O'REILLY®

Dịch vụ vi mô AntiPatterns và cạm bẫy

Mark Richards

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

AntiPatterns và cạm bẫy của microservice

Mark Richards

Bắc Kinh Boston Farnham Sebastopol Tokyo O'REILLY®

Các mô hình vi mô và cạm bẫy của Mark Richards

Được in tại Hoa Kỳ.

Được xuất bản bởi O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Biên tập: Brian Foster

Biên tập sản xuất: Melanie Yarbrough

Biên tập viên: Christina Edwards

Người hiệu đính: Amanda Kersey

Nhà thiết kế nội thất: David Futato

Thiết kế bìa: Karen Montgomery

Người minh họa: Rebecca Demarest

Tháng 7 năm 2016: Ấn bản đầu tiên

Lịch sử sửa đổi cho lần xuất bản đầu tiên

06-07-2016: Phát hành lần đầu

978-1-491-96331-9

[LSI]

Mục lục

Lời nói đầu

Vào cuối năm 2006, kiến trúc hướng dịch vụ (SOA) đã trở thành cơn sốt. Các công ty đã nhảy vào cuộc và nắm lấy SOA trước khi hiểu đầy đủ những ưu điểm và nhược điểm của phong cách kiến trúc rất phức tạp này. Những công ty bắt tay vào các dự án SOA thường gặp phải những khó khăn liên tục với mức độ chi tiết của dịch vụ, hiệu suất, di chuyển dữ liệu và đặc biệt là sự thay đổi về mặt tổ chức xảy ra với SOA. Kết quả là, nhiều công ty hoặc từ bỏ nỗ lực SOA của họ hoặc xây dựng các kiến trúc lai không đáp ứng được tất cả những hứa hẹn của SOA.

Ngày nay, chúng tôi sẵn sàng lặp lại trải nghiệm tương tự này với một phong cách kiến trúc tương đối mới được gọi là microservice. Dịch vụ vi mô hiện đang là xu hướng hiện nay trong ngành và giống như SOA vào giữa những năm 2000, tất cả đều là cơn sốt. Do đó, nhiều công ty đang hướng tới phong cách kiến trúc này để tận dụng những lợi ích do microservice mang lại như dễ thử nghiệm, triển khai nhanh chóng và dễ dàng, khả năng mở rộng chi tiết, tính mô đun hóa và tính linh hoạt tổng thể. Tuy nhiên, giống như SOA, các công ty phát triển dịch vụ vi mô đang gặp khó khăn với những thứ như mức độ chi tiết của dịch vụ, di chuyển dữ liệu, thay đổi tổ chức và các thách thức xử lý phân tán.

Giống như bất kỳ công nghệ, phong cách kiến trúc hoặc phương pháp thực hành mới nào, các mô hình phản mẫu và cạm bẫy thường xuất hiện khi bạn tìm hiểu thêm về nó và trải nghiệm nhiều “bài học kinh nghiệm” trong suốt quá trình. Mặc dù phản khuôn mẫu và cạm bẫy có vẻ giống nhau, nhưng giữa chúng có một sự khác biệt tinh tế. Andrew Koenig định nghĩa một phản mẫu là một thứ gì đó có vẻ như là một ý tưởng hay khi bạn bắt đầu, nhưng lại khiến bạn gặp rắc rối, trong khi bạn tôi Neal Ford định nghĩa cạm bẫy là một điều gì đó không bao giờ là một ý tưởng hay, kể cả ngay từ đầu. Đây là

một sự khác biệt quan trọng bởi vì bạn có thể không gặp phải những kết quả tiêu cực từ một phản mẫu cho đến khi bạn bước vào vòng đời phát triển hoặc thậm chí là đi vào sản xuất, trong khi đó, với một cạm bẫy, bạn thường phát hiện ra rằng mình đang đi sai đường tương đối nhanh chóng.

Báo cáo này sẽ giới thiệu một số lỗi sai và cạm bẫy phổ biến hơn thường xuất hiện khi sử dụng vi dịch vụ. Mục tiêu của tôi với báo cáo này là giúp bạn tránh những sai lầm tốn kém bằng cách không chỉ giúp bạn hiểu khi nào một phản mẫu hoặc cạm bẫy đang xảy ra, mà quan trọng hơn là giúp bạn hiểu các kỹ thuật và cách thực hành để tránh những phản mẫu và cạm bẫy này.

Mặc dù tôi không có thời gian trong báo cáo này để trình bày chi tiết về tất cả các phản mẫu và cạm bẫy khác nhau mà bạn có thể gặp phải với các dịch vụ vi mô, nhưng tôi cũng đề cập đến một số dịch vụ phổ biến hơn. Chúng bao gồm các phản mẫu và cạm bẫy liên quan đến mức độ chi tiết của dịch vụ (Chương 5, Hạt cát), di chuyển dữ liệu (Chương 1, AntiPattern di chuyển theo hướng dữ liệu), độ trễ truy cập từ xa (Chương 9, Chúng ta có phải là cạm bẫy), báo cáo (Chương 4) , AntiPattern báo cáo tiếp cận), tạo phiên bản hợp đồng (Chương 8, e Cạm bẫy hợp đồng tĩnh), khả năng đáp ứng dịch vụ (Chương 2, e Timeout AntiPattern) và nhiều thứ khác.

Gần đây tôi đã quay một video cho O'Reilly có tên là Microservices AntiPatterns and Cạm bẫy: Học cách tránh những sai lầm tốn kém chứa đầy đủ các phản mẫu và cạm bẫy mà bạn có thể gặp phải khi sử dụng vi dịch vụ, cũng như cái nhìn sâu hơn về từng vi dịch vụ.
Bao gồm trong video là sổ làm việc tự đánh giá bao gồm các nhiệm vụ phân tích và mục tiêu xoay quanh việc phân tích ứng dụng hiện tại của bạn. Bạn có thể sử dụng sổ làm việc đánh giá này để xác định xem liệu bạn có đang gặp phải bất kỳ sai lầm và cạm bẫy nào được giới thiệu trong video hay không cũng như cách tránh chúng.

Các quy ước được sử dụng trong cuốn sách này

Các quy ước đánh máy sau đây được sử dụng trong cuốn sách này:

Nghiêng

Cho biết các điều khoản, URL, địa chỉ email, tên tệp và tệp mới phần mở rộng.

Độ rộng không đổi

Được sử dụng cho danh sách chương trình, cũng như trong các đoạn văn để chỉ các thành phần chương trình như tên biến hoặc hàm, cơ sở dữ liệu, kiểu dữ liệu, biến môi trường, câu lệnh và từ khóa.

in đậm có chiều rộng không đổi

Hiển thị các lệnh hoặc văn bản khác mà người dùng phải nhập theo nghĩa đen.

Nghiêng có chiều rộng không đổi

Hiển thị văn bản cần được thay thế bằng giá trị do người dùng cung cấp hoặc bằng giá trị được xác định theo ngữ cảnh.

Sách Safari® trực tuyến

Sách Safari trực tuyến là một thư viện kỹ thuật số theo yêu cầu cung cấp nội dung chuyên môn dưới dạng sách và video từ các tác giả hàng đầu thế giới về công nghệ và kinh doanh.

Các chuyên gia công nghệ, nhà phát triển phần mềm, nhà thiết kế web cũng như các chuyên gia kinh doanh và sáng tạo sử dụng Safari Books Online làm tài nguyên chính cho nghiên cứu, giải quyết vấn đề, học tập và đào tạo cấp chứng chỉ.

Safari Books Online cung cấp nhiều gói và mức giá khác nhau cho doanh nghiệp , chính phủ, giáo dục, và cá nhân.

Các thành viên có quyền truy cập vào hàng nghìn cuốn sách, video đào tạo và bản thảo chuẩn bị xuất bản trong một cơ sở dữ liệu có thể tìm kiếm đầy đủ từ các nhà xuất bản như O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, và hàng trăm trang khác. Để biết thêm thông tin về Safari Books Online, vui lòng truy cập trực tuyến của chúng tôi.

Cách liên hệ với chúng tôi

Vui lòng giải quyết các nhận xét và câu hỏi liên quan đến cuốn sách này tới nhà xuất bản:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (ở Hoa Kỳ hoặc Canada) 707-829-0515 (quốc tế hoặc địa phương) 707-829-0104 (fax)

Để nhận xét hoặc đặt câu hỏi kỹ thuật về cuốn sách này, hãy gửi email đến bookquestions@oreilly.com.

Để biết thêm thông tin về sách, khóa học, hội nghị và tin tức của chúng tôi, hãy xem trang web của chúng tôi tại http://www.oreilly.com.

Tìm chúng tôi trên Facebook: http://facebook.com/oreilly

Theo dõi chúng tôi trên Twitter: http://twitter.com/oreillymedia

Xem chúng tôi trên YouTube: http://www.youtube.com/oreillymedia

Lời cảm ơn

Tôi muốn cảm ơn một số người đã giúp làm cho báo cáo này thành công. Đầu tiên, tôi xin cảm ơn Matt Stine về phần đánh giá kỹ thuật của báo cáo. Kiến thức kỹ thuật và kiến thức chuyên môn của ông về các dịch vụ vi mô không chỉ giúp xác thực các khía cạnh kỹ thuật khác nhau của báo cáo mà còn nâng cao một số chương nhất định bằng thông tin chi tiết và tài liệu bổ sung. Tiếp theo, tôi muốn cảm ơn người bạn và đồng phạm Neal Ford của tôi vì đã giúp tôi hiểu sự khác biệt giữa cạm bẫy và phản mẫu, đồng thời giúp tôi phân loại chính xác từng phản mẫu và cạm bẫy mà tôi đã viết trong báo cáo này. Tôi cũng muốn cảm ơn đội ngũ biên tập tại O'Reilly vì sự giúp đỡ của họ trong việc làm cho quá trình soạn thảo trở nên dễ dàng và suôn sẻ nhất có thể. Cuối cùng, tôi muốn cảm ơn vợ và các con tôi vì đã cùng tôi thực hiện một dự án khác (dù nhỏ) khiến tôi phải rời xa điều tôi thích làm nhất—ở bên gia đình.

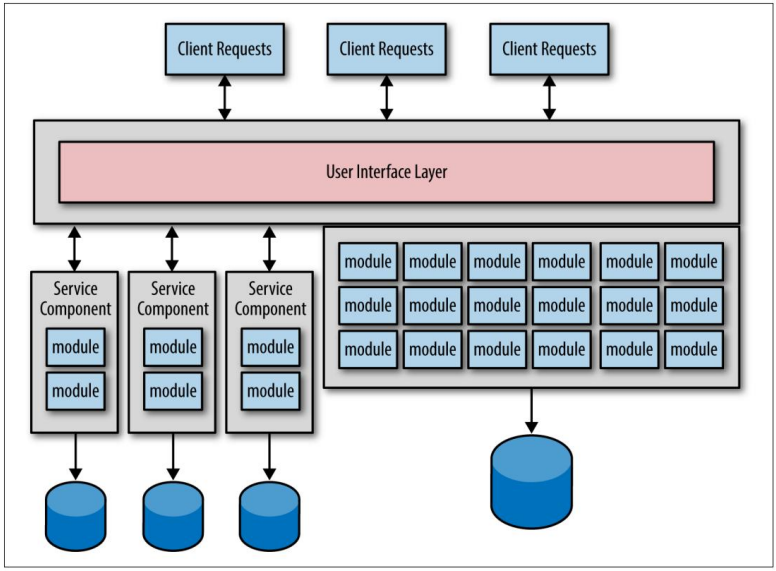
CHƯƠNG 1

AntiPattern di chuyển theo hướng dữ liệu

Microservices là việc tạo ra nhiều dịch vụ nhỏ, có mục đích đơn lẻ, phân tán, trong đó mỗi dịch vụ sở hữu dữ liệu riêng. Sự kết hợp dữ liệu và dịch vụ này hỗ trợ khái niệm về bối cảnh bị giới hạn và kiến trúc không chia sẻ gì, trong đó mỗi dịch vụ và dữ liệu tương ứng của nó được ngăn cách và hoàn toàn độc lập với tất cả các dịch vụ khác, chỉ hiển thị một giao diện được xác định rõ ràng (hợp đồng). Bối cảnh giới hạn này cho phép phát triển, thử nghiệm và triển khai nhanh chóng và dễ dàng với mức độ phụ thuộc tối thiểu.

Phản mẫu di chuyển theo hướng dữ liệu xảy ra chủ yếu khi bạn di chuyển từ một ứng dụng nguyên khối sang kiến trúc vi dịch vụ. Lý do đây là một phản mẫu là vì lúc đầu, có vẻ như là một ý tưởng hay để di chuyển cả chức năng dịch vụ và dữ liệu tương ứng với nhau khi tạo vi dịch vụ, nhưng như bạn sẽ tìm hiểu trong chương này, điều này sẽ dẫn bạn đến một con đường xấu là có thể dẫn đến rủi ro cao, chi phí vượt mức và nỗ lực di chuyển bổ sung.

Có hai mục tiêu chính trong mọi nỗ lực chuyển đổi vi dịch vụ. Mục tiêu đầu tiên là chia chức năng của ứng dụng nguyên khối thành các dịch vụ nhỏ, phục vụ một mục đích. Mục tiêu thứ hai là di chuyển dữ liệu nguyên khối vào cơ sở dữ liệu nhỏ (hoặc các lược đồ riêng biệt) do mỗi dịch vụ sở hữu. Hình 1-1 cho thấy một quá trình di chuyển thông thường sẽ trông như thế nào khi cả mã dịch vụ và dữ liệu tương ứng được di chuyển cùng một lúc.

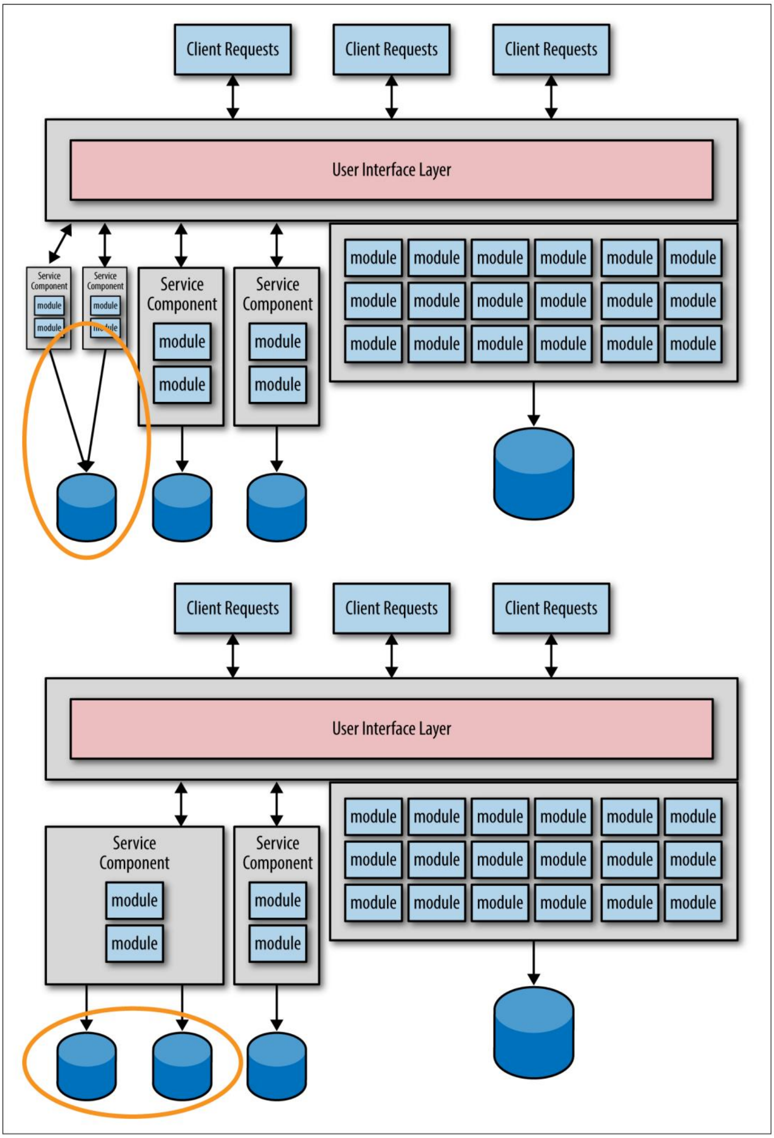


Hình 1-1. Di chuyển dịch vụ và dữ liệu

Lưu ý rằng có ba dịch vụ được tạo từ ứng dụng nguyên khối cùng với ba cơ sở dữ liệu riêng biệt. Đây là quá trình di chuyển tự nhiên vì bạn đang tạo bối cảnh giới hạn quan trọng giữa mỗi dịch vụ và dữ liệu tương ứng của dịch vụ đó. Tuy nhiên, các vấn đề bắt đầu nảy sinh với cách thực hành phổ biến này, do đó dẫn bạn đến mô hình phản mẫu di chuyển theo hướng dữ liệu.

Quá nhiều lần di chuyển dữ liệu

Vấn đề chính với loại đường dẫn di chuyển này là bạn sẽ hiếm khi nhận được thông tin chi tiết của từng dịch vụ ngay lần đầu tiên. Biết rằng bắt đầu với một dịch vụ chi tiết hơn luôn là một ý tưởng hay và chia nhỏ nó ra nếu cần khi bạn tìm hiểu thêm về dịch vụ đó, bạn có thể thường xuyên điều chỉnh mức độ chi tiết của các dịch vụ của mình. Hãy xem xét việc di chuyển được minh họa trong Hình 1-1, tập trung vào dịch vụ ngoài cùng bên trái. Giả sử sau khi tìm hiểu thêm về dịch vụ, bạn phát hiện ra rằng nó quá thô sơ và cần được chia thành hai dịch vụ nhỏ hơn. Ngoài ra, bạn có thể thấy rằng hai dịch vụ còn lại quá chi tiết và cần được hợp nhất. Trong cả hai trường hợp, bạn phải đối mặt với hai nỗ lực di chuyển—một cho chức năng dịch vụ và một cho cơ sở dữ liệu. Kịch bản này được minh họa trong Hình 1-2.



Hình 1-2. Di chuyển dữ liệu bổ sung sau khi điều chỉnh mức độ chi tiết của dịch vụ

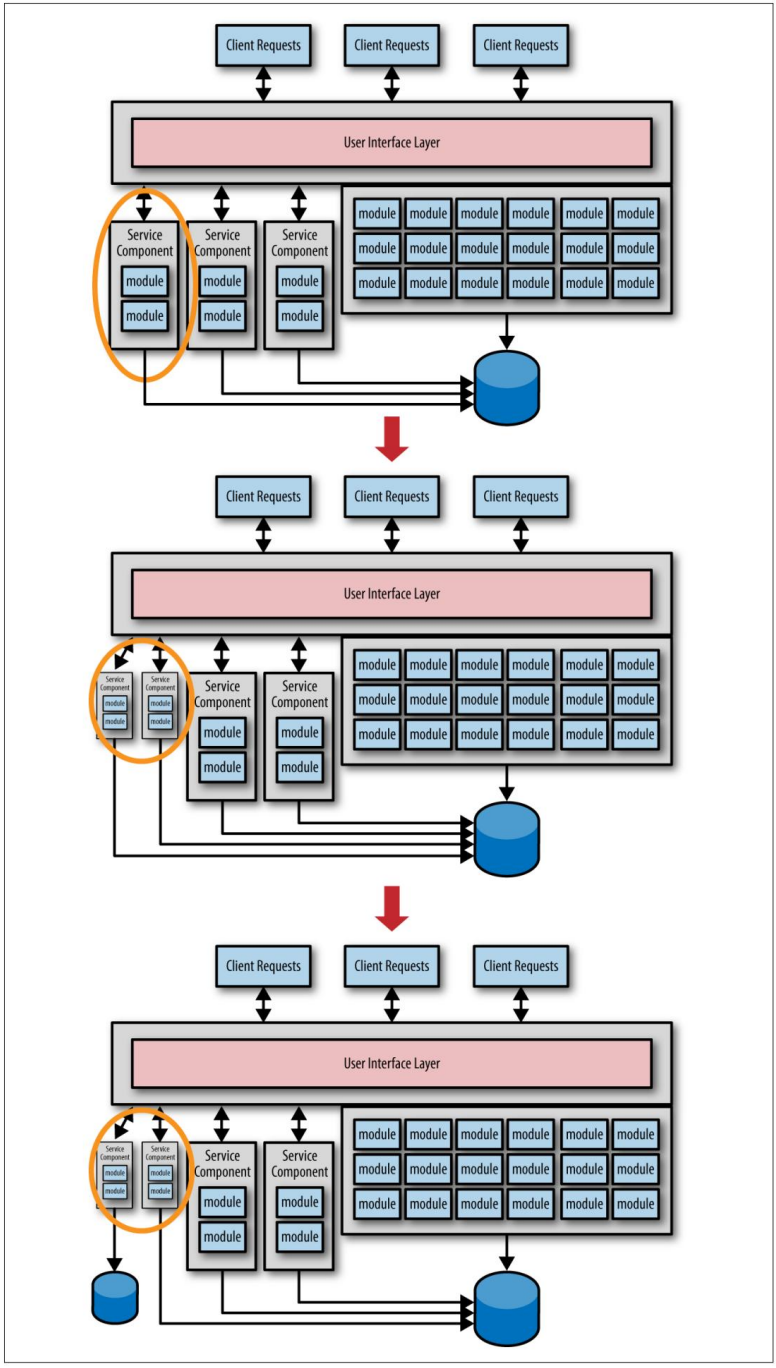
Người bạn tốt của tôi và cũng là tác giả O'Reilly Alan Beaulieu (Học SQL) đã từng nói với tôi "Dữ liệu là tài sản của công ty, không phải tài sản ứng dụng". Dựa trên tuyên bố của Alan, bạn có thể hiểu rõ hơn về rủi ro liên quan và những lo ngại nảy sinh khi dữ liệu được di chuyển liên tục. Quá trình di chuyển dữ liệu rất phức tạp và dễ xảy ra lỗi—còn hơn thế nữa

hơn là di chuyển mã nguồn. Tốt nhất là bạn chỉ muốn di chuyển dữ liệu cho mỗi dịch vụ một lần. Hiểu những rủi ro liên quan đến việc di chuyển dữ liệu và tầm quan trọng của "dữ liệu hơn chức năng" là bước đầu tiên để tránh mô hình phản mẫu này.

Chức năng đầu tiên, dữ liệu cuối cùng

Kỹ thuật tránh chính cho phản mẫu này là di chuyển chức năng của dịch vụ trước và lo lắng về bối cảnh bị giới hạn giữa dịch vụ và dữ liệu sau đó. Sau khi tìm hiểu thêm về dịch vụ, bạn có thể sẽ thấy cần phải điều chỉnh mức độ chi tiết thông qua hợp nhất dịch vụ hoặc tách dịch vụ. Sau khi bạn hài lòng rằng bạn có mức độ chi tiết chính xác, hãy di chuyển dữ liệu, từ đó tạo ra bối cảnh giới hạn rất cần thiết giữa dịch vụ và dữ liệu.

Kỹ thuật này được minh họa trong Hình 1-3. Lưu ý cách cả ba dịch vụ đã được di chuyển nhưng vẫn đang kết nối với dữ liệu nguyên khối. Điều này hoàn toàn phù hợp đối với giải pháp tạm thời vì giờ đây bạn có thể tìm hiểu thêm về cách sử dụng dịch vụ và loại yêu cầu nào sẽ được mỗi dịch vụ xử lý.



Hình 1-3. Di chuyển chức năng dịch vụ trước, sau đó là phần dữ liệu sau

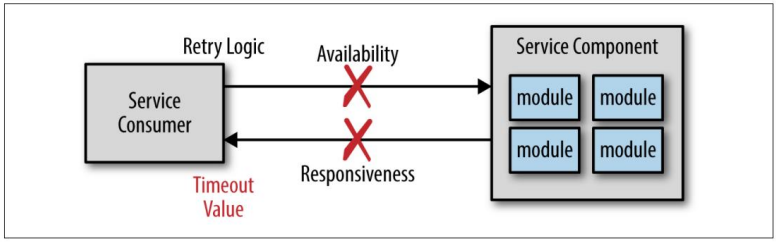
Trong Hình 1-3, hãy chú ý thấy dịch vụ này quá thô sơ và do đó được chia thành hai dịch vụ nhỏ hơn. Bây giờ, mức độ chi tiết đã chính xác, dữ liệu có thể được di chuyển để tạo bối cảnh giới hạn giữa dịch vụ và dữ liệu tương ứng.

Kỹ thuật này tránh việc di chuyển dữ liệu tốn kém và lặp đi lặp lại, đồng thời giúp điều chỉnh mức độ chi tiết của dịch vụ khi cần dễ dàng hơn. Mặc dù không thể nói phải đợi bao lâu trước khi di chuyển dữ liệu, nhưng điều quan trọng là phải hiểu hậu quả của kỹ thuật né tránh này—một bối cảnh bị giới hạn kém. Khoảng thời gian từ khi dịch vụ được tạo đến khi dữ liệu cuối cùng được di chuyển sẽ tạo ra sự ghép nối dữ liệu giữa các dịch vụ. Điều này có nghĩa là khi lược đồ cơ sở dữ liệu được thay đổi, tất cả các dịch vụ sử dụng lược đồ đó phải được điều phối từ quan điểm phát hành và kiểm soát thay đổi, điều mà bạn muốn tránh với kiến trúc vi dịch vụ. Tuy nhiên, sự đánh đổi này rất xứng đáng với việc giảm thiểu rủi ro liên quan đến việc tránh di chuyển cơ sở dữ liệu tốn kém nhiều lần.

CHƯƠNG 2

AntiPattern hết thời gian chờ

Microservices là một kiến trúc phân tán, nghĩa là tất cả các thành phần (tức là dịch vụ) được triển khai dưới dạng các ứng dụng riêng biệt và được truy cập từ xa thông qua một số loại giao thức truy cập từ xa. Một trong những thách thức của bất kỳ kiến trúc phân tán nào là quản lý tính khả dụng và khả năng phản hồi của quy trình từ xa. Mặc dù tính sẵn có của dịch vụ và khả năng đáp ứng của dịch vụ đều liên quan đến giao tiếp dịch vụ nhưng chúng là hai thứ rất khác nhau. Tính khả dụng của dịch vụ là khả năng người tiêu dùng dịch vụ kết nối với dịch vụ và có thể gửi yêu cầu cho dịch vụ đó, như trong Hình 2-1. Mặt khác, khả năng đáp ứng của dịch vụ là thời gian cần thiết để dịch vụ phản hồi một yêu cầu nhất định sau khi bạn đã liên lạc với nó.



Hình 2-1. Tính sẵn có của dịch vụ so với khả năng đáp ứng

Nếu người tiêu dùng dịch vụ không thể kết nối hoặc nói chuyện với dịch vụ (tức là tính khả dụng), người tiêu dùng dịch vụ thường được thông báo ngay lập tức trong vòng một phần nghìn giây, như Hình 2-1 cho thấy. Người sử dụng dịch vụ có thể chọn chuyển lỗi này sang máy khách hoặc thử lại kết nối nhiều lần trước khi từ bỏ và đưa ra một số loại lỗi kết nối. Tuy nhiên, giả sử đã đạt được dịch vụ và

yêu cầu đã được thực hiện, điều gì sẽ xảy ra nếu dịch vụ không phản hồi? Trong trường hợp này, người sử dụng dịch vụ có thể chọn chờ vô thời hạn hoặc tận dụng một số loại giá trị hết thời gian chờ.

Sử dụng giá trị thời gian chờ để đáp ứng dịch vụ có vẻ là một ý tưởng hay nhưng có thể dẫn bạn vào con đường xấu được gọi là mô hình chống thời gian chờ.

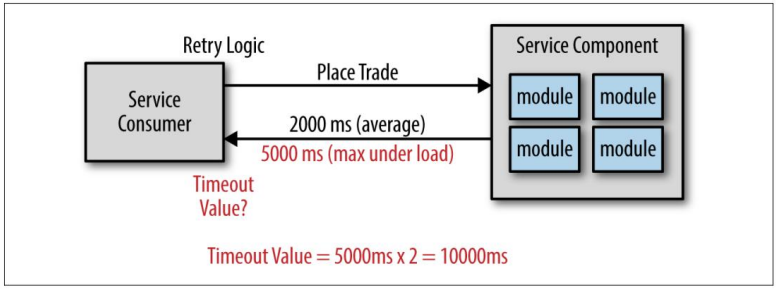
Sử dụng giá trị thời gian chờ

Bạn có thể hơi bối rối vào thời điểm này. Rốt cuộc, việc đặt giá trị thời gian chờ có phải là điều tốt không? Có thể, nhưng trong hầu hết các trường hợp, nó có thể dẫn bạn vào con đường xấu. Hãy xem xét ví dụ khi bạn đưa ra yêu cầu dịch vụ để mua 1000 cổ phiếu Apple (AAPL). Điều cuối cùng bạn muốn làm với tư cách là người tiêu dùng dịch vụ là hết thời gian yêu cầu ngay khi dịch vụ đã thực hiện giao dịch thành công và chuẩn bị cung cấp cho bạn số xác nhận. Bạn có thể thử gửi lại giao dịch nhưng bạn phải tăng thêm độ phức tạp đáng kể vào dịch vụ của mình để xác định xem đây là giao dịch mới hay giao dịch trùng lặp. Hơn nữa, vì bạn không có số xác nhận từ giao dịch đầu tiên nên rất khó để biết liệu giao dịch đó có thực sự thành công hay không.

không.

Vì vậy, vì bạn không muốn hết thời gian yêu cầu quá sớm, nên giá trị thời gian chờ là bao nhiêu? Có một số kỹ thuật để giải quyết vấn đề này. Đầu tiên là tính toán thời gian chờ của cơ sở dữ liệu trong dịch vụ và sử dụng nó làm cơ sở để xác định thời gian chờ của dịch vụ. Giải pháp thứ hai, cho đến nay là kỹ thuật phổ biến nhất, là tính toán thời gian tối đa khi tải và nhân đôi nó, từ đó cung cấp cho bạn bộ đệm bổ sung trong trường hợp đôi khi mất nhiều thời gian hơn.

Hình 2-2 minh họa kỹ thuật này. Lưu ý rằng trung bình dịch vụ sẽ phản hồi trong vòng 2 giây để thực hiện giao dịch. Tuy nhiên, dưới tải thời gian tối đa quan sát được là 5 giây. Do đó, bằng cách sử dụng kỹ thuật nhân đôi, giá trị thời gian chờ đối với người sử dụng dịch vụ sẽ là 10 giây. Một lần nữa, mục đích của kỹ thuật này là tránh việc hết thời gian yêu cầu trong khi trên thực tế nó đã thành công và đang trong quá trình gửi lại cho bạn số xác nhận.



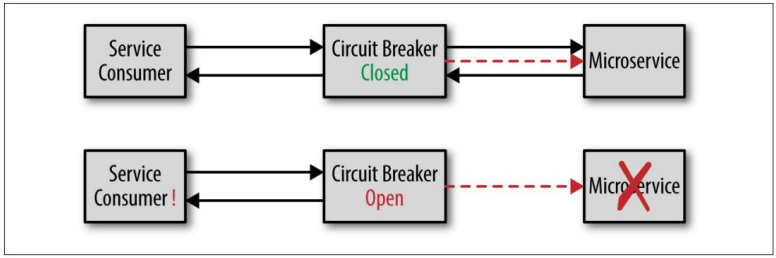
Hình 2-2. Tính giá trị thời gian chờ

Bây giờ chúng ta đã rõ tại sao cách tiếp cận này là một phản mẫu. Mặc dù đây có vẻ là một giải pháp hoàn toàn hợp lý cho vấn đề hết thời gian chờ, nhưng nó khiến mọi yêu cầu từ người tiêu dùng dịch vụ phải đợi 10 giây chỉ để biết dịch vụ không phản hồi. Mười giây là một thời gian dài để chờ đợi một lỗi. Trong hầu hết các trường hợp, người dùng sẽ không đợi quá 2 đến 3 giây trước khi nhấn lại nút gửi hoặc bỏ cuộc và đóng màn hình. Phải có cách tốt hơn để giải quyết vấn đề phản hồi của máy chủ.

Sử dụng mẫu ngắt mạch

Thay vì dựa vào các giá trị thời gian chờ cho các cuộc gọi dịch vụ từ xa, cách tiếp cận tốt hơn là sử dụng một thứ gọi là mẫu ngắt mạch. Mẫu phần mềm này hoạt động giống như một cầu dao trong nhà bạn. Khi nó đóng lại, dòng điện chạy qua nó, nhưng khi nó mở ra thì không có dòng điện nào có thể truyền qua cho đến khi cầu dao đóng lại. Tương tự, nếu bộ ngắt mạch phần mềm phát hiện một dịch vụ không phản hồi, nó sẽ mở ra, từ chối các yêu cầu đến dịch vụ đó. Khi dịch vụ trở nên phản hồi, bộ ngắt sẽ đóng lại, cho phép các yêu cầu được thực hiện.

Hình 2-3 minh họa cách hoạt động của mô hình ngắt mạch. Bộ ngắt mạch liên tục giám sát dịch vụ từ xa, đảm bảo rằng nó vẫn hoạt động và phản hồi (sẽ nói thêm về phần đó sau). Trong khi dịch vụ vẫn đáp ứng, bộ ngắt sẽ bị đóng, cho phép các yêu cầu được thực hiện. Nếu dịch vụ từ xa đột nhiên không phản hồi, bộ ngắt mạch sẽ mở ra, do đó ngăn các yêu cầu được thực hiện cho đến khi dịch vụ phản hồi lại. Tuy nhiên, không giống như cầu dao trong nhà bạn, cầu dao phần mềm có thể tiếp tục giám sát dịch vụ và tự đóng khi dịch vụ từ xa hoạt động trở lại.



Hình 2-3. Mô hình ngắt mạch

Tùy thuộc vào cách thực hiện, người sử dụng dịch vụ sẽ luôn kiểm tra cầu dao trước để xem nó đang mở hay đóng. Điều này cũng có thể được thực hiện thông qua mẫu chặn để người tiêu dùng dịch vụ không cần biết bộ ngắt mạch nằm trong đường dẫn yêu cầu. Trong cả hai trường hợp, lợi thế đáng kể của kiểu ngắt mạch so với các giá trị thời gian chờ là người tiêu dùng dịch vụ biết ngay rằng dịch vụ đã không phản hồi thay vì phải đợi giá trị thời gian chờ. Trong ví dụ trước, nếu sử dụng bộ ngắt mạch thay vì giá trị thời gian chờ, người tiêu dùng dịch vụ sẽ biết trong vòng một phần nghìn giây rằng dịch vụ đặt vị trí giao dịch không phản hồi thay vì phải đợi 10 giây (10.000 mili giây) để nhận được thông tin tương tự. thông tin.

Bộ ngắt mạch có thể giám sát dịch vụ từ xa theo nhiều cách. Cách đơn giản nhất là thực hiện kiểm tra nhịp tim đơn giản trên dịch vụ từ xa (ví dụ: ping). Mặc dù việc này tương đối dễ dàng và không tốn kém, nhưng tất cả những gì nó làm là thông báo cho bộ ngắt mạch rằng dịch vụ từ xa vẫn hoạt động chứ không nói gì về khả năng phản hồi của yêu cầu dịch vụ thực tế. Để có được thông tin tốt hơn về khả năng đáp ứng yêu cầu, bạn có thể sử dụng các giao dịch tổng hợp. Giao dịch tổng hợp là một kỹ thuật giám sát khác mà bộ ngắt mạch có thể sử dụng khi giao dịch giả mạo được gửi định kỳ đến dịch vụ (ví dụ: 10 giây một lần). Giao dịch giả mạo thực hiện tất cả chức năng được yêu cầu trong dịch vụ đó, cho phép bộ ngắt mạch có được thước đo phản hồi chính xác. Giao dịch tổng hợp có thể rất phức tạp và khó thực hiện vì tất cả các phần của ứng dụng hoặc hệ thống đều cần biết về giao dịch tổng hợp. Loại giám sát thứ ba là giám sát người dùng theo thời gian thực, trong đó các giao dịch sản xuất thực tế được giám sát về khả năng phản hồi. Khi đạt đến ngưỡng, bộ ngắt sẽ chuyển sang trạng thái được gọi là trạng thái nửa mở, trong đó chỉ một số lượng giao dịch nhất định được thực hiện (ví dụ 1 trên 10).

Sau khi khả năng đáp ứng của dịch vụ trở lại bình thường, bộ ngắt sẽ được đóng lại, cho phép tất cả các giao dịch được thực hiện.

Có một số triển khai nguồn mở của mẫu ngắt mạch, bao gồm Hystrix từ Netflix và rất nhiều triển khai GitHub. Khung Akka bao gồm việc triển khai bộ ngắt mạch như một phần của khung được triển khai thông qua lớp Akka CircuitBreaker .

Bạn có thể biết thêm thông tin về mẫu cầu dao thông qua các tài nguyên sau:

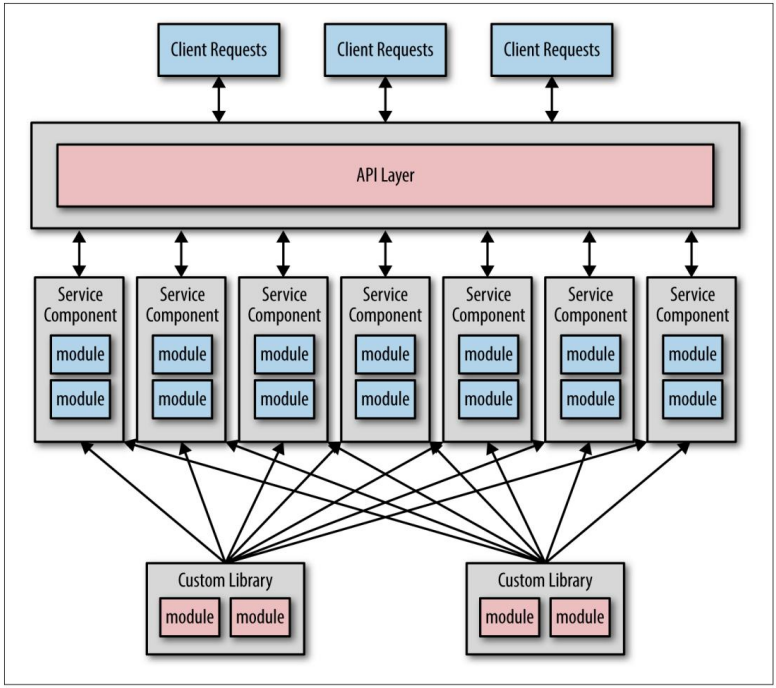
- Cuốn sách xuất sắc của Michael Nygard Release It!
- Bài đăng trên blog về cầu dao của Martin Fowler
- Thư viện MSDN của Microsoft

CHƯƠNG 3

Bài “Tôi được dạy để chia sẻ” Chống mẫu

Microservices được biết đến như một kiến trúc “không chia sẻ gì”. Về mặt thực tế, tôi thích nghĩ về nó như một kiến trúc “chia sẻ ít nhất có thể” bởi vì sẽ luôn có một số cấp độ mã được chia sẻ giữa các dịch vụ vi mô. Ví dụ: thay vì có một dịch vụ bảo mật chịu trách nhiệm xác thực và ủy quyền, bạn có thể có mã nguồn và chức năng bảo mật được gói trong một tệp JAR có tên security.jar mà tất cả các dịch vụ đều sử dụng. Giả sử bảo mật được xử lý ở cấp dịch vụ, đây thường là một phương pháp hay vì nó loại bỏ nhu cầu thực hiện cuộc gọi từ xa tới dịch vụ bảo mật cho mọi yêu cầu, từ đó tăng cả hiệu suất và độ tin cậy.

Tuy nhiên, nếu đi quá xa, bạn sẽ gặp phải cơn ác mộng về sự phụ thuộc như được minh họa trong Hình 3-1, trong đó mọi dịch vụ đều phụ thuộc vào nhiều thư viện chia sẻ tùy chỉnh.

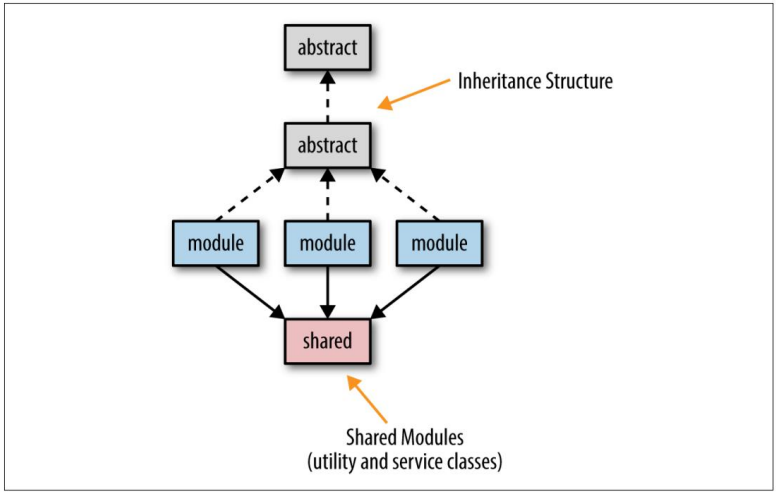


Hình 3-1. Chia sẻ nhiều thư viện tùy chỉnh

Mức độ chia sẻ này không chỉ phá vỡ bối cảnh giới hạn của từng dịch vụ mà còn gây ra một số vấn đề, bao gồm độ tin cậy tổng thể, kiểm soát thay đổi, khả năng kiểm tra và triển khai.

Quá nhiều sự phụ thuộc

Nếu bạn xem xét cách hầu hết các ứng dụng phần mềm hướng đối tượng được phát triển, không khó để nhận ra các vấn đề với việc chia sẻ, đặc biệt khi chuyển từ kiến trúc phân lớp nguyên khối sang kiến trúc vi dịch vụ. Một trong những điều cần phấn đấu trong hầu hết các ứng dụng nguyên khối là tái sử dụng và chia sẻ mã. Hình 3-2 minh họa hai tạo phẩm chính (các lớp trừu tượng và các tiện ích dùng chung) cuối cùng được chia sẻ trong hầu hết các kiến trúc phân lớp nguyên khối.



Hình 3-2. Chia sẻ cấu trúc kế thừa và các lớp tiện ích

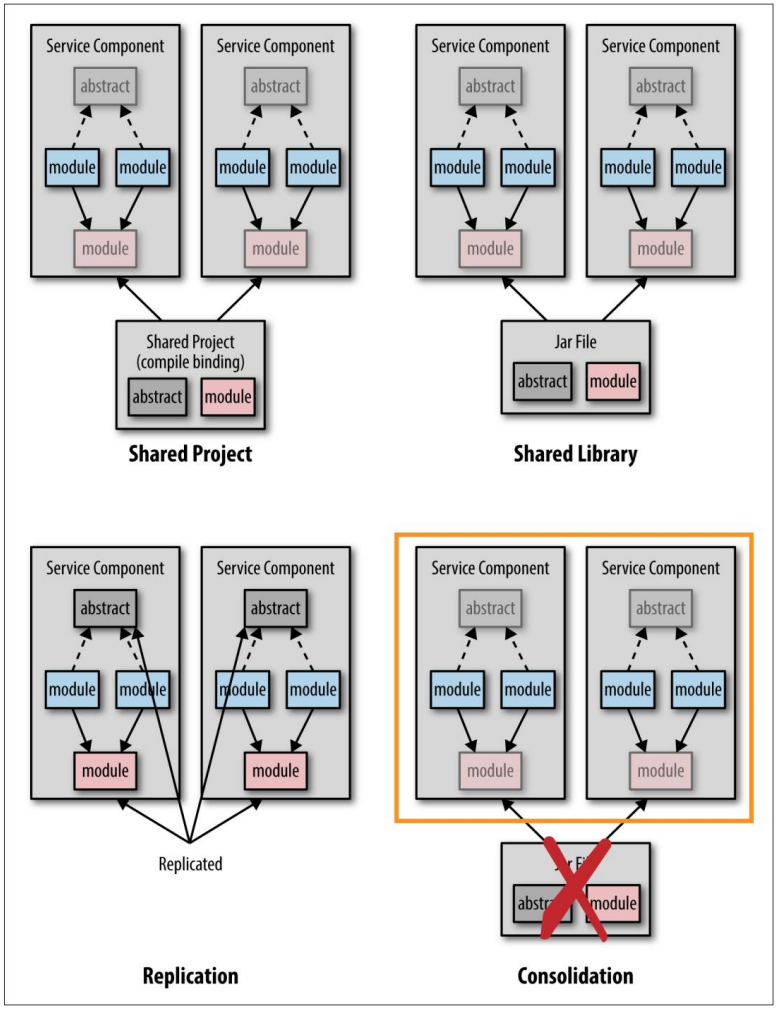
Mặc dù việc tạo các lớp và giao diện trừu tượng là một phương pháp phổ biến với hầu hết các ngôn ngữ lập trình hướng đối tượng, nhưng chúng lại gây cản trở khi cố gắng di chuyển các mô-đun sang kiến trúc vi dịch vụ. Điều tương tự cũng xảy ra với các lớp và tiện ích được chia sẻ tùy chỉnh như các tiện ích chuỗi hoặc ngày tháng chung và các tiện ích tính toán. Bạn sẽ làm gì với mã cần được chia sẻ bởi hàng trăm dịch vụ?

Một trong những mục tiêu chính của phong cách kiến trúc microservices là chia sẻ càng ít càng tốt. Điều này giúp duy trì bối cảnh giới hạn của từng dịch vụ, mang đến cho bạn khả năng thực hiện thử nghiệm và triển khai nhanh chóng. Với microservice, tất cả đều tập trung vào việc thay đổi quyền kiểm soát và các phần phụ thuộc. Bạn càng có nhiều sự phụ thuộc giữa các dịch vụ thì càng khó tách biệt các thay đổi của dịch vụ, gây khó khăn cho việc kiểm tra và triển khai riêng lẻ các dịch vụ riêng lẻ. Việc chia sẻ quá nhiều sẽ tạo ra quá nhiều sự phụ thuộc giữa các dịch vụ, dẫn đến hệ thống dễ vỡ, rất khó kiểm tra và triển khai.

Kỹ thuật chia sẻ mã

Thật dễ dàng để nói rằng cách tốt nhất để tránh phản mẫu này đơn giản là không chia sẻ mã giữa các dịch vụ. Tuy nhiên, như tôi đã nói ở đầu chương này, về mặt thực tế sẽ luôn có một số mã cần được chia sẻ. Mã chia sẻ đó nên đi đâu?

Hình 3-3 minh họa bốn kỹ thuật cơ bản để giải quyết vấn đề chia sẻ mã: dự án dùng chung, thư viện dùng chung, sao chép và hợp nhất dịch vụ.



Hình 3-3. Kỹ thuật chia sẻ mô-đun

Việc sử dụng một dự án dùng chung sẽ tạo ra sự ràng buộc về thời gian biên dịch giữa mã nguồn chung nằm trong một dự án dùng chung và từng dự án dịch vụ. Mặc dù điều này giúp dễ dàng thay đổi và phát triển phần mềm nhưng đây là kỹ thuật chia sẻ mà tôi ít yêu thích nhất vì nó gây ra các vấn đề tiềm ẩn và bất ngờ trong thời gian chạy, khiến ứng dụng kém mạnh mẽ hơn. Vấn đề chính với kỹ thuật dự án chia sẻ là

giao tiếp và kiểm soát—rất khó để biết những mô-đun chia sẻ nào đã thay đổi và tại sao, đồng thời cũng khó kiểm soát liệu bạn có muốn thay đổi cụ thể đó hay không. Hãy tưởng tượng bạn sẵn sàng phát hành microservice của mình chỉ để phát hiện ra ai đó đã thực hiện một thay đổi đột phá đối với mô-đun dùng chung, yêu cầu bạn thay đổi và kiểm tra lại mã của mình trước khi triển khai.

Cách tiếp cận tốt hơn nếu bạn phải chia sẻ mã là sử dụng thư viện dùng chung (ví dụ: tập hợp .NET hoặc tệp JAR). Cách tiếp cận này khiến việc phát triển trở nên khó khăn hơn vì đối với mỗi thay đổi được thực hiện đối với một mô-đun trong thư viện dùng chung, trước tiên nhà phát triển phải tạo thư viện, sau đó khởi động lại dịch vụ rồi kiểm tra lại. Tuy nhiên, ưu điểm của kỹ thuật thư viện dùng chung là các thư viện có thể được phiên bản hóa, cung cấp khả năng kiểm soát tốt hơn đối với hành vi triển khai và thời gian chạy của một dịch vụ. Nếu một thay đổi được thực hiện đối với thư viện dùng chung và được lập phiên bản, chủ sở hữu dịch vụ có thể đưa ra quyết định về thời điểm kết hợp thay đổi đó.

Kỹ thuật thứ ba phổ biến trong kiến trúc microservice là vi phạm nguyên tắc không lặp lại (DRY) và sao chép mô-đun dùng chung trên tất cả các dịch vụ cần chức năng cụ thể đó. Mặc dù kỹ thuật sao chép có vẻ rủi ro nhưng nó tránh được việc chia sẻ phụ thuộc và duy trì bối cảnh giới hạn của dịch vụ.

Các vấn đề phát sinh với kỹ thuật này khi mô-đun sao chép cần được thay đổi, đặc biệt là khi có lỗi. Trong trường hợp này tất cả các dịch vụ cần phải thay đổi. Vì vậy, kỹ thuật này chỉ thực sự hữu ích đối với những module chia sẻ rất ổn định, có ít hoặc không có sự thay đổi.

Kỹ thuật thứ tư đôi khi có thể thực hiện được là sử dụng hợp nhất dịch vụ. Giả sử hai hoặc ba dịch vụ đều chia sẻ một số mã chung và các mô-đun chung đó thường xuyên thay đổi. Vì tất cả các dịch vụ đều phải được kiểm tra và triển khai với sự thay đổi mô-đun chung, nên bạn cũng có thể hợp nhất chức năng thành một dịch vụ duy nhất, từ đó loại bỏ thư viện phụ thuộc.

Một lời khuyên liên quan đến thư viện dùng chung—tránh kết hợp tất cả mã dùng chung của bạn vào một thư viện dùng chung duy nhất như common.jar.
Việc sử dụng một thư viện chung khiến bạn khó biết liệu bạn có cần kết hợp mã được chia sẻ hay không và khi nào. Một kỹ thuật tốt hơn là tách các thư viện dùng chung của bạn thành những thư viện có ngữ cảnh. Ví dụ: tạo các thư viện dựa trên ngữ cảnh như security.jar, Persistence.jar, dateutils.jar, v.v. Điều này tách biệt mã không thay đổi thường xuyên với mã thay đổi thường xuyên, giúp dễ dàng hơn

xác định xem có nên kết hợp thay đổi ngay lập tức hay không và bối cảnh của thay đổi là gì.

CHƯƠNG 4

AntiPattern báo cáo tiếp cận

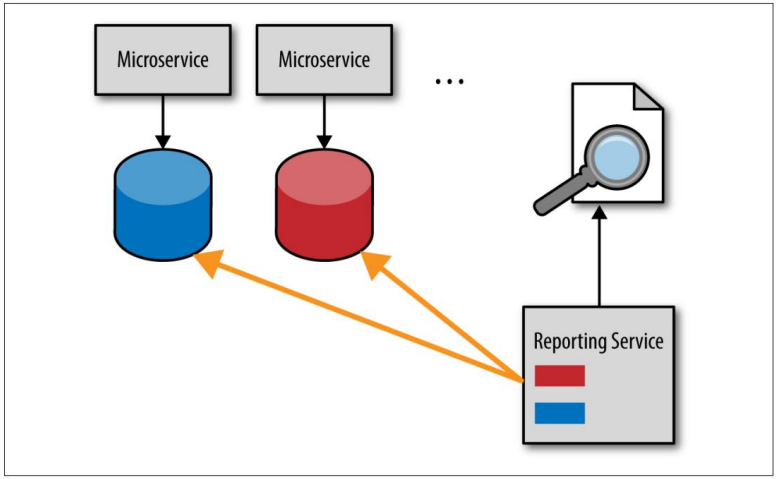
Với kiểu kiến trúc microservices, các dịch vụ và dữ liệu tương ứng được chứa trong một bối cảnh giới hạn duy nhất, nghĩa là dữ liệu thường được di chuyển sang các cơ sở dữ liệu (hoặc lược đồ) riêng biệt. Mặc dù điều này hoạt động tốt đối với các dịch vụ nhưng nó lại gây ảnh hưởng đến việc báo cáo trong kiến trúc vi dịch vụ.

Có bốn kỹ thuật chính để xử lý báo cáo trong kiến trúc dịch vụ vi mô: mô hình kéo cơ sở dữ liệu, mô hình kéo HTTP, mô hình kéo hàng loạt và cuối cùng là mô hình đẩy dựa trên sự kiện. Ba kỹ thuật đầu tiên lấy dữ liệu từ mỗi cơ sở dữ liệu dịch vụ, do đó có tên phản mẫu là “báo cáo phạm vi tiếp cận”. Vì ba mô hình đầu tiên thể hiện vấn đề liên quan đến phản mẫu này, nên trước tiên chúng ta hãy xem xét các kỹ thuật đó để xem tại sao chúng lại khiến bạn gặp rắc rối.

Sự cố với Báo cáo vi dịch vụ

Vấn đề với việc báo cáo có hai mặt: làm cách nào để bạn có được dữ liệu báo cáo kịp thời mà vẫn duy trì bối cảnh giới hạn giữa dịch vụ và dữ liệu của dịch vụ đó? Hãy nhớ rằng bối cảnh bị giới hạn trong vi dịch vụ bao gồm dịch vụ và dữ liệu tương ứng của dịch vụ đó và việc duy trì dịch vụ đó là rất quan trọng.

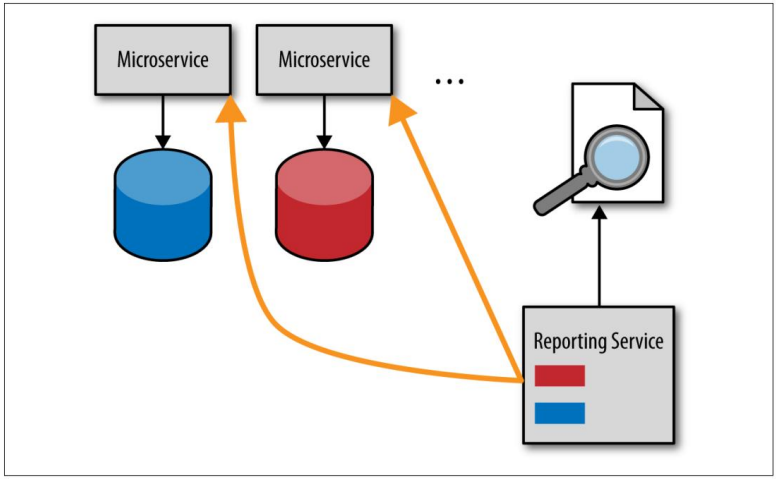
Một trong những cách báo cáo thường được xử lý trong kiến trúc vi dịch vụ là sử dụng mô hình kéo cơ sở dữ liệu, trong đó dịch vụ báo cáo (hoặc yêu cầu báo cáo) lấy dữ liệu trực tiếp từ cơ sở dữ liệu dịch vụ. Kỹ thuật này được minh họa trong Hình 4-1.



Hình 4-1. Mô hình báo cáo kéo cơ sở dữ liệu

Về mặt logic, cách nhanh nhất và dễ dàng nhất để có được dữ liệu kịp thời là truy cập trực tiếp vào dữ liệu đó. Mặc dù điều này có vẻ là một ý tưởng hay vào thời điểm đó nhưng nó dẫn đến sự phụ thuộc lẫn nhau đáng kể giữa các dịch vụ và dịch vụ báo cáo. Đây là cách triển khai điển hình của kiểu tích hợp cơ sở dữ liệu dùng chung, kết hợp các ứng dụng với nhau thông qua cơ sở dữ liệu dùng chung. Điều này có nghĩa là các dịch vụ không còn sở hữu dữ liệu của họ nữa. Bất kỳ thay đổi lược đồ cơ sở dữ liệu dịch vụ hoặc tái cấu trúc cơ sở dữ liệu nào cũng phải bao gồm các sửa đổi dịch vụ báo cáo, phá vỡ bối cảnh giới hạn quan trọng giữa dịch vụ và dữ liệu.

Cách để tránh vấn đề ghép dữ liệu là sử dụng một kỹ thuật khác gọi là mô hình kéo HTTP. Với mô hình này, thay vì truy cập trực tiếp vào từng cơ sở dữ liệu dịch vụ, dịch vụ báo cáo sẽ thực hiện lệnh gọi HTTP ổn định tới từng dịch vụ, yêu cầu dữ liệu của dịch vụ đó. Mô hình này được minh họa trong Hình 4-2.

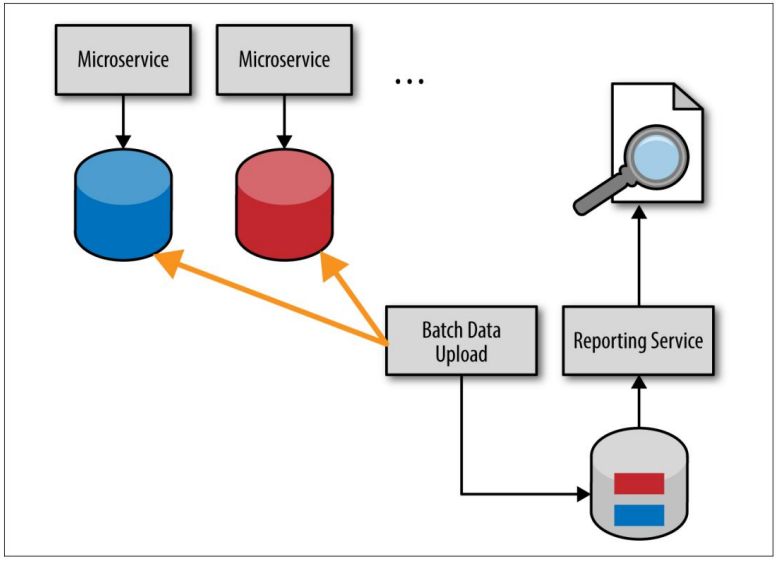


Hình 4-2. Mô hình báo cáo kéo HTTP

Mặc dù mô hình này duy trì bối cảnh giới hạn của từng dịch vụ nhưng rất tiếc là nó quá chậm, đặc biệt đối với các yêu cầu báo cáo phức tạp. Hơn nữa, tùy thuộc vào báo cáo được yêu cầu, khối lượng dữ liệu có thể quá lớn so với tải trọng đối với một lệnh gọi HTTP đơn giản.

Tùy chọn thứ ba để giải quyết các vấn đề liên quan đến mô hình kéo HTTP là sử dụng mô hình kéo hàng loạt được minh họa trong Hình 4-3. Lưu ý rằng mô hình này sử dụng cơ sở dữ liệu báo cáo hoặc kho dữ liệu riêng biệt chứa dữ liệu báo cáo tổng hợp và rút gọn.

Cơ sở dữ liệu báo cáo thường được điền thông qua một công việc hàng loạt chạy vào buổi tối để trích xuất tất cả dữ liệu báo cáo đã thay đổi, tổng hợp và giảm bớt dữ liệu đó, đồng thời chèn dữ liệu đó vào cơ sở dữ liệu báo cáo hoặc kho dữ liệu.

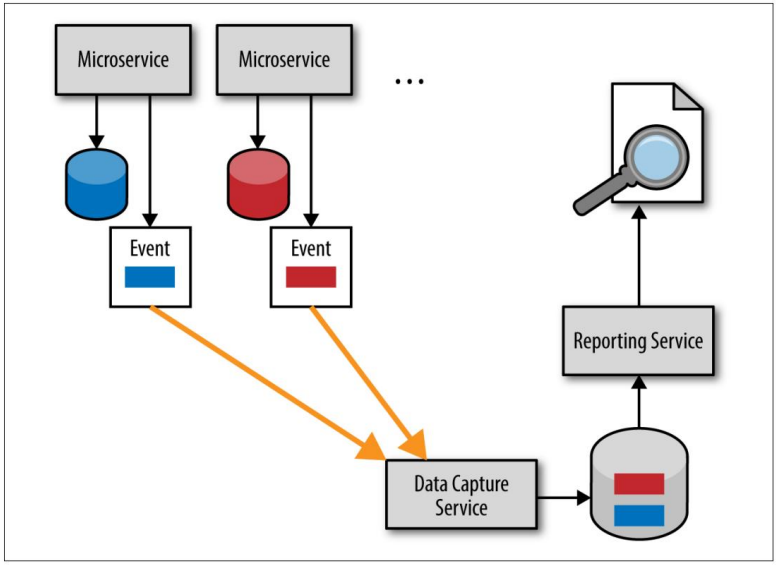


Hình 4-3. Mô hình báo cáo kéo hàng loạt

Mô hình kéo hàng loạt có cùng một vấn đề với mô hình kéo HTTP—cả hai đều triển khai kiểu tích hợp cơ sở dữ liệu dùng chung—do đó phá vỡ bối cảnh giới hạn của từng dịch vụ. Nếu lược đồ cơ sở dữ liệu dịch vụ thay đổi thì quá trình tải lên dữ liệu hàng loạt cũng phải thay đổi. thôi.

Đẩy sự kiện không đồng bộ

Giải pháp để tránh phản mẫu báo cáo phạm vi tiếp cận là sử dụng cái được gọi là mô hình đẩy dựa trên sự kiện. Sam Newman, trong cuốn sách Xây dựng vi dịch vụ của mình, gọi kỹ thuật này là máy bơm dữ liệu. Mô hình này, được minh họa trong Hình 4-4, dựa vào xử lý sự kiện không đồng bộ để đảm bảo cơ sở dữ liệu báo cáo có thông tin chính xác càng sớm càng tốt.



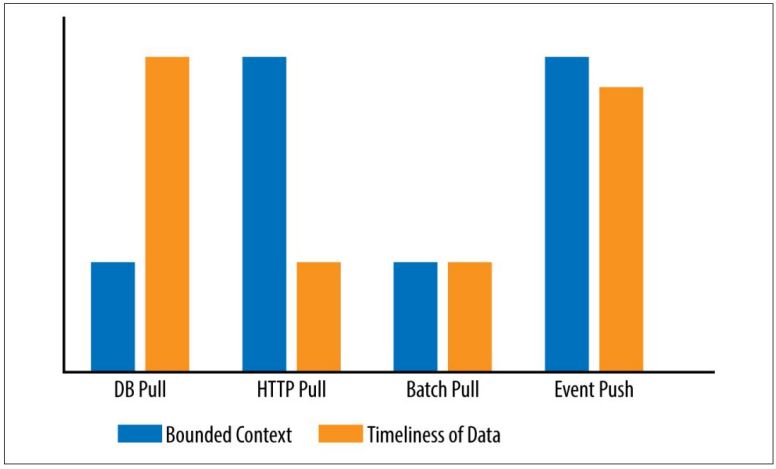
Hình 4-4. Mô hình báo cáo đẩy dựa trên sự kiện

Mặc dù đúng là mô hình đẩy dựa trên sự kiện tương đối phức tạp để triển khai nhưng nó vẫn duy trì bối cảnh giới hạn của từng dịch vụ đồng thời đảm bảo tính kịp thời hợp lý của dữ liệu.
Giống như mô hình kéo hàng loạt, mô hình này cũng có cơ sở dữ liệu báo cáo riêng do dịch vụ báo cáo sở hữu. Tuy nhiên, thay vì lấy dữ liệu theo lô, mỗi vi dịch vụ sẽ gửi không đồng bộ các cập nhật dữ liệu đáng chú ý của nó (ví dụ: dữ liệu mà dịch vụ báo cáo cần) dưới dạng một sự kiện riêng biệt đến dịch vụ thu thập dữ liệu, sau đó giảm dữ liệu và cập nhật cơ sở dữ liệu báo cáo.

Mô hình đẩy dựa trên sự kiện yêu cầu một hợp đồng giữa mỗi vi dịch vụ và dịch vụ thu thập dữ liệu đối với dữ liệu mà nó gửi không đồng bộ, nhưng hợp đồng đó tách biệt với lược đồ cơ sở dữ liệu do dịch vụ sở hữu. Tuy nhiên, các dịch vụ có phần kết hợp với nhau ở chỗ mỗi dịch vụ phải biết khi nào nên gửi thông tin nào cho mục đích báo cáo.

Trong biểu đồ ở Hình 4-5, bạn có thể thấy rằng mô hình kéo cơ sở dữ liệu tối đa hóa tính kịp thời của dữ liệu nhưng lại phá vỡ bối cảnh bị giới hạn. Mô hình kéo HTTP duy trì bối cảnh bị giới hạn nhưng có các vấn đề liên quan đến thời gian chờ và khối lượng dữ liệu. Mô hình kéo theo lô hóa ra là mô hình ít được mong muốn nhất trong số bốn tùy chọn vì không tối ưu hóa bối cảnh bị giới hạn cũng như tính kịp thời của

dữ liệu. Chỉ mô hình đẩy dựa trên sự kiện mới tối đa hóa cả bối cảnh bị ràng buộc của từng dịch vụ và tính kịp thời của dữ liệu báo cáo.



Hình 4-5. So sánh các mô hình báo cáo

CHƯƠNG 5

Hạt cát cạm bẫy

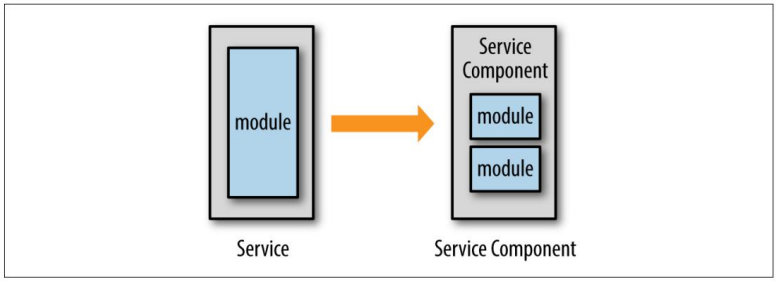
Có lẽ một trong những thách thức lớn nhất mà các kiến trúc sư và nhà phát triển phải đối mặt khi tạo ứng dụng bằng kiến trúc microservices là mức độ chi tiết của dịch vụ. Một dịch vụ nên lớn đến mức nào? Nó nên nhỏ đến mức nào? Việc chọn mức độ chi tiết phù hợp cho dịch vụ của bạn là rất quan trọng đối với sự thành công của bất kỳ nỗ lực dịch vụ vi mô nào. Mức độ chi tiết của dịch vụ có thể tác động đến hiệu suất, độ bền, độ tin cậy, kiểm soát thay đổi, khả năng kiểm thử và thậm chí cả việc triển khai.

Những cạm bẫy cát xảy ra khi các kiến trúc sư và nhà phát triển tạo ra các dịch vụ quá mịn. Đợi đã—chẳng phải đó là lý do tại sao ngay từ đầu nó được gọi là microservices sao? Từ "vi mô" ngụ ý rằng một dịch vụ nên rất nhỏ, nhưng "nhỏ" đến mức nào?

Một trong những lý do chính khiến cạm bẫy này xảy ra là do các nhà phát triển thường nhầm lẫn giữa dịch vụ với một lớp. Đã rất nhiều lần tôi thấy các nhóm phát triển tạo ra các dịch vụ bằng cách nghĩ rằng lớp triển khai mà họ đang viết thực sự là dịch vụ. Không có gì có thể hơn từ sự thật.

Một dịch vụ phải luôn được coi là một thành phần dịch vụ. Thành phần dịch vụ là thành phần của kiến trúc thực hiện một chức năng cụ thể trong hệ thống. Thành phần dịch vụ phải có tuyên bố trách nhiệm và vai trò rõ ràng, ngắn gọn và có một tập hợp các hoạt động được xác định rõ ràng. Nhà phát triển có quyền quyết định cách triển khai thành phần dịch vụ và cần bao nhiêu lớp triển khai cho dịch vụ.

Như Hình 5-1 cho thấy, một thành phần dịch vụ được triển khai thông qua một hoặc nhiều mô-đun (ví dụ: các lớp Java). Việc triển khai một thành phần dịch vụ bằng cách sử dụng mối quan hệ một-một giữa mô-đun và thành phần dịch vụ không chỉ dẫn đến các thành phần quá chi tiết mà còn dẫn đến thực tiễn lập trình kém. Các dịch vụ được triển khai thông qua một lớp duy nhất có xu hướng có các lớp quá lớn và mang quá nhiều trách nhiệm, khiến chúng khó bảo trì và kiểm tra.



Hình 5-1. Mối quan hệ giữa các module và một dịch vụ

Số lượng các lớp triển khai không phải là đặc điểm xác định để xác định mức độ chi tiết của dịch vụ. Một số dịch vụ có thể chỉ cần một tệp lớp duy nhất để triển khai tất cả chức năng kinh doanh, trong khi những dịch vụ khác có thể cần sáu lớp trở lên.

Nếu số lượng các lớp triển khai không ảnh hưởng đến mức độ chi tiết của một dịch vụ thì điều gì sẽ xảy ra? May mắn thay, có ba thử nghiệm cơ bản mà bạn có thể sử dụng để xác định mức độ chi tiết phù hợp cho dịch vụ của mình: phạm vi và chức năng dịch vụ, nhu cầu giao dịch cơ sở dữ liệu và cuối cùng là mức độ hoạt động của dịch vụ.

Phân tích phạm vi và chức năng dịch vụ

Cách đầu tiên để xác định xem dịch vụ của bạn có mức độ chi tiết phù hợp hay không là phân tích phạm vi và chức năng của dịch vụ. Dịch vụ này làm gì? Hoạt động của nó là gì? Ghi lại hoặc nêu rõ phạm vi và chức năng của dịch vụ là một cách tuyệt vời để xác định xem dịch vụ có đang hoạt động quá nhiều hay không. Việc sử dụng các từ như “và” và “ngoài ra” thường là dấu hiệu tốt cho thấy dịch vụ đó có thể đang hoạt động quá nhiều.

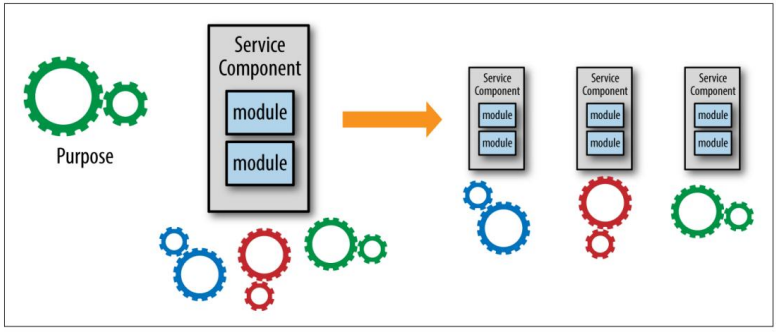
Sự gắn kết cũng đóng một vai trò liên quan đến phạm vi và chức năng dịch vụ. Sự gắn kết được định nghĩa là mức độ và cách thức mà

hoạt động của dịch vụ có mối quan hệ với nhau. Bạn muốn cố gắng đạt được sự gắn kết chặt chẽ trong các dịch vụ của mình. Ví dụ: giả sử bạn có một dịch vụ khách hàng với các hoạt động sau:

- add_customer •
- update_customer •
- get_customer •
- thông báo cho khách hàng • record_customer_comments •
- get_customer_comments

Trong ví dụ này, ba hoạt động đầu tiên có liên quan với nhau vì chúng đều liên quan đến việc duy trì và truy xuất thông tin khách hàng. Tuy nhiên, ba cái cuối cùng (notify_customer, record_customer_comments và get_customer_comments) không liên quan đến các hoạt động CRUD cơ bản trên dữ liệu khách hàng cơ bản. Khi phân tích mức độ gắn kết của các hoạt động trong dịch vụ này, có thể thấy rõ rằng dịch vụ ban đầu có lẽ nên được chia thành ba dịch vụ riêng biệt (bảo trì khách hàng, thông báo cho khách hàng và nhận xét của khách hàng).

Hình 5-2 minh họa quan điểm rằng, nói chung, khi phân tích phạm vi và chức năng dịch vụ, bạn có thể sẽ thấy rằng các dịch vụ của mình quá thô và bạn sẽ chuyển sang các dịch vụ chi tiết hơn.



Hình 5-2. Tác động của việc phân tích chức năng và phạm vi dịch vụ

Sam Newman đưa ra một số lời khuyên hữu ích trong lĩnh vực này—hãy bắt đầu từ chi tiết thô hơn và chuyển sang chi tiết hơn khi bạn tìm hiểu thêm về dịch vụ. Làm theo lời khuyên này sẽ giúp bạn bắt đầu xác định các thành phần dịch vụ của mình mà không phải lo lắng quá nhiều về mức độ chi tiết ngay lập tức.

Mặc dù việc phân tích phạm vi và chức năng của dịch vụ là một khởi đầu tốt nhưng bạn không muốn dừng lại ở đó. Sau khi xem xét phạm vi dịch vụ, bạn cần phân tích nhu cầu giao dịch cơ sở dữ liệu của mình.

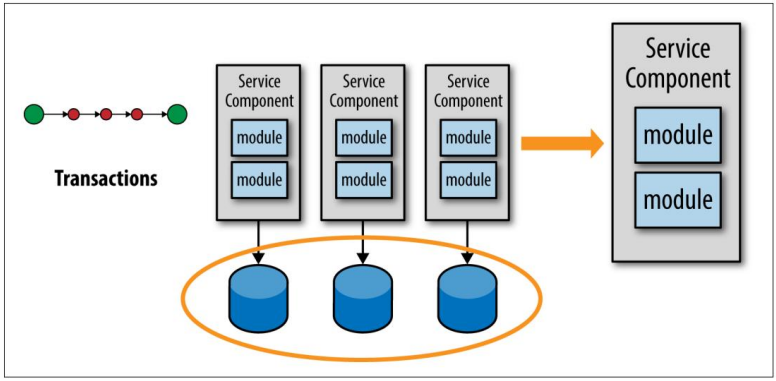
Phân tích giao dịch cơ sở dữ liệu

Một thử nghiệm khác để xác thực mức độ chi tiết của dịch vụ là nhu cầu giao dịch cơ sở dữ liệu cho một số hoạt động nhất định. Các giao dịch cơ sở dữ liệu được gọi chính thức hơn là các giao dịch ACID (tính nguyên tử, tính nhất quán, sự cô lập và độ bền). Các giao dịch ACID phối hợp nhiều bản cập nhật cơ sở dữ liệu vào một đơn vị công việc duy nhất. Các bản cập nhật cơ sở dữ liệu được cam kết dưới dạng toàn bộ hoặc được khôi phục nếu xảy ra tình trạng lỗi.

Bởi vì các dịch vụ trong kiến trúc microservice được phân phối và triển khai dưới dạng các ứng dụng riêng biệt nên việc duy trì giao dịch ACID giữa hai hoặc nhiều dịch vụ từ xa là vô cùng khó khăn. Vì lý do này, kiến trúc microservice thường dựa vào một kỹ thuật được gọi là giao dịch BASE (tính khả dụng cơ bản, trạng thái mềm và tính nhất quán cuối cùng). Bất chấp điều đó, thường sẽ có lúc bạn yêu cầu giao dịch ACID cho một số hoạt động kinh doanh nhất định.

Nếu bạn thấy mình liên tục gặp phải các vấn đề xung quanh ACID so với ACID. Các giao dịch BASE và bạn cần phối hợp nhiều bản cập nhật, rất có thể bạn đã làm cho dịch vụ của mình trở nên quá chi tiết.

Khi phân tích nhu cầu giao dịch của bạn và thấy rằng bạn không thể sống với sự nhất quán cuối cùng, bạn thường sẽ chuyển từ các dịch vụ chi tiết hơn sang các dịch vụ chi tiết hơn, do đó giữ nhiều bản cập nhật được phối hợp trong một bối cảnh dịch vụ duy nhất, như minh họa trong Hình 5-3 .



Hình 5-3. Tác động của việc phân tích các giao dịch cơ sở dữ liệu

Lưu ý rằng từ quan điểm giao dịch ACID, việc bạn hợp nhất các cơ sở dữ liệu riêng biệt hay giữ chúng dưới dạng các cơ sở dữ liệu riêng lẻ không thành vấn đề. Nói chung, bạn cũng sẽ muốn hợp nhất cơ sở dữ liệu, nhưng đây không phải là yêu cầu để duy trì giao dịch ACID (giả sử cơ sở dữ liệu và trình quản lý giao dịch bạn đang sử dụng hỗ trợ XA—ví dụ: giao dịch cam kết hai pha).

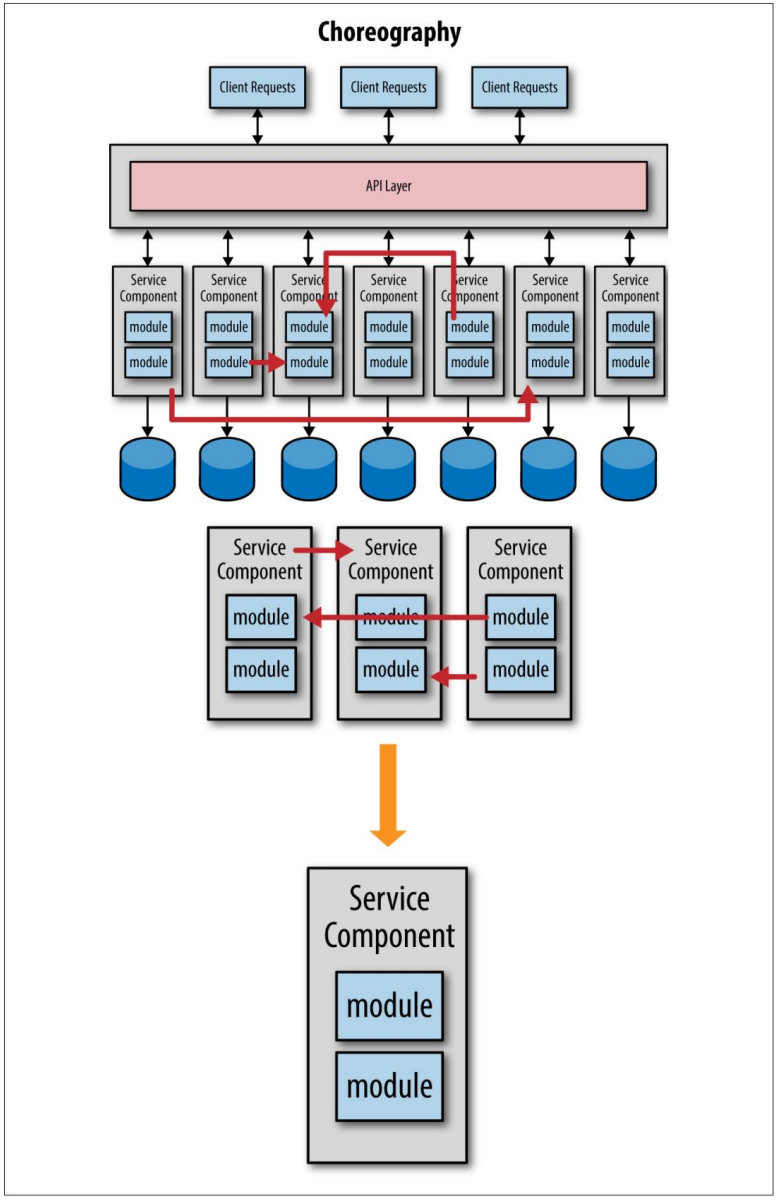
Sau khi bạn đã phân tích nhu cầu giao dịch của mình, đã đến lúc chuyển sang thử nghiệm thứ ba, cách thực hiện dịch vụ.

Phân tích biên đạo dịch vụ

Thử nghiệm thứ ba mà bạn có thể sử dụng để xác thực mức độ chi tiết của dịch vụ là cách thực hiện dịch vụ. Biên đạo dịch vụ đề cập đến sự giao tiếp giữa các dịch vụ, cũng thường được gọi là giao tiếp giữa các dịch vụ. Biên đạo dịch vụ nói chung là điều bạn muốn cẩn thận trong kiến trúc vi dịch vụ. Trước hết, nó làm giảm hiệu suất tổng thể của ứng dụng của bạn vì mỗi cuộc gọi đến dịch vụ khác đều là một cuộc gọi từ xa. Ví dụ: giả sử phải mất 100 mili giây để thực hiện một cuộc gọi yên tĩnh đến một dịch vụ khác, thì việc thực hiện năm cuộc gọi dịch vụ từ xa chỉ mất nửa giây trong thời gian truy cập từ xa.

Vấn đề khác với việc sử dụng quá nhiều dịch vụ là nó có thể ảnh hưởng đến độ tin cậy và độ bền tổng thể của hệ thống của bạn. Bạn càng thực hiện nhiều cuộc gọi từ xa cho một yêu cầu kinh doanh thì khả năng một trong những cuộc gọi từ xa đó sẽ không thành công hoặc hết thời gian chờ càng cao.

Nếu bạn thấy mình phải giao tiếp với quá nhiều dịch vụ để hoàn thành các yêu cầu kinh doanh đơn lẻ thì có thể bạn đã làm cho dịch vụ của mình quá chi tiết. Khi phân tích mức độ của vũ đạo dịch vụ, nhìn chung bạn sẽ chuyển từ các dịch vụ chi tiết hơn sang các dịch vụ chi tiết hơn, như minh họa trong Hình 5-4.



Hình 5-4. Tác động của việc phân tích vũ đạo dịch vụ

Bằng cách hợp nhất các dịch vụ và chuyển sang các dịch vụ thô hơn, bạn có thể cải thiện hiệu suất và tăng độ tin cậy cũng như độ bền tổng thể cho các ứng dụng của mình. Bạn cũng loại bỏ

sự phụ thuộc giữa các dịch vụ, cho phép kiểm soát thay đổi, thử nghiệm và triển khai tốt hơn.

Cách tiếp cận khác khi xử lý vũ đạo dịch vụ nhằm giúp khắc phục các vấn đề về hiệu suất và độ tin cậy là tận dụng xử lý song song không đồng bộ kết hợp với các kỹ thuật kiến trúc phản ứng để xử lý lỗi. Việc thực hiện nhiều yêu cầu cùng lúc sẽ làm tăng khả năng phản hồi tổng thể, cho phép bạn phối hợp nhiều dịch vụ trong một yêu cầu kinh doanh một cách kịp thời. Điểm mấu chốt ở đây là hiểu và phân tích sự cân bằng liên quan đến hoạt động dịch vụ để đảm bảo vừa đủ khả năng đáp ứng cho người dùng vừa đủ độ tin cậy tổng thể cho hệ thống của bạn.

CHƯƠNG 6

Cạm bẫy của nhà phát triển không có nguyên nhân

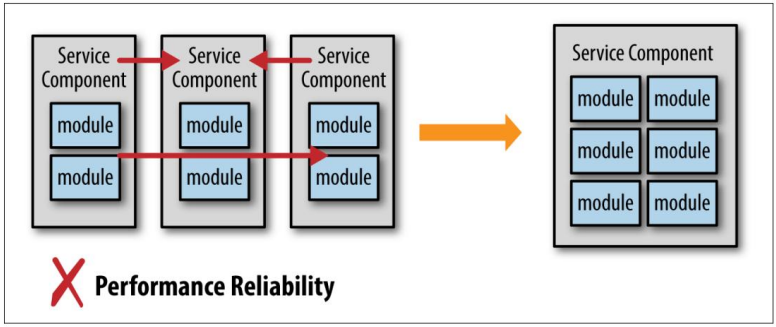
Lần đầu tiên tôi nhìn thấy James Dean trong bộ phim Rebel Without a Cause khi tôi chỉ là một cậu bé, nhưng tôi vẫn nhớ mọi thứ về bộ phim. Khi nghĩ về tên cho phản mẫu này, tôi nghĩ ngay đến James Dean – một thanh niên gặp rắc rối và đưa ra những quyết định sai lầm. Hoàn hảo.

Tôi đã quan sát nhiều lần hơn rằng tôi có thể thấy các kiến trúc sư và nhà phát triển đưa ra quyết định về các khía cạnh khác nhau của microservice, đặc biệt là liên quan đến mức độ chi tiết của dịch vụ và các công cụ dành cho nhà phát triển, vì tất cả các lý do sai lầm. Tất cả đều quy về sự đánh đổi. Rich Hickey nói "Các lập trình viên biết lợi ích của mọi thứ và sự đánh đổi của không có gì". Bạn tôi, Neal Ford, thích theo dõi câu nói của Rich bằng cách nói "Kiến trúc sư phải hiểu cả hai". Tôi khẳng định rằng các nhà phát triển cũng nên biết cả hai điều này.

Đưa ra những quyết định sai lầm

Hình 6-1 minh họa một tình huống phổ biến trong đó các dịch vụ được phát hiện là quá chi tiết, do đó ảnh hưởng đến hiệu suất và độ tin cậy tổng thể do lượng giao tiếp giữa các dịch vụ giữa chúng. Trong kịch bản này, nhà phát triển hoặc kiến trúc sư đưa ra quyết định rằng các dịch vụ này nên được hợp nhất thành một

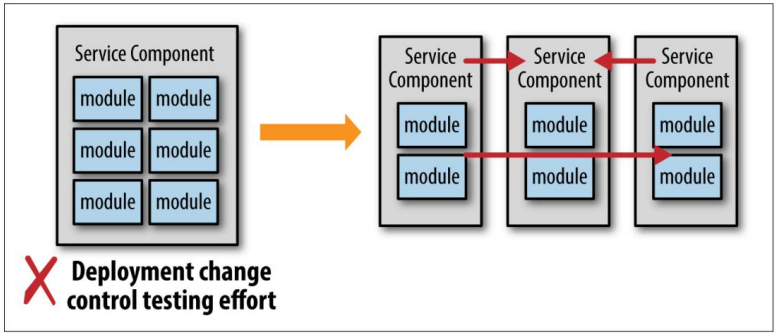
dịch vụ đơn lẻ, chi tiết hơn để giải quyết các vấn đề về hiệu suất và độ tin cậy.



Hình 6-1. Chuyển từ hạt mịn sang hạt thô

Mặc dù đây có vẻ là một quyết định hợp lý nhưng sự đánh đổi của việc làm này lại bị bỏ qua. Việc triển khai, kiểm soát thay đổi và thử nghiệm đều bị ảnh hưởng khi chuyển sang một dịch vụ chi tiết đơn lẻ. Câu hỏi là, điều gì là quan trọng nhất?

Hãy xem xét ví dụ được minh họa trong Hình 6-2 trong đó tình huống ngược lại xảy ra. Trong kịch bản này, các dịch vụ quá thô, do đó ảnh hưởng đến nỗ lực thử nghiệm tổng thể và sự phối hợp triển khai. Trong trường hợp này, kiến trúc sư hoặc nhà phát triển đưa ra quyết định rằng dịch vụ nên được chia thành các dịch vụ nhỏ hơn để giảm phạm vi của từng dịch vụ, do đó giúp chúng dễ dàng kiểm tra và triển khai hơn.



Hình 6-2. Chuyển từ hạt thô sang hạt mịn

Mặc dù chúng ta có thể hoan nghênh kiến trúc sư hoặc nhà phát triển đã đưa ra quyết định này nhưng sự đánh đổi một lần nữa lại bị lãng quên. Mặc dù các dịch vụ chắc chắn dễ kiểm tra và triển khai hơn nhờ thay đổi này nhưng ứng dụng đột nhiên gặp vấn đề về hiệu suất và độ tin cậy do sự gia tăng trong quy trình dịch vụ. Cái nào quan trọng hơn?

Hiểu động lực kinh doanh

Hiểu được động lực kinh doanh đằng sau việc lựa chọn microservice là chìa khóa để tránh cạm bẫy này. Mọi kiến trúc sư và nhà phát triển trong nhóm nên biết câu trả lời cho từng câu hỏi sau:

- Tại sao bạn làm microservices? • Động lực kinh doanh chính là gì? • Đặc điểm kiến trúc nào là quan trọng nhất?

Sử dụng khả năng triển khai, hiệu suất, độ bền và khả năng mở rộng làm đặc điểm kiến trúc chính, hãy xem xét các tình huống sau đây trong đó đã biết trình điều khiển kinh doanh. Lưu ý rằng các yếu tố thúc đẩy kinh doanh là yếu tố thúc đẩy quyết định về hợp nhất dịch vụ hoặc chia tách dịch vụ chứ không phải bản thân các đặc điểm.

Kịch bản 1: Lý do chuyển sang microservice là để đạt được thời gian tiếp thị tốt hơn thông qua quy trình triển khai hiệu quả.

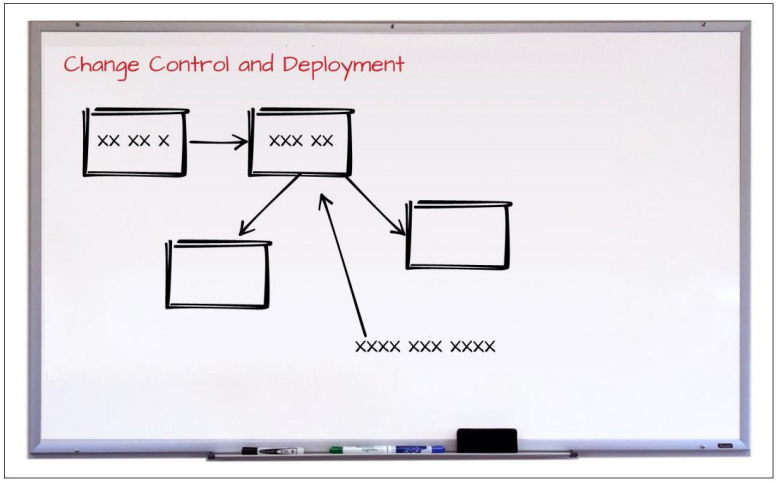
Trong kịch bản này, khả năng triển khai của từng dịch vụ quan trọng hơn hiệu suất, độ tin cậy và khả năng mở rộng, do đó, với động lực kinh doanh này, bạn sẽ có xu hướng tạo ra nhiều dịch vụ chi tiết hơn, đánh đổi sự gia tăng tiềm năng trong quy trình dịch vụ (và do đó tác động đến hiệu suất). và độ tin cậy). Quay lại Hình 6-1, với trình điều khiển này, nhà phát triển thực sự đã đưa ra quyết định sai lầm khi hợp nhất các dịch vụ.

Kịch bản 2: Lý do chuyển sang microservice là để tăng độ tin cậy và tính mạnh mẽ tổng thể của ứng dụng.

Tình huống này là lý do phổ biến khiến các công ty chuyển từ ứng dụng nguyên khối sang kiến trúc vi dịch vụ, chủ yếu là do các vấn đề với kiến trúc nguyên khối xung quanh khả năng ghép nối chặt chẽ và do đó các ứng dụng dễ vỡ. Trong trường hợp này, người điều khiển doanh nghiệp nêu rõ nhu cầu về độ tin cậy và độ mạnh mẽ, nghĩa là bạn có thể sẽ đánh đổi việc dễ dàng thử nghiệm và triển khai để có độ tin cậy và độ bền cao hơn, do đó ưu tiên các dịch vụ chi tiết thô hơn thay vì các dịch vụ chi tiết hơn.

Một kỹ thuật mà tôi thường sử dụng là viết các động lực kinh doanh bằng chữ lớn màu đỏ lên trên bảng trắng chung của nhóm như minh họa trong Hình 6-3. Sau đó, bất cứ khi nào có quyết định về mức độ chi tiết của dịch vụ hoặc lựa chọn công cụ, nhóm luôn có thể tra cứu, tham khảo bảng trắng và nói “ồ, vâng, đúng vậy. Được rồi, hãy giữ các dịch vụ

chi tiết hơn và tìm ra cách khác để giải quyết các vấn đề về hiệu suất và độ tin cậy.”



Hình 6-3. Đặt các động lực kinh doanh lên bảng trắng

CHƯƠNG 7

Nhảy vào cạm bẫy Bandwagon

Bạn phải nắm lấy microservice. Không thể phủ nhận đó là xu hướng mới nhất trong ngành, mọi người khác đang làm theo, và bên cạnh đó, thật tuyệt khi có trong sơ yếu lý lịch của bạn.

Để vượt qua cạm bẫy của đám đông, tất cả là về việc tận dụng các dịch vụ vi mô trước khi phân tích nhu cầu kinh doanh, động lực kinh doanh cũng như cơ cấu tổ chức tổng thể và môi trường công nghệ. Mặc dù kiến trúc microservices là một phong cách kiến trúc rất mạnh mẽ và phổ biến nhưng nó không phù hợp với mọi ứng dụng hoặc môi trường.

Bạn có thể dễ dàng tránh được cạm bẫy này bằng cách hiểu rõ những ưu điểm và nhược điểm của microservice. Sau đó, khi đã hiểu đầy đủ về microservice là gì, bạn có thể kết hợp nhu cầu và mục tiêu kinh doanh của mình với các đặc điểm kiến trúc để xác định xem microservice có phù hợp với tình huống và tổ chức của bạn hay không. Bạn cũng có thể tránh cạm bẫy này bằng cách tìm hiểu thêm về các mẫu kiến trúc khác có thể phù hợp hơn với trường hợp của bạn.

Ưu điểm và nhược điểm

Bước đầu tiên để tránh cạm bẫy này là hiểu những ưu điểm và nhược điểm của phong cách kiến trúc microservices. Sau đây là một số lợi ích quan trọng hơn mà bạn nên biết:

Triển khai Dễ

dàng triển khai là một trong những động lực chính để chuyển sang kiến trúc vi dịch vụ. Microservice có kích thước nhỏ, đơn lẻ

dịch vụ mục đích được triển khai như các ứng dụng riêng biệt. Việc triển khai một dịch vụ đơn lẻ sẽ dễ dàng hơn và ít rủi ro hơn nhiều so với việc triển khai toàn bộ ứng dụng nguyên khối. Trên thực tế, toàn bộ khái niệm về phân phối liên tục một phần là nguyên nhân thúc đẩy việc tạo ra phong cách kiến trúc microservices.

Khả năng
kiểm thử Dễ kiểm thử là một lợi thế lớn khác của kiến trúc microservice. Phạm vi nhỏ của một dịch vụ cùng với việc thiếu sự phụ thuộc chung với các dịch vụ khác khiến chúng tương đối dễ kiểm tra. Tuy nhiên, có lẽ một trong những khía cạnh quan trọng hơn của đặc điểm này là với microservice, bạn có khả năng thực hiện kiểm tra hồi quy hoàn chỉnh hơn so với các ứng dụng nguyên khối lớn hơn.

Kiểm soát thay
đổi Với microservice, việc kiểm soát những gì sẽ thay đổi khi thêm chức năng mới sẽ dễ dàng hơn. Điều này một lần nữa là do phạm vi dịch vụ hạn chế và bối cảnh giới hạn được duy trì bởi mỗi dịch vụ. Có các dịch vụ nhỏ, độc lập và ít phụ thuộc lẫn nhau đồng nghĩa với việc có ít sự phối hợp hơn để phát triển, thử nghiệm và đưa ra các thay đổi.

Microservices
có tính mô-đun là một phong cách kiến trúc có tính mô-đun cao, từ đó dẫn đến các ứng dụng có tính linh hoạt cao. Sự nhanh nhẹn được định nghĩa tốt nhất là khả năng phản ứng nhanh với sự thay đổi. Kiến trúc càng mô-đun thì khả năng phát triển, thử nghiệm và đưa ra các thay đổi càng nhanh. Phong cách kiến trúc microservices có lẽ là kiến trúc mô đun nhất trong số tất cả các mẫu kiến trúc do mức độ chi tiết của dịch vụ.

Khả năng
mở rộng Bởi vì microservice là các dịch vụ đơn mục đích chi tiết được triển khai riêng biệt nên kiểu kiến trúc này có khả năng mở rộng ở mức cao nhất trong số tất cả các mẫu kiến trúc. Tương đối dễ dàng để mở rộng quy mô một phần chức năng cụ thể với phong cách kiến trúc microservice, một phần do tính chất được chứa trong cấu trúc liên kết dịch vụ và các công cụ giám sát tinh vi cho phép bạn khởi động và dừng dịch vụ một cách linh hoạt thông qua tự động hóa.

Mặc dù những ưu điểm này có thể thuyết phục bạn rằng microservice là giải pháp tốt nhất cho trường hợp của bạn, nhưng hãy xem xét danh sách nhược điểm sau đây.

Thay đổi tổ chức

Microservices yêu cầu thay đổi tổ chức ở nhiều cấp độ.

Các nhóm phát triển phải được cơ cấu lại và tổ chức lại thành các nhóm đa chức năng hơn để các nhóm nhỏ có thể sở hữu các khía cạnh kỹ thuật toàn diện của dịch vụ mà họ chịu trách nhiệm, bao gồm giao diện người dùng, xử lý phụ trợ, xử lý quy tắc và cơ sở dữ liệu. xử lý và mô hình hóa. Mô hình nhóm phát triển doanh nghiệp truyền thống gồm các nhóm giao diện người dùng, nhóm phát triển phụ trợ và kỹ sư/quản trị viên cơ sở dữ liệu đơn giản là không hoạt động với kiến trúc vi dịch vụ. Ngoài ra, cơ cấu tổ chức liên quan đến việc phát hành phần mềm cũng phải thay đổi. Với microservice, việc sử dụng các quy trình vòng đời phát triển phần mềm truyền thống tồn tại với các kiến trúc nguyên khối, phân lớp là không khả thi. Thay vào đó, bạn phải tận dụng khả năng tự động hóa và tận dụng các công cụ cũng như phương pháp thực hành của nhà phát triển để phát triển quy trình triển khai hiệu quả nhằm phát hành các dịch vụ vi mô.

Hiệu suất Bởi

vì mỗi vi dịch vụ là một ứng dụng được triển khai riêng biệt nên việc giao tiếp đến và đi từ các dịch vụ cũng như giao tiếp giữa các dịch vụ đều diễn ra từ xa. Hiệu suất có thể bị ảnh hưởng đáng kể tùy thuộc vào môi trường của bạn và số lượng dịch vụ bạn có trong ứng dụng vi dịch vụ của mình. Điều quan trọng là phải hiểu độ trễ truy cập từ xa của bạn (xem “Chúng ta có gặp bẫy không”) và mức độ liên lạc dịch vụ mà bạn sẽ cần (xem Grains of Sand Pitfall) để hiểu đầy đủ tác động hiệu suất của việc sử dụng microservi-

những cái này.

Độ tin cậy

Vì những lý do tương tự khiến hiệu suất có thể bị ảnh hưởng khi sử dụng vi dịch vụ, độ tin cậy tổng thể cũng tương tự như vậy.
Vì mọi yêu cầu đều là một cuộc gọi truy cập từ xa nên bạn có nguy cơ gặp phải rủi ro là một trong các dịch vụ bạn cần liên lạc để hoàn thành một yêu cầu công việc không có sẵn hoặc không phản hồi.

Với kiến trúc vi dịch vụ, bạn có thể có từ hàng trăm đến thậm chí hàng nghìn vi dịch vụ. Do

với số lượng lớn dịch vụ mà bạn có thể có, việc quản lý thủ công hàng trăm chu kỳ phát hành và triển khai đồng thời là không khả thi. Tự động hóa và cộng tác liên tục giữa các nhà phát triển, người thử nghiệm và kỹ sư phát hành là yếu tố quan trọng đối với sự thành công của bất kỳ nỗ lực dịch vụ vi mô nào. Vì lý do này, bạn cần sử dụng nhiều công cụ và phương pháp thực hành liên quan đến hoạt động khác nhau, đây có thể là một nhiệm vụ rất phức tạp. Có khoảng 12 loại công cụ và khung liên quan đến hoạt động khác nhau được sử dụng trong kiến trúc vi dịch vụ và mỗi danh mục đó chứa hàng chục lựa chọn công cụ và sản phẩm. Ví dụ: có các công cụ giám sát, công cụ đăng ký và khám phá dịch vụ, công cụ triển khai, v.v. Cái nào là tốt nhất cho môi trường và hoàn cảnh của bạn? Câu trả lời cho câu hỏi này đòi hỏi vài tháng nghiên cứu, nỗ lực chứng minh khái niệm và phân tích đánh đổi để xác định sự kết hợp tốt nhất giữa các công cụ và khuôn khổ cho ứng dụng và môi trường của bạn.

Phù hợp với nhu cầu kinh doanh

Sau khi hiểu những ưu điểm và nhược điểm của phong cách kiến trúc dịch vụ vi mô, bạn phải phân tích nhu cầu và mục tiêu kinh doanh của mình để xác định xem liệu vi dịch vụ có phải là phương pháp phù hợp cho vấn đề bạn đang cố gắng giải quyết hay không. Khi xác định liệu microservice có phù hợp hay không, hãy tự hỏi mình những câu hỏi sau:

- Mục tiêu kinh doanh và kỹ thuật của tôi là gì?
- Tôi đang cố gắng đạt được điều gì với microservice?
- Những điểm khó khăn hiện tại và có thể dự đoán trước của tôi là gì?
- Đặc điểm kiến trúc điều khiển chính của ứng dụng này là gì (ví dụ: hiệu suất, khả năng mở rộng, khả năng bảo trì, v.v.)?

Trả lời những câu hỏi này có thể giúp bạn kết hợp nhu cầu và mục tiêu kinh doanh của mình với những ưu điểm và nhược điểm của dịch vụ vi mô để xác định xem nó có thực sự phù hợp với tình huống của bạn hay không.

Các mẫu kiến trúc khác

Phong cách kiến trúc microservices là một phong cách rất mạnh mẽ, mang theo nhiều ưu điểm, nhưng nó không phải là phong cách kiến trúc duy nhất hiện có. Một điều khác bạn có thể làm để tránh cạm bẫy này là hiểu và phân tích các mẫu kiến trúc khác để xác định xem liệu một trong số đó có phù hợp hơn với tình huống của bạn hay không.

Bên cạnh microservice, còn có bảy mẫu kiến trúc phổ biến khác mà bạn có thể muốn xem xét cho ứng dụng hoặc hệ thống của mình:

- Kiến trúc dựa trên dịch vụ
- Kiến trúc hướng dịch vụ
- Kiến trúc phân lớp
- Kiến trúc vi nhân
- Kiến trúc dựa trên không gian
- Kiến trúc hướng sự kiện
- Kiến trúc đường ống

Tất nhiên, bạn không cần phải chọn một mẫu kiến trúc duy nhất cho ứng dụng của mình. Bạn chắc chắn có thể kết hợp các mẫu để tạo ra một giải pháp hiệu quả. Một số ví dụ là vi dịch vụ hướng sự kiện, vi nhân dựa trên sự kiện, kiến trúc dựa trên không gian phân lớp và vi nhân đường ống.

Sử dụng các tài nguyên sau để tìm hiểu thêm về các mẫu kiến trúc khác:

- Nguyên tắc cơ bản về kiến trúc phần mềm: Tìm hiểu những điều cơ bản
- Nguyên tắc cơ bản về kiến trúc phần mềm: Ngoài những điều cơ bản
- Nguyên tắc cơ bản về Kiến trúc Phần mềm: Kiến trúc Dựa trên Dịch vụ
- Các mẫu kiến trúc phần mềm)
- Vi dịch vụ so với Kiến trúc hướng dịch vụ

CHƯƠNG 8

Cạm bẫy hợp đồng tĩnh

Tất cả các microservice đều có hợp đồng giữa người tiêu dùng dịch vụ và microservice. Hợp đồng thường chứa một lược đồ chỉ định dữ liệu đầu vào và đầu ra dự kiến và đôi khi là tên của thao tác (tùy thuộc vào cách bạn triển khai dịch vụ của mình).
Các hợp đồng thường thuộc quyền sở hữu của dịch vụ và có thể được thể hiện thông qua các định dạng như XML, JSON hoặc thậm chí là đối tượng Java hoặc C#. Và tất nhiên, những hợp đồng đó không bao giờ thay đổi phải không? Sai.

Cạm bẫy hợp đồng tĩnh xảy ra khi bạn không thể phiên bản hợp đồng dịch vụ của mình ngay từ đầu, hoặc thậm chí không hề. Phiên bản hợp đồng là cực kỳ quan trọng để không chỉ tránh vi phạm các thay đổi (thay đổi hợp đồng và phá vỡ tất cả người tiêu dùng sử dụng hợp đồng đó), mà còn để duy trì tính linh hoạt bằng cách hỗ trợ khả năng tương thích ngược.

Đây là ví dụ minh họa cách bạn có thể gặp rắc rối nếu không lập phiên bản hợp đồng của mình. Giả sử bạn có một vi dịch vụ được truy cập bởi ba máy khách khác nhau (máy khách 1, máy khách 2 và máy khách 3).
Khách hàng 1 muốn thay đổi hợp đồng dịch vụ ngay. Bạn kiểm tra với khách hàng 2 và khách hàng 3 để xem liệu họ có thể chấp nhận thay đổi hay không và cả hai khách hàng đều thông báo cho bạn rằng sẽ mất vài tuần để thực hiện thay đổi đó do những vấn đề khác đang xảy ra với những khách hàng đó. Bây giờ bạn phải thông báo cho khách hàng 1 rằng sẽ mất vài tuần để thực hiện thay đổi đó vì bạn cần phối hợp cập nhật với khách hàng 2 và 3. Tuy nhiên, khách hàng 1 không thể đợi hàng tuần.

Bằng cách cung cấp phiên bản trong hợp đồng của bạn và do đó cung cấp khả năng tương thích ngược, giờ đây bạn có thể linh hoạt hơn về mặt môi trường.

yêu cầu của 1. Sự nhanh nhẹn được định nghĩa là tốc độ bạn có thể phản ứng với sự thay đổi. Nếu bạn lập phiên bản hợp đồng của mình đúng cách ngay từ đầu, bạn có thể phản hồi ngay yêu cầu thay đổi hợp đồng của khách hàng 1 bằng cách tạo một phiên bản mới của hợp đồng, chẳng hạn như phiên bản 1.1. Khách hàng 2 và 3 đều đang sử dụng phiên bản 1.0 của hợp đồng nên giờ đây bạn có thể thực hiện thay đổi ngay lập tức mà không cần phải đợi khách hàng 2 hoặc khách hàng 3 phản hồi. Ngoài ra, bạn có thể thực hiện thay đổi mà không cần thực hiện cái gọi là "thay đổi đột phá".

Có hai kỹ thuật cơ bản để lập phiên bản hợp đồng: lập phiên bản ở cấp tiêu đề và lập phiên bản trong chính lược đồ hợp đồng. Trong chương này tôi sẽ trình bày chi tiết từng kỹ thuật này, nhưng trước tiên hãy xem một ví dụ.

Thay đổi hợp đồng

Để minh họa vấn đề không lập phiên bản hợp đồng, tôi sẽ sử dụng ví dụ về việc mua một số lượng cổ phiếu nhất định của cổ phiếu phổ thông Apple (AAPL). Lược đồ cho yêu cầu này có thể trông giống như thế này:

```
{
     "$schema": "http://json-schema.org/draft-04/schema#", "properties":
     { "acct": {"type":
        "number"}, "cusip": {"type ":
        "chuỗi"}, "chia sẻ": {"type": "số",
        "tối thiểu": 100}

    }, "bắt buộc": ["tài khoản", "cusip", "chia sẻ"]
}
```

Trong trường hợp này để mua cổ phiếu, bạn phải chỉ định tài khoản môi giới (acct), cổ phiếu bạn muốn mua ở định dạng CUSIP (Ủy ban về Thủ tục nhận dạng bảo mật thống nhất) (cusip) và cuối cùng là số lượng cổ phiếu (cổ phiếu), phải là lớn hơn 100. Cả ba trường đều bắt buộc.

Đoạn mã thực hiện yêu cầu mua 1000 cổ phiếu Apple (CUSIP 037833100) đối với tài khoản môi giới 12345 sử dụng REST sẽ trông như thế này:

```
ĐĂNG /giao dịch/mua
Chấp nhận: application/json
{ "acct": "12345",
   "cusip": "037833100",
   "shares": "1000" }
```

Bây giờ, giả sử dịch vụ thay đổi hợp đồng để chấp nhận SEDOL (Danh sách chính thức hàng ngày của Sở giao dịch chứng khoán) thay vì CUSIP, đây là một cách tiêu chuẩn khác của ngành để xác định một công cụ cụ thể sẽ được giao dịch. Bây giờ hợp đồng trông như thế này:

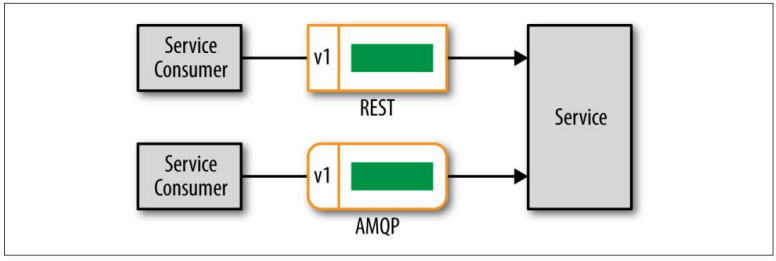
```
{
     "$schema": "http://json-schema.org/draft-04/schema#", "properties":
     { "acct": {"type":
        "number"}, "sedol": {"type ":
        "chuỗi"}, "chia sẻ": {"type": "số",
        "tối thiểu": 100}

    }, "bắt buộc": ["acct", "sedol", "chia sẻ"]
}
```

Đây sẽ được coi là một thay đổi đột phá vì mã máy khách trước đây sẽ không thành công vì nó vẫn đang sử dụng CUSIP. Những gì bạn cần làm là sử dụng phiên bản để phiên bản 1 sử dụng CUSIP và phiên bản 2 sử dụng SEDOL để xác định cổ phiếu đang được giao dịch.

Phiên bản tiêu đề

Kỹ thuật đầu tiên để tạo phiên bản hợp đồng là đặt số phiên bản hợp đồng vào tiêu đề của giao thức truy cập từ xa như minh họa trong Hình 8-1. Tôi muốn gọi đây là phiên bản hợp đồng nhận biết giao thức vì thông tin về phiên bản hợp đồng bạn đang sử dụng được chứa trong tiêu đề của giao thức truy cập từ xa (ví dụ: REST, SOAP, AMQP, JMS, MSMQ, v.v.) .



Hình 8-1. Phiên bản hợp đồng tiêu đề

Khi sử dụng REST, bạn có thể sử dụng loại mime của nhà cung cấp để chỉ định phiên bản hợp đồng bạn muốn sử dụng trong tiêu đề chấp nhận của yêu cầu:

```
POST /trade/buy
Chấp nhận: application/vnd.svc.trade.v2+json
```

Bằng cách sử dụng loại mime của nhà cung cấp (vnd) trong tiêu đề chấp nhận của URI, bạn có thể chỉ định số phiên bản của hợp đồng, từ đó hướng dẫn dịch vụ thực hiện xử lý dựa trên số phiên bản hợp đồng đó. Tương ứng, dịch vụ sẽ cần phân tích tiêu đề chấp nhận để xác định số phiên bản. Một ví dụ về điều này là sử dụng biểu thức chính quy để tìm phiên bản như minh họa bên dưới:

```
phiên bản def
    request.headers ["Chấp
    nhận"][/^application/vnd.svc.trade.v(d)/, 1].to_i end
```

Thật không may đó là phần dễ dàng; phần khó khăn là mã hóa tất cả độ phức tạp theo chu kỳ vào dịch vụ để cung cấp quá trình xử lý có điều kiện dựa trên phiên bản hợp đồng (ví dụ: nếu phiên bản 1 thì... nếu không thì nếu phiên bản 2 thì...). Vì lý do này, chúng tôi cần một số loại chính sách ngừng sử dụng phiên bản để kiểm soát mức độ phức tạp theo chu kỳ mà bạn đưa vào mỗi dịch vụ.

Khi sử dụng tính năng nhắn tin, bạn sẽ cần cung cấp số phiên bản trong phần thuộc tính của tiêu đề thư. Đối với JMS 2.0 nó sẽ trông giống như thế này:

```
Chuỗi msg = createJSON( "acct","12345",
   "sedol","2046251",
   "shares","1000")};
```

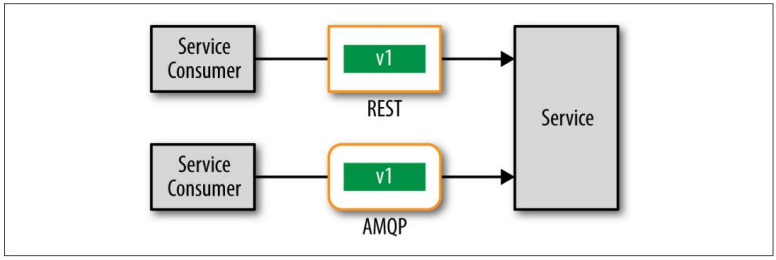
```
jmsContext.createProducer() .setProperty("version", 2) .send(queue, msg);
```

Mỗi tiêu chuẩn nhắn tin sẽ có cách thiết lập tiêu đề này riêng. Điều quan trọng cần nhớ ở đây là bất kể tiêu chuẩn nhắn tin là gì, thuộc tính phiên bản là một giá trị chuỗi cần khớp chính xác với những gì dịch vụ mong đợi, bao gồm cả phân biệt chữ hoa chữ thường. Vì lý do này, thông thường không nên cung cấp phiên bản mặc định nếu không thể tìm thấy số phiên bản trong tiêu đề.

Phiên bản lược đồ

Một kỹ thuật tạo phiên bản hợp đồng khác là thêm số phiên bản vào chính lược đồ thực tế. Kỹ thuật này được minh họa trong Hình 8-2. Tôi thường gọi kỹ thuật này là khái niệm bất khả tri về giao thức.

lập phiên bản đường truyền vì việc nhận dạng phiên bản hoàn toàn độc lập với giao thức truy cập từ xa. Không cần chỉ định gì trong tiêu đề của giao thức truy cập từ xa để sử dụng phiên bản.



Hình 8-2. Phiên bản hợp đồng dựa trên lược đồ

Bằng cách sử dụng lập phiên bản dựa trên lược đồ, lược đồ được sử dụng trong ví dụ trước sẽ trông như thế này:

```
{
     "$schema": "http://json-schema.org/draft-04/schema#", "thuộc tính":
     {
        "phiên bản": {"type": "số nguyên"},
        "acct": {"type": "number"},
        "cusip": {"type": "string"},
        "sedol": {"type" : "chuỗi"}, "chia
        sẻ": {"type": "số", "tối thiểu": 100}

    }, "bắt buộc": ["phiên bản", "tài khoản", "chia sẻ"]
}
```

Lưu ý rằng lược đồ thực sự chứa trường số phiên bản (phiên bản) dưới dạng giá trị số nguyên. Vì hiện tại bạn chỉ có một lược đồ nên bạn sẽ cần thêm tất cả các kết hợp khả năng vào lược đồ. Trong ví dụ trên, cả CUSIP và SEDOL đều được thêm vào lược đồ vì đó là điểm khác nhau giữa các phiên bản.

Ưu điểm lớn của kỹ thuật này là lược đồ (bao gồm cả phiên bản) độc lập với giao thức truy cập từ xa. Điều này có nghĩa là cùng một lược đồ chính xác có thể được sử dụng bởi nhiều giao thức. Ví dụ: REST và JMS 2.0 có thể sử dụng cùng một lược đồ mà không cần bất kỳ sửa đổi nào đối với các tiêu đề giao thức truy cập từ xa:

```
ĐĂNG /giao dịch/mua
Chấp nhận: application/json
{ "version": "2",
   "tài khoản": "12345",
```

```
   "sedol": "2046251",
   "chia sẻ": "1000" }


Chuỗi msg =


   createJSON( "version","2",
   "acct","12345",
"sedol","2046251", "shares","1000")}; jmsContext.createProducer().send(queue, msg);
```

Thật không may, kỹ thuật này có rất nhiều nhược điểm liên quan đến nó. Trước tiên, bạn phải phân tích tải trọng thực tế của tin nhắn để trích xuất số phiên bản. Điều này ngăn cản việc sử dụng những thứ như các thiết bị XML (ví dụ: DataPower) để thực hiện định tuyến và cũng có thể gây ra sự cố khi cố gắng phân tích cú pháp lược đồ (đặc biệt là với XML).

Thứ hai, các lược đồ có thể khá phức tạp, gây khó khăn cho việc chuyển đổi tự động lược đồ (ví dụ: đối tượng JSON sang Java).

Cuối cùng, dịch vụ có thể yêu cầu xác thực tùy chỉnh để xác thực lược đồ. Trong ví dụ trên, dịch vụ sẽ phải xác thực rằng CUSIP hoặc SEDOL được điền dựa trên số phiên bản.

CHƯƠNG 9

Chúng ta có ở đó chưa? Cạm bẫy

Với kiến trúc vi dịch vụ, mọi dịch vụ đều được triển khai dưới dạng một ứng dụng riêng biệt, nghĩa là tất cả giao tiếp với dịch vụ vi mô từ máy khách hoặc lớp API, cũng như giao tiếp giữa các dịch vụ, đều yêu cầu thực hiện lệnh gọi từ xa.

Cạm bẫy này xảy ra khi bạn không biết cuộc gọi truy cập từ xa kéo dài bao lâu. Bạn có thể cho rằng độ trễ là khoảng 50 mili giây, nhưng bạn đã bao giờ đo nó chưa? Bạn có biết độ trễ trung bình đối với môi trường cụ thể của bạn là bao nhiêu không? Bạn có biết độ trễ "đuôi dài" là bao nhiêu (ví dụ: 95, 99, 99,5 phần trăm) đối với môi trường của bạn không? Việc đo lường cả hai số liệu này đều quan trọng, bởi vì ngay cả với độ trễ trung bình tốt, độ trễ dài có thể hủy hoại bạn.

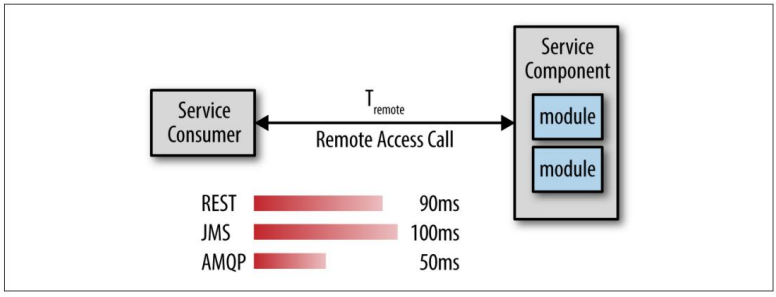
Đo độ trễ

Việc đo độ trễ truy cập từ xa khi đang tải trong môi trường sản xuất của bạn (hoặc môi trường giống như môi trường sản xuất) là rất quan trọng để hiểu được hồ sơ hiệu suất của ứng dụng của bạn. Ví dụ: giả sử một yêu cầu kinh doanh cụ thể yêu cầu sự phối hợp của bốn vi dịch vụ. Giả sử độ trễ truy cập từ xa của bạn là 100 mili giây, yêu cầu công việc cụ thể đó sẽ tiêu tốn 500 mili giây chỉ tính riêng độ trễ truy cập từ xa (yêu cầu ban đầu cộng với bốn cuộc gọi từ xa giữa các dịch vụ). Đó là nửa giây thời gian yêu cầu mà không có một dòng mã nguồn nào được thực thi để xử lý yêu cầu kinh doanh thực tế. Hầu hết các ứng dụng đơn giản là không thể hấp thụ được loại độ trễ đó.

Bạn có thể nghĩ cách để tránh cạm bẫy này chỉ đơn giản là đo và xác định độ trễ trung bình cho giao thức truy cập từ xa đã chọn (ví dụ: REST). Tuy nhiên, điều đó chỉ cung cấp cho bạn một thông tin—độ trễ trung bình của giao thức truy cập từ xa cụ thể mà bạn đang sử dụng. Nhiệm vụ khác là điều tra độ trễ so sánh bằng cách sử dụng các giao thức truy cập từ xa khác như Dịch vụ tin nhắn Java (JMS), Giao thức xếp hàng tin nhắn nâng cao (AMQP) và Hàng đợi tin nhắn của Microsoft (MSMQ).

So sánh các giao thức

Độ trễ so sánh sẽ khác nhau rất nhiều dựa trên cả môi trường của bạn và tính chất của yêu cầu công việc, vì vậy điều quan trọng là phải thiết lập các điểm chuẩn này cho nhiều yêu cầu công việc khác nhau với các cấu hình tải khác nhau.



Hình 9-1. So sánh độ trễ truy cập từ xa

Khi xem ví dụ giả định trong Hình 9-1, bạn nhận thấy rằng AMQP trên thực tế nhanh gần gấp đôi REST. Giờ đây, bạn có thể tận dụng thông tin này để đưa ra những lựa chọn thông minh về việc yêu cầu nào sẽ sử dụng giao thức truy cập từ xa nào. Ví dụ: bạn có thể chọn sử dụng REST cho tất cả giao tiếp từ yêu cầu của khách hàng đến vi dịch vụ và AMQP cho tất cả giao tiếp giữa các dịch vụ để tăng hiệu suất trong ứng dụng của bạn.

Hiệu suất không phải là yếu tố duy nhất được cân nhắc khi chọn giao thức truy cập từ xa của bạn. Như bạn sẽ thấy trong Chương 10, bạn có thể muốn tận dụng tính năng nhắn tin để cung cấp các khả năng bổ sung cho ứng dụng của mình.

CHƯƠNG 10

Hãy cho nó một cạm bẫy nghỉ ngơi

Sử dụng REST cho đến nay là lựa chọn phổ biến nhất để truy cập các dịch vụ vi mô và liên lạc giữa các dịch vụ. Đây là một lựa chọn phổ biến đến mức hầu hết các khung mẫu phổ biến (ví dụ: DropWizard, Spring Boot, v.v.) đều có quyền truy cập REST đã được tích hợp sẵn trong các mẫu dịch vụ. Nếu REST là một lựa chọn phổ biến như vậy thì tại sao nó lại là một cạm bẫy? Cạm bẫy khiến nó ngừng hoạt động là việc sử dụng REST làm giao thức giao tiếp duy nhất và bỏ qua sức mạnh của việc nhắn tin để nâng cao kiến trúc vi dịch vụ của bạn. Ví dụ: trong kiến trúc vi dịch vụ RESTful, bạn sẽ xử lý giao tiếp không đồng bộ như thế nào? Còn nhu cầu về khả năng phát sóng thì sao?

Bạn sẽ làm gì nếu cần quản lý nhiều lệnh gọi RESTful từ xa trong một đơn vị công việc giao dịch?

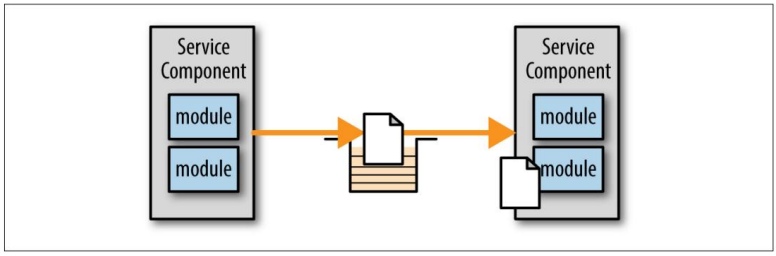
Có hai loại tiêu chuẩn nhắn tin mà bạn cần lưu ý khi cân nhắc sử dụng tính năng nhắn tin cho kiến trúc vi dịch vụ của mình—tiêu chuẩn dành riêng cho nền tảng và tiêu chuẩn độc lập với nền tảng. Các tiêu chuẩn dành riêng cho nền tảng bao gồm JMS cho nền tảng Java và MSMQ cho nền tảng .NET. Cả hai đều mô tả một API tiêu chuẩn được sử dụng trong nền tảng, độc lập với nhà cung cấp dịch vụ nhắn tin (nhà cung cấp) mà bạn đang sử dụng. Ví dụ: trong nền tảng Java, bạn có thể trao đổi các nhà môi giới (ví dụ: ActiveMQ, HornetQ, v.v.) mà không cần thay đổi API. Mặc dù API là tiêu chuẩn và vẫn giữ nguyên, nhưng giao thức độc quyền cơ bản giữa các nhà môi giới này lại khác nhau (đó là lý do tại sao bạn cần có cùng JAR máy khách và JAR máy chủ cho cùng một nhà cung cấp). Với các giao thức nhắn tin tiêu chuẩn nền tảng, bạn quan tâm nhiều hơn đến tính di động thông qua một giao thức chung

API chứ không phải sản phẩm thực tế của nhà cung cấp mà bạn đang sử dụng hoặc các giao thức cấp dây được sử dụng.

Tiêu chuẩn độc lập với nền tảng hiện tại là AMQP. AMQP, tiêu chuẩn hóa giao thức cấp dây chứ không phải API. Điều này cho phép các nền tảng không đồng nhất giao tiếp với nhau, bất kể bạn đang sử dụng sản phẩm của nhà cung cấp nào. Ví dụ: một máy khách sử dụng Rab-bitMQ có thể dễ dàng giao tiếp với máy chủ StormMQ (giả sử họ đang sử dụng cùng một phiên bản giao thức). AMQP sử dụng RabbitMQ hiện là lựa chọn phổ biến nhất để nhắn tin trong kiến trúc vi dịch vụ, chủ yếu là do tính chất độc lập với nền tảng của nó.

Yêu cầu không đồng bộ

Việc cân nhắc đầu tiên khi sử dụng tính năng nhắn tin trong kiến trúc vi dịch vụ của bạn là giao tiếp không đồng bộ. Với các yêu cầu không đồng bộ, người gọi dịch vụ không cần đợi phản hồi từ dịch vụ khi thực hiện yêu cầu, như minh họa trong Hình 10-1. Điều này đôi khi được gọi là quá trình "bắn và quên".

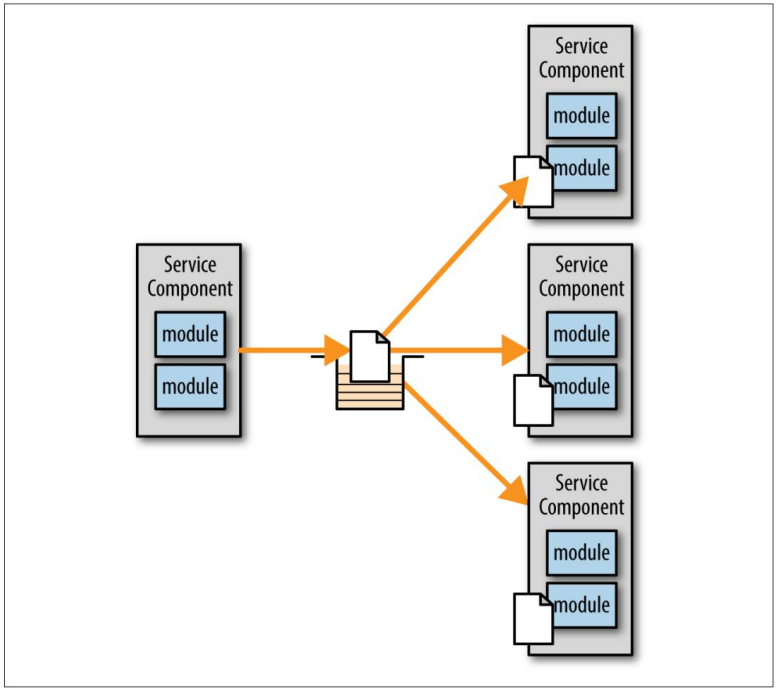


Hình 10-1. Truyền thông không đồng bộ sử dụng tin nhắn

Quá trình xử lý không đồng bộ không chỉ làm tăng hiệu suất tổng thể mà còn tăng thêm yếu tố độ tin cậy cho hệ thống của bạn. Hiệu suất được tăng lên vì người gọi không phải đợi phản hồi nếu không cần thiết. Thông qua việc gửi tin nhắn được đảm bảo, người môi giới tin nhắn đảm bảo rằng dịch vụ cuối cùng sẽ nhận được tin nhắn. Độ tin cậy được tăng lên vì người gọi không cần phải lo lắng về việc đặt giá trị thời gian chờ hoặc sử dụng mẫu ngắt mạch khi liên lạc với một dịch vụ (xem Chương 2).

Khả năng phát sóng

Một tính năng nhắn tin rất mạnh mẽ khác không có trong REST là khả năng truyền phát tin nhắn tới nhiều dịch vụ. Điều này được biết đến trong nhắn tin là tin nhắn "xuất bản và đăng ký" và thường liên quan đến các chủ đề và người đăng ký (tùy thuộc vào tiêu chuẩn nhắn tin bạn đang sử dụng). Hình 10-2 minh họa hoạt động cơ bản của tin nhắn quảng bá.

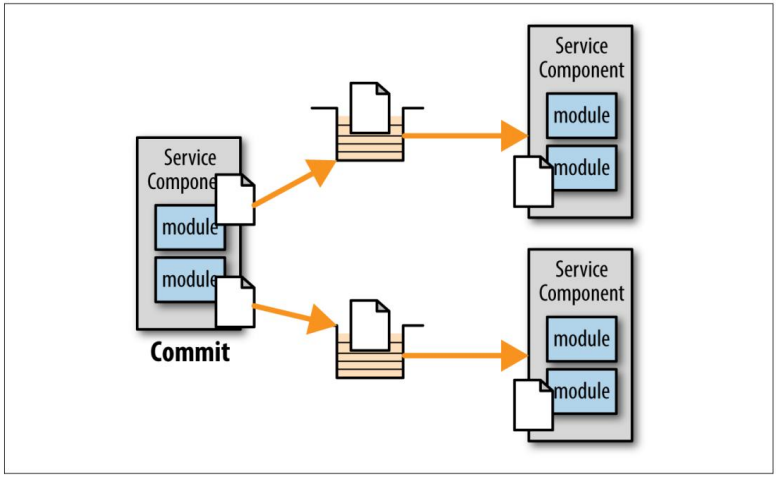


Hình 10-2. Khả năng phát sóng bằng tin nhắn

Tin nhắn quảng bá liên quan đến việc người tạo tin nhắn gửi cùng một tin nhắn đến nhiều người nhận tin nhắn (tức là các dịch vụ). Người tạo tin nhắn thường không biết ai đang chấp nhận tin nhắn hoặc họ sẽ làm gì với nó. Ví dụ: một nhà sản xuất tin nhắn có thể phát đi một tin nhắn thông báo cho người tiêu dùng về việc chia tách cổ phiếu của cổ phiếu Apple (AAPL). Người tạo tin nhắn chỉ có trách nhiệm xuất bản tin nhắn đến một chủ đề (JMS), một fanout hoặc trao đổi chủ đề (AMQP) hoặc hàng đợi phát đa hướng (MSMQ). Thông báo chia cổ phiếu có thể được nhiều người tiêu dùng chấp nhận hoặc không có người tiêu dùng nào cả.

Yêu cầu đã giao dịch

Hệ thống nhắn tin hỗ trợ khái niệm tin nhắn được giao dịch, nghĩa là nếu tin nhắn được gửi đến nhiều hàng đợi hoặc chủ đề trong bối cảnh giao dịch thì các tin nhắn sẽ không thực sự được dịch vụ nhận cho đến khi người gửi thực hiện cam kết với giao dịch đó. Người sử dụng dịch vụ gửi một tin nhắn đến dịch vụ đầu tiên và sau đó gửi một tin nhắn khác đến dịch vụ thứ hai, như minh họa trong Hình 10-3. Cho đến khi người sử dụng dịch vụ thực hiện một cam kết, những tin nhắn đó sẽ được giữ trong hàng đợi. Sau khi người sử dụng dịch vụ thực hiện một cam kết, cả hai thông báo sẽ được phát hành.



Hình 10-3. Khả năng giao dịch của tin nhắn

Nếu người tiêu dùng dịch vụ trong Hình 10-3 gửi tin nhắn đến hàng đợi đầu tiên, nhưng sau đó gặp phải một số loại lỗi, thì người tiêu dùng dịch vụ có thể thực hiện khôi phục giao dịch nhắn tin, điều này sẽ loại bỏ tin nhắn khỏi hàng đợi đầu tiên một cách hiệu quả.

Việc triển khai loại khả năng giao dịch này bằng REST sẽ rất khó khăn, về cơ bản yêu cầu người tiêu dùng dịch vụ đưa ra các yêu cầu đền bù để đảo ngược các cập nhật được thực hiện theo mỗi yêu cầu.
Do đó, bạn nên cân nhắc sử dụng tin nhắn đã giao dịch bất kỳ lúc nào người tiêu dùng dịch vụ cần sắp xếp nhiều yêu cầu từ xa.

Về tác giả

Mark Richards là một kiến trúc sư phần mềm thực hành, giàu kinh nghiệm, tham gia vào kiến trúc, thiết kế và triển khai các kiến trúc dịch vụ vi mô, kiến trúc hướng dịch vụ và các hệ thống phân tán trong J2EE và các công nghệ khác. Ông làm việc trong ngành phần mềm từ năm 1983 và có nhiều kinh nghiệm cũng như kiến thức chuyên môn về ứng dụng, tích hợp và kiến trúc doanh nghiệp.

Mark từng là chủ tịch của Nhóm người dùng Java New England từ năm 1999 đến năm 2003. Ông là tác giả của nhiều sách và video kỹ thuật, bao gồm Chuỗi video Cơ bản về Kiến trúc Phần mềm (video O'Reilly), Tin nhắn Doanh nghiệp (video O'Reilly) , Dịch vụ tin nhắn Java, Phiên bản thứ 2 (O'Reilly) và là tác giả đóng góp cho 97 bài viết Mọi kiến trúc sư phần mềm nên biết (O'Reilly). Mark có bằng thạc sĩ về khoa học máy tính và có nhiều chứng chỉ về kiến trúc cũng như nhà phát triển từ IBM, Sun, The Open Group và BEA. Ông là diễn giả hội nghị thường xuyên tại Chuỗi hội nghị chuyên đề No Fluff Just Stuff (NFJS) và đã phát biểu tại hơn 100 hội nghị và nhóm người dùng trên khắp thế giới về nhiều chủ đề kỹ thuật liên quan đến doanh nghiệp. Khi không làm việc, người ta thường thấy Mark đang dạy các lớp kiến trúc cơ bản và đi bộ đường dài ở White Mountains ở New Hampshire và dọc theo Đường mòn Appalachian.