Generating and Running Code in Different Languages (Java, Python, PHP, Javascript)

**Java**

Java is another programming language, which is extremely popular these days. Used not only for desktop applications, but also in the web applications in the embedded devices or on the mobile phones and tablets (Android being a notable case), Apache Thrift is used in all of those scenarios.

**Generating the code**

Apache Thrift's compiler offers lots of options for Java. Run the following command to see them:

**$ thrift --help**

Look for the information about Java generators:

**java (Java):**

**beans: Members will be private, and setter methods will return void.**

**private-members: Members will be private, but setter methods will return 'this' like usual.**

**nocamel: Do not use CamelCase field accessors with beans.**

**fullcamel: Convert underscored\_accessor\_or\_service\_names to camelCase.**

**android: Generated structures are Parcelable.**

**android\_legacy: Do not use java.io.IOException(throwable) (available for Android 2.3 and above).**

**java5: Generate Java 1.5 compliant code (includes android\_ legacy flag).**

**reuse-objects: Data objects will not be allocated, but existing instances will be used (read and write).**

**sorted\_containers:**

**Use TreeSet/TreeMap instead of HashSet/HashMap as an implementation of set/map.**

Some of the options may cater to your project's specific needs and they are self-explanatory.

You need to pick the options from the list to match your project's needs. It is possible to use more than one, for example, if you would like your code to have private members and have the structures parsable, use:

**$ thrift --gen java:private-members,android myfirst.thrift**

Be aware that you won't get any error message if you misspell the parameter; it will just be ignored. It's hard to debug such a situation, so just check your parameters twice.

**Examining the code**

The code that you have generated will be in the gen-java directory, or if you chose the beans option, then in the gen-javabean directory. Inside the directory, there will be a namespace structure (if you chose any) containing the Java files for each of the complex variable types (such as structs, exceptions, enums, and so on), constants, and services that you declared.

You can examine the files containing the variables and constants to see an elaborate implementation of the behavior intended by Apache Thrift.

Now, let's have a look at the service file which, in the case of our example, is MyFirstService.java. It contains the MyFirstService class. This class contains other important interfaces and classes, notably the Iface interface:

public interface Iface {

public void log(String filename) throws org.apache.thrift. TException;

public int multiply(int number1, int number2) throws org.apache. thrift.TException;

public int get\_log\_size(String filename) throws MyError, org.apache. thrift.TException;

}

An interface is a description of the elements that the class which extends it has to have. Therefore, you need to implement all of the methods of your interface for the service to exist; in this case the log, multiply, and get\_log\_size methods you declared in your Apache Thrift document.

The MyFirstService class also contains the Client subclass, which implements the Iface interface which you will use in your client application.

**Transports**

There are a lot of transports available in the Java implementation of Apache Thrift. You can look them up in the lib/java/src/org/apache/thrift/transport directory. There are all of the most popular transports.

There are also a few nonstandard transports that you may find particularly useful. They are:

1. TFastFramedTransport: This transport is compatible with TFramedTransport, but offers some optimizations which are beneficial in terms of memory usage, when your messages are of more-or-less similar size.
2. TSaslClientTransport and TSaslServerTransport: Those are layered transports for client and server that are used if you need to provide security through the **Simple Authentication and Security Layer** (**SASL**) framework.

If you intend to use one of the transports, I suggest you examine the implementation to be sure how it works and if it fits your specific needs.

**Protocols**

In addition to the standard protocols, which were discussed in *Chapter 4*, *Understanding How Apache Thrift Works*, Java implementation of Apache Thrift offers TMultiplexedProtocol, which is a decorator that helps you deal with multiplexing—multiple services on one server.

Implementations of the protocols are in the lib/java/src/org/apache/thrift/ protocol directory, where you can examine them before using them.

**Servers**

The Java implementation of Apache Thrift offers all of the basic servers mentioned in *Chapter 4*, *Understanding How Apache Thrift Works*. Additionally, there are some other servers that you might find useful.

The first of them is TThreadPoolServer, which uses Java's Thread Pools to manage the workers.

Two others are implementing the Half-Sync/Half-Async architectural pattern to provide concurrent I/O, which combines asynchronous handling of connections with synchronous processing of the following requests:

1. THsHaServer: This is an extension of TNonblockingServer.
2. TThreadedSelectorServer: This performs better than THsHaServer in multicore environments.

You can examine the implementations of the servers in the lib/java/src/org/ apache/thrift/server directory.

**Implementing and running the service**

Implementing the service is done by creating a handler class, which implements the interface that was generated by Apache Thrift; in our case, it is MyFirstService. Iface. Then the server needs to be created.

The following source code indicates how to perform the first step. For brevity, the implementation details of the methods were omitted:

import org.apache.thrift.TException;

// import code generated by Apache Thrift compiler

import myfirst.\*;

public class MyFirstHandler implements MyFirstService.Iface {

public MyFirstHandler() {

}

public void log(String filename) {

// implementation omitted

}

public int multiply(int number1, int number2) {

// implementation omitted

}

public int get\_log\_size(String filename) {

// implementation omitted

}

}

Let's save this handler to a file, that is, MyFirstHandler.java.

The second part is to create the server:

import org.apache.thrift.server.TServer;

import org.apache.thrift.server.TServer.Args;

import org.apache.thrift.server.TThreadPoolServer;

import org.apache.thrift.transport.TServerSocket;

import org.apache.thrift.transport.TServerTransport;

// import code generated by Apache Thrift compiler

import myfirst.\*;

public class MyFirstServer {

public static MyFirstHandler handler;

public static MyFirstService.Processor processor;

public static void myserver(MyFirstService.Processor processor) {

TServerTransport serverTransport = new TServerSocket(8080);

TServer server = new TThreadPoolServer(new TThreadPoolServer. Args(serverTransport).processor(processor));

server.serve();

}

// main function

public static void main(String [] args) {

handler = new MyFirstHandler();

processor = new MyFirstService.Processor(handler);

Runnable server = new Runnable() {

public void run() {

myserver(processor);

}

};

new Thread(server).start();

}

}

Save it to the file, for example, MyFirstServer.java. Now, you can compile the code and run the server:

**$ javac MyFirstServer.java**

**$ java MyFirstServer**

**Implementing and running the client**

Running the client using the Java implementation of Apache Thrift is easy: you need to create a proper environment (transport, protocol) and use the client (in our case MyFirstService.Client), which exposes the service's methods.

Here's the code example:

// Import code generated by Apache Thrift compiler

import myfirst.\*;

import org.apache.thrift.transport.TTransport;

import org.apache.thrift.transport.TSocket;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

public class MyFirstClient {

public static void main(String [] args) {

TTransport transport = new TSocket("localhost", 8080);

transport.open();

TProtocol protocol = new TBinaryProtocol(transport);

MyFirstService.Client client = new MyFirstService. Client(protocol);

// call remote functions

client.log("logfile.log");

System.out.println(client.multiply(14,3));

transport.close();

}

}

You can embed your client's code in your application or just save it to the MyFirstClient.java file, compile, and run:

**$ javac MyFirstClient.java**

**$ java MyFirstClient**

**Python**

Python is used universally in server scripting, web and desktop applications, networking, natural language processing, statistical analysis, machine learning, and lots of other applications. This makes it a great tool for distributed applications with strong support in Apache Thrift.

**Generating the code**

Apache Thrift's compiler offers lots of options for Python. Run the following command to see them:

**$ thrift --help**

Look for the information about Python generators:

**py (Python):**

**new\_style: Generate new-style classes.**

**twisted: Generate Twisted-friendly RPC services.**

**tornado: Generate code for use with Tornado.**

**utf8strings: Encode/decode strings using utf8 in the generated code.**

**slots: Generate code using slots for instance members.**

**dynamic: Generate dynamic code, less code generated but slower.**

**dynbase=CLS Derive generated classes from class CLS instead of TBase.**

**dynexc=CLS Derive generated exceptions from CLS instead of TExceptionBase.**

**dynimport='from foo.bar import CLS'**

**Add an import line to generated code to find the dynbase class.**

Some of the options may cater to your project's specific needs. Here's a brief explanation of the most important of them, which we will use:

new\_style: The classes will be generated in Python's "new style", which boils down to inheriting from the object (for more information about new style classes in Python, refer to the Python wiki: https://wiki.python.org/ moin/NewClassVsClassicClass)

twisted: The generated code will be compatible with the Twisted asynchronous networking framework (https://twistedmatrix.com/)

tornado: The generated code will be compatible with the Tornado framework (http://www.tornadoweb.org/)

utf8strings: It's an important option when developing applications in languages other than English that all the strings are properly encoded and decoded using the UTF-8 codec

slots: Python's slots are used for instance members (see https://docs. python.org/2/reference/datamodel.html#slots) to save space when creating multiple objects

You need to pick the options from the list to match your project's needs. It is possible to use more than one, for example, if you need your code to have new style classes and have the strings encoded and decoded in UTF-8, run the following command:

**$ thrift --gen py:new\_style,utf8strings myfirst.thrift**

Watch out, because you won't get any error message if you misspell the parameter; it will just will ignored. It's hard to debug such a situation, so just check your parameters twice.

**Examining the code**

The code that you generated will be in the following directory:

1. gen-py.twisted: If you wanted to have the Twisted-compatible code (twisted option)
2. gen-py.tornado: If you generated the Tornado-compatible code (tornado option)
3. gen-py: In any other case

Inside the directory, there will be a namespace structure (if you chose any) containing two Python files for each of the services that you declared. Separately, there is a ttypes.py file, which contains the variables generated from your definition and a constants.py file, which contains all of the constants. You can note that while consts are defined very simply as a variable (there are no constants in Python), the variables are elaborate with a huge amount of code to handle them. Note that there are the \_\_ init\_\_.py files present, which make a module out of the code that was generated.

Let us have a look at the service files. In our example, the MyFirstService.py file contains the Iface class (because in Python there is no equivalent to interfaces known from other object-oriented programming languages):

class Iface:

def log(self, filename):

"""

Parameters:

- filename

"""

pass

def multiply(self, number1, number2):

"""

Parameters:

- number1

- number2

"""

pass

def get\_log\_size(self, filename):

"""

Parameters:

- filename

"""

pass

In your implementation of the service, you are going to define all the methods, which currently do nothing (just pass); in this case the log, multiply, and get\_log\_size methods you declared in your Apache Thrift document. There are annotations that will suggest you what parameters the methods will take, however, contrary to the implementation for other languages, the suggestions for the types of parameters and return value are missing.

Another important element in the MyFirstService.py file is the Client class. You will use this class in your client script to call the service and run remote methods. In fact, the creators of the Python implementation went an extra step forward and have the compiler to generate a file with the service name suffixed with -remote (in our case, MyFirstService-remote). This script contains an implementation of the client; more on that later.

**Transports**

In addition to the most popular transports described in *Chapter 4*, *Understanding How Apache Thrift Works*, the Python implementation of Apache Thrift offers some other transports as well. You can look up the list in the code repository in the lib/py/src/ transport directory. Some of the basic transports are the classes in the TTransport. py file, while the others are in separate files.

Some of the nonstandard transports you may find particularly useful are:

TMemoryBuffer: The wrapper for the StringIO object, so you are able to write to or read from the memory; if you pass the value in the constructor it will be a transport for reading; otherwise, for writing

TSSLSocket: Used for creating sockets wrapped in the SSL security layer

TSaslClient: Similarly to in Java, this is a layered transport for the client which is used if you need to provide security through the SASL framework

TZlibTransport: A layered transport that compresses the transport that it gets using Python's zlib library

THttpClient: Implements TTransportBase to provide communication over HTTP or HTTPS

For THttpClient, use URI in the constructor:

THttpClient('http://host:port/path')

So, for example, if the server is running on the localhost, on port 8080 and its path is /MyFirstServer, the instantiation of the client class would be:

client = THttpClient('http://localhost:8080/MyFirstServer')

If you intend to use one of those transports, I suggest you examine the implementation to be sure how it works and if it fits your specific needs.

**Protocols**

Python implementation of Apache Thrift offers all of the standard protocols mentioned in *Chapter 4*, *Understanding How Apache Thrift Works*. Additionally, it provides TMultiplexedProtocol, which is not a standalone protocol, but a decorator that helps you deal with complex scenarios when you want to use multiple services on one server (called **multiplexing**).

Implementations of the protocols are in the lib/py/src/protocol directory, where you can examine them before using.

**Servers**

The Python implementation of Apache Thrift contains all of the popular servers that we mentioned in *Chapter 4*, *Understanding How Apache Thrift Works*. Moreover, it contains a few more, which you may find useful in certain scenarios:

TThreadedServer: A threaded server that spawns a new thread for each connection

TForkingServer: Forks a new process for each request, and is more scalable than TThreadedServer

TProcessPoolServer: A server with a fixed pool of worker subprocesses to serve the requests

THttpServer: A simple HTTP server, not very perfromant

You can examine the implementations of the servers in the lib/py/src/server directory of the Apache Thrift package. Some of the basic servers are the classes in the TServer.py file, while the others are in separate files.

**Building the libraries**

To run the Python code, we need to have the Apache Thrift libraries built. Instead of installing it system-wise, we can have them in the local copy.

Copy the archive that you downloaded from the Apache Thrift website to your project created directory and decompress it:

**$ tar -xvzf thrift-0.9.2.tar.gz**

Now, we will have full Apache Thrift in the thrift-0.9.2 directory (the name may differ, depending on the version, substitute it in all commands). The Python libraries are in the thrift-0.9.2/lib directory and they need to be built. Enter the thrift-0.9.2/lib/py directory and run the setup command. This library will not be installed system-wise, but just built in place:

**$ cd thrift-0.9.2/lib/py**

**$ python setup.py build**

**$ cd ../../..**

**Implementing and running the service**

As in other languages, implementing the service is done by creating a handler that implements the interface that was generated by Apache Thrift; in our case, it is MyFirstService.Iface. The methods should accept and return the variables of declared types.

The following source code indicates how to do it. For brevity, the implementation details of the methods were omitted:

import sys, glob

# path for file generated by Apache Thrift Compiler

sys.path.append('gen-py')

# add path where built Apache Thrift libraries are

sys.path.insert(0, glob.glob('thrift-0.9.2/lib/py/build/lib.\*')[0])

from myfirst import MyFirstService

from myfirst.ttypes import \*

from myfirst.constants import \*

from thrift.transport import TSocket

from thrift.transport import TTransport

from thrift.protocol import TBinaryProtocol

from thrift.server import TServer

class MyFirstHandler(MyFirstService.Iface):

def \_\_init\_\_(self):

pass

def log(self, filename):

# implement log here

pass

def multiply(self, number1, number2):

# implement multiply here

pass

def get\_log\_size(self, filename):

# implement get\_log\_size here

pass

handler = MyFirstHandler()

processor = MyFirstService.Processor(handler)

transport = TSocket.TServerSocket(port=8080)

tfactory = TTransport.TBufferedTransportFactory()

pfactory = TBinaryProtocol.TBinaryProtocolFactory()

server = TServer.TSimpleServer(processor, transport, tfactory, pfactory)

server.serve()

To start the server, save this code to the MyFirstServer.py file and run the following command:

**$ python MyFirstServer.py**

Your server will be running on the localhost, port 8080.

**Implementing and running the client**

As I mentioned earlier, the Apache Thrift compiler generates an example client script, which is in the file prefixed with -remote, in our case, MyFirstService-remote. This client runs out of the box and allows you to test your service. For example, to test the multiply method, you can run:

**$ ./MyFirstService-remote multiply 7 6**

To see the list of possible methods, run:

**$ ./MyFirstService-remote --help**

This script is a great resource to learn how the client code should be written.

To run the client using the PHP Apache Thrift implementation, you need to prepare the network stack. Then, after connecting, you can call your remote procedures using the instance of your service's client that was generated by the compiler; in our case, it is MyFirstService.Client. Let's have a look at the example code, which is very simple:

import sys, glob

# add path with Apache Thrift compiler generated files

sys.path.append('gen-py')

# add path where built Apache Thrift libraries are

sys.path.insert(0, glob.glob('thrift-0.9.2/lib/py/build/lib.\*')[0])

from myfirst import MyFirstService

from myfirst.ttypes import \*

from myfirst.constants import \*

from thrift import Thrift

from thrift.transport import TSocket

from thrift.transport import TTransport

from thrift.protocol import TBinaryProtocol

transport = TSocket.TSocket('localhost', 8080)

transport = TTransport.TBufferedTransport(transport)

protocol = TBinaryProtocol.TBinaryProtocol(transport)

client = MyFirstService.Client(protocol)

transport.open()

client.log('logile.log')

print client.multiply(2,21)

transport.close()

If you want to use the variable types defined by you, that is, our MyStruct struct, it is very easy to instantiate them:

ms = MyStruct()

Then, work with it as with any other class to read or write values, for example:

ms.myi32 = 42

print ms.myi32

You can run your client by simply saving the code to the file, that is, MyFirstClient.py and running the Python file from the command line:

**$ python MyFirstClient.py**

Of course, you can also embed the client code in your application depending on your needs.

**PHP**

PHP is one of the most popular programming languages used mainly for server-side scripting of web applications (however, it may be used also as a general purpose language). It is relatively easy to learn and simple to use with thousands of popular applications written in it.

**Generating the code**

Apache Thrift's compiler offers a bunch of options for PHP. Run the following command to see them:

**$ thrift --help**

Look for the information about PHP generators given below:

**php (PHP):**

**inlined: Generate PHP inlined files**

**server: Generate PHP server stubs**

**oop: Generate PHP with object oriented subclasses**

**rest: Generate PHP REST processors**

**nsglobal=NAME: Set global namespace**

**validate: Generate PHP validator methods**

**json: Generate JsonSerializable classes (requires PHP >= 5.4)**

Some of the options may cater to your project's specific needs. Their descriptions may be cryptic, so here's some extra explanation:

inlined: The data encoding is done inline in the generated PHP file.

oop: The generated code is somewhat more object-oriented. With classes extending TBase, it is mutually exclusive with inlined.

server: This adds a service processor, which is required to run the service (you don't need it if you want to implement PHP only in the client).

rest: Some extra parameter processing is added, so the received values are casted at the proper types.

validate: Extra validation is added; so if the received value is null, an exception is thrown.

nsglobal=NAME: An extra namespace is added at the top of the already defined namespaces.

json: The generated classes implement PHP's JsonSerializable interface present in PHP >= 5.4 (read more on http://php.net/manual/en/class. jsonserializable.php).

You need to pick the options from the list to match your project's needs. It is possible to use more than one, for example, if you would like your code to not only be JsonSerializable, but also contain a service class, run the following command:

**$ thrift --gen php:json,server myfirst.thrift**

**Examining the code**

The code that you generated will be in the gen-php directory (or, if you chose the inlined option, then in the gen-phpi directory). Inside the directory, there will be a namespace structure (if you chose any) containing the PHP files for each of the services that you declared. Separately, there is the Types.php file, which contains the variables generated from your definition.

Let's have a look at this file. As PHP is dynamically typed, the types of variables are really a matter of convention and adding extra checking when reading or writing. As a result, the simple structure becomes an elaborate class with various methods handling the variables.

Now, let's have a look at the MyFirstService.php file. It contains the MyFirstServiceIf interface:

interface MyFirstServiceIf {

/\*\*

\* @param string $filename

\*/

public function log($filename);

/\*\*

\* @param int $number1

\* @param int $number2

\* @return int

\*/

public function multiply($number1, $number2);

/\*\*

\* @param string $filename

\* @return int

\* @throws \myfirst\MyError

\*/

public function get\_log\_size($filename);

}

In object-oriented programming, an interface is a description of the elements that the class that implements it has to have. Therefore, you need to implement all of the methods of your interface for the service to exist; in this case the log, multiply, and get\_log\_size methods you declared in your Apache Thrift document. There are annotations that will suggest you the behavior of the service.

Another important element in the MyFirstService.php file is the MyFirstServiceClient class. This class implements the service's interface as well, with the purpose of exposing public methods that you can use in your client script.

If you chose the server compiler option, you will also have the MyFirstServiceProcessor class generated. This is the processor that is needed to run the server for your service.

**Transports**

PHP implementation of Apache Thrift offers a variety of transports. You can always look them up in the lib/php/lib/Thrift/Transport. There are all of the most popular transports described in *earlier.*

One transport specific for PHP is TPhpStream, which reads from and writes to PHP's standard streams php://input and php://output. In this way, you don't need to run your own server to provide the service; you can use an existing HTTP server such as nginx or Apache HTTP Server.

If you intend to use one of the transports, I suggest you examine the implementation to be sure how it works and if it fits your specific needs.

**Protocols**

PHP implementation of Apache Thrift offers all of the standard protocols mentioned in *Chapter 4*, *Understanding How Apache Thrift Works*. Additionally, it provides TMultiplexedProtocol, which is not a standalone protocol, but a decorator that helps you deal with complex scenarios when you want to use multiple services on one server (called **multiplexing**).

Implementations of the protocols are in the lib/php/lib/Thrift/Protocol directory where you can examine them before using.

**Servers**

In PHP, there are the implementations of TSimpleServer and TForkingServer. Additionally, you can use any HTTP server already running by using the TPhpStream transport.

Implementations of the servers are relatively simple and you can examine them in the lib/php/lib/Thrift/Server directory.

**Implementing and running the service**

Implementing the service is done by creating a handler implementing the interface that was generated by Apache Thrift; in our case, it is MyFirstServiceIf. The methods should accept and return the variables of the declared types.

The source code below indicates how to do it. For the sake of brevity, the implementation details of the methods were omitted:

#!/usr/bin/env php

<?php

# path to your Apache Thrift library

define('THRIFT\_PHP\_LIB', \_\_DIR\_\_.'/thrift-0.9.2/lib/php/lib');

# path to the files generated by the Apache Thrift compiler

define('GEN\_PHP\_DIR', \_\_DIR\_\_.'/gen-php');

require\_once THRIFT\_PHP\_LIB.'/Thrift/ClassLoader/ThriftClassLoader. php';

use Thrift\ClassLoader\ThriftClassLoader;

$loader = new ThriftClassLoader();

$loader->registerNamespace('Thrift', THRIFT\_PHP\_LIB);

// register your namespace

$loader->registerDefinition('myfirst', GEN\_PHP\_DIR);

$loader->register();

// include here the protocols and transports that you need

use Thrift\Protocol\TBinaryProtocol;

use Thrift\Transport\TPhpStream;

use Thrift\Transport\TBufferedTransport;

// implementing the service interface

class MyFirstHandler implements \myfirst\MyFirstServiceIf {

public function log($filename) {

// implementation of log function

}

public function multiply($number1, $number2) {

// implementation of multiply function

}

public function get\_log\_size($filename) {

// implementation of get\_log\_size function

}

};

header('Content-Type', 'application/x-thrift');

echo "\r\n";

// instantiation of our handler

$handler = new MyFirstHandler();

$processor = new \myfirst\MyFirstServiceProcessor($handler);

$transport = new TBufferedTransport(new TPhpStream(TPhpStream::MODE\_R | TPhpStream::MODE\_W));

$protocol = new TBinaryProtocol($transport, true, true);

$transport->open();

$processor->process($protocol, $protocol);

$transport->close();

In this case, the service is run through the other already installed HTTP server. Save this code to the file (for example, MyFirstServer.php), upload it to the server, and point any clients to the address representing this file.

**Implementing and running the client**

To run the client using the PHP Apache Thrift implementation, you need to prepare the environment in the same manner as with a server. Then, you can call your remote procedures using the instance of your service's client, which was generated by the compiler; in our case, it is MyFirstServiceClient. Let's have a look at the example:

#!/usr/bin/env php

<?php

# path to your Apache Thrift library

define('THRIFT\_PHP\_LIB', \_\_DIR\_\_.'/thrift-0.9.2/lib/php/lib');

# path to the files generated by the Apache Thrift compiler

define('GEN\_PHP\_DIR', \_\_DIR\_\_.'/gen-php');

require\_once THRIFT\_PHP\_LIB.'/Thrift/ClassLoader/ThriftClassLoader. php';

use Thrift\ClassLoader\ThriftClassLoader;

$loader = new ThriftClassLoader();

$loader->registerNamespace('Thrift', THRIFT\_PHP\_LIB);

// register your namespace

$loader->registerDefinition('myfirst', GEN\_PHP\_DIR);

$loader->register();

// include here the protocols and transports that you need

use Thrift\Protocol\TBinaryProtocol;

use Thrift\Transport\TSocket;

use Thrift\Transport\THttpClient;

use Thrift\Transport\TBufferedTransport;

use Thrift\Exception\TException;

// provide hostname, port number and URL of your service

$server = new THttpClient('localhost', 8080, '/MyFirstServiceServer. php');

// create connection

$transport = new TBufferedTransport($server, 1024, 1024);

$protocol = new TBinaryProtocol($transport);

$client = new \myfirst\MyFirstServiceClient($protocol);

$transport->open();

// run remote methods with $client->methodname($param)

$client->log('lofgile.log');

print $client->multiply(2,21);

$transport->close();

If you want to use the custom variable types defined by you, that is, our MyStruct struct, you can instantiate it with:

$ms = new \myfirst\MyStruct();

Then work with it as with any other class to read or write values, for example:

$ms->myi32 = 42;

print $ms->myi32;

Save your client code to the file, MyFirstClient.php, and simply run the PHP file from the command line:

php -f MyFirstClient.php

Of course, you can also embed the client code in your application, depending on your needs.

**JavaScript**

JavaScript is a scripting language used mainly for frontend development in web applications, but it has also gained popularity recently in server-side solutions with Node.js.

JavaScript code generated by the Apache Thrift compiler (Node.js aside) is strictly client-side, intended to be used in the web browser against the services written in other languages.

**Generating the code**

Apache Thrift's compiler offers few options for JavaScript. Run the following command to see them:

**$ thrift --help**

Look for the information about JavaScript generators:

**js (Javascript):**

**jquery: Generate jQuery compatible code.**

**node: Generate node.js compatible code.**

**ts: Generate TypeScript definition files.**

Running the generator without any options will provide you with just plain JavaScript code. Let's explain the extra options:

Jquery: The generated code will be jQuery compatible

Node: Code for Node.js will be generated; technically, this is still JavaScript code, however this is a completely different runtime environment, so we will won't discuss it here. If you use this option, your file will be created in the gen-nodejs directory.

ts: Definition files for TypeScript will be generated. Useful if you use TypeScript in your project.

You need to pick the options from the list to match your project's needs. There is not much to choose from, but if you need to have your files both jQuery compatible and have TypeScript definition files, just use:

**$ thrift --gen js:jquery,ts myfirst.thrift**

Watch out, because you won't get any error message if you misspell the parameter; it will just be ignored. It's hard to debug such a situation, so just check your parameters twice.

**Examining the code**

The code that you generated will be in the gen-js directory. Inside the directory, there will be at least two files prefixed with your namespace.

One of them is suffixed with \_types.js and contains the variables and constant that you created (in our example, this file's name is myfirst\_types.js). Note that there are complex structures to handle the variables. Constants, on the other hand, are handled simply as JavaScript variables, for example:

myfirst.PI = 3.14159;

The other file bears the service's name (in our example, MyFirstService.js) and contains the client's objects.

If you declared more than one service or namespaces, more files will be generated accordingly.

If you used the ts option during compilation, there are also the TypeScript definition files with names ending in .d.ts.

**Transport, protocol, and servers**

Due to the simplicity of the implementation of JavaScript code to be used in the web browsers, the developer will only use one of the transport and protocol objects: Thrift.Transport and Thrift.Protocol, respectively. You can examine the implementation in the lib/js/src/thrift.js file, where the whole code is stored.

For obvious reasons, there is no server implementation in the client-side JavaScript.

**Implementing and running the client**

JavaScript client for Apache Thrift-enabled services is run by the web browser. The common scenario is to have a web application that collects some information from or posts to the service, which is going to be implemented in any other language.

When developing services to be consumed by JavaScript from web applications that will be publicly available, please consider the security and performance of the setup.

From the security point of view, you should bear in mind that your whole service will be accessible for everyone, not only for those users who will use your web applications, but for anyone who would want to write their own client. Having this in mind, think about possible vulnerabilities: methods that modify the data or expose them in bulk. Consider exposing only those methods that won't do any harm if misused by somebody, and move other methods to the server side (that is, add an extra layer in PHP or Python). You shouldn't embed any authorization code (that is, usernames, passwords, API keys, and so on) in the JavaScript code, as it is easily readable by anyone.

Taking into consideration that the performance is prepared to serve a lot of requests, use a server that is prepared to handle such a load. There are Apache Thrift servers in Python, PHP, or Java, that we mentioned, which can be used to do so.

To construct the client for the Apache Thrift service in JavaScript, we will need the following components:

1. The thrift.js library that can be obtained from the lib/js/src/ directory in your Apache Thrift library
2. The gen-js directory with files generated by the compiler
3. An HTML page that contains the client code

As you already have the first two components, let's prepare the HTML document. For brevity, the layout is reduced to be as simple as possible; you are encouraged to work your way towards the expected result.

Let's have a look at the HTML code:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www. w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>MyFirstService example</title>

<!-- include the Apache Thrift library -->

<script src="thrift.js" type="text/javascript"></script>

<!-- include files generated by the Apache Thrift compiler –

<script src="gen-js/myfirst\_types.js" type="text/ javascript"></script>

<script src="gen-js/MyFirstService.js" type="text/ javascript"></script>

<script type="text/javascript" src="http://code.jquery.com/ jquery-1.11.3.min.js"></script>

<script type="text/javascript" charset="utf-8">

// provide the service location - let's assume it's on http://localhost:8080/MyFirstService

var transport = new Thrift.Transport("http:// localhost:8080/MyFirstService");

var protocol = new Thrift.Protocol(transport);

var client = new MyFirstServiceClient(protocol);

client.log("logfile.log");

var mresult = client.multiply(6,7);

$('#result').val(result);

</script>

</head>

<body>

<p>result of 6x7: <input type="text" id="result" value=""/></ p>

</body>

</html>

As you see, this code is relatively simple in comparison to implementation in other languages.

If you want to use the variable types defined by you, that is, our MyStruct struct, it is very easy to instantiate them:

var ms = new MyStruct();

Then, work with it as with any other class to read or write values, for example:

ms.myi32 = 42;

$('#result').val(ms.myi32);

alert(ms.myi32);

To run the code, save it to the file, for example, MyFirstClient.html. Ensure that all the files that we listed before are accessible. Now, open your file in the web browser through a web server or directly. The client code will be run in the browser and the service will be called. The result will appear in the text field.