SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

GRPC STREAMING
SEMESTRÁLNA PRÁCA

M. Bilka, M. Kudlačík,
P. Sobota, A. Štang, A. Tóth

2022

Obsah

Zoznam skratiek

HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
REST	Representational state transfer
RPC	Remopte Procedure Call
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Zoznam výpisov

1 REST

REST je typ architektúry ktorý popisuje komunikáciu medzi klientom a serverom. REST používa komunikačný protokol HTTP 1.1 na prenos údajov. Koncept REST predstavil v roku 2000 Roy Fielding, ktorý spolupracoval aj na vývoji HTTP protokolu. Namiesto toho aby sa všetky požiadavky (requesty) posielali na jeden centrálny bod, tak sa ich použije viacej. Každý zdroj (resource) má práve jeden koncový bod (endpoint) definovaný pomocou URI. Zdroj (resource) je akýkoľvek objekt, dáta alebo servis ku ktorému môžeme pristúpiť cez príslušnú URI. URI je jednoznačný identifikátor pre každý zdroj (resource). Tento princíp definuje jednotné rozhranie čo znamená že, bez ohľadu na to odkiaľ príde požiadavka (request), vždy by mala byť rovnaká pre získanie prístupu ku konkrétnemu zdroju. Taktiež to zabezpečuje že klient a server sú úplne oddelený a nezávislý. REST rovnako definuje aj bezstavovosť, čiže server si neukladá žiadne informácie o stave klienta. Server bude s každým requestom od klienta zaobchádzať ako z úplne novým, čiže klient musí v requeste poskytnúť všetky potrebné informácie serveru pre spracovanie requestu. Formát na výmenu dát sa najčastejšie používa JSON alebo XML, čo sa zväčša definuje v hlavičke požiadavky. REST využíva metódy HTTP protokolu na popis čo sa má z dopytovaným zdrojom stať. Poznáme základné 4 HTTP metódy ktorými sú GET – na získavanie zdroju, POST – na pridanie zdroju, PUT – na úpravu zdroju a DELETE na vymazanie zdroju.

1.1 Výhody a nevýhody

Jednou z výhod REST-u je že, je plne podporovaný webovými prehliadačmi vďaka používaniu HTTP 1.1, kde gRPC potrebuje gRPC-web a proxy vrstvu na uskutočnenie konverzie z HTTP 1.1 na http 2. Ďalšou výhodou je už spomínaná bezstavovosť, ktorá zabezpečuje nezávislosť serveru a klienta. Netreba zabudnúť aj na to že REST vyniká svojou jednoduchosťou, ľahko sa modifikuje a je škálovateľný čiže ľahko sa dá rozšíriť o nové komponenty. Formáty na výmenu dát ktoré používa REST čiže, JSON a XML sú na jednej strane čitateľné pre ľudí, na druhej strane sa musia pred odosielaním serializovať a pri prijatí deserializovať, čo zaberá nejaký čas naviac. Keďže REST využíva request-response komunikačný model ktorý beží na HTTP 1.1 protokole, to znamená že, jeden servis môže naraz spracovať len jednu požiadavku, v tom prípade ak príde naraz viacero požiadaviek na jeden servis tak dochádza k spomaleniu. REST taktiež pri každom requeste musí vytvoriť nový TCP Hanshake, keďže si neuchováva žiaden stav, čo spôsobuje ďalšie spomalenie

procesu.

1.2 Použitie

Keďže bol tento projekt zameraný na JVM, rozhodli sme sa ho implementovať v jazyku Kotlin pre jeho efektívnu syntax. Na tvorbu serverovej časti sme využili Ktor čo je framework jazyka Kotlin pre tvorbu web aplikácií ktorý podporuje aj rest. Na správu závislostí sme využili systém Gradle. Pre použitie rest servisov sme si najprv museli zadefinovať závislosti pre framework Ktor a serializáciu json formátu ako môžeme vidieť na obrázku nižšie.

```
dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation("io.ktor:ktor-server-core:$ktorVersion")
    implementation("io.ktor:ktor-server-netty:$ktorVersion")
    implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
    implementation("io.ktor:ktor-server-content-negotiation:$ktorVersion")
}
```

Po zadefinovaní závislostí sme si vytvorili funkciu ktorá nám povolí používanie JSON serializácie. JSON serializáciu nám poskytuje knižnica Serialization ktorú sme si importovali v predchádzajúcom kroku. Taktiež sme si vytvorili funkciu v ktorej si budeme definovať jednotlivé endpointy. Máme konkrétne dva endpointy a to vypísanie všetkých lietadiel a vypísanie konkrétneho lietadla podľa id.

```
fun Application.configureSerialization() {
    install(ContentNegotiation) {
        json()
    }
}
fun Application.configureRouting() {
    routing {
        getPlaneById()
        getAllPlanes()
        get("/") {
            call.respondText("Hello, world!")
        }
    }
```

```
}
```

Následne sme si vytvorili dátovú triedu lietadla s názvom PlaneInfo, ktorá obsahuje informácie o lietadle. Objekty vytvorené z tejto triedy sa pri odosielaní odpovedi serverom serializujú na JSON objekty zodpovedajúce tejto triede a odošlú sa ako odpoveď klientovi. Ako si môžeme všimnúť nad dátovou triedou sme použili anotáciu @Serializable ktorú nám poskytuje knižnica ktorú sme si importovali v predchádzajúcich krokoch a postará sa o serializáciu dát.

```
@Serializable
data class PlaneInfo(
    private val id: Long,
    private val type: String,
    private val name: String,
    private val capacity: Int,
    private val startDestination: String,
    private val finalDestination: String,
    private var latitude: Double,
    private var longitude: Double,
    private var status: String
)
```

Ako posledné je treba vytvoriť jednotlivé endpointy ktoré sme si definovali v predošlej funkcii. Na obrázku nižšie je ukážka krátkeho endpointu ktorý vracia informácie o všetkých lietadlách.

```
fun Route.getAllPlanes() {
    get("/planes"){
        call.respond(
            HttpStatusCode.OK,
            Planes.allPlaneInformation().map {
                PlaneInfo.fromGrpcResponses(it.first, it.second)
            }
        )
    }
}
```

2 gRPC

GRPC je framework od spoločnosti Google, ktorý dovoľuje jednoducho implementovať komunikáciu medzi klientom a serverom pomocou techniky RPC. RPC je technika využívaná pri vývoji distribuovaných systémov, prvý krát bola presne popísaná v roku 1981 B. Nelsonom [1]. RPC umožnuje klientskému zariadeniu vykonať volanie subrutiny bez toho aby klientské zariadenie vedelo, že ide o vzdialené volanie a teda bez potreby riešiť problémy spojené s komunikáciou so vzdialeným zariadením [2].

2.1 Volania a streaming

Okrem synchrónnych blokujúcich volaní definovaných v RPC umožňuje gRPC aj asynchrónne volania bez blokovania volajúceho vlákna. GRPC poskytuje dva základné druhy volaní procedúr, unárne a streaming. Unárne volania sa podobajú volaniam architektúry REST, kde je ku každej poslanej požiadavke odoslaná práve jedna odpoveď. Pre nás zaujímavejším typom je práve streaming, ktorý umožňuje obísť toto viazanie požiadaviek a odpovedí unárnych volaní. Pri streamingu je možné k niekoľkým požiadavkám obdržať niekoľko odpovedí. Streaming sa pri gRPC delí na:

- streaming na klientskej strane - klient odošle niekoľko požiadaviek a od serveru obdrží jednu odpoveď
- streaming na strane servera - klient odošle jednu požiadavku a server odpovie tokom niekoľkých odpovedí
- obojstranný streaming - klient aj server odosielajú požiadavky a odpovede v tokoch, môžu posielať a prijímať dáta nezávislo na druhom zariadení

2.2 Výhody a nevýhody

Veľkou výhodou, ktorá umožňuje napríklad aj už popísané streamovanie a prenos niekoľkých požiadaviek paralelne je vytvorenie komunikačného kanála pri pripojení klienta k serveru. Tento kanál je potom využívaný na ďalšiu komunikáciu [3], nie je teda potrebné spojenie vytvárať pri každej odoslanej požiadavke ako napríklad pri použití rozhraní na báze REST. Takéto prepoužitie jedného kanála je gRPC dosiahlo použitím komunikačného protokolu HTTP verzie 2.0 [4]. Ďalšou veľkou výhodou gRPC je využitie technológie protocol buffers, ďalej iba protobuf, ako prenosového formátu a prevodníka dát. Narozdiel od rozhraní

na báze REST, kde sa Často využívajú formáty ako XML a JSON, ktoré je potrebné serializovať a deserializovať pri prechode z aplikačnej do prenosovej vrstvy. Protobuf požaduje presne určiť formát dát, a teda poskytuje typovú bezpečnosť dát. Protobuf kódujú dáta do binárnej podoby a prenášané dáta sú teda kompaktnejšie. O serializáciu aj deserializáciu rieši samotná knižnica protobuf, z ktorých sú vygenerované priamo pomocné triedy a súbory pre ľubovoľný podporovaný jazyk. Vytvorené proto súbory je teda možné prepoužiť na viacerých platformách a klientská a serverová časť môžu byť na sebe teda technologicky nezávislé.

Nevýhodou gRPC je hlavne menšia podpora, kým bežné HTTP rozhrania je možné dotazovať napríklad bežnými webovými prehliadačmi, na vykonanie gRPC dotazovej sú potrebné implementovať klientskú aplikáciu alebo použiť špecializované nástroje. Rovnako chýbajú aj dokumentačné nástroje, pre gRPC napríklad neexistuje priama náhrada nástroja OpenAPI. Pri hľadaní implementačných chýb môže predstavovať komplikáciu aj binárna reprezentácia odosielaných dát. Formát je síce efektívnejší pri prenose, no je pre človeka nečitateľný.

2.3 Použitie

Ako už bolo spomenuté, rozhodli sme sa ho implementovať v jazyku Kotlin pre jeho efektívnu syntax. Framework gRPC priamo podporuje jazyk Kotlin, integrácia teda nebola náročná. Prvý krokom bolo získať závislosti, na ich správu sme využili systém Gradle. Pre generáciu Kotlin tried z nášho protobuf predpisu rozhrania boli potrebné nasledovné závilosti. Ako môžeme vidieť, okrem závislostí frameworku gRPC sa v využíva aj balík `kotlinx-coroutines-core`. Tento balík poskytuje prístup ku takzvaným korutinám, čo je mechanizmus konkurentného programovania jazyka Kotlin. Tieto funkcionality sú používané na zabezpečenie konkurentného spracovania požiadaviek ale poskytujú aj triedy využívané pri implementácie gRPC stream.

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutinesVersion")
    implementation("io.grpc:grpc-netty:$grpcVersion")
    implementation("io.grpc:grpc-stub:$grpcVersion")
    implementation("io.grpc:grpc-protobuf:$grpcVersion")
    implementation("com.google.protobuf:protobuf-kotlin:$protobufVersion")
    implementation("io.grpc:grpc-kotlin-stub:$grpcKotlinVersion")
}
```

Predpis rozhrania je potrebné písať vo formáte už spomínaného balíka protobuf, príklad je možné vidieť nižšie. V prvom rade je potrebné zadať verziu syntaxe, možné je použiť aj rôznu konfiguráciu, v našom prípade sme uznali za vhodné generované triedy generovať do viacerých súborov. Potom už stačí definovať služby a jednotlivé nimi poskytované procedúry. Za generáciu tried je zodpovedné rozšírenie `com.google.protobuf`, príklad generovanej triedy nebudeme uvádzať, keďže generované triedy majú veľké množstvo dostupných metód aby bolo možné ich pohodlnejšie používať.

```
syntax = "proto3";

package sk.stuba.fei.asos.project24.plane;

option java_multiple_files = true;

service Plane {
  rpc FollowLocation(LocationRequest) returns (stream Location);
  rpc Information(PlaneInfoRequest) returns (PlaneInfo);
  rpc Register(RegisterRequest) returns (RegisterResponse);
}

message LocationRequest {

}

message Location {
  double latitude = 1;
  double longitude = 2;
  Status status = 3;
}
```

Po vygenerovaní tried rozhrania môžu všetky komponenty, či už klient alebo server aplikácie, použiť balík ako závislosť a implementovať potrebnú funkcionalitu. V prípade serverovej časti je potrebné implementovať definované služby a obsluu jednotlivých procedúr, príklad môžeme vidieť nižšie.

```
class PlaneService(
    coroutineContext: CoroutineContext = Dispatchers.Default
): PlaneGrpcKt.PlaneCoroutineImplBase(coroutineContext) {
```

```
    override fun followLocation(request: LocationRequest) = flow {
        while (true) {
            emit(PlaneLocation.currentLocation)
            delay(Config.delaySeconds.seconds)
        }
    }

    override suspend fun information(request: PlaneInfoRequest): PlaneInfo {
        return PlaneInfo.newBuilder()
            .setId(PlaneData.id)
            .setName(PlaneData.name)
            .setType(PlaneData.type)
            .setCapacity(PlaneData.capacity ?: 0)
            .setStartCity(PlaneData.start)
            .setEndCity(PlaneData.destination)
            .build()
    }

    override suspend fun register(request: RegisterRequest): RegisterResponse {
        return RegisterResponse.newBuilder().setId(PlaneData.id).build()
    }
}
```

Ako si môžeme všimnúť, procedúry ktoré nevyužívajú streaming majú návratovú hodnotu s typom odpovede, ktorú má server poslať. Procedúry využívajúce streaming majú návratovú hodnotu typu `Flow`, tento dátový typ v knižnici jazyka Kotlin predstavuje obálku nad tokom dát kde dáta môžu priebežne prichádzať.

Po implementácii je potrebné spustiť server, ktorý bude obsluhovať požiadavky na dané služby, príklad takéhoto spustenia môžeme vidieť nižšie.

```
private val server: Server = ServerBuilder
        .forPort(Config.grpcPort)
        .addService(PlaneService(Dispatchers.Default))
        .build()
```

Po imlementácii servera je na rade posledný krok našej implementácie, implementácia klientskej časti. Nižšie môžeme vidieť príklad vytvorenia komunikačného kanála medzi serverom a klientom. Rovnako môžeme vidieť aj vytvorenie takzvaného objektu `stub`,

tento objekt predstavuje samotného klienta a vytvára pre klientsku stranu ilúziu lokálneho volania procedúr. Pre zachovanie optimálneho výkonu je potrebné kanál ako aj stub používať čo najdlhšie bez opätovnej inicializácie.

```
private val channel = ManagedChannelBuilder.forAddress(clientAddress,
    clientPort).usePlaintext().build()
private val stub = PlaneGrpcKt.PlaneCoroutineStub(channel)
```

Bežným postupom je vytvorenie klientskej triedy ktorá s danými prvkami narába a poskytuje už čisté rozhranie pre ďalšie triedy, ktoré by procedúry potrebovali využívať. Ako príklad, viditeľný nižšie, uvádzame inicializáciu komunikácie s inštanciou lietadla.

```
suspend fun initPlane(): Long {
        return stub.register(RegisterRequest.getDefaultInstance()).id
            .also {
                CoroutineScope(Dispatchers.Default).launch {
                    stub.followLocation(LocationRequest.getDefaultInstance()).collect{
                        currentLocation = it }
                }
            }
    }
```

V tomto príklade môžeme vidieť spracovanie unárneho aj streamovacieho volania. Volanie `register` je unárne a môžeme vidieť, že k požadovanému atribútu `id` môžeme pristúpiť jednoducho ako k atribútu bežného objektu jazyka Kotlin. Volanie `followLocation` je rovnaké volanie, ktoré sme spomínali vyššie a návratová hodnota je už spomínaného typu `Flow`. Prichádzajúce hodnoty teda vieme jednoducho ukladať do bežnej premennej v čase aktualizácie zo strany servera a ďalej jednoducho využiť v ďalších triedach. Aktualizácia sa ale musí diať v kontexte už spomínaných korutín aby bol zabezpečený neblokujúci chod spracovania údajov konkurentne s hlavný vláknom.

3 Docker

Docker je open-source platforma pre vytváranie, spúšťanie a spravovanie kontajnerov. Kontajnery sú malé a izolované aplikácie, ktoré môžu byť spustené a prevádzkované na rovnakom stroji bez toho, aby došlo k prepísaniu systémových súborov. Je to efektívny spôsob, ako zabezpečiť, aby aplikácia fungovala rovnako, či je nasadená kdekoľvek. Docker sa obvykle používa pre aplikácie s viacerými časťami, ktoré je možné nakonfigurovať pomocou rôznych kontajnerov pre rôzne časti.

3.1 Docker Compose

Docker Compose je nástroj na správu kontajnerov pomocou súboru v jazyku YAML. Poskytuje jednoduchý spôsob, ako definovať a spustiť viac kontajnerov súčasne. Používa sa pre vývoj a nasadenie aplikácií zložených z viacerých kontajnerov. Pomocou Docker Compose môžete definovať všetky kontajnery, ktoré potrebujete pre aplikáciu, a zavolať jediný príkaz na spustenie všetkých.

3.2 Výhody a nevýhody

Docker Compose umožňuje ľahkú správu a konfiguráciu viacerých kontajnerov, automatizáciu procesov spustenia a údržby kontajnerov, spravovanie služieb s podsieťou viacerých kontajnerov, nastavenie vzdialenej infraštruktúry, nakonfigurovanie sieťových nastavení pre kontajner, jednoduché nasadenie aplikácie do viacerých prostredí, ľahké zdieľanie kontajnerov medzi používateľmi a ľahké spustenie aplikácií viacerých virtuálnych strojov. Avšak konfigurácia zložitých aplikácií je komplikovaná, sledovanie a ladenie aplikácie je obtiažne, nie je kompatibilný so všetkými verziami Dockeru, vyžaduje vyššiu úroveň znalostí pre správu kontajnerov a niekedy sa môže stať, že sa kontajnery po reštarte neobnovia[5].

3.3 Využitie

Rozhodli sme sa použiť Docker Compose pre dispečing a jednotlivé lietadla, nakoľko nám umožňuje izoláciu jednotlivých častí, je rýchly a kompatibilný s väčšinou operačných systémov, vrátane Linuxu, MacOS a Windows. V nasej demo aplikácii využívame 4 kontajnery a tie sú konkrétne dispatching, plane-1, plane-2 a plane-3.

Zoznam použitej literatúry

1. NELSON, Bruce Jay. Remote procedure call. In: 1981.
2. *What is Remote Procedure Call?* [B.r.]. Dostupné tiež z: `https://www.w3.org/History/1992/nfs_dxcern_mirror/rpc/doc/Introduction/WhatIs.html`.
3. *Core Concepts, architecture and Lifecycle*. 2022. Dostupné tiež z: `https://grpc.io/docs/what-is-grpc/core-concepts/`.
4. BANSAL, Anshul. *Rest vs. gRPC*. 2021. Dostupné tiež z: `https://www.baeldung.com/rest-vs-grpc`.
5. *Docker documentation*. 2022. Dostupné tiež z: `https://docs.docker.com/`.