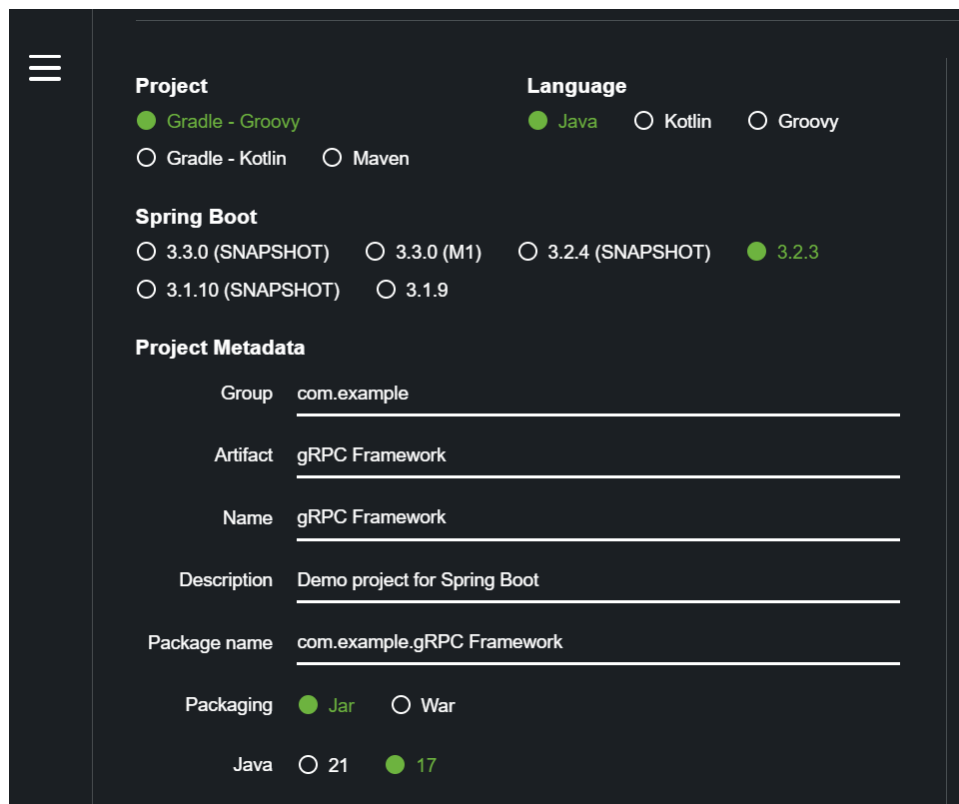


MidEng GK861 gRPC Framework [GK/EK]

von Ivan Milev

Projet Setup



The screenshot shows a project setup form with the following sections:

- Project**: Radio buttons for ☒ Gradle - Groovy, ☐ Gradle - Kotlin, and ☐ Maven.
- Language**: Radio buttons for ☒ Java, ☐ Kotlin, and ☐ Groovy.
- Spring Boot**: Radio buttons for ☐ 3.3.0 (SNAPSHOT), ☐ 3.3.0 (M1), ☐ 3.2.4 (SNAPSHOT), ☒ 3.2.3, ☐ 3.1.10 (SNAPSHOT), and ☐ 3.1.9.
- Project Metadata**:
 - Group**:
 - Artifact**:
 - Name**:
 - Description**:
 - Package name**:
 - Packaging**: Radio buttons for ☒ Jar and ☐ War.
 - Java**: Radio buttons for ☐ 21 and ☒ 17.

Protoc Installieren

<https://www.geeksforgeeks.org/how-to-install-protocol-buffers-on-windows/>

Richtige Dependencies installieren und die build.gradle Datei anpassen:

```

dependencies {
    // https://mvnrepository.com/artifact/com.google.protobuf/protobuf-javalite
    implementation group: 'com.google.protobuf', name: 'protobuf-javalite', version: '3.11.0'
    implementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    runtimeOnly 'io.grpc:grpc-netty-shaded:1.61.0'
    implementation 'io.grpc:grpc-protobuf:1.61.0'
    implementation 'io.grpc:grpc-stub:1.61.0'
    compileOnly 'org.apache.tomcat:annotations-api:6.0.53' // needed for grpc
}

tasks.named('test') {
    useJUnitPlatform()
}

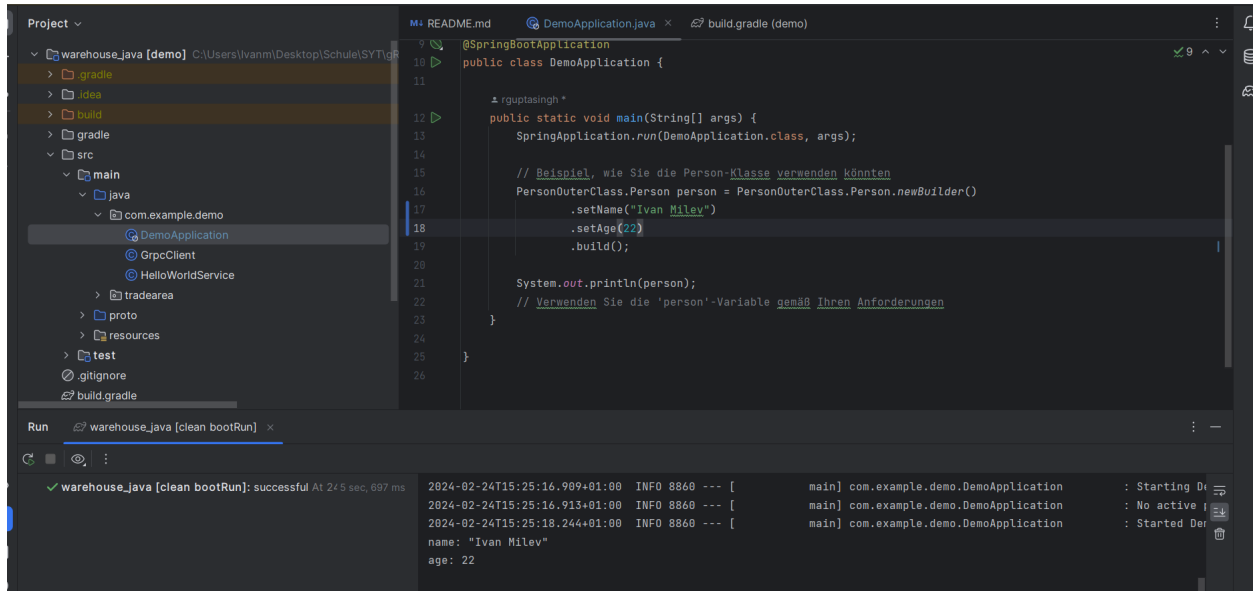
protobuf {
    generateProtoTasks {
        { task ->
            task.plugins {
                grpc {
                    // Use subdirectory 'grpcjava' instead of 'grpc'
                    outputSubDir = 'grpcjava'
                }
            }
        }
    }
}

sourceSets {
    main {
        java {
            srcDirs 'build/generated/source/proto/main/java', 'src/main/java'
        }
    }
}

```

```
}
}
```

Demo Application



Man sieht jetzt schön wie das gesamte Projekt läuft und das Demo Projekt ausgeführt wird.

GKV

In einer `.proto`-Datei werden die Datenstrukturen, Felder, Nachrichten und Dienste definiert, die in einem bestimmten Anwendungsfall verwendet werden sollen. Diese Dateien enthalten oft einfache, aber präzise Syntax, um Datenstrukturen zu beschreiben.

```
syntax = "proto3";

package tradearea;

// Definition für ProductData
```

```

message ProductData {
    string productId = 1;
    string productName = 2;
    string productCategory = 3;
    string productAmount = 4;
    string productUnit = 5;
}

// Definition für WarehouseData
message WarehouseData {
    string warehouseID = 1;
    string warehouseName = 2;
    string timestamp = 3;
    string warehouseCountry = 4;
    string warehouseCity = 5;
    string address = 6;
    repeated ProductData productData = 7;
}

// Service-Definition
service WarehouseService {
    rpc GetWarehouseData (WarehouseRequest) returns (WarehouseData);
}

// Anfrageformat für WarehouseData
message WarehouseRequest {
    string warehouseID = 1;
}

```

Grpc Server config

```

@Configuration
public class GrpcServerConfig {

```

```

@Bean(destroyMethod = "shutdown")
public Server grpcServer() throws IOException {
    return ServerBuilder.forPort(50051)
        .addService(new WarehouseServiceImpl())
        .build()
        .start();
}
}

```

Die Klasse `GrpcServerConfig` ist eine Konfigurationsklasse, die durch die Annotation `@Configuration` gekennzeichnet ist.

Die Methode

`grpcServer()` innerhalb dieser Konfigurationsklasse wird durch die Annotation `@Bean` gekennzeichnet. Diese Annotation gibt an, dass die Methode ein Spring-Bean bereitstellt, das vom Spring-Anwendungskontext verwaltet wird.

In diesem speziellen Fall erstellt die Methode `grpcServer()` ein gRPC-Serverobjekt. Der gRPC-Server wird mit einem Port konfiguriert (hier Port 50051), und ein Dienst (`WarehouseServiceImpl`) wird als Dienstanbieter zum Server hinzugefügt.

WarehouseService Implementation

```

public class WarehouseServiceImpl extends WarehouseServiceImplBase {

    @Override
    public void getWarehouseData(WarehouseRequest request, StreamObserver<WarehouseData> responseObserver) {
        // Hier können Sie die Logik hinzufügen, um die Daten zu abrufen
        // Zum Beispiel könnte es so aussehen:
        WarehouseData data = WarehouseData.newBuilder()
            .setWarehouseID(request.getWarehouseID())
            .setWarehouseName("Ivan Milevs Lagerhaus")
            .setWarehouseCity("Sofia")
            .setWarehouseCountry("Bulgaria")
            .setAddress("Polizeistraße 123")
            .setTimestamp("2024")
            .build();
    }
}

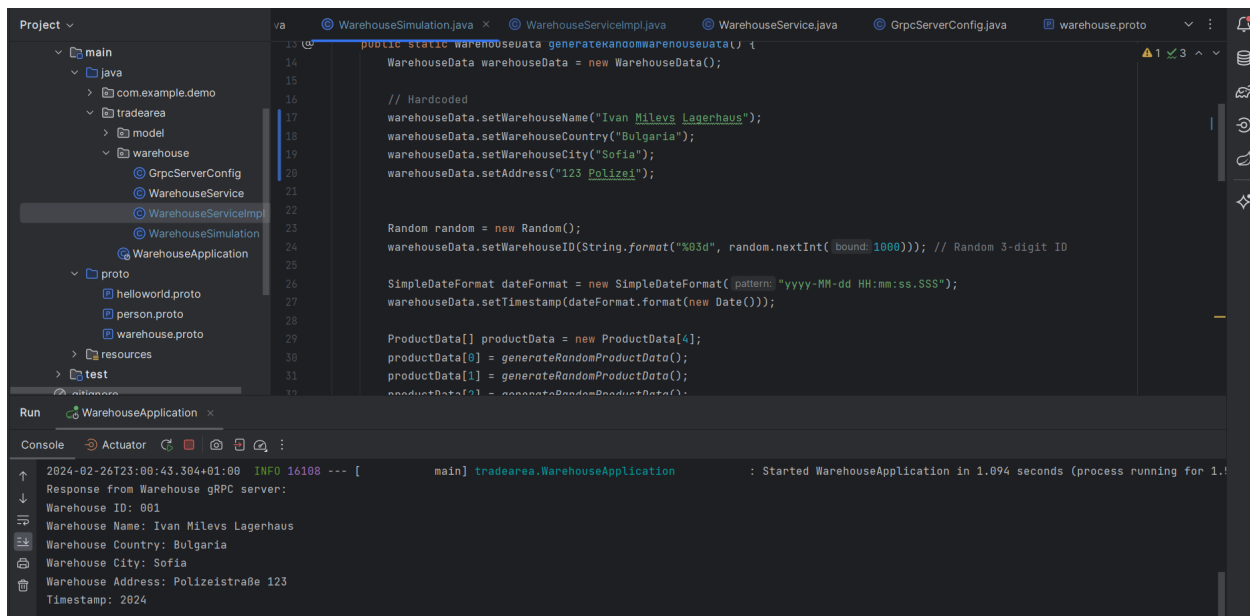
```

```

        responseObserver.onNext(data);
        responseObserver.onCompleted();
    }
}

```

Die Klasse `WarehouseServiceImpl` stellt eine Implementierung eines gRPC-Dienstes dar, der Methoden zum Abrufen von Lagerhausdaten bereitstellt. Sie ist dafür verantwortlich, Anfragen von Clientanwendungen entgegenzunehmen, die mit dem gRPC-Server kommunizieren möchten, und darauf zu reagieren, indem sie die angeforderten Daten zurücksendet.



Fragen

What is gRPC

gRPC ist ein leistungsstarkes, Open-Source-Universal-RPC (Remote Procedure Call)-Framework, das von Google entwickelt wurde. Es ermöglicht es Server- und Client-Anwendungen, transparent zu kommunizieren, und erleichtert den Aufbau verbundener Systeme. gRPC funktioniert über Sprachen und Plattformen hinweg,

indem es Protocol Buffers (Protobuf) als seine Schnittstellendefinitionssprache verwendet. Dies ermöglicht es Entwicklern, ihre Dienstmethoden und Nachrichtentypen auf eine sprach- und plattformneutrale Weise zu definieren. Dieser Ansatz stellt sicher, dass Dienste, die in verschiedenen Sprachen erstellt wurden, nahtlos miteinander kommunizieren können.

Why does it work across languages and different platforms

gRPC wurde so konzipiert, dass es sprachagnostisch und plattformunabhängig ist. Dies wird durch die Verwendung von Protocol Buffers erreicht, die als universelle Sprache für die Definition von Service-Schnittstellen und Nachrichtenstrukturen dienen. Der Protobuf-Compiler (protoc) kann Client- und Servercode aus .proto-Dateien für eine Vielzahl von Programmiersprachen generieren. Dies ermöglicht es Entwicklern, Dienste und Clients in ihrer bevorzugten Sprache zu erstellen, während gleichzeitig die Interoperabilität gewährleistet wird.

RPC-Lebenszyklus, beginnend mit dem RPC-Client:

1. **Service-Definition:** Bevor RPC-Aufrufe durchgeführt werden, werden Dienste und ihre Methoden in einer .proto-Datei unter Verwendung der Syntax von Protocol Buffers definiert.
2. **Initialisierung des Client-Stubs:** Der Client initialisiert einen Stub für den entfernten Dienst. Dieser Stub stellt Methoden bereit, die den in der .proto-Datei definierten Dienstmethoden entsprechen.
3. **Durchführung des Aufrufs:** Der Client führt einen Remote-Prozeduraufruf durch, indem er eine Methode auf dem Stub aufruft und die erforderlichen Anforderungsparameter als Argumente übergibt.
4. **Anforderungsserialisierung:** Der Client-Stub serialisiert die Anforderungsnachricht in ein binäres Format (unter Verwendung von Protocol Buffers) und sendet sie über das Netzwerk an den Server.
5. **Serververarbeitung:** Der Server empfängt die Anfrage, deserialisiert sie und ruft die entsprechende Dienstmethode auf.

6. **Antwort:** Der Server serialisiert die Antwort in ein binäres Format und sendet sie zurück an den Client.
7. **Client erhält Antwort:** Der Client-Stub deserialisiert die Antwort, und der Aufruf ist abgeschlossen – der Client hat jetzt das Ergebnis des Remote-Prozeduraufrufs.

Arbeitsablauf von Protocol Buffers:

1. **Nachricht und Dienst definieren:** Entwickler definieren Nachrichtenformate und Dienstschnittstellen in einer .proto-Datei unter Verwendung der Protobuf-Sprache.
2. **Codegenerierung:** Mit dem Protobuf-Compiler wird Quellcode aus der .proto-Datei für die gewünschten Sprachen generiert. Dieser generierte Code enthält Datenzugriffsklassen für jede Nachricht und Client-/Servercode für jeden Dienst.
3. **Implementierung:** Entwickler implementieren die generierten Server-Schnittstellen und verwenden die Client-Stubs in ihrem Anwendungscode, um RPCs durchzuführen und zu behandeln.
4. **Serialisierung/Deserialisierung:** Nachrichten werden automatisch von/zu binärem Format von den Client-/Server-Stubs serialisiert/deserialisiert, wenn RPC-Aufrufe durchgeführt oder empfangen werden.

Vorteile der Verwendung von Protocol Buffers:

1. **Effizienz:** Protobuf ist sowohl in Größe als auch Geschwindigkeit effizienter im Vergleich zu JSON und XML. Es handelt sich um ein binäres Format, das schneller zu serialisieren/deserialisieren ist und weniger Bandbreite erfordert.
2. **Unterstützung für verschiedene Sprachen:** Protobuf unterstützt die Codegenerierung in mehreren Sprachen, was eine nahtlose Kommunikation zwischen in verschiedenen Sprachen geschriebenen Diensten ermöglicht.
3. **Vorwärts- und Rückwärtskompatibilität:** Entwickler können neue Felder zu ihren Nachrichtendefinitionen hinzufügen, ohne die Rückwärtskompatibilität mit Diensten zu brechen, die gegen ältere Definitionen kompiliert wurden.

When Is the Use of Protocol Not Recommended?

- **Human Readability:** If human readability of the data format is a primary requirement, Protobuf might not be the best choice due to its binary nature¹⁸.
- **Small Projects or Microservices:** For small-scale projects or microservices where the overhead of defining Protobuf schemas and generating code might not be justified¹⁹.

3 Different Data Types That Can Be Used With Protocol Buffers

1. **Basic Types:** Such as `int32`, `float`, `double`, `bool`, and `string`, which cover most of the primitive data types needed in applications²⁰.
2. **Complex Types:** Custom defined `message` types, which allow the composition of basic types and other messages to create complex data structures²¹.
3. **Enumerations (`enum`):** These allow the definition of named constants, which can make protocols clearer and more readable by using symbolic names instead of integers²².