



gRPC - Communication Patterns

Sistemas Distribuidos



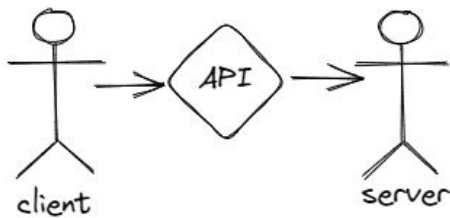


Separación en proyectos



Separación en proyectos

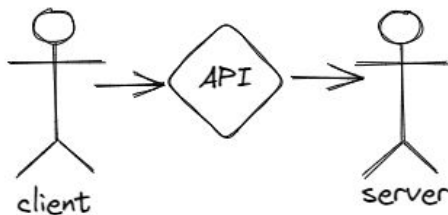
Queremos separar el código del servicio y del cliente para simular la situación real donde serían proyectos probablemente mantenidos por personas diferentes



Por lo cual generaremos un proyecto por rol

Separación en proyectos

En gRPC ambos roles/proyectos deben tener la definición de los servicios (el .proto) y generar el código del servicio a partir de esa definición. Sobre esto cada uno genera su código particular



Para no “copy-pastear” vamos a generar un tercer proyecto que contenga dicho código y cada uno lo podrá incorporar como dependencia

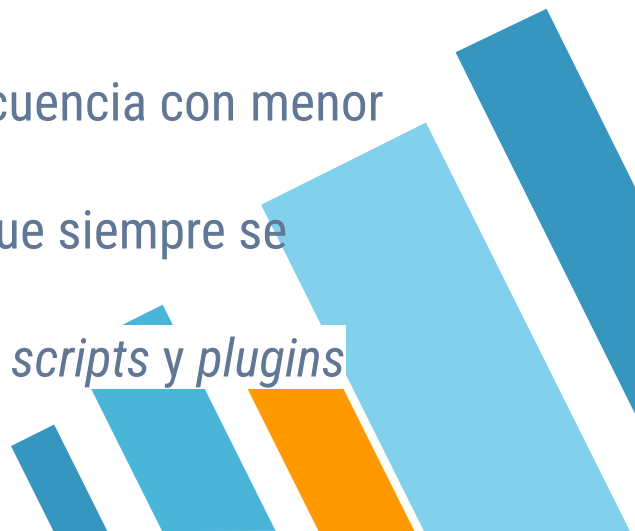


Separación en proyectos

Para que la generación de los proyectos no sea un proceso tedioso vamos a usar **archetypes de Maven**.

Un arquetipo es un proyecto de Maven que permite generar proyectos a partir de un *template*.

Esto permite que:

- El proceso de creación sea automático y en consecuencia con menor posibilidad de error.
 - Se pueden agregar al template dependencias que siempre se usan más allá del proyecto.
 - También se pueden agregar otros elementos como *scripts* y *plugins* que pueden ser útiles para el proyecto
- 

Ejercicio 1 - Instalar el archetype

1. Bajar el archivo **pod-grpc-archetype.zip** y descomprimir.
2. Incorporar al IDE y analizar la estructura del proyecto.
3. Cada vez que se modifica el arquetipo hay que instalarlo usando:

```
$> mvn clean install
```

4. La primera vez que se lo instala puede ser necesario correr el comando:

```
$> mvn archetype:crawl
```

Ejercicio 2 - Usar el archetype

1. Ir a una carpeta dentro de la cual se quiera crear el proyecto y correr

```
mvn archetype:generate -DarchetypeCatalog=local
```

2. Seleccionar el arquetipo:

```
local -> ar.edu.itba.pod:pod-mp-grpc-archetype (pod)
```

3. Crear un proyecto con los siguientes parámetros

```
'groupId': ar.edu.itba.pod.grpc  
'artifactId': grpc-com-patterns  
'version': 2025.1Q  
'package' ar.edu.itba.pod.grpc
```



Proyectos generados por Maven


Los proyectos vienen con el ***plugin maven-assembly configurado***, este *plugin* permite empaquetar proyectos con sus dependencias en un tar.gz:

A medida que se va generando el código para el proyecto se puede:

```
$> mvn clean install
```

para construir el proyecto, correr tests y empaquetar.


Genera un archivo **tar.gz** con las clases y jars de dependencias listo para usar, uno por cada proyecto.





Proyectos generados por Maven

Para ejecutar los proyectos **client** y **server**

1. Entrar en la carpeta **target** de cada proyecto.
 2. Descomprimir los ***-bin.tar.gz** (usando `tar -xzf <path>`).
 3. Entrar en la carpeta que aparece y dar permisos de ejecución a los scripts **run-*.sh** (`chmod u+x <path>`).
 4. Revisar dentro de los scripts que el Fully Qualified Name de la clase a ejecutar sea el correcto.
 5. Una vez hechos estos pasos, se puede ejecutar en 2 consolas diferentes:
- 

Ejercicio 3 - Finalizar proyecto

1. Bajar de campus el archivo **health_service.proto** e implementar el servicio.
2. Probar correr cliente y servidor desde la consola.



Comunicación






gRPC - Comunicación

Como dijimos, uno de los beneficios de utilizar un framework como gRPC es la existencia del *middleware* que nos aísla de la forma de comunicación haciendo que nuestro trabajo sea relativamente parecido a trabajar “local”.

gRPC basa la comunicación en dos protocolos rápidos y eficientes:


- **HTTP/2:** para la comunicación y transporte
 - **Protocol Buffers:** para la serialización de la información
- 



gRPC - Transporte

HTTP/2 es el protocolo de transporte. Es una mejora sobre el protocolo HTTP que permite generar aplicaciones más robustas, rápidas y eficientes.

Sus principales características son:

- **Mantener la semántica de HTTP**
 - **Una única conexión “permanente”**
 - **Multiplexed streams**
 - **Server push**
 - **Compresión de HEADERS**
 - **Formato binario**
- 

Marshalización / Serialización


- **Serializar** un objeto significa convertir su estado en un *stream* de bytes que pueda luego ser utilizado para obtener una copia de la información del objeto.
- **Marshalizar:** codifica no sólo la información que contiene el objeto, sino que también la información de la definición del objeto (o cómo construirlo o al menos cómo encontrar la referencia a la definición).
- **De-serializar** un objeto significa tomar un *stream* de bytes (que corresponden a un objeto serializado previamente) y obtener a partir de él un objeto con la información que tenía el objeto original.
- **“Un-marshalling”** es similar a de-serializar, salvo que además permite obtener la definición del objeto.

En algunos lenguajes y situaciones los términos se utilizan como sinónimos



gRPC - Protocol Buffer


Como herramienta de serialización Protocol Buffer es


- **Independiente:** del lenguaje y la plataforma.
 - **Tipado:** permite validar y optimizar por tipos
 - **Binario:** Al no ser un protocolo textual es más rápido de leer y escribir para las máquinas (Machine readable).
 - **Provee generadores:** para la serialización/deserialización en la mayoría de los lenguajes de programación.
- 




gRPC - Flujo

Cuando un cliente llama a un método del servicio (utilizando el *stub*) el proceso que se realiza es:

1. El **stub** serializa los parámetros del método del servicio.
 2. El **stub** genera un POST request usando HTTP, se genera la conexión y envía headers que indican el tipo de llamado, el método a correr remotamente, etc.
 3. Se envía el mensaje por la red.
 4. Al recibir el mensaje el servidor lee los headers para ver el método a llamar e invoca al **skeleton** correspondiente pasándole el mensaje.
 5. El **skeleton** de-serializa los parámetros y realiza la llamada local a la implementación del *Servant* del método remoto.
 6. Con la respuesta del *Servant* se genera el proceso inverso para que la respuesta le llegue al cliente
- 



Protocol Buffer - Definición de Mensajes



Protocol Buffer - Servicio

Un servicio se define con un nombre y un listado de métodos que contienen un mensaje de entrada y uno de salida

```
service UserService {  
    rpc DoLogin(LoginInformation) returns (User);  
    rpc GetRoles(User) returns (UserRoles);  
}
```

Protocol Buffer - Mensajes

Cada mensaje es un nombre y el conjunto de los campos que lo componen

```
/*  
 * Representation of the User in the system.  
 */  
message User {  
    int32 id = 1; //id is a positive integer  
    string user_name = 2;  
}
```

Recordamos:

- Los números son el identificador de campo, así que deben ser diferentes comenzando en 1.
- Los comentarios son estilo C

Protocol Buffer - Primitivas

El protocolo permite utilizar las siguientes primitivas:

.proto Type	Notes	Java Type
double		double
float		float
int32	Uses variable-length encoding. if your field is likely to have negative values, use sint32 instead.	int
int64	Uses variable-length encoding. – if your field is likely to have negative values, use sint64 instead.	long
uint32	Uses variable-length encoding.	int
uint64	Uses variable-length encoding.	long
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 228.	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 256.	long
sfixed32	Always four bytes.	int
sfixed64	Always eight bytes.	long
bool		boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 232.	String
bytes	May contain any arbitrary sequence of bytes no longer than 232.	ByteString

gRPC - Wrappers

Como los servicios solo pueden recibir y devolver mensajes el mismo protocolo provee una serie de wrappers que definen mensajes para:

- primitivas
- objeto vacío

```
message StringValue {  
    // The string value.  
    string value = 1;  
}
```

```
message Empty {  
}
```

Entonces se los usa en los servicios:

```
service PingService {  
    rpc Ping(google.protobuf.Empty) returns (google.protobuf.StringValue);  
}
```

Protocol Buffer - Campos de tipo Mensaje

Se puede utilizar un mensaje como tipo de un campo de otro mensaje

```
message LoginInformation {  
    string user_name = 1;  
    string password = 2;  
}
```

```
message User {  
    int32 id = 1;  
    LoginInformation login_information = 2;  
    string display_name = 3;  
}
```

Protocol Buffer - Enum

Protobuf permite usar **enum** como tipo para definir un tipo con un número finito de posibles elementos

```
enum AccountStatus {  
    ACCOUNT_STATUS_UNSPECIFIED = 0;  
    ACCOUNT_STATUS_PENDING = 1;  
    ACCOUNT_STATUS_ACTIVE = 2;  
}
```

```
message User {  
    int32 id = 1;  
    string user_name = 2;  
    string display_name = 3;  
    AccountStatus status = 4;  
}
```

A diferencia de los campos en el mensaje, para los enums el número 0 debe estar presente y debería terminar con el sufijo UNSPECIFIED

Protocol Buffer - Colecciones

Protocol Buffer permite dos tipos de colecciones:

Listas

```
message User {  
    int32 id = 1;  
    string user_name = 2;  
    repeated string preferences = 5;  
}
```

Mapas

```
message UserRoles {  
    map<string, Role> roles_by_site = 1;  
}
```


Ejercicio 4 - UserService

1. Agregar al proyecto el **user_service.proto** e implementar los métodos del mismo.



Llamados Asincrónicos





gRPC - Llamados Asincrónicos

Para situaciones donde queremos utilizar llamados no bloqueantes gRPC provee dos estrategias de llamados asincrónicos:


- Resolución por Futures
 - Resolución por Observer / Callback
- 



gRPC - Llamados Asincrónicos

De esta manera podemos aprovechar las ventajas de realizar llamados asincrónicos y liberar los recursos del sistema mientras el servidor procesa el pedido.

En principio para usar llamadas asincrónicas el código en el servidor no varía, si cambia el cliente.





gRPC - FutureStub

El cliente instancia un nuevo *stub* llamado ***FutureStub** a partir del método `newFutureStub` del código generado

```
UserServiceGrpc.UserServiceFutureStub userServiceFutureStub =  
    UserServiceGrpc.newFutureStub(channel);
```





gRPC - FutureStub

Al realizar un llamado a los métodos remotos del **FutureStub** el mismo retorna un **ListenableFuture**

```
public interface ListenableFuture <V extends Object> extends Future<V> {  
    void addListener(Runnable runnable, Executor executor);  
}
```



gRPC - FutureStub

Por lo que permite acceder a la respuesta de dos maneras:

- Utilizando las funciones de la clase Futures.
- Agregando un *callback*.

```
Futures.addCallback(userFuture, userCallback, executorService);
```



gRPC - FutureStub

El callback es un FutureCallback que provee un método para el caso de error y otro para el *success*.

```
class UserCallback implements FutureCallback<User> {  
    private static Logger logger =  
        LoggerFactory.getLogger(UserCallback.class);  
  
    @Override  
    public void onSuccess(final User user) {  
        this.logger.info("received user {}", user);  
    }  
  
    @Override  
    public void onFailure(final Throwable throwable) {  
        this.logger.error("reaching for user", throwable);  
    }  
}
```


Ejercicio 5 - Future Stub

1. Agregar un nuevo método al cliente que solicite los roles utilizando un callback que loguee los mismos en consola.

gRPC - Async Stub

La estrategia alternativa para llamados asincrónicos es utilizar el *Stub*

```
UserServiceGrpc.UserServiceStub userServiceStub =  
    UserServiceGrpc.newStub(channel);
```

En este caso los métodos definidos por el servicio además del parámetro de entrada reciben un segundo parámetro de tipo **StreamObserver** que funciona como *callback*

```
userServiceStub.doLogin(loginInformation, observer);
```

gRPC - Observer

La interfaz Stream Observer se utiliza para definir al callback.

```
public interface StreamObserver<V> {  
    void onNext(V var1);  
    void onError(Throwable var1);  
    void onCompleted();  
}
```

Se llama

- **onNext:** Se llama por cada elemento enviado.
- **onError:** Se llama en caso de un error.
- **onCompleted:** Se llama al terminar el proceso para indicar que se terminó de enviar la respuestas

Ejercicio 6 - Async Stub

1. Implementar el cliente de asincrónico no bloqueante para el UserService