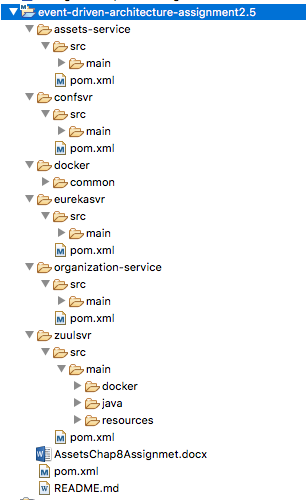
**Term Project Part 3, Assignment-3.2**

In the Assignment-3.2,you’ll examine different aspects of Microservices Event-Driven Architecture.

Below is the application directory structure.

****

**Use Case 1: Developing a simple message producer and consumer**

You’ll pass a message from your **organization service** to your **assets service**. The only thing you’ll do with the message in the assets service is to print a log message to the console.

In addition, because you’re only going to have one Spring Cloud Stream source (the message producer) and sink (message consumer) in this Use Case, you’re going to start this Case with a few simple Spring Cloud shortcuts that will make setting up the source in the organization service and a sink in the assets service trivial.

In the organization service’s Maven pom.xml file, add dependencies on spring-cloud-stream and spring-cloud-starter-stream-kafka.

In organization-service Application.java tell your application that it’s going to bind to a Spring Cloud Stream message broker by using annotation @EnableBinding.

Use class **SimpleSourceBean** to publish a message to a message broker.

Use class **OrganizationService** to publish a message in your organization service.

At this point, you need to have your assets service consume the message published by the organization service.

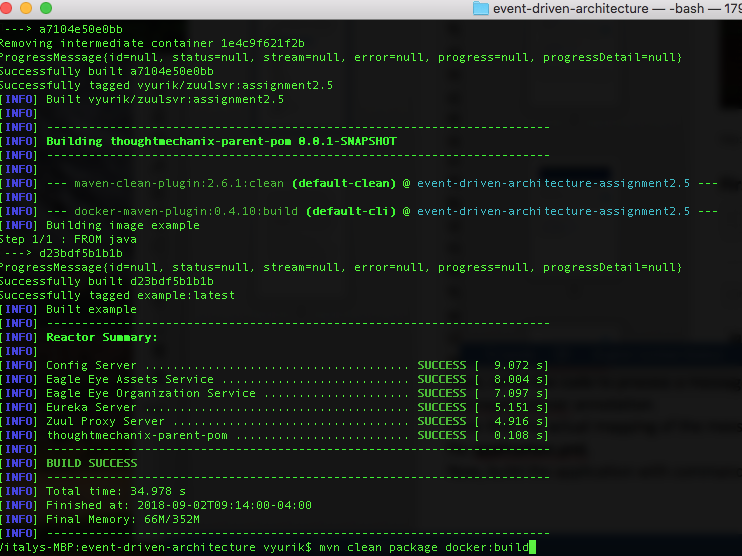
You need to add to your assets service’s Maven pom.xml file the same dependencies that you added to your organization service.

Like the organization service, you need to annotate the assets service Application.java with @EnableBinding, but here you need to pass Sink.class to the annotation.

Then, write the code to process a message coming off the Sink input channel. To do this, use the Spring Cloud Stream @StreamListener annotation.

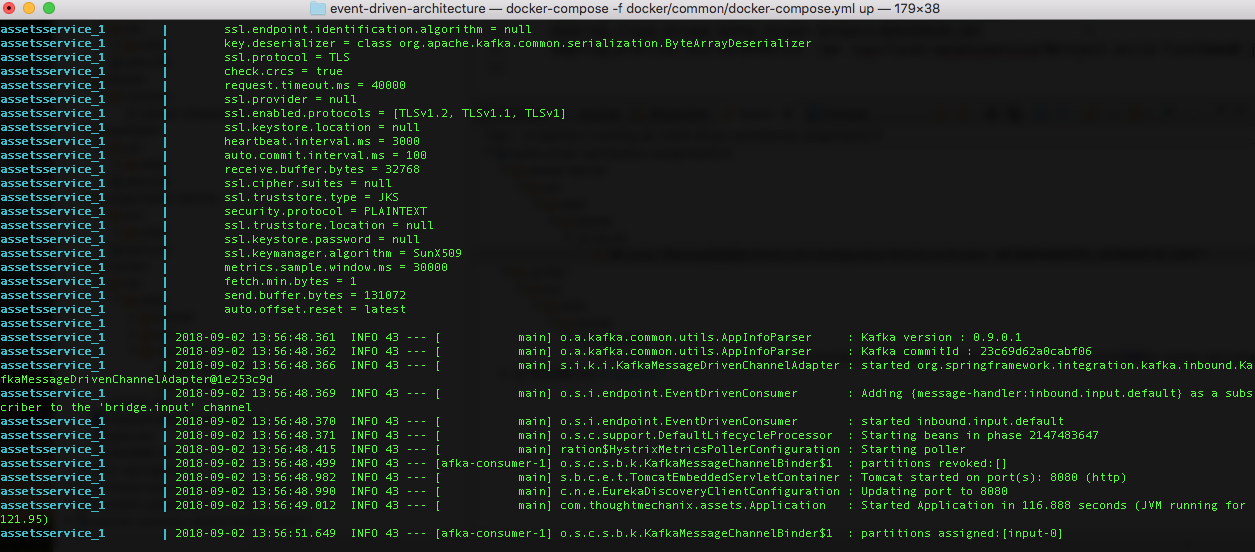
Note that the actual mapping of the message broker’s topic to the input channel is done in the assets service’s configuration file **application.yml.**

*Note: Asset Service Id must be different from Licensing Service Id used in the application examined in class*.Now, build the application with command: **mvn clean package docker:build**



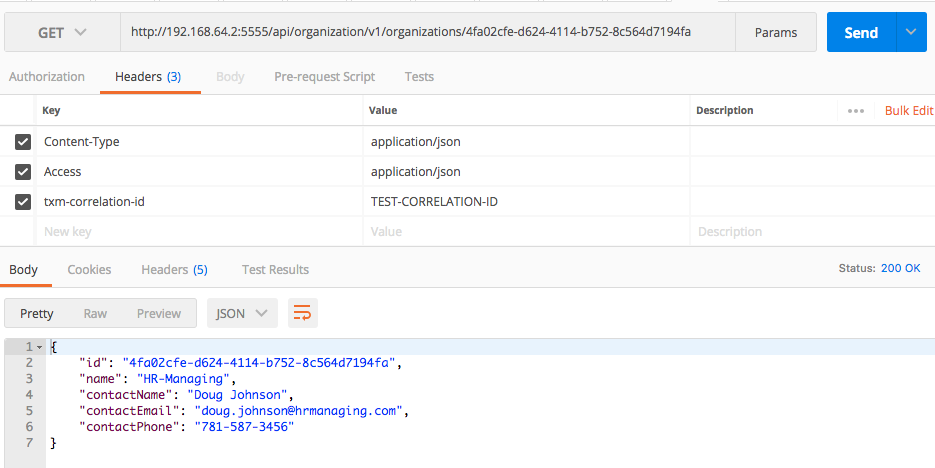
Run the application with command

**docker-compose -f docker/common/docker-compose.yml up**

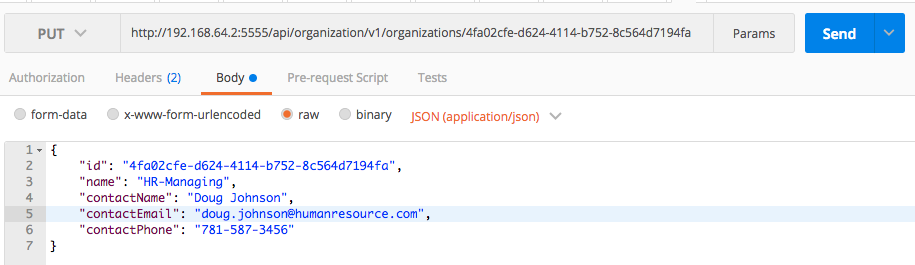
****

Verify that you can access this endpoint**:** [**http://192.168.64.2:5555/api/organization/v1/organizations/4fa02cfe-d624-4114-b752-8c564d7194fa**](http://192.168.64.2:5555/api/organization/v1/organizations/4fa02cfe-d624-4114-b752-8c564d7194fa)

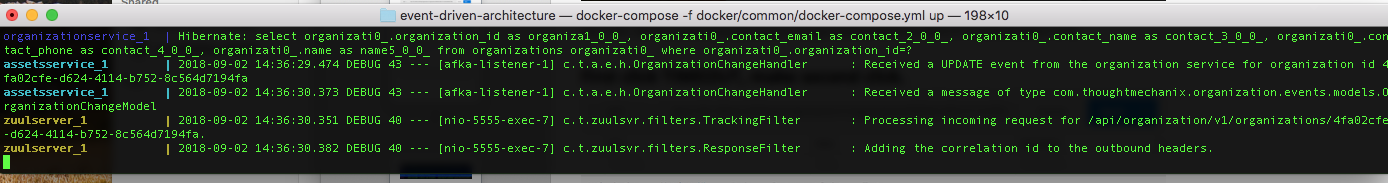
If you have on first click TIMEOUT, make second click.

****Select PUT command, then select Body link and paste an updated contact info. Change email

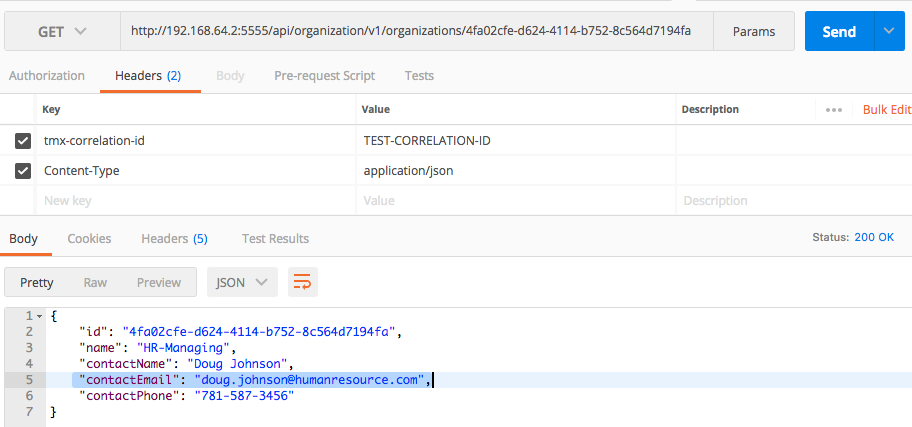
Make sure you selected raw radio button and JSON (application/json) option



The output shows that we updated the email address



Make sure the email has changed. On GET we return the changed email (several TIMEOUT, then the result)

****

**Use Case 2: Using Redis to cache lookups tmx-correlation-id →**a18bee153cc0cdfc

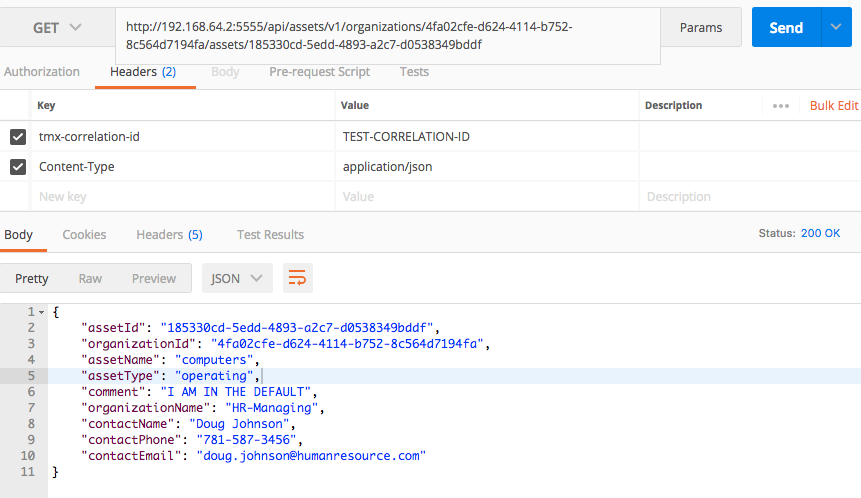
Set up Redis cache code as examined in the Class examined application.

Now**,** hit the asset service

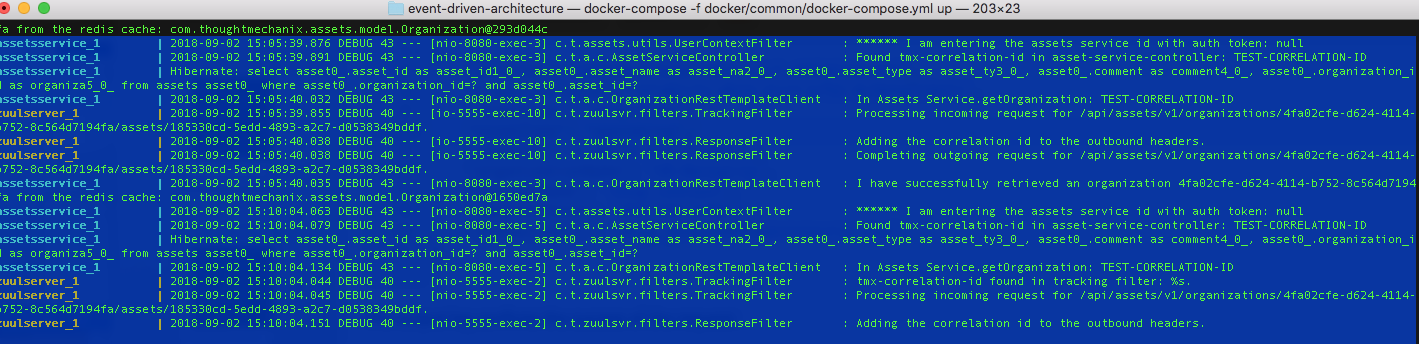
<http://192.168.64.2:5555/api/assets/v1/organizations/4fa02cfe-d624-4114-b752-8c564d7194fa/assets/185330cd-5edd-4893-a2c7-d0538349bddf>

After couple of TIMEOUT, you’ll see the response

http://192.168.64.2:5555/api/assets/v1/organizations/4fa02cfe-d624-4114-b752-8c564d7194fa/assets/185330cd-5edd-4893-a2c7-d0538349bddf



**The output log confirms the success of the request:**

****