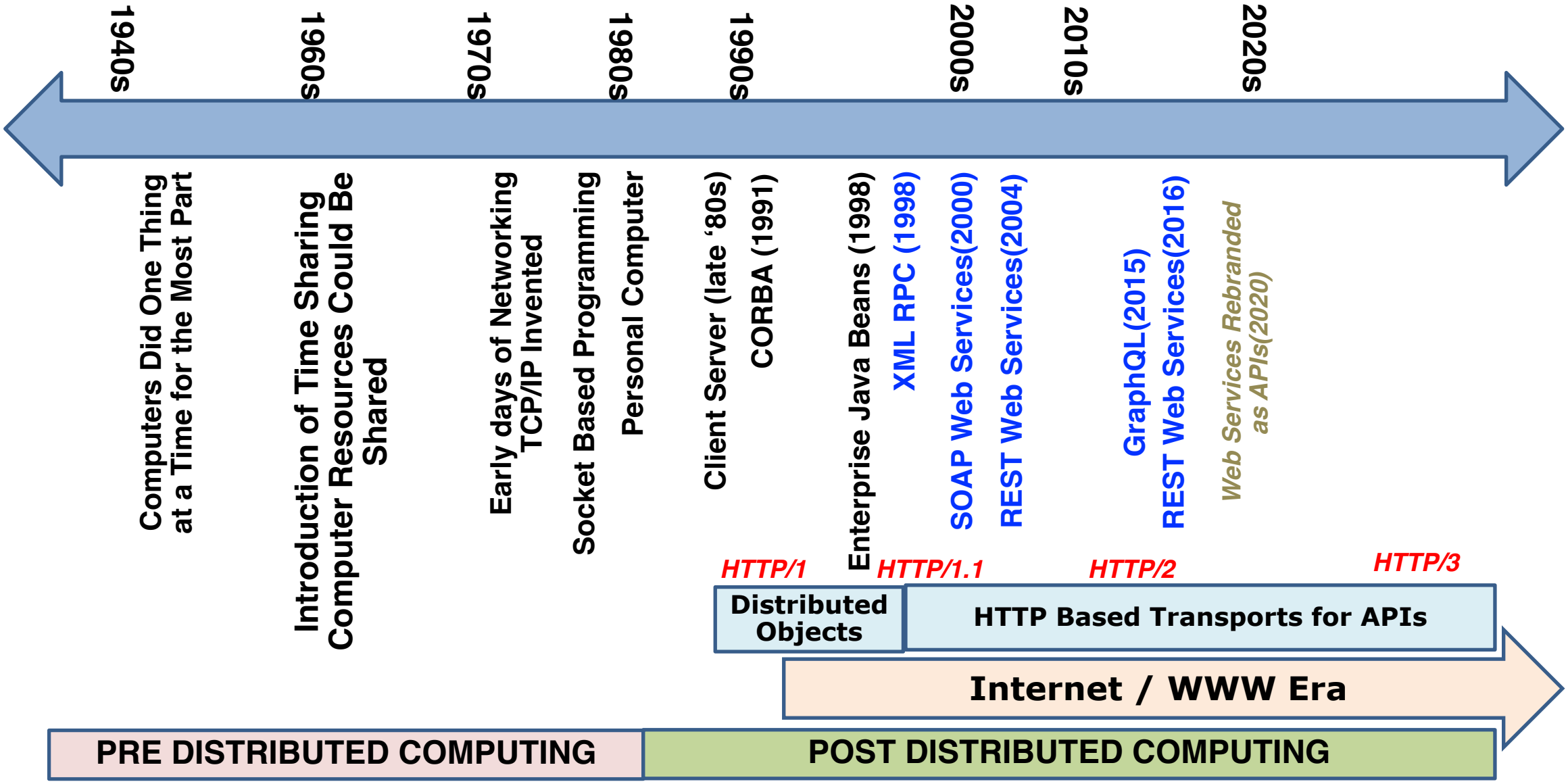
SE 577
Software Architecture

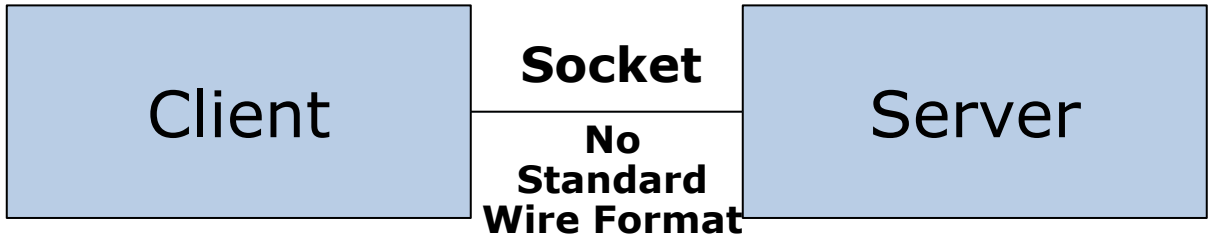
API Architecture

A historical context

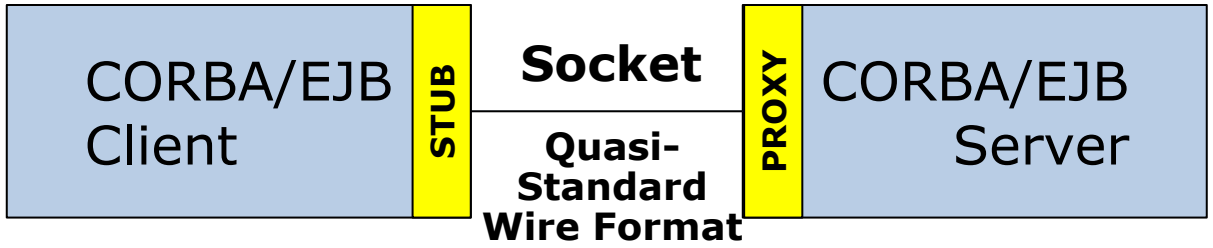


Versions of Distributed Client/Server Interoperability

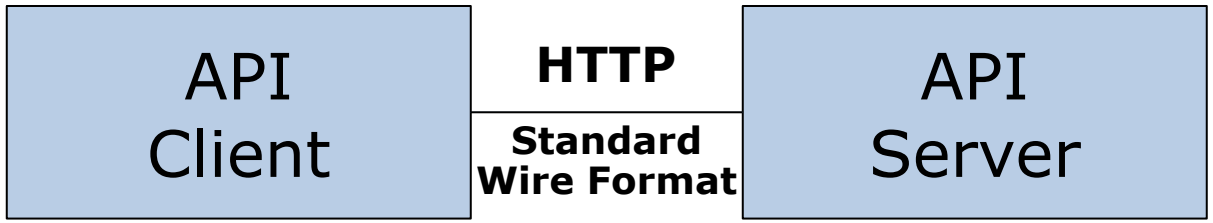
Roll your own Distributed Program



Distributed Objects

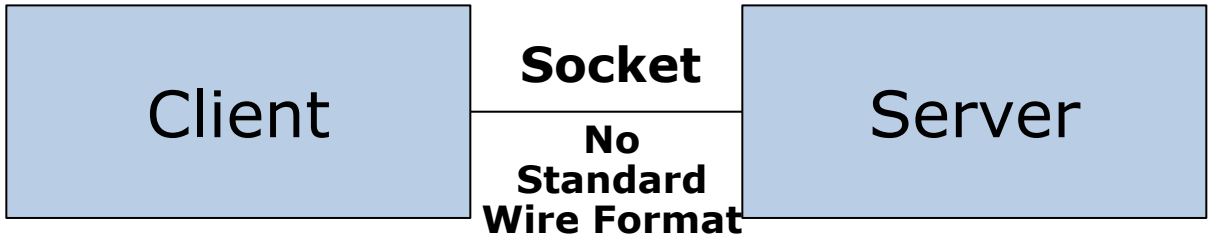


Client/Server Over Web Protocols



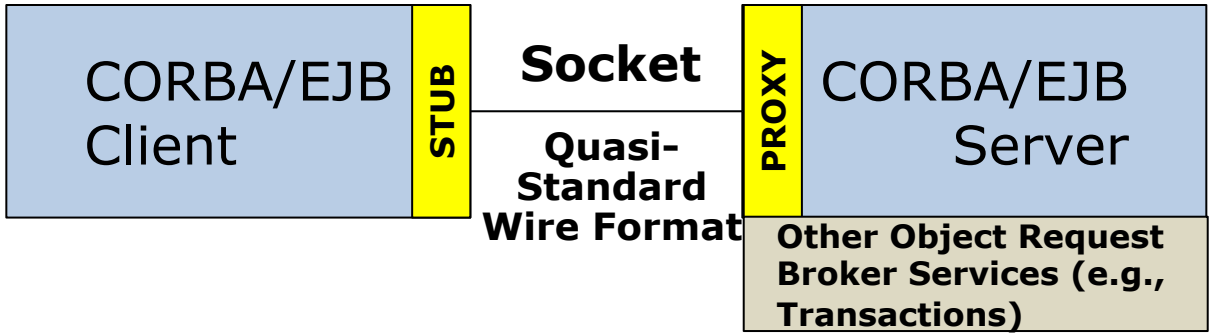
Architecture Challenges addressed with modern Web-Based APIs

Roll your own Distributed Program



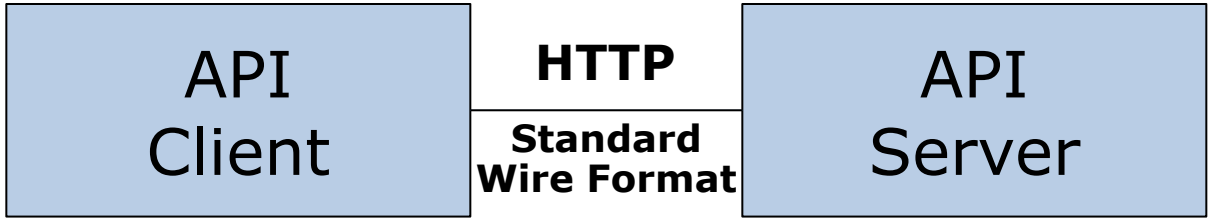
- No tooling, develop at a very low level
- Client and server very tightly coupled because of no OSI standard layer 7 protocol
- No support for resilience and scale

Distributed Objects



- Provides some benefits, tooling to serialize/deserialize objects
- CORBA multi-language, EJB just Java
- Provides higher level lifecycle services such as transactions
- Interoperability was largely vendor specific / many didn't follow standards
- Vendor specific ways to manage resilience and scale

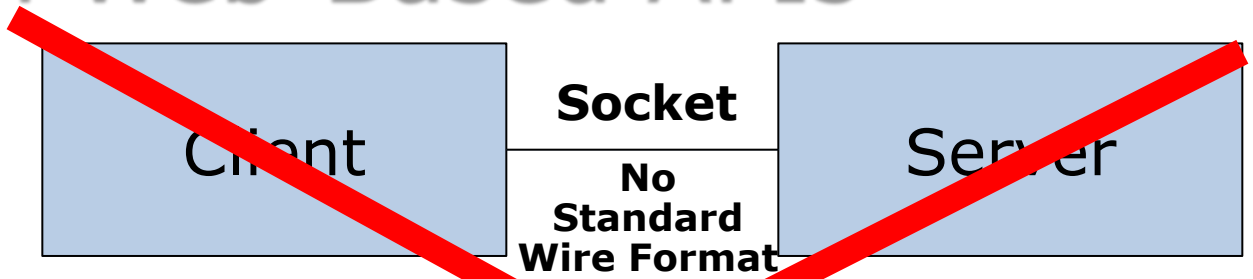
Client/Server Over Web Protocols



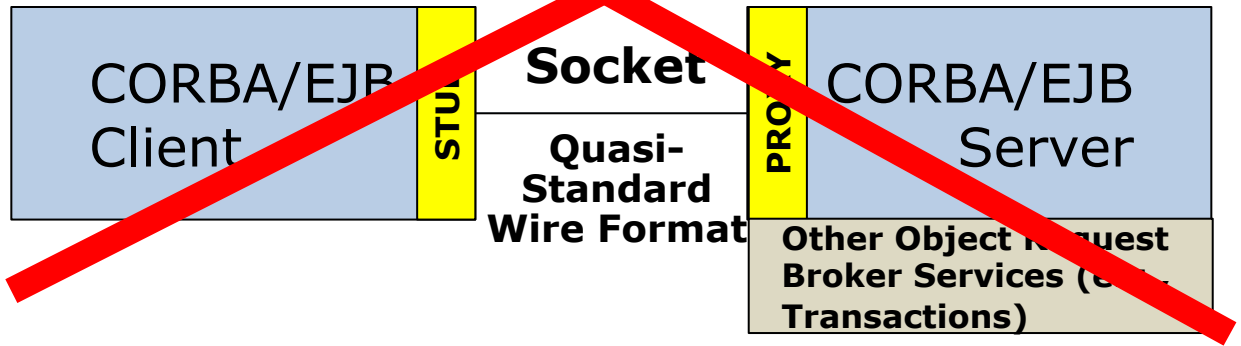
- APIs ride over all of the innovations and scale that the internet provides
- APIs are interoperate because the HTTP protocol is consistently implemented
- Vendor differentiation moved to tooling and injecting API specific features into middleware that don't impact interoperability (e.g., Load Balancing, Traffic Routing, etc)

Architecture Challenges addressed with modern Web-Based APIs

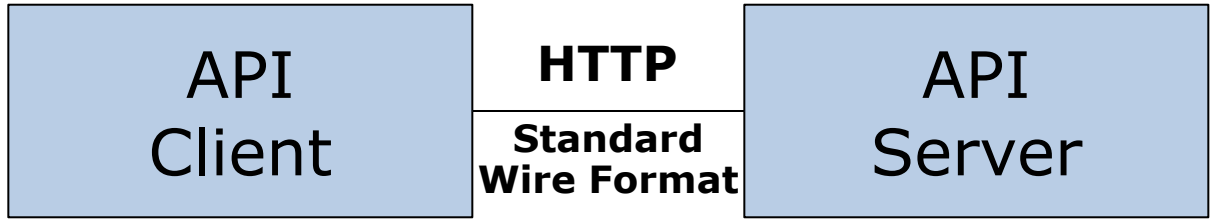
Roll your own Distributed Program



Distributed Objects



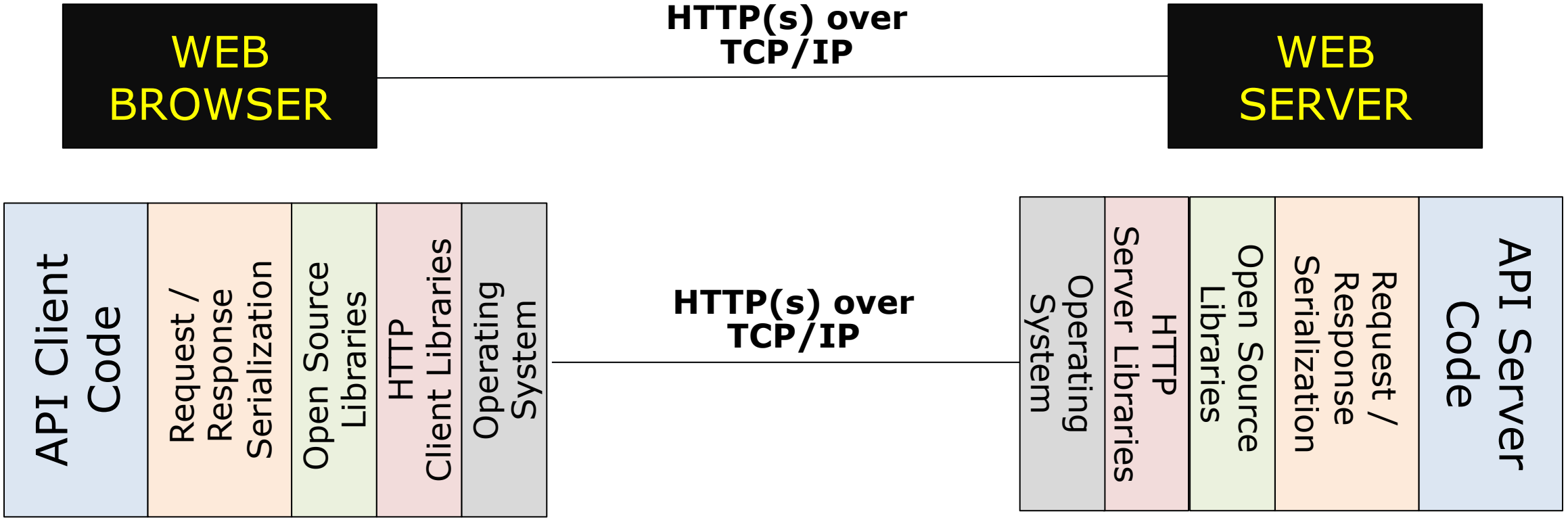
Client/Server Over Web Protocols



THESE ARE INTERESTING HISTORICALLY BUT WE WILL NOT BE DISCUSSING IN CLASS

OUR AREA OF FOCUS

The Primary Architecture Pattern of Modern APIs is Client/Server and Layered



If we consider the basic web architecture as a black box (top), we can see that the architecture of APIs basically can ride on top. Think about all of the innovations over the past 10 years to deploy, manage, scale and run web applications, APIs get all of the benefits out-of-the-box. As well as future ones that may come later.

Lets start by taking a look at network protocols – Why?

- Network protocols have interesting architectures
- Understanding aspects of network protocols will lead to a deeper foundational understanding of web services and APIs
- This material will also help with some other content covered in this class – web architectures, cloud native architectures, reactive architectures in particular

Modern APIs take advantage of the OSI Model from the 1970s – a Layered Architecture

7. Application	• End User layer • HTTP, FTP, IRC, SSH, DNS
6. Presentation	• Syntax layer • SSL, SSH, IMAP, FTP, MPEG, JPEG
5. Session	• Synch & send to port • API's, Sockets, WinSock
4. Transport	• End-to-end connections • TCP, UDP
3. Network	• Packets • IP, ICMP, IPSec, IGMP
2. Data Link	• Frames • Ethernet, PPP, Switch, Bridge
1. Physical	• Physical structure • Coax, Fiber, Wireless, Hubs, Repeaters

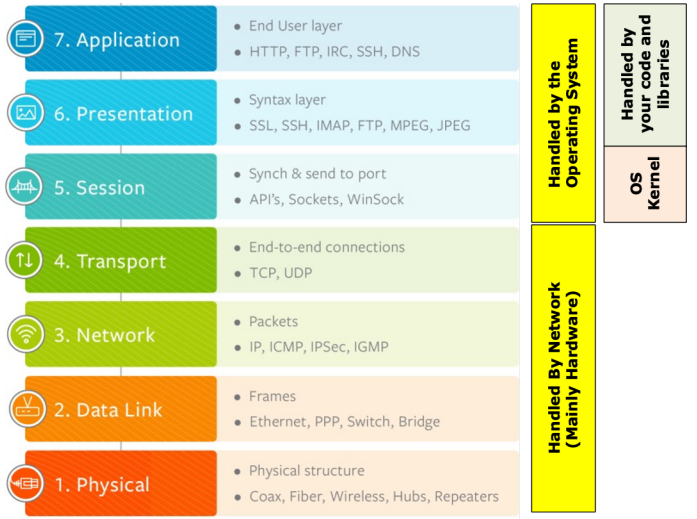
Handled by the Operating System

Handled by your code and libraries

OS Kernel

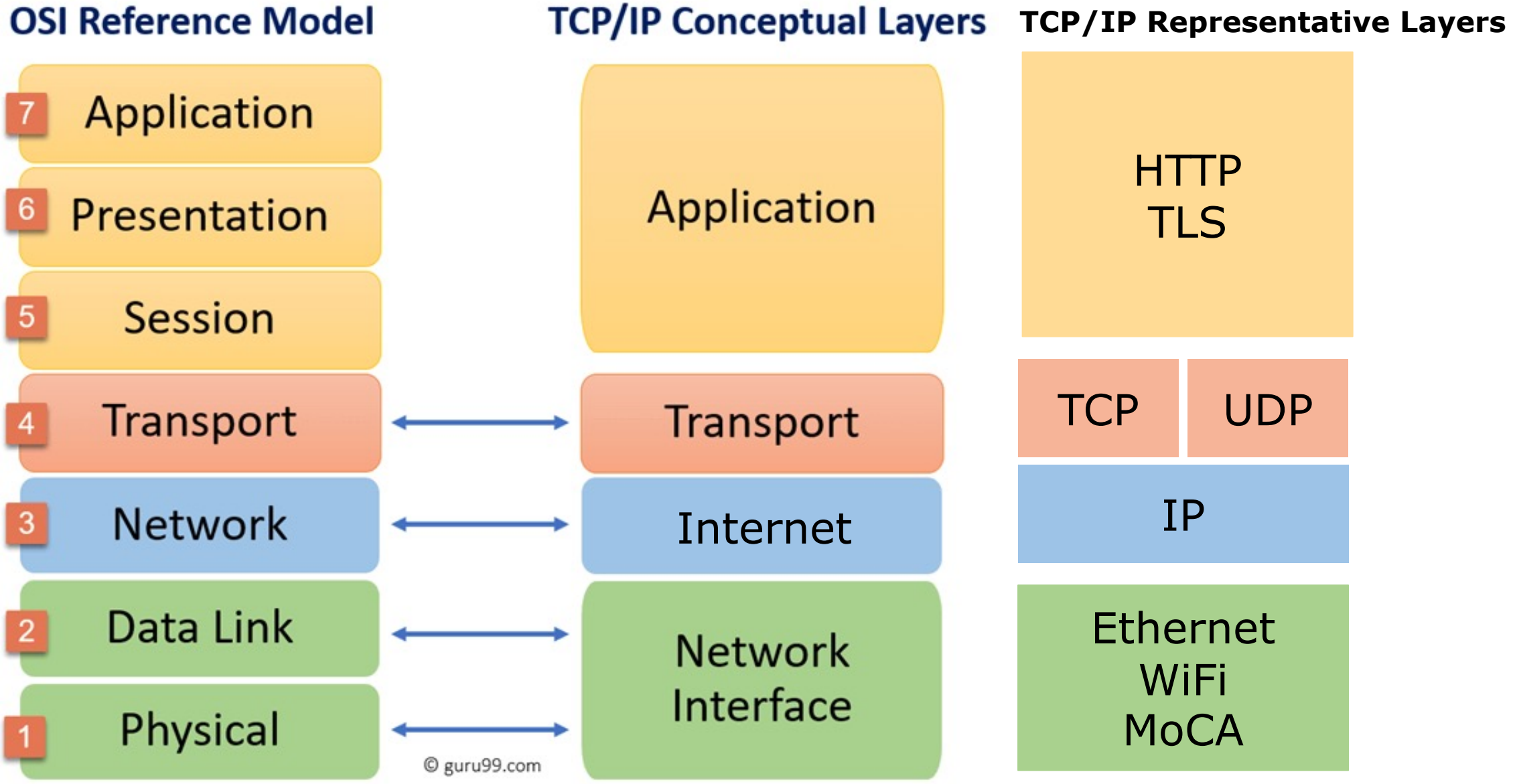
Handled By Network (Mainly Hardware)

API Clients & Servers



OSI architecture mapped to TCP/IP and UDP/IP

OSI is a REFERENCE ARCHITECTURE; TCP/IP is a concreate architecture



TCP vs UDP (this will be helpful later)

TCP

- **Rides on top of IP – IP Routes packets**
- **Designed in the 1970s, adopted as a standard in the 1980s**
- **TCP is a connection-oriented protocol –aka Connections are stood up, used and torn down**
- **TCP provides a variety of different services**
  - Reliable delivery
  - Congestion control
  - Messages delivered in order

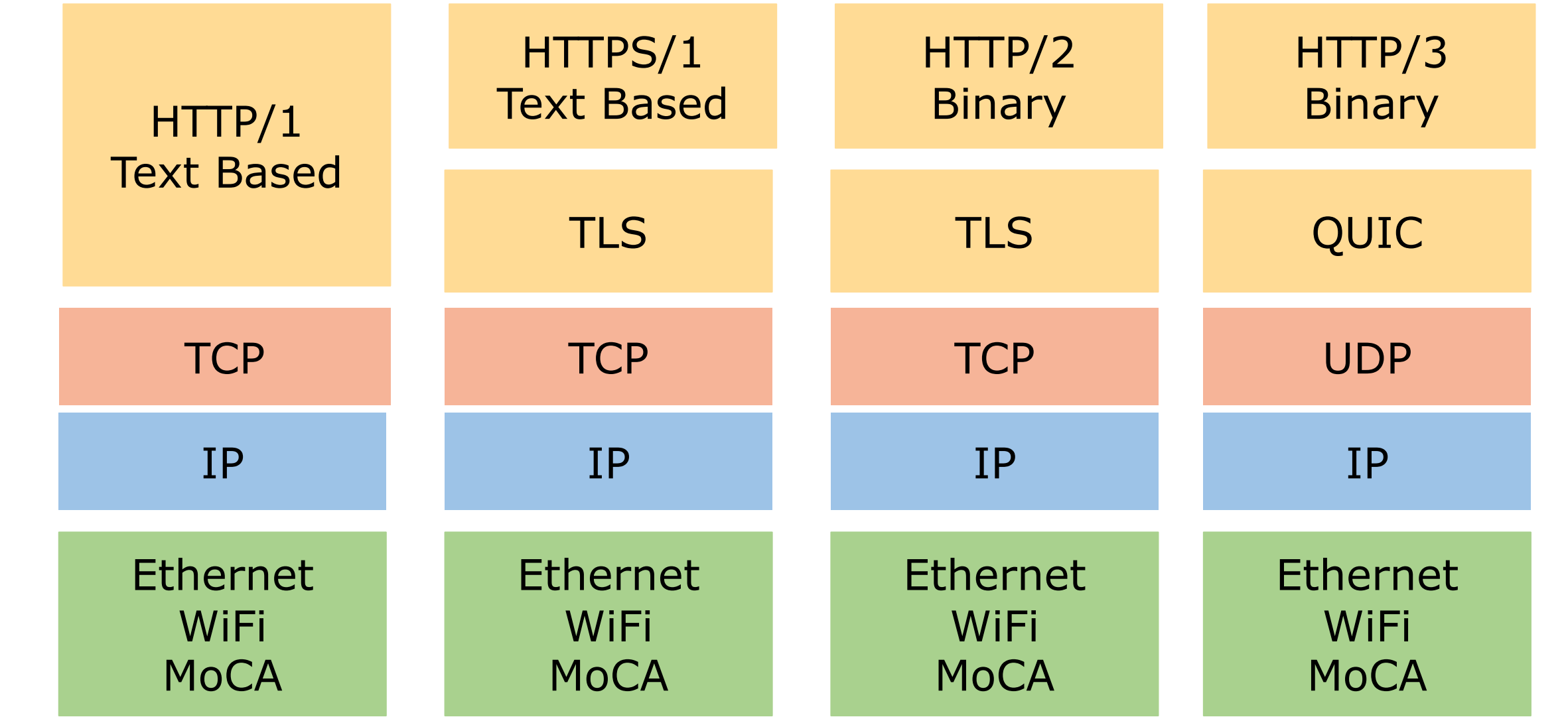
UDP

- **Like TCP, Rides on top of IP – IP Routes packets**
- **Designed in the 1970s, standardized in 1980**
- **TCP is a connectionless protocol –aka Connection setup is not required, you just send messages**
- **Since its connectionless, its significantly faster than TCP, and includes other features not found in TCP, and supports use cases that don't require TCPs features**
  - Due to efficiency, many use cases can absorb packet loss – e.g., video streaming
  - Support for broadcast and multicast, TCP is generally peer to peer

In many cases the performance benefits of UDP don't matter, the main place TCP differs from UDP is connection setup. TCP connection setup requires 3 messages, TCP with TLS connection setup requires 8 messages. UDP has no connection setup requirement!

APIs are very heavy in connection setup and tear down!

HTTP has variations V1.1, 2, and 3 (emerging)



1997
Note HTTP/1 is really version 1.1 – 1.0 deprecated

1997

2015

Not officially Released – Reference Implementations Exist

HTTP Protocol - L7 of OSI; App layer of TCP/IP

HTTP VERB
PARAMETERS
PROTOCOL

GET https://api.github.com/users/ArchitectingSoftware HTTP/1.1

The current HTTP verb set is GET, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS, and TRACE

HEADERS

Headers are key/value pairs that provide metadata on what is being request, or what is being retuned. Example: "content-length: 350". See https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers for details.

Note you can add custom headers but they should start with x- for example "x-course-id: se577"

Payload

HTTP itself says nothing about its payload structure and format. The payload can be HTML, JSON, etc

HTTP/1.1 specifies that the payload is not binary, so if binary data is sent over HTTP/1.1 it must be base-64 encode.

HTTP/2 and HTTP/3 use binary formats for the payload which are more efficient to send over a network.

Also note the https:// in the above example. With HTTP/1.1 sending data encrypted is optional so HTTPS indicates secure where HTTP is just plain text. With HTTP/2 and HTTP/3, https is the only option, as non encrypted headers and payloads are not supported

HTTP Protocol Versions Can be Negotiated Client Requested Example

One cool feature of HTTP is that via headers and status codes, HTTP protocol versions can be negotiated between clients and servers. HTTP status codes are well defined in the standard. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
-> GET foo.com HTTP/1.1

<- Status-Code: 200/OK
```

Simple exchange, server supports protocol requested by client

```
-> GET foo.com HTTP/1.1
   Upgrade: http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client requests server to upgrade to http/2, server indicates that upgrade is OK

```
-> GET foo.com HTTP/1.1
   Upgrade: http/3, http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client requests server to upgrade and gives server a preferred list of protocols, server picks the one it wants and returns it

HTTP Protocol Versions Can be Negotiated Server Enforced Example

One cool feature of HTTP is that via headers and status codes, HTTP protocol versions can be negotiated between clients and servers. HTTP status codes are well defined in the standard. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
-> GET foo.com HTTP/1.1


<- Status-Code: 426/Upgrade-Required
   Upgrade: http/3, http/2

-> Get foo.com HTTP/1.1
   Upgrade: http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

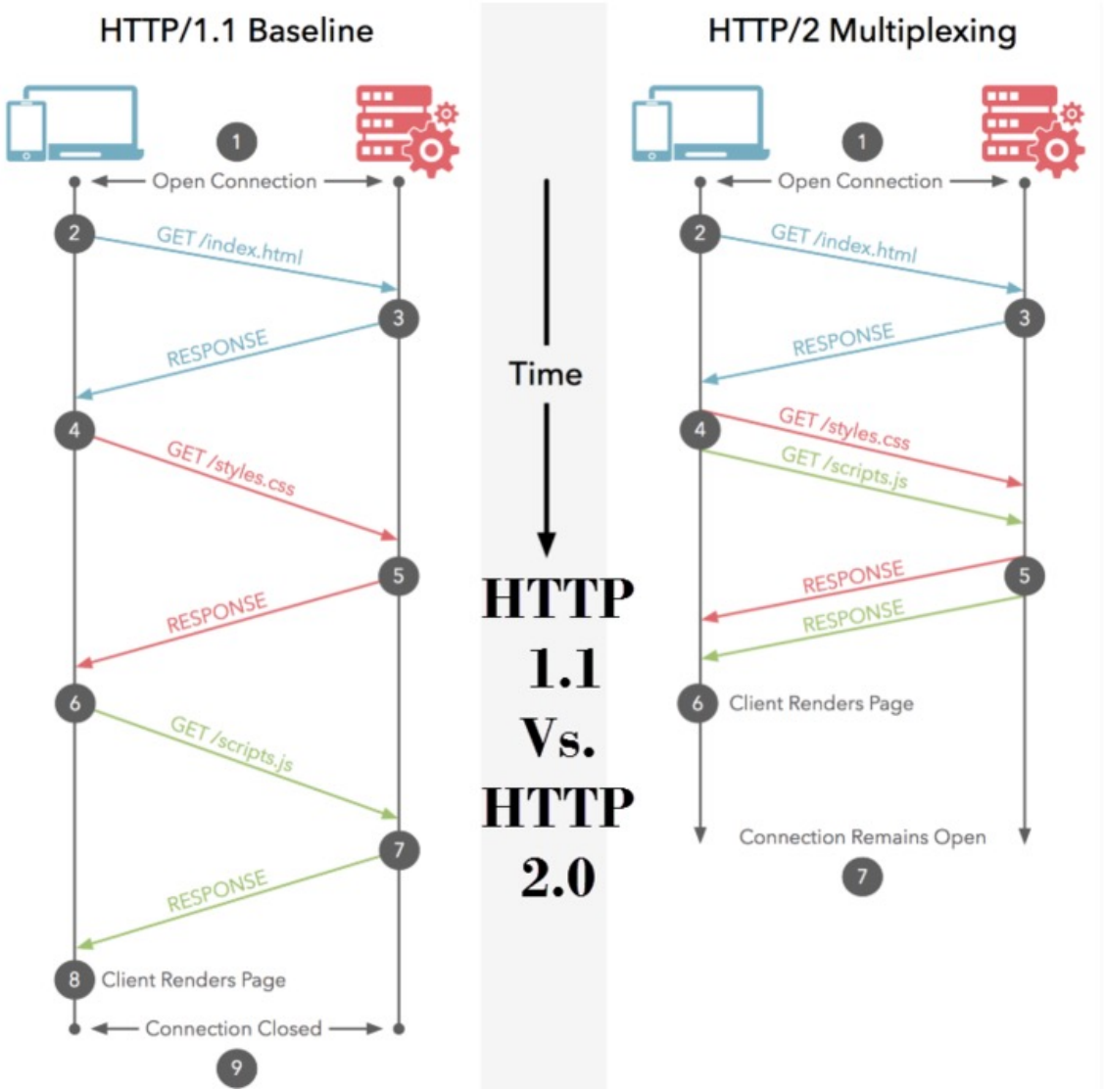
Client wants to use 1.1, but server responds that it supports http/3 and http/2 with http3 being preferred

Client then initiates protocol transfer to what it wants, http/2 in this case

```
-> GET foo.com HTTP/1.1


<- Status-Code: 426/Upgrade-Required
   Upgrade: http/2

-> Get foo.com HTTP/1.1
   Upgrade: http/3
   Connection: Upgrade

<- Status-Code: 505/HTTP-Version-Not-Supported
   Upgrade: http/2, http/1.1
```

The client is trying to upgrade to http/3 but the server does not support it, so it returns a 505 with a list of protocols it supports in preferred order

HTTP Evolution – 1.1 to 2.0



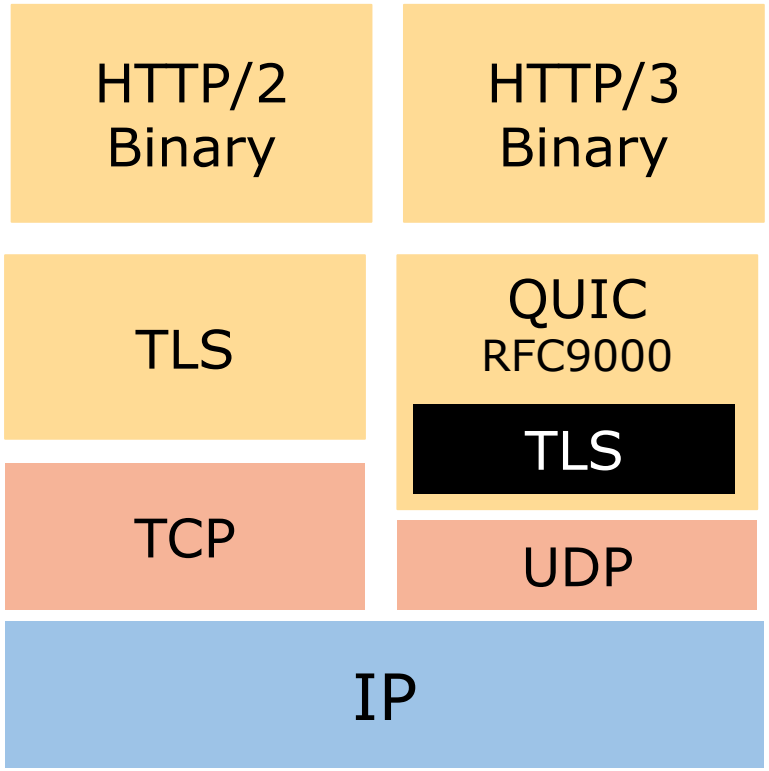
HTTP/1.1 sets up and tears down a TCP/IP connection for each request; HTTP/2.0 sets up a persistent connection and keeps it open – big benefits:

- Consider a web page or making multiple API calls, TCP setup is expensive
- Allows requests to be **multiplexed** (see picture on left) vs making one request at a time serially

HTTP/2.0 is a binary protocol, which is more efficient to send over the wire than text, which is the HTTP/1.1 standard

HTTP/2 only supports encrypted traffic; this is optional in HTTP/1.1 via http vs https – note HTTP/2 supports non-encrypted payloads for localhost testing only

HTTP Evolution – 2 to 3 – Still under development

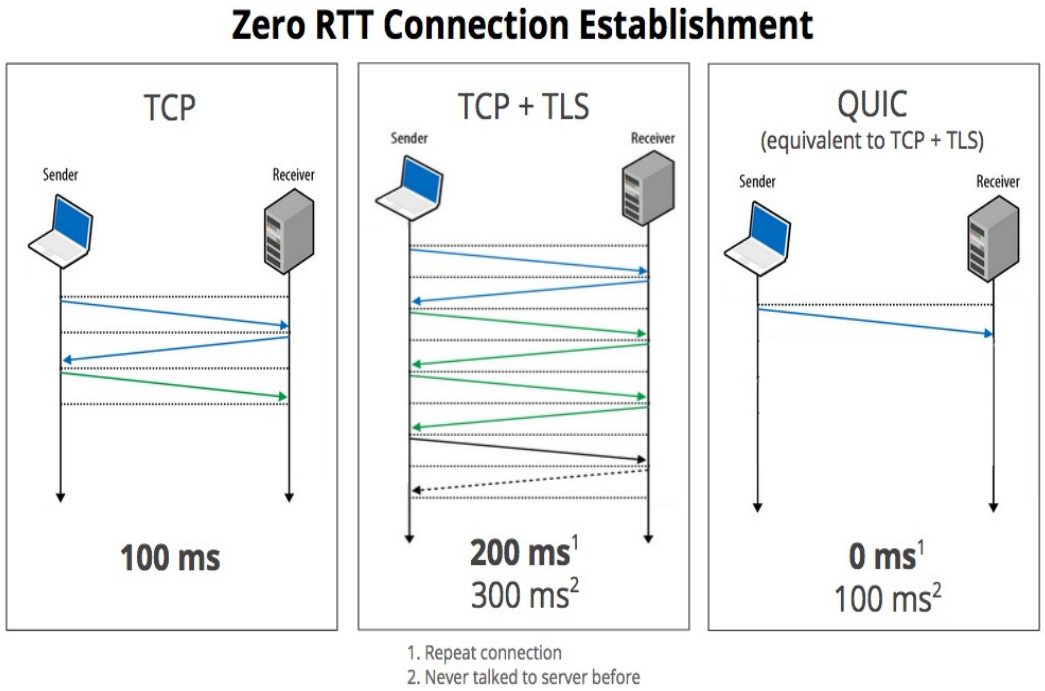


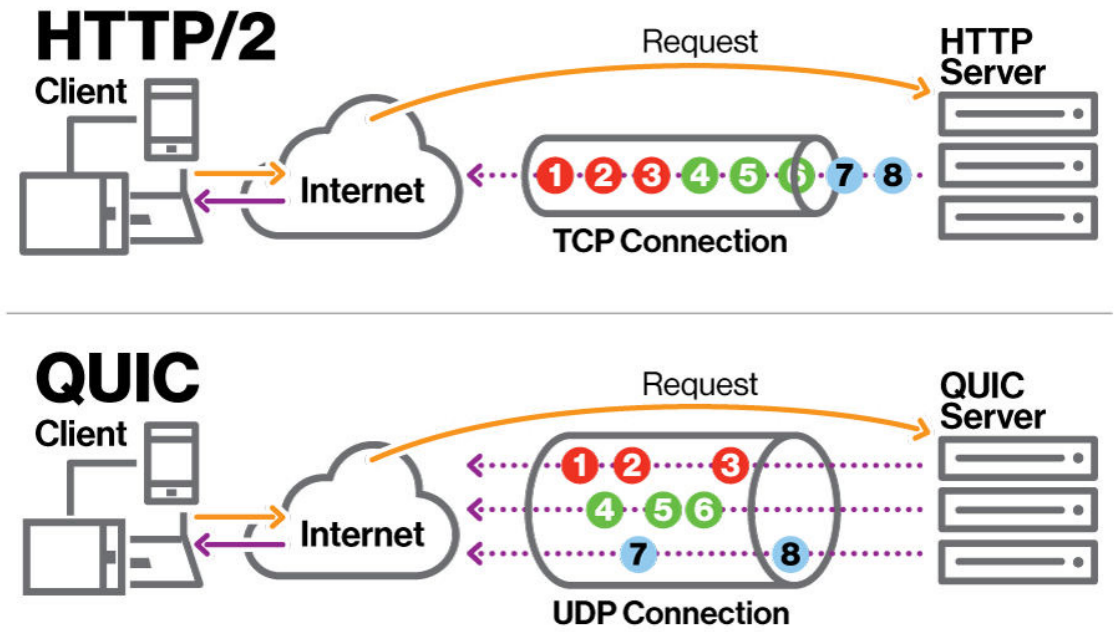
With HTTP/3, QUIC replaces TCP for establishing reliable connections Between clients and servers

Super novel ideal, proposed by Google and working its way through the standards bodies:

- Architecturally, TCP really cant be replaced given its baked into intermediary router hardware, it can only be bypassed by using UDP
- QUIC provides connection reliability just like TCP
- TCP is 40 years old, and has to be general enough to handle any network traffic
- We also have 20 years of experience with HTTP and can benefit from a modern connection-oriented protocol that is purposed built for Web and API traffic vs general traffic
- Like HTTP/2 encryption is not optional
- QUIC can adjust behavior based on connection quality – think wired, vs wifi, vs mobile

Specific Benefits of QUIC vs TCP for HTTP

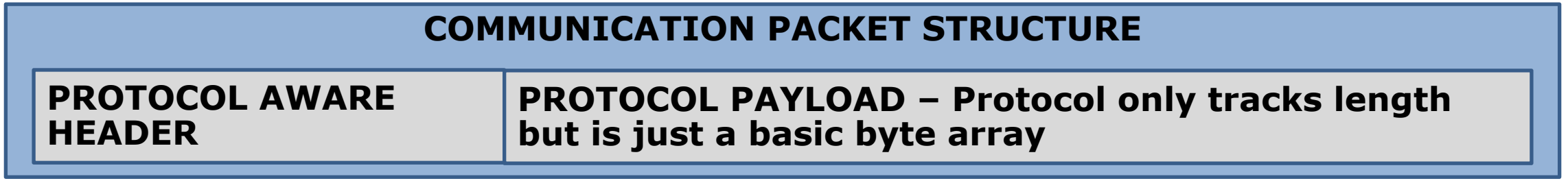




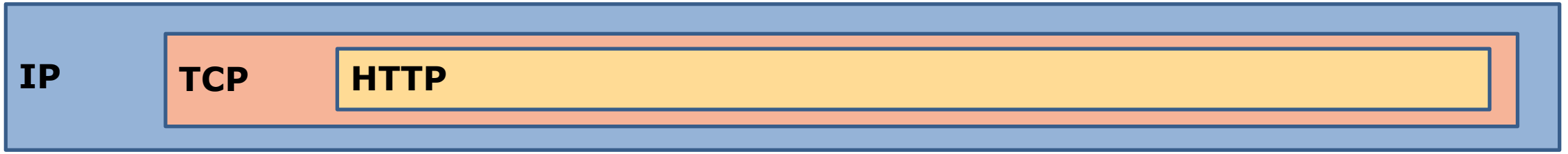
BIG BENEFIT #1
Since QUIC has knows both sides of the connection will be encrypted it does not have to negotiate it, improving connection setup time dramatically

BIG BENEFIT #2
Since QUIC uses fast and cheap UDP connections it can use individual UDP sockets to multiplex client calls, since TCP is expensive to setup, multiplexing gets serialized on the TCP connection, leading to buffering inefficiencies such as "head of line" blocking

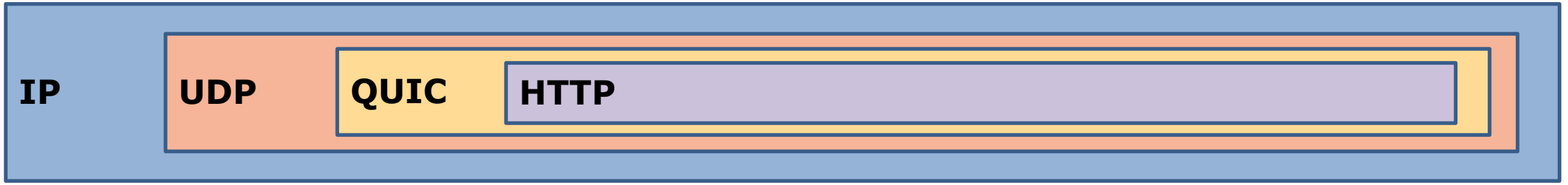
Wrapping up HTTP



THUS protocols can be "stacked" where a nested protocol can be put in the payload of a parent protocol

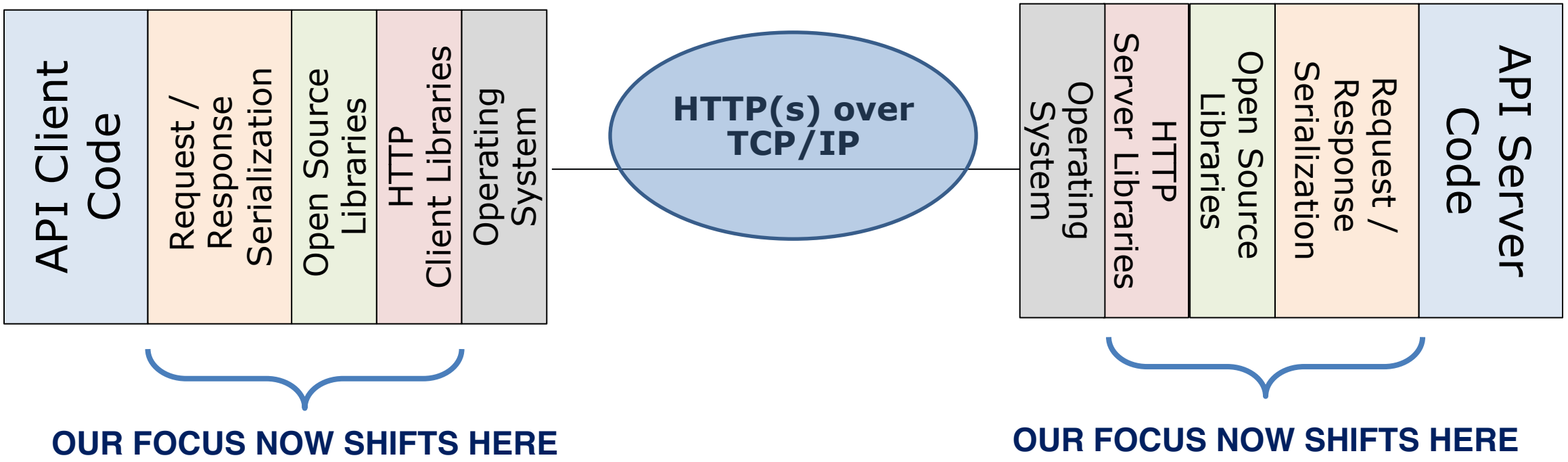


HTTP/1.1 and HTTP/2



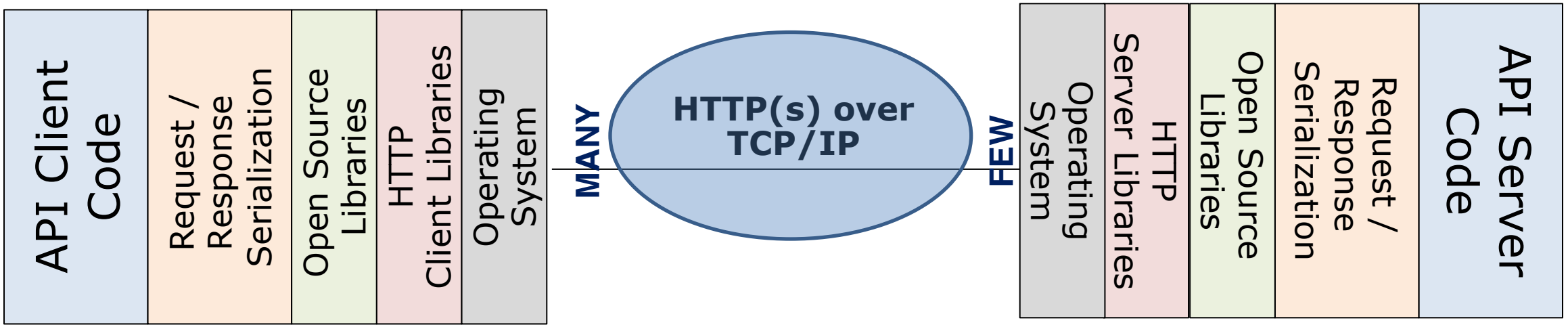
HTTP/3

BACK to APIs now that we covered HTTP



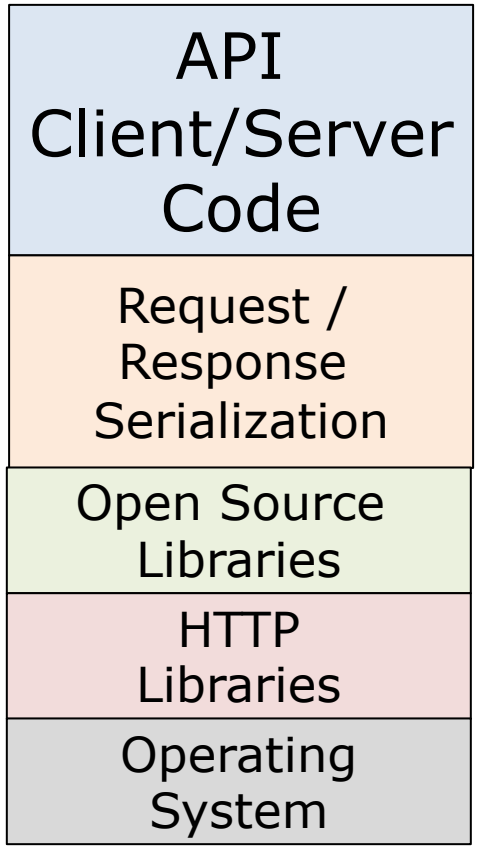
Certain aspects of API architectures have evolved over the past 20 years, others have not changed…

WHAT HAS NOT CHANGED: The basic architecture shown BELOW has not changed in the past 20 years. One or more API Clients interact with One or more API servers. In general, the number of API clients exceeds the number of API servers dramatically



WHAT HAS CHANGED: (1) VOLUME – 20 years ago we measured volume in dozens of concurrent connections and hundreds-of-thousands to low millions of calls per year… now we measure volume in thousands of concurrent connections and billions of calls per year. (2) PERFORMANCE – 20 years ago 3-5 seconds was acceptable, now anything over 500 ms is considered slow, we target under 100ms in general. (3) RESILIENCY – 20 years ago 95+% availability was OK; now anything less than 99.9% is considered bad. (4) SECURITY – 20 years ago APIs were internal only, now they are a major attack vector so security is essential.

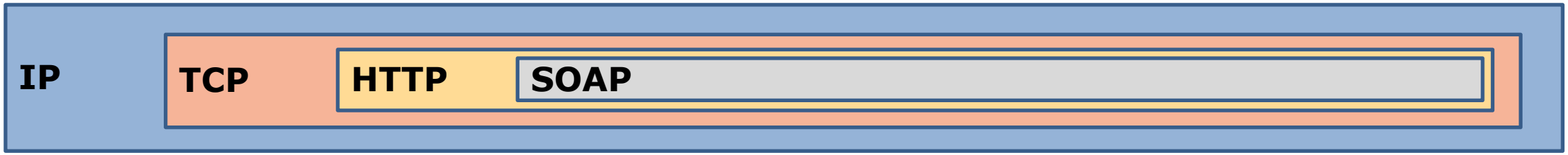
API Components



For API-based architectures we need:

- A way to serialize and deserialize data structures that will be exchanged between the client and server that is language agnostic
- Libraries that provide simple abstractions to send and receive data without knowledge of the underlying network protocol requirements
- The ability to support important resiliency patterns such as health checks, retries, timeouts, circuit-breakers, etc given this is a distributed system
- The ability to observe and trace distributed calls end-to-end for performance optimization and debugging
- The ability to supervise servers to ensure that they are running and are healthy, and to automatically restart and scale them out as needed
- The ability to ensure that the clients and servers trust each other from a security perspective, and are hardened to common attacks

XML-RPC and SOAP (late '90s – early '00s)

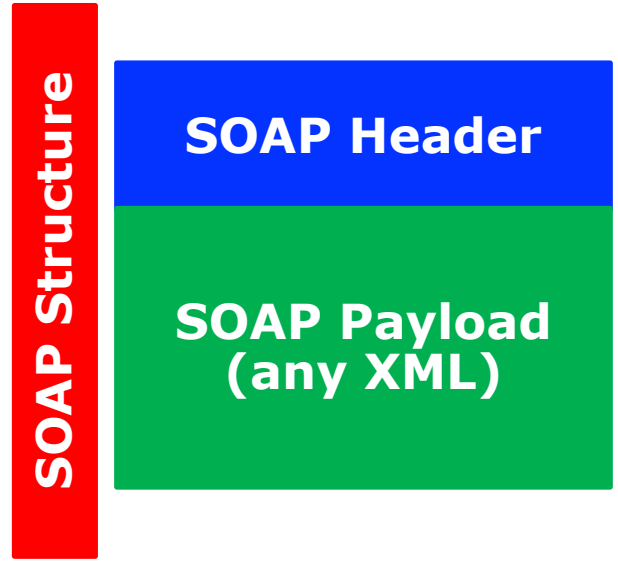


For the most part, SOAP was a protocol encapsulated in HTTP and was sent from clients to servers in an HTTP POST

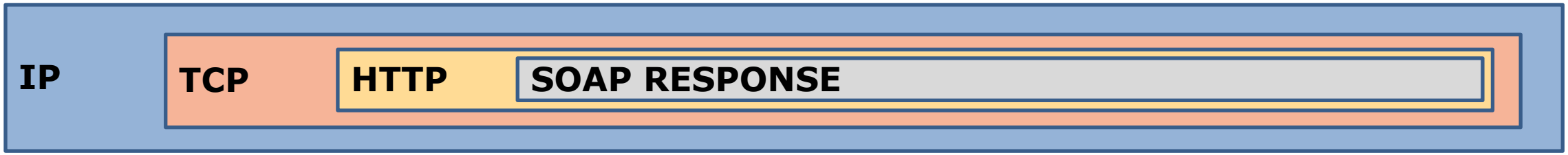
Like all other protocols, SOAP separated Headers that were used as Metdata, from the payload which was used to communicate request calls, and responses in XML

EXAMPLE: A SOAP REQUEST

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header>
      <h:LevelOfDetail> Summary </h:LevelOfDetail>
   </soap:Header>
   <soap:Body>
      <m:GetUser>
        <m:UserId>123</m:UserId>
      </m:GetUser>
   </soap:Body>
</soap:Envelope>
```



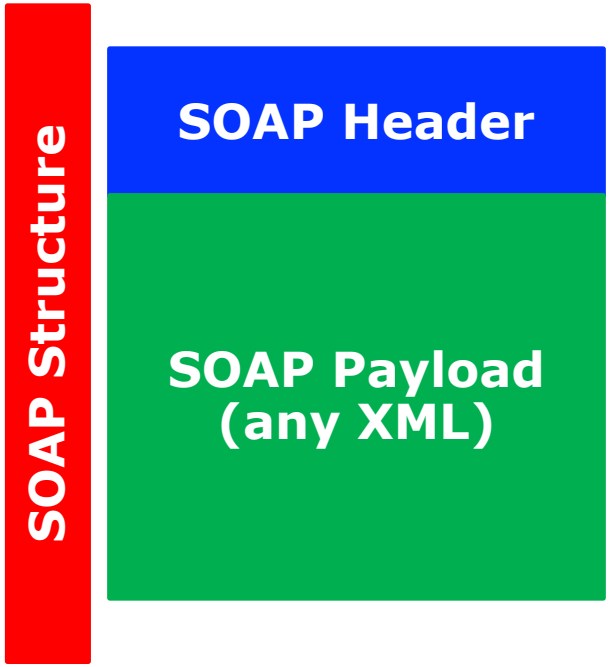
XML-RPC and SOAP (late '90s – early '00s)



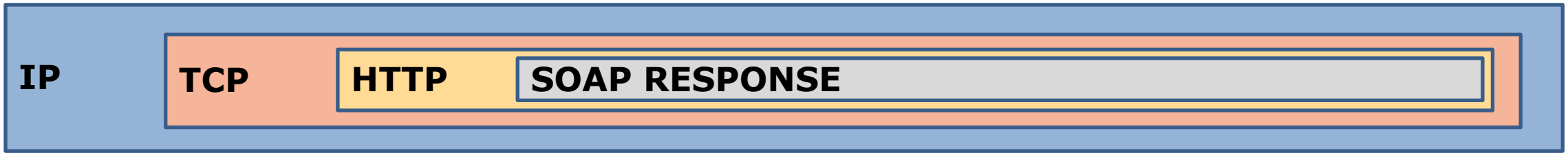
SOAP RESPONSES TOOK THE SAME STRUCTURE AS A REQUEST BUT HAD AN OPTIONAL OTHER VERB CALLED A "FAULT" THAT COULD BE RETURNED IF AN ERROR HAPPENED

EXAMPLE: A SOAP RESPONSE

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header>
      <h:LevelOfDetail> Summary </h:LevelOfDetail>
   </soap:Header>
   <soap:Body>
      <m:GetUserResponse>
         <m:First-Name>B rian</m:First-Name>
         <m:Last-Name>Mitchell</m:Last-Name>
         <m:Gender>Male</m:Gender>
         <m:OtherStuff>Foo-Bar</m:OtherStuff>
      </m:GetUserResponse>
   </soap:Body>
</soap:Envelope>
```



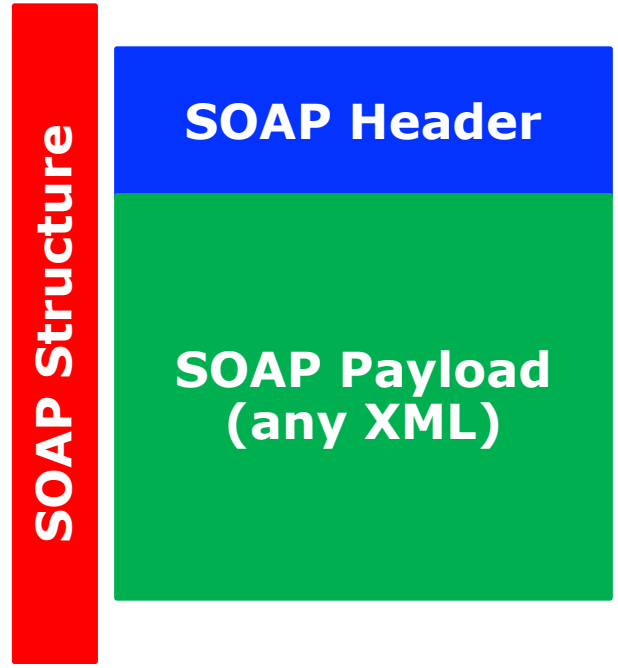
XML-RPC and SOAP (late '90s – early '00s)



SOAP RESPONSES TOOK THE SAME STRUCTURE AS A REQUEST BUT HAD AN OPTIONAL OTHER VERB CALLED A "FAULT" THAT COULD BE RETURNED IF AN ERROR HAPPENED

EXAMPLE: A SOAP RESPONSE WITH A FAULT

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header>
      <h:LevelOfDetail> Summary </h:LevelOfDetail>
   </soap:Header>
   <soap:Body>
   </soap:Body>
   </soap:Fault>
      <Code>1234</Code>
      <Message>User Not Found</Message>
   </soap:Fault>
</soap:Envelope>
```



SOAP Protocol

- The Soap protocol used XML and wrapped SOAP messages in the <Envelope> tag
- The <Envelope> tag was allowed to encapsulate the <Header>, <Body>, and <Fault> tags
- ISSUES
  - The data contained under the standard tags was not standardized beyond having to be valid XML, thus techniques had to be used to communicate data schema expectations between the client and the server
  - Every request customized what was expected of the server, and every response had to react to the request
  - XML processing is verbose and tends not to be super performant for large payloads
  - SOAP was encapsulated inside of the HTTP packet, yet it took zero advantage of the HTTP protocol itself

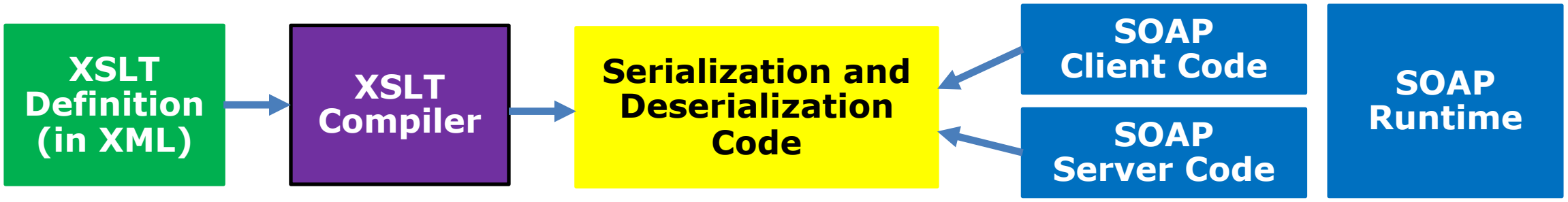
What about Security with SOAP?

```
<soapenv:Header>
  <wsse:Security xmlns:wsse="http://...oasis-200401-wss-wssecurity-utility-1.0.xsd">
     <wsse:UsernameToken wsu:Id="UsernameToken-1">
       <wsse:Username>login</wsse:Username>
       <wsse:Password Type="http://...#PasswordText">XXXX</wsse:Password>
     </wsse:UsernameToken>
  </wsse:Security>
<soapenv:Header>
```

- Since SOAP is a protocol, security assertions are defined in a standard way nested under the SOAP header
- A variety of different security schemes have been standardized including basic authorization (shown above),and SAML
- Because SOAP is embedded in HTTP, any SOAP messages with security should be transmitted encrypted – SOAP doesn't do encryption, HTTPS is utilized

SOAP – Design by Contract

- To address the fact that every SOAP request and response could be structured in any way, best practice evolved to "design by contract"
- The definition of the structure and semantics of requests and replies was defined in an XML quasi-schema language called XSLT
- Tools parsed XSLT and generated custom proxies and stubs to serialize and deserialize to SOAP request and response structures



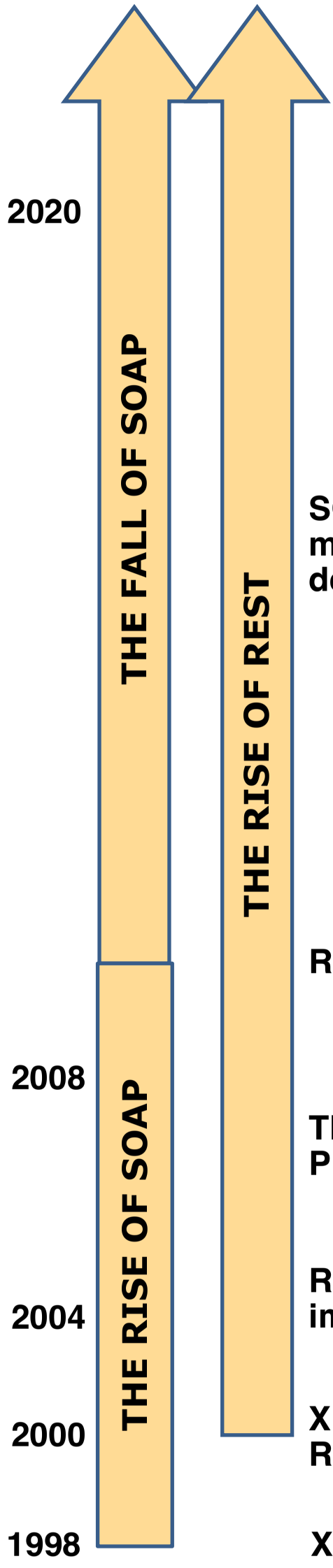
SOAP – Challenges with SOAP Design

- SOAP clients made request to the SOAP server using “verbs”, AKA – go do this, go do that, which lead to massive sprawl and no standards for how to interoperate
  - GetCustomerByFirstName, GetCustomerByID, GetCustomerByAddress, …
- SOAP did not use any of HTTP to manage traffic, scale, routing, etc so custom SOAP servers were required – e.g., IBM WebSphere, etc. Each one of these customer vendor offerings had their own story around how/why they scaled and ran better than other vendor offerings
- Because the serializers, and deserializers were provided by the vendor there were interoperability challenges if you tried to use code compiled against IBM Websphere with BEA Weblogic – this was by vendor design
- XML lead to big payloads, and in the early 2000’s networks were not what they are today

SOAP – Architecture Summary

Feature	Description	Architecture Implications
Protocol	SOAP is a well defined protocol, consisting of a header and a body/payload, most of the time encapsulated in HTTP	• Given SOAP rides on HTTP, and we know how to scale HTTP, SOAP scales • Since its a "wrapped" protocol, we need libraries to encode/decode SOAP messages, these libraries can be of varying quality and performance
Payload / Encoding	The SOAP standard specifies XML as the standard encoding	• Unlike other protocols we will look at that offer multiple encoding types, SOAP is XML-Only • XML includes many nice features such as schema definition and enforcement • XML is bulky, can lead to performance issues to parse and transmit, especially over mobile networks
Client/Server Semantics	SOAP requests tend to be biased towards "do this", or "do that". In other words they are verbs	• Given the types of operations that generally come up in real applications, the number of actions we want to expose tend to become large, having a SOAP API for each one of them leads to MANY APIs • APIs could be reduced by developing complex request schemes that convey semantic intent
Current Status of SOAP	SOAP has been replaced by REST (discussed next) as a best practice, but well performing SOAP APIs don't require replacement	• SOAP has proven to be able to support modern workloads – many organizations still maintain and extend SOAP services, and they work fine • Newer techniques have proven to be simplier, so they are common practice these days

Then came REST



Other alternatives to REST introduced – GRPC, GraphQL but REST still the most common API technique

SOAP Services go into “maintain” mode, just about all new APIs developed in REST

REST WINS

The REST vs SOAP DEBATE Plays out

Roy Fieldings PhD work implemented – REST born

XML RPC Standardized as SOAP
Roy Fielding PhD Published

XML RPC Introduced

REST 101 – What is a Resource? It's the most important concept that you need to know

- A resource is an abstract concept that can be thought of as anything important enough to be referenced as a thing in itself
- They are generally identified by mining for nouns in your design space. E.g, persons, users, locations, items, orders, etc
- Resources can also be thought of as entities in a database, thus they come in collections – notice above I refer to them in Plural form, not singular form
- Because they come in collections, resources should be uniquely identifiable – think of an ID or a Key
- Resources are naturally hierarchical, a resource can be linked to one or more child resources, which in turn could have children (a tree)

REST 101 – Representing a resource in REST

A resource in REST is directly mapped to the HTTP protocol

```
HTTP_VERB LOCATION/RESOURCE-COLLECTION/ID?PARAM-KEY=PARAM-VALUE&... HTTP/VERSION
HEADERS ...
PAYLOAD ..l
```

```
GET api.github.com/users/ArchitectingSoftware HTTP/1.1
```

```
GET api.github.com/users/ArchitectingSoftware/Repos HTTP/1.1
```

```
PATCH api.github.com/users/ArchitectingSoftware HTTP/1.1
HEADERS ...
PAYLOAD { updated_user_json}
```

```
PUT api.github.com/users/ArchitectingSoftware2 HTTP/1.1
HEADERS ...
PAYLOAD { new_user_json}
```

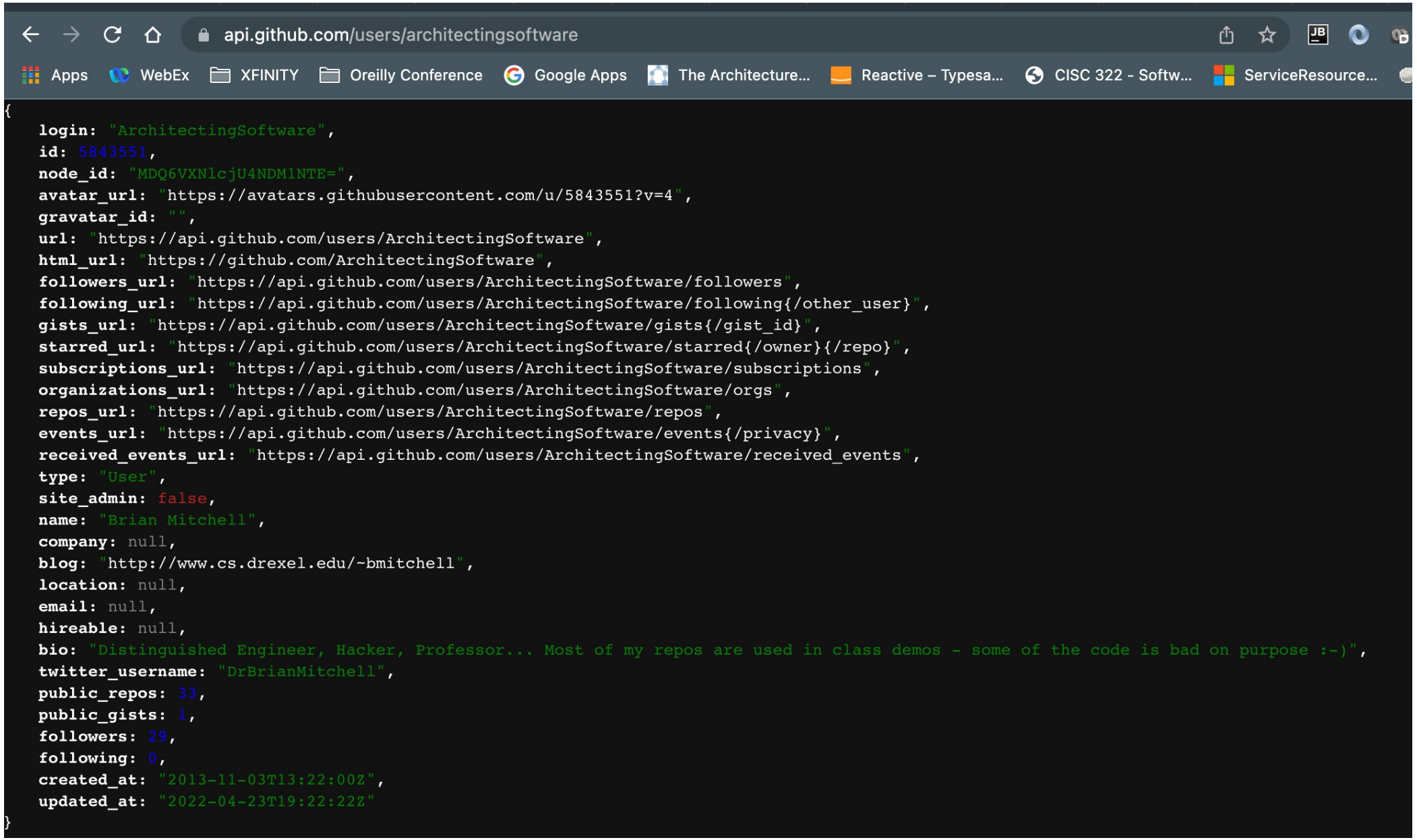
HTTP VERB MEANING

GET: Query
PUT: Create
PATCH: Update
DELETE: Deleate

HTTP RESOURCES

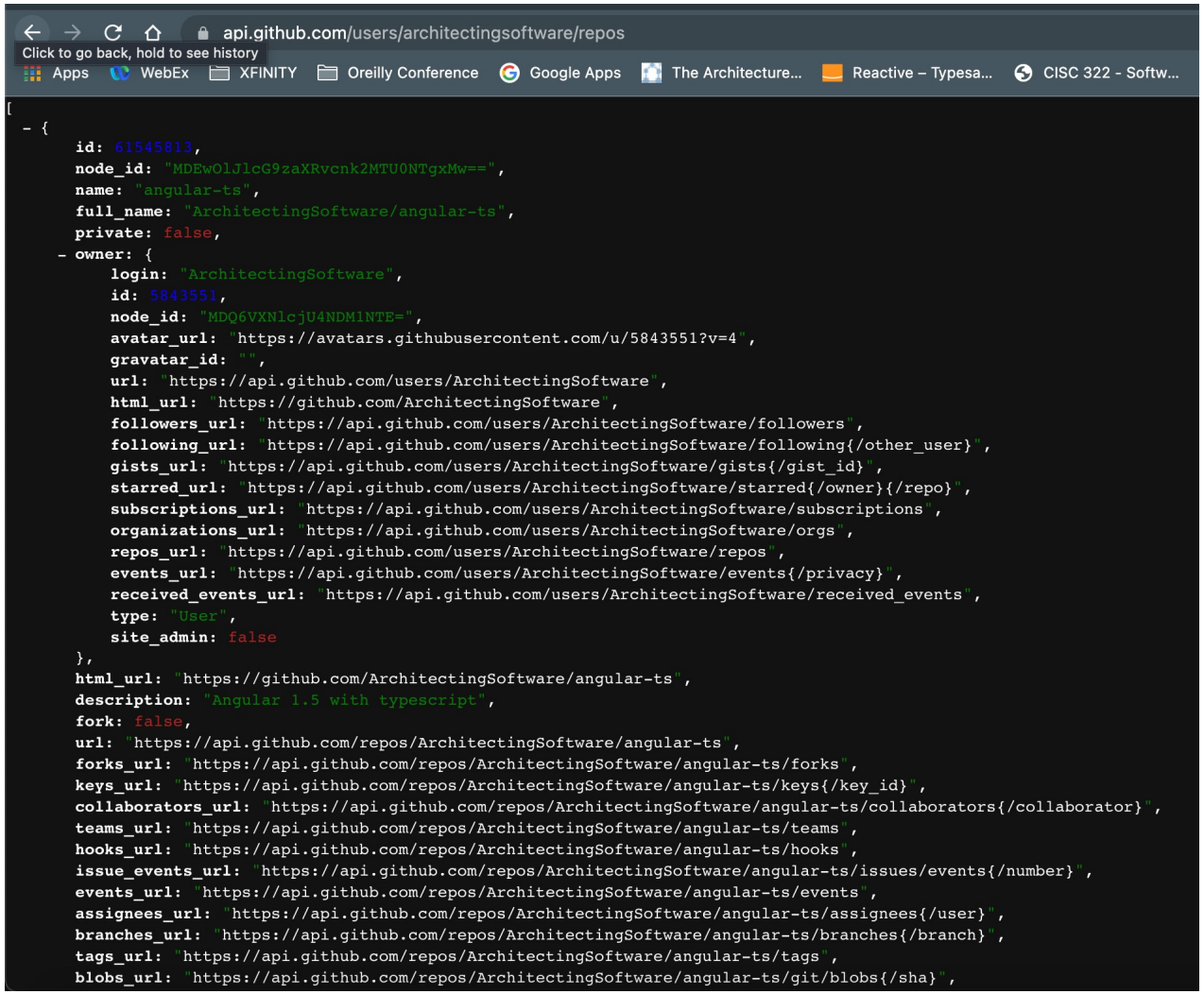
GET: Query for 1…n
PUT: Create 1
PATCH: Update 1
DELETE: Delete 1

REST 101 – Example



REST 101 – Example

Nested Resource Example
GET api.github.com/users/ArchitectingSoftware/repos



Notice the LINK header that GitHub returned – because it didn't return the entire collection, it gives you links to the next and last page to support navigation/pagnation

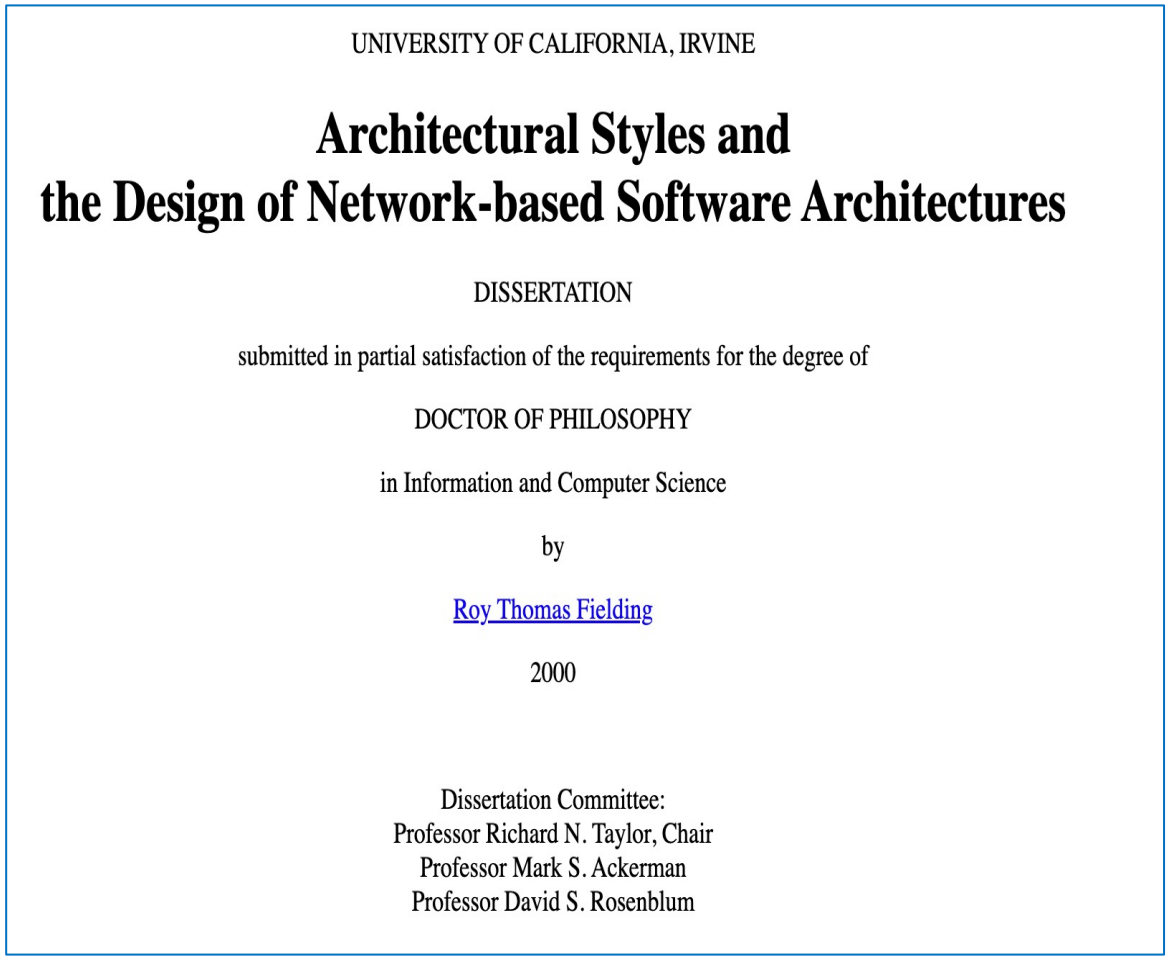
THIS IS CALLED HYPERMEDIA

```
8   X-GitHub-Media-Type: github.v3; format=json
9   Link: <https://api.github.com/user/5843551/repos?page=2>; rel="next", <https://a
    pi.github.com/user/5843551/repos?page=2>; rel="last"
```

REST is like SOAP, but simplifies things, A LOT

Feature	Description	Architecture Simplification Implications
Protocol	SOAP was a custom protocol encapsulated in HTTP, REST uses HTTP directly	• Eliminates the need for software that can build and understand SOAP protocols • Eliminates a layer, leads to better performance
Payload / Encoding	The SOAP standard specifies XML as the standard encoding, REST makes no claim on payload encoding, just uses HTML content-type header to describe	• SOAP is XML, Early REST was also XML, but could be easily migrated to anything else, these days JSON, which is more compact is the standard • JSON is more compact, easier to parse, and is naturally aligned to the web architecture – javascript objects are JSON
Client/Server Semantics	SOAP semantics are "verb" based – do this, do that. REST semantics are Noun based, here is a structure, and here is how I want to operate on it.	• Verb based semantics based lead to a significant API sprawl, where each operation lead to its own service, or complex request semantics to convey intent • Noun semantics simplified things a lot, REST operates on resources/entities, and uses standard HTTP Verbs– GET, POST, PATCH, DELETE to query and manipulate them (think CRUD)
Evolvability	SOAP being a custom protocol is supported by libraries "compiled into" the services themselves. Rest requires no libraries, it conforms to HTTP standards	• SOAP can be difficult to version and evolve, quality depends on library implementations and their interoperability • SOAP says nothing about how to negotiate protocol versions, this is built into the HTTP standard that REST uses

So where did the principals of rest come from?



UNIVERSITY OF CALIFORNIA, IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

2000

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

- At the same time engineers started to use SOAP, Roy Fielding wrote his PhD thesis (2000) that suggested an alternative architecture
- By 2004 people were starting to play with this model, and it became the defacto-standard a few years later
- It was called REST, **RE**presentional **S**tate **T**ransfer
- REST is an architecture for distributed hypermedia systems

Guiding Architecture Principals for REST

- Uniform Interfaces in REST – REST manipulates RESOURCES
- Resources are like objects in OO, or an entity in a DB
  - Identification: The interface must uniquely identify each resource
  - Manipulation: Servers should return resources to clients in a uniform way, and clients should use these representations in a standard way to manipulate the state of the resource
  - Self-Descriptive Messages: Resource messages should be self descriptive, they should not only contain information on the resource itself, but also include additional actions that a client can perform
  - Hypermedia as the engine of Application State (HATEOAS): The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks

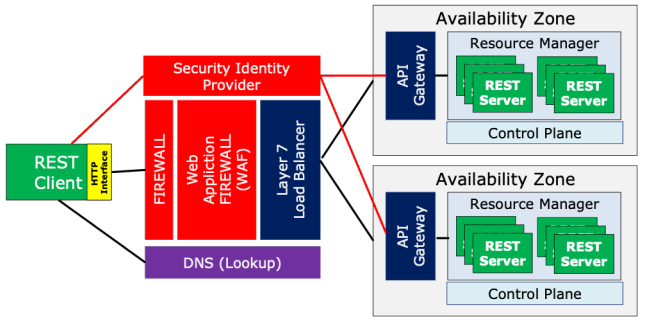
Guiding Architecture Principals for REST

- Embrace the client/server architecture pattern
  - Put smarts in the endpoints – the client and server, and allow them to evolve independently
  - Separate the interface from the storage and resource lifecyle management concerns. Are resources stored in a database? A cache? Nowhere, just computational (e.g., my blockchain demo)?
  - Evolvability, discovery and versioning of interfaces. Keep interfaces compatible where it makes sense, make breaking changes on purpose
- Statelessness
  - Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.
  - This is a key attribute to scaling REST-based services

Guiding Architecture Principals for REST

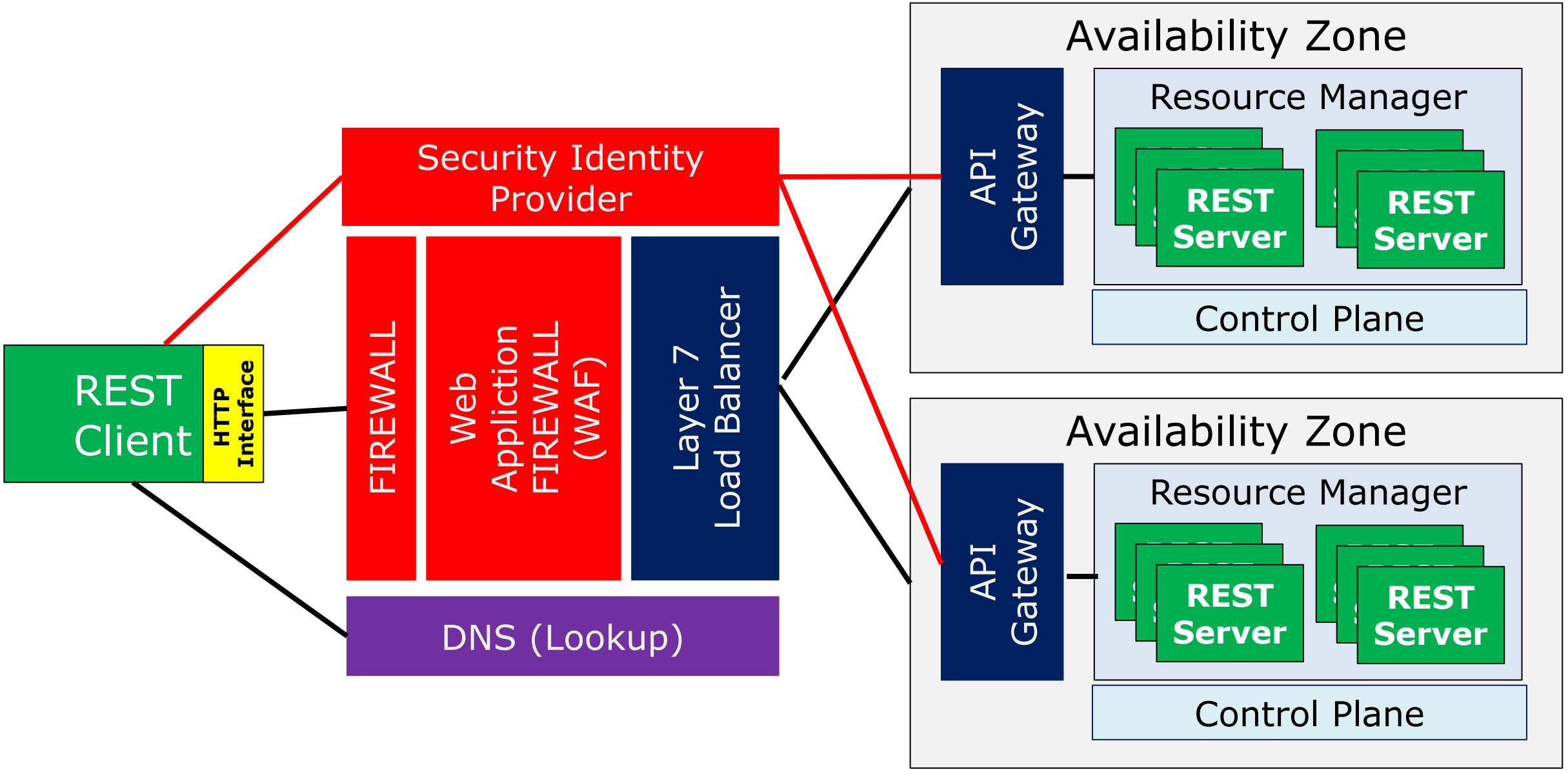
- Cacheable
  - The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable
  - Done via the cache-control header. This header indicates if the resource is cacheable, and if so for how long, and if the caching can only happen at the client and server itself vs also at intermediaries.
- Embrace, and don't work around HTTP
  - Use HTTP to the fullest extent possible – headers, response codes, etc. HTTP works and scales so use it
  - API specific control data and detailed error messages can be exchanged as metadata on the API call. Generally this is done with X- headers or a custom "metadata{}" object in the response

REST Architecture Components



Component	Description
REST Client	Component that makes API calls and receives responses from the REST Server
Firewall	Layer 2/3 firewall that deals with IP addressing, IP Routing and Port Filtering
WAF	Layer 7 (e.g, HTTP aware) firewall that supports protecting against HTTP attacks – SQL Injection, Bots, etc
L7 Load Balancer	Load balancer that distributes load and ensures health of downstream components – eg., the API Gateway. It works at Layer 7 so its optimized for HTTP protocol
Identity Provider (IdP)	A RESTFul security endpoint that handles authentication and authorization requests. Typically they issue tokens that can be used to assert trust
DNS	Location services for REST
API Gateway	An intelligent proxy that can load balance REST services, act as a security enforcement point, apply policies around traffic management and shaping, etc
Resource Manager/ Control Plane	An intelligent runtime that supervises running services. It can react to traffic congestion, errors, etc and can scale up/down running instances as needed
REST Server	Component that receives and processes REST Client calls

Reference Architecture for REST



Current State of REST

- The RESTful architecture for APIs is considered best practice currently
- However, there are some things that REST does not do well, so some options exist
  - Type safety for data structures
  - Data structure discovery
  - Sometimes request/reply semantics do not fit our needs
  - Support for streaming
  - Conventions for handling things like paging, filtering, etc

Current State of REST – Some challenges

Example REST Response – from one of the demos I have been using
Challenge is around data structures and data types


```
{
 "blockHash": "000bac8172d58c3c0c2a25e1deafadd6a225e2ebdd75b412ae3e8de39b3234a6",
 "blockId": "18f3a534-3f69-469d-bc22-981a01d12ac2",
 "executionTimeMs": 6,
 "found": true,
 "nonce": 6922,
 "parentHash": "0000000000000000000000000000000000",
 "query": "hello-there”
}
```


- How do I know that these attributes are returned from the API?
  For the most part, this is handled via documentation in REST
- How do I know which fields are strings, booleans, numbers, etc?
  For the most part this is handled via documentation in REST

RESTFul models generally rely on documentation to define resource structures, header standards, Data types, etc. This is just the way it is, but there is some good tooling out there to generate documentation from code like swagger. There are also other solutions, although not widely used to help with this – google “JSON Schema”

Current State of REST – Some challenges

Example REST Request – I used this earlier to demonstrate
Challenge is large collections

```
GET api.github.com/users/ArchitectingSoftware/repos
```

- Sometimes REST responses deal with, and need to return, large collections. Best practice is to use custom paging technique. Some common practices
  - Use the standard Link header to specify links to get the first, previous, next and last pages – Github uses this technique
  - Use GET parameters like limit and offset to self navigate. For example: …/users/ArchitectingSoftware/repos?limit=5&offset=10 to return items 10-15
- How do I know the technique used for managing large collections with paging? You got it, read the documentation?

Current State of REST – Some challenges

Example REST Request – I used this earlier to demonstrate
REST is Uni-Directional Request/Reply

```
-> GET api.github.com/users/ArchitectingSoftware/repos
...SOME TIME
<- { RESULTS }
```

- REST by its very nature is request/reply. Everything is requested from the client, and fulfilled by the server – sometimes this does not meet your needs
- For other situations, REST is not a good solution, but there are ways to support these things:
  - What if I want to be notified directly from the server when something happens? Solutions: HTTP server push, only on HTTP/2+, Webhooks
  - What if I want to keep a long connection open between the client and server for something like a chat application. Solution: Websockets

Current State of REST – Some challenges

Example REST Request – I used this earlier to demonstrate
Streaming Requests and Responses

- Sometimes a client may want to stream data to a server, or a server might want to stream data to a client.
- REST by its definition really does not support this concept given it expects a well defined request and a well defined response.
- REST responses can be large collections, rather than sending the entire result, it can be paged
- Its interesting how REST does not support streaming but HTTP does. In HTTP/1.1 there was a simple streaming standard that used a special header, Transfer-Encoding: Chunked. HTTP/2 supports streaming in its core protocol, but REST really doesn't take advantage of it

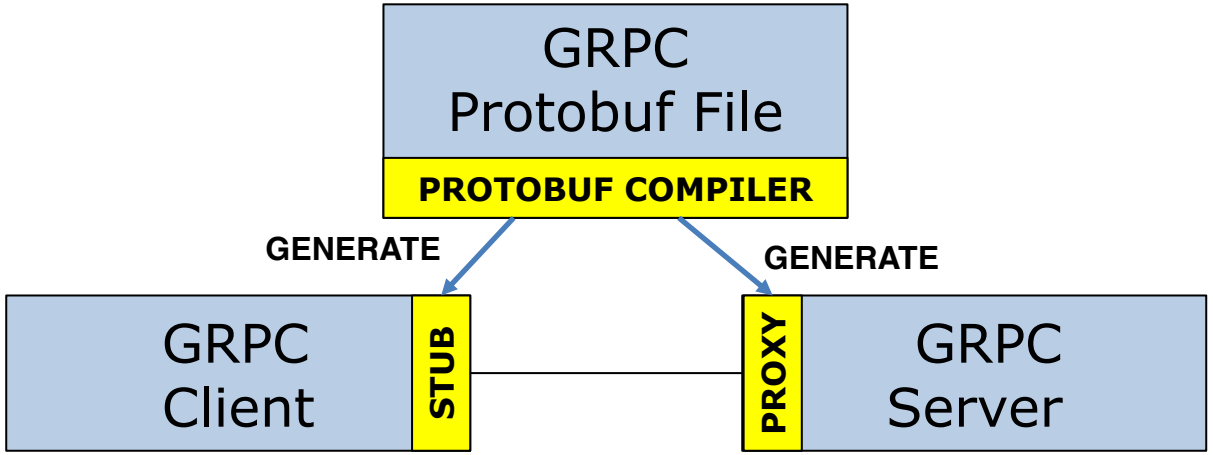
REST Summary

- Even with all of the shortcoming highlighted, REST is the defacto standard these days given its proven to run reliably, at scale, and has very good tooling to support creating APIs following the REST architecture principals.
- There are many good resources on the web to learn more about best practices for RESTful design and architecture. Here is just one: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

Modern REST Alternative 1 - GRPC

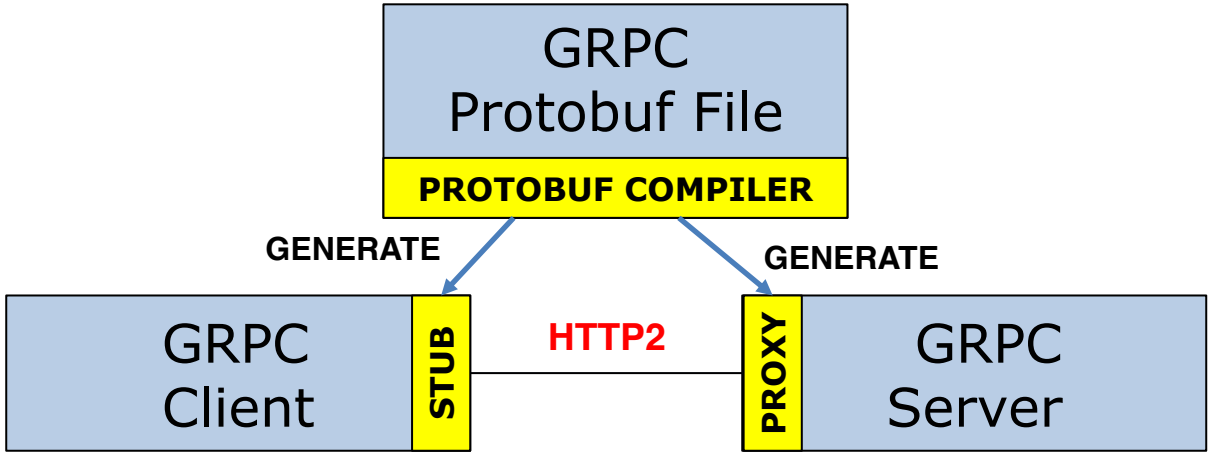
See: https://grpc.io/

- What if you could improve the performance of REST by more than 10x, and introduce type safety at the same time? Interested?
- GRPC was introduced by Google in 2016 and is an essential component in any modern web service architecture
- GRPC has an amazing feature set, but has one significant caveat that currently positions it as a complement to REST, and not a replacement for REST.



GRPC Architecture

- With GRPC you define your client/server interactions in a protobuf file
- You then compile that protobuf file into code that can be imported into your client and server. This code facilitates all the communication
- If you are using a strongly typed language, the generated code enforces this at compile time – a big benefit over REST
- This architecture has been used many times before and has not worked out – MSRPC (2006), CORBA (1990s) – Whats Different? **The communication between the client and server flows over HTTP/2 (vs TCP/IP sockets) – which works at amazing scale!**



What does a protobuf file look like?

```
syntax = "proto3";

package BCGrpc;


service BCSolver {
  rpc BlockSolver (BcRequest) returns (BcResponse) {}
}

message BcRequest {
  string query = 1;
  string parent_block_id = 2;
  string block_id = 3;
  uint64 max_tries = 4;
  string complexity = 5;
}

message BcResponse {
  string block_hash = 1;
  string block_id = 2;
  int64 exec_time_ms = 3;
  bool found = 4;
  uint64 nonce = 5;
  string parent_block_id = 6;
  string query = 7;
}
```

Defines the client/server interactions

Defines the message structures and types

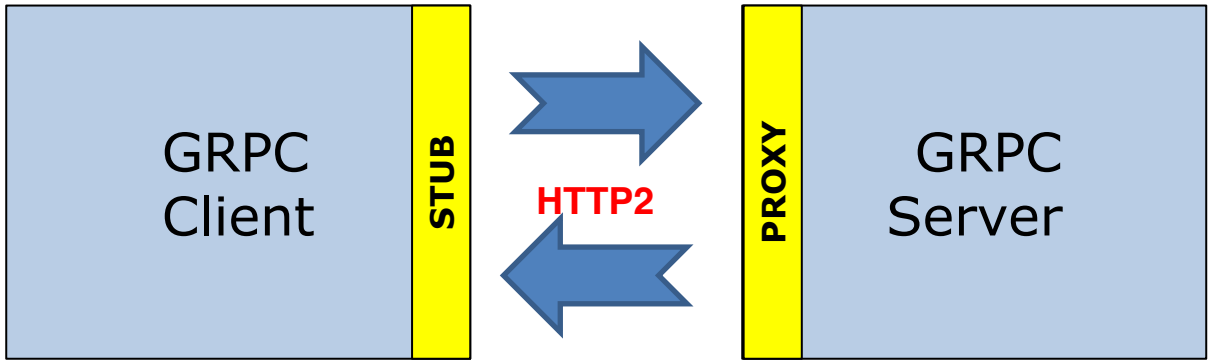
Note that each attribute has a unique number, that is used to optimize binary serialization and deserialization

What about Streaming?

```
service BCSolver {
  rpc NoStreamHelloWorld (HelloReq) returns (HelloRsp) {}
  rpc CliStreamHelloWorld (stream HelloReq) returns (HelloRsp) {}
  rpc SvrStreamHelloWorld (HelloReq) returns (stream HelloRsp) {}
  rpc BiDirectionStreamHelloWorld (stream HelloReq) returns (stream HelloRsp) {}
}

message HelloReq {
  ...
}

message HelloRsp {
  ...
}
```

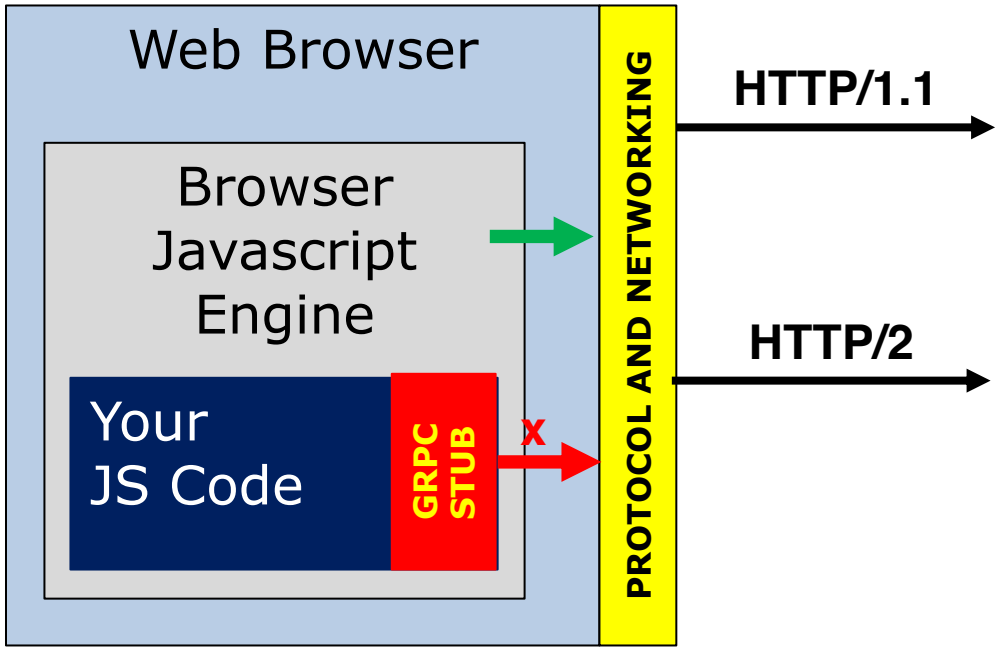
GRPC includes the stream keyword to enable streaming, it can be one way or bidirectional

Under the covers native HTTP/2 streaming is used



Why doesn't GRPC just outright replace REST?

GRPC Supports Dozens of Different Programming Languages, even Javascript and Typescript; But currently it CANNOT BE USED DIRECTLY FROM BROWSERS, which is a SIGNIFICANT BLOCKER for GRPC to directly replace REST

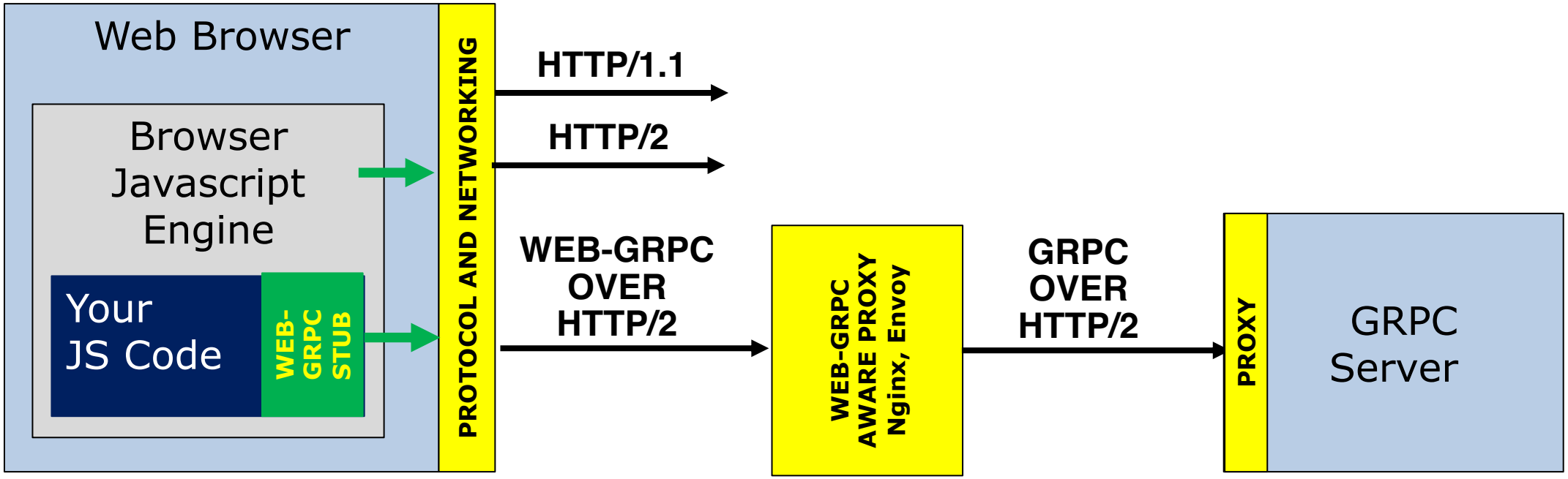


- GRPC requires HTTP/2, and all modern browsers support HTTP/2
- GRPC can generate Javascript, and all modern browsers support Javascript
- ISSUE: The javascript engines embedded in browsers blocks access to the APIs that GRPC requires to manage HTTP/2 connections

Since the browser is the most common client these days, does this make GRPC useless?

Workarounds for Browser Restrictions on GRPC

An Open Source Project Exists to Enable GRPC from the web, its called Web-GRPC
https://github.com/grpc/grpc-web
I personally consider this a quasi-architecture-hack and don't like it

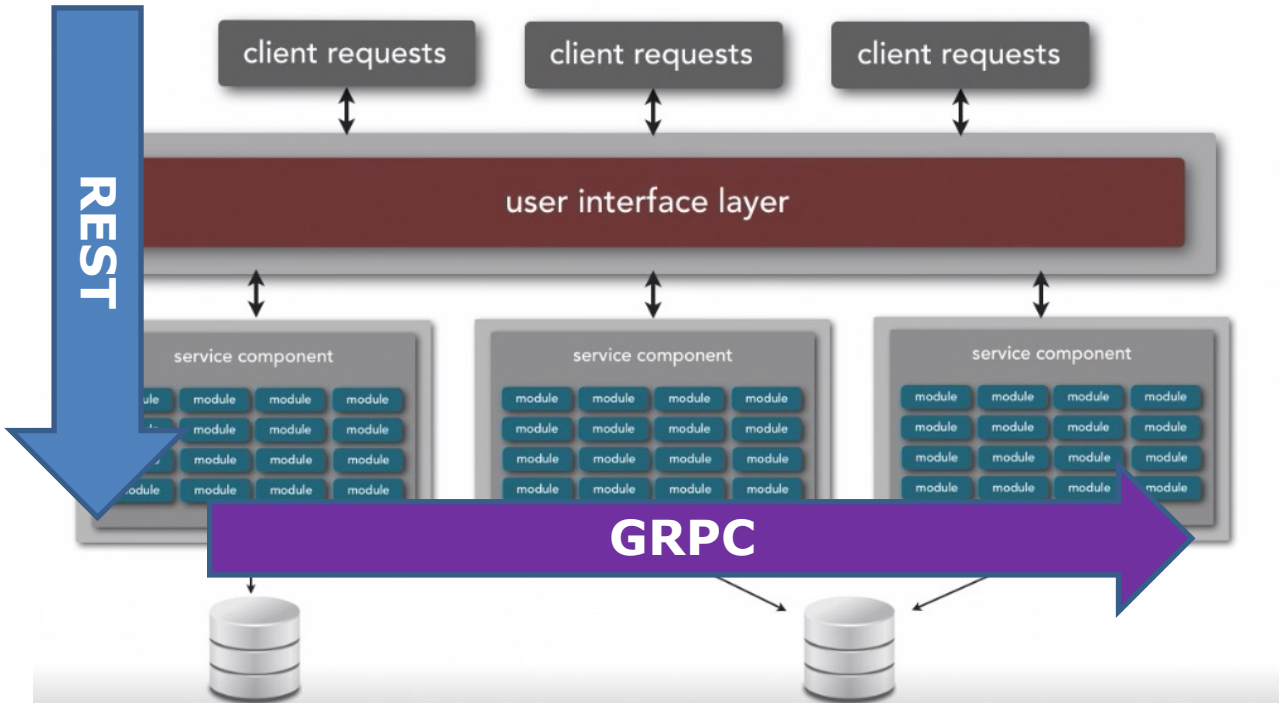


GRPC-Web works around the restrictions of the browser javascript-http/2 APIs and emulates the functionality not supported. HOWEVER, this is not standard GRPC and you need to introduce a proxy to translate between GRPC-WEB and proper GRPC.

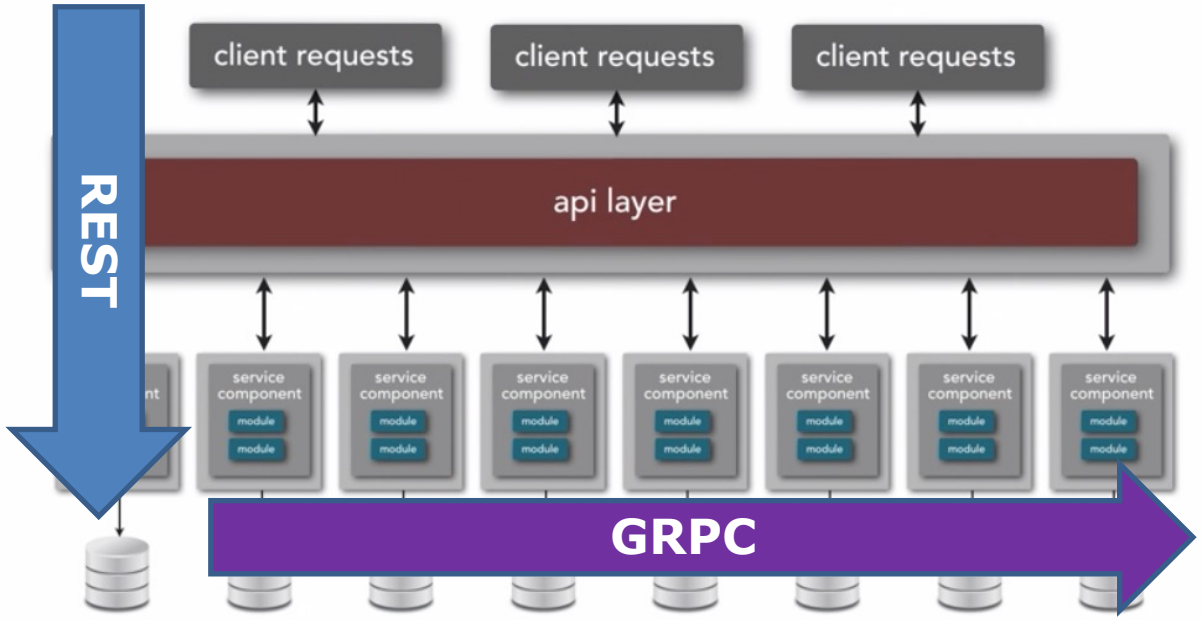
If you are programming in GO there is a wrapper for GRPC that also supports GRPC-WEB that you can play with, but its only available in GoLang
https://github.com/improbable-eng/grpc-web/tree/master/go/grpcweb

The GRPC Sweet Spot – GRPC Complements REST

Service Based Architecture Style



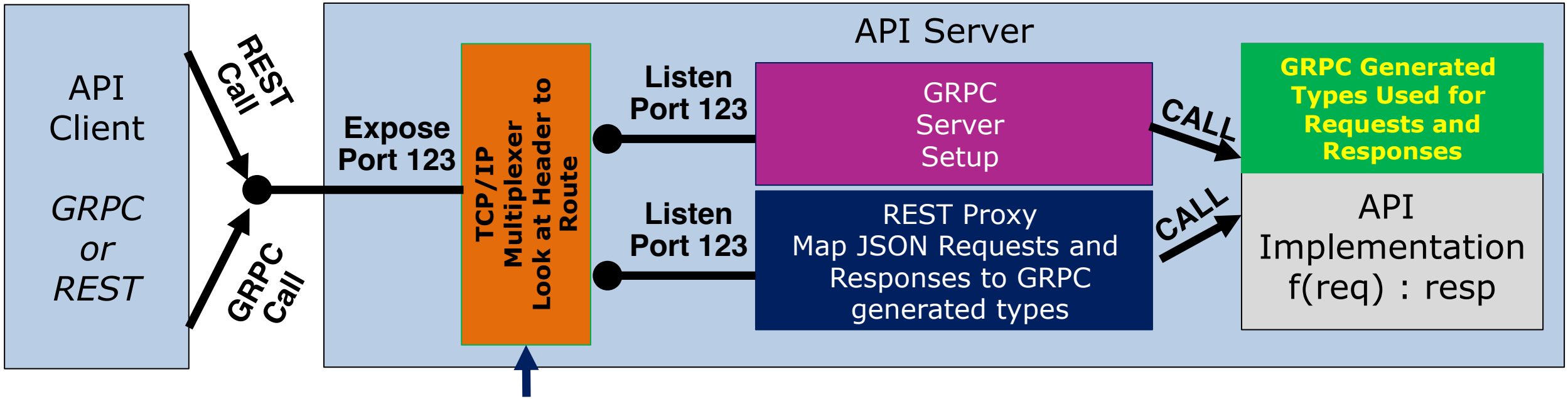
Microservice Architecture Style



This is a very common architectural approach –
Use REST from the client to the service tier, and use GRPC for service-to-service APIs

Do you have to choose – GRPC or REST?

Servers can be built to handle either GRPC or REST – even over the same port!
Given HTTP is used, we know its GRPC if its over HTTP/2 and `Content-Type: application/grpc`
REST would be over HTTP/1.1 or HTTP/2 with `Content-Type application/json`



Multiplexer probes HTTP protocol 1.1 or 2 and Content-Type Header to decide

To see a demo of this approach:
https://github.com/ArchitectingSoftware/se577-webservices-demo/tree/main/grpc/combined-server

Finally GraphQL

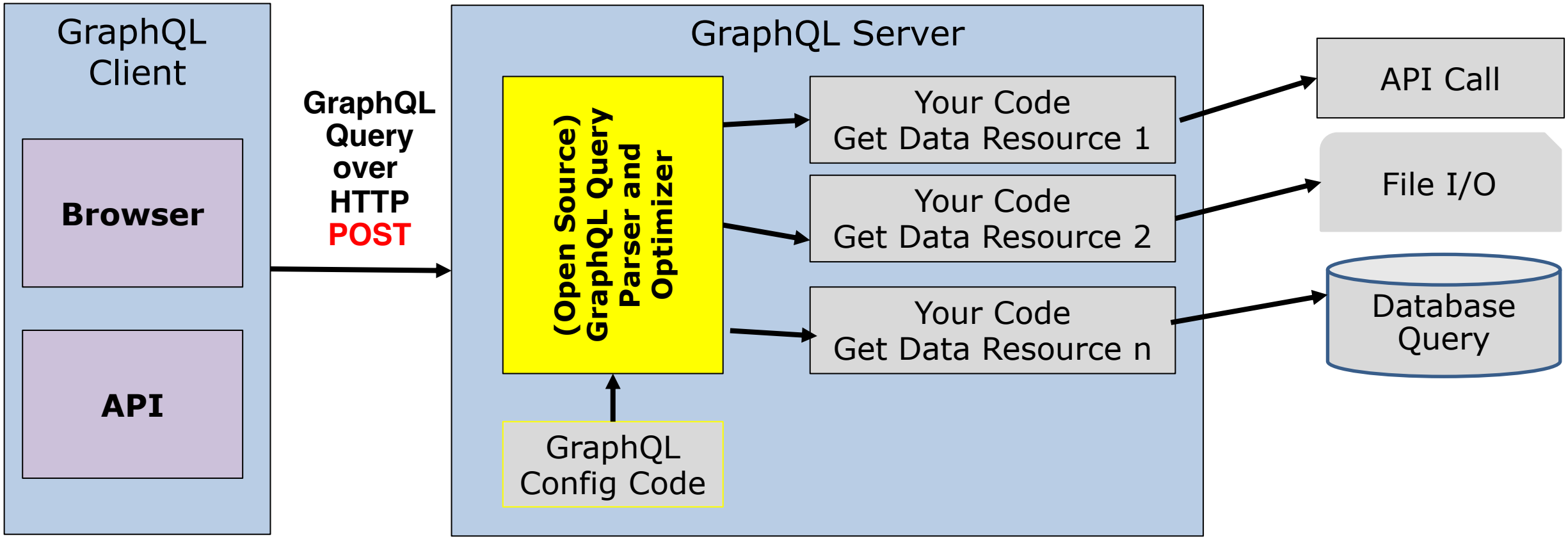
REST 101 – What is a Resource? It's the most important concept that you need to know

- A resource is an abstract concept that can be thought of as anything important enough to be referenced as a thing in itself
- They are generally identified by mining for nouns in your design space. E.g, persons, users, locations, items, orders, etc
- Resources can also be thought of as entities in a database, thus they come in collections – notice above I refer to them in Plural form, not singular form
- Because they come in collections, resources should be uniquely identifiable – think of an ID or a Key
- Resources are naturally hierarchical, a resource can be linked to one or more child resources, which in turn could have children (a tree)

Remember this slide – the definition of a resource, look at the 3rd bullet point
"Resources can be thought of as entities in a database, …"
Databases have standard Query langugages that can search and mutate data.
Why cant we have the same thing for Resources?
We can – its called GraphQL!

GrpahQL is a standard Query Syntax for Resources – Think SQL for Resources

Im not going to get into the details of the query language, google "GraphQL" and you will get a lot of content



GraphQL Architecture

GrpahQL Query Examples

```
{
  viewer {
    repositories(first: 100) {
      totalCount
      nodes {
        nameWithOwner
      }
      pageInfo {
        hasNextPage
      }
    }
  }
}
```

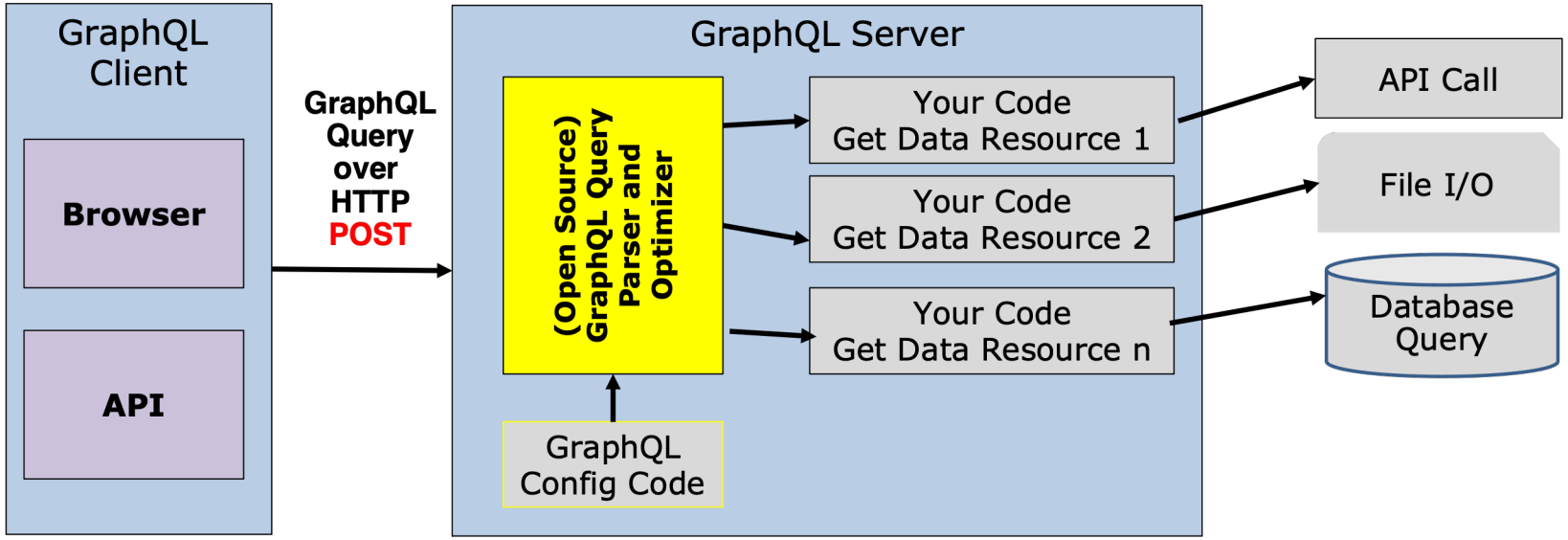
GitHub Supports GraphQL Queries


```
query {
  viewer {
    login
  }
}
```

Try the GitHub API Playground at:
https://docs.github.com/en/graphql/overview/explorer

GraphQL Architecture Analysis

Is GraphQL good or bad from an architecture perspective



From a client usability perspective, its pretty awesome, GraphQL has an amazing query syntax – its extremely powerful – see: https://graphql.org/. There is not much NOT TO LIKE

However, from a server perspective, you introduce a pretty complicated piece of software – shown in yellow that parses and optimizes queries – the quality of libraries that supports this varies dramatically

You also need to write all of the code in gray that gets all of the data to execute the query. You have little control over the efficiency of the query request to return the desired response

Overall opinion – GraphQL is hard to get right as its hard to optimize, REST is much simpler, so you decide I am a fan in very specific situations only!