# API Interaction

Vedang Sharma  25th of March 2020

## Executive Summary

The document proposes solutions to build a server application that accepts requests over the network and sends those requests to a second server. In turn the second server responds to the requests of the client via the first server. The service allows users to perform basic operations like multiplication, etc.

## PFS Review Status

| Date of Event | Event |
|---|---|
| 29-03-2020 | Document created by Vedang Sharma |
| | |

## Overview

We have to write a server application that accepts requests over the network and sends those requests to a second server. Have the second server respond to the requests of the client via the first server.

- The request should be a simple operation e.g. multiply two numbers.
- The application should be able to handle multiple concurrent requests.
- The exercise can be over engineered to show specific design choices.

### Use cases

As a User

- I can call the Network Service and perform multiplication operation
- I can perform several operations in parallel and get results for them.

## Proposal

We would build both the services using the Dropwizard framework. Following are some configurations which would be common to both the applications.

### Logging

Dropwizard provides support for logging out of the box. From their document:

> Dropwizard uses Logback for its logging backend. It provides an slf4j implementation, and even routes all `java.util.logging`, Log4j, and Apache Commons Logging usage through Logback.

Note: All the HTTP requests would be automatically logged in slf4j format. There is no need to do it separately. Although, we can add custom logs as per application requirements.

### Swagger

To simplify API development we would be integrating with the Swagger toolset. We would be integrating swagger bundles in both the servers using Smoketurner swagger implementation. How to use it?

### Common design pattern

We would be implementing **Builder Pattern** in the apps using Lombok. This Java library will help us by automatically creating Getter, Setter as well as implementing Builder for the POJO.

### HTTP response status codes

The server should always return the right HTTP status code to the client. Both the app would be returning the correct status code to the client based on JAX-RS Response.Status enum.

| | | |
|---|---|---|
| 200 | OK | Everything is working |
| 400 | Bad Request | The request was invalid or cannot be served |
| 500 | Internal Server Error | RuntimeException while processing request |

**Use Error payloads**

All exceptions should be mapped in an error payload which is an implementation of ErrorMessage from Jersey. Here is an example of what a JSON payload should look like.

```
{
  "message": "Exception from Data Server: Unknown Exception from Server",
  "code": 500
}
```

**Testing**

We would be using junit to implement unit testing on both the APIs. Additionally, we would also be using Mockito & Mockwebserver when needed to mock any resource or API calls respectively.

**Coverage**

We would add Jacoco's java maven plugin to generate coverage reports as a part of build steps.

## API Specification:

Since the purpose of this exercise is to show how servers interact we would be implementing the same specification in both APIs

**GET** /operation

Perform requested operation with the given query params

**Request:**

GET /operation?type=MULTIPLY&operand1=3&operand2=5

**Query Params:**

| | | | |
|---|---|---|---|
| type | Optional (default: MULTIPLY) | Enum | Type of operation: ADD, DELETE, MULTIPLY, SUBTRACT |
| operand1 | Mandatory | Double | |
| operand2 | Mandatory | Double | |

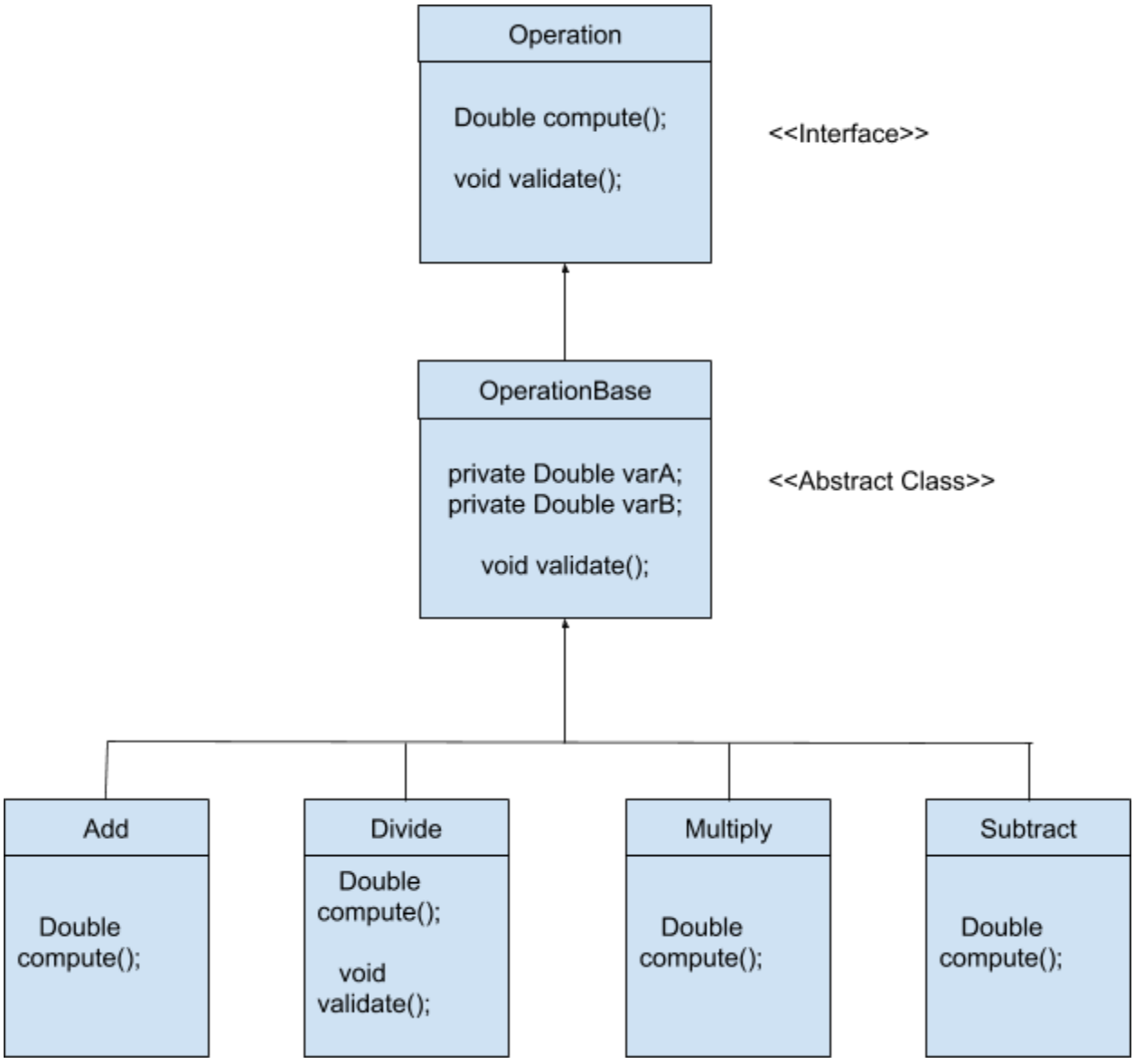
**Response:**

```
{
  "type": "MULTIPLY",
  "result": 15.0
}
```

| | | |
|---|---|---|
| type | Enum | Type of operation: ADD, DELETE, MULTIPLY, SUBTRACT |
| result | Double | Output of calculation |

## Data Service

To implement the handling of request and performing the requested operation we will use the [Factory Method Pattern]




## Network Service

We would be using a type-safe [Retrofit] client to call the DataServer API. This will help us in converting our HTTP API into a Java Interface. The Retrofit client in turn will be using [OkHTTP] client to perform the request to DataServer over HTTP.

**Error Handling:** We would be injecting a custom Interceptor on the OkHTTP client to intercept all the requests going through it and return the corresponding proper response to the client. We

can do something like

```
final static OkHttpClient HTTP_CLIENT = new OkHttpClient.Builder()
      .addInterceptor(new Interceptor() {
         @Override
         public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();
            Response response = chain.proceed(request);
            if (response.code() != 200) {
               throw new DataServiceException(response.message(),
                                    javax.ws.rs.core.Response.Status.fromStatusCode(response.code()));
            }
            return response;
         }
      })
      .build();
```