



Luca Cabibbo  
Architettura  
dei Sistemi  
Software

# Invocazione remota: gRPC

**dispensa asw830**  
ottobre 2023

*These are my principles.  
If you don't like them,  
I have others.*  
*Groucho Marx*

1

Invocazione remota: gRPC

Luca Cabibbo ASW



## - Riferimenti

- Luca Cabibbo. **Architettura del Software: Strutture e Qualità**. Edizioni Efestò, 2021.
  - Capitolo 23, **Invocazione remota**
- gRPC
  - <https://grpc.io/>
- Protocol Buffers
  - <https://developers.google.com/protocol-buffers/docs/overview>
- Fowler, M. **Presentation Model**. 2004.
  - <https://martinfowler.com/eaaDev/PresentationModel.html>

2

Invocazione remota: gRPC

Luca Cabibbo ASW



## - Obiettivi e argomenti

### □ Obiettivi

- presentare gRPC – un framework per l'invocazione remota
- introdurre il pattern Presentation Model

### □ Argomenti

- introduzione a gRPC
- esempi
- discussione



## \* Introduzione a gRPC

- **gRPC** è un framework per l'invocazione remota (RPC) moderno, ad alte prestazioni e interoperabile
  - inizialmente sviluppato da Google, ora è open source
  - utilizza i Protocol Buffers come formato di interscambio e come IDL (Interface Definition Language) – ed inoltre HTTP/2 come protocollo di trasporto
  - consente di generare proxy (sia lato server che lato client) per una decina di linguaggi di programmazione – tra cui C#, Go, Java, Node.js e Python
  - fornisce caratteristiche come l'autenticazione, lo streaming bidirezionale e il controllo del flusso, invocazioni bloccanti e non bloccanti, timeout e cancellazioni
  - gli scenari di utilizzo includono l'invocazione remota nei sistemi distribuiti (ad es., tra microservizi) e la comunicazione tra dispositivi mobili e client web con i servizi di backend



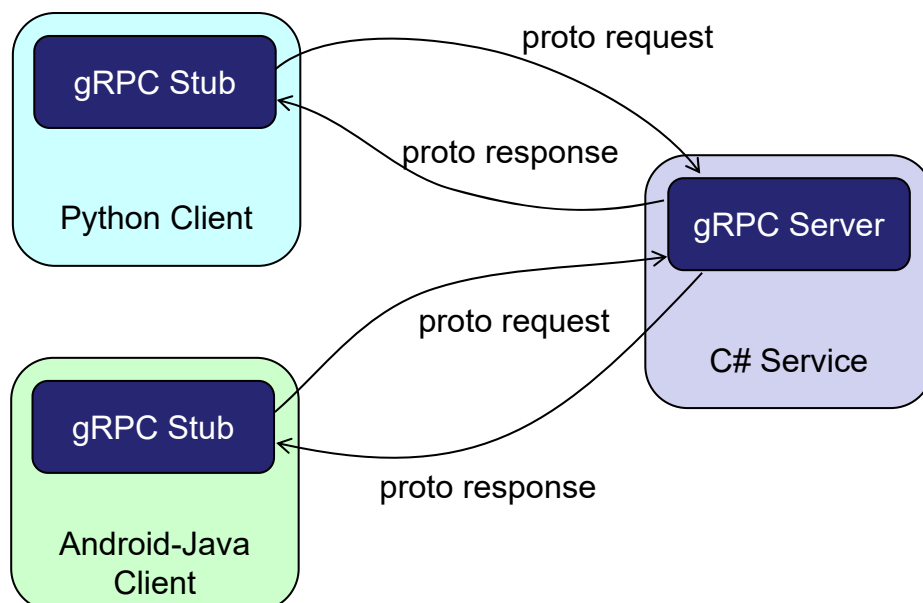
## - gRPC

- In pratica, gRPC consente a un'applicazione o servizio client di chiamare un metodo di un'applicazione o servizio server, in esecuzione su un computer remoto, come se fosse una chiamata locale – per semplificare la realizzazione di sistemi distribuiti
  - per definire un servizio distribuito, bisogna prima specificare la sua interfaccia – ovvero le operazioni che possono essere invocate remotamente, con i loro parametri e i loro tipi di ritorno
  - a partire da questa interfaccia, gRPC genera i proxy (lato server e lato client) per l'invocazione remota
  - nell'architettura esagonale, è possibile poi utilizzare questi proxy per realizzare gli adattatori – inbound (lato server) e outbound (lato client) – per completare i connettori tra i servizi applicativi server e client



## gRPC

- I client e i server gRPC possono essere definiti in modo flessibile
  - possono essere realizzati in linguaggi di programmazione differenti, e possono comunicare in una varietà di ambienti





## - Introduzione a Protocol Buffers

- Il framework gRPC adotta (per default) Protocol Buffers come formato per la serializzazione e l'interscambio dei dati tra servizi remoti
  - è però possibile usare anche altri linguaggi e formati dei dati – ad es., JSON
- *Protocol Buffers* è un linguaggio per la serializzazione di dati strutturati, da usare (ad es.) nei protocolli di comunicazione
  - è neutrale rispetto ai linguaggi di programmazione e alle piattaforme, ed è estensibile
  - si pensi a XML – ma più semplice e più veloce



## Protocol Buffers

- In pratica, con *Protocol Buffers* la struttura dei dati da serializzare va specificata in un file *proto* – un file di testo con estensione *.proto*
    - i dati sono strutturati in messaggi – ogni *messaggio* è sostanzialmente un record, composto da una sequenza di campi
- ```
message Person {  
    int32 id = 1;  
    string firstName = 2;  
    string lastName = 3;  
}
```
- dopo aver specificato le strutture di dati di interesse, il compilatore *protoc* consente di generare il codice per gestire tali strutture di dati nel linguaggio di programmazione preferito



# Protocol Buffers

```
message Person {
  int32 id = 1;
  string firstName = 2;
  string lastName = 3;
}
```

- Con riferimento al codice generato da **protoc** (in questo esempio, per Java), ecco come creare una struttura di dati che rappresenta una persona – si usa il design pattern **Builder** [GoF]

```
Person person =
  Person.newBuilder()
    .setId(42)
    .setFirstName("Mario")
    .setLastName("Rossi")
    .build();
```

- ed ecco come accedere a questa struttura di dati

```
... person.getId() ...
... person.getFirstName() ...
... person.getLastName() ...
```



## - Definizione di un servizio gRPC

- gRPC adotta Protocol Buffers anche come IDL (Interface Definition Language) per specificare l'interfaccia dei servizi remoti
  - l'interfaccia di un servizio distribuito può essere specificata in un file **proto** – bisogna definire le operazioni del servizio, nonché i loro messaggi di richiesta e quelli di risposta

```
/* Il servizio HelloService. */
service HelloService {
  /* il servizio definisce una sola operazione sayHello */
  rpc sayHello(HelloRequest) returns (HelloReply) {}
}

/* Il messaggio di richiesta di sayHello contiene il nome. */
message HelloRequest {
  string name = 1;
}

/* Il messaggio di risposta di sayHello contiene il saluto. */
message HelloReply {
  string greeting = 1;
}
```



## \* Esempi

- Vengono ora mostrati alcuni esempi di utilizzo di gRPC
  - un semplice servizio per saluti
  - il servizio **restaurant-service** per la gestione di un insieme di ristoranti – nell'ambito di un'applicazione **efood** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale



## - Il servizio Hello

- Si consideri un semplice servizio per generare dei saluti, la cui logica di business è definita come segue

```
package asw.hello.domain;

import org.springframework.stereotype.Service;

@Service
public class HelloService {

    public String sayHello(String name) {
        return "Hello, " + name + "!";
    }

}
```

- vogliamo esporre questo servizio come un servizio remoto gRPC



## - Definizione dell'interfaccia del servizio

- Per prima cosa, bisogna specificare l'interfaccia del servizio mediante un file *proto* – il file **HelloService.proto**

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "asw.hello.grpc.proto";

/* Il servizio HelloService. */
service HelloService {
    /* il servizio definisce una sola operazione sayHello */
    rpc sayHello(HelloRequest) returns (HelloReply) {}
}

/* Il messaggio di richiesta di sayHello contiene il nome. */
message HelloRequest {
    string name = 1;
}

/* Il messaggio di risposta di sayHello contiene il saluto. */
message HelloReply {
    string greeting = 1;
}
```

13

Invocazione remota: gRPC

Luca Cabibbo ASW



## Definizione dell'interfaccia del servizio

- A partire dal file **HelloService.proto**, il compilatore di interfacce gRPC per Java genera il package **asw.hello.grpc.proto** che contiene
  - una classe **HelloServiceGrpc** – che rappresenta il servizio e i suoi proxy (sia lato server che lato client) – con le seguenti classi interne
    - una classe **HelloServiceGrpc>HelloServiceImplBase** – lo skeleton (proxy lato server) per il servizio
    - le classi **HelloServiceGrpc>HelloServiceBlockingStub** e **HelloServiceGrpc>HelloServiceFutureStub** – due stub (proxy lato client) per il servizio, da usare con modalità diverse
  - le classi **HelloRequest** e **HelloReply** – che rappresentano i messaggi (di richiesta e risposta) per l'invocazione del servizio
  - questo package può essere utilmente condiviso tra il client e il server – ciascuno utilizzerà solo le classi di interesse

14

Invocazione remota: gRPC

Luca Cabibbo ASW



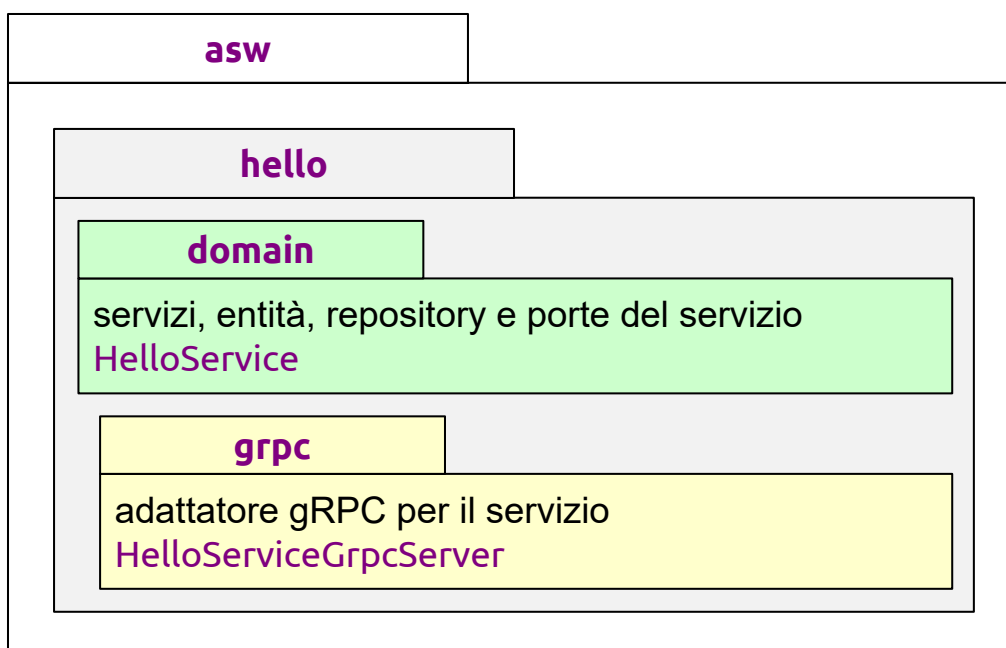
## - Server Grpc

- Lato server, bisogna definire un server gRPC per il servizio
  - lo realizziamo come un'applicazione Spring Boot
  - applicando l'architettura esagonale, dobbiamo definire un inbound adapter per questo servizio
    - a tal fine, definiamo il package `asw.hello.grpc` con una classe `HelloServiceGrpcServer` che implementa il server gRPC (ovvero, l'adattatore inbound gRPC) per il servizio Hello



## Architettura esagonale

- Architettura esagonale del servizio **hello**







## Server Grpc

- Lato server, la classe `HelloServiceGrpcServer` implementa il server gRPC

```
package asw.hello.grpc;

import asw.hello.domain.HelloService;

import asw.hello.grpc.proto.*;
import io.grpc.*;
import io.grpc.stub.*;
import jakarta.annotation.*;
import java.io.IOException;

import org.springframework.beans.factory.annotation.Value;

@Component
public class HelloServiceGrpcServer {

    ... vedi dopo ...

}
```

in **rosso** indichiamo  
il codice che dipende  
da gRPC

17

Invocazione remota: gRPC

Luca Cabibbo ASW



## Server Grpc

- Nella classe `HelloServiceGrpcServer` sono necessarie alcune definizioni preliminari standard

```
@Autowired
private HelloService helloService;

@Value("${asw.helloservice.grpc.port}")
private int port;

private Server server;

@PostConstruct
public void start() throws IOException {
    this.server = ServerBuilder.forPort(port)
        .addService( new HelloServiceImpl() )
        .build().start();
}

@PreDestroy
public void stop() {
    if (server != null) { server.shutdown(); }
}
```

# application.properties  
asw.helloservice.grpc.port=50051

classe che  
estende lo  
skeleton, da  
definire

18

Invocazione remota: gRPC

Luca Cabibbo ASW



## Parentesi: @PostConstruct e @PreDestroy

- Nel framework Spring (ma anche in altri framework a componenti e basati su contenitori) è comune l'uso di alcune annotazioni
  - l'inizializzazione dello stato di un componente (o bean) va effettuata in un metodo annotato **@PostConstruct**
    - i metodi annotati in questo modo vengono eseguiti dopo la costruzione (da parte del contenitore) di un'istanza del componente – dopo l'iniezione delle dipendenze, ma prima che sia possibile invocare i metodi del componente
    - a tal fine, non è invece opportuno definire dei costruttori
  - l'eventuale deallocazione delle risorse del componente (o bean) va effettuata in un metodo annotato **@PreDestroy**
    - i metodi annotati in questo modo vengono eseguiti subito prima della distruzione (da parte del contenitore) di un'istanza del componente



## Server Grpc

- Il collegamento tra il server gRPC e il servizio **helloService** viene realizzato mediante la classe interna **HelloServiceImpl** – che deve estendere lo skeleton **HelloServiceGrpc>HelloServiceImplBase**

```
private class HelloServiceImpl
    extends HelloServiceGrpc>HelloServiceImplBase {

    @Override
    public void sayHello(HelloRequest request,
        StreamObserver<HelloReply> responseObserver) {

        String name = request.getName();
        String greeting = helloService.sayHello(name);
        HelloReply reply = HelloReply.newBuilder()
            .setGreeting(greeting)
            .build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();

    }
}
```



## Server Grpc

- Il collegamento tra il server gRPC e il servizio `helloService` viene realizzato mediante la classe interna `HelloServiceImpl` – che deve estendere lo skeleton `HelloServiceGrpc>HelloServiceImplBase`
  - questa classe è il cuore dell'adattatore (lato server) – il ruolo di un adattatore inbound è quello di interpretare richieste del client (in questo caso, richieste gRPC), trasformarle in richieste all'oggetto adattato (in questo caso, il servizio `helloService`), ottenere risposte dall'adattato, trasformarle in risposte al client
  - si noti pertanto l'adattamento svolto dal metodo `sayHello` per gestire una chiamata tramite gRPC
    - estrae i parametri della chiamata dall'oggetto richiesta `request`
    - invoca l'operazione richiesta del servizio `helloService` e ottiene il risultato
    - a partire dal risultato, crea l'oggetto risposta `reply`
    - richiede la trasmissione della risposta `reply` al client

21

Invocazione remota: gRPC

Luca Cabibbo ASW



## - Client Grpc

- Il lato client è evidentemente relativo a un'altra applicazione o servizio che vuole accedere, in questo esempio, al servizio Hello
  - supponiamo che sia un'altra applicazione Spring Boot, il cui package di base è `asw.samplehelloclient`
  - nel suo dominio va definita un'interfaccia richiesta (una porta) `HelloClientPort` per accedere al servizio Hello

```
package asw.samplehelloclient.domain;

public interface HelloClientPort {
    public String sayHello(String name);
}
```

  - applicando l'architettura esagonale, va definito un outbound adapter per accedere al servizio Hello, che implementa questa porta/interfaccia
    - definiamo il package `asw.samplehelloclient.helloclient.grpc` con la classe `HelloClientGrpcAdapter`

22

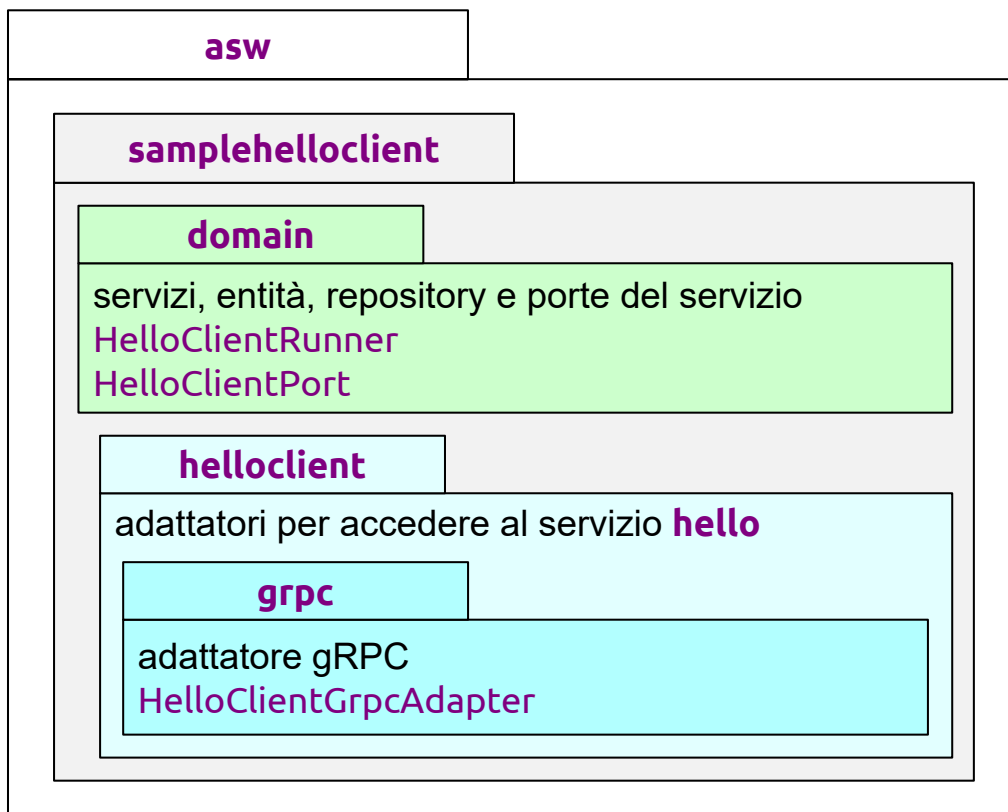
Invocazione remota: gRPC

Luca Cabibbo ASW



# Architettura esagonale

## □ Architettura esagonale del servizio **sample-hello-client**



23

Invocazione remota: gRPC

Luca Cabibbo ASW



# Client Grpc

- Che cosa faccia il client finale di questo servizio è, in effetti, poco rilevante per la nostra discussione
  - tuttavia, ecco una porzione di esempio del nostro client – si noti la dipendenza dalla sola porta **HelloClientPort** – e soprattutto che il client non dipende in alcun modo da gRPC

```
package asw.samplehelloclient.domain;

import ...;

@Component
public class HelloClientRunner implements CommandLineRunner {

    @Autowired
    private HelloClientPort helloClientAdapter;

    public void run(String[] args) {

        ... helloClientAdapter.sayHello("Luca") ...
        ... helloClientAdapter.sayHello("World") ...

    }
}
```

24

Invocazione remota: gRPC

Luca Cabibbo ASW



## Client Grpc

- Lato client, la classe **HelloClientGrpcAdapter** implementa l'adattatore gRPC per l'accesso al servizio Hello

```
package asw.samplehelloclient.helloclient.grpc;

import asw.samplehelloclient.domain>HelloClientPort;
import asw.hello.grpc.proto.*;

import io.grpc.*;

import java.util.concurrent.*;
import com.google.common.util.concurrent.ListenableFuture;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Value;
import jakarta.annotation.*;

@Service
public class HelloClientGrpcAdapter implements HelloClientPort {
    ... vedi dopo ...
}
```

25

Invocazione remota: gRPC

Luca Cabibbo ASW



## Client Grpc

- Anche nella classe **HelloClientGrpcAdapter** sono necessarie alcune definizioni preliminari standard

```
@Value("${asw.helloservice.grpc.host}")
private String host;
@Value("${asw.helloservice.grpc.port}")
private int port;
```

```
private ManagedChannel channel;
private>HelloServiceGrpc>HelloServiceBlockingStub blockingStub;
private>HelloServiceGrpc>HelloServiceFutureStub futureStub;
```

# application.properties  
asw.helloservice.grpc.host=localhost  
asw.helloservice.grpc.port=50051

- intuitivamente

- il **channel** rappresenta una connessione gRPC con uno specifico server (localizzato mediante host e porta)
- i due **stub** (**blocking** e **future**) sono dei proxy per l'invocazione remota, in modalità sincrona e asincrona, rispettivamente – in effetti, è sufficiente usare solo lo stub di interesse

26

Invocazione remota: gRPC

Luca Cabibbo ASW



## Client Grpc

- Anche nella classe `HelloClientGrpcAdapter` sono necessarie alcune definizioni preliminari standard
  - metodi di supporto all'inizializzazione e alla deallocazione della connessione gRPC

```
@PostConstruct
public void init() {
    this.channel = ManagedChannelBuilder.forAddress(host, port)
        .usePlaintext()
        .build();
    this.blockingStub = HelloServiceGrpc.newBlockingStub(channel);
    this.futureStub = HelloServiceGrpc.newFutureStub(channel);
}

@PreDestroy
public void shutdown() throws InterruptedException {
    channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
}
```



## Client Grpc

- Ecco il metodo `sayHello` della classe `HelloClientGrpcAdapter`, che implementa l'accesso all'operazione remota

```
public String sayHello(String name) {
    String greeting = null;
    HelloRequest request = HelloRequest.newBuilder()
        .setName(name).build();
    try {
        HelloReply reply =
            blockingStub.sayHello(request); // bloccante
        greeting = reply.getGreeting();
    } catch (StatusRuntimeException e) {
        ... gRPC failed ...
    }
    return greeting;
}
```

- in questo caso è stato utilizzato il `blockingStub`



## Client Grpc

- In questo caso, l'adapter (lato client) realizza un adattamento tra il client – che è la logica di business del servizio (che vuole fare chiamate Java) – e l'adattato – che è il server remoto gRPC (che vuole ricevere chiamate gRPC)
  - il ruolo di un adattatore outbound è quello di interpretare richieste della logica di business tramite una porta inbound (in questo caso, richieste Java), trasformarle in richieste all'entità esterna adattata (in questo caso, richieste gRPC al servizio **hello**), ottenere risposte dall'entità esterna adattata, trasformarle in risposte alla logica di business



## Client Grpc

- In questo caso, l'adapter (lato client) realizza un adattamento tra il client – che è la logica di business del servizio (che vuole fare chiamate Java) – e l'adattato – che è il server remoto gRPC (che vuole ricevere chiamate gRPC)
  - si noti pertanto l'adattamento svolto dal metodo **sayHello** per effettuare una chiamata tramite gRPC
    - crea l'oggetto richiesta **request** che codifica i parametri dell'invocazione remota
    - invoca l'operazione remota tramite lo stub – in questo caso il **blockingStub** – e ottiene la risposta **reply** dal server
    - estrae il risultato dalla risposta **reply** e lo restituisce al suo chiamante



## Client Grpc

### □ Alcune osservazioni

- il server gRPC non riceverà questo oggetto richiesta **request** – infatti potrebbe essere realizzato con un linguaggio di programmazione differente! – piuttosto, riceverà un oggetto che è un clone di questa richiesta **request**
- in modo analogo, il client gRPC non riceverà l'oggetto risposta **reply** che viene creato dal server – piuttosto, riceverà un oggetto che è un clone di una tale risposta **reply**
- tra il servizio client e il servizio server (che, in questo caso, vogliono uno fare richieste Java e l'altro ricevere richieste Java) ci sono in mezzo due adattatori – uno lato client (da Java a gRPC) e uno lato server (da gRPC a Java)
- i messaggi scambiati in rete per l'invocazione del servizio sono conformi alla specifica di Protocol Buffers



## Client Grpc

### □ Una variante del metodo **sayHello**, che implementa l'accesso all'operazione remota mediante il **futureStub**

```
public String sayHello(String name) {  
    String greeting = null;  
    HelloRequest request = HelloRequest.newBuilder()  
        .setName(name).build();  
    try {  
        ListenableFuture<HelloReply> futureReply =  
            futureStub.sayHello(request);    // non bloccante  
        ... qui è possibile eseguire altre azioni ...  
        HelloReply reply = futureReply.get();    // bloccante  
        greeting = reply.getGreeting();  
    } catch (StatusRuntimeException e) {  
        ... gRPC failed ...  
    } catch (InterruptedException | ExecutionException e) {  
        ... other exceptions ...  
    }  
    return greeting;  
}
```





## - Discussione

- Ecco alcune considerazioni su gRPC
  - consente di esporre un servizio e di invocarlo remotamente
  - l'interfaccia del servizio viene definita (in modo neutrale rispetto ai linguaggi di programmazione supportati) usando un file *proto*
  - dal file *proto* vengono generati i proxy lato server e lato client per il servizio
  - è necessario scrivere del codice aggiuntivo sia lato server che lato client, per collegare la logica di business con i proxy
    - nell'architettura esagonale, vanno realizzati gli adapter inbound e outbound per il servizio, in due "esagoni" diversi



## Discussione

- Ulteriori considerazioni su gRPC
  - nel file *proto*, è possibile specificare messaggi con campi ripetuti (come sarà discusso più avanti)
  - le operazioni del servizio possono effettuare lo streaming lato client (il client invia uno stream di richieste), lo streaming lato server (il server invia uno stream di risposte) e lo streaming bidirezionale
    - qui consideriamo solo l'RPC "unario" – senza streaming
  - le operazioni possono essere invocate in modo bloccante oppure non bloccante
  - è possibile definire dei timeout e richiedere la cancellazione delle invocazioni (sia da parte del client che da parte del server)
  - è possibile l'autenticazione
  - un'invocazione remota può terminare con un'eccezione **StatusRuntimeException** (discusso più avanti)



## Discussione – errori

- ❑ gRPC può generare degli errori (eccezioni remote) in varie circostanze – ad es., un fallimento della rete o connessioni non autenticate – a cui sono associati diversi codici di stato
  - **errori generali**

| Case                                                                                                 | Status code                   |
|------------------------------------------------------------------------------------------------------|-------------------------------|
| Client application cancelled the request                                                             | GRPC_STATUS_CANCELLED         |
| Deadline expired before server returned status                                                       | GRPC_STATUS_DEADLINE_EXCEEDED |
| Method not found on server                                                                           | GRPC_STATUS_UNIMPLEMENTED     |
| Server shutting down                                                                                 | GRPC_STATUS_UNAVAILABLE       |
| Server threw an exception (or did something other than returning a status code to terminate the RPC) | GRPC_STATUS_UNKNOWN           |



## Discussione – errori

- **errori di rete**

| Case                                                                                                                                                             | Status code                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| No data transmitted before deadline expires. Also applies to cases where some data is transmitted and no other failures are detected before the deadline expires | GRPC_STATUS_DEADLINE_EXCEEDED |
| Some data transmitted (for example, the request metadata has been written to the TCP connection) before the connection breaks                                    | GRPC_STATUS_UNAVAILABLE       |



## Discussione – errori

- errori di protocollo

| Case                                                             | Status code                    |
|------------------------------------------------------------------|--------------------------------|
| Could not decompress but compression algorithm supported         | GRPC_STATUS_INTERNAL           |
| Compression mechanism used by client not supported by the server | GRPC_STATUS_UNIMPLEMENTED      |
| Flow-control resource limits reached                             | GRPC_STATUS_RESOURCE_EXHAUSTED |
| Flow-control protocol violation                                  | GRPC_STATUS_INTERNAL           |
| Error parsing returned status                                    | GRPC_STATUS_UNKNOWN            |
| Unauthenticated: credentials failed to get metadata              | GRPC_STATUS_UNAUTHENTICATED    |
| Invalid host set in authority metadata                           | GRPC_STATUS_UNAUTHENTICATED    |
| Error parsing response protocol buffer                           | GRPC_STATUS_INTERNAL           |
| Error parsing request protocol buffer                            | GRPC_STATUS_INTERNAL           |



## Discussione – semantica

- Semantica dell'invocazione remota di gRPC
  - *attenzione, la fonte sono alcune discussioni su Stack Overflow, ma non ho trovato documentazione ufficiale in proposito*
  - la semantica di default di gRPC sembra essere *maybe* – anche se qualcuno parla di *at-most once*
  - inoltre, è possibile configurare un client gRPC per la ritrasmissione automatica dei messaggi in caso di errori – in questo caso la semantica sembra essere *at-least once*



## Discussione – concorrenza

- ❑ Con gRPC, il server esegue le operazioni remote in modo concorrente
  - il server gRPC vive in un processo distinto da quello dei client gRPC – che possono essere molti, e accedere al server in modo concorrente
  - ciascuna diversa invocazione remota viene eseguita nell’ambito di un thread (lato server) differente e separato
    - dunque, il server gRPC potrebbe essere occupato nell’esecuzione concorrente di più operazioni remote
  - attenzione dunque a possibili interferenze nell’esecuzione di operazioni concorrenti
    - ad es., potrebbe essere utile (ma talvolta è invece dannoso) dichiarare **synchronized** i metodi remoti – i metodi “sincronizzati” di uno stesso oggetto vengono sempre eseguiti in modo mutuamente esclusivo da thread separati



## Discussione – presentation model

- ❑ Con gRPC e Protocol Buffers, la comunicazione tra client e server avviene mediante delle strutture di dati che sono nettamente distinte da quelle usate nel dominio delle applicazioni client e server
  - in questo semplice esempio
    - le strutture di dati usate nel dominio sia del client che del server sono delle semplici stringhe (per il nome e per il saluto)
    - le strutture di dati utilizzate nella comunicazione corrispondono ai messaggi del file *proto* – che rappresentano richieste, risposte e altri messaggi
  - questa è un’applicazione del pattern **Presentation Model** (discusso di seguito)



## - Presentation model

### □ Pattern *Presentation Model*

- all'interno di un servizio applicativo, i dati sono organizzati sulla base di un “modello di dominio”
- tuttavia, nell'interazione con altri servizi applicativi o entità esterne, questo pattern suggerisce di organizzare i dati sulla base di un “modello di presentazione”
  - un “modello di presentazione” è basato su una “rappresentazione” degli oggetti di dominio specifica per l'interazione con altre entità – che è in genere differente dalla rappresentazione del modello di dominio
  - questo è utile, ad es., se gli oggetti di dominio contengono dati privati che non è opportuno trasmettere in rete, oppure per consentire di variare le strutture di dati usate nel dominio di un servizio senza dover cambiare l'interfaccia del servizio



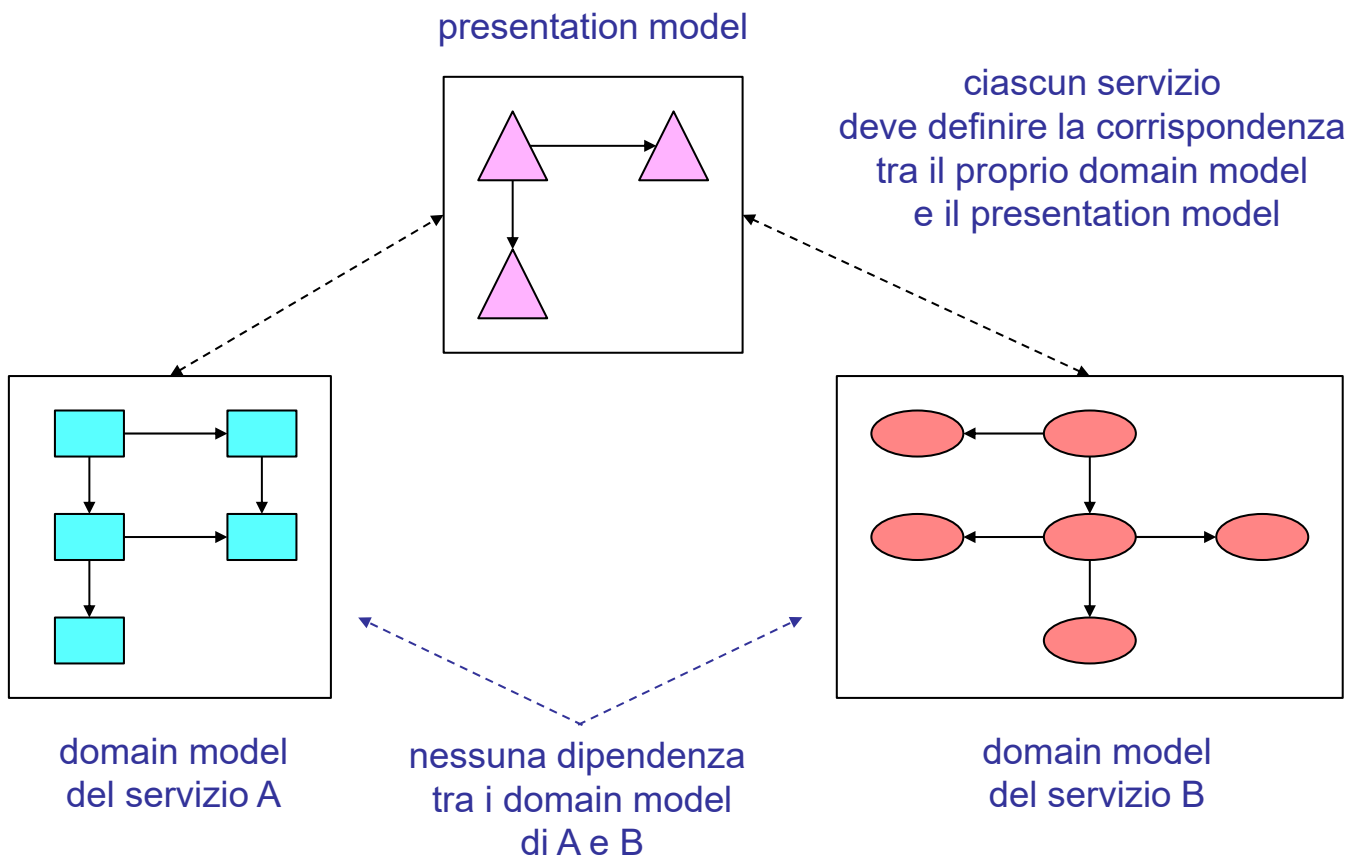
## Presentation model

### □ Pattern *Presentation Model*

- l'interazione con altri servizi applicativi o entità esterne può essere basata sull'utilizzo di adapter
  - ciascun adapter deve in genere effettuare una trasformazione tra il modello di dominio del proprio servizio e il modello di presentazione usato nella comunicazione con un altro servizio
  - nell'interazione tra due servizi, ciascun servizio deve conoscere solo il proprio modello di dominio (“privato”) e il presentation model (condiviso con l'altro servizio) – pertanto ognuno dei servizi è completamente disaccoppiato dal modello di dominio dell'altro servizio



# Presentation model



43

Invocazione remota: gRPC

Luca Cabibbo ASW



## - Il servizio restaurant-service

- Consideriamo ora il servizio **restaurant-service** per la gestione di un insieme di ristoranti – nell’ambito di un’applicazione **efood** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale – già introdotto in una dispensa precedente
  - la gestione dei ristoranti avviene tramite il servizio **RestaurantService**
  - i ristoranti sono definiti come un’entità JPA **Restaurant** – con attributi **id**, **name** e **location**
  - internamente al servizio, i ristoranti vengono acceduti da una base di dati mediante un repository **RestaurantRepository**

44

Invocazione remota: gRPC

Luca Cabibbo ASW



## Il servizio RestaurantService

- La gestione dei ristoranti avviene tramite il servizio **RestaurantService**, con le seguenti operazioni

```
package asw.efood.restaurant.service.domain;

import ...

@Service
public class RestaurantService {

    ...

    public Restaurant createRestaurant(String name, String location) {
        ...
    }

    public Restaurant getRestaurant(Long id) { ... }

    public Collection<Restaurant> getAllRestaurants() { ... }

}
```



## - Definizione dell'interfaccia del servizio

- Il file **RestaurantService.proto** specifica l'interfaccia del servizio

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "asw.efood.restaurant.service.api.grpc";

service RestaurantService {

    rpc createRestaurant(CreateRestaurantRequest)
        returns (CreateRestaurantReply) {}

    rpc getRestaurant(GetRestaurantRequest)
        returns (GetRestaurantReply) {}

    rpc getAllRestaurants(GetAllRestaurantsRequest)
        returns (GetAllRestaurantsReply) {}

}
```

- da questo, verrà generato il package **asw.efood.restaurant.service.api.grpc** con la classe **RestaurantServiceGrpc** per i proxy – da condividere tra server e client



# Definizione dell'interfaccia del servizio

- Il file `RestaurantService.proto` specifica l'interfaccia del servizio

```
message CreateRestaurantRequest {
  string name = 1;
  string location = 2;
}

message CreateRestaurantReply {
  int64 restaurantId = 1;
}
```

```
message GetRestaurantRequest {
  int64 restaurantId = 1;
}

message GetRestaurantReply {
  int64 restaurantId = 1;
  string name = 2;
  string location = 3;
}
```

```
message GetAllRestaurantsRequest {
}

message GetAllRestaurantsReply {
  repeated GetRestaurantReply restaurants = 1;
}
```

- questi messaggi definiscono un modello di presentazione “neutrale” per interagire con il nostro servizio



## - Server Grpc

- Lato server, la classe `RestaurantServiceGrpcServer` nel package `asw.efood.restaurantservice.grpc` implementa l'adattatore inbound per gRPC per il nostro servizio

```
package asw.efood.restaurantservice.grpc;

import asw.efood.restaurantservice.domain.*;
import asw.efood.restaurantservice.api.grpc.*;
import ...

@Component
public class RestaurantServiceGrpcServer {
    ... vedi dopo ...
}
```





## Server Grpc

- Nella classe **RestaurantServiceGrpcServer** è necessaria innanzitutto l'iniezione del servizio **RestaurantService**

```
@Autowired
private RestaurantService restaurantService;
```

- in questo modo, l'adattatore può invocare le operazioni definite dalla logica di business del servizio



## Server Grpc

- Nella classe **RestaurantServiceGrpcServer** sono poi necessarie anche alcune definizioni preliminari standard per l'utilizzo di GRPC

```
@Value("${asw.efood.restaurantservice.grpc.port}")
private int port;

private Server server;

@PostConstruct
public void start() throws IOException {
    this.server = ServerBuilder.forPort(port)
        .addService( new RestaurantServiceImpl() )
        .build().start();
}

@PreDestroy
public void stop() {
    if (server != null) { server.shutdown(); }
}
```

# application.properties  
asw.efood.restaurantservice.grpc.port=50052



## Server Grpc

- Il collegamento tra il server gRPC e il servizio `restaurantService` è realizzato mediante la classe interna `RestaurantServiceImpl`

```
private class RestaurantServiceImpl
    extends RestaurantServiceGrpc.RestaurantServiceImplBase {

    @Override
    public void createRestaurant(...) { ... }

    @Override
    public void getRestaurant(...) { ... }

    @Override
    public void getAllRestaurants(...) { ... }
}
```

- nell'implementazione di queste operazioni, si noti l'adattamento tipico di un adapter (in questo caso di tipo inbound, lato server) – e l'adattamento tra il modello di dominio interno del servizio e il modello di presentazione usato con gRPC



## Server Grpc

- L'operazione `createRestaurant`

```
public void createRestaurant(CreateRestaurantRequest request,
    StreamObserver<CreateRestaurantReply> responseObserver) {

    String name = request.getName();
    String location = request.getLocation();
    Restaurant restaurant =
        restaurantService.createRestaurant(name, location);
    CreateRestaurantReply reply =
        CreateRestaurantReply.newBuilder()
            .setRestaurantId(restaurant.getId())
            .build();
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
}
```



# Server Grpc

## □ L'operazione `getRestaurant`

```
public void getRestaurant(GetRestaurantRequest request,
    StreamObserver<GetRestaurantReply> responseObserver) {

    Long restaurantId = request.getRestaurantId();
    Restaurant restaurant =
        restaurantService.getRestaurant(restaurantId);
    GetRestaurantReply reply = GetRestaurantReply.newBuilder()
        .setRestaurantId(restaurant.getId())
        .setName(restaurant.getName())
        .setLocation(restaurant.getLocation())
        .build();
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
}
```



# Server Grpc

## □ L'operazione `getAllRestaurants`

```
public void getAllRestaurants(GetAllRestaurantsRequest request,
    StreamObserver<GetAllRestaurantsReply> responseObserver) {

    Collection<Restaurant> restaurants =
        restaurantService.getAllRestaurants();
    List<GetRestaurantReply> rr = restaurants.stream()
        .map(restaurant -> GetRestaurantReply.newBuilder()
            .setRestaurantId(restaurant.getId())
            .setName(restaurant.getName())
            .setLocation(restaurant.getLocation())
            .build())
        .collect(Collectors.toList());
    GetAllRestaurantsReply reply =
        GetAllRestaurantsReply.newBuilder()
            .addAllRestaurants(rr)
            .build();
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
}
```



# Architettura esagonale del servizio

- Architettura esagonale del servizio applicativo **restaurant-service**



55

Invocazione remota: gRPC

Luca Cabibbo ASW



## - Client Grpc

- Il lato client è un'altra applicazione o servizio che vuole accedere, in questo esempio, al servizio **restaurant-service**
  - supponiamo che sia un'altra applicazione Spring Boot, il cui package di base è `asw.efood.samplerestaurantclient`
  - nel package `asw.efood.samplerestaurantclient.domain` va definita un'interfaccia richiesta (una porta) `RestaurantClientPort` per accedere al servizio dei ristoranti
  - poi, applicando l'architettura esagonale, va definito un outbound adapter per accedere al servizio dei ristoranti
    - a tal fine, definiamo il package `asw.efood.samplerestaurantclient.restaurantclient.grpc` con la classe `RestaurantClientGrpcAdapter`

56

Invocazione remota: gRPC

Luca Cabibbo ASW



## Client Grpc

- Lato client, l'interfaccia **RestaurantClientPort**

```
package asw.efood.samplerestaurantclient.domain;
import java.util.*;

public interface RestaurantClientPort {
    Long createRestaurant(String name, String location);
    Restaurant getRestaurant(Long restaurantId);
    List<Restaurant> getAllRestaurants();
}
```

- questa è sostanzialmente la controparte dell'interfaccia del servizio **RestaurantService** nel dominio del server



## Client Grpc

- Lato client, nello stesso package, definiamo anche una classe **Restaurant**
  - che è la controparte dall'entità **Restaurant** nel dominio del server

```
public class Restaurant {
    private Long id;
    private String name;
    private String location;
    ... costruttori e metodi get, set e toString ...
}
```

- si tratta però una classe diversa dalla classe **Restaurant** nel dominio del server – ad es., non è un'entità



## Client Grpc

- Attenzione, l'interfaccia lato client per il servizio dei ristoranti potrebbe anche essere definita diversamente dall'interfaccia del servizio lato server – e potrebbe anche far riferimento a un modello di dominio differente
  - ad es., potrebbe essere definita così – ma noi consideriamo la definizione precedente

```
/* Definizione alternativa dell'interfaccia del servizio
 * per i ristoranti. */
public interface ServizioRistoranti {

    Long crea(String nome, String città);
    Ristorante getRistorante(Long id);
    List<Ristorante> getRistoranti();

}

public class Ristorante {
    ...
}
```

59

Invocazione remota: gRPC

Luca Cabibbo ASW



## Client Grpc

- Anche in questo caso, che cosa faccia il client finale di questo servizio è poco rilevante per la nostra discussione
  - tuttavia, ecco una porzione di esempio del nostro client – si noti la dipendenza dall'interfaccia/porta **RestaurantClientPort** – e soprattutto che il client non dipende in alcun modo da gRPC

```
package asw.efood.samplerestaurantclient.domain;

import ...;

@Component
public class RestaurantClientRunner implements CommandLineRunner {

    @Autowired
    private RestaurantClientPort restaurantClientAdapter;

    public void run(String[] args) {

        ... restaurantClientAdapter.getRestaurant(42L) ...

    }

}
```

60

Invocazione remota: gRPC

Luca Cabibbo ASW



## Client Grpc

- Lato client, la classe **RestaurantClientGrpcAdapter**

```
package asw.efood.samplerestaurantclient.restaurantclient.grpc;
import asw.efood.samplerestaurantclient.domain.*;
import asw.efood.restaurantservice.api.grpc.*;
import ...;
@Service
public class RestaurantClientGrpcAdapter
    implements RestaurantClientPort {
    ... vedi dopo ...
}
```



## Client Grpc

- Alcune definizioni preliminari nella classe **RestaurantClientGrpcAdapter**

```
@Value("${asw.efood.restaurantservice.grpc.host}")
private String host;
@Value("${asw.efood.restaurantservice.grpc.port}")
private int port;
```

```
# application.properties
asw.efood.restaurantservice.grpc.host=localhost
asw.efood.restaurantservice.grpc.port=50052
```

```
private ManagedChannel channel;
private HelloServiceGrpc.HelloServiceBlockingStub blockingStub;
```



## Client Grpc

- Alcune definizioni preliminari nella classe

### RestaurantClientGrpcAdapter

- metodi di supporto all'inizializzazione ed alla deallocazione della connessione con il server gRPC

```
@PostConstruct
public void init() {
    this.channel = ManagedChannelBuilder.forAddress(host, port)
        .usePlaintext()
        .build();
    this.blockingStub =
        RestaurantServiceGrpc.newBlockingStub(channel);
}

@PreDestroy
public void shutdown() throws InterruptedException {
    channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
}
```



## Client Grpc

- Il metodo `createRestaurant`

```
public Long createRestaurant(String name, String location) {
    Long restaurantId = null;
    CreateRestaurantRequest request =
        CreateRestaurantRequest.newBuilder()
            .setName(name)
            .setLocation(location)
            .build();
    try {
        CreateRestaurantReply reply =
            blockingStub.createRestaurant(request);
        restaurantId = reply.getRestaurantId();
    } catch (StatusRuntimeException e) { ... gRPC failed ... }
    return restaurantId;
}
```

- si noti l'adattamento tipico di un adapter (in questo caso outbound, lato client) – e l'adattamento tra il modello di dominio interno e il modello di presentazione usato con gRPC





# Client Grpc

## Il metodo `getRestaurant`

```
public Restaurant getRestaurant(Long restaurantId) {
    Restaurant restaurant = null;
    GetRestaurantRequest request =
        GetRestaurantRequest.newBuilder()
            .setRestaurantId(restaurantId).build();
    try {
        GetRestaurantReply reply =
            blockingStub.getRestaurant(request);
        if (reply!=null) {
            restaurant = new Restaurant(
                reply.getRestaurantId(),
                reply.getName(),
                reply.getLocation());
        }
    } catch (StatusRuntimeException e) { ... gRPC failed ... }
    return restaurant;
}
```



# Client Grpc

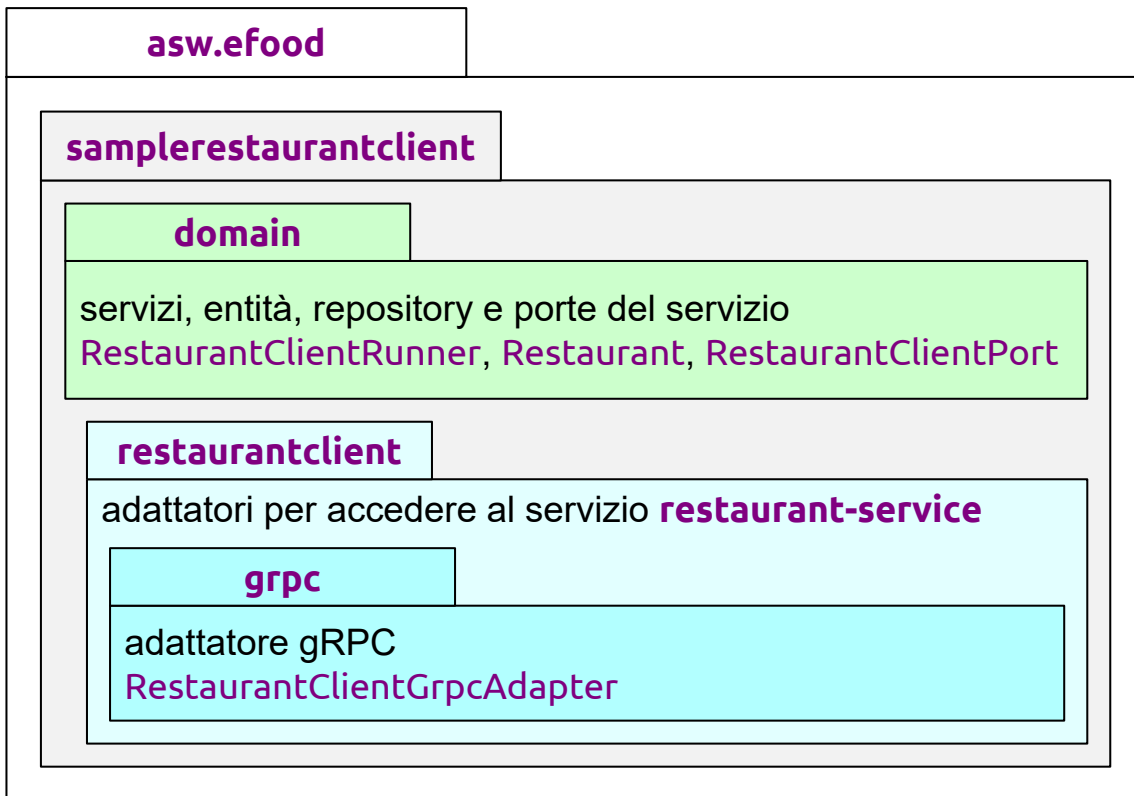
## Il metodo `getAllRestaurants`

```
public List<Restaurant> getAllRestaurants() {
    List<Restaurant> restaurants = null;
    GetAllRestaurantsRequest request =
        GetAllRestaurantsRequest.newBuilder().build();
    try {
        GetAllRestaurantsReply reply =
            blockingStub.getAllRestaurants(request);
        if (reply!=null) {
            restaurants = reply.getRestaurantsList().stream()
                .map( restaurant -> new Restaurant(
                    restaurant.getRestaurantId(),
                    restaurant.getName(),
                    restaurant.getLocation() ) )
                .collect(Collectors.toList());
        }
    } catch (StatusRuntimeException e) { ... gRPC failed ... }
    return restaurants;
}
```



# Architettura esagonale del servizio

- Architettura esagonale del servizio **sample-restaurant-client**



67

Invocazione remota: gRPC

Luca Cabibbo ASW



## - Esercizio

- In un precedente esercizio è stato richiesto di estendere il servizio per la gestione dei ristoranti con la gestione dei menu dei ristoranti
  - nel dominio del servizio dei ristoranti, il menu **RestaurantMenu** di un ristorante **Restaurant** è un elenco di **Menuitem** (ciascuno con un id, un nome e un prezzo)
  - andavano definite un'operazione per trovare il menu di un ristorante e un'operazione per creare il menu di un ristorante
  - qui si chiede
    - lato server, di esporre queste funzionalità mediante gRPC
    - lato client, di invocare queste funzionalità
    - si supponga che, nel dominio del client, un ristorante **Restaurant** abbia una lista di **RestaurantMenuitem** (ciascuno con un id, una descrizione e un prezzo) – infatti, il modello di dominio del server e del client possono essere strutturati in modi differenti

68

Invocazione remota: gRPC

Luca Cabibbo ASW



## \* Discussione

- gRPC è un framework open source per l'invocazione remota (RPC), moderno, ad alte prestazioni e interoperabile
  - l'uso di gRPC (come di altri framework e strumenti di RPC e RMI) richiede
    - la specifica dell'interfaccia del servizio da esporre remotamente, mediante un IDL – in questo caso, basata su Protocol Buffers
    - la scrittura di adapter lato server e lato client per il servizio – scritti con riferimento ai proxy (lato server e lato client) generati automaticamente da gRPC
    - i dati scambiati tra server e client vengono in genere rappresentati da un opportuno Presentation Model – in questo caso, quello generato automaticamente tramite Protocol Buffers a partire dal file *proto*
    - attenzione alla semantica dell'invocazione remota