Back-End Developer Case Study

# Overview

These two small applications are a case study and both can easily be extended to work for larger input, different clients or to be parallelized in case it is necessary.

We have two different applications: a Java application, that acts as a master and it is the only one that can be run and a C++ DLL application, which acts as a library for the Java application. The Java application gets some dummy input and it invokes the C++ application for persistency, as required in the specifications.

# Java application

The Java application ensures receiving the events over HTTP, passes these events to the DLL C++ application fort persistency and it also acts as a dummy server part. In case of an extension of the application, the server and client part should be detached and they should work as two independent applications. In what follows, a short explanation of the code and how it implements each requirement is presented.

## 2.1 Application input

As presented, the input is dummy and done inside the application. I have used the event given as example in the specifications as the template event for this test case, but it can be extended to basically any type of events, as specified in the “.proto” file of the Google Protocol Buffers. For the current test case, the following “.proto” file model was used:

syntax = "proto2";

package company;

option java\_package = "com.company";

option java\_outer\_classname = "LogEvents";

message Events {

required int32 count = 1;

message Event {

required int64 timestamp = 1;

required int64 userId = 2;

required string event = 3;

}

repeated Event events = 4;

}

This file, after being compiled with the protoc compiler, produced a class called ‘LogEvents’, containing the Events class (the set of events sent by a user) and Event (a single event, as described in the example in the case study).

## 2.2 Client side

In Main.java, we have the start point of the application. The function main creates the server and then it creates the dummy input. Two similar dummy input sets are sent, one after creating the server and the other one after 20 s. For creating this sort of input, the static function demoSendPost was used (part of JsonProcesser class).

This function is responsible with the client side of the application. It first invoked the method ‘getDummyElements’, which creates our dummy input. We basically create a set of 10 user-defined objects (a list of dummy objects).

These objects represent the model of our application (class UserObject) and contains the three elements of our message type (timestamp, user id and the message). For our case study, timestamp is the current time as provided by the system (‘System.currentTimeMillis()’), the user id is hard-coded to 1 and a message changes with the index of our current loop.

In order to mimic sending of JSON messages, we converted this list of user objects to JSON objects. For each of the user objects, a key-set pair was created inside the JSON object HashMap. Each of these objects are added thereafter to the JSONArray object. This part could have been extended by reading user input from file in JSON format, but for the current case study I have chosen this simpler version.

The next part of the client function makes the HTTP connection part. We are building an HTTP client and we set the content type of its body to ‘application/json’ type. We pass the JSON array as a JSON string to the body of our request. The request is a POST request, connection to localhost, on port 8081, at /requests address. In order to confirm that the request was processed, after executing it, we are printing the response status code on the console.

## 2.3 Server side

In Main.java, we are also executing code for creating our server.In function ‘CreateServer’, we basically create an HttpServer, on port 8081, on localhost. We are using the same JsonProcesser class for processing, but this time we use an instance object of this class, which implements ‘HttpHandler’ interface. We then run our server on a separate thread, that can be terminated only by terminating the entire application.

Our server will wait for client requests and when the client sends its POST request to our server, ‘handle’ method is executed. Here, I read the requestBody (method ‘getRequestBody’) and after some stream processing, we get a StringBuffer with our content. In this case, as we made sure we sent the correct JSON format, we will parse it (‘parseJson’ call) and get the JSONArray object. In case wrong input is provided, I send an error code 400, with the message “Syntax error” (function ‘sendErrorCode’).

The next is to convert the json array to Google Protocol Buffers format. For this operation, we recreate our list of user objects in function ‘getListOfJSONObjects’ (for each JSONObject, we read from the mash map our fields and we build the UserObject) and pass it to an instance of GPBProcesser. This class is responsible with GPB operations and it will report if it is successful. If conversion succeeded, GPB buffers are sent to the C++ operation for saving on the disk, if it did not succeed, a 400 status code with the message “GPB did not succeed” is sent back to the client.

GPBProcesser class deals with the conversion to Google Protocol Buffers format and with sending these buffers to C++. Method ‘convertJSONToGPB’ makes the actual conversion from our list of user objects to GPB. We use a builder for Events and another one for one single event. For each user object, we instantiate the fields of our event with the corresponding UserObject fields and we add these events to the Events builder. We also count each added element in Events. After processing all UserObject objects, we build the Events and store the Events object I our instance of GPBProcesser.

GPBProcesser class is also responsible with sending information to C++ part. I have used JNI as communication protocol. I have two native methods for this part, one is startListening, which is called at the beginning of the application, to inform C++ that it should start listening to list of events that may come (we basically start the C++ DLL, as we can see in the following sections).

Method ‘passMessagesToCpp’ is used to actually pass the Events to C++. I have chosen to use ‘toByteArray’ method, thus sending the buffer in its byte serialized version.

## 2.4 Other libraries

For creating this project, other libraries were used. I will mention here ‘com.google.protobuf’, used for Google Protocol Buffers, commons-logging, http-client jar file from Apache, httpcore jar from Apache and also json-simple.

# C++ application

The C++ application is designed as a DLL application, used as a library by our Java application. It only takes the byte array sent by Java in Google Protocol Buffers and prints the events to a file.

## 3.1 Application input

For the C++ part, our input is the native function call through JNI and the actual proto file.

For the proto file part, I have used the same file as for the Java application. Based on that model file, by running it through the compiler, We have obtained ‘log.pb.h’ and ‘log.pb.cc’ files. It generated our two classes, called here ‘Events’ and ‘Events\_Event’ for a single event, which also contains our required timestamp, user id and message fields.

The other input comes from Java and is the native function calls. ‘Java\_com\_company\_GPBProcesser\_startListening’ is the function that makes the C++ application to start listening. This function starts a new thread, which will execute the loop function. ‘loop’ function is just an infinite loop that, based on the process static bool variable, decides when to start processing and when to stop processing. Whenever some new input arrives, process is set back to false and no other client should process until further notice. This part of the code should be protected from synchronization issues in a further improvement of the application.

The actual input from Java comes through ‘Java\_com\_company\_GPBProcesser\_passMessagesToCpp’, which contains the byte array sent by Java, in the form of jByteArray. This function will actually trigger processing.

## 3.2 C++ application’s functionality

Inside the native call mentioned above, we process the jByteArray. Using the ‘env’ pointer provided by the native function, I create a normal (native) array with ‘GetByteArrayElements’ function. This will create our jbyte array, which can then be used for processing. We save this array to our GPBProcesser instance and mark process to true to allow starting processing.

GPBProcesser class is a simple class responsible for saving the byte array between native function calls , processing the GPB input and writing input to file. After saving the byte array, processing will start and ‘vProcessGPBInput’ function is called. In case we have a valid byte array, we create an output string stream of binary type and, for each non-null character in our byte array, we write the byte to our string and count it. When \0 character is encountered, our byte array is finished. If our stream is ok, we create an input stream for Google Protocol Buffers parser. We create an Events object and we call its ‘ParseFromArray’ function. As specified, this function will create our events from the binary format that was sent by Java application. The format is good in this case, as the C++ application is called only from our Java application. In future, further checks should be done to enforce correct parsing.

## 3.3 Application output

The last part of our application is writing to file. I have decide to append user logs to the same file, for a better storage. This is what function ‘appendEventsToFile’ actually does. In file “test.txt”, events are appended. We get a list of ‘RepeatedPtrField’ objects and we read each element in turn in order to print its content to file.