Công ty cổ phần VCCorp

BÁO CÁO TUẦN 4

Tìm hiểu về k8s, jwt, oauth, sso, basic auth, session auth và graphql

Tác giả: Ngô Minh Đức

Người hướng dẫn: Anh Ngô Văn Vĩ

Hà Hội, 7/2025

Tóm tắt nội dung

Báo cáo này trình bày những nội dung quan trọng về khái niệm, đăc điểm, cấu trúc, các phương pháp và ứng dụng của k8s, jwt, oauth, sso, basic auth, session auth và graphql

MỤC LỤC

Chương 1: Kunbernetes – K8s

1.1 Giới thiệu về Kunbernetes

- Kubernetes (viết tắt là K8s) là một nền tảng mã nguồn mở (open-source platform) được phát triển bởi Google và hiện được quản lý bởi Cloud Native Computing Foundation (CNCF). Mục tiêu chính của Kubernetes là tự động hóa việc triển khai, quản lý, mở rộng và vận hành các ứng dụng container hóa — chẳng hạn như các ứng dụng được đóng gói bằng Docker.
- Kubernetes cung cấp một cơ chế mạnh mẽ để:
    - Triển khai và quản lý các container theo cụm (cluster).
    - Tự động phân bổ tài nguyên, đảm bảo hiệu suất tối ưu.
    - Giám sát trạng thái hoạt động của container và tự động thay thế nếu cần.
    - Tự động mở rộng hoặc thu hẹp quy mô hệ thống tùy theo lưu lượng thực tế.

- Triển khai liên tục (rolling updates) và rollback dễ dàng nếu có sự cố.

- Kubernetes mang lại rất nhiều lợi ích thiết thực trong việc vận hành các hệ thống hiện đại dựa trên microservices và container. Dưới đây là những lý do chính nên sử dụng Kubernetes:

- Tự động hóa triển khai và quản lý container: Kubernetes cho phép bạn triển khai các container một cách tự động thông qua file cấu hình YAML (Deployment, Pod, Service, v.v.). Các container sẽ được tự động phân phối (scheduling) lên các node còn tài nguyên, tự động restart nếu gặp lỗi hoặc bị crash, tự động thay thế hoặc rollback nếu phiên bản mới hoạt động không ổn định, tự động cập nhật cấu hình hoặc hình ảnh container (image). Ví dụ: Nếu một Pod (đơn vị nhỏ nhất trong Kubernetes) bị crash, Kubernetes sẽ tự động tạo lại Pod đó để duy trì trạng thái ổn định cho hệ thống.
- Khả năng mở rộng linh hoạt: Kubernetes hỗ trợ scale theo chiều ngang (horizontal scaling) một cách dễ dàng và nhanh chóng. Nó Có thể scale thủ công (bằng lệnh kubectl scale) hoặc tự động thông qua Horizontal Pod Autoscaler (HPA), dựa trên các chỉ số như CPU, RAM, hoặc custom metrics và dễ dàng mở rộng số lượng bản sao của ứng dụng khi có nhiều traffic, và thu gọn lại khi traffic giảm cho nên giúp tiết kiệm chi phí tài nguyên và đảm bảo hiệu suất hệ thống được tối ưu hóa.
- Đảm bảo tính khả dụng cao: Kubernetes được thiết kế để duy trì độ sẵn sàng của hệ thống (uptime) ở mức cao. Nó tự động phân phối workload đến các node hoạt động tốt; chạy nhiều bản sao (replicas) của ứng dụng để dự phòng nếu một instance gặp sự cố; tái cân bằng tải (load balancing) giữa các Pod một cách tự động; hỗ trợ rolling update và zero-downtime deployment để cập nhật ứng dụng mà không gây gián đoạn dịch vụ cho nên hệ thống có thể tiếp tục hoạt động ổn định ngay cả khi một phần hạ tầng gặp sự cố
- Tương thích và triển khai linh hoạt trên nhiều môi trường: Kubernetes có thể chạy gần như ở mọi nơi như trên máy cá nhân sử dụng Minikube, Kind hoặc Docker Desktop (dùng cho mục đích phát triển); trên máy chủ vật lý (on-premise) với các cụm K8s riêng; trên đám mây (cloud, Google Kubernetes Engine, Amazon Elastic Kubernetes Service, Azure Kubernetes Service) cho nên giúp dễ dàng di chuyển workload giữa các môi trường (hybrid cloud, multi-cloud) mà không phải thay đổi nhiều trong cấu hình ứng dụng.

- Hệ sinh thái mạnh mẽ và mở rộng: Kubernetes không chỉ là một công cụ điều phối container mà còn là trung tâm của một hệ sinh thái đa dạng gồm nhiều công cụ hỗ trợ như trình quản lý package cho Kubernetes (Helm); giám sát hệ thống và vẽ biểu đồ metrics (Prometheus + Grafana); Service Mesh dùng để điều phối lưu lượng nội bộ (Istio / Linkerd); triển khai GitOps CI/CD tự động (ArgoCD / FluxCD)

1.2 Kiến trúc Kunbernetes

- Cluster: một Cluster Kubernetes là một tập hợp các máy (node) làm việc cùng nhau để chạy các ứng dụng container. Mỗi cluster bao gồm:
    - Control Plane (Master Node): Chịu trách nhiệm điều khiển và quản lý toàn bộ hệ thống.
    - Worker Nodes: Thực thi và vận hành các ứng dụng thực tế (Pod/Container).

  Các thành phần trong cluster giao tiếp với nhau qua API, và toàn bộ hệ thống có thể được vận hành thông qua công cụ dòng lệnh kubectl.
- Node: một máy tính vật lý hoặc máy ảo trong cluster Kubernetes. Mỗi node đều có:
    - Kubelet: Quản lý và thực thi các container trong node.
    - Kube-proxy: Điều phối mạng, forwarding request.
    - Container Runtime: Ví dụ Docker, containerd, CRI-O.

  Có 2 loại node chính:

    - Master Node (Control Plane): Đây là trung tâm điều khiển (brain) của cluster, chịu trách nhiệm quản lý và điều phối mọi hoạt động nhưng không trực tiếp chạy ứng dụng người dùng, bao gồm các thành phần:

Thành phần	**Mô tả**
API Server	Cổng giao tiếp chính giữa user và cluster (qua kubectl, dashboard hoặc CI/CD).
Scheduler	Gán các Pod đến node phù hợp dựa trên tài nguyên và policy.

Controller Manager	Theo dõi trạng thái cluster và đảm bảo trạng thái mong muốn (desired state).
etcd	Cơ sở dữ liệu key-value phân tán, lưu trữ toàn bộ cấu hình và trạng thái của cluster.

- Worker Node: Đây là nơi thực sự chạy các ứng dụng container hóa của người dùng. Mỗi worker node có:

Thành phần	Vai trò
Kubelet	Agent chạy trên node, nhận lệnh từ API Server và thực thi container.
Kube-proxy	Xử lý traffic và load balancing giữa các Pod trong node.
Container Runtime	Chạy container thực tế (ví dụ: Docker, containerd, CRI-O).

- Pod: đơn vị nhỏ nhất có thể triển khai được trong Kubernetes. Mỗi Pod có thể chứa một hoặc nhiều container, chia sẻ network, storage và namespace. Các container trong một Pod luôn được triển khai cùng nhau, trên cùng một node. Ví dụ: một Pod có thể chứa 1 container ứng dụng chính và 1 container sidecar (như logging hoặc monitoring). Kubernetes sẽ quản lý vòng đời của Pod và thay thế khi có lỗi.
- **Deployment**: là một đối tượng trong Kubernetes dùng để khai báo số lượng **replica (bản sao)** của ứng dụng, quản lý quá trình **rolling update, rollback** và đảm bảo luôn có số lượng Pod cần thiết chạy tại mọi thời điểm.
  Cấu hình Deployment thường được khai báo bằng YAML, ví dụ:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
```

```
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app-container
          image: my-app:1.0
```

- Service: là một lớp trừu tượng giúp expose các Pod ra ngoài để các thành phần bên trong hoặc bên ngoài cluster có thể truy cập được. Dù Pod có thể thay đổi IP mỗi khi restart, Service đảm bảo một endpoint cố định giúp hệ thống hoạt động ổn định. Các loại service chính:

Loại Service	Mô tả
ClusterIP	Mặc định, chỉ truy cập được bên trong cluster.
NodePort	Mở một port trên mỗi node để truy cập từ bên ngoài.
LoadBalancer	Sử dụng Load Balancer của Cloud Provider để expose service.
ExternalName	Trỏ đến một DNS name bên ngoài cluster.

- Mối quan hệ giữa các thành phần trong kiến trúc Kubernetes được tổ chức chặt chẽ theo luồng điều phối từ người dùng đến ứng dụng. Khi người dùng gửi yêu cầu thông qua kubectl hoặc API, yêu cầu này sẽ được chuyển đến API Server, trung tâm giao tiếp chính của hệ thống. Tại đây, Control Plane tiếp nhận và xử lý thông tin, sau đó Scheduler sẽ lựa chọn node phù hợp để triển khai Pod dựa trên tài nguyên sẵn có. Tiếp theo, Controller Manager đảm bảo trạng thái mong muốn bằng cách tạo Pod và gửi chúng đến Worker

Node. Trên node, Kubelet nhận nhiệm vụ khởi chạy và giám sát container bên trong Pod. Cuối cùng, Service đảm nhận vai trò expose các Pod, cung cấp địa chỉ truy cập ổn định để người dùng hoặc các hệ thống khác có thể truy cập ứng dụng, bất kể Pod thay đổi IP hay bị thay thế.

1.3 Thao tác cơ bản

- Tạo Cluster Kubernetes: Minikube là công cụ đơn giản để tạo Kubernetes cluster ngay trên máy local. Nó mô phỏng một cluster gồm một node, rất thích hợp để học và phát triển. Yêu cầu có: VirtualBox (Docker hoặc Hypervisor) và CLI để giao tiếp với Kubernetes (kubectl) với lệnh cài đặt:
  # Trên macOS
  brew install minikube
  # Trên Ubuntu
  curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
  sudo install minikube-linux-amd64 /usr/local/bin/minikube
  Sau khi cài xong, bạn có thể kiểm tra bằng lệnh:
  minikube version
  Minikube sẽ tạo một máy ảo để khởi chạy Kubernetes master & node đồng thời thiết lập kubectl để liên kết với cluster bằng lệnh:
  minikube start
  Sau khi hoàn tất, bạn có thể kiểm tra node bằng:
  kubectl get nodes
- Triển khai ứng dụng: để tạo một Deployment cần dùng lệnh sau để triển khai một ứng dụng mẫu, ví dụ tạo một Deployment với tên hello-node, chạy container từ image mẫu echoserver:1.4:
  kubectl create deployment hello-node --image=k8s.gcr.io/echoserver:1.4
  Sau đó kiểm tra trạng thái: Các lệnh này giúp bạn xem danh sách các deployment và pod hiện đang chạy trong cluster.
  kubectl get deployments
  kubectl get pods
- Expose Ứng dụng ra bên ngoài: Để truy cập ứng dụng từ trình duyệt, bạn cần expose nó qua một Service dùng lệnh:
  kubectl expose deployment hello-node --type=LoadBalancer --port=8080
  Mở trình duyệt mở rộng: mở ứng dụng tại địa chỉ public do Minikube tạo ra:

minikube service hello-node

- Mở rộng Ứng dụng: tăng số lượng Pod để xử lý nhiều request hơn bằng cách scale deployment:
  kubectl scale deployment hello-node --replicas=3
  Kiểm tra lại số lượng Pod đang chạy:
  kubectl get pods
- Cập nhật Ứng dụng: cập nhật phiên bản image của ứng dụng, sử dụng lệnh sau:
  kubectl set image deployment/hello-node hello-node hello-node=k8s.gcr.io/echoserver:1.10
  Kubernetes sẽ tự động thực hiện rolling update để đảm bảo không gây downtime, để kiểm tra tiến trình update bằng:
  kubectl rollout status deployment/hello-node
- Xóa Cluster: xóa cluster và giải phóng tài nguyên bằng:
  minikube delete hoặc minikube stop

1.4 Cấu hình trong Kubernetes

- ConfigMap là một đối tượng trong Kubernetes dùng để lưu trữ dữ liệu cấu hình không nhạy cảm dưới dạng key-value. Nó giúp tách biệt cấu hình khỏi ứng dụng, cho phép thay đổi cấu hình mà không cần rebuild image, tái sử dụng cấu hình giữa nhiều Pod và cập nhật cấu hình chỉ bằng YAML hoặc lệnh CLI. Có thể tạo ConfigMap từ một file, một biến môi trường, hoặc trực tiếp từ lệnh*: kubectl create configmap my-config --from-literal=APP_ENV =production --from-literal=PORT=8080* và gắn ConfigMap vào Pod bằng 2 cách thông qua biến môi trường hoặc mount vào volume.
- Cập nhật cấu hình qua ConfigMap: thay đổi ConfigMap bằng lệnh hoặc YAML. Nếu Pod đang sử dụng ConfigMap qua volume, bạn cần restart Pod để cập nhật và để cập nhật rolling nên dùng lệnh*: kubectl rollout restart deployment <tên-deployment>*.

- Sidecar container: là một mô hình triển khai trong đó một Pod chứa nhiều container và các container này hỗ trợ lẫn nhau. Thay vì nhồi toàn bộ chức năng vào một container, bạn có thể chia nhỏ chức năng thành các container riêng biệt nhưng hoạt động cùng một Pod.

1.5 Bảo mật trong Kunbernetes

- Hệ thống bảo mật trong Kubernetes bao gồm các lớp kiểm soát sau:

Lớp bảo vệ	Mục tiêu chính
Cluster-level security	Bảo vệ toàn bộ cụm (node, control plane)
Namespace-level security	Phân vùng bảo mật giữa các môi trường
Pod-level security	Giới hạn hành vi của Pod, container
Container-level security	Hạn chế tài nguyên và quyền truy cập của container

- Pod Security Standards (PSS) là các chính sách định nghĩa những gì Pod được và không được phép làm trong cluster. Có ba cấp độ chính:

Cấp độ	Mô tả
Privileged	Không hạn chế – phù hợp môi trường test, dev.
Baseline	Áp dụng các chính sách cơ bản – phù hợp production.
Restricted	Bảo mật nghiêm ngặt nhất – áp dụng cho môi trường cần độ an toàn cao.

  Khi áp dụng PSS ở cấp độ cluster, tất cả các namespace mới tạo ra sẽ kế thừa chính sách này, trừ khi được ghi đè. PSS ở cấp độ Namespace giúp phân chia môi trường phát triển (dev), kiểm thử (staging), và sản xuất (prod) với các mức độ bảo mật khác nhau.
- AppArmor: là một công cụ bảo mật kernel-level trên Linux, cho phép bạn xác định những hành vi nào của container là hợp lệ, như: những file nào có thể đọc/ghi, những process nào có thể chạy, những network nào được phép truy cập
- Seccomp (Secure Computing Mode) cho phép giới hạn hệ thống lời gọi (syscalls) mà container được phép sử dụng, từ đó ngăn chặn các hành vi nguy hiểm ở cấp độ kernel. Seccomp thường được dùng để giảm thiểu bề mặt tấn công cho container, đặc biệt với các ứng dụng có quyền truy cập sâu vào hệ thống.

Chương 2: JSON Web Token – JWT

2.1 Giới thiệu về JWT

Trong các hệ thống web hiện đại, vấn đề xác thực và ủy quyền (authentication & authorization) là vô cùng quan trọng để đảm bảo an toàn cho hệ thống cũng như dữ liệu của người dùng. Một trong những cơ chế xác thực phổ biến, nhẹ và hiệu quả hiện nay là sử dụng JSON Web Token (JWT).JWT là một tiêu chuẩn mã hóa mở (RFC 7519) dùng để truyền tải thông tin xác thực giữa các bên một cách an toàn và không cần trạng thái (stateless). JWT (JSON Web Token) là một chuỗi ký tự mã hóa nhỏ gọn, được sử dụng để trao đổi thông tin giữa các hệ thống một cách an toàn. JWT thường được sử dụng trong quá trình đăng nhập và bảo mật API. Điểm đặc biệt của JWT là nó không yêu cầu lưu trạng thái (stateless) trên server. Điều này rất hữu ích với các hệ thống phân tán hoặc microservices.

2.2 Cấu trúc của JWT

- Một JWT gồm 3 phần, được nối với nhau bằng dấu chấm (.): <Header>.<Payload>.<Signature>
- Header: thường chứa 2 thông tin:
    - alg: Thuật toán ký (ví dụ: HS256, RS256)
    - typ: Loại token (mặc định là JWT)

  Sau đó, phần header sẽ được mã hóa bằng Base64Url.
- Payload chứa các thông tin (claims) muốn truyền tải, được mã hóa bằng Base64Url. Có 3 loại claims:
    - Registered claims: Các claim chuẩn như iss (issuer), exp (expiration), sub (subject), aud (audience)…
    - Public claims: Thông tin công khai do người dùng định nghĩa (ví dụ: userId, email, roles…)
    - Private claims: Thông tin riêng giữa các bên (ví dụ: companyId, isPremiumUser…)
- Signature: giúp xác thực token và đảm bảo tính toàn vẹn dữ liệu. Nếu dữ liệu bị chỉnh sửa, signature sẽ không khớp và token bị từ chối.

2.3 Ưu nhược điểm

- Ưu điểm:
    - Stateless: Server không cần lưu phiên làm việc (session).
    - Tương thích đa nền tảng: Dễ dàng sử dụng trong web, mobile, microservices.
    - Bảo mật với chữ ký số: Đảm bảo dữ liệu không bị chỉnh sửa.

    - Truyền tải nhanh: Dạng chuỗi gọn nhẹ, truyền qua HTTP Header dễ dàng.
- Nhược điểm:
    - Không thể thu hồi token (trừ khi lưu blacklist).
    - Token có dung lượng lớn hơn session ID.
    - Nếu token bị lộ thì toàn bộ quyền truy cập bị xâm phạm cho đến khi token hết hạn.

2.4 Ứng dụng

- Xác thực người dùng: Khi người dùng đăng nhập thành công, server tạo JWT chứa thông tin người dùng và gửi về client. Client lưu token (trong localStorage, AsyncStorage…) và gửi kèm theo mỗi request: Authorization: Bearer <JWT>
- Phân quyền API: Dựa trên payload (ví dụ role: admin), backend có thể cấp quyền truy cập các API khác nhau.
- Microservices Communication: Các service có thể xác thực lẫn nhau bằng cách sử dụng JWT mà không cần truy vấn database mỗi lần.

2.5 Một số thuật toán ký

- HS256 – HMAC with SHA-256 (đối xứng): Cả máy chủ ký và xác minh đều sử dụng cùng một chuỗi bí mật (secret key) có tốc độ rất nhanh, hiệu quả cao; dễ dùng, đơn giản, không cần cặp khóa công khai/bí mật nhưng nếu secret key bị lộ, toàn bộ hệ thống xác thực bị phá vỡ. Nó phù hợp cho hệ thống nội bộ, microservices tin cậy lẫn nhau.
- RS256 – RSA SHA-256 (bất đối xứng): dùng Private key để ký và Public key để xác minh có tính an toàn cao hơn HS256 vì public key có thể công khai, không lo bị lộ như secret key nhưng chậm hơn HS256 do thuật toán RSA nặng hơn. Nó phù hợp cho ứng dụng web có front-end/back-end riêng, hệ thống phân tán, SSO (Single Sign-On).
- Mã ví dụ sử dụng JWT trong Node.js + Express:

```
import express from 'express';
import jwt from 'jsonwebtoken';

const app = express();
const PORT = 3000;
```

```
// Dữ liệu gốc (payload)
const payload = { userId: 'abc123', role: 'admin' };

// Đối với HS256
const secretKey = 'yourSecretKey'; // KHÔNG được lộ
const token = jwt.sign(payload, secretKey, { algorithm: 'HS256',
expiresIn: '1h' });

// Đối với RS256
// const fs = require('fs');
// const privateKey = fs.readFileSync('./private.pem');
// const publicKey = fs.readFileSync('./public.pem');
// const token = jwt.sign(payload, privateKey, { algorithm: 'RS256',
expiresIn: '1h' });

app.get('/token', (req, res) => {
  res.send({ token });
});

app.get('/verify', (req, res) => {
  try {
    const decoded = jwt.verify(token, secretKey); // Hoặc publicKey nếu
dùng RS256
    res.send({ decoded });
  } catch (err) {
    res.status(401).send('Invalid token');
  }
});

app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

Chương 3: OAuth

3.1 Tổng quan về OAuth

- OAuth là một giao thức ủy quyền (authorization protocol), cho phép ứng dụng bên thứ ba truy cập vào tài nguyên của người dùng (như email, danh bạ, hình ảnh...) mà không cần biết hoặc lưu trữ mật khẩu. Ví dụ: Một ứng dụng lịch muốn truy cập lịch Google của bạn. Bạn đăng nhập Google và đồng ý cấp quyền. Ứng dụng được phép truy cập mà không cần biết mật khẩu của bạn.
- Thành phần chính trong hệ thống OAuth:

Thành phần	Mô tả
Resource Owner	Người dùng – chủ sở hữu dữ liệu
Client	Ứng dụng bên thứ ba (ví dụ: app lịch, app đọc email)
Authorization Server	Máy chủ xác thực và cấp token
Resource Server	API chứa tài nguyên (ví dụ: Google Drive, GitHub)

3.2 Quy trình hoạt động

- Một quy trình tiêu chuẩn trong OAuth 2.0 là Authorization Code Flow, thường được dùng cho các ứng dụng web hoặc server-side. Trong quy trình này, người dùng truy cập vào ứng dụng và được chuyển hướng đến trang đăng nhập của máy chủ ủy quyền (Authorization Server). Sau khi đăng nhập và đồng ý cấp quyền, người dùng được chuyển ngược lại ứng dụng kèm theo một mã tạm thời gọi là authorization code. Ứng dụng sau đó dùng mã này để yêu cầu access token từ máy chủ ủy quyền. Cuối cùng, access token được dùng để gọi các API trên resource server thay mặt người dùng.
- Khi quyền truy cập được cấp, máy chủ sẽ cung cấp hai loại token:
    - Access Token: là mã truy cập chính, được sử dụng để gọi API và thường có thời gian sống ngắn (thường từ vài phút đến một giờ).
    - Refresh Token: cho phép ứng dụng xin cấp lại access token mới mà không cần người dùng đăng nhập lại, giúp nâng cao trải nghiệm người dùng nhưng cũng cần được bảo vệ kỹ càng.
- OAuth hỗ trợ nhiều loại luồng khác nhau, phù hợp với các loại ứng dụng khác nhau:

- Authorization Code Flow: Phù hợp cho ứng dụng web có backend (bảo mật cao).
- Implicit Flow: Dành cho ứng dụng JavaScript client-side, tuy nhiên đã lỗi thời và không còn được khuyến nghị do vấn đề bảo mật.
- Client Credentials Flow: Dùng cho hệ thống server-to-server, không có người dùng tham gia.
- Resource Owner Password Credentials Flow: Cho phép ứng dụng gửi trực tiếp username/password để lấy token. Tuy nhiên, điều này làm mất ưu điểm "không lưu mật khẩu" nên không nên dùng trừ khi trong hệ thống nội bộ và tin cậy.

3.3 Ứng dụng

OAuth 2.0 được sử dụng rộng rãi trong các dịch vụ hiện đại. Một trong những ứng dụng phổ biến nhất là "Đăng nhập bằng Google/Facebook", nơi người dùng không cần tạo tài khoản mới mà chỉ cần xác nhận cấp quyền cho ứng dụng. Ngoài ra, OAuth 2.0 còn được dùng để cấp quyền cho ứng dụng truy cập dữ liệu riêng tư như danh bạ, email, tài liệu Google Drive, v.v. Nó cũng là nền tảng cho các hệ thống Single Sign-On (SSO), giúp người dùng chỉ cần đăng nhập một lần cho nhiều ứng dụng.

3.4 Bảo mật

Để bảo đảm an toàn khi triển khai OAuth 2.0, cần tuân thủ một số nguyên tắc bảo mật như: luôn sử dụng HTTPS để bảo vệ dữ liệu truyền đi, hạn chế thời gian sống của access token, không lưu trữ token trong những nơi dễ bị lộ (như localStorage), và sử dụng cơ chế PKCE (Proof Key for Code Exchange) trong các ứng dụng client như mobile hoặc SPA. Việc bảo vệ refresh token cũng vô cùng quan trọng, vì nếu bị lộ có thể dẫn đến việc lạm dụng quyền truy cập mà không cần người dùng tương tác.

Chương 4: Single Sign-On (SSO)

4.1 Tổng quan về SSO

- SSO (viết tắt của Single Sign-On) là một cơ chế xác thực cho phép người dùng chỉ cần đăng nhập một lần duy nhất để có thể truy cập nhiều ứng dụng

hoặc hệ thống khác nhau mà không cần phải đăng nhập lại cho từng hệ thống riêng lẻ.

- Khi người dùng đăng nhập vào một ứng dụng hoặc hệ thống trong môi trường có hỗ trợ SSO, thông tin xác thực sẽ được chia sẻ hoặc tái sử dụng giữa các dịch vụ. Điều này giúp tiết kiệm thời gian, giảm rào cản truy cập và nâng cao trải nghiệm người dùng.
- SSO hoạt động dựa trên các giao thức xác thực và ủy quyền tiêu chuẩn như:
    - OAuth: Giao thức ủy quyền, cấp access token cho ứng dụng.
    - OpenID Connect (OIDC): Lớp mở rộng của OAuth 2.0 để xác thực danh tính người dùng.
    - SAML (Security Assertion Markup Language): Giao thức dựa trên XML được sử dụng phổ biến trong các hệ thống doanh nghiệp.
- Các thành phần chính trong hệ thống SSO:

Thành phần	Mô tả
User	Người dùng cần xác thực
Service Provider (SP)	Ứng dụng hoặc hệ thống yêu cầu xác thực
dentity Provider (IdP)	Máy chủ trung tâm xác thực và cấp token
Token/Assertion	Dữ liệu đại diện cho việc xác thực thành công, có thể là JWT, SAML, hoặc ID Token

4.2 Quy trình hoạt động

1. Người dùng truy cập một ứng dụng (gọi là ứng dụng client).
2. Ứng dụng kiểm tra xem người dùng đã đăng nhập chưa. Nếu chưa, nó chuyển hướng người dùng đến Identity Provider (IdP) – máy chủ xác thực trung tâm.
3. Người dùng đăng nhập tại IdP (có thể là Google, Azure AD, hoặc máy chủ nội bộ).
4. Sau khi xác thực thành công, IdP cấp một token hoặc assertion để chứng minh danh tính người dùng.
5. Ứng dụng nhận token và cho phép người dùng truy cập.

6. Trong các lần truy cập sau (dù là ứng dụng khác), nếu phát hiện người dùng đã đăng nhập tại IdP, hệ thống sẽ tự động cấp quyền mà không yêu cầu đăng nhập lại.

4.3 Ưu nhược điểm

- Ưu điểm:
    - Về trải nghiệm người dùng: giảm phiền phức bằng việc không cần nhớ và nhập nhiều mật khẩu; truy cập nhanh chóng, chuyển giữa các dịch vụ dễ dàng mà không bị ngắt quãng.
    - Về bảo mật và quản trị: giảm thiểu tấn công đánh cắp mật khẩu (phishing): người dùng không phải nhập lại mật khẩu nhiều nơi; quản lý truy cập tập trung: dễ kiểm soát người dùng, phân quyền và thu hồi quyền khi cần; tích hợp xác thực đa yếu tố (MFA) dễ dàng hơn tại một điểm duy nhất (IdP).
- Nhược điểm:
    - Điểm lỗi duy nhất (Single Point of Failure): Nếu IdP gặp sự cố hoặc bị tấn công, toàn bộ hệ thống bị ảnh hưởng.
    - Token bị rò rỉ: Nếu token không được bảo vệ đúng cách, hacker có thể dùng để truy cập trái phép.
    - Cần triển khai và cấu hình chính xác: Sai sót trong thiết lập SAML, OIDC, token timeout… có thể gây lỗi hoặc lỗ hổng bảo mật

4.4 Ứng dụng

SSO được triển khai rộng rãi trong cả môi trường doanh nghiệp và các dịch vụ web. Một số ứng dụng thực tế bao gồm:

- Hệ thống nội bộ doanh nghiệp: Nhân viên chỉ cần đăng nhập một lần để sử dụng hệ thống email, quản lý nhân sự, CRM, tài liệu nội bộ,…
- Các hệ thống có nhiều subdomain: Ví dụ, người dùng đăng nhập tại portal.company.com và tự động truy cập chat.company.com, docs.company.com, mail.company.com mà không cần đăng nhập lại.
- Các nền tảng SaaS tích hợp Google, Microsoft: Đăng nhập Google → truy cập mọi dịch vụ có tích hợp Google SSO như Zoom, Slack, Trello...

Chương 5: Basic Auth

5.1 Tổng quan về Basic Auth

Basic Authentication là một phương pháp xác thực HTTP đơn giản, được định nghĩa trong chuẩn HTTP 1.0. Trong phương pháp này, client gửi thông tin xác thực dưới dạng username và password, được mã hóa bằng Base64, trong phần Authorization header của HTTP request. Cú pháp của header thường có dạng:

Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

Trong đó, phần dXNlcm5hbWU6cGFzc3dvcmQ= là chuỗi Base64 của username:password.

5.2 Quy trình hoạt động

1. Client gửi HTTP request với header Authorization: Basic <base64-encoded-credentials>.
2. Server giải mã chuỗi base64: tách thành username và password.
3. Server xác thực thông tin: nếu đúng thì cho phép truy cập, nếu sai thì trả lỗi 401 Unauthorized.
4. Mỗi request tiếp theo đều cần gửi lại thông tin xác thực.

5.3 Hạn chế

- Không an toàn nếu không có HTTPS: Vì Base64 chỉ là mã hóa đơn giản, ai cũng có thể giải mã nếu bắt được gói tin → phải kết hợp với HTTPS để bảo vệ đường truyền.
- Không có cơ chế session hoặc token: Mỗi request đều phải gửi username và password → tăng nguy cơ bị rò rỉ.
- Không có cơ chế logout: Không có cách nào để "hủy" quyền truy cập ngoài cách xoá dữ liệu cache trình duyệt.
- Không scale tốt: Việc xác thực mỗi request, nhất là với hệ thống lớn, gây tải cao cho server nếu không có cache hoặc giới hạn kết nối.

5.4 Ứng dụng

Basic Auth đôi khi vẫn được sử dụng trong các tình huống đơn giản như:

- Các dịch vụ nội bộ không cần xác thực phức tạp.

- API RESTful nhỏ, có cơ chế bảo vệ khác (IP whitelist, firewall).
- Kết hợp trong CI/CD (truyền token mã hóa kiểu Basic).

Tuy nhiên, trong các hệ thống sản phẩm lớn hoặc môi trường production hiện đại, Basic Auth gần như bị thay thế bởi các phương thức xác thực mạnh hơn như Token, OAuth, hoặc Session.

Chương 6: Session Auth

6.1 Tổng quan

Session-based Authentication là một phương pháp xác thực truyền thống, phổ biến trong các ứng dụng web cổ điển. Phương pháp này dựa trên cơ chế lưu trạng thái đăng nhập (session) của người dùng tại phía server, và dùng một session ID (thường được gửi qua cookie) để quản lý người dùng đã đăng nhập.

6.2 Quy trình hoạt động

1. Người dùng gửi thông tin đăng nhập (username/password) thông qua form.
2. Server kiểm tra thông tin và nếu hợp lệ: tạo một session (có thể lưu trong RAM, Redis, Database...).
3. Server gửi về client một cookie chứa session ID.
4. Các request tiếp theo: trình duyệt tự động gửi lại cookie đó.
5. Server xác minh session ID và nếu hợp lệ thì cho phép truy cập tài nguyên.

6.3 Ưu nhược điểm

- Ưu điểm:
    - Không cần gửi password mỗi request: Giảm nguy cơ lộ thông tin nhạy cảm.
    - Có thể kiểm soát session tập trung: Admin có thể xóa session, giới hạn thời gian sống session, hoặc theo dõi hoạt động.
    - Hỗ trợ dễ dàng logout: Chỉ cần huỷ session tại server.
- Nhược điểm:

- Phụ thuộc vào trạng thái server (stateful): Không phù hợp với hệ thống phân tán hoặc microservices nếu không dùng shared session storage (Redis…).
- Khó scale nếu không dùng session store: Vì mỗi request cần kiểm tra session tại server.
- Dễ bị tấn công CSRF (Cross Site Request Forgery): Nếu không có bảo vệ (CSRF token, SameSite cookie, v.v.).
- Bảo mật cookie: Cookie cần được cấu hình đúng (HttpOnly, Secure, SameSite) để tránh bị đánh cắp hoặc khai thác.

6.4 Ứng dụng

Session-based authentication là phương thức mặc định trong nhiều framework web cổ điển như:

- PHP ($_SESSION)
- Express.js (với middleware như express-session)
- Django, Ruby on Rails, ASP.NET WebForms, v.v.

Các hệ thống admin nội bộ, web quản lý nhân sự, CRM truyền thống... thường sử dụng session auth do tính đơn giản và dễ triển khai.

Chương 7: GraphQL

7.1 Tổng quan

- GraphQL là một ngôn ngữ truy vấn dành cho API (Application Programming Interface – Giao diện lập trình ứng dụng) và là một runtime phía máy chủ để thực thi các truy vấn dựa trên một hệ thống kiểu dữ liệu (type system) được định nghĩa bởi chính bạn.
- GraphQL được phát triển bởi Facebook vào năm 2012 và chính thức open-source vào năm 2015. Hiện nay, nó đã được triển khai rộng rãi bằng nhiều ngôn ngữ lập trình khác nhau như JavaScript, Python, Java, Go, PHP,...
- Khác với REST – mô hình API truyền thống dựa trên các endpoint cụ thể – GraphQL cho phép client yêu cầu chính xác dữ liệu mà họ cần, không nhiều hơn, không ít hơn.
- Các đặc điểm chính:

- Mô tả API bằng hệ thống kiểu dữ liệu: Một dịch vụ GraphQL được xây dựng thông qua việc định nghĩa các type (kiểu dữ liệu) và các field (trường dữ liệu) tương ứng. Mỗi field sẽ được ánh xạ với một hàm "resolver" để cung cấp dữ liệu.
- Truy vấn chính xác dữ liệu cần thiết: Client có thể gửi các truy vấn đến máy chủ GraphQL thông qua một endpoint duy nhất (thường là /graphql). Truy vấn sẽ được kiểm tra tính hợp lệ dựa trên schema định nghĩa sẵn, sau đó thực thi để trả về kết quả. Dữ liệu trả về có cùng cấu trúc với truy vấn và client chỉ nhận những gì họ yêu cầu.
- Phát triển API mà không cần versioning: GraphQL hỗ trợ phát triển và mở rộng API linh hoạt mà không cần phải tạo ra các phiên bản mới (v1, v2, v3…).

7.2 Kiến trúc của GraphQL

7.2.1 Schema

- Trong GraphQL, schema là trung tâm mô tả toàn bộ khả năng của một API. Nó chỉ rõ:
  - Những kiểu dữ liệu nào có thể được truy vấn.
  - Mối quan hệ giữa các kiểu dữ liệu.
  - Những trường (fields) nào khả dụng và kiểu dữ liệu của từng trường.
  - Cách thức client có thể gửi truy vấn để nhận kết quả dự đoán được.
- Schema được định nghĩa bằng một ngôn ngữ định nghĩa lược đồ gọi là SDL (Schema Definition Language), một cú pháp trực quan, gần gũi với truy vấn GraphQL và không phụ thuộc vào ngôn ngữ lập trình cụ thể.
- Hệ thống kiểu trong GraphQL bao gồm **6 loại định nghĩa kiểu có tên** (named type definitions):

Loại kiểu dữ liệu	**Mục đích sử dụng**
Object Type	Mô tả các đối tượng có trường (field)
Scalar Type	Đại diện cho các giá trị đơn giản (Int, String,...)
Enum Type	Tập hợp giá trị cố định (rất hữu ích cho validate)

Interface Type	Mô tả các trường chung cho nhiều kiểu đối tượng cụ thể
Union Type	Kết hợp nhiều Object Type không có trường chung
Input Object Type	Cho phép truyền đối tượng phức tạp vào arguments

- Object Type và Field: Đây là thành phần cơ bản của bất kỳ schema nào. Ví dụ:

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

  - Character là một Object type.
  - name là trường kiểu String! (không được null).
  - appearsIn là một List chứa các giá trị kiểu Episode!, cũng không null.
- Root Types: Schema GraphQL cần có ít nhất một kiểu gốc (root type):
  - Query: dùng để truy vấn dữ liệu.
  - Mutation: dùng để thay đổi dữ liệu (create/update/delete)
  - Subscription: hỗ trợ dữ liệu thời gian thực.

  Ví dụ:

```
type Query {
  droid(id: ID!): Droid
}
schema {
  query: MyQuery
  mutation: MyMutation
}
```

- Scalar Types: các Scalar types cơ bản trong GraphQL:

Kiểu	Mô tả

Int	Số nguyên 32-bit có dấu
Float	Số thực 64-bit
String	Chuỗi ký tự UTF-8
Boolean	True hoặc false
ID	Định danh duy nhất, thường dùng cho cache hoặc truy vấn

- Enum Types: dạng đặc biệt của scalar – chỉ cho phép một số giá trị cố định để ràng buộc giá trị, tăng độ tự mô tả của schema, hỗ trợ mạnh cho client thông qua introspection. Vi dụ:
  enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
  }
- Type Modifiers: gồm Non-Null và List
  - Non-Null: Field này không bao giờ được trả về null. Nếu null thì lỗi thực thi. Ví dụ: *name: String!*
  - List: Danh sách không null và mỗi phần tử trong danh sách cũng không null. Ví dụ: *appearsIn: [Episode!]!*
- Interface Types: Interface giúp định nghĩa bộ trường chung cho nhiều Object Type khác nhau:
  interface Character {
  id: ID!
  name: String!
  appearsIn: [Episode!]!
  }
  Một Object muốn triển khai Interface thì phải chứa đầy đủ các trường:
  type Human implements Character {
  id: ID!
  name: String!
  appearsIn: [Episode!]!
  totalCredits: Int
  }

Client chỉ có thể truy vấn các field nằm trong Interface:

```
{
  hero {
    name
    ... on Droid {
      primaryFunction
    }
  }
}
```

- Union Types: Giống Interface, nhưng không yêu cầu các field chung. Chỉ định danh sách các Object Type. Ví dụ: *union SearchResult = Human | Droid | Starship*

  Client phải dùng inline fragment để truy xuất:

```
{
  search(text: "an") {
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
  }
}
```

- Input Object Types: Dùng để truyền object phức tạp làm tham số. Ví dụ:

```
input ReviewInput {
  stars: Int!
  commentary: String
}

type Mutation {
  createReview(episode: Episode, review: ReviewInput!): Review
}
```

- Directives: Cho phép gắn metadata hoặc tùy biến hành vi truy vấn. Ví dụ phổ biến nhất:
  ```
  type User {
    name: String @deprecated(reason: "Use `fullName`.")
  }
  directive @deprecated(reason: String = "No longer supported")
    on FIELD_DEFINITION | ENUM_VALUE
  ```
- Documentation: GraphQL hỗ trợ mô tả trực tiếp trong schema bằng Markdown. Những mô tả này có thể được hiển thị trong các công cụ như GraphiQL hoặc Playground.
  ```
  """
  Một nhân vật trong Star Wars
  """
  type Character {
    "Tên nhân vật"
    name: String!
  }
  ```
- Comment: SDL cho phép dùng # để viết chú thích (không hiển thị cho client):
  ```
  # Đây là comment
  type Character {
    name: String!
  }
  ```

7.2.2 Query

- Fields: GraphQL cho phép bạn yêu cầu các trường cụ thể trên một đối tượng. Ví dụ:
  ```
  {
    hero {
      name
    }
  }
  ```
  Kết quả trả về có cùng hình dạng (shape) với truy vấn:
  ```
  {
    "data": {
  ```

```
  "hero": {
    "name": "R2-D2"
  }
}
```

- Nested Queries: Các trường có thể trả về đối tượng khác, cho phép truy vấn sâu hơn. Điều này giúp GraphQL vượt trội so với REST: chỉ cần một request để lấy nhiều dữ liệu liên quan:

```
{
  hero {
    name
    friends {
      name
    }
  }
}
```

- Arguments: có thể truyền tham số vào bất kỳ trường nào và không chỉ giới hạn ở root-level. Tham số có thể là kiểu dữ liệu đơn giản (scalar), enum, hoặc custom input type. Điều này loại bỏ nhu cầu gọi nhiều endpoint như trong REST:

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

- Operation Type và Name: có thể đặt tên truy vấn để dễ debug và theo dõi. Có các loại: query dùng đọc dữ liệu; mutation dùng ghi dữ liệu; subscription dùng lắng nghe dữ liệu theo thời gian thực:

```
query HeroNameAndFriends {
  hero {
    name
    friends {
      name
    }
```

```
  }
}
```

- Aliases: cho phép truy vấn cùng một trường nhiều lần với tham số khác nhau. Kết quả trả về với key theo alias:

```
query {
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
```

- Variables: có thể sử dụng biến thay vì hardcode giá trị trong query giúp tránh phải build query string động (insecure):

```
query Hero($episode: Episode) {
  hero(episode: $episode) {
    name
  }
}
```

- Default Variables: có thể đặt mặc định cho biến

```
query Hero($episode: Episode = JEDI) {
  hero(episode: $episode) {
    name
  }
}
```

- Fragment: giúp tái sử dụng một nhóm trường, hữu ích khi UI phức tạp với nhiều component chia sẻ chung schema:

```
fragment comparisonFields on Character {
  name
  friends {
    name
  }
}
```

```
query {
  hero1: hero(episode: EMPIRE) {
    ...comparisonFields
  }
  hero2: hero(episode: JEDI) {
    ...comparisonFields
  }
}
```

- Inline Fragments: Khi kết quả trả về là một Interface hoặc Union cần dùng inline fragment:

```
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      height
    }
  }
}
```

- Meta-fields: có thể truy vấn meta-field đặc biệt __typename để biết rõ kiểu thực sự của đối tượng:

```
{
  search(text: "an") {
    __typename
    ... on Human { name }
    ... on Droid { name }
  }
}
```

- Directives: GraphQL hỗ trợ chỉ thị (directive) để kiểm soát logic trong query, giúp kiểm soát linh hoạt việc hiển thị dữ liệu mà không cần thay đổi cấu trúc query.

```
query Hero($withFriends: Boolean!) {
```

```
  hero {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}
```

7.2.3 Mutation

- Trong khi phần lớn sự chú ý trong GraphQL tập trung vào việc truy vấn dữ liệu (Query), thì Mutation đóng vai trò cực kỳ quan trọng trong việc ghi dữ liệu (Write) , từ thêm mới, cập nhật cho đến xoá bỏ thông tin trên server. Trong REST, GET không nên có tác dụng phụ (side-effect), còn POST/PUT/PATCH/DELETE là các phương thức thay đổi trạng thái. Trong GraphQL, chỉ các trường top-level của Mutation được phép có tác dụng phụ.
- Cấu trúc mutation: Giống như query, mutation cũng bắt đầu bằng từ khóa mutation, tiếp theo là tên mutation và danh sách các trường bạn muốn trả về. Ví dụ: Mutation tạo mới Review:
    - Định nghĩa Schema:

      ```
      input ReviewInput {
        stars: Int!
        commentary: String
      }

      type Mutation {
        createReview(episode: Episode, review: ReviewInput!): Review
      }
      ```
    - Thao tác

      ```
      mutation CreateReviewForEpisode($ep: Episode!, $review:
      ReviewInput!) {
        createReview(episode: $ep, review: $review) {
          stars
          commentary
      ```

```
  }
}
```

- Biến

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

- Kết quả: ReviewInput là một Input Object Type – cho phép truyền đối tượng có cấu trúc, tương tự DTO trong lập trình hướng đối tượng.

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

- Cập nhật dữ liệu: có thể định nghĩa các mutation để cập nhật thông tin:

```
type Mutation {
  updateHumanName(id: ID!, name: String!): Human
}
mutation UpdateHumanName($id: ID!, $name: String!) {
  updateHumanName(id: $id, name: $name) {
    id
    name
  }
}
{
  "id": "1000",
  "name": "Luke Starkiller"
}
```

```
{
  "data": {
    "updateHumanName": {
      "id": "1000",
      "name": "Luke Starkiller"
    }
  }
}
```

- Purpose-built mutations: Thay vì chỉ có mutation tổng quát như updateHuman, bạn nên định nghĩa mutation cụ thể như updateHumanName, updateHumanAge,... Giúp biểu đạt rõ mục đích, có thể sử dụng Non-Null Argument (!) và dễ kiểm soát logic và validation. Ví dụ mutation đánh giá phim:

```
mutation RateFilm($episode: Episode!, $rating: FilmRating!) {
  rateFilm(episode: $episode, rating: $rating) {
    episode
    viewerRating
  }
}
```

- Xoá dữ liệu: GraphQL không có keyword riêng như DELETE, bạn vẫn dùng mutation:

```
type Mutation {
  deleteStarship(id: ID!): ID!
}
mutation DeleteStarship($id: ID!) {
  deleteStarship(id: $id)
}
```

- Gọi nhiều mutation trong một request: có thể chạy nhiều mutation trong cùng 1 request, các trường thực thi tuần tự (serially), giúp tránh race condition, nhưng không đảm bảo tính toàn vẹn giao dịch (transaction):

```
mutation {
  firstShip: deleteStarship(id: "3001")
  secondShip: deleteStarship(id: "3002")
}
```

- Server-side thực thi mutation: Ví dụ server xử lý mutation createReview:

```
const Mutation = {
  createReview(_obj, args, context, _info) {
    return context.db
      .createNewReview(args.episode, args.review)
      .then(reviewData => new Review(reviewData));
  }
};
```

7.2.4 Subscription

- Subscription là loại thao tác GraphQL cho phép client nghe các thay đổi từ server theo thời gian thực. Subscription là một operation (tương tự Query/Mutation), nhưng sử dụng từ khóa subscription. Mỗi Subscription duy trì kết nối lâu dài với server (thường là WebSocket) để nhận dữ liệu tự động mỗi khi có sự kiện xảy ra. Đây là điểm giúp GraphQL trở nên mạnh mẽ hơn REST trong các ứng dụng cần real-time updates như chat, thông báo, dashboard, game...

```
subscription NewReviewCreated {
  reviewCreated {
    rating
    commentary
  }
}
```

- Cách hoạt động phía server: Khi mutation createReview được gọi thành công, server phát sự kiện qua một hệ thống pub/sub. Subscription sẽ lắng nghe sự kiện đó và gửi dữ liệu cho tất cả client đang đăng ký. PubSub có thể là: Memory PubSub (cho demo), Redis Pub/Sub, Kafka, NATS, MQTT,…
- Giao thức và triển khai: GraphQL không quy định giao thức (transport), nhưng thực tế thường dùng:
    - Websocket: hai chiều (bi-directional), phổ biến nhất cho real-time.
    - Server-Sent Events: Một chiều (server → client), đơn giản, dùng cho push nhẹ nhàng
    - HTTP Long Polling: Giải pháp tạm thời nếu không có WebSocket hoặc SSE

- Mỗi Subscription chỉ được có 1 root field. Với nhiều subscription, mỗi cái phải có tên riêng và chỉ một cái được chọn khi gửi.
- Các tình huống sử dụng Subscriptions: nhận tin nhắn mới ngay lập tức, hiển thị thông báo hệ thống theo thời gian, cập nhật số liệu, traffic, giá cổ phiếu, trạng thái các player khác,…
- Mặc dù mạnh mẽ, Subscriptions yêu cầu kiến trúc phức tạp hơn. Vì vậy, bạn cần các công cụ: Redis hoặc Kafka để chia sẻ pub/sub giữa các server; Load balancer hỗ trợ sticky session (hoặc sử dụng WebSocket-aware); Caching phía client để merge dữ liệu từ subscription.
- Không nên dùng subscription khi cập nhật không thường xuyên, gửi một chiều (như tin khuyến mãi), tải lại toàn bộ trang khi có thay đổi, dữ liệu nhạy cảm, cần kiểm soát kỹ hơn.

7.2.5 Resolver

- Resolver là một hàm xử lý được gọi mỗi khi một field trong GraphQL schema được yêu cầu trong một query/mutation/subscription.
- Cấu trúc: Resolver luôn nhận vào 4 tham số
    - Parent: Kết quả từ field cha
    - Args: Đối số của field (GraphQL argument)
    - Context: Dữ liệu toàn cục cho toàn request (user, db, token, logger...)
    - Info: Metadata GraphQL (tên field, type, query AST...)
- Nếu bạn không khai báo resolver cho một field, GraphQL sẽ sử dụng default resolver
- Nếu bạn không khai báo resolver cho một field, GraphQL sẽ sử dụng default resolver:

```
query {
  user(id: "1") {
    name
    posts {
      title
      comments {
        content
      }
    }
  }
```

}

- Resolver và Context: Context là nơi lý tưởng để chia sẻ dữ liệu như: token đã decode, database connection, Service layer (UserService, AuthService, MailService,...).
- Resolver và DataLoader: Trong các field lồng nhau (ví dụ User.posts → Post.comments), GraphQL dễ gặp vấn đề N+1 query nên sử dụng dataloader để batch và cache query.

7.3 Các hoạt động

1. Client gửi một truy vấn (query) đến server GraphQL.
2. Server kiểm tra truy vấn dựa trên schema đã định nghĩa.
3. Gọi các resolver để lấy dữ liệu từ database hoặc API khác.
4. Server trả về dữ liệu đúng định dạng yêu cầu từ client.

7.4 Công cụ hoạt động

- Client Tools: Giao tiếp với GraphQL từ phía người dùng
    - Apollo Client: Thư viện client phổ biến nhất hiện nay. Cung cấp: cache, pagination, optimistic UI, state management.
    - Relay: Thư viện client của Facebook, mạnh về performance, pagination và GraphQL fragments.
    - Urql: Nhẹ hơn Apollo, phù hợp với React & mobile apps.
- Server Tools: Xây dựng GraphQL API backend
    - Apollo Server: Thư viện phổ biến để triển khai GraphQL server với Node.js, hỗ trợ context, schema stitching, federation.
    - Express-GraphQL: Plugin đơn giản tích hợp với Express.js – lý tưởng cho học tập hoặc server nhỏ.
    - GraphQL Yoga: Server nhẹ, built-in support WebSocket, subscriptions.
    - Mercurius: GraphQL server nhanh, phù hợp Fastify framework.
    - NestJS GraphQL: Module của NestJS hỗ trợ schema-first & code-first, phù hợp enterprise.
- GraphQL IDE & Playground: Công cụ kiểm thử truy vấn
    - GraphQL Playground: Giao diện thử truy vấn GraphQL thân thiện, hỗ trợ auto-complete, query history, variables.

    - GraphiQL: IDE tích hợp trong trình duyệt, đơn giản và hiệu quả, thường được bundle sẵn trong các GraphQL server.
    - Postman: Có hỗ trợ GraphQL nhưng không mạnh như Playground hoặc GraphiQL
- Auto-generated GraphQL Backend: Tự động tạo API GraphQL từ database
    - Hasura: Tạo GraphQL API real-time từ PostgreSQL, hỗ trợ subscription, role-based permission, metadata.
    - PostGraphile: Tạo GraphQL server từ PostgreSQL schema (code-first), phù hợp với backend tùy chỉnh.
    - GraphQL Mesh: Kết nối nhiều nguồn dữ liệu khác nhau (REST, gRPC, OpenAPI) thành GraphQL API duy nhất.
- Schema Design & Visualization: GraphQL Voyager, GraphQL Inspector, GraphQL Editor.
- Monitoring & DevOps: Apollo Studio, GraphQL Hive, New Relic, Datadog, Prometheus.
- Security & Validation: graphql-shield, GraphQL Depth Limit, Persisted Queries.

7.5 Bảo mật

- Các rủi ro bảo mật phổ biến:
    - Query quá sâu (Deep Query): Truy vấn quá nhiều lớp liên kết (nested fields) có thể khiến server phải thực hiện nhiều phép truy cập database hoặc tính toán đệ quy nên dễ bị DoS (Denial of Service).
    - Vòng lặp truy vấn vô hạn: Nếu schema cho phép truy vấn liên tục các quan hệ 2 chiều như user -> friends -> user -> friends, client có thể vô tình hoặc cố ý gây infinite recursion.
    - Truy vấn phức tạp (Overly Complex Queries): Một truy vấn có thể yêu cầu hàng ngàn trường dữ liệu hoặc các tính toán nặng làm tốn CPU/RAM, ảnh hưởng đến hệ thống.
    - Truy cập trái phép: Nếu không có phân quyền rõ ràng, người dùng có thể truy cập dữ liệu không thuộc quyền sở hữu.
    - Rò rỉ schema (Introspection): Nếu không vô hiệu hóa introspection, attacker có thể dễ dàng liệt kê toàn bộ schema, từ đó lên kế hoạch tấn công.

    - Injection thông qua arguments: Nếu không validate arguments đúng cách, có thể dẫn đến NoSQL Injection / SQL Injection
    - Quá tải server qua truy vấn hàng loạt: Client gửi hàng trăm operation trong một request GraphQL làm server bị choke.
    - WebSocket Subscriptions lỏng lẻo: Nếu không xác thực chặt, attacker có thể lắng nghe subscription của người khác.
- Các biện pháp phòng ngừa:
    - Giới hạn độ sâu truy vấn: Dùng để ngăn chặn các truy vấn có độ sâu lớn gây tốn tài nguyên.
    - Giới hạn độ phức tạp: Đánh trọng số cho từng field/truy vấn bằng cách tính tổng complexity và từ chối truy vấn vượt mức.
    - Xác thực và phân quyền: Xác minh người dùng là ai bằng JWT (JSON Web Token), OAuth2 / OpenID Connect, Session, API Key. Phân quyền: Kiểm tra người dùng có quyền làm hành động đó không, có thể dùng middleware hoặc schema-based.
    - Rate Limiting: Giới hạn số lượng truy vấn mỗi user trong 1 thời gian cụ thể, có thể dùng Redis + express-rate-limit, API Gateway, Token Bucket / Leaky Bucket Algorithm.
    - Persisted Queries: Chỉ cho phép client gửi ID của truy vấn đã đăng ký trước giúp ngăn chặn việc gửi truy vấn độc hại, có thể dùng Apollo Persisted Queries hoặc tự triển khai
    - Vô hiệu hóa introspection schema: Ngăn không cho người lạ dò được toàn bộ schema qua introspection queries.
    - Secure WebSocket Subscription: Xác thực người dùng trước khi kết nối subscription.

7.6 Ưu nhược điểm

- Ưu điểm:
    - Truy vấn chính xác dữ liệu cần thiết
    - Một endpoint duy nhất
    - Tự động sinh documentation
    - Realtime với Subscription
    - Dễ mở rộng và tích hợp

- Nhược điểm:
    - Phức tạp khi bắt đầu
    - Tối ưu hóa caching khó hơn REST
    - Cần thiết kế schema tốt
    - Không thích hợp cho tất cả hệ thống nhỏ