# Quiz

**API Monitoring**

# API Security

## Security Mechanisms

## Security Mechanisms

Threats can be countered by applying security mechanisms that ensure that particular security goals are met

In this section we will run through the most common security mechanisms that you will generally find in every well-designed API:

**Encryption** **Authentication** **Access Control** **Audit Logging**

**Rate Limiting**

In almost all scenarios, your API should have all the 5 implemented (except for access control)

## Security Mechanisms

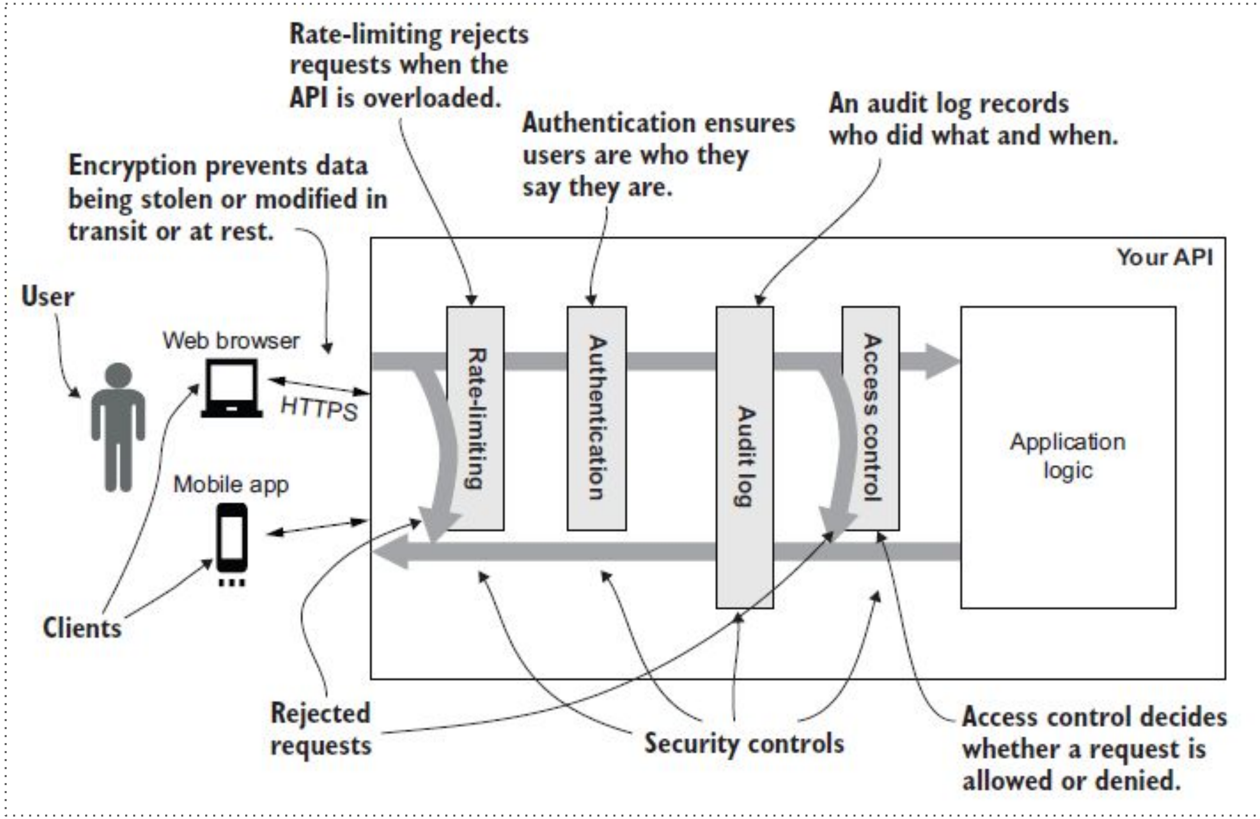
When processing a request, a secure API will apply some standard steps

Requests and responses are encrypted using the HTTPS protocol

Rate-limiting is applied to prevent DoS attacks

Then users and clients are identified and authenticated, and a record is made of the access attempt in an access or audit log

Finally, checks are made to decide if this user should be able to perform this request


Rate-limiting rejects requests when the API is overloaded.
Authentication ensures users are who they say they are.
An audit log records who did what and when.
Encryption prevents data being stolen or modified in transit or at rest.
Your API
User
Web browser
HTTPS
Mobile app
Clients
Rate-limiting
Authentication
Audit log
Access control
Application logic
Rejected requests
Security controls
Access control decides whether a request is allowed or denied.

## High Level Description

**Of what part of the problem do we aim to solve with Logging, Rate Limiting, Encryption, Hashing, etc**

## Security Mechanisms: Encryption (Protect Data In-Transit)

Encryption ensures that data can’t be read by unauthorized parties, either when it is being transmitted from the API to a client or at rest in a database or filesystem

Encryption is used to protect data when it is outside your API. There are two main cases in which data may be at risk:

- Requests and responses to an API may be at risk as they travel over networks, such as the internet. Encrypting data in transit is used to protect against these threats
- Data may be at risk from people with access to the disk storage that is used for persistence. Encrypting data at rest is used to protect against these threats

TLS should be used to encrypt data in transit and is covered in next sections.
Encrypting data at rest is a complex topic with many aspects to consider and is largely beyond the scope of this training but we will look at it at a high level.

## Security Mechanisms: Authentication (Who is trying to access)

Authentication is the process of verifying whether a user is who they say they are

Why do we need to authenticate the users of an API in the first place?

- You want to record which users performed what actions to ensure accountability
- You may need to know who a user is to decide what they can do (authorization)
- You may want to only process authenticated requests to avoid anonymous attacks

Authentication is usually achieved by asking the user to present some kind of credentials that prove that the claims are correct such as providing a password along with the username that only that user would know

## Security Mechanisms: Authentication Factors

There are many ways of authenticating a user, which can be divided into three broad categories known as authentication factors

**Something you know,** such as a secret password

**Something you have**, like a key or physical device

**Something you are**. This refers to biometric factors, such as your unique fingerprint or iris pattern

Any individual factor of authentication may be compromised. People choose weak passwords or write them down on notes attached to their computer screen, and they mislay physical devices and although biometric factors can be appealing, they often have high error rates

## Security Mechanisms: Authentication Factors

For this reason, the most secure authentication systems require two or more different factors. For example, your bank may require you to enter a password and then use a device with your bank card to generate a unique login code. This is known as two-factor authentication (2FA) or multi-factor authentication (MFA)

Two-factor authentication (2FA) or multi-factor authentication (MFA) require a user to authenticate with two or more different factors so that a compromise of any one factor is not enough to grant access to a system

Authenticating with two different passwords would still be considered a single factor, because they are both based on something you know

On the other hand, authenticating with a password and a time-based code generated by an app on your phone counts as 2FA because the app on your phone is something you have and password is something you know

## Security Mechanisms: Authorization (Who Can Do What)

In order to preserve confidentiality and integrity of your assets, it is usually necessary to control who has access to what and what actions they are allowed to perform

There are two primary approaches to access control that are used for APIs

***Identity-based access*** control first identifies the user and then determines what they can do based on who they are. A user can try to access any resource but may be denied access based on access control rules

***Capability-based access*** control uses special tokens or keys known as capabilities to access an API. The capability itself says what operations the bearer can perform rather than who the user is.
In a capability based system, permissions are based on keys that user generate/receive. A user or an application can only read a file, etc if they hold a capability that allows them to read that specific file.

## Security Mechanisms: Audit Logging (Who is doing what)

An audit log is a record of every operation performed using your API

The purpose of an audit log is to ensure accountability. It can be used after a security breach as part of a forensic investigation to find out what went wrong

Logs are also analyzed in real-time by log analysis tools to identity attacks in progress and other suspicious behavior

A good audit log can be used to answer the following kinds of questions

Who performed the action and what client did they use?

When was the request received?

What resource was being accessed?

Was the request successful? If not, why?

What kind of request was it, C, R, U or D?

## Security Mechanisms: Rate Limiting (Who is doing how much)

Rate limiting talks about preserving availability in the face of malicious or accidental DoS attacks

A DoS attack works by exhausting some finite resource that your API requires to service legitimate requests like CPU, memory, power, disk space, etc

By flooding your API with bogus requests, these resources become tied up servicing those requests and not others

The key to fending off these attacks is to recognize that a client (or group of clients) is using more than their fair share of some resource: time, memory, number of connections, and so on

Once a user has authenticated, your application can enforce quotas that restrict what they are allowed to do

For example, you might restrict each user to a certain number of API requests per hour, preventing them from flooding the system with too many requests

## Rate Limiting: Throttling

Once a predefined limit is reached then the system rejects new requests until the rate falls back under the limit

A rate-limiter can either completely close connections when the limit is exceeded or else slow down the processing of requests, a process known as throttling

**Throttling** is a process by which a client's requests are slowed down without disconnecting the client completely. Throttling can be achieved either by queueing requests for later processing, or else by responding to the requests with a status code telling the client to slow down. If the client doesn't slow down, then subsequent requests are rejected

## Security Considerations via Validations

Everytime you create/fetch/change any resources via your API you should do few validations

**REQUEST**

- Schema (keys of JSON)
- Data Types
- Data Size (length of user names, passwords, etc)
- A check on special characters, allowed, un-allowed characters

**RESPONSE**

- Never attach error logs of code
- Always respond in right format (json, xml)

# API Security

## Security Mechanism: Deep Dive and Hands Ons

## Securing our API: Introduction

In the last section we learned how to develop the functionality of your API while avoiding common security flaws

In this section we will go beyond basic functionality and see how proactive security mechanisms can be added to your API to ensure all requests are from genuine users and properly authorized

We will protect the our  API applying effective ***password authentication*** and preventing ***denial of service attacks***

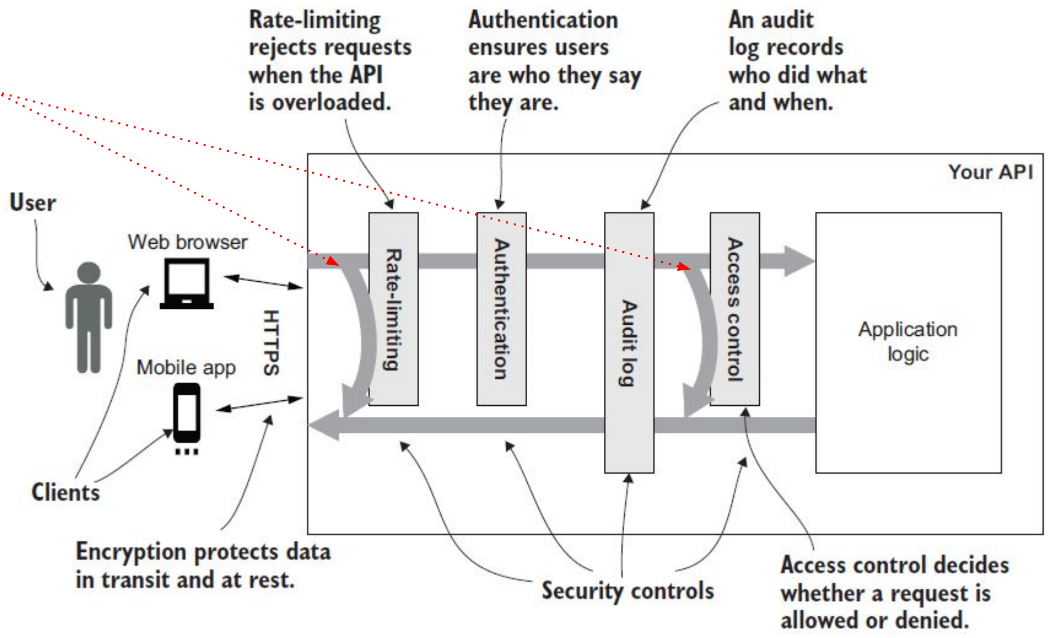
## Addressing threats with security controls

We'll protect our API against common threats by applying some basic security mechanisms (also known as security controls)

An important detail, shown in figure, is that only rate-limiting and access control directly reject requests

A failure in authentication does not immediately cause a request to fail, but a later access control decision may reject a request if it is not authenticated

This is important because we want to ensure that even failed requests are logged, which they would not be if the authentication process immediately rejected unauthenticated requests



Rate-limiting rejects requests when the API is overloaded.
Authentication ensures users are who they say they are.
An audit log records who did what and when.
Your API
User
Web browser
Mobile app
HTTPS
Rate-limiting
Authentication
Audit log
Access control
Application logic
Clients
Encryption protects data in transit and at rest.
Security controls
Access control decides whether a request is allowed or denied.

## Rate Limiting Steps, Actions and Hands On

There are lot of libraries available to rate-limit your API like guava in Java, flask-rate-limiter in Python.
We don't need to implement our own rate limiter

When a client breaches their limit you can then either block and wait until the rate reduces, or you can simply reject the request

The standard **HTTP 429 Too Many Requests** status code can be used to indicate that rate-limiting has been applied and that the client should try the request again later

You can also send a Retry-After header to indicate how many seconds the client should wait before trying again

The rate limiter should be the very first filter defined in your API, because even ***authentication and audit logging services have to be protected against DoS attacks***

## Hands On

**Rate Limiting**
**Our Payments API is insecure and vulnerable against DoS attacks.**
**Anyone can write a script to make 1 million requests and bring it down.**
**Let's protect it with rate-limiting**

| Simple Rate Limiting | Rate Limiting with Redis | Rate Limiting Per Route | Rate Limiting by username |
|---|---|---|---|
| Easiest to implement. Loses validity on application restart | Withstands app restarts. Keys gets created on IP address TTL per IP per endpoint | Different routes can have different rate-limiting | Rate limit on username provided in Header. If not provided - fall back on IP address |

Never try a DoS attack of any public URL. In most countries a DoS attacks are illegal. For example in USA: according to the Federal Computer Fraud and Abuse Act, an unauthorized DoS attack can lead to up to 10 years in prison and a $500,000 fine.

## Authentication

Basic Auth

Stateful Token

Stateless Tokens

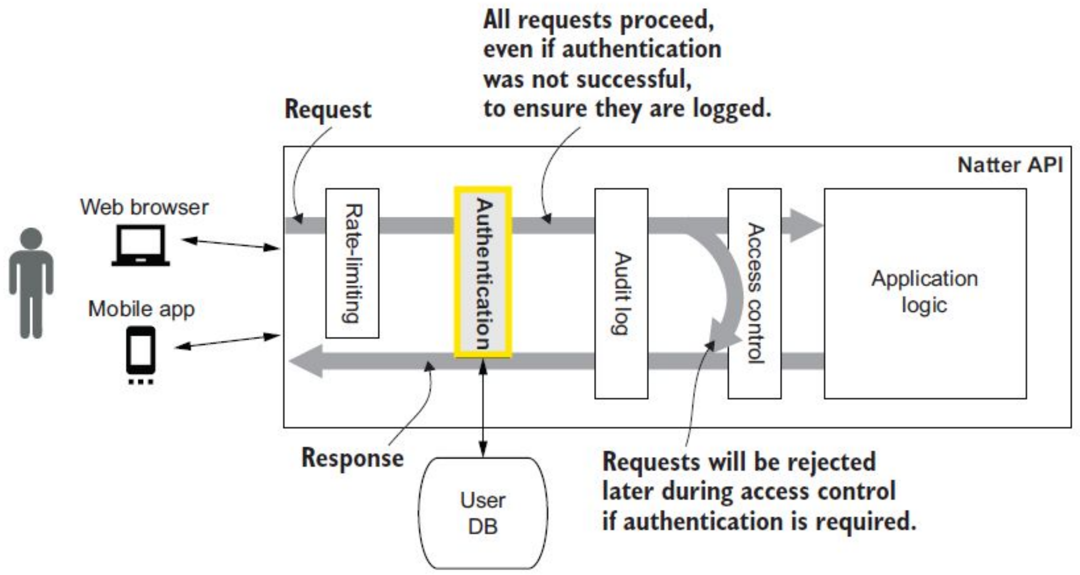
JWTs

## Authentication to prevent spoofing

Almost all operations in our API need to know who is performing them

Almost all security starts with authentication, which is the process of verifying that a user is who they say they are

Figure shows how authentication fits within the security controls that you'll add to the API in this section. Apart from rate-limiting (which is applied to all requests regardless of who they come from), authentication is the first process we perform

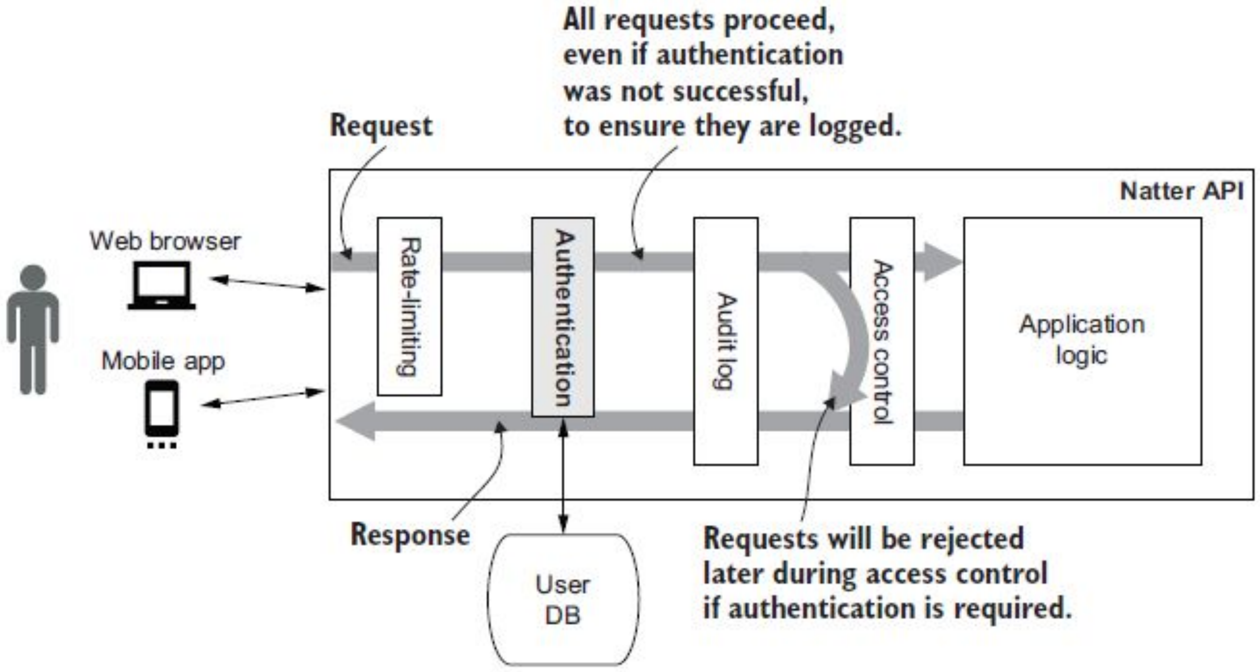

All requests proceed, even if authentication was not successful, to ensure they are logged.
Request
Natter API
Web browser
Mobile app
Rate-limiting
Authentication
Audit log
Access control
Application logic
Response
User DB
Requests will be rejected later during access control if authentication is required.

## Securing an API: Authentication

It is important to realize that the authentication phase itself shouldn't reject a request even if authentication fails

Deciding whether any particular request requires the user to be authenticated is the job of access control

Instead, the **authentication process will populate the request with attributes indicating whether the user was correctly authenticated** that can be used by these downstream processes


All requests proceed, even if authentication was not successful, to ensure they are logged.
Request
Natter API
Web browser
Mobile app
Rate-limiting
Authentication
Audit log
Access control
Application logic
Response
User DB
Requests will be rejected later during access control if authentication is required.

## HTTP Basic authentication

There are many ways of authenticating a user, but one of the most widespread is simple ***username*** and ***password*** authentication

In a web application with a user interface, we might implement this by presenting the user with a form to enter their username and password

An API is not responsible for rendering a UI, so you can instead use the standard **HTTP Basic authentication mechanism** to prompt for a password in a way that doesn’t depend on any UI

This is the mechanism in which the **username and password are encoded** (using Base64 encoding) and sent in a header. An example of a Basic authentication header for the **username demo** and **password changeit** is as follows

```
Authorization: Basic ZGVtbzpjaGFuZ2VpdA==
```

Encoding and Encryption are two different things

## HTTP Basic authentication

The **Authorization** header is a standard HTTP header **for sending credentials to the Server** (just like content-type is a header to communicate representation protocol)

**Authorization**: Basic base64(username:password)

HTTP Basic credentials are easy to decode for anybody able to read network messages between the client and the server. You should only ever send passwords over an encrypted connection. We will add encryption to the API communications

The reason for encoding a password before sending it is that HTTP headers are text-based and can contain only certain characters. Passwords can contain special characters and encoding them makes sure they are converted in characters that are acceptable in HTTP header.

# Basic HTTP Authentication

## Hands On

Rate Limiting -> Authentication?
Authentication -> Rate Limiting?

1. Start the App
2. In Postman - go to Authorization Tab
3. Try to make request to both /apiStatus and /payments (Check the response code)
4. In the Auth Type - Select Basic Auth and add username and password
5. Make the request
6. Check Postman Console for Authorization Header

## Securing Password Storage

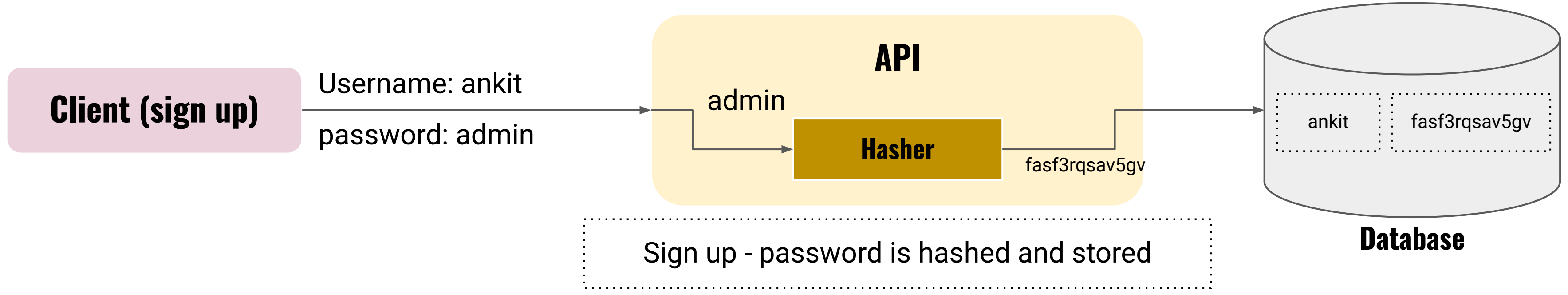
Client can easily send a username and password to the API using the request headers, but you need to securely store and validate that password

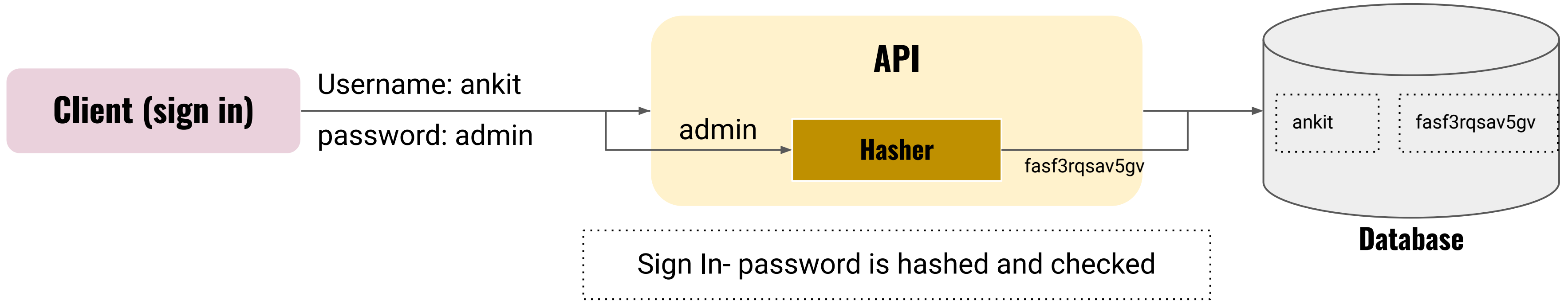
A password hashing algorithm converts each password into a fixed-length random-looking string. We never store the password in clear-text formats in a database (what is database is stolen/compromised)

When the user tries to login, the password they present is hashed using the same algorithm and compared to the hash stored in the database. This allows the password to be checked without storing it directly

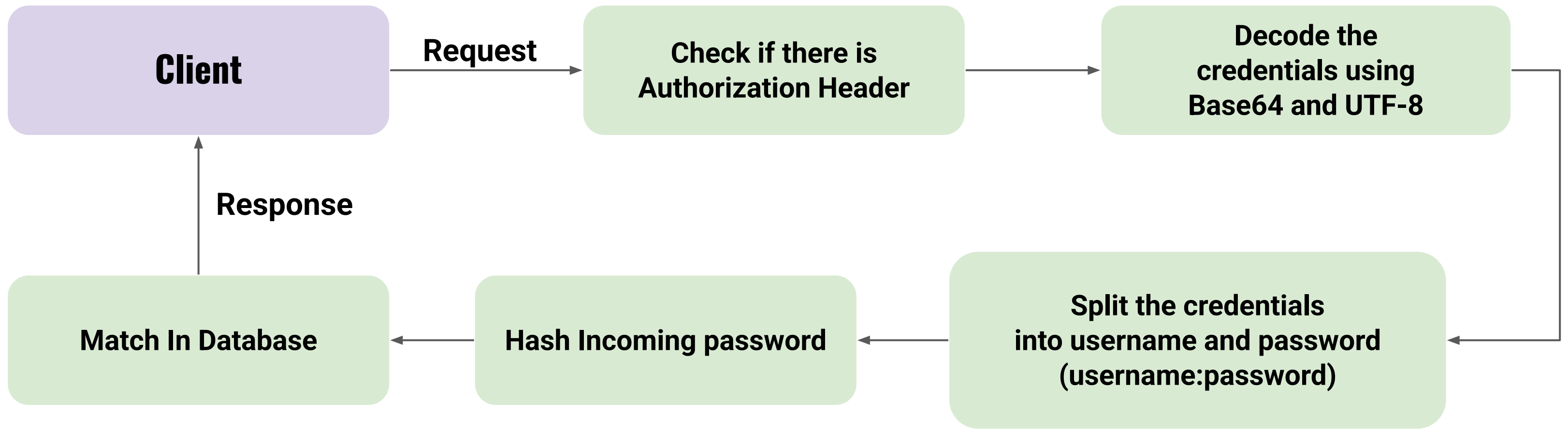
Note: unlike encryption which is two way process (encrypt and decrypt) hashing is a one way process. Once hashed, a string cannot be unhashed.
***Encoding ≠ Encryption ≠ Hashing***

# Password Hashing Flow





## Authentication Flow


Client
Request
Check if there is Authorization Header
Decode the credentials using Base64 and UTF-8
Split the credentials into username and password (username:password)
Hash Incoming password
Match In Database
Response

# Password Hashing

Hands On

## Authentication

Basic Auth

Stateful Token

Stateless Tokens

JWTs

## Token Based Authentication

So far, you have required API clients to submit a username and password on every API request to enforce authentication

Although simple, this approach has several downsides from both a **security** and **usability** point of view

In this section, you'll learn about those downsides and implement an alternative known as token-based authentication, where the username and password are supplied once to a dedicated login endpoint

A time-limited token is then issued to the client that can be used in place of the user's credentials for subsequent API calls

## 2 Drawbacks of HTTP Basic Authentication

The user's password is sent on every API call, increasing the chance of it **accidentally being exposed** by a bug in one of those operations

**Verifying a password is an expensive operation**, (specially because everytime password has to be **hashed** and then matched) and performing this validation on every API call adds a lot of overhead.
Check the response time - without hashing it would be around 10-20ms and with hashing it would go to 100ms+.
Modern password-hashing algorithms are designed to take around 100ms for interactive logins, which limits your API

## What is Token Based Authentication?

Let’s suppose that your users are complaining about the drawbacks of HTTP Basic authentication in your API and want a better authentication experience

The CPU overhead of all this password hashing on every request is killing performance and driving up energy costs too

What you want is a way for users to login once and then be trusted for the next hour or so while they use the API

This is the purpose of token-based authentication

## Flow of Token Based Authentication

When a user logs in by presenting their username and password, the API will generate a random string (the token) and give it to the client.

The client then presents the token on each subsequent request, and the API can look up the token in a database on the server to see which user is associated with that session

When the user logs out, or the token expires, it is deleted from the database, and the user must log in again if they want to keep using the API

To switch to token-based authentication, we generally implement a dedicated new login endpoint.
This endpoint could be a new route within an existing API or a brand-new API running as its own microservice

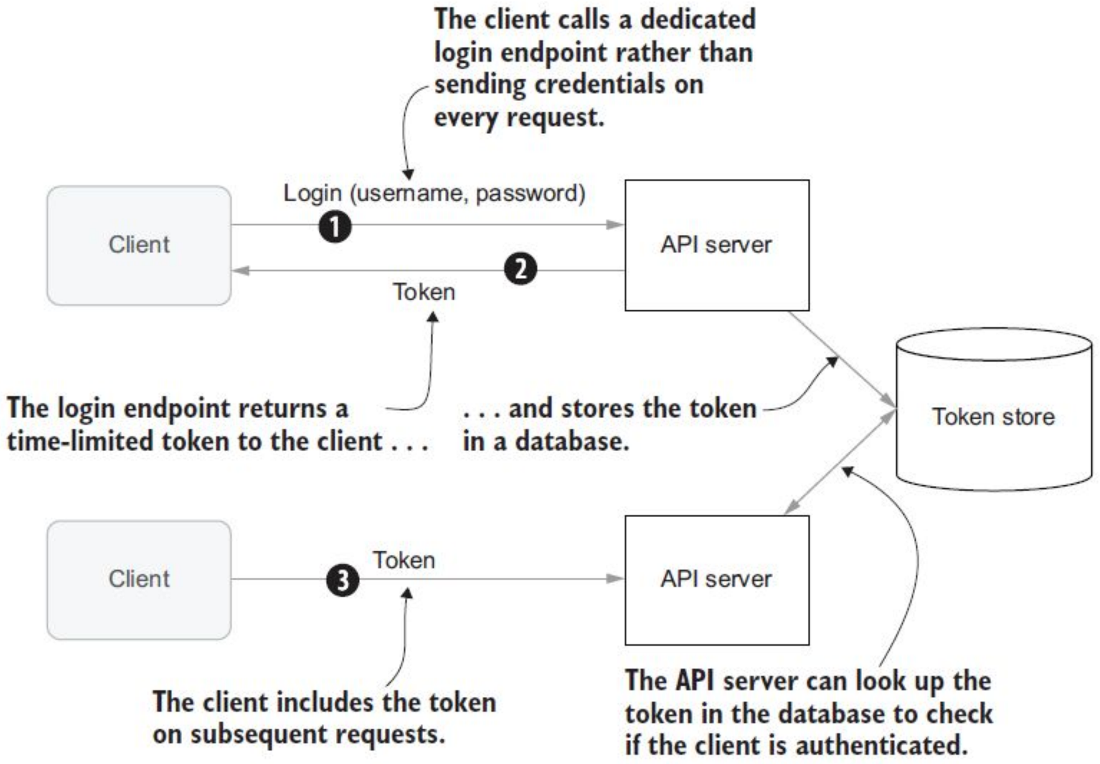
## Flow of Token Based Authentication

Token-based authentication is a little more complicated than the HTTP Basic authentication you have used so far, but the basic flow, shown in figure

Rather than send the username and password directly to each API endpoint, the client instead sends them to a dedicated login endpoint

The login endpoint verifies the username and password and then issues a time-limited token

The client then includes that token on subsequent API requests to authenticate

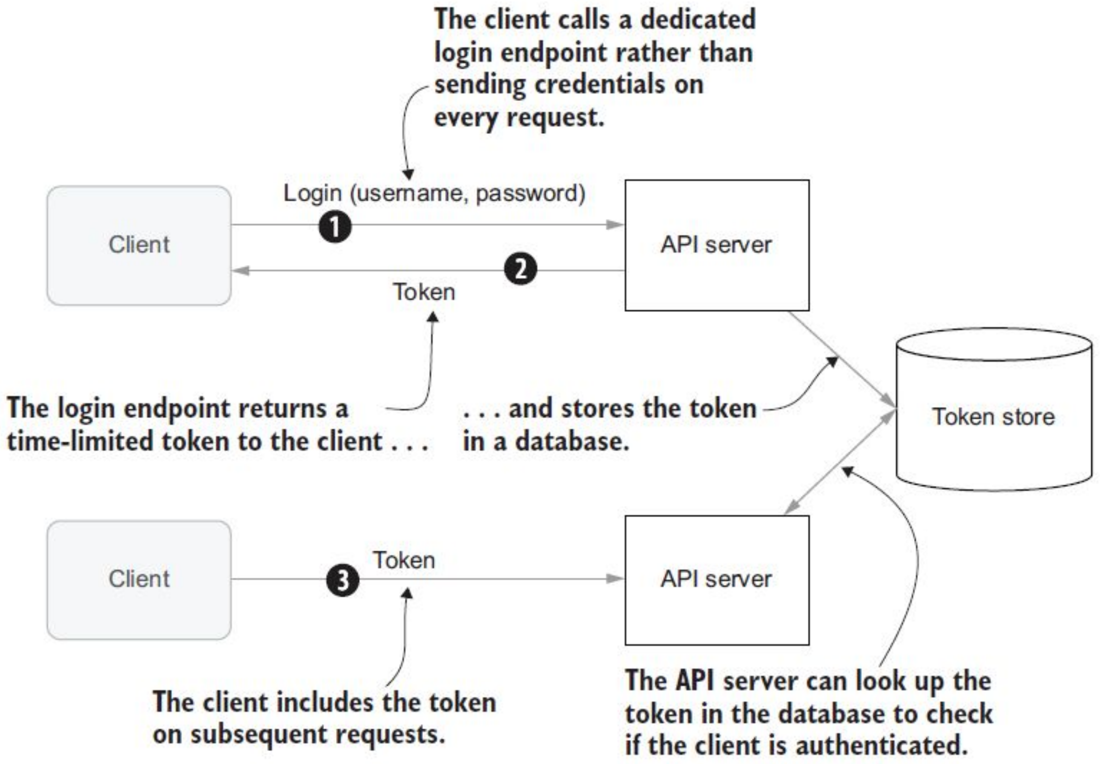

The client calls a dedicated login endpoint rather than sending credentials on every request.
Login (username, password)
1
Client
API server
2
Token
The login endpoint returns a time-limited token to the client . . .
. . . and stores the token in a database.
Token store
Client
3
Token
API server
The client includes the token on subsequent requests.
The API server can look up the token in the database to check if the client is authenticated.

## Flow of Token Based Authentication

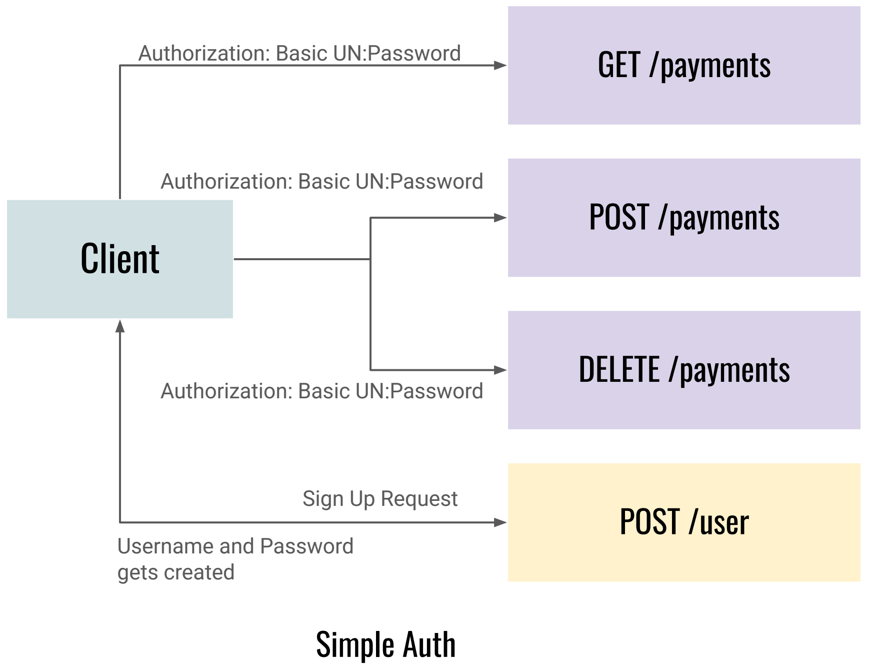
The API endpoint can validate the token because it is able to talk to a token store that is shared between the login endpoint and the API endpoint

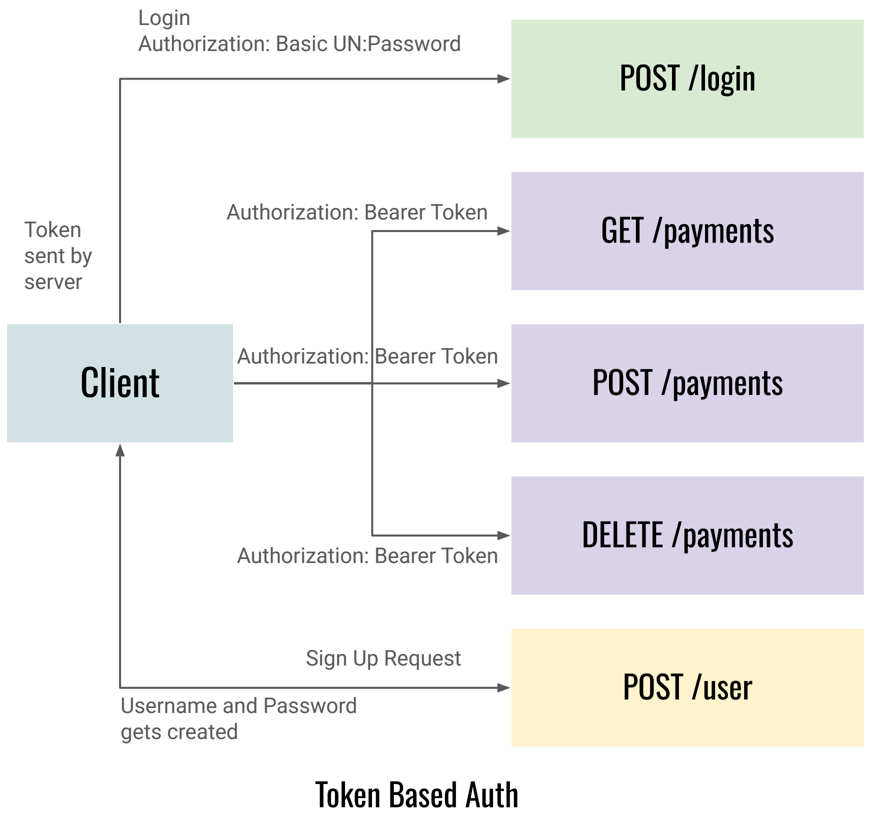
Generally, this token store is a shared database indexed by the token ID

The major scalability comes from the fact that in token based authentication - the server does not need to work on hashing the password everytime it gets any request


The client calls a dedicated login endpoint rather than sending credentials on every request.
Login (username, password)
1
Client
API server
2
Token
The login endpoint returns a time-limited token to the client . . .
. . . and stores the token in a database.
Token store
Client
3
Token
API server
The client includes the token on subsequent requests.
The API server can look up the token in the database to check if the client is authenticated.

# Simple Auth vs Token Based Auth


Authorization: Basic UN:Password
GET /payments
Authorization: Basic UN:Password
POST /payments
Client
DELETE /payments
Authorization: Basic UN:Password
Sign Up Request
POST /user
Username and Password gets created
Simple Auth


Login
Authorization: Basic UN:Password
POST /login
Authorization: Bearer Token
GET /payments
Token sent by server
Authorization: Bearer Token
Client
POST /payments
DELETE /payments
Authorization: Bearer Token
Sign Up Request
POST /user
Username and Password gets created
Token Based Auth

## Token Store Schema Highlights

A token store (token table) typically has following attributes or columns:

token_id

user_id

expiry_timestamp

attributes

Once the token is received by the client, subsequent request are made with adding token in the **Authorization header (just like username password)**

Token store should periodically be deleted based on expiry timestamps. This is important because as more users interact with API more token needs to be created and stored. If we don’t clean it up regularly, we might end up using all our disk space.

## Simple Token

### Hands On

## Self Contained Tokens (JWTs)

(Client Side Tokens)

## Good News, Bad News

We have shifted from the simple http-based-authentication to a token based authentication where the tokens are stored in databases and Web Storage (local storage/cookies, etc)

If your API is a hit - that’s a good news, but the bad news is that the token database is struggling to cope with this level of traffic

You’ve evaluated different database backends, but you’ve heard about stateless tokens that would allow you to get rid of the database entirely.

Without a database slowing you down, your API will be able to scale up as the user base continues to grow

## Storing token state on the client

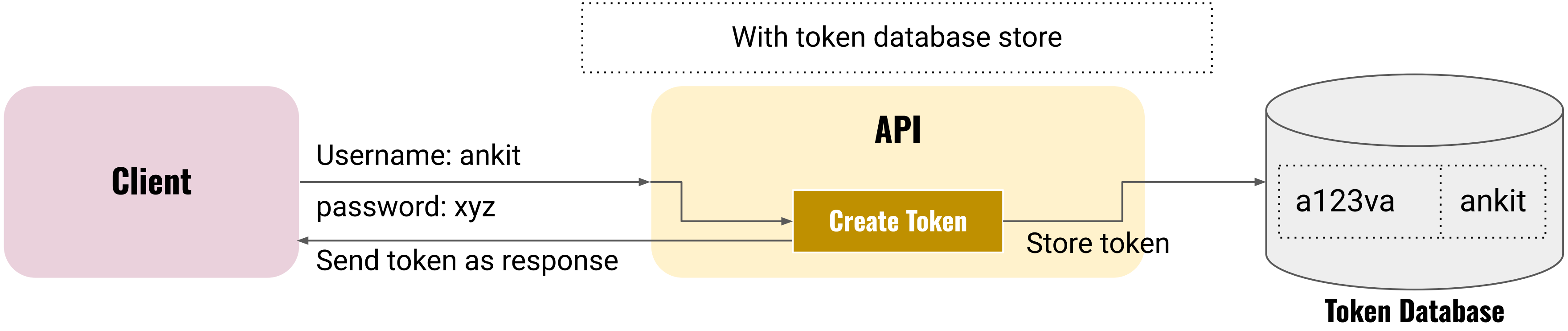
The idea behind stateless tokens is simple

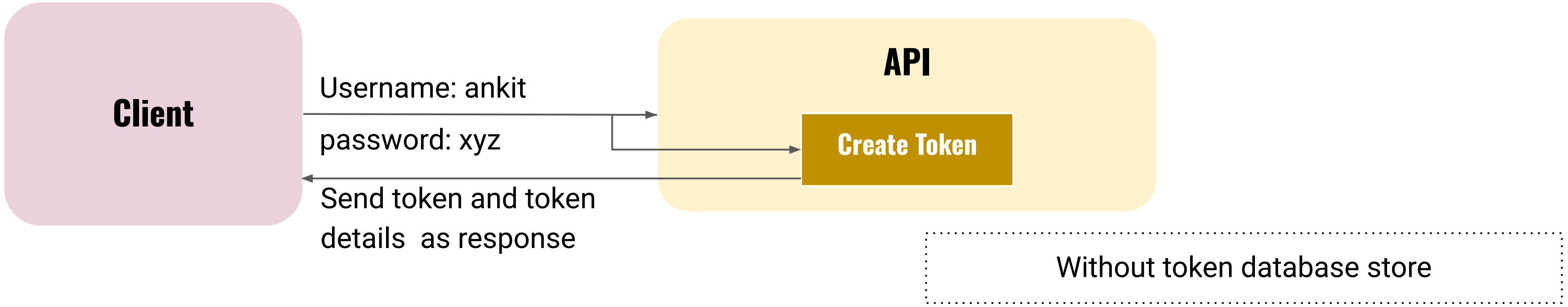
Rather than store the token state in the database, you can instead encode that state directly into the token ID and send it to the client

For example, you could serialize the token fields into a **JSON object**, which
you then **Base64url-encode** to create a string that you can use as the token ID
Example: { sub: username, exp: exiryTS, attributes: token.details }
(The keys of the json are also called ***claims***)

When the token is presented back to the API, you then simply decode the token and parse
the JSON to recover the attributes of the session

# With and Without Token DB Store




Client
Username: ankit
password: xyz
Send token and token details as response
API
Create Token
Without token database store

## There is a problem!

The idea of sending all the token details (id, expiry, username, etc) is completely unsafe

The client can modify the expiry time and keep using the same token forever, or can attach different username to the token and because server has no storage of token on any DB, there is no way server can find if the token is altered

We prevent these kind of token tampering with a HMAC (hash based message authentication code)

Let’s check how this HMAC protection works on the next slide

# Protecting Token With HMAC

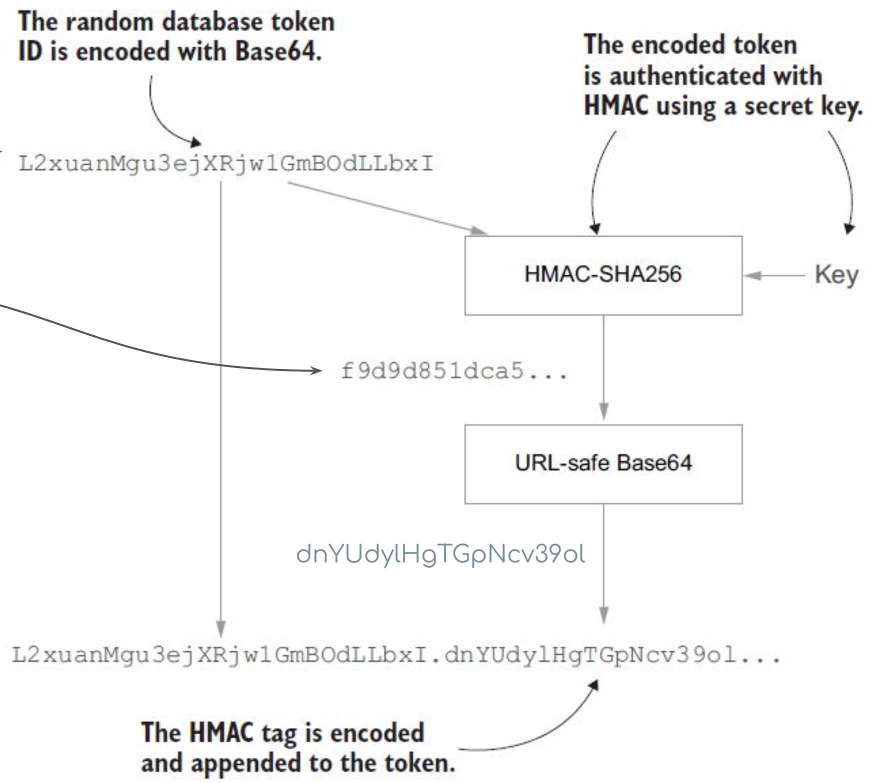
Token is generated

API maintains a secret key known only to them

With the token and Key as input to the HMAC generator - a code is generated

That code (HMAC tag) is added to the original token and sent to the client

Let’s see how it protects from token tampering



The random database token ID is encoded with Base64.
The encoded token is authenticated with HMAC using a secret key.
L2xuanMgu3ejXRjw1GmBOdLLbxI
HMAC-SHA256
Key
f9d9d851dca5...
URL-safe Base64
dnYUdylHgTGpNcv39ol
L2xuanMgu3ejXRjw1GmBOdLLbxI.dnYUdylHgTGpNcv39ol...
The HMAC tag is encoded and appended to the token.

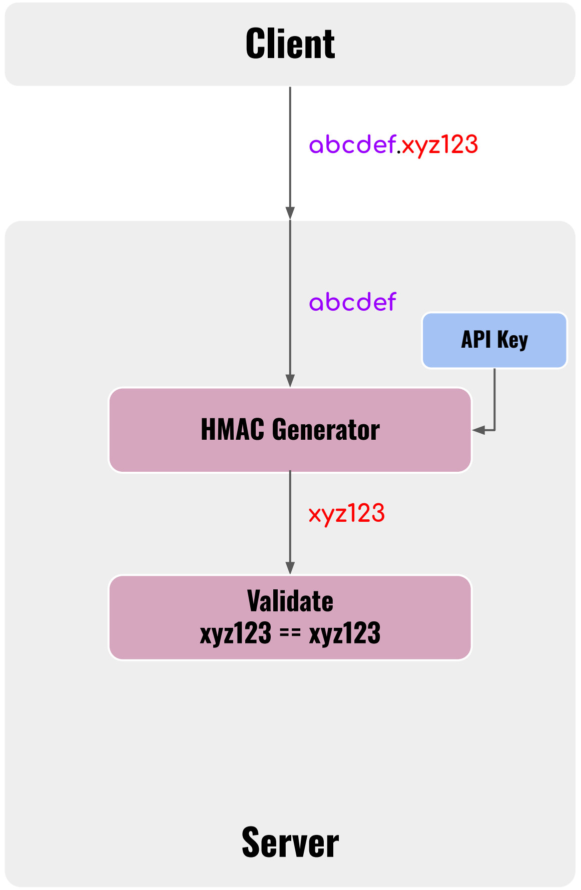
## Protecting Token With HMAC

Client sends the token along with HMAC tag to the server

Server extracts the token and calculates the HMAC tag with the API

The server then compares the calculated HMAC tag with the received HMAC tag

If the client tampers the token even by a single character, the newly calculate HMAC tag will not match with received HMAC tag and thus will be rejected



## Standardised Token: JSON Web Tokens (JWT)

Authenticated client-side tokens have become very popular in recent years, thanks in part to the standardization of JSON Web Tokens in 2015

JWTs are very similar to the JSON tokens you have just produced, but have standard features

- A standard header format that contains metadata about the JWT, such as which MAC or encryption algorithm was used
- A set of standard claims that can be used in the JSON content of the JWT, with defined meanings, such as exp to indicate the expiry time and sub for the subject

Because JWTs are standardized, they can be used with lots of existing tools, libraries, and services. JWT libraries exist for most programming languages now, and many API frameworks

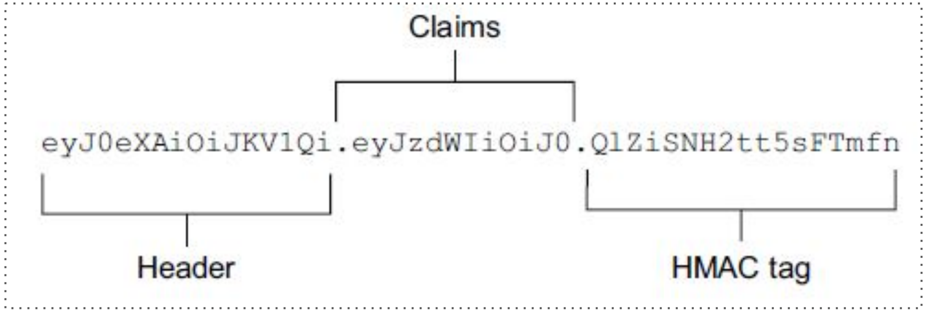
## JSON Web Tokens (JWT)

JWT contain standard set of keys (claims) inside them as listed below

| Claim | Name | Description |
|---|---|---|
| iss | issuer | Indicates who created the JWT (url of API) |
| iat | Issued-At | The timestamp at which the JWT was created |
| nbf | Not-Before | The JWT should be rejected if used before this time |
| exp | Expiry | The timestamp at which the JWT expires and should be rejected by recipients |
| sub | subject | The identity of the subject of the JWT. A string. Usually a username |

**Standard JWT Claims**

## JSON Web Tokens (JWT)



**Header**

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

**Claims**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

## JWT Hands On

Generate JWT Token

Decode JWT

Change anything in encoded JWT to see the error

Implement JWT in REST API (Next Slide)

Code

## JWT Based API Hands On

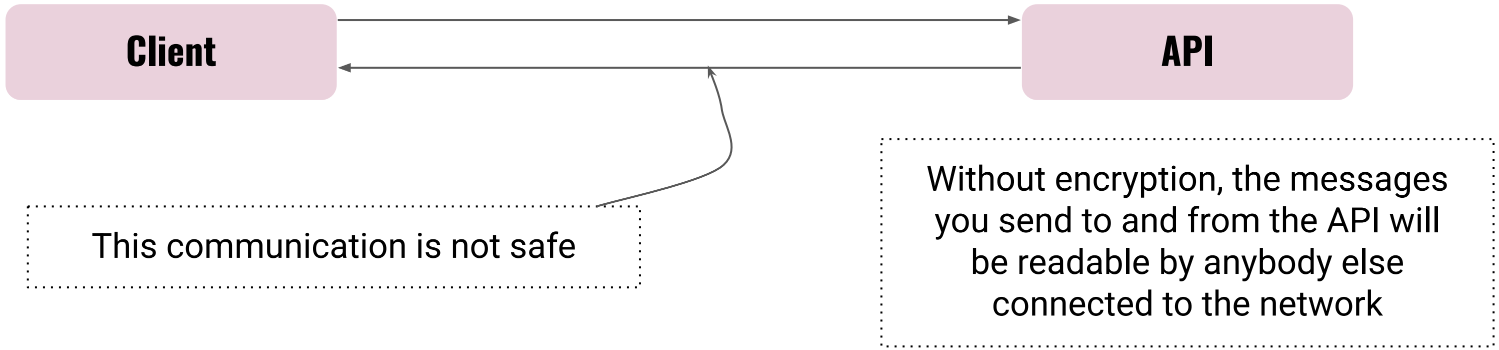
Code

## Encryption

## Using encryption to keep data private

Introducing authentication into your API protects against spoofing threats

However, requests to the API, and responses from it, are not protected in any way, leading to tampering and information disclosure threats



This communication is not safe

Without encryption, the messages you send to and from the API will be readable by anybody else connected to the network

## Using encryption to keep data private

Your simple password authentication scheme is also vulnerable to this snooping, as an attacker with access to the network can simply read your Base64-encoded passwords as they go by

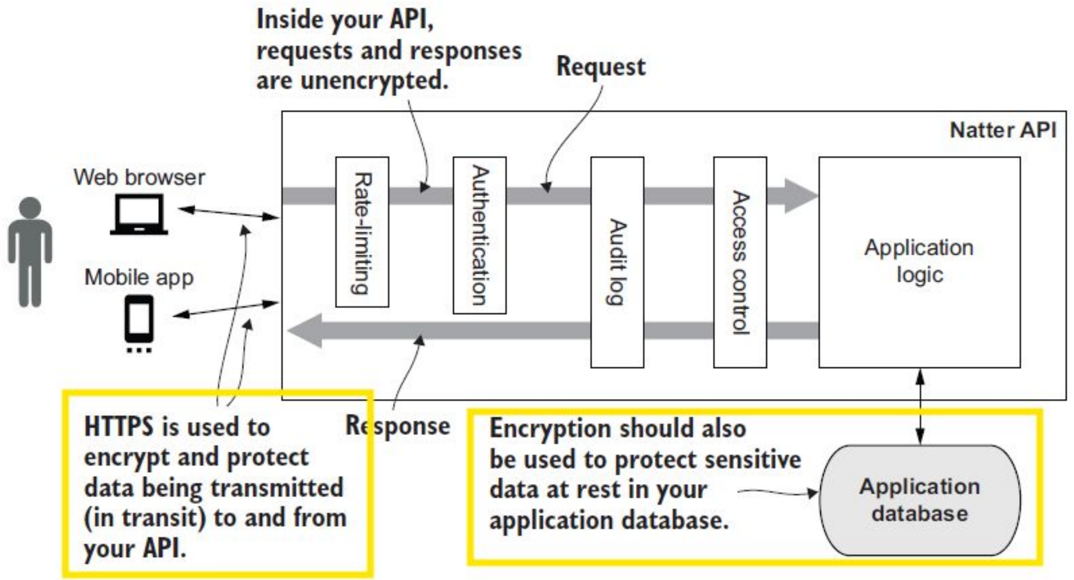
They can then impersonate any user whose password they have stolen

In this case, sending passwords in clear text is a pretty big vulnerability, so let’s fix that by enabling HTTPS

HTTPS is normal HTTP, just that the data before sending it out over the network is protected by encryption

## Using encryption to keep data private

Figure below shows how HTTPS fits into the picture, protecting the connections between your users and the API


Inside your API, requests and responses are unencrypted.
Request
Natter API
Web browser
Mobile app
Rate-limiting
Authentication
Audit log
Access control
Application logic
HTTPS is used to encrypt and protect data being transmitted (in transit) to and from your API.
Response
Encryption should also be used to protect sensitive data at rest in your application database.
Application database

## Steps Involved in HTTPS communication: High Level

First, you need to generate a certificate that the API will use to authenticate itself to its clients

When a client connects to your API it will use a URI that includes the hostname of the server the API is running on, for example api.npci.com

The server must present a certificate, signed by a trusted certificate authority (CA), that says that it really is the server for api.npci.com.

If an invalid certificate is presented, or it doesn't match the host that the client wanted to connect to, then the client will abort the connection. Without this step, the client might be tricked into connecting to the wrong server and then send its password or other confidential data to the imposter.

## The 3 Parties in HTTPS Communication: Server Not Ready for HTTPS

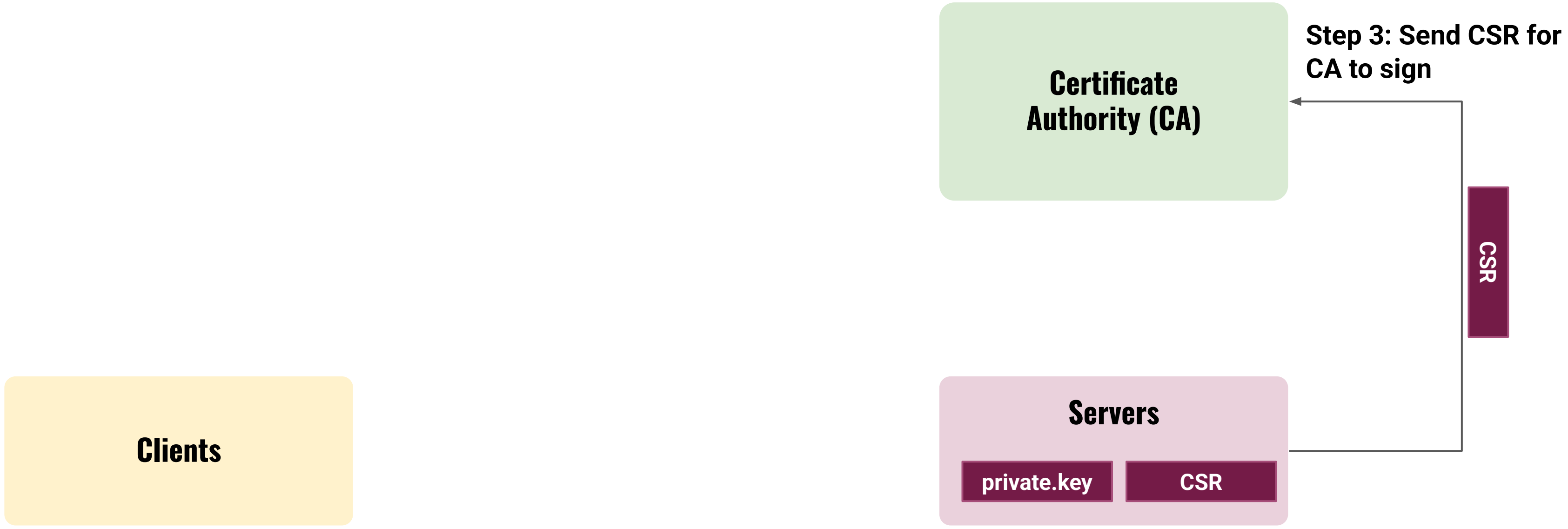
Certificate Authority (CA)
Clients
Servers

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS

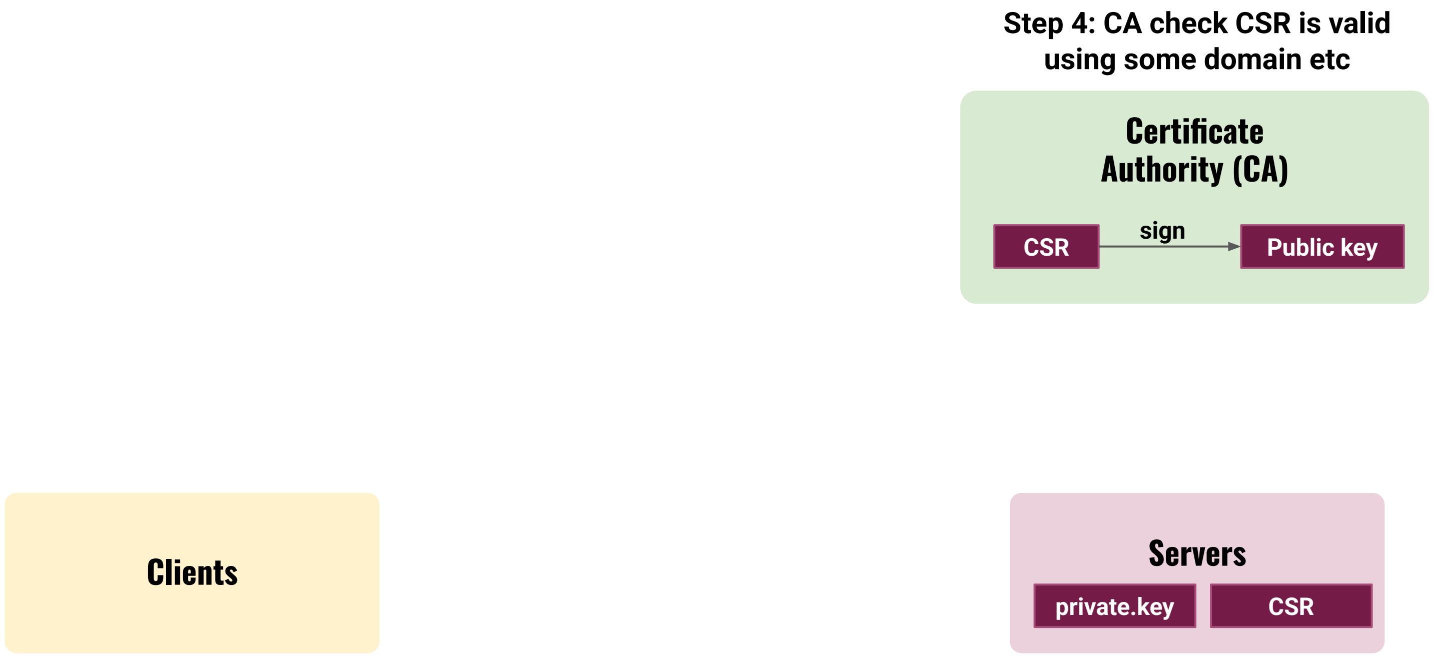
Certificate Authority (CA)
Clients
Servers
private.key
CSR

**Step 1: Create a private key**
**Step 2: Using private key create CSR (certificate signing request) which is public key ready to be signed**

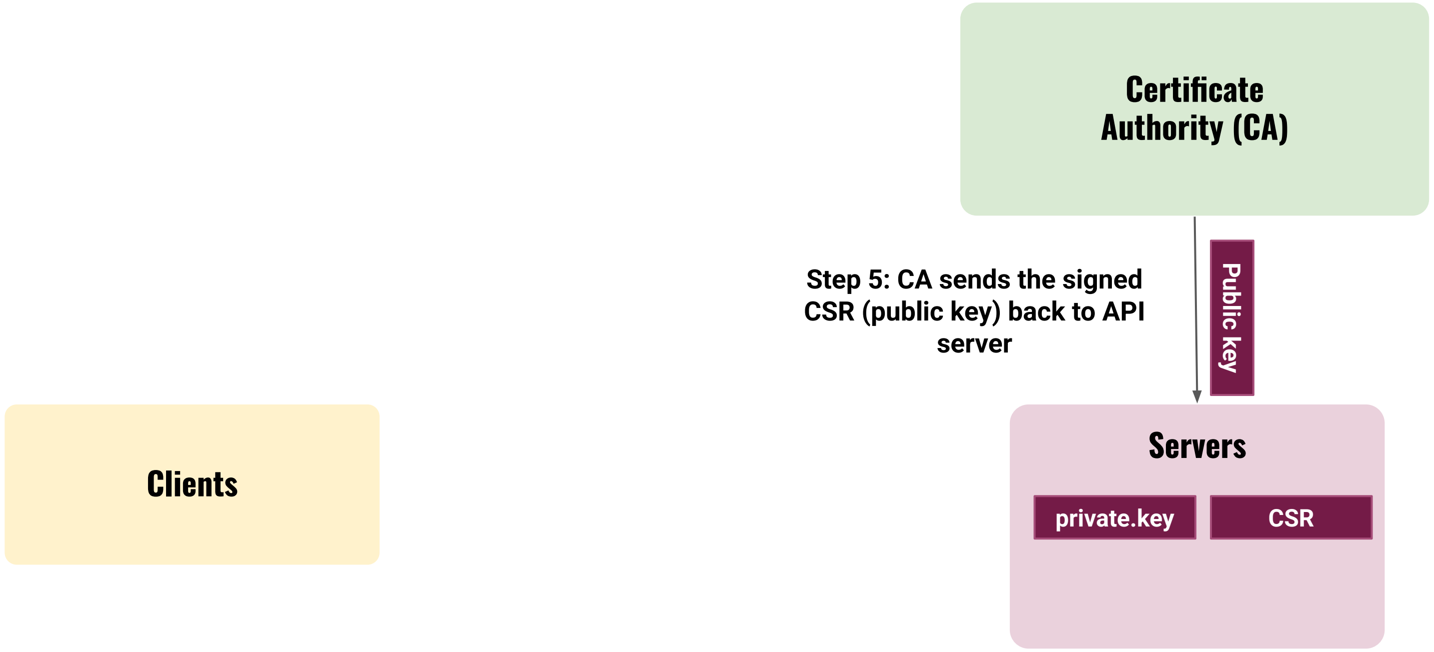
# The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



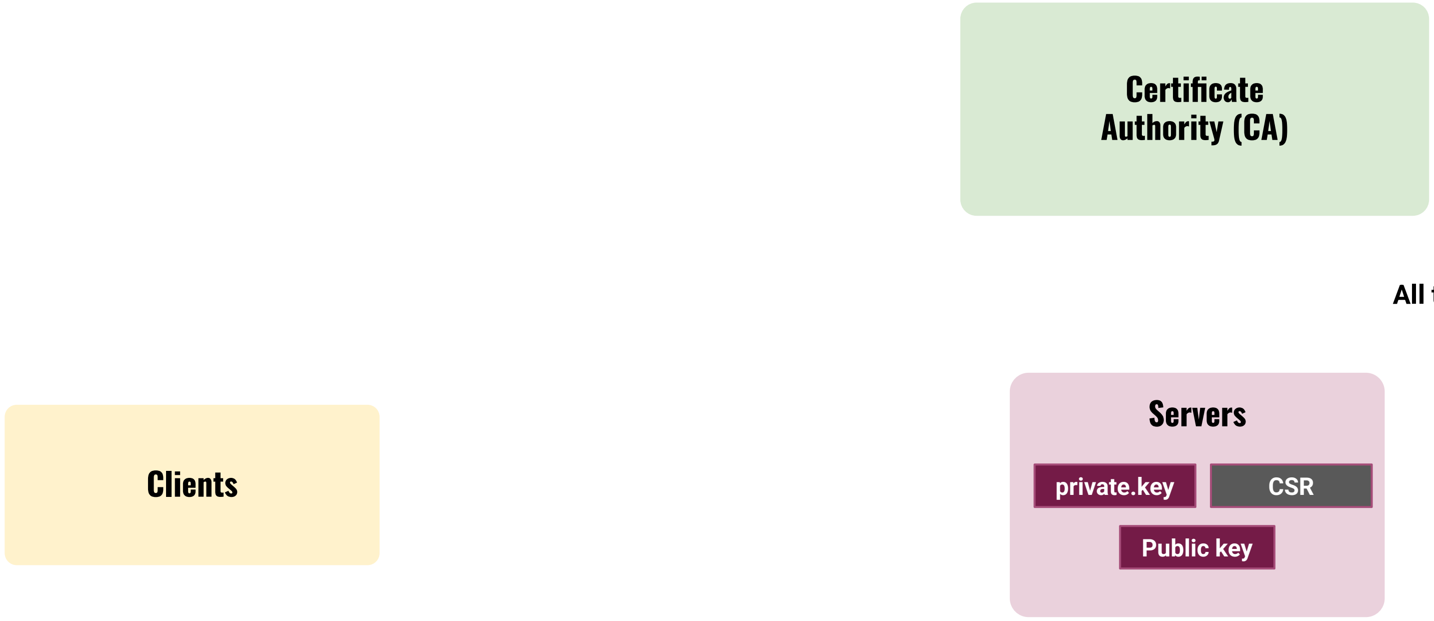
## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS


Step 4: CA check CSR is valid using some domain etc
Certificate Authority (CA)
CSR
sign
Public key
Clients
Servers
private.key
CSR

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS


Certificate Authority (CA)
Step 5: CA sends the signed CSR (public key) back to API server
Public key
Servers
private.key
CSR
Clients

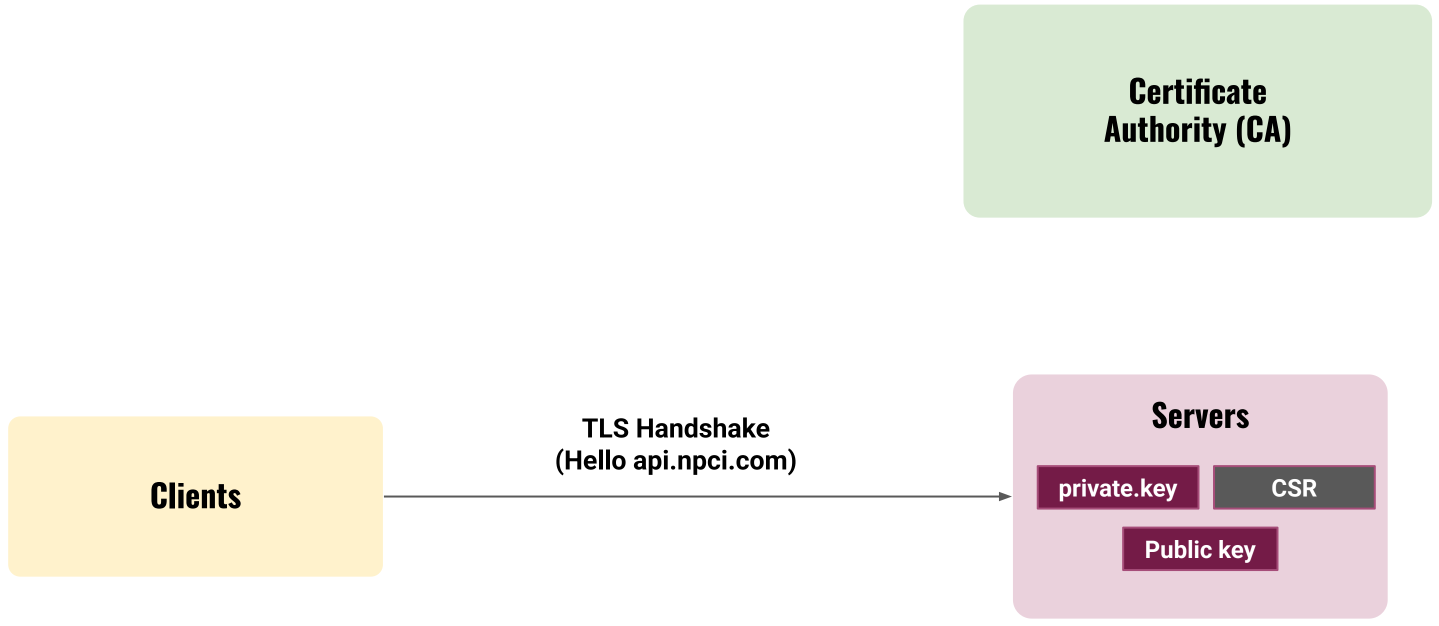
# The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



All this is just one time activity

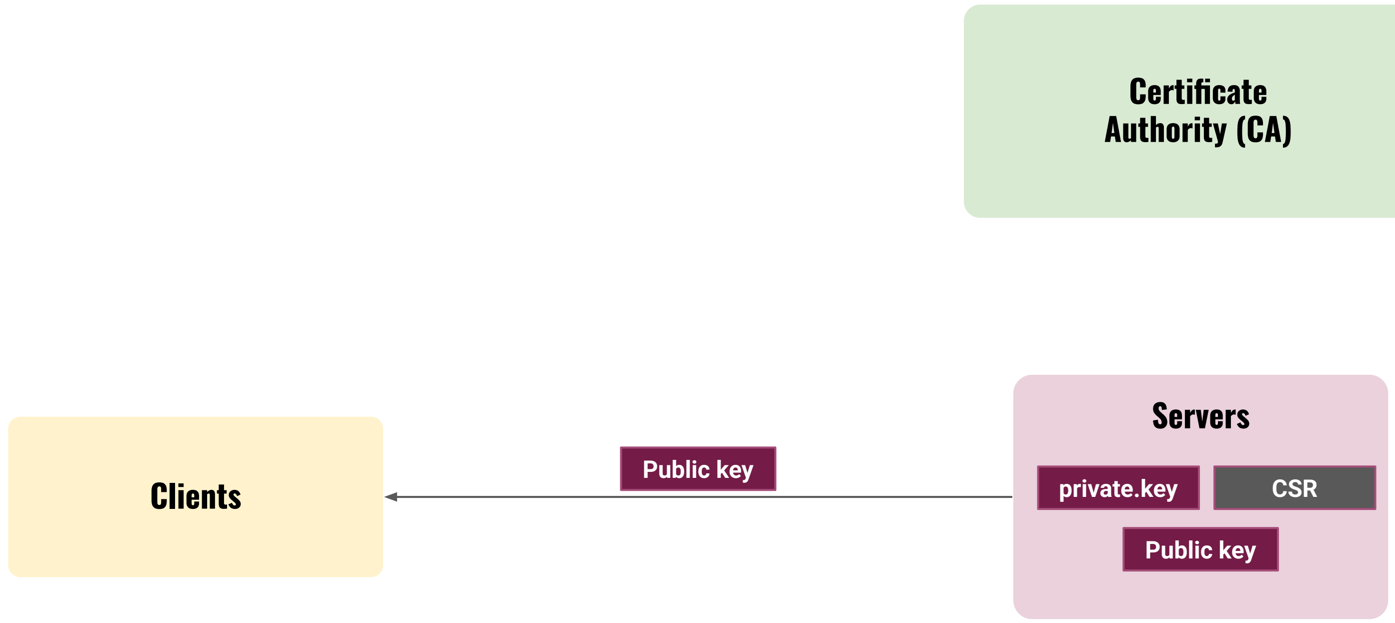
Server has a public key and private key

# The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS


Certificate Authority (CA)
TLS Handshake (Hello api.npci.com)
Clients
Servers
private.key
CSR
Public key

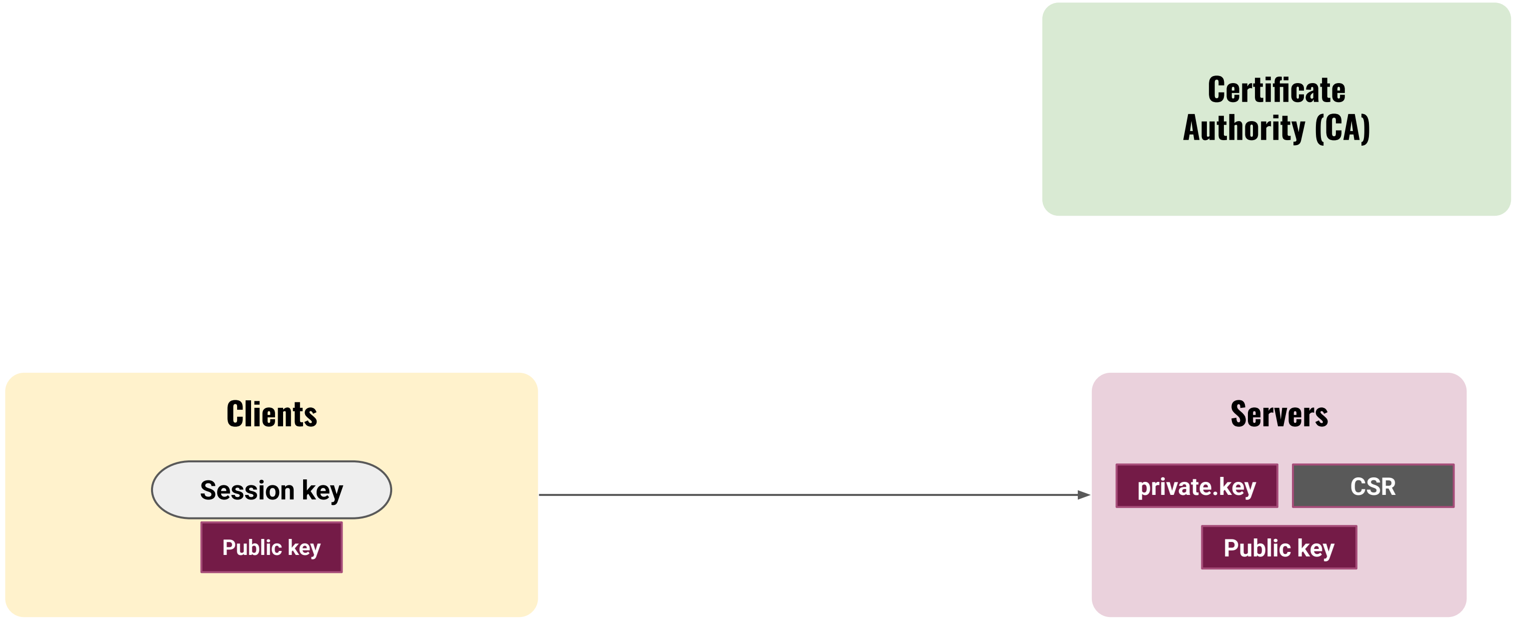
Before client can make an actual API request - client will send a handshake request

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



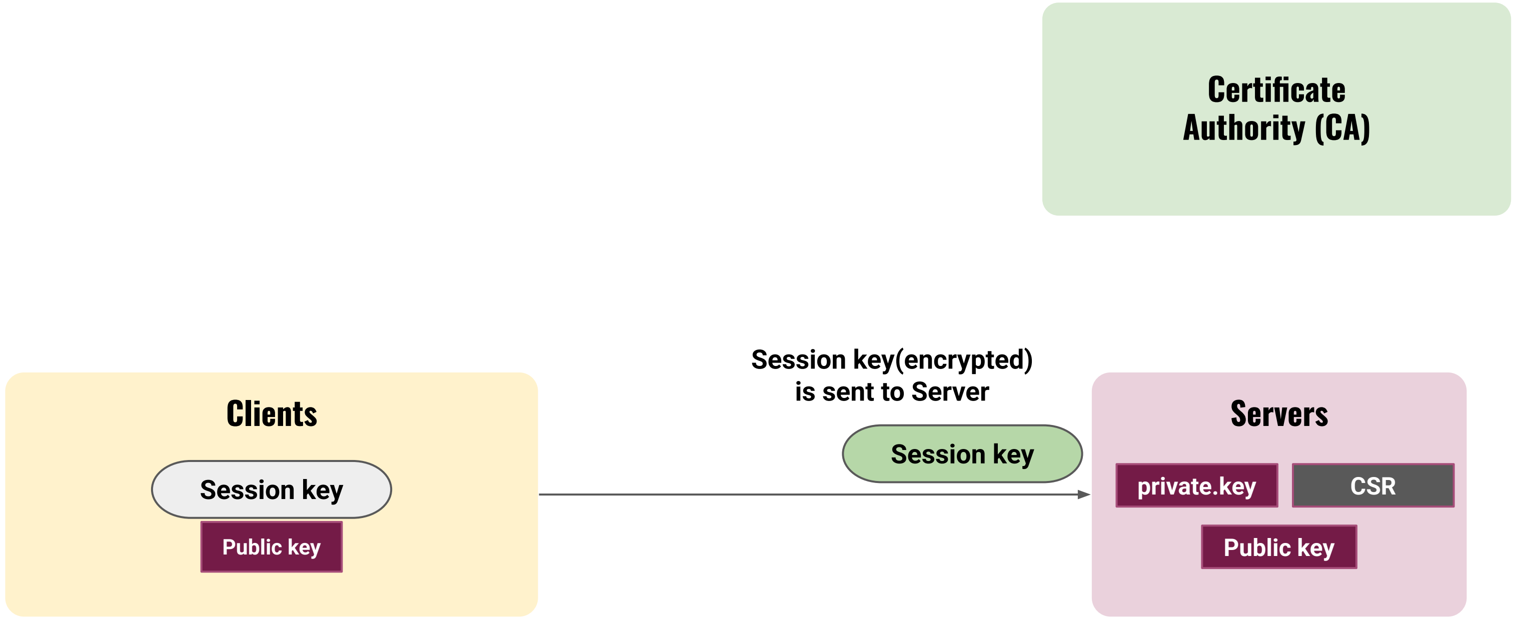
In response to the handshake request, API server will send it's public certificate to the Client

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS

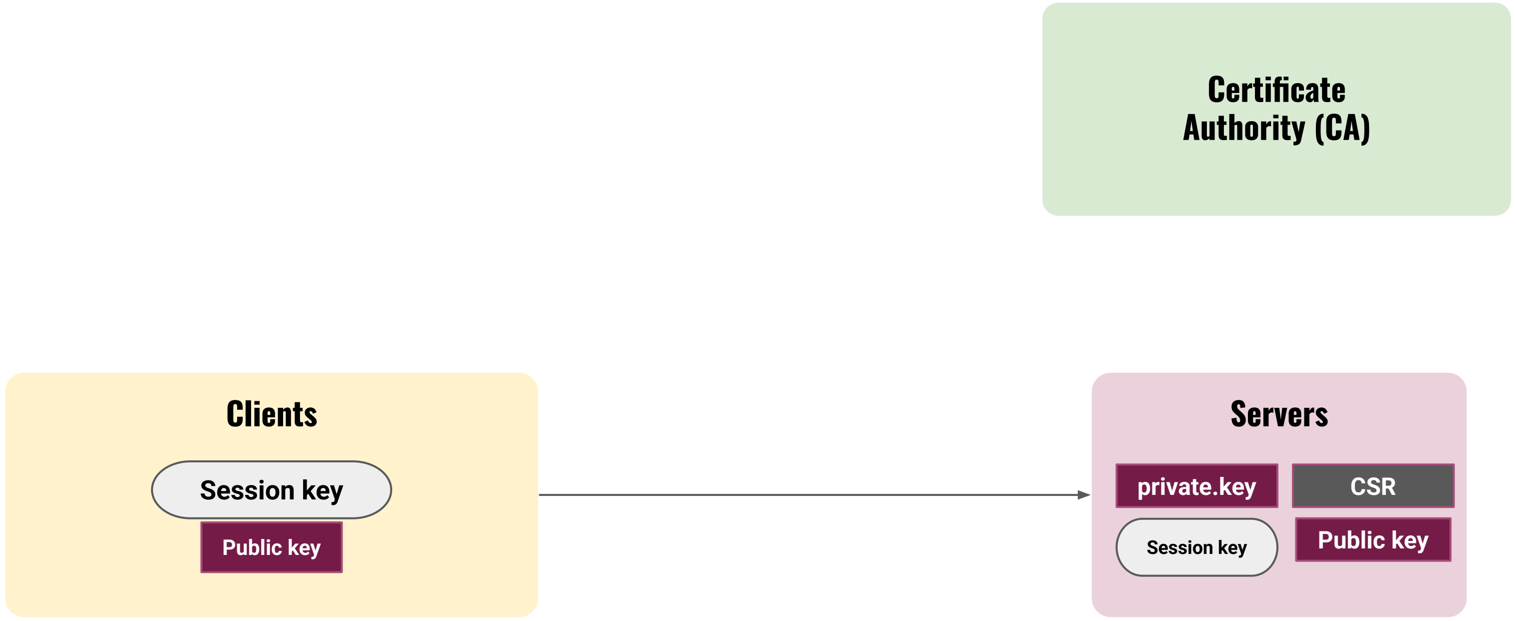


A session key is created and encrypted using public key

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS

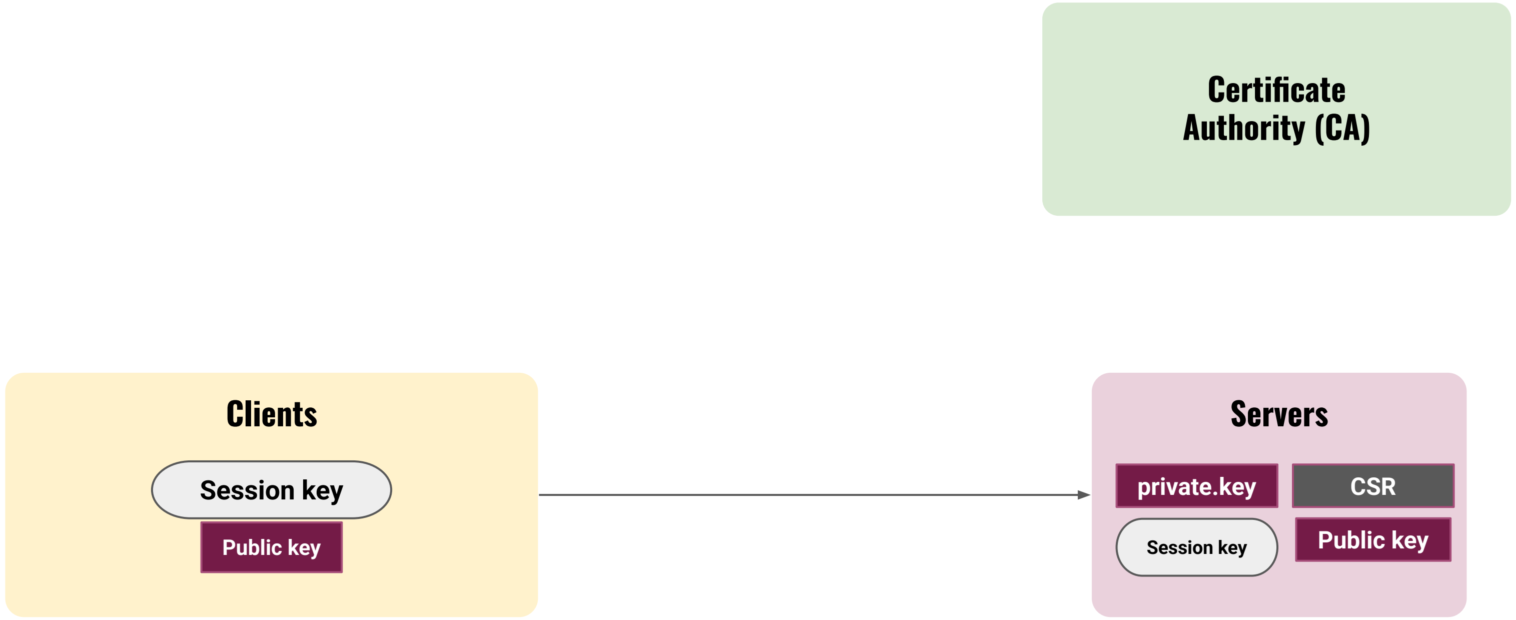


## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



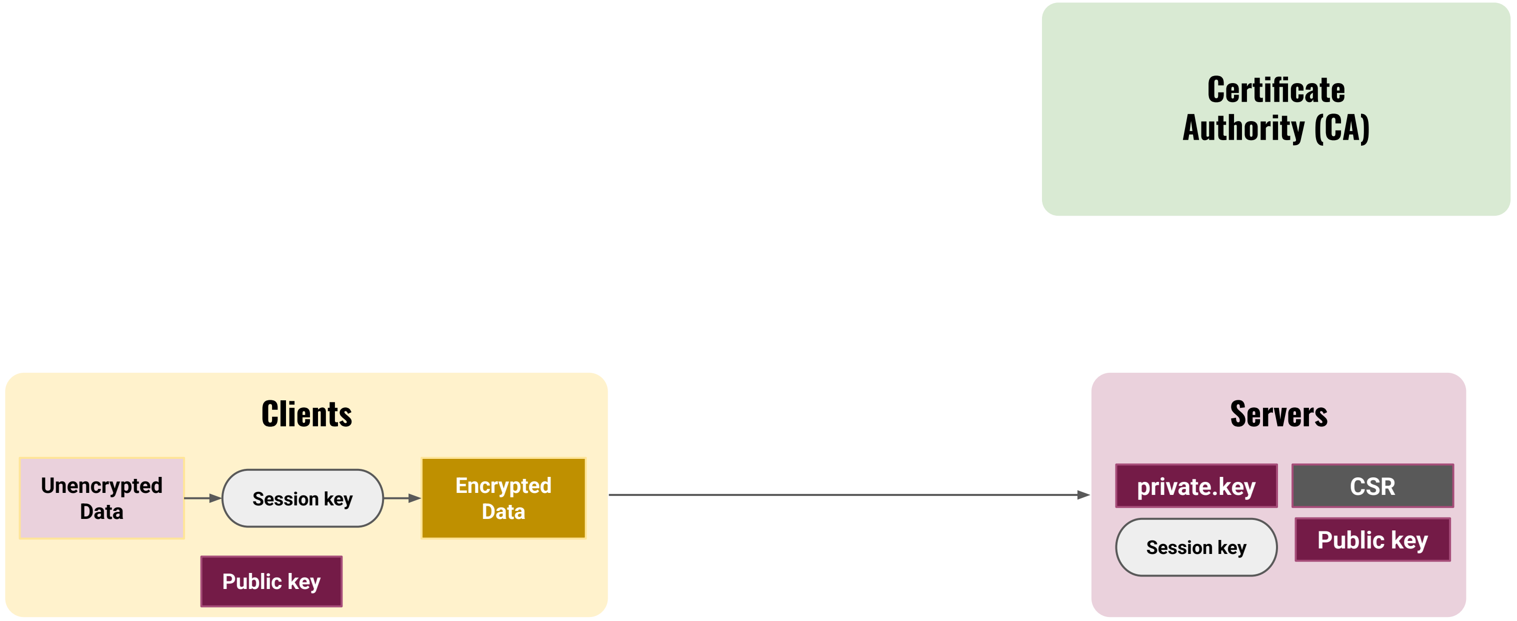
Session key is decrypted using private key

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



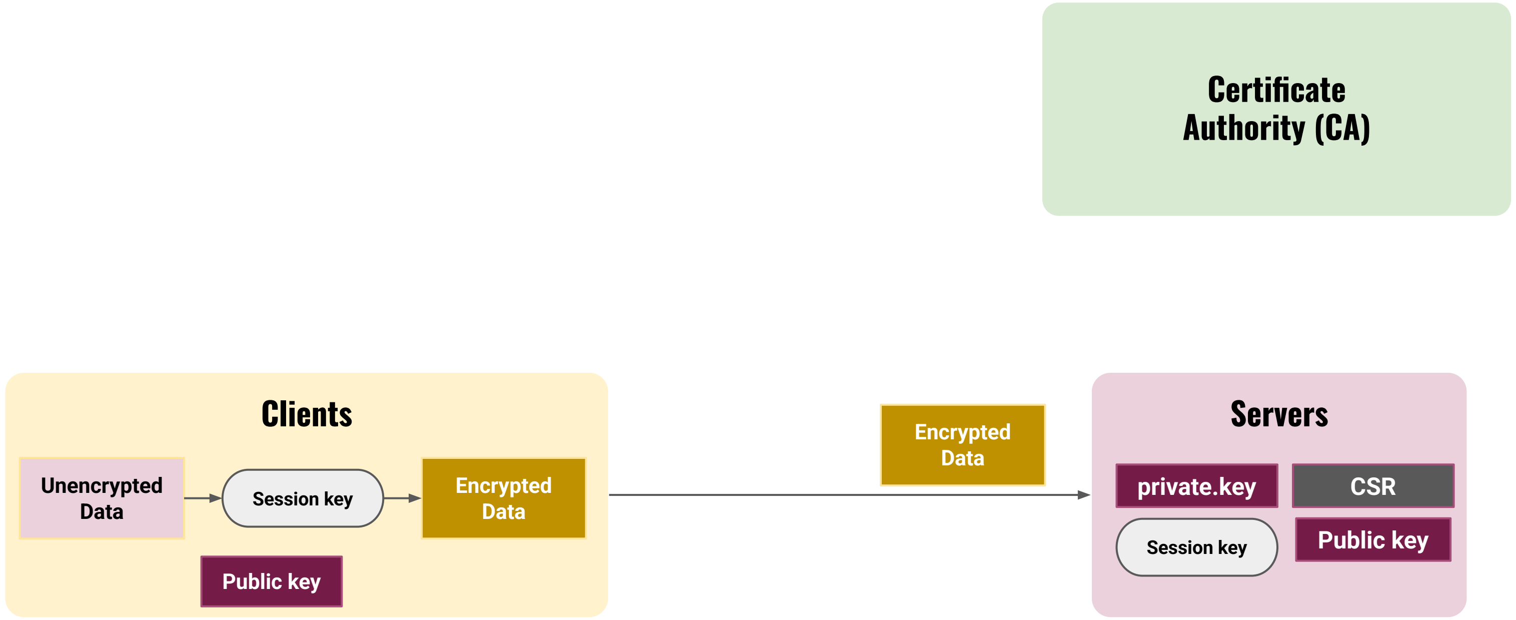
Both Client and server has the session key for 1 session

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS

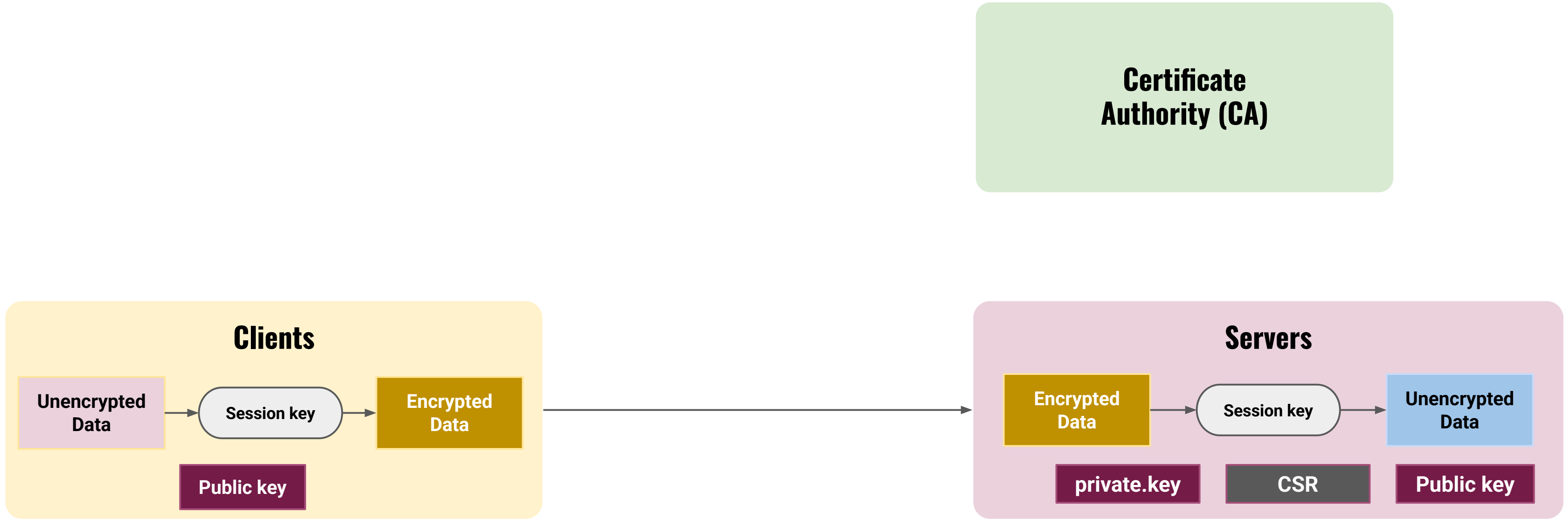


Client encrypts the data using session key

## The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



# The 3 Parties in HTTPS Communication: Server Is Ready for HTTPS



**Server decrypts the data using session key**

**Before sending the response server also encrypts the data using session key**

## Deployment

**Taking API to production - our last mile!**

## Taking your API to Production

Once your API is ready in terms of development, security, testing, monitoring, etc there is one last step, deployment

In terms of deployment there are two main things that we want our API to be in Production: **Scalable** and **Durable**

In **scalability**, in most cases we want horizontal scalability (in short deploy API multiple times) - but we don't want to manually start API 5-10 times

In **durability**, we want that if for any reason our API crashes - it restarts itself. Also in case of server reboots - the API should restart itself

We will achieve **scalability with gunicorn** - a production grade web server and **durability with systemd** - a readily available process manager in all Linux distributions

## Deployment

**Taking API to production - our last mile!**

**Hands On**

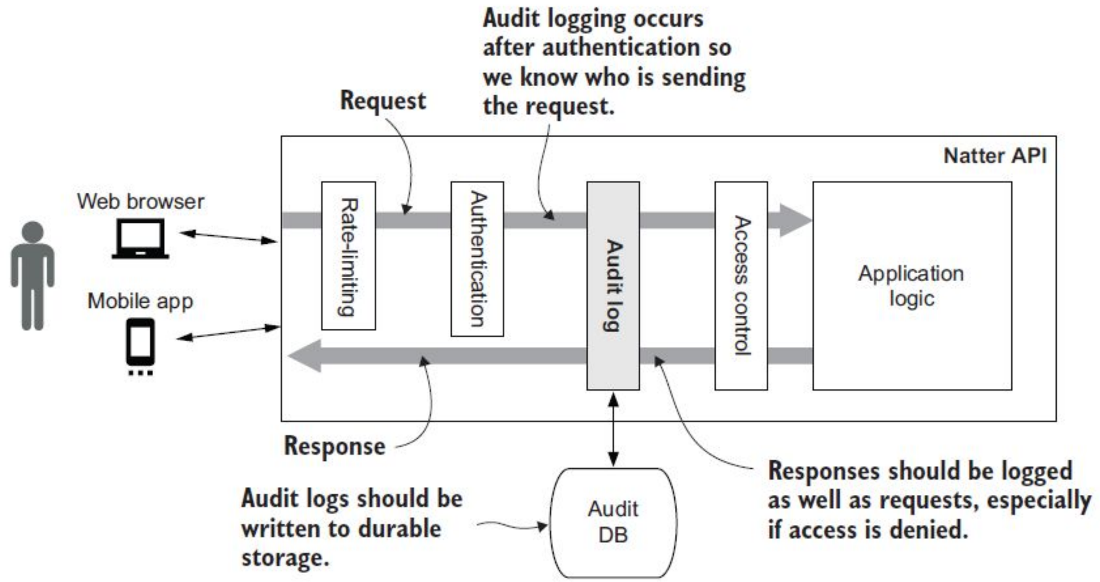
## Audit Logging

## Audit logging for accountability

Accountability relies on being able to determine who did what and when

The simplest way to do this is to keep a log of actions that people perform using your API, known as an audit log

Audit logging should occur after authentication, so that you know who is performing an action, but before you make authorization decisions that may deny access.


Audit logging occurs after authentication so we know who is sending the request.
Request
Natter API
Web browser
Mobile app
Rate-limiting
Authentication
Audit log
Access control
Application logic
Response
Audit DB
Audit logs should be written to durable storage.
Responses should be logged as well as requests, especially if access is denied.

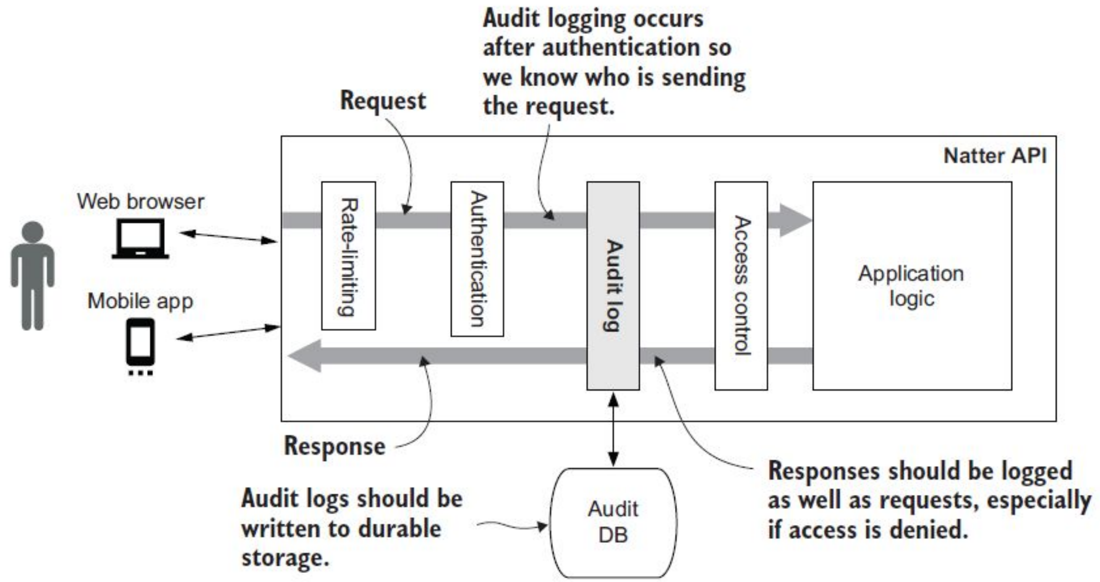
## Audit logging for accountability

The reason for this is that you want to record all attempted operations, not just the successful ones

Unsuccessful attempts to perform actions may be indications of an attempted attack

It’s difficult to overstate the importance of good audit logging to the security of an API

Audit logs should be written to durable storage, such as the file system or a database, so that the audit logs will survive if the process crashes for any reason


Audit logging occurs after authentication so we know who is sending the request.
Request
Natter API
Web browser
Mobile app
Rate-limiting
Authentication
Audit log
Access control
Application logic
Response
Audit logs should be written to durable storage.
Audit DB
Responses should be logged as well as requests, especially if access is denied.

## Keep Your Audit Logs Safe

Audits logs should be replicated and stored durably

Regular backup should be taken

It’s advisable to store them in more durable storage like S3 or Cloud Storage

## What to Log?

**REQUEST**

- Audit ID (Request ID)
- Method (GET, POST, PUT, Delete, etc)
- Path: /apiStatus, /payments, /payments{transaction_id} , etc
- Timestamps
- A check on special characters, allowed, un-allowed characters
- User Details
- Request Body (It's not a common practice to log request body due to size constraints)

**RESPONSE**

- Status Codes
- Response Timestamps
- Response Body (Uncommon)
- Business Domain (Payment done, user created, user removed, etc)

## Audit Logging

### Hands On

1. Add a before_request and after_request handler
2. Give a GUID to each request to track it - very important for microservices
3. Check the audit.log file