Department of Computer Science
University of California, Los Angeles

Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

Lecture 9: April 28, 2025

Outline

Beyond REST

Last time we discussed the following:

1. Using a REST API via the Fetch API, and then really quickly Axios
2. The tradeoffs of using Fetch and Axios
3. How to build a REST API using Express
4. REST API best practices

Beyond REST (contd.)

We covered the most important conceptual aspects of REST APIs.

On Project 2, you will learn the rest.

Beyond REST (contd.)

Today, we will move past REST and discuss other API paradigms, including some late breaking ones.

Then we will discuss streaming data from servers to clients.

Beyond REST (contd.)

Today's topics:

1. GraphQL
2. gRPC
3. WebSockets
4. Server-Sent Events
5. WebTransport via HTTP/3
6. Socket.IO

Remember to REST

Remember the following about REST:

- It is based on resources, things you want to expose to the user.
- Communication is based on endpoints, simple declarative URLs.
- HTTP verbs: `GET, POST, PUT, PATCH, DELETE`
- Stateless communication
- Structured URLs and status codes

Too Much REST Is Bad

REST APIs are great, they are simple to use. But they cause some problems:

- Over-fetching
- Under-fetching
- Sometimes we need realtime updates
- Client flexibility
- We often need a strong contract between client and server

GraphQL

GraphQL is a query language for APIs and a runtime for executing those queries with your existing data.

GraphQL (contd.)

GraphQL features:

- Query language for APIs
- Single endpoint (not multiple)
- Flexible queries
- Client defines the schema they want the data in
- The server resolves requested fields
- Flexible, avoids over-fetching and under-fetching
- Strongly typed schema
- **Tradeoff:** Adds complexity

GraphQL (contd.)

```
query {
  trails(difficulty: "Beginner", open: true) {
    name
  }
}
```

Note that the server must support filtering queries. In REST this would require request parameters, or a new endpoint.

We control what we fetch. In REST, we would get other needless fields.

GraphQL (contd.)

Response:

```
{
  "data": {
    "trails": [
      {
          "name": "Easy Street"
      }
    ]
  }
}
```

GraphQL (contd.)

If the server does not support filtering though, we must over-fetch and then filter client side:

```
query {
  trails {
    name
    difficulty
    open
  }
}
const beginnerOpenTrails = data.resort.trails.filter(
  trail => trail.difficulty === "Beginner" && trail.open
);
```

GraphQL in Express

There are a few ways to work with GraphQL in Express:

1. Apollo Server
2. Express GraphQL (graphql-http)

`graphql-http` is a simple middleware for Express that allows you to create a GraphQL server and is recommended for small projects and beginners. We will use Apollo which is a more powerful and flexible solution.

GraphQL in Express (contd.)

Dependencies for Apollo, GraphQL and Express 5 are frustrating right now. To install:

```
npm install graphql@^16.6.0 @apollo/server
```

GraphQL in Express (contd.)

GraphQL has two main components:

1. **Schema:** defines the types and relationships of your data
2. **Resolvers:** functions that resolve the data for each field in the schema

GraphQL: Schema

The schema contains types and fields.

There are scalar types: `String, Int, Float, Boolean, ID`

There are also custom shapes and relationships:

GraphQL: Schema (contd.)

A canonical example:

```
type Lodge {
  // Assigned by the server
  id: ID!
  name: String!
  lifts: [Lift!]!
  trails: [Trail!]!
}

type Trail {
  id: ID!
  name: String!
  difficulty: String!
  status: String!
  features: [String!]
  lodges: [Lodge!]!
}
```

```
type Lift {
  id: ID!
  name: String!
  capacity: Int!
  status: String!
  lastUpdated: String!
  lodges: [Lodge!]!
}
```

GraphQL: Resolvers

Resolvers are functions that tell the server how to fetch the data for each field. Every query, mutation and nested field can have a resolver.

1. Every (nested) field on every query or mutation, can have a resolver.
2. If a field does not have a resolver, GraphQL will use the default resolver for that field.
3. Looks like `(parent, args, context, info) => result`
    - `parent` is the result of the previous resolver
    - `args` are the arguments passed to the field (from the user)
    - `context` is an object shared by all resolvers in a query
    - Parent object flows down nested resolvers

GraphQL: Resolvers (contd.)

Example resolver for `Lift`.

```
const resolvers = {
  // Lift ID -> Lift object
  Query: {
    trail: (parent, args, context) => context.db.getTrailById(args.id),
  },
  // Lift object -> features and lodge object
  Trail: {
    // What is returned
    features: (parent) => parent.features, // Already part of the object
    lodges: (parent, args, context) => context.db.getLodgesForTrail(parent.id),
  },
};
```

GraphQL: Mutation

Mutation is just a fancy name for a query that modifies data.

```
mutation {
  addTrail(name: "Wazoo",
    difficulty: "Difficult",
    status: "CLOSED FOR SEASON",
    features: ["Uphill", "Groomer"]
  }) {  // Returned to user after mutation
    id
    name
  }
}
```

The mutation `addLift` is specified in the schema.

GraphQL: General Workflow

When the user executes a GraphQL query, the server does the following:

1. The query is received
2. The resolver is executed on all fields in the query
3. Any `context` functions are executed (e.g. database access)
4. The expected behavior is executed (e.g. database access)
5. The data is returned to the client

GraphQL: General Workflow (contd.)

Question: Why is this approach useful?

Demo: GraphQL

Let's take a look at both for the ski resort example...

When to use GraphQL

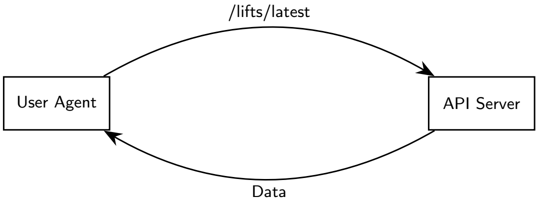
GraphQL has some specific use cases:

1. Developing specifically for mobile. **Why?**
2. Need flexibility in the data you fetch
3. Heterogeneous users and use cases dictated by client
4. Need to aggregate data from multiple sources
5. Note that *caching* is harder than in REST. **Why?**

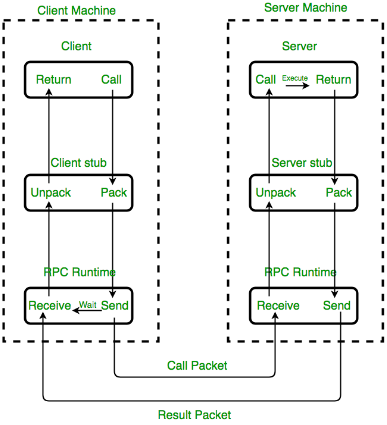
Remote Procedure Call (RPC)

RPC is a protocol that allows an application to execute a procedure (e.g. function) on a remote server as if it is executed locally.

Unlike REST which provides resources that the user/app exploits in a fetch, RPC is an **action** and is typically and HTTP POST. The **payload** is the function name and arguments.



Remote Procedure Call (RPC) (contd.)



Implementation of RPC mechanism

Remote Procedure Call (RPC) (contd.)

RPC is useful when we want a **strict contract** for I/O between the client and the server.

This is typically the case when two components of a (distributed) system need to communicate.

Remote Procedure Call (RPC) (contd.)

The three most popular RPC protocols are:

1. gRPC
2. Apache Thrift
3. tRPC (well, gaining popularity)

We will discuss gRPC a bit (as some of you may end up interning or working at Google or somewhere else that uses it), as well as tRPC.

gRPC

gRPC is an open-source version of RPC from Google. It uses protocol buffers (protobuf) and HTTP/2.

gRPC (contd.)

I used to write a lot of code like this. . .

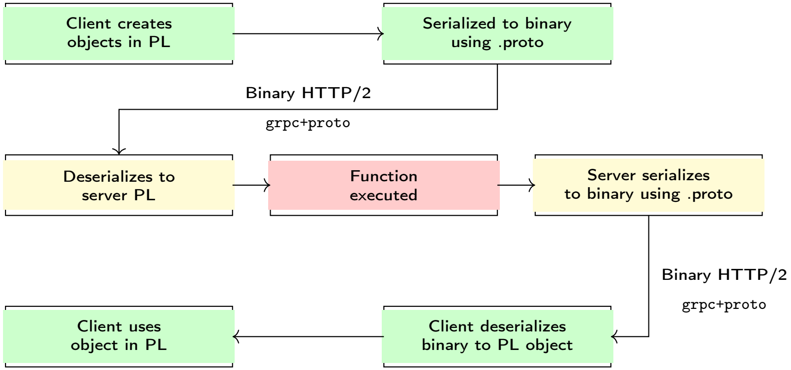
While I prefer REST, there are several things that are beautiful about gRPC over REST.

gRPC (contd.)

First, let's consider the workflow of gRPC:

It all starts with a .proto file that defines the contract between the client and server and message (data) types.

gRPC (contd.)


Client creates objects in PL
Serialized to binary using .proto
Binary HTTP/2
grpc+proto
Deserializes to server PL
Function executed
Server serializes to binary using .proto
Binary HTTP/2
grpc+proto
Client uses object in PL
Client deserializes binary to PL object

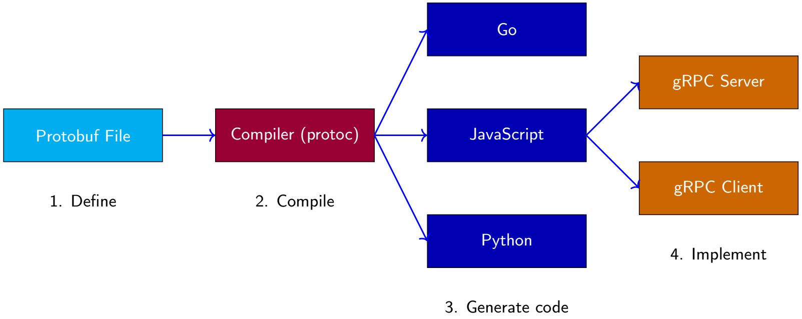
gRPC (contd.)

So why do I find this so beautiful?

1. The client and server can be written in different languages!
2. The developer does not need to worry about what happens on the server side, just that they get a response.
3. The developer never needs to worry about missing fields or data type mismatches.
4. Protobufs are much smaller than JSON.
5. HTTP/2 allows the server to push multiple responses to the client at once.
6. Streaming

gRPC (contd.)

The Protobuf compiler generates the client and server code for you in your language of choice.



The `proto` definition file creates a contract between the client and server.

gRPC (contd.)

Another thing that is beautiful about gRPC is that all you need to do is commit and deploy the `proto` file when changes are made!

Both the client and the server must compile the `proto` file to generate the code for their language.

```
syntax = "proto3";

service SkiResortService {
  rpc GetLatestTrails (Empty) returns (TrailBatch);
  rpc GetLatestLifts (Empty) returns (LiftBatch);
  rpc GetOpenTrails (Empty) returns (TrailBatch);
  rpc GetOpenLifts (Empty) returns (LiftBatch);
  rpc GetTrailAtTimestamp (TimestampRequest) returns (TrailBatch);
  rpc GetLiftAtTimestamp (TimestampRequest) returns (LiftBatch);
  rpc GetTrailStatus (NameRequest) returns (Trail);
  rpc GetLiftStatus (NameRequest) returns (Lift);
  rpc UpdateTrailStatus (Trail) returns (UpdateResponse);
  rpc UpdateLiftStatus (Lift) returns (UpdateResponse);
}

message Empty {}

message TimestampRequest {
  string timestamp = 1;
}
```

```
message Trail {
  string name = 1;
  string status = 2;
  optional repeated string features = 3; // Features are optional
}

message Lift {
  string name = 1;
  string status = 2;
  string misc = 3;
  int32 capacity = 4;
}

message TrailBatch {
  string id = 1;
  string timestamp = 2;
  string type = 3;
  repeated Trail data = 4;
}
```

gRPC

Server Skeleton:

```
const server = new grpc.Server();
server.addService(SkiResortService, {
  getLatestTrails: (call, callback) => {
    const trails = fetchLatestTrails(); // you write this
    callback(null, trails);
  }
  // plus many more
});
```

gRPC (contd.)

Client Stub:

```
const client = new SkiResortServiceClient('localhost:50051'

client.getLatestTrails(new Empty(), (error, response) => {
  console.log(response.getDataList()); // access trails dir
});
```

It feels like calling a function locally…

Apache Thrift

Developed by Facebook in 2007, Thrift is a framework for scalable cross-language services development.

Apache Thrift (contd.)

It is still in use by Facebook and other companies. It is similar to gRPC, but it is not as popular nowadays.

- Similar to gRPC it has its own language and can auto-generate code for multiple languages.
- Thrift can serialize to multiple formats (e.g. JSON, binary, compact)
- The protocol (e.g. HTTP) and transport layer (e.g. TCP) are pluggable, which is not the case for gRPC.
- gRPC is focuses on HTTP/2, streaming and standardization. Thrift is more configurable but complex.
- Has been eclipsed by gRPC in popularity.

tRPC

tRPC is a TypeScript-first RPC framework that is gaining traction.

tRPC (contd.)

It has some of the guarantees of gRPC, but with the flexibility of REST.

1. Unlike gRPC and more similar to REST (decoupled), client/server must be written in TypeScript.
2. Like gRPC, end-to-end typesafety.
3. Like GraphQL, there are mutations.
4. No schemas, client and server are expected to be in same codebase.
5. Like gRPC, it is better for internal component communication. **Why?**
6. **Great for web apps that are already migrating or migrated to TypeScript**
7. tRPC is no longer "just for Next.js" — it's expanding into Vite, Svelte, Bun, SolidJS, serverless, and even OpenAPI world.

tRPC (contd.)

There are a few important components of tRPC we will see in a bit:

1. Procedures
2. Routers
3. Client
4. Context

tRPC (contd.)

Procedures:

```
t.procedure
  .input(z.object({ trailId: z.string() }))
  .query(({ input }) => {
    // do something
  });
```

Routers and Procedures

```
export const appRouter = t.router({
  getTrails: t.procedure
    .input(z.object({ trailId: z.string() }))
    .query(({ input }) => {
      // Retrieve trail by ID
      return trails.find(trail => trail.id === input.trailId);
    }),

  addTrail: t.procedure
    .input(z.object({
      name: z.string(),
      difficulty: z.enum(["easy", "intermediate", "advanced"]),
      length: z.number().positive(),
      isOpen: z.boolean().default(false)
    }))
    .mutation(({ input, ctx }) => {
      // Create a new trail with the input data
      const newTrail = {
        id: generateId(),
        createdAt: new Date(),
        ...input
      };
```

tRPC

Client:

```
const trails = await client.getTrails.query();
const added = await client.addTrail.mutate({
  name: "New Trail", ... });
```

tRPC (contd.)

Context:

```
t.router({
  getUserData: t.procedure.query(({ ctx }) => {
    return ctx.user;
  }),
});
```

tRPC (contd.)

DEMO if time allows.

tRPC: Example

After seeing the demo, some questions:

1. Is this something that can be used on the web?
2. Is this as easy to test as REST?

Summary

We saw a few alternatives to REST:

1. GraphQL: schema, resolvers, mutations, flexible queries
2. gRPC: strong contract, different languages, protocol buffers, binary HTTP/2
3. Apache Thrift: similar to gRPC, but more complex and less popular
4. tRPC: TypeScript-first, end-to-end type safety, procedures, routers, contex, internal only

Streaming

Streaming was game changing, and still is. It breaks the traditional request/response model of web applications.

- Streaming is a method of processing data in real-time as it arrives.
- It allows for immediate analysis and response to incoming data.
- Commonly used in applications like chat, video streaming, online gaming, and real-time analytics.
- Allows a web app to process the most recent data without ”pulling” or polling.

Note that is similar, but separate from the concept of streaming architectures involving Kafka, Spark, etc.

Technologies for Today

There are many approaches to streaming data:

1. WebSockets (briefly)
2. WebTransport via HTTP/3
3. Socket.io
4. Server-Sent Events (SSE)

WebSockets

WebSockets (HTML5) are a protocol for full-duplex communication channels over a single TCP connection.

WebSockets (contd.)

WebSockets:

- require an "upgrade" from HTTP
- are persistent connections until closed
- are low latency
- allow a client or server to send messages at any time during the connection

WebSockets (contd.)

WebSockets are used for many interesting applications such as:

1. Real-time chat
2. Online gaming
3. Live data feeds (e.g. lift status, stock prices, sports scores)
4. Collaborative applications (e.g. Google Docs)
5. Real-time data visualization
6. Audio/video streaming
7. Real-time location apps and customer tracking

WebSockets (contd.)

```
ws = new WebSocket("wss://skiresort.com/status");

ws.onopen = () => {
    console.log("Connection opened");
    ws.send("Hello, please send me the gondola status.");
}

ws.onmessage = (event) => {
    console.log("Data received", event.data);
    ws.close(); // We got what we wanted.
}

ws.onclose = (event) => {
    console.log("Connection closed", event.code,
        event.reason, event.wasClean);
}

ws.onerror = () => { console.log("Connection closed due to error"); }
```

Flask WebSockets Server

```python
from flask import Flask
from flask_socketio import SocketIO, emit, send

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your-secret-key'
socketio = SocketIO(app, cors_allowed_origins="*")

# Simulate some ski resort data
resort_status = {
    "Gondola": "WIND HOLD",
    "Broadway Express": "OPEN",
    "Schoolyear Express": "CLOSED FOR SEASON",
}

@app.route('/')
def index():
    return "Ski Resort WebSocket Server"

@socketio.on('connect')
def handle_connect():
    print("Client connected")
    emit('status_update', resort_status)
```

Server-Sent Events (SSE)

Server-Sent Events (SSE) are a standard for server-to-client streaming. There are a better technologies out there today, but they are legacy and worth mentioning as they are still in use.

One reason other technologies are preferred is that SSE only allows one-way communication from server to client and functionality is limited.

Server-Sent Events (SSE) (contd.)

SSE is a standard for server-to-client streaming. It is a simple and efficient way to send real-time updates from the server to the client over HTTP.

- Native browser support (EventSource API)
- Server to Client only
- Easy for notifications and feeds
- Largely replaced today by WebSocket, gRPC, and HTTP/2

Server-Sent Events (SSE) (contd.)

```
const eventSource = new EventSource('/api/lift-status');

// Handle incoming messages (status updates)
eventSource.addEventListener('message', (event) => {
  const liftData = JSON.parse(event.data);
  console.log('Lift status update received:', liftData);

  // Update UI with new lift statuses
  updateLiftDisplay(liftData);
});
```

Server-Sent Events (SSE) (contd.)

Notifications, eh? So is that how apps send notifications? Here is how Facebook does it:

- On Desktop, they use WebSockets under normal conditions
- On Mobile, they tunnel MQTT over WebSockets (lightweight, tiny packets)
- On weak or lossy connections, they use long-polling.
- For push, MQTT over a persistent TCP connection, fallback to WebSockets.
- For internal services, typically Apache Thrift or gRPC
- Native mobile apps use a broker: Apple Push Notification for iOS and Firebase Cloud Messaging for Android

WebTransport and HTTP/3, QUIC

WebTransport is a new protocol for streaming data over HTTP/3. It is pretty hot right now but not universally supported.

WebTransport and HTTP/3, QUIC (contd.)

Some features of WebTransport:

- Reliable streams and unreliable datagrams (though not "raw" like UDP)
- Encryption and congestion control
- Origin-based security model: same-origin and CORS
- Multiplexing: multiple streams over a single connection
- Can resume partially completed transfers after disruptions
- No separate upgrade handshake (unlike WebSockets)
- Built on top of QUIC (multiplexed, replaces TCP+TLS by combining them)

WebTransport and HTTP/3, QUIC (contd.)

Feature	**WebSocket**	**WebTransport**
Transport Protocol	TCP	QUIC (over UDP)
Multiplexing	No	Yes (many streams possible)
Latency	Moderate (TCP handshake)	Very Low (QUIC handshake)
Server Push	No	Yes
Future-proof	Limited	High (designed into HTTP/3)

WebTransport and HTTP/3, QUIC (contd.)

WebTransport is great for:

- sending/receiving high-frequency, small messages that don't need to be reliable
- sending/receiving low-latency media
- transferring files

Socket.io

Socket.io is a library that enables real-time, bidirectional communication between web clients and servers. It was originally build on top of WebSockets.

Socket.io (contd.)

Socket.io now provides a multifaceted approach to real-time communication.

1. It first tries to use WebSockets
2. If that fails, it falls back to other techniques like long-polling

v4.5+ has started experimenting with WebTransport, but it is not yet stable. **This is why we are covering it last today.**

Socket.io (contd.)

It provides a lot of features that are not included in WebSockets:

1. Automatic reconnection
2. Fallback options (e.g. long-polling)
3. Broadcasting to multiple clients
4. Event-driven API (similar to SSE)
5. Supports rooms and namespaces for organizing connections

Socket.io (contd.)

DEMO

Socket.io (contd.)

Socket.io is great for:

- Reliable real-time communication without managing reconnections
- Want to separate clients into rooms (e.g. groups like game or chat rooms)
- Broad browser support
- Rapid development and prototyping

Summary

Architecture Summary:

Protocol	Best for
REST	CRUD APIs
GraphQL	Flexible client-driven queries
RPC (gRPC, Thrift, tRPC)	Tight internal integration between services
WebSockets	Real-time bi-directional updates
Socket.IO	Reliable real-time communication with fallbacks and rooms
WebTransport	Future scalable real-time streams over HTTP/3

Summary (contd.)

Choosing the right tool:

- **Simple CRUD operations?** REST
- **Complex nested queries?** GraphQL
- **Internal services/microservices?** RPC
- **Live bidirectional data?** WebSockets or Socket.io
- **Future-proof streaming over HTTP/3?** WebTransport
- As with all systems, expect to mix and match.

References

- GraphQL Official Site
- tRPC Documentation
- Using WebSockets API
- WebTransport API
- Socket.IO Documentation