

# gRPC

---

## 1. Introduction to GRPC

---

GRPC is a Remote Procedure Call framework that can run in any environment. It connects services in and across data centers.

### 1.1 The main usage scenarios

---

- Efficiently connecting polyglot services in microservices style architecture
- Connecting mobile devices, browser clients to backend services
- Generating efficient client libraries

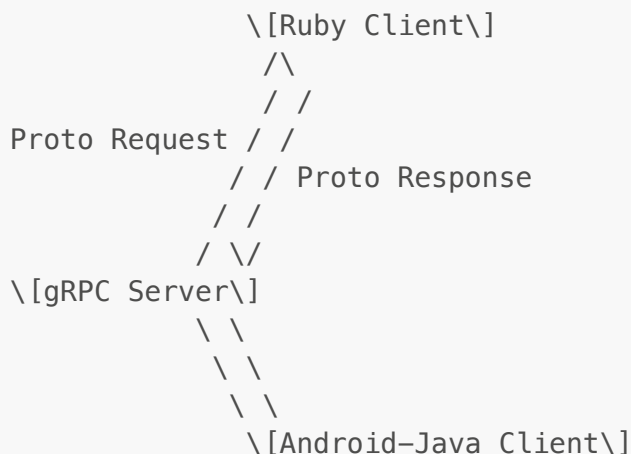
### 1.2 Core features

---

- Idiomatic client libraries in 11 languages
- Highly efficient on wire and with a simple service definition framework
- Bi-directional streaming with http/2 based transport
- Pluggable auth, tracing, load balancing and health checking

## 2. Architecture

---



In gRPC a client can call a Method on a server application as if it were a local Object. It makes it easier to create distributed applications and services.

We have to specify a Interface for the client that can be called remotely. On the Server side we have to implement that interface.

## 2.1 Protocol Buffers

---

By default, gRPC uses [Protocol Buffers](#). Heres a quick intro how that works.

With protocol buffers we define the data structures we serialize. This is a normal text file with the *.proto* extension.

```
message Person {  
  string name = 1;  
  int32 id = 2;  
  bool has_ponycopter = 3;  
}
```

Once created, we can use the protocol buffer compiler *protoc* to compile these files into access classes for your language.

You can define gRPC services in *.proto* files.

```
syntax = "proto3";  
  
package hello;  
  
service Greeter {  
  // Sends a greeting  
  rpc SayHello (HelloRequest) returns (HelloReply) {}  
}  
  
// The request message containing the user's name.  
message HelloRequest {  
  string name = 1;  
}  
  
// The response message containing the greetings  
message HelloReply {  
  string message = 1;  
}
```

## 3. Implementation

---

### 3.1 Java

### 3.1.1 Service Implementation

```
public class HelloWorldServiceImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(Hello.HelloRequest request,
        StreamObserver<Hello.HelloReply> responseObserver) {
        Hello.HelloReply reply =
        Hello.HelloReply.newBuilder().setMessage("Hello " +
        request.getName()).build();

        // Send the reply back to the client.
        responseObserver.onNext(reply);

        // Indicate that no further messages will be sent to the client.
        responseObserver.onCompleted();
    }
}
```

Here the Greeter Service gets Implemented. The HelloWorldServiceImpl class extends the GreeterGrpc.GreeterImplBase class to override the sayHello Method. I can determine what the sayHello() Method does and define the intended behaviour.

### 3.1.2 Server

```
public class HelloWorldServer {
    private static final int PORT = 50051;
    private Server server;
    public void start() throws IOException {
        server = ServerBuilder.forPort(PORT)
            .addService(new HelloWorldServiceImpl())
            .build()
            .start();
    }
}
```

The Server gets built by the ServerBuilder Class, I can define the Port with the forPort() function and add the Service with the addService() function. After I have defined what port I want to use and what service to use I can build and start the Server;

### 3.1.3 Client

```
public class HelloWorldClient {

    public static void main(String[] args) {

        ManagedChannel channel =
        ManagedChannelBuilder.forAddress("localhost", 50051)
```

```
        .usePlaintext()
        .build();

    GreeterGrpc.GreeterBlockingStub stub =
GreeterGrpc.newBlockingStub(channel);

    Hello.HelloReply helloResponse =
stub.sayHello(Hello.HelloRequest.newBuilder()
        .setName("Max")
        .build());
    System.out.println( helloResponse.getMessage() );
    channel.shutdown();

}

}
```

The Client opens a connection to the Service with a ManagedChannel and ManagedChannelBuilder. Then I create a blocking stub with the newBlockingStub() Method. On the stub I can call the sayHello() Method.