Universidade Federal do Ceará – Campus de Quixadá

Disciplina: Sistemas Distribuídos - 2021.1

Prof. Dr. Antonio Rafael Braga

Tutorial *Protocol Buffers – Java*

Importante: este tutorial não tem como objetivo dispensar o acesso ao site do ProtocolBuffers (https://developers.google.com/protocol-buffers/docs/javatutorial), pelo contrário, cita-o várias vezes seu conteúdo.

Escopo:

1) Downloads e Instalação
2) Como definir as mensagens no formato *.proto*
3) Como usar o compilador *protoc* para geração automática de código
4) Como usar a API JAVA *Protocol Buffer* para escrever e ler mensagens

a. Empacotar (*marshalling*) e Desempacotar (*unmarshalling*) mensagens

5) Rodar o exemplo do tutorial Oficial

1) Downloads e Instalação

- Para nosso trabalho vamos utilizar o Eclipse que o download pode ser feito no link https://www.eclipse.org/downloads/
- O Compilador ProtocolBuffers pode ser encontrado no link https://github.com/protocolbuffers/protobuf/releases, em nosso trabalho utilizamos o "protobuf-java-3.17.3.zip".

o Extrai para o diretório C: e adiciona nas variáveis de ambiente do windows: C:\protoc-3.17.3-win64\bin

2) Definição das mensagens que representam os objetos a serem serializados

```
syntax = "proto2";

package tutorial;

option java_multiple_files = true;
option java_package = "br.ufc.quixada.models";
option java_outer_classname = "AddressBookProtos";

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
```

```
  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

- Nesse exemplo é definido uma estrutura de dados que armazena uma Agenda Telefônica. Repare que uma Agenda (`message AddressBook`) pode ter várias Pessoas (`repeated message Person`). Cada pessoa pode ter, além de nome, id e e-mail, vários números de telefone(`message PhoneNumber`) de diferentes tipos (`enum PhoneType`).
- ***message****:* indica um dado estruturado, semelhante à *Struct* em C.
  - O código gerado para cada *message* será uma classe que a representa e um *Builder* para construí-la, além dos métodos para empacota-la e desempacota-la (Seção 3 deste tutorial).
  - Pode-se aninhar *messages,* como no exemplo acima, para criar dados mais complexos. O resultado em código gerado serão referências aninhadas entre objetos.
  - Herança não é suportada, mas pode-se reusar *messages* definidas em outros arquivos *.proto.* É possível, inclusive, reusar apenas partes aninhadas de uma *message.*
  - Mais detalhes em: https://developers.google.com/protocol-buffers/docs/proto
- ***required***: atributo obrigatório. Sempre deve ser "setado", caso contrário a mensagem não será construída.
- ***optional***: atributo pode ser "setado" ou não.
- ***repeated***: o atributo pode ser repetido inúmeras vezes (inclusive 0). Implementa uma coleção do tipo do atributo.

EXTRA – Trabalho (Definição das mensagens de requisição/resposta):

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*Int*
arguments	*array of bytes*

Para o nosso trabalho, devemos definir um arquivo *.proto* que represente as Mensagens trocadas entre os processos cliente e servidor (como ilustrado na figura acima). Os atributos *objectRefrence* e *methodId* podem ser *Strings*. O atributo *arguments* é do tipo *bytes* (*ByteString* em java) e receberá os argumentos de requisição (parâmetros dos métodos) ou de resposta (retorno dos métodos) já empacotados (*array* de *bytes*). Para conhecer os tipos suportados pelo *protocolbuffer* e seus respecitvos em Java, acesse: https://developers.google.com/protocol-buffers/docs/proto

3) Compilando as mensagens com o compilador protoc

```
protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/addressbook.proto
```

- *$SRC_DIR*: Path do diretório da aplicação
- *$DST_DIR*: Path do diretório que receberá os códigos gerados (geralmente igual a SRC_DIR)
- *$SRC_DIR*: Path do diretório onde está armazenado o arquivo .proto

O resultado da compilação são códigos(Classes e *Builders*) que representam as *messages* definidas no arquivo *.proto*, bem como métodos para empacotamento e desempacotamento. Para cada arquivo *.proto* definido, apenas uma arquivo *.Java* é gerado, independentemente de quantas *messages* foram definidas.

4) Protocol Buffer API

Builder

As classes geradas pelo compilador *protoc* são imutáveis (como *String* em java). Uma vez que o objeto é construído, ele não pode mais ter seus atributos modificados. Para construir um objeto, temos que primeiro construir seu Builder, "setar" os atributos e finalmente chamar build().

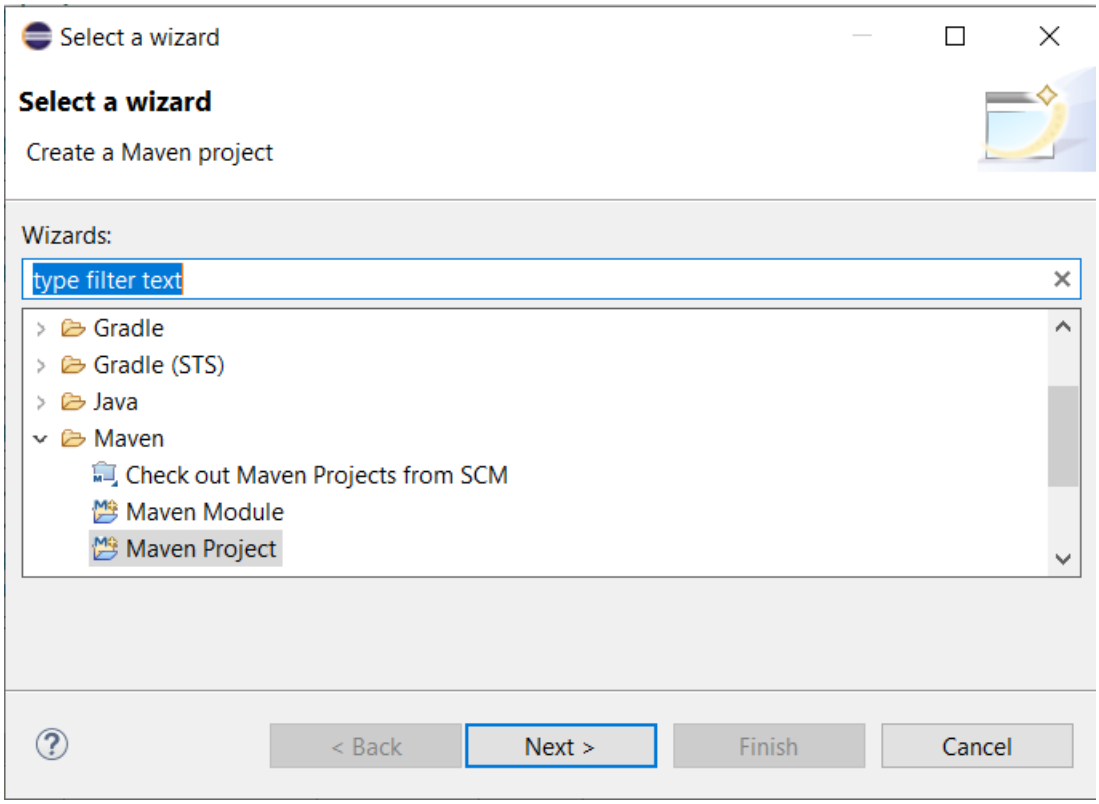
```
Person john =
  Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .addPhone(
      Person.PhoneNumber.newBuilder()
        .setNumber("555-4321")
        .setType(Person.PhoneType.HOME))
    .build();
```

Empacotamento(Serialização) e Desempacotamento(Parsing)

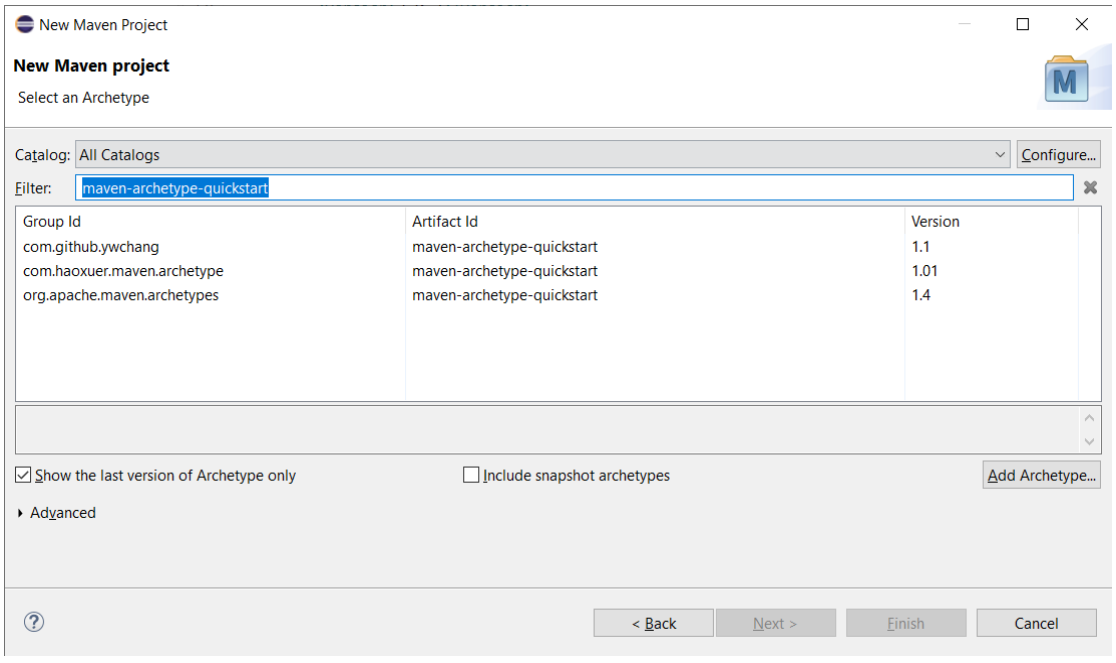
- **byte[] toByteArray();**
  - Serializa o objeto em array de bytes.
- **static Object parseFrom(byte[] data);:**
  - A partir de uma array de bytes, constrói o objeto correspondente.
- **void writeTo(OutputStream output);:**
  - serializa o objetos e escreve em um OutputStream.
- **static Object parseFrom(InputStream input);:**
  - Lê um InputStream e constrói o objeto relacionado.

5) Rodando o exemplo do tutorial

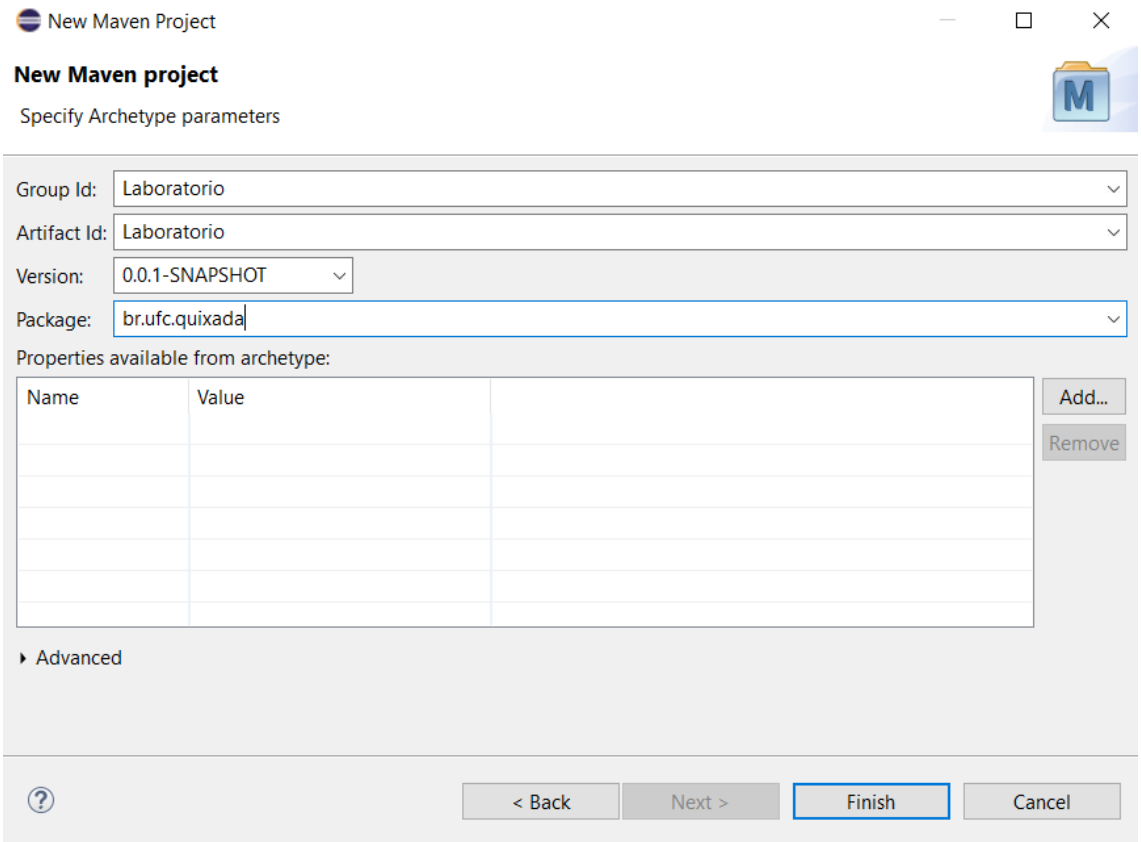
- Abre o eclipse e vamos criar um novo projeto. Utilizaremos o Maven pois possui um gerenciador de pacotes que facilita nosso projeto.



- Ao selecionar Next aparecerá uma nova tela, deixa como está e **Next** e chegamos no catálogo de pacotes do maven. Buscar pelo "**maven-archetype-quickstart**". Selecionamos a versão 1.4, mas pode selecionar outra versão e **Next**.



- Nesse próximo passo é nomeado o Projeto e o pacote (o pacote deve ser o mesmo definido no arquivo .proto) que usaremos. Defina e **Finish.**



- Precisamos editar nosso arquivo pom.xml gerado automaticamente para adicionar o pacote “com.google.protobuf” para incluir a dependência do protobuffer.

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
      <groupId>com.google.protobuf</groupId>
      <artifactId>protobuf-java</artifactId>
      <version>3.17.0</version>
  </dependency>
</dependencies>
```

Esse código pode ser encontrado aqui, tanto para o Maven como outros gerenciadores de pacotes.

- Utilizando nosso código proto citado no início criamos o arquivo “addressbook.proto” no diretório do nosso projeto: “$DiretorioDoProjeto$/src/main/java”.
- Agora vamos no Terminal entramos no diretório acima e executar o comando:

  # protoc --java_out=. .\addressbook.proto

Que gera as classes java necessárias para nosso projeto.

- Dentro de nosso pacote br.ufc.quixada criamos o arquivo que fará a serialização AddPerson.java

```
package br.ufc.quixada;

import br.ufc.quixada.models.AddressBook;
import br.ufc.quixada.models.Person;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;

class AddPerson {
  // This function fills in a Person message based on user input.
  static Person PromptForAddress(BufferedReader stdin,
                                 PrintStream stdout) throws IOException {
    Person.Builder person = Person.newBuilder();

    stdout.print("Enter person ID: ");
    person.setId(Integer.valueOf(stdin.readLine()));

    stdout.print("Enter name: ");
    person.setName(stdin.readLine());

    stdout.print("Enter email address (blank for none): ");
    String email = stdin.readLine();
    if (email.length() > 0) {
      person.setEmail(email);
    }

    while (true) {
      stdout.print("Enter a phone number (or leave blank to finish): ");
      String number = stdin.readLine();
      if (number.length() == 0) {
        break;
      }

      Person.PhoneNumber.Builder phoneNumber =
        Person.PhoneNumber.newBuilder().setNumber(number);

      stdout.print("Is this a mobile, home, or work phone? ");
      String type = stdin.readLine();
      if (type.equals("mobile")) {
        phoneNumber.setType(Person.PhoneType.MOBILE);
      } else if (type.equals("home")) {
        phoneNumber.setType(Person.PhoneType.HOME);
      } else if (type.equals("work")) {
        phoneNumber.setType(Person.PhoneType.WORK);
```

```
      } else {
        stdout.println("Unknown phone type.  Using default.");
      }

      person.addPhones(phoneNumber);
    }

    return person.build();
  }

  // Main function:  Reads the entire address book from a file,
  //   adds one person based on user input, then writes it back out to the same
  //   file.
  public static void main(String[] args) throws Exception {

//         if (args.length != 1) {
//       System.err.println("Usage:  AddPerson ADDRESS_BOOK_FILE");
//       System.exit(-1);
//     }
//         args = [];

    AddressBook.Builder addressBook = AddressBook.newBuilder();

    // Read the existing address book.
    try {
      addressBook.mergeFrom(new FileInputStream("teste_proto"));
    } catch (FileNotFoundException e) {
      System.out.println("teste_proto" + ": File not found.  Creating a new
file.");
    }

    // Add an address.
    addressBook.addPeople(
      PromptForAddress(new BufferedReader(new InputStreamReader(System.in)),
                       System.out));

    // Write the new address book back to disk.
    FileOutputStream output = new FileOutputStream("teste_proto");
    addressBook.build().writeTo(output);
    output.close();
  }
}
```

Para testar basta executar o arquivo criado.

- Para podermos desserializar criamos o arquivo ListPeople.java que mostra os dados que escrevemos no arquivo AddPerson.java.

```
package br.ufc.quixada;

import br.ufc.quixada.models.AddressBook;
import br.ufc.quixada.models.Person;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

class ListPeople {
  // Iterates though all people in the AddressBook and prints info about them.
  static void Print(AddressBook addressBook) {
    for (Person person: addressBook.getPeopleList()) {
```

```
      System.out.println("Person ID: " + person.getId());
      System.out.println("  Name: " + person.getName());
      if (person.hasEmail()) {
        System.out.println("  E-mail address: " + person.getEmail());
      }

      for (Person.PhoneNumber phoneNumber : person.getPhonesList()) {
        switch (phoneNumber.getType()) {
          case MOBILE:
            System.out.print("  Mobile phone #: ");
            break;
          case HOME:
            System.out.print("  Home phone #: ");
            break;
          case WORK:
            System.out.print("  Work phone #: ");
            break;
        }
        System.out.println(phoneNumber.getNumber());
      }
    }
  }

  // Main function:  Reads the entire address book from a file and prints all
  //   the information inside.
  public static void main(String[] args) throws Exception {
//    if (args.length != 1) {
//      System.err.println("Usage:  ListPeople ADDRESS_BOOK_FILE");
//      System.exit(-1);
//    }

    // Read the existing address book.
    AddressBook addressBook =
      AddressBook.parseFrom(new FileInputStream("teste_proto"));

    Print(addressBook);
  }
}
```

Ao executar esse arquivo teremos a lista de Pessoas que foram serializadas.