Practical Backend & Systems Study Guide – Easy Mode

Modern backend, infra, auth, security,
and system patterns explained in plain English.

{ HTTP • JWT • OAuth2 • Kafka • SSO • Docker • K8s }

☁ Cloud ▯ Docker ☸ K8s ▯ Auth ⚡ Kafka ▯ Patterns

by Ana Fernandes

How to Use This Guide

Purpose of this guide:

This guide is designed to explain modern backend, security, infrastructure and authentication concepts in clear, direct language. It is meant for developers, tech learners, and engineers who want to understand how real systems work without drowning in jargon.

How to read it:

• Each topic is self-contained. You do not have to read in order.

• The language is intentionally simple and direct. No fluff, no buzzwords unless they are defined.

• Bullet points are used instead of long paragraphs so you can revise quickly.

What you should get from this guide:

• A working mental model of how backend systems fit together: front-end → API → business logic → data → infra.

• A practical understanding of security basics: JWT, OAuth 2.0, SSO, sessions, tokens.

• Confidence talking about real production terms like Kafka, Kubernetes, rate limiting, etc.

How to use this for interviews / work:

• Use the summaries at the end of each section to describe concepts out loud in plain English.

• Learn the 'memory trick' lines — they’re built to stick and they’re great for explaining ideas to non-technical people (or managers).

• For each topic, ask yourself: Could I explain this to someone else in 30 seconds?

Who this is for:

• Developers who want to speak more confidently about systems design.

• People moving toward backend / infra / platform engineering.

• Anyone who has felt talked down to in tech and just wants straight answers.

Final note:

This guide is not theory for theory’s sake. Everything in here maps to what modern teams are actually doing in production today.

Table of Contents

1. PRESENTATION LAYER (UI / UX)

What it is:

- This is what the user can see and touch.
- Example: buttons, screens, forms, pages.

Why it matters:

- If this part is ugly or confusing, people leave.

Tools people use to build this:

- React, Next.js
- Angular
- Flutter (works on iPhone and Android)
- SwiftUI (iPhone / iPad apps)
- Kotlin Compose (Android apps)

In real life:

- It is like the shop front.
- How the shop window looks.
- How easy the door is to open.

2. EDGE & DELIVERY

What it is:

- This makes your app load fast everywhere in the world.
- It also blocks attacks.

Why it matters:

- If someone is in London and your server is in New York, this helps it still feel fast.

Tools people use:

- Cloudflare
- AWS CloudFront
- Akamai
- Fastly

In real life:

- It is like having mini-copies of your website in many countries, so no one has to wait.
- Also like a security guard at the door.

3. INTEGRATION LAYER (API)

What it is:

- This is how the front-end (the screen) talks to the back-end (the brain / server).
- It is like an interpreter.

Why it matters:

- Without this, the button on the screen cannot do anything.

Tools / styles:

- API Gateway
- GraphQL
- REST / Swagger
- gRPC
- WebSocket

In real life:

- Imagine you press 'Buy Now'.
- The API sends: 'User wants to buy item 123' to the system that handles money.

4. MESSAGING & ASYNC PROCESSING

What it is:

- This means: send a job to be done in the background.

Why it matters:

- Some jobs are too slow to do while the user is waiting.
- Examples: send an email, resize an image, charge a card, update reports.
- So instead of making the user stare at a loading circle, we say:
- 'Put this job on the list and do it shortly.'

Tools people use:

- Kafka
- RabbitMQ
- AWS SQS / SNS
- Google Pub/Sub
- Redis Streams
- Celery
- KEDA

In real life:

- It is like a queue at the post office.
- Everyone takes a ticket.
- Jobs are handled one by one, calmly, in the back.

5. BUSINESS LOGIC LAYER

What it is:

- This is the rules of the business. The brain.
- This is where the app decides things.

What it decides:

- Can this user log in?
- Do they have money?
- Are they allowed to see this data?
- What happens next?

Tools people use to build this brain:

- Node.js (Express, NestJS)
- Python (FastAPI, Django)
- Java (Spring Boot)
- .NET
- Go
- Serverless functions

In real life:

- Think of a worker in an office who says:
- 'Yes, that is approved' or 'No, that breaks the rules.'
- That worker is this layer.

6. DATA ACCESS LAYER

What it is:

- This is the helper between the brain and the database.

The job is:

- Ask the database for the right info.
- Make it fast.
- Keep it tidy.
- Sometimes it also keeps a quick copy of data in memory,
- so you do not have to keep asking the database again and again.

Tools people use:

- Redis (fast temporary memory)
- Elasticsearch / OpenSearch (fast search)
- Prisma
- SQLAlchemy
- Hibernate

In real life:

- It is like a librarian who knows exactly which shelf to go to,
- so you do not waste time looking.

7. DATA STORAGE LAYER

What it is:

- This is where the data actually lives.

Things it stores:

- User accounts
- Orders
- Messages
- Photos
- Reports

Tools people use:

- PostgreSQL, MySQL (classic databases with tables)
- MongoDB, DynamoDB, Firestore (more flexible shape)
- BigQuery, Snowflake, ClickHouse (for big data and reports)

In real life:

- This is the filing cabinet.
- Or the big hard drive in the back office.

8. ANALYTICS & ML

What it is:

- This means using the data to learn things.

Examples:

- Which users are most active?
- Who is likely to buy again?
- Should we recommend this product to you?
- Can we spot fraud?

Tools people use:

- Spark
- PyTorch
- TensorFlow
- Vertex AI / SageMaker (train AI models)
- Looker, Power BI (dashboards and charts)
- LangChain + Vector DBs (AI that can chat with your data)

In real life:

- This is the 'insights' team.
- They look at all the data and say:
- 'This is what is really happening. Here is what we should do next.'

9. INFRASTRUCTURE LAYER (HOSTING / RUNTIME)

What it is:

- This is where everything runs.
- The machines. The cloud. The containers.
- The tools to deploy and update.

Tools people use:

- AWS, Azure, GCP (clouds)
- Docker (package the app so it can run anywhere)
- Kubernetes (manage lots of containers)
- Terraform, Helm, Argo CD (keep setups the same and ship updates safely)

In real life:

- Imagine you own 1,000 little computers around the world.

- You need a way to start apps on them, stop apps, update apps, and not break anything.
- This layer is the team that keeps the lights on.

10. QUICK SUMMARY (TOP TO BOTTOM)

1. USER INTERFACE
- What the user sees.

2. SPEED AND SECURITY AT THE EDGE
- Make it fast and safe.

3. API / CONNECTION LAYER
- Pass the message from screen to system.

4. BACKGROUND JOBS
- Do slow work later, not while the user waits.

5. BUSINESS RULES / LOGIC
- Decide what should happen.

6. DATA ACCESS HELPER
- Fetch info from storage, quickly and cleanly.

7. DATABASES / STORAGE
- All the data lives here.

8. ANALYTICS AND AI
- Learn from the data. Predict. Recommend.

9. INFRASTRUCTURE / PLATFORM
- All of this runs on computers somewhere. That 'somewhere' is this layer.

1. API (APPLICATION PROGRAMMING INTERFACE)

Plain meaning:

- An API is a way for two different computer systems to talk to each other.
- Example: You press a button in App A, and App A asks App B to do something.

Purpose:

- Let one system ask another system for information or to do an action.
- Example: A phone app calls a payment service to take money.

How do you know how to use it?

- You read the docs or manual that a human wrote.
- The docs tell you what to send, where to send it, and what you get back.

Is it always the same style?

- No. There are different types (flavours) of API.
- Here are the main ones and what they mean:

REST (common web style):

- You go to an address like /users/123 and get data back, usually in JSON.
- Very common on the internet.
- Good for: 'Give me info' or 'Update this thing.'

GraphQL (only what I ask for):

- You ask one question and say exactly which fields you want back.
- Example: 'Give me the user's name and email only, nothing else.'
- Good for apps that do not want a lot of extra data.

gRPC (super fast service-to-service):

- Used for very fast talking between services, often inside big systems.
- More strict and more compact, which makes it fast.
- Good for speed and performance in the backend.

SOAP (older, very strict):

- Very formal and uses XML, which is very wordy text.
- Still used in banks, government, and older systems where rules must be followed very tightly.

Short version of all four:

- REST = common, simple web style.
- GraphQL = 'only give me exactly what I ask for.'
- gRPC = very fast, computer-to-computer talking.
- SOAP = older, strict, but trusted in serious places.

2. MCP (MODEL CONTEXT PROTOCOL)

Plain meaning:

- MCP is a newer way for AI tools and AI assistants to talk to other systems.
- It is made for AI, not just normal apps.

Example:

- An AI inside your code editor asks a tool: 'Run this check for me,' and gets the answer.

Purpose:

- Let AI clients (AI bots, coding assistants, large language models) talk to servers and tools.
- The AI can say things like:
- 'Please run this command.'
- 'Please get me this file.'
- 'Please give me this data.'

How do you know how to use it?

- MCP can explain itself.
- It can tell you:
- 'Here are the tools I have.'
- 'Here is what each tool does.'
- 'Here is what I need from you when you call it.'
- So you do not always need a big PDF manual.

Is it always the same style?

- Yes. MCP tries to be one standard way to talk about:
- Resources (things you can read / get),
- Tools (things you can run / do),
- Prompts (things you can ask / request).
- So everything looks the same shape to the AI.

3. WHY THIS MATTERS

APIs:

- APIs are everywhere already.
- They are trusted and used in almost every normal app, website, and company system today.
- APIs run most of the world right now.

MCP:

- MCP is built for AI work.
- It is easier for AI to understand what it can do.
- It is easier for AI tools to plug into new systems without a human reading long docs.
- It tries to keep one way to talk, not many different styles.

4. SUPER SIMPLE SUMMARY

API =

- Apps talking to apps.
- Needs human-written docs.
- Many styles (REST, GraphQL, gRPC, SOAP).

MCP =

- AI talking to tools.
- Explains itself.
- One style.

1. COOKIES

Plain meaning:

- A cookie is a tiny piece of data saved in your web browser (for example, Chrome or Safari).
- Websites use cookies to remember who you are.

How it works (step by step):

- You log in to a website.
- The website sends a cookie to your browser.
- Your browser saves that cookie.
- Next time you click something on that website, your browser sends the cookie back to the website automatically.
- The website reads the cookie and says, 'Oh, it's you again. I know who you are.'

Good things about cookies:

- They are simple to use.
- Good for small and short-term data, like 'this user is logged in' or 'dark mode on/off.'

Problems with cookies:

- If the cookie is not protected (not encrypted / not secured), someone else could steal it and pretend to be you.
- So cookies can be risky if not done safely.

2. SESSIONS

Plain meaning:

- A session means the real data about you is kept on the server, not in your browser.
- Your browser only keeps a small ID number called a 'session ID.'

How it works (step by step):

- You log in.
- The server creates a session and stores your info there (for example: user id, role, cart, etc.).
- The server sends your browser a session ID (just an ID, like 'abc123').
- Your browser keeps that ID and sends it with every request.
- The server looks at the ID and pulls your data from its memory or database.

Good things about sessions:

- More secure, because your personal data stays on the server side.
- If someone steals just the session ID, the server can also block it or kill that session.

Problems with sessions:

- The server must store a lot of user sessions in memory or in a database.
- That uses more server resources, especially if you have lots of users logged in at once.
- It also needs more management to clean up old sessions.

3. COOKIES VS SESSIONS (QUICK VIEW)

Main difference:

- Cookies = data is stored in the browser (on the user's device).
- Sessions = data is stored on the server (the website's system).

Which is better?

- Cookies are easier and lighter if you only need to remember small stuff.
- Sessions are usually safer for important info, like user identity or permissions.

Which one should I use?

- Most real apps use both together.
- Example: the browser keeps only a small session ID in a cookie, and the real user info lives safely on the server.
- The right choice depends on how big your app is, how secure it must be, and how fast it needs to be.

TOPIC: HTTP METHODS (WEB REQUEST METHODS)

Plain meaning:

- When your app talks to a server, it sends a request.
- The request must say WHAT it wants to do.
- HTTP methods are the words for WHAT you want to do.

Example:

- Do you want to GET data?
- Do you want to SAVE new data?
- Do you want to CHANGE data?
- Do you want to DELETE data?

Why this matters:

- You will see these methods in web development and APIs all the time.
- They tell the server exactly what action to take.

1. GET

What it does:

- GET asks the server for data.
- It does NOT change anything on the server.

Easy way to think about it:

- Like reading a book.
- You are only looking. You are not editing the book.

Common use:

- Get user info.
- Get a list of products.
- Load a web page.

2. POST

What it does:

- POST sends NEW data to the server.
- Normally used to create something new.

Easy way to think about it:

- Like adding a new contact to your phone.
- You are creating a new item that was not there before.

Common use:

- Sign up a new user.
- Create a new order.
- Add a new comment.

3. PUT

What it does:

- PUT replaces an existing item with a full new version.
- You send the full updated data.

Easy way to think about it:

- Imagine you have a note.
- You throw away the old note and write a new one from scratch.
- So the whole thing is replaced.

Common use:

- Update ALL fields of a user profile in one go.

4. PATCH

What it does:

- PATCH changes part of an item.
- You ONLY send what needs to change.

Easy way to think about it:

- Instead of rewriting the whole note, you just fix one line.
- Small edit, not full replace.

Common use:

- Update just the email of a user.
- Update just the delivery status of an order.

5. DELETE

What it does:

- DELETE removes something from the server.
- You are saying: 'Take this away.'

Easy way to think about it:

- Like deleting a photo from your phone.
- Once it's gone on the server, it stays gone.

Common use:

- Delete a user account.
- Delete a product.
- Delete a post or comment.

6. OPTIONS

What it does:

- OPTIONS asks the server: 'What am I allowed to do here?'

Easy way to think about it:

- Like looking at a restaurant menu before ordering.
- You are not ordering yet. You are just asking, 'What can I ask for?'

Common use:

- Used a lot in browsers to check security rules (CORS).
- The browser might send OPTIONS before sending the real request.

7. HEAD

What it does:

- HEAD is like GET but without the body/content.
- It only asks for the headers (information about the content).

Easy way to think about it:

- Like looking at the book cover without opening the book.
- You can see info ABOUT the thing, but not the thing itself.

Common use:

- Check if something exists.
- Check size/type of a file before downloading it.

8. TRACE

What it does:

- TRACE is used to test and debug the path of the request.
- The server sends back what it got, so you can see how the request travelled.

Easy way to think about it:

- Like tracking the route a message took on its way to someone.
- You are checking: 'What happened to my request on the way there?'

Common use:

- Mostly for debugging/network testing, not for normal app features.

9. CONNECT

What it does:

- CONNECT asks the server to open a direct tunnel (a direct line) between you and the server.

Easy way to think about it:

- Like saying, 'Make a private channel so I can talk securely.'

Common use:

- Often used for secure HTTPS connections (encrypted traffic).

FAST RECAP

Most common in normal work:

- GET = read data.
- POST = create new data.
- PUT = replace whole thing.
- PATCH = change part of a thing.
- DELETE = remove it.

Used more for special jobs:

- OPTIONS = ask 'what can I do?'.
- HEAD = just the info, not the full content.
- TRACE = debug path.
- CONNECT = make a secure tunnel.

API SECURITY: 12 SIMPLE TIPS

Plain meaning:

- These are basic rules to help keep your API safe.
- An API is how apps talk to each other.
- If your API is not protected, attackers can steal data or break things.

Goal:

- Keep bad people out.
- Protect user data.
- Make sure only the right people can do the right things.

1. RATE LIMITING AND THROTTLING

What it means:

- Do not let someone call your API too many times in a short time.

Why it matters:

- Stops abuse and bots that try to break your system.
- Helps slow down DDoS (flood) attacks.

In plain English:

- Limit how fast people can knock on the door.

2. INPUT VALIDATION

What it means:

- Always check the data your API receives from the user.

Why it matters:

- Prevents attacks like SQL injection or XSS (bad code sent into your system).
- Only allow clean, expected data. Block the rest.

In plain English:

- Do not trust random input. Check it first.

3. USE HTTPS

What it means:

- Always use HTTPS, not HTTP.

Why it matters:

- HTTPS encrypts data while it is moving between the user and the server.
- This stops other people from reading sensitive info, like passwords.

In plain English:

- Lock the conversation so no one can listen in.

4. AUTHENTICATION

What it means:

- Check: 'Who are you?' before letting someone use the API.

How:

- Use strong login methods like OAuth 2.0, API keys, tokens, etc.

Why it matters:

- Only real, allowed users or systems should be able to call your API.

In plain English:

- No ID = no entry.

5. AUTHORIZATION

What it means:

- After the user is logged in, limit what they can do or see.

Why it matters:

- Not every user should see everything.
- Give each user only what they actually need for their role.

In plain English:

- Even if you are inside the building, you still cannot open every door.

6. DATA ENCRYPTION AT REST

What it means:

- Encrypt (lock) sensitive data in the database or storage.

Why it matters:

- If someone gets into the database, the data should still look unreadable to them.
- So stolen data is harder to use.

In plain English:

- The files are saved in a locked box, not left open.

7. API GATEWAY

What it means:

- Put an API gateway in front of your API.

Why it matters:

- It helps control traffic.
- It can handle login/authentication, limits, and monitoring for you.

In plain English:

- It is like a front desk that checks everyone before they get inside.

8. REGULAR SECURITY AUDITS

What it means:

- Test your API often to look for problems.

Why it matters:

- Find holes early and fix them before attackers find them.

In plain English:

- Do routine health checks for your API.

9. DEPENDENCY MANAGEMENT

What it means:

- Keep all libraries, packages, and tools up to date.

Why it matters:

- Old versions can have known security bugs.
- Attackers search for systems still using old, weak versions.

In plain English:

- Do your updates. Don't stay on old, broken stuff.

10. LOGGING AND MONITORING

What it means:

- Record what is happening in your API.

Why it matters:

- You can spot weird or dangerous behaviour (for example, too many failed logins).
- You can react fast if something goes wrong.

In plain English:

- Keep security cameras on. Watch the door.

11. API VERSIONING

What it means:

- Give your API clear versions like v1, v2, v3.

Why it matters:

- You can upgrade safely without breaking old apps that still use the older version.
- Lets you fix security problems in new versions while keeping old ones separate.

In plain English:

- Move forward without breaking people still on the older path.

12. CONTINUOUS IMPROVEMENT

What it means:

- Security is not 'do it once and forget.'

Why it matters:

- New threats appear all the time.
- You should review your API security often and improve your rules and tools.

In plain English:

- Keep updating your locks. Attackers keep getting smarter.

FAST RECAP

Core ideas to remember:

- Limit traffic (rate limiting).
- Check data (validation).
- Protect data in transit (HTTPS).
- Protect data at rest (encryption).
- Make sure only the right people get in (auth).
- Watch what is happening (logging).
- Keep things updated and improving.

DOCKER VS KUBERNETES (PLAIN ENGLISH)

The big idea:

- Docker is about putting your app in a container so it can run the same anywhere.
- Kubernetes is about running LOTS of containers across LOTS of machines and keeping them healthy.

Why people get confused:

- People say 'we run Docker' and 'we run Kubernetes' like it's the same thing.
- They are related, but they are not the same job.

1. DOCKER

Plain meaning:

- Docker lets you package your app together with everything it needs so it does not break when you move it.
- It can run on your laptop, on a server, or in the cloud the same way.

How Docker works (step by step):

- You write a Dockerfile. This says what your app needs (code, libraries, versions).
- You build an image. The image is like a 'ready-to-run kit' for your app.
- You run that image as a container. The container is the app actually running.
- Networking connects containers to each other and to the outside world.

Result:

- Your app runs the same everywhere.
- No more 'it works on my machine but not on yours' problems.

In plain English:

- Docker = put the whole app in a little box so you can run that box anywhere.

2. KUBERNETES

Plain meaning:

- Kubernetes is for when you have MANY containers and MANY machines.
- It keeps them running, fixes problems, and scales them up or down.

Why you need it:

- One or two containers? Docker alone is fine.
- Hundreds of containers across lots of servers? You need a manager → Kubernetes.

How Kubernetes works (step by step):

- You start with the same Docker image you already built.
- A control plane (API server, etcd, controller manager, scheduler) decides what should run where.
- Worker nodes are the machines that actually run your containers.
- Pods are the small units that hold one or more containers.
- Kubelet (on each worker) keeps the Pods running.
- kube-proxy helps with networking so things can talk to each other.
- If something crashes, Kubernetes can restart it automatically.
- If you need more copies because of high traffic, Kubernetes can make more.

In plain English:

- Kubernetes = the boss that runs and watches many containers on many computers.

3. DOCKER VS KUBERNETES (QUICK VIEW)

Docker does this:

- Builds the container image.
- Runs a container on a single machine (like your laptop or one server).

Kubernetes does this:

- Runs lots of containers across lots of machines.
- Scales them up and down.
- Heals them if they crash.
- Keeps everything online for users.

Super simple memory trick:

- Docker = one box.
- Kubernetes = a warehouse full of boxes, with robots keeping it organised.

APACHE KAFKA: HOW IT WORKS (PLAIN ENGLISH)

Big picture:

- Kafka is a system for sending A LOT of messages in real time.
- Apps can SEND messages into Kafka (they are called producers).
- Other apps can READ those messages from Kafka (they are called consumers).
- This lets you handle things like clicks, payments, sensor data, logs, etc. very fast.

Why people use Kafka:

- It can handle millions of events per second.
- It does not crash easily because it runs across many machines.
- Many different systems can listen to the same data stream.

1. PRODUCER CREATES EVENTS

Plain meaning:

- A producer is something that sends data into Kafka.

Example:

- Your website sends 'user clicked buy button'.
- Your payment system sends 'payment success'.
- An IoT device (sensor) sends 'temperature is 22.4C'.

In plain English:

- Producer = the talker.

2. SERIALIZATION

Plain meaning:

- Before sending, the data is turned into bytes (raw computer format).

Why it matters:

- Kafka does not care about objects or Python classes or whatever.
- It only moves bytes. Bytes are fast to send and store.

In plain English:

- Turn the message into a clean, standard package so Kafka can carry it.

3. PARTITIONING

Plain meaning:

- Kafka groups messages by topic (like a named channel).
- Each topic is split into smaller pieces called partitions.

Why it matters:

- Different partitions can be handled at the same time, by different consumers.
- This makes it scale — more partitions = more speed.

In plain English:

- Think of a topic like 'orders', and partitions are like lanes of traffic for those orders.

4. PUBLISHING TO THE KAFKA CLUSTER

Plain meaning:

- Kafka does not run on just one machine.
- It runs on many servers working together. This is called a cluster.

Details:

- Each server in the cluster is called a broker.
- Producers send messages to the brokers in the cluster.

In plain English:

- You don't send data to one box. You send it to a whole team of boxes.

5. STORAGE & REPLICATION

Plain meaning:

- Brokers save the data from each partition onto disk (they store it).

Replication:

- Kafka makes copies of the data on other brokers.
- So if one machine dies, your data is still safe on another machine.

In plain English:

- Important messages are backed up in more than one place so you don't lose them.

6. ORDERED STORAGE

Plain meaning:

- Inside each partition, messages are stored in order (1, then 2, then 3...).

Why it matters:

- A consumer that reads that partition will see messages in the same order they were written.

In plain English:

- For one lane of traffic, you always know who came first, second, third.

7. CONSUMER GROUPS

Plain meaning:

- A consumer is something that reads data from Kafka.
- A consumer group is a team of consumers working together.

Why it matters:

- Each consumer in the group takes some of the partitions.
- They share the work so no one machine is overloaded.

In plain English:

- Consumer group = the listeners team.

8. PARALLEL CONSUMPTION

Plain meaning:

- Different consumers can read different partitions at the same time.

Why it matters:

- This means you can process huge amounts of data very fast.
- Work is split across many consumers in parallel.

In plain English:

- Many people reading many lanes at once, instead of one person reading one lane slowly.

9. REAL-TIME PROCESSING

Plain meaning:

- As soon as data arrives, something can act on it right away.

Examples:

- Live dashboards update right now.
- Fraud alerts trigger right now.
- Microservices react right now (for example: send email, update stock, flag suspicious payments).

In plain English:

- Kafka lets systems know 'this just happened' in real time, not later.

FAST RECAP

Remember these roles:

- Producer = sends messages in.
- Kafka cluster (brokers) = stores and shares the messages safely.
- Consumer group = reads the messages out.
- Everything is ordered in partitions and can run in parallel for speed.

JWT (JSON WEB TOKEN): PLAIN ENGLISH

Big picture:

- A JWT is a small string (a token) that proves who you are.
- Apps use it so they know: 'Yes, this is the right user.'
- It is used for login / access / permissions.

Why people use JWT:

- It can carry user info safely between systems.
- The token is signed so you can tell if someone tried to change it.
- It is common in modern web apps and APIs.

WHAT IS A JWT?

Plain meaning:

- A JWT is a compact, URL-safe token. (So you can send it in headers, links, etc.)
- It contains trusted information about the user or the session.
- The server can read it and know who you are and what you are allowed to do.

Important idea:

- A JWT has 3 parts glued together with dots:
- header.payload.signature

PART 1. HEADER

What it is:

- This tells you HOW the token was signed.
- Example: HS256 or RS256. (This is the algorithm.)
- It also says 'this is a JWT'.
- The header is JSON, then it is Base64-encoded (turned into safe text).

In plain English:

- Header = 'How was this token made and signed?'

PART 2. PAYLOAD

What it is:

- The payload has the 'claims'.
- A claim is a fact, like 'user id is 123', 'role is admin', 'token expires at 10:30', etc.

Types of claims:

- Registered claims: standard fields like 'iss' (who made the token) and 'exp' (when it expires).
- Public claims: custom but agreed/shared across systems.
- Private claims: app-specific data that only your app cares about.

In plain English:

- Payload = 'Who is this user, and what are they allowed to do?'

PART 3. SIGNATURE

What it is:

- The signature proves the token was not changed.
- It is made by taking:
-   - the header,
-   - the payload,
- and signing them with a secret or private key.

Why it matters:

- If someone edits the payload (for example, tries to make themselves admin), the signature will no longer match.
- So the server will reject it.

In plain English:

- Signature = 'Is this real, or did someone mess with it?'

HOW JWTs GET SIGNED

1. SYMMETRIC SIGNATURE (HMAC):

- One shared secret key is used to sign AND to check the token.
- All services that need to check the token must know the same shared secret.
- Easy to set up, but you must protect that secret.

In plain English (symmetric):

- One password is used by both sides.

2. ASYMMETRIC SIGNATURE (RSA / ECDSA):

- You have two keys:
-   - Private key: used to sign the token (keep this secret).
-   - Public key: used to verify the token (this can be shared).
- Great when many different services need to check the token, but you do NOT want to share your private key with them.

In plain English (asymmetric):

- I keep the pen, you get the stamp.
- Only I can write a real signature, but everyone can check if it's real.

WHY USE JWT?

Stateless:

- The server does not need to remember a session for every user.
- The token itself carries the info.

Self-contained:

- All the info the server needs is inside the token (who you are, when it expires, etc.).

Scales well:

- Works well for microservices and big systems where many services need to trust the same login without sharing a central session store.

In plain English:

- JWT lets different parts of your system trust that you are you, without asking the main server every time.

FAST RECAP

Remember:

- JWT = header + payload + signature.
- Header = how it was signed.
- Payload = who you are / what you can do.
- Signature = proof it was not changed.
- Signed with a key so others can trust it.
- Great for login and permission checks across many services.

SSO (SINGLE SIGN-ON): PLAIN ENGLISH

Big picture:

- SSO means: you log in once, and then you can open many apps without logging in again each time.
- Example: You sign in once, and now Gmail, Slack, Drive, etc. all know who you are.

Who is involved:

- Service Provider (SP): the app you want to use (for example, Gmail or Slack).
- Identity Provider (IdP): the place that actually checks your username + password and proves your identity.

Why people use SSO:

- Better user experience (less logging in).
- Better security (one strong login instead of many weak ones).
- Easier to block access if someone leaves the company.

STEP-BY-STEP: FIRST LOGIN TO ONE APP

1. User tries to access the app (Service Provider / SP):

- You open a protected app, like Gmail.
- Gmail is the Service Provider (SP).
- Gmail sees you are not logged in yet.

2. Redirect to the Identity Provider (IdP):

- Gmail says, 'Go ask the Identity Provider (IdP) to prove who you are.'
- Your browser is sent to the IdP login page.
- The IdP is the trusted system that knows logins.

3. Login at the IdP:

- The IdP shows a login form.
- You type your email + password (or use 2FA, etc.).

4. Token issuance (IdP creates a token):

- If your login is correct, the IdP creates a secure token.
- That token includes who you are and what you are allowed to do.
- The IdP also creates a session so it remembers you for next time.
- The token is sent back to your browser, and then sent on to Gmail.

5. Token validation at the SP:

- Gmail checks the token.
- It checks that the token is valid and really came from the trusted IdP.
- If it's valid, Gmail says: 'Access granted □' and logs you in.

NOW YOU OPEN ANOTHER APP (E.G. SLACK)

6. New app, same IdP:

- You now go to Slack.
- Slack is also a Service Provider (SP).
- Slack sees: 'No local login here yet.'

- Slack sends you to the same IdP to check who you are.

7. No need to log in again:

- The IdP still remembers you (you already have a session there).
- So the IdP does NOT ask for your password again.
- The IdP just creates a new token, this time for Slack.
- The browser sends that token to Slack.

8. Token validation at Slack:

- Slack checks the token the same way Gmail did.
- Slack says: 'Token is trusted and came from the IdP.'
- Slack lets you in instantly, without another login.

FAST RECAP

Main ideas:

- SP (Service Provider) = the app you want to use (Gmail, Slack, etc.).
- IdP (Identity Provider) = the place that actually logs you in and proves who you are.
- The IdP gives the SP a token that says, 'This user is real. Let them in.'

Why SSO feels 'magic':

- You only type your password once at the IdP.
- After that, every other app just asks the IdP for a token.
- So you keep getting in automatically.

In plain English:

- Log in once → trusted everywhere in that group of apps.

OAUTH 2.0: PLAIN ENGLISH

Big picture:

- OAuth 2.0 lets an app get access to your data on another service WITHOUT learning your password.
- Example: 'Sign in with Google' or 'Connect with GitHub'.
- You say what the app is allowed to see or do.

Why this matters:

- Your password stays with Google / Facebook / GitHub.
- The app only gets a token with limited permissions.
- This is safer than giving your password to every app.

STEP-BY-STEP FLOW (AUTHORIZATION CODE GRANT)

1. Client requests access:

- In your app you click 'Connect with Facebook' (or Google, GitHub, etc.).
- Your app is called the Client.

2. Redirect to the Authorization Server:

- Your app sends you (the user) to the provider's login + consent page over HTTPS.
- This page is controlled by the provider (for example, Facebook).
- The Authorization Server is the part that handles login and permission asking.

3. User grants or denies permission:

- You log in there (not in the Client app).
- You see a screen like: 'This app wants your email and friends list.'
- You choose Allow or Deny.

4. Authorization code issued:

- If you click Allow, the provider sends you BACK to the Client app.
- It includes a short-lived code (the 'authorization code').
- This code proves: 'The user said yes.'

5. Exchange code for access token:

- Now the Client's backend talks directly to the Authorization Server.
- It sends the code + its own secret (client credentials).
- In return it gets an access token, and sometimes a refresh token.
- This step is server-to-server, not seen by the user.

6. Access protected resources:

- The Client now uses the access token to ask the Resource Server for data.
- Example: 'Give me this user's profile info' or 'friends list'.
- The Resource Server is the thing that actually stores the data.

TOKEN EXPIRATION & REFRESH

How tokens behave:

- Access tokens expire. This limits damage if someone steals one.
- If the Client also got a refresh token, it can ask for a new access token later.

- This can happen quietly in the background without asking the user to log in again.

In plain English:

- Access token = short-term pass.
- Refresh token = way to get a new pass without bugging the user.

KEY PLAYERS (ROLES IN OAUTH 2.0)

Client:

- The app that wants access (for example, 'My Cool App').

Resource Owner:

- The user. You own the data.

Authorization Server:

- The system that logs you in and issues tokens (for example, accounts.google.com).

Resource Server:

- The API that holds the data you are trying to access (for example, Google Contacts API).

In plain English:

- User says yes → Authorization Server gives token → Client uses token to call Resource Server.

GRANT TYPES (COMMON FLOWS)

Authorization Code:

- Used by web apps.
- Most secure standard flow.
- The token is given to the server, not directly to the browser.

Implicit:

- Used by browser-only apps in the past.
- Now mostly discouraged because tokens were exposed directly in the browser URL.

Client Credentials:

- Used for machine-to-machine (no user).
- One service talks to another service using its own credentials.

Resource Owner Password:

- User gives their username/password directly to the Client.
- ONLY used when nothing else works. Not recommended today.

WHY OAUTH 2.0?

Secure:

- The app never learns your real password for Google / Facebook / etc.

Granular (not all or nothing):

- The app can ask only for what it needs using scopes.

- Example: 'Can I read your email address?' not 'Can I control your whole account?'

Flexible:
- Works for web apps, mobile apps, and server-to-server systems.

In plain English:
- OAuth 2.0 = safe permission sharing without giving away your password.

FAST RECAP

Remember this flow:
- 1. User clicks 'Connect with X'.
- 2. User is sent to X to log in and approve.
- 3. X sends back a code.
- 4. The app swaps that code for a token.
- 5. The app uses the token to access allowed data.
- 6. Tokens expire, and refresh tokens can get new ones.

AUTHENTICATION METHODS: PLAIN ENGLISH

Big picture:

- Authentication = proving who you are when you log in.
- Different systems do this in different ways.
- Some are old-school (sessions), some are modern (tokens, JWT), some are super smooth (SSO, OAuth 2.0, QR).

Why this matters:

- Good auth should be secure, fast, and easy for the user.
- You should not have to type your password 10 times a day.
- You should not leak your password to apps that should not have it.

1. SESSION-BASED AUTH

Plain meaning:

- You log in once. The server creates a session for you and keeps your info on the server side.

How it works:

- The server stores 'this user is logged in' in memory or a database.
- The server gives your browser a session ID (often in a cookie).
- Your browser sends that ID with every request.
- The server looks up the ID and knows it's you.

Good for:

- Classic web apps (traditional websites).

Not so good for:

- Lots of devices / mobile apps / microservices / modern APIs that want to be stateless.

In plain English:

- The server remembers you. You just carry a claim ticket (session ID).

2. TOKEN-BASED AUTH

Plain meaning:

- Instead of the server keeping all the login info, it gives YOU a token.
- You send that token with each request to prove who you are.

How it works:

- You log in. The server gives you a token.
- That token says who you are (encoded inside it).
- For the next requests, you send that token instead of logging in again.

Why people like it:

- The server does not have to keep a session for every user.
- Works well for APIs and mobile apps.

Watch out:

- Tokens must be stored safely.

- They should expire, so stolen tokens do not work forever.

In plain English:

- You carry the proof with you. The server checks it each time.

3. JWT (JSON WEB TOKEN)

Plain meaning:

- JWT is a specific kind of token format used a lot in modern APIs.

Key idea:

- The JWT is signed (has a digital signature).
- That means the server can check if it was changed or faked.

How it's different:

- The JWT itself contains claims like user id, role, and expiry time.
- Because of the signature, the server can trust the token WITHOUT looking in a database first.

Why people like it:

- Fast, stateless, used in systems like Auth0 and Firebase.
- Good for microservices because different services can all verify the same token.

In plain English:

- JWT = a signed ID card you carry. Anyone can check it, nobody can edit it.

4. SSO (SINGLE SIGN-ON)

Plain meaning:

- You log in once, and then you can get into many apps without typing your password again.

Who is involved:

- There is an Identity Provider (IdP) like Google, Okta, etc.
- Apps like Gmail, Slack, Notion etc. trust that IdP.

How it feels to the user:

- You sign in once with the IdP.
- Then other apps say: 'Oh, the IdP has already confirmed you. Come in.'

Why people like it:

- Less password typing.
- Easy to manage access for employees in a company.
- If someone leaves the company, turning off the IdP blocks all apps in one go.

In plain English:

- One login to rule them all.

5. OAUTH 2.0

Plain meaning:

- OAuth 2.0 lets an app get access to some part of your account on another service WITHOUT knowing your password.

Example:

- You click 'Login with Google' or 'Sign in with GitHub'.
- Google / GitHub asks 'Is it OK if this app can read your email address?'
- You say yes (or no).

Why it's good:

- The app never sees your real password.
- You can choose what access to allow (read-only, email only, etc.).

In plain English:

- You say: 'Yes, this app can see THIS part of my data. But it does not get my password.'

6. QR CODE LOGIN

Plain meaning:

- Log in by scanning a QR code with your phone instead of typing a password on the device you're using.

How it works:

- The computer screen shows a QR code that includes a random one-time token.
- You scan it with your phone (where you're already logged in).
- Your phone tells the server: 'Yes, this is me. Approve that device.'
- The other device is now logged in without typing password on it.

Why it's nice:

- Great for public computers, TV apps, consoles, kiosks.
- No typing password on an unsafe keyboard.

In plain English:

- Your phone vouches for you. The QR is just the link between them.

FAST RECAP

Session-based auth:

- Server keeps your login state. You get a session ID cookie. Best for classic web.

Token-based auth:

- You carry a token. Server checks it each time. Good for APIs/mobile.

JWT:

- Signed token with user info inside. Easy to verify. No DB lookup needed.

SSO:

- Log in once, use many apps. Company-style, less password pain.

OAuth 2.0:

- Let one app access certain data from another app without sharing your password.

QR code login:

- Use your phone to log in on another device with no password typing.

SOFTWARE DESIGN PATTERNS: PLAIN ENGLISH

What this is about:

- Design patterns are common ways to solve common coding problems.
- They are not code you copy and paste.
- They are ideas / shapes you can use in your own code.

Why they matter:

- They help keep code clean and easier to change later.
- They help reduce bugs and mess.
- They help teams write code in the same 'language'.

Simple goal:

- Write code you are not ashamed of in 6 months.

1. FACTORY PATTERN

Plain meaning:

- A Factory creates objects for you.
- You ask the Factory for 'a thing', and it gives you the right kind of object.

Why it helps:

- You do not have to say 'new ThisClass' or 'new ThatClass' everywhere.
- Your code becomes easier to change and extend.

Memory trick:

- Factory = a factory that makes different products on demand.

2. OBSERVER PATTERN

Plain meaning:

- One object is the 'main source' (subject).
- Other objects 'subscribe' to it (observers).
- When the subject changes, all observers get notified.

Why it helps:

- Great for events, updates, live data (like UI refreshing when data changes).

Memory trick:

- Like subscribing to a YouTube channel. When they post, you get notified.

3. SINGLETON PATTERN

Plain meaning:

- Singleton makes sure there is only ONE object of a certain type in the whole app.

Why it helps:

- Good for shared things like: database connection, config, logger.
- Everyone uses the same single instance.

Memory trick:

- Singleton = 'the one and only'.

4. BUILDER PATTERN

Plain meaning:

- Builder helps you build a complicated object step by step.

Why it helps:

- You do not have to pass 20 things into one giant constructor.
- You can create different versions (with/without certain parts) in a clean way.

Memory trick:

- Builder = like building with LEGO bricks, one piece at a time.

5. ADAPTER PATTERN

Plain meaning:

- Adapter lets two things talk to each other even if they do not match.
- It converts one interface into another interface that the caller expects.

Why it helps:

- You can plug in old code or third-party code without rewriting everything.

Memory trick:

- Adapter = phone charger plug adapter. UK plug ↔ EU socket.

6. DECORATOR PATTERN

Plain meaning:

- Decorator lets you add extra behaviour to an object without changing the original class.

Why it helps:

- You do not need to make lots of child classes just to add small features.
- You can 'wrap' the object and give it more powers.

Memory trick:

- Decorator = adding toppings to a pizza. Same pizza base, more extras.

7. PROXY PATTERN

Plain meaning:

- Proxy is a stand-in / middle person for a real object.

Why it helps:

- You can control access (check permission first).
- You can delay loading something heavy until you really need it (lazy loading).
- You can cache results.

Memory trick:

- Proxy = personal assistant who takes calls for the boss.

8. STRATEGY PATTERN

Plain meaning:

- Strategy lets you pick which algorithm to use at run time.

Why it helps:

- You can swap in different behaviour without rewriting the whole function.
- Good when there are many 'ways to do it'.

Memory trick:

- Strategy = different ways to sort, search, pay, route, etc.

9. COMMAND PATTERN

Plain meaning:

- Command wraps an action/request as its own object.

Why it helps:

- You can save it, queue it, log it, undo it, redo it later.
- Great for undo/redo systems, task queues, job runners.

Memory trick:

- Command = a to-do item your program can run later.

10. TEMPLATE PATTERN

Plain meaning:

- Template defines the general steps of an algorithm.
- Some steps are fixed, some steps can be customised by child classes.

Why it helps:

- You get shared structure and shared logic, but still allow custom behaviour in certain parts.

Memory trick:

- Template = a workflow with fill-in-the-blanks steps.

11. ITERATOR PATTERN

Plain meaning:

- Iterator lets you loop through items in a collection without caring how the collection is built inside.

Why it helps:

- You get 'next item, next item, next item' in a safe way.
- You don't expose private data structure details.

Memory trick:

- Iterator = tour guide walking you through a museum room by room.

12. STATE PATTERN

Plain meaning:

- State lets an object change how it behaves based on its current state.

Why it helps:

- You avoid giant 'if...else...if...else' chains everywhere.
- Each state has its own logic, which keeps code clean.

Memory trick:

- State = traffic light. Red, amber, green all behave differently.

FAST RECAP

Easy grouping:

- Creation patterns: Factory, Builder, Singleton (how to create objects).
- Structure patterns: Adapter, Decorator, Proxy (how objects are wrapped/connected).
- Behaviour patterns: Strategy, Command, Template, Observer, Iterator, State (how objects act and change).

Main goal:

- Make code easier to extend, test, and maintain over time.