Công ty Cổ phần VCCorp

BÁO CÁO TUẦN 4

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

Tác giả: Trần Văn Toàn

Người hướng dẫn: Anh Ngô Văn Vĩ

Hà Nội, tháng 7 năm 2025

Tóm tắt nội dung

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

Mục lục

Chương 1

Tìm hiểu cơ bản về Kubernetes - K8s

1.1 Định nghĩa Kubernetes

Định nghĩa

Kubernetes là một hệ thống mã nguồn mở phục vụ mục đích triển khai tự động (automating deployment), mở rộng hệ thống (scaling) và quản lý các container hóa ứng dụng.

Nên sử dụng Kubernetes khi nào?

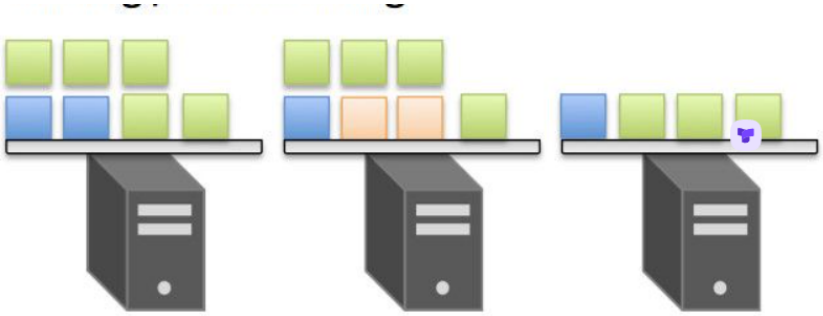
- Các doanh nghiệp lớn, có nhu cầu thực sự phải scaling hệ thống nhanh chóng, và đã sử dụng container (Docker).
- Các dự án cần chạy >= 5 container cùng loại cho 1 dịch vụ
- Các startup tân tiến, chịu đầu tư vào công nghệ để dễ dàng auto scale về sau.

1.2 Chức năng của Kubernetes - K8s

1.2.1 Scaling / Auto Scaling

Kubernetes quản lý các docker host và cấu trúc container cluster. Ngoài ra, khi thực thi các container trên K8s, bằng cách thực hiện replicas (tạo ra nhiều container giống nhau)

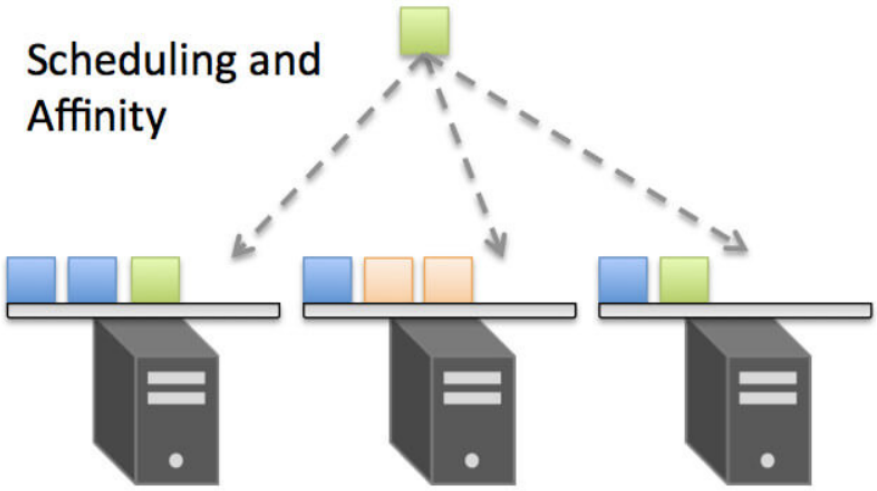
làm cho hệ thống có sức chịu lỗi cao và tự động thực hiện load balancing. Thông qua cơ chế load balancing, chúng ta có thể tăng giảm số lượng container replica (auto scaling).



Hình 1.1: Scaling / Auto Scaling

1.2.2 Scheduling and affinity

Khi thực hiện phân chia container vào các Node (docker host), dựa trên các loại docker host kiểu như "Disk SSD" hay "số lượng clock của CPU cao"... Hoặc dựa trên loại Workload kiểu như "Disk I/O quá nhiều", "Băng thông đến một container chỉ định quá nhiều" ... K8s sẽ ý thức được việc affinity hay anti-affinity và thực hiện Scheduling một cách hợp lý cho chúng ta

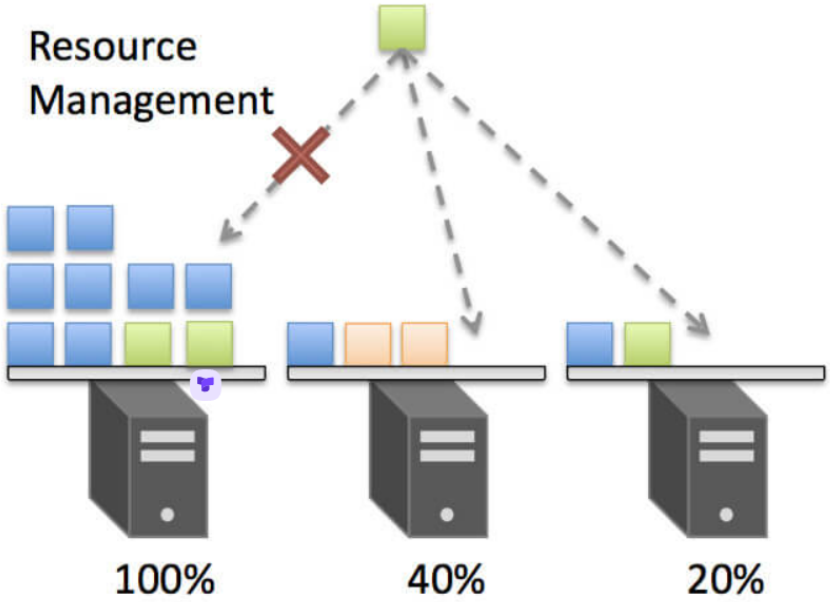


Hình 1.2: Scheduling and affinity

1.2.3 Resource Management

Trong trường hợp không được chỉ định host cụ thể, K8s sẽ thực hiện scheduling tuỳ thuộc vào tình trạng CPU, memmory của docker host có trống hay không. Vì vậy, chúng ta không cần quan tâm đến việc quản lý bố trí container vào các docker host như thế nào.

Hơn nữa, trường hợp resource không đủ, thì việc auto scheduling của K8s cluster cũng sẽ được thực hiện tự động.



Hình 1.3: Resource Management

1.3 Các khái niệm cơ bản

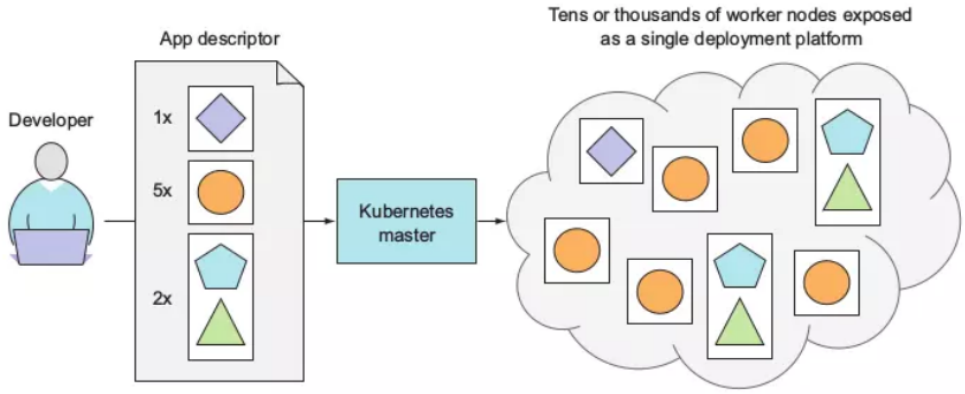
1.3.1 Cluster

Định nghĩa

Kubernetes cluster là một tập hợp các máy ảo/vật lý được cài đặt Kubernetes dùng để chạy các ứng dụng. Các máy ảo/máy vật lý này được gọi là các nodes, được phân chia thành master node và các worker nodes.

Như mô tả trong Hình 1.4, hệ thống bao gồm Kubernetes Master Node và các Worker Nodes. Trong thực tế, một cluster có thể có hàng chục, thậm chí hàng ngàn worker nodes

tùy thuộc vào quy mô của hệ thống.



Hình 1.4: Kiến trúc Kubernetes Cluster với Master Node và Worker Nodes

Master Node

Master Node là thành phần điều khiển chính của Kubernetes cluster, chịu trách nhiệm:

- Quản lý và điều phối các hoạt động của cluster
- Lập lịch triển khai các container lên các worker nodes
- Quản lý API server, scheduler và controller manager
- Lưu trữ trạng thái của cluster trong etcd

Worker Node

Worker Node là nơi thực sự chạy các ứng dụng container, bao gồm:

- **Kubelet**: Agent chạy trên mỗi node để giao tiếp với master
- **Container Runtime**: Môi trường chạy container (Docker, containerd, etc.)
- **Kube-proxy**: Quản lý network rules và load balancing

1.3.2 Node

Định nghĩa

Node là đơn vị nhỏ nhất của phần cứng máy tính trong Kubernetes, đại diện cho một máy duy nhất trong Kubernetes cluster.

Bản chất của Node

Node trong Kubernetes có thể là:

- **Máy vật lý**: Server thực tế trong data center
- **Máy ảo trên cloud**: Như Google Cloud Platform (GCP), Amazon Web Services (AWS)
- **Máy ảo local**: Được tạo bởi VirtualBox, VMware trên máy cá nhân

Yêu cầu kỹ thuật

Mỗi node cần được cài đặt:

- **Kubernetes**: Bao gồm các thành phần cốt lõi
- **Docker**: Hoặc container runtime tương đương
- **Docker host**: Mỗi node hoạt động như một Docker host độc lập

Tài nguyên và tính linh hoạt

Định nghĩa

Tài nguyên Node: Mỗi node có thể được xem như một tập hợp tài nguyên CPU và RAM có thể sử dụng.

Định nghĩa

Tính thay thế: Trong Kubernetes cluster, bất kỳ node nào cũng có thể thay thế bất kỳ node nào khác, nhờ vào tính đồng nhất về tài nguyên và cấu hình.

Điều này mang lại các lợi ích:

- Khả năng mở rộng linh hoạt
- Tính sẵn sàng cao của hệ thống
- Dễ dàng bảo trì và thay thế phần cứng
- Phân bổ workload tự động

1.3.3 Pods

Định nghĩa

Pod là đơn vị nhỏ nhất có thể triển khai trong Kubernetes, đại diện cho một nhóm các container chia sẻ tài nguyên và network.

Khái niệm cơ bản

Khi một ứng dụng được đóng gói (containerized), nó có thể chạy trên một container độc lập. Tuy nhiên, Kubernetes sử dụng khái niệm Pod để nhóm các container liên quan lại với nhau.

Đặc điểm của Pod:

- Là một nhóm gồm một hoặc nhiều container
- Các container trong Pod chia sẻ tài nguyên và network
- Các container có thể giao tiếp với nhau như trên cùng một máy chủ
- Vẫn duy trì sự độc lập cần thiết giữa các container

Cách tổ chức Pod

Định nghĩa

Nhóm theo mục đích: Các Pod thường nhóm các container có cùng mục đích sử dụng hoặc cùng thuộc một ứng dụng.

Ví dụ về cách tổ chức Pod:

- **Pod backend**: Tập hợp 4 container chạy nginx + backend services
- **Pod frontend**: Tập hợp 2 container chạy frontend + nginx proxy
- **Pod database**: Container database + container backup/monitoring

Lợi ích của Pod

- **Đơn giản hóa việc Scale**
- **Quản lý đơn giản**: Quản lý nhóm container như một đơn vị
- **Chia sẻ tài nguyên**: Storage, network, và configuration
- **Lifecycle đồng bộ**: Các container trong Pod có cùng lifecycle
- **Co-location**: Các container luôn được đặt trên cùng một node

Tài nguyên chia sẻ trong Pod

Network:

- Các container chia sẻ cùng IP address
- Giao tiếp qua localhost với các port khác nhau
- Chia sẻ network namespace

Storage:

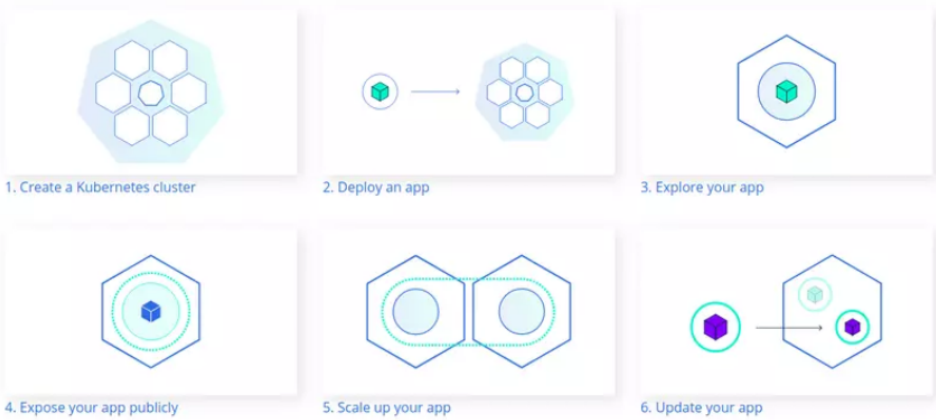
- Chia sẻ volumes được mount
- Có thể chia sẻ dữ liệu giữa các container
- Persistent storage cho toàn bộ Pod

Configuration:

- Chia sẻ environment variables
- Chia sẻ secrets và configmaps
- Cùng security context

1.4 Một số module cơ bản của Kubernetes

- Create Kubernetes cluster
- Deploy an app
- Explore app
- Expose app publicity
- Scale up app
- Update app



Hình 1.5: Kubernetes Basics Modules

1.4.1 Image

Định nghĩa

Image là phần mềm chạy ứng dụng đã được đóng gói thành một chương trình để có thể chạy dưới dạng container. Các Pod sẽ sử dụng các Image này để khởi chạy container.

Quản lý Image: Các Image thường được quản lý tại các kho lưu trữ tập trung, ví dụ:

- **Docker Hub**: Chứa Images của các ứng dụng phổ biến
- **Ví dụ Images**: nginx, mysql, wordpress, redis, mongodb
- **Private Registry**: Kho lưu trữ riêng của tổ chức

1.4.2 Deployment

Định nghĩa

Deployment là cách thức để triển khai, cập nhật và quản trị Pod một cách khai báo (declarative) trong Kubernetes.

Chức năng của Deployment:

- Triển khai ứng dụng với số lượng replica mong muốn
- Cập nhật ứng dụng theo chiến lược rolling update
- Rollback về phiên bản trước khi cần thiết
- Quản lý lifecycle của Pod

1.4.3 Replica Controller

Định nghĩa

Replica Controller là thành phần quản trị bản sao của Pod, đảm bảo số lượng Pod replica luôn duy trì theo mong muốn.

Chức năng chính:

- Nhân bản Pod khi cần scale up
- Giảm số lượng Pod khi cần scale down
- Tự động tạo lại Pod khi Pod bị lỗi
- Đảm bảo tính sẵn sàng của ứng dụng

1.4.4 Service

Định nghĩa

Service là thành phần mạng (network) của Kubernetes, cung cấp điểm truy cập ổn định cho các Pod và thực hiện load balancing.

Chức năng của Service:

- Giúp các Pod gọi nhau một cách ổn định
- Load balancing giữa nhiều bản sao của Pod
- Dẫn traffic từ người dùng vào ứng dụng (Pod)
- Cung cấp service discovery cho các ứng dụng

Các loại Service:

- **ClusterIP**: Truy cập nội bộ trong cluster
- **NodePort**: Truy cập từ bên ngoài qua port của node
- **LoadBalancer**: Sử dụng load balancer của cloud provider
- **ExternalName**: Mapping tới external service

1.4.5 Label

Định nghĩa

Label là hệ thống nhãn key-value được sử dụng để phân loại và quản lý các đối tượng trong Kubernetes, đặc biệt là Pod.

Mục đích của Label:

- Phân loại Pod theo chức năng
- Quản lý Pod theo môi trường

- Hỗ trợ selector trong Service và Deployment
- Tổ chức và tìm kiếm resources

Ví dụ về Label:

- **Theo chức năng**: `app=frontend`, `app=backend`
- **Theo môi trường**: `env=dev`, `env=qc`, `env=uat`, `env=production`
- **Theo version**: `version=v1.0`, `version=v2.0`
- **Theo team**: `team=web`, `team=api`

Chương 2

JSON Web Token (JWT)

2.1 Định nghĩa

Định nghĩa

JSON Web Mã (JWT) là một chuẩn mở (RFC 7519) định nghĩa một cách nhỏ gọn và khép kín để truyền một cách an toàn thông tin giữa các bên dưới dạng đối tượng JSON. Thông tin này có thể được xác minh và đáng tin cậy vì nó có chứa chữ ký số. JWTs có thể được ký bằng một thuật toán bí mật (với thuật toán HMAC) hoặc một cặp khóa công khai/riêng tư sử dụng mã hóa RSA.

2.2 Cấu trúc một JWT

Ví dụ

Cấu trúc một JWT **Cấu trúc một JWT:**

```
1 <base64-encoded header>.<base64-encoded payload>.<base64-
    encoded signature>
```

Header: Header bao gồm hai phần chính: loại token (mặc định là JWT - Thông tin này cho biết đây là một Token JWT) và thuật toán đã dùng để mã hóa (HMAC SHA256 - HS256 hoặc RSA).

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

Giải thích các trường:

- `"alg": "HS256"` - Thuật toán mã hóa sử dụng HMAC SHA256
- `"typ": "JWT"` - Loại token, luôn là JWT

Payload: Payload chứa các claims. Claims là một các biểu thức về một thực thể (chẳng hạn user) và một số metadata phụ trợ. Có 3 loại claims thường gặp trong Payload: reserved, public và private claims.

Reserved Claims: Đây là một số metadata được định nghĩa trước, trong đó một số metadata là bắt buộc, số còn lại nên tuân theo để JWT hợp lệ và đầy đủ thông tin:

- `iss` (issuer) - Đơn vị phát hành token
- `iat` (issued-at time) - Thời gian phát hành token
- `exp` (expiration time) - Thời gian hết hạn token
- `sub` (subject) - Chủ thể của token
- `aud` (audience) - Đối tượng mục tiêu của token
- `jti` (Unique Identifier) - Mã định danh duy nhất cho JWT. Có thể được sử dụng để ngăn JWT bị phát lại. Hữu ích cho token sử dụng một lần.

Ví dụ

Ví dụ về Reserved Claims

```
{
    "iss": "jira:1314039",
    "iat": 1300819370,
    "exp": 1300819380,
    "qsh": "8063
       ff4ca1e41df7bc90c8ab6d0f6207d491cf6dad7c66ea797b4614b71922e9
       ",
    "sub": "batman",
    "context": {
        "user": {
            "userKey": "batman",
            "username": "bwayne",
            "displayName": "Bruce Wayne"
        }
    }
}
```

Ví dụ

Ví dụ khác về Reserved Claims

```
{
  "iss": "scotch.io",
  "exp": 1300819380,
  "name": "Chris Sevilleja",
  "admin": true
}
```

Public Claims Claims được cộng đồng công nhận và sử dụng rộng rãi. Các claims này phải được định nghĩa trong IANA JSON Web Token Registry hoặc được định nghĩa như một URI chứa namespace chống xung đột.

Private Claims Claims tự định nghĩa (không được trùng với Reserved Claims và Public Claims), được tạo ra để chia sẻ thông tin giữa 2 parties đã thỏa thuận và thống nhất trước đó. Đây là các claims tùy chỉnh được tạo ra để chia sẻ thông tin giữa các bên tham gia giao tiếp.

Signature Chữ ký Signature trong JWT là một chuỗi được mã hóa bởi header, payload cùng với một chuỗi bí mật theo nguyên tắc sau:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Giải thích công thức:

- `HMACSHA256`: Thuật toán mã hóa HMAC với SHA256
- `base64UrlEncode(header)`: Header được mã hóa base64url
- `base64UrlEncode(payload)`: Payload được mã hóa base64url
- `secret`: Khóa bí mật dùng để ký
- Dấu `"."` là ký tự phân cách giữa header và payload

Mục đích của Signature:

Do bản thân Signature đã bao gồm cả header và payload nên Signature có thể dùng để kiểm tra tính toàn vẹn của dữ liệu khi truyền tải. Điều này đảm bảo rằng:

- Dữ liệu không bị thay đổi trong quá trình truyền
- Token được tạo bởi nguồn đáng tin cậy (có khóa bí mật)
- Xác thực tính hợp lệ của JWT

2.3 How do JSON Web Tokens work?

Ví dụ cụ thể ứng dụng của JWT trong bài toán Authenticate (xác thực):

Quá trình Authentication với JWT

Quá trình xác thực với JWT:

1. **Đăng nhập**: Khi user đăng nhập thành công (Browser sẽ post username và mật khẩu về Server), Server sẽ trả về một chuỗi JWT về Browser.
2. **Lưu trữ Token**: Token JWT này cần được lưu lại trong Browser của người dùng (thường là LocalStorage hoặc Cookies), thay vì cách truyền thống là tạo một session trên Server và trả về Cookie.
3. **Truy cập tài nguyên bảo vệ**: Bất cứ khi nào mà User muốn truy cập vào Route được bảo vệ (mà chỉ có User đã đăng nhập mới được phép), Browser sẽ gửi token JWT này trong Header Authorization, Bearer schema của request gửi đi.
4. **Xác thực**: Server sẽ kiểm tra Token JWT này có hợp lệ hay không (Bởi vì JWT có tính chất self-contained, mọi thông tin cần thiết để kiểm tra JWT đều đã được chứa trong Token JWT).

2.3.1 Định dạng Authorization Header

Khi gửi request đến server, JWT được đính kèm trong header Authorization theo định dạng Bearer:

```
Authorization: Bearer <token>
```

Ví dụ cụ thể:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
   eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyf0
   .SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

2.3.2 Tính chất Stateless

Đây là cách mà **stateless** (phi trạng thái) authentication làm việc:

- Trạng thái của user không được lưu trong bộ nhớ của Server
- Thông tin được đóng gói hẳn vào trong JWT

- Server chỉ cần xác minh tính hợp lệ của token mà không cần lưu trữ session

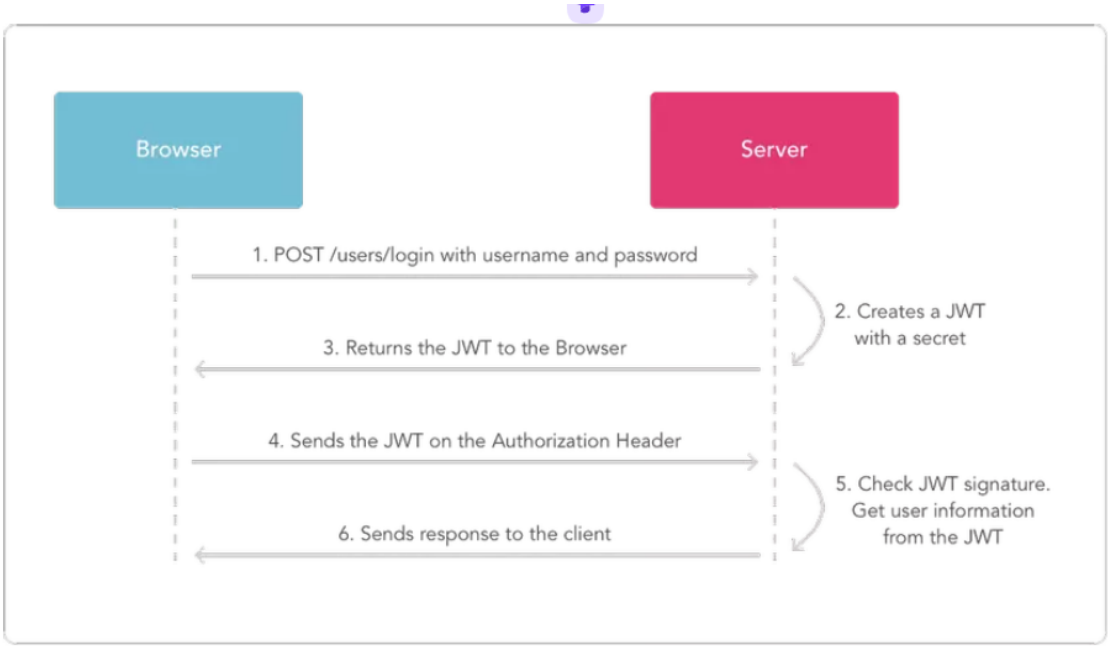
Lợi ích của JWT

Lợi ích của tính chất stateless:

- **Không lo lắng về domains**: Không còn phải lo lắng về domains nào được sử dụng cho API của bạn
- **Giải quyết vấn đề CORS**: Không còn gặp rắc rối với CORS (Cross-Origin Resource Sharing) vì nó không sử dụng cookies
- **Khả năng mở rộng**: Dễ dàng scale horizontal vì không cần chia sẻ session giữa các server
- **Tính độc lập**: Các microservice có thể xác thực độc lập mà không cần truy cập database session

2.3.3 Sơ đồ quá trình

Sơ đồ dưới đây cho thấy quá trình này:



Hình 2.1: Sơ đồ quá trình chạy

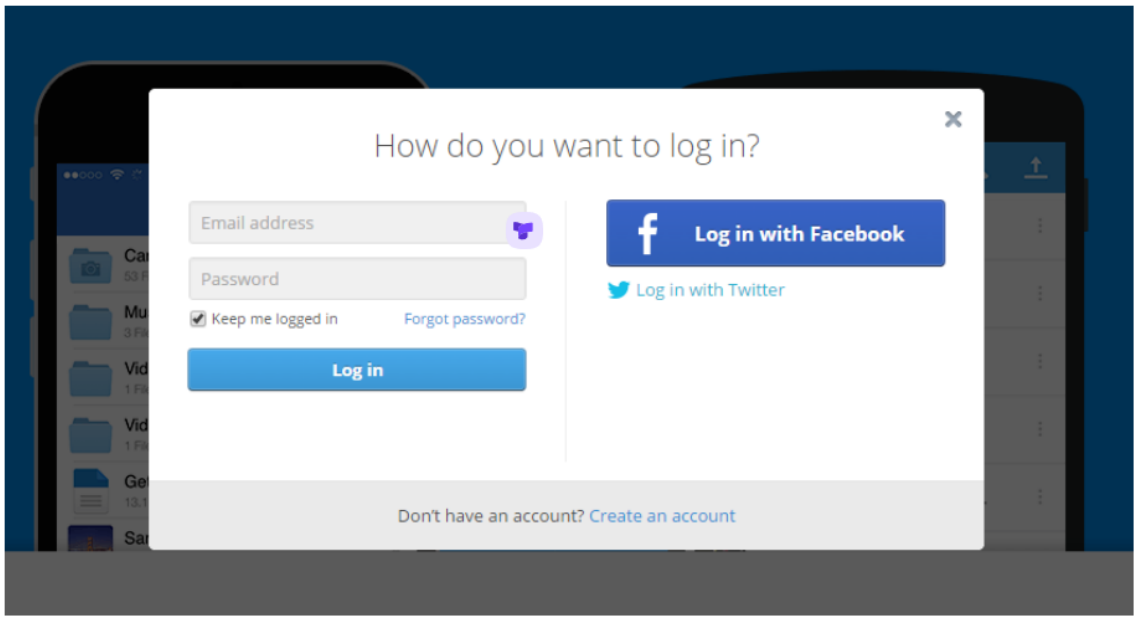
Chương 3

OAuth

3.1 OAuth là gì?

Định nghĩa

OAuth là một cách xác thực mở (open standard for authorization). Nếu bạn thấy một website (hay phần mềm) nào đó cho phép đăng nhập bằng tài khoản Facebook hay Google mà không cần tạo nick mới thì rất có khả năng đó chính là OAuth. Đây là một chuẩn xác thực mở được rất nhiều các website và phần mềm sử dụng.



Hình 3.1: Ví dụ về OAuth

3.2 Phân biết hai loại OAuth

- **2-legged OAuth:** là kiểu Authorization trong đó vai trò của bạn và Client là như nhau. Tức là Client chính là bạn và Server. Đó là kịch bản Client-Server thông thường.
- **3-legged OAuth:** là kiểu Authorization trong đó Bạn và Client là phân biệt. Client muốn bạn chia sẻ tài nguyên đã có bên phía Server.

3.3 Cách OAuth hoạt động

A.2-legged OAuth

- Client đăng ký sử dụng dịch vụ với Server
- Server cho Client (CONSUMER-KEY, CONSUMER-SECRET-KEY)
- Client sử dụng các keys trên để thực hiện Authorization

B.3-legged OAuth

- Client đăng kí sử dụng dịch vụ với Server
- Server cho Client (CONSUMER-KEY, CONSUMER-SECRET-KEY)
- Bạn có tài nguyên ở Server
- Bạn sử dụng dịch vụ ở Client, Client yêu cầu bạn cho phép khai thác tài nguyên của bạn ở Server
- Client yêu cầu Server gửi REQUEST-TOKEN cho bạn
- Client chuyển bạn đến Server Authentication
- Bạn đăng nhập vào Server, Server hỏi bạn có muốn chia sẻ quyền khai thác dữ liệu cho Client hay không
- You đồng ý, Server chuyển You về Client kèm theo ACCESS-TOKEN
- Client sử dụng ACCESS-TOKEN để thực hiện thao tác trên các tài nguyên của You thuộc Server

3.4 Kết luận

- Tiện lợi, nhanh chóng: không cần phải tạo nick mới mỗi khi đăng nhập vào một website hay phần mềm.
- An toàn: Hạn chế được tình trạng đánh cắp mật khẩu, thông tin cá nhân.

Chương 4

SSO - Đăng nhập một lần

4.1 SSO là gì?

Định nghĩa

SSO là viết tắt của "Single Sign-On", đồng nghĩa với "Đăng nhập một lần duy nhất". Nó là một phương pháp xác thực và ủy quyền cho phép người dùng truy cập vào nhiều ứng dụng và hệ thống khác nhau chỉ bằng cách đăng nhập một lần duy nhất với một thông tin đăng nhập duy nhất.

4.2 Tầm quan trọng của SSO

- **Tiện lợi cho người dùng:** Single Sign-On (SSO) mang lại sự tiện lợi cho người dùng bằng cách cho phép họ chỉ cần đăng nhập một lần duy nhất và sau đó có thể truy cập vào nhiều ứng dụng khác mà không cần phải đăng nhập lại. Điều này giúp tiết kiệm thời gian và gia tăng tính tiện ích.
- **Cải thiện an ninh:** SSO giúp cải thiện mức độ an ninh trong hệ thống. Thay vì phải sử dụng nhiều tài khoản và mật khẩu khác nhau cho từng ứng dụng, người dùng chỉ cần quản lý một bộ thông tin đăng nhập duy nhất. Điều này giúp giảm khả năng người dùng sử dụng mật khẩu yếu hoặc tái sử dụng mật khẩu giữa các ứng dụng khác nhau. Hơn nữa, việc quản lý quyền truy cập cũng trở nên dễ dàng hơn, vì người quản trị chỉ cần thực hiện quyền truy cập một lần cho mỗi người dùng.

- **Tăng hiệu quả và giảm chi phí:** SSO giúp tăng hiệu quả làm việc bằng cách giảm thời gian để đăng nhập vào các ứng dụng khác nhau. Nó cũng giảm khối lượng công việc cho người quản trị hệ thống, vì họ chỉ cần quản lý một bộ thông tin đăng nhập cho mỗi người dùng thay vì nhiều tài khoản khác nhau. Điều này giúp giảm chi phí vận hành và hỗ trợ kỹ thuật.
- **Tích hợp dễ dàng:** SSO cung cấp khả năng tích hợp dễ dàng với các ứng dụng và dịch vụ khác nhau. Hầu hết các hệ thống và ứng dụng hiện đại hỗ trợ các giao thức và tiêu chuẩn SSO phổ biến như SAML (Security Assertion Markup Language) và OAuth (Open Authorization). Điều này giúp việc triển khai SSO trở nên đơn giản và linh hoạt.

4.3 Cơ chế hoạt động của SSO (Single Sign-On)

Định nghĩa

Single Sign-On (SSO) là một cơ chế xác thực cho phép người dùng đăng nhập một lần duy nhất để truy cập vào nhiều ứng dụng và hệ thống khác nhau mà không cần phải đăng nhập lại cho từng ứng dụng riêng biệt.

Cơ chế hoạt động của SSO thường bao gồm các bước sau:

Bước 1: Truy cập ứng dụng: Người dùng truy cập vào một ứng dụng hoặc hệ thống SSO.

Bước 2: Yêu cầu xác thực: Hệ thống SSO yêu cầu người dùng cung cấp thông tin đăng nhập, chẳng hạn như tên người dùng và mật khẩu.

Bước 3: Tạo phiên làm việc: Sau khi người dùng cung cấp thông tin đăng nhập và xác thực thành công, hệ thống SSO tạo ra một phiên làm việc hoặc mã thông báo xác thực.

Bước 4: Truy cập ứng dụng khác: Khi người dùng truy cập vào một ứng dụng hoặc hệ thống khác, ứng dụng đó sẽ gửi yêu cầu đến hệ thống SSO để xác thực người dùng.

Bước 5: Kiểm tra xác thực: Hệ thống SSO kiểm tra phiên làm việc hoặc mã thông báo xác thực của người dùng. Nếu phiên làm việc hoặc mã thông báo hợp lệ, hệ

thống SSO chứng thực người dùng và chuyển tiếp yêu cầu đến ứng dụng hoặc hệ thống yêu cầu.

Bước 6: Cấp quyền truy cập: Ứng dụng hoặc hệ thống nhận được xác thực từ hệ thống SSO và cho phép người dùng truy cập vào tài nguyên hoặc chức năng tương ứng.

4.4 Phân loại

- **SAML**: SAML là một giao thức phổ biến trong việc cung cấp SSO cho các ứng dụng web. Nó sử dụng các thông điệp XML để chuyển đổi thông tin xác thực giữa các thành phần trong hệ thống SSO, bao gồm người dùng, nhà cung cấp dịch vụ và nhà cung cấp xác thực. SAML cho phép người dùng đăng nhập một lần duy nhất vào hệ thống SSO, sau đó có thể truy cập vào các ứng dụng khác mà không cần phải đăng nhập lại.
- **OAuth (Open Authorization):** OAuth là một giao thức phổ biến trong việc ủy quyền và chia sẻ tài nguyên trên Internet. Nó không chỉ hỗ trợ việc xác thực người dùng mà còn cho phép người dùng ủy quyền truy cập vào tài nguyên của mình từ các ứng dụng và dịch vụ khác. OAuth sử dụng mã thông báo (token) để xác thực và ủy quyền người dùng và cung cấp khả năng truy cập an toàn và kiểm soát cho các ứng dụng và API.
- **OIDC (OpenID Connect):** OIDC là một giao thức xác thực dựa trên OAuth 2.0. Nó kết hợp khả năng ủy quyền và xác thực người dùng từ OAuth với các tính năng quản lý danh tính mở rộng. OIDC cung cấp một lớp bổ sung của thông tin xác thực và thông tin người dùng, cho phép ứng dụng xác thực người dùng và chứng nhận thông tin về người dùng từ nhà cung cấp dịch vụ.
- **Kerberos:** Kerberos là một giao thức xác thực mạng phân tán được sử dụng rộng rãi trong môi trường doanh nghiệp. Nó cung cấp khả năng xác thực và ủy quyền cho người dùng trong mạng nội bộ. Sử dụng mô hình máy chủ trung tâm, Kerberos cho phép người dùng đăng nhập một lần duy nhất và sau đó có thể truy cập vào các tài nguyên trong mạng mà không cần phải đăng nhập lại. Kerberos sử dụng các phiên làm việc (tickets) để xác thực và ủy quyền người dùng.

4.5 Kết luận

SSO không chỉ mang lại tiện lợi và hiệu quả trong quản lý đăng nhập mà còn đóng vai trò quan trọng trong việc giảm chi phí và tăng cường an toàn thông tin. Qua đó, hy vọng rằng độc giả đã có cái nhìn rõ ràng hơn về cách SSO hoạt động và ảnh hưởng của nó đối với môi trường công nghiệp và doanh nghiệp hiện đại.

Chương 5

Basic Authentication và Session-based Authentication

5.1 Basic Authentication

5.1.1 Định nghĩa

Định nghĩa

HTTP Basic Authentication yêu cầu client cung cấp username và password mỗi lần gọi request. Hiểu một cách đơn giản thì nó là phương thức để xác thực người dùng khi truy cập tài nguyên thông qua HTTP(s). Thông tin đăng nhập được gửi kèm theo mỗi request.

5.1.2 Cấu trúc Header

Cấu trúc header sẽ có thêm:

```
Authorization: Basic <Base64 encode {username:password}>
```

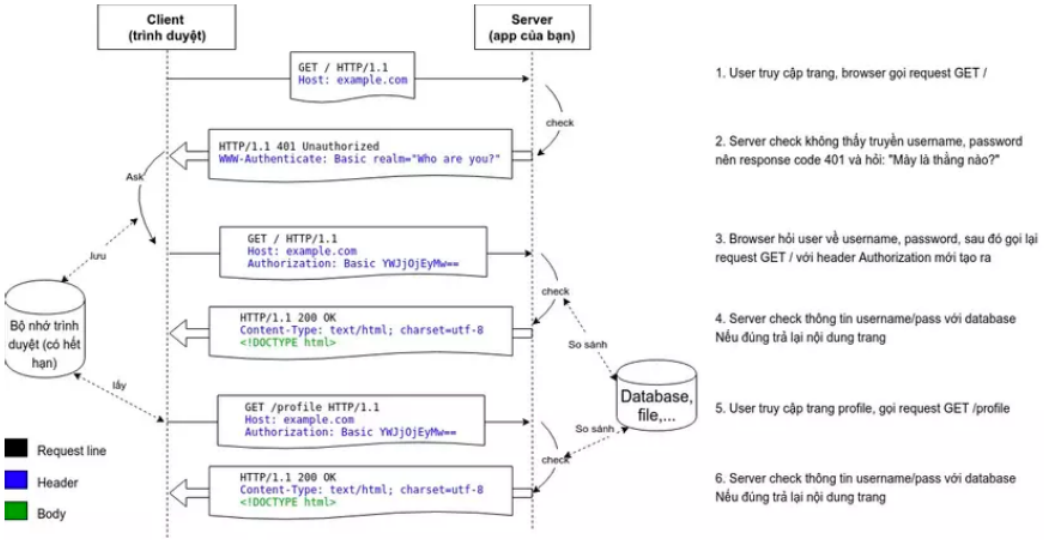
Ví dụ về Basic Authentication Header

Giả sử username là `admin` và password là `password123`:

```
# Chuoi goc: admin:password123
# Sau khi ma hoa: YWRtaW46cGFzc3dvcmQxMjM=

Authorization: Basic YWRtaW46cGFzc3dvcmQxMjM=
```

5.1.3 Cách hoạt động



Hình 5.1: Sơ đồ hoạt động của Basic Authentication

5.1.4 Ưu điểm

Ưu điểm của Basic Authentication:

- **Đơn giản**: Do đó được hầu hết các trình duyệt, webserver (nginx, apache,...) hỗ trợ. Bạn có thể dễ dàng config cho webserver sử dụng Basic Auth với một vài dòng config.
- **Dễ dàng kết hợp**: Do đã được xử lý mặc định trên trình duyệt và webserver thông qua truyền tải HTTP header, các bạn có thể dễ dàng kết hợp phương pháp này với các phương pháp sử dụng cookie, session, token,...
- **Không cần lưu trữ phía server**: Không cần quản lý session hay token ở phía server.

5.1.5 Nhược điểm

Nhược điểm của Basic Authentication:

- **Username/password dễ bị lộ**: Do mỗi request đều phải truyền username và password nên sẽ tăng khả năng bị lộ qua việc bắt request, log server,...
- **Không thể logout**: Vì việc lưu username, password dưới trình duyệt được thực hiện tự động và không có sự can thiệp của chủ trang web. Do vậy không có cách nào logout được người dùng ngoại trừ việc tự xóa lịch sử duyệt web hoặc hết thời gian lưu của trình duyệt.
- **Không thân thiện với người dùng**: Việc hiển thị hộp thoại đăng nhập cũng như thông báo lỗi của trình duyệt, như các bạn đã biết là vô cùng nhàm chán, không chứa đựng nhiều thông tin cho người dùng.
- **Bảo mật kém**: Thông tin đăng nhập chỉ được mã hóa Base64 (không phải encryption) nên dễ dàng giải mã.

5.2 Session-based Authentication

5.2.1 Định nghĩa

Định nghĩa

Session-based Authentication là cơ chế đăng nhập người dùng dựa trên việc tạo ra session của người dùng ở phía server. Sau quá trình xác thực người dùng thành công (username/password,...) thì phía server sẽ tạo và lưu ra một session chứa thông tin của người dùng đang đăng nhập và trả lại cho client session ID để truy cập session cho những request sau.

5.2.2 Nơi lưu trữ

Tại Server

- Lưu dữ liệu của session trong database

- Lưu trong file hệ thống
- Lưu trong RAM (memory)
- Sử dụng Session ID để tìm kiếm thông tin session

Tại Client

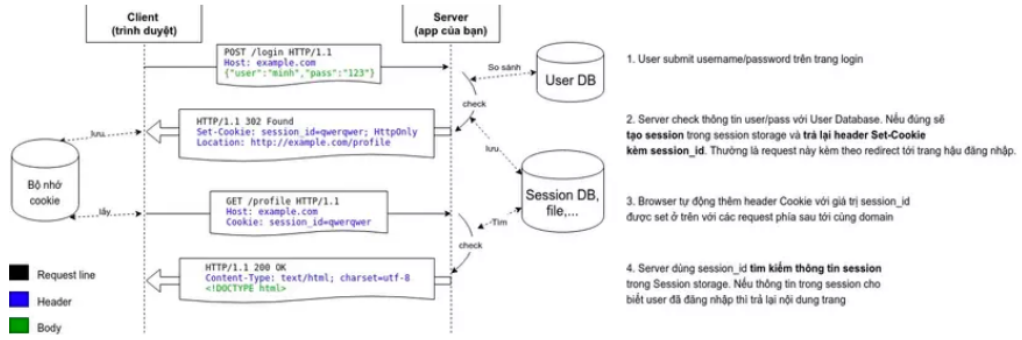
- Lưu Session ID trong cookie của browser
- Lưu trong URL trang web
- Lưu trong form field ẩn

Truyền tải

Session ID sẽ xuất hiện trong các HTTP request tiếp theo:

- **Cookie**: `Cookie: SESSION_ID=abc`
- **URL**: `/profile?session_id=abc`
- **Body**: Form field ẩn

5.2.3 Cách hoạt động



Hình 5.2: Sơ đồ hoạt động của Session-based Authentication

5.2.4 Ưu điểm

Ưu điểm của Session-based Authentication:

- **Thông tin được giấu kín**: Client chỉ được biết tới Session ID thường là một chuỗi random không mang thông tin gì của người dùng, còn mọi thông tin khác của phiên đăng nhập hay người dùng hiện tại đều được lưu phía server nên cơ chế này giữ kín được thông tin của người dùng trong quá trình truyền tải.
- **Dung lượng truyền tải nhỏ**: Bởi vì tự thân Session ID không mang theo thông tin gì, thông thường chỉ là một chuỗi ký tự unique khoảng 20-50 ký tự, do vậy việc gắn Session ID vào mỗi request không làm tăng nhiều độ dài request, do đó việc truyền tải sẽ diễn ra dễ dàng hơn.
- **Không cần tác động client**: Theo mình thì để sử dụng cơ chế session này bạn chủ yếu chỉ cần sửa phía server. Client mà cụ thể là browser hầu như không cần phải xử lý gì thêm bởi đã được tích hợp tự động (đối với cookie), hoặc response trả về của server đã có sẵn (đối với session ID ở URL hoặc hidden form).
- **Fully-controlled session**: Tính chất này có thể cho phép hệ thống quản trị TẤT CẢ các hoạt động liên quan tới phiên đăng nhập của người dùng như thời gian login, force logout,...

5.2.5 Nhược điểm

Nhược điểm của Session-based Authentication:

- **Chiếm nhiều bộ nhớ**: Với mỗi phiên làm việc của user, server sẽ lại phải tạo ra một session và lưu vào bộ nhớ trên server. Số data này có thể còn lớn hơn cả user database của bạn do mỗi user có thể có vài session khác nhau. Do vậy việc tra cứu đối với các hệ thống lớn nhiều người dùng sẽ là vấn đề.
- **Khó scale**: Vì tính chất stateful của việc lưu session data ở phía server, do đó bạn sẽ khó khăn hơn trong việc scale ngang ứng dụng, tức là nếu bạn chạy ứng dụng của bạn ở 10 máy chủ, hay 10 container, thì:
  - Bạn phải dùng chung chỗ lưu session, hoặc
  - Nếu không dùng chung bộ nhớ session thì phải có giải pháp để ghi nhớ user đã kết nối tới server nào

  Nếu không, rất có thể chỉ cần ấn refresh, user kết nối với server khác khi cân bằng tải là sẽ như chưa hề có cuộc login.

- **Phụ thuộc domain**: Vì thường sử dụng cookie, mà cookie lại phụ thuộc vào domain, do vậy khả năng sử dụng phiên đăng nhập của bạn sẽ bị giới hạn ở đúng domain được set cookie. Điều này không phù hợp với các hệ thống phân tán hoặc tích hợp vào ứng dụng bên thứ 3.
- **CSRF (Cross-Site Request Forgery)**: Session ID thường được lưu vào Cookie, và cookie mới là thứ dễ bị tấn công kiểu này. Vì cookie được tự động gắn vào các request tới domain của bạn.

Ví dụ về tấn công CSRF

Kịch bản tấn công CSRF:

1. User vừa login vào `my-bank.com` và được set cookie: `session_id=123`
2. User vào trang web `taolahacker.com` xem nội dung
3. Trên `taolahacker.com`, hacker ngầm gửi một request ajax tới domain `my-bank.com`
4. Vì browser tự động thêm cookie `session_id=123` vào request ajax trên, do vậy request của hacker có thể thao tác mọi thứ NHƯ USER

Cách phòng chống:

- Sử dụng CSRF Token
- Kiểm tra Referer header
- Sử dụng SameSite cookie attribute

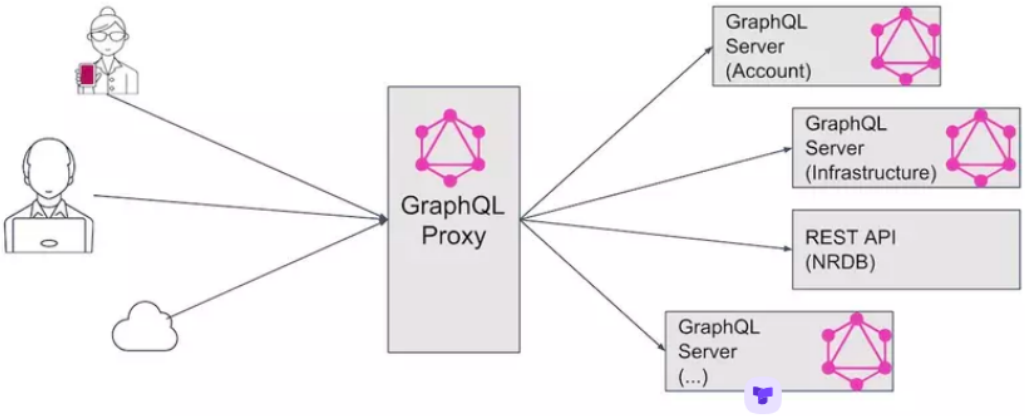
Chương 6

GraphQL

6.1 Định nghĩa

Định nghĩa

GraphQL là ngôn ngữ thao tác và truy vấn dữ liệu nguồn mở cho API, cung cấp cho client 1 cách thức dễ dàng để request chính xác những gì họ cần, giúp việc phát triển API dễ dàng hơn theo thời gian. GraphQL được Facebook phát triển nội bộ vào năm 2012 trước khi phát hành công khai vào năm 2015.



Hình 6.1: Sơ đồ hoạt động của Session-based Authentication

GraphQL bao gồm 3 điểm đặc trưng bao gồm:

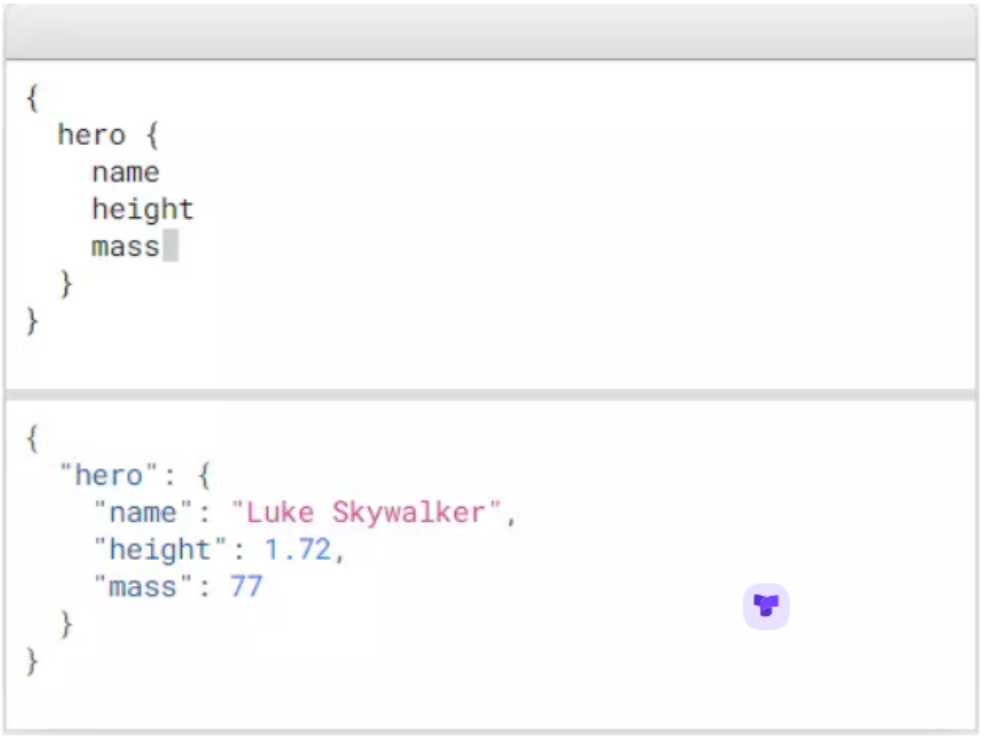
- Cho phép client xác định chính xác những dữ liệu gì họ cần
- GraphQL làm cho việc tổng hợp dữ liệu từ nhiều nguồn dễ dàng hơn
- Sử dụng một type system để khai báo dữ liệu.

6.2 Trả về chính xác những gì bạn gửi request

Khi bạn gửi một request GraphQL đến API, bạn sẽ nhận được **chính xác** những gì bạn yêu cầu trong request, không hơn không kém. Điều này giúp cho các truy vấn GraphQL luôn trả về kết quả có thể dự đoán được.

Lợi ích:

- Các ứng dụng sử dụng GraphQL rất nhanh và ổn định vì GraphQL kiểm soát dữ liệu mà nó nhận được, chứ không phải máy chủ.
- Giảm tải cho mạng và client vì chỉ lấy đúng dữ liệu cần thiết.
- Giúp phát triển frontend linh hoạt, chủ động hơn với dữ liệu.



Hình 6.2: Ví dụ

6.3 Nhận nhiều dữ liệu trong một request duy nhất

Các câu query GraphQL không chỉ cho phép truy xuất các thuộc tính của một đối tượng, mà còn có thể truy vấn đồng thời nhiều đối tượng liên quan trong cùng một request.

Lợi ích:

- Trong khi API REST truyền thống phải gửi nhiều request tới các URL khác nhau để lấy đủ dữ liệu, thì với GraphQL, bạn chỉ cần một request duy nhất để lấy tất cả thông tin ứng dụng cần.
- Tăng hiệu suất xử lý, giảm độ trễ và số lần round-trip giữa client và server.
- Các ứng dụng sử dụng GraphQL có tốc độ xử lý rất nhanh, ngay cả trên các kết nối mạng chậm.

6.4 Hướng dẫn sử dụng GraphQL

6.4.1 Cấu trúc cơ bản

GraphQL có ba loại operation chính:

1. Query (Đọc dữ liệu)

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

2. Mutation (Thay đổi dữ liệu)

```
mutation {
  createUser(input: {
```

```
    name: "John Doe"
    email: "john@example.com"
  }) {
    id
    name
    email
  }
}
```

3. Subscription (Lắng nghe thay đổi)

```
subscription {
  postAdded {
    id
    title
    author {
      name
    }
  }
}
```

6.4.2 Schema Definition

Ví dụ về GraphQL Schema

```
type User {
  id: ID!
  name: String!
  email: String!
  posts: [Post!]!
  createdAt: DateTime!
}

type Post {
  id: ID!
  title: String!
  content: String!
  author: User!
  createdAt: DateTime!
}

type Query {
  user(id: ID!): User
  users: [User!]!
  post(id: ID!): Post
  posts: [Post!]!
}

type Mutation {
  createUser(input: CreateUserInput!): User!
  updateUser(id: ID!, input: UpdateUserInput!): User!
  deleteUser(id: ID!): Boolean!
  createPost(input: CreatePostInput!): Post!
}

input CreateUserInput {
  name: String!
  email: String!
}

input UpdateUserInput {
  name: String
```

6.4.3 Cách thực hiện Query

Query đơn giản

```
# Request
query {
  user(id: "1") {
    name
    email
  }
}

# Response
{
  "data": {
    "user": {
      "name": "John Doe",
      "email": "john@example.com"
    }
  }
}
```

Query với tham số

```
# Request  v i  variables
query GetUser($userId: ID!) {
  user(id: $userId) {
    name
    email
    posts(limit: 5) {
      title
      createdAt
    }
  }
}

# Variables
```

```
{
  "userId": "1"
}
```

Query lồng nhau (Nested Query)

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

6.4.4 Fragments và Aliases

Fragments - Tái sử dụng trường

```
fragment UserInfo on User {
  id
  name
  email
  createdAt
}

query {
  user1: user(id: "1") {
    ...UserInfo
  }
```

```
  user2: user(id: "2") {
    ...UserInfo
  }
}
```

Aliases - Đặt tên cho trường

```
query {
  currentUser: user(id: "1") {
    fullName: name
    emailAddress: email
  }
  otherUser: user(id: "2") {
    fullName: name
    emailAddress: email
  }
}
```

6.5 So sánh GraphQL với REST API

6.5.1 Bảng so sánh tổng quan

Tiêu chí	REST API	GraphQL
Số endpoint	Nhiều URL khác nhau	Một endpoint duy nhất
Dữ liệu trả về	Cố định theo endpoint	Linh hoạt theo query
Over-fetching	Thường xảy ra	Không xảy ra
Under-fetching	Cần nhiều request	Một request duy nhất
Caching	Dễ dàng (HTTP caching)	Phức tạp hơn
Learning curve	Đơn giản	Phức tạp hơn
Versioning	Cần version API	Không cần versioning
File upload	Dễ dàng	Phức tạp
Real-time	Cần WebSocket riêng	Có sẵn subscription

Bảng 6.1: So sánh chi tiết REST API và GraphQL

6.5.2 Ưu và nhược điểm

REST API

Ưu điểm của REST:

- **Đơn giản**: Dễ hiểu, dễ học, dễ implement
- **Caching**: HTTP caching hoạt động tốt với GET requests
- **Tooling**: Có nhiều tools hỗ trợ (Postman, Swagger, etc.)
- **Stateless**: Mỗi request độc lập
- **Mature**: Đã được sử dụng rộng rãi, có nhiều best practices
- **File upload**: Dễ dàng upload files

Nhược điểm của REST:

- **Over-fetching**: Lấy nhiều dữ liệu không cần thiết
- **Under-fetching**: Cần nhiều requests để lấy đủ dữ liệu
- **Versioning**: Cần quản lý versions của API
- **Multiple endpoints**: Phải quản lý nhiều URLs
- **Documentation**: Cần maintain documentation riêng

GraphQL

Ưu điểm của GraphQL:

- **Precise data fetching**: Lấy đúng dữ liệu cần thiết
- **Single request**: Một request cho nhiều tài nguyên
- **Strong typing**: Type safety với schema
- **Real-time**: Subscriptions cho real-time updates
- **Introspection**: Self-documenting API
- **Developer experience**: Tools tốt như GraphiQL, Apollo DevTools

- **Versionless**: Không cần versioning API

Nhược điểm của GraphQL:

- **Complexity**: Learning curve cao hơn REST
- **Caching**: HTTP caching không hiệu quả, cần custom solutions
- **File upload**: Phức tạp hơn so với REST
- **Query complexity**: Có thể tạo ra queries rất phức tạp
- **N+1 problem**: Cần giải quyết vấn đề N+1 queries
- **Server resources**: Có thể tốn nhiều tài nguyên server

6.6 Khi nào nên sử dụng GraphQL vs REST

6.6.1 Nên sử dụng GraphQL khi:

- **Mobile applications**: Cần optimize bandwidth và battery
- **Multiple clients**: Có nhiều client với nhu cầu dữ liệu khác nhau
- **Rapid development**: Cần phát triển frontend nhanh chóng
- **Real-time features**: Cần real-time updates
- **Complex data relationships**: Dữ liệu có quan hệ phức tạp
- **Microservices**: Cần aggregate data từ nhiều services

6.6.2 Nên sử dụng REST khi:

- **Simple applications**: Ứng dụng đơn giản với CRUD operations
- **File operations**: Cần upload/download files thường xuyên
- **Caching requirements**: Cần HTTP caching mạnh mẽ
- **Team expertise**: Team đã quen với REST
- **Third-party integrations**: Cần integrate với nhiều third-party APIs
- **Compliance**: Cần tuân thủ các chuẩn REST có sẵn

6.7 Kết luận

GraphQL và REST đều có những ưu nhược điểm riêng và phù hợp với những tình huống khác nhau. Việc lựa chọn giữa chúng phụ thuộc vào:

- **Yêu cầu của dự án**: Độ phức tạp, performance requirements
- **Kinh nghiệm của team**: Khả năng học và maintain
- **Ecosystem**: Tools và libraries có sẵn
- **Long-term vision**: Hướng phát triển của sản phẩm

Trong nhiều trường hợp, có thể sử dụng cả hai approach trong cùng một hệ thống: REST cho những operations đơn giản và GraphQL cho những queries phức tạp.