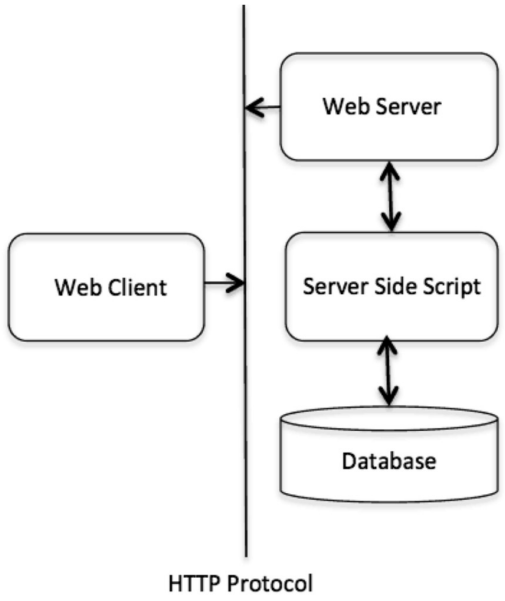
WORKING WITH REST APIS

TRAINOCATE

HTTP Overview

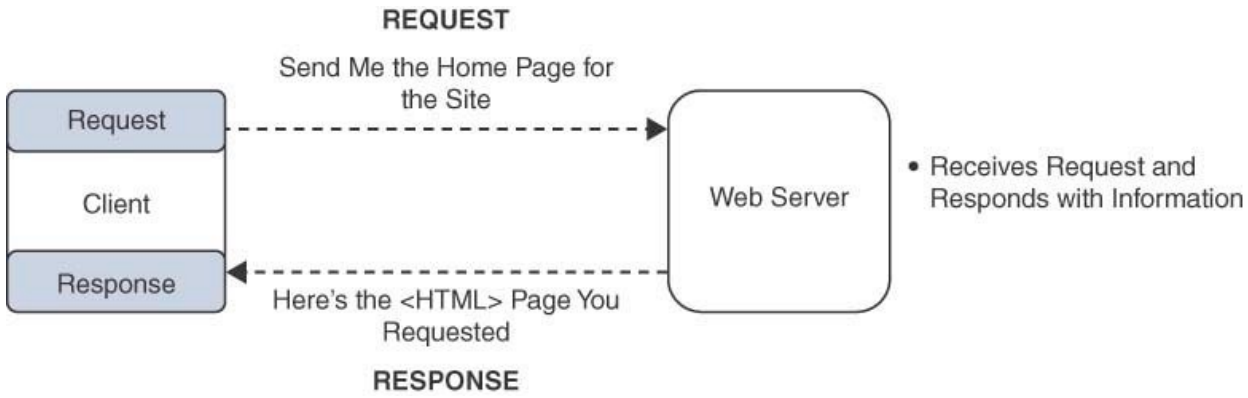
HTTP (Hypertext Transfer Protocol)

- The HTTP protocol is a request/response protocol based on the client/server-based architecture where web browsers, robots and search engines, etc. act like HTTP clients, and the Web server acts as a server.



Request-Response Cycle

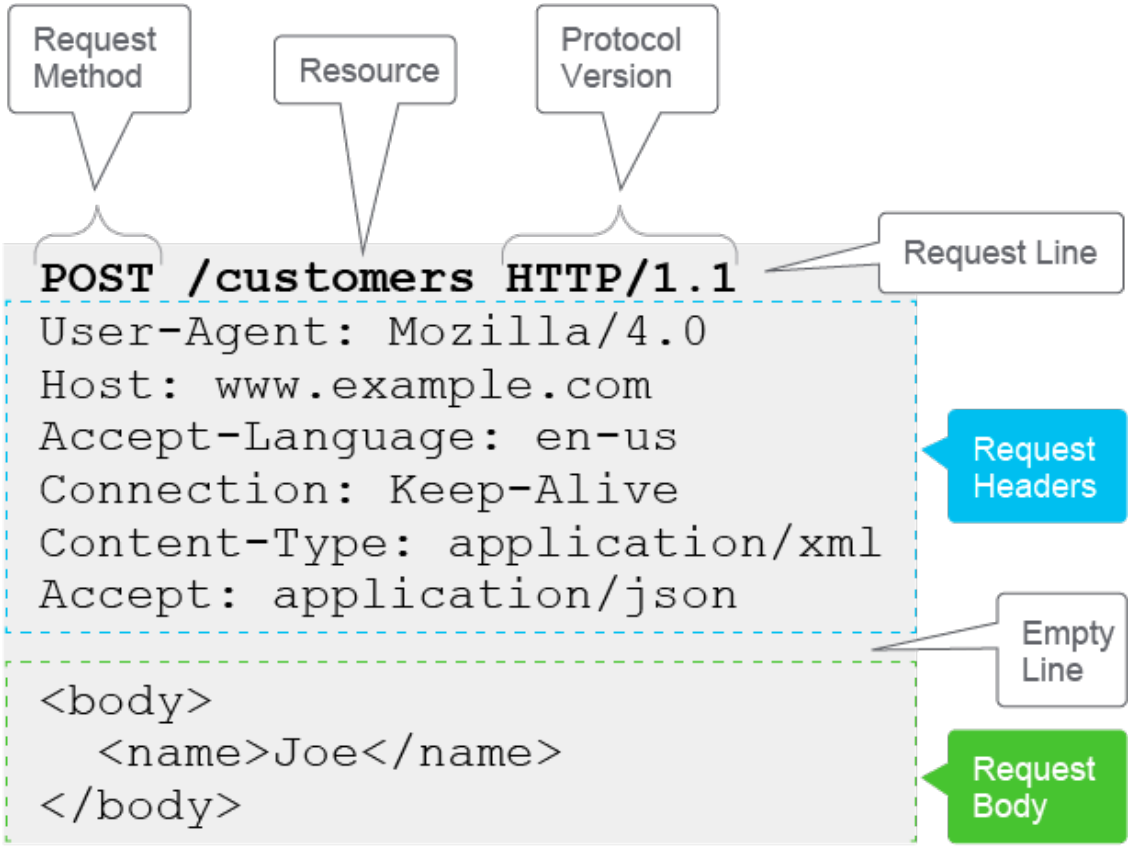
- The data is exchanged via HTTP requests and HTTP responses, which are specialized data formats used for HTTP communication. A sequence of requests and responses is called an HTTP session and is initiated by a client by establishing a connection to the server.
- Process of the request-response cycle:
  - Client sends an HTTP request to the web.
  - Server receives the request.
  - Server processes the request.
  - Server returns an HTTP response.
  - Client receives the response.



HTTP Request

- An HTTP request is the message sent by the client. The request consists of four parts:
  - Request-line, which specifies the method and location of accessing the resource. It consists of the request method (or HTTP verb), request Universal Resource Identifier (URI), and protocol version, in that order.
  - Zero or more HTTP headers. These contain additional information about the request target, authentication, taking care of content negotiation, and so on.
  - Empty line, indicating the end of the headers.
  - Message body, which contains the actual data transmitted in the transaction. It is optional and mostly used in HTTP POST requests.

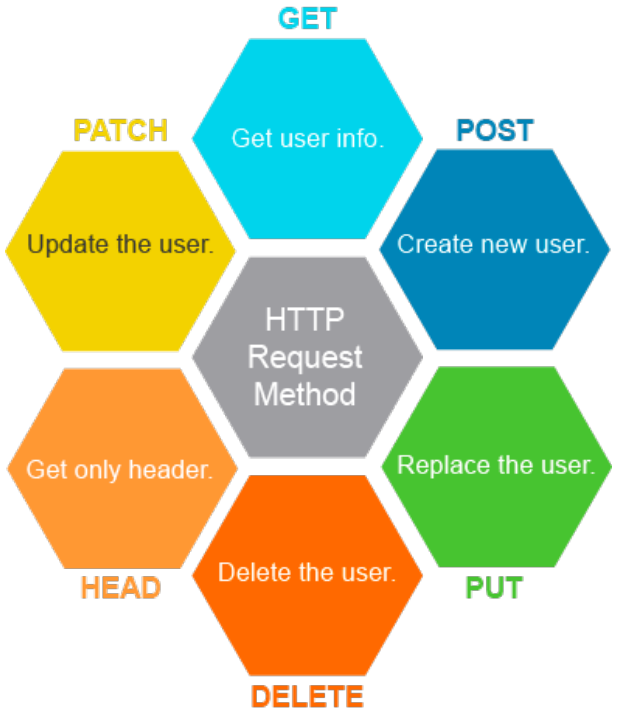
HTTP Request



GET	Request URI	HTTP Version
Request Header (Optional)		
Request Body (Optional)		

POST	Request URI	HTTP Version
Content Type		
Content Length		
Request Header (Optional)		
Request Body (Optional)		

HTTP Methods



HTTP Method	Function
GET	Requests a representation of a specific resource. Should only retrieve data and is considered safe and idempotent.
POST	Used to submit an entity to the specified resource, often causing a state change or side effects on the server. Requests made with POST should include a request body.
DELETE	Deletes the specified resource. Subsequent calls should not cause any side effects.
PUT	Replaces all current representations of the target resource with the request payload.
HEAD	Asks for a response identical to that of a GET request, but without the response body. Useful for validating resource availability.
PATCH	Applies partial modification to a resource. Useful for instances where using PUT might be too cumbersome. PATCH is not an idempotent method and is used for merging resources.

HTTP Headers

- The headers are a list of key-value pairs that the client and server use to pass additional information or metadata between them in requests. They consist of a case-insensitive name, followed by a colon (":") and then its value. There are dozens of different headers—some defined by the HTTP standard, and others defined by specific applications—so only the most common ones will be mentioned.
- General headers:
  - Headers from this category are not specific to any particular kind of message.
  - They are primarily used to communicate information about the message itself and how to process it.
  - **Cache-Control:** Specifies caching parameters.
  - **Connection:** Defines connection persistency.
  - **Date:** A datetime time stamp.
- Request headers:
  - These headers carry information about the resource to be fetched.
  - They also contain the information about the client.
  - **Accept-(*):** A subset of headers that define the preferred response format.
  - **Authorization:** Usually contains a Base64-encoded authentication string, composed of username and password for basic HTTP authentication.
  - **Cookie:** Contains a list of key-value pairs that contain additional information about the current session, user, browsing activity, or other stateful information.
  - **Host:** Used to specify the Internet host and port number of the resource being requested. This header is required in request messages.
  - **User-Agent:** Contains the information about the user agent originating the request.

HTTP Response

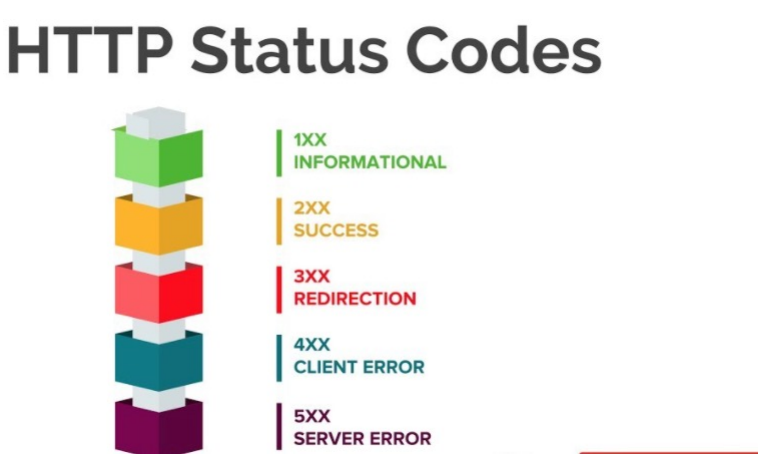
An HTTP response is the reply to the HTTP request and is sent by the server. The structure is similar to that of the request and consists of the following parts:

- Status-line, which consists of the protocol version, a response code (called HTTP Response Code), and a human-readable reason phrase that summarizes the meaning of the code.
- Zero or more HTTP headers. These contain additional information about the response data.
- Empty line, indicating the end of the headers.
- Message body, which contains the response data transmitted in the transaction.

HTTP Response

HTTP Status Codes

HTTP response status codes are a predefined set of numerical codes that indicate the status of a specific HTTP request in the response header. Status codes are separated into five classes (or categories) by functionality. You can create your own status codes, but it is strongly advised that you do not, because most user agents will not know how to handle them.



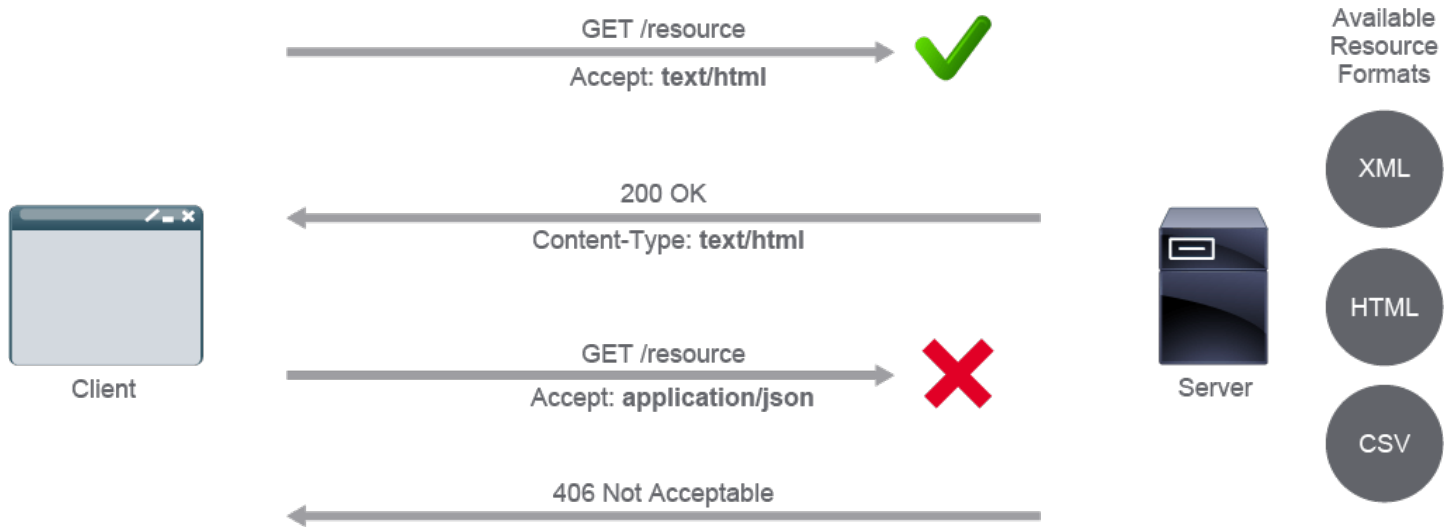
Return Codes	Meaning	Explanation
100	Continue	The server received the request and in the process of giving the response.
200	OK	The request is fulfilled.
301	Move Permanently	The resource requested for has been permanently moved to a new location. The URL of the new location is given in the response 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.

HTTP Headers

- Response headers:
  - These headers hold additional information about the response and the server providing it.
  - **Age:** Conveys the amount of time since the response was generated.
  - **Location:** Used to redirect the client to a location other than the request URI from a header.
  - **Server:** Contains the information about the software used by the origin server to handle the request.
  - **Set-Cookie:** Used to send cookies from the server to the client. It contains a list of key-value pairs, called *cookies*.
- Entity headers:
  - these headers contain information about the response body.
  - **Allow:** Lists the supported methods identified by the requested resource.
  - **Content-Type:** Indicates the media type of the body (also called Multipurpose Internet Mail Extensions [MIME] type), sent to the recipient. Used for content negotiation.
  - **Content-Language:** Describes the language of the intended audience for the enclosed body.
  - **Content-Length:** Indicates the size of the body.
  - **Content-Location:** Used to supply the resource location for the entity that is accessible from somewhere else than the request URI.
  - **Expires:** Gives the datetime after which the response is considered stale.
  - **Last-Modified:** Indicates the date and time at which the origin server believes the variant was last modified.

HTTP Content Negotiation

Basic HTTP Content Negotiation



HTTP Content Negotiation

HTTP headers that take care of content negotiation are:

- **Accept:** This header denotes the preferred media type (MIME type) for the response. A media type represents a general category and the subtype, which identifies the exact kind of data. A general type can be either discrete (representing a single resource) or multipart, where a resource is broken into pieces, often using several different media types (for example, multipart/form-data).
- Some useful discrete general types are:
  - **Application:** Any kind of binary data, that does not fall explicitly into other types. Data will be either executed or interpreted in a way that requires a specific application or category. Generic binary data has the type application/octet-stream, while more standardized formats include JSON (application/json) or XML (application/xml).
  - **Audio:** Audio or music data (for example, audio/mpeg).
  - **Image:** Image or graphical data, including both bitmap (image/bmp) and vector still images, and animated versions of still-image formats, such as animated GIF (image/gif).
  - **Text:** Text-only data, including any human readable content (text/plain), source code (text/javascript, text/html), or formatted data (text/csv).
  - **Video:** Video data or files (for example, video/mp4).
- **Accept-Charset:** Sets the preferred character sets, such as UTF-8 or ISO 8859-1. It is important when displaying resources in languages that include special characters.
- **Accept-Datetime:** Requests a previous version of the resource, denoted by the point in time with datetime. The value must always be older than the current datetime.
- **Accept-Encoding:** Sets the preferred encoding type for the content.
- **Accept-Language:** The preferred natural language. Useful for various localizations.

HTTP URL

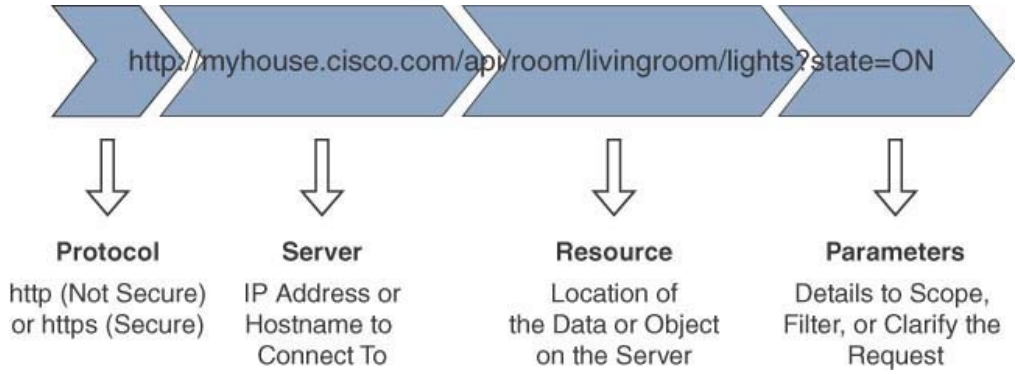
HTTP requests use an URL to identify and locate the resources targeted by the request. The "resource" term in the URL is very broadly defined, so it can represent almost anything—a simple web page, an image, a web service, or something else.

TRAINOCATE

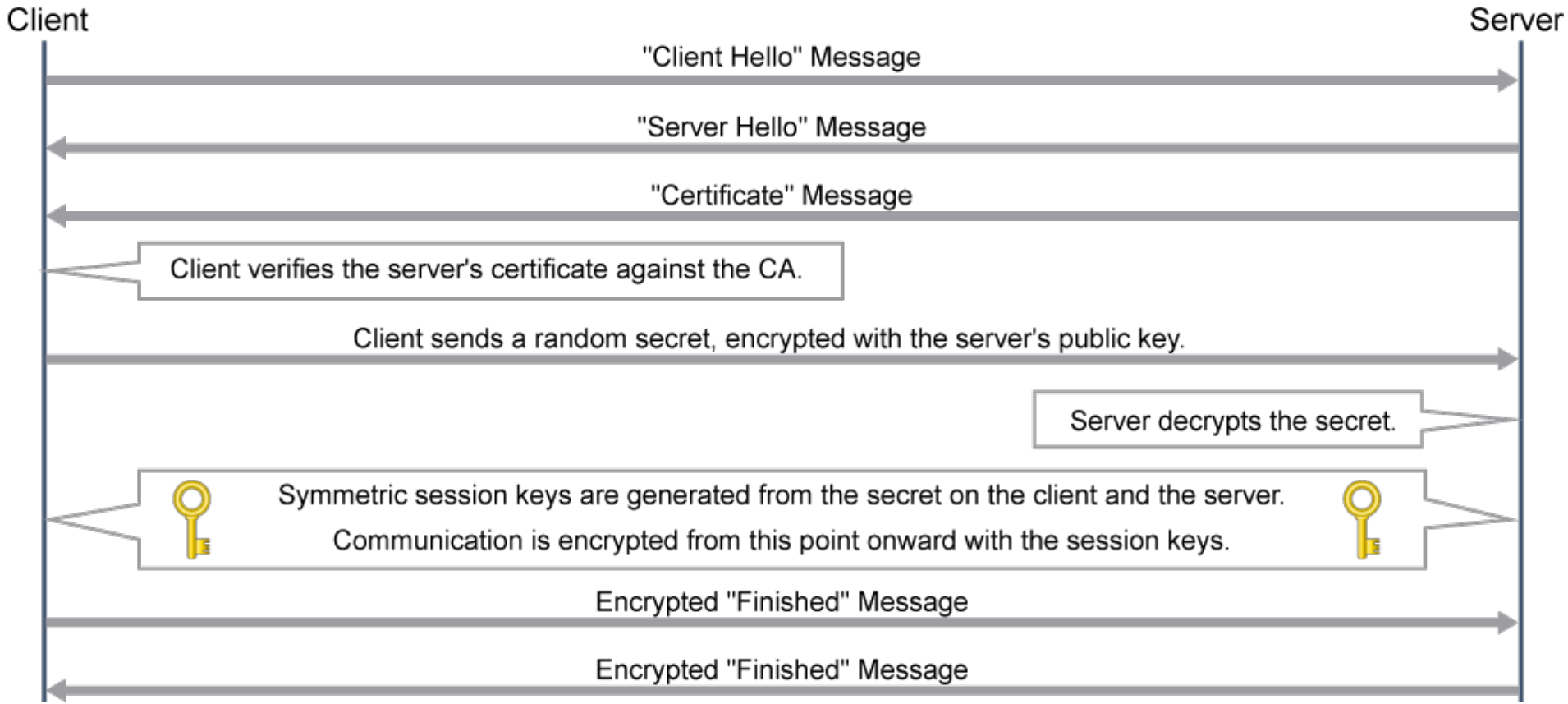
HTTP URL

URLs are composed from predefined URI components:

- **Scheme:** Each URL begins with a scheme name that refers to a specification for assigning identifiers within that scheme. Examples of popular schemes are http, https, mailto, ftp, data, and so on.
- **Host:** A URL host can be a fully qualified domain name (FQDN) or an IPv4 or IPv6 public address.
- **Port:** An optional parameter that specifies the connection port. If no port is set, the default port for the scheme is taken (default port is 80 for HTTP).
- **Resource path:** A sequence of hierarchical path segments, separated by a slash (/). It is always defined, although it may have zero length (for example, https://www.example.com/).
- **Query:** An optional parameter, preceded by the question mark (?) passed to the server that contains a query string of nonhierarchical data.
- **Fragment:** Also an optional parameter, the fragment starts with a hash (#) and provides directions to a secondary resource (for example, specific page in a document). It is processed by the client only.



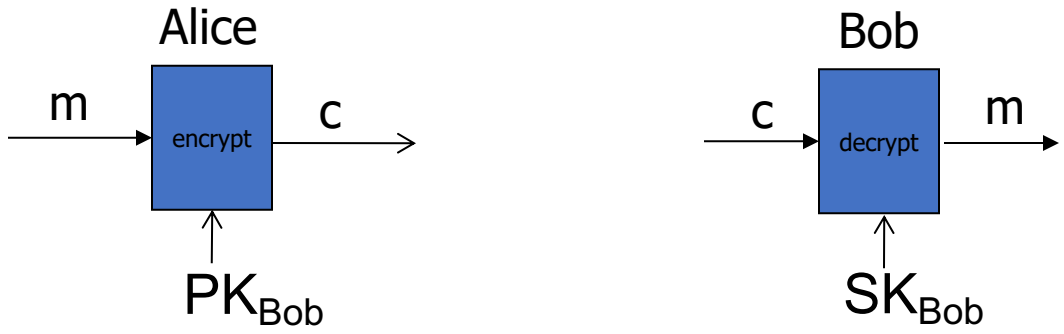
Leveraging HTTPS for Security



Leveraging HTTPS for Security

SSL/TLS overview

Public-key encryption:

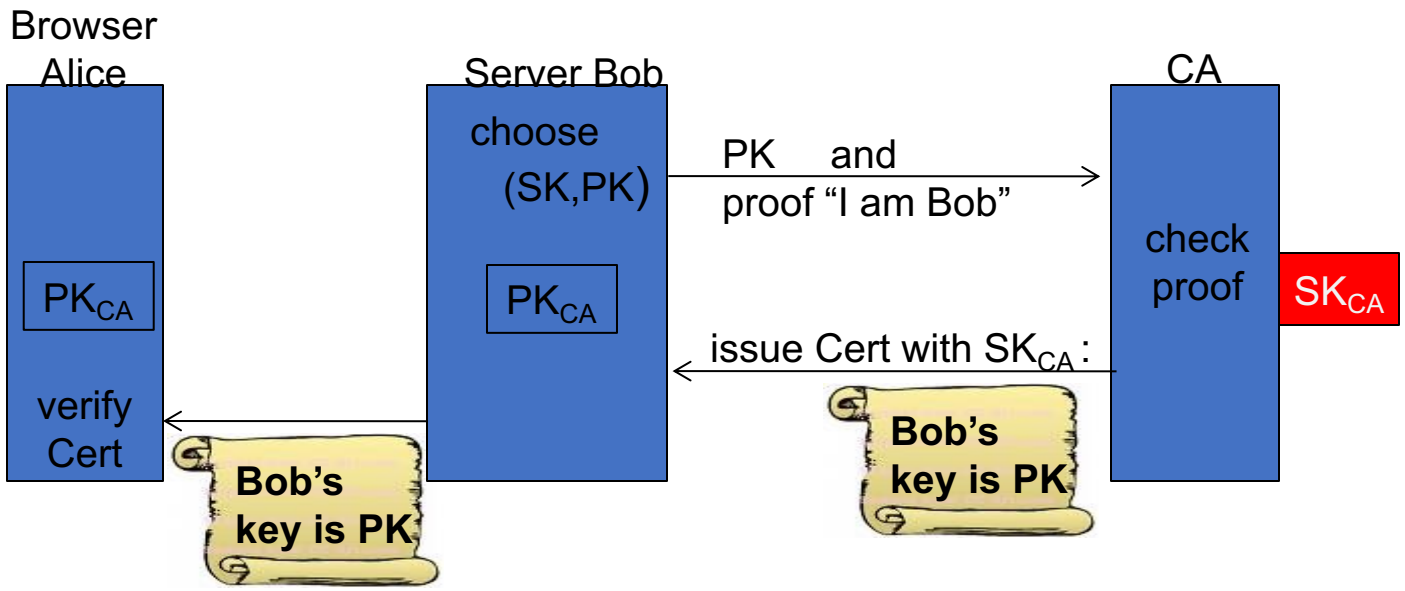


- Bob generates (SK_{Bob} , PK_{Bob})
- Alice: using PK_{Bob} encrypts messages and only Bob can decrypt

(SK: secure key, PK: public key)

Certificates

- How does Alice (browser) obtain PK_{Bob} ?

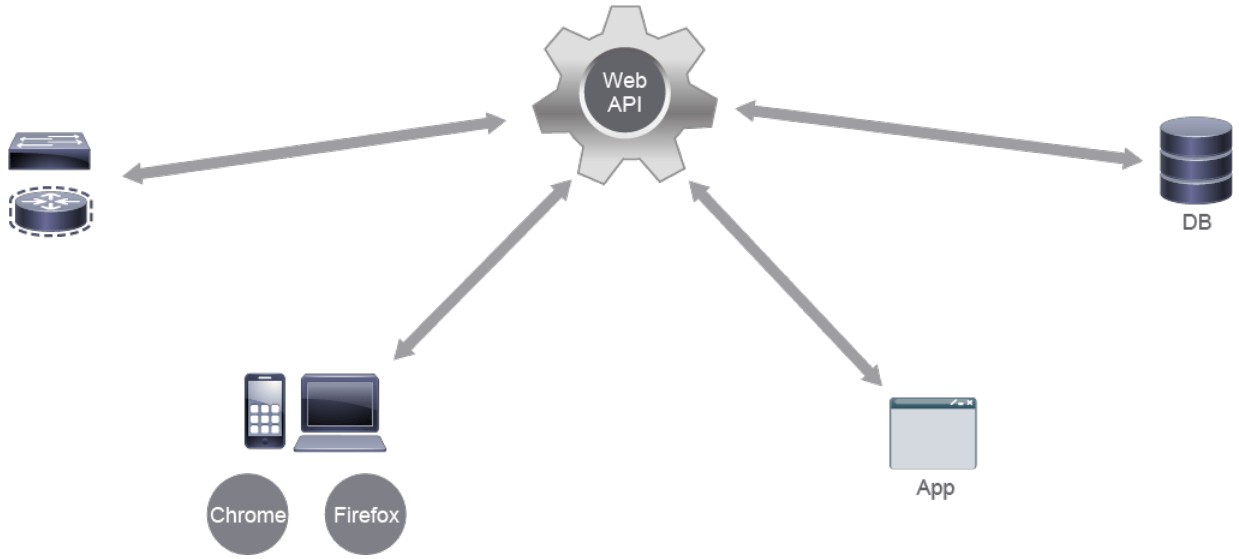


Bob uses Cert for an extended period (e.g. one year)

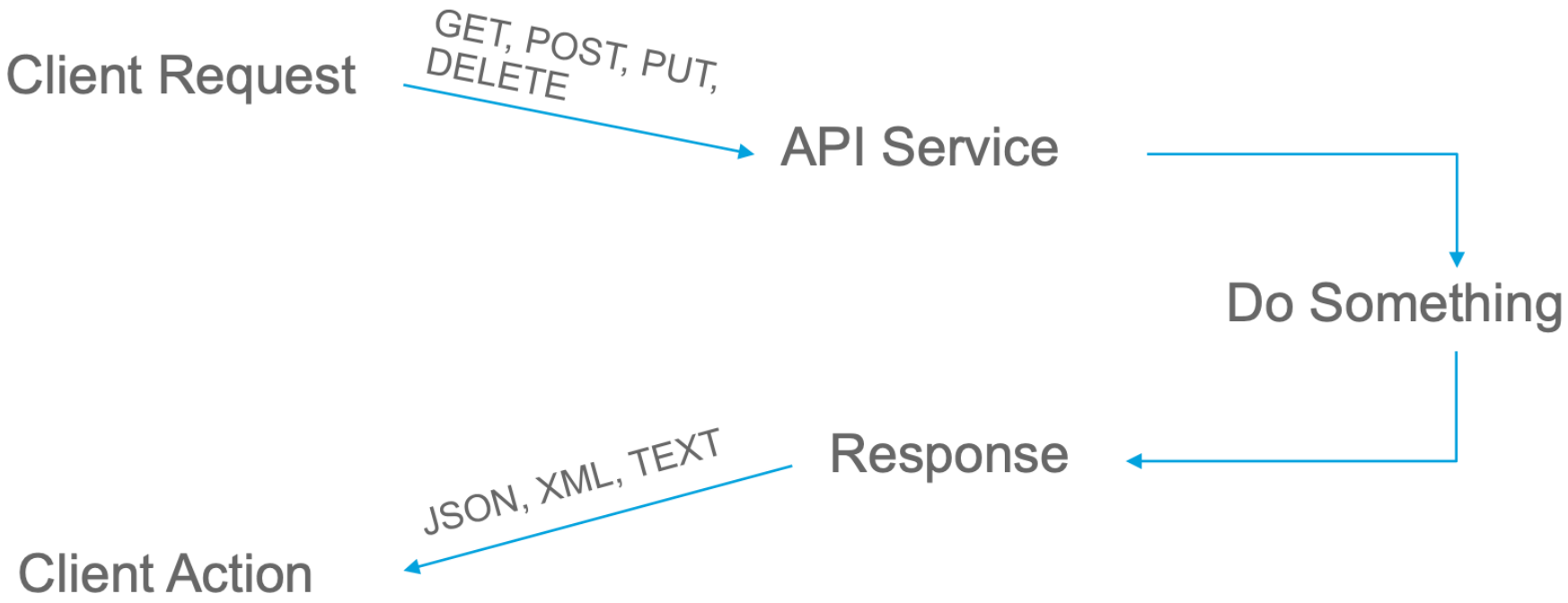
HTTP APPLIED TO WEB-BASED APIS

HTTP Applied to Web-Based APIs

- Web APIs are a subset of APIs, accessible over HTTP. Web APIs are software concepts that usually consist of one or more publicly exposed endpoints, their request and response structures, and abstraction of underlying layers.
- To communicate with these endpoints in an efficient and standardized way, HTTP request and response messages are used.



REST - How does it work



API AUTHENTICATION

API key authentication

- Can be accomplished in three ways
  - String

    GET /something?api_key=abcdef12345
  - Request header

    GET /something HTTP/1.1

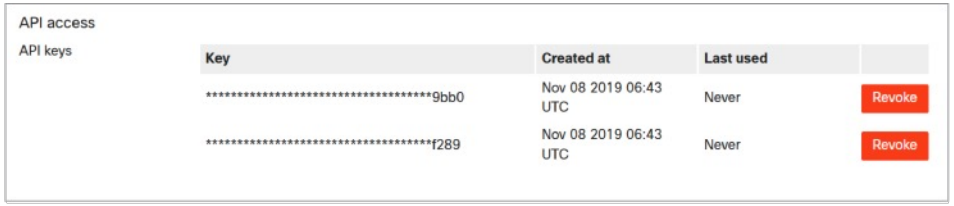
    X-API-Key: abcdef12345
  - Cookie

    GET /something HTTP/1.1

    Cookie: X-API-KEY=abcdef12345

API key authentication

- API key authentication uses a unique, pregenerated, cryptographically strong string as the authentication key, which is encoded in Base64 encoding schema, so it can be transported using HTTP.
- Clients encode the key either in the HTTP request headers as a cookie or as an individual header, or as a URL parameter. When the API request arrives at the server, the key is first parsed from the request, decoded from Base64, and compared to a database table, containing valid keys.
- These authentication keys are generated on the server on demand by the API administrator. A common practice is to generate the keys on a per-user or per-service basis. It means that you should avoid using a single key that has authorization for every possible API action and using it for everything. The scope of the keys should be limited to their intended use.
- API keys are usually only given to the users at the time of their creation for security reasons. If a user has to figure out which key does what later, they can still identify the keys by their prefixes; for example, only the first five characters of the key are displayed, which mostly eliminates duplicates but still guarantees uniqueness. If the key gets lost, a new one can easily be generated. This authentication method is not as safe as it could be. Because the authentication key is an immutable string that stays the same until it is deleted from the server, it can be intercepted by an attacker and used to create API calls in the name of the user for whom the key was generated.




```
GET /users?api_key=28fnhu783hb39bb0
```

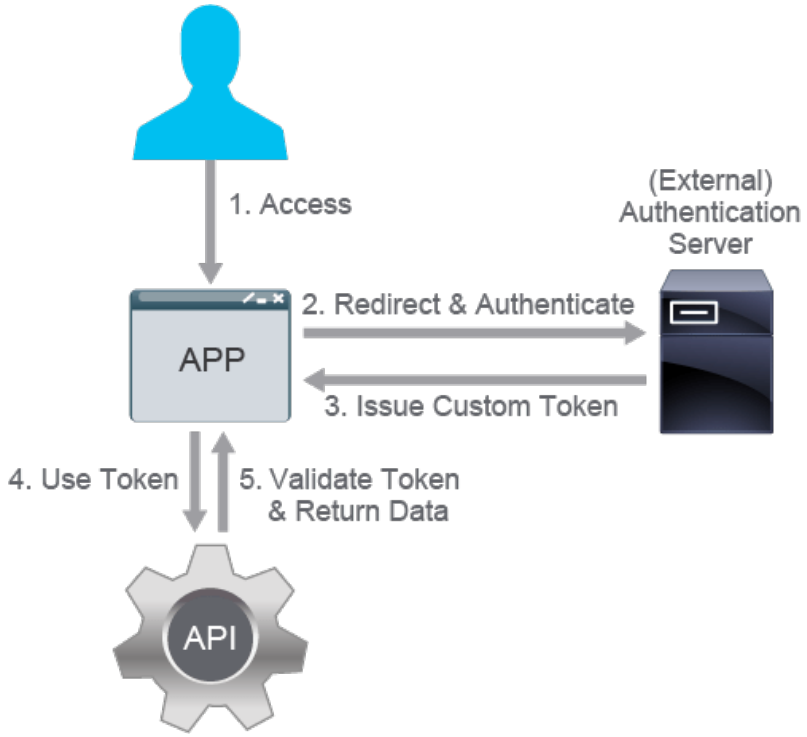
```
GET /users
Cookie: api_key=28fnhu783hb39bb0
```

API key authentication

- In case the API key is invalid or not valid anymore, a 401 Unauthorized HTTP error will notify you. If the key is valid but just not authorized for the resource you are trying to access, a 403 Forbidden error should appear. Because the implementation of error codes is mostly up to the developer, you might encounter more generic HTTP codes for such errors with some APIs (for example, just returning a code 400 or even 200 and then appending a message such as "Authentication unsuccessful" as the response body).
- The problem with both the basic HTTP authentication and API key authentication is that the authentication details have to be sent (and therefore processed and compared) with every API call. This action creates a weak point in security, because every request contains this data and is thus a viable attack surface

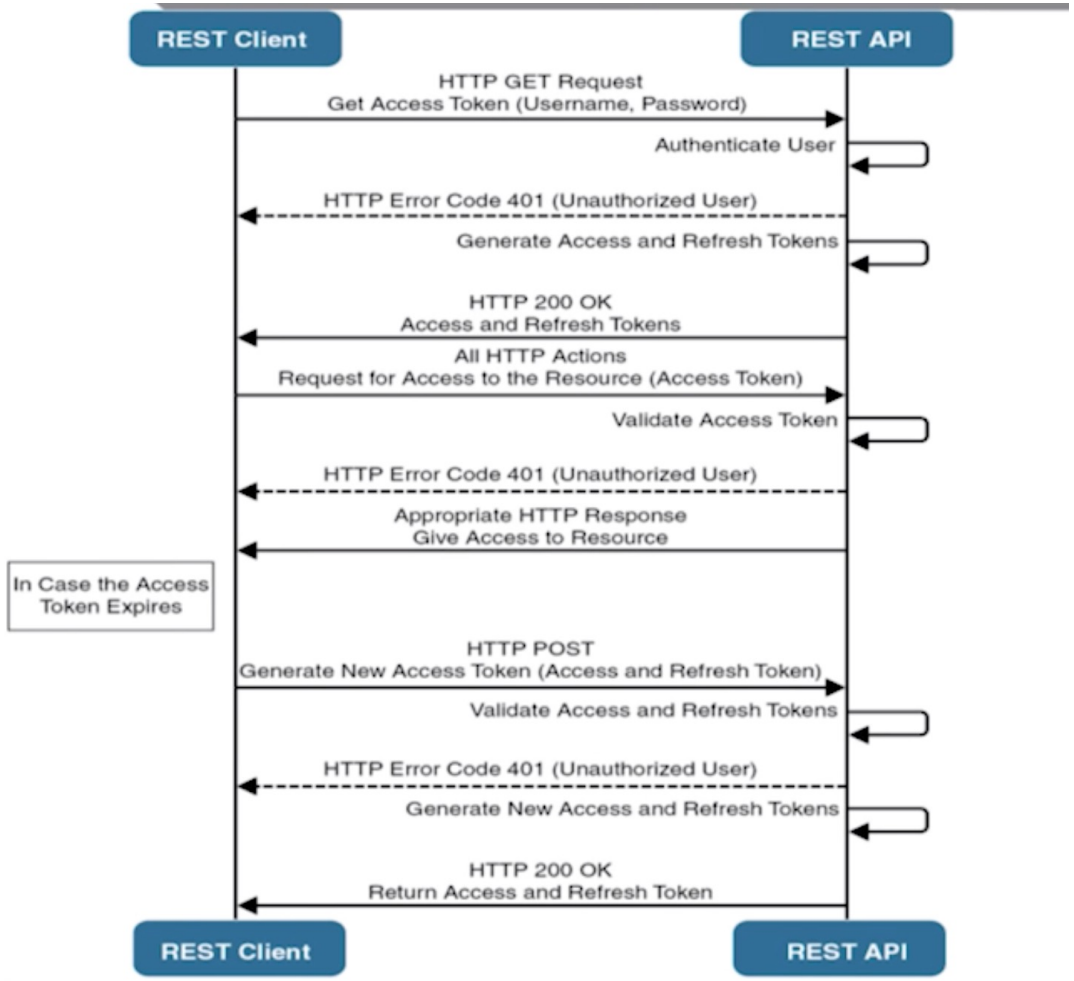
Custom token authentication

- The client tries to access the application.
- Because the client is not authenticated, it redirects it to an authentication server, where the user provides the username and password.
- The authentication server validates the credentials. If they are valid, a custom, time-limited, and signed authentication token is issued and returned to the user.
- The client request now contains the custom token, so the authenticated request is passed to the API service.
- The API service validates the token and serves the required data.



TRAINOCATE

Custom token authentication



TRAINOCATE

Custom token authentication

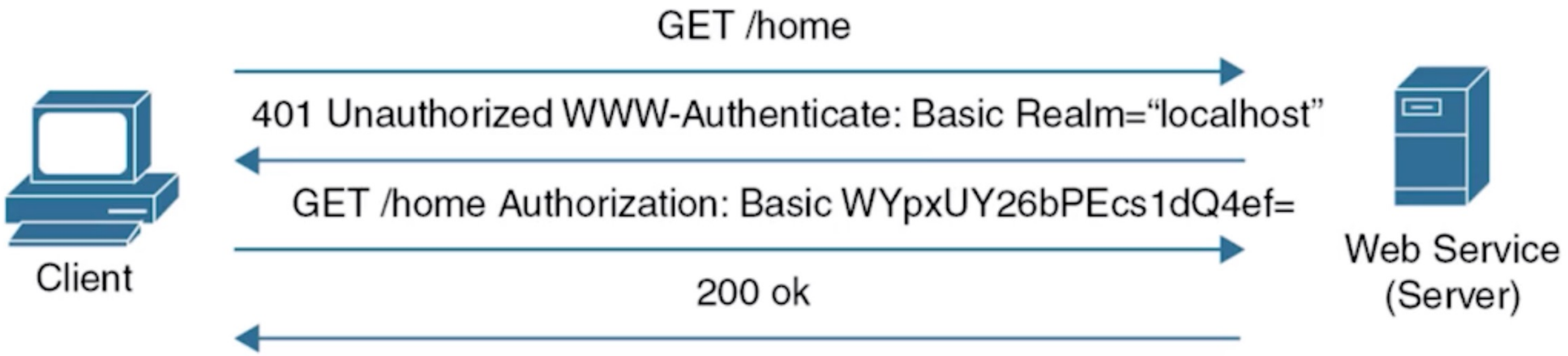
- The authentication server is often an external service (for example, OpenID) that is used in combination with an authorization mechanism (for example, OAuth 2.0). The tokens themselves often contain some authorization data as well. Authentication servers are commonly used to issue tokens that can be used with many different services from different vendors (for example, using Google authentication to log in to an unrelated site that has implemented Google sign-in).
- The term that is commonly used for this type of authentication (and authorization) is single sign-on (SSO). SSO is gaining in popularity, because it is very useful for services that require third-party authorization (for example, sharing an article from a news site on a social media account) or for businesses that offer several different services but want users to use the same identity for all of them (for example, Google Apps, your company enterprise environment, and so on).

Custom token authentication

The pros of custom token authentication include:

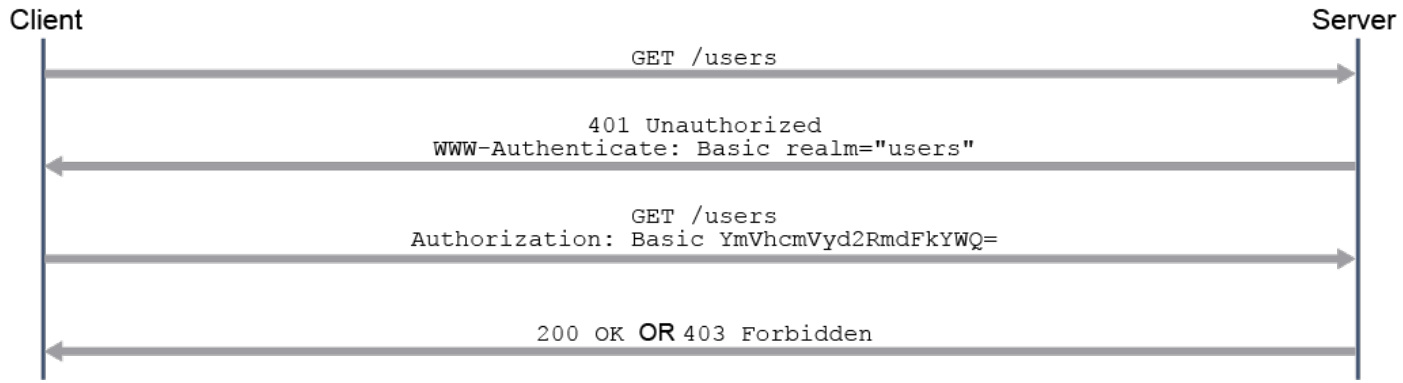
- **Faster response times:** Only the token (and maybe its session) has to be validated instead of the credentials. Sessions are often kept in very fast in-memory databases (for example, Redis), while credentials are kept in nonvolatile memory relational databases.
- **Simplicity of cross-service use:** This reduces the number of the authentication systems needed in an organization and unifies authentication and authorization policies across the services.
- **Increased security:** The credentials are not passed with every request, but rather only once in the beginning and periodically after that when the token expires. This way, it reduces the chance of success and viable time frame for any man-in-the-middle or cross-site attacks.

Basic authentication



Using HTTP Authentication

- When an unauthenticated (and consequently unauthorized) client makes a request toward the HTTP server, the server in turn challenges the received HTTP request. The challenge is done by using an error code (401 Unauthorized) and the *WWW-Authenticate* HTTP header in the response. This header defines which authentication method should be used to gain access to the resource specified in the request, because several methods exist.
- The client must then send another request, this time using the *Authorization* header, which contains the authentication type and the authentication credentials. The realm is a string that denotes some protected space on the server to which only authorized users have access; think of the realm as a group of resources, a specific server, and so on.
- If the credentials within the request Authorization header are valid, the server responds with a 200 OK HTTP response (or 403 Forbidden if the opposite occurs).



Using HTTP Authentication

- HTTP authentication uses several different authentication schemes:
  - Anonymous (no authentication)
  - Basic (Base64-encoded credentials as username:password)
  - Bearer (HTTP implementation of custom token authentication)
  - Digest (MD5-hashed credentials)
  - Mutual (two-way authentication)
  - More uncommon schemes (HMAC, HOBA, AWS, OAuth, Negotiate, and so on)
- It is important to note that no HTTP authentication schema is secure by itself. At the very least, Transport Layer Security (TLS) should be used to encrypt the connection (forming an HTTPS connection).
- Some schemas are inherently less secure than others. For example, the Basic schema uses Base64 to encode credentials, but because Base64 is just an encoding, not an encryption; the username and password can easily be decoded. Hashed credentials are more secure because the credentials are never sent in a clear or encoded form but follow a hashing algorithm, which can be a big obstacle for any potential attackers.

Handling Secrets for API Consumption

- API secrets, or in general, any kind of credentials and authentication data, is one of the most sensitive pieces of information, because it often grants you partial or full access to a system or an API. Yet, they are sometimes found written on sticky notes, in credentials.txt files, or as an admin:admin combination. These are, however, beginner mistakes when it comes to credential management, and as a software developer, you need to be aware of best practices regarding advanced credentials handling.
- One of the most basic rules is *avoid hardcoding*. Hardcoding is the practice of embedding variable values directly into the source code. While it is tempting—and sometimes useful or even mandatory—to store the credentials directly where they are needed, it should be avoided and is considered a very bad idea for multiple reasons:

Handling Secrets for API Consumption

Hardcoding

```
def api_call():

    api_key = 'c2H4svKvZln099HuaReb'
    url = 'api.mycompany.example/api/users'

    cookies = {'Cookies': 'api_key={}'.format(api_key)}
    body = {'limit': 10}

    response = requests.get(url, cookies=cookies, data=body)
    if response.status_code != 200:
        raise Exception('API call error: {}'.format(response))
    else:
        return response.data
```

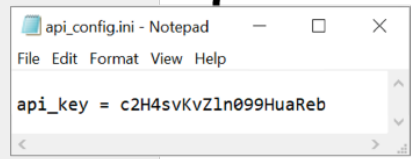
softcoding

```
import configparser

def api_call(context, result_limit=10):
    config = configparser.read('api_config.ini')
    api_key = config['api_key']
    url = context.url

    cookies = {'cookies': 'api_key={}'.format(api_key)}
    body = {'limit': result_limit}

    response = requests.get(url, cookies=cookies, data=body)
    if response.status_code != 200:
        raise Exception('API call error: {}'.format(response))
    else:
        return response.data
```



TRAINOCATE

API structure example (Webex rooms API)

PATH	METHOD	PURPOSE
/rooms	GET	List all rooms
/rooms	POST	Create a new room
/rooms/id	GET	Get Room Details
/rooms/id	DELETE	Delete a room
/rooms/id	UPDATE/PATCH	update a room


TRAINOCATE