# BACKEND

## ▼ Microservice ≠ Monolithic

### ▼ Monolithic

- là kiến trúc phần mềm dạng nguyên khối, nghĩa là mọi tính năng sẽ nằm trong một project.
- Giả sử mình có một project web bán hàng triển khai theo kiến trúc monolithic, thì các module như khách hàng, đơn hàng, sản phẩm,… sẽ được gói gọn trong project đó.
- **Ưu điểm:**
  - Dễ phát triển vì các stack công nghệ thống nhất ở tất cả các layer.
  - Dễ test do toàn bộ project được đóng gói trong một package nên dễ dàng chạy test integrantion và test end-to-end.
  - Deploy đơn giản và nhanh chóng nếu bạn chỉ có một package để bận tâm.
  - Dễ scale vì chúng ta có thể có nhiều instance cho load banlancer.
  - Yêu cầu team size nhỏ cho việc maintain app.
  - Team member có thể chia sẻ ít nhiều về skill.
  - Tech stack đơn giản và đa số là dễ học.
  - Phát triển ban đầu nhanh hơn do đó có thể đem sale hoặc marketing nhanh hơn.
  - Yêu cầu cơ sở hạ tầng đơn giản. Thậm chí một container đơn giản cũng đủ để chạy ứng dụng.
- **Nhược điểm:**
  - Các component được liên kết chặt chẽ với nhau dẫn đến side effect không mong muốn như khi thay đổi một component ảnh hưởng đến một component khác.
  - Theo thời gian thì project trở nên phức tạp và lớn dần. Các tính năng mới sẽ mất nhiều thời gian hơn để phát triển và tái cấu trúc các tính năng hiện có sẽ nhiều khó khăn hơn.
  - Toàn bộ ứng dụng cần được triển khai lại cho bất kỳ thay đổi nào.
  - Không hề dễ để hiểu project do các module liên quan chặt chẽ lẫn nhau. Một issue nhỏ cũng có thể làm chết toàn bộ ứng dụng.
  - Áp dụng công nghệ mới khó khăn vì toàn bộ ứng dụng phải thay đổi. Do đó nhiều ứng dụng một khối thường phụ thuộc một công nghệ cũ và lỗi thời.
  - Các service quan trọng không thể scale riêng dẫn đến lãng phí tài nguyên vì toàn bộ ứng dụng phải scale theo.
  - Các ứng dụng một khối lớn sẽ có thời gian khởi động lâu và tốn tài nguyên CPU cũng như bộ nhớ.
  - Các team tham gia vào dự án phải phụ thuộc lẫn nhau và tất khó để mở rộng quy mô team.

### ▼ Microservice

- là kiến trúc chia dự án thành nhiều service nhỏ
- Các service trong kiến trúc microservice là độc lập với nhau, chúng có thể có kiến trúc khác nhau, sử dụng công nghệ khác nhau hoặc thậm chí có database riêng

- Chúng trao đổi thông tin với nhau thông qua môi trường mạng (có thể bằng end point Restful API hoặc các message queue)
- **Ưu điểm:**
  - Các component có kết nối lỏng lẻo dẫn đến dễ cách ly, dễ test và khởi động nhanh.
  - Vòng đời phát triển nhanh hơn. Tính năng mới được phát triển nhanh hơn và tính năng cũ được cấu trúc lại dễ hơn.
  - Các service có thể deploy độc lập nên ứng dụng dễ đọc, dễ tạo các bản vá hơn.
  - Những issue, ví dụ liên quan đến memory leak một trong các service, bị cô lập và có thể không làm sập ứng dụng.
  - Việc áp dụng các công nghệ mới dễ hơn. Các component có thể được nâng cấp độc lập với nhau.
  - Các mô hình scale phức tạp và hiệu quả hơn có thể được thiết lập. Các service quan trọng có thể scale hiệu quả hơn. Các component riêng sẽ khởi động nhanh hơn và cải thiện thời gian khởi động của cả hệ thống.
  - Các team tham gia sẽ ít phụ thuộc lẫn nhau. Kiến trúc này rất thích hợp cho các đội Agile.
- **Nhược điểm:**
  - Phức tạp hơn về mặt tổng thể vì các component khác nhau có các stack công nghệ khác nhau nên buộc team phải tập trung đầu tư thời gian để theo kịp công nghệ.
  - Khó thực hiện test end-to-end và integration test vì có nhiều stack công nghệ khác nhau.
  - Deploy toàn bộ ứng dụng phức tạp hơn vì có nhiều container và nền tảng ảo hóa liên quan.
  - Ứng dụng được scale hiệu quả hơn nhưng thiết lập nâng cấp sẽ phức tạp hơn vì nó sẽ yêu cầu nâng cao nhiều tính năng như truy tìm dịch vụ (service discovery), định tuyến DNS,…
  - Yêu cầu một team-size lớn để maintain ứng dụng vì có nhiều component và công nghệ khác nhau.
  - Các thành viên trong team chia sẻ các skill khác nhau dựa trên component họ làm nên sẽ tạo ra sự khó khăn khi thay thế và chia sẻ kiến thức.
  - Stack công nghệ phức tạp và khó để học hơn.
  - Thời gian phát triển ban đầu là chậm nên thời gian để có thể làm marketing lâu hơn.
  - Yêu cầu cơ sở hạ tầng phức tạp. Thông thường sẽ yêu cầu nhiều container (Docker) và nhiều máy JVM để chạy.

▼ **Rest ≠ graph**

- https://2kvn.com/graphql-vs-rest-apis-p5f31353330
- https://viblo.asia/p/so-sanh-graphql-voi-rest-V3m5WLv8KO7
- https://codelearn.io/sharing/graphql-va-uu-diem-so-voi-rest-api

▼ **CORS**

- (Cross-Origin Resource Sharing) là một kĩ thuật được sinh ra để làm cho việc tương tác giữa client và server được dễ dàng hơn, nó cho phép JavaScript ở một trang web có thể tạo request lên một REST API được host ở một domain khác.
- **Cơ chế hoạt động:**
  - Trong trường hợp đơn giản nhất, phía client (tức là cái web app chạy ở browser đó) sẽ tạo request GET, POST, PUT, HEAD, etc để yêu cầu server làm một việc gì đó.

    - Những request này sẽ được đính kèm một header tên là `Origin` để chỉ định origin của client code (giá trị của header này chính là domain của trang web).
    - Server sẽ xem xét `Origin` để biết được nguồn này có phải là nguồn hợp lệ hay không.
    - Nếu hợp lệ, server sẽ trả về response kèm với header `Access-Control-Allow-Origin`.
    - Header này sẽ cho biết xem client có phải là nguồn hợp lệ để browser tiếp tục thực hiện quá trình request.
    - Trong trường hợp thông thường, `Access-Control-Allow-Origin` sẽ có giá trị giống như `Origin`, một số trường hợp giá trị của `Access-Control-Allow-Origin` sẽ nhìn giống giống như Regex hay chỉ đơn giản là `*`
    - Tuy nhiên thì cách dùng `*` thường được coi là không an toàn, ngoại trừ trường hợp API của bạn được public hoàn toàn và ai cũng có thể truy cập được.
    - Nếu không có header `Access-Control-Allow-Origin` hoặc giá trị của nó không hợp lệ thì browser sẽ báo lỗi.
- **Pre-flight request:**
    - Khi thực hiện những request ảnh hưởng đến data như: POST, PUT, DELETE, ... thì browser sẽ tự động thực hiện một request gọi là `preflight request` trước khi thực sự thực hiện request để kiểm tra xem phía server đã thực hiện CORS hay chưa, cũng như để biết được rằng request này có hợp lệ hay không.
    - Ngoài ra thì nếu bạn có thêm những custom header vào trong request thì việc gửi một `preflight request` cũng là cần thiết.
    - Preflight request được gửi lên server với dạng là `OPTIONS` (đây là lý do tại sao khi bạn debug ở client bạn thường thấy có hai request giống nhau nhưng khác request method, một cái là `OPTIONS` một cái là method thật sự bạn muốn gửi).
    - **Ví dụ:** bạn muốn gửi request `DELETE` lên server.
        - Browser sẽ tự tạo một request `OPTIONS` sẽ hỏi xem server có cho phép việc gửi request `DELETE` hay không.
        - Nếu server cho phép, nó sẽ gửi về response đính kèm những header như `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, `Access-Control-Max-Age`, etc.
            - Access-Control-Allow-Methods: mô tả những method nào client có thể gửi đi.
            - Access-Control-Max-Age: mô tả thời gian hợp lệ của `preflight request`, nếu quá hạn, browser sẽ tự tạo một `preflight request` mới.
        - Sau đó browser sẽ có thể gửi request `DELETE` và nhận response như bình thường. Và ngược lại, browser sẽ báo lỗi

## ▼ Middleware

- Middleware đóng vai trò trung gian giữa **request/response** (tương tác với người dùng) và các xử lý logic bên trong web server.
- Middleware sẽ là các hàm được dùng để tiền xử lý, lọc các request trước khi đưa vào xử lý logic hoặc điều chỉnh các response trước khi gửi về cho người dùng.
- Các hàm middleware có thể thực thi ở đầu, giữa, hoặc cuối vòng đời của một request. Trong stack các middleware function luôn được thêm vào theo thứ tự mà chúng ta mong muốn ngay ban đầu.
- Một số middleware phổ biến trong nodejs như: route, cors, auth, logger, helmet, ...

## ▼ Helmet

- Helmet là một package được viết để giúp bạn bảo vệ ứng dụng của mình khỏi những lỗ hổng đã biết bằng cách thiết lập các Http headers một cách phù hợp.

- Thực tế thì Helmet chỉ là một tập hợp các Middleware nhỏ làm nhiệm vụ thiết lập các Http headers liên quan đến bảo mật.
- Nó giúp ẩn đi những thông tin không cần thiết trong header và thiết lập lại một số thuộc tính giúp cho server an toàn hơn.

▼ **Cookies**

- **Cookie** là những tập tin một trang web gửi đến máy người dùng và được lưu lại thông qua trình duyệt khi người dùng truy cập trang web đó.
- Cookie được dùng để lưu trữ với rất nhiều mục đích như lưu phiên đăng nhập, hoạt động của người dùng khi truy cập trang web.
- **Cookie** có nhiều loại khác nhau và phân chia theo từng mục đích sử dụng. Một số cookies phổ biến như:
  - **Session Cookie:** chỉ tồn tại tạm thời trong bộ nhớ của trình duyệt và sẽ bị trình duyệt tự xóa khi người dùng hết phiên đăng nhập, thông thường loại cookie này không có thời hạn.
  - **Third-party cookie :** thông thường cookie của trang web sẽ trùng với thanh địa chỉ của trình duyệt nhưng có một vài trường hợp sử dụng cookie bên thứ 3 có tên miền khác với url trang web
  - **Secure cookie:** một loại cookie HTTP có bộ thuộc tính secure giới hạn phạm vi của cookie đối với trình duyệt web.

▼ **Sessions**

- Session dùng để lưu trữ dữ liệu trên Server và đồng thời nó sẽ có một đoạn code dữ liệu được lưu trữ ở client (cookie).
- Một số cách lưu trữ sessions ở server:
  - **Cookie**: Chúng ta có thể store sesion trên cookie session nodejs mỗi trình duyệt nhưng chú ý rằng tất cả đều nằm ở Clients.
  - **Memory Cache**: Như chúng ta đã biết, Cache được lưu trữ trong bộ nhớ. Chúng ta cũng có thể sử dụng thêm những cache module như Redis và Memcached.
  - **Database**: (ít được sử dụng)

▼ **WebSocket**

- **WebSocket** là công nghệ hỗ trợ giao tiếp hai chiều giữa client và server bằng cách sử dụng một TCP socket để tạo một kết nối hiệu quả và ít tốn kém.
- Mặc dù được thiết kế để chuyên sử dụng cho các ứng dụng web, lập trình viên vẫn có thể đưa chúng vào bất kì loại ứng dụng nào

▼ **Cache**

- Khi cache client cần truy cập data, việc đầu tiên là check cache. Khi request data tìm thấy dữ liệu cần thiết trong Cache, nó được gọi là ***Cache hit***. Tỉ lệ của kết quả tìm kiếm **cache hit** được biết đến như là *cache hit rate* hay *ratio*.
- Nếu việc tìm kiếm data không thành công, nó gọi là ***Cache miss*** - từ đây dữ liệu sẽ được kéo từ bộ nhớ chính sang bộ nhớ cache. Việc giữ dữ liệu nào cần, hay xoá khỏi bộ nhớ điệm để nhường chỗ cho dữ liệu mới sẽ tùy thuộc vào thuật toán mà system sử dụng.
- **Cache Replacement Policy** nghĩa nôm na là các thuật toán để thay thế giá trị hoặc xóa các giá trị cũ để thêm giá trị mới vào.
  - LRU, LFU, MRU, FIFO, LIFO, ...

- **Memoization** là một kỹ thuật tối ưu hóa, giúp tăng tốc các ứng dụng bằng cách lưu trữ kết quả của các lệnh gọi hàm (mà các hàm này được gọi là `expensive function`) và trả về kết quả được lưu trong bộ nhớ cache khi có cùng một đầu vào yêu cầu (đã được thực thi ít nhất 1 lần trước đó rồi).
- Một số thuật toán điều khiển cache:
  - **Least Frequently Used (LFU)** : theo dõi tần suất truy cập một dữ liệu. Các dữ liệu có số lần truy cập thấp nhất được loại bỏ đầu tiên.
  - **Least Recently Used (LRU)** : lưu trữ các dữ liệu được truy cập gần đây gần đầu bộ đệm. Khi bộ đệm đạt đến giới hạn của nó, các liệu được truy cập gần đây nhất sẽ bị xóa.
  - **Most Recently Used (MRU)** : loại bỏ các dữ liệu truy cập gần đây nhất đầu tiên. Cách tiếp cận này là tốt nhất khi các data cũ có nhiều khả năng được sử dụng.
- Các loại cache hay sử dụng:
  - **cache server** : Một dedicated network server hoặc dịch vụ chuyên dụng hoạt động như một máy chủ lưu các trang web hoặc nội dung internet khác cục bộ. Một cache server đôi khi được gọi là proxy cache.
  - **Cache memory**: Random access memory, hay còn được gọi là **RAM**, Cache memory thường được gắn trực tiếp vào CPU và được sử dụng để cập nhanh các dữ liệu trong CPU.
  - **Flash cache**: Temporary storage of data on NAND flash memory chips -- thường được sử dụng ở **solid-state drives (SSDs) ** , thực hiện các request dữ liệu nhanh hơn có thể nếu bộ đệm nằm trên ổ đĩa cứng truyền thống (HDD)

  https://pymi.vn/blog/memoization/

▼ **Stream, Buffer, Pipe**

- **Stream**: là các đối tượng cho phép bạn đọc dữ liệu từ một nguồn và ghi dự liệu đến một đích (*là một chuỗi dữ liệu sẵn có qua thời gian, hay có thể hình dung stream là một đối tượng chứa dữ liệu sẽ được truyền đi từ nơi này đến nơi khác.*)
  - Có 4 loại Stream trong Nodejs:
    - **Readable**: Là Stream được sử dụng để cho hoạt động đọc
    - **Writable**: Là Stream được sử dụng cho hoạt động ghi
    - **Duplex**: Là Stream được sử dụng cho cả mục đích ghi và đọc
    - **Transform**: Đây là một kiểu Duplex Stream, khác ở chỗ là kết quả đầu ra được tính toán dựa trên dữ liệu bạn đã nhập vào.
  - Mỗi loại Stream là một sự thể hiện của đối tượng **EventEmitter** và ném một vài sự kiện tại các thời điểm khác nhau
  - **Piping Stream:** là một kỹ thuật. Với kỹ thuật này, chúng ta cung cấp kết quả đầu ra của một Stream để làm dữ liệu đầu vào cho một Stream khác. Không có giới hạn nào về hoạt động Piping này, tức là quá trình trên có thể vẫn tiếp tục.
  - **Chaining Stream:** là một kỹ thuật để kết nối kết quả đầu ra của một Stream tới một Stream khác và tạo một chuỗi bao gồm nhiều hoạt động Stream. Thường thì nó được sử dụng với các hoạt động Piping
- **Buffer**: là một vùng dự trữ tạm thời chứa các dữ liệu đang được chuyển từ nơi này đến nơi khác.
  - Buffer có kích thước xác định và giới hạn.
  - Kích thước của buffer được xác định bằng những thuật toán cho từng trường hợp cụ thể.

- Buffer là một kỹ thuật được phát triển nhằm ngăn chặn sự tắc nghẽn dữ liệu khi truyền từ nơi này đến nơi khác.
- Ví dụ:
  - Trong thực tế, khi chúng ta xem một đoạn phim trên mạng.
  - Nếu đường truyền mạng chúng ta đủ nhanh thì tốc độ stream video sẽ kịp thời làm đầy các buffer (vùng nhớ tạm thời trên RAM) và đoạn dữ liệu này sẽ được gửi đến trình media player để chạy đoạn dữ liệu vừa được làm đầy trong buffer.
  - Trong lúc phát nội dung đó, buffer sẽ trống và lại được làm đầy.
  - Cứ như vậy cho đến khi kết thúc stream.

▼ **Queue**

- **Khái niệm**:
  - Message Queue nôm na là Queue (hàng đợi), chứa Message (Tin nhắn) như hộp thư.
  - Và nó cho phép các thành phần/service trong một hệ thống (hoặc nhiều hệ thống), trao đổi thông tin cho nhau.
  - Ý nghĩa của queue (hàng đợi) là nó thực hiện việc lấy message theo cơ chế vào trước thì ra trước ( First In First Out ).
- **Thành phần của 1 MQ:**
  - **Message**: Thông tin được gửi đi (có thể là text, binary hoặc JSON)
  - **Message Queue**: Nơi chứa những message này, cho phép producer và consumer có thể trao đổi với nhau
  - **Producer**: Chương trình/service tạo ra thông tin, đưa thông tin vào message queue
  - **Consumer**: Chương trình/service nhận message từ message queue và xử lý
  - Một chương trình/service có thể **vừa là producer, vừa là consumer**
- **MQ giải quyết được các vấn đề:**
  - **Đảm bảo duration/recovery**: Do message đã được lưu trong queue, khi 1 service đang xử lý nhưng bị crash hoặc lỗi, ta không lo bị mất dữ liệu.
    - Vì có thể lấy message từ trong queue ra và chạy lại.
    - Trong 1 hệ thống có nhiều consumer, nếu 1, 2 consume bị crash cũng không làm sụp toàn hệ thống
  - **Phân tách hệ thống**: Giúp phân tách hệ thống thành nhiều service nhỏ hơn, mỗi service chỉ xử lý 1 chức năng nhất định
  - **Hỗ trợ rate limit, batching**: Trong nhiều trường hợp, năng lực xử lý hệ thống có hạn (chỉ có thể xử lý 300 đơn hàng/s).
    - Với message queue, ta có thể dần dần lấy đơn hàng trong queue ra xử lý, không sợ thất lại.
    - Hoặc thay vì mỗi lần gửi email mất thời gian lâu, ta có thể đợi message queue có yêu cầu gửi 200 email rồi gửi luôn 1 lượt.
  - **Dễ scaling hệ thống**: Vào giờ cao điểm, nhiều truy vấn, ta có thể tăng số lượng consumer lên để xử lý được nhiều messege hơn. Khi không cần ta có thể giảm lại.
- **Một số MQ phổ biến:**

    - **Kafka**
    - **RabittMQ**
    - **Bull**
    - ...

▼ **Single sign-on**

- **SSO** là viết tắt của Single Sign On, có nghĩa là bạn có thể đăng nhập vào một hệ thống trong một nhóm ứng dụng đa hệ thống (google, youtube, console...) và bạn có thể nhận ủy quyền trong tất cả các hệ thống khác mà không cần đăng nhập lại.
- *SSO là cơ chế cho phép người dùng có thể truy cập nhiều trang web, ứng dụng mà chỉ cần đăng nhập một lần. Một khi đã được định danh ở một trang website A, thì cũng sẽ được định danh tương tự ở website B mà không cần lặp lại thao tác đăng nhập*
- Ưu điểm:
    - Giảm số lượng username và pasword mà người dùng cần phải ghi nhớ
    - Giảm số lần phải nhập thông tin username và pasword
    - Rủi ro về việc lộ thông tin người dùng cũng được tiết chế lại
- Nhược điểm:
    - Chi phí phát triển khi thông qua service thứ ba
    - Phụ thuộc vào service bên ngoài
- SSO là một phần của hệ thống nhận dạng liên kết (**Federated Identity Glossary**), có liên quan chặt chẽ với việc xác thực thông tin người dùng. Nó sẽ định danh người dùng, và sau đó chia sẻ thông tin định danh được với các hệ thống con.
    - Hệ thông nhận dang liên kết là nơi tập trung và liên kết thông tin người dùng. Có 4 yếu tố nền tảng cấu thành:
        - Xác thực (Authentication)
        - Phân quyền (Authorization)
        - Trao đổi thông tin người dùng (User attributes exchange)
        - Quản lí người dùng (User management)
- **Cơ chế:**
    - https://www.google.com/search?q=single+sign+on+la+gi&oq=single&aqs=chrome.0.69i59j69i57j69i60l3.1218j0j1&sourceid=chrome&ie=UTF-8

▼ **JWT, auth**

- link

▼ **Single thread, multiple threads**

- https://viblo.asia/p/javascript-single-thread-lieu-da-loi-thoi-gAm5yxwkldb
- **Lấy một ví dụ để minh họa JS chạy trên môi trường trình duyệt web chrome là single thread?**

```
let loop = true;
```

```
setTimeout(() => {  // callback execution is unreachable because main thread is still busy with below infinite loop
 code.
    console.log('It will never reach here :ohh');
    loop = false;
});

while (loop) {
    console.log('loop', loop);  // Infinite loop
}

console.log('after loop');  // Unreachable code
```

- *Single thread theo hướng event-driven nó vậy đó fence, các hoạt động i/o như gọi API thì js nó không tự tạo thread để xử lý mà bắn cho thằng khác(trình duyệt) lo còn bản thân js nó ngồi chơi xơi nước đợi kết quả không à. Ngược lại nếu gọi 1 hàm đệ quy tính toán nặng trong main thread(js tự thân vận động) thì web nó treo đến khi hàm đó chạy xong thì thôi. Đấy mới chính là cái khẳng định js trên browser đơn thuần là single thread. Còn việc có thể tạo lập các service worker thì nó là runtime của browser, browser có hỗ trợ thì dùng, không thì thôi nên không thể nói nó là của js đc.*
-
-

▼ **pub & sub system**

- **Publish/subscribe messaging** là một pattern mà đặc trưng bởi việc gửi (publisher) data (message) mà ko chỉ định người nhận rõ ràng .
- Thay vào đó người gửi sẽ phân loại tin nhắn thành các lớp và bằng cách nào đó mà người nhận (subscriber) phải đăng kí vào lớp nhất định để nhận tin nhắn ở lớp đấy.
- Hệ thống pub/sub thường có một broker (nhà phân phối) , là trung tâm nơi mà các message được phân phối.

▼ **ACL, RBAC**

▼ **ACL**: là hình thức phân quyền dựa trên một danh sách các quyền truy cập.

- `Subject` có thể `Action` tới `Object`
- Dựa vào người dùng và nhóm người dùng
- **Ví dụ**:
  - Cho phép Nguyễn Văn A tạo bài viết

```
'Subject' : 'Nguyễn Văn A'
'Action' : 'Tạo'
'Đối tượng' : 'Bài viết'
```

  - Nguyễn Văn A có thể tạo bài viết..
  - Ví dụ : Phân quyền trong `MySql`.

▼ **RBAC**: là hình thức phân quyền dựa vào vai, mỗi Subject sẽ thuộc một hoặc nhiều Role. Mỗi Role lại có một hoặc nhiều Permission thực thi Action tới Object.

- Một `Subject` thuộc một hoặc nhiều `Role`
- Một `Role` có một hoặc nhiều `Permission`.
- **Ví dụ**:
  - Người dùng Nguyễn Văn A có quyền Admin, User.

    - Người dùng Lê Văn B có quyền User.
    - User có quyền `Đọc` bài viết.
    - Admin có quyền `Đọc`, `Thêm`. `Sửa`, `Xóa` bài viết.

    ⇒

    - Người dùng A có quyền `Đọc`, `Thêm`. `Sửa`, `Xóa` bài viết.
    - Người dùng Lê Văn B có quyền `Đọc` bài viết.

  https://www.phamduytung.com/blog/2021-07-02-mo-hinh-phan-quyen/

  https://kipalog.com/posts/Bai-toan-phan-quyen-van-de-muon-thuo-va-rat-kho-hieu

▼ **Một số cách secure trong nodejs**

- Validate user input to limit SQL injections and XSS attacks
- Implement strong authentication
- Avoid errors that reveal too much
- Run automatic vulnerability scanning
- Avoid data leaks
- Set up logging and monitoring
- Use security linters (eslint)
- Avoid secrets in config files
- Implement HTTP response headers
- Don’t run Node.js as root
- Protect and observe your Node.js apps in production

https://blog.sqreen.com/nodejs-security-best-practices/

---

- Chống DOS, DDOS hay brute-force mật khẩu
- Lọc dữ liệu người dùng gửi lên server
- Sử dụng helmet
- Sử dụng `express-validator` để data gửi lên server
- Sử dụng thư viện ORD/ODM để chống SQL/NoSQL Injection
- Sử dụng https
- Sử dụng biến môi trường
- Dùng **bcrypt** hoặc **pbkdf2** để băm mật khẩu
- Giới hạn kích thước payload request gửi lên server

https://hocweb.vn/toi-uu-bao-mat-app-nodejs-tot-hon

▼ **CronJob**

- Cron là chương trình để xử lý các tác vụ lặp đi lặp lại ở lần sau.
- Cron Job đưa ra một lệnh để lên lịch “làm việc” cho một hành động cụ thể, tại một thời điểm cụ thể mà cần lặp đi lặp lại.

- Giả sử ứng dụng của bạn có chức năng lưu tạm file, vậy mỗi lần người dùng lưu tạm miết vậy và không dùng, đến một lúc nào đó nó sẽ đầy và tốn dùng lượng.
- Lúc này bạn cần một công việc tự động là 3 ngày nó sẽ dọn các file tạm đó đi.
- Do đó, đối với các công việc định kì, lặp đi lặp lại thì cron là giải pháp hoàn hảo.

▼ **IoC**

- **Inversion of Control** (*IoC - Đảo ngược điều khiển*) là một nguyên lý thiết kế trong công nghệ phần mềm trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.

▼ **Dependency Injection**

- https://2kvn.com/di-la-gi
- **Dependency Injection** (DI) là 1 design pattern được sử dụng để thực hiện Inversion of Control(IoC). DI giúp loại bỏ việc phụ thuộc trực tiếp của class với đối tượng.
- Có thể hiểu Dependency Injection một cách đơn giản như sau:
  - Các module không giao tiếp trực tiếp với nhau, mà thông qua interface. Module cấp thấp sẽ implement interface, module cấp cao sẽ gọi module cấp thấp thông qua interface.
  - Việc khởi tạo các module cấp thấp sẽ do DI Container thực hiện.
  - Việc Module nào gắn với interface nào sẽ được config trong code hoặc trong file XML
  - DI được dùng để làm giảm sự phụ thuộc giữa các module, dễ dàng hơn trong việc thay đổi module, bảo trì code và testing.

https://toidicodedao.com/2015/11/03/dependency-injection-va-inversion-of-control-phan-1-dinh-nghia/

https://kipalog.com/posts/Dependency-injection---Nhung-thu-co-the-ban-bo-qua

▼ **Push notification & real-time**

https://kipalog.com/posts/Realtime--Pusher-va-ke-thay-the-Slanger

- Một số cách push notification từ server đến client
  - setInterval gọi 5s hay 10s gì đó thì call api lên server
  - Sử dụng Nodejs với websocket
  - Sử dụng cơ chế pub sub của redis
  - Sử dụng một Pusher (trả phí), Slanger (open source)
  - Sử dụng firebase
  - ...