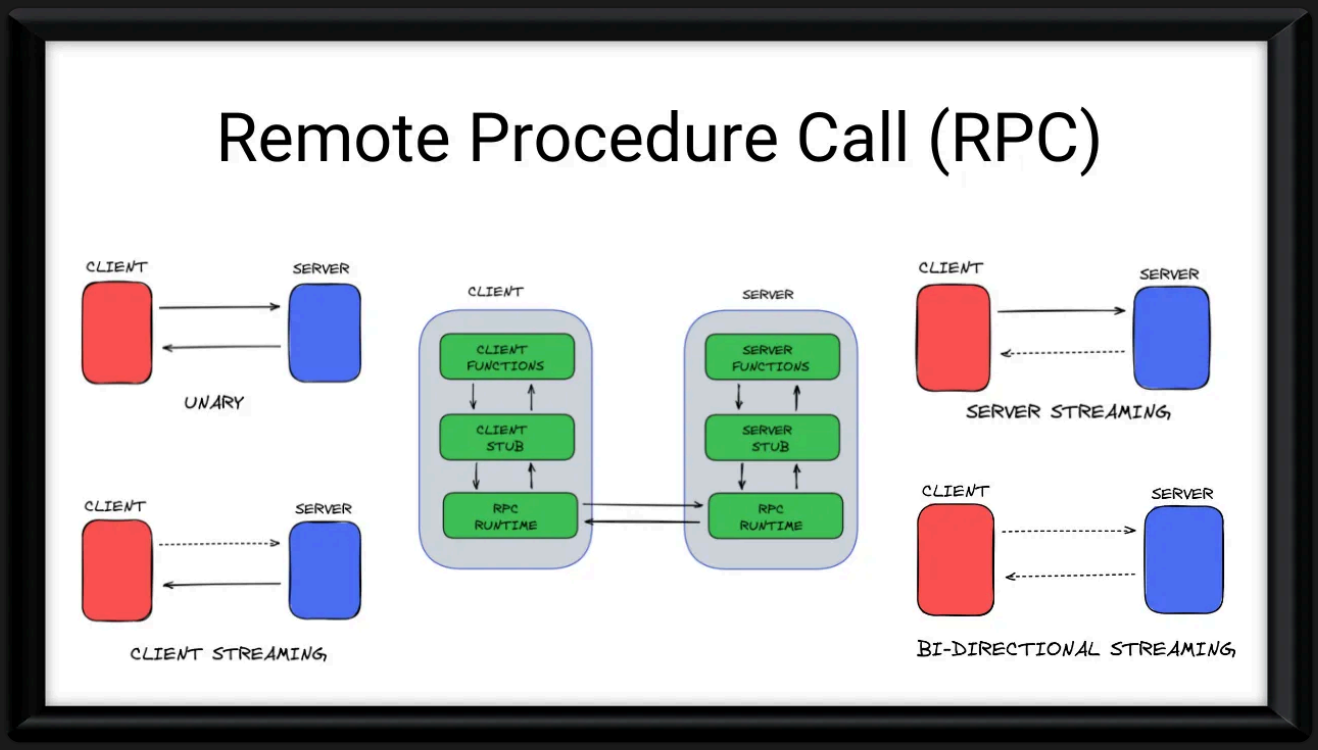
Week 24.1

In this lecture, Harkirat covers `Remote Procedure Calls` (RPC), a powerful technique for enabling communication between distributed systems. He explains the need for RPC and its advantages over traditional communication methods. Harkirat demonstrates how to `Implement a simple RPC in TypeScript`, showcasing the concept of `auto-generated clients`. He then introduces `Protocol Buffers`, a language-neutral, platform-neutral, extensible mechanism for serializing structured data, and its role in RPC. He then explores common `RPC models, including JSON-RPC, tRPC, and gRPC`, highlighting their differences and use cases. Additionally, he delves into defining types in Protocol Buffers and `Implementing services using gRPC`, incorporating type safety and code generation.

Understanding RPC

client, making it appear as if the procedure is being executed locally.



Why use RPC?

1. **Language and Platform Independence**: RPC allows applications written in different programming languages and running on different platforms to communicate seamlessly. This promotes code reusability and interoperability across diverse systems.
2. **Abstraction of Communication Details**: RPC abstracts away the complexities of network communication, such as socket programming, serialization, and deserialization of data. Developers can focus on the application logic rather than low-level communication details.
3. **Distributed Computing**: RPC enables the distribution of computational tasks across multiple systems, allowing for better resource utilization, load balancing, and scalability.
4. **Code Modularity**: RPC promotes code modularity by separating the client and server components. This separation of concerns makes the codebase more maintainable and easier to evolve.
5. **Efficiency**: RPC can be more efficient than traditional HTTP-based communication, especially for tightly coupled systems or high-performance scenarios, as it avoids the overhead of HTTP headers and parsing.

Drawbacks of HTTP for Backend Communication

1. **Lack of Type Safety**: HTTP requests and responses are typically transmitted as plain text or JSON, which lacks type safety. This can lead to runtime errors and make it harder to ensure data integrity
2. **Overhead**: HTTP has additional overhead due to headers, parsing, and serialization/deserialization of data, which can impact performance, especially in high-throughput scenarios.
3. **Language Dependency**: HTTP libraries and their usage can vary across programming languages, making it harder to maintain consistent communication patterns across different backend systems.
4. **Limited Functionality**: HTTP is primarily designed for request-response communication, which may not be suitable for more complex scenarios like bi-directional streaming or long-lived connections.

RPC addresses these limitations by providing a more efficient, language-agnostic, and type-safe communication mechanism for backend systems.

To illustrate RPC in action, let's consider making a request to `https://sum-server.100xdevs.com/todos` from a Node.js server using the built-in `http` module:

```
const https = require('https');

const options = {
  hostname: 'sum-server.100xdevs.com',
  port: 443,
  path: '/todos',
  method: 'GET'
};

const req = https.request(options, (res) => {
  console.log(`Status Code: ${res.statusCode}`);

  res.on('data', (chunk) => {
    console.log(`Body: ${chunk}`);
  });

  res.on('end', () => {
    console.log('No more data in response.');
  });
```

```
  console.error(`problem with request: ${e.message}`);
});

req.end();
```

This code sends an HTTP GET request to the specified URL and logs the response status code and body to the console. While this approach works, it requires handling low-level details like creating the request options, managing the response data, and handling errors explicitly.

With RPC, the communication between the client and server would be abstracted away, allowing developers to focus on the application logic rather than the underlying communication details.

Implementing a Simple RPC

The idea behind implementing a simple RPC is to generate client code that can be used by different programming languages to call functions on a remote service without worrying about the underlying communication details, such as making HTTP requests or handling serialization/deserialization.

Autogenerated Client

Let's consider an autogenerated client in TypeScript that can fetch a list of todos from a remote service:

```
// rpc.ts (autogenerated)
import axios from "axios";

interface Todo {
  id: string;
  title: string;
  description: string;
  completed: boolean;
}
```

```
  let todos = response.data.todos;
  return todos;
}
```

In this example, the `getTodos` function is autogenerated and uses the `axios` library to make an HTTP GET request to the specified URL. The function returns a `Promise` that resolves with an array of `Todo` objects, where the `Todo` interface is also autogenerated based on the expected response shape.

To use this autogenerated client, you can import the `getTodos` function and call it like this:

```
// index.ts
import { getTodos } from "./rpc";

const todos = await getTodos();
console.log(todos);
```

Benefits of the Autogenerated Client

1. **Better Type Safety:** The `getTodos` function has an associated type for the data being returned (`Todo[]`), which provides better type safety and helps catch errors during development.
2. **Abstraction of Communication Details:** Developers no longer need to worry about using libraries like `axios` or `fetch` directly. They can simply call the `getTodos` function, which abstracts away the underlying communication details.
3. **Language Agnostic:** By autogenerating clients for different programming languages, this approach becomes language-agnostic, allowing backend systems written in different languages to communicate seamlessly.

Sample Clients in Other Languages

To illustrate the language-agnostic nature of this approach, here are sample clients for the same `getTodos` function in Rust and Go:

Rust:

```
#[derive(Debug)]
struct Todo {
    id: String,
    title: String,
    description: String,
    completed: bool,
}

async fn get_todos() -> Result<Vec<Todo>, Error> {
    let response = reqwest::get("<https://sum-server.100xdevs.c

    let todos: Vec<Todo> = response.json().await?;
    Ok(todos)
}
```

Go:

```
import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
)

type Todo struct {
    ID          string `json:"id"`
    Title       string `json:"title"`
    Description string `json:"description"`
    Completed   bool   `json:"completed"`
}

func getTodos() ([]Todo, error) {
    response, err := http.Get("<https://sum-server.100xdevs.com
    if err != nil {
        return nil, err
    }
    defer response.Body.Close()
```

```
    }

    var todos struct {
        Todos []Todo `json:"todos"`
    }
    if err := json.Unmarshal(body, &todos); err != nil {
        return nil, err
    }

    return todos.Todos, nil
}
```

These examples demonstrate how the autogenerated client can be used in different programming languages, providing a consistent and language-agnostic way to communicate with the remote service.

> While this approach is a step towards a more efficient and type-safe communication mechanism, it still relies on JSON for serialization/deserialization, which can be slow compared to other formats like Protocol Buffers or gRPC. In the next section, we'll explore how to improve performance by using more efficient serialization formats.

Protocol Buffers

Protocol Buffers (Protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. They provide an efficient and type-safe way to serialize and deserialize data, making them a popular choice for communication between different systems or programming languages.

Defining the Schema

Protocol Buffers use a schema definition language (`.proto` files) to define the structure of data. Here's an example of a simple `.proto` file:

```
// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
}

service PersonService {
  // Add a person to the address book.
  rpc AddPerson(Person) returns (Person);

  // Get a person from their name
  rpc GetPersonByName(GetPersonByNameRequest) returns (Person);
}

message GetPersonByNameRequest {
  string name = 1;
}
```

In this example, we define a `Person` message type with two fields: `name` (string) and `age` (int32). We also define a `PersonService` with two remote procedure calls (RPCs): `AddPerson` and `GetPersonByName`. The `GetPersonByNameRequest` message is used as the input for the `GetPersonByName` RPC.

Field Numbers

Each field within a message type is assigned a unique numerical identifier called a field number or tag. These field numbers serve several purposes:

1. **Efficient Encoding**: Field numbers are used during serialization and deserialization to efficiently encode and decode the data. Instead of including field names in the serialized data, Protocol Buffers use field numbers, which are typically more compact and faster to process.
2. **Backward Compatibility**: Field numbers are stable identifiers that remain consistent even if you add, remove, or reorder fields within a message type. This means that old serialized data can still be decoded correctly by newer versions of your software, even if the message type has changed.
3. **Language Independence**: Field numbers provide a language-independent way to refer to fields within a message type. Regardless of the programming language used to

Serializing and Deserializing Data

Protocol Buffers provide a binary serialization format that is more compact and efficient compared to text-based formats like XML and JSON. Here's an example of how to serialize and deserialize data using the `protobufjs` library in Node.js:

```
const protobuf = require('protobufjs');

// Load the Protocol Buffers schema
protobuf.load('a.proto')
  .then(root => {
    // Obtain the Person message type
    const Person = root.lookupType('Person');

    // Create a new Person instance
    const person = { name: "Alice", age: 30 };

    // Serialize Person to a buffer
    const buffer = Person.encode(person).finish();

    // Write buffer to a file
    require('fs').writeFileSync('person.bin', buffer);

    console.log('Person serialized and saved to person.bin');

    // Read the buffer from file
    const data = require('fs').readFileSync('person.bin');

    // Deserialize buffer back to a Person object
    const deserializedPerson = Person.decode(data);

    console.log('Person deserialized from person.bin:', deseria
  })
  .catch(console.error);
```

In this example, we load the `a.proto` schema, obtain the `Person` message type, create a `Person` instance, and serialize it to a binary buffer using

Size Comparison

One of the advantages of Protocol Buffers is their compact binary serialization format, which results in smaller data sizes compared to text-based formats like JSON. Let's compare the size of the serialized `Person` data with a JSON representation:

```
{
  "name": "Alice",
  "age": 31
}
```

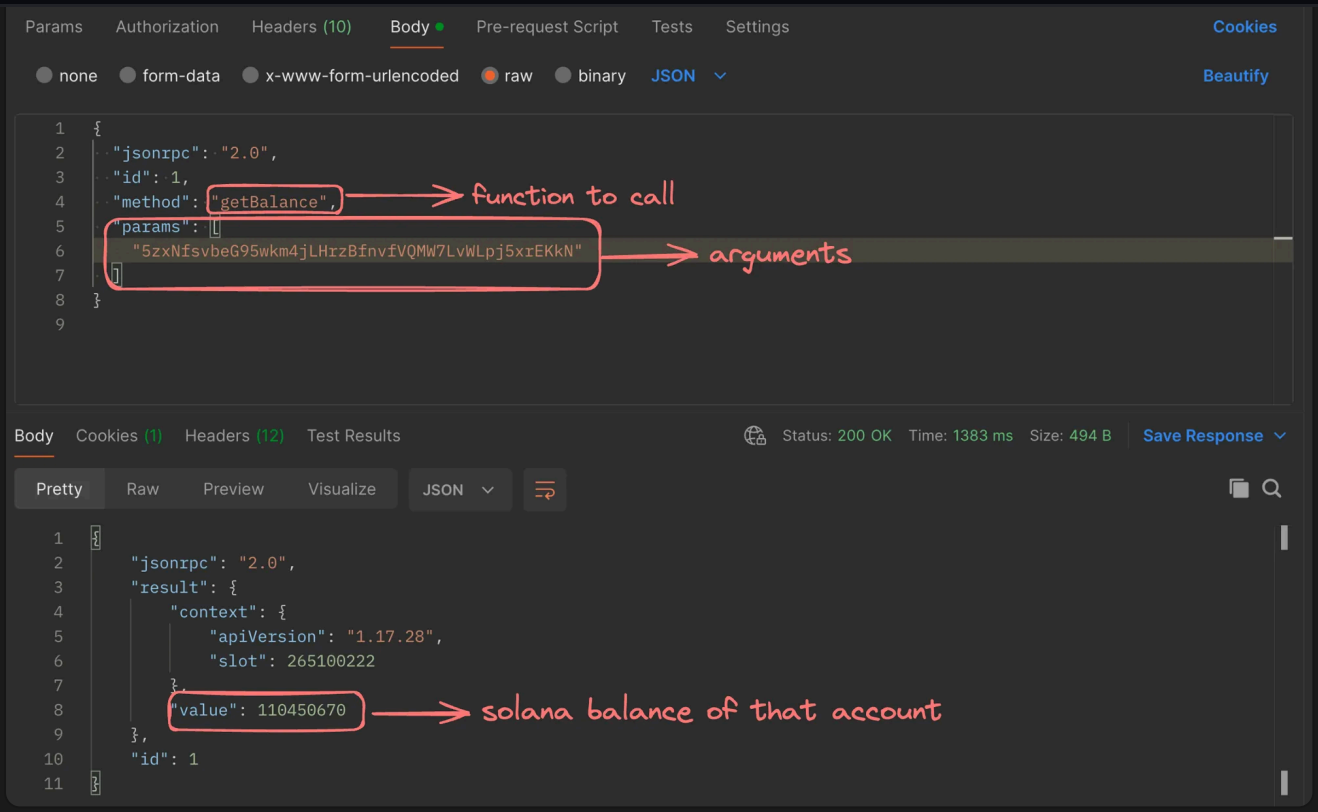
The size of the `person.bin` file (serialized with Protocol Buffers) is typically smaller than the JSON representation, as shown in the provided images.

Some Common RPC Protocols

There are several RPC protocols available, each with its own strengths and use cases. Here are some common RPC protocols:

JSON-RPC

JSON-RPC is a lightweight remote procedure call protocol that uses JSON as the data format for requests and responses. It is widely used in various applications, including blockchain platforms like Ethereum and Solana.



Creating a JSON-RPC Server

Here's an example of creating a simple JSON-RPC server using Express.js in Node.js:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;

// Parse JSON bodies
app.use(bodyParser.json());

// Define a sample method
function add(a, b) {
    return a + b;
}

// Handle JSON-RPC requests
app.post('/rpc', (req, res) => {
    const { jsonrpc, method, params, id } = req.body;
```

```
    }

    // Execute the method
    let result;
    switch (method) {
        case 'add':
            result = add(params[0], params[1]);
            break;
        default:
            res.status(404).json({ jsonrpc: '2.0', error: { cod
            return;
    }

    // Send back the response
    res.json({ jsonrpc: '2.0', result, id });
});

// Start the server
app.listen(port, () => {
    console.log(`JSON-RPC server listening at <http://localhost
});
```

In this example, we define an `add` function and handle JSON-RPC requests at the `/rpc` endpoint. The server expects a JSON-RPC request body with the `jsonrpc` version, `method` name, `params` array, and an `id`. If the request is valid, the server executes the corresponding method and sends back the result.

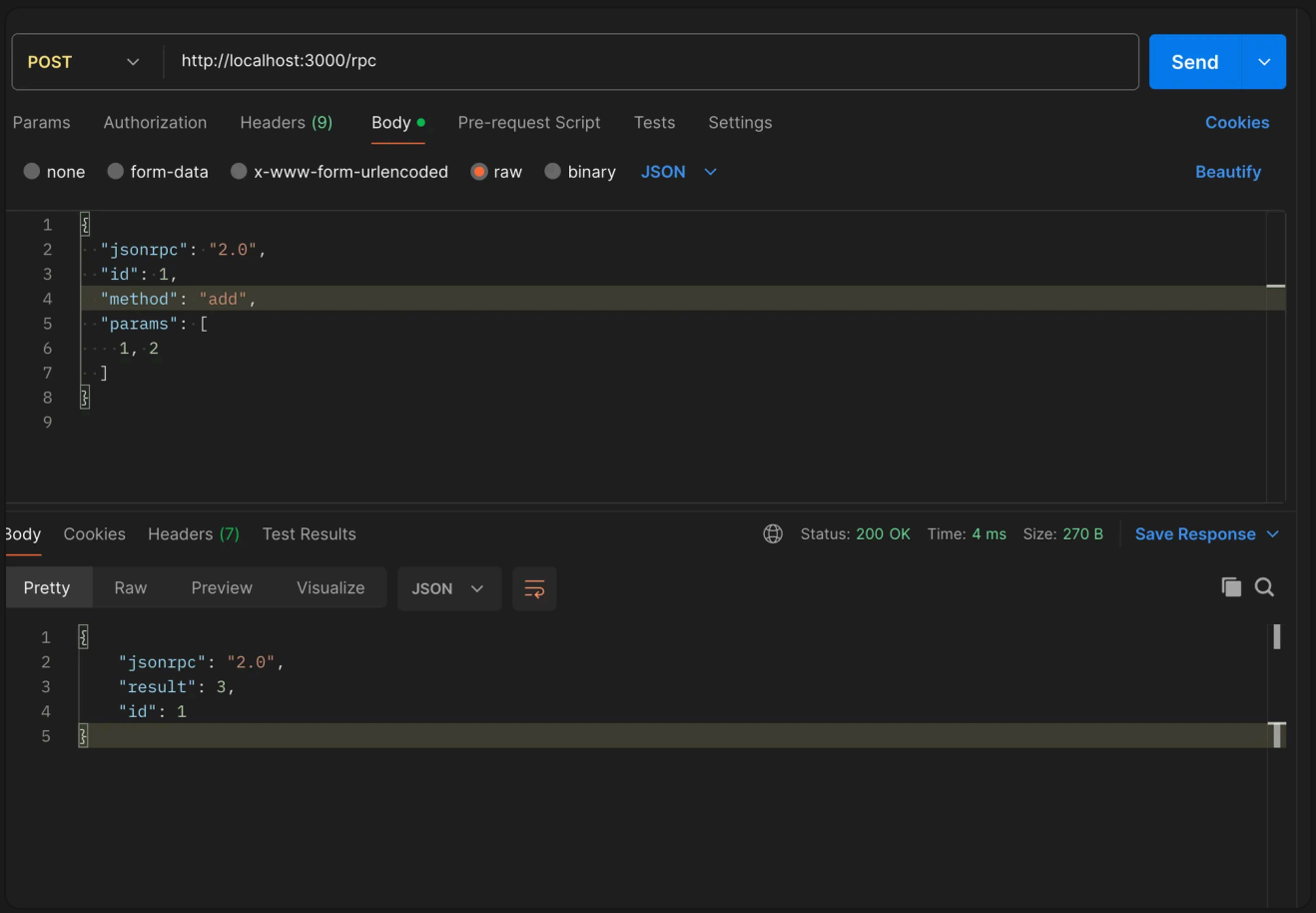
To test the server, you can send a JSON-RPC request like this:

```
  "id": 1,
  "method": "add",
  "params": [
    1, 2
  ]
}
```

The server should respond with the result:

```
{
  "jsonrpc": "2.0",
  "result": 3,
  "id": 1
}
```

JSON-RPC is a simple and lightweight protocol, making it suitable for various use cases, including blockchain interactions and general-purpose RPC communication.

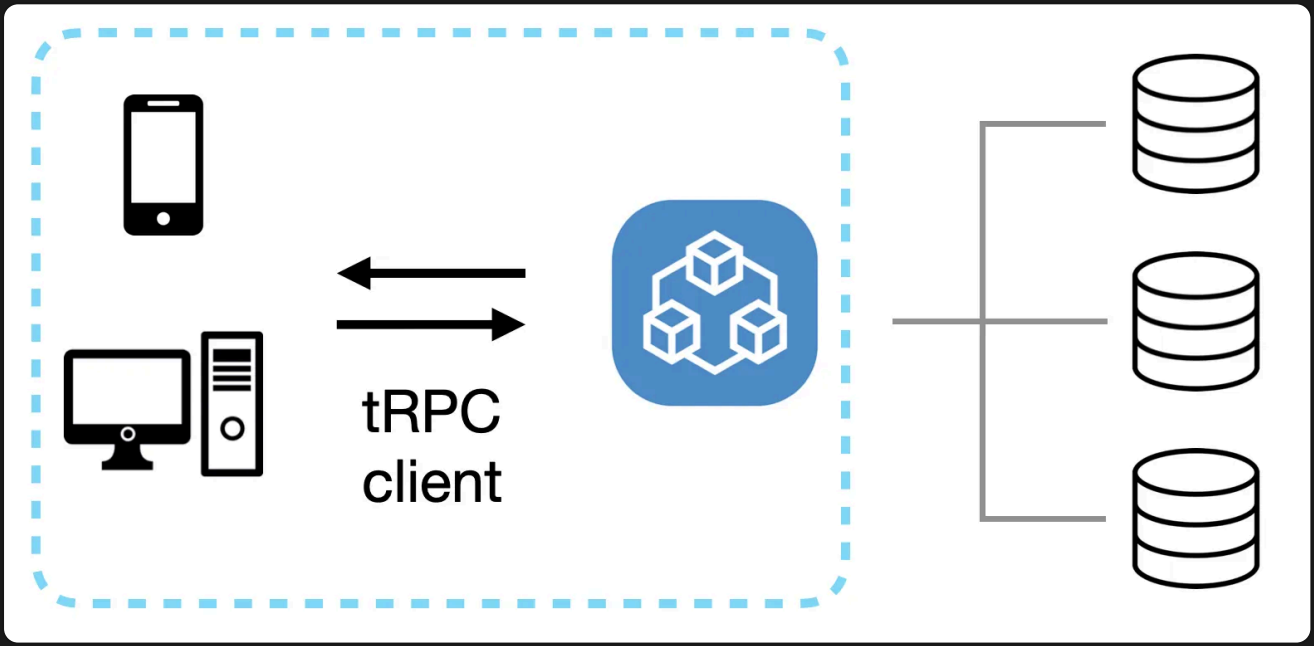


Google. It uses Protocol Buffers for efficient data serialization and provides features like streaming, load balancing, and authentication.

gRPC is widely used in microservices architectures and high-performance distributed systems due to its efficiency and language support. It generates client and server code in various programming languages based on the defined service definitions in Protocol Buffers.

tRPC

tRPC (TypeScript RPC) is a framework for building end-to-end type-safe APIs in TypeScript. It is designed for full-stack JavaScript/TypeScript applications and provides type safety on both the frontend and backend.



tRPC allows you to define your API routes and data types on the server-side, and it automatically generates TypeScript types for the client-side. This ensures that your client code is always in sync with the server API, reducing the risk of runtime errors and improving developer productivity.

tRPC is particularly useful for building full-stack applications with a shared codebase between the frontend and backend, as it eliminates the need for separate API documentation and client-side type definitions.

Types in Protocol Buffers

Scalar Types

Scalar types are the basic data types in Protocol Buffers. They include:

- `int32`, `int64`, `uint32`, `uint64`: Signed and unsigned integers of various sizes.
- `float`, `double`: Floating-point numbers.
- `bool`: Boolean values (`true` or `false`).
- `string`: Unicode text strings.
- `bytes`: Arbitrary binary data.

Here's an example of using scalar types in a `.proto` file:

```
syntax = "proto3";

// Define a message type representing an address.
message Address {
  string street = 1;
  string city = 2;
  string state = 3;
  string zip = 4;
}

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
  Address address = 3;
}
```

In this example, the `Address` message type contains four `string` fields, and the `Person` message type contains a `string` field for the name, an `int32` field for the age, and an `Address` message field.

Message Types

```
message Person {
  string name = 1;
  int32 age = 2;
  repeated string phone_numbers = 3;
}
```

In this example, the `Person` message type has a `name` field of type `string`, an `age` field of type `int32`, and a repeated field `phone_numbers` of type `string` to store multiple phone numbers.

Enum Types

Enum types define a set of named constant values. You define enum types using the `enum` keyword followed by the name of the enum type and its values.

```
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}
```

In this example, the `PhoneType` enum defines three values: `MOBILE`, `HOME`, and `WORK`.

Maps

Protocol Buffers also support map types, which are associative collections that map keys to values. You define map types using the `map` keyword followed by the key and value types.

```
message MapMessage {
  map<string, int32> id_to_age = 1;
}
```

Combining Types

You can combine these different types to create more complex message structures. Here's an example that combines scalar types, message types, enum types, and repeated fields:

```
syntax = "proto3";

// Define an enum representing the type of phone numbers.
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

// Define a message type representing a phone number.
message PhoneNumber {
  string number = 1;
  PhoneType type = 2;
}

// Define a message type representing an address.
message Address {
  string street = 1;
  string city = 2;
  string state = 3;
  string zip = 4;
}

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
  repeated PhoneNumber phone_numbers = 3;
  Address address = 4;
}
```

In this example, the `Person` message type contains a `name` field of type `string`, an `age` field of type `int32`, a repeated field `phone_numbers` of type `PhoneNumber` (which itself

> your data, and then use the generated code to create, serialize, and deserialize instances of these message types in your application.

Implementing Services

In Protocol Buffers, the `service` section defines the interface for the remote procedure calls (RPCs) that a server can handle. However, Protocol Buffers itself does not provide an implementation for these services. Instead, it relies on other RPC frameworks or custom implementations to handle the actual communication and execution of the defined services.

Here's an example of a `.proto` file that defines a service:

```
syntax = "proto3";

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
}

service AddressBookService {
  // Add a person to the address book.
  rpc AddPerson(Person) returns (Person);

  // Get a person from their name
  rpc GetPersonByName(string) returns (Person);
}
```

In this example, the `AddressBookService` defines two RPCs: `AddPerson` and `GetPersonByName`. The `AddPerson` RPC takes a `Person` message as input and returns a `Person` message as output, while the `GetPersonByName` RPC takes a `string` (representing the person's name) as input and returns a `Person` message as output.

Implementing Services Using gRPC

gRPC (Google Remote Procedure Call) is a popular framework for implementing services defined in Protocol Buffers. It provides a high-performance, efficient, and language-agnostic way to build distributed systems and microservices. Here's an example of how to implement services using gRPC in Node.js with TypeScript.

Setup

1. Initialize a new Node.js project:

```
npm init -y
```

1. Initialize TypeScript:

```
npx tsc --init
```

1. Install the required dependencies:

```
npm i @grpc/grpc-js @grpc/proto-loader
```

Define the Protocol Buffers File

Create a file named `a.proto` and define the service and message types:

```
syntax = "proto3";

// Define a message type representing a person.
message Person {
  string name = 1;
  int32 age = 2;
}

service AddressBookService {
  // Add a person to the address book.
```

```
  rpc GetPersonByName(GetPersonByNameRequest) returns (Person);
}

message GetPersonByNameRequest {
  string name = 1;
}
```

Implement the Server

Create a file named `index.ts` and implement the server:

```
import path from 'path';
import * as grpc from '@grpc/grpc-js';
import { GrpcObject, ServiceClientConstructor } from "@grpc/grp
import * as protoLoader from '@grpc/proto-loader';

const packageDefinition = protoLoader.loadSync(path.join(__dirn

const personProto = grpc.loadPackageDefinition(packageDefinitio

const PERSONS = [
    {
        name: "harkirat",
        age: 45
    },
    {
      name: "raman",
      age: 45
    },
];

//@ts-ignore
function addPerson(call, callback) {
  console.log(call)
    let person = {
      name: call.request.name,
      age: call.request.age
    }
```

```
const server = new grpc.Server();

server.addService((personProto.AddressBookService as ServiceCli
server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.create
    server.start();
});
```

Here's what's happening in the code:

1. We load the Protocol Buffers schema using `protoLoader.loadSync` and `grpc.loadPackageDefinition`.
2. We define a simple in-memory array `PERSONS` to store person objects.
3. We implement the `addPerson` function, which handles the `AddPerson` RPC. It takes the request data from `call.request`, creates a new person object, adds it to the `PERSONS` array, and sends the new person object back as the response using the `callback` function.
4. We create a new gRPC server instance using `new grpc.Server()`.
5. We add the `AddressBookService` to the server using `server.addService`, providing the implementation for the `addPerson` method.
6. We bind the server to listen on `0.0.0.0:50051` using `server.bindAsync` and start the server using `server.start()`.

Run the Server

1. Compile the TypeScript code:

```
tsc -b
```

1. Run the server:

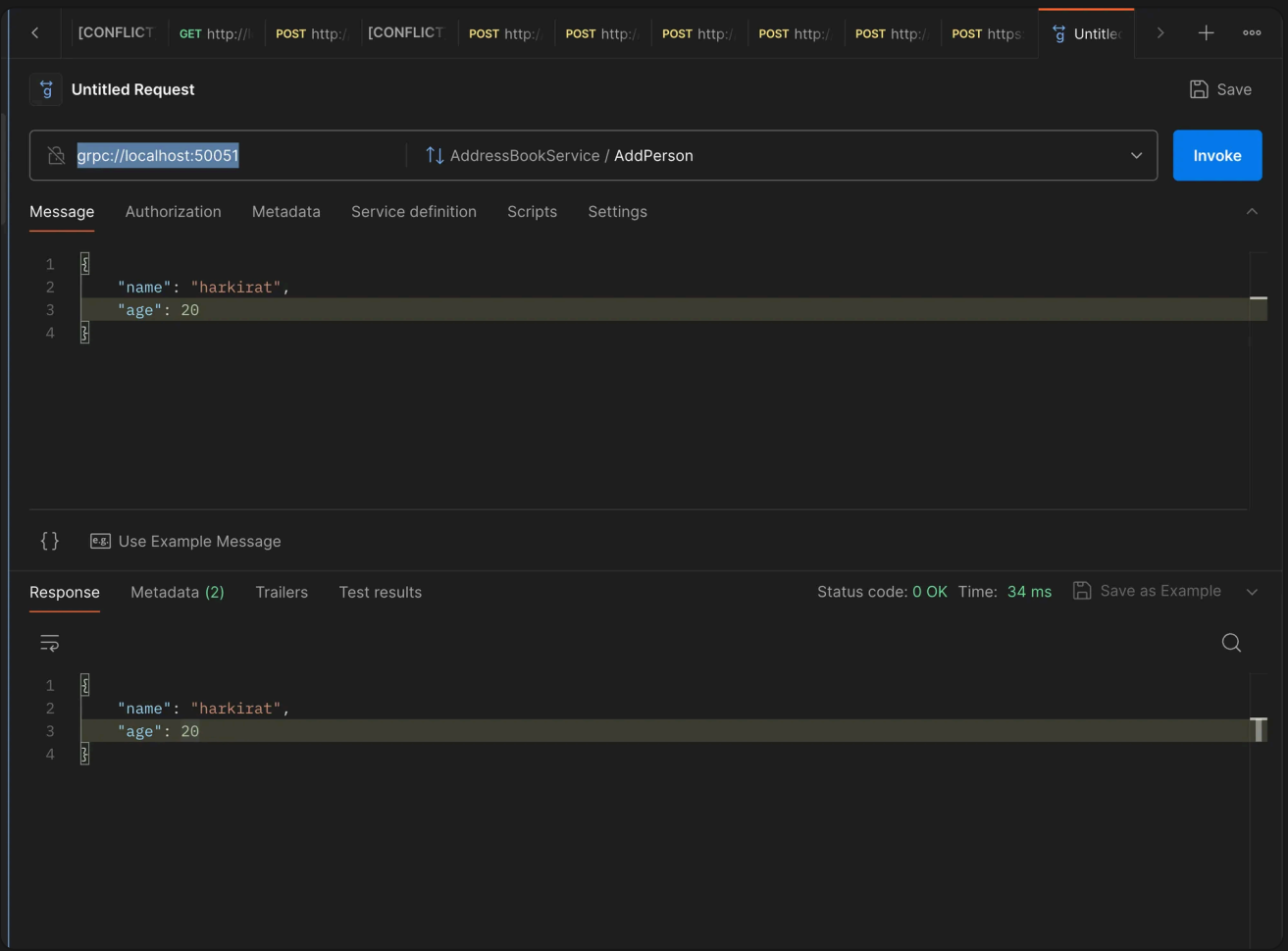
```
node index.js
```

Test the Server

You can test the server using a gRPC client like Postman or BloomRPC. Here's how to test it using Postman:

1. Open Postman and create a new gRPC request.
2. Import the `a.proto` file by going to `File` > `New` > `GRPC` and selecting the `a.proto` file.
3. Set the URL to `grpc://localhost:50051`.
4. Select the `AddressBookService` service and the `AddPerson` method.
5. Fill in the request data with a `Person` object, e.g., `{ "name": "Alice", "age": 30 }`.
6. Send the request.

You should see the response containing the `Person` object you added.



This example demonstrates how to implement a simple gRPC server in Node.js using TypeScript. You can extend this example to implement additional RPCs, handle errors, and

Adding Types

To improve type safety and developer experience when working with gRPC in Node.js, we can generate TypeScript types from our Protocol Buffers definition file (`.proto`). This allows us to leverage the benefits of static typing and autocompletion in our code.

Example Usage

Generate the types:

```
$(npm bin)/proto-loader-gen-types --longs=String --enums=String --defaults
```

Consume the types:

```
import * as grpc from '@grpc/grpc-js';
import * as protoLoader from '@grpc/proto-loader';
import type { ProtoGrpcType } from './proto/example.ts';
import type { ExampleHandlers } from './proto/example_package/Example.ts';

const exampleServer: ExampleHandlers = {
  // server handlers implementation...
};

const packageDefinition = protoLoader.loadSync('./proto/example.proto');
const proto = (grpc.loadPackageDefinition(
  packageDefinition
) as unknown) as ProtoGrpcType;

const server = new grpc.Server();
server.addService(proto.example_package.Example.service, exampleServer);
```

Generating Types

The `@grpc/proto-loader` package provides a command-line tool called `proto-loader-gen-types` that can generate TypeScript types from a `.proto` file. Here's how you can use it:

1. Install the `@grpc/proto-loader` package if you haven't already:

1. Run the `proto-loader-gen-types` command, providing the necessary options and the path to your `.proto` file:

```
./node_modules/@grpc/proto-loader/build/bin/proto-loader-gen-ty
```

This command generates TypeScript types for the messages, enums, and services defined in the `a.proto` file and saves them in the `generated` directory.

Updating the Code

After generating the types, you can update your code to use them:

```
import path from 'path';
import * as grpc from '@grpc/grpc-js';
import { GrpcObject, ServiceClientConstructor } from "@grpc/grp
import * as protoLoader from '@grpc/proto-loader';
import { ProtoGrpcType } from './proto/a';
import { AddressBookServiceHandlers } from './proto/AddressBook
import { Status } from '@grpc/grpc-js/build/src/constants';

const packageDefinition = protoLoader.loadSync(path.join(__dirn

const personProto = (grpc.loadPackageDefinition(packageDefiniti

const PERSONS = [
    {
        name: "harkirat",
        age: 45
    },
    {
      name: "raman",
      age: 45
    },
];

const handler: AddressBookServiceHandlers = {
```

```
      age: call.request.age
    }
    PERSONS.push(person);
    callback(null, person)
  },
  GetPersonByName: (call, callback) => {
    let person = PERSONS.find(x => x.name === call.request.name
    if (person) {
      callback(null, person)
    } else {
      callback({
        code: Status.NOT_FOUND,
        details: "not found"
      }, null);
    }
  }
}

const server = new grpc.Server();

server.addService((personProto.AddressBookService).service, han
server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.create
    server.start();
});
```

Comment

In this updated code:

1. We import the generated types `ProtoGrpcType` and `AddressBookServiceHandlers` from the `./proto/a` and `./proto/AddressBookService` modules, respectively.
2. We cast the result of `grpc.loadPackageDefinition` to `ProtoGrpcType` to get access to the generated types.
3. We define the `handler` object as an instance of `AddressBookServiceHandlers`, which ensures that the `AddPerson` and `GetPersonByName` methods have the correct signatures and types.

By using the generated types, you get the benefits of type safety, autocompletion, and better tooling support in your gRPC implementation. This can help catch errors during development and improve the overall maintainability of your code.