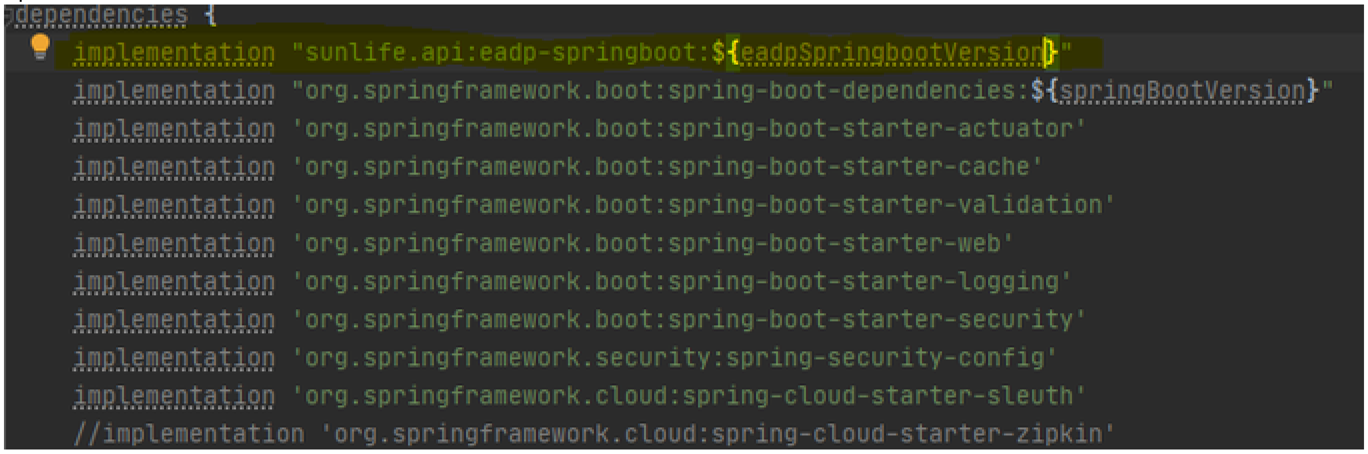
```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=http://artifactory.sunlifecorp.com/artifactory/gradle-
services-cache/distributions/gradle-6.3-bin.zip
```

Now you can open the project in your preferred IDE, and let **Gradle** sync do it's thing. The first thing you will notice is an issue regarding **org.gradle.java.home**. To fix this, we will need to open the **gradle.properties** file found in the root directory, and set **org.gradle.java.home** variable to the location of our **JDK** installation.

```
org.gradle.java.home=C:/Program Files/Java/JDK_11.04
```

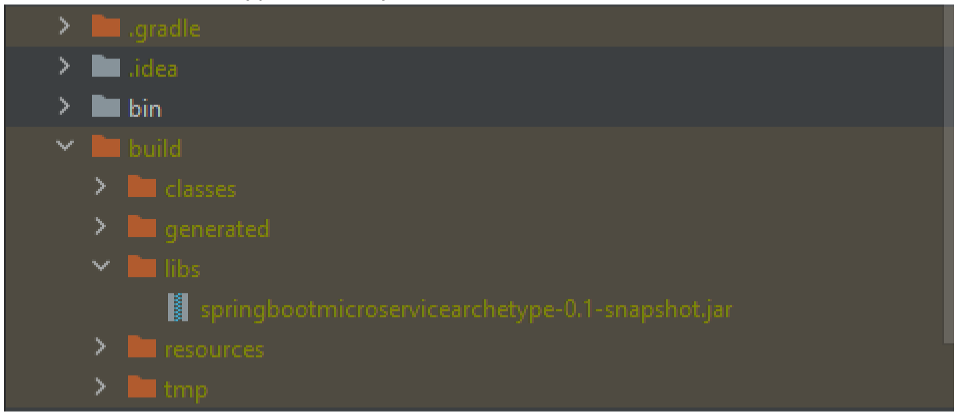
Note: this would differ from machine to machine

The final adjustment we need to make is to our dependencies found in **build.gradle**. We need to add the **eadp-springboot** library to the implementation list.



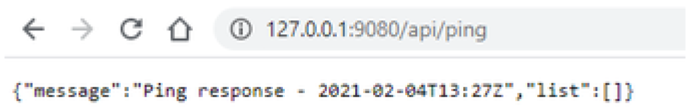
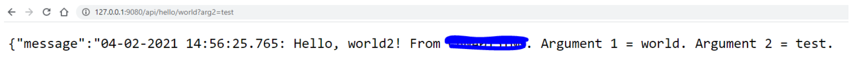
```
dependencies {
    implementation "sunlife.api:eadp-springboot:${eadpSpringbootVersion}"
    implementation "org.springframework.boot:spring-boot-dependencies:${springBootVersion}"
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    implementation 'org.springframework.boot:spring-boot-starter-cache'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-logging'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.security:spring-security-config'
    implementation 'org.springframework.cloud:spring-cloud-starter-sleuth'
    //implementation 'org.springframework.cloud:spring-cloud-starter-zipkin'
```

Now, we are going to build our API to make sure everything is set properly. Open your terminal, navigate to your project directory, and run `gradlew instatiateArchetype`. The built jar file can be found here:



```
> .gradle
> .idea
> bin
v build
    > classes
    > generated
    v libs
        springbootmicroservicearchetype-0.1-snapshot.jar
    > resources
    > tmp
```

Lets test our built service by running the command `gradlew springbootStart` in our terminal. By default, the application runs on port 9080. We can test the endpoints using the following:

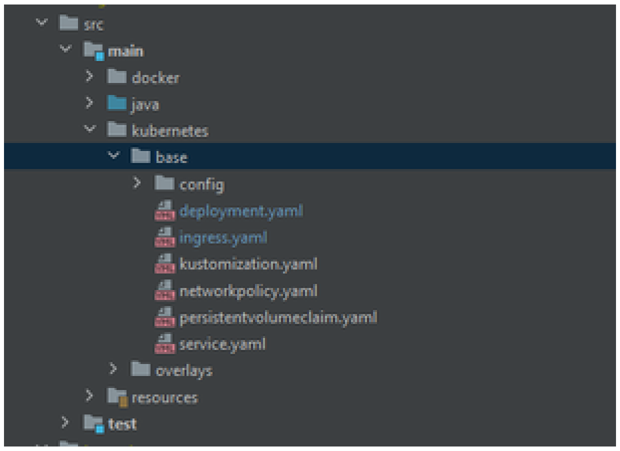
Get Request	Expected Response
`127.0.0.1:9080/api/ping`	 127.0.0.1:9080/api/ping `{"message":"Ping response - 2021-02-04T13:27Z","list":[]}`
`http://127.0.0.1:9080/api/hello/world?arg2=test`	 127.0.0.1:9080/api/hello/world?arg2=test `{"message":"04-02-2021 14:56:25.765: Hello, world2! From [redacted]. Argument 1 = world. Argument 2 = test.`

2. Containerizing Your API

Goal	• How to prepare an API for a containerized environment
Prerequisites	• *Creating a Simple API Using Spring Boot* must be completed • **Docker**

Now that we have our service running locally, let’s prepare the our API to run in a container. Since we are using the **spring-microservice-archetype**, the required files are are already generated for us when we built our API.

Let’s take a look at the kubernetes kustomize configurations in the **src/kubernetes** directory. Here you can see two subfolders, one called **base,** and another called **overlays**. Base is the base configuration required for our API to run, while overlays are different configuration options that fit certain 'scenarios'. Kubernetes kustomization is done this way so we can reuse configuration files.


```
src
  main
    docker
    java
    kubernetes
      base
        config
        deployment.yaml
        ingress.yaml
        kustomization.yaml
        networkpolicy.yaml
        persistentvolumeclaim.yaml
        service.yaml
      overlays
    resources
  test
```

Configuration File	Description
deployment.yaml	• Defines the object state of your API • Contains basic information (ie. name) • More information found here
ingress.yaml	• Defines the network traffic rules • How your API will be access externally • More information found here
kustomization.yaml	• Manages what configuration files your deployment will use • More information found here
networkpolicy.yaml	• Controls specific network flow at the IP and/or port level • More information here
persistentvolumeclaim.yaml	• Define a request for a specific amount of resources needed for your API • Requests can be made for specific levels of CPU and memory • Claims can request specific size and access modes
service.yaml	• Abstract definitions of logical sets of pods and by which to access them • HTTP port is 9080 by default • Actuator port is 9081 by default

Now that we have all the config files in place, and we have some understanding of what they do. Before we can build, we need to modify one option in our **Docker** file found in **src/main/docker/**. You will need to update the value for the tomcat image that is pulled from Sunlife's docker repos. For this tutorial, we will use the following option:

```
FROM prod-dtr-ca.sunlifecorp.com/whs/tomcatbase:2.3.4.release-
20201015141323
```

Note: Check to see if there are other/more recent images available here under *tags*

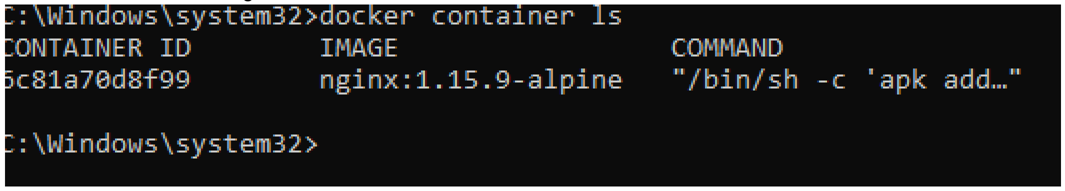
Now that we updated that value, lets try to build our docker container with our service. Make sure the **Docker** service is active, and run the following command:

```
gradew applyKubernetesVariantToLocal -Pvariant=local
```

Now, let’s check to see if our container is running correctly. You can do this by opening a new terminal, and running the command:

```
docker container ls
```

You should see something like this:



```
C:\Windows\system32>docker container ls
CONTAINER ID        IMAGE                 COMMAND
5c81a70d8f99        nginx:1.15.9-alpine   "/bin/sh -c 'apk add…"

C:\Windows\system32>
```

This means our container instance is running successfully. You can further validate by going to **127.0.0.1:9080/api/ping** just like in part 1 of our training.

---

3. Integrating Light4j Sidecar Proxy With Your API

Goal	• How to prepare an API for a containerized environment
Prerequisites	• *Creating a Simple API Using Spring Boot* must be complete • *Containerizing Your API* must be complete

Now that we have our basic API running on a containerized environment, we will need to make a few changes in order to integrate Light4j Sidecar Proxy completely. Luckily for us, the bulk of the changes have already been made for us in the **eadp-springboot** common library we imported earlier.

Most features implemented by Light4j require no changes to your API. But, there are changes required for some. If you want a more in-depth description of all the technical handlers Light4j Sidecar Proxy offers, you can go <<here>>

Light4j Sidecar Proxy Handler	Summary of changes required
Correlation Handler	• We need to accept **X-Correlation-Id** in our header • We need to change our logs to include the new header value
Traceability Handler	• We need to accept **X-Traceability-Id** in our header • We need to change our logs to include the new header value
HTTP Error Handler	• Our error codes need to be formatted to the API Platform **error code standard** • We need to make sure our API responds with the **eadp-common** response wrapper

Starting with the **correlation handler**, we will need to change what our endpoints accept as header parameters, as well as how they will be logged in our API.

1. Implement eadp-springboot
   - import the required modules to your springboot app
2. Modify your endpoints to accept Light4J Sidecar Proxy values
   - eg. X-Traceability-Id
3. Update your logs to reflect the additional light-proxy values
   a. mention sunlife logging standards
4. Link Light4j Side Proxy and your API through configurations
   a. proxy points to the business API
5. Test the local instance of your API with a local instance of light-proxy

    a. try out some requests and some invalid requests
    b. test light-proxy handlers
    c. showcase the features that business API did not have to implement

---

4. Preparing Your API For 'Bundled' Deployment

Goal	• Adding the required configuration files for deployment with Light4j Sidecar Proxy
Prerequisites	• *Integrating Light4j Sidecar Proxy With Your API* must be complete

1. Add K8s Folder
    a. show new file structure as a whole
2. Create Base and Overlay folders
3. copy the Kubernetes/Base files to your K8s/Base folder
4. Add api-config to your K8s/Base folder
5. Add proxy-config folder
6. configure values.yml and openapi.yml
    a. show the important values
    b. link to page that contains all optional values
    c. explain openapi and validator handler

---

5. Testing Your API

Goal	• How to test your API and view logs
Prerequisites	• *Preparing Your API For 'Bundled' Deployment* must be complete

1. Testing the endpoints of the API
2. Make a request and show response
    a. mention the format coming back
3. Make an invalid request
    a. mention the error code format
4. Log viewing
    a. Splunk
    b. Kubernetes Dashboard
    c. environment/system generated logs
    d. etc.
