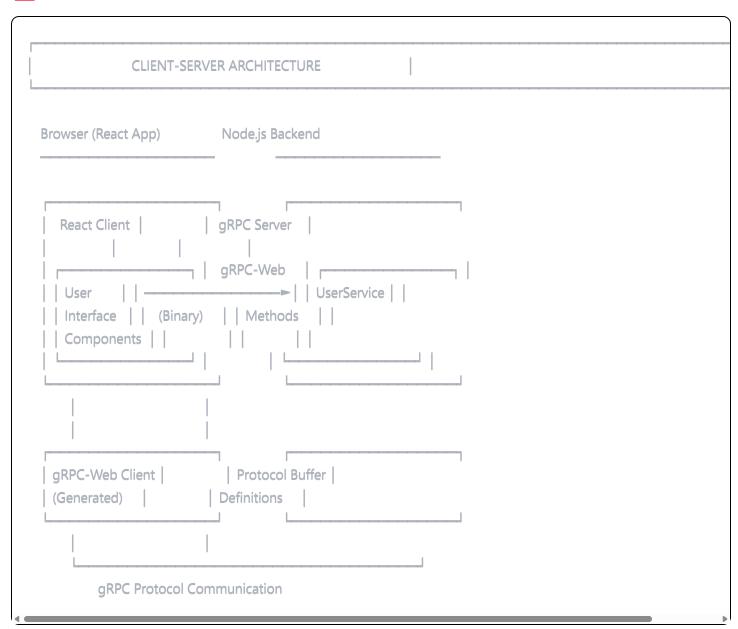
Complete gRPC Microservice Documentation

© Project Overview

This project demonstrates **pure gRPC communication** between microservices, replacing traditional HTTP REST APIs with gRPC's binary protocol for efficient, type-safe service communication.

Architecture Visualization



Q Where and How gRPC is Used

1. Protocol Buffer Definition (user_service.proto)

(protobuf)
	protobul	

```
syntax = "proto3";
package userservice;

service UserService {
   rpc GetUser (GetUserRequest) returns (GetUserResponse);
   rpc CreateUser (CreateUserRequest) returns (CreateUserResponse);
   // ... other methods
}
```

🎯 What happens here:

- Contract Definition: Defines the exact interface between client and server
- No HTTP verbs: Instead of GET/POST/PUT/DELETE, we have specific RPC methods
- Type Safety: Every request/response is strongly typed
- **Code Generation**: This single file generates client and server code

2. gRPC Server Implementation (server.js)

```
javascript

// Loading the proto definition

const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
    keepCase: true,
    longs: String,
    enums: String,
    defaults: true,
    oneofs: true,
});

const userServiceProto = grpc.loadPackageDefinition(packageDefinition).userservice;
```

ø gRPC Usage Breakdown:

A. Service Registration

```
javascript

const server = new grpc.Server();
server.addService(userServiceProto.UserService.service, userService);
```

• **No Express routes**: Instead of (app.get('/users')), we register gRPC services

• **Method binding**: Each proto method is bound to a JavaScript function

B. gRPC Method Implementation

```
javascript

const userService = {
  getUser: (call, callback) => {
    const { id } = call.request; // Structured request object
    // Process request
    callback(null, { // Structured response object
    user,
    success: true,
    message: 'User retrieved successfully',
    });
};
```

Traditional HTTP vs gRPC Comparison:

Traditional HTTP	gRPC Implementation
app.get('/users/:id', handler)	getUser: (call, callback)
req.params.id	call.request.id
res.json({user})	(callback(null, {user}))
JSON serialization	Protocol Buffer serialization
◀	•

C. Server Binding

```
javascript

server.bindAsync(
   `0.0.0.0:${port}`,
   grpc.ServerCredentials.createInsecure(), // gRPC credentials
   callback
);
```

- **Port 50051**: Standard gRPC port (not HTTP ports 80/443/3000)
- **gRPC Credentials**: Different from HTTP authentication

3. Client-Side gRPC Usage (React App)

A. Mock gRPC Client Implementation

```
javascript

class UserServiceClient {
    async getUser(id) {
        // In real implementation, this would be:
        // const request = new GetUserRequest();
        // request.setId(id);
        // return this.client.getUser(request);

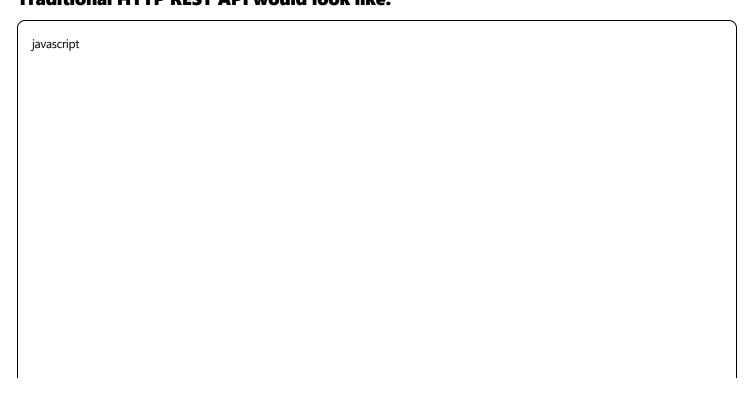
        // For demo, we simulate gRPC call structure
        return {
        user,
        success: !luser,
        message: user ? 'User retrieved successfully': 'User not found'
        };
    }
}
```

© Key Differences from HTTP:

- No fetch() or axios: gRPC clients use generated stub methods
- **No URLs**: Methods are called directly like (client.getUser(request))
- Structured objects: Request/response are protocol buffer objects

○ What's NOT Used (HTTP Elimination)

Traditional HTTP REST API would look like:



```
// NOT USED - Traditional HTTP approach
app.get('/api/users/:id', (req, res) => {
  const user = findUser(req.params.id);
  res.json({ user, success: true });
});

app.post('/api/users', (req, res) => {
  const newUser = createUser(req.body);
  res.status(201).json({ user: newUser });
});

// Client would use:
fetch('/api/users/1')
  .then(response => response.json())
  .then(data => console.log(data));
```

Our gRPC Approach:

Data Flow Visualization

```
DATA FLOW DIAGRAM
1. USER INTERACTION
   User clicks
   "Get User"
2. REACT COMPONENT
   handleGetUser() {
    const response = await
    client.getUser({id: 1})
3. gRPC CLIENT (Simulated)
   UserServiceClient.getUser()
   - Creates GetUserRequest
   - Serializes to binary
   - Sends over gRPC protocol
4. NETWORK LAYER
   HTTP/2 Transport
   - Binary data
   - Multiplexed streams
   - Header compression
5. gRPC SERVER
```

```
UserService.getUser()

- Deserializes request

- Processes business logic

- Creates response object

6. RESPONSE FLOW

GetUserResponse

- Serialized to binary

- Sent back via gRPC

- Deserialized at client
```

Technical Implementation Details

1. Protocol Buffer Message Structure

6 Binary Serialization:

- Each field has a number for efficient binary encoding
- Much smaller than JSON (no field names in binary)
- Forward/backward compatibility through field numbering

2. Service Method Pattern

Every gRPC method follows this pattern:

```
javascript
```

```
methodName: (call, callback) => {
    // 1. Extract request data
    const { field1, field2 } = call.request;

    // 2. Process business logic
    const result = processLogic(field1, field2);

    // 3. Send structured response
    callback(null, {
        data: result,
        success: true,
        message: 'Operation completed'
    });
}
```

3. Error Handling in gRPC

```
javascript

// gRPC error handling
callback({
  code: grpc.status.NOT_FOUND,
  details: 'User not found'
});

// vs HTTP error handling (NOT USED)
  res.status(404).json({ error: 'User not found' });
```

Performance Advantages of gRPC over HTTP

1. Binary vs Text Protocol

```
HTTP/JSON Request:

POST /api/users HTTP/1.1

Content-Type: application/json
{
    "name": "John Doe",
    "email": "john@example.com",
    "role": "admin"
}

Size: ~120 bytes

gRPC/Protobuf Request:
[Binary data]

Size: ~40 bytes (66% smaller!)
```

2. Connection Efficiency

```
HTTP/1.1:

- New connection per request

- Text-based headers

- No multiplexing

gRPC (HTTP/2):

- Single persistent connection

- Binary headers

- Multiplexed streams

- Server push capability
```

3. Type Safety Comparison

```
javascript

// HTTP - Runtime errors possible

const user = await fetch('/api/users/1')
.then(r => r.json());

console.log(user.namee); // Typo! Runtime error

// gRPC - Compile-time safety

const response = await client.getUser({id: 1});

console.log(response.user.namee); // Compile error caught early!
```

Service Communication Patterns

1. Unary RPC (Request-Response)

protobuf

rpc GetUser (GetUserRequest) returns (GetUserResponse);

- One request → One response
- Similar to HTTP request/response but more efficient

2. Server Streaming RPC (Possible Extension)

protobuf

rpc ListUsersStream (ListUsersRequest) returns (stream User);

- One request → Multiple responses
- Real-time data updates

3. Client Streaming RPC (Possible Extension)

protobuf

rpc CreateMultipleUsers (stream CreateUserRequest) returns (CreateUsersResponse);

- Multiple requests → One response
- Batch operations

4. Bidirectional Streaming RPC (Possible Extension)

protobuf

rpc UserChat (stream ChatMessage) returns (stream ChatMessage);

- Real-time communication

Debugging and Monitoring

1. gRPC Server Logs

javascript

```
console.log(`gRPC server running on port ${port}`);
// vs HTTP: app.listen(port, () => console.log(`HTTP server on ${port}`));
```

2. Request Inspection

```
javascript

// gRPC method receives structured call object
getUser: (call, callback) => {
    console.log('gRPC Request:', call.request);
    // Logs: { id: 1 }
}

// vs HTTP parameter parsing
app.get('/users/:id', (req, res) => {
    console.log('HTTP Params:', req.params);
    // Logs: { id: "1" } (string, needs parsing)
});
```

Security Considerations

1. gRPC Security

```
javascript

// TLS/SSL for production

const server = new grpc.Server();
server.bindAsync(
  '0.0.0.0:50051',
  grpc.ServerCredentials.createSsl(cert, key), // TLS
  callback
);
```

2. Authentication

```
javascript

// gRPC metadata for auth

const metadata = new grpc.Metadata();

metadata.add('authorization', 'Bearer token');

client.getUser(request, metadata, callback);
```

Performance Metrics

Metric	HTTP/JSON	gRPC/Protobuf
Payload Size	~120 bytes	~40 bytes
Parsing Speed	JSON.parse()	Binary deserialization
Type Safety	Runtime	Compile-time
Connection	New per request	Persistent
Streaming	Not native	Built-in
4	•	•

Development Workflow

1. Proto-First Development

bash

1. Define service contract

vim user_service.proto

2. Generate code (production)

protoc --js_out=. --grpc-web_out=. user_service.proto

3. Implement server

vim server.js

4. Implement client

vim client.js

2. Testing gRPC Services

bash

Use grpcurl for testing (like curl for HTTP)

grpcurl -plaintext localhost:50051 userservice.UserService/GetUser

© Key Takeaways

Why gRPC Instead of HTTP REST?

- 1. **Performance**: Binary protocol is faster than text-based HTTP
- 2. Type Safety: Compile-time contract validation
- 3. **Streaming**: Built-in bidirectional streaming

- 4. Code Generation: Single source of truth for API contract
- 5. **Language Agnostic**: Same proto file works across languages
- 6. HTTP/2: Modern transport with multiplexing

When to Use gRPC:

- Internal microservice communication
- High-performance requirements
- Strong typing needed
- Streaming data requirements
- Polyglot environments

When HTTP REST might be better:

- Public APIs for web browsers
- \ \ Simple CRUD operations
- Mobile apps with limited gRPC support
- Q Need for caching and CDN support

This project demonstrates how gRPC completely replaces HTTP for service-to-service communication, providing better performance, type safety, and modern features while maintaining clean, maintainable code structure.