GROUP 3

1 NSHIMIYIMANA Jean Nepo
2 HAKIZIMANA Theogene
3 AKIMANA Gabriel
4 HABIMANA Sixbert

Task

Group 3: Custom RPC Framework with Marshalling

Design and implement a simple RPC system with client/server stubs.

Include marshalling/unmarshalling for complex data types. Calculate serialization overhead and compare with existing frameworks (gRPC, Thrift).

FRAMEWORK OVERVIEW

What is tRPC?

tRPC (TypeScript Remote Procedure Call) is a framework designed to build end-to-end type-safe APIs without schemas or code generation.

> *"Move Fast and Break Nothing. Automatic typesafety & autocompletion inferred from your API-paths."*

End-to-End Type Safety
Types are automatically inferred from your router directly to your client.

No Code Generation
Forget about `codegen.yml` or OpenAPI schemas. Just import the type definition.

Runtime Validation
Deep integration with validation libraries like **Zod** for bulletproof inputs/outputs.

Great DX
Full IDE autocompletion, jump-to-definition, and typed errors out of the box.

IMPLEMENTATION DETAIL

Server Code Walkthrough

A breakdown of how tRPC is initialized, how routers are defined, and how it integrates with Express.

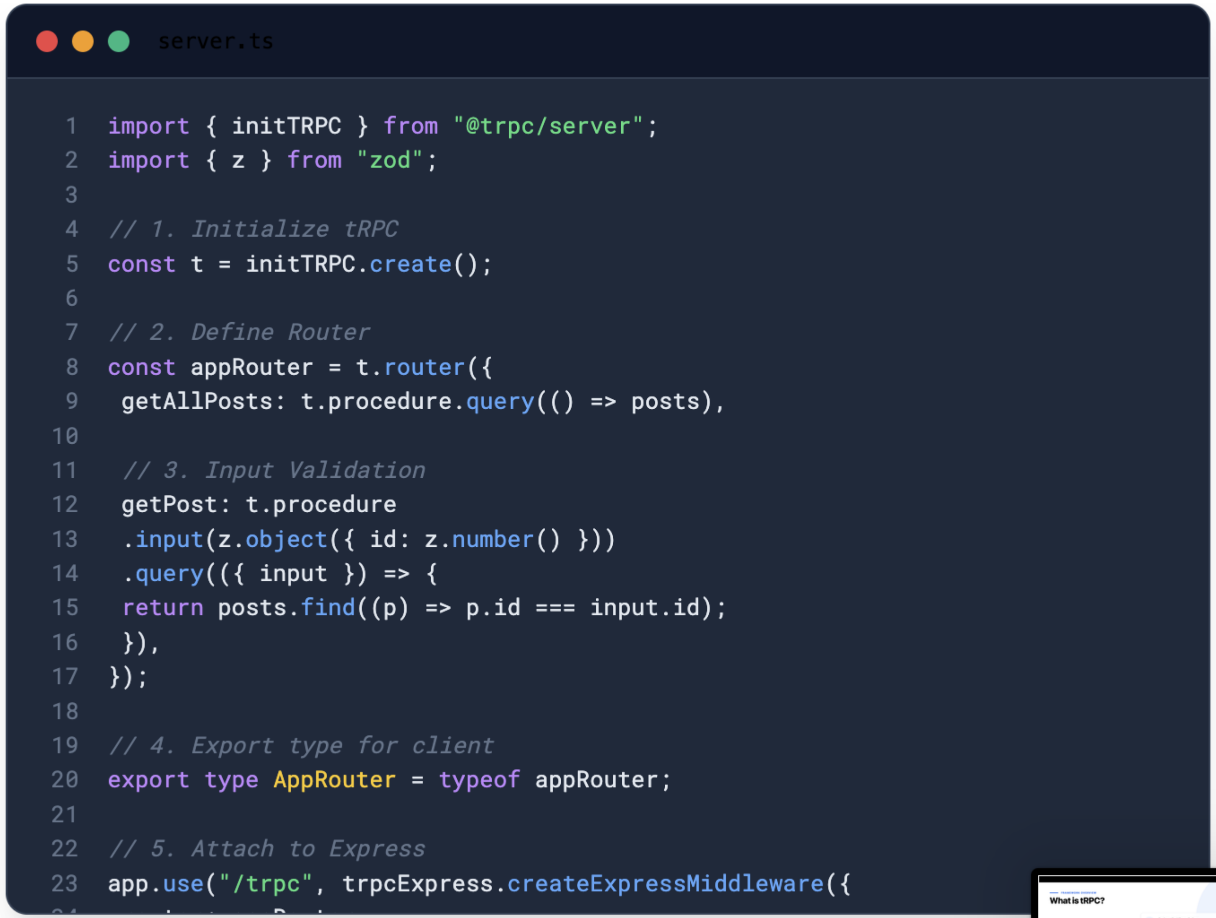
1. **Initialization**
   `initTRPC.create()` sets up the instance. We extract `router` and `publicProcedure` builders.

2. **Router Definition**
   The `appRouter` contains all API endpoints. Endpoints are just functions (queries or mutations).

3. **Input Validation (Zod)**
   `.input(z.object({...}))` validates parameters *before* your handler runs. Provides type safety inside the handler.

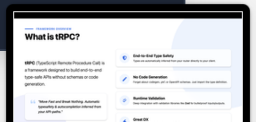
4. **Type Export**
   `export type AppRouter` allows the client to import **only the types**, not the server code.

5. **Express Adapter**
   Connects tRPC router to Express at `/trpc` via `createExpressMiddleware`.



server.ts

```
import { initTRPC } from "@trpc/server";
import { z } from "zod";

// 1. Initialize tRPC
const t = initTRPC.create();

// 2. Define Router
const appRouter = t.router({
 getAllPosts: t.procedure.query(() => posts),

 // 3. Input Validation
 getPost: t.procedure
 .input(z.object({ id: z.number() }))
 .query(({ input }) => {
 return posts.find((p) => p.id === input.id);
 }),
});

// 4. Export type for client
export type AppRouter = typeof appRouter;

// 5. Attach to Express
app.use("/trpc", trpcExpress.createExpressMiddleware({
```



What is tRPC?

DATA FLOW

Unmarshalling: Client ➜ Server

Converting wire format (JSON) back into usable in-memory objects.

1. **FUNCTION CALL**
   Client invokes typed procedure
   `getPost({id: 2})`

2. **MARSHALLING**
   Input converted to payload
   `JSON.stringify`

3. **TRANSPORT**
   HTTP Request sent
   `GET /trpc/getPost...`

4. **PARSING**
   Express/tRPC receive body
   `body-parser`

5. **UNMARSHALLING**
   JSON to JS Object
   `JSON.parse()`

6. **VALIDATION**
   Zod schema check
   `z.object({id: z.number()})`

7. **EXECUTION**
   Procedure runs logic
   `posts.find(...)`

RESPONSE FLOW

Marshalling: Server ➜ Client

Converting in-memory objects back to wire format (JSON) for transport.

1

PROCEDURE RETURN

Server logic returns a native JS object

```
return post;
```

2

MARSHALLING

tRPC converts object to JSON string

```
JSON.stringify(data)
```

3

TRANSPORT

Response sent over the wire

```
HTTP 200 OK
```

4

UNMARSHALLING

Client infers types from AppRouter

```
const { data } = trpc...
```

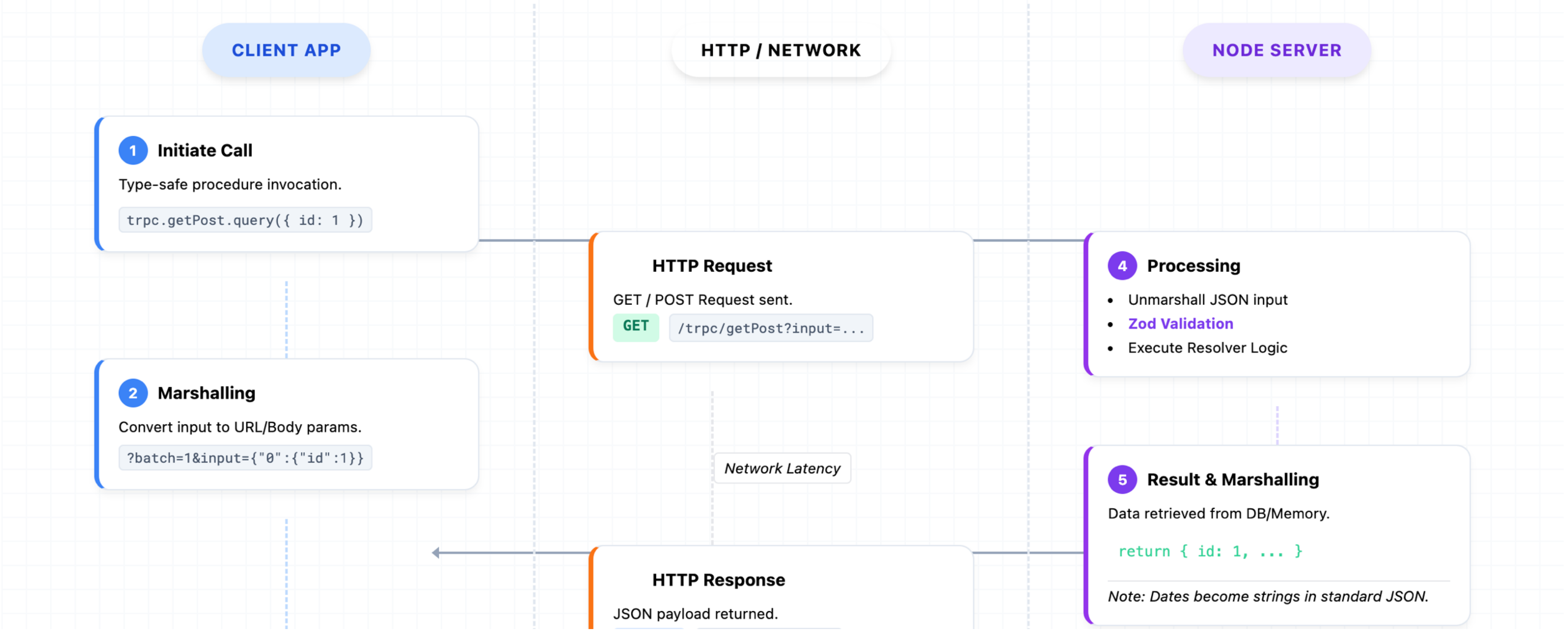
Serialization Warning: Dates & Special Types

By default, JSON marshalling converts `Date` objects to ISO strings (e.g., "2024-01-15..."). To preserve true Date objects on the client, use a transformer like **superjson**.

END-TO-END CYCLE

Request & Response Flow

Complete journey of data types over the wire.



CLIENT APP

HTTP / NETWORK

NODE SERVER

1 Initiate Call

Type-safe procedure invocation.

```
trpc.getPost.query({ id: 1 })
```

2 Marshalling

Convert input to URL/Body params.

```
?batch=1&input={"0":{"id":1}}
```

HTTP Request

GET / POST Request sent.

`GET` `/trpc/getPost?input=...`

Network Latency

HTTP Response

JSON payload returned.

4 Processing

- Unmarshall JSON input
- **Zod Validation**
- Execute Resolver Logic

5 Result & Marshalling

Data retrieved from DB/Memory.

```
return { id: 1, ... }
```

Note: Dates become strings in standard JSON.

WHY TRPC?

Key Benefits in This Setup

Combining tRPC, Zod, and Express creates a powerful environment for reliable application development.

E2E Type Safety

Types are inferred directly from your server router. No manual type definitions or `codegen` required to keep client and server in sync.

Runtime Integrity

Zod validates data at the I/O boundary. Malformed JSON is caught immediately before it hits your business logic, preventing runtime crashes.

Minimal Boilerplate

No separate DTO files, no interface duplication. The marshalling and unmarshalling logic is abstracted away by the framework adapters.

Superior DX

Get autocompletion for API endpoints in your frontend IDE instantly. Rename a property on the server, and see the red squiggly line on the client.

Flexible Serialization

Use standard JSON for speed, or plug in libraries like **SuperJSON** to seamlessly transport `Date`, `Map`, and `Set` objects without manual conversion.

Typed Error Handling

Throw `TRPCError` on the server to return consistent, typed error shapes to the client, simplifying exception handling logic.

SUMMARY

Conclusion & Next Steps

Wrapping up the data flow and looking ahead.

Core Concept Recap

UNMARSHALLING

The journey from `JSON String` (Wire) to `JS Object` (Server). Handled automatically by tRPC + Zod validation.

MARSHALLING

The return trip from `JS Object` (Server) to `JSON String` (Client). By default, strips rich types like Dates.

TYPE SAFETY

The `AppRouter` type export acts as the contract, ensuring client and server speak the exact same language without schemas.

Implementation Roadmap

Add Rich Types (SuperJSON)

Configure `transformer: superjson` to automatically handle `Date`, `Map`, and `Set` without manual conversion.

Typed Error Handling

Use `TRPCError` to return standardized error codes (e.g., `NOT_FOUND`) that the client can handle gracefully.

Frontend Integration

Import `AppRouter` in your React/Vue client to get instant autocompletion on `trpc.getPost.useQuery(...)`.