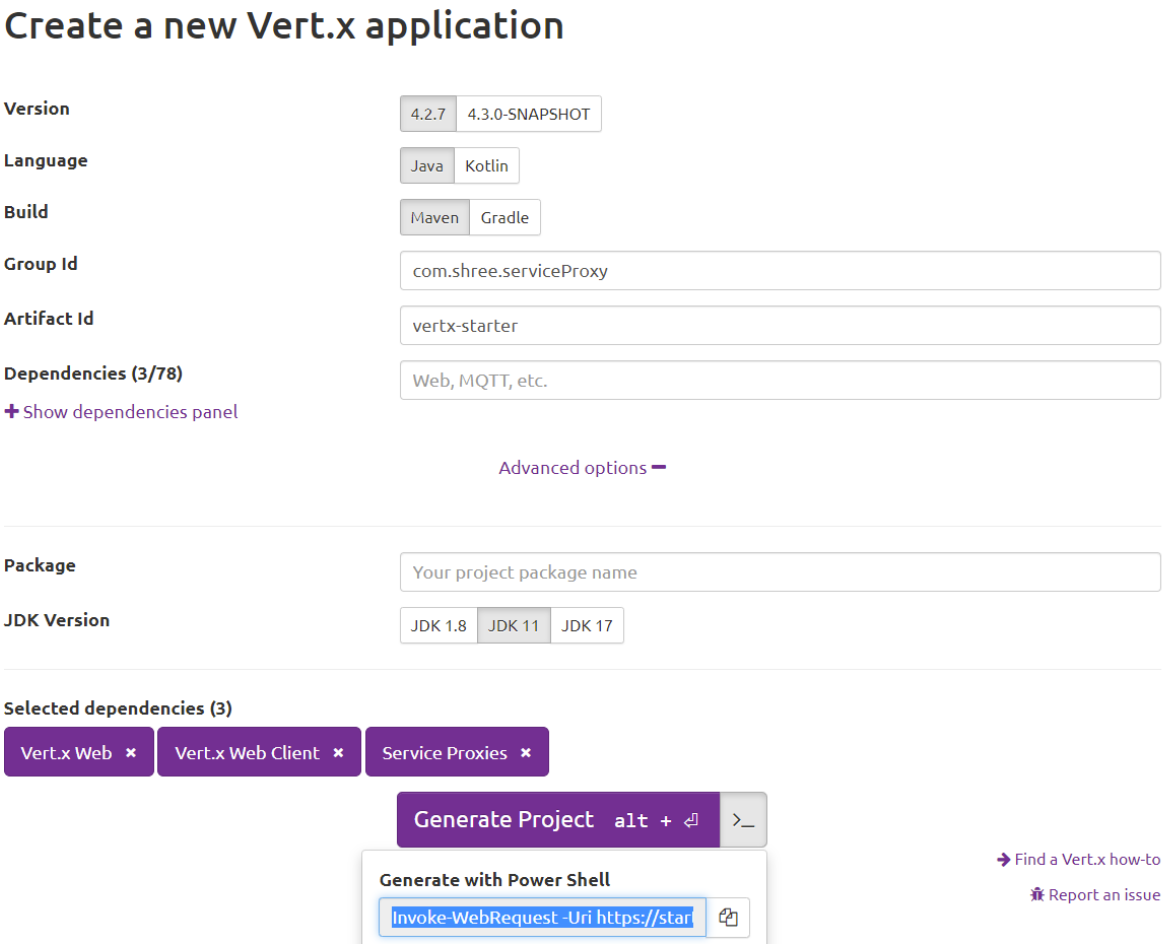
Vertx Service Proxy

Extra Links : https://how-to.vertx.io/service-proxy-howto/
http://rick-hightower.blogspot.com/2016/02/step-5-using-vertx-service-proxies.html

Generated the project from start.vertx.io



Got the zip folder by executing the command in Intellij Terminal

Invoke-WebRequest -Uri https://start.vertx.io/starter.zip -Body @{ groupId='com.shree.serviceProxy'; artifactId='vertx-starter'; vertxVersion='4.2.7'; vertxDependencies='vertx-web,vertx-web-client,vertx-service-proxy'; language='java'; jdkVersion='11'; buildTool='maven' } -OutFile vertx-starter.zip

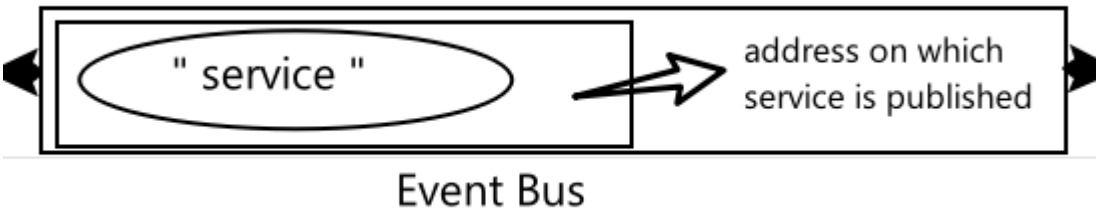
and unzipped it manually
Added service proxy as a dependency in the maven project

To do

- [ ] Pom.xml | bootstrap the application from vertx.io : ***Done***
- [ ] Psvm in the main verticle + deploy the verticle : ***Done***
- [ ] HTTP GET Endpoint : ***Done***
- [ ] use the router and attach it to the HTTP Server : ***Done***
- [ ] Write JUnit Test cases to check if add, sub, multiply and divide methods are working properly
- [ ] Package-info.java| package-info.java with @ModuleGen annotation : ***Done***
- [ ] @ProxyClose annotation : ***Done***
- [ ] service async interface : **Done**
- [ ] compile the source to generate the stub and proxies : ***Done***
- [ ] "register" your service on the event bus : ***Done***
- [ ] create a proxy : **Done**
- [ ] ServiceException.fail
- [ ] Client side error handling : ***Done***
- [ ] MainVerticle exposes the service | Output verticle does the calculations
- [ ] createProxy(Vertx vertx, String address) method in service interface : **Done**
- [ ] Implement the Interface methods : ***Done***
- [ ] Deploy it on Docker

Notes

- While creating a vertx application, isolate a functionality and make it available to the rest of your application. This is the main purpose of service proxy
-



- it lets you expose a *service* on the event bus, so, any other Vert.x component can consume it, as soon as they know the *address* on which the service is published.
- Service is described in Java Interface containing methods following async pattern (Future and Promise)
- To invoke the service, call proxy using API from Service interface (Under the hood messages are sent on Event Bus)
- To use proxies add dependency in the Maven's pom.xml
- In Vertx 3, you can instead use service proxies, which allow you to create code that looks more like a service and less like tons of callback registry with the event bus.
- You will also need a package-info.java file somewhere in (or above) the package your interface is defined in.

- Given the interface, Vert.x will generate all the boilerplate code required to access your service over the event bus, and it will also generate a client side proxy for your service, so your clients can use a rich idiomatic API for your service instead of having to manually craft event bus messages to send. The client side proxy will work irrespective of where your service actually lives on the event bus (potentially on a different machine).
- To be used by the service-proxy generation, the service interface must comply to a couple of rules. First it should follow the async pattern. To return a result, the method should declare a Future<ResultType> return type.
- You can also declare that a particular method unregisters the proxy by annotating it with the @ProxyClose annotation. The proxy instance is disposed when this method is called.
- Service annotated with @ProxyGen annotation trigger the generation of the service helper classes:
- The service proxy: a **compile time *generated proxy*** that uses the EventBus to interact with the service via messages
- The service handler: a **compile time *generated EventBus handler*** that reacts to events sent by the proxy
- Generated proxies and handlers are named after the service class, for example if the service is named MyService the handler is called MyServiceProxyHandler and the proxy is called MyServiceEBProxy.
- The codegen annotation processor generates these classes at compilation time. It is a feature of the Java compiler so no extra step is required, it is just a matter of configuring correctly your build:
- Return types must be one of:
- void
- @Fluent and return reference to the service (this):
- This is because methods must not block and it’s not possible to return a result immediately without blocking if the service is remote.
- You will build a Vert.x application that exposes and uses Service/Interface, an Event Bus Service that generates captures the input json value and manages calculation

Exposing your service

- Once you have your service interface, compile the source to generate the stub and proxies. Then, you need some code to "register" your service on the event bus:
- This can be done in a verticle, or anywhere in your code.
- Once registered, the service becomes accessible. If you are running your application on a cluster, the service is available from any host.

Proxy creation

- Now that the service is exposed, you probably want to consume it. For this, you need to create a proxy. The proxy can be created using the
- ServiceProxyBuilder
- The second method takes an instance of DeliveryOptions where you can configure the message delivery (such as the timeout).
- Alternatively, you can use the generated proxy class. The proxy class name is the service interface class name followed by VertxEBProxy.
- For instance, if your service interface is named SomeDatabaseService, the proxy class is named SomeDatabaseServiceVertxEBProxy
- Generally, service interface contains a createProxy static method to create the proxy. But this is **not** required:

Error Handling

- Service methods may return errors to the client by passing a failed Future containing a
- ServiceException
- instance to the method's Handler. A ServiceException contains an int failure code, a message, and an optional JsonObject containing any extra information deemed important to return to the caller.
- For convenience, the
- ServiceException.fail
- factory method can be used to create an instance of ServiceException already wrapped in a failed Future. For example:
- The client side can then check if the Throwable it receives from a failed Future is a ServiceException, and if so, check the specific error code inside. It can use this information to differentiate business logic errors from system errors (like the service not being registered with the Event Bus), and to determine exactly which business logic error occurred.

Restrictions for service interface

There are restrictions on the types and return values that can be used in a service method

Convention for invoking services over the event bus (without proxies)

Service Proxies assume that event bus messages follow a certain format so they can be used to invoke services.

- In order for services to be interacted with in a consistent way the following message formats must be used for any Vert.x services.

- There should be a header called action which gives the name of the action to perform.
- The body of the message should be a JsonObject, there should be one field in the object for each argument needed by the action.

Create a Vert.x Event Bus Service with Service Proxy

```
static BarmanService createProxy(Vertx vertx, String address) { // (5)
  return new BarmanServiceVertxEBProxy(vertx, address);
}
```

- Add @VertxGen and @ProxyGen to trigger code generation
- Specify the method that creates a new proxy.

Expose the service

- Now we can expose the service inside the BarmanVerticle with the ServiceBinder class
- Instantiate the BarmanServiceImpl
- Instantiate the ServiceBinder
- Configure the service address
- Register the service on the event bus

Use the service | Consume the Service (DrunkVerticle)

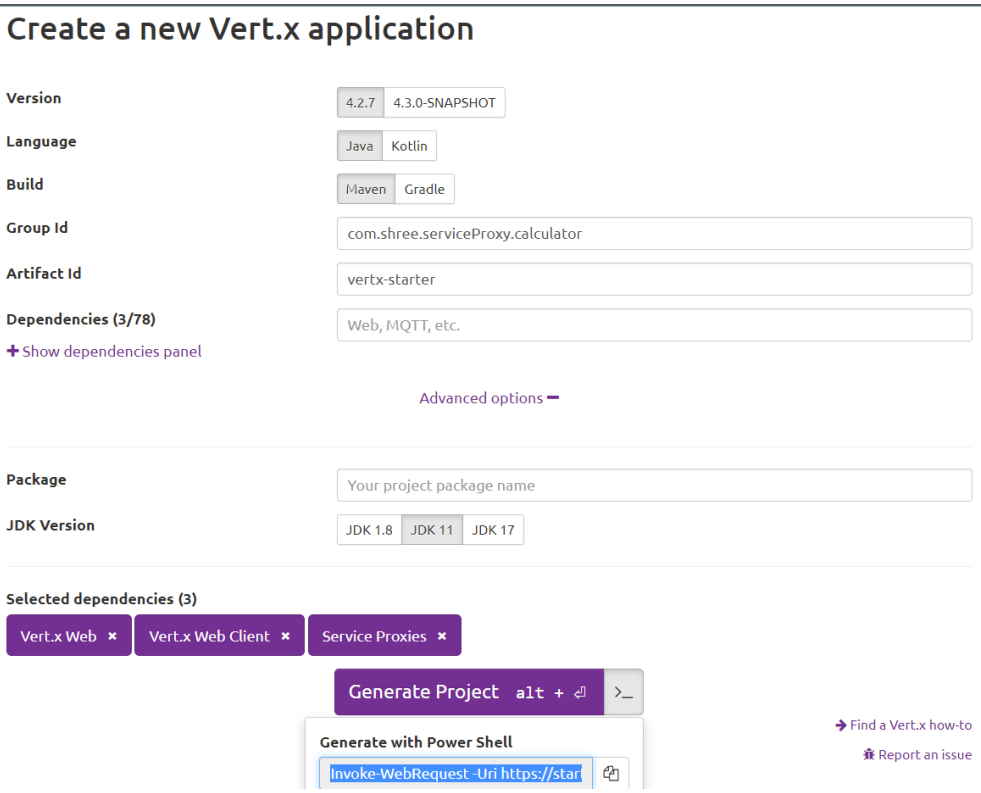
```
startPromise.complete();
```

- Instantiate the proxy
- Let's try the first beer 🍺
- Something went wrong during beer retrieval, fail the verticle start
- Drinking the first one 🍻
- Give me another one 🍺
- Drinking the second one 🍻
- Time to go home. Give me the bill
- Pay the bill

Running the application

- **To run the application deploy the BarmanVerticle and subsequently the DrunkVerticle**

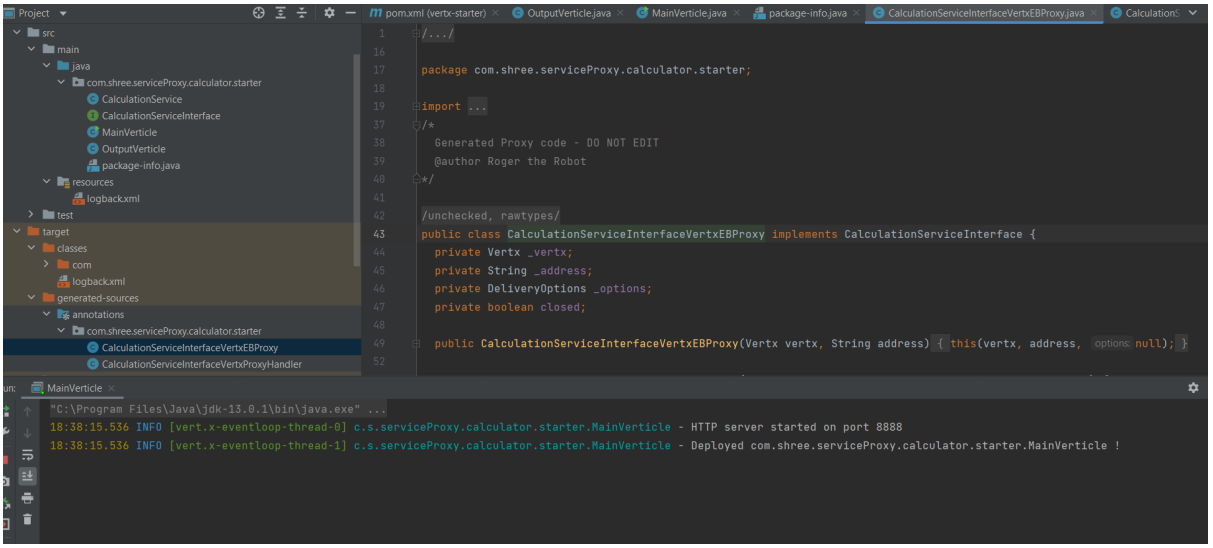
My Implementation



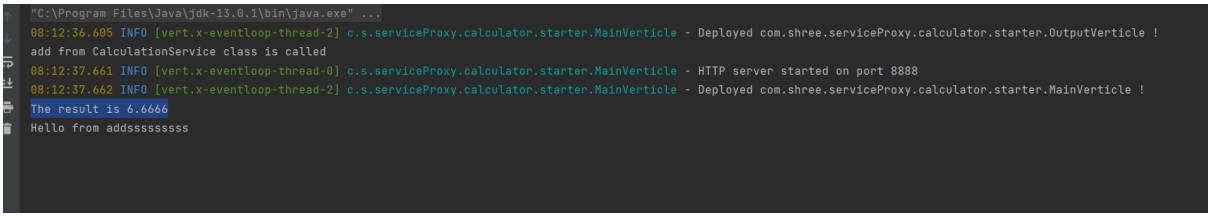
- Updating pom.xml by adding and logback classic dependency + adding logback.xml in resources folder under main folder

<dependency>
<groupId>io.vertx</groupId>
<artifactId>vertx-codegen</artifactId>
<classifier>processor</classifier>
</dependency>

- Create an interface which is a blueprint for all the calculations and close proxy
- To trigger the code generation, you must also add in the same package of the interface a package-info.java with @ModuleGen annotation:
- After the interface, implement the Class which implements interface / service
- I ran the main vertical and proxies are generated !
-



- To build a calculator application we want add, subtract, divide and multiply to be exposed to the event Bus
- Whenever there is error in EBProxy and VertxProxyHandler, do a mvn clean command in the terminal
- Or mvn clean package in the Intellij terminal
- There will be no errors in the auto generated proxy classes



Problem :
Initially I set up router to get HTTP endpoints and direct it to add method in the MainVerticle

```
pom.xml (vertx-starter) | OutputVerticle.java | MainVerticle.java | logback.xml | CalculationServiceInterface.java | package-info

    }

    @Override
    public void start(Promise<Void> startPromise) throws Exception {
      router.get("/add/:num1/:num2").handler(this :: add);
      router.get("/sub/:num1/:num2").handler(this :: sub);
      router.get("/mul/:num1/:num2").handler(this :: mul);
      router.get("/div/:num1/:num2").handler(this :: div);


      new ServiceBinder(vertx)
        .setAddress(ADD_ADDRESS)
        .register(CalculationServiceInterface.class, calculationServiceInterface);
```

I also added new ServiceBinder and registered it to event bus with ADD_ADDRESS

In output verticle a proxy is created, to call add method from the Implementation of the interface (CalculationService)

```
ServiceProxyBuilder builder = new ServiceProxyBuilder(vertx)
  .setAddress(MainVerticle.ADD_ADDRESS);

CalculationServiceInterface service = builder.build(CalculationServiceInterface.class);
service.add( num1: 3.0,  num2: 3.0, add1 -> {
  if(add1.failed()){
    LOG.error("Can't add :( because ", add1.cause());
    return;
  }
  System.out.println("The result is "+ add1.result());
});
```

Inside start method of Output Verticle

But before I write the HTTP request and hit enter, it had already registered the

ADD_ADDRESS to the event Bus through
ServiceBinder.setAddress(address).register(Interface.class)

And it goes to the OutputVerticle to

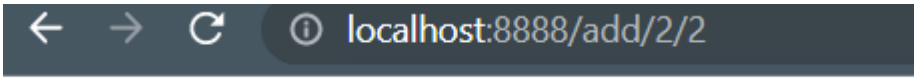
Build the service proxy through serviceProxyBuilderObject.setAddress(ADD_ADDRESS)
and it already called the service.add method and returned the result

Thennn | it started HTTP Service on port 8888 !
And got the result from ServiceImplementation class (CalculatorService)
And went to the add handler method in the MainVerticle to print “Hello from addssssss”

```
"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" ...
08:12:36.605 INFO [vert.x-eventloop-thread-2] c.s.serviceProxy.calculator.starter.MainVerticle - Deployed com.shree.serviceProxy.calculator.starter.OutputVerticle !
add from CalculationService class is called
08:12:37.661 INFO [vert.x-eventloop-thread-0] c.s.serviceProxy.calculator.starter.MainVerticle - HTTP server started on port 8888
08:12:37.662 INFO [vert.x-eventloop-thread-2] c.s.serviceProxy.calculator.starter.MainVerticle - Deployed com.shree.serviceProxy.calculator.starter.MainVerticle !
The result is 6.6666
Hello from addsssssssss
```

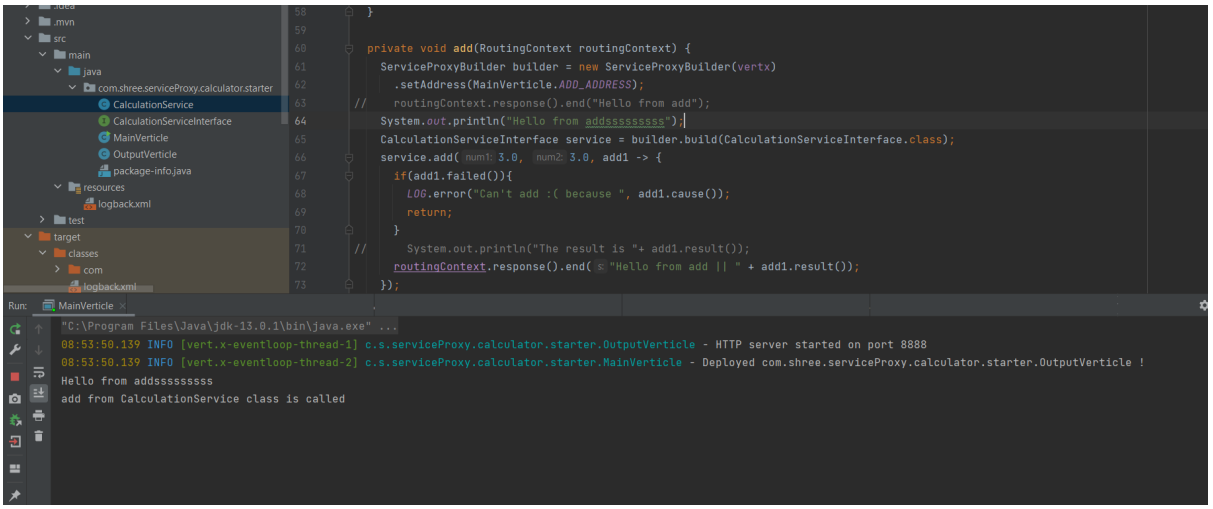
The problem here is it is calculating the result before HTTP server is started and does not take input from the User HTTP endpoint

How do I make it so that the service is called only after we get path parameters and return the result as JSON object



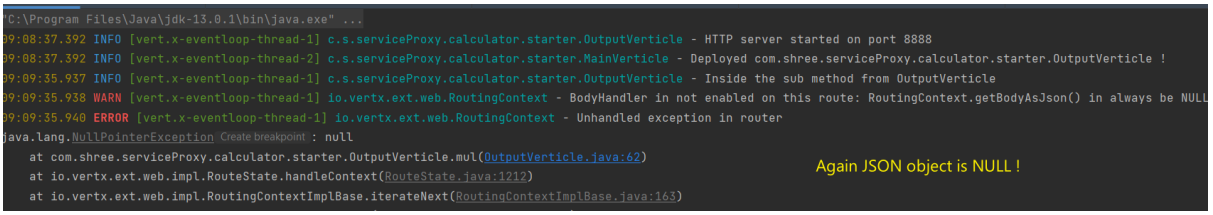
Resource not found

Now, I have added the router.get and the listener in the output verticle



Yaaay ! Now the add method is called after the HTTP request is entered by the user

Now, I'll use JSON objects only
No browser work from now on :)



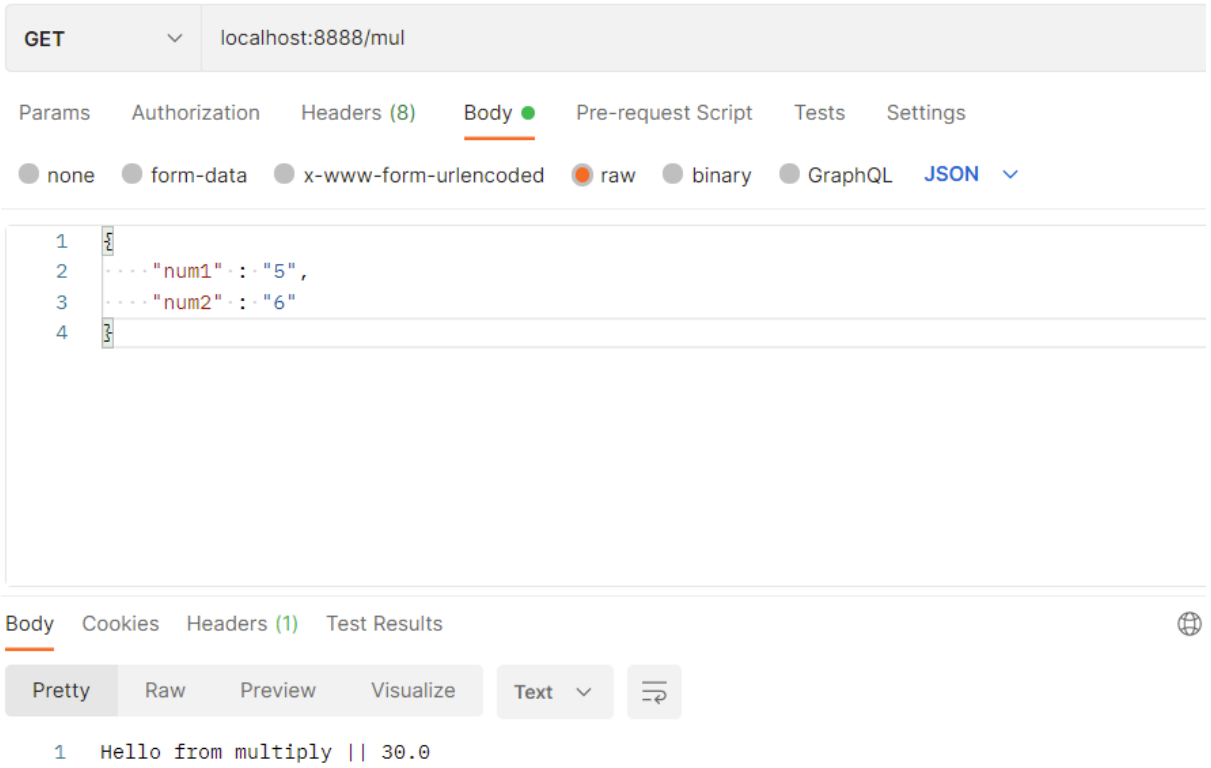
Problem : .getBodyAsJson is returning NULL !

What did I do ????

```
    router.get("/add/:num1/:num2").handler(this :: add);
    router.get("/sub").handler(this :: sub);
    router.get("/mul").handler(BodyHandler.create()).handler(this :: mul);
    router.get("/div/:num1/:num2").handler(this :: div);
//    router.route().handler(BodyHandler.create());
```

I added this line because the solution was given in this https://stackoverflow.com/questions/30858917/unable-to-access-request-body-using-getbodyasjson-in-vert-x-3-0-0

And it works fine :)



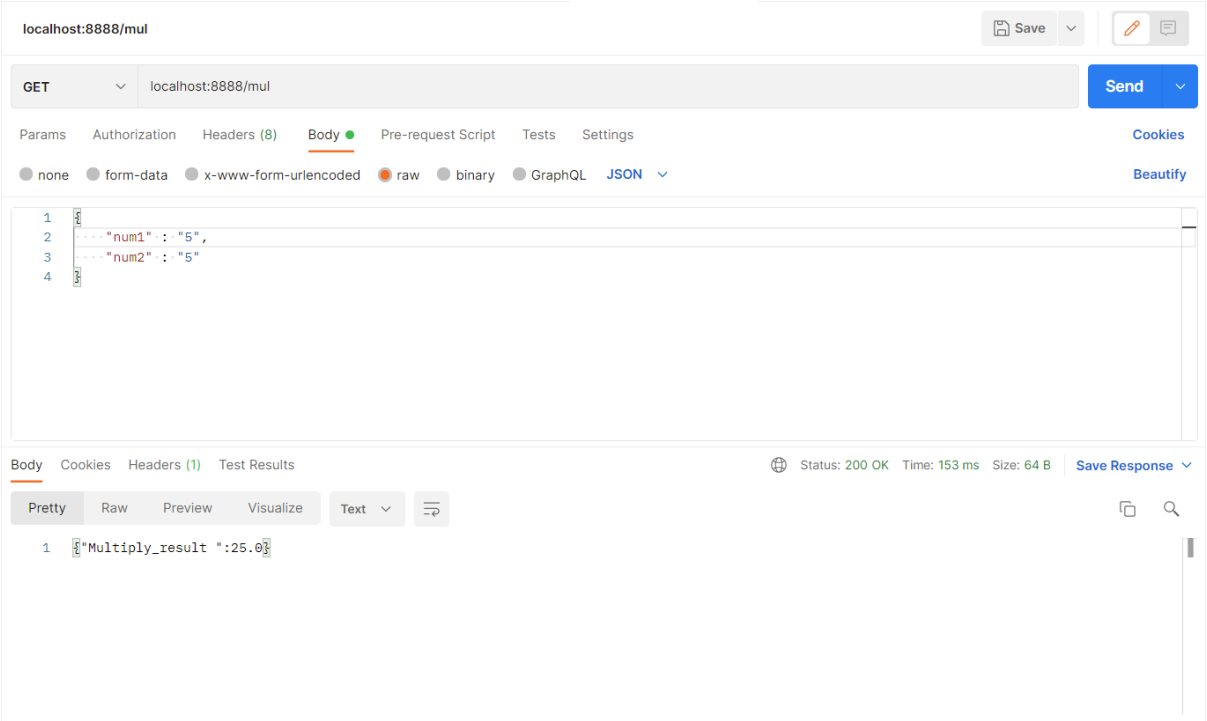
I also wanted to return JSON object from the add, multiply, subtract, method methods from the ServiceImpl class i.e., (CalculationService) but

```
@Override
public  void multiply(Double num1, Double num2, Handler<AsyncResult<Double>> resultHandler) {
  Double result = num1 * num2;
  System.out.println("multiply from CalculationService class is called");
  JsonObject json_result = new JsonObject();
  json_result.put("Multiply_result ", result);
  resultHandler.handle(Future.succeededFuture(json_result));
}
```

I should also change the return type to Handler<AsyncResult<JsonObject>> result handler And the proxy should have to be build once again ! and also there is restriction on the return type i.e.,
```

There are a couple of rules you must follow while defining the service interface. Look at Restrictions for service interface for more info. In our interface we are using a couple of primitives and `Beer`, a POJO annotated as `@DataObject`. If you want to use `@DataObject`s inside your interface, they must define both a constructor with only a `JsonObject` parameter and a `toJson()` method that returns a `JsonObject`
```

So, I'm not going to do it

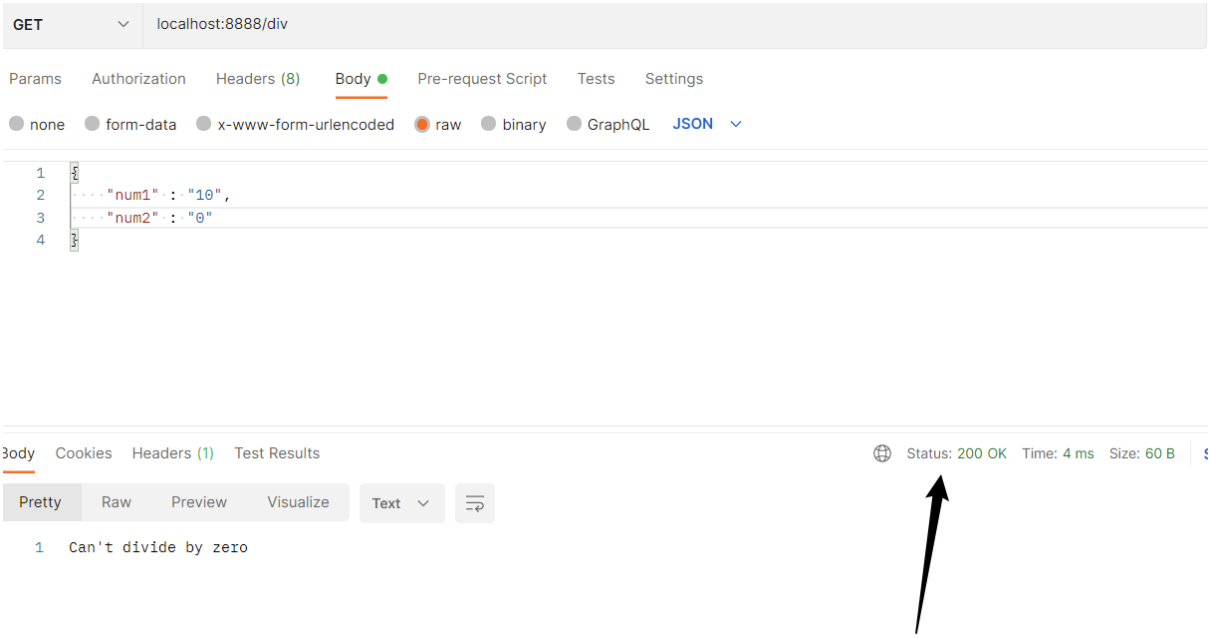


Got the expected result !

```
private void mul(RoutingContext routingContext) {
  ServiceProxyBuilder builder = new ServiceProxyBuilder(vertx)
    .setAddress(MainVerticle.MULTIPLY_ADDRESS);
  LOG.info("Inside the sub method from OutputVerticle");
  CalculationServiceInterface service = builder.build(CalculationServiceInterface.cla

  JsonObject jsonObject =  routingContext.getBodyAsJson();
  String num1 = jsonObject.getString( key: "num1");
  String num2 = jsonObject.getString( key: "num2");
  double num1_double = Double.parseDouble(num1);
  double num2_double = Double.parseDouble(num2);
  JsonObject json_result = new JsonObject();

  service.multiply(num1_double, num2_double, mul1 -> {
    if(mul1.failed())
    {
      LOG.error("Can't multiply :( because ", mul1.cause());
      return;
    }
    json_result.put("Multiply_result ", mul1.result());
//      routingContext.response().end("Hello from multiply || "+ mul1.result());
    routingContext.response().end(json_result.encode());
  });
}
```



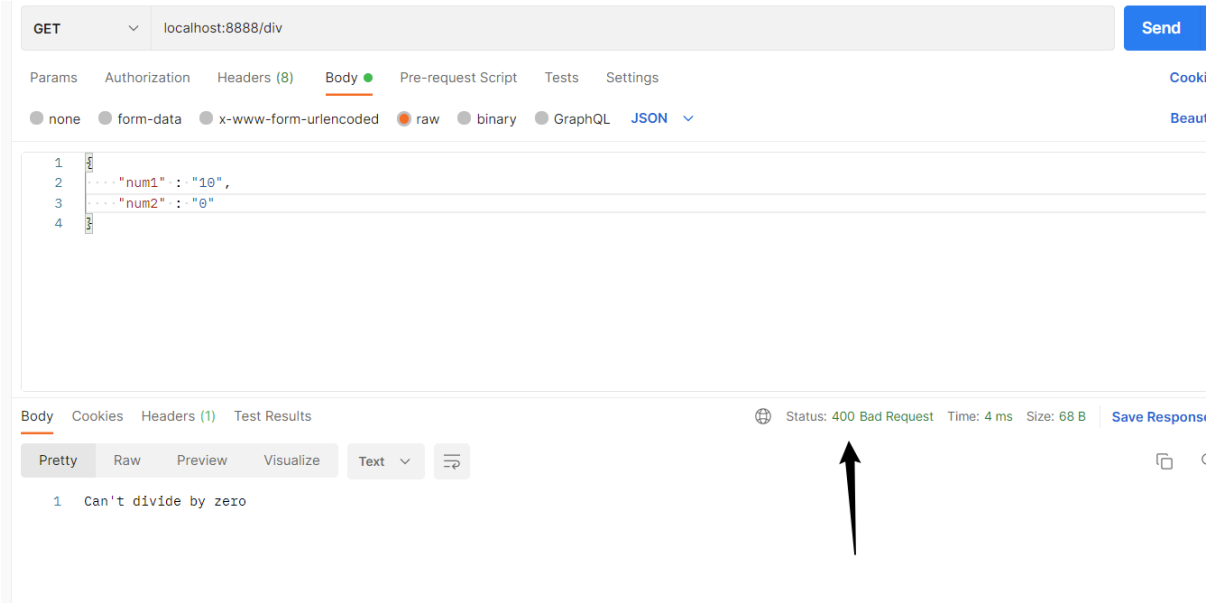
Setting HTTP Response code to 400 because it should indicate error

400 (Bad Request)

400 is the generic client-side error status, used when no other 4xx error code is appropriate. Errors can be like malformed request syntax, invalid request message parameters, or deceptive request routing etc.

The client SHOULD NOT repeat the request without modifications.

From https://restfulapi.net/http-status-codes/



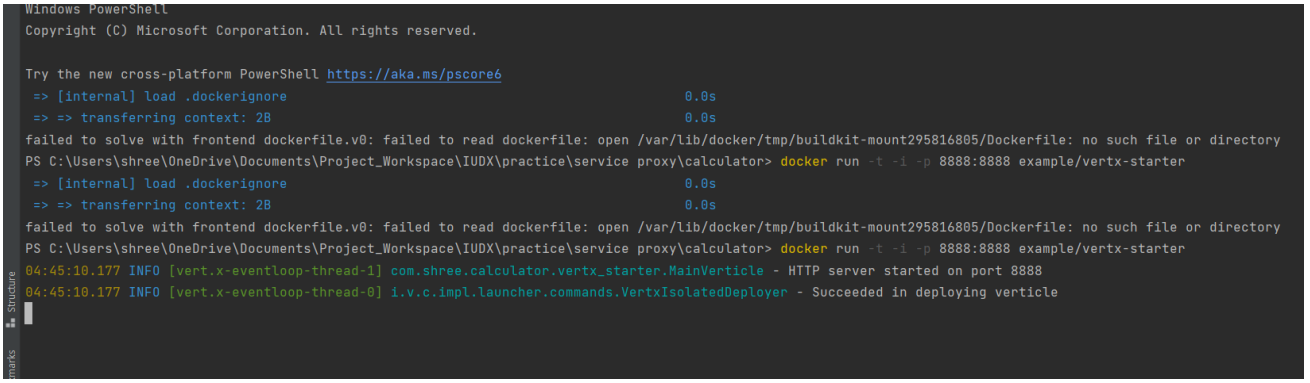
Done !

```
if(num2_double == 0.0)
{
  routingContext.response().setStatusCode(400).setStatusMessage("Bad Request").end( s: "Can't divide by zero");
      response().end("Can't divide by zero ");
  LOG.error("Division by zero error ");
  return;
}
```

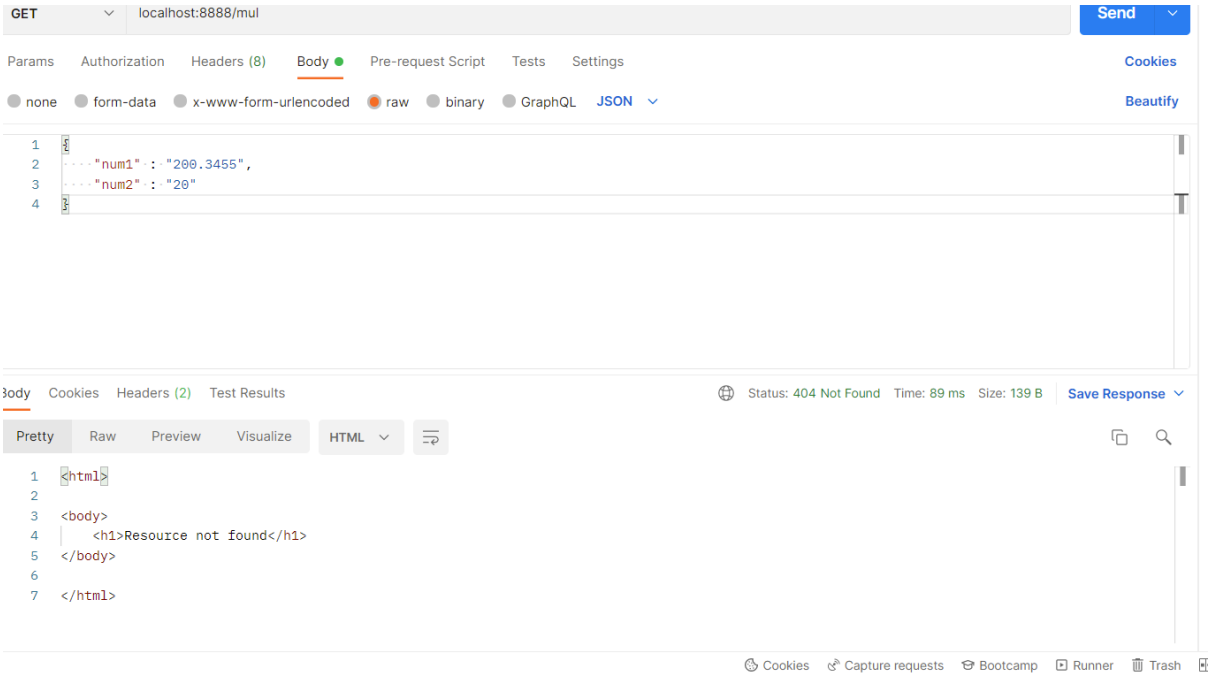
Got help from
Link : https://vertx.io/docs/apidocs/io/vertx/ext/web/client/HttpResponse.html#statusCode–

From Udemy course
“these days most of the web applications are running on cloud, the de facto standard is to bundle a docker container and deploy it”

“ Here we will see how to bundle a vert.x fat jar into a docker container”
Used the same Dockerfile from Udemy



Output after deploying on docker



Whenever I deploy it on docker, postman says the resource is not found even if the MainVerticle has started !

Anyways I’ll push this repository in GitHub now