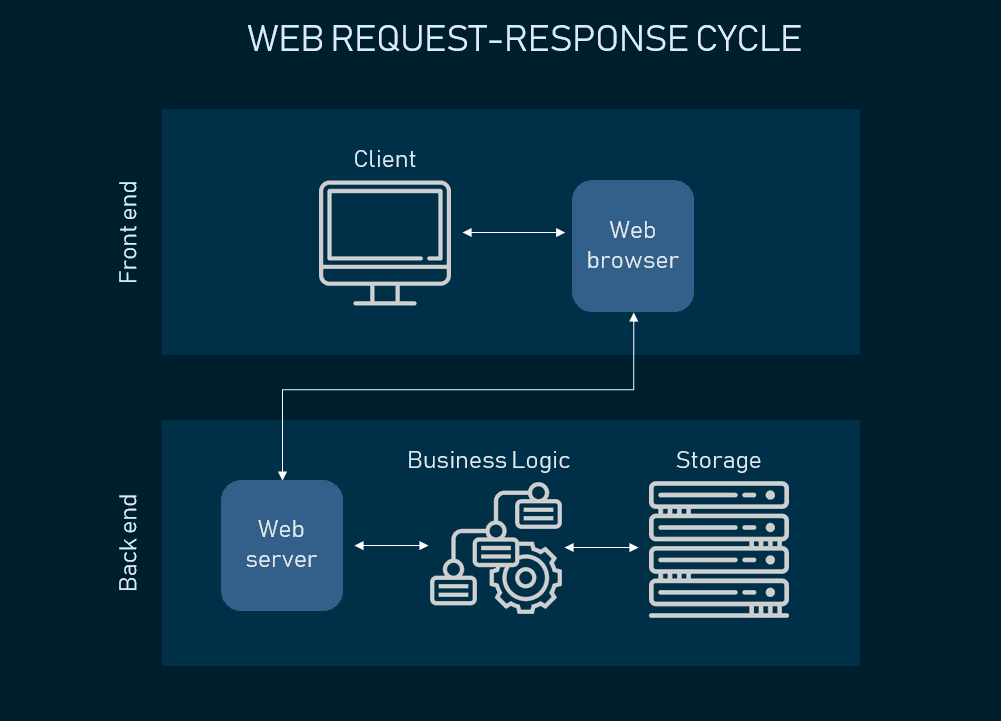
**ADVANCED WEB PROGRAMMING**

**Architecture of the Web**

* From a software engineer’s perspective, a web application should be scalable, functional, and able to withstand high traffic loads.
* All these issues are addressed in the web application’s architecture.
* **Web application architecture** is a mechanism that determines how application components communicate with each other. Or, in other words, the way the client and the server are connected is established by web application architecture.
* To understand the components of web application architecture, we need to understand how they are used in performing the most basic action – receiving and responding to a web request.

**Client-Server Architecture**



**Three-Tier Architecture**

Most web applications are developed by separating its main functions into layers, or tiers. This allows you to easily replace and upgrade each layer independently. This architectural pattern is called Multi- or Three-Tier Architecture.

**Presentation layer**

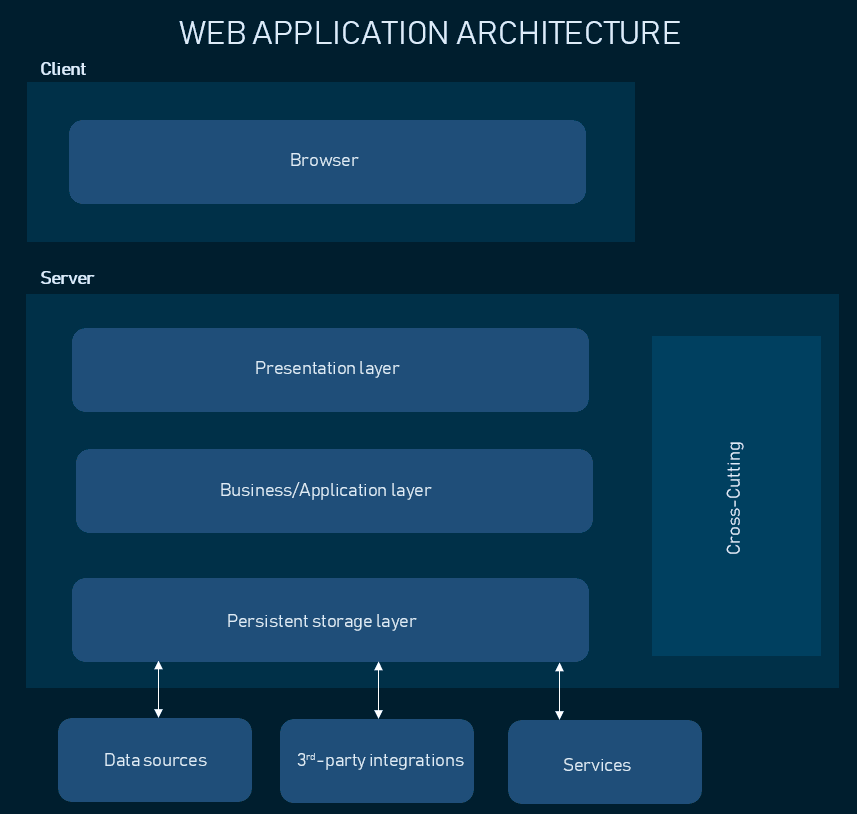
The presentation layer is accessible to users via a browser and consists of user interface components and UI process components that support interaction with the system. It’s developed using three core technologies: HTML, CSS, and JavaScript. While HTML is the code that determines what your website will contain, CSS controls how it will look. JavaScript and its frameworks make your website interactive – responsive to a user’s actions. Developers use JavaScript frameworks such as **Angular** and **React** to make the content on the page dynamic.

**Business layer**

This layer, also called Business Logic or Domain Logic or Application Layer, accepts user requests from the browser, processes them, and determines the routes through which the data will be accessed. The workflows by which the data and requests travel through the back end are encoded in a business layer.

**Persistence layer**

Also called the storage or data access layer, the persistence layer is a centralized location that receives all data calls and provides access to the persistent storage of an application. The persistence layer is closely connected to the business layer, so the logic knows which database to talk to and the data retrieving process is more optimized.



The data storage infrastructure includes a server and a Database Management System, software to communicate with the database itself, applications, and user interfaces to obtain data and parse it. Typically you can store your data either in owned hardware servers or in the cloud – meaning, that you purchase data center management and maintenance services while accessing your storage virtually. Using the services of [cloud technology providers](https://www.altexsoft.com/blog/cloud/comparing-serverless-architecture-providers-aws-azure-google-ibm-and-other-faas-vendors/) such as Amazon, Google, or Microsoft, you can utilize Infrastructure-as-a-Service, Platform-as-a-Service, or serverless approaches to cloud management.

There are also components that usually exist in all web applications but are separated from the main layers:

**Cross-cutting code.**This component handles other application concerns such as communications, operational management, and security. It affects all parts of the system but should never mix with them.

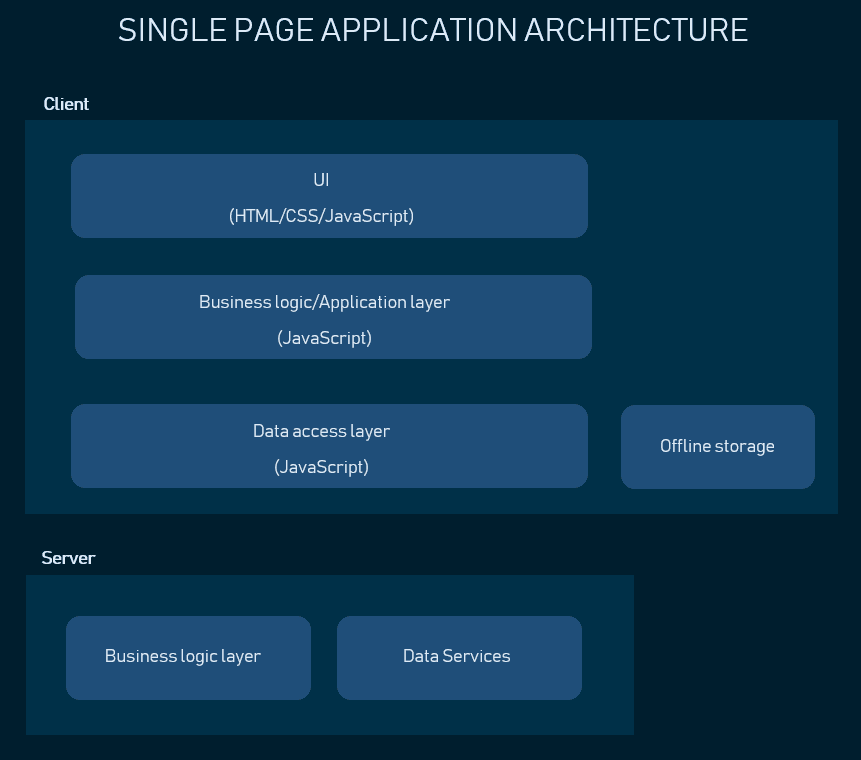
**Third-party integrations**. Payment gateways, social logins etc. are all integrations connected to the application’s back end via pieces of code called APIs. They allow your software to source data from other software and widen your functionality without coding it from scratch

SPAs, and MPAs

* There are two main design patterns for web apps: multi-page application (MPA) and single-page application (SPA).
* The application’s front end can serve either static or dynamic content. In most cases, it’s a combination of both. Static Web Pages exist on a server as they are and contain information that doesn’t change.
* Dynamic Web Pages change information every day or in response to a user’s request .The combination of dynamic and static content makes up a web application.
* The simplest example of a web application with dynamic content is a Single Page Application.

Single Page Applications

* A single-page application(SPA) is an app that works inside a browser and does not require page reloading during use. we are using this type of applications every day. These are, for instance: Gmail, Google Maps, Facebook or GitHub.
* The main purpose of SPAs is the ability to access all information from a single HTML page. Having moved the application logic to the client-side and using server-side only as data storage, developers can make the website run faster and ease the load off the server.
* The front end, aside from HTML and CSS, is written on a single framework, which dynamically generates content and transmits it to a user.
* Dependencies between components are tight. This means that making changes to one of the UI elements necessitates rewriting the whole front end code.
* Since SPAs move the logic to the client-side, they have to be written using client-side scripting.
* when a user requests content, a server simply transmits this data back to the browser, which renders it according to the templates. This significantly reduces the server load, as opposed to server-side scripting.
* SPA requests the markup and data independently and renders pages straight in the browser. Advanced JavaScript frameworks like AngularJS, Ember.js, Meteor.js, Knockout.js Single-page sites help keep the user in one, comfortable web space where content is presented to the user in a simple, easy and workable fashion.
* The core technology of client-side scripting is JavaScript. Along with its many frameworks(AngularJs), this language allows creation of both small and robust applications.



* When the role of the server is reduced to data services, this is sometimes called thin server architecture.

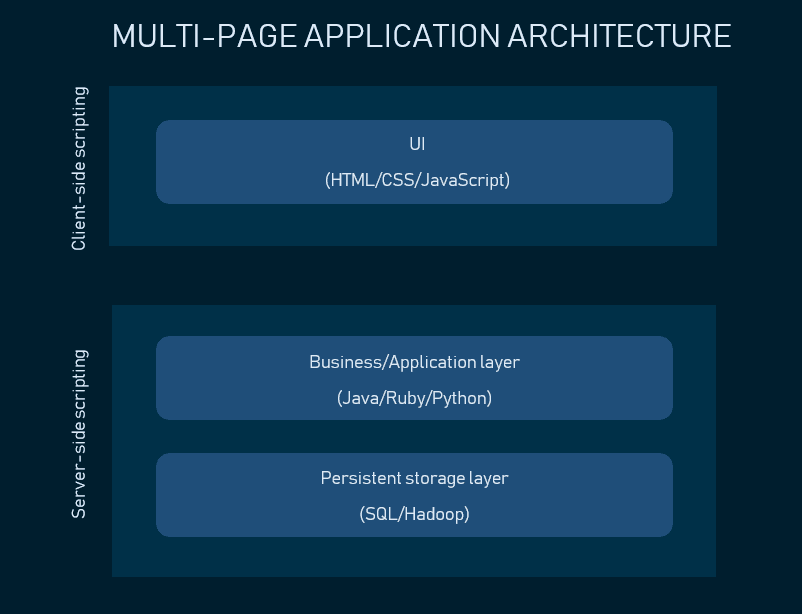
# Pros of the Single-Page Application:

* SPA is fast, as most resources (HTML+CSS+Scripts) are only loaded once throughout the lifespan of application. Only data is transmitted back and forth.
* The development is simplified and streamlined. There is no need to write code to render pages on the server. It is much easier to get started because you can usually kick off development from a file file://URI, without using any server at all.
* It’s easier to make a mobile application because the developer can reuse the same backend code for web application and native mobile application.
* SPA can cache any local storage effectively. An application sends only one request, store all data, then it can use this data and works even offline.

# Cons of the Single-Page Application:

* It is slow to download because heavy client frameworks are required to be loaded to the client.
* It requires JavaScript to be present and enabled. If any user disables JavaScript in his or her browser, it won’t be possible to present application and its actions in a correct way.
* Compared to the “traditional” application, SPA is less secure.It enables attackers to inject client-side scripts into web application by other users.
* Memory leak in JavaScript can even cause powerful system to slow down.

### Multi-Page Applications

* In Multi-Page Applications, some content requests need a whole new web page to be retrieved from the server.
* These are massive applications with multi-layered UI. AJAX technology solves the difficulties of complex applications transferring a huge amount of data between server and browser, refreshing only selective elements of the application. 
* Server-side scripting means that all operations are performed on the server’s end, so when you request content, it processes the script, retrieves data from the storage and chooses the content to display. Server-scripting languages you should be familiar with include PHP, Java, Python, Ruby, C#, and more.

# Pros of the Multiple-Page Application:

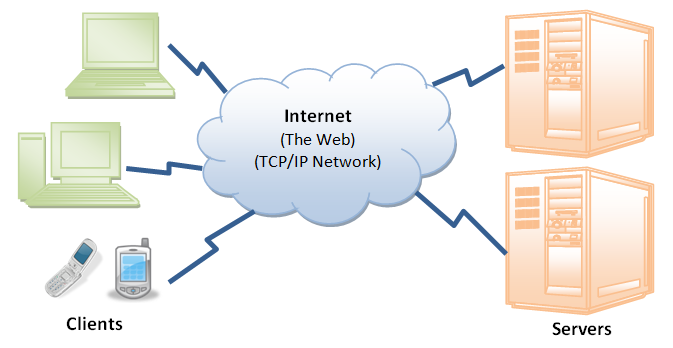
* It’s the perfect approach for users who need a visual map of where to go in the application. Solid, few level menu navigation is an essential part of traditional Multi-Page Application.
* Very good and easy for proper SEO management. It gives better chances to rank for different keywords since an application can be optimized for one keyword per page.

# Cons of the multiple-page application:

* There is no option to use the same backend with mobile applications.
* Frontend and backend development are tightly coupled.
* The development becomes quite complex. The developer needs to use frameworks for either client and server side. This results in the longer time of application development.

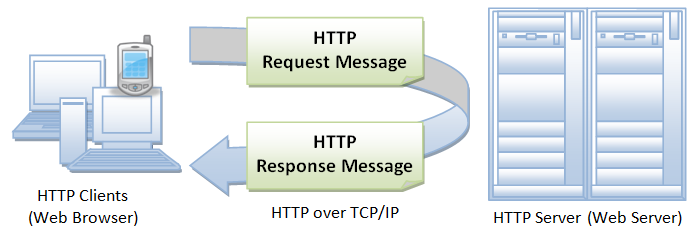
# HTTP (HyperText Transfer Protocol)

* Internet (or The Web) is a massive distributed client/server information system as depicted in the following diagram.



* Many applications are running concurrently over the Web, such as web browsing/surfing, e-mail, file transfer, audio & video streaming, and so on.
* In order for proper communication to take place between the client and the server, these applications must agree on a specific application-level protocol such as HTTP, FTP, SMTP, POP, and etc.

#### HyperText Transfer Protocol (HTTP)

* HTTP (Hyper Text Transfer Protocol) is the application layer protocol which runs over TCP/IP and used to transfer hyper texts or hyperlinks.
* 

* HTTP is a state less protocol that means each command executed independently without keeping relation with previous command, which makes server available for the request from different clients (browsers) as the server is not bind to any specific client.
* Invented by Tim Berners-Lee at CERN in the years 1989–1991, HTTP (Hypertext Transfer Protocol) is the underlying communication protocol of World Wide Web. HTTP functions as a request–response protocol in the client–server computing model.
* HTTP has four versions — HTTP/0.9, HTTP/1.0, HTTP/1.1, and HTTP/2.0. Today the version in common use is HTTP/1.1 and the future will be HTTP/2.0.

## HTTP/0.9 — The One-line Protocol

* Initial version of HTTP — a simple client-server, request-response, telenet-friendly protocol
* Request nature: single-line (method + path for requested document)
* Methods supported: GET only
* Response type: hypertext only
* Connection nature: terminated immediately after the response
* No HTTP headers (cannot transfer other content type files), No status/error codes, No URLs, No versioning.

**HTTP/1.0 — Building extensibility**

* Browser-friendly protocol
* Provided header fields including rich metadata about both request and response (HTTP version number, status code, content type)
* Response: not limited to hypertext (Content-Type header provided ability to transmit files other than plain HTML files — e.g. scripts, stylesheets, media)
* Methods supported: GET , HEAD , POST
* Connection nature: terminated immediately after the response

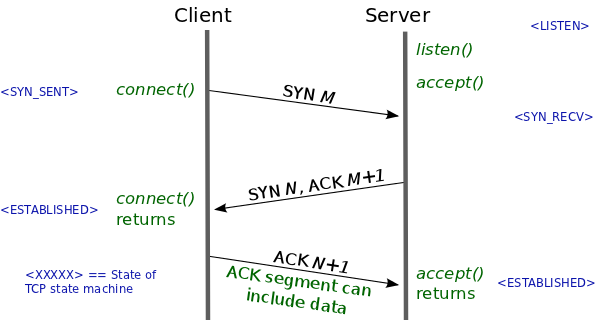
# HTTP/1.1 — The standardized protocol

* This is the HTTP version currently in common use.
* Introduced critical performance optimizations and feature enhancements — persistent and pipelined connections, chunked transfers, compression/decompression, content negotiations, virtual hosting (a server with a single IP Address hosting multiple domains), faster response and great bandwidth savings by adding cache support.
* Methods supported: GET , HEAD , POST , PUT , DELETE , TRACE , OPTIONS
* Connection nature: long-lived

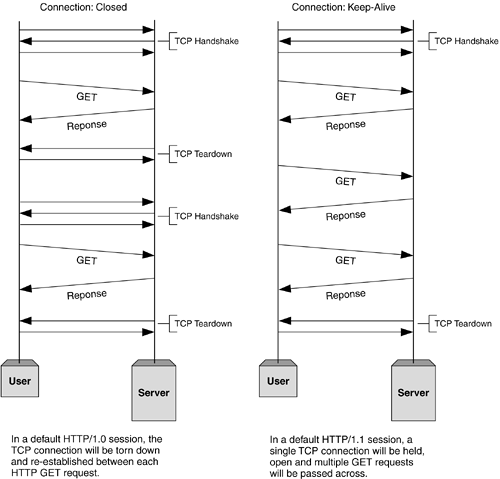
Persistent connections:

The first response that a client receives on an HTTP GET request is often not the fully rendered page. Instead, it contains links to additional resources needed by the requested page. The client discovers that the full rendering of the page requires these additional resources from the server only after it downloads the page. Because of this, the client will have to make additional requests to retrieve these resources. In HTTP/1.0, the client had to break and remake the TCP connection with every new request, a costly affair in terms of both time and resources.

HTTP/1.1 takes care of this problem by introducing persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each, greatly improving the performance of HTTP/1.1 over HTTP/1.0.



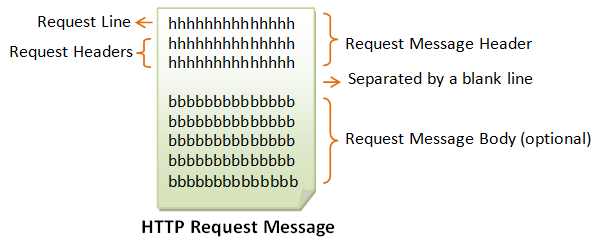
***A typical TCP Three-way Handshake***



#### Before establishing any connection, a TCP three-way handshake happens. At the end, after sending all data to Client, Server sends a message saying there’s no more data to send. Then the client closes the connection (TCP teardown). The problem in HTTP/1.0 is, for each request-response cycle, a connection needs to be opened and closed. And the advantage of using HTTP/1.1 is, we can reuse the same open connection for multiple request-response cycles. (Image from [informit.com](http://www.informit.com/articles/article.aspx?p=169578))

#### HTTP Request Message

The format of an HTTP request message is as follow:



##### Request Line

The first line of the header is called the request line, followed by optional request headers.

The request line has the following syntax:

**request-method-name request-URI HTTP-version**

* request-method-name: HTTP protocol defines a set of request methods, e.g., GET, POST, HEAD, and OPTIONS. The client can use one of these methods to send a request to the server.
* request-URI: specifies the resource requested.
* HTTP-version: Two versions are currently in use: HTTP/1.0 and HTTP/1.1.

**Examples** of request line are:

GET / HTTP/1. test.html 1

HEAD /query.html HTTP/1.0

POST /index.html HTTP/1.1

##### **Request Headers**

The request headers are in the form of name:value pairs. Multiple values, separated by commas, can be specified.

**request-header-name: request-header-value1, request-header-value2, ...**

**Examples** of request headers are:

Host: www.xyz.com

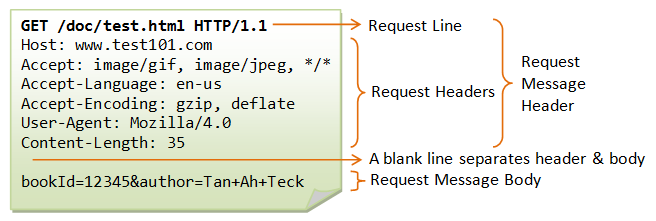
Connection: Keep-Alive

Accept: image/gif, image/jpeg, \*/\*

Accept-Language: us-en, fr, cn

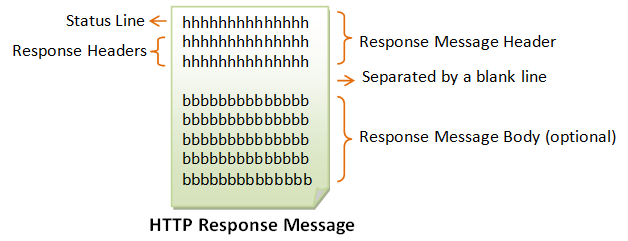
##### **Example**

The following shows a sample HTTP request message:



#### HTTP Response Message

The format of the HTTP response message is as follows:



##### **Status Line**

The first line is called the status line, followed by optional response header(s).

The status line has the following syntax:

**HTTP-version status-code reason-phrase**

* HTTP-version: The HTTP version used in this session. Either HTTP/1.0 and HTTP/1.1.
* status-code: a 3-digit number generated by the server to reflect the outcome of the request.
* reason-phrase: gives a short explanation to the status code.
* Common status code and reason phrase are "200 OK", "404 Not Found", "403 Forbidden", "500 Internal Server Error".

**Examples** of status line are:

HTTP/1.1 200 OK

HTTP/1.0 404 Not Found

HTTP/1.1 403 Forbidden

##### **Response Headers**

The response headers are in the form name:value pairs:

**response-header-name: response-header-value1, response-header-value2, ...**

**Examples** of response headers are:

Content-Type: text/html

Content-Length: 35

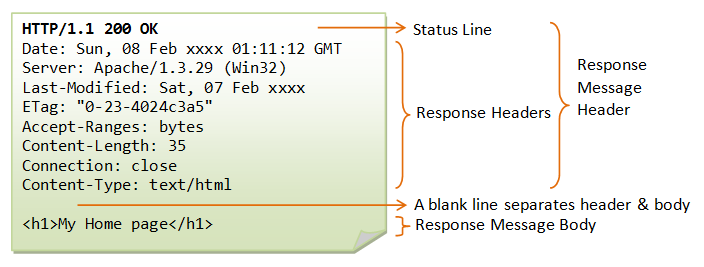
Connection: Keep-Alive

Keep-Alive: timeout=15, max=100

The response message body contains the resource data requested.

##### **Example**

The following shows a sample response message:



**HTTP Request Methods**

HTTP protocol defines a set of request methods. A client can use one of these request methods to send a request message to an HTTP server. The methods are:

* GET: A client can use the GET request to get a web resource from the server.
* HEAD: A client can use the HEAD request to get the header that a GET request would have obtained. Since the header contains the last-modified date of the data, this can be used to check against the local cache copy.
* POST: Used to post data up to the web server.
* PUT: Ask the server to store the data.
* DELETE: Ask the server to delete the data.
* TRACE: Ask the server to return a diagnostic trace of the actions it takes.
* OPTIONS: Ask the server to return the list of request methods it supports.

#### Response Status Code

The first line of the response message (i.e., the status line) contains the response status code, which is generated by the server to indicate the outcome of the request.

The status code is a 3-digit number:

* 1xx (Informational): Request received, server is continuing the process.
* 2xx (Success): The request was successfully received, understood, accepted and serviced.
* 3xx (Redirection): Further action must be taken in order to complete the request.
* 4xx (Client Error): The request contains bad syntax or cannot be understood.
* 5xx (Server Error): The server failed to fulfill an apparently valid request.

Some commonly encountered status codes are:

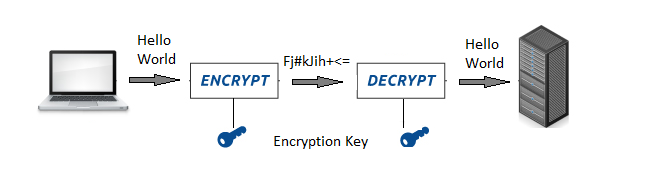
* 100 Continue: The server received the request and in the process of giving the response.
* 200 OK: The request is fulfilled.
* 301 Move Permanently: The resource requested for has been permanently moved to a new location. The URL of the new location is given in the response header called Location. The client should issue a new request to the new location. Application should update all references to this new location.
* 302 Found & Redirect (or Move Temporarily): Same as 301, but the new location is temporarily in nature. The client should issue a new request, but applications need not update the references.
* 304 Not Modified: In response to the If-Modified-Since conditional GET request, the server notifies that the resource requested has not been modified.
* 400 Bad Request: Server could not interpret or understand the request, probably syntax error in the request message.
* 401 Authentication Required: The requested resource is protected, and require client’s credential (username/password). The client should re-submit the request with his credential (username/password).
* 403 Forbidden: Server refuses to supply the resource, regardless of identity of client.
* 404 Not Found: The requested resource cannot be found in the server.
* 405 Method Not Allowed: The request method used, e.g., POST, PUT, DELETE, is a valid method. However, the server does not allow that method for the resource requested.
* 408 Request Timeout:
* 414 Request URI too Large:
* 500 Internal Server Error: Server is confused, often caused by an error in the server-side program responding to the request.
* 501 Method Not Implemented: The request method used is invalid (could be caused by a typing error, e.g., "GET" misspell as "Get").
* 502 Bad Gateway: Proxy or Gateway indicates that it receives a bad response from the upstream server.
* 503 Service Unavailable: Server cannot response due to overloading or maintenance. The client can try again later.
* 504 Gateway Timeout: Proxy or Gateway indicates that it receives a timeout from an upstream server.

**HTTP vs HTTPS**

Various encryption techniques were introduced like Symmetric and Asymmetric encryption to make HTTP a secure channel.

Symmetric Encryption

* This is a simple kind of encryption in which only one key is required to encrypt and decrypt the message. Secrete key can either be a number or string of random letters.
* The key and the message to encrypt goes through series of mathematical calculations to generate the ciphered message. The receiver uses the same secrete key and decipher that message to original.
* Both sender and receiver should have to exchange key first before sending any encrypted message. Sharing secret key over unsecured line (HTTP) before starting secure communication is still not a perfect secure solution, to overcome this disadvantage, asymmetric encryption was officially published in 1976 by [Whitfield Diffie](https://en.wikipedia.org/wiki/Whitfield_Diffie) and [Martin Hellman](https://en.wikipedia.org/wiki/Martin_Hellman).



# Asymmetric Encryption / Public-key Encryption

* Unlike symmetric encryption, asymmetric encryption uses two different keys to encrypt and decrypt the data in which one key is published publicly called public key and other key is kept private called private key.
* Data that is encrypted using public key can only be decrypted by private key and vice-versa. So public key is made freely available to anyone who might want to send data.
* The sender then encrypts the data with public key of receiver. This encrypted data can only be decrypted by the private key of receiver.



* Although asymmetric encryption makes communication channel secure but still not prevented from [Man-in-the-middle](https://en.wikipedia.org/wiki/Man-in-the-middle_attack) attack. Attacker, lying in between client and server, could generate a key pair and publish its public key (to client) pretending that the public key belongs to someone else (the actual server). The client believes it and encrypts it’s private data with the attackers public key, resulting in the attacker being able to read the client’s private data.
* To overcome this problem HTTPS over TLS or SSL is introduced. TLS (Transport Layer Security) is just an updated, more secure, version of SSL (Secure Sockets Layer). SSL/TLS provides identification to both the communicating parties, letting them know who they are communicating with.

**HTTPS**

* Hypertext Transfer Protocol Secure (HTTPS) is an extension of HTTP for secure communication over a computer network, and is widely used in the internet.
* Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP. It uses SSL/TLS for secure encrypted communications.
* Originally developed by Netscape in mid-1990s, SSL (Secure Socket Layer) is a cryptographic protocol enhancement to HTTP, which defines how client and server should communicate with each other securely. TLS (Transport Layer Security) is the successor of SSL.
* An HTTPS connection can protect the data transfer from the man-in-the-middle attacks and common security threats by providing bidirectional encryption for communications between a client and server.
* The handshake begins when a client connects to a TLS-enabled server requesting a secure connection and the client presents a list of supported cipher suites (ciphers and hash functions). From this list, the server picks a cipher and hash function that it also supports and notifies the client of the decision.
* The server usually then provides identification in the form of a [digital certificate](https://en.wikipedia.org/wiki/Public_key_certificate). The certificate contains the server name, the trusted [certificate authority](https://en.wikipedia.org/wiki/Certificate_authority) (CA), and the server’s public encryption key. The client confirms the validity of the certificate before proceeding.
* To generate the session keys used for the secure connection, the client either: encrypts a [random number](https://en.wikipedia.org/wiki/Random_number_generation) with the server’s public key and sends the result to the server (which only the server should be able to decrypt with its private key), both parties then use the random number to generate a unique session key for subsequent encryption and decryption of data during the session uses [Diffie–Hellman key exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange" \t "_blank) to securely generate a random and unique session key for encryption and decryption.
* Although HTTPS is secure by its design, the SSL/TLS handshake process consumes a significant time before establishing an HTTPS connection. It normally costs 1–2 seconds and drastically slows down the startup performance of a website.

### HTTP/2

HTTP/2 began as the SPDY protocol, developed primarily at Google with the intention of reducing web page load latency by using techniques such as compression, multiplexing, and prioritization. This protocol served as a template for HTTP/2 when the Hypertext Transfer Protocol working group httpbis of the [IETF (Internet Engineering Task Force)](https://www.ietf.org/) put the standard together, culminating in the publication of HTTP/2 in May 2015. From the beginning, many browsers supported this standardization effort, including Chrome, Opera, Internet Explorer, and Safari. Due in part to this browser support, there has been a significant adoption rate of the protocol since 2015, with especially high rates among new sites.

From a technical point of view, one of the **most significant features that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer,** which can be thought of as a part of the application layer in the internet protocol stack. As opposed to **HTTP/1.1, which keeps all requests and responses in plain text format, HTTP/2 uses the binary framing layer to encapsulate all messages in binary format**, while still maintaining HTTP semantics, such as verbs, methods, and headers. An application level API would still create messages in the conventional HTTP formats, but the underlying layer would then convert these messages into binary. This ensures that web applications created before HTTP/2 can continue functioning as normal when interacting with the new protocol.

The conversion of messages into binary allows HTTP/2 to try new approaches to data delivery not available in HTTP/1.1, a contrast that is at the root of the practical differences between the two protocols. The next section will take a look at the delivery model of HTTP/1.1, followed by what new models are made possible by HTTP/2.

## Delivery Models

As mentioned in the previous section, HTTP/1.1 and HTTP/2 share semantics, ensuring that the requests and responses traveling between the server and client in both protocols reach their destinations as traditionally formatted messages with headers and bodies, using familiar methods like GET and POST. But while HTTP/1.1 transfers these in plain-text messages, HTTP/2 encodes these into binary, allowing for significantly different delivery model possibilities. In this section, we will first briefly examine how HTTP/1.1 tries to optimize efficiency with its delivery model and the problems that come up from this, followed by the advantages of the binary framing layer of HTTP/2 and a description of how it prioritizes requests.

### HTTP/1.1 — Pipelining and Head-of-Line Blocking

The first response that a client receives on an HTTP GET request is often not the fully rendered page. Instead, it contains links to additional resources needed by the requested page. The client discovers that the full rendering of the page requires these additional resources from the server only after it downloads the page. Because of this, the client will have to make additional requests to retrieve these resources. In HTTP/1.0, the client had to break and remake the TCP connection with every new request, a costly affair in terms of both time and resources.

HTTP/1.1 takes care of this problem by introducing persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each, greatly improving the performance of HTTP/1.1 over HTTP/1.0.

Unfortunately, there is a natural bottleneck to this optimization strategy. Since multiple data packets cannot pass each other when traveling to the same destination, there are situations in which a request at the head of the queue that cannot retrieve its required resource will block all the requests behind it. This is known as *head-of-line (HOL) blocking*, and is a significant problem with optimizing connection efficiency in HTTP/1.1. Adding separate, parallel TCP connections could alleviate this issue, but there are limits to the number of concurrent TCP connections possible between a client and server, and each new connection requires significant resources.

These problems were at the forefront of the minds of HTTP/2 developers, who proposed to use the aforementioned binary framing layer to fix these issues, a topic you will learn more about in the next section.

### HTTP/2 — Advantages of the Binary Framing Layer

In HTTP/2, the binary framing layer encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer.

Let’s take a closer look at how this works. As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple *streams* of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called *frames*: