Guía paso a paso para recrear el proyecto taller-grpc-envoy (servidor gRPC Java ↔ proxy Envoy con transcodificación REST)
==========

1) Crear carpeta base y entrar
```bash
mkdir grpc-envoy && cd grpc-envoy
```

2) Crear pom.xml

3) Sustituir el contenido de pom.xml por lo siguiente
```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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.ejemplo</groupId>
  <artifactId>grpc-native-workshop</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <grpc.version>1.56.0</grpc.version>
    <protobuf.version>3.23.4</protobuf.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-netty-shaded</artifactId>
      <version>${grpc.version}</version>
    </dependency>
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-protobuf</artifactId>
```

```
      <version>${grpc.version}</version>
    </dependency>
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-stub</artifactId>
      <version>${grpc.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.annotation</groupId>
      <artifactId>javax.annotation-api</artifactId>
      <version>1.3.2</version>
    </dependency>
  </dependencies>

  <build>
    <extensions>
      <extension>
        <groupId>kr.motd.maven</groupId>
        <artifactId>os-maven-plugin</artifactId>
        <version>1.7.0</version>
      </extension>
    </extensions>
    <plugins>
      <plugin>
        <groupId>org.xolstice.maven.plugins</groupId>
        <artifactId>protobuf-maven-plugin</artifactId>
        <version>0.6.1</version>
        <configuration>

<protocArtifact>com.google.protobuf:protoc:${protobuf.version}:exe:${os.detecte
d.classifier}</protocArtifact>
          <pluginId>grpc-java</pluginId>
          <pluginArtifact>io.grpc:protoc-gen-grpc-
java:${grpc.version}:exe:${os.detected.classifier}</pluginArtifact>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>compile-custom</goal>
            </goals>
```

```
            </execution>
          </executions>
        </plugin>

        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-shade-plugin</artifactId>
          <version>3.4.1</version>
          <executions>
            <execution>
              <phase>package</phase>
              <goals>
                <goal>shade</goal>
              </goals>
              <configuration>
                <transformers>
                  <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTr
ansformer">
                    <mainClass>com.ejemplo.ecommerce.GrpcServer</mainClass>
                  </transformer>
                </transformers>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
</project>
```

4) Crear estructura de carpetas
```bash
mkdir -p src/main/java/com/ejemplo/ecommerce
mkdir -p src/main/proto/google/api
mkdir -p envoy
```

5) Crear src/main/proto/order_service.proto

6) Sustituir el contenido de src/main/proto/order_service.proto por lo siguiente

```proto
syntax = "proto3";

package ecommerce;

import "google/api/annotations.proto";

option java_multiple_files = true;
option java_package = "com.ejemplo.ecommerce";

service OrderService {

 // 1. CREAR: Mapea un POST HTTP. El body "*" significa que todo el JSON
 // entrante se intentará mapear al mensaje CreateOrderRequest.
 rpc CreateOrder (CreateOrderRequest) returns (OrderResponse) {
  option (google.api.http) = {
   post: "/v1/orders"
   body: "*"
  };
 }

 // 2. CONSULTAR: Mapea un GET HTTP con parámetro de ruta (path parameter).
 rpc GetOrder (GetOrderRequest) returns (OrderResponse) {
  option (google.api.http) = {
   get: "/v1/orders/{order_id}"
  };
 }
}

// -- Mensajes de Datos --

enum OrderStatus {
 PENDING = 0;
 SHIPPED = 1;
 DELIVERED = 2;
 CANCELLED = 3;
}

message OrderItem {
 string product_id = 1;
 int32 quantity = 2;
```

```
  double price = 3;
}

message CreateOrderRequest {
  string customer_id = 1;
  repeated OrderItem items = 2; // Lista de objetos
}

message GetOrderRequest {
  string order_id = 1;
}

message OrderResponse {
  string order_id = 1;
  string customer_id = 2;
  repeated OrderItem items = 3;
  OrderStatus status = 4; // Enum
  double total_amount = 5;
}
```

7) Crear src/main/proto/google/api/annotations.proto

8) Sustituir el contenido de src/main/proto/google/api/annotations.proto
```proto
// Copyright 2025 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
// See the License for the specific language governing permissions and
// limitations under the License.

syntax = "proto3";
```

```
package google.api;

import "google/api/http.proto";
import "google/protobuf/descriptor.proto";

option go_package =
"google.golang.org/genproto/googleapis/api/annotations;annotations";
option java_multiple_files = true;
option java_outer_classname = "AnnotationsProto";
option java_package = "com.google.api";
option objc_class_prefix = "GAPI";

extend google.protobuf.MethodOptions {
  // See `HttpRule`.
  HttpRule http = 72295728;
}
```

9) Crear src/main/proto/google/api/http.proto

10) Sustituir el contenido de src/main/proto/google/api/http.proto

```proto
// Copyright 2025 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
// See the License for the specific language governing permissions and
// limitations under the License.

syntax = "proto3";

package google.api;
```

```
option go_package =
"google.golang.org/genproto/googleapis/api/annotations;annotations";
option java_multiple_files = true;
option java_outer_classname = "HttpProto";
option java_package = "com.google.api";
option objc_class_prefix = "GAPI";

// Defines the HTTP configuration for an API service. It contains a list of
// [HttpRule][google.api.HttpRule], each specifying the mapping of an RPC method
// to one or more HTTP REST API methods.
message Http {
  // A list of HTTP configuration rules that apply to individual API methods.
  //
  // **NOTE:** All service configuration rules follow "last one wins" order.
  repeated HttpRule rules = 1;

  // When set to true, URL path parameters will be fully URI-decoded except in
  // cases of single segment matches in reserved expansion, where "%2F" will be
  // left encoded.
  //
  // The default behavior is to not decode RFC 6570 reserved characters in multi
  // segment matches.
  bool fully_decode_reserved_expansion = 2;
}

// gRPC Transcoding
//
// gRPC Transcoding is a feature for mapping between a gRPC method and one or
// more HTTP REST endpoints. It allows developers to build a single API service
// that supports both gRPC APIs and REST APIs. Many systems, including [Google
// APIs](https://github.com/googleapis/googleapis),
// [Cloud Endpoints](https://cloud.google.com/endpoints), [gRPC
// Gateway](https://github.com/grpc-ecosystem/grpc-gateway),
// and [Envoy](https://github.com/envoyproxy/envoy) proxy support this feature
// and use it for large scale production services.
//
// `HttpRule` defines the schema of the gRPC/REST mapping. The mapping
specifies
// how different portions of the gRPC request message are mapped to the URL
// path, URL query parameters, and HTTP request body. It also controls how the
```

```
// gRPC response message is mapped to the HTTP response body. `HttpRule` is
// typically specified as an `google.api.http` annotation on the gRPC method.
//
// Each mapping specifies a URL path template and an HTTP method. The path
// template may refer to one or more fields in the gRPC request message, as long
// as each field is a non-repeated field with a primitive (non-message) type.
// The path template controls how fields of the request message are mapped to
// the URL path.
//
// Example:
//
//    service Messaging {
//      rpc GetMessage(GetMessageRequest) returns (Message) {
//        option (google.api.http) = {
//            get: "/v1/{name=messages/*}"
//        };
//      }
//    }
//    message GetMessageRequest {
//      string name = 1; // Mapped to URL path.
//    }
//    message Message {
//      string text = 1; // The resource content.
//    }
//
// This enables an HTTP REST to gRPC mapping as below:
//
// - HTTP: `GET /v1/messages/123456`
// - gRPC: `GetMessage(name: "messages/123456")`
//
// Any fields in the request message which are not bound by the path template
// automatically become HTTP query parameters if there is no HTTP request body.
// For example:
//
//    service Messaging {
//      rpc GetMessage(GetMessageRequest) returns (Message) {
//        option (google.api.http) = {
//            get:"/v1/messages/{message_id}"
//        };
//      }
//    }
```

```
//    message GetMessageRequest {
//      message SubMessage {
//        string subfield = 1;
//      }
//      string message_id = 1; // Mapped to URL path.
//      int64 revision = 2;    // Mapped to URL query parameter `revision`.
//      SubMessage sub = 3;    // Mapped to URL query parameter `sub.subfield`.
//    }
//
// This enables a HTTP JSON to RPC mapping as below:
//
// - HTTP: `GET /v1/messages/123456?revision=2&sub.subfield=foo`
// - gRPC: `GetMessage(message_id: "123456" revision: 2 sub:
// SubMessage(subfield: "foo"))`
//
// Note that fields which are mapped to URL query parameters must have a
// primitive type or a repeated primitive type or a non-repeated message type.
// In the case of a repeated type, the parameter can be repeated in the URL
// as `...?param=A&param=B`. In the case of a message type, each field of the
// message is mapped to a separate parameter, such as
// `...?foo.a=A&foo.b=B&foo.c=C`.
//
// For HTTP methods that allow a request body, the `body` field
// specifies the mapping. Consider a REST update method on the
// message resource collection:
//
//    service Messaging {
//      rpc UpdateMessage(UpdateMessageRequest) returns (Message) {
//        option (google.api.http) = {
//          patch: "/v1/messages/{message_id}"
//          body: "message"
//        };
//      }
//    }
//    message UpdateMessageRequest {
//      string message_id = 1; // mapped to the URL
//      Message message = 2;   // mapped to the body
//    }
//
// The following HTTP JSON to RPC mapping is enabled, where the
// representation of the JSON in the request body is determined by
```

```
// protos JSON encoding:
//
// - HTTP: `PATCH /v1/messages/123456 { "text": "Hi!" }`
// - gRPC: `UpdateMessage(message_id: "123456" message { text: "Hi!" })`
//
// The special name `*` can be used in the body mapping to define that
// every field not bound by the path template should be mapped to the
// request body.  This enables the following alternative definition of
// the update method:
//
//    service Messaging {
//      rpc UpdateMessage(Message) returns (Message) {
//        option (google.api.http) = {
//          patch: "/v1/messages/{message_id}"
//          body: "*"
//        };
//      }
//    }
//    message Message {
//      string message_id = 1;
//      string text = 2;
//    }
//
//
// The following HTTP JSON to RPC mapping is enabled:
//
// - HTTP: `PATCH /v1/messages/123456 { "text": "Hi!" }`
// - gRPC: `UpdateMessage(message_id: "123456" text: "Hi!")`
//
// Note that when using `*` in the body mapping, it is not possible to
// have HTTP parameters, as all fields not bound by the path end in
// the body. This makes this option more rarely used in practice when
// defining REST APIs. The common usage of `*` is in custom methods
// which don't use the URL at all for transferring data.
//
// It is possible to define multiple HTTP methods for one RPC by using
// the `additional_bindings` option. Example:
//
//    service Messaging {
//      rpc GetMessage(GetMessageRequest) returns (Message) {
//        option (google.api.http) = {
```

```
//         get: "/v1/messages/{message_id}"
//         additional_bindings {
//           get: "/v1/users/{user_id}/messages/{message_id}"
//         }
//       };
//     }
//   }
//   message GetMessageRequest {
//     string message_id = 1;
//     string user_id = 2;
//   }
//
// This enables the following two alternative HTTP JSON to RPC mappings:
//
// - HTTP: `GET /v1/messages/123456`
// - gRPC: `GetMessage(message_id: "123456")`
//
// - HTTP: `GET /v1/users/me/messages/123456`
// - gRPC: `GetMessage(user_id: "me" message_id: "123456")`
//
// Rules for HTTP mapping
//
// 1. Leaf request fields (recursive expansion nested messages in the request
//   message) are classified into three categories:
//   - Fields referred by the path template. They are passed via the URL path.
//   - Fields referred by the [HttpRule.body][google.api.HttpRule.body]. They
//   are passed via the HTTP
//     request body.
//   - All other fields are passed via the URL query parameters, and the
//     parameter name is the field path in the request message. A repeated
//     field can be represented as multiple query parameters under the same
//     name.
//  2. If [HttpRule.body][google.api.HttpRule.body] is "*", there is no URL
//  query parameter, all fields
//    are passed via URL path and HTTP request body.
//  3. If [HttpRule.body][google.api.HttpRule.body] is omitted, there is no HTTP
//  request body, all
//    fields are passed via URL path and URL query parameters.
//
// Path template syntax
//
```

```
//    Template = "/" Segments [ Verb ] ;
//    Segments = Segment { "/" Segment } ;
//    Segment  = "*" | "**" | LITERAL | Variable ;
//    Variable = "{" FieldPath [ "=" Segments ] "}" ;
//    FieldPath = IDENT { "." IDENT } ;
//    Verb     = ":" LITERAL ;
//
// The syntax `*` matches a single URL path segment. The syntax `**` matches
// zero or more URL path segments, which must be the last part of the URL path
// except the `Verb`.
//
// The syntax `Variable` matches part of the URL path as specified by its
// template. A variable template must not contain other variables. If a variable
// matches a single path segment, its template may be omitted, e.g. `{var}`
// is equivalent to `{var=*}`.
//
// The syntax `LITERAL` matches literal text in the URL path. If the `LITERAL`
// contains any reserved character, such characters should be percent-encoded
// before the matching.
//
// If a variable contains exactly one path segment, such as `"{var}"` or
// `"{var=*}"`, when such a variable is expanded into a URL path on the client
// side, all characters except `[-_.~0-9a-zA-Z]` are percent-encoded. The
// server side does the reverse decoding. Such variables show up in the
// [Discovery
// Document](https://developers.google.com/discovery/v1/reference/apis) as
// `{var}`.
//
// If a variable contains multiple path segments, such as `"{var=foo/*}"`
// or `"{var=**}"`, when such a variable is expanded into a URL path on the
// client side, all characters except `[-_.~/0-9a-zA-Z]` are percent-encoded.
// The server side does the reverse decoding, except "%2F" and "%2f" are left
// unchanged. Such variables show up in the
// [Discovery
// Document](https://developers.google.com/discovery/v1/reference/apis) as
// `{+var}`.
//
// Using gRPC API Service Configuration
//
// gRPC API Service Configuration (service config) is a configuration language
// for configuring a gRPC service to become a user-facing product. The
```

```
// service config is simply the YAML representation of the `google.api.Service`
// proto message.
//
// As an alternative to annotating your proto file, you can configure gRPC
// transcoding in your service config YAML files. You do this by specifying a
// `HttpRule` that maps the gRPC method to a REST endpoint, achieving the same
// effect as the proto annotation. This can be particularly useful if you
// have a proto that is reused in multiple services. Note that any transcoding
// specified in the service config will override any matching transcoding
// configuration in the proto.
//
// The following example selects a gRPC method and applies an `HttpRule` to it:
//
//    http:
//      rules:
//       - selector: example.v1.Messaging.GetMessage
//         get: /v1/messages/{message_id}/{sub.subfield}
//
// Special notes
//
// When gRPC Transcoding is used to map a gRPC to JSON REST endpoints, the
// proto to JSON conversion must follow the [proto3
// specification](https://developers.google.com/protocol-
buffers/docs/proto3#json).
//
// While the single segment variable follows the semantics of
// [RFC 6570](https://tools.ietf.org/html/rfc6570) Section 3.2.2 Simple String
// Expansion, the multi segment variable **does not** follow RFC 6570 Section
// 3.2.3 Reserved Expansion. The reason is that the Reserved Expansion
// does not expand special characters like `?` and `#`, which would lead
// to invalid URLs. As the result, gRPC Transcoding uses a custom encoding
// for multi segment variables.
//
// The path variables **must not** refer to any repeated or mapped field,
// because client libraries are not capable of handling such variable expansion.
//
// The path variables **must not** capture the leading "/" character. The reason
// is that the most common use case "{var}" does not capture the leading "/"
// character. For consistency, all path variables must share the same behavior.
//
```

```
// Repeated message fields must not be mapped to URL query parameters,
because
// no client library can support such complicated mapping.
//
// If an API needs to use a JSON array for request or response body, it can map
// the request or response body to a repeated field. However, some gRPC
// Transcoding implementations may not support this feature.
message HttpRule {
 // Selects a method to which this rule applies.
 //
 // Refer to [selector][google.api.DocumentationRule.selector] for syntax
 // details.
 string selector = 1;

 // Determines the URL pattern is matched by this rules. This pattern can be
 // used with any of the {get|put|post|delete|patch} methods. A custom method
 // can be defined using the 'custom' field.
 oneof pattern {
  // Maps to HTTP GET. Used for listing and getting information about
  // resources.
  string get = 2;

  // Maps to HTTP PUT. Used for replacing a resource.
  string put = 3;

  // Maps to HTTP POST. Used for creating a resource or performing an action.
  string post = 4;

  // Maps to HTTP DELETE. Used for deleting a resource.
  string delete = 5;

  // Maps to HTTP PATCH. Used for updating a resource.
  string patch = 6;

  // The custom pattern is used for specifying an HTTP method that is not
  // included in the `pattern` field, such as HEAD, or "*" to leave the
  // HTTP method unspecified for this rule. The wild-card rule is useful
  // for services that provide content to Web (HTML) clients.
  CustomHttpPattern custom = 8;
 }
```

```
  // The name of the request field whose value is mapped to the HTTP request
  // body, or `*` for mapping all request fields not captured by the path
  // pattern to the HTTP body, or omitted for not having any HTTP request body.
  //
  // NOTE: the referred field must be present at the top-level of the request
  // message type.
  string body = 7;

  // Optional. The name of the response field whose value is mapped to the HTTP
  // response body. When omitted, the entire response message will be used
  // as the HTTP response body.
  //
  // NOTE: The referred field must be present at the top-level of the response
  // message type.
  string response_body = 12;

  // Additional HTTP bindings for the selector. Nested bindings must
  // not contain an `additional_bindings` field themselves (that is,
  // the nesting may only be one level deep).
  repeated HttpRule additional_bindings = 11;
}

// A custom pattern is used for defining custom HTTP verb.
message CustomHttpPattern {
  // The name of this custom HTTP verb.
  string kind = 1;

  // The path matched by this custom verb.
  string path = 2;
}
```

11) Generar el descriptor binario necesario para Envoy

```bash
protoc -I=src/main/proto --include_imports --include_source_info --
descriptor_set_out=envoy/proto.pb src/main/proto/order_service.proto
```

12) Crear src/main/java/com/ejemplo/ecommerce/GrpcServer.java

13) Sustituir el contenido de
src/main/java/com/ejemplo/ecommerce/GrpcServer.java
```java
package com.ejemplo.ecommerce;

import io.grpc.Server;
import io.grpc.ServerBuilder;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

public class GrpcServer {

  private Server server;

  private void start() throws IOException {
    /* The port on which the server should run */
    int port = 9090;

    // Construye e inicia el servidor gRPC.
    server = ServerBuilder.forPort(port)
        .addService(new OrderServiceImpl()) // Registra la implementación del
servicio.
        .build()
        .start();

    System.out.println("Servidor gRPC iniciado en el puerto " + port);

    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
      // Hook para un apagado ordenado (ej. al recibir Ctrl+C).
      System.err.println("*** Apagando servidor gRPC porque la JVM se está
cerrando");
      try {
        GrpcServer.this.stop();
      } catch (InterruptedException e) {
        e.printStackTrace(System.err);
      }
      System.err.println("*** Servidor apagado");
    }));
  }

  private void stop() throws InterruptedException {
```

```
        if (server != null) {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        }
    }

    /**
     * Espera a que el hilo principal termine (esto mantiene la app viva).
     */
    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }

    public static void main(String[] args) throws IOException, InterruptedException {
        final GrpcServer server = new GrpcServer();
        server.start();
        server.blockUntilShutdown();
    }
}
```

14) Crear src/main/java/com/ejemplo/ecommerce/OrderServiceImpl.java

15) Sustituir el contenido de
src/main/java/com/ejemplo/ecommerce/OrderServiceImpl.java

```java
package com.ejemplo.ecommerce;

import io.grpc.Status;
import io.grpc.stub.StreamObserver;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.ConcurrentHashMap;

public class OrderServiceImpl extends OrderServiceGrpc.OrderServiceImplBase {

    // Simulación de Base de Datos
    private final Map<String, OrderResponse> orderRepository = new
ConcurrentHashMap<>();
```

```
  @Override
  public void createOrder(CreateOrderRequest request,
StreamObserver<OrderResponse> responseObserver) {
    // Lógica de negocio: Calcular total y generar ID
    double total = request.getItemsList().stream()
        .mapToDouble(item -> item.getPrice() * item.getQuantity())
        .sum();

    String generatedId = UUID.randomUUID().toString();

    OrderResponse newOrder = OrderResponse.newBuilder()
        .setOrderId(generatedId)
        .setCustomerId(request.getCustomerId())
        .addAllItems(request.getItemsList()) // Copiar lista compleja
        .setStatus(OrderStatus.PENDING) // Asignar Enum por defecto
        .setTotalAmount(total)
        .build();

    // Guardar
    orderRepository.put(generatedId, newOrder);
    System.out.println("Orden creada: " + generatedId);

    // Responder
    responseObserver.onNext(newOrder);
    responseObserver.onCompleted();
  }

  @Override
  public void getOrder(GetOrderRequest request,
StreamObserver<OrderResponse> responseObserver) {
    String id = request.getOrderId();

    if (orderRepository.containsKey(id)) {
      responseObserver.onNext(orderRepository.get(id));
      responseObserver.onCompleted();
    } else {
      // Retornar un error gRPC específico.
      // Envoy traducirá Status.NOT_FOUND a HTTP 404 Not Found
automáticamente.
      responseObserver.onError(Status.NOT_FOUND
          .withDescription("La orden con ID " + id + " no existe.")
```

```
                .asRuntimeException());
      }
    }
}
```

16) Empaquetar Java (genera código gRPC y JAR sombreado)
```bash
mvn clean package
```

17) Crear envoy/envoy.yaml

18) Sustituir el contenido de envoy/envoy.yaml
```yaml
admin:
 access_log_path: /tmp/admin_access.log
 address:
  socket_address: { address: 0.0.0.0, port_value: 9901 }

static_resources:
 listeners:
 - name: listener_0
  address:
   socket_address: { address: 0.0.0.0, port_value: 8080 } # Puerto entrada REST
  filter_chains:
  - filters:
   - name: envoy.filters.network.http_connection_manager
     typed_config:
      "@type":
type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v
3.HttpConnectionManager
       stat_prefix: grpc_json
       codec_type: AUTO
       route_config:
        name: local_route
        virtual_hosts:
        - name: local_service
          domains: ["*"]
          routes:
          - match: { prefix: "/" }
```

```
            route:
              cluster: grpc_service
              timeout: 60s
        http_filters:
        # ---------------------------------------------------------------
        # FILTRO JSON TRANSCODER
        # Convierte HTTP/JSON -> gRPC y viceversa
        # ---------------------------------------------------------------
        - name: envoy.filters.http.grpc_json_transcoder
          typed_config:
            "@type":
type.googleapis.com/envoy.extensions.filters.http.grpc_json_transcoder.v3.GrpcJs
onTranscoder
            proto_descriptor: "/etc/envoy/proto.pb"   # El archivo binario compilado
            services: ["ecommerce.OrderService"]     # Nombre exacto package.Service

            # Opciones de formato JSON
            print_options:
              add_whitespace: true               # JSON "bonito" con saltos de línea
              always_print_primitive_fields: true    # Muestra campos aunque sean 0 o
false
              always_print_enums_as_ints: false      # Muestra "PENDING" en vez de 0
              preserve_proto_field_names: true       # Usa snake_case (product_id)

            # Configuración estricta de rutas
            match_incoming_request_route: false
            ignore_unknown_query_parameters: true

        - name: envoy.filters.http.router
          typed_config:
            "@type": type.googleapis.com/envoy.extensions.filters.http.router.v3.Router

  clusters:
  - name: grpc_service
    connect_timeout: 1.25s
    type: LOGICAL_DNS
    lb_policy: ROUND_ROBIN
    dns_lookup_family: V4_ONLY

    # Esta opción fuerza a Envoy a usar HTTP/2 para hablar con el backend Java
    # Es obligatorio para que gRPC funcione.
```

```
  http2_protocol_options: {}

  load_assignment:
      cluster_name: grpc_service
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                # ANTES: address: java-app
                # AHORA: Usamos el DNS especial de Docker para ver el host
                address: host.docker.internal
                port_value: 9090
```

19) Crear docker-compose.yml

20) Sustituir el contenido de docker-compose.yml

```yaml
version: '3.8'

services:
 # ------------------------------------------------------------
 # 2. Envoy Proxy (Traductor REST <-> gRPC)
 # ------------------------------------------------------------
 envoy:
  image: envoyproxy/envoy:v1.26-latest
  container_name: envoy-proxy
  ports:
   - "8080:8080"          # Puerto PÚBLICO para peticiones HTTP/REST
   - "9901:9901"          # Interfaz de administración de Envoy
  volumes:
   # Montamos la configuración
   - ./envoy/envoy.yaml:/etc/envoy/envoy.yaml
   # Montamos el descriptor binario (¡CRÍTICO para la traducción!)
   - ./envoy/proto.pb:/etc/envoy/proto.pb
  extra_hosts:
   - "host.docker.internal:host-gateway" # Asegura que Linux/Windows resuelvan
bien la IP
```

21) Crear pruebas_grpc_envoy.bat o pruebas_grpc_envoy.sh

22) Sustituir el contenido de pruebas_grpc_envoy.bat o pruebas_grpc_envoy.sh

```bat
@echo off
setlocal EnableDelayedExpansion
CLS

REM --- CONFIGURACION ---
set "URL=http://localhost:8080/v1/orders"

echo ============================================================
echo   TEST TALLER gRPC-REST (Modo Verboso)
echo ============================================================
echo.

REM ------------------------------------------------------------
REM PASO 1: CREAR ORDEN (POST)
REM ------------------------------------------------------------
echo [PASO 1] Creando una Orden...
echo.

REM 1. Preparamos el JSON en un archivo (para evitar problemas de comillas en CMD)
set "JSON_CONTENT={"customer_id": "Cliente_CMD_Visual", "items": [{"product_id": "Laptop_Pro", "quantity": 1, "price": 1200.50}]}"
echo %JSON_CONTENT% > orden_temp.json

REM 2. Mostramos el comando que vamos a ejecutar (Simulacion visual)
echo ------------------------------------------------------------
echo [COMANDO A EJECUTAR]:
echo curl -X POST %URL% ^
echo     -H "Content-Type: application/json" ^
echo     -d '%JSON_CONTENT%'
echo ------------------------------------------------------------
echo.

REM 3. Ejecutamos el comando real
curl -X POST %URL% -H "Content-Type: application/json" -d @orden_temp.json

echo.
```

```
echo.
echo ======================================================
echo  ARRIBA DEBERIAS VER EL JSON DE RESPUESTA.
echo  BUSCA EL "order_id" Y COPIALO.
echo ======================================================
echo.

REM --- INPUT MANUAL ---
set /p id=">> PEGA EL ID AQUI Y DALE ENTER: "
echo.

REM ------------------------------------------------------
REM PASO 2: CONSULTAR (GET)
REM ------------------------------------------------------
echo.
echo [PASO 2] Consultando la Orden...
echo.

REM Mostramos el comando
echo ------------------------------------------------------
echo [COMANDO A EJECUTAR]:
echo curl -X GET %URL%/%id%
echo ------------------------------------------------------
echo.

REM Ejecutamos
curl -X GET %URL%/%id%

echo.
echo.

REM ------------------------------------------------------
REM PASO 3: PROBAR ERROR 404
REM ------------------------------------------------------
echo.
echo [PASO 3] Probando Error 404 (ID Inexistente)...
echo.

REM Mostramos el comando
echo ------------------------------------------------------
echo [COMANDO A EJECUTAR]:
```

```
echo curl -v -X GET %URL%/ID_FALSO_123
echo ------------------------------------------------------
echo.

REM Ejecutamos (filtramos para resaltar el 404 si es posible, o mostramos todo)
curl -v -X GET %URL%/ID_FALSO_123 2>&1 | findstr "HTTP/1.1 404"

echo.
echo.
echo [FIN DEL TEST]
REM Limpieza
if exist orden_temp.json del orden_temp.json
pause
```
```bash
#!/bin/bash

# Limpiamos la pantalla
clear

# --- CONFIGURACION ---
URL="http://localhost:8080/v1/orders"

echo "============================================================"
echo "   TEST TALLER gRPC-REST (Modo Verboso - Linux/Mac)"
echo "============================================================"
echo ""

# ------------------------------------------------------------
# PASO 1: CREAR ORDEN (POST)
# ------------------------------------------------------------
echo "[PASO 1] Creando una Orden..."
echo ""

# 1. Preparamos el JSON en un archivo
# Nota: Usamos comillas simples en 'JSON_CONTENT' para que bash no interprete las comillas dobles internas
JSON_CONTENT='{"customer_id": "Cliente_Bash_Visual", "items": [{"product_id": "Laptop_Pro", "quantity": 1, "price": 1200.50}]}'
echo "$JSON_CONTENT" > orden_temp.json
```

```
# 2. Mostramos el comando (Simulacion visual)
echo "-------------------------------------------------------"
echo "[COMANDO A EJECUTAR]:"
echo "curl -X POST $URL \\"
echo "     -H \"Content-Type: application/json\" \\"
echo "     -d '$JSON_CONTENT'"
echo "-------------------------------------------------------"
echo ""

# 3. Ejecutamos el comando real
curl -X POST "$URL" -H "Content-Type: application/json" -d @orden_temp.json

echo ""
echo ""
echo "======================================================="
echo "  ARRIBA DEBERIAS VER EL JSON DE RESPUESTA."
echo "  BUSCA EL 'order_id' Y COPIALO."
echo "======================================================="
echo ""

# --- INPUT MANUAL ---
# 'read -p' es el equivalente a 'set /p'
read -p ">> PEGA EL ID AQUI Y DALE ENTER: " id
echo ""

# -------------------------------------------------------
# PASO 2: CONSULTAR (GET)
# -------------------------------------------------------
echo ""
echo "[PASO 2] Consultando la Orden..."
echo ""

# Mostramos el comando
echo "-------------------------------------------------------"
echo "[COMANDO A EJECUTAR]:"
echo "curl -X GET $URL/$id"
echo "-------------------------------------------------------"
echo ""

# Ejecutamos
curl -X GET "$URL/$id"
```

```
echo ""
echo ""

# ------------------------------------------------------
# PASO 3: PROBAR ERROR 404
# ------------------------------------------------------
echo ""
echo "[PASO 3] Probando Error 404 (ID Inexistente)..."
echo ""

# Mostramos el comando
echo "------------------------------------------------------"
echo "[COMANDO A EJECUTAR]:"
echo "curl -v -X GET $URL/ID_FALSO_123"
echo "------------------------------------------------------"
echo ""

# Ejecutamos.
# Nota: '2>&1' redirige stderr a stdout. 'grep' reemplaza a 'findstr'.
curl -v -X GET "$URL/ID_FALSO_123" 2>&1 | grep "HTTP/1.1 404"

echo ""
echo ""
echo "[FIN DEL TEST]"

# Limpieza
if [ -f orden_temp.json ]; then
   rm orden_temp.json
fi

# Pausa final (read sin variable espera un enter)
read -p "Presiona ENTER para salir..."
```

23) Levantar componentes
```bash
# Terminal 1: proxy Envoy (requiere docker)
docker-compose up

# Terminal 2: servidor gRPC Java
```

java -jar target/grpc-native-workshop-1.0-SNAPSHOT.jar
```

24) Probar flujo desde Windows (opcional)
```bash
pruebas_grpc_envoy.bat
```

25) Probar flujo desde Linux/macOS (opcional)
```bash
# Ejecutar el script Bash que se encuentra en la raíz del proyecto
./pruebas_grpc_envoy.sh
```

Con estos pasos recreas el proyecto original con exactamente los mismos ficheros y contenidos. Copia/pega cada bloque en el archivo indicado para que se cree con el contenido correcto.
```