Exercise 1

gRPC

Author: Mariusz Fraś

1 Objectives of the exercise

The purpose of the exercise is:

1. Familiarization with the development environment for gRPC applications.
2. Understanding the general architecture of the gRPC application.
3. Mastering the basic techniques of creating gRPC applications.

2 Development environment

The developing of gRPC applications can be performed using various platforms. Here is considered the following environment:

- Windows 7 (or higher) Operating System (also possible MacOS or Linux).
- Visual Studio 2017/2019/2022 (with proper NuGet packages).
- Java programming platform – e.g. IntelliJ.IDEA (with proper plugins).

3 Exercise – Part I gRPC application project

3.1 Java based platform – maven project

Basic packages

Necessary modules (packages/libs) to build gRPC applications are:

- **io.grpc:grpc-protobuf** and **io.grpc:grpc-stub** – to build and use stubs for calling remote procedure,
  grpc-stub – include stub classes for gRPC that provide type-safe bindings.
- **grpc-netty** – **netty** server hosting gRPC service with defined remote procedures (also grpc-netty-shaded can be used).

In recommended and proposed here structure of maven project there are also necessary tools to compile and generate necessary modules for building application:

- **kr.motd.maven:os-maven-plugin** (added as an extension) – used to detect operating system and platform version,
- **org.xolstice.maven.plugins:protobuf-maven-plugin** plugin –used to compile stubs from **proto file**.

os-maven-plugin – is a Maven plugin that generates various useful platform-dependent project properties. It detects the information about the current operating system and normalize it. The one used in project is **os.detected.classifier** –a shortcut for OS version and machine architecture. It can be referenced I POMs as: **${os.detected.classifier}**.

protobuf-maven-plugin plugin compile stubs with use of **protoc** compiler.

Including above components in dependences (and extension) will cause downloading proper components from repositories and activate necessary steps to generate project.

Maven Protocol Buffers plugin

New versions of **protoc** binary executables for major operating systems are available as artifacts in Maven repository. These artifacts can be referenced in plugin configuration. Xolstice's **protobuf-maven-plugin** plugin uses Protocol Buffer Compiler **protoc** to generate Java source files from **.proto** file (protocol buffer definition).

The Plugin has the several goals – among others:
protobuf:compile – compiles main .proto definitions into Java source files and attaches the generated sources to the project.
protobuf:compile-custom – compiles main .proto definitions using a specified custom protoc plugin.
*protobuf:test-compi*le – compiles test .proto definitions into Java source files and attaches the generated test sources to the project.
protobuf:test-compile-custom – compiles test .proto definitions using a specified custom protoc plugin.

To compile code the goals should be included in **protobuf-maven-plugin** in its **<executions>** section:

```
<executions>
    <execution>
        <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
            <goal>test-compile</goal>
            <goal>test-compile-custom</goal>
        </goals>
    </execution>
</executions>
```

The plugin automatically resolves and downloads the **protoc** executable, and uses it for compiling protobuf definitions. To do this it must be included **<configuration>** node/section in plugin's section in the following form.

```
<configuration>
  <protocArtifact>
    com.google.protobuf:protoc:${protobuf.version}:exe:
                                ${os.detected.classifier}
  </protocArtifact>
</configuration>
```

(***note**: put the code without spaces in one line!*)

Parameter **protocArtifact** is used for specifying artifact coordinates in a format:

`groupId:artifactId:version[:type[:classifier]]`

(square bracket means optional use – here we use **classifier**).

It defines to use **com.gogle.protobuf.protoc** plugin to which is a compiler for .proto files. It generates language-specific code for Protobuf messages and RPC interfaces.

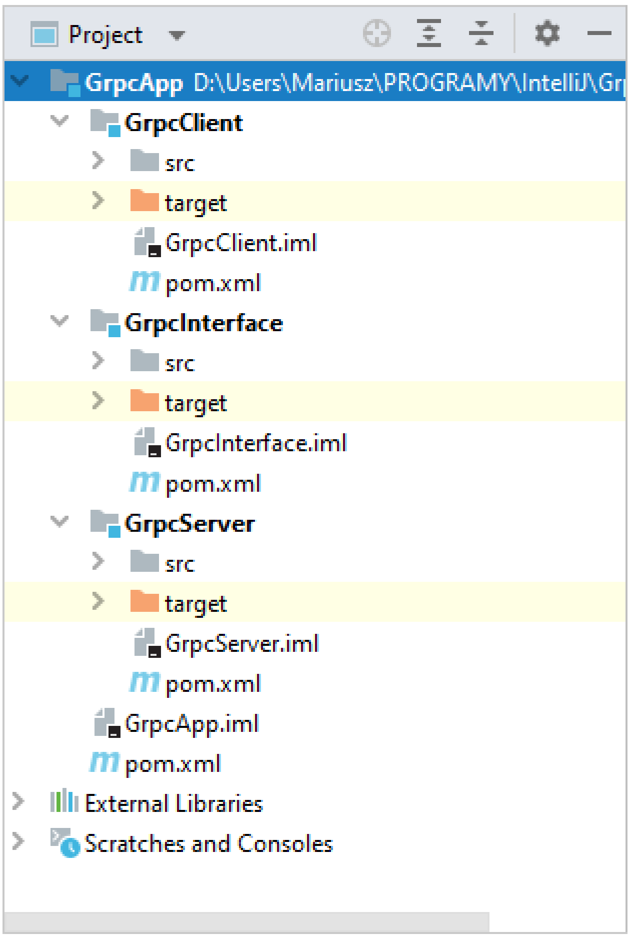
The same way must be defined to use **io.grpc: protoc-gen-grpc-java** which is the protoc plugin for gRPC Java:

```
<configuration>
  <pluginId>grpc-java</pluginId>
  <protocArtifact>
    io.grpc:protoc-gen-grpc-java:${grpc.version}:exe:
                                ${os.detected.classifier}
  </protocArtifact>
</configuration>
```

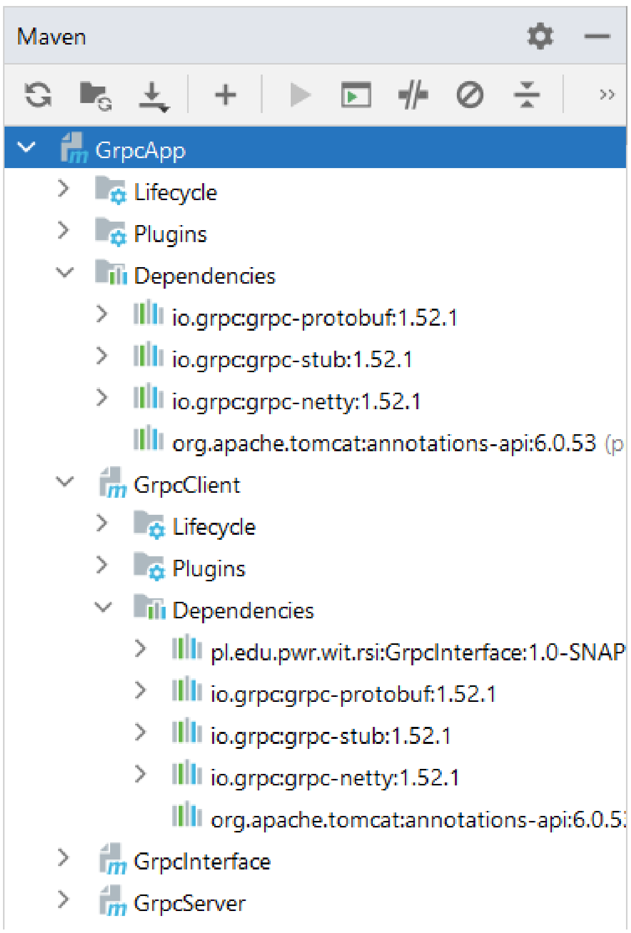
<pluginId> is a unique id that identifies the plugin to protoc.

3.1.1 General project structure

The Java app project is multi-module maven project. It consist of **GrpcInterface**, **GrpcServer**, and **GrpcClient** modules as presented in the figure (on the left side project structure, on the right some maven dependences – defined latter). It contains one shared/main POM file and three POMs of each module. In shared POM are defined shared components.



```
Project
GrpcApp D:\Users\Mariusz\PROGRAMY\IntelliJ\Gr
  GrpcClient
    src
    target
    GrpcClient.iml
    pom.xml
  GrpcInterface
    src
    target
    GrpcInterface.iml
    pom.xml
  GrpcServer
    src
    target
    GrpcServer.iml
    pom.xml
  GrpcApp.iml
  pom.xml
External Libraries
Scratches and Consoles
```



```
Maven
GrpcApp
  Lifecycle
  Plugins
  Dependencies
    io.grpc:grpc-protobuf:1.52.1
    io.grpc:grpc-stub:1.52.1
    io.grpc:grpc-netty:1.52.1
    org.apache.tomcat:annotations-api:6.0.53 (p
  GrpcClient
    Lifecycle
    Plugins
    Dependencies
      pl.edu.pwr.wit.rsi:GrpcInterface:1.0-SNAP
      io.grpc:grpc-protobuf:1.52.1
      io.grpc:grpc-stub:1.52.1
      io.grpc:grpc-netty:1.52.1
      org.apache.tomcat:annotations-api:6.0.5
  GrpcInterface
  GrpcServer
```

- Create the maven project composed of mentioned modules.
  Note: on of possible way is to create new maven project (here named **GrpcApp**) with default module, next remove src node/folder (leave pom.xml), and then add three mentioned child modules (their parent is GrpcApp).

3.1.2 POMs

The main POM (pom.xml) defines basic elements for the app project

- In the main POM define the following:
  – three component modules (usually added by platform when modules are added)

```
<modules>
  <module>GrpcServer</module>
  <module>GrpcClient</module>
  <module>GrpcInterface</module>
</modules>
```

  – few properties – values used in other places (mostly plugin version numbers)

```
<properties>
  <protobuf.maven.plugin.version>0.6.1</protobuf.maven.plugin.version>
  <grpc.version>1.52.1</grpc.version>
  <protobuf.version>3.22.2</protobuf.version>
  <os.maven.plugin.version>1.7.1</os.maven.plugin.version>
  <tomcat.annotations.api.version>6.0.53</tomcat.annotations.api.version>
</properties>
```

- necessary dependences – packages used to build application:
  - **grpc-protobuf** and **grpc-stub** – to build and use stubs,
  - **grpc-netty** – **netty** server hosting app service with defined remote procedures.

```
<dependencies>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>${grpc.version}</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>${grpc.version}</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty</artifactId>
    <version>${grpc.version}</version>
  </dependency>
  <dependency> <!-- necessary for Java 9+ -->
    <groupId>org.apache.tomcat</groupId>
    <artifactId>annotations-api</artifactId>
    <version>6.0.53</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

The Tomcat **annotations-api** are necessary for compatibility with Java 9 and above.

Interface module

The **GrpcInterface** module defines .proto components and the its POM must contain elements to generate proper code and actions.

- In the **interface** POM define the following:
  - parent module section (added by platform during project configuration),
  - in the **<build>** section an **os-maven-plugin** extension – plugin used to detect operating system and platform version:

```
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>${os.maven.plugin.version}</version>
    </extension>
  </extensions>
...
</build>
```

  - in **<plugins>** section a Xolstice **protobuf-maven-plugin** plugin (used to compile stubs from **proto file** with use of **protoc** compiler) with proper configuration (see description above) and goals:

```
<project ...>
...
  <build>
    ...
    <plugins>
      <plugin>
        <groupId>org.xolstice.maven.plugins</groupId>
        <artifactId>protobuf-maven-plugin</artifactId>
        <version>${protobuf.maven.plugin.version}</version>
        <configuration>
          <protocArtifact>com.google.protobuf:
                  protoc:${protobuf.version}:
                  exe:${os.detected.classifier}</protocArtifact>
          <pluginId>grpc-java</pluginId>
          <pluginArtifact>io.grpc:protoc-gen-grpc-java:${grpc.version}:
                  exe:${os.detected.classifier}</pluginArtifact>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>compile-custom</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Server module

The **GrpcServer** module contains service code (remote procedures) and host for services (Netty server).

- In the **server** POM (pom.xml) define the following:
  - parent module section (added by platform during project configuration),
  - dependency of **GrpcInterface** interface module – to contain generated stub:

```
<dependencies>
  <dependency>
    <groupId>pl.edu.pwr.wit.rsi</groupId>
    <artifactId>GrpcInterface</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Additionally you may include plugin to build complete jar to run application independently (outside IntelliJ environment).

- in the **<build><plugins>** section include **maven-assembly-plugin** assembly plugin with definition to build jar with all app components with dependences:

```
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-assembly-plugin</artifactId>
          <version>3.3.0</version>
          <configuration>
            <descriptorRefs>
              <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
            <archive>
              <manifest>
                <mainClass>GrpcServer</mainClass>
              </manifest>
            </archive>
          </configuration>
          <executions>
            <execution>
              <id>make-assembly</id>
              <phase>package</phase>
              <goals>
                <goal>single</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </project>
```

This part is not necessary if we run application only from platform (here IntelliJ.IDEA). This is necessary if we may want run the application from console with **java –jar …** command or distribute full package.

<u>Client module</u>

The **GrpcClient** module just makes requests for remote procedures and consumes responses. The POM needn't to contain anything more but parent section.

Similarly you may include plugin to build complete jar to run client independently (outside IntelliJ environment).

- In the **client** POM (pom.xml) define the maven-assembly-plugin similarly to server's POM.

May be omitted if not run outside the IntelliJ.IDEA platform.

3.1.3 Proto file

In the **proto file** the interface for remote procedures is defined. On its basis the proper stubs (java classes and methods) are built as defined in POM files.

- In the **GrpcInterface** interface module in **src\main** directory/node create new **proto** directory.
- In **src\main\proto** directory/node add new file proto file (here named **GrpcInterface.proto**).
- In the proto file enter the code defining syntax, service (here named **ServiceName**) and procedure (here named **unaryProcedure**), and messages (inputs and outputs). In the file you may also define same additional elements (e.g. java package).

```
syntax = "proto3";
option java_multiple_files = true;
option java_outer_classname = "GrpcAppProto";
option objc_class_prefix = "GAP";
// The service definition.
service ServiceName {
  // Remote procedures:
  rpc unaryProcedure (TheRequest) returns (TheResponse) {}
}
// The request message containing the user's name and age.
message TheRequest {
  string name = 1;
  int32 age = 2;
}
// The response message containing the hello text
message TheResponse {
  string message = 1;
}
```

***Important remark**:*
*Note the remote procedure name starts with small letter. **Don't** use first capital letter to avoid some compilation problems (probably caused by imperfect app building components).*

Instead ServiceName you may use your own service name.

3.1.4 Server code

In server code file (here **GrpcServer** class) you must minimally define:

– the class extending generated from .proto file service class, with the method corresponding to remote procedure defined in .proto file – in this method the request must be consumed and the response be produced with use of **StreamObserver**, [the names of classes and methods correspond to names used in .proto file]
– in the main method build server object working on given port
– add the build service implementation (class object) to the server,
– start the server (and optionally finally down).

The class containing remote procedure must extend generated (by protoc compiler) inner class of ***ServiceName*****Grpc** class i.e.:
ServiceName**Grpc.*****ServiceName*****ImplBase**

- Define the service class (here named **ServiceNameImpl**) with :

```
public class GrpcServer {
  [...]
  static class MyServiceImpl extends ServiceNameGrpc.ServiceNameImplBase {
    public void unaryProcedure(TheRequest req,
                         StreamObserver<TheResponse> responseObserver) {
      String msg;
      System.out.println("...called UnaryProcedure - start");
      if (req.getAge() > 18)
        msg = "Mr/Ms "+ req.getName();
      else
        msg = "Boy/Girl";
      TheResponse response = TheResponse.newBuilder()
                                           .setMessage("Hello " + msg)
                                           .build();
      try {
          Thread.sleep(2000);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      responseObserver.onNext(response);
      responseObserver.onCompleted();
      System.out.println("...called UnaryProcedure - end");
    }
  }
}
```

Note:

- The class extends the class built from **proto file**. The extended class name is composed of ***Xxx*Grpc.*Xxx*ImplBase** where *Xxx* is the name of service defined in proto file (here: **ServiceName**). The *Xxx*ImplBase is inner class of *Xxx*Grpc.
- The class' method (remote procedure) name is the same as in proto file, and the parameters types (request and response) are the same as in proto file.
- The procedure takes **StreamObserver** as parameter. The **StreamObserver** is used to consume/produce data from/to observed stream used to data exchange
- The response is built with use of **newBuilder()** method (as the response class implements specific builder interface).
- Typical getters and setters are used to access/set values of message data.
- The response is passed to data exchange stream with **StreamObserver's onNext(...)** method what makes the client take the data. The response must be built, then **onNext(...)** and **onCompleted()** methods must be called.

The server hosts the defined service (with remote procedure) and works on given port.

- In the server module class in **main** method build **Server** object with use of **ServerBuilder** (from io.grpc),
  – at this statement create service object specifying connection port and add service implementation (class object with remote procedure) to the server,

- Start the server and wait for termination.

```
public class GrpcServer {
  public static void main(String[] args) {
    int port = 50001;
    System.out.println("Starting gRPC server...");
    Server server = ServerBuilder.forPort(port)
                           .addService(new ServiceNameImpl()).build();
    try {
      server.start();
      System.out.println("...Server started");
      server.awaitTermination();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  static class MyServiceImpl extends ... {[...previous code...]}
}
```

3.1.5 Client code

<u>**The blocking (synchronous) calls**</u>

In client java code file (here **GrpcClient** class) you must:

- build the **channel** for communication with use of **ManagedChannelBuilder**,
- build the client stub – ***ServiceName*BlockingStub** object – with use of **newBlockingStub()** method (of ***ServiceName*Grpc** class) with **channel** parameter.
- build the request object (message to send) with **newBuilder()** method,
- finally call the remote procedure with use of stub object and request as parameter.

```
public class GrpcClient {
  public static void main(String[] args) {
    String address = "localhost"; //here we use service on the same host
    int port = 50001;
    ServiceNameGrpc.ServiceNameBlockingStub bStub;
    System.out.println("Running gRPC client...");
    ManagedChannel channel = ManagedChannelBuilder
        .forAddress(address, port) .usePlaintext().build();
    bStub = ServiceNameGrpc.newBlockingStub(channel);
    TheRequest request = TheRequest.newBuilder().setName("Mariusz")
                                              .setAge(33).build();
    System.out.println("...calling unaryProcedure");
    TheResponse response = bStub.unaryProcedure(request);
    System.out.println("...after calling unaryProcedure");
    System.out.println("--> Response: " + response);
    channel.shutdown();
  }
}
```

- Run the application – first server, next client – and check the operation.

The non-blocking (asynchronous) calls

To perform non-blocking (asynchronous) calls in the client you must:

– define and create non-blocking stub – the ***ServiceName*Stub** object – with use of **newStub()** method (of ***ServiceName*Grpc** class) with **channel** as a parameter:

– define the class implementing **StreamObserver** – the class must implement the **onNext(...)** method in which gets the response from server, passed to the method as a parameter. **onNext(...)** is called when data from channel are ready to get.

– build the request and call the remote procedure with use of non-blocking stub object and request and stream observer as parameters.

- Define the non-blocking stub and create the variable object:

```
...
ServiceNameGrpc.ServiceNameStub nonbStub;
...
nonbStub = ServiceNameGrpc.newStub(channel);
...
```

- Define the class implementing **StreamObserver<TheResponse>** here named **UnaryObs**. In the class define:
  – the **onNext(...)** method which takes the response (a message) as a parameter and takes the data from the response.
  – mandatory on **onError(...)** method.
  – mandatory **onCompleted(...)** method.

```
private static class UnaryObs implements StreamObserver<TheResponse> {
  public void onNext(TheResponse theResponse) {
    System.out.println("-->async unary onNext: " +
                       theResponse.getMessage());
  }
  public void onError(Throwable throwable) {
    System.out.println("-->async unary onError");
  }
  public void onCompleted() {
    System.out.println("-->async unary onCompleted");
  }
}
```

- Build the request (here we use the same request message as previously) and call the procedure in asynchronous (non-blocking) mode:

```
...
System.out.println("...async calling unaryProcedure");
nonbStub.unaryProcedure(request, new UnaryObs());
System.out.println("...after async calling unaryProcedure");
```

- Before channel shutdown add the code to prevent too early finishing client app.
- Run the application – first server, next client – and check the operation.

The result of operation should be similar to the following:

```
Starting gRPC server...
...Server started
...called unaryProcedure - start
...called unaryProcedure - end
...called unaryProcedure - start
...called unaryProcedure - end
```

```
Running gRPC client...
...calling unaryProcedure
...after calling unaryProcedure
-->message: "Hello Mr/Ms Mariusz"
...async calling unaryProcedure
...after async calling unaryProcedure
==> Press <ENTER> to finish <==
-->async unary onNext: Hello Mr/Ms Mariusz
-->async unary onCompleted
```

3.1.6 The streaming calls (streaming from server to client)

Streaming from server is made very similarly as for unary procedures. It is because **StreamObserver** used to send/receive response/request data works (puts/takes data to/from stream) in non-blocking way until **onCompleted(...)** method is called.

To implement stream procedure on the client side you can use almost the same code as for unary procedure with the following difference:

– in the procedure you must repeatedly prepare streamed data **chunks** to send (create response messages) and call **onNext(...)** method in loop.

– call **onCompleted()** method after all chunks are sent – the method notifies of successful stream completion – after this method is called onNext(...) can't be called.

- Define in **.proto** file the procedure to send from server to client stream of data chunks (just simple strings) here named **streamProcedure**.

```
service ServiceName {
  ...
  rpc streamProcedure (TheRequest) returns (stream TheResponse) {}
}
```

  Here we use the same types for messages (request and response) to stream text.

- Define in the server module the **streamProcedure** procedure to send stream of strings to client.
  Use almost the same code as for unaryProcedure with distinction to produce messages and send them in loop:

```
public void streamProcedure(TheRequest req,
                            StreamObserver<TheResponse> responseObserver) {
...
  for( int i=0; i<NUM_OF_CHUNKS; i++) {
    TheResponse response = TheResponse.newBuilder()
                                 .setMessage("Stream chunk " + (i+1)).build();
    [enter here Thread.sleep to easier trace the operation]
    responseObserver.onNext(response);
  }
  responseObserver.onCompleted();
  ...
}
```

On the client side calling and receiving streamed response is very similar to unary one with respect to getting multiple chunk data. There are distinguished two cases: synchronous (more precisely synchronous data consumption) and asynchronous (non-blocking).

To get stream in synchronous manner (in fact request is non-blocked but data collecting makes whole process synchronous) you don't use own StreamObserver but need to collect chunks in iterator like variable, making the request as for synchronous unary.

To get stream in non-blocking manner the code structure is exactly the same as for unary case, because of non-blocking implementation of StreamObserver and it contains **onNext(…)** method, that in stream case is called as many times as needed to transfer all chunks. The only additional step may be to perform some operation on completion of transferring chunks. It is usually done in **onCompleted()** method.

- Add to client the code of blocking request for streamed data chunks (text strings):

```
...
Iterator<TheResponse> respIterator;
System.out.println("...calling streamProcedure");
respIterator = bStub.streamProcedure(request);
System.out.println("...after calling streamProcedure");
TheResponse strResponse;
while(respIterator.hasNext()) {
    strResponse = respIterator.next();
    System.out.println("-->" + strResponse.getMessage());
}
...
```

- Add to the client code the request for stream using non-blocking stub and StreamObserver:

```
...
System.out.println("...async calling streamProcedure");
nonbStub.streamProcedure(request, new UnaryObs());
System.out.println("...after async calling streamProcedure");
...
```

Here you may use the same StreamObserver object as previously or may define exactly the same separate one. Using separate StreamObservers make easier control multithreading processing in complex application.

- Compose the request in client in the following order:
  - unaryProcedure synchronous call,
  - streamProcedure synchronous call,
  - unaryProcedure asynchronous call,
  - streamProcedure asynchronous call.
- Run the application and check the operation. Pay attention on the order of printed comments and responses on client side.

3.1.7 The streaming from client to server

Streaming to server requires programing of sort reversely to previous case i.e.:

In .proto file:

– declaration of procedure that takes stream of messages as parameter:

```
rpc streamToSrv (stream TheRequest) returns (TheResponse) {}
```

here named **streamToSrv** (here we use the same message types because we stream strings of text).

In server code:

– implementing on server side the observer for the request (more precisely messages being chunks of the sent stream) – here e.g.:

```
StreamObserver<TheRequest> srvObserver = new StreamObserver<TheRequest>()
{...}
```

– implemented stream procedure (here **streamToSrv**) returns above observer and takes client's response observer as a parameter:

```
public StreamObserver<TheRequest> streamToSrv(StreamObserver<TheResponse>
                                               responseObserver)
{  ...
   return srvObserver;
}
```

So the client that calls above procedure puts its own response observer as a parameter, and receives the server's observer (to use them – see later),

– the server's request observer contains the code:
    - In **onNext(TheRequest the Request)** method that takes request stream chunk message consumes received message (here e.g. collects send chunks of string; other example: write received chunks to file).
    - In **onComplete()** method performs necessary steps for all received chunks and produces final response to client – i.e. calls **client's observer's onNext(…)** method (here e.g. sends back sorted concatenated strings; other example: finally close the file), and finally close interaction calling **client's onCompleted()** method:

```
public void onCompleted() {
    ...
    TheResponse response = TheResponse.newBuilder().….build();
    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

    - in **onError(…)** method handle errors in standard way.

In client code:

– Define the response observer exactly the same as for non-blocking (asynchronous) unary call (here below named **StrObs2**),

– Call the procedure passing client's response observer and get the server's observer:

```
StreamObserver<TheRequest> strReqObserver = nonbStub.streamToSrv(
                                                    new StrObs2());
```

- Next produce message chunks (here e.g. bulid messages with text string; other example read chunks of data from file) to send them to server in stream by calling server's request observer's **onNext(...)** method:

  ```
  TheRequest rrr = TheRequest.newBuilder().set... .build();
  ...
  strReqObserver.onNext(rrr);
  ```

  Do the above **onNext(...)** calls in loop as many times as needed to send all data.

- Finally call server's request observer's **onComplete()** method to close streamin and optionally receive final response (in client's response observer):

  ```
  strReqObserver.onComplete();
  ```

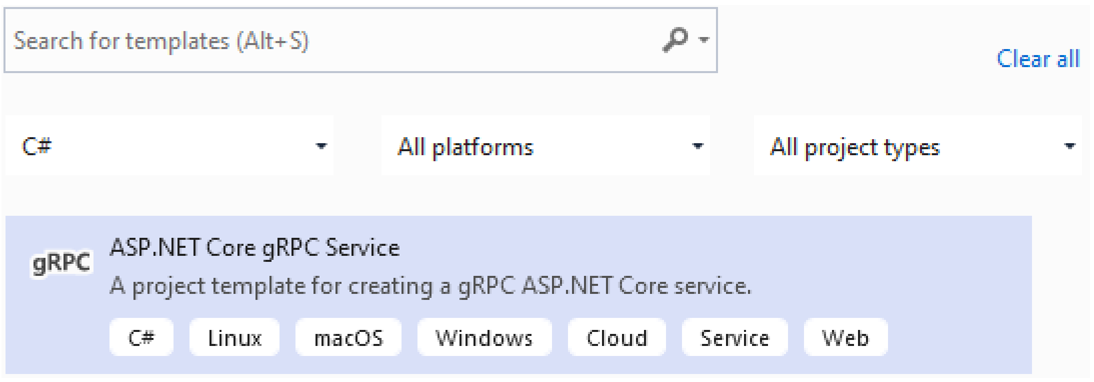
3.2 Visual Studio 2019

Visual Studio 2019 contains dedicated gRPC service application project. It is also possible to use several NuGet packages to build gRPC application components.

Note (from nugget.org):
"*NuGet is the package manager for .NET. The NuGet client tools provide the ability to produce and consume packages. The NuGet Gallery is the central package repository used by all package authors and consumers.*"
NuGet is a package manager aimed to enable developers to share reusable code. NuGet's client (nuget.exe) is a free and open-source command-line app that can create and consume packages. NuGet is distributed as a Visual Studio extension, and it can natively consume NuGet packages.

3.2.1 C# project (new solution) for gRPC service (server).

- To create new gRPC app select *ASP.NET Core gRPC Service*. You can use Search field to find the project faster.



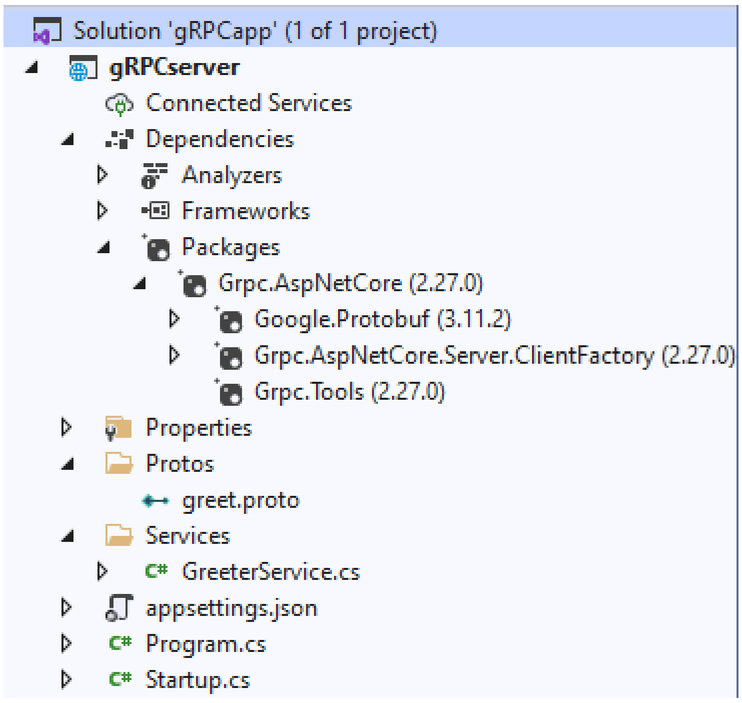
ASP.NET Core gRPC Service
A project template for creating a gRPC ASP.NET Core service.

- After selection of names (in next window – here project name gRPCserver in gRPCapp solution) you can specify *Target Framework* (.Net 5.0 or .NET Core 3.1), and eventually enable *Docker* option (don't do it now).

3.2.2 The project structure

The initial default project consist of (among the others):

- already included necessary packages – **Grpc.AspNetCore**.
- the initial proto file (file defining procedure interface) – **greet.proto**,
- the initial gRPC service - **GreeterService.cs**,
- **appSettings.json** – contains configuration data
- the code for service host (by default Kestrel web server is used) – **Program.cs**,
- the service configuration code – **Startup.cs**,



```
Solution 'gRPCapp' (1 of 1 project)
◢ gRPCserver
      Connected Services
   ◢ Dependencies
      ▷ Analyzers
      ▷ Frameworks
      ◢ Packages
         ◢ Grpc.AspNetCore (2.27.0)
            ▷ Google.Protobuf (3.11.2)
            ▷ Grpc.AspNetCore.Server.ClientFactory (2.27.0)
              Grpc.Tools (2.27.0)
   ▷ Properties
   ◢ Protos
         greet.proto
   ◢ Services
      ▷ GreeterService.cs
   ▷ appsettings.json
   ▷ Program.cs
   ▷ Startup.cs
```

To create your own solution you can as well edit *.cs and proto files as rename or add new ones.

<u>Note the following</u>:

- The default main NuGet package for the default gRPC service project is *GrpcASPNetCore* package which include: *Google.Protobuf*, *Grpc.Tools*, *Grpc/ASPNetCore.Server*, and some other packages.
- The project includes files with methods called by the runtime. It includes:
  - **`Startup`** class that configures services and the app's request pipeline. It has two default methods:
    - **`Configure`** method to create the app's request processing pipeline,
    - **`ConfigureServices`** optional method to configure the app's services. Services are registered here.
  - Enabling gRPC service
    gRPC is enabled with the **`AddGrpc`** method in `ConfigureServices` method.
  - Routing
    Routing in ASP..NET Core is responsible for matching incoming requests (especially HTTP ones) and dispatching those requests to the app's service endpoints.

    The service hosted by ASP.NET Core gRPC, should be added to the routing pipeline with use of the **`UseRouting()`**, the **`UseEndpoints()`**, and the **`MapGrpcService<TService>()`** methods called in the `Configure` method. It adds endpoints to the gRPC service.

- Startup class is typically called by `UseStartup<TStartup>` method when the app's host is built.
- Hosting the service

  Host is the component that, among the others, encapsulates (and run) the services – when it is run it calls `StartAcync` on each service implementation..

  - **`Program`** class (with **`Main`** method) is used to create and run the host for gRPC service.
  - The default host for gRPC service is Kestrel – a cross-platform web server for ASP.NET Core (other host options are also available). Kestrel doesn't have some of the advanced features but provides the best performance and memory utilization. It requires HTTP/2 transport and should be secured with TLS. **Attention**: MacOS doesn't support ASP.NET Core gRPC with TLS.
  - Creation and running the host

    In the main method it is built and run the host with builder method:

    `CreateHostBuilder``(args).Build().Run();`

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
         {
           webBuilder.UseStartup<Startup>();
         }
    );
```

  - Defining the host

    The default HTTP host (Kestrel Web server) is built using:

    `CreateDefaultBuilder(args).`**`ConfigureWebHostDefaults`**`(…);`

    In the method it is specified `Startup` as the startup class with use of `UseStartup<TStartup>` method.

3.2.3 gRPC procedure interface defined in proto file

In proto file the interface used for creating stubs for server and client is defined. In the example the remote procedure named **`GrpcProc`** with two parameters is defined. The procedure returns some calculation result and some message (string value).

- Access the **proto.file** and enter (or modify) the following content that defines:
  - Protocol Buffers version,
  - service and procedure definition (with input and output parameters (messages)),

- input (request) message – parameters in call,
- output (response) message – response for call.

```
syntax = "proto3";
option csharp_namespace = "gRPCserver";
package mygrpc;
// Service definition.
service GrpcService {
  rpc GrpcProc (GrpcRequest) returns (GrpcResponse);
}
// The request message
message GrpcRequest {
  string name = 1;
  int32 age = 2;
}
// The response message
message GrpcResponse {
  string message = 1;
  int32 days = 2;
}
```

- Ensure that namespace in proto file is the same as for gRPC service implementation. When changing names in proto file they must be changed in server implementation accordingly.

3.2.4 The gRPC service (gRPC procedure)

In the solution's Services node skeleton code of remote gRPC procedure is generated.

The class must extend the base class with the name of ***service name defined in proto file*** tagged with ***Base*** word. This class is defined in the class generated from proto file named as ***service name defined in proto file***. (i.e. ***SrvName.SrvName*****Base** class where ***SrvName*** is the name of the service defined in **proto file**.

In the class the method of remote procedure (with the name as in proto file) takes the request and returns the Task object with response.
Note:
- the `Task<…>` is used to allow asynchronous processing,
- the `ServerCallContext` is not used here (is for potential future use).

- Add/modify the code of GrpcProc that returns some message and number of days for provided years:

```
public class MyGrpcService : GrpcService.GrpcServiceBase
{
  ...
  public override Task<GrpcResponse> GrpcProc(GrpcRequest request,
                                              ServerCallContext context)
  {
        string msg;
        int val;
        val = request.Age * 12 * 365;
        msg = "Hello "+request.Name+" being "+request.Age+" years old.";
        return Task.FromResult(new GrpcResponse { Message=msg, Days=val });
      }
    }
```

Pay attention that request and response field names are with capital first letter!

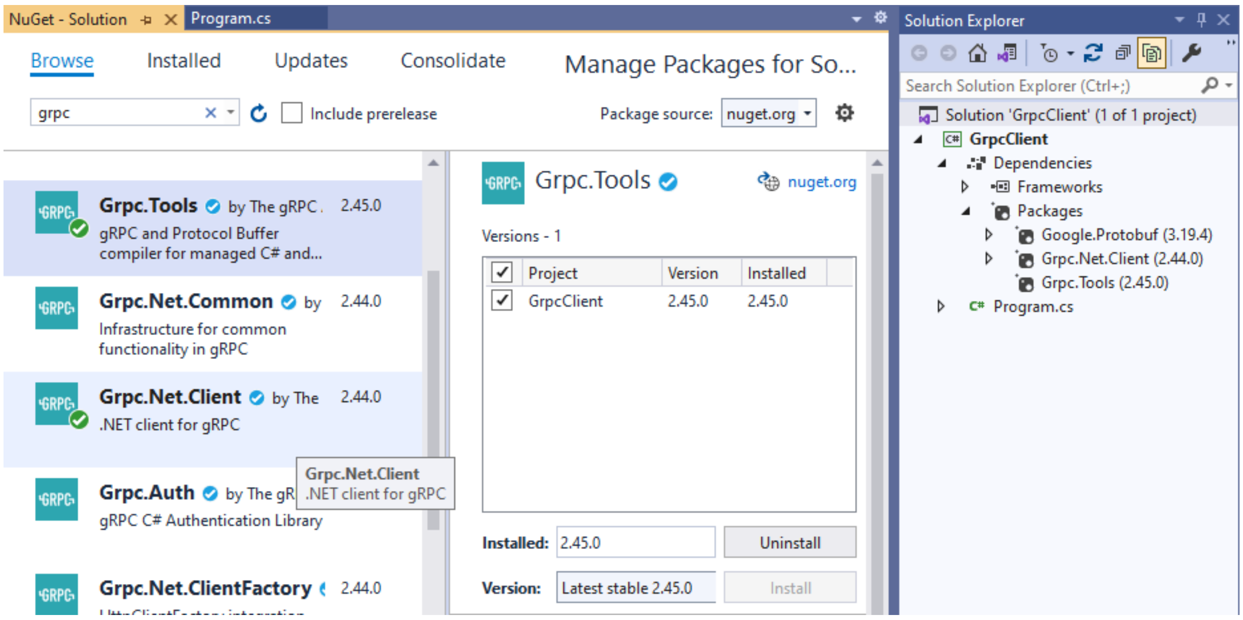
- Build and run the application to verify the correctness.

3.2.5 The gRPC client

Here the client .Net console application will be created.

In order to compile proto files it must be included in the project several packages with use of NuGet manager.

- Create a new project (eventually add one to previous) a *Console Application (.NET Core)* project – here named **gRPCclient**.
- Add NuGet packages: Grpc.Net.Client, Grpc.Tools, Google.Protobuf.
  - Select the project in Solution Explorer window, and from platform menu or context menu select *Manage NuGet Packages* option (or select this option from bar menu).
  - Select *Browse* tab, enter **grpc** in *Search* field.

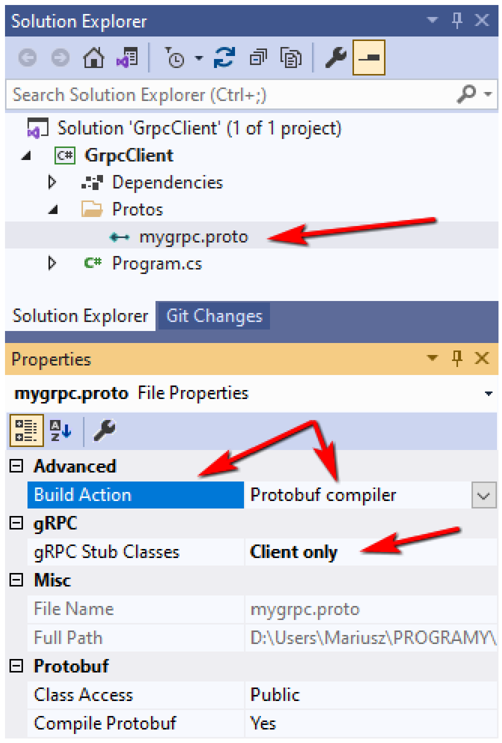


  - Select and add the following packages to the project:
    - Grpc.Net.Client – .Net client for gRPC,
    - Grpc.Tools – gRPC and Protocol Buffers compiler.
  - Enter **protobuf** in *Search* field.
    Select and add the following package:
    - Google.Protobuf – C# runtime library for Protocol Buffers.

- Add/create new folder (e.g. Protos) in project and copy/create **greet.proto** proto file. Ensure (change) the namespace to the same as for the client.
- Do one of the following:
  - Select the proto file in *Solution Explorer* window and bellow in *Properties* window select in *Build Action* field the *Protobuf compiler* action.

  or:

  - Select the client project node in *Solution Explorer*, select the context menu *Edit Project File*, and ensure (alternatively enter/correct) the in the file is included section:



Advanced	
Build Action	Protobuf compiler
gRPC	
gRPC Stub Classes	**Client only**
Misc	
File Name	mygrpc.proto
Full Path	D:\Users\Mariusz\PROGRAMY\
Protobuf	
Class Access	Public
Compile Protobuf	Yes

```
<ItemGroup>
  <Protobuf Include="Protos\mygrpc.proto" GrpcServices="Client" />
</ItemGroup>
```

Especially check if **`Client`** is set as `GrpcServices` value.

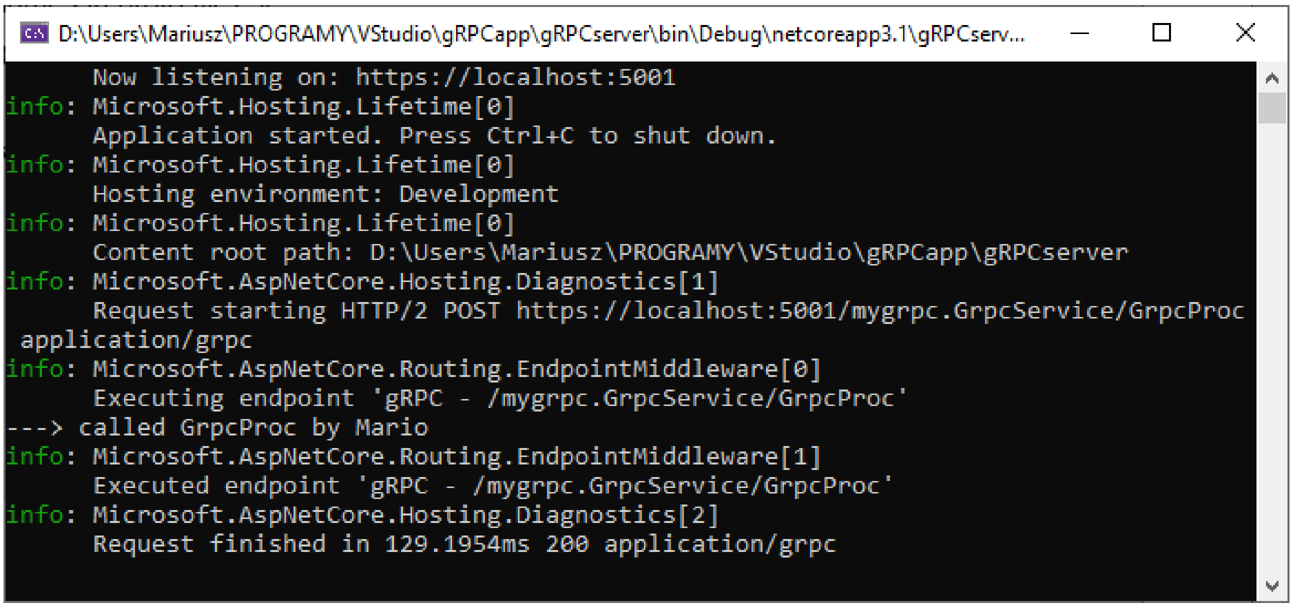
- Enter the code of client `Main` method in which you will:
  - create a channel for communication with the server (specifying the address and the same port as in server (here we run the server on localhost),
  - create client object – to make requests,
  - call the remote procedure asynchronously passing input parameters,
  - display results,
  - close the channel when no more used.

```
static async Task Main(string[] args)
{
  Console.WriteLine("Starting gRPC Client");

  using var channel = GrpcChannel.ForAddress("https://localhost:5001");
  var client = new GrpcService.GrpcServiceClient(channel);

  Console.Write("Enter the name: ");
  String str = Console.ReadLine();
  int val = 21;

  var reply = await client.GrpcProcAsync(new GrpcRequest { Name=str ,
                                                          Age=val});

  Console.WriteLine("From server: " + reply.Message);
  Console.WriteLine("From server: "+val+" years = "+reply.Days+" days");
  Console.WriteLine("Press any key to exit...");
  Console.ReadKey();

  channel.ShutdownAsync().Wait();
}
```

Note:
The client class is generated from the interface defined in proto file. This class is defined in the class named as ***service name in proto file***. The client class name is ***service name in proto file*** tagged with ***Client*** word. Here the client object is built with the GrpcService.GrpcServiceClient class generated from proto file (names correspond to names in proto file).

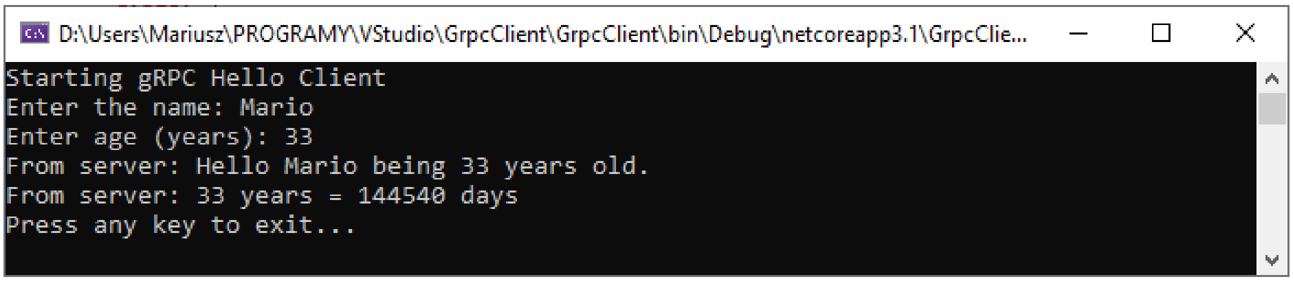
1. Build the solution (compile/build all projects) and run the application (server first, and then client).
   - Check the operation of the application.
   - The result should be similar to below figures:

Server running in separate window (one process):


D:\Users\Mariusz\PROGRAMY\VStudio\gRPCapp\gRPCserver\bin\Debug\netcoreapp3.1\gRPCserv...

```
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\Users\Mariusz\PROGRAMY\VStudio\gRPCapp\gRPCserver
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5001/mygrpc.GrpcService/GrpcProc
 application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /mygrpc.GrpcService/GrpcProc'
---> called GrpcProc by Mario
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /mygrpc.GrpcService/GrpcProc'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 129.1954ms 200 application/grpc
```

Client running in separate window (other process):


D:\Users\Mariusz\PROGRAMY\VStudio\GrpcClient\GrpcClient\bin\Debug\netcoreapp3.1\GrpcClie...

```
Starting gRPC Hello Client
Enter the name: Mario
Enter age (years): 33
From server: Hello Mario being 33 years old.
From server: 33 years = 144540 days
Press any key to exit...
```

4 Exercise – Part II

The detailed and final requirements for Part II are set by the teacher.

A. Develop or prepare to develop a program according to the teacher's instructions.