## **STEP A: Prepare the grpc-proto Repository**

**Structure of the repo :**

| grpc-proto/ ├── src/main/proto/ │ └── billing-service.proto ├── build.gradle ├── settings.gradle └── README.md |
| --- |

This step involves creating a dedicated repository for your Protobuf definitions. This ensures a single, centralized source of truth for all your service contracts.

### **A.1: Create the Proto File**

Create a directory named grpc-proto and inside it, create the file src/main/proto/billing\_service.proto. The content should be exactly as provided:

Protocol Buffers

| syntax = "proto3";  option java\_multiple\_files = true; option java\_package = "com.pm.billing"; option java\_outer\_classname = "BillingProto";  service BillingService {  rpc createBillingAccount (BillingRequest) returns (BillingResponse); }  message BillingRequest {  string patientId = 1;  string name = 2;  string email = 3; }  message BillingResponse {  string accountId = 1;  string status = 2; } |
| --- |

This file defines the BillingService with a single RPC method, createBillingAccount, and the corresponding request and response messages.

### 

### **A.2: Create the build.gradle File**

In the grpc-proto root directory, create a build.gradle file to configure the Protobuf compilation and Maven publication. This Gradle script uses the com.google.protobuf plugin to generate Java code from the .proto file and the maven-publish plugin to create a publishable artifact.

Gradle

| plugins {  id 'java'  id 'com.google.protobuf' version '0.9.4'  id 'maven-publish' }  group = 'com.pm' version = '1.0.0'  repositories {  mavenCentral() }  dependencies {  implementation "io.grpc:grpc-netty-shaded:1.69.0"  implementation "io.grpc:grpc-protobuf:1.69.0"  implementation "io.grpc:grpc-stub:1.69.0"  implementation "com.google.protobuf:protobuf-java:3.25.3" }  protobuf {  protoc {  artifact = "com.google.protobuf:protoc:3.25.3"  }  plugins {  grpc {  artifact = "io.grpc:protoc-gen-grpc-java:1.69.0"  }  }  generateProtoTasks {  all().each { task ->  task.plugins {  grpc {}  }  }  } }  java {  withSourcesJar()  withJavadocJar() }  publishing {  publications {  mavenJava(MavenPublication) {  from components.java  artifactId = 'pm-protos'  }  }  repositories {  // We'll configure GitHub Packages here later  } } |
| --- |

Also, create a settings.gradle file in the same directory with rootProject.name = 'grpc-proto'.

## **STEP B: Local Testing and Publication 🧪**

Before publishing to a remote repository, it's good practice to test the build and publication process locally. This step publishes the generated artifact to your local Maven repository.

Run the following command from the grpc-proto root:

Bash

| ./gradlew clean build publishToMavenLocal |
| --- |

This command performs several actions:

1. **Generates Java classes**: It compiles billing\_service.proto into Java files, including BillingServiceGrpc.java, and places them in build/generated/.
2. **Builds a JAR**: It packages the generated Java classes and the source code into a JAR file.
3. **Publishes to local Maven repo**: It installs the built JAR (com.pm:pm-protos:1.0.0) into your local Maven repository, typically located at ~/.m2/repository.

To verify the process, check if the file ~/.m2/repository/com/pm/pm-protos/1.0.0/pm-protos-1.0.0.jar exists. You can also inspect the contents of this JAR to confirm the generated Java classes are present.

## **STEP C: Publish to GitHub Packages 📦**

This step outlines how to publish the pm-protos artifact to GitHub Packages so that other services can consume it.

### **C.1: Create a GitHub Personal Access Token (PAT)**

You need a GitHub PAT with the write:packages scope to publish and read:packages to consume artifacts. Go to **GitHub** -> **Settings** -> **Developer settings** -> **Personal access tokens** -> **Tokens (classic)** to create a new token. Be sure to copy the token immediately, as you won't be able to see it again.

### **C.2: Store Credentials for Gradle**

For Gradle to authenticate with GitHub Packages, store your credentials in your ~/.gradle/gradle.properties file:

Ini, TOML

| gpr.user=YOUR\_GITHUB\_USERNAME gpr.key=YOUR\_PERSONAL\_ACCESS\_TOKEN |
| --- |

Replace the placeholders with your actual GitHub username and PAT. This is the recommended and most secure method.

### **C.3: Add GitHub Packages to build.gradle**

Update the publishing block in grpc-proto/build.gradle to include the GitHub Packages repository configuration:

Gradle

| publishing {  publications {  mavenJava(MavenPublication) {  from components.java  artifactId = 'pm-protos'  }  }  repositories {  maven {  name = "GitHubPackages"  url = uri("https://maven.pkg.github.com/OWNER/grpc-proto")  credentials {  username = project.findProperty("gpr.user") ?: System.getenv("GPR\_USER") ?: System.getenv("GITHUB\_ACTOR")  password = project.findProperty("gpr.key") ?: System.getenv("GPR\_KEY") ?: System.getenv("GITHUB\_TOKEN")  }  }  } } |
| --- |

***Important : Replace OWNER with your GitHub username or organization name.***

### **C.4: Publish to GitHub Packages**

Run the following command from the grpc-proto root:

Bash

| ./gradlew clean build publish |
| --- |

If successful, the pm-protos artifact (version 1.0.0) will be available in your GitHub repository's **Packages** section.

## **STEP D: Consume the Artifact in Services 🤝**

Now, configure both the patient-service and billing-service to use the pm-protos artifact.

### **D.1: Update build.gradle in Both Services**

In each service's build.gradle file, add the repositories and the dependency.

First, add the repositories block:

Gradle

| repositories {  mavenLocal() // useful for local dev (optional)  mavenCentral()  maven {  url = uri("https://maven.pkg.github.com/OWNER/grpc-proto")  credentials {  username = project.findProperty("gpr.user") ?: System.getenv("GPR\_USER") ?: System.getenv("GITHUB\_ACTOR")  password = project.findProperty("gpr.key") ?: System.getenv("GPR\_KEY") ?: System.getenv("GITHUB\_TOKEN")  }  } } |
| --- |

Then, add the dependency:

Gradle

| dependencies {  implementation "com.pm:pm-protos:1.0.0"  // other dependencies... } |
| --- |

**Important**: Make sure to remove any existing local copies of billing\_service.proto or generated Java files from these service repositories to prevent classpath conflicts.

### **D.2: Verify Compilation**

From each service's root directory, run:

Bash

| ./gradlew clean build -x test |
| --- |

**If something doesn’t work try hardcoding the value (Not recommended).**

This command will fetch the pm-protos artifact from either your local Maven repository or GitHub Packages and compile the service code.

## **STEP E: Build and Dockerize the Services 🐳**

This section focuses on preparing the services for containerization. The best practice here is to build the JARs locally and then copy them into a minimal Docker image.

### **E.1: Build JARs Locally**

From the root of each service, run the following command to build the executable JAR files.

For billing-service:

Bash

| ./gradlew clean bootJar -x test |
| --- |

For patient-service:

Bash

| ./gradlew clean bootJar -x test |
| --- |

This will create a JAR file in the build/libs/ directory of each service.

### **E.2: Create Dockerfiles**

Create a simple Dockerfile for each service that copies the pre-built JAR into the image.

billing-service/Dockerfile:

Dockerfile

| FROM eclipse-temurin:21-jre-jammy WORKDIR /app COPY build/libs/\*.jar app.jar EXPOSE 4001 9001 ENTRYPOINT ["java","-jar","/app/app.jar"] |
| --- |

patient-service/Dockerfile:

Dockerfile

| FROM eclipse-temurin:21-jre-jammy WORKDIR /app COPY build/libs/\*.jar app.jar EXPOSE 4000 ENTRYPOINT ["java","-jar","/app/app.jar"] |
| --- |

These Dockerfiles are minimal and only include the Java Runtime Environment and the application JAR, resulting in smaller, more secure images.

## **STEP F: Configure docker-compose.yml 📝**

A docker-compose.yml file is used to orchestrate the services and their dependencies. This file defines the patient-service, billing-service, and a PostgreSQL database.

Create a docker-compose.yml file in the root directory of your project:

YAML

| version: '3.8'  services:  patient-service-db:  image: postgres:16  environment:  POSTGRES\_DB: db  POSTGRES\_USER: admin\_user  POSTGRES\_PASSWORD: password  ports:  - "5432:5432"   billing-service:  build:  context: ./billing-service  container\_name: billing-service  environment:  SERVER\_PORT: 4001  GRPC\_SERVER\_PORT: 9001  SPRING\_PROFILES\_ACTIVE: docker  ports:  - "4001:4001" # HTTP  - "9001:9001" # gRPC   patient-service:  build:  context: ./patient-service  container\_name: patient-service  depends\_on:  - patient-service-db  - billing-service  environment:  SPRING\_DATASOURCE\_URL: jdbc:postgresql://patient-service-db:5432/db  SPRING\_DATASOURCE\_USERNAME: admin\_user  SPRING\_DATASOURCE\_PASSWORD: password  BILLING\_SERVICE\_ADDRESS: billing-service  BILLING\_SERVICE\_GRPC\_PORT: 9001  ports:  - "4000:4000" |
| --- |

The key configuration here is BILLING\_SERVICE\_ADDRESS: billing-service, which allows patient-service to communicate with billing-service using Docker's internal DNS.

## **STEP G: Start and Verify Containers ▶️**

With the docker-compose.yml file in place, you can build and run the services.

**Build images**: Make sure you've built the JARs as described in STEP E, and then run:  
Bash

| docker compose build --no-cache |
| --- |

**Start containers**:  
Bash

| docker compose up |
| --- |

1. **Check logs**: Use docker compose logs <service-name> -f to monitor the startup process for each service. This helps in diagnosing any startup failures.

## **STEP H: Test gRPC and HTTP Flow ✅**

After the containers are up, test the communication between them and the external endpoints.

### **H.1: Test gRPC with grpcurl**

You can use grpcurl to inspect and interact with the gRPC service directly.

**List services**:  
Bash  
grpcurl -plaintext localhost:9001 list

1. This should return BillingService.

**Call createBillingAccount**:  
Bash  
grpcurl -plaintext -d '{"patientId":"p1","name":"John","email":"john@ex.com"}' localhost:9001 BillingService/createBillingAccount

An example successful response would be:  
JSON

| {  "accountId": "1234",  "status": "SUCCESS" } |
| --- |

1. This confirms the billing-service is correctly handling gRPC requests.

### **H.2: Call Patient Service HTTP Endpoint**

Finally, test the end-to-end flow by calling the patient service's HTTP endpoint.

Bash

| curl -v -X POST http://localhost:4000/patients \  -H "Content-Type: application/json" \  -d '{"name":"Jane Doe","email":"jane@example.com"}' |
| --- |

This should trigger the patient service to save the patient to its database and then make a gRPC call to the billing service. Check the logs of both services to confirm the full workflow executed successfully.