System Analysis and Design

API Design

By: Vahid Rahimian

Spring 2022

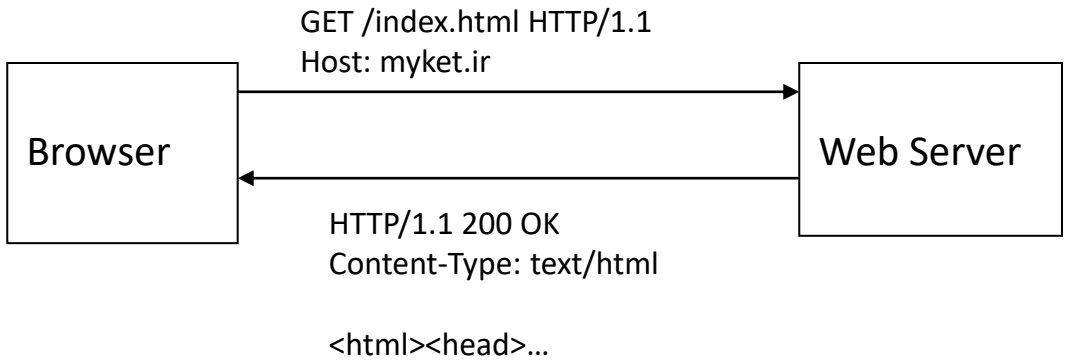
Agenda

- HTTP, REST Basics
- REST API Design Guide
- SOAP, REST, GraphQL, and gRPC
- Which API Format?

HTTP, REST Basics

Hypertext Transfer Protocol (HTTP)

- A communications protocol
- Allows retrieving inter-linked text documents (hypertext)
  - World Wide Web.
- HTTP Verbs
  - HEAD
  - **GET**
  - **POST**
  - PUT
  - DELETE
  - TRACE
  - OPTIONS
  - CONNECT



Representational State Transfer (REST)

- A style of software architecture for distributed hypermedia systems such as the World Wide Web.
- Introduced in the doctoral dissertation of Roy Fielding
  - One of the principal authors of the HTTP specification.
- A collection of network architecture principles which outline how resources are defined and addressed

REST and HTTP

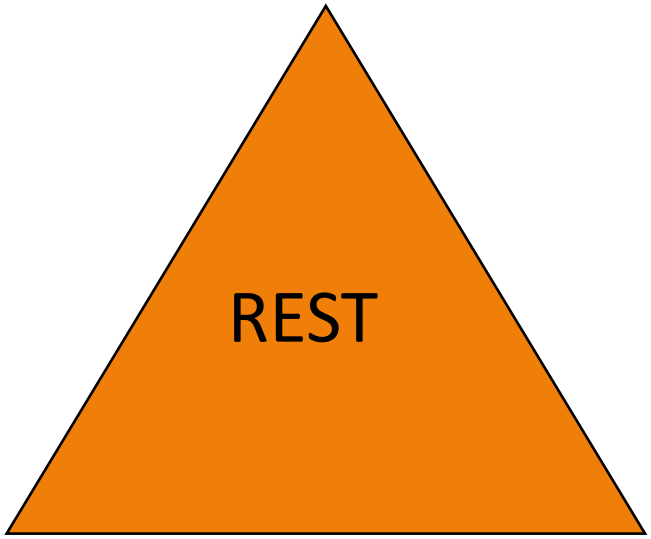
- The motivation for REST was to capture the characteristics of the Web which made the Web successful.
  - URI Addressable resources
  - HTTP Protocol
  - Make a Request – Receive Response – Display Response
- Exploits the use of the HTTP protocol beyond HTTP POST and HTTP GET
  - HTTP PUT, HTTP DELETE

REST - not a Standard

- REST is not a standard
  - JSR 311: JAX-RS: The Java™ API for RESTful Web Services
- But it uses several standards:
  - HTTP
  - URL
  - XML/HTML/GIF/JPEG/etc (Resource Representations)
  - text/xml, text/html, image/gif, image/jpeg, etc (Resource Types, MIME Types)

Main Concepts

Nouns (Resources)
unconstrained
i.e., http://example.com/employees/12345



Verbs
constrained
i.e., GET

Representations
constrained
i.e., XML

Resources

- The key abstraction of information in REST is a resource.
- A resource is a conceptual mapping to a set of entities
  - Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on
- Represented with a global identifier (URI in HTTP)
  - http://myket.ir/games/clash-of-clans

Naming Resources

- REST uses URI to identify resources
  - https://myket.ir/games/
  - https://myket.ir/games/clash-of-clans/
  - https://myket.ir/games/clash-of-clans/comments/
  - http://sharif.edu/classes
  - http://sharif.edu/classes/cs40418-2
  - http://sharif.edu/classes/cs40418-2/students
- As you traverse the path from more generic to more specific, you are navigating the data

Verbs

- Represent the actions to be performed on resources

- HTTP GET
- HTTP POST
- HTTP PUT
- HTTP DELETE

HTTP GET

- How clients ask for the information they seek.
- Issuing a GET request transfers the data from the server to the client in some representation

- GET http://taaghche.com/books
  - Retrieve all books
- GET http://taaghche.com /books/ISBN-0011021
  - Retrieve book identified with ISBN-0011021
- GET http://taaghche.com/books/ISBN-0011021/authors
  - Retrieve authors for book identified with ISBN-0011021

HTTP PUT, HTTP POST

- HTTP POST creates a resource
- HTTP PUT updates a resource

- POST http://admin.taaghche.com/books/
  - Content: {title, authors[], …}
  - Creates a new book with given properties

- PUT http://taaghche.com/books/isbn-111
  - Content: {isbn, title, authors[], …}
  - Updates book identified by isbn-111 with submitted properties

HTTP DELETE

- Removes the resource identified by the URI

- DELETE http://admin.taaghche.com/books/ISBN-0011
  - Delete book identified by ISBN-0011

Representations

- How data is represented or returned to the client for presentation.
- Two main formats:
  - JavaScript Object Notation (JSON)
  - XML
- It is common to have multiple representations of the same data

Representations

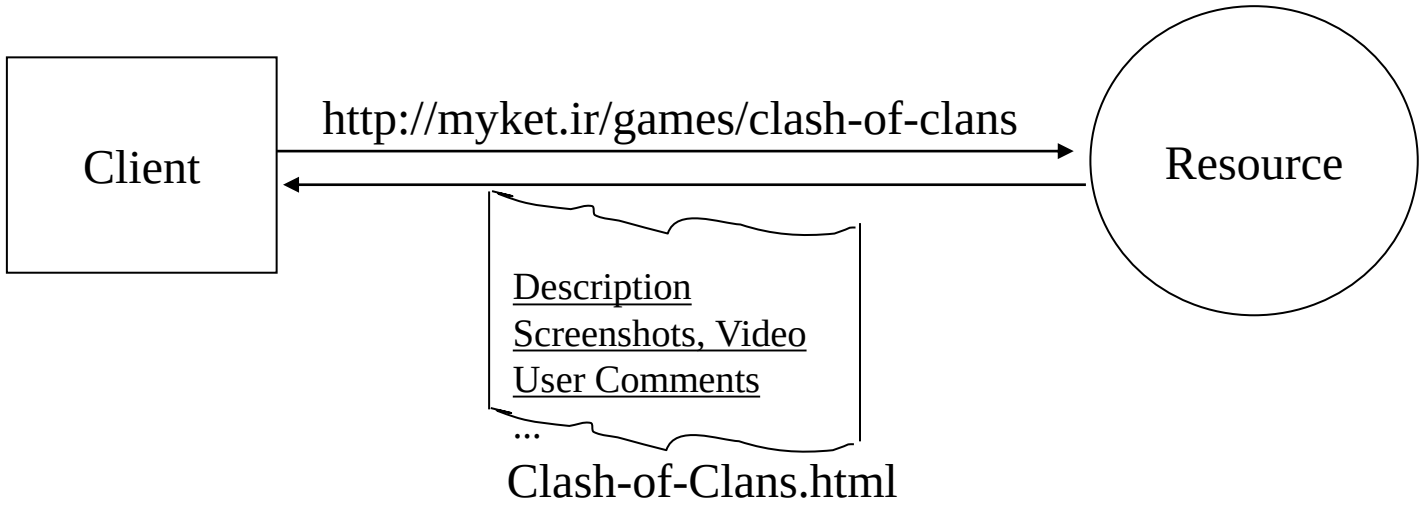
- XML

```
<COURSE>
    <ID> CS40418-2 </ID>
    <NAME> System Analysis and Design </NAME>
    <INSTRUCTOR> Vahid Rahimian </INSTRUCTOR>
</COURSE>
```

- JSON

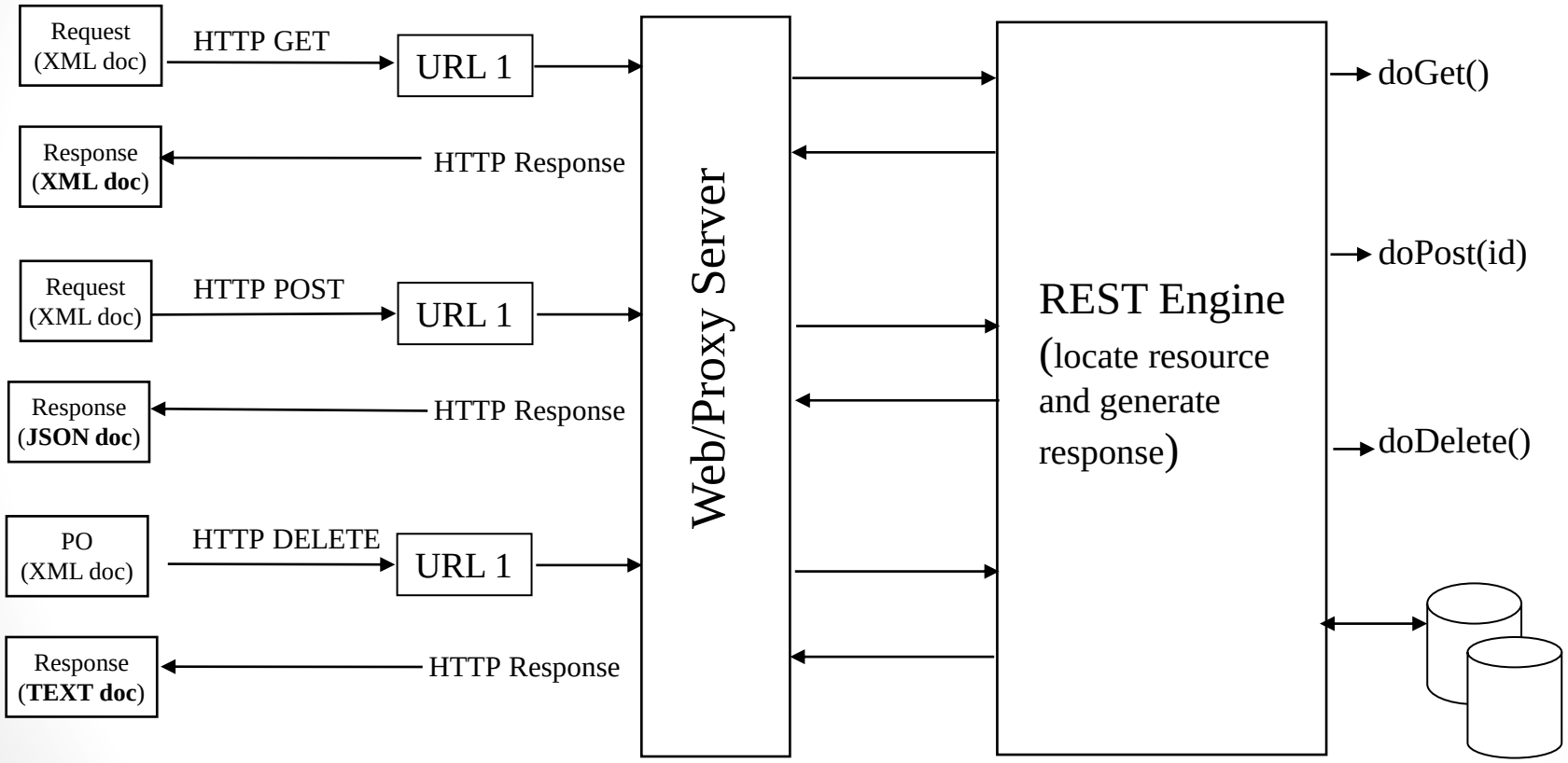
```
{
    “id”: “CS40418-2”,
    “name”: “System Analysis and Design”,
    “instructor”: “Vahid Rahimian”
}
```

Representational State Transfer



The Client references a Web resource using a URL. A **representation** of the resource is returned (in this case as an HTML document).
The representation (e.g., Clash-Of-Clans.html) places the client application in a **state**. The result of the client traversing a hyperlink in Clash-Of-Clans.html is another resource accessed. The new representation places the client application into yet another state. Thus, the client application changes (**transfer**s) state with each resource representation --> Representation State Transfer!

Architecture Style



Real Life Examples

- Google Maps
- Google AJAX Search API
- Amazon Web Services
- Trello API

REST and the Web

- The Web is an example of a REST system!
- All of those Web services that you have been using all these many years - book ordering services, search services, online dictionary services, etc - are REST-based Web services.
- Alas, you have been using REST, building REST services and you didn't even know it.

REST API Design Guide

Organize the API around resources

- Focus on the business entities that the web API exposes. For example, in an e-commerce system, the primary entities might be customers and orders.
- Avoid requiring resource URIs more complex
  than *collection/item/collection*.
-

```
HTTP
https://adventure-works.com/orders // Good

https://adventure-works.com/create-order // Avoid
```

Define operations in terms of HTTP methods

- GET retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- POST creates a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.

Define operations in terms of HTTP methods

- PUT either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- PATCH performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
- DELETE removes the resource at the specified URI.

Define operations in terms of HTTP methods

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

Conform to HTTP semantics: Media Types

- In the HTTP protocol, formats are specified through the use of *media types*, also called MIME types.
- For non-binary data, most web APIs support JSON (media type = application/json) and possibly XML (media type = application/xml).
- The Content-Type header in a request or response specifies the format of the representation.

Conform to HTTP semantics: Media Types

HTTP

Copy

```
POST https://adventure-works.com/orders HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

If the server doesn't support the media type, it should return HTTP status code 415 (Unsupported Media Type).

Conform to HTTP semantics: Media Types

HTTP

Copy

```
GET https://adventure-works.com/orders/2 HTTP/1.1
Accept: application/json
```

- A client request can include an Accept header that contains a list of media types the client will accept from the server in the response message.
- If the server cannot match any of the media type(s) listed, it should return HTTP status code 406 (Not Acceptable).

Conform to HTTP semantics: POST methods

- If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

- If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

- If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

Conform to HTTP semantics: PATCH methods

- With a PATCH request, the client sends a set of updates to an existing resource, in the form of a patch document. The server processes the patch document to perform the update.

Error condition	HTTP status code
The patch document format isn't supported.	415 (Unsupported Media Type)
Malformed patch document.	400 (Bad Request)
The patch document is valid, but the changes can't be applied to the resource in its current state.	409 (Conflict)

Conform to HTTP semantics: Async operations

- Sometimes a POST, PUT, PATCH, or DELETE operation might require processing that takes a while to complete.
- If you wait for completion before sending a response to the client, it may cause unacceptable latency.
- If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed

Filter and paginate data

- GET requests over collection resources can potentially return a large number of items.
- You should design a web API to limit the amount of data returned by any single request.
- Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection.

HTTP Copy

```
/orders?limit=25&offset=50
```

Support partial responses for large binary resources

- A resource may contain large binary fields, such as files or images.
- To overcome problems caused by unreliable and intermittent connections and to improve response times, consider enabling such resources to be retrieved in chunks.
- To do this, the web API should support the Accept-Ranges header for GET requests for large resources.
- This header indicates that the GET operation supports partial requests. The client application can submit GET requests that return a subset of a resource, specified as a range of bytes.

Support partial responses for large binary resources

HTTP

Copy

```
GET https://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=0-2499
```

HTTP

Copy

```
HTTP/1.1 206 Partial Content

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580

[...]
```

Support partial responses for large binary resources

- Also, consider implementing HTTP HEAD requests for these resources.
- A HEAD request is similar to a GET request, except that it only returns the HTTP headers that describe the resource, with an empty message body.
- A client application can issue a HEAD request to determine whether to fetch a resource by using partial GET requests.

Support partial responses for large binary resources

HTTP

Copy

```
HEAD https://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

HTTP

Copy

```
HTTP/1.1 200 OK

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 4580
```

Use HATEOAS to enable navigation to related resources

JSON Copy


```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

Versioning a RESTful web API

HTTP Copy

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

If the `DateCreated` field is added to the schema of the customer resource, then the response would look like this:

HTTP Copy

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":"1 Mic
```

Versioning a RESTful web API

- URI Versioning

https://adventure-works.com/v2/customers/3

- Query string versioning

https://adventure-works.com/customers/3?version=2

- Header versioning

HTTP Copy

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=2
```

Versioning a RESTful web API

- Media type versioning

HTTP

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

SOAP, REST, GraphQL, and gRPC

SOAP

- Simple Object Access Protocol (SOAP)
- A protocol for exchanging information encoded in Extensible Markup Language (XML) between a client and a procedure or service that resides on the Internet

SOAP

- SOAP is typically used with the Web Service Description Language (WSDL).

- WSDL describes how to structure the SOAP request and response messages

Sample WSDL

```
<definitions name = "HelloService"
   targetNamespace = "http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns = "http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns = "http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

   <message name = "SayHelloRequest">
      <part name = "firstName" type = "xsd:string"/>
   </message>

   <message name = "SayHelloResponse">
      <part name = "greeting" type = "xsd:string"/>
   </message>

   <portType name = "Hello_PortType">
      <operation name = "sayHello">
         <input message = "tns:SayHelloRequest"/>
         <output message = "tns:SayHelloResponse"/>
      </operation>
   </portType>
```

Sample WSDL (cont'd)

```
<binding name = "Hello_Binding" type = "tns:Hello_PortType">
   <soap:binding style = "rpc"
      transport = "http://schemas.xmlsoap.org/soap/http"/>
   <operation name = "sayHello">
      <soap:operation soapAction = "sayHello"/>
      <input>
         <soap:body
            encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
            namespace = "urn:examples:helloservice"
            use = "encoded"/>
      </input>

      <output>
         <soap:body
            encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
            namespace = "urn:examples:helloservice"
            use = "encoded"/>
      </output>
   </operation>
</binding>
```

Sample WSDL (cont'd)

```
   <service name = "Hello_Service">
      <documentation>WSDL File for HelloService</documentation>
      <port binding = "tns:Hello_Binding" name = "Hello_Port">
         <soap:address
            location = "http://www.examples.com/SayHello/" />
      </port>
   </service>
</definitions>
```

Sample SOAP Request

```
POST /Quotation HTTP/1.0
Host: www.xyz.org
Content-Type: text/xml; charset = utf-8
Content-Length: nnn

<?xml version = "1.0"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
   SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">

   <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotations">
      <m:GetQuotation>
         <m:QuotationsName>MiscroSoft</m:QuotationsName>
      </m:GetQuotation>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

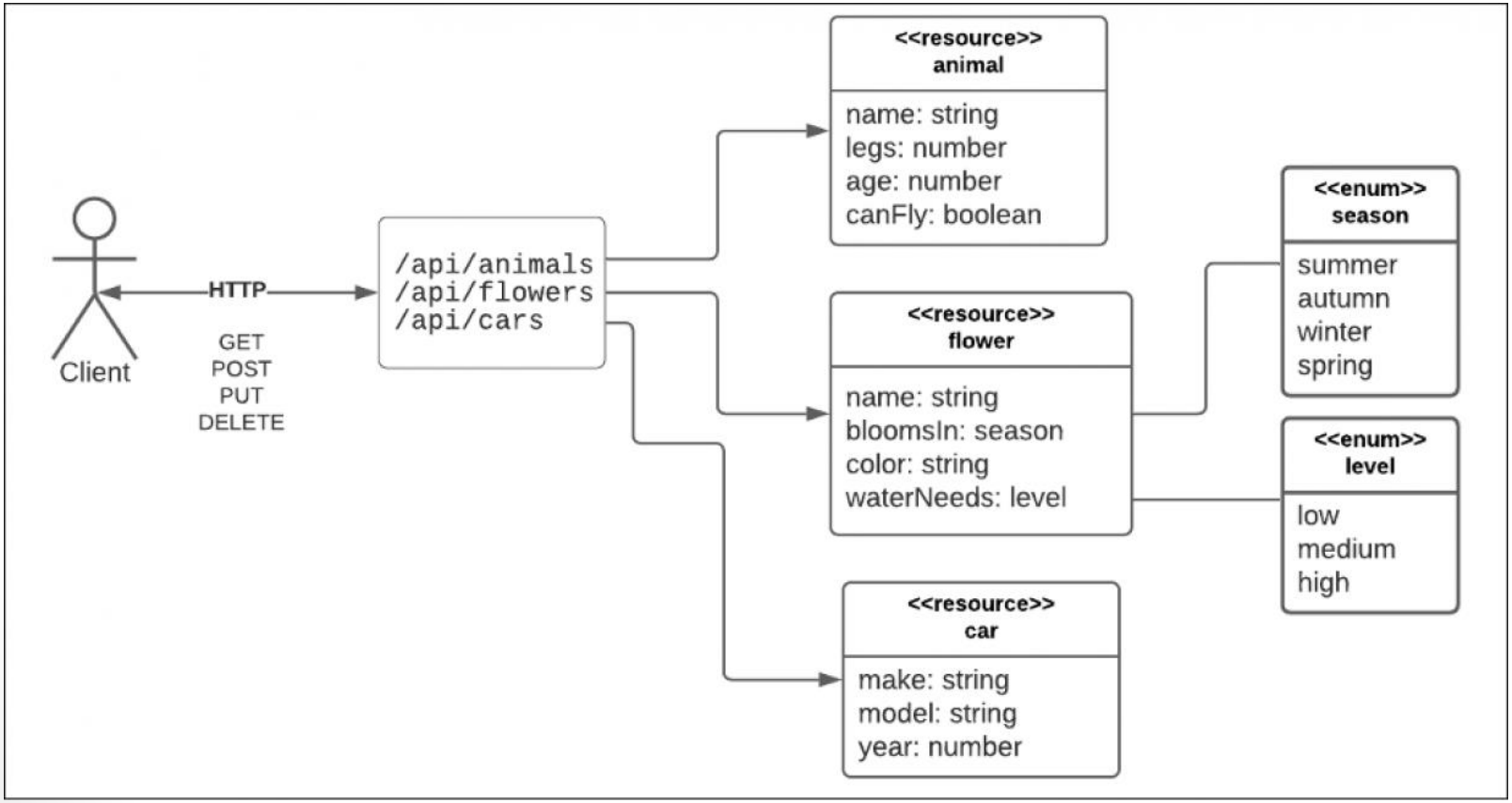
Sample SOAP Response

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset = utf-8
Content-Length: nnn

<?xml version = "1.0"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV = "http://www.w3.org/2001/12/soap-envelope"
   SOAP-ENV:encodingStyle = "http://www.w3.org/2001/12/soap-encoding">

   <SOAP-ENV:Body xmlns:m = "http://www.xyz.org/quotation">
      <m:GetQuotationResponse>
         <m:Quotation>Here is the quotation</m:Quotation>
      </m:GetQuotationResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

REST

- Representational State Transfer
- An architectural style devised by Roy Fielding in his 2000 Ph.D. thesis.
- Use the standard HTTP methods, GET, POST, PUT and DELETE, to query and mutate resources represented by URIs on the Internet.

REST


Client
HTTP
GET
POST
PUT
DELETE
/api/animals
/api/flowers
/api/cars
<<resource>>
animal
name: string
legs: number
age: number
canFly: boolean
<<resource>>
flower
name: string
bloomsIn: season
color: string
waterNeeds: level
<<resource>>
car
make: string
model: string
year: number
<<enum>>
season
summer
autumn
winter
spring
<<enum>>
level
low
medium
high

HATEOAS

- Hypermedia as the Engine of Application State
- a REST response can contain links that describe operations or followup workflow steps relevant to the given resource.

REST can use HATEOAS concept

- REST uses HATEOAS to define operations and workflow tasks that are relevant to a resource

```
{
  "car": {
    "vin": "KNDJT2A23A7703818",
    "make": "kia",
    "model": "soul",
    "year": 2010,
    "links": {
      "service": "/cars/KNDJT2A23A7703818/service",
      "sell": "/cars/KNDJT2A23A7703818/sell",
      "clean": "/cars/KNDJT2A23A7703818/sell"
    }
  }
}
```

Sample REST Request

```
GET /      HTTP/1.1
Host: https://api.github.com/

accept: text/html,image/webp,image/png
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9,fa;q=0.8
cache-control: no-cache
```

Sample REST Response


```
{
  "current_user_url": "https://api.github.com/user",
  "current_user_authorizations_html_url": "https://github.com/settings/connections/applications{/client_id}",
  "authorizations_url": "https://api.github.com/authorizations",
  "code_search_url": "https://api.github.com/search/code?q={query}{&page,per_page,sort,order}",
  "commit_search_url": "https://api.github.com/search/commits?q={query}{&page,per_page,sort,order}",
  "emails_url": "https://api.github.com/user/emails",
  "emojis_url": "https://api.github.com/emojis",
  "events_url": "https://api.github.com/events",
  "feeds_url": "https://api.github.com/feeds",
  "followers_url": "https://api.github.com/user/followers",
  "following_url": "https://api.github.com/user/following{/target}",
  "gists_url": "https://api.github.com/gists{/gist_id}",
  "hub_url": "https://api.github.com/hub",
  "issue_search_url": "https://api.github.com/search/issues?q={query}{&page,per_page,sort,order}",
  "issues_url": "https://api.github.com/issues",
  "keys_url": "https://api.github.com/user/keys",
  "label_search_url": "https://api.github.com/search/labels?q={query}&repository_id={repository_id}{&page,per_page}",
  "notifications_url": "https://api.github.com/notifications",
  "organization_url": "https://api.github.com/orgs/{org}",
  "organization_repositories_url": "https://api.github.com/orgs/{org}/repos{?type,page,per_page,sort}",
  "organization_teams_url": "https://api.github.com/orgs/{org}/teams",
  "public_gists_url": "https://api.github.com/gists/public",
  "rate_limit_url": "https://api.github.com/rate_limit",
  "repository_url": "https://api.github.com/repos/{owner}/{repo}",
  "repository_search_url": "https://api.github.com/search/repositories?q={query}{&page,per_page,sort,order}",
  "current_user_repositories_url": "https://api.github.com/user/repos{?type,page,per_page,sort}",
  "starred_url": "https://api.github.com/user/starred{/owner}{/repo}",
  "starred_gists_url": "https://api.github.com/gists/starred",
  "user_url": "https://api.github.com/users/{user}",
  "user_organizations_url": "https://api.github.com/user/orgs",
  "user_repositories_url": "https://api.github.com/users/{user}/repos{?type,page,per_page,sort}",
  "user_search_url": "https://api.github.com/search/users?q={query}{&page,per_page,sort,order}"
}
```

GraphQL

- GraphQL is a technology that came out of Facebook but is now open-source specification.
- The underlying mechanism for executing queries and mutations is the HTTP POST verb.
- GraphQL requests can be sent via HTTP POST or HTTP GET requests.

GraphQL

- as the name implies, GraphQL is intended to represent data in a graph.
- Instead of the columns and rows found in a relational database or the collection of structured documents found in a document-centric database such as MongoDB, a graph database is a collection of nodes and edges.

GraphQL Query

- Unlike REST, in which the caller has no control over the structure of the returned dataset (maybe just ‘fields’), GraphQL allows you to define the structure of the returned data explicitly in the query itself.

GraphQL Query

query	result
{ venues{ name city state_province } }	{ "data": { "venues": [{ "name": "Capitol Theater", "city": "West Mallie", "state_province": "New Hampshire" }, { "name": "Floydmouth Arena", "city": "Port Floydmouth", "state_province": "Alabama" }, { "name": "Symphony Hall", "city": "Blockborough", "state_province": "Alaska" }] } }

GraphQL Query

query	result

{
  venues{
  name
  postal_code
  }
}
	{
	  "data": {
	  "venues": [
	  {
	  "name": "Capitol Theater",
	  "postal_code": "33865"
	  },
	  {
	  "name": "Floydmouth Arena",
	  "postal_code": "05642-4932"
	  },
	  {
	  "name": "Symphony Hall",
	  "postal_code": "45651-1205"
	  }
	  ]
	  }
	}

GraphQL Request

- POST requests sent with the Content-Type header application/graphql must have a POST body content as a GraphQL query string.

```
query {
  getTask(id: "0x3") {
    id
    title
    completed
    user {
      username
      name
    }
  }
}
```

GraphQL Request

- POST requests sent with the Content-Type header application/json must have a POST body in the following JSON format:

```
{
  "query": "...",
  "operationName": "...",
  "variables": { "var": "val", ... }
}
```

GraphQL Request

- In GET requests, the query, variables, and operation are sent as URL-encoded query parameters in the URL.

```
http://localhost:8080/graphql?query={...}&variables={...}&operation=...
```

GraphQL Response

- The “data” field contains the result of your GraphQL request.
- The “extensions” field contains extra metadata for the request with metrics and trace information for the request.
- The “errors” field is a JSON list where each entry has a "message" field that describes the error.

Sample GraphQL Response

```
{
  "data": {
    "getTask": {
      "id": "0x3",
      "title": "GraphQL docs example",
      "completed": true,
      "user": {
        "username": "dgraphlabs",
        "name": "Dgraph Labs"
      }
    }
  }
}
```

Sample GraphQL Response


```
{
  "data": {
    "getTask": {
      "id": "0x3",
      "title": "GraphQL docs example",
      "completed": true,
      "user": {
        "username": "dgraphlabs",
        "name": "Dgraph Labs"
      }
    }
  },
  "extensions": {
    "touched_uids": 9,
    "tracing": {
      "version": 1,
      "startTime": "2020-07-29T05:54:27.784837196Z",
      "endTime": "2020-07-29T05:54:27.787239465Z",
      "duration": 2402299,
      "execution": {
        "resolvers": [
          {
            "path": [
              "getTask"
            ],
            "parentType": "Query",
            "fieldName": "getTask",
```

Sample GraphQL Response (cont'd)


```
            "parentType": "Query",
            "fieldName": "getTask",
            "returnType": "Task",
            "startOffset": 122073,
            "duration": 2255955,
            "dgraph": [
              {
                "label": "query",
                "startOffset": 171684,
                "duration": 2154290
              }
            ]
          }
        ]
      }
    }
  }
}
```

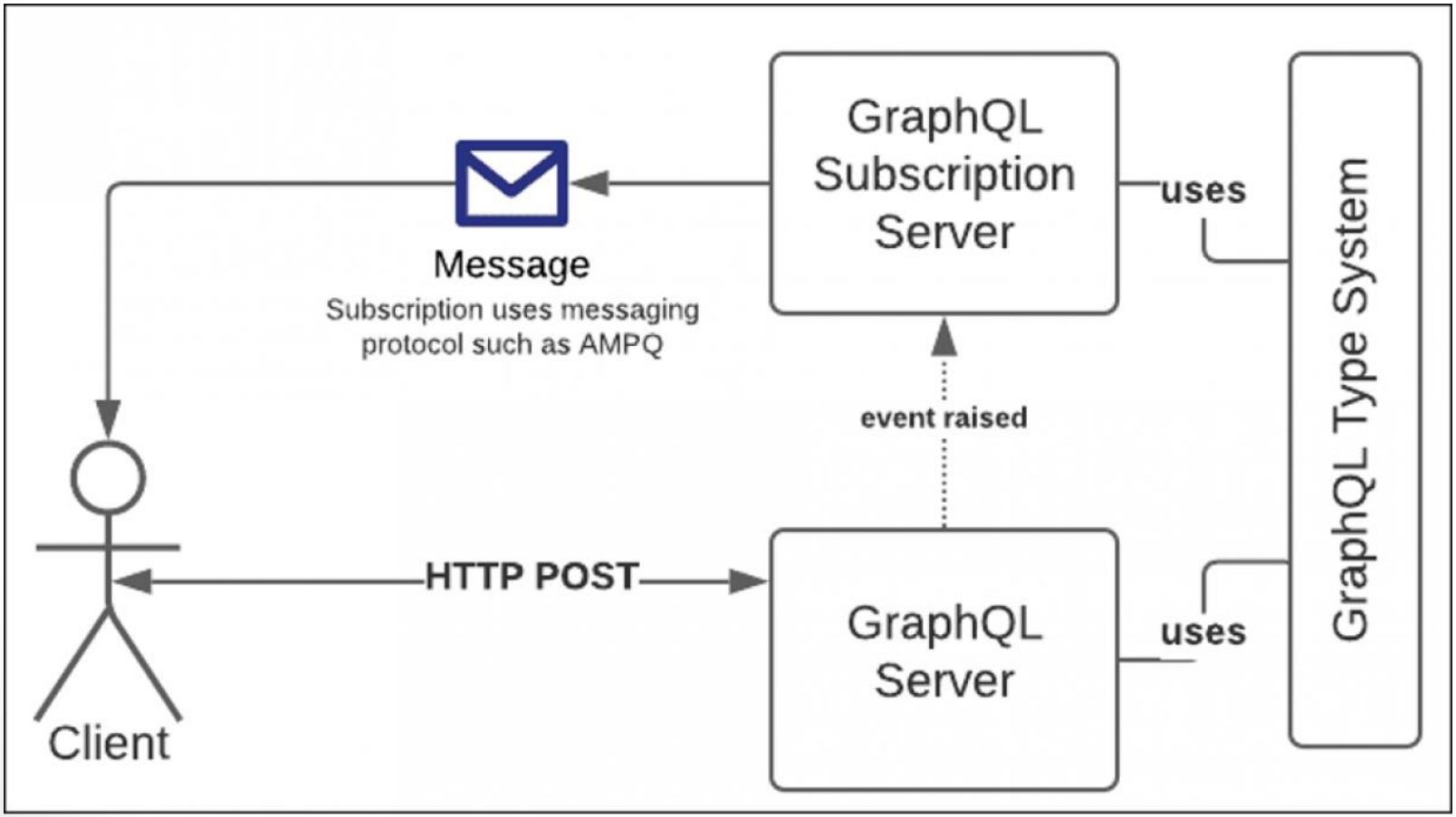
Sample GraphQL Response


```
{
  "errors": [
    {
      "message": "Field \"getTask\" argument \"id\" of type \"ID!\" is required but not provided.",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ]
    }
  ]
}
```

GraphQL Subscriptions

- Opens the door to asynchronous messaging.
- Query and mutation data exchange under GraphQL is synchronous due to the request-response pattern inherent in the HTTP/1.1 protocol.
- However, GraphQL allows users to receive messages asynchronously when a specific event is raised on the server-side

GraphQL Subscriptions



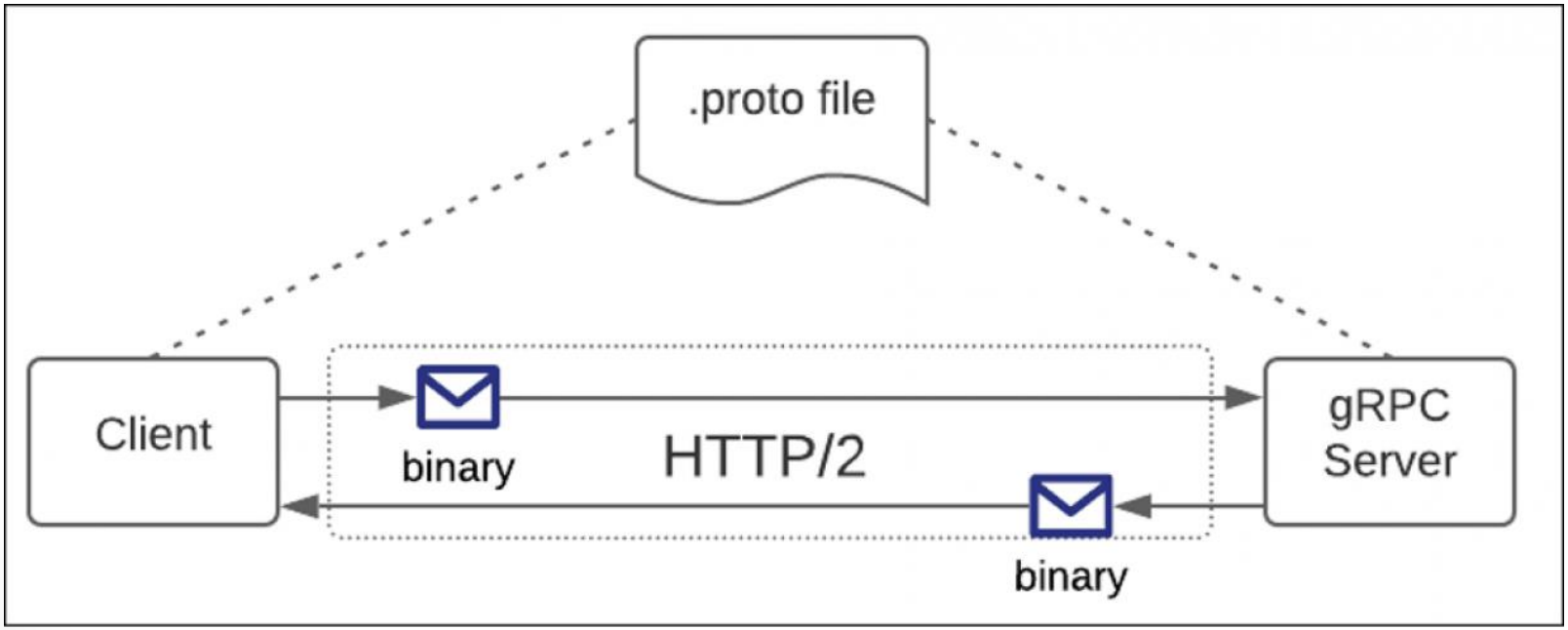
gRPC

- A data exchange technology developed by Google and then later made open-source.
- Like GraphQL, it's a specification that's implemented in a variety of languages.
- Unlike REST and GraphQL, which use text-based data formats, gRPC uses binary format (increases performance)

gRPC Protocol Buffers

- gRPC uses the Protocol Buffers binary format.
- Both the client and server in a gRPC data exchange shall have access to the same schema definition
- By convention, a Protocol Buffers definition is defined in a .proto file.
- The .proto file provides the "dictionary" by which data is encoded and decoded to and from the Protocol Buffers binary format.

gRPC



gRPC and HTTP/2

- In addition to using Protocol Buffers to encode data and thus increase performance, gRPC has another benefit.
- It supports bidirectional, asynchronous data exchange. This is because gRPC is based on the HTTP/2 protocol.

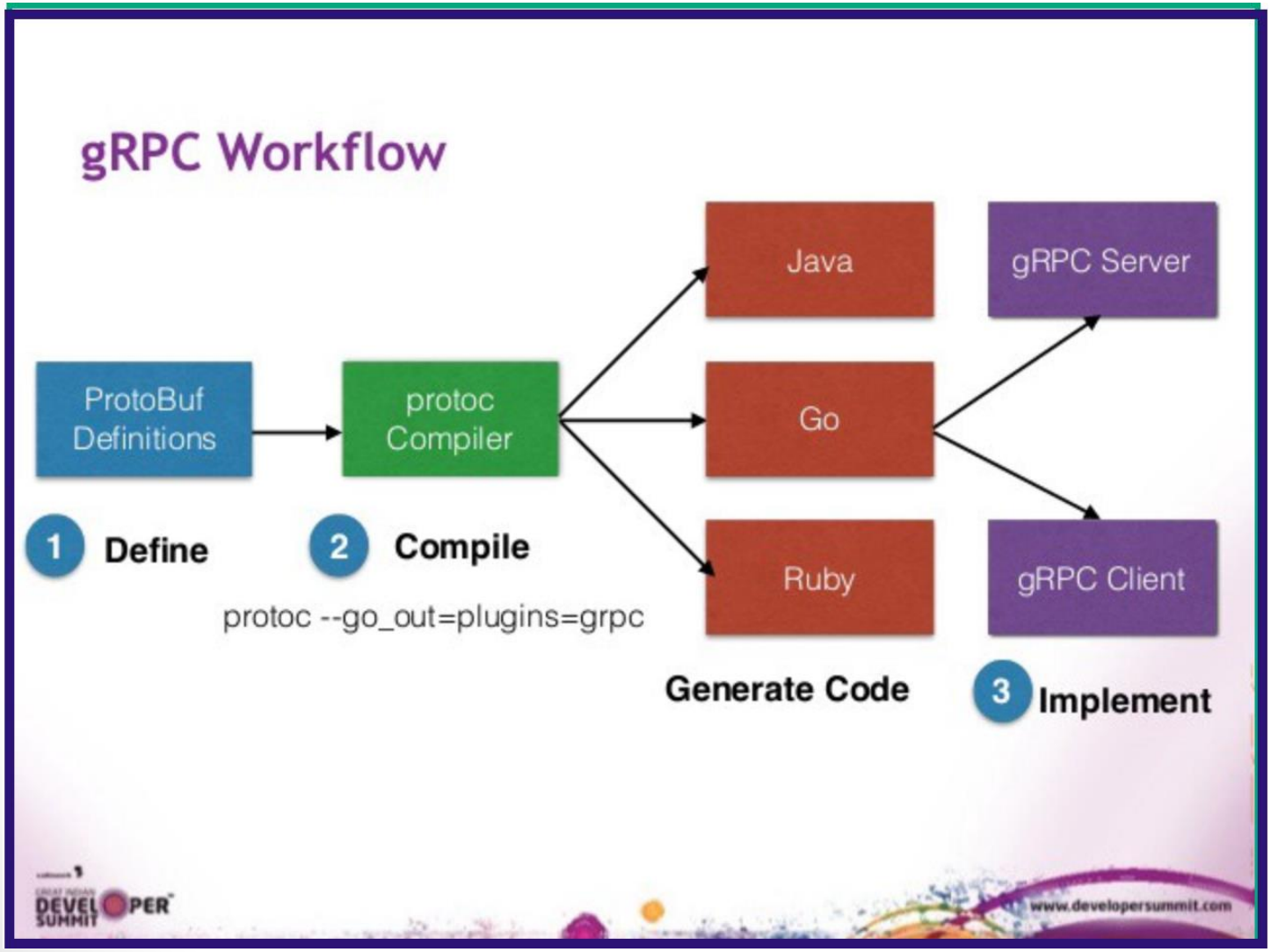
HTTP/2

- Unlike HTTP/1.1, which supports only a request-response interaction over a single connection, HTTP/2 supports any number of requests and responses over a single connection.
- Connections can also be bidirectional.
- under HTTP/2, a client opens a connection to a target server, and that connection stays open until either the client or server closes it.
- gRPC allows data streams as well. The steam can emanate from the client or from the server.

gRPC schema in protocol buffer language format

```
1   //defines an Animal response
2   message AnimalResponse {
3       Animal animal = 1;
4   }
5
6   //defines cars returned as a single array
7   message CarResponse {
8       repeated Car car = 1;
9   }
10
11
12  service SimpleService {
13      rpc GetAnimals () returns (stream AnimalResponse) {
14      }
15
16      rpc GetFlowers () returns (stream FlowerResponse) {
17      }
18
19      rpc GetCars (SearchCarRequest) returns (CarResponse) {
20      }
21  }
```

gRPC workflow



Sample ProtoBuff Message

```
message Test1 {
  optional int32 a = 1;
}
```

- In an application, you create a Test1 message and set a to 150. You then serialize the message to an output stream. If you were able to examine the encoded message, you'd see three bytes:

```
08 96 01
```

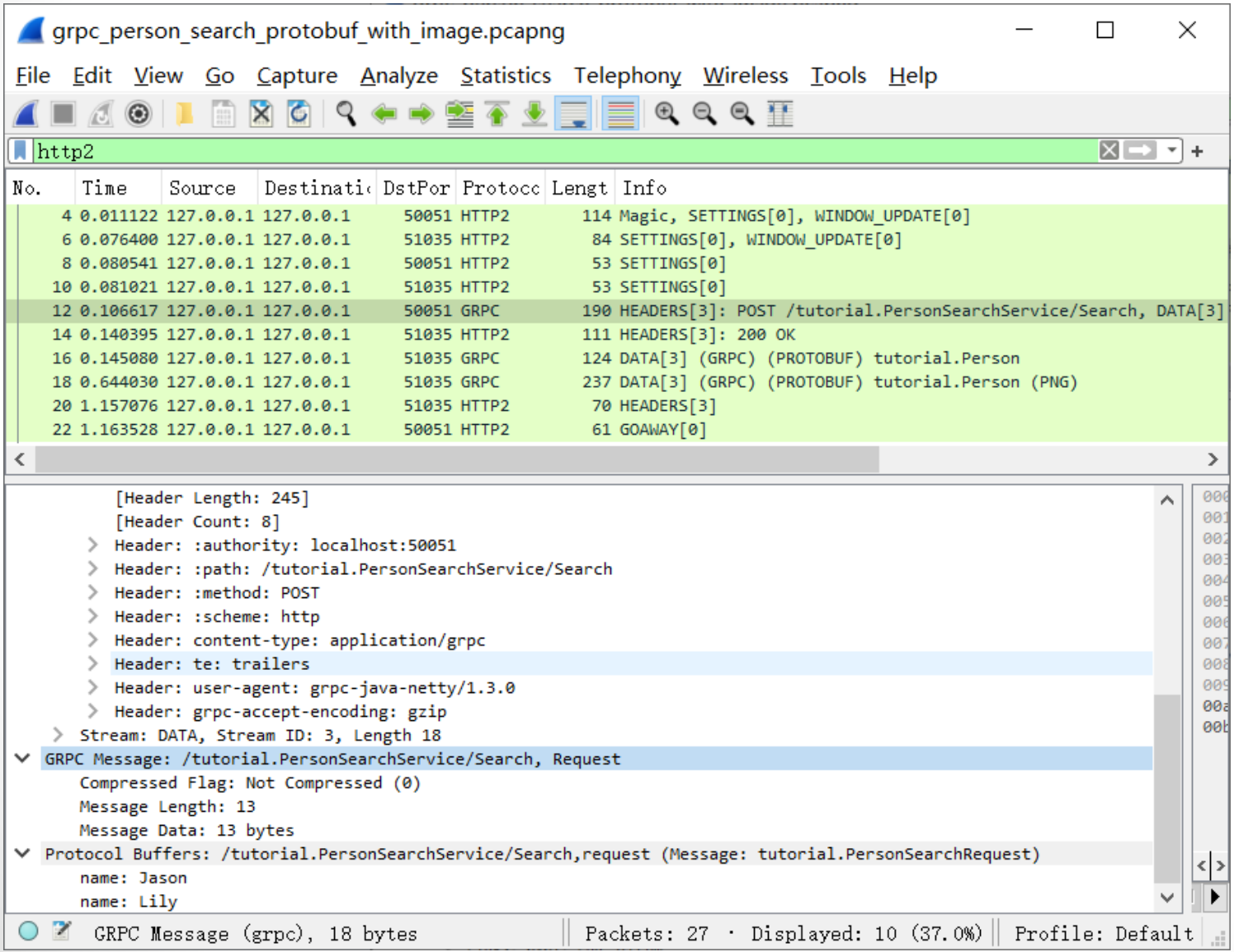
Sample ProtoBuff Message

```
message Test4 {
  repeated int32 d = 4 [packed=true];
}
```

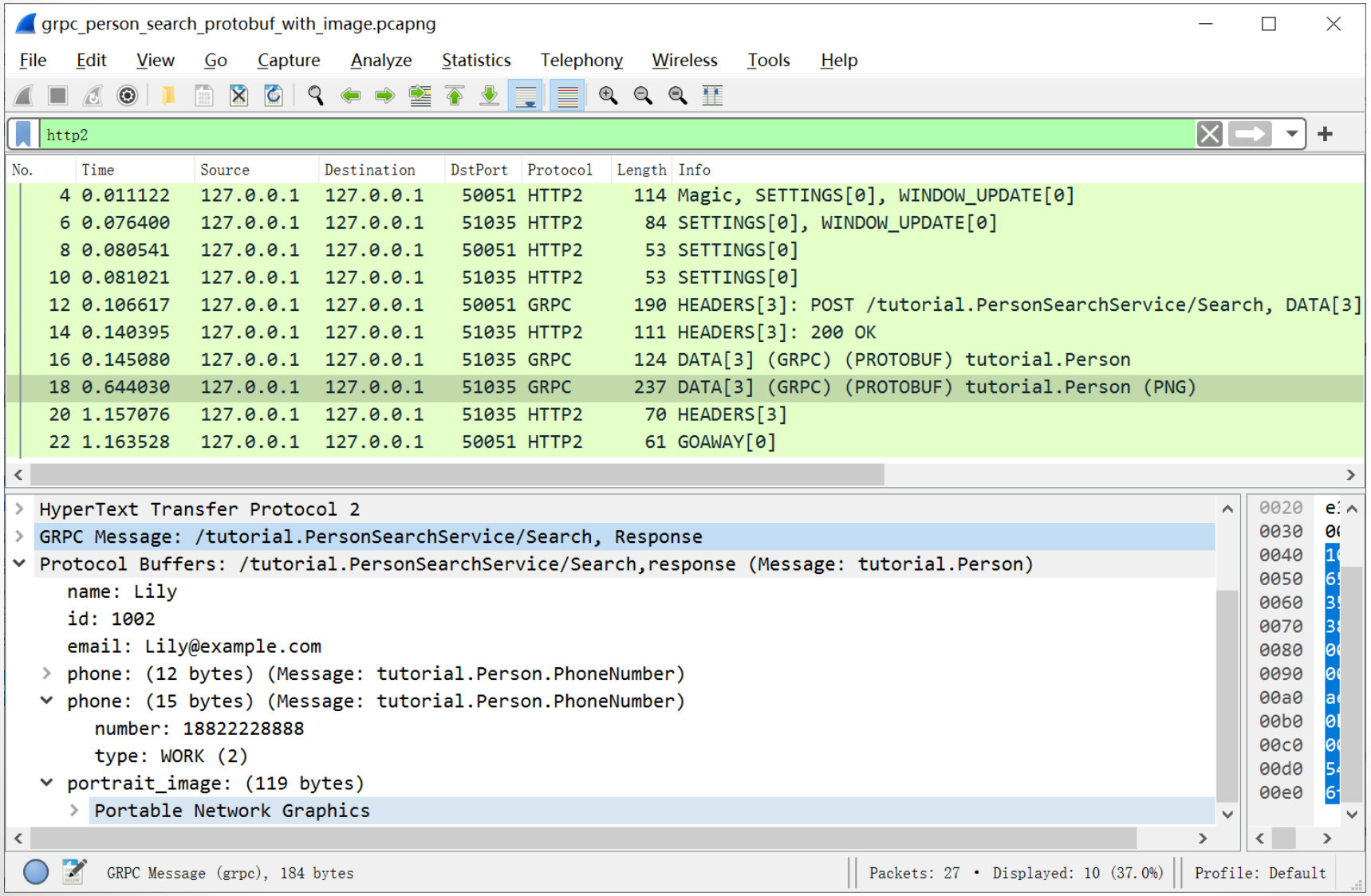
- Now let's say you construct a Test4, providing the values 3, 270, and 86942 for the repeated field d. Then, the encoded form would be:

```
22        // key (field number 4, wire type 2)
06        // payload size (6 bytes)
03        // first element (varint 3)
8E 02     // second element (varint 270)
9E A7 05  // third element (varint 86942)
```

Sample gRPC Request



Sample gRPC Response



API Communication

- a REST client written in Go can communicate with a REST server written in Node.JS. Or, you can execute a query or mutation from the curl command.
- Same goes for GraphQL, gRPC, and SOAP

Which API Format?

SOAP: Pros

- SOAP can be implemented using a variety of protocols, not only HTTP but SMTP and FTP as well.
- SOAP supports discovery via WSDL and it's language agnostic.
- SOAP has been around for a while. There is still a good deal of legacy SOAP implementations that need to be maintained.

SOAP: Cons

- SOAP can be considered a complex message format with a lot of ins and outs to the specification.
- The verbose nature of XML which is the format upon which SOAP is based, coupled with the reliance on external namespaces to extend the basic message format makes the protocol difficult to manage.
- SOAP messages can get quite large.
- Moving bulky, text based, SOAP messages between source and target takes a long time in comparison to binary messaging protocols such as gRPC

SOAP: Cons

- SOAP is a legacy protocol. While there's a lot of maintenance work to be done with those systems that use it, new architectures are taking a more modern approach to inter-service communication.

REST: Pros

- REST is simple, well-known, and widely used.
- You make a call on a resource represented by a URL on the Internet using an HTTP verb and get a response back in JSON or XML.
- Productivity under REST is almost immediate.

REST: Cons

- REST is immutable in terms of the data structure of a response.
- Given the response/response aspect of HTTP/1.1, REST can be slow.

GraphQL: Pros

- GraphQL is flexible and growing in popularity.
- The latest version of GitHub's API is published using GraphQL. Yelp publishes its API in GraphQL, as does Shopify. The list continues to grow.
- The GraphQL specification covers every aspect of API implementation, from Scalars, Types, Interfaces, Unions, Directives, …

GraphQL: Cons

- QraphQL is complex and hard to implement. While the specification allows for customization, the basic framework cannot be avoided. You have to do things according to the GraphQL way.
- REST, on the other hand, has a limited rule set to follow.

GraphQL vs REST

- It's the difference between making a skateboard and making an automobile. No matter what, you need four wheels as well as a way to start and stop, but a skateboard (REST) is far easier to make and operate than an automobile (GraphQL).
- It's a question of tradeoffs and making sure the benefits of use outweigh the cost of implementation.
- Once GraphQL is implemented, users find it a better developer experience than REST.

gRPC: Pros

- gRPC is exact and wicked fast.
- It's become a de facto standard for inter-service data exchange on the backend.
- Bidirectional streaming capabilities that are provided by HTTP/2 allow gRPC to be used in situations where REST or GraphQL can't even be considered.

gRPC: Cons

- Both client and server need to support the same Protocol Buffers specification. This is a significant undertaking in terms of version control.
- Under REST or GraphQL, one can add a new attribute(s) to a resource (REST) or type (GraphQL) without running much risk of breaking the existing code. Making such additions in gRPC can have a detrimental impact. Thus, updates to the .proto file need to be carefully coordinated.

gRPC: Cons

- Another challenge is that HTTP/2 does not have universal support for public-facing client-server interactions on the Internet.
- Not all websites on the Internet support HTTP/2.

gRPC: Cons

- gRPC is that it takes time to attain mastery.
- Some time can be saved by using the protoc tool. protoc will auto-generate gRPC client and server code according to a particular programming language based on a specific .proto file.
- It's useful for creating boilerplate code, but doing more complex programming requires a lot more work.

gRPC as a Backend Technology

- gRPC is best suited to situations where developers control both client and server data exchange activities. Typically such boundaries exist on the backend. Hence, the prominence of gRPC as a backend technology.

gRPC: Performance over Flexibility

- gRPC is a very particular API format that provides lightning-fast execution at the expense of flexibility.
- Yet, if you have an application in which nanoseconds count, gRPC includes speed that is hard to match when using REST or GraphQL.

Any Questions?

Your time is limited, don't waste it living someone else's life

Steve Jobs, Stanford University speech, 2005