Kiểm thử phần mềm trong kiến trúc Microservice

Đỗ Đức Bích Ngân [a*]

[a]*Khoa Công nghệ Thông tin, Trường Đại học Ngoại ngữ - Tin học Thành phố Hồ Chí Minh*
[*]*Tác giả liên hệ: Email: ngan.ddb@huflit.edu.vn | Điện thoại: 077.3144.044*

Tóm *tắt*

Kiến trúc microservices không phải là một kiến trúc mới. Nó đã được sử dụng trong hơn một thập kỷ nay bởi những nhà công nghệ hàng đầu như Amazon, Google và Facebook. Ví dụ khi ta tra cứu một nội dung nào đó trên Google, để truy xuất kết quả có liên quan, Google sẽ gọi tới gần 70 microservices khác nhau. Một đơn hàng được đặt qua hệ thống Amazon cũng liên kết hơn 40 dịch vụ vi mô này.Với tốc độ phát triển công nghệ hiện nay, quy mô phần mềm ngày càng trở nên rộng lớn và phức tạp, kết nối bao phủ hầu hết mọi vấn đề liên quan đến đời sống, học tập, khoa học…. Tốc độ xây dựng phần mềm kiến trúc microsoft đang có xu hướng tăng dần làm cho nhu cầu kiểm thử phần mềm kiến trúc này ngày càng phổ biến và phát triển.

Từ khoá: microservice; microservice testing strategies; unit testing;intergrate testing, component testing; E2E testing

1. GIỚI THIỆU

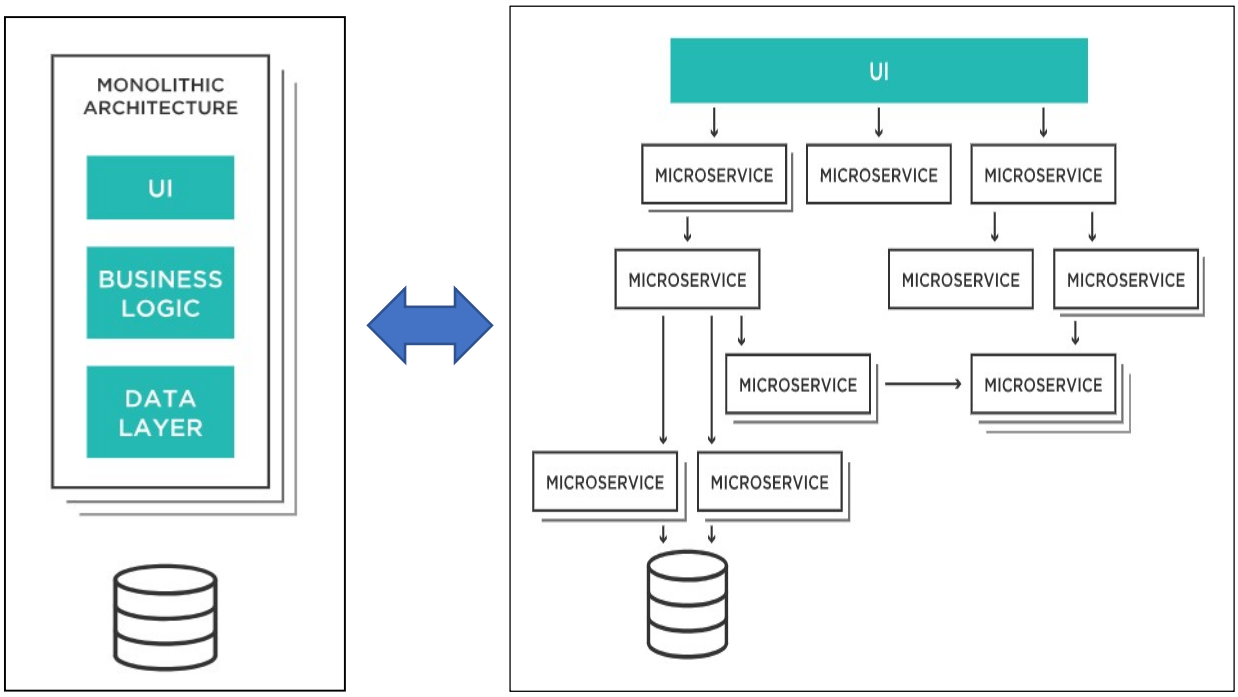
Vậy kiến trúc microservices là gì?

Kiến trúc microservice, hay đơn giản là microservices, là một phương pháp phát triển hệ thống phần mềm đặc biệt cố gắng tập trung vào việc xây dựng các mô-đun đơn chức năng hoặc mục đích đơn với các giao diện và hoạt động được xác định rõ ràng. Xu hướng này đã trở nên phổ biến trong những năm gần đây khi các doanh nghiệp mong muốn trở nên linh hoạt hơn và hướng tới cách tiếp cận DevOps và kiểm thử liên tục. Microservices có thể giúp tạo phần mềm có thể mở rộng, có thể kiểm tra và có thể được phân phối thường xuyên hầu như hàng tuần, thậm chí là hàng ngày.

Thông thường, các microservices hiển thị cấu trúc bên trong tương tự bao gồm một số hoặc tất cả các lớp được hiển thị.

Bất kỳ chiến lược thử nghiệm nào được sử dụng đều phải nhằm mục đích cung cấp phạm vi bao phủ cho từng lớp và giữa các lớp của dịch vụ trong khi vẫn giữ được trọng lượng nhẹ.

Vì mỗi dịch vụ có một chức năng hạn chế, nên nó có kích thước và độ phức tạp nhỏ hơn nhiều. Thuật ngữ microservice xuất phát từ thiết kế chức năng rời rạc này, không phải từ kích thước vật lý của nó.



Hình 1. So sánh kiến trúc truyền thống nguyên khối với kiến trúc microservices

Rõ ràng với kiến trúc microservices, các nhà phát triển phần mềm có thể hoạch định phần mềm thành các môđun nhỏ, thậm chí là rất nhỏ, tăng tính khả thi và phân phối nguồn lực lẫn thời gian thực hiện. Một số lợi ích trực tiếp từ kiến trúc này có thể tóm tắt:

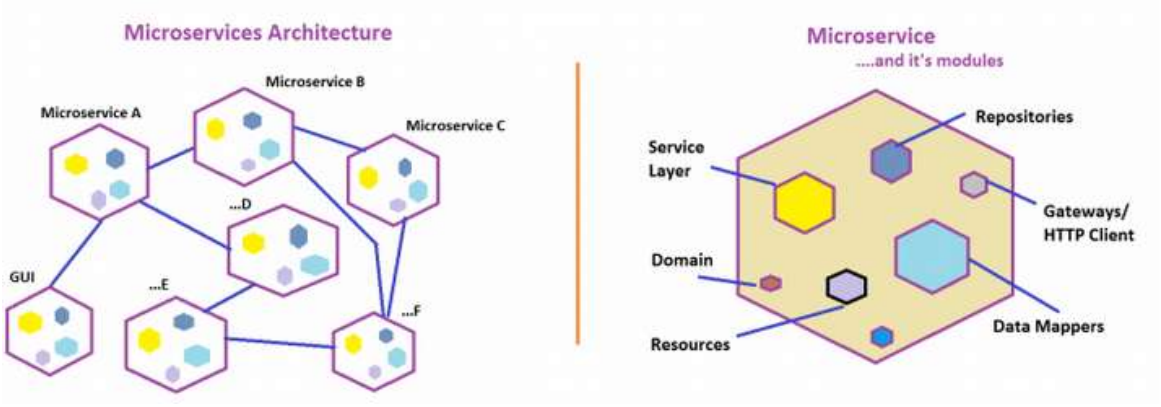
- Các thành phần phần mềm (mỗi microservice) đảm nhận một hoặc một vài nhiệm vụ độc lập và chạy trong tiến trình của nó, giao tiếp bên ngoài với cơ chế gọn nhẹ.
- Các thành phần của ứng dụng có thể được xây dựng trên các ngôn ngữ lập trình khác nhau, trên những công nghệ và lưu trữ cơ sở dữ liệu khác nhau.
- Khả năng mở rộng ứng dụng cực kỳ linh hoạt.
- Có thể triển khai theo nguyên tắc cloud-native[3] và có thể triển khai liên tục.
- Chi phí về nhân lực, thời gian thực hiện thấp, tính tái sử dụng cao.
- Sự cô lập trong các dịch vụ vi mô giúp hạn chế hoặc cách ly lỗi.
- Liên kết ràng buộc lỏng cho phép nâng cấp và cải tiến theo từng phần

Tính linh hoạt khi triển khai và sự gia tăng của các tùy chọn triển khai không cần máy chủ và chức năng như một dịch vụ trên đám mây (chẳng hạn như AWS Lambda và Microsoft Azure Cloud Functions) đã tạo ra môi trường hoàn hảo cho các microservices phát triển trong bối cảnh CNTT ngày nay. Các nền tảng đám mây này cho phép các dịch vụ và chức năng vi mô được điều chỉnh từ không hoạt động thành khối lượng lớn và hoạt động trở lại trong khi khách hàng chỉ trả tiền cho dung lượng máy tính họ sử dụng. Khi các doanh nghiệp liên tục tìm cách linh hoạt hơn, giảm tắc nghẽn và cải thiện thời gian phân phối ứng dụng, kiến trúc microservices tiếp tục ngày càng phổ biến.

Hiện nay có rất nhiều thông tin, kỹ thuật về việc xây dựng, phát tiển mô hình microservices, tuy nhiên việc kiểm thử ứng dụng kiến trúc microsrevice testing đang trở thành thách thức hiện nay. Mỗi dịch vụ vi mô xem như một đơn vị nhỏ của hệ thống. Kiểm thử đơn vị là yêu cầu đòi hỏi đối với người lập trình, nó thường chiếm nhiều thời gian, chưa kể giai đoạn kiểm thử tích hợp và kiểm thử hệ thống. Do sự chia vi mô, nên số lượng kết

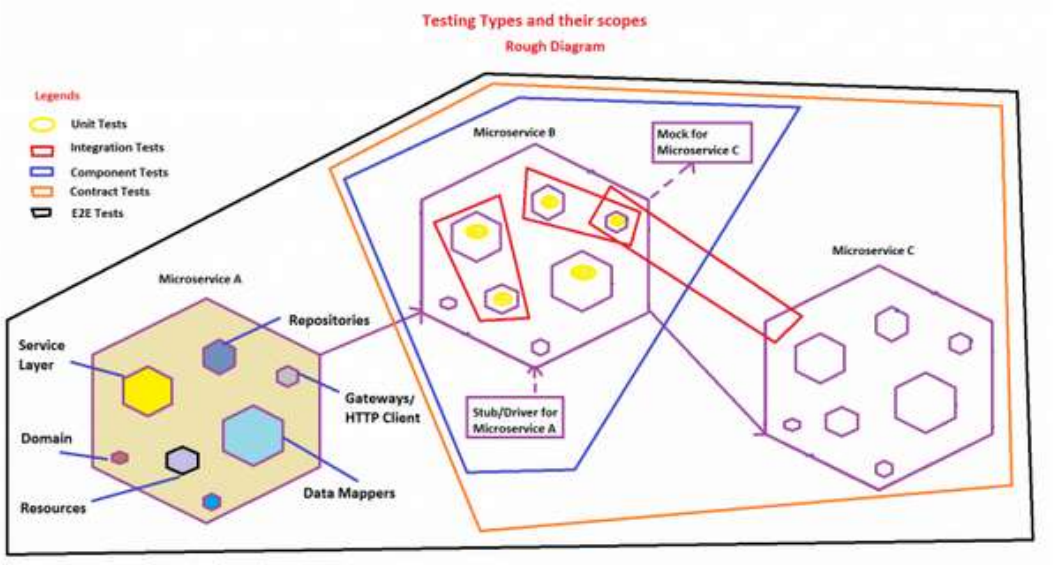
[3] https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition

hợp và phụ thuộc gia tăng đáng kể và có thể là rất lớn đối với các ứng dụng phức tạp và quy mô và nó thay đổi văn hóa làm việc của các nhóm theo cách các nhóm làm việc cùng nhau.



Hình 2. Cấu trúc bên trong kiến trúc microservice

Các loại kiểm tra khác nhau cần được thực hiện ở các lớp vi dịch vụ khác nhau. Giả sử Microservices A, B và C đang hoạt động cùng với nhiều Microservices khác (không được hiển thị trong hình). Sơ đồ sau minh họa tất cả các loại thử nghiệm và ranh giới của chúng bằng cách lấy một microservice (Microservice B) làm ví dụ.



Hình 3. Các loại kiểm thử thực hiện trên các đơn vị microservices

Một số khái niệm chung

Test double: còn gọi là *stunt double* hay đóng thế kép, là một thuật ngữ chung cho bất kỳ trường hợp nào người ta thay thế một đối tượng sản phẩm bằng một cái giả khác vì mục đích thử nghiệm. Có nhiều loại *test double* khác nhau:

- ***Dummy object:*** là đối tượng giả lập cho nó có mặt nhưng không thực sự được sử dụng, thường dùng để điền các thông số trong danh sách bắt buộc phải có để kiểm nghiệm sản phẩm chính

- ***Fake** object:* Các đối tượng giả thực sự có triển khai hoạt động, nhưng thường sử dụng một số phím tắt khiến chúng không phù hợp để sản xuất ví dụ In- memory database[4]

- ***Stubs** objects:* cung cấp các chế độ trả lời (respond) đã được soạn trước cho các cuộc gọi dùng trong kiểm thử, thường không cung cấp bất cứ gì thêm ngoại trừ các yêu cầu lập trình để kiểm tra.

- ***Spies*** là dạng ***stubs*** mà có ghi lại một số thông tin dựa trên cách chúng được gọi, ví dụ log hay dịch vụ email ghi lại bao nhiêu thư mà nó đã được gửi đi.

- ***Mocks***: được lập trình trước theo đặc tả kỹ thuật của nó tương ứng với yêu cầu thực tiễn của object sản phẩm tuy nhiên vẫn là giả lập, thường gia lập các tình huống xử lý các ngoại lệ hoặc các tình huống không mong đợi trong quá trình kiểm thử.

2. CHIẾN LƯỢC KIỂM THỬ MICROSERVICE

2.1. Những thách thức trong việc triển khai chiến lược thử nghiệm dịch vụ vi mô

Kiến trúc Microservices bao gồm nhiều dịch vụ độc lập nhỏ tích hợp với nhau để tạo thành toàn bộ ứng dụng. Các dịch vụ nhỏ này tương tác với nhau trong môi trường sản xuất cho các chức năng của ứng dụng, mặc dù các dịch vụ nhỏ này rất đơn giản trong khi giao tiếp với nhau lại nảy sinh nhiều phức tạp. Khi mức độ chi tiết của ứng dụng tăng lên.

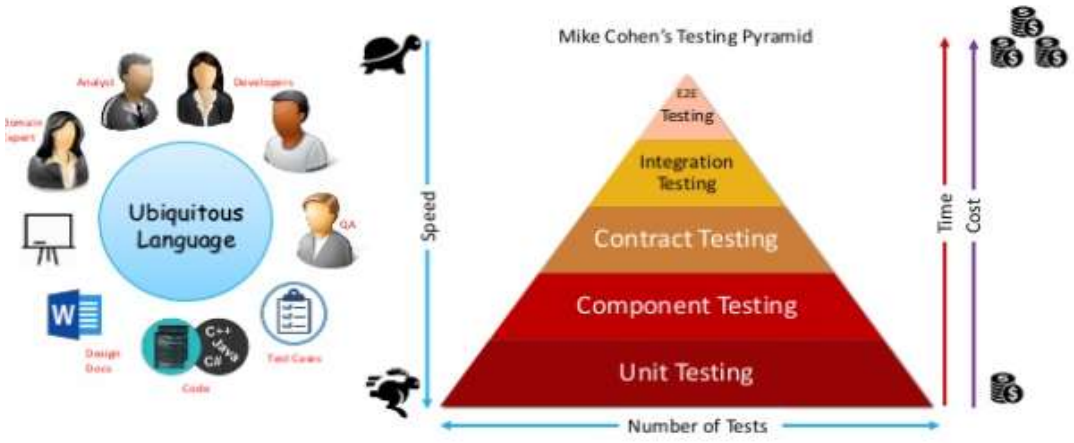
Kiến trúc microservices có ảnh hưởng đến sự phát triển và kiểm thử nghiệm của các ứng dụng phần mềm. Ranh giới, trước đây vốn bị ẩn, giờ sẽ hiển thị khi ta chuyển sang kiến trúc microservices tùy thuộc vào mức độ chi tiết. Điều này đòi hỏi các nhà quản lý kiểm thử phải sử dụng một cách tiếp cận khác để kiểm thử nghiệm microservices.

Yêu cầu khi kiểm thử microservice phải đạt được:

- Code nên làm những gì nó nên làm
- Cung cấp phản hồi nhanh chóng, đáng tin cậy và chính xác
- Giúp bảo trì tổng thể dễ dàng hơn

Vì kiến trúc microservices dựa nhiều hơn vào sự phụ thuộc over-the-wire (từ xa) và ít hơn là phụ thuộc vào các thành phần trong quy trình, nên chiến lược thử nghiệm và môi trường thử nghiệm cũng cần phải thích ứng với những thay đổi này.

[4] https://en.wikipedia.org/wiki/In-memory_database



Hình 4. Chiến lược kiểm thử kiến trúc microservice

2.2. Chiến lược kiểm thử trong kiến trúc Microservice

2.2.1. Unit Testing – Kiểm thử đơn vị

Unit test còn gọi là kiểm thử đơn vị - thực hiện kiểm thử phần nhỏ nhất của phần mềm có thể kiểm tra trong ứng dụng để xác định xem nó có hoạt động như mong đợi hay không. Kiểm tra đơn vị thường được viết ở cấp lớp hoặc xung quanh một nhóm nhỏ các lớp có liên quan. Đơn vị được thử nghiệm càng nhỏ, thì càng dễ dàng thể hiện hành vi bằng cách sử dụng thử nghiệm đơn vị. Với kiểm tra đơn vị, bạn sẽ thấy sự khác biệt quan trọng dựa trên việc đơn vị được kiểm tra có bị cô lập với các đối tác của nó hay không.

Unit test được viết cho từng mô-đun bên trong của một microservice. *Test doubles*, *stubs* và *smocking* [5] có thể được kết hợp để thay thế bất kỳ phần phụ thuộc nào. Thông thường, các nhà lập trình viết unit test và với tư cách là một kỹ sư QA (Auality Assurance – Kỹ sư bảo đảm chất lượng phần mềm), họ muốn đảm bảo rằng mã chương trình của họ được đảm bảo mức độ tin cậy dau khi đã kiểm thử.

2.2.2. Component Testing – Kiểm tra thành phần

Với unit test và intergration test, người ta có thể tự tin về chất lượng của các mô-đun bên trong microservice, nhưng vẫnchưa thể đảm bảo rằng microservice đó đáp ứng các yêu cầu nghiệp vụ.

Component testing là loại kiểm thử từ-đầu-đến-cuối(end-to-end:E2E) một microservice bằng cách cô lập nó khỏi các phụ thuộc bên ngoài và các thành phần đối tác hay cộng tác khác.

Trong kiến trúc Microservice, các *components* là các *services themselves – dịch vụ bản thân*. Viết các bài kiểm thử ở mức chi tiết này, các API được kiểm thử theo quan điểm của người sử dụng, cô lập dịch vụ bằng các thay thế các cộng tác bên ngoài bằng test-doubles, stubs hay các API endpoint nội bộ để kiểm tra hay thiết lập cấu hình dịch vụ.

Theo sơ đồ microservice ở trên, các bài kiểm tra *component* tập trung vào một microservice tại một thời điểm. Ví dụ ban đầu ta kiểm thử microservice A một cách riêng biệt, sau đó là B, rồi đến C, v.v. Nếu microservice A có tương tác với microservice B, ta sẽ

[5] https://en.wikipedia.org/wiki/Test_double

thay thế các microservice B bằng một microservice giả lập, rồi tích hợp vào microservice A sau đó tiến hành kiểm thử end-to-end.

Kiểm thử *component* như vậy sẽ có những lợi ích:

Bằng cách giới hạn phạm vi cho một thành phần đơn, ta có thể thực hiện kiểm thử chấp nhận cho các hành vi được đóng gói bởi thành phần đó trong khi duy trì kiểm tra để thực thi nhanh hơn.

Cách ly thành phần với các thành phần khác bằng cách sử dụng test-doubles / test stubs mô phỏng thử nghiệm để tránh bất kỳ hành vi phức tạp nào mà chúng có thể có. Nó cũng giúp cung cấp một môi trường thử nghiệm được kiểm soát cho thành phần, kích hoạt mọi trường hợp lỗi áp dụng theo cách có thể lặp lại.

Nhóm QA dự kiến sẽ xác định phạm vi bao phủ, viết các kịch bản thử nghiệm và tự động hóa tất cả. Họ có thể sử dụng bất kỳ công cụ kiểm tra API phổ biến nào như SoapUI, ReadyAPI, REST-secure và HTTP Client, và có thể xem đây là một thử nghiệm dịch vụ web điển hình. Điều này liên quan đến việc sử dụng rộng rãi các dịch vụ giả để thay thế các phụ thuộc để kiểm tra dịch vụ vi mô một cách riêng biệt. Wiremock, MockLab, MockableIO, ServiceV là một vài ví dụ.

2.2.3. Contract Testing

Contract testing là một phương pháp để đảm bảo rằng hai thành phần riêng biệt (chẳng hạn như hai dịch vụ vi mô) tương thích và có thể giao tiếp với nhau.

Với Unit testing, Component testing, intergrate testing có thể đạt được mức độ bao phủ cao của các mô-đun tạo nên microservice . Nhưng chất lượng của giải pháp hoàn chỉnh không được đảm bảo trừ khi các thử nghiệm của mình bao gồm tất cả các dịch vụ nhỏ hoạt động cùng nhau. Contract testing sẽ tập trung đối với các phụ thuộc bên ngoài và kiểm tra đầu cuối của toàn bộ hệ thống để đảm bảo vấn đề này.

Nếu hai microservices hoặc một microservice và các service phụ thuộc của nó đang cộng tác, thì giữa chúng sẽ hình thành một *contract*. Ví dụ microservice A và B đang cộng tác trong đó microservice A tương tác với B bằng cách gửi hoặc tìm kiếm một số dữ liệu và B phản hồi cho A tất cả các yêu cầu gửi đến nó, thì sẽ có một *contract* giữa A và B. Ở đây, A là được biết đến với tư cách là người tiêu dùng và B là nhà cung cấp. Contract testing diễn ra theo hai bước:

- Đầu tiên, người tiêu dùng sẽ xuất một hợp đồng cho nhà cung cấp của họ. Hợp đồng này trông giống như một lược đồ API điển hình (có thể là một tệp json) với tất cả các yêu cầu, dữ liệu phản hồi và định dạng có thể có. Điều này bao gồm tiêu đề, nội dung, mã trạng thái, uri[6], đường dẫn, động từ, v.v. Ở cuối bước đầu tiên, các hợp đồng được xuất bản trực tiếp đến nhà cung cấp hoặc đến một vị trí trung tâm.

- Bước thứ hai là nơi nhà cung cấp truy cập vào các hợp đồng do (những) người tiêu dùng cung cấp và xác minh chúng dựa trên mã mới nhất của họ. Nếu nó tìm thấy bất kỳ lỗi nào, thì đó có thể là do một số thay đổi vi phạm mà nhà cung cấp đã thực hiện đối với lược đồ API của họ.

Đội QA của cả hai bên (sẽ đưa ra các thử nghiệm contract. Kiểm thử contract cũng sử dụng một dịch vụ giả tạm thời hoạt động như một cầu nối giữa người tiêu dùng và nhà

[6] Uri: Uniform Resource Identifier: một chuỗi chứa các ký tự xác định một tài nguyên vật lý hoặc logic

cung cấp vì quá trình thử nghiệm diễn ra theo hai bước trong các mốc thời gian khác nhau. PACT là một ví dụ của thực hiện kiểm thử contract.[7]

Contract testing không phải là kiểm tra các chức năng, chúng được thực hiện cho toàn bộ các microservices và chúng tập trung nhiều vào các định dạng các loại đầu vào hoặc đầu ra hơn là các giá trị thực tế. Do đó, kích thước bộ kiểm thử sẽ nhỏ hơn các bộ thử nghiệm thành phần.

2.2.4. Integration Testing – Kiểm thử tích hợp

Tuy nhiên, chỉ kiểm thử đơn vị không đảm bảo về hoạt động của hệ thống, cần phải kiểm tra sự liên kết của chúng khi hoạt động cùng nhau để tạo thành một Microservice hoàn chỉnh và đồng thời giải quyết các tương tác mà mô-đun thực hiện với các phụ thuộc hay ràng buộc bên ngoài.

Kiểm tra tích hợp xác minh các đường liên lạc và tương tác giữa các mô-đun và giữa các phần phụ thuộc của chúng.

Kiểm tra tích hợp là xem các mô-đun khi kết hợp lại với nhau thành một hệ thống con chúng có cộng tác và hoạt động như dự định để có thể tích hợp lại thành một thành phần (component) lớn hơn. Ví dụ về kiểm tra tích hợp giữa các mô-đun gắn kết như Kho lưu trữ & Lớp dịch vụ và Lớp dịch vụ với lớp Tài nguyên, v.v. Kiểm tra tích hợp cho các phần phụ thuộc bên ngoài bao gồm kiểm tra giữa các cơ sở dữ liệu, bộ nhớ đệm và các dịch vụ nhỏ khác.

2.2.5. End-to-end Testing (E2E)

End-to-end Testing, Kiểm thử một quy trình hoàn thiện thông qua ứng dụng hoặc microservice. Thường được sử dụng để kiểm thử các đường (path) "vàng" hoặc để xác minh rằng ứng dụng đáp ứng các yêu cầu bên ngoài. Kiểm thử E2E kiểm tra toàn bộ hệ thống từ đầu đến cuối, nó xác minh rằng toàn bộ hệ thống đáp ứng các yêu cầu bên ngoài và cuối cùng đạt được mục tiêu của nó mà không cần bận tâm về kiến trúc bên trong.

Hệ thống khi này đã được triển khai đầy đủ và được coi như một hộp đen và quá trình kiểm tra được thực hiện thông qua GUI hoặc các API, E2E đối mặt nhiều vấn đề nghiệp vụ hơn, Kiểm thử này còn kiểm tra rằng tường lửa, proxy và bộ cân bằng tải có được định cấu hình chính xác không. Trong kiến trúc microservice, đối với một hành vi, có nhiều microservice tương tác để phản hồi lại hành vi đó, thử nghiệm E2E còn giúp phủ thêm các trường hợp kiểm tra các khoảng hở (gaps) giữa chúng bên trong hệ thống.

3. KIỂM THỬ TỰ ĐỘNG TRONG KIẾN TRÚC MICROSERVICES?

Phần mềm xây dựng bằng kiến trúc microservice vẫn có thể áp dụng kiểm thử tự động. Tuy nhiên cần theo nguyên tắc:

- Xử lý từng dịch vụ như một mô-đun phần mềm.
- Xác định các liên kết quan trọng trong kiến trúc và ưu tiên kiểm tra chúng.
- Không nên kiểm thử kiến trúc microservice trong môi trường thử nghiệm có quy mô nhỏ.

[7] https://docs.pact.io/

- Kiểm thử ở nhiều chế độ cài đặt khác nhau.
- Tận dụng tối đa canary testing cho code mới.

(Canary testing là phương thức kiểm thử cho phép xác thực một tính năng mới bằng cách thử nghiệm nó trong môi trường trực tiếp (live) của mình. Điều này có thể hiển thị các vấn đề liên quan đến máy chủ bị bỏ sót trong quá trình thử nghiệm.)

Việc kiểm thử tự động đều có thể áp dụng tất cả các mức trong chiến lược kiểm thử microservice từ unit testing, component testing, contract testing, intergrate testing, end-to-end testing.

4. KẾT LUẬN

Kiến trúc Microservices là một cách tiếp cận phân tán được thiết kế để khắc phục những hạn chế của kiến trúc nguyên khối truyền thống. Kiến trúc Microservices mang lại những lợi ích không thể phủ nhận cho việc phát triển ứng dụng, nó giúp mở rộng quy mô các ứng dụng và tổ chức, tuy nhiên, chúng cũng đi kèm với một số thách thức có thể gây ra thêm sự phức tạp về kiến trúc và gánh nặng hoạt động, đó cũng những thách thức mới đối với QA và thử nghiệm

Với sự hiểu biết về dịch vụ vi mô là gì, cách kiểm tra chúng, loại kiểm tra nào để kết hợp, nơi thực hiện kiểm tra với môi trường, ta có thể đề xuất chiến lược và kế hoạch kiểm tra tốt nhất phù hợp với nhu cầu của tổ chức hay khách hàng của mình và điều này có thể tùy chỉnh và sửa đổi dựa trên nhiều yếu tố như chi phí liên quan đến việc triển khai tất cả các loại thử nghiệm, khả năng tài nguyên sẵn có, thời gian và quy mô thời gian phân phối dự án, chi phí kỹ sư tự động hóa thử nghiệm, cơ sở hạ tầng và phần cứng (ví dụ: dịch vụ đám mây) , ngân sách, và các yếu tố khác.

Kiến trúc Microservices chắc chắn đã ảnh hưởng đến công việc của các chuyên gia QA. Vì microservices thường được triển khai trong các môi trường sử dụng vùng chứa như Docker, điều này buộc tất cả các QA phải tìm hiểu cách họ có thể thiết lập một môi trường thử nghiệm như vậy.

5. TÀI LIỆU THAM KHẢO

[1] Harsh Upreti – 2018 - https://smartbear.com/blog/4-essential-strategies-for-testing-microservices/

[2] XenonStackAugust 24, 2017 - https://www.xenonstack.com/blog/microservices-testing/

[3] Amir Ghahrai, 2017 - https://devqa.io/testing-microservices-beginners-guide/

[4] Martin Fowler - https://martinfowler.com/bliki/TestDouble.html

[5] Nathan Jakubiak - https://www.parasoft.com/how-to-approach-testing-microservices/

Scalable web application

Đỗ Đức Bích Ngân [a*]

[a]Khoa Công nghệ Thông tin, Trường Đại học Ngoại ngữ - Tin học Thành phố Hồ Chí Minh
[*]Tác giả liên hệ: Email: ngan.ddb@huflit.edu.vn | Điện thoại: 077.3144.044

Tóm tắt

Khả năng mở rộng là vấn đề xử lý khi có sự tăng trưởng nhất là đáp ứng được theo thời gian, lượng thông tin lưu trữ ngày càng nhiều, các yêu cầu phần mềm ngày càng mở rộng. Một ứng dụng web thành công cần phải đáp ứng liên tục và hiệu quả sự phát triển cũng như được thiết kế với khả năng mở rộng như vậy. Một ứng dụng web được thiết kế và xây dựng có khả năng mở rộng sẽ có thể xử lý khi có sự gia tăng người dùng và tải mà không làm gián đoạn người dùng cuối đó là Scalable website.

Từ khoá: Scalable; Web scaling; Vertical scacling; Horizontal scaling; Diagonal scaling

1. GIỚI THIỆU

Với tốc độ phát triển của ngành công nghiệp phần mềm, đặc biệt các website, nhất là các website thương mại, mọi người muốn dữ liệu của họ ngay lập tức được xử lý, không có khái niệm chờ đợi, một hình ảnh để tải lên, hoặc một biểu mẫu xử lý, hay thông tin bị tải quá lâu. Nếu một ứng dụng không được thiết kế đúng cách và có thể xử lý số lượng người dùng và khối lượng công việc tăng lên, thì chắc chắn ứng dụng sẽ khó tồn tại lâu dài hoặc sẽ phải tốn tiền nâng cấp hay đập bỏ và xây lại.

Khả năng mở rộng là rất quan trọng đối với tuổi thọ của bất kỳ ứng dụng web nào, không còn như xưa với request ít ỏi hoặc chỉ cần nâng ram, nâng cấp CPU. Hiểu được khả năng mở rộng là gì và cách khai thác các nguyên tắc và tiêu chuẩn từ khâu thiết kế cho đến khi triển khai là điểm mà hầu hết các nhà làm web phải lưu tâm và có kế hoạch thiết kế và xây dựng ngay từ đầu.

Nhìn vào các trang web của các công ty, không phải lúc nào ta cũng nhận ra nó phải chịu được khối lượng công việc ra sao. Người ta thường không nghĩ đến tầm quan trọng của hiệu suất cao và khả năng sử dụng đối với sự thành công của công ty. Do đó, khi những người mới bắt đầu xây dựng các trang web đầu tiên của họ, họ không biết về một số vấn đề mà họ có thể gặp phải sau này:

- Sự gia tăng số lượng người truy cập làm giảm hiệu suất của các ứng dụng web;
- Mở rộng phạm vi sản phẩm ảnh hưởng tiêu cực đến thời gian tải trang; ví dụ cập nhật hàng tồn kho của một cửa hàng thương mại điện tử trở nên có vấn đề;
- Thay đổi cấu trúc mã trở nên nguy hiểm và phức tạp; việc thêm một sản phẩm hoặc dịch vụ mới sẽ mất quá nhiều thời gian và trở nên tốn kém, đồng thời khả năng thực hiện các thử nghiệm A / B (thử nghiệm A/B (hay còn được gọi là split testing) là một quy trình mà trong đó hai phiên bản (A và B) sẽ được cùng so