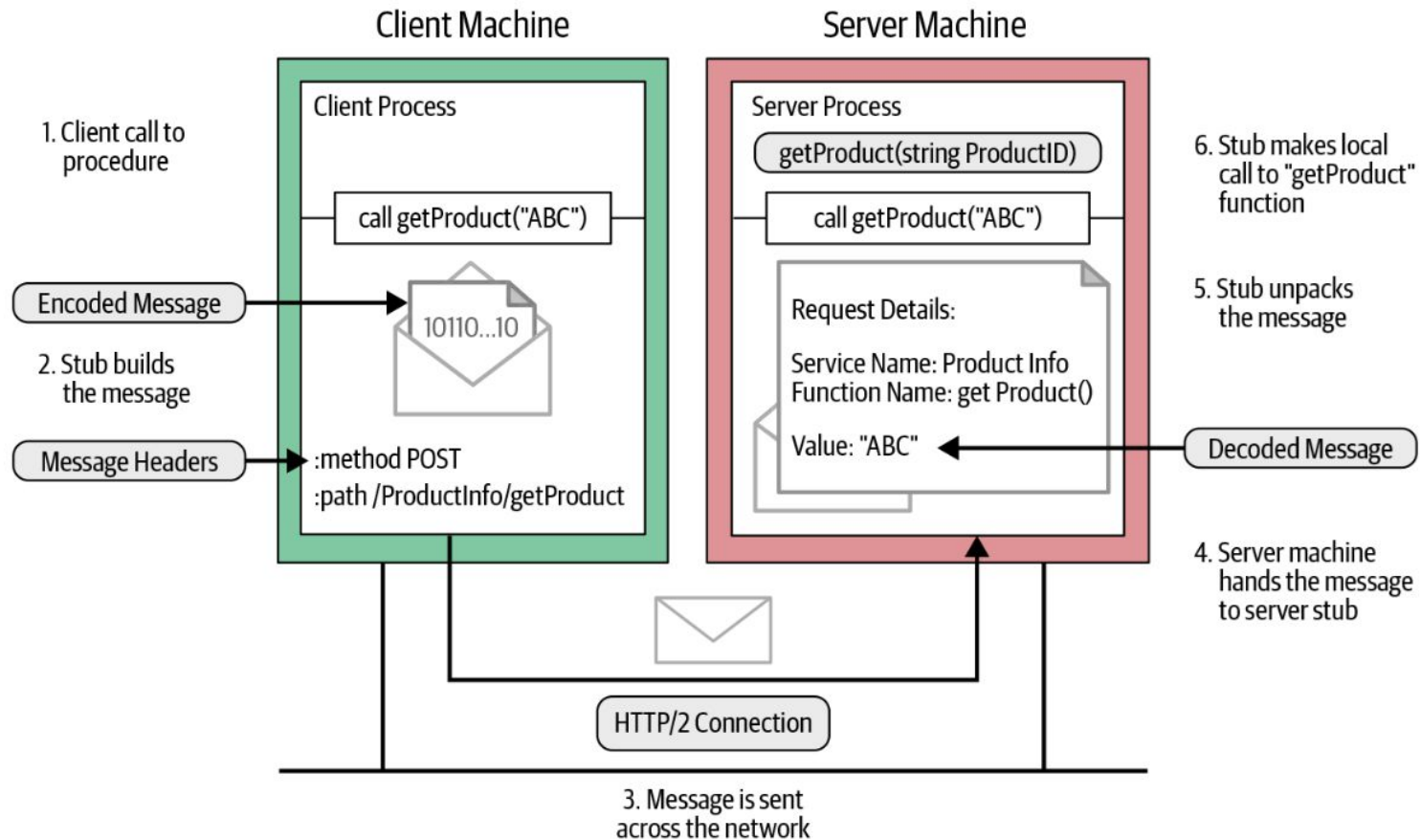


# gRPC : Google Remote Procedure Call

Prof. (Dr.) Vipul Dabhi  
Dept. of Information Technology,  
D. D. University

# How RPC Works?



# Communication between Services

- In microservice architecture, each functionality is developed as a separate service. The interaction between these services becomes an important aspect of development.
- Microservices must exchange data, call each other, and maintain high performance, reliability, and scalability.
- There are multiple ways for communication between services
  - RESTful Services
    - RESTful services can be bulky and inefficient.
    - The bulkiness of RESTful services is because of the fact that most implementation relies on JSON, a text-based encoding format.
    - The JSON format uses lots of space compared to the binary format.
    - Another issue with using JSON-based API is that schema support is optional.
  - SOAP (XML Based) Services
  - gRPC
- gRPC provides better features compare to other ways of message exchange

# Public Vs Private APIs

- There are two types of APIs:
  - Public API (REST API)
    - These APIs are consumed by client applications, typically browsers, mobile applications, or other applications.
    - The de facto standard for using Public API is REST over HTTP.
  - Private API (gRPC API)
    - These APIs are not exposed to the outside world, and it's mainly used for inter-service communication within your application
    - For private APIs, you can think of using gRPC.
- REST APIs and gRPC APIs refer to different Architectural Styles for building APIs.
- Both these APIs are used to connect microservices or applications. However, the way these APIs connect microservices is different.

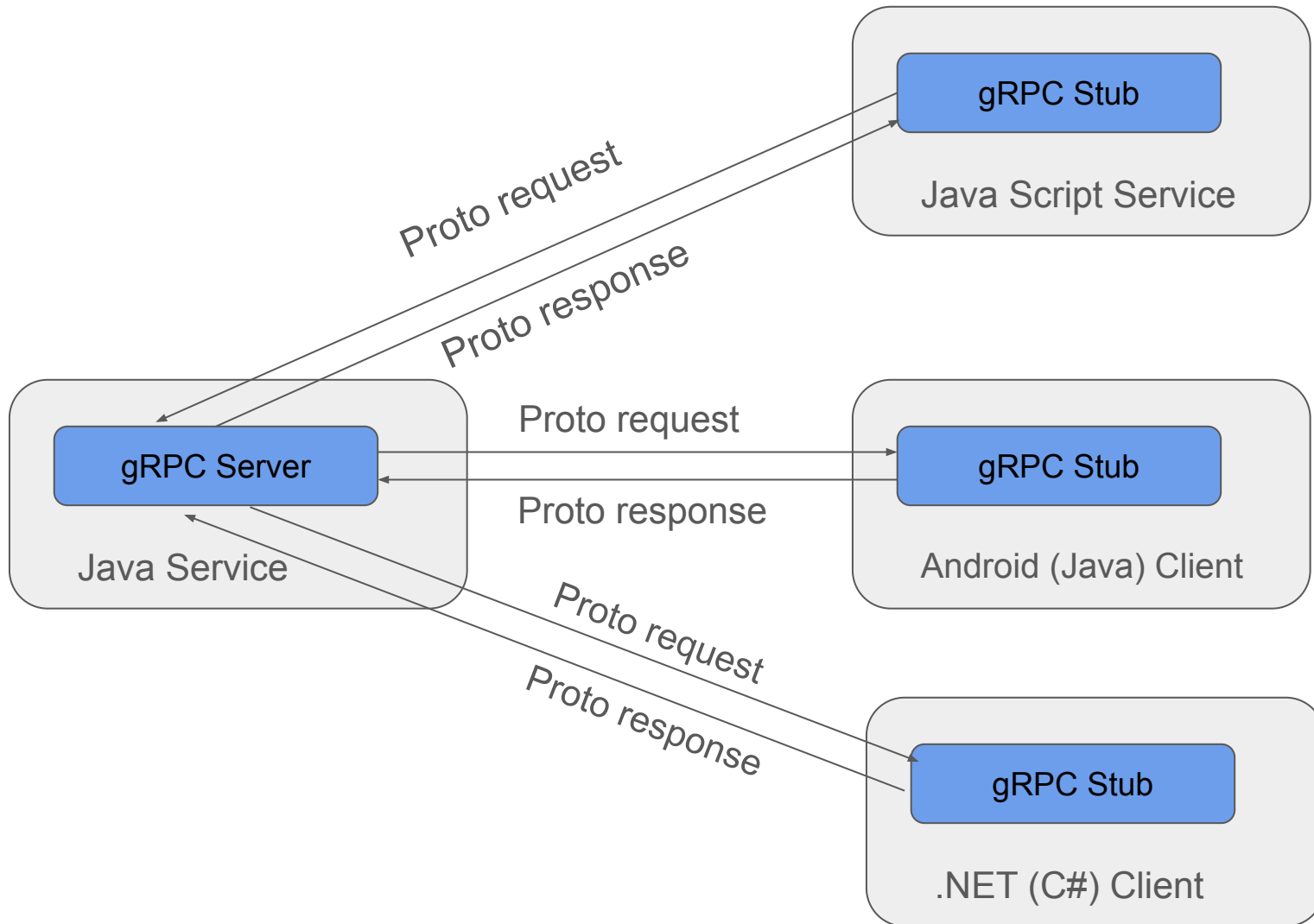
# Features of gRPC

- The gRPC uses HTTP/2 protocol as transport for communication which is faster than REST
- Message exchanged in the form of Protocol Buffers (Google's mature open source mechanism for serializing structured data)
- We can generate the code using .proto files
- Allows bi-directional streaming
- Requires gRPC -web to invoke

Reference:

<https://medium.com/@ankithahjpgowda/grpc-implementation-in-springboot-and-microservices-366dc7a66c5a>

# gRPC Architecture

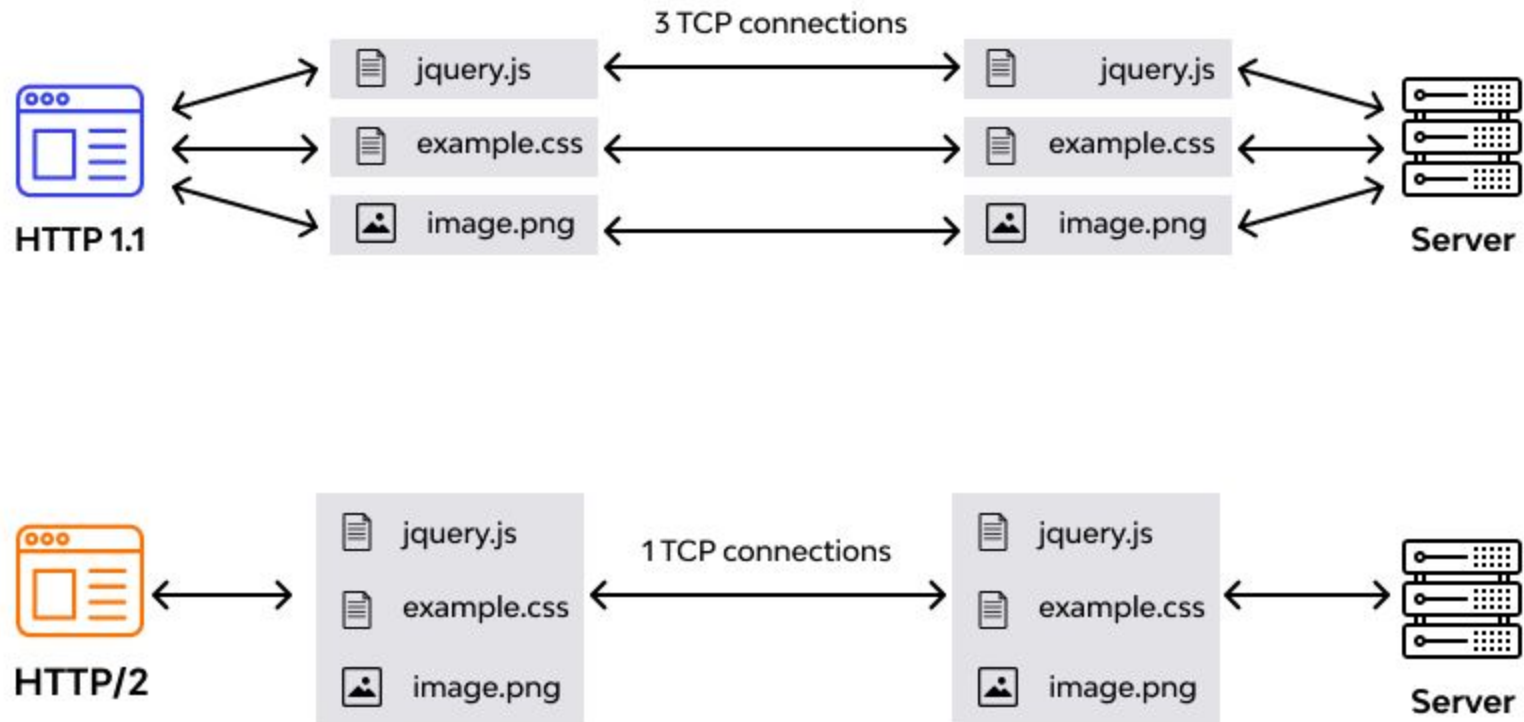


# HTTP/1.1 Vs HTTP/2

- HTTP 1.1 has
  - Textual format
  - Plain Text Headers
  - TCP connection requires 3 way handshake process (single request and response with 1 single TCP connection)
- HTTP/2 has
  - Binary format
  - Header Compression
  - Flow Control
  - Multiplexing (Same TCP connection can be reused for multiplexing. Server streaming — Client streaming — Bi-directional streaming is possible)
- HTTP/2 is supported on almost all popular web browsers, such as Chrome, Firefox, Internet Explorer, and Safari

# HTTP/1.1 Vs HTTP/2

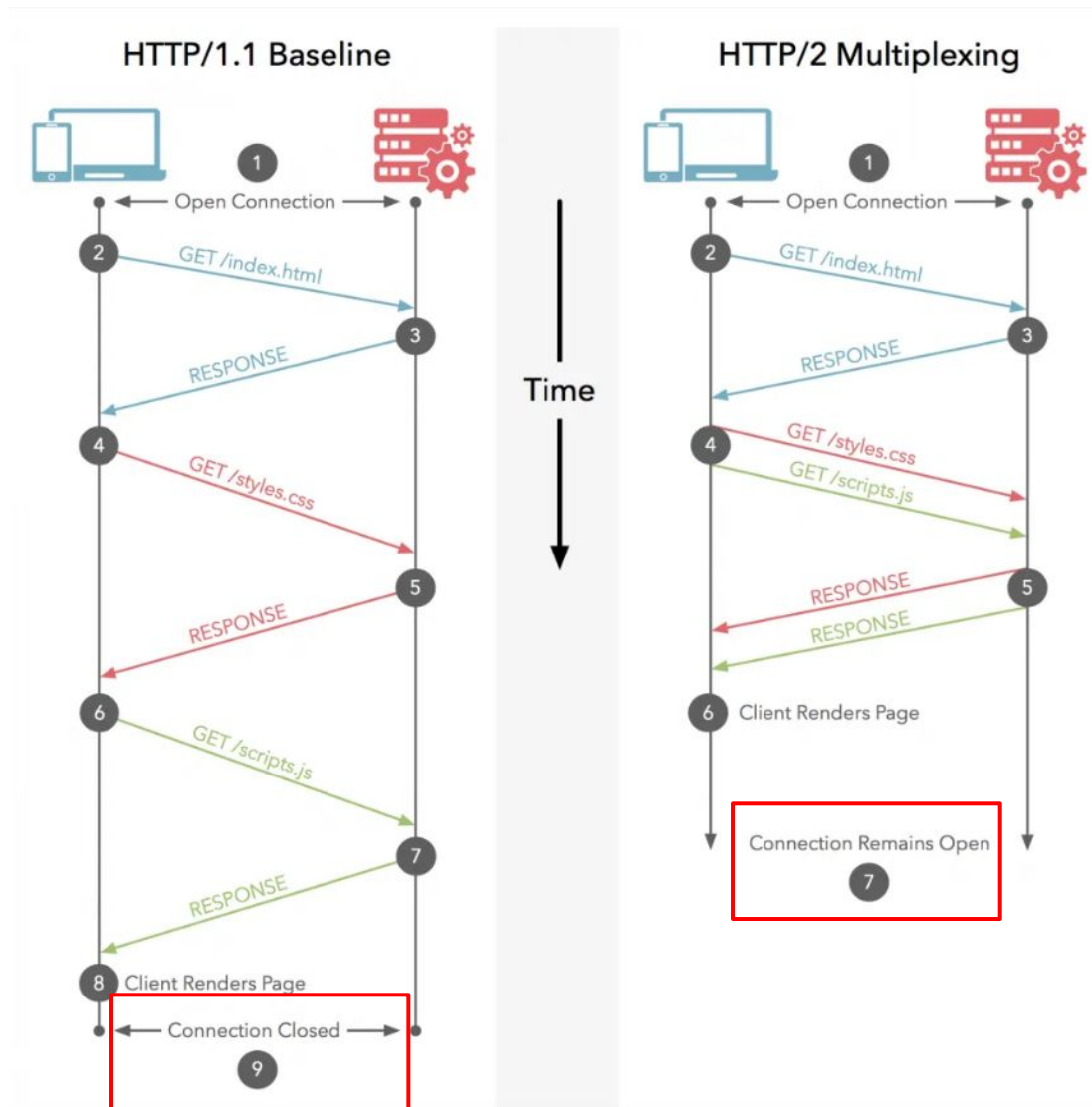
## Multiplexing



Source: <https://www.wallarm.com/what/what-is-http-2-and-how-is-it-different-from-http-1>



# HTTP/1.1 Vs HTTP/2

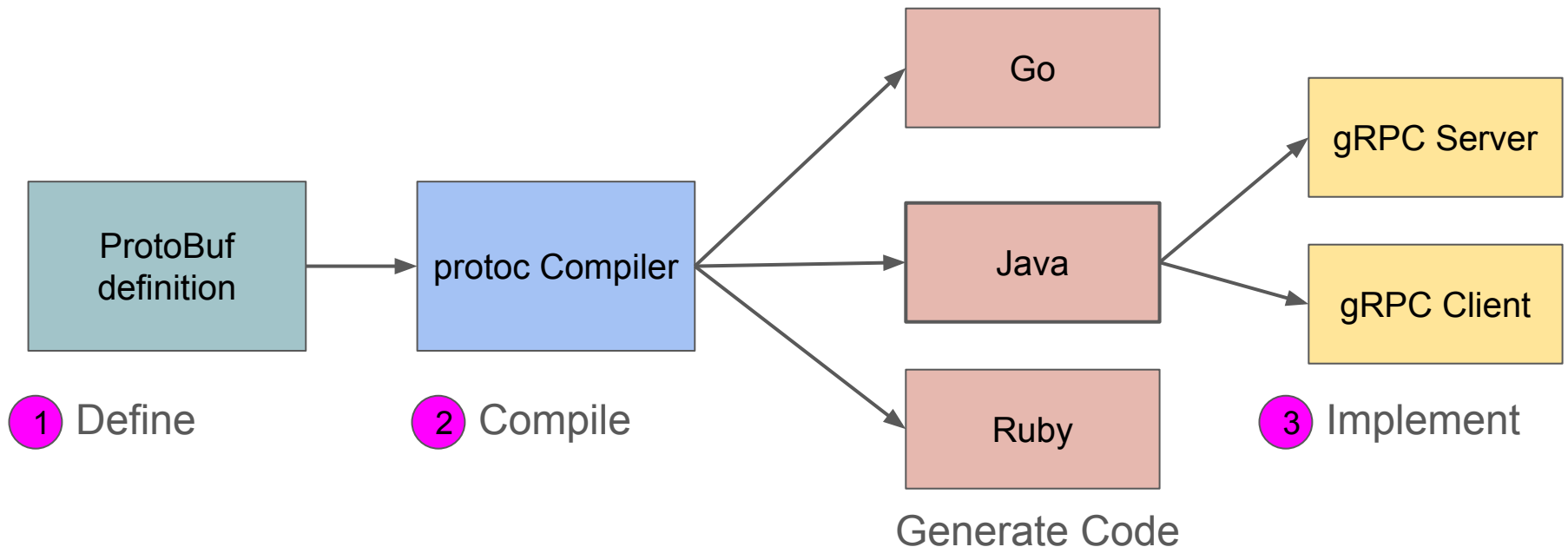


# gRPC

- gRPC is a protocol for transmitting messages between systems in an efficient manner.
- It leverages
  - HTTP/2 for transport
  - employs Protocol Buffers as the Interface Definition Language (IDL)
  - Provides features such as authentication, load balancing, and more.
- Using gRPC protocol, services can communicate in real-time with low latency and high throughput.

# gRPC

- Protocol Buffers (protobufs for short) allow you to define simple data structures in a language-neutral way.
- After defining data structures, you can use the protobuf compiler to generate data access classes in various languages.
- This approach ensures that gRPC can work seamlessly across different platforms and languages, providing a universal language for microservices communication.



# gRPC

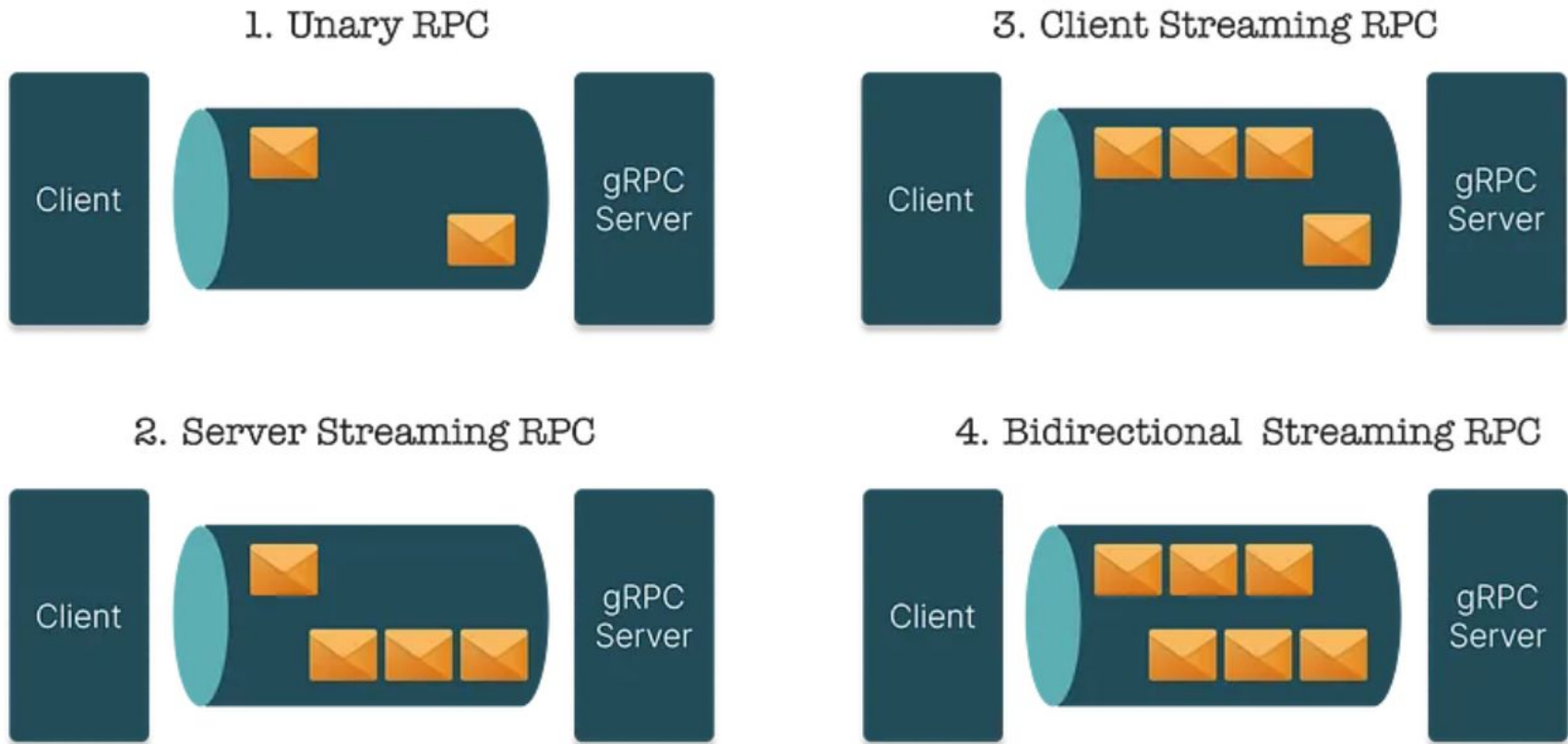
- The gRPC protocol is built on top of HTTP/2, which provides a foundation for long-lived, real-time communication streams.
- The protocol introduces three concepts: channels, remote procedure calls (RPCs), and messages.
- Each channel may have a number of RPCs, while each RPC may have a number of messages.



# Client - Server Communication in gRPC

- There are four different ways for client - server communication in gRPC
  - Unary RPC:
    - The client sends a single request and gets back a single response.
  - Server Streaming RPC
    - Similar to a unary RPC, except that the server returns a stream of messages in response to a client's request.
  - Client Streaming RPC
    - Similar to a unary RPC, except that the client sends a stream of messages to the server and the server responds with a single message.
  - Bidirectional Streaming RPC
    - In a bidirectional streaming RPC, the call is initiated by the client invoking the method. The two streams are independent, so the client and server can read and write messages in any order.

# Client - Server Communication in gRPC



Client-Server communication in gRPC

# @GrpcService Annotation in Spring Boot

- @GrpcService annotation is a part of the gRPC Spring Boot starter library
  - It provides a seamless integration between gRPC and Spring Boot
  - Developers can define gRPC services within a Spring Boot application using this annotation
  - This annotation handles much of the boilerplate and configuration automatically.
  - This annotation tells Spring that the annotated class is a gRPC service, and Spring automatically configures the server to handle the gRPC requests directed at the service
  - Allows developers to build faster and more efficient microservices

# Auto-Configuration

- By including the `grpc-spring-boot-starter` in our project, Spring Boot's auto-configuration mechanism will be available.
  - Conditional on Class and Property: The auto-configuration classes provided by the `grpc-spring-boot-starter` are conditional. They check for the presence of gRPC classes in the classpath and certain properties in our application's configuration. If the conditions are met, the auto-configuration is activated.
  - Creating gRPC Server Bean: The starter configures a Server bean (a gRPC server instance) in the Spring application context. The configuration details, such as the server's port, security settings, and others, can be specified in the application's properties file.
  - Configuring Server with SSL/TLS (Optional): We can set up the gRPC server to use SSL/TLS, (ensuring encrypted communication) by setting SSL / TLS settings in our `application.properties` file.



# Service Discovery and Registration

- Once the gRPC server is auto-configured, the next step is to discover and register your gRPC services so they can handle incoming RPC calls. This involves finding your service implementations within the Spring application context and registering them with the gRPC server
  - Scanning for @GrpcService Annotations: The starter scans the Spring application context for beans annotated with @GrpcService. This custom annotation is provided by the grpc-spring-boot-starter and is used to **mark gRPC service implementations for automatic discovery**.
  - Instantiating Service Definitions: For each discovered service, **the starter instantiates a service definition. This is a combination of the service implementation (your subclass of the generated base class) and metadata about the service (such as its name and methods)**. The instantiation involves creating a wrapper around your implementation that adheres to the gRPC framework's requirements.
  - Registering Services with the Server: **Each instantiated service definition is registered with the gRPC server. This registration is essential for the server to know which service implementation to delegate the incoming RPC calls to.** The gRPC framework uses the service definition to match incoming requests with the correct service and method, and then invokes the appropriate method on your implementation.

# Application Context


- With the gRPC services registered, the final step is starting the gRPC server:
  - Lifecycle Management: The `grpc-spring-boot-starter` ties the lifecycle of the gRPC server to the Spring application's lifecycle. This ensures that the server starts after all beans are initialized and configured and shuts down gracefully when the application stops.
  - Listening for Requests: Once started, the gRPC server listens on the configured port for incoming gRPC calls. The HTTP/2 protocol, used by gRPC, allows for efficient, multiplexed communication, making it well-suited for microservices architectures.

# Spring Initializr

← → ↺ start.spring.io/

☰

🕒

 **spring** initializr

**Project**

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ **Java** ☐ Kotlin ☐ Groovy

☒ **Maven** ← 1. Select Maven

**Spring Boot**

☐ 3.3.2 (SNAPSHOT) ☒ **3.3.1** ☐ 3.2.8 (SNAPSHOT) ☐ 3.2.7

**Project Metadata**

Group

Artifact  ← 2. Provide artifact for server

Name

Description

Package name  ← 3. Check package name

Packaging ☒ **Jar** ☐ War

Java ☐ 22 ☐ 21 ☒ **17**

**Dependencies** ADD DEPENDENCIES... CTRL + B

No dependency selected

4. Click Explore

🐙

**GENERATE** CTRL + G **EXPLORE** CTRL + SPACE **SHARE...**

# Explore pom.xml

The screenshot shows the start.spring.io web interface. On the left is a file explorer for a project named 'server.zip'. The files listed are .gitignore, .mvn, HELP.md, mvnw, mvnw.cmd, pom.xml (highlighted), and src. On the right is a code editor displaying the content of pom.xml. The code includes properties for java.version, dependencies for spring-boot-starter and spring-boot-starter-test, and a build section with a spring-boot-maven-plugin. At the top right of the code editor are 'DOWNLOAD' and 'COPY' buttons. At the bottom of the interface are 'DOWNLOAD CTRL + ⌘' and 'CLOSE ESC' buttons. A red arrow points from the text 'Press Close' to the 'CLOSE ESC' button.

start.spring.io/

server.zip

- .gitignore
- .mvn
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml**
- src

29 <properties>  
30 <java.version>17</java.version>  
31 </properties>  
32 <dependencies>  
33 <dependency>  
34 <groupId>org.springframework.boot</groupId>  
35 <artifactId>spring-boot-starter</artifactId>  
36 </dependency>  
37  
38 <dependency>  
39 <groupId>org.springframework.boot</groupId>  
40 <artifactId>spring-boot-starter-test</artifactId>  
41 <scope>test</scope>  
42 </dependency>  
43 </dependencies>  
44  
45 <build>  
46 <plugins>  
47 <plugin>  
48 <groupId>org.springframework.boot</groupId>  
49 <artifactId>spring-boot-maven-plugin</artifactId>

DOWNLOAD COPY

Press Close

DOWNLOAD CTRL + ⌘ CLOSE ESC

# Generate zip file

start.spring.io/

**Project**

☐ Gradle - Groovy ☒ **Java** ☐ Kotlin ☐ Groovy

☐ Gradle - Kotlin ☒ **Maven**

**Spring Boot**

☐ 3.3.2 (SNAPSHOT) ☒ **3.3.1** ☐ 3.2.8 (SNAPSHOT) ☐ 3.2.7

**Project Metadata**

Group

Artifact

Name

Description

Package name

Packaging ☒ **Jar** ☐ War

Java ☐ 22 ☐ 21 ☒ **17**

**Dependencies** [ADD DEPENDENCIES...](#) CTRL + B








No dependency selected

**1. Click GENERATE**

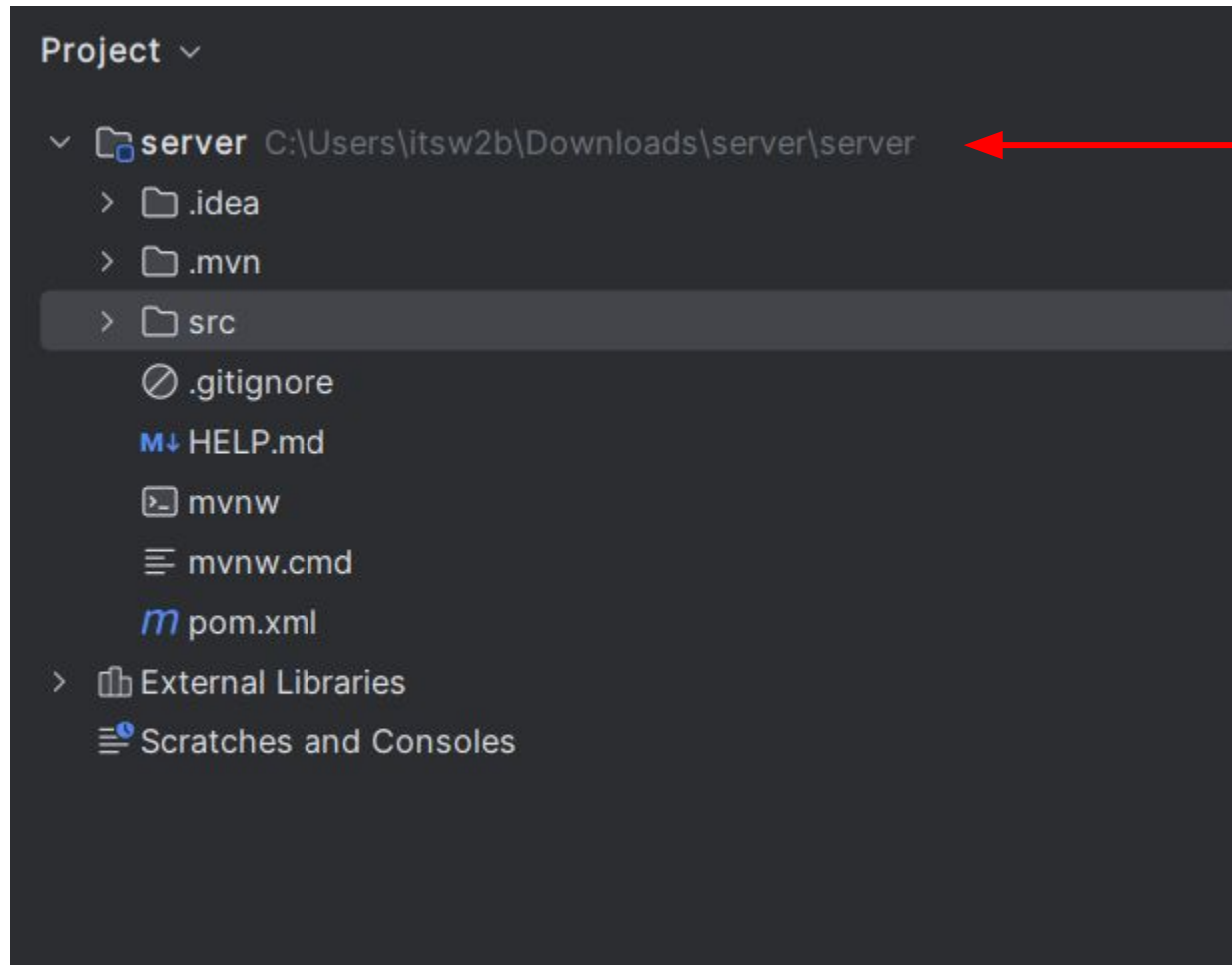
**2. Downloading of zip file will start**

**GENERATE** CTRL + G **EXPLORE** CTRL + SPACE **SHARE...**

# Download / Extract zip file

Name	Date modified	Type	Size
Today			
 .gitignore	28-06-2024 16:54	Git Ignore Source ...	1 KB
 HELP	28-06-2024 16:54	Markdown Source ...	1 KB
 mvnw	28-06-2024 16:54	File	11 KB
 mvnw	28-06-2024 16:54	Windows Comma...	7 KB
 pom.xml	28-06-2024 16:54	xmlfile	2 KB
 .mvn	28-06-2024 16:54	File folder	
 src	28-06-2024 16:54	File folder	

# Open root folder of project in IntelliJ Idea



Open root folder of the project in IntelliJ Idea

# Adding gRPC Server Dependencies in pom.xml

```
<dependency>
  <groupId>net.devh</groupId>
  <artifactId>grpc-server-spring-boot-starter</artifactId>
  <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-netty</artifactId>
  <version>1.63.0</version>
</dependency>
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

These dependencies are useful to define and implement gRPC services



# Add the plugins in pom.xml

```
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.7.1</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.graalvm.buildtools</groupId>
      <artifactId>native-maven-plugin</artifactId>
      <configuration>
        <buildArgs>--trace-class-initialization=org.slf4j.LoggerFactory -H:+ReportExceptionStackTraces</buildArgs>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>add-source</id>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-source</goal>
          </goals>
          <configuration>
            <sources>
              <source>${project.build.directory}/generated-sources/protobuf/java</source>
            </sources>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

# Add the plugins in pom.xml

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.6.1</version>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>compile-custom</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <protocArtifact>com.google.protobuf:protoc:3.25.1:exe:${os.detected.classifier}</protocArtifact>
    <pluginId>grpc-java</pluginId>
    <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.63.0:exe:${os.detected.classifier}</pluginArtifact>
  </configuration>
</plugin>
</plugins>
</build>
```

# Sync changes of pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>server</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>server</name>
    <description>Demo project for GRPC Spring Boot </description>
    <url/>
    <licenses>
        <license/>
    </licenses>
    <developers>
        <developer/>
    </developers>
    <scm>
        <connection/>
        <developerConnection/>
    </scm>

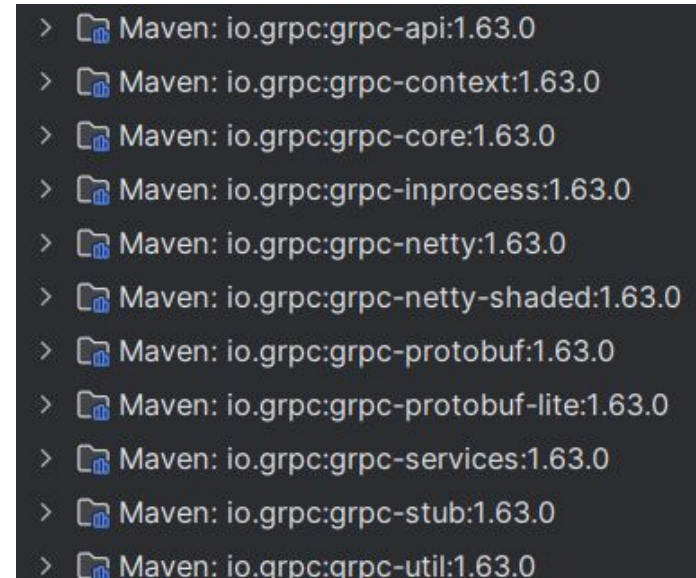
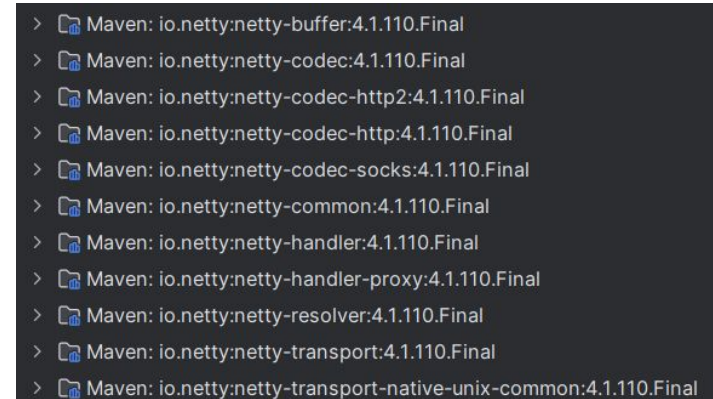
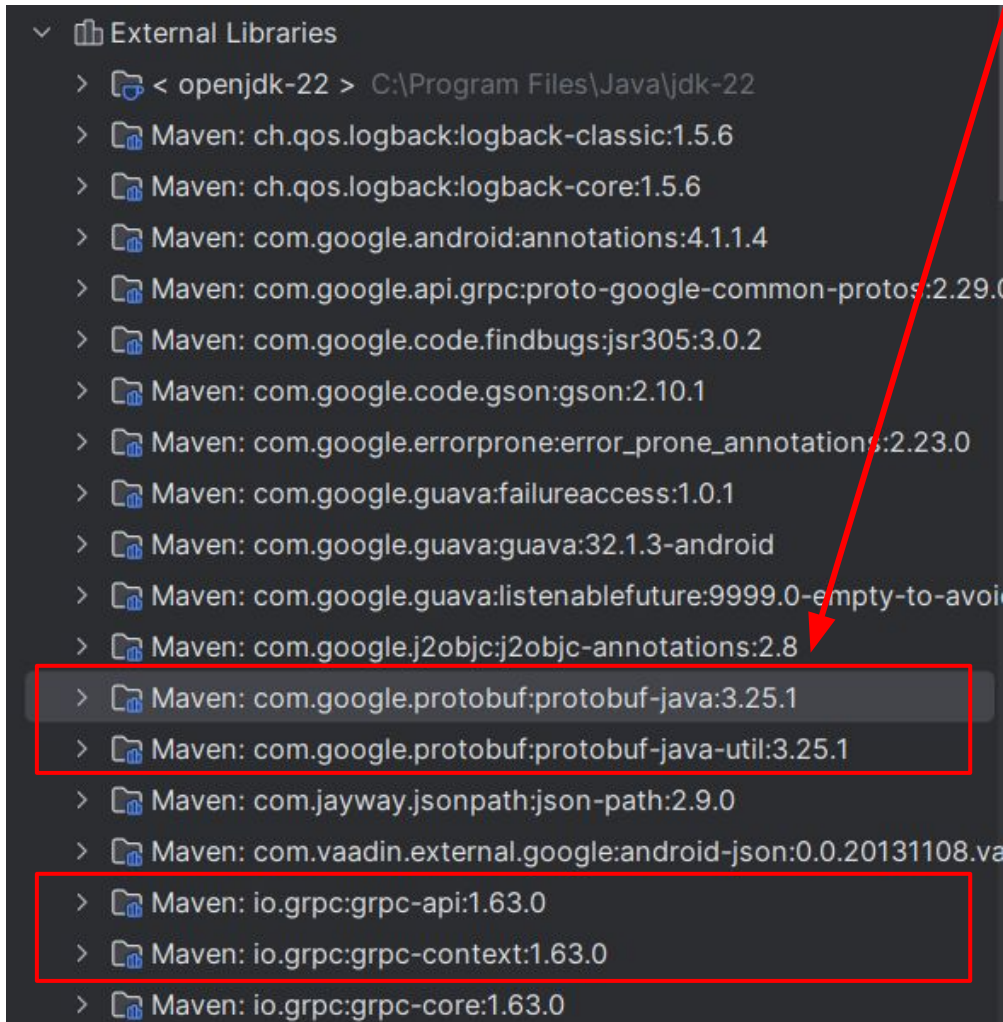
```

Sync changes made in pom.xml

Load Maven Change  
Maven project struct  
changed. Load chang  
IDEA to make it work

# Check External Libraries

Check required dependencies (jar files) downloaded under External Libraries.

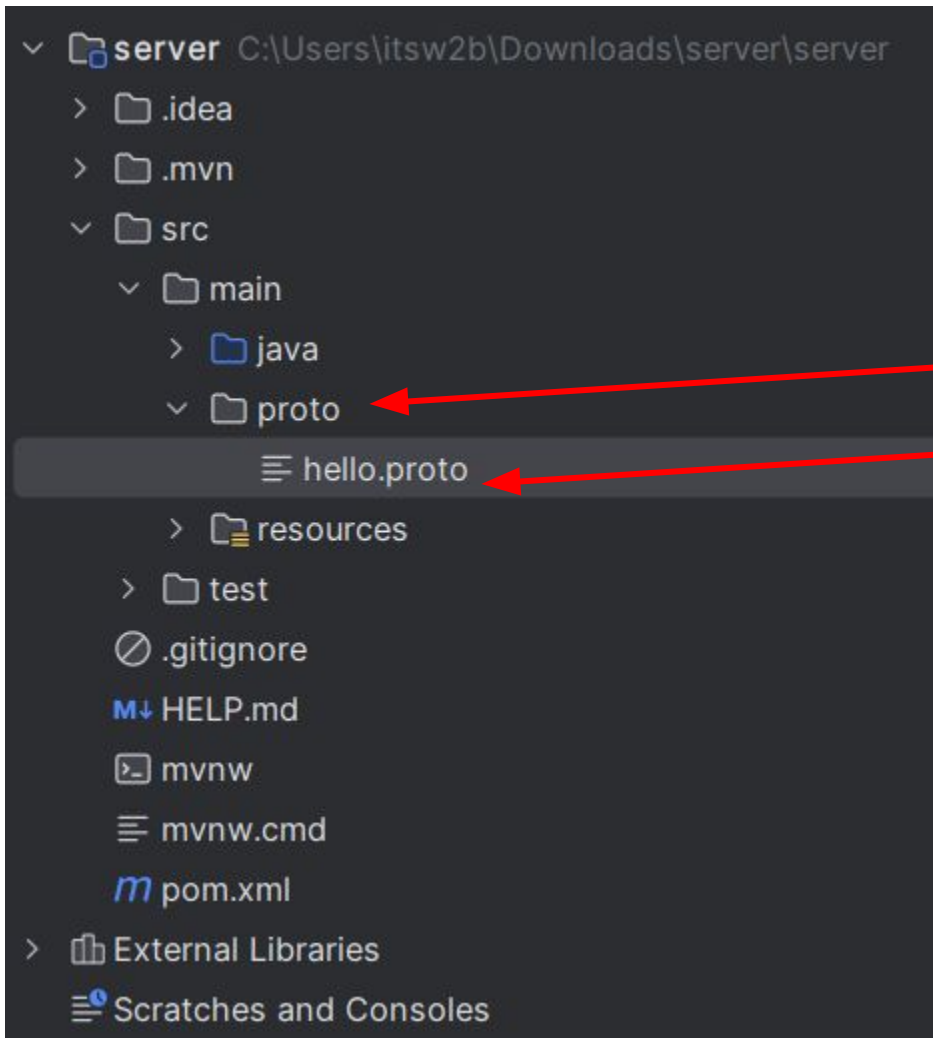


# Implement gRPC Service

The implementation of gRPC Service involves two steps:

- (1) definition of the service in a Protocol Buffer (.proto) file
  - (a) Create a New Directory to hold hello.proto file
  - (b) Write hello.proto file
  - (c) Compile hello.proto file
- (2) implementation of the service using the `@GrpcService` annotation

# Create a proto folder and add hello.proto file



1. Create a proto folder under main directory

2. Create hello.proto folder under proto directory

(1) Create a New Directory:  
/src/main/proto

(a) This directory will hold .proto file:  
hello.proto



# Write hello.proto file

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.example.server.proto";
option java_outer_classname = "HelloWorldProto";

service Simple {
  rpc SayHello (HelloRequest) returns (HelloReply) {
  }
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

option java\_multiple\_files = true;

The compiler will create separate .java files for each of the classes which it will generate for each top-level message and service declared in the .proto file.

Service name

Method name

# Compile hello.proto file

## (3) Compile hello.proto file

Use the Protocol Buffer compiler (protoc) to generate the data access classes in your specified language (Java in this case).

This can often be automated within your build process, with plugins available for Maven and Gradle.

Use the protobuf-maven-plugin to generate Java classes from your .proto file.

Run the command :

```
.\mvnw package
```



# Build Files

The screenshot shows an IDE interface with a project explorer on the left, a code editor on the right, and a terminal at the bottom.

**Project Explorer:** The project structure includes folders for `.idea`, `.mvn`, `src` (containing `main` and `test`), `.gitignore`, `HELP.md`, `mvnw`, `mvnw.cmd`, `pom.xml`, and `External Libraries`. The `src/main/proto` directory is expanded, showing `hello.proto` and `resources/application.properties`.

**Code Editor:** The `hello.proto` file is open. It contains the following content:

```
1 syntax = "proto3";
2
3 option java_multiple_files = true;
4 option java_package = "com.example.server.proto";
5 option java_outer_classname = "HelloWorldProto";
6
7 service Simple {
8     rpc SayHello (HelloRequest) returns (HelloReply) {
9     }
10 }
11
12 message HelloRequest {
13     string name = 1;
14 }
15
16 message HelloReply {
17     string message = 1;
18 }
```

A notification bar at the top of the code editor states: "Plugins supporting \*.proto files found."

**Terminal:** The terminal shows the command `PS C:\Users\itsw2b\Downloads\server\server> .\mvnw package` being entered. The command is highlighted with a red box.

Observe that there is no directory "target" before executing "mvnw package" command.

# Build Files

The screenshot shows an IDE interface. On the left, the project structure is displayed. The 'target' folder is highlighted with a red rectangle. Inside 'target', there are several subfolders including 'classes', 'generated-sources', 'generated-test-sources', 'maven-archiver', 'maven-status', 'protoc-dependencies', 'protoc-plugins', 'surefire-reports', 'test-classes', and 'test-ids'. At the bottom of the 'target' folder, there are two files: 'server-0.0.1-SNAPSHOT.jar' and 'server-0.0.1-SNAPSHOT.jar.original'. A red arrow points from the 'target' folder to a text box on the right that says: "Different directories containing source code gets generated under 'target' folder."

On the right side of the IDE, a snippet of Protobuf code is shown:

```
4 option java_package = "com.example.server.proto";
5 option java_outer_classname = "HelloWorldProto";
6
7 service Simple {
8   rpc SayHello (HelloRequest) returns (HelloReply) {
9   }
10 }
11
12 message HelloRequest {
13   string name = 1;
14 }
15
16 message HelloReply {
17   string message = 1;
18 }
```

At the bottom, a terminal window shows the output of a Maven command:

```
S C:\Users\itsw2b\Downloads\server\server> .\mvnw package
[INFO] Scanning for projects...
[INFO] -----
[INFO] Detecting the operating system and CPU architecture
[INFO] -----
[INFO] os.detected.name: windows
[INFO] os.detected.arch: x86_64
[INFO] os.detected.bitness: 64
```

# Code of ServerApplication.java file

```
package com.example.server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.example.server.proto.HelloReply;
import com.example.server.proto.HelloRequest;

import io.grpc.stub.StreamObserver;
import net.devh.boot.grpc.server.service.GrpcService;

import com.example.server.proto.SimpleGrpc;

@SpringBootApplication
public class ServerApplication {

    public static void main(String[] args) { SpringApplication.run(ServerApplication.class, args); }

}

@GrpcService no usages
class GrpcServerService extends SimpleGrpc.SimpleImplBase {

    @Override 1 usage
    public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
        HelloReply reply = HelloReply.newBuilder().setMessage("Hello Faculty ==> " + req.getName()).build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }

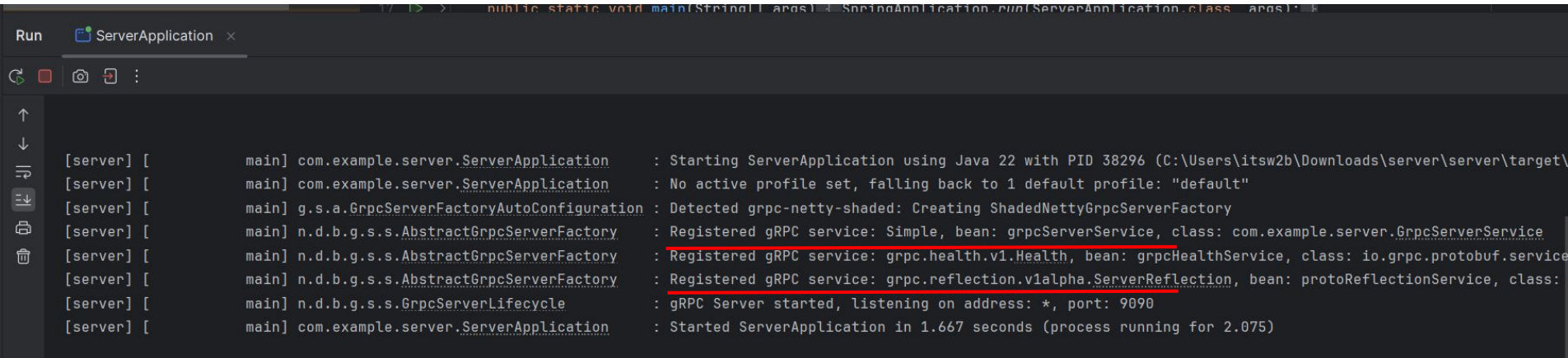
}
```

# Run Application

```
13
14 @SpringBootApplication
15 ▶ Run 'ServerApplicat....main()'
16   Debug 'ServerApplicat....main()'
17   Run 'ServerApplicat....main()' with Coverage
18   Run with Profiler Ctrl+Shift+F10
19   g[] args) { SpringApplication.run(ServerApplication.class, args); }
20
21 }
22
23 @GrpcService no usages
24 class GrpcServerService extends SimpleGrpc.SimpleImplBase {
25
26     @Override 1 usage
27     Ⓜ@ public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
28         HelloReply reply = HelloReply.newBuilder().setMessage("Hello Faculty ==> " + req.getName()).build();
29         responseObserver.onNext(reply);
30         responseObserver.onCompleted();
31     }
32 }
```

# Run Application

- Run your Spring Boot application and observe the console logs to verify that the gRPC server starts successfully. Look for log entries indicating that the gRPC server has started and is listening on the configured port.



The screenshot shows an IDE window titled "Run" with a tab for "ServerApplication". The console displays the following log output:

```
[server] [main] com.example.server.ServerApplication : Starting ServerApplication using Java 22 with PID 38296 (C:\Users\itsw2b\Downloads\server\server\target\
[server] [main] com.example.server.ServerApplication : No active profile set, falling back to 1 default profile: "default"
[server] [main] g.s.a.GrpcServerFactoryAutoConfiguration : Detected grpc-netty-shaded: Creating ShadedNettyGrpcServerFactory
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: Simple, bean: grpcServerService, class: com.example.server.GrpcServerService
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: grpc.health.v1.Health, bean: grpcHealthService, class: io.grpc.protobuf.service
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: grpc.reflection.v1alpha.ServerReflection, bean: protoReflectionService, class:
[server] [main] n.d.b.g.s.s.GrpcServerLifecycle : gRPC Server started, listening on address: *, port: 9090
[server] [main] com.example.server.ServerApplication : Started ServerApplication in 1.667 seconds (process running for 2.075)
```

# Configure the Server application

- Stop the Server and configure application.properties file
- In your **application.properties** or **application.yml** file, define the gRPC server properties
  - `# application.properties`
  - `grpc.server.port=9999`
  - `grpc.server.inProcessName=test`
- gRPC server will run on port 9999 and has an in-process name of 'test'.
- You can configure more based on your project requirements.



# Run the Configured Server

```
1 spring.application.name=server
2 grpc.server.port=9999
3 grpc.server.inProcessName=test
```

← application.properties file

```
Run  ServerApplication x
[server] [main] g.s.a.GrpcServerFactoryAutoConfiguration : 'grpc.server.in-process-name' is set: Creating InProcessGrpcServerFactory
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: Simple, bean: grpcServerService, class: com.example.server.GrpcServerService
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: grpc.health.v1.Health, bean: grpcHealthService, class: io.grpc.protobuf.services
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: grpc.reflection.v1alpha.ServerReflection, bean: protoReflectionService, class: i
[server] [main] n.d.b.g.s.s.GrpcServerLifecycle : gRPC Server started, listening on address: *, port: 9999
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: Simple, bean: grpcServerService, class: com.example.server.GrpcServerService
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: grpc.health.v1.Health, bean: grpcHealthService, class: io.grpc.protobuf.services
[server] [main] n.d.b.g.s.s.AbstractGrpcServerFactory : Registered gRPC service: grpc.reflection.v1alpha.ServerReflection, bean: protoReflectionService, class: i
[server] [main] n.d.b.g.s.s.GrpcServerLifecycle : gRPC Server started, listening on address: in-process:test, port: -1
[server] [main] com.example.server.ServerApplication : Started ServerApplication in 1.745 seconds (process running for 2.105)
```

# Test Server Using Postman

The screenshot shows the Postman web interface. At the top, there's a navigation bar with 'Home', 'Workspaces', and 'Explore'. A search bar labeled 'Search Postman' is on the right, along with 'Sign In' and 'Create Account' buttons. A blue banner below the navigation bar states: 'You are using the Lightweight API Client, sign in or create an account to work with collections, environments and unlock all free features in Postman.'

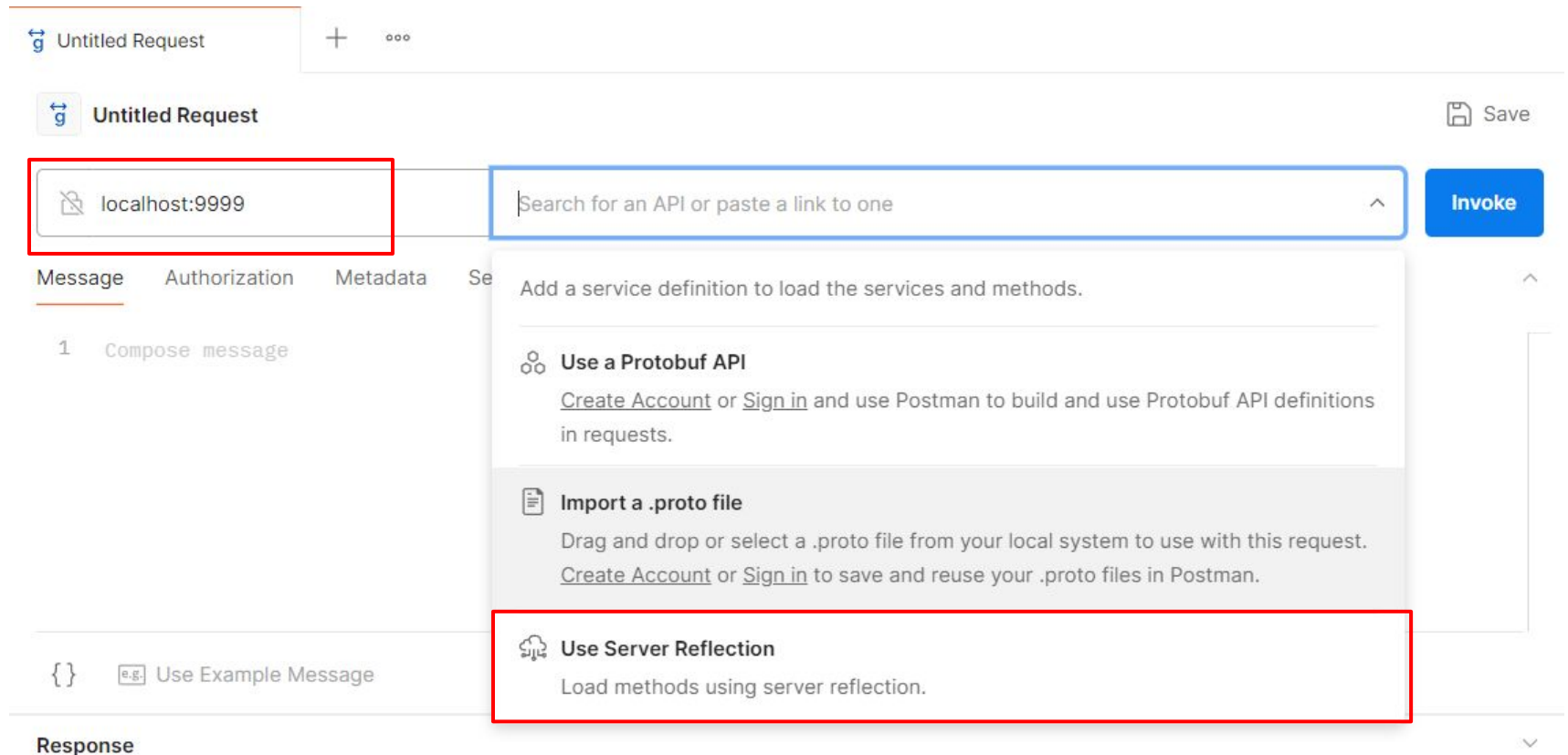
On the left, the 'History' panel is visible, showing a list of requests to 'localhost:9090' grouped by date (Today, June 12, June 11, June 10). Above this list are 'New' and 'Import' buttons. A red arrow points to the 'New' button with the text '1. Click New'.

A modal window is open in the center, displaying various API protocols: HTTP, WebSocket, Socket.IO, GraphQL, gRPC, MQTT, Collection, Environment, API, Flows, and Workspace. The 'gRPC' option is highlighted with a blue border. A red arrow points to this highlighted option with the text '2. Select gRPC'.

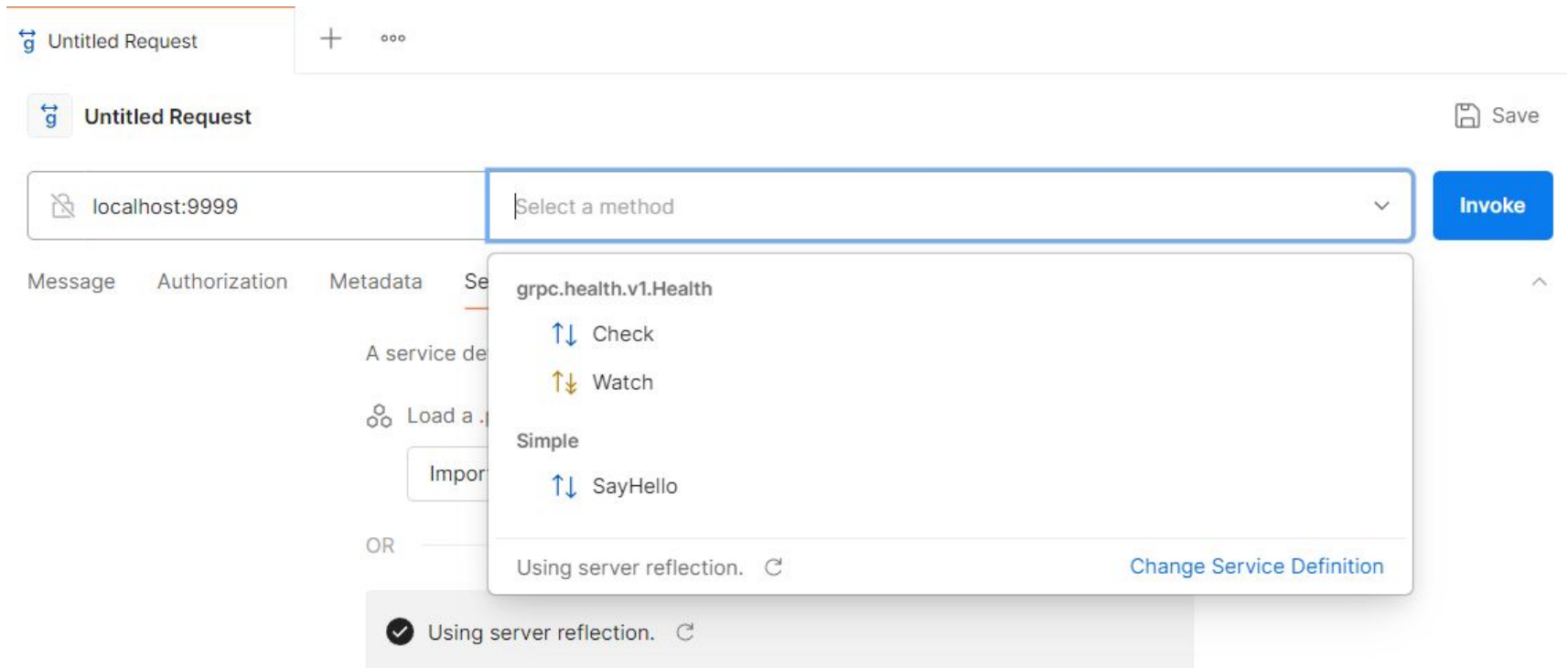
At the bottom of the modal, there is a text box that reads: 'gRPC is a highly performant RPC framework often used to build microservices. Test your gRPC APIs with a gRPC request.'



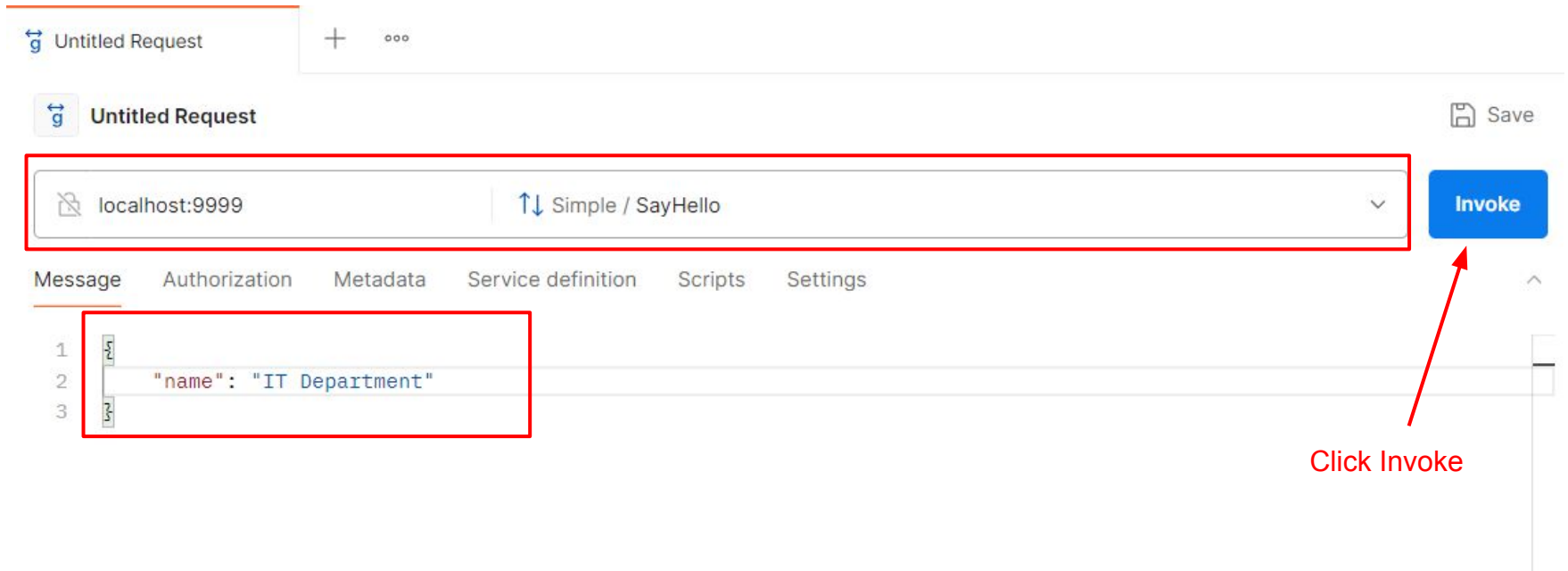
# Test Server Using Postman



# Test Server Using Postman

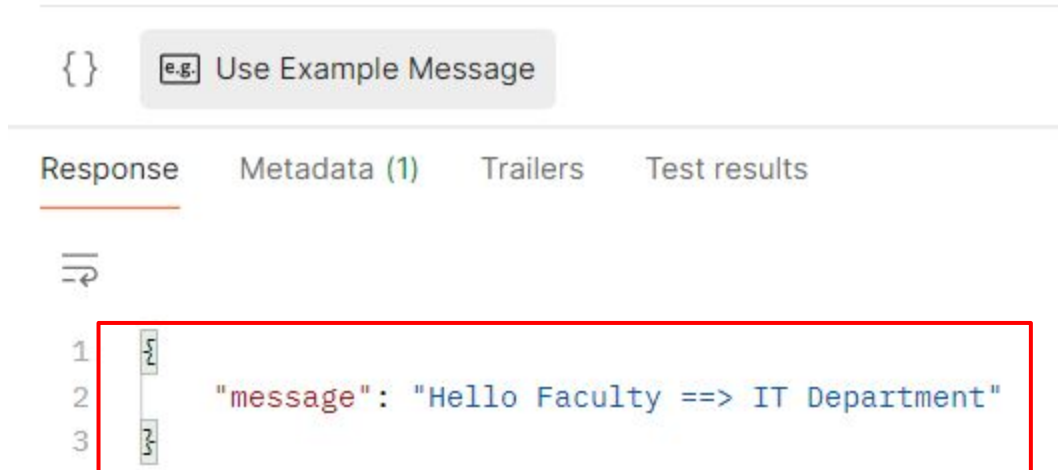


# Test Server Using Postman



# Test Server Using Postman

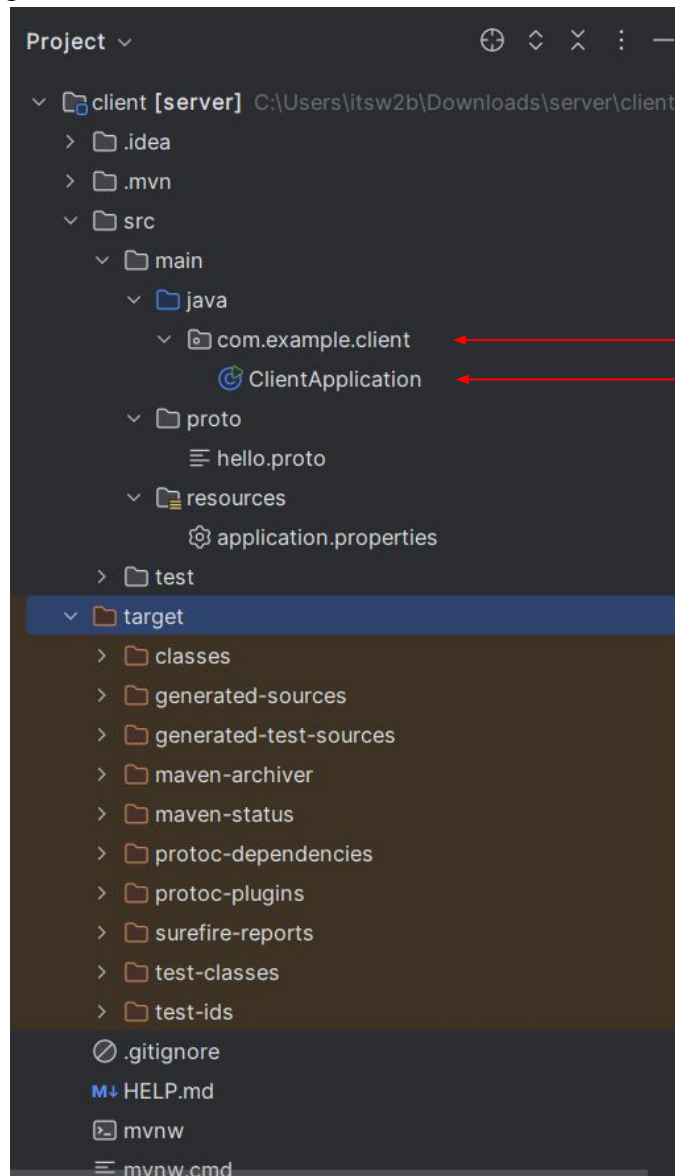
Observe the Response of gRPC invocation



# Test Server using gRPC Client

Copy the project files of server to client

# Directory Structure of Client Application



Change package name to client from server

Change package and application name to client from server

# Add the dependencies in pom.xml file

```
<dependency>
  <groupId>net.devh</groupId>
  <artifactId>grpc-client-spring-boot-starter</artifactId>
  <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>
<dependency>
```

# Plugin details in pom.xml file

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>add-source</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>add-source</goal>
            </goals>
            <configuration>
                <sources>
                    <source>${project.build.directory}/generated-sources/protobuf/java</source>
                </sources>
            </configuration>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.xolstice.maven.plugins</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <version>0.6.1</version>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>compile-custom</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.25.1:exe:${os.detected.classifier}</protocArtifact>
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.63.0:exe:${os.detected.classifier}</pluginArtifact>
    </configuration>
</plugin>
</plugins>
</build>
```



# gRPC - Client: application.properties file

```
spring.application.name=client  
grpc.client.helloService.address=static://localhost:9999  
grpc.client.helloService.negotiation-type=plaintext
```

# ClientApplication.java

```
package com.example.client;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.example.server.proto.HelloRequest;
import com.example.server.proto.SimpleGrpc;
import net.devh.boot.grpc.client.inject.GrpcClient;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class ClientApplication {

    public static void main(String[] args) { SpringApplication.run(ClientApplication.class, args); }

    @Bean no usages
    ApplicationRunner clientRunner(@GrpcClient("helloService") SimpleGrpc.SimpleBlockingStub simpleBlockingStub) {
        return new ApplicationRunner() {
            @Override
            public void run(ApplicationArguments args) throws Exception {
                System.out.println(simpleBlockingStub.sayHello(HelloRequest.newBuilder().setName("Vipul Dabhi").build()));
            }
        };
    }
}
```

# Stub Class Name

- The stub class names are derived from the service name used in your proto file.
- The blocking stub name is derived from
  - `package.name.serviceNameGrpc.serviceNameBlockingStub`.
  - For our example of Simple service, the stub name would be
    - `com.example.server.proto.SimpleGrpc.SimpleBlockingStub`

```
option java_package = "com.example.server.proto";  
  
service Simple {  
  rpc SayHello (HelloRequest) returns (HelloReply) {  
  }  
}
```

# Run Client Application

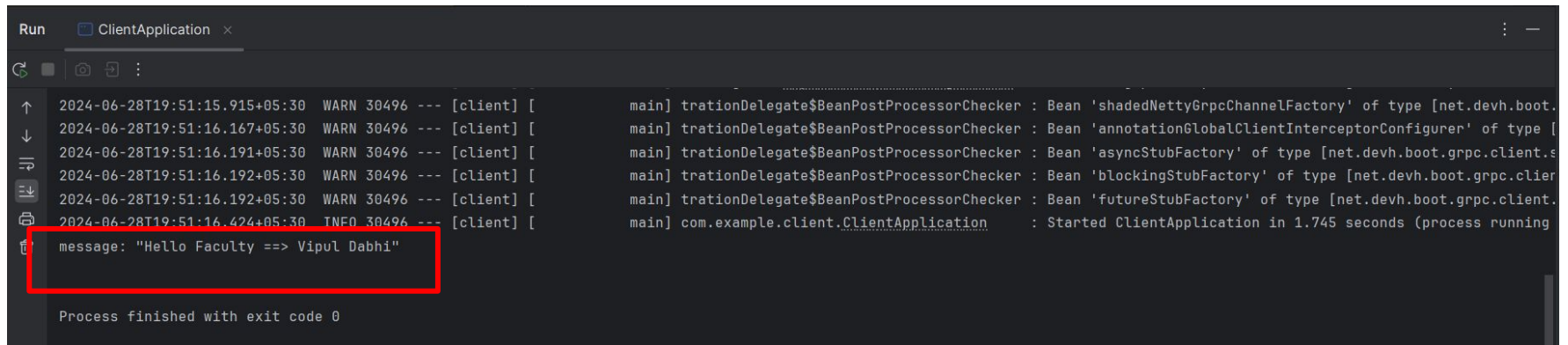
```
7  import com.example.server.proto.SimpleGrpc;
8  import net.devh.boot.grpc.client.inject.GrpcClient;
9  import org.springframework.boot.ApplicationArguments;
10 import org.springframework.boot.ApplicationRunner;
11 import org.springframework.context.annotation.Bean;
12
13
14 @SpringBootApplication
15 public class ClientApplication {
16
17     public static void main(String[] args) { SpringApplication.run(ClientApplication.class, args); }
18
19
20
21
22 @Bean no usages
23 ApplicationRunner clientRunner(@GrpcClient("helloService") SimpleGrpc.SimpleBlockingStub simpleBlockingStub){
24     return new ApplicationRunner(){
25
26         @Override
27         public void run(ApplicationArguments args) throws Exception {
28             System.out.println(simpleBlockingStub.sayHello(HelloRequest.newBuilder().setName("Vipul Dabhi").build()));
29         }
30     };
31 }
32 }
```

Run Client  
Application

Stub

# Run Client Application

Run Client Application and Observe the output



```
Run ClientApplication x
2024-06-28T19:51:15.915+05:30 WARN 30496 --- [client] [main] trationDelegate$BeanPostProcessorChecker : Bean 'shadedNettyGrpcChannelFactory' of type [net.devh.boot.
2024-06-28T19:51:16.167+05:30 WARN 30496 --- [client] [main] trationDelegate$BeanPostProcessorChecker : Bean 'annotationGlobalClientInterceptorConfigurer' of type [
2024-06-28T19:51:16.191+05:30 WARN 30496 --- [client] [main] trationDelegate$BeanPostProcessorChecker : Bean 'asyncStubFactory' of type [net.devh.boot.grpc.client.s
2024-06-28T19:51:16.192+05:30 WARN 30496 --- [client] [main] trationDelegate$BeanPostProcessorChecker : Bean 'blockingStubFactory' of type [net.devh.boot.grpc.clie
2024-06-28T19:51:16.192+05:30 WARN 30496 --- [client] [main] trationDelegate$BeanPostProcessorChecker : Bean 'futureStubFactory' of type [net.devh.boot.grpc.client.
2024-06-28T19:51:16.424+05:30 INFO 30496 --- [client] [main] com.example.client.ClientApplication : Started ClientApplication in 1.745 seconds (process running
message: "Hello Faculty ==> Vipul Dabhi"
Process finished with exit code 0
```