1. Chuẩn thiết kế API: RESTful API, gRPC

1. RESTful API là gì? Các nguyên tắc chính?

RESTful API là một kiểu kiến trúc API dựa trên REST – Representational State Transfer. Nó cho phép client và server giao tiếp với nhau qua HTTP, sử dụng các phương thức như GET, POST, PUT, DELETE. REST tuân theo 6 nguyên tắc chính:

- client-server (tách client và server)
- stateless (không lưu trạng thái)
- cacheable (hỗ trợ lưu cache)
- uniform interface (giao diện đồng nhất)
- layered system (hệ thống nhiều lớp)
- code on demand (tùy chọn tải code từ server).

"RESTful API là API dựa trên kiến trúc REST, cho phép client và server giao tiếp qua HTTP. Nó tuân theo các nguyên tắc như client-server, stateless, cacheable, uniform interface, layered system và code on demand."

2. Idempotent và Safe methods là gì?

Idempotent methods là những phương thức HTTP mà dù gọi nhiều lần cũng không làm thay đổi kết quả so với lần đầu, ví dụ như GET, PUT hay DELETE.

Safe methods là những phương thức không thay đổi tài nguyên, tức là chỉ đọc dữ liệu mà không sửa gì, ví dụ GET và HEAD.

"Idempotent là phương thức mà gọi nhiều lần không thay đổi kết quả (GET, PUT, DELETE). Safe là phương thức không thay đổi tài nguyên (GET, HEAD)."

3. Khi nào nên dùng PUT vs PATCH?

PUT được dùng để cập nhật toàn bộ resource, tức là thay thế hoàn toàn dữ liệu cũ bằng dữ liệu mới.

PATCH được dùng khi chỉ cần cập nhật một phần của resource, thay vì thay thế toàn bộ.

"Dùng PUT để thay thế toàn bộ resource, dùng PATCH để cập nhật một phần."

4. Cấu trúc URL tốt cho RESTful API?

Cấu trúc URL tốt cho RESTful API nên dùng danh từ số nhiều, ví dụ /users/{id}/posts, tránh dùng động từ trong URL. Dùng query parameters cho các chức năng như lọc, sắp xếp, phân trang. Ngoài ra, có thể thêm versioning như /api/v1/... để dễ quản lý phiên bản.

"URL RESTful nên dùng danh từ số nhiều, tránh động từ, dùng query params cho filter/sort/paginate và có thể thêm version như /api/v1/...."

5. HTTP status code hay gặp và khi nào dùng?

Các HTTP status code thường gặp gồm:

2xx: thành công, ví dụ 200 OK, 201 Created khi tạo tài nguyên, 204 No Content khi không có dữ liệu trả về.

4xx: lỗi phía client, ví dụ 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict.

5xx: lỗi phía server, ví dụ 500 Internal Server Error, 503 Service Unavailable.

"2xx là thành công (200, 201, 204), 4xx là lỗi client (400, 401, 403, 404, 409), 5xx là lỗi server (500, 503)."

6. Làm thế nào để thiết kế API dễ mở rộng và backward-compatible?

Để thiết kế API dễ mở rộng và backward-compatible, bạn có thể version hóa API, ví dụ /v1, /v2, không xóa các field cũ mà giữ chúng để tránh phá client hiện tại, dùng feature flags hoặc HATEOAS để thêm tính năng mới mà không ảnh hưởng cũ, và document rõ ràng bằng Swagger hoặc OpenAPI để client dễ hiểu.

"Dùng versioning, giữ field cũ, feature flags/HATEOAS và document rõ ràng để API dễ mở rộng và backward-compatible."

7. RESTful API caching — thiết kế và lợi ích?

Caching trong RESTful API thường dựa vào HTTP headers như ETag, Last-Modified, Cache-Control. Thiết kế cache hợp lý giúp giảm tải server, tăng tốc độ phản hồi và tiết kiệm băng thông.

"Dùng HTTP headers (ETag, Last-Modified, Cache-Control) để cache, giúp giảm tải server và tăng tốc phản hồi."

8. Authentication vs Authorization trong API?

Authentication (AuthN) xác định ai đang đăng nhập, còn Authorization (AuthZ) xác định người đó có quyền làm gì trên hệ thống.

"AuthN là xác thực ai đang đăng nhập, AuthZ là xác định quyền của họ."

9. gRPC là gì? Khác gì với REST?

gRPC là framework RPC hiện đại dùng HTTP/2 và Protocol Buffers để truyền dữ liệu nhị phân, hỗ trợ streaming hai chiều và tốc độ nhanh hơn.

Khác với REST, REST dùng HTTP/1.1 và JSON, dễ debug và phổ biến hơn, nhưng thường chậm hơn và không hỗ trợ streaming phức tạp.

"gRPC dùng HTTP/2 + protobuf, nhị phân, hỗ trợ streaming, nhanh hơn. REST dùng JSON + HTTP/1.1, dễ debug, phổ biến hơn."

10. Các loại RPC trong gRPC?

gRPC hỗ trợ 4 loại RPC:

Unary: client gửi 1 request, server trả 1 response.

Server streaming: client gửi 1 request, server trả nhiều response.

Client streaming: client gửi nhiều request, server trả 1 response.

Bidirectional streaming: cả client và server gửi nhiều message song song.

"4 loại RPC trong gRPC: Unary, Server streaming, Client streaming, Bidirectional streaming."

11. Khi nào nên chọn gRPC thay vì REST?

Bạn nên chọn gRPC khi làm microservices nội bộ, cần hiệu năng cao, low latency hoặc streaming dữ liệu, và không cần dữ liệu dễ đọc bằng tay.

REST phù hợp hơn cho public API, khi client đa dạng và cần dễ test/debug.

"Chọn gRPC cho microservices nội bộ, streaming, low latency; chọn REST cho public API, client đa dạng, dễ test/debug."

12. gRPC có hỗ trợ load balancing, retry, timeout không?

gRPC có hỗ trợ load balancing, retry và timeout thông qua gRPC Channel và interceptors. Bạn có thể cấu hình retry policy, timeout, và load-balancing ở phía client để tối ưu hiệu năng và độ ổn định.

"gRPC hỗ trợ load balancing, retry, timeout qua Channel và interceptors, có thể config client-side."

13. Làm sao định nghĩa và sinh code gRPC?

Để định nghĩa gRPC, bạn viết file .proto xác định service, RPC method và message. Sau đó dùng protoc compiler để sinh code stub cho client và server, giúp triển khai dễ dàng.

"Viết file .proto rồi dùng protoc để sinh code stub cho client và server."

14. gRPC reflection, health check và interceptors dùng làm gì?

Trong gRPC:

Reflection cho phép introspection, giúp debug và các tool tự động khám phá service.

Health check dùng để xác định service còn hoạt động hay không.

Interceptor giống middleware, dùng cho logging, tracing, authentication và các xử lý chung khác.

"Reflection cho debug/tooling, health check xác định service còn chạy, interceptor là middleware cho logging, tracing, auth..."

15. Làm thế nào bảo vệ API khỏi tấn công?

Để bảo vệ API, bạn nên dùng HTTPS để mã hóa dữ liệu, sử dụng JWT hoặc OAuth2 cho authentication/authorization, áp dụng rate limiting và IP filtering để chống spam, validate input để tránh injection, và nếu có giao diện web thì thêm CSRF/XSS protection.

"Dùng HTTPS, JWT/OAuth2, rate limiting, validate input, và thêm CSRF/XSS protection nếu có web."

16. Pagination, Filtering, Sorting nên implement như thế nào?

Pagination, filtering và sorting thường implement qua query parameters. Ví dụ /users?page=2&limit=20&sort=created_at&order=desc. Thiết kế như vậy giúp tránh trả về toàn bộ dataset, giảm tải server và tăng hiệu năng.

"Dùng query params cho pagination, filtering, sorting, ví dụ /users?page=2&limit=20&sort=created_at&order=desc, tránh trả về toàn bộ dataset."

17. Rate limiting là gì? Làm sao triển khai?

Rate limiting là giới hạn số request trong một khoảng thời gian để tránh abuse và bảo vệ server. Có thể triển khai bằng Redis, NGINX, hoặc middleware như Kong, Envoy.

"Rate limiting giới hạn số request/time để tránh abuse, triển khai bằng Redis, NGINX hoặc middleware như Kong/Envoy."

18. Làm sao đo hiệu năng API?

Để đo hiệu năng API, dùng các công cụ APM như NewRelic, Datadog hoặc Prometheus. Theo dõi các chỉ số quan trọng như latency (P95/P99), throughput và error rate để đánh giá và tối ưu API.

"Dùng APM (NewRelic, Datadog, Prometheus) theo dõi latency, throughput và error rate để đo hiệu năng API."

19. Bạn đã từng xử lý backward compatibility khi thay đổi API chưa?

Khi thay đổi API, tôi đảm bảo backward compatibility bằng cách giữ nguyên version cũ, dùng feature flag để triển khai dần các tính năng mới, hoặc sử dụng content negotiation với header Accept: application/vnd.company.v2+json để client vẫn có thể dùng version cũ.

"Giữ version cũ, dùng feature flag hoặc Accept: application/vnd.company.v2+json để đảm bảo backward compatibility."

20. gRPC Gateway là gì?

gRPC Gateway là một công cụ cho phép expose gRPC service ra các HTTP RESTful endpoint. Điều này giúp các client không hỗ trợ gRPC vẫn có thể gọi API một cách bình thường.

"gRPC Gateway expose gRPC service ra RESTful HTTP, cho client không hỗ trợ gRPC vẫn gọi được API."

21. Nếu hệ thống RESTful API hiện tại quá chậm, bạn sẽ làm gì?

Nếu RESTful API hiện tại quá chậm, tôi sẽ bắt đầu phân tích bottleneck ở database, network hoặc code. Sau đó áp dụng các giải pháp như caching, batching, sử dụng async queue (Kafka/RabbitMQ), dùng CDN hoặc tách hệ thống thành microservices để cải thiện hiệu năng.

"Phân tích bottleneck, thêm caching/batching/async queue, dùng CDN hoặc tách microservices để cải thiện hiệu năng API."

22. Làm sao migrate từ REST sang gRPC dần dần?

Để migrate từ REST sang gRPC dần dần, tôi sẽ dùng gRPC Gateway hoặc triển khai dual-stack API layer. Giữ REST cho client public để không phá vỡ client hiện tại, trong khi dùng gRPC cho nội bộ để tối ưu hiệu năng và streaming.

"Dùng gRPC Gateway hoặc dual-stack API, giữ REST cho client public, gRPC cho nội bộ."

23. Nếu API có nhiều version, bạn quản lý chúng thế nào?

Khi API có nhiều version, tôi quản lý bằng cách dùng folder hoặc module riêng cho từng version, duy trì backward compatibility, và document rõ ràng kèm migration guide để client dễ nâng cấp.

"Dùng folder/module riêng, giữ backward compatibility, document migration guide rõ ràng."

24. Làm sao test REST và gRPC?

Để test REST API, tôi thường dùng Postman, curl hoặc Newman cho tự động hóa. Với gRPC, có thể dùng BloomRPC, grpcurl, hoặc viết các integration test tự động để đảm bảo service hoạt động đúng.

"Test REST bằng Postman, curl, Newman; test gRPC bằng BloomRPC, grpcurl hoặc integration test tự động."

25. Bạn sẽ làm gì nếu gRPC client gặp lỗi "deadline exceeded"?

Nếu gRPC client gặp lỗi deadline exceeded, tôi sẽ kiểm tra cấu hình timeout, độ trễ mạng, và tốc độ xử lý của server. Đồng thời dùng logging và tracing (ví dụ OpenTelemetry) để tìm ra bottleneck và tối ưu hiệu năng.

"Kiểm tra timeout, network latency, server chậm, dùng logging/tracing để tìm bottleneck."

2. Quy trình phát triển phần mềm: Scrum.

1. Scrum là gì? Mục tiêu chính của Scrum?

Scrum là một framework thuộc Agile, giúp phát triển phần mềm theo hướng lặp (iterative) và tăng dần (incremental). Mục tiêu chính là nhanh chóng tạo giá trị cho khách hàng, nhận phản hồi liên tục và thích ứng với thay đổi.

"Scrum là framework Agile, phát triển phần mềm lặp và tăng dần, mục tiêu tạo giá trị nhanh, nhận phản hồi liên tục và thích ứng với thay đổi."

2. Scrum có những vai trò nào?

Trong Scrum có ba vai trò chính:
Product Owner (PO) quản lý backlog và xác định thứ tự ưu tiên.
Scrum Master đảm bảo team tuân theo Scrum và loại bỏ trở ngại.
Development Team chịu trách nhiệm xây dựng sản phẩm.

"Ba vai trò chính: PO quản lý backlog, Scrum Master đảm bảo Scrum, Development Team xây dựng sản phẩm."

3. Các artefacts chính trong Scrum là gì?

Trong Scrum có ba artefacts chính:
Product Backlog: danh sách tất cả các tính năng, yêu cầu cần phát triển.
Sprint Backlog: các công việc team cam kết hoàn thành trong sprint.
Increment: sản phẩm hoàn chỉnh, có thể release, tích lũy từ các sprint trước.

"Ba artefacts chính: Product Backlog (danh sách tính năng), Sprint Backlog (công việc sprint), Increment (sản phẩm hoàn chỉnh, có thể release)."

4. Một Sprint là gì? Thời gian thường kéo dài bao lâu?

Sprint là một chu kỳ phát triển lặp lại trong Scrum, thường kéo dài từ 1 đến 4 tuần. Kết thúc mỗi sprint, team phải có một Increment hoàn chỉnh, đạt tiêu chí 'Done'.

"Sprint là chu kỳ phát triển lặp, 1–4 tuần, kết thúc có Increment hoàn chỉnh đạt 'Done'."

5. Scrum có những sự kiện (ceremonies) nào?

Trong Scrum có các sự kiện chính:
Sprint Planning: lên kế hoạch công việc cho sprint.
Daily Scrum (Daily Standup): họp ngắn hằng ngày để cập nhật tiến độ.
Sprint Review: xem lại sản phẩm và nhận phản hồi.
Sprint Retrospective: đánh giá cách làm việc và cải thiện quy trình.
Sprint: vòng đời lặp của Scrum.
"Các sự kiện Scrum: Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective, và Sprint là vòng đời lặp."

6. Mục tiêu của Sprint Planning là gì?

Mục tiêu của Sprint Planning là định nghĩa Sprint Goal, chọn các Product Backlog Items (PBI) để thực hiện trong sprint, và lập kế hoạch chi tiết công việc cho team.
"Sprint Planning định nghĩa Sprint Goal, chọn PBI và lập kế hoạch công việc cho sprint."

7. Daily Scrum diễn ra như thế nào?

Daily Scrum diễn ra 15 phút mỗi ngày, trong đó team chia sẻ: hôm qua đã làm gì, hôm nay dự định làm gì, và có trở ngại nào không. Mục đích là đồng bộ tiến độ và nhận biết sớm vấn đề.
"Daily Scrum 15 phút, team chia sẻ hôm qua làm gì, hôm nay làm gì, có trở ngại không."

8. Sprint Review khác gì Sprint Retrospective?

Sprint Review là buổi trình bày sản phẩm cho Product Owner và stakeholders để lấy phản hồi về Increment.
Sprint Retrospective là buổi team tự xem xét cách làm việc, nhận biết vấn đề và tìm cách cải thiện quy trình cho sprint tiếp theo.
"Sprint Review trình bày sản phẩm, lấy feedback; Retrospective team tự xem cách làm việc để cải thiện quy trình."

9. “Definition of Done (DoD)” là gì? Vì sao quan trọng?

Definition of Done (DoD) là tiêu chí xác định khi nào một task hoặc Increment thực sự hoàn thành. Nó quan trọng vì giúp đảm bảo chất lượng đồng nhất, tránh tình trạng 'xong nhưng chưa test' hoặc chưa đáp ứng yêu cầu.
"DoD là tiêu chí xác định task hoàn thành, giúp đảm bảo chất lượng đồng nhất và tránh việc xong nhưng chưa test."

10. Làm sao ước lượng công việc trong Scrum?

Trong Scrum, công việc thường được ước lượng bằng Story Points dựa trên độ phức tạp, effort và rủi ro. Các công cụ phổ biến gồm Planning Poker hoặc T-shirt sizing, giúp team đưa ra ước lượng tương đối và đồng thuận.
"Ước lượng công việc bằng Story Points, dùng Planning Poker hoặc T-shirt sizing, dựa trên độ phức tạp, effort và rủi ro."

11. Nếu PO thay đổi yêu cầu giữa Sprint, bạn xử lý thế nào?

Theo Scrum, Sprint Goal không nên thay đổi giữa Sprint. Nếu Product Owner thay đổi yêu cầu quan trọng, PO và team sẽ họp lại để đánh giá, và có thể hủy Sprint hiện tại và bắt đầu một Sprint mới.

"Sprint Goal không thay đổi; nếu yêu cầu quan trọng thay đổi, họp lại và có thể hủy Sprint để bắt đầu Sprint mới."

12. Làm sao xử lý khi team không hoàn thành toàn bộ Sprint Backlog?

Nếu team không hoàn thành toàn bộ Sprint Backlog, chúng tôi sẽ phân tích nguyên nhân trong Sprint Retrospective. Các item chưa xong sẽ được đưa trở lại Product Backlog và ưu tiên lại cho các sprint tiếp theo. Đồng thời, team sẽ cải thiện dần velocity và planning accuracy.

"Phân tích nguyên nhân trong Retrospective, đưa item chưa xong về Product Backlog, cải thiện velocity và planning accuracy."

13. Velocity là gì? Dùng để làm gì?

Velocity là tổng số Story Points mà team hoàn thành trong một Sprint. Nó giúp ước lượng năng lực thực tế của team và lập kế hoạch cho các Sprint tiếp theo một cách chính xác hơn.

"Velocity là tổng Story Points hoàn thành trong Sprint, dùng để ước lượng năng lực team cho Sprint sau."

14. Làm sao bạn xử lý khi Scrum Master không chủ động hỗ trợ team?

Nếu Scrum Master không chủ động hỗ trợ team, team có thể nêu vấn đề trong Sprint Retrospective. Ngoài ra, có thể báo Product Owner hoặc quản lý để hỗ trợ đào tạo Scrum Master hoặc điều phối lại vai trò để đảm bảo team được hỗ trợ đầy đủ.

"Team nêu vấn đề trong Retrospective hoặc báo PO/management để đào tạo hoặc điều phối lại Scrum Master."

15. Làm sao đảm bảo chất lượng khi deadline gấp?

Khi deadline gấp, tôi đảm bảo Definition of Done không thay đổi. Thay vì bỏ qua test hoặc code review, tôi cắt giảm phạm vi (scope) và ưu tiên các tính năng cốt lõi có giá trị cao nhất cho khách hàng.

"Giữ DoD, cắt scope thay vì bỏ test, ưu tiên tính năng core có giá trị cao."

16. Nếu bạn và PO không đồng ý về độ ưu tiên backlog thì làm sao?

Nếu có bất đồng với PO về độ ưu tiên backlog, tôi sẽ thảo luận dựa trên giá trị kinh doanh (business value) và rủi ro. Developer có thể giải thích khía cạnh kỹ thuật, nhưng PO quyết định cuối cùng để đảm bảo Product Goal được tối ưu.

"Thảo luận dựa trên business value và risk; developer giải thích kỹ thuật, PO quyết định cuối cùng."

17. Khi có thành viên trong team không hợp tác, bạn làm gì?

Khi có thành viên không hợp tác, tôi sẽ thảo luận riêng để lắng nghe nguyên nhân và tìm giải pháp. Nếu tình hình không cải thiện, báo Scrum Master để can thiệp và hỗ trợ team duy trì hiệu quả làm việc.

"Thảo luận riêng, lắng nghe nguyên nhân; nếu không cải thiện, báo Scrum Master can thiệp."

18. Làm sao đo hiệu quả của một Sprint?

Hiệu quả của một Sprint được đo dựa trên mức độ hoàn thành Sprint Goal, chất lượng deliverable, số lượng bug, và velocity ổn định của team.

"Đo hiệu quả Sprint dựa trên Sprint Goal, chất lượng sản phẩm, số bug và velocity ổn định."

19. Bạn đã bao giờ tham gia Retrospective thực tế chưa? Team bạn thường cải thiện điều gì?

Tôi đã tham gia Retrospective thực tế. Team chúng tôi thường cải thiện các khía cạnh như chất lượng code review, tối ưu thời gian CI/CD, giảm thời gian họp không cần thiết, và cải thiện độ chính xác trong ước lượng công việc.

"Đã tham gia Retrospective; team cải thiện code review, CI/CD, giảm meeting time, và tối ưu estimate."

20. Scrum có bắt buộc dùng Task board không?

Scrum không bắt buộc phải dùng Task board, nhưng nó rất hữu ích. Các công cụ như Jira, Trello, ClickUp giúp minh bạch tiến độ và dễ visual hóa trạng thái của các task trong sprint.

"Scrum không bắt buộc Task board, nhưng dùng Jira/Trello/ClickUp giúp minh bạch tiến độ và visual hóa task."

21. Scrum khác gì Kanban?

Scrum có Sprint cố định, các vai trò và nghi thức rõ ràng. Trong khi đó, Kanban không có Sprint, luồng công việc liên tục và tập trung vào giới hạn 'work in progress' để tối ưu hóa hiệu suất.

"Scrum có Sprint cố định, vai trò và nghi thức rõ; Kanban luồng liên tục, tập trung vào work in progress."

22. Nếu khách hàng yêu cầu gấp feature trong 2 ngày, bạn làm gì?

Khi khách hàng yêu cầu gấp feature trong 2 ngày, tôi sẽ xem nó có phù hợp với Sprint Goal không. Nếu quan trọng, PO và team sẽ thương lượng để đổi scope hoặc triển khai hotfix riêng ngoài Sprint hiện tại.

"Kiểm tra phù hợp Sprint Goal; nếu quan trọng, thương lượng đổi scope hoặc làm hotfix ngoài Sprint."

23. Làm sao bạn xử lý technical debt trong Scrum?

Để xử lý technical debt trong Scrum, tôi liệt kê các khoản nợ kỹ thuật trong Product Backlog và gắn nhãn 'tech debt'. Mỗi sprint, team dành khoảng 10–20% effort để xử lý những khoản này, vừa đảm bảo chất lượng vừa không ảnh hưởng quá nhiều đến tiến độ.
"Liệt kê tech debt trong Product Backlog, gắn nhãn, dành 10–20% effort mỗi sprint để xử lý."

24. Scrum Master không phải là Project Manager — giải thích?

Scrum Master không phải Project Manager vì họ không ra lệnh hay quản lý deadline. Scrum Master giúp team tự tổ chức, loại bỏ trở ngại, và huấn luyện team tuân thủ Scrum framework.
"Scrum Master không quản lý deadline; họ hỗ trợ team tự tổ chức, loại bỏ trở ngại và huấn luyện Scrum."

25. Làm sao áp dụng Scrum khi team có nhiều timezone khác nhau?

Khi team có nhiều timezone, chúng tôi giữ Daily Scrum vào giờ cố định hợp lý cho đa số thành viên, đồng thời dùng các công cụ async như Slack standup bot hoặc cập nhật Jira. Ưu tiên communication rõ ràng và minh bạch để mọi người đồng bộ thông tin.
"Giữ Daily Scrum giờ cố định, dùng Slack/Jira async, ưu tiên communication rõ ràng và minh bạch."

3. DevOps, Git, Jenkins, GitLab CI.

1. DevOps là gì? Mục tiêu chính của DevOps?

DevOps là văn hoá và quy trình kết hợp giữa Development và Operations. Mục tiêu chính là rút ngắn vòng đời phát triển, tăng tần suất release, đồng thời đảm bảo chất lượng và độ ổn định của hệ thống.
"DevOps là văn hoá kết hợp Dev và Ops, mục tiêu rút ngắn vòng đời phát triển, tăng release, và đảm bảo chất lượng/ổn định."

2. Sự khác biệt giữa Agile và DevOps?

Agile tập trung vào phát triển phần mềm nhanh và nhận phản hồi liên tục từ khách hàng. DevOps tập trung vào tự động hóa và vận hành liên tục, giúp phần mềm được triển khai, giám sát và vận hành hiệu quả sau khi code xong.
"Agile tập trung phát triển nhanh, nhận phản hồi; DevOps tập trung tự động hóa và vận hành liên tục để triển khai và giám sát phần mềm."

3. CI/CD là gì?

CI/CD gồm:
Continuous Integration (CI): merge code thường xuyên và chạy test tự động để đảm bảo code luôn ổn định.
Continuous Delivery/Deployment (CD): tự động build, test, và deploy phần mềm lên môi trường staging hoặc production.
"CI là merge code thường xuyên và test tự động; CD là build, test, deploy tự động lên staging/production."

4. Các lợi ích chính của CI/CD pipeline?

CI/CD pipeline mang lại nhiều lợi ích: tự động hóa giúp giảm lỗi con người, phản hồi sớm khi build hoặc test thất bại, dễ rollback khi có sự cố, và tăng tốc độ release phần mềm.
"CI/CD giảm lỗi, phản hồi sớm khi build/test fail, dễ rollback, và tăng tốc độ release."

5. Công cụ phổ biến trong hệ sinh thái DevOps?

Trong hệ sinh thái DevOps, một số công cụ phổ biến gồm:
CI/CD: Jenkins, GitLab CI, GitHub Actions
Container & Orchestration: Docker, Kubernetes
Infrastructure as Code: Terraform, Ansible
Monitoring & Logging: Prometheus, Grafana
"DevOps phổ biến: CI/CD (Jenkins, GitLab CI, GitHub Actions), Container (Docker, Kubernetes), Infra (Terraform, Ansible), Monitoring (Prometheus, Grafana)."

6. Git hoạt động như thế nào?

Git là một Distributed Version Control System (DVCS), nghĩa là mỗi developer có repository riêng. Git lưu thay đổi dưới dạng snapshot của toàn bộ project, chứ không lưu đơn thuần theo file diff, giúp theo dõi lịch sử và quản lý nhánh hiệu quả.
"Git là DVCS, mỗi dev có repo riêng, lưu thay đổi theo snapshot, không chỉ file diff."

7. Git branching strategy nào bạn thường dùng?

Tôi thường sử dụng các branching strategy như:
Git Flow với các nhánh main, develop, feature/, release/, hotfix/*
Trunk-based development, merge trực tiếp vào main sau khi CI pass
GitLab Flow, kết hợp feature branch với environment branch để deploy linh hoạt.
"Thường dùng Git Flow, Trunk-based hoặc GitLab Flow, tuỳ team và CI/CD setup."

8. Rebase vs Merge — khác nhau thế nào?

Merge giữ nguyên lịch sử commit, dễ theo dõi nhưng có thể tạo nhiều merge commit. Rebase gộp lại lịch sử, giúp lịch sử thẳng hàng và gọn hơn. Thường dùng rebase cho branch cá nhân và merge khi tích hợp vào main.
"Merge giữ lịch sử, dễ xem; Rebase gộp lịch sử, thẳng hàng. Rebase cho branch cá nhân, merge vào main."

9. Khi nào nên dùng git cherry-pick?

Git cherry-pick dùng khi cần lấy một commit cụ thể từ branch khác. Ví dụ, khi có một hotfix quan trọng cần đưa nhanh lên production mà không muốn merge toàn bộ branch.
"Dùng cherry-pick để lấy commit cụ thể từ branch khác, ví dụ hotfix lên production."

10. Làm sao xử lý conflict khi rebase hoặc merge?

Khi gặp conflict trong rebase hoặc merge, tôi dùng git status để xác định các file xung đột. Sau đó chỉnh sửa file, kiểm tra lại, rồi dùng git add và git rebase --continue (hoặc git commit nếu merge) để hoàn tất.
"Dùng git status xác định conflict, chỉnh sửa, test, rồi git add + git rebase --continue (hoặc git commit nếu merge)."

11. Git tag dùng để làm gì?

Git tag dùng để đánh dấu các mốc version release, ví dụ v1.0.0. Tag có thể là lightweight hoặc annotated, kèm metadata như tác giả, ngày tháng và message.

"Git tag đánh dấu mốc release (v1.0.0), có thể lightweight hoặc annotated."

12. Jenkins là gì? Dùng trong giai đoạn nào?

Jenkins là một automation server dùng trong CI/CD pipeline. Nó tự động build, test và deploy phần mềm khi có thay đổi code, giúp tăng tốc độ phát triển và giảm lỗi.

"Jenkins là automation server cho CI/CD, tự động build, test và deploy khi code thay đổi."

13. Jenkinsfile là gì?

Jenkinsfile là file định nghĩa pipeline sử dụng DSL, có thể theo Declarative hoặc Scripted. Nó thường nằm ở root của repository để Jenkins tự động đọc và chạy pipeline.

"Jenkinsfile định nghĩa pipeline bằng DSL (Declarative hoặc Scripted), thường ở root repo để Jenkins chạy tự động."

14. Các stage phổ biến trong Jenkins pipeline?

Các stage phổ biến trong Jenkins pipeline gồm: Checkout (lấy code), Build, Test, Package, Deploy, và Notify (thông báo kết quả).

"Stage phổ biến: Checkout → Build → Test → Package → Deploy → Notify."

15. Jenkins có thể tích hợp với những công cụ nào?

Jenkins có thể tích hợp với nhiều công cụ khác nhau, ví dụ:
SCM: Git, GitHub, GitLab
Build tools: Maven, Gradle, npm
Container/Orchestration: Docker, Kubernetes
Notification: Slack, Email

"Jenkins tích hợp SCM (Git/GitHub/GitLab), Build (Maven/Gradle/npm), Container (Docker/Kubernetes), Notification (Slack/Email)."

16. Jenkins agent (node) là gì?

Jenkins agent là máy con thực thi các job. Jenkins Master chỉ điều phối, còn agent chịu trách nhiệm chạy build, test và các tác vụ thực tế.

"Jenkins agent chạy job thực tế; Master chỉ điều phối."

17. Bạn có biết "Blue Ocean" trong Jenkins không?

Blue Ocean là giao diện hiện đại của Jenkins, giúp visualize pipeline trực quan hơn, dễ theo dõi các stage và trạng thái build.

"Blue Ocean là UI hiện đại của Jenkins, visualize pipeline trực quan."

18. Khi build Jenkins fail, bạn làm gì?

Khi build Jenkins fail, tôi sẽ kiểm tra logs (Console Output) để xác định nguyên nhân. Xem commit gần nhất, test case nào fail, hoặc vấn đề môi trường. Nếu lỗi do script, tôi chạy local để debug trước khi sửa.

"Kiểm tra logs, commit/test fail, môi trường; nếu lỗi script, chạy local để debug."

19. Làm sao để Jenkins pipeline chỉ chạy khi có thay đổi trong thư mục cụ thể?

Để Jenkins pipeline chỉ chạy khi có thay đổi trong thư mục cụ thể, có thể dùng when { changeset '**/path/**' } trong Declarative Jenkinsfile hoặc cấu hình pollSCM với filter để kiểm tra thư mục cần theo dõi.

*"Dùng when { changeset '**/path/**' } hoặc pollSCM filter để pipeline chạy chỉ khi thư mục thay đổi."*

20. GitLab CI/CD khác gì Jenkins?

Sự khác nhau giữa GitLab CI/CD và Jenkins gồm:
Cấu hình: Jenkins dùng Jenkinsfile (DSL), GitLab CI dùng .gitlab-ci.yml
Cài đặt: Jenkins cần cài đặt riêng, GitLab CI tích hợp sẵn
Runner/Node: Jenkins agent, GitLab Runner
Giao diện: Jenkins dựa vào plugin, GitLab CI có native UI tích hợp sẵn

"Jenkins dùng Jenkinsfile, cần cài đặt, agent riêng, UI plugin-based; GitLab CI dùng .gitlab-ci.yml, tích hợp sẵn, GitLab Runner, UI native."

21. .gitlab-ci.yml là gì?

.gitlab-ci.yml là file định nghĩa pipeline trong GitLab CI/CD, bao gồm các stages, jobs, dependencies và rules. GitLab sẽ tự động đọc file này mỗi khi có code được push để chạy pipeline.

".gitlab-ci.yml định nghĩa pipeline (stages, jobs, rules), GitLab tự đọc khi push code."

22. Các stage phổ biến trong GitLab pipeline?

Các stage phổ biến trong GitLab pipeline gồm: Build, Test và Deploy, tương ứng với các bước chính từ compile, kiểm thử đến triển khai phần mềm.

"Stage phổ biến: Build → Test → Deploy."

23. Artifacts trong GitLab CI là gì?

Artifacts trong GitLab CI là các output của job, như build, binary, logs hoặc reports, có thể được lưu trữ và chia sẻ giữa các stage trong pipeline.

"Artifacts là output của job (build, logs, reports) có thể chia sẻ giữa các stage."

24. GitLab Runner là gì?

GitLab Runner là máy agent thực thi các job trong GitLab CI/CD. Có thể là Shared Runner dùng chung toàn hệ thống hoặc Specific Runner gắn riêng cho một project.

"GitLab Runner là agent chạy job, có thể Shared (dùng chung) hoặc Specific (riêng project)."

25. Làm sao triển khai CI/CD bảo mật trong GitLab?

Để triển khai CI/CD bảo mật trong GitLab, tôi dùng Protected Variables và Environment scopes, không hard-code token trong code, và chỉ cho phép các stage deploy chạy trên protected branches.

"Dùng Protected Variables, Environment scopes, không hard-code token, deploy chỉ trên protected branches."

26. Cách rollback khi deploy fail?

Khi deploy fail, tôi rollback bằng cách dùng versioned artifacts hoặc Docker image tags, kết hợp với chiến lược Blue-Green deployment hoặc Canary release. Trong GitLab, cũng có thể dùng manual rollback job để quay lại phiên bản trước.

"Rollback bằng versioned artifacts/Docker tags, Blue-Green hoặc Canary, hoặc manual rollback job trong GitLab."

27. Làm sao tối ưu pipeline build chạy nhanh hơn?

Để tối ưu pipeline build, tôi dùng caching (npm cache, Docker layers), chạy test song song (parallel jobs), chia pipeline thành các stages nhỏ và độc lập, và tái sử dụng artifacts thay vì rebuild toàn bộ.

"Dùng caching, parallel jobs, stages nhỏ & độc lập, reuse artifacts để pipeline chạy nhanh hơn."

28. Làm sao triển khai CI/CD cho microservices?

Với microservices, mỗi service nên có pipeline riêng. Có thể dùng template .gitlab-ci.yml chung để tái sử dụng cấu hình, và dùng tag để build & deploy từng service một cách độc lập.

"Mỗi microservice pipeline riêng, dùng template chung, tag để build & deploy độc lập."

29. Bạn có từng xử lý secret trong CI/CD không?

Tôi xử lý secret bằng cách dùng Vault, GitLab CI Variables hoặc AWS Secret Manager. Không bao giờ commit secret vào repository và thường dùng dotenv hoặc .env file được mã hóa để quản lý.

"Dùng Vault, GitLab CI Variables, AWS Secret Manager; không commit secret, dùng dotenv/.env mã hóa."

30. Monitoring sau deploy — bạn làm gì?

Sau khi deploy, tôi theo dõi các metrics như CPU, memory, latency, sử dụng Prometheus + Grafana, kết hợp log aggregation với ELK stack. Nếu phát hiện lỗi tăng, có thể tự động rollback để đảm bảo hệ thống ổn định.

"Theo dõi metrics (CPU/memory/latency) với Prometheus/Grafana, log aggregation ELK, rollback nếu lỗi tăng."

4. Công nghệ container: Docker, Kubernetes.

1. Docker là gì?

Docker là nền tảng container hóa, giúp đóng gói ứng dụng cùng tất cả dependencies vào một container, từ đó chạy đồng nhất trên mọi môi trường, từ local đến production.

"Docker container hóa app cùng dependencies, chạy đồng nhất mọi môi trường."

2. Container khác VM ở điểm nào?

Container và VM khác nhau ở một số điểm chính: Container dùng chung OS kernel, nhẹ và khởi động nhanh, sử dụng tài nguyên hiệu quả hơn. VM có OS riêng, nặng hơn và khởi động chậm hơn, tốn nhiều RAM/CPU hơn.

"Container dùng chung OS, nhẹ, khởi động nhanh, hiệu quả tài nguyên; VM có OS riêng, nặng, tốn RAM/CPU."

3. Docker image là gì?

Docker image là một mẫu (template) để tạo container, bao gồm code, môi trường runtime, thư viện và cấu hình cần thiết để container chạy đúng.

"Docker image là template chứa code, runtime và config để tạo container."

4. Dockerfile là gì?

Dockerfile là file định nghĩa cách build Docker image, gồm các lệnh như FROM, RUN, COPY, CMD,... để xác định base image, cài đặt dependencies, copy code và chỉ định lệnh chạy khi container khởi động.

"Dockerfile định nghĩa cách build image với lệnh FROM, RUN, COPY, CMD,..."

5. Các lệnh Docker cơ bản?

Các lệnh Docker cơ bản gồm: docker build (build image), docker run (chạy container), docker ps (liệt kê container), docker exec (chạy lệnh trong container), docker logs (xem log), docker stop (dừng container), và docker rm (xóa container).

"Docker cơ bản: build, run, ps, exec, logs, stop, rm."

6. Sự khác nhau giữa CMD và ENTRYPOINT?

CMD chỉ định lệnh mặc định khi container chạy, nhưng có thể bị override bởi docker run <command>. ENTRYPOINT xác định command cố định, thường dùng để wrap logic chính, và CMD có thể cung cấp default arguments cho ENTRYPOINT.

"CMD là lệnh mặc định có thể override; ENTRYPOINT cố định command, CMD có thể là default args."

7. Docker volume dùng để làm gì?

Docker volume dùng để lưu trữ dữ liệu bền vững, giữ nguyên giữa các container. Có thể gắn volume khi chạy container bằng -v hoặc --mount.

"Docker volume lưu dữ liệu persist giữa container, gắn bằng -v hoặc --mount."

8. Docker network có mấy loại?

Docker network có một số loại chính: bridge (mặc định cho container trên cùng host), host (container dùng network của host), none (không có network), và overlay (dùng cho multi-host, như Swarm hoặc Kubernetes).

"Docker network: bridge, host, none, overlay (multi-host)."

9. Multi-stage build là gì?

Multi-stage build là kỹ thuật dùng nhiều FROM trong Dockerfile, giúp chỉ copy artifact cần thiết vào image cuối cùng, từ đó giảm kích thước image và tách biệt các bước build/phát triển.

"Multi-stage build dùng nhiều FROM, chỉ copy artifact cần thiết, giảm size image."

10. Làm sao giảm kích thước Docker image?

Để giảm kích thước Docker image, tôi thường dùng base image nhỏ như Alpine, áp dụng multi-stage build, xóa cache sau khi cài dependencies (apt-get clean), và gộp nhiều lệnh RUN vào một để giảm layer.

"Dùng base image nhỏ, multi-stage build, xóa cache, gộp RUN để giảm size image."

11. Docker Compose là gì?

Docker Compose là công cụ để định nghĩa và chạy multi-container application bằng file YAML (docker-compose.yml). Dùng lệnh docker-compose up để khởi chạy toàn bộ stack.

"Docker Compose định nghĩa multi-container app bằng YAML, dùng docker-compose up chạy stack."

12. Docker registry là gì?

Docker registry là nơi lưu trữ Docker image, có thể là public như Docker Hub hoặc private như GitLab Registry, AWS ECR,... để pull/push image giữa các môi trường.

"Docker registry lưu trữ image, ví dụ Docker Hub, GitLab Registry, AWS ECR."

13. Bạn xử lý thế nào khi container “crash loop”?

Khi container bị crash loop, tôi sẽ kiểm tra log (docker logs <container_id>), xem Dockerfile CMD/ENTRYPOINT, kiểm tra port mapping, biến môi trường, file bị thiếu hoặc các lỗi cấu hình khác để xác định nguyên nhân và sửa.

"Check logs, CMD/ENTRYPOINT, port mapping, env, file missing để xử lý crash loop."

14. Kubernetes là gì?

Kubernetes là hệ thống orchestration giúp quản lý, triển khai và scale container tự động. Nó đảm bảo ứng dụng luôn sẵn sàng, cân bằng tải và dễ dàng cập nhật mà không gián đoạn dịch vụ.

"Kubernetes orchestration quản lý, deploy và scale container tự động."

15. Các thành phần chính trong kiến trúc K8s?

Kiến trúc Kubernetes gồm hai phần chính:
Control Plane: API Server, Scheduler, Controller Manager, ETCD.
Node: Kubelet, Kube Proxy, Container Runtime, chịu trách nhiệm chạy container.

"Control Plane: API Server, Scheduler, Controller Manager, ETCD; Node: Kubelet, Kube Proxy, Container Runtime."

16. Pod là gì?

Pod là đơn vị nhỏ nhất trong Kubernetes, có thể chứa một hoặc nhiều container, chia sẻ cùng network namespace và volume, giúp các container trong cùng Pod giao tiếp dễ dàng và đồng bộ dữ liệu.

"Pod là đơn vị nhỏ nhất, chứa 1 hoặc nhiều container chia sẻ network và volume."

17. Deployment là gì?

Deployment là tài nguyên trong Kubernetes quản lý lifecycle của Pod, bao gồm tạo mới, cập nhật và rollback. Nó hỗ trợ rolling update tự động, đảm bảo ứng dụng luôn sẵn sàng khi thay đổi phiên bản.

"Deployment quản lý Pod lifecycle, hỗ trợ rolling update và rollback tự động."

18. ReplicaSet dùng để làm gì?

ReplicaSet đảm bảo số lượng Pod luôn đúng như mong muốn. Ví dụ, nếu muốn có 3 replicas, ReplicaSet sẽ tạo thêm Pod mới nếu có Pod nào bị chết, hoặc xóa bớt nếu vượt quá số lượng.

"ReplicaSet đảm bảo số lượng Pod luôn đúng, tự động thêm/xóa Pod khi cần."

19. Service trong Kubernetes là gì?

Service là một abstraction trong Kubernetes giúp expose Pod ra bên ngoài hoặc giữa các Pod trong cluster. Các loại phổ biến gồm ClusterIP, NodePort và LoadBalancer, đảm bảo giao tiếp ổn định dù Pod bị recreate.

"Service expose Pod ra bên ngoài hoặc giữa Pod, loại ClusterIP, NodePort, LoadBalancer."

20. Ingress là gì?

Ingress là entry point cho HTTP/HTTPS traffic vào Kubernetes cluster, dùng để điều hướng yêu cầu đến các Service khác nhau dựa trên host hoặc path, đồng thời hỗ trợ SSL/TLS termination, load balancing và routing nâng cao.

"Ingress là entry point HTTP/HTTPS, route traffic đến các Service trong cluster."

21. ConfigMap vs Secret khác nhau thế nào?

ConfigMap và Secret đều lưu cấu hình cho Pod, nhưng khác nhau: ConfigMap lưu dữ liệu không nhạy cảm, dùng cho cấu hình chung; Secret lưu dữ liệu nhạy cảm như token, password và được mã hóa base64 để bảo mật.

"ConfigMap lưu config không nhạy cảm; Secret lưu token/password, mã hóa base64."

22. Namespace trong K8s dùng để làm gì?

Namespace trong Kubernetes dùng để phân vùng tài nguyên, giúp quản lý các đối tượng theo team hoặc môi trường khác nhau như dev, staging, production, đồng thời hỗ trợ phân quyền và quota.

"Namespace phân vùng tài nguyên, quản lý theo team hoặc môi trường (dev/staging/prod)."

23. StatefulSet khác gì Deployment?

StatefulSet dùng cho ứng dụng stateful như database, Redis hay Kafka. Nó đảm bảo mỗi Pod có identity cố định và volume riêng biệt, trong khi Deployment thích hợp cho ứng dụng stateless và Pod có thể recreate bất kỳ.

"StatefulSet cho app stateful, Pod có identity và volume cố định; Deployment cho stateless."

24. DaemonSet là gì?

DaemonSet đảm bảo mỗi node trong cluster chạy đúng một Pod. Thường dùng cho các agent như logging, monitoring hoặc network daemons để triển khai trên tất cả node.

"DaemonSet chạy 1 Pod trên mỗi node, dùng cho logging/monitoring agent."

25. CronJob trong K8s là gì?

CronJob trong Kubernetes là một loại resource dùng để chạy các tác vụ theo lịch định kỳ, giống như cron trong Linux. Nó thường được dùng cho các công việc lặp lại như backup dữ liệu, dọn dẹp log, hoặc tạo báo cáo. CronJob sẽ tự động tạo Job vào đúng thời điểm đã được định nghĩa trong biểu thức cron.

"CronJob là resource trong Kubernetes dùng để chạy tác vụ theo lịch định kỳ, ví dụ như backup hoặc dọn dẹp hệ thống, tương tự như cron trong Linux."

26. Kubernetes có những kiểu Service nào?

Trong Kubernetes có 4 loại Service chính:
ClusterIP: Dùng để truy cập nội bộ trong cluster, là kiểu mặc định.
NodePort: Mở port trên mỗi node để truy cập service từ bên ngoài.
LoadBalancer: Dùng để expose service ra Internet thông qua load balancer của cloud provider.
ExternalName: Dùng để ánh xạ service tới một tên miền hoặc dịch vụ bên ngoài cluster.

"Kubernetes có 4 loại Service: ClusterIP (nội bộ), NodePort (mở port node), LoadBalancer (ra Internet), và ExternalName (map ra domain ngoài)."

27. Horizontal Pod Autoscaler (HPA) hoạt động như thế nào?

Horizontal Pod Autoscaler, hay HPA, là cơ chế trong Kubernetes giúp tự động scale số lượng Pod lên hoặc xuống dựa trên các metric như CPU, bộ nhớ, hoặc metric tùy chỉnh. HPA thường theo dõi các chỉ số này định kỳ, rồi so sánh với ngưỡng (threshold) được cấu hình để quyết định scale in hoặc scale out pod cho phù hợp với tải thực tế.

"HPA theo dõi các metric như CPU, memory và tự động tăng hoặc giảm số Pod dựa trên ngưỡng đã đặt, giúp ứng dụng co giãn theo tải."

28. PersistentVolume (PV) và PersistentVolumeClaim (PVC)?

PersistentVolume (PV) trong Kubernetes là tài nguyên mô tả vùng lưu trữ vật lý — ví dụ như ổ đĩa, NFS, hay volume từ cloud.
Còn PersistentVolumeClaim (PVC) là yêu cầu từ người dùng hoặc Pod để sử dụng một phần dung lượng từ PV. Nói cách khác, PV là "ổ đĩa", còn PVC là "phiếu mượn ổ đĩa" mà Pod dùng để gắn vào.

"PV là vùng lưu trữ vật lý, còn PVC là yêu cầu dùng vùng lưu trữ đó — Pod sẽ gắn PVC để có không gian lưu trữ ổn định."

29. Liveness vs Readiness Probe khác nhau thế nào?

Liveness Probe dùng để kiểm tra xem container có đang "sống" hay không — nếu probe thất bại, Kubernetes sẽ tự động restart container.
Còn Readiness Probe kiểm tra xem container đã sẵn sàng nhận traffic chưa. Nếu chưa sẵn sàng, Pod sẽ tạm thời bị loại khỏi load balancer cho đến khi sẵn sàng trở lại.
"Liveness kiểm tra container có còn sống không, còn Readiness kiểm tra container đã sẵn sàng nhận traffic chưa."

30. NetworkPolicy là gì?

NetworkPolicy trong Kubernetes là một tài nguyên dùng để kiểm soát lưu lượng mạng giữa các Pod, hoặc giữa Pod với bên ngoài. Nó định nghĩa những traffic nào được phép đi vào (ingress) hoặc đi ra (egress), giúp tăng cường bảo mật nội bộ trong cluster và tránh các kết nối không mong muốn.
"NetworkPolicy quy định traffic nào được phép đi giữa các Pod, giúp kiểm soát và bảo mật mạng trong cluster."

31. Làm sao bạn triển khai app trong K8s?

Để triển khai ứng dụng trong Kubernetes, tôi thường viết các file YAML mô tả tài nguyên như Deployment để tạo Pod, Service để expose nội bộ hoặc ra ngoài, và Ingress nếu cần route từ domain.
Sau đó tôi dùng lệnh kubectl apply -f <tên-file>.yaml để áp dụng và tạo các resource đó trong cluster.
"Viết YAML cho Deployment, Service, và Ingress rồi dùng kubectl apply -f để triển khai app lên cluster."

32. Làm sao rollback khi bản deploy lỗi?

Khi bản deploy bị lỗi, ta có thể rollback bằng lệnh kubectl rollout undo deployment <tên-deployment>.
Lệnh này giúp quay lại revision trước đó. Ngoài ra có thể dùng kubectl rollout history để xem các version và rollback về revision cụ thể nếu cần.
"Dùng kubectl rollout undo deployment <name> để rollback về bản deploy trước đó khi gặp lỗi."

33. Cách debug Pod đang lỗi?

Khi Pod bị lỗi, tôi thường debug theo 3 bước:
Dùng kubectl describe pod <tên-pod> để xem event và lý do Pod fail (ví dụ image pull, scheduling...).
Dùng kubectl logs <tên-pod> để xem log của container.
Nếu cần kiểm tra bên trong container, tôi dùng kubectl exec -it <tên-pod> -- bash để truy cập và kiểm tra trực tiếp.
"Dùng describe xem event, logs xem log container, và exec vào pod để kiểm tra trực tiếp bên trong."

34. Làm sao tối ưu chi phí cluster Kubernetes?

Để tối ưu chi phí cluster Kubernetes, tôi thường:
Đặt resource request/limit hợp lý để tránh lãng phí CPU và RAM.
Bật autoscaling (HPA cho pod, Cluster Autoscaler cho node) để co giãn tài nguyên theo tải.
Dọn orphan resources như PVC, image cũ, hay pod không còn dùng.
Và dùng spot/preemptible nodes cho workload không cần tính ổn định cao để tiết kiệm chi phí.
"Tối ưu bằng cách set request/limit hợp lý, bật autoscaling, dọn tài nguyên thừa và dùng spot nodes để giảm chi phí."

35. Best practices khi deploy production với Docker + K8s?

Khi deploy production với Docker và Kubernetes, tôi tuân theo một số best practices:
Dùng image tag version cụ thể, tránh dùng latest để dễ quản lý và rollback.
Scan image bảo mật bằng các công cụ như Trivy để phát hiện lỗ hổng.
Không chạy container bằng root để tăng bảo mật.
Giới hạn CPU và RAM bằng request/limit để tránh ảnh hưởng đến cluster.
Thiết lập centralized logging và monitoring để theo dõi ứng dụng và dễ debug khi xảy ra sự cố.
"Best practices: dùng tag version, scan image bảo mật, không chạy root, giới hạn resource, và setup logging/monitoring trung tâm."

5. Database: PostgreSQL, MySQL, MSSQL Server, Redis.

1. RDBMS là gì? So sánh với NoSQL?

RDBMS (Relational Database Management System) là hệ quản trị cơ sở dữ liệu quan hệ, lưu trữ dữ liệu theo bảng với schema cố định và truy vấn bằng SQL.
Ngược lại, NoSQL là loại cơ sở dữ liệu phi quan hệ, dữ liệu có schema linh hoạt, thường dùng cho dữ liệu phi cấu trúc hoặc bán cấu trúc, và dễ scale horizontal hơn khi cần mở rộng.
"RDBMS lưu trữ dữ liệu theo bảng với schema cố định và dùng SQL, còn NoSQL linh hoạt về schema, dữ liệu phi cấu trúc và dễ scale horizontal."

2. ACID là gì? Giải thích từng thành phần.

ACID là tập hợp các đặc tính đảm bảo tính nhất quán của giao dịch trong database:
Atomicity: giao dịch thực hiện tất cả hoặc không thực hiện gì.
Consistency: dữ liệu luôn ở trạng thái hợp lệ trước và sau transaction.
Isolation: các transaction thực hiện đồng thời không ảnh hưởng lẫn nhau.
Durability: dữ liệu đã commit sẽ được lưu bền, ngay cả khi hệ thống crash.
"ACID gồm: Atomicity (tất cả hoặc không), Consistency (dữ liệu hợp lệ), Isolation (transaction độc lập), Durability (dữ liệu commit bền)."

3. Isolation level trong RDBMS?

Trong RDBMS, isolation level xác định mức độ tách biệt giữa các transaction đồng thời, giúp kiểm soát phenomena như dirty read, non-repeatable read, phantom read.
Các mức phổ biến là:
Read Uncommitted: có thể đọc dữ liệu chưa commit (dirty read).
Read Committed: chỉ đọc dữ liệu đã commit.
Repeatable Read: đảm bảo đọc cùng dữ liệu nhiều lần là giống nhau trong cùng transaction.
Serializable: cao nhất, các transaction như thực hiện tuần tự, tránh tất cả các anomaly.
Chọn isolation level là trade-off giữa consistency và concurrency.
"Isolation level kiểm soát cách transaction đồng thời ảnh hưởng lẫn nhau: Read Uncommitted, Read Committed, Repeatable Read, Serializable; trade-off giữa consistency và concurrency."

4. Normalization là gì? Tại sao cần?

Normalization là quá trình chia dữ liệu thành các bảng nhỏ hơn, loại bỏ redundancy và đảm bảo dữ liệu nhất quán.
Quá trình này thường thực hiện qua các bước:
1NF: loại bỏ repeating group, mỗi cột chỉ chứa giá trị nguyên tử.
2NF: loại bỏ partial dependency trên primary key.
3NF (hoặc BCNF): loại bỏ transitive dependency.
Normalization giúp tối ưu lưu trữ, giảm lỗi dữ liệu và cải thiện maintainability.
"Normalization là chia dữ liệu thành bảng nhỏ để giảm redundancy và tăng consistency, thường qua 1NF → 2NF → 3NF hoặc BCNF."

5. Khi nào nên denormalize?

Denormalize là quá trình kết hợp các bảng lại để tăng hiệu năng đọc.
Chúng ta thường denormalize khi ứng dụng read-heavy, hoặc khi join quá nhiều bảng gây chậm, nhằm giảm độ phức tạp của truy vấn và cải thiện tốc độ truy xuất dữ liệu.
Tuy nhiên, denormalize có thể làm tăng redundancy và khó maintain hơn.
"Denormalize khi cần tối ưu đọc (read-heavy) và giảm join phức tạp, đổi lại có thể tăng redundancy."

6. Index là gì? Có mấy loại index phổ biến?

Index là cấu trúc dữ liệu giúp tìm kiếm dữ liệu trong bảng nhanh hơn, tương tự như mục lục trong sách.
Các loại index phổ biến:
B-tree: hỗ trợ range query và lookup nhanh.
Hash: tìm kiếm bằng key, không hỗ trợ range query.
GiST, GIN: thường dùng trong PostgreSQL cho dữ liệu phức tạp như text search, array, JSON.
Full-text: tối ưu tìm kiếm văn bản.
Clustered/Non-clustered: trong MySQL/MSSQL, xác định cách lưu dữ liệu vật lý theo index.
"Index giúp tìm dữ liệu nhanh hơn; các loại phổ biến: B-tree, Hash, GiST/GIN, Full-text, Clustered/Non-clustered."

7. Primary Key vs Unique Key vs Foreign Key?

Primary Key (PK): định danh duy nhất cho mỗi row, không được phép null.
Unique Key: cũng đảm bảo giá trị duy nhất, nhưng có thể có giá trị null.
Foreign Key (FK): ràng buộc quan hệ giữa các bảng, dùng để liên kết dữ liệu từ bảng này sang bảng khác và đảm bảo referential integrity.
"PK duy nhất và không null, Unique key duy nhất nhưng có thể null, FK ràng buộc quan hệ giữa bảng."

8. Composite key là gì? Khi nào dùng?

Composite Key là khóa chính gồm nhiều cột kết hợp lại để định danh duy nhất một row trong bảng.
Nó thường được dùng khi không có một cột nào duy nhất có thể làm primary key, hoặc khi muốn kết hợp nhiều thuộc tính để đảm bảo tính duy nhất của dữ liệu.
"Composite Key gồm nhiều cột, dùng khi không có cột nào duy nhất để làm primary key."

9. Transaction log / WAL (Write-Ahead Logging) là gì?

Transaction log hay Write-Ahead Logging (WAL) là cơ chế ghi lại các thay đổi của database vào log trước khi commit.
Mục đích là đảm bảo dữ liệu có thể phục hồi nếu hệ thống crash, giúp database duy trì ACID, đặc biệt là durability.
"WAL là ghi log các thay đổi trước khi commit, để có thể phục hồi dữ liệu khi crash."

10. Deadlock là gì? Làm sao phát hiện và xử lý?

Deadlock xảy ra khi 2 hoặc nhiều transaction chờ nhau release resource, dẫn đến tình trạng block lẫn nhau.
Trong PostgreSQL/MySQL, hệ thống thường tự phát hiện deadlock và rollback một transaction để giải phóng vòng chờ.
Best practice để tránh deadlock gồm: lock resource theo cùng thứ tự, tránh transaction quá dài, và thiết kế truy vấn hạn chế cạnh tranh tài nguyên.
"Deadlock là khi 2+ transaction chờ nhau release resource; DBMS tự detect và rollback; best practice là lock theo thứ tự và tránh long transaction."

11. Explain plan / EXPLAIN dùng để làm gì?

EXPLAIN hoặc EXPLAIN PLAN dùng để hiển thị execution plan của một query trong database.
Nó cho biết cách database thực hiện truy vấn, bao gồm index nào được dùng, thứ tự join, và cost estimate.
Thông tin này giúp developer tối ưu query, cải thiện hiệu năng và sử dụng index hiệu quả hơn.
"EXPLAIN hiển thị execution plan của query để tối ưu hiệu năng, xem index dùng và join strategy."

12. Stored procedure & Trigger khác nhau thế nào?

Stored Procedure là đoạn code SQL được lưu trong database và gọi thủ công khi cần. Nó có thể nhận tham số và trả kết quả.
Trigger là đoạn code SQL tự động thực thi khi một event xảy ra trên bảng, như INSERT, UPDATE, DELETE.
Nói cách khác, procedure do người gọi, trigger do sự kiện kích hoạt.
"Procedure gọi thủ công và có thể trả kết quả; Trigger tự chạy khi event INSERT/UPDATE/DELETE xảy ra."

13. View là gì? Khi nào dùng?

View là một virtual table trong database, được tạo ra từ một query.
View giúp ẩn sự phức tạp của join hoặc tính toán, cải thiện readability và tái sử dụng truy vấn.
Ngoài ra, một số DBMS như PostgreSQL hỗ trợ materialized view, lưu kết quả query nặng để tăng tốc truy xuất.
"View là virtual table giúp ẩn complexity và cải thiện readability; materialized view dùng để cache kết quả query nặng."

14. Làm sao tối ưu query performance?

Để tối ưu query performance, có thể làm:
Dùng index phù hợp cho các cột thường dùng trong WHERE, JOIN, ORDER BY.
Tránh SELECT * để chỉ lấy cột cần thiết.
Partition table để giảm dữ liệu scan.
Viết lại query (query rewrite) cho hiệu quả hơn.
Sử dụng caching với materialized view hoặc Redis.
Dùng EXPLAIN để phân tích execution plan và cải thiện chiến lược truy vấn.
*"Tối ưu query bằng index phù hợp, tránh SELECT *, partition table, query rewrite, caching, và dùng EXPLAIN để kiểm tra execution plan."*

15. Partition table là gì? Khi nào dùng?

Partition table là kỹ thuật chia một table lớn thành các phần nhỏ hơn dựa trên một key, range, hoặc hash.
Mỗi partition có thể được quản lý và truy vấn riêng, giúp tối ưu query và dễ quản lý dữ liệu.
Partition thường dùng với bảng có dữ liệu lớn, hoặc dữ liệu theo thời gian để cải thiện hiệu năng và bảo trì.
"Partition table chia table lớn thành nhỏ theo key/range/hash để tối ưu query và quản lý dữ liệu dễ hơn."

16. PostgreSQL có hỗ trợ JSON không?

PostgreSQL hỗ trợ dữ liệu JSON và JSONB.
JSON lưu dữ liệu dạng text.
JSONB lưu dạng binary, tối ưu query và index hơn.
Cả hai có thể dùng để lưu và truy vấn dữ liệu semi-structured, kết hợp với các operator, function và indexing để truy xuất hiệu quả.
"PostgreSQL hỗ trợ JSON và JSONB, giúp lưu, query và index dữ liệu semi-structured."

17. PostgreSQL vs MySQL khác nhau điểm nào?

PostgreSQL là RDBMS ACID strict, hỗ trợ nhiều tính năng nâng cao như CTE, window function, JSONB, thích hợp cho các ứng dụng cần consistency cao và dữ liệu phức tạp.
MySQL phổ biến, thường nhanh với workload read-heavy, hỗ trợ nhiều storage engine như InnoDB và MyISAM, dễ triển khai và quen thuộc với nhiều developer.
"PostgreSQL ACID strict, nhiều tính năng nâng cao; MySQL phổ biến, nhanh read-heavy, nhiều engine."

18. CTE (Common Table Expression) trong PostgreSQL/MySQL dùng khi nào?

CTE (Common Table Expression) là cách khai báo tạm thời một result set trong SQL để dùng trong query chính.
Nó giúp chia nhỏ query phức tạp, làm code dễ đọc và maintain.
Ngoài ra, CTE còn hỗ trợ recursive query, rất hữu ích cho các dữ liệu dạng cây hoặc đồ thị.
"CTE dùng để chia nhỏ query phức tạp hoặc viết recursive query dễ đọc và maintain."

19. MySQL engine InnoDB vs MyISAM?

MySQL có nhiều storage engine, phổ biến nhất là InnoDB và MyISAM:
InnoDB: hỗ trợ transaction, foreign key, và row-level locking, phù hợp cho ứng dụng cần consistency và concurrent write.
MyISAM: không hỗ trợ transaction hay foreign key, sử dụng table-level locking, thường nhanh cho read-heavy workload nhưng kém concurrency.
"InnoDB có transaction, foreign key, row-level lock; MyISAM không transaction, không foreign key, table-level lock."

20. Làm sao replication trong MySQL/PostgreSQL?

Trong MySQL, replication có thể triển khai theo nhiều cách: Master-Slave, Group Replication hoặc dùng GTID để quản lý transaction.
Trong PostgreSQL, replication thường dùng Streaming Replication cho copy toàn bộ database và Logical Replication để replicate dữ liệu theo table hoặc schema.
Replication giúp high availability, backup, và scale đọc (read scaling).
"MySQL dùng Master-Slave, Group Replication, GTID; PostgreSQL dùng Streaming và Logical Replication để đảm bảo HA và scale đọc."

21. Tối ưu bulk insert thế nào?

Để tối ưu bulk insert trong database:
Chia dữ liệu thành các batch nhỏ thay vì insert một lần lượng lớn.
Tạm thời disable index hoặc constraints để giảm overhead.
Với PostgreSQL, có thể dùng COPY command để load dữ liệu nhanh hơn từ file.
Những cách này giúp giảm thời gian insert và tránh lock lâu trên table.
"Tối ưu bulk insert bằng batch, tạm thời disable index/constraints, hoặc dùng COPY command trong PostgreSQL."

22. VACUUM trong PostgreSQL là gì?

VACUUM trong PostgreSQL là quá trình dọn các “dead tuple” sinh ra sau các lệnh UPDATE/DELETE, giúp reclaim space và tối ưu hiệu năng truy vấn.
Ngoài ra, có thể dùng VACUUM FULL để nén table và giải phóng không gian vật lý hoàn toàn, nhưng tốn thời gian và lock table.
"VACUUM dọn dead tuple sau update/delete để giải phóng space và tối ưu performance."

23. Transaction isolation trong PostgreSQL mặc định là gì?

Mức transaction isolation mặc định trong PostgreSQL là Read Committed.
Nghĩa là mỗi transaction chỉ nhìn thấy các dữ liệu đã được commit, giúp tránh dirty read nhưng vẫn có thể gặp non-repeatable read hoặc phantom read.
Nếu cần strict hơn, có thể chuyển sang Repeatable Read hoặc Serializable.
"Mặc định PostgreSQL dùng Read Committed, chỉ nhìn thấy dữ liệu đã commit, tránh dirty read."

24. MSSQL khác MySQL/PostgreSQL điểm chính?

MSSQL khác MySQL/PostgreSQL ở chỗ nó hướng mạnh vào enterprise features, như SSIS (integration), SSRS (reporting), T-SQL nâng cao, và CLR integration để chạy code .NET trong database.
Nó cũng có các tính năng quản lý dữ liệu nâng cao như clustered/non-clustered index, snapshot isolation, và transaction log mạnh mẽ để đảm bảo durability và hỗ trợ backup/restore linh hoạt.
"MSSQL có enterprise features (SSIS, SSRS, T-SQL, CLR), clustered/non-clustered index, snapshot isolation và transaction log mạnh, khác MySQL/PostgreSQL."

25. SQL Server có những loại index nào?

SQL Server hỗ trợ nhiều loại index để tối ưu truy vấn:
Clustered: dữ liệu vật lý trong table sắp xếp theo index.
Non-clustered: lưu pointer đến dữ liệu, không sắp xếp dữ liệu vật lý.
Columnstore: tối ưu truy vấn analytical, lưu dữ liệu theo column.
Full-text: tìm kiếm văn bản nhanh.
XML: index cho dữ liệu XML trong table.
"SQL Server có Clustered, Non-clustered, Columnstore, Full-text và XML index."

26. Deadlock & isolation level trong MSSQL?

Trong MSSQL, deadlock xảy ra khi hai hay nhiều transaction chờ nhau release resource, gây block lẫn nhau. SQL Server có deadlock detection tự động và sẽ rollback một transaction để giải phóng vòng chờ.
Về isolation level, SQL Server hỗ trợ nhiều mức, trong đó Snapshot Isolation giúp giảm blocking và deadlock bằng cách đọc dữ liệu từ snapshot thay vì chặn row.
Kết hợp isolation level và thiết kế transaction cẩn thận giúp cải thiện concurrency và tránh deadlock.
"MSSQL tự detect deadlock và rollback 1 process; snapshot isolation giúp giảm blocking và deadlock."

27. Temp table & Table variable khác nhau?

Trong MSSQL:
Temp table (#temp) là table tạm, có thể tạo index, có statistics, và tồn tại trong scope session. Thích hợp cho các query phức tạp hoặc dữ liệu lớn.
Table variable (@table) nhẹ hơn, tồn tại trong scope batch, không có statistics, và query optimizer không tối ưu nhiều. Thường dùng cho dữ liệu nhỏ hoặc biến tạm trong procedure.
"Temp table (#temp) hỗ trợ index/statistics, scope session; table variable (@table) nhẹ, scope batch, không tối ưu nhiều query."

28. How to backup & restore database in MSSQL?

Trong MSSQL, backup có thể thực hiện theo nhiều loại: Full, Differential, và Transaction Log.
Để restore database, có thể dùng SQL Server Management Studio (GUI) hoặc T-SQL với lệnh RESTORE DATABASE.
Quy trình backup/restore giúp bảo vệ dữ liệu và đảm bảo khả năng phục hồi sau sự cố.
"MSSQL backup: Full, Differential, Transaction log; restore bằng SSMS hoặc T-SQL RESTORE DATABASE."

29. Redis là gì? Key features?

Redis là một in-memory datastore cực nhanh, thường dùng để cache, message broker hoặc lưu dữ liệu tạm thời.
Key features bao gồm: key-value, hash, list, set, sorted set, và pub/sub.
Redis cũng hỗ trợ persistence qua RDB/AOF, replication, và clustering để scale và đảm bảo high availability.
"Redis là in-memory datastore nhanh, hỗ trợ key-value, hash, list, set, sorted set, pub/sub."

30. Redis dùng để làm gì?

Redis thường được dùng để:
Caching dữ liệu để giảm truy vấn database.
Session store cho web application.
Leaderboards hoặc tính toán điểm xếp hạng với sorted set.
Rate-limiting request trong API.
Message broker / pub-sub cho hệ thống real-time.
Redis nổi bật nhờ tốc độ in-memory và đa dạng cấu trúc dữ liệu.
"Redis dùng làm cache, session store, leaderboards, rate-limiting, và message broker."

31. Redis persistence có mấy loại?

Redis hỗ trợ persistence để dữ liệu không mất khi restart:
RDB (snapshotting): định kỳ lưu toàn bộ dataset vào file.
AOF (Append Only File): ghi log tất cả command để replay lại khi khởi động.
Có thể kết hợp RDB + AOF để vừa backup định kỳ vừa đảm bảo dữ liệu gần như realtime, tăng độ an toàn.
"Redis persistence gồm RDB snapshot (lưu định kỳ) và AOF (ghi log command); có thể kết hợp cả hai."

32. Redis pub/sub là gì?

Redis Pub/Sub là cơ chế Publish/Subscribe: client có thể publish message lên một channel, và các client subscribe channel đó sẽ nhận message ngay lập tức.
Đây là message broker nhẹ (lightweight) rất hữu ích cho real-time communication, như chat app, notification, hay update dashboard.
"Redis Pub/Sub là cơ chế publish/subscribe, dùng làm lightweight message broker cho real-time communication."

33. Atomic operation trong Redis là gì?

Atomic operation trong Redis là các lệnh thực hiện nguyên tử, nghĩa là hoàn toàn thành công hoặc không thay đổi gì, không lo race condition khi nhiều client cùng thao tác dữ liệu.
Ví dụ các lệnh như INCR, DECR, SETNX đều là atomic, giúp đảm bảo consistency trong môi trường concurrent.
"Atomic operation là lệnh Redis thực hiện nguyên tử, như INCR, DECR, SETNX, tránh race condition."

34. Redis Cluster khác standalone thế nào?

Redis Cluster phân mảnh dữ liệu (sharding) trên nhiều node, cho phép scale horizontal và tăng high availability.
Ngược lại, Redis standalone chỉ có một node, dữ liệu tập trung, dễ triển khai nhưng khó scale và thiếu tính sẵn sàng cao.
"Redis Cluster phân mảnh dữ liệu, scale horizontal; standalone chỉ 1 node, khó scale và ít HA."

35. Cache eviction policies?

Cache eviction policy quyết định cách xóa dữ liệu khi cache đầy:
LRU (Least Recently Used): xóa key ít được truy cập nhất.
LFU (Least Frequently Used): xóa key ít được sử dụng nhất.
TTL (Time To Live): xóa key sau một khoảng thời gian xác định.
Noeviction: không xóa key, insert mới sẽ báo lỗi khi full.
Chọn policy phù hợp giúp tối ưu cache hit rate và quản lý memory hiệu quả.
"Cache eviction policies gồm LRU, LFU, TTL, noeviction, quyết định cách xóa key khi cache đầy."

36. Redis transaction & Lua script?

Trong Redis:
Transaction dùng MULTI/EXEC để nhóm nhiều lệnh lại và thực hiện liên tục, đảm bảo tất cả lệnh trong transaction được thực hiện cùng nhau.
Lua script cho phép thực hiện multi-command operation nguyên tử, giúp tránh race condition khi thao tác phức tạp hoặc nhiều key cùng lúc.
Kết hợp transaction và Lua script giúp đảm bảo atomicity trong môi trường concurrent.
"Redis transaction dùng MULTI/EXEC; Lua script đảm bảo multi-command operation nguyên tử, tránh race condition."

37. Làm sao tối ưu Redis memory?

Để tối ưu Redis memory:
Gán TTL cho key để tự động xóa khi hết hạn.
Dùng hashes thay vì nhiều key kiểu string để tiết kiệm overhead.
Chọn eviction policy phù hợp (LRU, LFU…) để quản lý khi full memory.
Monitor memory usage thường xuyên và tối ưu data structure theo workload.
Những cách này giúp Redis hoạt động hiệu quả và tránh out-of-memory.
"Tối ưu Redis memory bằng TTL cho key, dùng hashes, eviction policy phù hợp, và monitor memory."

38. Deadlock trong MySQL/PostgreSQL xảy ra khi nào và cách debug?

Deadlock xảy ra khi 2 hoặc nhiều transaction chờ nhau release resource, dẫn đến vòng chờ và block.
Để debug:
PostgreSQL: xem pg_locks, kiểm tra log deadlock để tìm transaction liên quan.
MySQL (InnoDB): dùng SHOW ENGINE INNODB STATUS để xem thông tin deadlock và transaction đang chờ.
Ngoài ra, thiết kế transaction hợp lý và lock theo thứ tự giúp giảm nguy cơ deadlock.
"Deadlock xảy ra khi 2 transaction chờ nhau lock resource; debug PostgreSQL bằng pg_locks/log, MySQL bằng SHOW ENGINE INNODB STATUS."

39. Khi query quá chậm, bạn làm gì?

Khi query quá chậm, tôi thường:
Dùng EXPLAIN plan để xem execution plan và xác định bottleneck.
Kiểm tra index đã phù hợp chưa, và xem join strategy có tối ưu không.
Xem xét partitioning table nếu dữ liệu lớn.
Sử dụng caching hoặc materialized view để giảm thời gian truy vấn.
Kết hợp các bước này giúp cải thiện hiệu năng query hiệu quả.
"Khi query chậm: check EXPLAIN plan, index, join strategy, partition, caching, hoặc materialized view."

40. Làm sao thiết kế schema chuẩn cho ứng dụng web high-load?

Khi thiết kế schema cho web application high-load:
Normalize ban đầu để giảm redundancy và tăng consistency.
Thêm index phù hợp cho các cột thường query hoặc join.
Cache dữ liệu read-heavy bằng Redis hoặc memcached.
Partition table lớn để tối ưu query và quản lý dữ liệu.
Scale read replicas để giảm load trên master và tăng throughput.
Kết hợp các bước này giúp hệ thống vừa tối ưu performance vừa đảm bảo maintainability.
"Normalize ban đầu, thêm index, cache read-heavy data (Redis), partition table lớn, scale read replicas."

6. Message queue: IBMMQ, Kafka

1. Message Queue là gì?

Message Queue là một loại middleware giúp tách rời (decouple) producers và consumers, cho phép asynchronous processing.
Producer gửi message vào queue, consumer lấy message để xử lý khi sẵn sàng.
MQ giúp tăng scalability, reliability, và giảm tight coupling giữa các thành phần hệ thống.
"Message Queue là middleware decouple producer & consumer, hỗ trợ xử lý bất đồng bộ (asynchronous)."

2. Pub/Sub khác gì Queue (Point-to-Point)?

Queue (Point-to-Point): mỗi message được gửi đến một consumer duy nhất. Thường dùng cho task processing hoặc workload balancing giữa nhiều worker.
Pub/Sub: mỗi message được gửi đến nhiều subscriber cùng lúc. Thường dùng cho event notification, broadcasting, hoặc real-time updates.
Nói cách khác, Queue là 1 message – 1 consumer, Pub/Sub là 1 message – nhiều consumer.
"Queue: 1 message → 1 consumer; Pub/Sub: 1 message → nhiều subscriber."

3. Khi nào nên dùng MQ?

Message Queue nên dùng khi:
Async task processing như gửi email, notification, background jobs.
Decouple services để giảm tight coupling giữa các thành phần hệ thống.
Smoothing traffic spikes, buffer lượng request lớn để xử lý dần.
Retry / error handling, đảm bảo message được xử lý ngay cả khi consumer gặp lỗi tạm thời.
MQ giúp hệ thống scalable, resilient, maintainable hơn.
"Dùng MQ cho async tasks, decouple services, smoothing spikes, retry/error handling."

4. Message delivery mode?

Message delivery mode xác định cách message được đảm bảo khi gửi:
At-most-once: message có thể bị mất, nhưng không bao giờ duplicate.
At-least-once: message có thể được gửi nhiều lần (duplicate), nhưng không bị mất.
Exactly-once: message không mất và không duplicate, ví dụ Kafka hỗ trợ transactional messaging.
Lựa chọn delivery mode phù hợp tùy vào business requirement về độ tin cậy và performance.
"At-most-once: có thể mất; At-least-once: có thể duplicate; Exactly-once: không mất, không duplicate (Kafka transactional)."

5. Ack/Nack là gì?

Trong Message Queue:
Ack (Acknowledgement) là khi consumer xác nhận đã xử lý message thành công, để broker có thể xóa message khỏi queue.
Nack (Negative Acknowledgement) là khi consumer từ chối message, broker có thể retry hoặc đưa vào dead-letter queue.
Cơ chế này giúp đảm bảo message không bị mất và xử lý lỗi hiệu quả.
"Ack: consumer xác nhận đã xử lý; Nack: consumer từ chối, message có thể retry hoặc dead-letter."

6. IBM MQ là gì?

IBM MQ là một enterprise message broker chuyên hỗ trợ queue-based messaging.
Nó đảm bảo ACID, reliable delivery, và hỗ trợ transactional messaging, giúp các hệ thống enterprise giao tiếp an toàn, đảm bảo message không mất và xử lý theo thứ tự.
IBM MQ thường được dùng trong các hệ thống tài chính, ngân hàng, hoặc logistics, nơi độ tin cậy và consistency cực kỳ quan trọng.
"IBM MQ là enterprise message broker queue-based, hỗ trợ ACID, reliable delivery, transactional."

7. Persistent vs Non-persistent message?

Persistent message được lưu vào disk, đảm bảo không mất dữ liệu ngay cả khi broker crash.
Non-persistent message chỉ tồn tại trong memory, nhanh hơn nhưng dễ mất nếu hệ thống gặp sự cố.
Lựa chọn giữa hai loại phụ thuộc vào yêu cầu reliability vs performance của ứng dụng.
"Persistent: lưu vào disk, không mất dữ liệu; Non-persistent: chỉ trong memory, nhanh nhưng dễ mất."

8. MQ Queue & Topic khác nhau?

Trong Message Queue:
Queue dùng Point-to-Point pattern, mỗi message chỉ được xử lý bởi một consumer duy nhất. Thường dùng cho task processing hoặc load balancing.
Topic dùng Pub/Sub pattern, mỗi message được gửi đến nhiều subscriber cùng lúc, phù hợp cho event broadcasting hoặc notification.
Nói cách khác, Queue là 1 message → 1 consumer, Topic là 1 message → nhiều subscriber.
"Queue: 1 message → 1 consumer; Topic: 1 message → nhiều subscriber."

9. MQ Transaction là gì?

MQ Transaction là cơ chế gộp nhiều message send/receive thành một đơn vị atomic.
Điều này có nghĩa là tất cả các message trong transaction sẽ được commit cùng nhau hoặc rollback nếu có lỗi xảy ra.
MQ transaction giúp đảm bảo consistency và độ tin cậy cao trong hệ thống messaging.
"MQ Transaction gộp nhiều message thành unit atomic, commit hoặc rollback cùng nhau."

10. Dead Letter Queue (DLQ) là gì?

Dead Letter Queue (DLQ) là queue dùng để chứa những message không thể deliver đến consumer.
Nguyên nhân có thể là: consumer lỗi, message hết hạn (expiry), hoặc vượt kích thước limit.
DLQ giúp theo dõi, debug và xử lý message thất bại mà không làm gián đoạn hệ thống chính.
"DLQ là nơi chứa message không thể deliver do lỗi consumer, expiry, hoặc size limit."

11. High Availability trong IBM MQ?

Để đảm bảo High Availability (HA) trong IBM MQ:
Dùng Queue Manager clustering để load balance và failover.
Replication dữ liệu giữa các node để tránh mất message.
Multi-instance Queue Manager: một instance active, một hoặc nhiều standby để failover tự động.
Cơ chế HA failover giúp hệ thống tiếp tục hoạt động khi có node hoặc instance gặp sự cố.
"IBM MQ HA: Queue Manager clustering, replication, multi-instance queue manager, HA failover."

12. Khi nào dùng IBM MQ thay Kafka?

Nên dùng IBM MQ thay vì Kafka khi:
Cần transactional processing và exactly-once delivery.
Tích hợp với enterprise system hoặc legacy system vốn dùng MQ.
Yêu cầu strict message ordering.
Latency thấp hơn Kafka là cần thiết (small latency).
IBM MQ phù hợp với các hệ thống tài chính, ngân hàng, logistics, nơi độ tin cậy và consistency quan trọng hơn throughput cực cao.
"Dùng IBM MQ khi cần transactional processing, enterprise/legacy integration, strict ordering, và small latency."

13. Kafka là gì?

Kafka là một distributed streaming platform dùng để xử lý high-throughput pub/sub messaging và real-time data streaming.
Kafka lưu dữ liệu theo log-based storage, cho phép replay messages, đảm bảo durability và scalability.
Nó thường dùng cho event streaming, log aggregation, real-time analytics, hoặc data pipeline.
"Kafka là distributed streaming platform, hỗ trợ high-throughput pub/sub, log-based storage, real-time streaming."

14. Topic, Partition & Offset là gì?

Trong Kafka:
Topic là kênh logic để phân loại message.
Partition là phần nhỏ của topic, giúp chia load, mỗi partition giữ order riêng của message.
Offset là vị trí của message trong partition, giúp consumer track progress và quản lý replay.
Sự kết hợp này giúp Kafka scale tốt và đảm bảo ordering trong partition.
“Topic: logical channel; Partition: chia topic, giữ order; Offset: vị trí message, track progress consumer.”

15. Kafka Delivery semantics?

Kafka delivery semantics xác định cách message được đảm bảo:
At-most-once: message có thể bị mất, không duplicate.
At-least-once: message có thể duplicate, nhưng không bị mất.
Exactly-once: không mất, không duplicate, hỗ trợ qua transactional producer/consumer.
Lựa chọn semantics phụ thuộc vào requirement về reliability và idempotency của ứng dụng.
"Kafka delivery: At-most-once (có thể mất), At-least-once (có thể duplicate), Exactly-once (không mất, không duplicate)."

16. Producer & Consumer Group?

Trong Kafka:
Producer gửi message vào topic.
Consumer Group là tập hợp nhiều consumer cùng group, chia partition để scale horizontally và đảm bảo mỗi message trong partition chỉ được xử lý bởi một consumer trong group.
Cơ chế này giúp Kafka vừa parallel processing vừa đảm bảo message không bị xử lý trùng lặp.
"Producer gửi message vào topic; Consumer Group chia partition giữa nhiều consumer để scale và tránh duplicate processing."

17. Kafka retention & compaction?

Trong Kafka:
Retention xác định thời gian hoặc dung lượng lưu trữ message trước khi xóa.
Compaction giữ bản ghi mới nhất theo key, xóa các bản cũ, giúp dữ liệu theo key luôn up-to-date.
Retention thường dùng cho log, event streaming, trong khi compaction dùng cho stateful data, như bảng lookup hoặc cache trong Kafka topic.
"Retention: giữ message theo thời gian/dung lượng; Compaction: giữ message mới nhất theo key, overwrite cũ."

18. High Availability & Fault-tolerance?

Để đảm bảo High Availability và Fault-tolerance trong Kafka:
Mỗi partition được replicate trên nhiều broker (leader + followers).
Leader chịu trách nhiệm đọc/ghi, followers replicate dữ liệu.
Nếu leader fail, một follower tự động lên làm leader để tiếp tục phục vụ.
Cơ chế này giúp Kafka resilient, không mất dữ liệu và đảm bảo throughput ngay cả khi broker gặp sự cố.
"Kafka HA: replication per partition, leader/follower, automatic failover."

19. Kafka vs IBM MQ khác gì?

So sánh Kafka và IBM MQ:
Messaging type: IBM MQ dùng queue-based, Kafka dùng log-based pub/sub.
Delivery guarantee: IBM MQ hỗ trợ exactly-once, ACID, Kafka hỗ trợ at-least-once và exactly-once transactional.
Throughput: Kafka rất cao, IBM MQ trung bình.
Persistence: Kafka mặc định lưu log, IBM MQ lưu tùy cấu hình.
Use case: IBM MQ dùng cho enterprise integration, transactional system, Kafka dùng cho real-time streaming, analytics, event sourcing.
"IBM MQ: queue-based, ACID, transactional, medium throughput, enterprise integration; Kafka: log-based pub/sub, high throughput, exactly-once transactional, real-time streaming."

20. Kafka consumer offset commit là gì?

Trong Kafka, consumer offset commit là cơ chế giúp consumer track progress của mình trong partition.
Auto-commit: Kafka tự động commit offset theo interval.
Manual commit: consumer chủ động commit offset sau khi xử lý message thành công, giúp đảm bảo message processing đúng và tránh mất message.
Offset commit quan trọng để replay message hoặc restart consumer mà không bỏ sót dữ liệu.
"Consumer offset commit: track progress bằng offset; có auto-commit hoặc manual commit để đảm bảo xử lý message đúng."

21. Kafka vs RabbitMQ/IBM MQ dùng khi nào?

Khi chọn giữa Kafka và RabbitMQ/IBM MQ:
Kafka phù hợp cho xử lý lượng message lớn, truyền dữ liệu thời gian thực, và lưu trữ message lâu dài. Thích hợp cho các trường hợp như event sourcing, phân tích dữ liệu, tổng hợp log.
RabbitMQ/IBM MQ phù hợp cho xử lý theo giao dịch, tích hợp hệ thống enterprise, yêu cầu giữ thứ tự message chính xác, xử lý message nhỏ, và đảm bảo mỗi message chỉ được xử lý một lần.
Tóm lại, Kafka ưu tiên throughput và streaming, MQ ưu tiên độ tin cậy và transactional.
"Kafka: xử lý lượng message lớn, truyền dữ liệu thời gian thực, lưu trữ lâu;
MQ: giao dịch, tích hợp enterprise, thứ tự message chính xác, message nhỏ."

22. Tại sao dự án lại dùng cả hai?

Dự án dùng cả Kafka và IBM MQ vì mỗi công cụ phục vụ mục đích khác nhau. IBM MQ được dùng cho các giao dịch quan trọng, cần đảm bảo tính toàn vẹn và thứ tự — phù hợp với hệ thống legacy và business-critical. Kafka thì dùng cho streaming, event sourcing và xử lý dữ liệu lớn theo thời gian thực.

"IBM MQ dùng cho transactional, critical business; Kafka cho streaming và event-driven analytics."

23. Pattern tích hợp MQ & Kafka?

Thường dùng pattern bridge giữa MQ và Kafka. Ví dụ: MQ → Kafka bridge, trong đó consumer đọc message từ IBM MQ rồi đẩy sang Kafka topic để xử lý realtime. Ngược lại, Kafka → MQ khi cần đưa event từ Kafka sang hệ thống legacy dùng MQ.

"Dùng MQ–Kafka bridge: MQ consumer đẩy message sang Kafka, hoặc Kafka stream gửi ngược về MQ để tích hợp hệ thống cũ."

24. Khi migrate từ MQ sang Kafka cần lưu ý gì?

Khi migrate từ MQ sang Kafka, cần chú ý đến delivery semantics (at least once / exactly once), đảm bảo thứ tự message (ordering guarantee), ranh giới transaction, và cơ chế retry – DLQ để không mất dữ liệu. MQ và Kafka có mô hình khác nhau, nên cần mapping cẩn thận về behavior.

"Cần chú ý delivery semantics, thứ tự message, transaction, và retry/DLQ để đảm bảo không mất dữ liệu khi chuyển từ MQ sang Kafka."

25. Message ordering đảm bảo thế nào ở Kafka & IBM MQ?

IBM MQ đảm bảo thứ tự message (FIFO) trong một queue, đặc biệt khi dùng transactional queue. Kafka cũng đảm bảo FIFO nhưng chỉ trong từng partition — nếu topic có nhiều partition thì sẽ mất thứ tự toàn cục, chỉ giữ ordering theo key.

"MQ giữ FIFO trong queue, còn Kafka chỉ giữ thứ tự trong từng partition, không đảm bảo global ordering."

26. Dead Letter Queue / Poison Message là gì?

Dead Letter Queue (DLQ) là hàng đợi đặc biệt lưu trữ những message không thể xử lý do lỗi như parse fail, timeout, hay retry nhiều lần vẫn thất bại. Nó giúp không làm gián đoạn luồng chính và cho phép xử lý thủ công hoặc debugging sau.

"DLQ chứa các message lỗi không xử lý được để xem lại và xử lý sau, tránh làm tắc nghẽn queue chính."

27. Retry & backoff strategy?

Retry & backoff strategy giúp xử lý message thất bại mà không làm tắc queue. Thường dùng exponential backoff hoặc fixed delay, giới hạn số lần retry tối đa, và nếu vẫn thất bại thì gửi message vào Dead Letter Queue để xử lý thủ công.

"Dùng exponential/fixed backoff, giới hạn retry, gửi message thất bại vào DLQ."

28. High throughput Kafka design?

Để đạt high throughput trên Kafka, thường tăng số partition, sử dụng batch produce, async produce, nén message (compression), và cân nhắc replication factor trade-off giữa hiệu năng và độ bền dữ liệu.

"High throughput Kafka: nhiều partition, batch/async produce, compression, replication factor hợp lý."

29. Monitoring & Metrics?

Khi monitoring:

Kafka: theo dõi consumer lag, throughput của consumer/producer, tình trạng broker và partition health.

IBM MQ: theo dõi queue depth, message age, channel status, và transaction success/failure.

"Kafka: lag, throughput, broker health; MQ: queue depth, message age, channel status."

30. Kafka exactly-once xử lý ra sao?

Để đạt exactly-once, Kafka dùng transactional producer kết hợp idempotent consumer. Producer đảm bảo atomic write trên một hoặc nhiều topic, còn consumer idempotent tránh duplicate khi xử lý message. Kết hợp này giúp dữ liệu được xử lý đúng một lần duy nhất.

"Kafka EOS: transactional producer + idempotent consumer → đảm bảo message xử lý đúng 1 lần."

31. IBM MQ transactional message?

IBM MQ hỗ trợ transactional message, trong đó commit/rollback đảm bảo atomic delivery. Nếu transaction commit, tất cả message được gửi hoặc nhận thành công; nếu rollback, mọi thay đổi đều bị hủy, đảm bảo consistency và reliability.

"IBM MQ transactional message dùng commit/rollback để đảm bảo atomic delivery."

32. Khi queue/topic full hoặc broker down?

Khi queue/topic đầy hoặc broker down:

IBM MQ: producer bị block hoặc message gửi vào DLQ; cần alert admin để xử lý.

Kafka: hệ thống tạo backpressure, producer retry, cần monitor disk usage và broker health để tránh data loss.

"MQ: block producer hoặc DLQ, alert admin; Kafka: backpressure, retry, monitor disk/broker."

33. Scaling Kafka & IBM MQ thế nào?

Để scale:

Kafka: tăng số partition và brokers, sử dụng consumer groups để phân phối load, đảm bảo high throughput.

IBM MQ: scale bằng cách clustering queue manager hoặc multi-instance, giúp tăng khả năng xử lý và độ sẵn sàng.

"Kafka: scale partition/brokers, consumer groups; MQ: queue manager clustering, multi-instance."

34. Tại sao dùng Kafka cho analytics nhưng vẫn dùng MQ cho core business?

Kafka dùng cho analytics vì có thể xử lý lượng sự kiện lớn, hỗ trợ streaming và real-time processing. IBM MQ vẫn dùng cho core business vì đảm bảo delivery, transactional safety và tích hợp với hệ thống legacy.

"Kafka xử lý high-volume events, streaming; MQ đảm bảo delivery, transaction, legacy integration."

35. Troubleshooting slow consumer?

Khi consumer chạy chậm, cần kiểm tra:

Kafka: consumer lag, partition imbalance.

IBM MQ: queue depth, message age.

Ngoài ra có thể tối ưu bằng parallelization, batch processing, và giảm thời gian xử lý mỗi message.

"Check lag (Kafka) hoặc queue depth (MQ); dùng parallelization, batching, tối ưu processing time."