Setting up gRPC with React

Prerequisites:

- Node.js and npm installed on your system.
- Basic knowledge of React and gRPC.

Step 1: Create a React Project

- You can create one using create-react-app:

```
npx create-react-app grpc-react-app
cd grpc-react-app
```

Step 2: Install gRPC Dependencies

- To use gRPC in a React project, you need to install the necessary packages:

```
npm install grpc-web

npm install google-protobuf
```

Step 3: Define Your gRPC Service

- Create a `.proto` file to define your gRPC service and messages. For example, example.proto:

```
syntax = "proto3";

package yourpackage;

service YourService {

  rpc YourMethod (YourRequest) returns (YourResponse);

}

message YourRequest {

  string input = 1;

}

message YourResponse {

  string output = 1;

}
```

Step 4: Generate gRPC Web Code

- Use the protoc compiler to generate the gRPC web code:

```
protoc --proto_path=proto --js_out=import_style=commonjs,binary:proto --grpc-
web_out=import_style=commonjs,mode=grpcwebtext:proto proto/example.proto
```

Step 5: Create a gRPC Service Client

- Create a gRPC service client in your React component. Import the generated service and messages:

```
import { YourServiceClient } from './your_generated_code_pb_service';

import { YourRequest, YourResponse } from './your_generated_code_pb';

// Use the Envoy Proxy address

const client = new YourServiceClient('http://localhost:8080');
```

Step 6: Make gRPC Calls

- Now, you can use the gRPC client to make calls to your server:

```
const request = new YourRequest();
request.setInput('Hello, gRPC!');

client.yourMethod(request, {}, (error, response) => {
  if (!error) {
    console.log('Response:', response.getOutput());
  } else {
    console.error('Error:', error);
  }
});
```

To complete your gRPC with React setup, you can also use Docker and Envoy Proxy to handle CORS (Cross-Origin Resource Sharing) issues. Here's how to add Docker configuration and Envoy Proxy for CORS:

Step 1: Envoy Proxy for CORS

- **Create an Envoy Configuration File**: Create an Envoy configuration file (envoy.yaml) to set up CORS. Here's a basic example:

```
admin:
  access_log_path: /tmp/admin_access.log
  address:
    socket_address: { address: 0.0.0.0, port_value: 9901 }

static_resources:
  listeners:
    - name: listener_0
      address:
        socket_address: { address: 0.0.0.0, port_value: 8080 } #your envoy port
      filter_chains:
        - filters:
          - name: envoy.filters.network.http_connection_manager
            typed_config:
              "@type":
type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v3.HttpConnect
ionManager
              codec_type: auto
              stat_prefix: ingress_http
              route_config:
                name: local_route
                virtual_hosts:
                  - name: local_service
                    domains: ["*"]
                    routes:
                      - match: { prefix: "/" }
                        route:
                          cluster: echo_service
                          timeout: 0s
                          max_stream_duration:
                            grpc_timeout_header_max: 0s
                    cors:
```

```
                    allow_origin_string_match:
                      - prefix: "*"
                    allow_methods: GET, PUT, DELETE, POST, OPTIONS
                    allow_headers: keep-alive,user-agent,cache-control,content-
type,content-transfer-encoding,custom-header-1,x-accept-content-transfer-encoding,x-accept-
response-streaming,x-user-agent,x-grpc-web,grpc-timeout
                    max_age: "1728000"
                    expose_headers: custom-header-1,grpc-status,grpc-message
            http_filters:
              - name: envoy.filters.http.grpc_web
                typed_config:
                  "@type":
type.googleapis.com/envoy.extensions.filters.http.grpc_web.v3.GrpcWeb
              - name: envoy.filters.http.cors
                typed_config:
                  "@type": type.googleapis.com/envoy.extensions.filters.http.cors.v3.Cors
              - name: envoy.filters.http.router
                typed_config:
                  "@type":
type.googleapis.com/envoy.extensions.filters.http.router.v3.Router
  clusters:
    - name: echo_service
      connect_timeout: 0.25s
      type: logical_dns
      http2_protocol_options: {}
      lb_policy: round_robin
      load_assignment:
        cluster_name: cluster_0
        endpoints:
          - lb_endpoints:
            - endpoint:
                address:
                  socket_address:
                    address: host.docker.internal
                    port_value: 11912  # Your gRPC server's port
```

Step 2: Dockerize the Envoy Proxy

- You can also create a Docker container for the Envoy Proxy and run it alongside your React app in a Docker Compose configuration for easier management. Here's a basic example of an Envoy Dockerfile:

```
version: "3"
services:
  envoy:
    image: envoyproxy/envoy:dev-34c68f9e87907e4259c13635cab7ad010bb2fc62
    volumes:
      - ./envoy.yaml:/etc/envoy/envoy.yaml
    ports:
      - "9901:9901"
      - "8080:8080"
  redis:
    image: bitnami/redis
    volumes:
      - ./redis:/bitnami/redis/data
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    ports:
      - "6379:6379"
```

Build and run the Envoy Docker container alongside your React app using Docker Compose.

That's it! You've added Docker configuration and set up Envoy Proxy to handle CORS for your gRPC-enabled React app. You can create a Word document with these instructions for reference.

Step 7: Start Your React App

- Start your React app to test the integration:
  npm start

Make sure your gRPC server is running and configured to accept requests from the React app.

That's it! You've set up gRPC with React.

Setting Up a gRPC Server in Python

Introduction

gRPC is a high-performance RPC (Remote Procedure Call) framework developed by Google. It allows you to define services and message types using Protocol Buffers (protobufs) and then generate server and client code in multiple programming languages. This guide will walk you through the process of setting up a gRPC server in Python.

Prerequisites

Before you begin, make sure you have the following installed:

- Python 3.x
- pip (Python package manager)
- Protocol Buffers compiler (protoc)
- gRPC Python library (grpcio)

You can install grpcio using pip:

```
pip install grpcio

pip install grpcio-tools
```

Define the gRPC Service

1. Create a .proto file to define your gRPC service and message types. For example, as shown above.
2. Generate Python code from the .proto file using this command:
   ```
   python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. example.proto
   ```

Implement the Server

1. Create a Python file for your gRPC server, e.g., server.py.

```
import grpc
import your_generated_code_pb2
import your_generated_code _pb2_grpc

class MyService(your_generated_code _pb2_grpc.MyServiceServicer):
    def MyMethod(self, request, context):
        return my_service_pb2.MyResponse(reply=f"Received: {request.message}")


def serve():
```

```
    server = grpc.server(grpc.insecure_channel())
    my_service_pb2_grpc.add_MyServiceServicer_to_server(MyService(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    print("Server started on port 50051...")
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

Run the Server

- Run the server script:

```
python server.py
```

Your gRPC server is now running on port 50051.

You've successfully set up a gRPC server in Python. You can now build gRPC clients to interact with this server and define more complex services and message types as needed for your application.