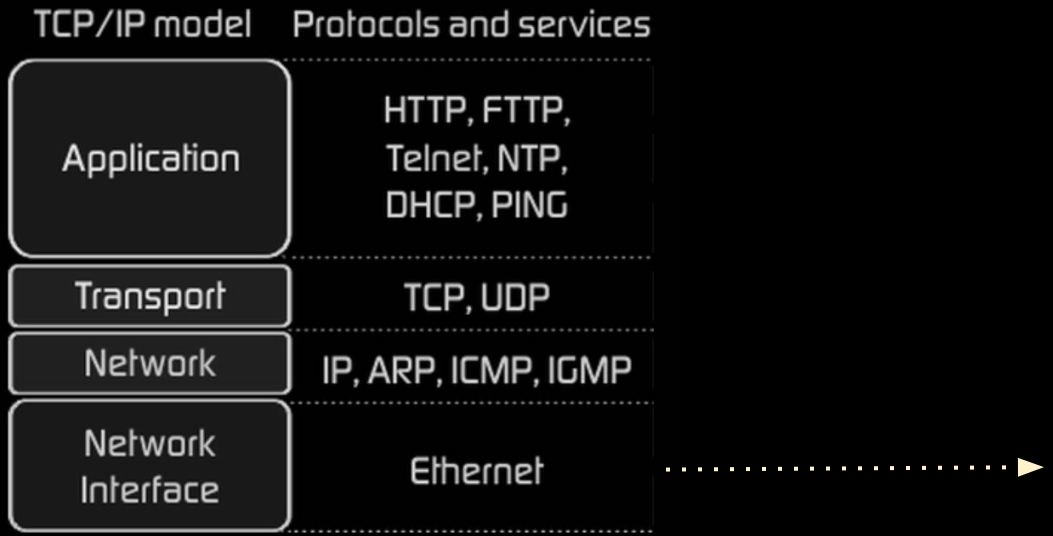
TEC | Tecnológico de Costa Rica

WEB SERVICES

CE-5508

BRIEF INTRODUCTION TO TCP/IP MODEL

The TCP/IP reference model is a layered model developed by the Defense Project Research Agency(ARPA or DARPA) of the United States as a part of their research project in 1960. Initially, it was developed to be used by defense only. But later on, it got widely accepted.

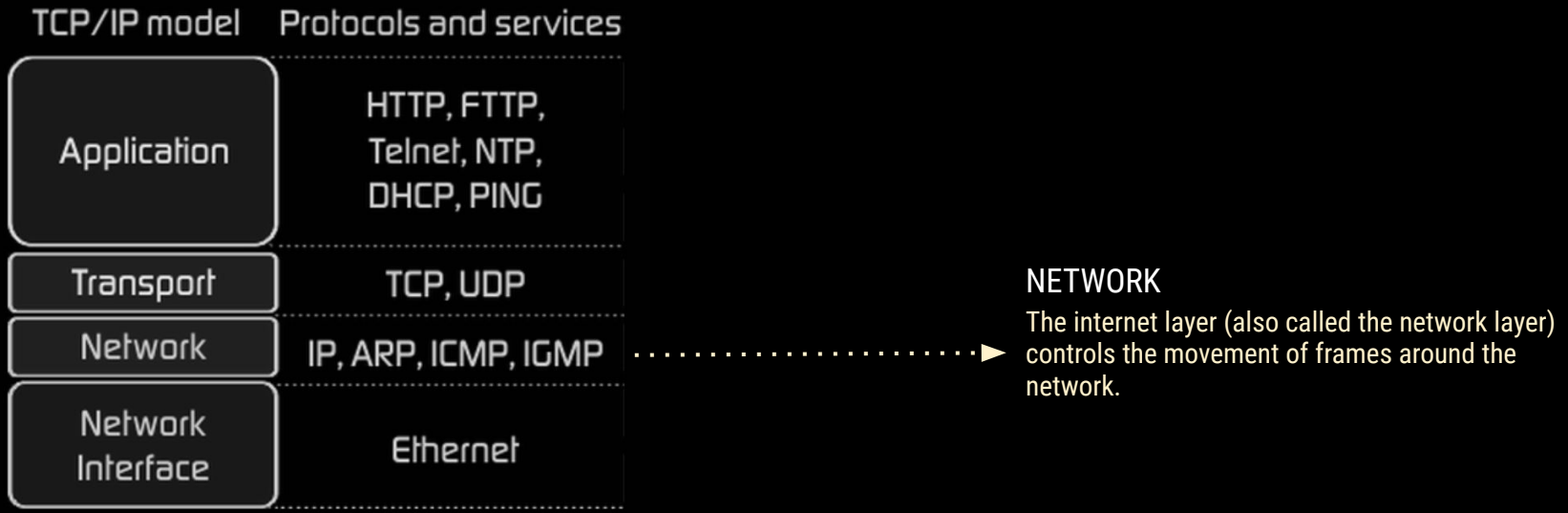


NETWORK INTERFACE

The data link layer (also called the link layer, network interface layer, or physical layer) is what handles the physical parts of sending and receiving data using the Ethernet cable, wireless network, network interface card, device driver in the computer, and so on.

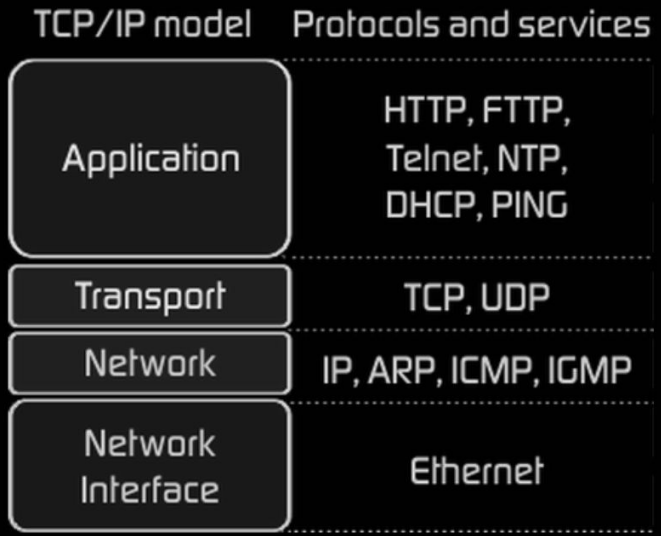
BRIEF INTRODUCTION TO TCP/IP MODEL

The TCP/IP reference model is a layered model developed by the Defense Project Research Agency(ARPA or DARPA) of the United States as a part of their research project in 1960. Initially, it was developed to be used by defense only. But later on, it got widely accepted.



BRIEF INTRODUCTION TO TCP/IP MODEL

The TCP/IP reference model is a layered model developed by the Defense Project Research Agency(ARPA or DARPA) of the United States as a part of their research project in 1960. Initially, it was developed to be used by defense only. But later on, it got widely accepted.

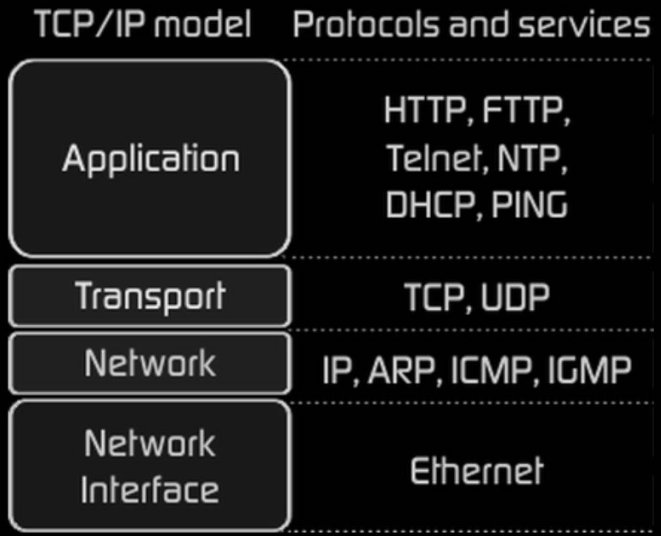


TRANSPORT

The transport layer is what provides a reliable data connection between two devices. It divides the data in packets, acknowledges the packets that it has received from the other device, and makes sure that the other device acknowledges the packets it receives.

BRIEF INTRODUCTION TO TCP/IP MODEL

The TCP/IP reference model is a layered model developed by the Defense Project Research Agency(ARPA or DARPA) of the United States as a part of their research project in 1960. Initially, it was developed to be used by defense only. But later on, it got widely accepted.

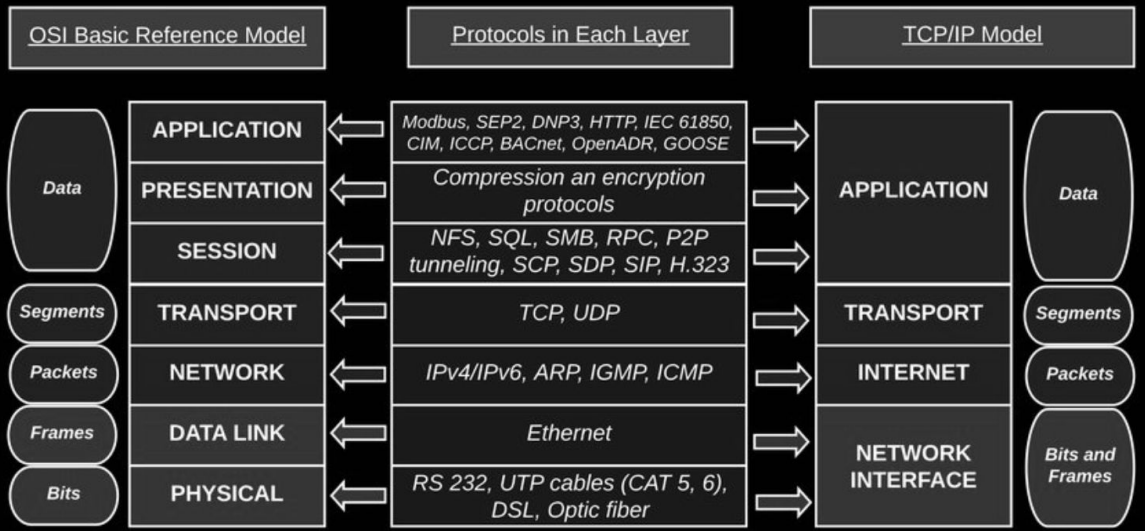


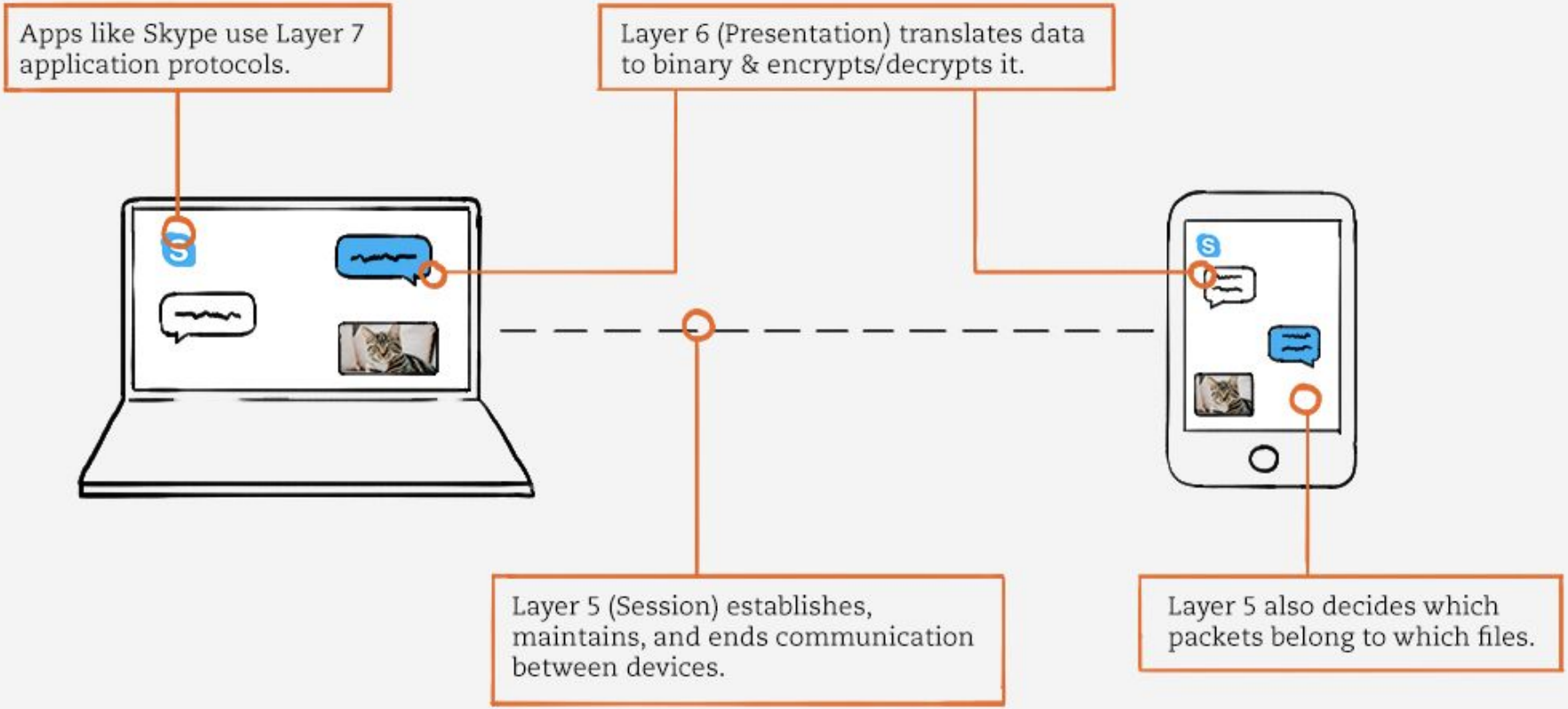
APPLICATION

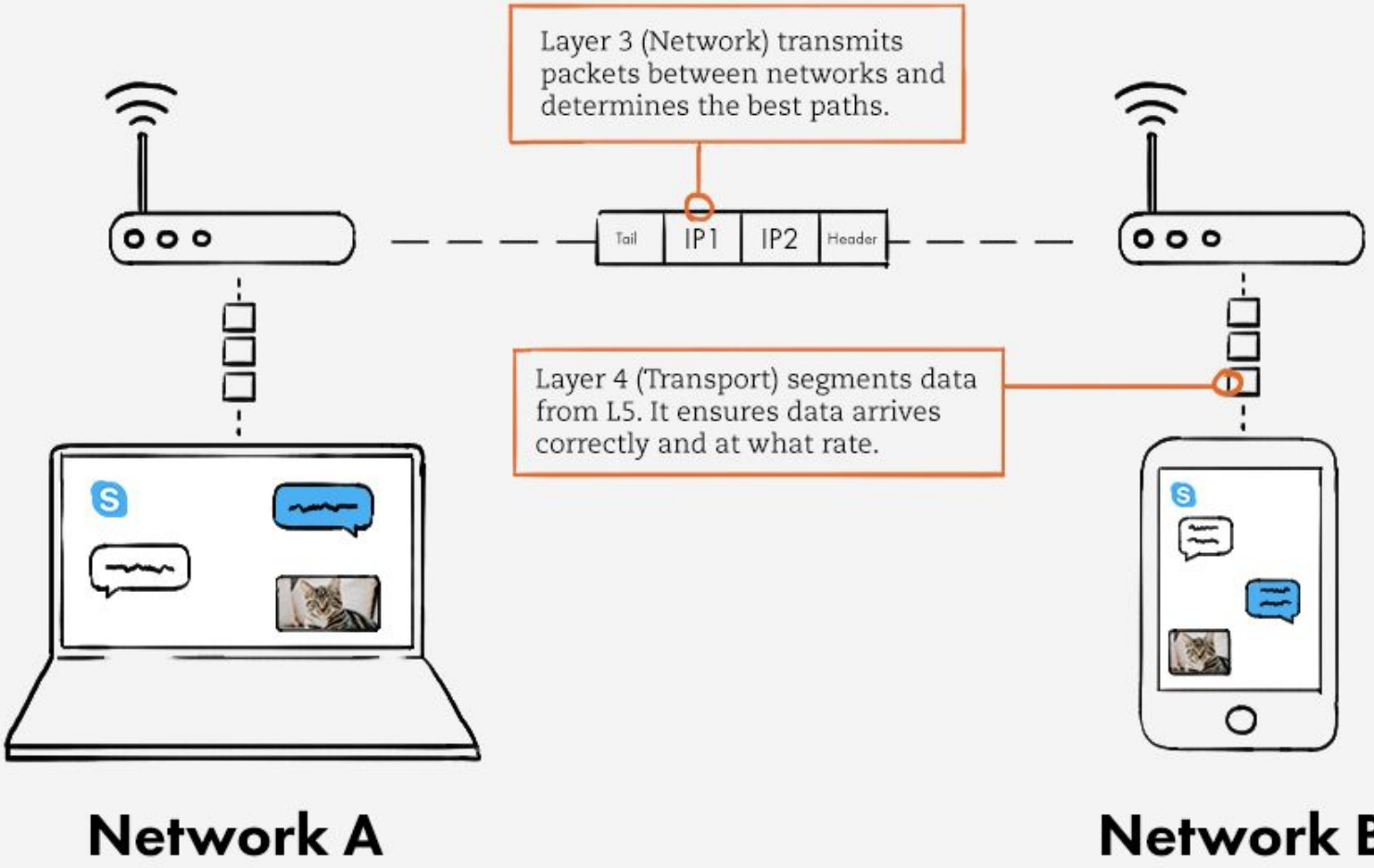
The application layer is the group of applications that require network communication. This is what the user typically interacts with, such as email and messaging. Because the lower layers handle the details of communication, the applications don't need to concern themselves with this

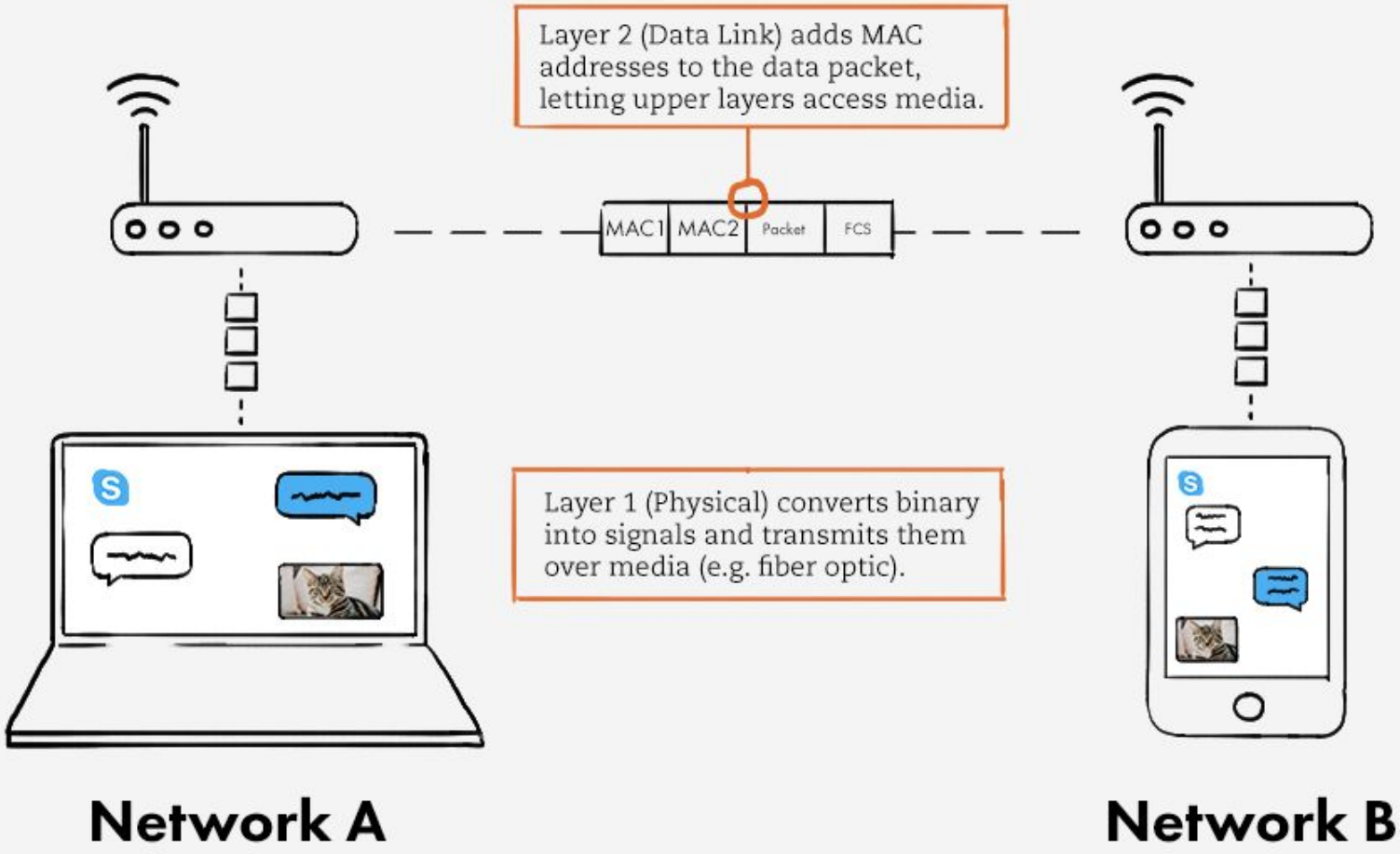
THE OSI MODEL

While TCP/IP is the newer model, the Open Systems Interconnection (OSI) model is still referenced a lot to describe network layers. The OSI model was developed by the International Organization for Standardization.




Apps like Skype use Layer 7 application protocols.
Layer 6 (Presentation) translates data to binary & encrypts/decrypts it.
Layer 5 (Session) establishes, maintains, and ends communication between devices.
Layer 5 also decides which packets belong to which files.


Layer 3 (Network) transmits packets between networks and determines the best paths.
Tail
IP1
IP2
Header
Layer 4 (Transport) segments data from L5. It ensures data arrives correctly and at what rate.
Network A
Network B


Layer 2 (Data Link) adds MAC addresses to the data packet, letting upper layers access media.
MAC1
MAC2
Packet
FCS
Layer 1 (Physical) converts binary into signals and transmits them over media (e.g. fiber optic).
Network A
Network B

HTTP: Hypertext Transfer Protocol

Is the foundation of data communication for WWW. This protocol defines how messages are formatted, transmitted, and processed over the Internet.

- There are three versions. HTTP/0.9 was the first documented version, which was released in the year 1991. This was very primitive and supported only the GET method. Later, HTTP/1.0 was released in the year 1996 with more features and corrections for the shortcomings in the previous release. HTTP/1.0 supported more request methods such as GET, HEAD, and POST. The next release was HTTP/1.1 in the year 1999. This was the revision of HTTP/1.0. This version is in common use today.

- HTTP/2 (originally named HTTP 2.0) is the next planned version. It is mainly focused on how the data is framed and transported between the client and the server.

HTTP: REQUEST-RESPONSE MODEL

The user enters the following URL in the browser, http://www.example.com/index.html, and then submits the request.

The browser establishes a connection with the server and sends a request to the server in the form of a request method, URI, and protocol version, followed by a message containing request modifiers, client information, and possible body content

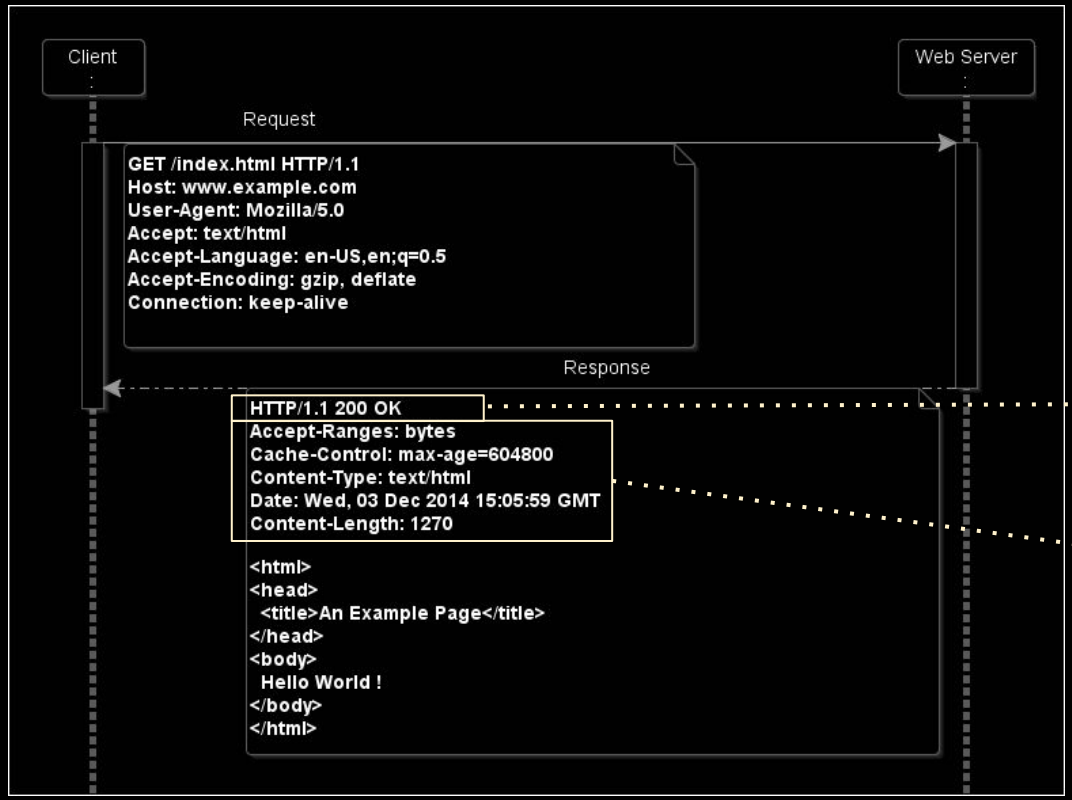
HTTP: REQUEST-RESPONSE MODEL

METHOD (VERB, COMMAND) URI HTTP-VERSION

HEADERS SENT TO THE SERVER

The server can use this information while preparing the response for the request.

HTTP: REQUEST-RESPONSE MODEL



STATUS LINE

The status code indicates one of the following parameters: informational codes, success of the request, client error, server error, or redirection of the request

HEADERS SENT TO THE CLIENT

Useful information about the resource being fetched, the server hosting the resource, and some parameters controlling the client behavior while dealing with the resource, such as content type, cache expiry, and refresh rate.

HTTP: REQUEST-RESPONSE MODEL

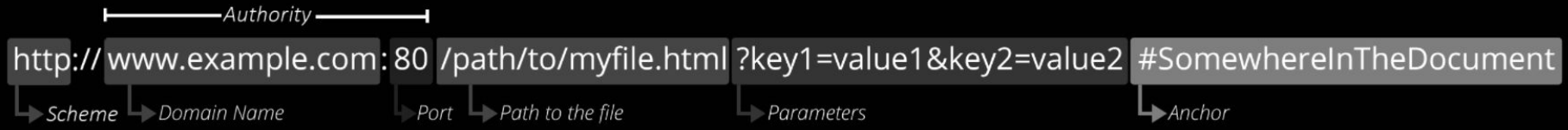
RESPONSE BODY

It can be HTML, binary data, image, video, text, XML, JSON, and so on. Once the response body has been sent to the requestor

HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

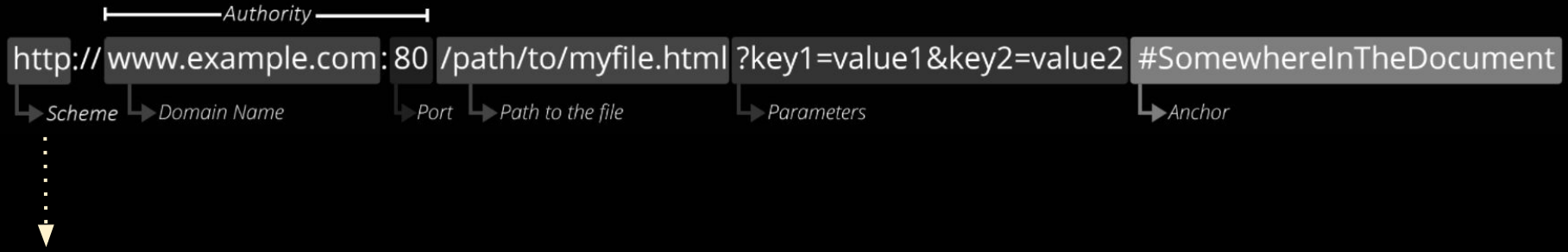
> In general, all URLs are URIs



HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

> In general, all URLs are URIs

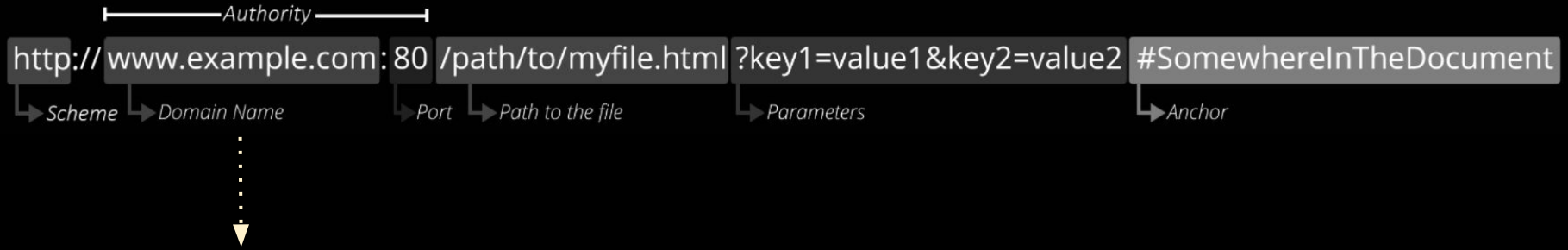


The first part of the URL is the *scheme*, which indicates the protocol that the browser must use to request the resource. Usually for websites the protocol is HTTPS or HTTP (its unsecured version). Addressing web pages requires one of these two, but browsers also know how to handle other schemes such as mailto: (to open a mail client) or ftp: to handle file transfer

HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

> In general, all URLs are URIs

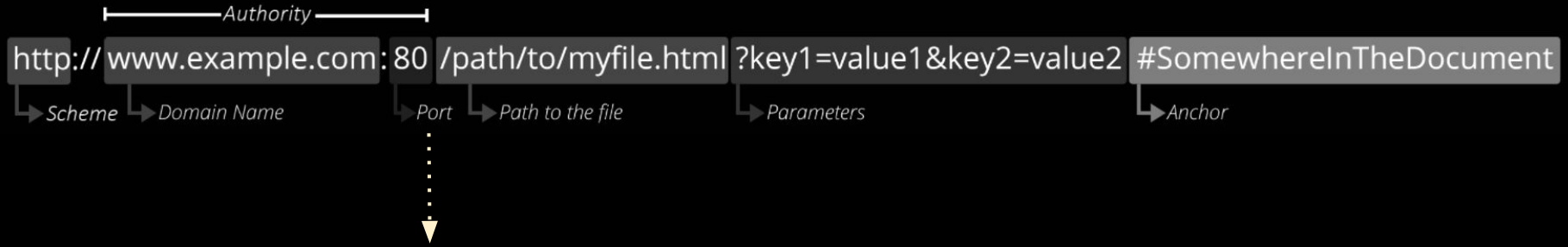


Indicates which Web server is being requested. Usually this is a domain name, but an IP address may also be used

HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

> In general, all URLs are URIs

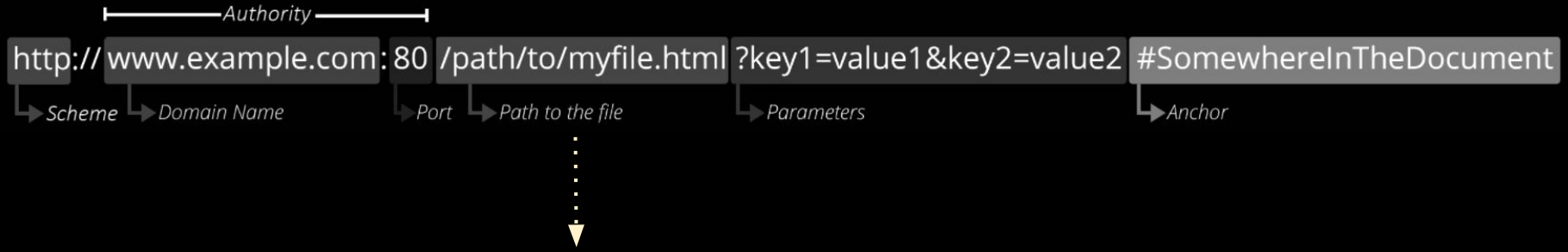


Indicates the technical "gate" used to access the resources on the web server. It is usually omitted if the web server uses the standard ports of the HTTP protocol (80 for HTTP and 443 for HTTPS) to grant access to its resources. Otherwise it is mandatory

HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

> In general, all URLs are URIs

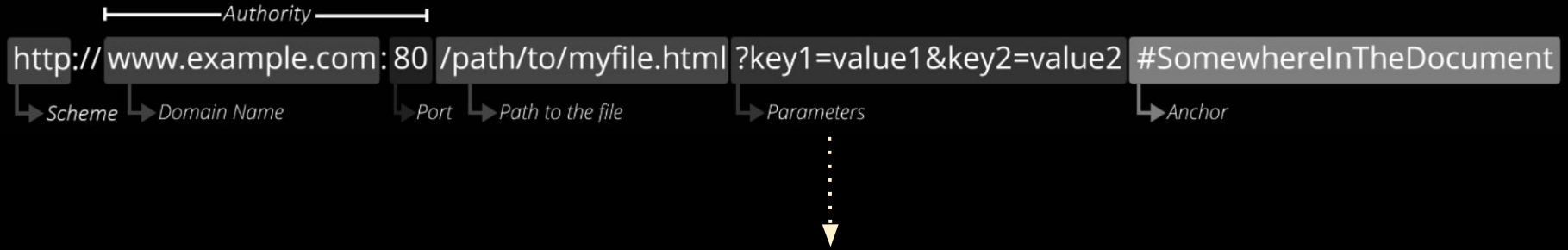


Is the path to the resource on the Web server. In the early days of the Web, a path like this represented a physical file location on the Web server. Nowadays, it is mostly an abstraction handled by Web servers without any physical reality

HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

> In general, all URLs are URIs



?key1=value1&key2=value2 are extra parameters provided to the Web server. Those parameters are a list of key/value pairs separated with the & symbol. The Web server can use those parameters to do extra stuff before returning the resource

HTTP: UNIFORM RESOURCE IDENTIFIER (URI)

A URI is a text that identifies any resource or name on the Internet. One can further classify a URI as a **Uniform Resource Locator** (**URL**) if the text used for identifying the resource also holds the means for accessing the resource such as HTTP or FTP

> In general, all URLs are URIs

Is an anchor to another part of the resource itself. An anchor represents a sort of "bookmark" inside the resource, giving the browser the directions to show the content located at that "bookmarked" spot

HTTP: REQUEST METHODS

GET
This method is used for retrieving resources from the server by using the given URI.

PUT
This method is used for updating the resource pointed at by the URI. If the URI does not point to an existing resource, the server can create the resource with that URI.

OPTIONS
This method returns the HTTP methods that the server supports for the specified URI.

HEAD
This method is the same as the GET request, but it only transfers the status line and the header section without the response body.

DELETE
This method deletes the resource pointed at by the URI.

CONNECT
This method is used for establishing a connection to the target server over HTTP.

POST
This method is used for posting data to the server. The server stores the data (entity) as a new subordinate of the resource identified by the URI. If you execute POST multiple times on a resource, it may yield different results

TRACE
This method is used for echoing the contents of the received request. This is useful for the debugging purpose with which the client can see what changes (if any) have been made by the intermediate servers.

PATCH
This method is used for applying partial modifications to a resource identified by the URI.

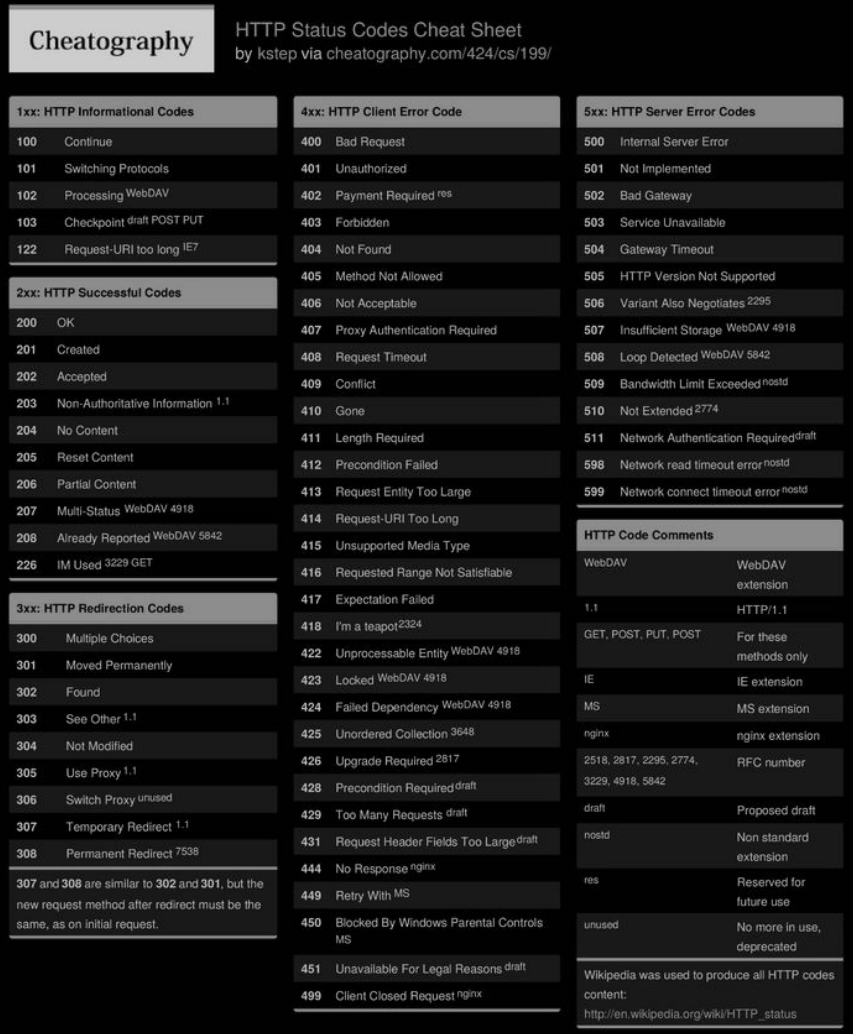
HTTP: CONTENT TYPES

The Content-Type header in an HTTP request or response describes the content type for the message body. The Accept header in the request tells the server the content types that the client is expecting in the response body. The content types are represented using the Internet media type. The Internet media type (also known as the MIME type) indicates the type of data that a file contains

- **text**: This type indicates that the content is plain text and no special software is required to read the contents. The subtype represents more specific details about the content, which can be used by the client for special processing, if any. For instance, Content-Type: text/html
- **multipart**: As the name indicates, this type consists of multiple parts of the independent data types. For instance, Content-Type: multipart/form-data is used for submitting forms that contain the files, non-ASCII data, and binary data.
- **image**: This type represents the image data. For instance, Content-Type: image/png indicates that the body content is a .png image.
- **audio**: This type indicates the audio data. For instance, Content-Type: audio/mpeg indicates that the body content is MP3 or other MPEG audio.
- **video**: This type indicates the video data. For instance, Content-Type: video/mp4 indicates that the body content is MP4 video.
- **application**: This type represents the application data or binary data. For instance, Content-Type: application/json; charset=utf-8 designates the content to be in the **JavaScript Object Notation** (**JSON**) format, encoded with UTF-8 character encoding.

HTTP: STATUS CODE

For every HTTP request, the server returns a status code indicating the processing status of the request.



Cheatography

HTTP Status Codes Cheat Sheet
by kstep via cheatography.com/424/cs/199/

1xx: HTTP Informational Codes

100	Continue
101	Switching Protocols
102	Processing WebDAV
103	Checkpoint draft POST PUT
122	Request-URI too long IE7

2xx: HTTP Successful Codes

200	OK
201	Created
202	Accepted
203	Non-Authoritative Information 1.1
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status WebDAV 4918
208	Already Reported WebDAV 5842
226	IM Used 3229 GET

3xx: HTTP Redirection Codes

300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other 1.1
304	Not Modified
305	Use Proxy 1.1
306	Switch Proxy unused
307	Temporary Redirect 1.1
308	Permanent Redirect 7538

307 and **308** are similar to **302** and **301**, but the new request method after redirect must be the same, as on initial request.

4xx: HTTP Client Error Code

400	Bad Request
401	Unauthorized
402	Payment Required res
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot 2324
422	Unprocessable Entity WebDAV 4918
423	Locked WebDAV 4918
424	Failed Dependency WebDAV 4918
425	Unordered Collection 3648
426	Upgrade Required 2817
428	Precondition Required draft
429	Too Many Requests draft
431	Request Header Fields Too Large draft
444	No Response nginx
449	Retry With MS
450	Blocked By Windows Parental Controls MS
451	Unavailable For Legal Reasons draft
499	Client Closed Request nginx

5xx: HTTP Server Error Codes

500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates 2295
507	Insufficient Storage WebDAV 4918
508	Loop Detected WebDAV 5842
509	Bandwidth Limit Exceeded nostd
510	Not Extended 2774
511	Network Authentication Required draft
598	Network read timeout error nostd
599	Network connect timeout error nostd

HTTP Code Comments

WebDAV	WebDAV extension
1.1	HTTP/1.1
GET, POST, PUT, POST	For these methods only
IE	IE extension
MS	MS extension
nginx	nginx extension
2518, 2817, 2295, 2774, 3229, 4918, 5842	RFC number
draft	Proposed draft
nostd	Non standard extension
res	Reserved for future use
unused	No more in use, deprecated

Wikipedia was used to produce all HTTP codes content:
http://en.wikipedia.org/wiki/HTTP_status

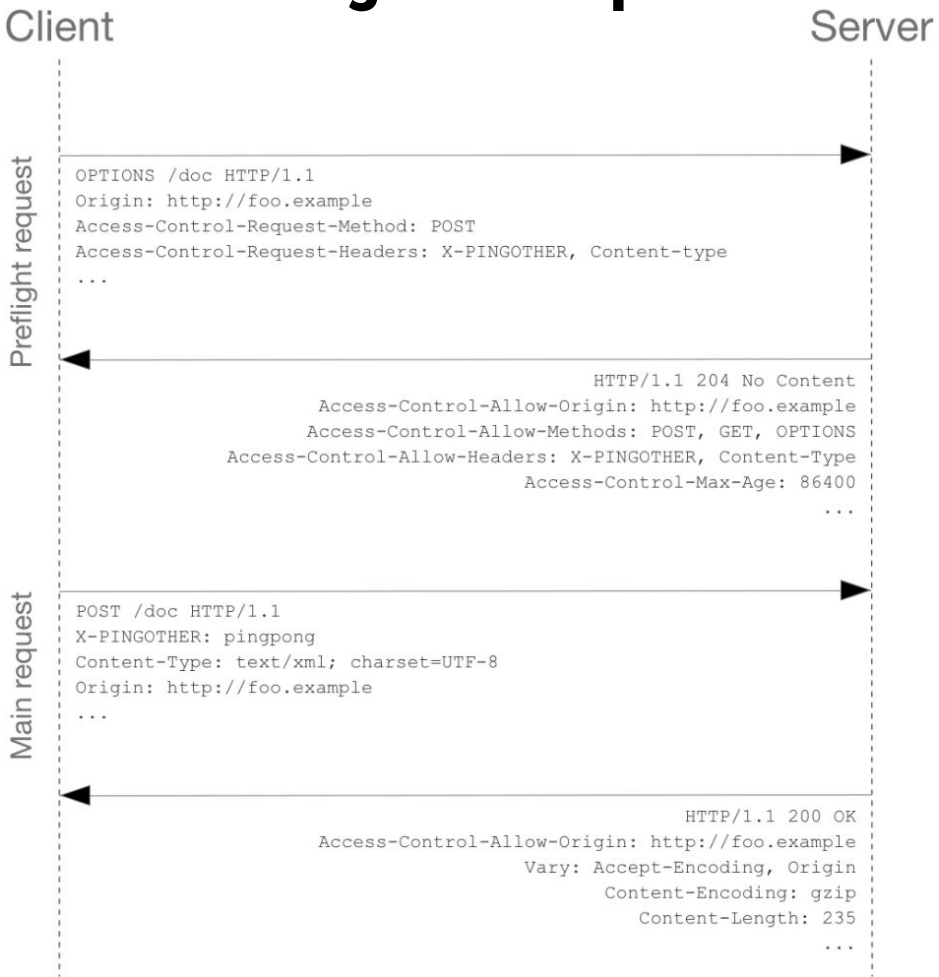
HTTP: CROSS ORIGIN RESOURCE SHARING (CORS)

HTTP-header based mechanism that allows a server to indicate any other origins than its own from which a browser should permit loading of resources. Browsers make a “preflight” request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

> For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequest and the Fetch API follow the same-origin policy.
>
> This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.

Simple Request (No Preflight)

Preflighted request



HTTP: COOKIES

A cookie

WEB SERVICE DEFINITION

> A Web service is a software package that is used for communicating between two devices or web entities lying on the network. They involve a service provider along with a service requester, i.e., the client.
>
> W3SCHOOLS.IN

> The World Wide Web is more and more used for application to application communication. The programmatic interfaces made available are referred to as Web services.
>
> W3C

WEB API

In simple terms, is the same as a Web Service. Web Service is mainly used to refer to SOAP and Web API refers to other types of Web Service. For our purposes, it is the same.

A BRIEF COMPARISON

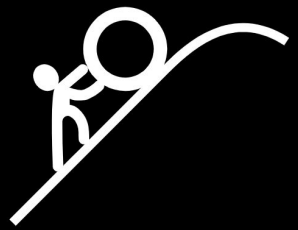
Architectural Styles ◄

	RPC	SOAP	REST	GraphQL
Organized in terms of	local procedure calling	enveloped message structure	compliance with six architectural constraints	schema & type system
Format	JSON, XML, Protobuf, Thrift, FlatBuffers	XML only	XML, JSON, HTML, plain text,	JSON
Learning curve	Easy	Difficult	Easy	Medium
Community	Large	Small	Large	Growing
Use cases	Command and action-oriented APIs; internal high performance communication in massive micro-services systems	Payment gateways, identity management CRM solutions financial and telecommunication services, legacy system support	Public APIs simple resource-driven apps	Mobile APIs, complex systems, micro-services

altexsoft
software r&d engineering

RPC: Remote Procedure Call

A **Remote Procedure Call** is a specification that allows for remote execution of a function in a different context. RPC extends the notion of local procedure calling but puts it in the context of an HTTP API.



Initial XML-RPC was problematic because ensuring data types of XML payloads is tough. So, later an RPC API started using a more concrete JSON-RPC specification which is considered a simpler alternative to SOAP.

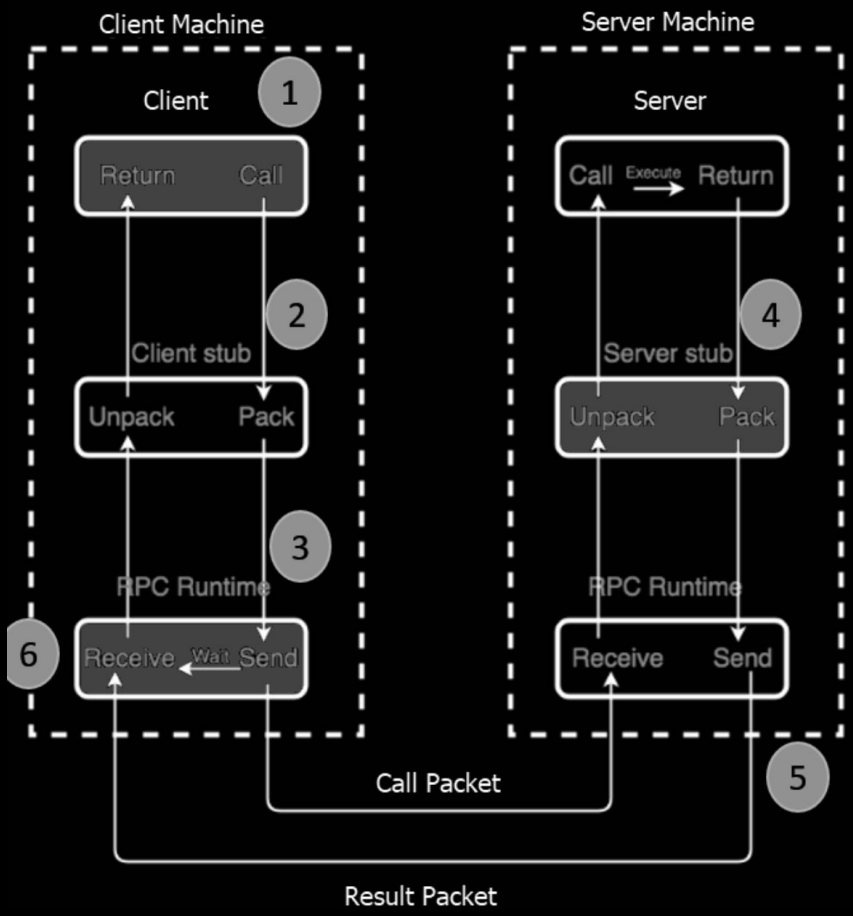
gRPC is the latest RPC version developed by Google in 2015. With pluggable support for load balancing, tracing, health checking, and authentication, gRPC is well-suited for connecting microservices.

RPC: HOW IT WORKS

A client invokes a remote procedure, serializes the parameters and additional information into a message, and sends the message to a server.

On receiving the message, the server deserializes its content, executes the requested operation, and sends a result back to the client.

The server stub and client stub take care of the serialization and deserialization of the parameters.



RPC: PROS

Straightforward and simple interaction. RPC uses GET to fetch information and POST for everything else.

Easy-to-add functions. If we get a new requirement for our API, we can easily add another endpoint executing this requirement: 1) Write a new function and throw it behind an endpoint and 2) now a client can hit this endpoint and get the info meeting the set requirement.

High performance. Lightweight payloads go easy on the network providing high performance. RPC is able to optimize the network layer and make it very efficient with sending tons of messages per day between different services.

RPC: CONS

Tight coupling to the underlying system. RPC's tight coupling to the underlying system doesn't allow for an abstraction layer between the functions in the system and the external API. This raises security issues as it's quite easy to leak implementation details about the underlying system into the API.

Low discoverability. In RPC there's no way to introspect the API or send a request and start understanding what function to call based on its requests.

Function explosion. It's so easy to create new functions. So, instead of editing the existing ones, we create new ones ending up with a huge list of overlapping functions that are hard to understand.

RPC: USE CASES

The RPC pattern started being used around the 80s, but this doesn't automatically make it obsolete. Big companies like Google, Facebook (Apache Thrift), and Twitch (Twirp) are using RPC high-performance variates internally to perform extremely high-performance, low-overhead messaging. Their massive microservices systems require internal communication to be clear while arranged in short messages.

Command API. An RPC is the proper choice for sending commands to a remote system. For instance, a Slack API is very command-focused: Join a channel, leave a channel, send a message.

Customer-specific APIs for internal microservices. Using HTTP 2 under the hood, gRPC is able to optimize the network layer and make it very efficient with sending tons of messages per day between different services..

SOAP: SIMPLE OBJECT ACCESS PROTOCOL

SOAP is an XML-formatted, highly standardized web communication protocol. Released by Microsoft a year after XML-RPC, SOAP inherited a lot from it. When REST followed, they were first used in parallel, but soon REST won the popularity contest.

HOW IT WORKS

XML data format drags behind a lot of formality. Paired with the massive message structure, it makes SOAP the most verbose API style. A SOAP message is composed of:

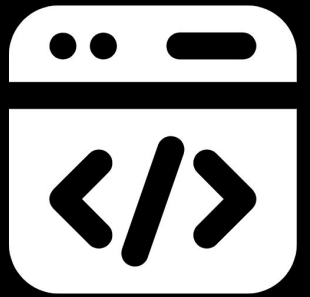
- An envelope tag that begins and ends every message,
- A body containing the request or response
- A header if a message must determine any specifics or extra requirements, and
- A fault informing of any errors that can occur throughout the request processing.

```
<?xml version='1.0' Encoding='UTF-8' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
   <m:dateAndTime>2007-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
   <n:name>Fred Bloggs</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2007-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2007-12-20</p:departureDate>
     <p:departureTime>mid-morning</p:departureTime>
     <p:seatPreference></p:seatPreference>
   </p:return>
  </p:itinerary>
 </env:Body>
</env:Envelope>
```

SOAP: WEB SERVICE DESCRIPTION LANGUAGE (WSDL)

The SOAP API logic is written in Web Service Description Language (WSDL). This API description language defines the endpoints and describes all processes that can be performed. This allows different programming languages and IDEs to quickly set up communication.

SOAP supports both stateful and stateless messaging. In a stateful scenario, the server stores the received information that can be really heavy. But it’s justified for operations involving multiple parties and complex transactions.



SOAP: PROS

Language- and platform-agnostic. The built-in functionality to create web-based services allows SOAP to handle communications and make responses language- and platform-independent.

Bound to a variety of transport protocols. SOAP is flexible in terms of transfer protocols to accommodate for multiple scenarios.

Built-in error handling. SOAP API specification allows for returning the Retry XML message with error code and its explanation.

A number of security extensions. Integrated with the WS-Security protocols, SOAP meets an enterprise-grade transaction quality. It provides privacy and integrity inside the transactions while allowing for encryption on the message level.

SOAP: CONS

XML only. SOAP messages contain a lot of metadata and only support verbose XML structures for requests and responses.

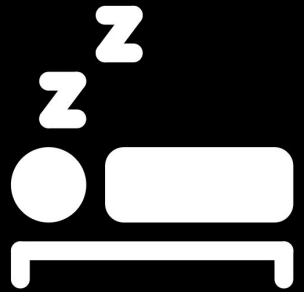
Heavyweight. Due to the large size of XML-files, SOAP services require a large bandwidth.

Narrowly specialized knowledge. Building SOAP API servers requires a deep understanding of all protocols involved and their highly restricted rules.

Tedious message updating. Requiring additional effort to add or remove the message properties, rigid SOAP schema slows down adoptio

REST: REPRESENTATIONAL STATE TRANSFER

REST is a self-explanatory API architectural style defined by a set of architectural constraints and intended for wide adoption with many API consumers. The most common API style today was originally described in 2000 by Roy Fielding in his doctoral dissertation. REST makes server-side data available representing it in simple formats, often JSON and XML



REST: HOW IT WORKS

RESTful architecture should comply with six architectural constraints:

- **uniform interface**
- **stateless**
- **caching**
- **client-server architecture**
- **layered system**

The uniform interface constraint is fundamental to the design of any RESTful system.[1] It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

Resource identification in requests

Individual resources are identified in requests, for example using URIs in RESTful Web services. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server could send data from its database as HTML, XML or as JSON—none of which are the server's internal representation.

Resource manipulation through representations

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource's state.

Self-descriptive messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type.[1]

Hypermedia as the engine of application state (HATEOAS)

Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other resources that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the application.[12]

REST: HOW IT WORKS

RESTful architecture should comply with six architectural constraints:

- **uniform interface**
- **stateless**
- **caching**
- **client-server architecture**
- **layered system**

In computing, a stateless protocol is a communications protocol in which no session information is retained by the receiver, usually a server. Relevant session data is sent to the receiver by the client in such a way that every packet of information transferred can be understood in isolation, without context information from previous packets in the session. This property of stateless protocols makes them ideal in high volume applications, increasing performance by removing server load caused by retention of session information.

REST: HOW IT WORKS

RESTful architecture should comply with six architectural constraints:

- **uniform interface**
- **stateless**
- **caching** ⋯⋯⋯⋯⋯⋯⋯⋯⋯⋯► As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from providing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.
- **client-server architecture**
- **layered system**

REST: HOW IT WORKS

RESTful architecture should comply with six architectural constraints:

- **uniform interface**
- **stateless**
- **caching**
- **client-server architecture** ⋯⋯► The principle behind the client–server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interfaces across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains
- **layered system**

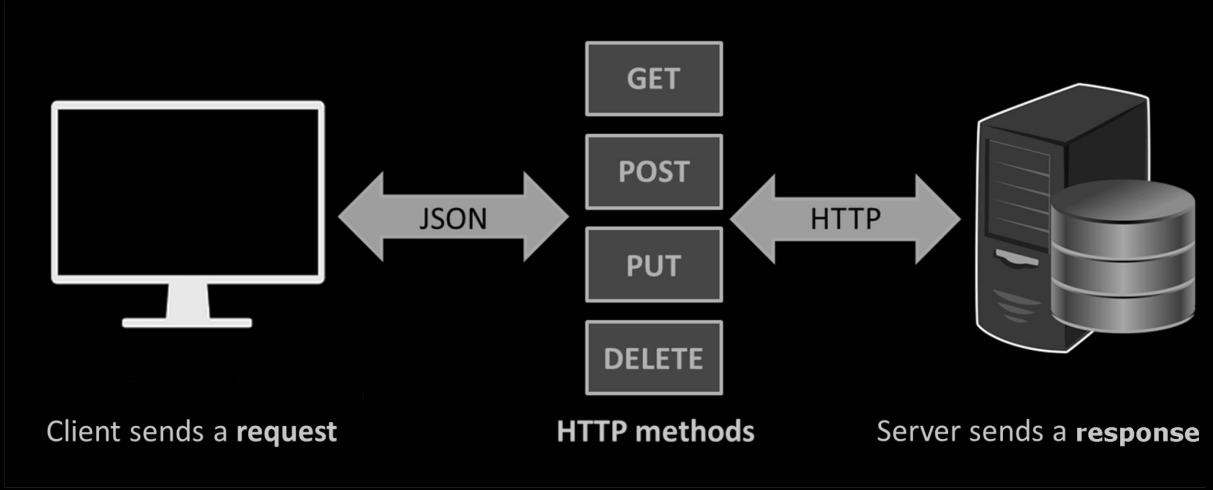
REST: HOW IT WORKS

RESTful architecture should comply with six architectural constraints:

- **uniform interface**
- **stateless**
- **caching**
- **client-server architecture**
- **layered system** . . . . . ► A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. If a proxy or load balancer is placed between the client and server, it won't affect their communications, and there won't be a need to update the client or server code. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. Also, security can be added as a layer on top of the web services, separating business logic from security logic. Adding security as a separate layer enforces security policies. Finally, intermediary servers can call multiple other servers to generate a response to the client.

REST: HOW IT WORKS

In REST, things are done using HTTP methods such as GET, POST, PUT, DELETE, OPTIONS, and, hopefully, PATCH.



REST: PROS

Decoupled client and server. Decoupling the client and the server as much as possible, REST allows for a better abstraction than RPC. A system with abstraction levels is able to encapsulate its details to better identify and sustain its properties. This makes a REST API flexible enough to evolve over time while remaining a stable system.

Discoverability. Communication between the client and server describes everything so that no external documentation is required to understand how to interact with the REST API.

Cache-friendly. Reusing a lot of HTTP tools, REST is the only style that allows caching data on the HTTP level. In contrast, caching implementation on any other API will require configuring an additional cache module.

Multiple formats support. The ability to support multiple formats for storing and exchanging data is one of the reasons REST is currently a prevailing choice for building public APIs.

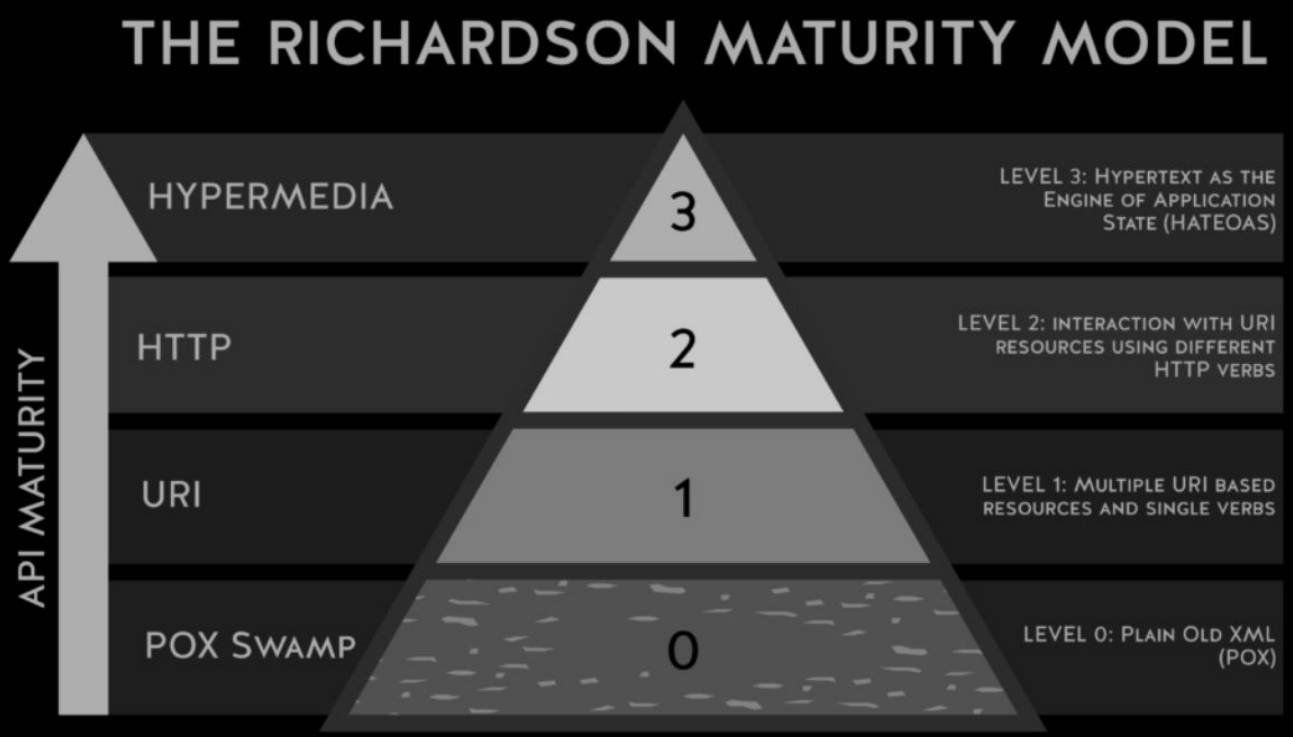
REST: CONS

No single REST structure. There's no exact right way to build a REST API. How to model resources and which resources to model will depend on each scenario. This makes REST simple in theory, but difficult in practice.

Big payloads. REST returns a lot of rich metadata so that the client can understand everything necessary about the state of the application just from its responses. And this chattiness is no big deal for a big network pipe with lots of bandwidth capacity. But that's not always the case. This was the key driving factor for Facebook coming up with the description of GraphQL style in 2012.

Over- and under-fetching problems. Containing either too much data or not enough of it, REST responses often create the need for another request.

REST: MATURITY



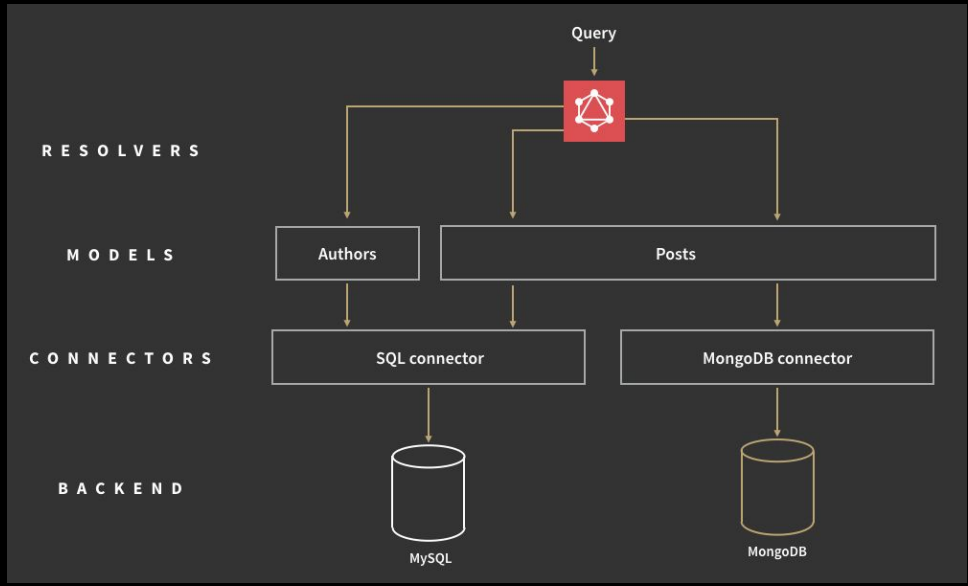
GRAPHQL

Is a syntax that describes how to make a precise data request. Implementing GraphQL is worth it for an application's data model with a lot of complex entities referencing each other.

GRAPHQL: HOW IT WORKS

GraphQL starts with building a *schema*, which is a description of all the queries you can possibly make in a GraphQL API and all the *types* that they return. Schema-building is hard as it requires strong typing in the Schema Definition Language (SDL).

Having the schema before querying, a client can validate their query against making sure the server will be able to respond to it. On reaching the backend application, a GraphQL operation is interpreted against the entire schema, and resolved with data for the frontend application. Sending one massive query to the server, the API returns a JSON response with exactly the shape of the data we asked for.



GRAPHQL: PROS

Typed schema. GraphQL publishes in advance what it can do, which improves its discoverability. By pointing a client at the GraphQL API, we can find out what queries are available.

Fits graph-like data very well. Data that goes far into linked relations but not good for flat data.

No versioning. The best practice with versioning is not to version the API at all. While REST offers multiple API versions, GraphQL uses a single, evolving version that gives continuous access to new features and contributes to cleaner, more maintainable server code.

Detailed error messages. In a similar fashion to SOAP, GraphQL provides details to errors that occurred. Its error message includes all the resolvers and refers to the exact query part at fault.

Flexible permissions. GraphQL allows for selectively exposing certain functions while preserving private information. Meanwhile, REST architecture doesn't reveal data in portions. It's either all or nothing.

GRAPHQL: CONS

Performance issues. GraphQL trades off complexity for its power. Having too many nested fields in one request can lead to system overload. So, REST remains a better option for complex queries.

Caching complexity. As GraphQL isn't reusing HTTP caching semantics, it requires a custom caching effort.

A lot of pre-development education. Not having enough time to figure out GraphQL niche operations and SDL, many projects decide to follow the well-known path of REST.

References

Comparing SOAP vs REST vs GraphQL vs RPC API | AltexSoft

Functions of Networking System: OSI Model | by Ravidu Perera | May, 2021 | Level Up Coding (gitconnected.com)

Network Layers Explained: OSI & TCP/IP Models [with examples] (plixer.com)

TEC | Tecnológico de Costa Rica

WEB SERVICES

CE-5508