Comunicação entre processos

- soluções para interação cliente-servidor:
  - com a mesma linguagem/tecnologia de ambos os lados
    - a representação externa de dados pode ser otimizada para essa tecnologia (ex: Serialização em Java), mas difícil juntar um processo baseado noutra tecnologia
  - cross-platform, ou independente da linguagem em cada ponto
    - SOAP Web Services
    - REST
    - Thrift

Apache Thrift

é um framework para desenvolvimento de serviços, cross-language, que combina bibliotecas com a geração automática de código. Desenvolvido inicialmente pelo Facebook, é agora mantido pela Apache.

Conceitos principais

- https://thrift.apache.org/docs/concepts.html

As aplicações que comunicam com Thrift podem ser baseadas em linguagens diferentes, incluindo C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml e Delphi.

Em comparação com REST, podemos ter ganho de desempenho, se escolhermos uma combinação para protocolo e transporte que permita formas de representação externa de dados mais rápidas de traduzir, em formato binário (*binary* ou *compact binary*) e não textual (JSON ou XML, como é usual em SOAP e REST). E mantém-se as conveniências que já tínhamos com REST, de escalabilidade e independência de linguagens de programação.

Mais informação:

- https://thrift.apache.org/
- https://thrift.apache.org/static/files/thrift-20070401.pdf

Como funciona

- interface: define as operações e tipos de dados envolvidos, num ou mais serviços em ficheiros .thrift, de acordo com a *Interface Description Language* (IDL)
- o compilador Thrift é aplicado sobre o thrift file (sobre a interface) e gera automaticamente código **para determinada linguagem de programação**, relacionado com o transporte e tradução de dados relacionados com o serviço (definidos na interface), bem como a interface do serviço
- a **biblioteca** (software stack), juntamente com o código gerado pelo compilador, permite ativar rapidamente um serviço, ou um cliente para um serviço numa linguagem de programação diferente
  -
  - dependência Maven
  - <dependency>

```
    <groupId>org.apache.thrift</groupId>
    <artifactId>libthrift</artifactId>
    <version>0.16.0</version>
</dependency>
```

  -
  - dependência Gradle
    - `implementation group: 'org.apache.thrift', name: 'libthrift', version: '0.14.1'`
- os servidores podem funcionar nos modos:
  - TSimpleServer – *for simple server*
  - TThreadPoolServer – *for multi-threaded server*
  - TNonblockingServer – *for non-blocking multi-threaded server*

Tipos

- básicos (i32, string...)
- structs
- containers/coleções/maps e pouco mais

Como instalar em Linux

$ sudo apt install thrift-compiler

$ thrift -help

aplicar o compilador à interface de um serviço

$ thrift --gen <language> <FILE.thrift>

A- Exemplo inicial

Considere um serviço com três operações (get, save, getList) relativamente a um objeto genérico com três campos (inteiro, string, string).

A interface para este serviço está aqui: service.thrift (guarde este ficheiro numa nova pasta).

1- Vamos aplicar o compilador thrift e gerar código para Java:

$ mkdir gen.java

$ thrift -r -out gen.java --gen java service.thrift

O resultado será:

```
└── com
    └── baeldung
        └── thrift
            └── impl
                ├── CrossPlatformResource.java  # o tipo de dados usado nas operações do serviço
                ├── CrossPlatformService.java   # o serviço (get, save, getList)
                └── InvalidOperationException.java
```

Aqui estão representadas uma classe de dados, uma classe com as operações do serviço e uma exceção. Todos os ficheiros dentro de pastas de acordo com o package Java (nome referido na interface, em namespace).

2- Este código pode ser agora integrado no projeto de software.
Use este código inicial (projeto maven) onde estes ficheiros já estão incluídos (para o caso de não ter o compilador thrift).

Comece por analisar o pom.xml, verificando a dependência com a biblioteca para thrift.

Implementação
No código gerado, não temos o modo como o serviço é implementado.

3- Assim, no mesmo package, crie uma nova classe CrossPlatformServiceImpl com este código.

Verifique que tudo compila.

Servidor
4- Adicione uma classe CrossPlatformServiceServer, com este código.

Verifique que tudo compila.

Tente executar o servidor, pelo IDE (para simplificar).

Para executar na linha de comandos, pode adicionar o plugin exec, em pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>3.1.0</version>
      <executions>
        <execution>
          <goals>
            <goal>exec</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

e na linha de comandos:

```
mvn exec:java -Dexec.mainClass=com.baeldung.thrift.impl.CrossPlatformServiceServer
```

Cliente
5- Adicione uma classe CrossPlatformServiceClient, com este código.

A operação ping() testa o uso do serviço.
Depois, o código obtém um resource, modifica-o e volta a gravá-lo junto do serviço.

6- a- Teste a execução de ambas as aplicações... (no IDE, Run File)
Confirme que tudo está a funcionar bem.

b- Altere o cliente para receber argumentos com o endereço e porto do servidor.
Teste uma ligação entre o seu cliente e o servidor de um colega, na sala.

Execute agora na linha de comandos... por exemplo com:

$ mvn exec:java -Dexec.mainClass=com.baeldung.thrift.impl.CrossPlatformServiceServer

$ mvn exec:java -Dexec.mainClass=com.baeldung.thrift.impl.CrossPlatformServiceClient -Dexec.args="hostname port"

B- Cross Language / Cliente noutra tecnologia

Suponhamos que o serviço não é nosso... e queremos apenas uma aplicação cliente na linguagem Python3.

Basta aplicar o compilador de Thrift sobre a interface do serviço, com a opção py:

```
$ mkdir gen.python
$ thrift -r -out gen.python --gen py service.thrift
```

Na pasta de destino fica:

```
├── __init__.py
└── service
    ├── constants.py
    ├── CrossPlatformService.py
    ├── CrossPlatformService-remote
    ├── __init__.py
    └── ttypes.py
```

Precisará ter a biblioteca/módulo Thrift para Python:

```
• $ sudo pip3 install thrift
```

O ficheiro ttypes.py tem um conjunto de classes relacionadas com o serviço que poderá ter de importar/usar no cliente.

Obtenha o código recomendado para o cliente em Python.
Este ficheiro é arrumado dentro de gen-pyhton (antes de service).

Teste a comunicação deste cliente com o servidor em Java:

$ python3 ResourcePythonClient.py

C- Segundo exemplo: Calculadora

Vamos agora para um exemplo do tutorial de Thrift.

O serviço tem operações de cálculo, com argumentos numéricos.
Deve descarregar:
- shared.thrift tutorial.thrift

Analise ambos os ficheiros. Note os vários tipos possíveis, referidos em comentário.

Em Python
Na pasta de trabalho atual, gerar código para Python:

```
$ thrift -r --gen  py  tutorial.thrift
```

Note o "-r", dado que este ficheiro inclui outro.

Use o código fornecido para as aplicações cliente e servidor:
- PythonServer.py
- PythonClient.py

Teste a execução de ambas as aplicações.

D- apenas o cliente, em Java

Suponhamos que o serviço já existe, mas desta vez pretendemos uma aplicação cliente em Java.

Partindo novamente da interface, no ficheiro .thrift, geramos código para Java:

```
$ thrift -r --gen java tutorial.thrift
```

E comece por usar o código semelhante ao mostrado no tutorial: JavaClient.java

Como compilar este código Java? **Opções**:
a) criar um projeto novo, com maven, e incluir o thrift nas dependências; depois inserir esta classe no projeto;
b) experiência na linha de comandos: ter as bibliotecas em modo *stand-alone*

Vamos fazer a opção b)

Voltando ao projeto anterior (com maven e thrift), vamos exportar as dependências do projeto, para termos todos os ficheiros .jar.

Na pasta base daquele projeto:

```
        mkdir libs
        mvn dependency:copy-dependencies -DoutputDirectory=libs
```

```
$ ls libs/
  commons-codec-1.11.jar   httpclient-4.5.10.jar    javax.annotation-api-1.3.2.jar
  slf4j-api-1.7.28.jar     tomcat-embed-core-8.5.46.jar
  commons-logging-1.2.jar  httpcore-4.4.12.jar
  libthrift-0.14.1.jar     tomcat-annotations-api-8.5.46.jar
```

Agora mova a pasta libs para dentro da pasta base onde está o cliente do serviço calculadora.
Para colocar tudo na classpath, podemos gerar uma variável:

```
$ export CP=""; for x in `ls -1 libs` ; do echo $x; export CP=$CP:libs/$x ; done
$ echo $CP
```

E finalmente podemos compilar e executar o cliente na linha de comandos:

```
$ javac -d . -classpath $CP JavaClient.java
```

```
$ java -classpath $CP:. JavaClient simple
```

O servidor (em Python) deve estar ativo, para responder a este cliente. (Vamos voltar a este código, mas abaixo.)

E- Exercício

Utilizando a tecnologia Thrift, implemente o serviço "ReservasParaJantar", que ofereça as operações:

- adicionar um nome para reserva, no final da fila
- obter a lista de nomes em espera
- verificar se um nome se encontra na fila
- obter o nº de reservas/nomes existentes
- remover N nomes do início da fila

Tente ativar o serviço em Java e aplicações cliente em Python e Java.

F- Extra: SSL

Se reparar no exemplo fornecido da Calculadora em Java, o serviço é oferecido em dois modos, com threads a escutarem em portos diferentes, sem e com SSL.
O método secure() usa as opções para SSL, para que o canal de comunicação não exponha os dados em trânsito.

```
    /*
     * Use TSSLTransportParameters to setup the required SSL parameters. In this example
     * we are setting the keystore and the keystore password. Other things like algorithms,
     * cipher suites, client auth etc can be set.
     */
    TSSLTransportParameters params = new TSSLTransportParameters();
    // The Keystore contains the private key
    params.setKeyStore("keystore", "thrift", null, null);
    /*
     * Use any of the TSSLTransportFactory to get a server transport with the appropriate
     * SSL configuration. You can use the default settings if properties are set in the command line.
     * Ex: -Djavax.net.ssl.keyStore=.keystore and -Djavax.net.ssl.keyStorePassword=thrift
     *
     * Note: You need not explicitly call open(). The underlying server socket is bound on return
     * from the factory class.
     */
    TServerTransport serverTransport = TSSLTransportFactory.getServerSocket(9091, 0, null, params);
    TServer server = new TSimpleServer(new Args(serverTransport).processor(processor));
```

Para que funcione, devemos ter:

- no servidor:
  - keystore: parâmetros para o SSL, como uma chave privada e certificado a apresentar ao cliente
- no cliente:
  - truststore: o(s) certificado e chave pública(s) em que confia

Compilação em linha de comandos

Na pasta base deste projeto, caso tenha já exportado as bibliotecas para uma pasta libs (ver acima), pode fazer:

$ javac -classpath .:libs/libthrift-0.14.1.jar:libs/slf4j-api-1.7.28.jar:libs/javax.annotation-api-1.3.2.jar *java

$ java -classpath .:libs/libthrift-0.14.1.jar:libs/slf4j-api-1.7.28.jar:libs/javax.annotation-api-1.3.2.jar **JavaServer**

$ java -classpath .:libs/libthrift-0.14.1.jar:libs/slf4j-api-1.7.28.jar:libs/javax.annotation-api-1.3.2.jar **JavaClient simple**

$ java -classpath .:libs/libthrift-0.14.1.jar:libs/slf4j-api-1.7.28.jar:libs/javax.annotation-api-1.3.2.jar **JavaClient secure**

Em alternativa pode executar no seu IDE, desde que tenha a biblioteca Thrift nas dependências.
Código do servidor, CalculatorHandler.java.

Nota: o código apresentado ao longo desta atividade vem de:

- https://github.com/apache/thrift/tree/master/tutorial/java/src
- https://www.baeldung.com/apache-thrift