Introduction to gRPC with Protocol Buffers

Efficient Communication with Modern APIs

What is gRPC?

- - gRPC is a high-performance RPC (Remote Procedure Call) framework
- - Developed by Google; uses HTTP/2 for transport
- - Uses Protocol Buffers (.proto) for message serialization
- - Supports multiple languages and bi-directional streaming

Why Use gRPC?

- - Strongly-typed contracts via .proto files
- - High performance and low latency
- - Supports streaming and multiplexing via HTTP/2
- - Built-in code generation for clients and servers

Key Advantages

- **Performance**: HTTP/2 multiplexing, header compression, binary protobuf serialization.
- **Strongly Typed Contracts**: Protobuf ensures clear and consistent service definitions.
- **Code Generation**: Protobuf compiler generates client and server stubs in various languages.
- **Streaming Capabilities**: Supports both unary (request/response) and streaming (multiple messages) communication.
- **Built-in Features**: Authentication, authorization, tracing, health checks.

Protocol Buffers (Protobuf)

- What is Protobuf?
  - Language-neutral, platform-neutral, extensible mechanism for serializing structured data.
  - Data is defined in .proto files.
  - The protobuf compiler (protoc) generates code in your chosen language.
- **Key Concepts:**
  - **Messages:** Define the structure of the data being exchanged (fields with types and unique numbers).
  - **Services:** Define the available RPC methods (name, request type, response type).
  - **Data Types:** Supports various scalar types (int32, string, bool, etc.) and complex types (messages, enums, etc.).

Components of .protofile

- **syntax**: Declares the version of the Protocol Buffers language you are using. proto3 is the current, recommended version.
- **package**: Acts like a C++ namespace or a Java package. It prevents name clashes between different .proto files.
- **option**: Provides instructions to the compiler for specific languages. This is a powerful feature for multi-language environments.
- **import**: Lets you re-use definitions from other .proto files, promoting modularity.

A message type

-
  ```
  message  User {
     // Field Definition: // [Type] [field_name] =   [Field_Number];
      string user_id = 1;
      string username = 2;
       bool is_active = 3;
     }
  ```

Protocol Buffers

- **Type**: The data type of the field. Can be a scalar type (string, int32, float, bool, bytes) or a complex type (another message, an enum).
- **field_name:** The name of the field. The convention is snake_case.
- **Field_Number:** This is the most important concept. It's a unique number used to identify the field in the binary wire format.
- Numbers 1-15 take only one byte to encode and should be reserved for your most frequently used fields.
- Once your API is in use, you must NEVER change or reuse a field number. This is the key to backward compatibility.

```
•   syntax = "proto3";

•   package greeter;

•   service Greeter {
•     rpc SayHello (HelloRequest) returns (HelloReply);
•   }

•   message HelloRequest {
•     string name = 1;
•   }

•   message HelloReply {
•     string message = 1;
•   }
```

Understanding Protocol Buffers (.proto)

- - Defines the structure of the messages and services
- - Used by gRPC to auto-generate code
- - Example syntax:

- syntax = "proto3";
- service BookService {
-   rpc GetBook (BookRequest) returns (BookResponse);
- }
- message BookRequest {
-   string book_id = 1;
- }
- message BookResponse {
-   string title = 1;
-   string author = 2;
- }

Code Generation with protoc

- - The `protoc` compiler generates code from .proto files
- - Example:
- protoc --python_out=. --grpc_python_out=. book.proto
- - Generated files: book_pb2.py, book_pb2_grpc.py
- - Can be used to implement server and client logic

Hands-On Activity: gRPC in Action

- 1. Create a simple .proto file for BookService
- 2. Generate code using protoc
- 3. Implement a gRPC server with sample data
- 4. Write a Python client to call GetBook
- 5. Run both and test communication
- 6. Observe the speed and type safety in gRPC

```
const grpc = require("@grpc/grpc-js");
const protoLoader = require("@grpc/proto-loader");

const packageDef = protoLoader.loadSync("book.proto");
const grpcObject = grpc.loadPackageDefinition(packageDef);
const bookPackage = grpcObject.BookService;

const books = {
  "1": { title: "Clean Code", author: "Robert C. Martin" },
  "2": { title: "The Pragmatic Programmer", author: "Andy Hunt" }
};

function getBook(call, callback) {
  const book = books[call.request.book_id] || {};
  callback(null, book);
}

const server = new grpc.Server();
server.addService(bookPackage.service, { GetBook: getBook });
server.bindAsync("0.0.0.0:50051", grpc.ServerCredentials.createInsecure(), () => {
  console.log("Server running at http://localhost:50051");
  server.start();
});
```

```
const grpc = require("@grpc/grpc-js");
const protoLoader = require("@grpc/proto-loader");

const packageDef = protoLoader.loadSync("book.proto");
const grpcObject = grpc.loadPackageDefinition(packageDef);
const client = new grpcObject.BookService("localhost:50051",
grpc.credentials.createInsecure());

client.GetBook({ book_id: "1" }, (err, response) => {
  if (err) console.error(err);
  else console.log("Book Info:", response);
});
```