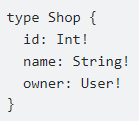
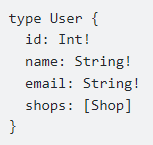
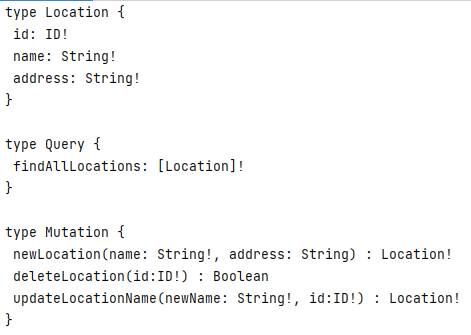
1. **In-memory database**
   * **IMDB:** tức là database được lưu trữ trực tiếp trên RAM của máy tính, thay vì lưu trữ trên disk cứng. IMDB sẽ cho tốc độ truy xuất nhanh hơn đáng kể cho với database lưu trữ trên disk truyền thống.
   * Nhưng nhược điểm là vì chúng lưu trên RAM nên khả năng lưu trữ có giới hạn và tính toàn vẹn dữ liệu không cao, khả năng bảo mật dữ liệu kém.
   * Một số IMDB: H2, Redis, SQLite, …
2. **Web services**
   * **Web services** **(dịch vụ web):** hiểu nôm na là những cách để cho 2 máy tính có thể giao tiếp với nhau (client – server) và truyền tải dữ liệu liền mạch mà không bị bó buộc vào hệ điều hành hay ngôn ngữ lập trình.
   * Ví dụ: tức là ta có thể dùng web service để cho app Java và app Python có thể giao tiếp với nhau. Hoặc cho một điện thoại có thể giao tiếp với một máy chủ server.
   * Cách thức: server định nghĩa một standard format (định dạng chuẩn) cho request và response. Khi user gửi request thông qua giao thức HTTP hoặc HTTPS, sử dụng ngôn ngữ thường là JSON hoặc XML, sau đó server xử lí request đó và trả về response.
   * **SOAP** (Simple Object Access Protocol): là phương thức gửi tin nhắn để giao tiếp giữa 2 hệ thống bằng ngôn ngữ XML.
   * **API** (Application Programming Interface): nhẹ hơn và sắp xếp hợp lý hơn web services
   * **Microservices:** giống với API. Tức là microservice bao gồm hoàn toàn những service riêng biệt. Tức là những service đó khi gom lại với nhau sẽ tạo thành một hệ thống hoàn chỉnh có đầy đủ chức năng,
3. **REST**
   * **REST (Representational State Transfer) :** REST hiểu đơn giản nó là một bộ các ràng buộc và quy định để thiết kế hệ thống. Hệ thống tuân theo REST, gọi là hệ thống RESTful, đặc trưng bởi việc chúng ko có trạng thái (stateless) và hoạt động theo mô hình client-server tách biệt.
   * Hệ thống hoạt động theo mô hình client-server, trong đó server là tập hợp các service nhỏ lắng nghe các request từ client. Với từng request khác nhau thì có thể một hoặc nhiều service xử lý.
   * Stateless: nghĩa là server và client ko lưu trạng thái của nhau => mỗi request lên server thì client phải đóng gói thông tin đầy đủ rồi gửi lên để server hiểu được.
   * Khả năng caching: các response có thể lấy ra từ cache. Bằng cách cache vào response, client gửi request mà lặp lại (kiểu như phân trang) thì có thể lấy dữ liệu từ cache => giảm tải việc server phải xử lí request, client cũng nhận được thông tin nhanh hơn.
   * Phân lớp hệ thống: tách biệt hoàn toàn các lớp (request handler, service handler, database handler), mỗi lớp chỉ giao tiếp với lớp ở trên hoặc dưới nó => tách biệt các thành phần => dễ mở rộng.
   * Trong kiến trúc REST, client gửi request tới server để modify resource và sau đó server sẽ gửi response tới client.
4. **GraphQL**
   * Nhược điểm của REST API là chúng luôn luôn trả về cấu trúc data cố định trong response mà dev define ra từ trước, mà điều này sẽ làm tốn thời gian khi ta gửi request lên và response trả về nhiều data thừa ko cần thiết, hoặc là thiếu những data mà ta cần.
   * VD: khi ta gửi cần request để get data bao gồm user, cart, product, ta sẽ phải gửi 3 request tới 3 endpoint khác nhau: /user, /cart, /product (bởi vì dev đã design API như vậy). GraphQL có thể linh hoạt hơn bằng cách gom cả 3 data cần get đó vào 1 request duy nhất và gửi lên server và nhận về response bao gồm cả user, cart, product.
   * GraphQL ra đời với mục đích có thể linh hoạt hơn trong việc lựa chọn ta muốn nhận data gì từ response.
   * GraphQL là query language cho API, dùng để request chính xác những data mà ta muốn từ server (có thể nhiều hơn hoặc ít hơn). Điều này sẽ làm tăng sự linh hoạt và performance.
   * **Schema:** hiểu nôm na giống như request body của REST API. Được viết bởi **SDL -** **Schema Definition Language.** Khá hao hao JSON.
   * GraphQL có một **hệ thống type** riêng để xác định **schema** của chúng. Schema bao gồm data point, data types, quan hệ giữa chúng và **Query** để select dữ liệu hay **Mutation** thao tác dữ liệu.
     + ****
     + ****
     + **!** là trường bắt buộc
     + owner: User để thể hiện foreign key owner của Shop là User.
     + shops: [Shop] để thể hiện quan hệ many rằng User có nhiều Shop. Array được kí hiệu bởi []
     + 1 schema chỉ có thể có 1 root Query và 1 root Mutation.
   * **GraphiQL** là một web app để giao tiếp với graphQL server và execute query hay mutation.
   * **Resolver:** có chức năng khá tương đồng với RestController, giúp cho app Spring có thể nhận diện và call đúng method ta muốn trong request cho GraphQL query. Chỉ khác là ta sẽ ko cần phải mapping endpoint như RestController mà ta chỉ cần map với tên method trong Resolver là được.
     + **QueryResolver** sẽ dùng để resolve query. Khi code ta sẽ tạo 1 component implement class **GraphQLQueryResolver,** và tên method trong đó sẽ match tới tên query trong schema.
     + **MutationResolver** sẽ dùng để resolve mutation. Khi code ta sẽ tạo 1 component implement class **GraphQLMuationResolver,** và tên method trong đó sẽ match tới tên mutation trong schema.
   * **GraphQLError:** khi có exception được throw ra, reponse trả về sẽ có thêm field là error và bên trong chứa message lỗi. Trong này sẽ chứa **extentions** có dạng key value giống như trên.
5. Annotation
   * **@RestController = @Controller + @ResponseBody**
     + Mặc định @Controller sẽ trả về **View** (tức là HTML thuần, chỉ dùng cho **HTML template engine** như **Thymleaf**…). Nếu muốn trả về kiểu JSON/XML thì thêm annotation @ResponseBody.
     + Còn @RestController là đã bao gồm @Controller + @ResponseBody rồi. Mặc định sẽ trả về JSON/XML thay vì HTML.
   * **@RequestMapping**
     + Có thể dùng bên trên class controller, để chỉ định endpoint gốc cho toàn bộ method bên trong nó
     + Hoặc vẫn có thể dùng bên trên method trong controller (ít dùng)
       1. @RequestMapping(value = "/users", method = RequestMethod.GET)
   * **@Entity**
     + Đánh dấu 1 class java thành một entity để có thể thao tác với database.
   * **Tại sao bỏ annotation @Repository mà app vẫn run được?** Vì bản chất repository khi viết trong code là interface, nhưng khi app SpringBoot chạy thì sẽ tự tạo ra class implementation của interface đó -> tức là đã tạo ra bean rồi. @Repository lúc này chỉ mang tính chất là marker anotation.
6. **Microservices Architecture (MSA)**
   * Trước khi kiến trúc microservice ra đời, các app được build theo **kiến trúc monolithic.**
   * **Monolithic architecture** nghĩa là toàn bộ app đó được build thành 1 file duy nhất là được deploy lên một server duy nhất. Nhược điểm của kiến trúc monolith là: code quá phụ thuộc vào nhau, chỉ cần một thay đổi nhỏ ở service này sẽ ảnh hưởng tới toàn bộ các service khác liên quan tới nó, dẫn tới việc phải test, build và deploy lại toàn bộ project đó. Việc này sẽ càng trở nên khó khăn khi app càng mở rộng
   * **Kiến trúc microservices** được ra đời để khắc phục những nhược điểm của kiến trúc monolith: hệ thống này sẽ phân tách các service thành các component hoàn toàn riêng biệt mà vẫn có thể giao tiếp với nhau.
   * Ưu điểm của microservice thì cũng nhiều:
     + Mỗi một service sẽ được tách thành một app riêng, được build trên một server riêng, và có database riêng biệt. Điều này sẽ giảm tải quá trình test, build và deploy khi code thay đổi.
     + Các service được tách thành các app riêng biệt, vậy nên ta có thể dùng các ngôn ngữ khác nhau cho từng service mà vẫn có thể giao tiếp được giữa các service với nhau.
     + Khi hệ thống mở rộng thì khi ta chỉnh sửa service nào sẽ ko ảnh hưởng tới các service còn lại vì vốn dĩ các service được build ở các server khác nhau.
     + Khi ta cần deploy một service lên 1 server khoẻ hơn thì với cloud computing ta chỉ cần thực hiện vài cú click.
   * Nhược điểm của kiến trúc này thì cũng có:
     + Sẽ làm hệ thống trở nên phức tạp hơn, code khó đọc hơn -> yêu cầu trình độ dev cao hơn để có thể hiểu luồng đi của code hay cách các service gọi nhau như thế nào.
     + Khi truy vấn mà phải đi qua nhiều service sẽ khiến code chạy chậm hơn.
     + Nếu hệ thống ko quá to mà build theo kiến trúc microservice thì hệ thống sẽ rối rắm một cách ko cần thiết và tốn thời gian dev.
7. **Eureka** **server**
   * Eureka server (được tạo ra bởi Netflix) là một server để đăng kí service trong hệ thống microservices. Tức là khi ta đăng kí service đó dưới một cái tên (name) thì khi cần gọi nó ta chỉ cần gọi thông qua name thay vì IP address và port.
   * Để apply hệ thống này ta cần deploy 2 thứ :
     + **Eureka server** : là server để các microservice register những service của mình vào
     + **Eureka client** : khi các service của microservices được register vào eureka serser, chúng sẽ được gọi là các eureka client.