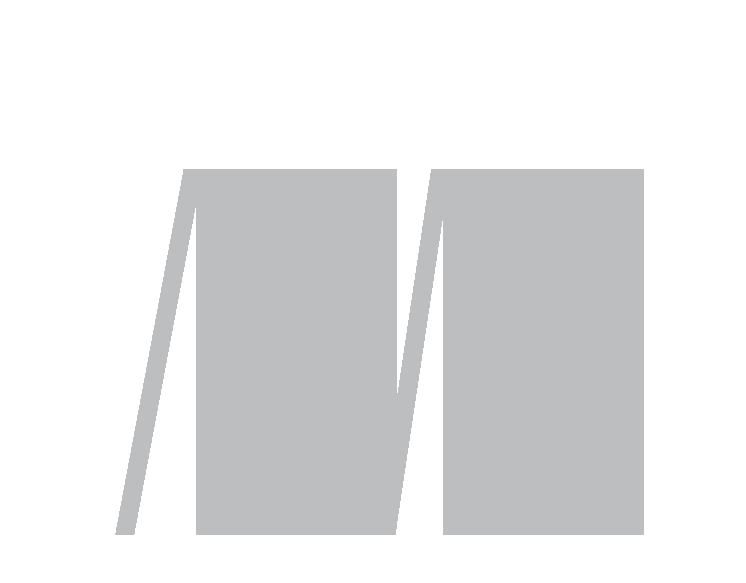
 [Translated from English to Vietnamese - www.onlinedoctranslator.com](https://www.onlinedoctranslator.com/en/?utm_source=onlinedoctranslator&utm_medium=docx&utm_campaign=attribution)



ChrisRichardson

## NGƯỜI ĐÀN ÔNG

#### Danh sáchcủa các mẫu

###### Các mẫu kiến ​​trúc ứng dụng

Kiến trúc nguyên khối (40)

Dịch vụ vi môkiến trúc (40)

###### Mẫu phân hủy

Phân tích theo năng lực kinh doanh (51) Phân tích theo tên miền phụ (54)

###### Mẫu phong cách nhắn tin

Nhắn tin (85)

Gọi thủ tục từ xa (72)

###### Mẫu giao tiếp đáng tin cậy

Cầu dao điện (78)

###### Các mẫu khám phá dịch vụ

Đăng ký bên thứ 3 (85) Khám phá phía máy khách (83)

Tự đăng ký (82)

Phát hiện phía máy chủ (85)

###### Mẫu tin nhắn giao dịch

Nhà xuất bản thăm dò (98) Theo dõi nhật ký giao dịch (99) Hộp thư đi giao dịch (98)

###### Mẫu thống nhất dữ liệu

Truyện dài (114)

###### Các mẫu thiết kế logic kinh doanh

Tổng hợp (150)

Sự kiện miền (160)

Mô hình miền (150)

Nguồn sự kiện(184)

Kịch bản giao dịch (149)

###### Mẫu truy vấn

Thành phần API(223)

Phân tách trách nhiệm truy vấn lệnh (228)

###### Mẫu API bên ngoài

Cổng API (259)

Backend cho frontend (265)

###### Mẫu thử nghiệm

Kiểm tra hợp đồng do người tiêu dùng thúc đẩy (302) Kiểm tra hợp đồng phía người tiêu dùng (303) Kiểm tra thành phần dịch vụ (335)

###### Mẫu bảo mật

Mã thông báo truy cập (354)

###### Các mẫu quan tâm cắt ngang

Cấu hình bên ngoài (361)

Khung máy vi dịch vụ (379)

###### Các mẫu quan sát được

Ứng dụngsố liệu (373)

Kiểm toánkhai thác gỗ (377)

Phân phối theo dõi (370)

Theo dõi ngoại lệ (376) API kiểm tra tình trạng (366) Tổng hợp nhật ký (368)

###### Các mẫu triển khai

Triển khai một dịch vụ như mộtcontainer (393) Triển khai dịch vụ dưới dạng VM (390)

Định dạng đóng gói theo ngôn ngữ cụ thể (387) Lưới dịch vụ (380)

Triển khai không cần máy chủ (416)

Xe đẩy có thùng (410)

###### Tái cấu trúc thành các mẫu dịch vụ vi mô

Lớp chống tham nhũng (447)

Ứng dụng siết cổ (432)

*Các mẫu dịch vụ vi mô*

*Dịch vụ vi mô Các mẫu*

**TITH VÍ DỤ TRONGJAVA**

CNhân sựRICHARDSON

NGƯỜI ĐÀN ÔNGG

SHÃNGTÔIĐẤT

Để biết thông tin trực tuyến và đặt mua sách này và các sách khác của Manning, vui lòng truy cập[www.manning.com](http://www.manning.com/). Nhà xuất bảncung cấp giảm giá cho cuốn sách này khi đặt hàng với số lượng lớn. Để biết thêm thông tin, vui lòng liên hệ

Phòng Bán Hàng Đặc Biệt Manning Publications Co. 20 Baldwin Road

Hộp thư bưu điện 761

Đảo Shelter,NY 11964 Email:[đơn hàng@manning.com](mailto:orders@manning.com)

©2019 của Chris Richardson. Bảo lưu mọi quyền.

Không được sao chép, lưu trữ trong hệ thống truy xuất hoặc truyền tải bất kỳ phần nào của ấn phẩm này dưới bất kỳ hình thức nào hoặc bằng phương tiện điện tử, cơ học, photocopy hoặc cách khác mà không có sự cho phép trước bằng văn bản của nhà xuất bản.

Nhiều tên gọi do nhà sản xuất và người bán sử dụng để phân biệt sản phẩm của họ được coi là nhãn hiệu. Khi những tên gọi đó xuất hiện trong sách và Manning Publications biết về khiếu nại về nhãn hiệu, thì những tên gọi đó được in bằng chữ in hoa đầu hoặc toàn bộ chữ in hoa.

Nhận ra tầm quan trọng của việc bảo tồn những gì đã được viết, chính sách của Manningđể các cuốn sách chúng tôi xuất bản được in trên giấy không axit và chúng tôi nỗ lực hết mình vì mục đích đó.

Nhận thức được trách nhiệm bảo tồn tài nguyên của hành tinh, sách Manning được in trên loại giấy được tái chế và xử lý ít nhất 15 phần trăm mà không sử dụng clo nguyên tố.

Xuất bản ManningCông ty Biên tập viên phát triển: Marina Michaels

20 BaldwinĐường Biên tập viên phát triển kỹ thuật: Christian Mennerich

Hộp thư bưu điện761 Biên tập viên đánh giá: Aleksandar Dragosavljevic´

Đảo Shelter, New York11964 Biên tập dự án: Lori Weidert Biên tập bản sao: Corbin Collins Người hiệu đính: Alyson Brener

Biên tập viên kỹ thuật: Andy Miles

Người sắp chữ: Dennis Dalinnik Thiết kế bìa: Marija Tudor

Mã số: 9781617294549

Được in tại Hoa Kỳ

1 2 3 4 5 6 7 89 10 – DP – 23 22 21 20 19 18

*Khi bạn thấy điều sai trái, bất bình đẳng hay bất công, hãy lên tiếng, vì đây là đất nước của bạn.*

*Đây là nền dân chủ của bạn. Hãy tạo ra nó. Hãy bảo vệ nó. Hãy truyền bá nó.*

— Thurgood Marshall, Thẩm phán Tòa án Tối cao

# nội dung tóm tắt

##### ■ [Thoát khỏi khối thống nhấtđịa ngục1](#_bookmark4)

1. ■ [Phân hủychiến lược33](#_bookmark177)

##### ■ [Giao tiếp giữa các tiến trình trong một dịch vụ vi mô](#_bookmark287) [kiến trúc65](#_bookmark287)

1. ■ [Quản lý giao dịch vớisagas110](#_bookmark473)

##### ■ [Thiết kế logic kinh doanh trong một dịch vụ vi mô](#_bookmark571) [kiến trúc146](#_bookmark571)

1. ■ [Phát triển logic kinh doanh với sự kiệnnguồn cung cấp183](#_bookmark677)

##### ■ [Triển khai truy vấn trong một dịch vụ vi môkiến trúc220](#_bookmark807)

1. ■ [API bên ngoàimẫu253](#_bookmark917)

##### ■ [Kiểm tra các dịch vụ vi mô: Phần1292](#_bookmark1043)

1. ■ [Kiểm tra các dịch vụ vi mô: Phần2318](#_bookmark1121)

##### ■ [Phát triển sản xuất sẵn sàngdịch vụ348](#_bookmark1179)

1. ■ [Triển khaidịch vụ vi mô383](#_bookmark1331)

##### ■ [Tái cấu trúc thànhdịch vụ vi mô428](#_bookmark1504)

**vii**

# nội dung

[*Lời nói đầu xvii*](#_bookmark0)[*lời cảm ơnxx*](#_bookmark1)[*về cuốn sách này xxii*](#_bookmark2)

[*về hình minh họa bìaxxvi*](#_bookmark3)

[***Thoát khỏi khối thống nhấtđịa ngục 1***](#_bookmark4)

[*1*](#_bookmark4)

##### [Cuộc hành quân chậm chạp hướng tới sự thống nhấtđịa ngục2](#_bookmark4)

[*Kiến trúc của ứng dụng FTGO 3*](#_bookmark12)■[*Những lợi ích của*](#_bookmark19)[*kiến trúc nguyên khối 4*](#_bookmark19)■[*Sống trong địa ngục nguyên khối 4*](#_bookmark21)

##### [Tại sao cuốn sách này có liên quan đến bạn 7](#_bookmark40)

* 1. [Những gì bạn sẽ học được trong nàyquyển 7](#_bookmark41)

##### [Kiến trúc vi dịch vụ chocứu8](#_bookmark42)

[*Scale cube và microservices 8*](#_bookmark49)■[*Các dịch vụ vi mô như một hình thức*](#_bookmark57)[*tính mô-đun 11*](#_bookmark57)■[*Mỗi dịch vụ có cơ sở dữ liệu riêng 12*](#_bookmark61)[*Kiến trúc dịch vụ vi mô FTGO 12*](#_bookmark63)■[*So sánh các*](#_bookmark66)[*Kiến trúc microservice và SOA 13*](#_bookmark66)

##### [Lợi ích và hạn chế của dịch vụ vi môkiến trúc14](#_bookmark72)

[*Lợi ích của kiến ​​trúc microservice 14*](#_bookmark73)■[*Nhược điểm của*](#_bookmark81)[*Kiến trúc dịch vụ vi mô 17*](#_bookmark81)

**ix**

**x MỤC LỤC**

##### [Mô hình kiến ​​trúc Microservicengôn ngữ19](#_bookmark91)

[*Kiến trúc microservice không phải là giải pháp hoàn hảo 19*](#_bookmark92)■[*Các mẫu và*](#_bookmark95)[*ngôn ngữ mẫu 20*](#_bookmark95)■[*Tổng quan về kiến ​​trúc Microservice*](#_bookmark113)[*ngôn ngữ mẫu 23*](#_bookmark113)

##### [Ngoài các dịch vụ vi mô: Quy trình vàtổ chức29](#_bookmark158)

[*Tổ chức phát triển và cung cấp phần mềm 29*](#_bookmark160)■[*Phần mềm*](#_bookmark165)[*quá trình phát triển và cung cấp 30*](#_bookmark165)■[*Mặt con người của*](#_bookmark173)[*áp dụng microservices 31*](#_bookmark173)

[*2*](#_bookmark182)

[***Phân hủychiến lược 33***](#_bookmark177)

##### [Kiến trúc microservice là gì?chính xác?34](#_bookmark182)

[*Kiến trúc phần mềm là gì và tại sao nó lại quan trọng? 34*](#_bookmark187)[*Tổng quan về phong cách kiến ​​trúc 37*](#_bookmark202)■ [*Kiến trúc dịch vụ vi mô*](#_bookmark218)[*là một phong cách kiến ​​trúc 40*](#_bookmark218)

##### [Xác định dịch vụ vi mô của ứng dụngkiến trúc44](#_bookmark232)

[*Xác định hoạt động của hệ thống 45*](#_bookmark235)■[*Định nghĩa dịch vụ theo*](#_bookmark241)[*áp dụng mô hình năng lực Phân tích theo doanh nghiệp 51*](#_bookmark241)[*Xác định các dịch vụ bằng cách áp dụng Phân tách theo miền phụ*](#_bookmark251)[*mẫu 54*](#_bookmark251)■[*Hướng dẫn phân hủy 56*](#_bookmark258)■[*Những trở ngại đối với*](#_bookmark268)[*phân tích một ứng dụng thành các dịch vụ 57*](#_bookmark268)■[*Định nghĩa dịch vụ*](#_bookmark276)[*API 61*](#_bookmark276)

[*3*](#_bookmark287)

[***Giao tiếp giữa các tiến trình trong một dịch vụ vi môngành kiến ​​​​trúc 65***](#_bookmark287)

##### [Tổng quan về giao tiếp giữa các tiến trình trong một dịch vụ vi mô](#_bookmark287) [kiến trúc66](#_bookmark289)

[*Phong cách tương tác 67*](#_bookmark291)■[*Định nghĩa API trong một dịch vụ vi mô*](#_bookmark298)[*kiến trúc 68*](#_bookmark298)■[*API phát triển 69*](#_bookmark304)■[*Định dạng tin nhắn 71*](#_bookmark313)

##### [Giao tiếp bằng cách sử dụng lệnh gọi thủ tục từ xa đồng bộmẫu72](#_bookmark324)

[*Sử dụng REST 73*](#_bookmark327)■[*Sử dụng gRPC 76*](#_bookmark335)■[*Xử lý lỗi một phần*](#_bookmark341)[*sử dụng mẫu cầu dao điện 77*](#_bookmark341)■[*Sử dụng dịch vụ khám phá 80*](#_bookmark354)

##### [Giao tiếp bằng cách sử dụng tin nhắn không đồng bộmẫu85](#_bookmark372)

[*Tổng quan về tin nhắn 86*](#_bookmark375)■[*Thực hiện các phong cách tương tác*](#_bookmark388)[*sử dụng tin nhắn 87*](#_bookmark388)■[*Tạo một đặc tả API cho một*](#_bookmark401)[*dịch vụ dựa trên tin nhắn API 89*](#_bookmark401)■[*Sử dụng một nhà môi giới tin nhắn 90*](#_bookmark408)[*Người nhận cạnh tranh và thứ tự tin nhắn94*](#_bookmark424)■[*Xử lý*](#_bookmark429)[*tin nhắn trùng lặp 95*](#_bookmark429)■[*Tin nhắn giao dịch 97*](#_bookmark434)[*Thư viện và khuôn khổ cho nhắn tin 100*](#_bookmark451)

##### [Sử dụng nhắn tin không đồng bộ để cải thiệnsự sẵn có 103](#_bookmark458)

[*Giao tiếp đồng bộ làm giảm tính khả dụng 103*](#_bookmark460)[*Loại bỏ tương tác đồng bộ 104*](#_bookmark463)

[***Quản lý giao dịch vớitruyện dài 110***](#_bookmark473)

[*4*](#_bookmark473)

##### [Quản lý giao dịch trong một dịch vụ vi mô](#_bookmark473) [kiến trúc111](#_bookmark481)

[*Nhu cầu về các giao dịch phân tán trong một dịch vụ vi mô*](#_bookmark484)[*kiến trúc 112*](#_bookmark484)■[*Rắc rối với phân phối*](#_bookmark486)[*giao dịch 112*](#_bookmark486)■[*Sử dụng mô hình Saga để duy trì*](#_bookmark494)[*tính nhất quán dữ liệu 114*](#_bookmark494)

##### [Phối hợpsagas117](#_bookmark504)

[*Truyện cổ tích dựa trên vũ đạo 118*](#_bookmark506)■ [*Các saga dựa trên sự phối hợp 121*](#_bookmark517)

##### [Xử lý sự thiếu hụtcô lập126](#_bookmark527)

[*Tổng quan về các bất thường 127*](#_bookmark531)■[*Các biện pháp đối phó để xử lý*](#_bookmark535)[*thiếu sự cô lập 128*](#_bookmark535)

##### [Thiết kế của Dịch vụ Đặt hàng và Tạo Đơn hàngTruyện dài132](#_bookmark551)

[*Lớp OrderService 133*](#_bookmark552)■[*Việc thực hiện Tạo*](#_bookmark555)[*Đặt hàng Saga 135*](#_bookmark555)■[*Lớp OrderCommandHandlers 142*](#_bookmark564)[*Lớp OrderServiceConfiguration 143*](#_bookmark567)

[***Thiết kế logic kinh doanh trong một dịch vụ vi môngành kiến ​​​​trúc 146***](#_bookmark571)

[*5*](#_bookmark571)

##### [Tổ chức logic kinh doanhmẫu hình 147](#_bookmark571)

[*Thiết kế logic kinh doanh bằng cách sử dụng mẫu tập lệnh Giao dịch 149*](#_bookmark578)[*Thiết kế logic kinh doanh bằng cách sử dụngMẫu mô hình miền 150*](#_bookmark586)[*Về Thiết kế theo miền 151*](#_bookmark591)

##### [Thiết kế mô hình miền sử dụng tổng hợp DDDmẫu 152](#_bookmark599)

[*Vấn đề với ranh giới mờ 153*](#_bookmark602)■[*Tổng hợp có*](#_bookmark606)[*ranh giới rõ ràng 154*](#_bookmark606)■[*Quy tắc tổng hợp 155*](#_bookmark611)■[*Tổng hợp*](#_bookmark614)[*độ chi tiết 158*](#_bookmark614)■ [*Thiết kế logic kinh doanh với tổng hợp 159*](#_bookmark616)

##### [Miền xuất bảnsự kiện160](#_bookmark620)

[*Tại sao phải công bố sự kiện thay đổi? 160*](#_bookmark623)■[*Tên miền là gì*](#_bookmark627)

[*sự kiện? 161*](#_bookmark627)■ [*Sự kiệnlàm giàu161*](#_bookmark629) ■[*Xác định miền*](#_bookmark632)[*sự kiện 162*](#_bookmark632)■[*Tạo và xuất bản sự kiện miền 164*](#_bookmark636)[*Tiêu thụ sự kiện miền 167*](#_bookmark645)

##### [Kinh doanh dịch vụ nhà bếplogic168](#_bookmark648)

[*Tổng hợp vé169*](#_bookmark650)

##### [Dịch vụ đặt hàng kinh doanhlogic173](#_bookmark661)

[*Tổng hợp đơn hàng 175*](#_bookmark663)■ [*Lớp OrderService 180*](#_bookmark673)

[*6*](#_bookmark677)

[***Phát triển logic kinh doanh với sự kiệnnguồn cung ứng 183***](#_bookmark677)

##### [Phát triển logic kinh doanh bằng cách sử dụng sự kiệnnguồn cung cấp184](#_bookmark677)

[*Rắc rối với sự kiên trì truyền thống 185*](#_bookmark682)■[*Tổng quan sự kiện*](#_bookmark691)[*nguồn cung ứng 186*](#_bookmark691)■[*Xử lý các bản cập nhật đồng thời bằng cách sử dụng sentiment*](#_bookmark707)[*khóa 193*](#_bookmark707)■ [*Sự kiện tìm nguồn và xuất bản sự kiện 194*](#_bookmark711)[*Sử dụng ảnh chụp nhanh để cải thiện hiệu suất195*](#_bookmark717)■[*Idempotent*](#_bookmark721)[*xử lý tin nhắn 197*](#_bookmark721)■ [*Miền đang phát triểnsự kiện 198*](#_bookmark725)[*Lợi ích của việc tìm nguồn sự kiện 199*](#_bookmark731)■[*Nhược điểm của sự kiện*](#_bookmark739)

[*nguồn cung cấp 200*](#_bookmark739)

##### [Thực hiện một sự kiệncửa hàng202](#_bookmark751)

[*Cửa hàng sự kiện Eventuate Local hoạt động như thế nào 203*](#_bookmark757)■[*Sự kiện kết thúc*](#_bookmark767)[*khung máy khách cho Java 205*](#_bookmark767)

##### [Sử dụng saga và nguồn sự kiệncùng nhau209](#_bookmark782)

[*Triển khai saga dựa trên vũ đạo bằng cách sử dụng nguồn sự kiện 210*](#_bookmark784)[*Tạo ra một câu chuyện dựa trên dàn nhạc211*](#_bookmark786)■ [*Thực hiệnMỘT*](#_bookmark791)[*người tham gia saga dựa trên sự kiện 213*](#_bookmark791)■[*Thực hiện saga*](#_bookmark797)[*người điều phối sử dụng sự kiện tìm nguồn 216*](#_bookmark797)

[*7*](#_bookmark807)

[***Triển khai truy vấn trong một dịch vụ vi môngành kiến ​​​​trúc 220***](#_bookmark807)

##### [Truy vấn bằng cách sử dụng thành phần APImẫu221](#_bookmark807)

[*Hoạt động truy vấn findOrder() 221*](#_bookmark813)■[*Tổng quan về API*](#_bookmark815)[*mẫu thành phần 222*](#_bookmark815)■[*Triển khai truy vấn findOrder()*](#_bookmark821)[*hoạt động sử dụng mẫu thành phần API 224*](#_bookmark821)■[*Giao diện lập trình ứng dụng (API)*](#_bookmark823)[*vấn đề thiết kế thành phần 225*](#_bookmark823)■ [*Những lợi ích và hạn chế*](#_bookmark830)[*của mẫu thành phần API 227*](#_bookmark830)

##### [Sử dụng CQRSmẫu228](#_bookmark838)

[*Động lực sử dụng CQRS 229*](#_bookmark840)■ [*Tổng quan vềCQRS232*](#_bookmark850) [*Lợi ích của CQRS 235*](#_bookmark856)■[*Nhược điểm của CQRS 236*](#_bookmark864)

##### [Thiết kế CQRSlượt xem 236](#_bookmark869)

[*Chọn kho dữ liệu xem237*](#_bookmark871)■ [*Dữ liệuthiết kế module truy cập 239*](#_bookmark874)[*Thêm và cập nhật chế độ xem CQRS 241*](#_bookmark878)

##### [Triển khai chế độ xem CQRS với AWSMáy phát điệnDynamoDB242](#_bookmark881)

[*Mô-đun OrderHistoryEventHandlers 243*](#_bookmark883)

[*Mô hình hóa dữ liệu và thiết kế truy vấn với DynamoDB 244*](#_bookmark886)[*Lớp OrderHistoryDaoDynamoDb 249*](#_bookmark907)

[***API bên ngoàimẫu hình 253***](#_bookmark917)

[*8*](#_bookmark917)

##### [Thiết kế API bên ngoàivấn đề254](#_bookmark917)

[*Các vấn đề thiết kế API cho máy khách di động FTGO 255*](#_bookmark922)■[*Thiết kế API*](#_bookmark926)[*các vấn đề cho các loại khách hàng khác 258*](#_bookmark926)

##### [Cổng APImẫu259](#_bookmark931)

[*Tổng quan về mẫu cổng API 259*](#_bookmark934)■[*Lợi ích và*](#_bookmark965)[*nhược điểm của cổng API 267*](#_bookmark965)■[*Netflix là một ví dụ*](#_bookmark968)[*của một cổng API 267*](#_bookmark968)■[*Các vấn đề thiết kế cổng API 268*](#_bookmark972)

##### [Triển khai APIcổng vào 271](#_bookmark985)

[*Sử dụng cổng API có sẵnsản phẩm/dịch vụ 271*](#_bookmark989)[*Phát triển cổng API của riêng bạn 273*](#_bookmark1000)■[*Thực hiện một*](#_bookmark1016)[*Cổng API sử dụng GraphQL 279*](#_bookmark1016)

[***Kiểm tra các dịch vụ vi mô: Phần1 292***](#_bookmark1043)

[*9*](#_bookmark1043)

##### [Chiến lược thử nghiệm cho dịch vụ vi môkiến trúc 294](#_bookmark1043)

[*Tổng quan về thử nghiệm 294*](#_bookmark1048)■[*Thách thức của việc thử nghiệm*](#_bookmark1064)[*dịch vụ vi mô 299*](#_bookmark1064)■[*Đường ống triển khai 305*](#_bookmark1079)

##### [Viết các bài kiểm tra đơn vị cho mộtdịch vụ 307](#_bookmark1091)

[*Phát triển các bài kiểm tra đơn vị cho các thực thể 309*](#_bookmark1095)■[*Viết các bài kiểm tra đơn vị cho giá trị*](#_bookmark1100)[*đối tượng 310*](#_bookmark1100)■[*Phát triển các bài kiểm tra đơn vị cho sagas 310*](#_bookmark1104)■ [*Viết*](#_bookmark1109)[*kiểm tra đơn vị cho dịch vụ miền 312*](#_bookmark1109)■[*Phát triển các bài kiểm tra đơn vị cho*](#_bookmark1113)[*bộ điều khiển 313*](#_bookmark1113)■[*Viết các bài kiểm tra đơn vị cho sự kiện và tin nhắn*](#_bookmark1118)[*người xử lý 315*](#_bookmark1118)

[***Kiểm tra các dịch vụ vi mô: Phần2 318***](#_bookmark1121)

[*10*](#_bookmark1121)

##### [Tích hợp viếtkiểm tra319](#_bookmark1121)

[*Kiểm tra tích hợp tính bền bỉ 321*](#_bookmark1124)■ [*Kiểm thử tích hợp dựa trên REST*](#_bookmark1127)[*tương tác theo kiểu yêu cầu/phản hồi322*](#_bookmark1127)■ [*Kiểm thử tích hợp*](#_bookmark1134)[*tương tác theo kiểu xuất bản/đăng ký 326*](#_bookmark1134)■ [*Hợp đồng tích hợp*](#_bookmark1142)[*kiểm tra tương tác yêu cầu/phản hồi không đồng bộ 330*](#_bookmark1142)

##### [Thành phần đang phát triểnkiểm tra335](#_bookmark1148)

[*Định nghĩa các bài kiểm tra chấp nhận 336*](#_bookmark1151)■[*Viết các bài kiểm tra chấp nhận bằng cách sử dụng*](#_bookmark1153)[*Dưa chuột 337*](#_bookmark1153)■[*Thiết kế thử nghiệm thành phần 339*](#_bookmark1157)■[*Viết*](#_bookmark1163)[*kiểm tra thành phần cho FTGO Order Service 340*](#_bookmark1163)

##### [Viết từ đầu đến cuốikiểm tra345](#_bookmark1169)

[*Thiết kế các bài kiểm tra đầu cuối 345*](#_bookmark1171)■[*Viết các bài kiểm tra đầu cuối 346*](#_bookmark1173)[*Chạy thử nghiệm đầu cuối 346*](#_bookmark1175)

[***Phát triển sản xuất sẵn sàngdịch vụ 348***](#_bookmark1179)

[*11*](#_bookmark1179)

##### [Đang phát triểndịch vụ an toàn349](#_bookmark1179)

[*Tổng quan về bảo mật trong ứng dụng đơn khối truyền thống 350*](#_bookmark1191)[*Triển khai bảo mật trong dịch vụ vi môkiến trúc 353*](#_bookmark1203)

##### [Thiết kế có thể định cấu hìnhdịch vụ360](#_bookmark1223)

[*Sử dụng cấu hình bên ngoài dựa trên đẩy 362*](#_bookmark1228)■[*Sử dụng kéo-*](#_bookmark1234)[*dựa trên cấu hình bên ngoài 363*](#_bookmark1234)

##### [Thiết kế có thể quan sát đượcdịch vụ364](#_bookmark1239)

[*Sử dụng mẫu API kiểm tra sức khỏe 366*](#_bookmark1247)■[*Áp dụng Nhật ký*](#_bookmark1256)[*mẫu tổng hợp 368*](#_bookmark1256)■[*Sử dụng theo dõi phân tán*](#_bookmark1273)[*mẫu 370*](#_bookmark1273)■[*Áp dụng mẫu số liệu ứng dụng 373*](#_bookmark1283)[*Sử dụng mẫu theo dõi Ngoại lệ 376*](#_bookmark1296)■[*Áp dụng Kiểm toán*](#_bookmark1301)[*mẫu ghi nhật ký 377*](#_bookmark1301)

##### [Phát triển dịch vụ bằng cách sử dụng khung Microservicemẫu378](#_bookmark1311)

[*Sử dụng khung máy vi dịch vụ379*](#_bookmark1315)■[*Từ khung gầm dịch vụ vi mô đến*](#_bookmark1322)[*lưới dịch vụ 380*](#_bookmark1322)

[*12*](#_bookmark1331)

[***Triển khai các dịch vụ vi mô 383***](#_bookmark1331)

##### [Triển khai dịch vụ bằng cách sử dụng gói Ngôn ngữ cụ thể](#_bookmark1331) [mẫu định dạng 386](#_bookmark1333)

[*Lợi ích của Dịch vụ như một mẫu gói ngôn ngữ cụ thể 388*](#_bookmark1337)[*Nhược điểm của Dịch vụ như một gói ngôn ngữ cụ thể*](#_bookmark1342)

[*mẫu 389*](#_bookmark1342)

##### [Triển khai dịch vụ bằng cách sử dụng mẫu Dịch vụ như máy ảo 390](#_bookmark1349)

[*Lợi ích của việc triển khai dịch vụ dưới dạng VM 392*](#_bookmark1356)■[*Những nhược điểm của*](#_bookmark1362)[*triển khai các dịch vụ như VM 392*](#_bookmark1362)

##### [Triển khai dịch vụ bằng cách sử dụng mô hình Dịch vụ như một container 393](#_bookmark1368)

[*Triển khai dịch vụ bằng Docker 395*](#_bookmark1371)■[*Lợi ích của việc triển khai*](#_bookmark1388)[*dịch vụ như container 398*](#_bookmark1388)■[*Nhược điểm của việc triển khai dịch vụ*](#_bookmark1390)[*như container 399*](#_bookmark1390)

##### [Triển khai ứng dụng FTGO vớiKubernetes 399](#_bookmark1393)

[*Tổng quan về Kubernetes 399*](#_bookmark1395)■ [*Triển khai dịch vụ Nhà hàng*](#_bookmark1406)[*trên Kubernetes 402*](#_bookmark1406)■[*Triển khai cổng API 405*](#_bookmark1414)

[*Triển khai không có thời gian chết 406*](#_bookmark1419)■ [*Sử dụng lưới dịch vụ*](#_bookmark1424)[*để tách triển khai khỏi bản phát hành 407*](#_bookmark1424)

##### [Triển khai dịch vụ bằng cách sử dụng triển khai Serverlessmẫu415](#_bookmark1451)

[*Tổng quan về triển khai không cần máy chủ với AWS Lambda 416*](#_bookmark1454)[*Phát triển hàm lambda 417*](#_bookmark1462)■[*Gọi lambda*](#_bookmark1468)[*chức năng 417*](#_bookmark1468)■[*Lợi ích của việc sử dụng hàm lambda 418*](#_bookmark1475)[*Nhược điểm của việc sử dụng hàm lambda 419*](#_bookmark1477)

##### [Triển khai dịch vụ RESTful bằng AWS Lambdavà AWSCổng vào 419](#_bookmark1482)

[*Thiết kế phiên bản AWS Lambda của Nhà hàngDịch vụ 419*](#_bookmark1484)[*Đóng gói dịch vụ dưới dạng tệp ZIP 424*](#_bookmark1498)■[*Triển khai lambda*](#_bookmark1500)[*chức năng sử dụng khung Serverless 425*](#_bookmark1500)

[*13*](#_bookmark1504)

[***Tái cấu trúc thànhdịch vụ vi mô 428***](#_bookmark1504)

##### [Tổng quan về tái cấu trúc đểdịch vụ vi mô429](#_bookmark1504)

[*Tại sao phải tái cấu trúc một khối đơn khối? 429*](#_bookmark1508)■[*Siết chặt khối đá nguyên khối 430*](#_bookmark1512)

##### [Các chiến lược để tái cấu trúc một khối đơn nguyên thànhdịch vụ vi mô433](#_bookmark1524)

[*Triển khai các tính năng mới nhưdịch vụ434*](#_bookmark1526) ■[*Trình bày riêng biệt*](#_bookmark1529)[*tầng từ phía sau 436*](#_bookmark1529)■[*Trích xuất khả năng kinh doanh vào*](#_bookmark1535)[*dịch vụ 437*](#_bookmark1535)

##### [Thiết kế cách thức dịch vụ và khối thống nhấtcộng tác443](#_bookmark1541)

[*Thiết kế keo tích hợp 444*](#_bookmark1543)■[*Duy trì dữ liệu*](#_bookmark1557)[*tính nhất quán trên một dịch vụ và một khối thống nhất 449*](#_bookmark1557)■[*Xử lý*](#_bookmark1572)[*xác thực và ủy quyền 453*](#_bookmark1572)

##### [Triển khai tính năng mới dưới dạng dịch vụ: xử lý giao nhầmđơn hàng455](#_bookmark1578)

[*Thiết kế dịch vụ giao hàng chậm 456*](#_bookmark1580)■[*Thiết kế*](#_bookmark1584)[*keo tích hợp cho Giao hàng chậm trễDịch vụ 457*](#_bookmark1584)

##### [Tách rờikhối đá nguyên khối: trích xuất giao hàngquản lý459](#_bookmark1589)

[*Tổng quan về chức năng quản lý giao hàng hiện có 460*](#_bookmark1591)[*Tổng quan về Giao hàngDịch vụ 462*](#_bookmark1598)■[*Thiết kế dịch vụ giao hàng*](#_bookmark1601)[*mô hình miền 463*](#_bookmark1601)■ [*Thiết kế tích hợp dịch vụ giao hàng*](#_bookmark1607)[*keo dán465*](#_bookmark1607)■ [*Thay đổi khối FTGOđể tương tác với Giao hàng*](#_bookmark1614)[*Dịch vụ 467*](#_bookmark1614)

[*chỉ số473*](#_bookmark1625)

# lời nói đầu

Một trong những câu trích dẫn yêu thích của tôi là

*Tương lai đã ở đây rồi—chỉ là nó không được phân bổ đồng đều mà thôi.*

—William Gibson, tác giả khoa học viễn tưởng

Bản chất của câu trích dẫn đó là những ý tưởng và công nghệ mới cần một thời gian để lan tỏa trong cộng đồng và được áp dụng rộng rãi. Một ví dụ điển hình về sự lan tỏa chậm rãi của các ý tưởng là câu chuyện về cách tôi khám phá ra các dịch vụ vi mô. Mọi chuyện bắt đầu vào năm 2006, khi được truyền cảm hứng từ một bài nói chuyện của một nhà truyền bá AWS, tôi bắt đầu đi theo con đường cuối cùng dẫn đến việc tôi tạo ra Cloud Foundry ban đầu. (Điểm chung duy nhất với Cloud Foundry ngày nay là tên gọi.) Cloud Foundry là Nền tảng dưới dạng dịch vụ (PaaS) để tự động triển khai các ứng dụng Java trên EC2. Giống như mọi ứng dụng Java doanh nghiệp khác mà tôi đã xây dựng, Cloud Foundry của tôi có kiến ​​trúc khối đơn bao gồm một tệp Lưu trữ ứng dụng web Java (WAR) duy nhất.

Việc đóng gói một tập hợp các chức năng đa dạng và phức tạp như cung cấp, cấu hình, giám sát và quản lý thành một khối thống nhất tạo ra cả thách thức về phát triển và vận hành. Ví dụ, bạn không thể thay đổi UI mà không thử nghiệm và triển khai lại toàn bộ ứng dụng. Và vì thành phần giám sát và quản lý dựa trên công cụ Xử lý sự kiện phức tạp (CEP) duy trì trạng thái trong bộ nhớ nên chúng tôi không thể chạy nhiều phiên bản của ứng dụng! Thật xấu hổ khi thừa nhận điều đó, nhưng tất cả những gì tôi có thể nói là tôi là một nhà phát triển phần mềm và "hãy để người không có tội ném đá đầu tiên".

**thứ 17**

**xviii LỜI NÓI ĐẦU**

Rõ ràng, ứng dụng đã nhanh chóng vượt ra khỏi kiến ​​trúc monolith của nó, nhưng giải pháp thay thế là gì? Câu trả lời đã có trong cộng đồng phần mềm trong một thời gian tại các công ty như eBay và Amazon. Ví dụ, Amazon đã bắt đầu di chuyển khỏi monolith vào khoảng năm 2002 ([https://plus.google.com/110981030061712822816/](https://plus.google.com/110981030061712822816/posts/AaygmbzVeRq) [bài viết/AaygmbzVeRq](https://plus.google.com/110981030061712822816/posts/AaygmbzVeRq)). Kiến trúc mới thay thế khối đơn bằng một tập hợp các dịch vụ được ghép nối lỏng lẻo. Các dịch vụ thuộc sở hữu của những gì Amazon gọi là nhóm hai chiếc bánh pizza—các nhóm đủ nhỏ để được nuôi bằng hai chiếc bánh pizza.

Amazon đã áp dụng kiến ​​trúc này để tăng tốc độ phát triển phần mềm để công ty có thể đổi mới nhanh hơn và cạnh tranh hiệu quả hơn. Kết quả thật ấn tượng: Amazon được báo cáo là triển khai các thay đổi vào sản xuất cứ sau 11,6 giây!

Vào đầu năm 2010, sau khi tôi chuyển sang các dự án khác, tương lai của kiến ​​trúc phần mềm cuối cùng đã bắt kịp tôi. Đó là lúc tôi đọc cuốn sách The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (Addison-Wesley Professional, 2009) của Michael T. Fisher và Martin L. Abbott. Một ý tưởng chính trong cuốn sách đó là khối lập phương tỷ lệ, như đã mô tả trong chương 2, là mô hình ba chiều để mở rộng quy mô ứng dụng. Tỷ lệ trục Y được xác định bởi khối lập phương tỷ lệ về mặt chức năng phân tích ứng dụng thành các dịch vụ. Nhìn lại, điều này khá hiển nhiên, nhưng đối với tôi vào thời điểm đó, đó là khoảnh khắc a-ha! Tôi có thể giải quyết những thách thức mà tôi đã phải đối mặt hai năm trước bằng cách thiết kế Cloud Foundry như một tập hợp các dịch vụ!

Vào tháng 4 năm 2012, tôi đã có bài nói chuyện đầu tiên về phương pháp tiếp cận kiến ​​trúc này, có tên là “Phân tích các ứng dụng về khả năng triển khai và khả năng mở rộng” ([www.slideshare.net/chris.e](http://www.slideshare.net/chris.e.richardson/decomposing-applications-for-scalability-and-deployability-april-2012)

[.richardson/decomposing-applications-for-scalability-and-deployability-april-2012](http://www.slideshare.net/chris.e.richardson/decomposing-applications-for-scalability-and-deployability-april-2012)).TạiVào thời điểm đó, không có thuật ngữ nào được chấp nhận chung cho loại kiến ​​trúc này. Đôi khi tôi gọi nó là kiến ​​trúc mô-đun, đa ngôn ngữ, vì các dịch vụ có thể được viết bằng nhiều ngôn ngữ khác nhau.

Nhưng trong một ví dụ khác về cách tương lai được phân bổ không đồng đều, thuật ngữ dịch vụ vi mô đã được sử dụng tại một hội thảo về kiến ​​trúc phần mềm năm 2011 để mô tả loại kiến ​​trúc này (<https://en.wikipedia.org/wiki/Microservices>). Lần đầu tiên tôi biết đến thuật ngữ này là khi nghe Fred George phát biểu tại Oredev 2013 và tôi thích nó!

Vào tháng 1 năm 2014, tôi đã tạo ra[https://microservices.io](https://microservices.io/)trang web để ghi lại kiến ​​trúc và các mẫu thiết kế mà tôi đã gặp. Sau đó vào tháng 3 năm 2014, James Lewis và Martin Fowler đã xuất bản một bài đăng trên blog về các dịch vụ vi mô ([https://martinfowler](https://martinfowler.com/articles/microservices.html)

[.com/articles/microservices.html](https://martinfowler.com/articles/microservices.html)). Bằng cách phổ biến thuật ngữ dịch vụ siêu nhỏ, bài đăng trên blog đã khiến cộng đồng phần mềm thống nhất xung quanh khái niệm này.

Ý tưởng về các nhóm nhỏ, liên kết lỏng lẻo, phát triển và cung cấp các dịch vụ siêu nhỏ nhanh chóng và đáng tin cậy đang dần lan tỏa trong cộng đồng phần mềm. Nhưng có khả năng là tầm nhìn về tương lai này khá khác so với thực tế hàng ngày của bạn. Ngày nay, các ứng dụng doanh nghiệp quan trọng đối với doanh nghiệp thường là các khối lớn do các nhóm lớn phát triển. Việc phát hành phần mềm diễn ra không thường xuyên và thường gây khó khăn cho tất cả mọi người liên quan. CNTT thường phải vật lộn để theo kịp nhu cầu của doanh nghiệp. Bạn đang tự hỏi làm thế nào bạn có thể áp dụng kiến ​​trúc dịch vụ siêu nhỏ.

**LỜI NÓI ĐẦU xix**

Mục tiêu của cuốn sách này là trả lời câu hỏi đó. Nó sẽ giúp bạn hiểu rõ về kiến ​​trúc vi dịch vụ, lợi ích và hạn chế của nó, cũng như thời điểm sử dụng nó. Cuốn sách mô tả cách giải quyết nhiều thách thức về thiết kế mà bạn sẽ phải đối mặt, bao gồm cách quản lý dữ liệu phân tán. Nó cũng đề cập đến cách tái cấu trúc một ứng dụng đơn khối thành kiến ​​trúc vi dịch vụ. Nhưng cuốn sách này không phải là bản tuyên ngôn về vi dịch vụ. Thay vào đó, nó được tổ chức xung quanh một tập hợp các mẫu. Mẫu là giải pháp có thể tái sử dụng cho một vấn đề xảy ra trong một bối cảnh cụ thể. Điểm hấp dẫn của mẫu là ngoài việc mô tả lợi ích của giải pháp, nó còn mô tả những hạn chế và các vấn đề bạn phải giải quyết để triển khai thành công một giải pháp. Theo kinh nghiệm của tôi, loại khách quan này khi suy nghĩ về các giải pháp dẫn đến việc ra quyết định tốt hơn nhiều. Tôi hy vọng bạn sẽ thích đọc cuốn sách này và nó dạy bạn cách phát triển thành công các vi dịch vụ.

# lời cảm ơn

Mặc dù viết là một hoạt động đơn độc nhưng cần rất nhiều người cùng chung tay để biến bản thảo thành một cuốn sách hoàn chỉnh.

Đầu tiên, tôi muốn cảm ơn Erin Twohey và Michael Stevens từ Manning vì sự động viên liên tục của họ để viết một cuốn sách khác. Tôi cũng muốn cảm ơn các biên tập viên phát triển của tôi, Cynthia Kane và Marina Michaels. Cynthia Kane đã giúp tôi bắt đầu và làm việc với tôi trong vài chương đầu tiên. Marina Michaels đã thay thế Cynthia và làm việc với tôi cho đến khi hoàn thành. Tôi sẽ mãi biết ơn những lời phê bình tỉ mỉ và mang tính xây dựng của Marina đối với các chương của tôi. Và tôi muốn cảm ơn những người còn lại trong nhóm Manning đã tham gia vào việc xuất bản cuốn sách này.

Tôi muốn cảm ơn biên tập viên phát triển kỹ thuật của tôi, Christian Mennerich, người hiệu đính kỹ thuật của tôi, Andy Miles, và tất cả những người đánh giá bên ngoài của tôi: Andy Kirsch, Antonio Pessolano, Areg Melik-Adamyan, Cage Slagel, Carlos Curotto, Dror Helper, Eros Pedrini, Hugo Cruz, Irina Romanenko, Jesse Rosalia, Joe Justesen, John Guthrie, Keerthi Shetty, Michele Mauro, Paul Grebenc, Pethuru Raj, Potito Coluccelli, Shobha Iyer, Simeon Leyzerzon, Srihari Sridharan, Tim Moore, Tony Sweets, Trent Whiteley, Wes Shaddix, William E. Wheeler và Zoltan Hamori.

Tôi cũng muốn cảm ơn tất cả mọi người đã mua MEAP và cung cấp phản hồi trên diễn đàn hoặc trực tiếp cho tôi.

Tôi muốn cảm ơn những người tổ chức và tham dự tất cả các hội nghị và buổi gặp gỡ mà tôi đã phát biểu vì đã cho tôi cơ hội trình bày và sửa đổi ý tưởng của mình. Và tôi muốn cảm ơn các khách hàng tư vấn và đào tạo của tôi trên toàn thế giới vì đã cho tôi cơ hội giúp họ đưa ý tưởng của tôi vào thực tế.

**xx**

**LỜI CẢM ƠN xxi**

Tôi muốn cảm ơn các đồng nghiệp Andrew, Valentin, Artem và Stanislav tại Eventuate, Inc. vì những đóng góp của họ cho sản phẩm Eventuate và các dự án nguồn mở.

Cuối cùng, tôi muốn cảm ơn vợ tôi, Laura, và các con tôi, Ellie, Thomas và Janet vì sự ủng hộ và thấu hiểu của họ trong suốt 18 tháng qua. Trong khi tôi dán mắt vào máy tính xách tay, tôi đã bỏ lỡ việc đến xem các trận bóng đá của Ellie, xem Thomas học lái máy bay trên máy bay mô phỏng và thử các nhà hàng mới với Janet.

Cảm ơn tất cả mọi người!

# về cuốn sách này

Mục tiêu của cuốn sách này là hướng dẫn bạn cách phát triển ứng dụng thành công bằng cách sử dụng kiến ​​trúc vi dịch vụ.

Nó không chỉ thảo luận về lợi ích của kiến ​​trúc microservice mà còn mô tả những nhược điểm. Bạn sẽ biết khi nào nên cân nhắc sử dụng kiến ​​trúc monolithic và khi nào nên sử dụng microservices.

#### Ai nên đọc cuốn sách này

Trọng tâm của cuốn sách này là về kiến ​​trúc và phát triển. Nó dành cho bất kỳ ai chịu trách nhiệm phát triển và cung cấp phần mềm, chẳng hạn như nhà phát triển, kiến ​​trúc sư, giám đốc công nghệ hoặc phó chủ tịch kỹ thuật.

Cuốn sách tập trung vào việc giải thích các mô hình kiến ​​trúc vi dịch vụ và các khái niệm khác. Mục tiêu của tôi là bạn có thể tìm thấy tài liệu này dễ tiếp cận, bất kể bạn sử dụng công nghệ nào. Bạn chỉ cần quen thuộc với những điều cơ bản về kiến ​​trúc và thiết kế ứng dụng doanh nghiệp. Đặc biệt, bạn cần hiểu các khái niệm như kiến ​​trúc ba tầng, thiết kế ứng dụng web, cơ sở dữ liệu quan hệ, giao tiếp giữa các quy trình bằng cách sử dụng tin nhắn và REST, và những điều cơ bản về bảo mật ứng dụng. Tuy nhiên, các ví dụ mã sử dụng Java và khung Spring. Để tận dụng tối đa chúng, bạn nên quen thuộc với khung Spring.

**xxii**

#### Lộ trình

Cuốn sách này gồm có 13 chương:

* Chương 1 mô tả các triệu chứng của địa ngục đơn khối, xảy ra khi một ứng dụng đơn khối phát triển vượt quá kiến ​​trúc của nó và tư vấn cách thoát khỏi bằng cách áp dụng kiến ​​trúc vi dịch vụ. Chương này cũng cung cấp tổng quan về ngôn ngữ mẫu kiến ​​trúc vi dịch vụ, là chủ đề tổ chức cho hầu hết cuốn sách.
* Chương 2 giải thích lý do tại sao kiến ​​trúc phần mềm lại quan trọng và mô tả các mẫu bạn có thể sử dụng để phân tích ứng dụng thành một tập hợp các dịch vụ. Chương này cũng giải thích cách vượt qua các trở ngại khác nhau mà bạn thường gặp phải trong quá trình này.
* Chương 3 mô tả các mô hình khác nhau cho giao tiếp liên tiến trình mạnh mẽ trong kiến ​​trúc dịch vụ vi mô. Chương này giải thích tại sao giao tiếp dựa trên tin nhắn không đồng bộ thường là lựa chọn tốt nhất.
* Chương 4 giải thích cách duy trì tính nhất quán của dữ liệu trên các dịch vụ bằng cách sử dụng mẫu Saga. Saga là một chuỗi các giao dịch cục bộ được phối hợp bằng cách sử dụng tin nhắn không đồng bộ.
* Chương 5 mô tả cách thiết kế logic kinh doanh cho dịch vụ bằng cách sử dụng các mẫu sự kiện miền và tổng hợp thiết kế theo miền (DDD).
* Chương 6 dựa trên chương 5 và giải thích cách phát triển logic kinh doanh bằng cách sử dụng mô hình Nguồn sự kiện, một phương pháp lấy sự kiện làm trung tâm để cấu trúc logic kinh doanh và lưu trữ các đối tượng miền.
* Chương 7 mô tả cách triển khai các truy vấn để truy xuất dữ liệu nằm rải rác trên nhiều dịch vụ bằng cách sử dụng mô hình thành phần API hoặc mô hình phân tách trách nhiệm truy vấn lệnh (CQRS).
* Chương 8 trình bày các mẫu API bên ngoài để xử lý các yêu cầu từ nhiều nhóm máy khách bên ngoài khác nhau, chẳng hạn như ứng dụng di động, ứng dụng Java-Script dựa trên trình duyệt và ứng dụng của bên thứ ba.
* Chương 9 là chương đầu tiên trong hai chươngvề các kỹ thuật kiểm thử tự động cho các dịch vụ vi mô. Nó giới thiệu các khái niệm kiểm thử quan trọng như kim tự tháp kiểm thử, mô tả tỷ lệ tương đối của từng loại kiểm thử trong bộ kiểm thử của bạn. Nó cũng chỉ ra cách viết các bài kiểm thử đơn vị, tạo thành cơ sở của kim tự tháp kiểm thử.
* Chương 10dựa trên chương 9 và mô tả cách viết các loại thử nghiệm khác trong kim tự tháp thử nghiệm, bao gồm thử nghiệm tích hợp, thử nghiệm hợp đồng người tiêu dùng và thử nghiệm thành phần.
* Chương 11 đề cập đến nhiều khía cạnh khác nhau của việc phát triển các dịch vụ sẵn sàng cho sản xuất, bao gồm bảo mật, mẫu cấu hình Externalized và các mẫu khả năng quan sát dịch vụ. Các mẫu khả năng quan sát dịch vụ bao gồm Tổng hợp nhật ký, Số liệu ứng dụng và Theo dõi phân tán.
* Chương 12 mô tả các mẫu triển khai khác nhau mà bạn có thể sử dụng để triển khai các dịch vụ, bao gồm máy ảo, vùng chứa và không có máy chủ. Nó cũng

**xxiv VỀ CUỐN SÁCH NÀY**

thảo luận về những lợi ích của việc sử dụng dịch vụ lưới, một lớp phần mềm mạng làm trung gian truyền thông trong kiến ​​trúc dịch vụ vi mô.

* Chương 13 giải thích cách cải tiến dần dần một kiến ​​trúc độc khối thành kiến ​​trúc vi dịch vụ bằng cách áp dụng mô hình ứng dụng Strangler: triển khai các tính năng mới dưới dạng dịch vụ và trích xuất các mô-đun ra khỏi khối và chuyển đổi chúng thành dịch vụ.

Khi bạn tìm hiểu các chương này, bạn sẽ tìm hiểu về các khía cạnh khác nhau của kiến ​​trúc vi dịch vụ.

#### Về mã

Cuốn sách này chứa nhiều ví dụ về mã nguồn trong cả danh sách được đánh số vàinline với văn bản bình thường. Trong cả hai trường hợp, mã nguồn được định dạng theo phông chữ có chiều rộng cố định như thế này để tách mã nguồn khỏi văn bản bình thường. Đôi khi, mã nguồn cũng được in đậm đến in nổi bật, mã nguồn đã thay đổi từ các bước trước đó trong chương, chẳng hạn như khi một tính năng mới được thêm vào một dòng mã hiện có. Trong nhiều trường hợp, mã nguồn gốc đã được định dạng lại; nhà xuất bản đã thêm ngắt dòng và thụt lề lại để phù hợp với không gian trang có sẵn trong sách. Trong một số trường hợp hiếm hoi, ngay cả điều này cũng không đủ và danh sách bao gồm các dấu tiếp tục dòng (➥). Ngoài ra, bình luậntrong mã nguồn thường bị xóa khỏi danh sách khi mã được mô tả trong văn bản. Chú thích mã đi kèm với nhiều danh sách, làm nổi bật các khái niệm quan trọng.

Mỗi chương, ngoại trừ chương 1, 2 và 13, đều chứa mã từ ứng dụng ví dụ đi kèm. Bạn có thể tìm thấy mã cho ứng dụng này trong kho lưu trữ GitHub:<https://github.com/microservices-patterns/ftgo-application>.

#### Diễn đàn sách

Việc mua Microservices Patterns bao gồm quyền truy cập miễn phí vào diễn đàn web riêng do Manning Publications điều hành, nơi bạn có thể đưa ra bình luận về cuốn sách, đặt câu hỏi kỹ thuật, chia sẻ giải pháp cho các bài tập và nhận trợ giúp từ tác giả và những người dùng khác. Để truy cập diễn đàn và đăng ký, hãy trỏ trình duyệt web của bạn tới<https://forums.manning.com/forums/microservices-patterns>. Bạn cũng có thể tìm hiểu thêm về diễn đàn Manning và các quy tắc ứng xử tại[https://diễn đàn](https://forums.manning.com/forums/about)

[.manning.com/diễn đàn/về](https://forums.manning.com/forums/about).

Cam kết của Manning đối với độc giả của chúng tôi là cung cấp một địa điểm nơi có thể diễn ra cuộc đối thoại có ý nghĩa giữa từng độc giả và giữa độc giả và tác giả. Đây không phải là cam kết về bất kỳ mức độ tham gia cụ thể nào từ phía tác giả, người mà sự đóng góp của họ cho diễn đàn vẫn là tự nguyện (và không được trả công). Chúng tôi đề xuất bạn thử đặt cho tác giả một số câu hỏi đầy thử thách để tránh mất hứng thú! Diễn đàn và kho lưu trữ các cuộc thảo luận trước đây sẽ có thể truy cập được từ trang web của nhà xuất bản miễn là cuốn sách vẫn được in.

#### Các nguồn tài nguyên trực tuyến khác

Một nguồn tài nguyên tuyệt vời khácđể học kiến ​​trúc microservice là trang web của tôi[http://](http://microservices.io/) [microservices.io](http://microservices.io/).

Nó không chỉ chứa ngôn ngữ mẫu hoàn chỉnh mà còn có liên kết đến các tài nguyên khác như bài viết, bài thuyết trình và mã ví dụ.

#### Về tác giả

Chris Richardson là một nhà phát triển và kiến ​​trúc sư. Ông là một Java Champion, một ngôi sao nhạc rock JavaOne và là tác giả của POJOs in Action (Manning, 2006), mô tả cách xây dựng các ứng dụng Java doanh nghiệp với các khuôn khổ như Spring và Hibernate.

Chris cũng là người sáng lập ra CloudFoundry.com ban đầu, một Java PaaS đầu tiên dành cho Amazon EC2.

Ngày nay, ông là một nhà tư tưởng được công nhận trong lĩnh vực dịch vụ vi mô và thường xuyên phát biểu tại các hội nghị quốc tế. Chris là người sáng tạo ra Microservices.io, một ngôn ngữ mẫu cho dịch vụ vi mô. Ông cung cấp dịch vụ tư vấn và đào tạo về dịch vụ vi mô cho các tổ chức trên toàn thế giới đang áp dụng kiến ​​trúc dịch vụ vi mô. Chris đang làm việc cho công ty khởi nghiệp thứ ba của mình: Eventuate.io, một nền tảng ứng dụng để phát triển dịch vụ vi mô giao dịch.

# về hình minh họa bìa

###### Jefferys

Hình ảnh trên trang bìa của Microservices Patterns có tiêu đề là “Thói quen của một người MoriscoNô lệ năm 1568.” Hình minh họa được trích từ Bộ sưu tập trang phục của các quốc gia khác nhau, cổ xưa và hiện đại (bốn tập) của Thomas Jefferys, London, xuất bản từ năm 1757 đến năm 1772. Trang tiêu đề nêu rõ rằng đây là những bản khắc đồng được tô màu thủ công, được tô điểm bằng gôm Ả Rập.

Thomas Jefferys (1719–1771) được gọi là “Nhà địa lý của Vua George III.” Ông là một nhà vẽ bản đồ người Anh, là nhà cung cấp bản đồ hàng đầu thời bấy giờ. Ông khắc và in bản đồ cho chính phủ và các cơ quan chính thức khác và sản xuất nhiều loại bản đồ và atlas thương mại, đặc biệt là của Bắc Mỹ. Công việc làm bản đồ của ông đã khơi dậy sự quan tâm đến phong tục ăn mặc địa phương của những vùng đất mà ông đã khảo sát và lập bản đồ, được thể hiện một cách xuất sắc trong bộ sưu tập này. Sự say mê với những vùng đất xa xôi và du lịch vì mục đích giải trí là những hiện tượng tương đối mới vào cuối thế kỷ 18, và những bộ sưu tập như thế này rất được ưa chuộng, giới thiệu cả khách du lịch cũng như du khách ngồi ghế bành đến với cư dân của các quốc gia khác.

Sự đa dạng của các bức vẽ trong các tập sách của Jefferys nói lên một cách sống động về sự độc đáo và cá tính của các quốc gia trên thế giới cách đây khoảng 200 năm. Quy định về trang phục đã thay đổi kể từ đó, và sự đa dạng theo vùng miền và quốc gia, vốn rất phong phú vào thời điểm đó, đã dần biến mất. Bây giờ, thường rất khó để phân biệt cư dân của một châu lục với một châu lục khác. Có lẽ, khi cố gắng nhìn nhận một cách lạc quan, chúng ta đã đánh đổi sự đa dạng về văn hóa và thị giác để lấy một cuộc sống cá nhân đa dạng hơn—hoặc một cuộc sống trí tuệ và kỹ thuật đa dạng và thú vị hơn.

**xxvi**

**VỀ BÌAHÌNH MINH HỌA xxvii**

Vào thời điểm khó có thể phân biệt một cuốn sách về máy tính với một cuốn sách khác, Manning lại tôn vinh tính sáng tạo và chủ động của ngành kinh doanh máy tính với những bìa sách dựa trên sự đa dạng phong phú của cuộc sống vùng miền cách đây hai thế kỷ, được tái hiện qua những bức tranh của Jeffreys.

*Thoát khỏi địa ngục đơn điệu*

***Chương này bao gồm***

* Các triệu chứng của địa ngục nguyên khối và cáchthoát khỏi nó bằng cách áp dụng kiến ​​trúc vi dịch vụ
* Các đặc điểm thiết yếu của dịch vụ vi môkiến trúc và những lợi ích và hạn chế của nó
* Làm thế nào các dịch vụ vi mô cho phépphong cách DevOps phát triển các ứng dụng lớn, phức tạp
* Ngôn ngữ mẫu kiến ​​trúc vi dịch vụvà tại sao bạn nên sử dụng nó

Mới chỉ là giờ ăn trưa thứ Hai, nhưng Mary, Giám đốc công nghệ của Food to Go, Inc. (FTGO), đã cảm thấy bực bội. Ngày của cô đã bắt đầu thực sự tốt đẹp. Cô đã dành tuần trước với các kiến ​​trúc sư phần mềm và nhà phát triển khác tại một hội nghị tuyệt vời để tìm hiểu về các kỹ thuật phát triển phần mềm mới nhất, bao gồm triển khai liên tục và kiến ​​trúc dịch vụ vi mô. Mary cũng đã gặp gỡ những người bạn học khoa học máy tính cũ của mình tại Đại học North Carolina A&T và chia sẻ những câu chuyện về cuộc chiến lãnh đạo công nghệ. Hội nghị đã khiến cô cảm thấy được trao quyền và háo hức cải thiện cách FTGO phát triển phần mềm.

**1**

Thật không may, cảm giác đó đã nhanh chóng tan biến. Cô vừa dành buổi sáng đầu tiên trở lại văn phòng trong một cuộc họp đau đớn khác với các kỹ sư và doanh nhân cấp cao. Họ đã dành hai giờ để thảo luận về lý do tại sao nhóm phát triển sẽ bỏ lỡ một ngày phát hành quan trọng khác. Đáng buồn thay, loại cuộc họp này đã trở nên ngày càng phổ biến trong vài năm qua. Mặc dù áp dụng phương pháp linh hoạt, tốc độ phát triển đang chậm lại, khiến việc đạt được các mục tiêu kinh doanh trở nên gần như không thể. Và, tệ hơn nữa, dường như không có giải pháp đơn giản nào.

Hội nghị đã khiến Mary nhận ra rằng FTGO đang phải chịu đựng một trường hợp địa ngục đơn khối và cách chữa trị là áp dụng kiến ​​trúc dịch vụ vi mô. Nhưng kiến ​​trúc dịch vụ vi mô và các hoạt động phát triển phần mềm tiên tiến liên quan được mô tả tại hội nghị giống như một giấc mơ khó nắm bắt. Mary không rõ làm thế nào cô có thể dập tắt các đám cháy ngày nay đồng thời cải thiện cách phát triển phần mềm tại FTGO.

May mắn thay, như bạn sẽ học được trong cuốn sách này, có một cách. Nhưng trước tiên, hãy xem xét các vấn đề mà FTGO đang phải đối mặt và cách họ gặp phải chúng.

* 1. ***Cuộc hành trình chậm chạp hướng tới địa ngục khổng lồ***

Kể từ khi ra mắt vào cuối năm 2005, FTGO đã phát triển vượt bậc. Ngày nay, nó là một trongcác công ty giao đồ ăn trực tuyến hàng đầu tại Hoa Kỳ. Doanh nghiệp thậm chí còn có kế hoạch mở rộng ra nước ngoài, mặc dù các kế hoạch đó đang gặp nguy hiểm vì sự chậm trễ trong việc triển khai các tính năng cần thiết.

Về cơ bản, ứng dụng FTGO khá đơn giản. Người tiêu dùng sử dụng trang web FTGO hoặc ứng dụng di động để đặt món ăn tại các nhà hàng địa phương. FTGO điều phối mạng lưới các đơn vị chuyển phát giao hàng. Nó cũng chịu trách nhiệm thanh toán cho các đơn vị chuyển phát và nhà hàng. Các nhà hàng sử dụng trang web FTGO để chỉnh sửa thực đơn và quản lý đơn hàng. Ứng dụng sử dụng nhiều dịch vụ web khác nhau, bao gồm Stripe để thanh toán, Twilio để nhắn tin và Amazon Simple Email Service (SES) để gửi email.

Giống như nhiều ứng dụng doanh nghiệp cũ khác, ứng dụng FTGO là một khối đơn, bao gồm một tệp Java Web Application Archive (WAR) duy nhất. Trong nhiều năm, nó đã trở thành một ứng dụng lớn, phức tạp. Bất chấp những nỗ lực hết mình của nhóm phát triển FTGO, nó đã trở thành một ví dụ về mô hình Big Ball of Mud ([www.laputan](http://www.laputan.org/mud/)

[.org/mud/](http://www.laputan.org/mud/)). Trích dẫn Foote và Yoder, tác giả của mô hình đó, đó là "một khu rừng mã spaghetti lộn xộn, hỗn độn, lộn xộn, băng keo và dây thép". Tốc độ phân phối phần mềm đã chậm lại. Tệ hơn nữa, ứng dụng FTGO đã được viết bằng một số khuôn khổ ngày càng lỗi thời. Ứng dụng FTGO đang biểu hiện tất cả các triệu chứng của địa ngục đơn khối.

Phần tiếp theo mô tả kiến ​​trúc của ứng dụng FTGO. Sau đó, phần này sẽ nói về lý do tại sao kiến ​​trúc monolithic hoạt động tốt lúc đầu. Chúng ta sẽ tìm hiểu cách ứng dụng FTGO phát triển vượt ra khỏi kiến ​​trúc của nó và cách điều đó dẫn đến địa ngục monolithic.

###### Kiến trúc của ứng dụng FTGO

FTGO là một ứng dụng Java doanh nghiệp điển hình. Hình 1.1 cho thấy kiến ​​trúc của nó. Ứng dụng FTGO có kiến ​​trúc lục giác, đây là kiểu kiến ​​trúc được mô tả chi tiết hơn trong chương 2. Trong kiến ​​trúc lục giác, cốt lõi của ứng dụng bao gồm logic nghiệp vụ. Bao quanh logic nghiệp vụ là nhiều bộ điều hợp triển khai UI và tích hợp với các hệ thống bên ngoài.

**Được gọi bởi các ứng dụng di động**



Người chuyển phát nhanh NGHỈ NGƠI

Giao diện lập trình ứng dụng (API)

|  |  |
| --- | --- |
| Đặt hàng  sự quản lý | Vận chuyển  sự quản lý |
|  |  |
| Thanh toán |

Người tiêu dùng

Ứng dụng FTGO

sự quản lý

Nhà hàng

Dịch vụ đám mây

Bộ chuyển đổi Twilio

Dịch vụ nhắn tin Twilio

Thông báo

Nhà hàng

Trang webGiao diện người dùng

MySQL

bộ chuyển đổi

Bộ chuyển đổi sọc

Amazon SES

bộ chuyển đổi

Dịch vụ thanh toán Stripe

Thanh toán

AWS SES

dịch vụ email

MySQL

**Bộ điều hợp gọi các dịch vụ đám mây.**

**Hình 1.1Ứng dụng FTGO có kiến ​​trúc lục giác. Nó bao gồm logic kinh doanh được bao quanh bởi các bộ điều hợp triển khai UI và giao diện với các hệ thống bên ngoài, chẳng hạn như ứng dụng di động và dịch vụ đám mây để thanh toán, nhắn tin và email.**

Logic kinh doanh bao gồm các mô-đun, mỗi mô-đun là một tập hợp các đối tượng miền. Ví dụ về các mô-đun bao gồm Quản lý đơn hàng, Quản lý giao hàng, Thanh toán và Thanh toán. Có một số bộ điều hợp giao tiếp với các hệ thống bên ngoài. Một số là bộ điều hợp đến, xử lý các yêu cầu bằng cách gọi logic kinh doanh, bao gồm các bộ điều hợp REST API và Web UI. Những bộ điều hợp khác là bộ điều hợp đi, cho phép logic kinh doanh truy cập cơ sở dữ liệu MySQL và gọi các dịch vụ đám mây như Twilio và Stripe.

Mặc dù có kiến ​​trúc mô-đun logic, ứng dụng FTGO được đóng gói dưới dạng một tệp WAR duy nhất. Ứng dụng này là một ví dụ về phong cách nguyên khối được sử dụng rộng rãi

của kiến ​​trúc phần mềm, cấu trúc hệ thống thành một thành phần có thể thực thi hoặc triển khai duy nhất. Nếu ứng dụng FTGO được viết bằng ngôn ngữ Go (GoLang), thì đó sẽ là một ứng dụng có thể thực thi duy nhất. Phiên bản Ruby hoặc NodeJS của ứng dụng sẽ là một hệ thống phân cấp thư mục duy nhất của mã nguồn. Kiến trúc nguyên khối không phải là xấu về bản chất. Các nhà phát triển FTGO đã đưa ra quyết định đúng đắn khi họ chọn kiến ​​trúc nguyên khối cho ứng dụng của mình.

###### Lợi ích của kiến ​​trúc nguyên khối

Vào những ngày đầu của FTGO, khi ứng dụng còn tương đối nhỏ, kiến ​​trúc độc khối của ứng dụng có rất nhiều lợi ích:

* + - * *Dễ dàng phát triển*—IDE và các công cụ phát triển khác tập trung vào việc xây dựng một ứng dụng duy nhất.
      * *Dễ dàng thực hiện những thay đổi cơ bảnđến ứng dụng*—Bạn có thể thay đổi mã và lược đồ cơ sở dữ liệu, xây dựng và triển khai.
      * *Dễ dàng để kiểm tra*—Các nhà phát triển đã viết các bài kiểm tra đầu cuối để khởi chạy ứng dụng, gọi REST API và kiểm tra giao diện người dùng bằng Selenium.
      * *Dễ dàng triển khai*—Tất cả những gì nhà phát triển phải làm là sao chép tệp WAR vào máy chủ đã cài đặt Tomcat.
      * *Dễ dàng mở rộng quy mô*—FTGO đã chạy nhiều phiên bản của ứng dụng đằng sau bộ cân bằng tải.

Tuy nhiên, theo thời gian, quá trình phát triển, thử nghiệm, triển khai vàviệc mở rộng quy mô trở nên khó khăn hơn nhiều. Hãy cùng xem lý do tại sao.

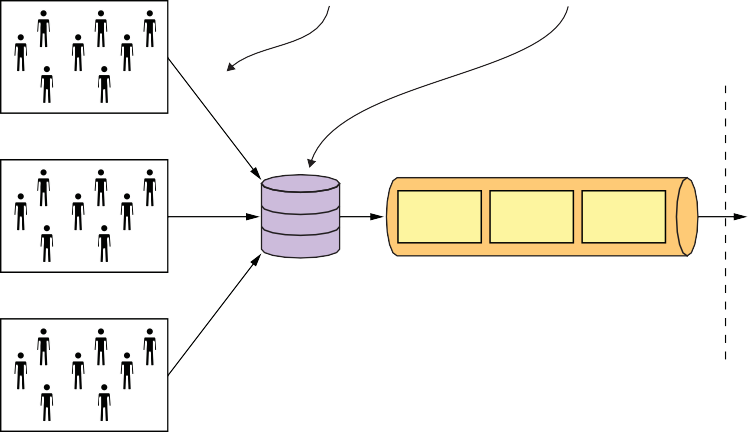
###### Sống trong địa ngục đơn điệu

Thật không may, như các nhà phát triển FTGO đã phát hiện ra, kiến ​​trúc monolithic có một hạn chế rất lớn. Các ứng dụng thành công như ứng dụng FTGO có thói quen phát triển vượt quá kiến ​​trúc monolithic. Mỗi lần chạy nước rút, nhóm phát triển FTGO triển khai thêm một vài câu chuyện, khiến cơ sở mã lớn hơn. Hơn nữa, khi công ty trở nên thành công hơn, quy mô của nhóm phát triển cũng tăng đều đặn. Điều này không chỉ làm tăng tốc độ phát triển của cơ sở mã mà còn làm tăng chi phí quản lý.

Như hình 1.2 cho thấy, ứng dụng FTGO nhỏ bé, đơn giản đã phát triển theo năm tháng thành một khối khổng lồ. Tương tự như vậy, nhóm phát triển nhỏ hiện đã trở thành nhiều nhóm Scrum, mỗi nhóm làm việc trên một lĩnh vực chức năng cụ thể. Do phát triển vượt quá kiến ​​trúc của mình, FTGO đang ở trong địa ngục khối. Phát triển chậm và đau đớn. Phát triển và triển khai theo phương pháp Agile là không thể. Hãy cùng xem tại sao điều này lại xảy ra.

**COMPLEXITY LÀM NHÀ PHÁT TRIỂN ĐÁNG SỢ**

Một vấn đề lớn với ứng dụng FTGO là nó quá phức tạp. Nó quá lớn để bất kỳ nhà phát triển nào có thể hiểu đầy đủ. Do đó, việc sửa lỗi và triển khai đúng các tính năng mới trở nên khó khăn và tốn thời gian. Không đáp ứng được thời hạn.



**Tổ chức phát triển lớn**

**Cơ sở mã đơntạo ra Giao tiếp lớn, phức tạpVà không đáng tin cậy, khó khăn**

**sự phối hợptrên không. để duy trì**

Đường ống triển khai

Jenkins tồn đọng Thủ côngKiểm tra Cl

Nguồn

kho lưu trữ mã

Sản xuất

**Con đường từ cam kết mã đến sản xuất rất gian nan.**

**Những thay đổi sẽ nằm trong hàng đợi cho đến khi có thể được kiểm tra thủ công.**

FTGO

ứng dụng

Phát triển FTGO

Đội ngũ quản lý đơn hàng

Đội ngũ quản lý nhà hàng

Đội ngũ quản lý giao hàng

**Hình 1.2 Một trường hợp địa ngục đơn khối. Nhóm phát triển FTGO lớn cam kết các thay đổi của họ vào một kho lưu trữ mã nguồn duy nhất. Đường dẫn từ cam kết mã đến sản xuất rất dài và gian nan và liên quan đến thử nghiệm thủ công. Ứng dụng FTGO lớn, phức tạp, không đáng tin cậy và khó bảo trì.**

Tệ hơn nữa, điều này thật áp đảođộ phức tạp có xu hướng đi xuống. Nếu cơ sở mã khó hiểu, nhà phát triển sẽ không thực hiện thay đổi đúng cách. Mỗi thay đổi làm cho cơ sở mã phức tạp hơn và khó hiểu hơn. Kiến trúc mô-đun sạch sẽ được hiển thị trước đó trong hình 1.1 không phản ánh thực tế. FTGO đang dần trở thành một quả cầu bùn khổng lồ, khó hiểu và lớn.

Mary nhớ lại gần đây đã tham dự một hội nghị, nơi cô gặp một nhà phát triển đang viết một công cụ để phân tích các mối phụ thuộc giữa hàng nghìn JAR trong ứng dụng hàng triệu dòng mã (LOC) của họ. Vào thời điểm đó, công cụ đó có vẻ như là thứ mà FTGO có thể sử dụng. Bây giờ cô không chắc lắm. Mary nghi ngờ một cách tiếp cận tốt hơn là di chuyển sang một kiến ​​trúc phù hợp hơn với một ứng dụng phức tạp: microservices.

**DSỰ PHÁT TRIỂNCHẬM**

Ngoài việc phải đấu tranh với sự phức tạp quá mức, các nhà phát triển FTGO thấy các tác vụ phát triển hàng ngày chậm chạp. Ứng dụng lớn làm quá tải và làm chậm IDE của nhà phát triển. Việc xây dựng ứng dụng FTGO mất nhiều thời gian. Hơn nữa, vì quá lớn nên ứng dụng mất nhiều thời gian để khởi động. Kết quả là, vòng lặp chỉnh sửa-xây dựng-chạy-kiểm tra mất nhiều thời gian, ảnh hưởng xấu đến năng suất.

**PATH TỪ CAM KẾT ĐẾN TRIỂN KHAI THỰC SỰ LÀ QUÁ TRÌNH DÀI VÀ KHÓ KHĂN**

Một vấn đề khác với ứng dụng FTGO là việc triển khai các thay đổi vào sản xuất là một quá trình dài và đau đớn. Nhóm thường triển khai các bản cập nhật vào sản xuất một lần một tháng, thường là vào đêm muộn thứ sáu hoặc thứ bảy. Mary tiếp tục đọc rằng tình trạng hiện đại nhất đối với các ứng dụng Phần mềm dưới dạng Dịch vụ (SaaS) là triển khai liên tục:

triển khai các thay đổi vào sản xuất nhiều lần trong ngày trong giờ làm việc. Rõ ràng là, tính đến năm 2011, Amazon.com đã triển khai một thay đổi vào sản xuất cứ sau 11,6 giây mà không bao giờ ảnh hưởng đến người dùng! Đối với các nhà phát triển FTGO, việc cập nhật sản xuất nhiều hơn một lần một tháng có vẻ như là một giấc mơ xa vời. Và việc áp dụng triển khai liên tục dường như là điều không thể.

FTGO đã áp dụng một phần phương pháp agile. Nhóm kỹ sư được chia thành các nhóm nhỏ và sử dụng sprint kéo dài hai tuần. Thật không may, hành trình từ khi hoàn thành code đến khi chạy trong production rất dài và gian nan. Một vấn đề với rất nhiều nhà phát triển cam kết với cùng một cơ sở code là bản build thường ở trạng thái không thể phát hành. Khi các nhà phát triển FTGO cố gắng giải quyết vấn đề này bằng cách sử dụng các nhánh tính năng, nỗ lực của họ đã dẫn đến việc hợp nhất kéo dài và đau đớn. Do đó, sau khi một nhóm hoàn thành sprint của mình, một thời gian dài thử nghiệm và ổn định code sẽ diễn ra.

Một lý do khác khiến việc đưa những thay đổi vào sản xuất mất nhiều thời gian là do việc thử nghiệm mất nhiều thời gian. Bởi vì cơ sở mã rất phức tạp và tác động của một thay đổi không được hiểu rõ, các nhà phát triển và máy chủ Tích hợp liên tục (CI) phải chạy toàn bộ bộ kiểm thử. Một số phần của hệ thống thậm chí còn yêu cầu kiểm thử thủ công. Cũng mất một thời gian để chẩn đoán và khắc phục nguyên nhân gây ra lỗi kiểm thử. Do đó, phải mất một vài ngày để hoàn thành một chu kỳ kiểm thử.

**SGỌI LÀ KHÓ KHĂN**

Nhóm FTGO cũng gặp vấn đề trong việc mở rộng ứng dụng của mình. Đó là vì các mô-đun ứng dụng khác nhau có các yêu cầu về tài nguyên xung đột. Ví dụ, dữ liệu nhà hàng được lưu trữ trong cơ sở dữ liệu lớn trong bộ nhớ, lý tưởng nhất là triển khai trên các máy chủ có nhiều bộ nhớ. Ngược lại, mô-đun xử lý hình ảnh sử dụng nhiều CPU và triển khai tốt nhất trên các máy chủ có nhiều CPU. Vì các mô-đun này là một phần của cùng một ứng dụng, FTGO phải thỏa hiệp về cấu hình máy chủ.

**DVIỆC CUNG CẤP MỘT MONOLITH ĐÁNG TIN CẬY LÀ MỘT THÁCH THỨC**

Một vấn đề khác với ứng dụng FTGO là thiếu độ tin cậy. Kết quả là, thường xuyên xảy ra tình trạng ngừng sản xuất. Một lý do khiến ứng dụng không đáng tin cậy là việc kiểm tra ứng dụng một cách kỹ lưỡng rất khó khăn do kích thước lớn của ứng dụng. Việc thiếu khả năng kiểm tra này có nghĩa là lỗi có thể xâm nhập vào quá trình sản xuất. Tệ hơn nữa, ứng dụng thiếu khả năng cô lập lỗi vì tất cả các mô-đun đều chạy trong cùng một quy trình. Thỉnh thoảng, một lỗi trong một mô-đun—ví dụ, rò rỉ bộ nhớ—sẽ làm sập tất cả các phiên bản của ứng dụng, từng cái một. Các nhà phát triển FTGO không thích bị gọi vào giữa đêm vì sự cố ngừng sản xuất. Những người kinh doanh thậm chí còn không thích mất doanh thu và mất lòng tin.

**LBỊ LẮP VÀO CÔNG NGHỆ NGÀY CÀNG LỖI THỜI**

Mặt cuối cùng của địa ngục nguyên khối mà nhóm FTGO trải qua là kiến ​​trúc buộc họ phải sử dụng một ngăn xếp công nghệ đang ngày càng trở nên lỗi thời. Kiến trúc nguyên khối khiến việc áp dụng các khuôn khổ và ngôn ngữ mới trở nên khó khăn. Sẽ cực kỳ tốn kém và rủi ro khi viết lại toàn bộ ứng dụng nguyên khối để nó có thể sử dụng một công nghệ mới và có lẽ là tốt hơn. Do đó, các nhà phát triển

***Những gì bạn sẽ học được trong nàysách* 7**

bị mắc kẹt với các lựa chọn công nghệ mà họ đã thực hiện khi bắt đầu dự án. Thông thường, họ phải duy trì một ứng dụng được viết bằng một ngăn xếp công nghệ ngày càng lỗi thời.

Khung Spring vẫn tiếp tục phát triển trong khi vẫn tương thích ngược, vì vậy về mặt lý thuyết FTGO có thể nâng cấp. Thật không may, ứng dụng FTGO sử dụng các phiên bản khung không tương thích với các phiên bản Spring mới hơn. Nhóm phát triển chưa bao giờ tìm được thời gian để nâng cấp các khung đó. Do đó, các phần chính của ứng dụng được viết bằng các khung ngày càng lỗi thời. Hơn nữa, các nhà phát triển FTGO muốn thử nghiệm với các ngôn ngữ không phải JVMchẳng hạn như GoLang và NodeJS. Đáng buồn thay, điều đó không thể thực hiện được với một ứng dụng độc lập.

#### Tại sao cuốn sách này có liên quan đến bạn

Có khả năng bạn là một nhà phát triển, kiến ​​trúc sư, giám đốc công nghệ hoặc phó chủ tịch kỹ thuật. Bạn chịu trách nhiệm cho một ứng dụng đã phát triển vượt ra khỏi kiến ​​trúc đơn khối của nó. Giống như Mary tại FTGO, bạn đang vật lộn với việc phân phối phần mềm và muốn biết cách thoát khỏi địa ngục đơn khối. Hoặc có lẽ bạn lo sợ rằng tổ chức của mình đang trên con đường đến địa ngục đơn khối và bạn muốn biết cách thay đổi hướng đi trước khi quá muộn. Nếu bạn cần thoát khỏi hoặc tránh địa ngục đơn khối, đây là cuốn sách dành cho bạn.

Cuốn sách này dành nhiều thời gian để giải thích các khái niệm về kiến ​​trúc microservice. Mục tiêu của tôi là bạn có thể tìm thấy tài liệu này dễ tiếp cận, bất kể bạn sử dụng công nghệ nào. Tất cả những gì bạn cần là phải quen thuộc với những điều cơ bản về kiến ​​trúc và thiết kế ứng dụng doanh nghiệp. Cụ thể, bạn cần biết những điều sau:

* Kiến trúc ba tầng
* Ứng dụng webthiết kế
* Làm thế nào để phát triển logic kinh doanh bằng cách sử dụng thiết kế hướng đối tượng
* Cách sử dụng RDBMS: Giao dịch SQL và ACID
* Cách sử dụng giao tiếp giữa các tiến trình bằng cách sử dụng môi giới tin nhắn và API REST
* Bảo mật, bao gồm xác thực và ủy quyền

Các ví dụ mã trong cuốn sách này được viết bằng Java và Spring framework. Điều đó có nghĩa là để tận dụng tối đa các ví dụ, bạn cũng cần phải quen thuộc với Spring framework.

#### Những gì bạn sẽ học được trong cuốn sách này

Khi bạn đọc xong cuốn sách này, bạn sẽ hiểusau đây:

* Các đặc điểm thiết yếu của kiến ​​trúc vi dịch vụ, lợi ích và hạn chế của nó, và thời điểm sử dụng nó
* Các mô hình quản lý dữ liệu phân tán
* Chiến lược thử nghiệm dịch vụ vi mô hiệu quả
* Tùy chọn triển khai cho các dịch vụ vi mô
* Các chiến lược để tái cấu trúc một ứng dụng độc lập thành kiến ​​trúc dịch vụ vi mô

Bạn cũng có thể thực hiện những điều sau:

* Thiết kế ứng dụng bằng cách sử dụng mô hình kiến ​​trúc vi dịch vụ
* Phát triển logic kinh doanh cho một dịch vụ
* Sử dụng sagas để duy trì tính nhất quán của dữ liệu trên các dịch vụ
* Triển khai các truy vấn mở rộng dịch vụ
* Kiểm tra hiệu quả các dịch vụ vi mô
* Phát triển các dịch vụ sẵn sàng sản xuất, an toàn, có thể định cấu hình và có thể quan sát được
* Tái cấu trúc một ứng dụng đơn khối hiện cóđến các dịch vụ

#### Kiến trúc vi dịch vụ để giải cứu

Mary đã đi đến kết luận rằng FTGO phải chuyển sang kiến ​​trúc dịch vụ vi mô.

Điều thú vị là kiến ​​trúc phần mềm có rất ít liên quan đến các yêu cầu chức năng. Bạn có thể triển khai một tập hợp các trường hợp sử dụng—các yêu cầu chức năng của ứng dụng—với bất kỳ kiến ​​trúc nào. Trên thực tế, các ứng dụng thành công, chẳng hạn như ứng dụng FTGO, thường là những quả bóng bùn lớn.

Tuy nhiên, kiến ​​trúc rất quan trọng vì nó ảnh hưởng đến cái gọi là chất lượng dịch vụ.*phó*yêu cầu, còn được gọi là yêu cầu phi chức năng, thuộc tính chất lượng hoặc khả năng. Khi ứng dụng FTGO phát triển, nhiều thuộc tính chất lượng đã bị ảnh hưởng, đáng chú ý nhất là những thuộc tính ảnh hưởng đến tốc độ phân phối phần mềm: khả năng bảo trì, khả năng mở rộng và khả năng kiểm tra.

Một mặt, một nhóm có kỷ luật có thể làm chậm tốc độ đi xuống địa ngục đơn khối của mình. Các thành viên trong nhóm có thể làm việc chăm chỉ để duy trì tính mô-đun của ứng dụng của họ. Họ có thể viết các bài kiểm tra tự động toàn diện. Mặt khác, họ không thể tránh khỏi các vấn đề của một nhóm lớn làm việc trên một ứng dụng đơn khối duy nhất. Họ cũng không thể giải quyết vấn đề của một ngăn xếp công nghệ ngày càng lỗi thời. Điều tốt nhất mà một nhóm có thể làm là trì hoãn điều không thể tránh khỏi. Để thoát khỏi địa ngục đơn khối, họ phải di chuyển sang một kiến ​​trúc mới: kiến ​​trúc Microservice.

Ngày nay, sự đồng thuận ngày càng tăng là nếu bạn đang xây dựng một ứng dụng lớn, phức tạp, bạn nên cân nhắc sử dụng kiến ​​trúc vi dịch vụ. Nhưng vi dịch vụ chính xác là gì? Thật không may, cái tên không giúp ích gì vì nó nhấn mạnh quá mức vào kích thước. Có rất nhiều định nghĩa về kiến ​​trúc vi dịch vụ. Một số người hiểu theo nghĩa đen của tên và cho rằng một dịch vụ phải rất nhỏ—ví dụ, 100 LOC. Những người khác cho rằng một dịch vụ chỉ mất hai tuần để phát triển. Adrian Cockcroft, trước đây làm việc cho Netflix, định nghĩa kiến ​​trúc vi dịch vụ là kiến ​​trúc hướng dịch vụ bao gồm các thành phần được ghép lỏng lẻo có ngữ cảnh bị giới hạn. Đó không phải là một định nghĩa tồi, nhưng hơi dày đặc. Chúng ta hãy xem liệu chúng ta có thể làm tốt hơn không.

###### Khối lập phương và dịch vụ vi mô

Định nghĩa của tôi về kiến ​​trúc vi dịch vụ được lấy cảm hứng từ cuốn sách tuyệt vời của Martin Abbott và Michael Fisher, The Art of Scalability (Addison-Wesley, 2015).

Cuốn sách mô tả một mô hình khả năng mở rộng ba chiều hữu ích:khối lập phương cân, được thể hiện ở hình 1.3.

Dịch vụ vi mô

Tỷ lệ trục Y,

còn gọi là chức năngsự phân hủy

Tỷ lệ bằng cách chia táchnhững thứ khác nhau, chẳng hạn như về chức năng.

Khối đá nguyên khối

Một

Nhiều

Một phân vùng

Nhiều phân vùng

Tỷ lệ trục Z,

còn gọi là phân vùng dữ liệu

Chia tỷ lệ bằng cách chia nhỏ những thứ tương tự, chẳng hạn như

ví dụ

Tỷ lệ trục X,

còn gọi là sự trùng lặp theo chiều ngangNhân bản bằng cách nhân bản.

trường hợp

theo mã khách hàng.

**Hình 1.3Khối lập phương tỷ lệ xác định ba cách riêng biệt để mở rộng quy mô ứng dụng: Tỷ lệ trục X cân bằng tải các yêu cầu trên nhiều phiên bản giống hệt nhau; Tỷ lệ trục Z định tuyến các yêu cầu dựa trên thuộc tính của yêu cầu; Trục Y phân tích chức năng một ứng dụng thành các dịch vụ.**

Mô hình xác định ba cách để mở rộng ứng dụng: X, Y và Z.

**X-YÊU CẦU CÂN BẰNG TẢI TRỤC MỞ RỘNG TRÊN NHIỀU TRƯỜNG HỢP**

*Trục X*scaling là một cách phổ biến để mở rộng ứng dụng đơn khối. Hình 1.4 cho thấy cách hoạt động của X-axis scaling. Bạn chạy nhiều phiên bản của ứng dụng sau một bộ cân bằng tải. Bộ cân bằng tải phân phối các yêu cầu giữa N phiên bản giống hệt nhau của ứng dụng. Đây là một cách tuyệt vời để cải thiện khả năng và tính khả dụng của ứng dụng.

**Z-YÊU CẦU TUYẾN ĐƯỜNG TRỤC MỞ RỘNG DỰA TRÊN THUỘC TÍNH CỦA YÊU CẦU**

*Trục Z*scaling cũng chạy nhiều phiên bản của ứng dụng monolith, nhưng không giống như X-axis scaling, mỗi phiên bản chỉ chịu trách nhiệm cho một tập hợp con dữ liệu. Hình 1.5 cho thấy cách hoạt động của Z-axis scaling. Bộ định tuyến trước các phiên bản sử dụng một thuộc tính yêu cầu để định tuyến đến phiên bản thích hợp. Ví dụ, một ứng dụng có thể định tuyến các yêu cầu bằng userId.

Trong ví dụ này, mỗi phiên bản ứng dụng chịu trách nhiệm cho một tập hợp con người dùng.

bộ định tuyến sử dụngID người dùngđược chỉ định bởi yêu cầuỦy quyềntiêu đề để chọn một trong

**Định tuyến các yêu cầu bằng thuật toán cân bằng tải.**

*N*các trường hợp ứng dụng giống hệt nhau

Bộ cân bằng tải

Ứng dụng thể hiện 3

Ứng dụng thể hiện 2

Ví dụ ứng dụng 1

Lời yêu cầu

Khách hàng

**Hình 1.4 Tỷ lệ trục X chạy nhiều trường hợp giống hệt nhau của khối đơnứng dụng đằng sau bộ cân bằng tải.**

**Sử dụng userId để quyết định nơi định tuyến yêu cầu**

*N*các trường hợp ứng dụng giống hệt nhau

Ví dụ ứng dụng 1

Lời yêu cầu:

Người dùng: a–h

LẤY /...

Ứng dụng thể hiện 2

Bộ định tuyến

Khách hàng

Quyền hạn: userId:password

Người dùng: ip

Ứng dụng thể hiện 3

**Mỗi trường hợp đều có trách nhiệm**

**cho một nhóm người dùng.**

Người dùng: r–z

**Hình 1.5Tỷ lệ trục Z chạy nhiều lần giống hệt nhau****các trường hợp ứng dụng độc lập đằng sau bộ định tuyến, định tuyến dựa trênlời yêu cầuthuộc tính. Mỗi trường hợp chịu trách nhiệm cho một tập hợp concủa dữ liệu.**

N phiên bản giống hệt nhau của ứng dụng. Việc mở rộng theo trục Z là một cách tuyệt vời để mở rộng ứng dụng nhằm xử lý khối lượng giao dịch và dữ liệu ngày càng tăng.

**Có-AXIS SCALING PHÂN TÍCH CHỨC NĂNG MỘT ỨNG DỤNG THÀNH CÁC DỊCH VỤ**

Việc mở rộng trục X và Z cải thiện khả năng và tính khả dụng của ứng dụng. Nhưng không có cách tiếp cận nào giải quyết được vấn đề tăng độ phức tạp của phát triển và ứng dụng. ĐểĐể giải quyết những vấn đề đó, bạn cần áp dụng phương pháp chia tỷ lệ trục Y hoặc phân tích chức năng. Hình 1.6 cho thấy cách thức hoạt động của phương pháp chia tỷ lệ trục Y: bằng cách chia một ứng dụng đơn khối thành một tập hợp các dịch vụ.

**Chức năng chia tỷ lệ trục Y phân tích ứng dụng thành các dịch vụ.**

Dịch vụ đặt hàng

Ứng dụng

Lời yêu cầu

Đặt hàng

yêu cầu

Khách hàng

yêu cầu

Ôn tập

yêu cầu

**Mỗi dịch vụ thường được chia tỷ lệ theo trục X và có thể theo trục Z.**

Khách hàng

Bộ cân bằng tải

Dịch vụ đánh giá

Dịch vụ khách hàng

Dịch vụ đặt hàng phiên bản 3

Dịch vụ đặt hàng phiên bản 2

Dịch vụ đặt hàng

Đặt hàng dịch vụ thể hiện 1

**Hình 1.6 Tỷ lệ trục Y chia ứng dụng thành một tập hợp các dịch vụ. Mỗi dịch vụ chịu trách nhiệmmột chức năng cụ thể. Một dịch vụ được chia tỷ lệ bằng cách sử dụng tỷ lệ trục X và có thể là tỷ lệ trục Z.**

Một dịch vụ là một ứng dụng nhỏ thực hiện chức năng tập trung hẹp, chẳng hạn nhưquản lý đơn hàng, quản lý khách hàng, v.v. Một dịch vụ được mở rộng quy mô bằng cách sử dụng tỷ lệ trục X, mặc dù một số dịch vụ cũng có thể sử dụng tỷ lệ trục Z. Ví dụ, dịch vụ Đơn hàng bao gồm một tập hợp các phiên bản dịch vụ được cân bằng tải.

Định nghĩa cấp cao về kiến ​​trúc vi dịch vụ (microservices) là một phong cách kiến ​​trúc phân tách chức năng của một ứng dụng thành một tập hợp các dịch vụ. Lưu ý rằng định nghĩa này không nói gì về kích thước. Thay vào đó, điều quan trọng là mỗi dịch vụ có một tập hợp các trách nhiệm tập trung và gắn kết. Sau này trong cuốn sách, tôi sẽ thảo luận về ý nghĩa của điều đó.

Bây giờ chúng ta hãy xem kiến ​​trúc vi dịch vụ là một dạng mô-đun như thế nào.

###### Các dịch vụ vi mô như một hình thức mô-đun

*Tính mô-đun*là điều cần thiết khi phát triển các ứng dụng lớn, phức tạp. Một ứng dụng hiện đại như FTGO quá lớn để một cá nhân có thể phát triển. Nó cũng quá phức tạp để một người có thể hiểu được. Các ứng dụng phải được phân tách thành các mô-đun được nhiều người khác nhau phát triển và hiểu. Trong một ứng dụng nguyên khối, các mô-đun được định nghĩa bằng cách kết hợp các cấu trúc ngôn ngữ lập trình (như các gói Java) và các hiện vật xây dựng (như các tệp Java JAR). Tuy nhiên, như các nhà phát triển FTGO đã phát hiện ra, cách tiếp cận này có xu hướng không hiệu quả trong thực tế. Các ứng dụng nguyên khối tồn tại lâu dài thường thoái hóa thành những quả bóng bùn lớn.

Kiến trúc microservice sử dụng các dịch vụ như là đơn vị của tính mô-đun. Một dịch vụ có một API, là ranh giới không thể xuyên thủng và khó có thể vi phạm. Bạn không thể bỏ qua

API và truy cập một lớp nội bộ như bạn có thể làm với một gói Java. Do đó, việc bảo toàn tính mô-đun của ứng dụng theo thời gian dễ dàng hơn nhiều. Có những lợi ích khác khi sử dụng các dịch vụ như các khối xây dựng, bao gồm khả năng triển khai và mở rộng chúng một cách độc lập.

###### Mỗi dịch vụ có cơ sở dữ liệu riêng

Một đặc điểm chính của kiến ​​trúc dịch vụ vi mô là các dịch vụ được ghép nối lỏng lẻo và chỉ giao tiếp qua API. Một cách để đạt được sự ghép nối lỏng lẻo là mỗi dịch vụ có kho dữ liệu riêng. Ví dụ, trong cửa hàng trực tuyến, Order Service cócơ sở dữ liệu bao gồm bảng ORDERS và Dịch vụ khách hàng có cơ sở dữ liệu của riêng mình, bao gồm bảng CUSTOMERS. Vào thời điểm phát triển, các nhà phát triển có thể thay đổi lược đồ của dịch vụ mà không cần phải phối hợp với các nhà phát triển đang làm việc trên các dịch vụ khác. Vào thời gian chạy, các dịch vụ được tách biệt với nhau—ví dụ, một dịch vụ sẽ không bao giờ bị chặn vì dịch vụ khác giữ khóa cơ sở dữ liệu.

**Đừng lo lắng: Sự kết hợp lỏng lẻo không làm Larry Ellison giàu hơn**

Yêu cầu mỗi dịch vụ phải có cơ sở dữ liệu riêng không có nghĩa là nó có máy chủ cơ sở dữ liệu riêng. Ví dụ, bạn không phải chi nhiều hơn 10 lần cho giấy phép Oracle RDBMS. Chương 2 sẽ khám phá chủ đề này một cách sâu sắc.

Bây giờ chúng ta đã định nghĩa kiến ​​trúc vi dịch vụ và mô tả một số đặc điểm cần thiết của nó, hãy cùng xem cách kiến ​​trúc này áp dụng vào ứng dụng FTGO.

###### Kiến trúc dịch vụ vi mô FTGO

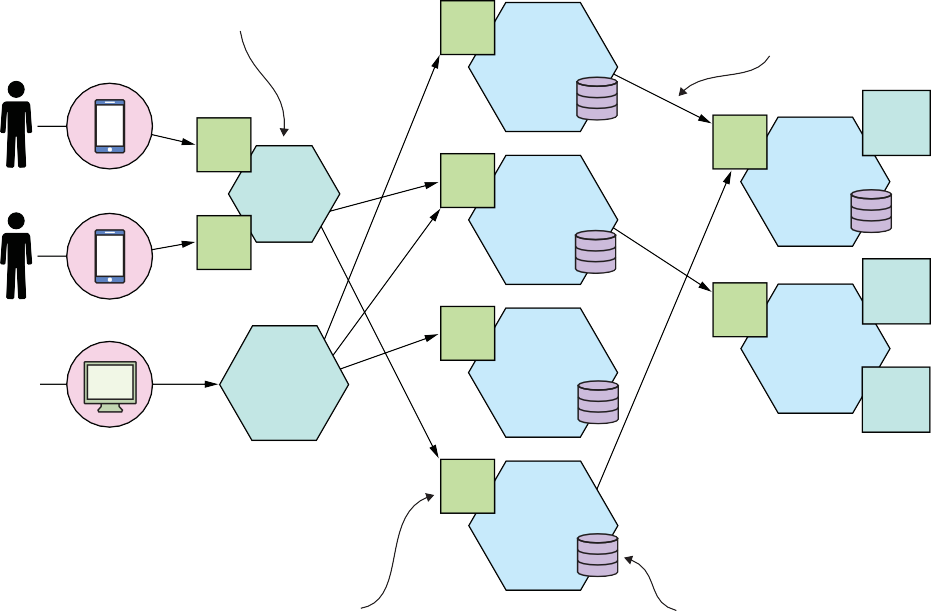
Phần còn lại của cuốn sách này thảo luận sâu về kiến ​​trúc dịch vụ vi mô của ứng dụng FTGO. Nhưng trước tiên, hãy nhanh chóng xem xét ý nghĩa của việc áp dụng tỷ lệ trục Y cho ứng dụng này. Nếu chúng ta áp dụng phân tích trục Y cho ứng dụng FTGO, chúng ta sẽ có được kiến ​​trúc được hiển thị trong hình 1.7. Ứng dụng phân tích bao gồm nhiều dịch vụ frontend và backend. Chúng ta cũng sẽ áp dụng tỷ lệ trục X và có thể là trục Z, để khi chạy sẽ có nhiều phiên bản của mỗi dịch vụ.

Các dịch vụ frontend bao gồm một cổng API và Restaurant Web UI. Cổng API, đóng vai trò là một mặt tiền và được mô tả chi tiết trong chương 8, cung cấp các REST API được sử dụng bởi các ứng dụng di động của người tiêu dùng và người giao hàng. Restaurant Web UI triển khai giao diện web được các nhà hàng sử dụng để quản lý thực đơn và xử lý đơn hàng.

Logic kinh doanh của ứng dụng FTGO bao gồm nhiều dịch vụ phụ trợ. Mỗi dịch vụ phụ trợ có một REST API và kho dữ liệu riêng. Các dịch vụ phụ trợ bao gồm:

* + - * Dịch vụ đặt hàng**—**Quản lý đơn hàng
      * Dịch vụ giao hàng—Quản lý việc giao hàng từ nhà hàng đến người tiêu dùng
        + Dịch vụ nhà hàng—Lưu giữ thông tin về nhà hàng
        + Dịch vụ nhà bếp—Quản lý việc chuẩn bị đơn hàng
        + Dịch vụ kế toán—Tay cầmthanh toán và lập hóa đơn

**API Gateway định tuyến các yêu cầu từ ứng dụng di động đến các dịch vụ.**



NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Dịch vụ đặt hàng

**Các dịch vụ tương ứng với các khả năng kinh doanh/các miền phụ thiết kế theo định hướng miền (DDD)**

Người chuyển phát nhanh

Người tiêu dùng

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Giao diện lập trình ứng dụng (API)

Cổng vào

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Nhà hàngDịch vụ

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Kế toánDịch vụ

Thông báo

Bộ chuyển đổi sọc

Bộ chuyển đổi Twilio

Nhà hàng

Nhà hàngGiao diện người dùng web

Phòng bếp

Dịch vụ

Dịch vụ

Amazon SES

Bộ chuyển đổi

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Vận chuyển

Dịch vụ

**Dịch vụ cóAPI.A dữ liệu của dịch vụ là riêng tư.**

**Hình 1.7 Một số dịch vụ của phiên bản dựa trên kiến ​​trúc vi dịch vụ của ứng dụng FTGO. Một API Gateway định tuyến các yêu cầu từ các ứng dụng di động đến các dịch vụ. Các dịch vụcộng tác thông qua API.**

Nhiều dịch vụ tương ứng với các mô-đun được mô tả trước đó trong chương này. Điểm khác biệt là mỗi dịch vụ và API của nó được định nghĩa rất rõ ràng. Mỗi dịch vụ có thể được phát triển, thử nghiệm, triển khai và mở rộng độc lập. Ngoài ra, kiến ​​trúc này thực hiện tốt việc bảo toàn tính mô-đun. Nhà phát triển không thể bỏ qua API của dịch vụ và truy cập các thành phần bên trong của dịch vụ đó. Chương 13 mô tả cách chuyển đổi ứng dụng đơn khối hiện có thành các dịch vụ siêu nhỏ.

###### So sánh kiến ​​trúc microservice và SOA

Một số nhà phê bình kiến ​​trúc vi dịch vụ cho rằng nó không có gì mới—đó là kiến ​​trúc hướng dịch vụ (SOA). Ở cấp độ rất cao, có một số điểm tương đồng. SOA và kiến ​​trúc vi dịch vụ là các phong cách kiến ​​trúc cấu trúc một hệ thống như một tập hợp các dịch vụ. Nhưng như bảng 1.1 cho thấy, khi bạn đào sâu, bạn sẽ gặp phải những khác biệt đáng kể.

**Bảng 1.1 So sánh SOA với microservices**

|  |  |  |
| --- | --- | --- |
|  | **SOA** | **Dịch vụ vi mô** |
| Giao tiếp giữa các dịch vụ | Các đường ống thông minh, chẳng hạn như Enterprise Service Bus, sử dụng các giao thức nặng như SOAP và các tiêu chuẩn WS\* khác. | Các đường ống câm, chẳng hạn như một môi giới tin nhắn hoặc giao tiếp trực tiếp giữa các dịch vụ, sử dụng các giao thức nhẹ như REST hoặc gRPC |
| Dữ liệu | Mô hình dữ liệu toàn cầu và dữ liệu được chia sẻcăn cứ | Mô hình dữ liệu và cơ sở dữ liệu theo dịch vụ |
| Dịch vụ tiêu biểu | Ứng dụng đơn khối lớn hơn | Dịch vụ nhỏ hơn |

SOA và kiến ​​trúc microservice thường sử dụng các ngăn xếp công nghệ khác nhau. Các ứng dụng SOA thường sử dụng các công nghệ nặng như SOAP và các chuẩn WS\* khác. Chúng thường sử dụng ESB, một đường ống thông minh chứa logic xử lý thông điệp và kinh doanh để tích hợp các dịch vụ. Các ứng dụng được xây dựng bằng kiến ​​trúc microservice có xu hướng sử dụng các công nghệ nhẹ, mã nguồn mở. Các dịch vụ giao tiếp thông qua các đường ống câm, chẳng hạn như các trình môi giới thông điệp hoặc các giao thức nhẹ như REST hoặc gRPC.

SOA và kiến ​​trúc microservice cũng khác nhau về cách chúng xử lý dữ liệu. Các ứng dụng SOA thường có mô hình dữ liệu toàn cục và chia sẻ cơ sở dữ liệu. Ngược lại, như đã đề cập trước đó, trong kiến ​​trúc microservice, mỗi dịch vụ đều có cơ sở dữ liệu riêng. Hơn nữa, như đã mô tả trong chương 2, mỗi dịch vụ thường được coi là có mô hình miền riêng.

Một điểm khác biệt chính giữa SOA và kiến ​​trúc microservice là quy mô của các dịch vụ. SOA thường được sử dụng để tích hợp các ứng dụng lớn, phức tạp, đơn khối. Mặc dù các dịch vụ trong kiến ​​trúc microservice không phải lúc nào cũng nhỏ, nhưng chúng hầu như luôn nhỏ hơn nhiều. Do đó, một ứng dụng SOA thường bao gồm một vài dịch vụ lớn, trong khi một ứng dụng dựa trên microservice thường bao gồm hàng chục hoặc hàng trăm dịch vụ nhỏ hơn.

#### Lợi ích và hạn chế của dịch vụ vi môngành kiến ​​​​trúc

Đầu tiên chúng ta hãy xem xétnhững lợi ích và sau đó chúng ta sẽ xem xét những hạn chế.

###### Lợi ích của kiến ​​trúc vi dịch vụ

Kiến trúc dịch vụ vi mô cónhững lợi ích sau:

* + - * Nó cho phép phân phối và triển khai liên tục các ứng dụng lớn, phức tạp.
      * Các dịch vụ có quy mô nhỏ và dễ bảo trì.
      * Các dịch vụ có thể triển khai độc lập.
      * Các dịch vụ có thể mở rộng độc lập.
      * Kiến trúc vi dịch vụ cho phép các nhóm có tính tự chủ.
      * Nó cho phép thử nghiệm dễ dàngvà áp dụng các công nghệ mới.
      * Nó có khả năng cô lập lỗi tốt hơn.

Hãy cùng xem xét từng lợi ích.

**ENABLES VIỆC GIAO HÀNG VÀ TRIỂN KHAI LIÊN TỤC CÁC,ỨNG DỤNG PHỨC TẠP**

Lợi ích quan trọng nhất của kiến ​​trúc microservice là nó cho phép phân phối và triển khai liên tục các ứng dụng lớn, phức tạp. Như đã mô tả sau trong phần 1.7, phân phối/triển khai liên tục là một phần của DevOps, một tập hợp các hoạt động để phân phối phần mềm nhanh chóng, thường xuyên và đáng tin cậy. Các tổ chức DevOps hiệu suất cao thường triển khai các thay đổi vào sản xuất với rất ít vấn đề sản xuất.

Có ba cách mà kiến ​​trúc vi dịch vụ cho phép triển khai/cung cấp liên tục:

* + - * + *Nó có khả năng kiểm tra được yêu cầu bởi việc phân phối/triển khai liên tục*—Tự độngkiểm thử là một hoạt động chính của việc phân phối/triển khai liên tục. Vì mỗi dịch vụ trong kiến ​​trúc dịch vụ vi mô tương đối nhỏ nên các bài kiểm tra tự động dễ viết hơn và thực hiện nhanh hơn nhiều. Do đó, ứng dụng sẽ có ít lỗi hơn.
        + *Nó có khả năng triển khai cần thiết cho việc phân phối/triển khai liên tục*—Mỗi dịch vụ có thể được triển khai độc lập với các dịch vụ khác. Nếu các nhà phát triển chịu trách nhiệm cho một dịch vụ cần triển khai một thay đổi cục bộ cho dịch vụ đó, họ không cần phải phối hợp với các nhà phát triển khác. Họ có thể triển khai các thay đổi của mình. Do đó, việc triển khai các thay đổi thường xuyên vào sản xuất dễ dàng hơn nhiều.
        + *Nó cho phép các nhóm phát triển có thể tự chủ và liên kết lỏng lẻo*—Bạn có thể cấu trúc tổ chức kỹ thuật như một tập hợp các nhóm nhỏ (ví dụ, two-pizza). Mỗi nhóm chịu trách nhiệm riêng cho việc phát triển và triển khai một hoặc nhiều dịch vụ liên quan. Như hình 1.8 cho thấy, mỗi nhóm có thể phát triển, triển khai và mở rộng quy mô dịch vụ của mình độc lập với tất cả các nhóm khác. Do đó, tốc độ phát triển cao hơn nhiều.

Khả năng thực hiện phân phối và triển khai liên tục có một sốLợi ích kinh doanh:

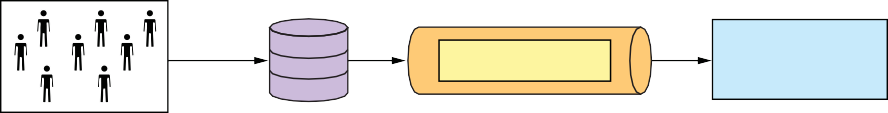
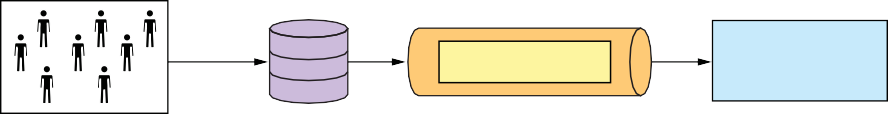
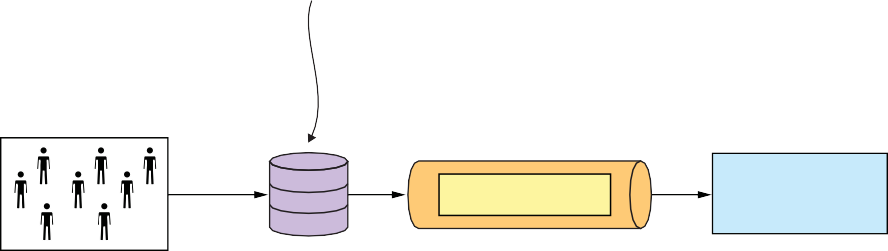
* + - * + Nó rút ngắn thời gian đưa sản phẩm ra thị trường, cho phép doanh nghiệp phản ứng nhanh chóng với phản hồi từ khách hàng.
        + Nó cho phép doanh nghiệp cung cấp loại dịch vụ đáng tin cậy mà khách hàng ngày nay mong đợi.
        + Sự hài lòng của nhân viên cao hơn vì họ dành nhiều thời gian hơn để cung cấp các tính năng có giá trị thay vì giải quyết vấn đề.

Kết quả là, kiến ​​trúc vi dịch vụ đã trở thành nền tảng cơ bản của mọi doanh nghiệp phụ thuộc vào công nghệ phần mềm.

**EDỊCH VỤ ACH NHỎ VÀ DỄ DÀNG BẢO TRÌ**

Một lợi ích khác của kiến ​​trúc microservice là mỗi dịch vụ tương đối nhỏ. Mã dễ hiểu hơn đối với nhà phát triển. Cơ sở mã nhỏ không làm chậm IDE, giúp nhà phát triển làm việc hiệu quả hơn. Và mỗi dịch vụ thường khởi động nhanh hơn nhiều so với một khối lớn, điều này cũng giúp nhà phát triển làm việc hiệu quả hơn và tăng tốc triển khai.

**Các nhóm nhỏ, tự chủ, liên kết lỏng lẻo**



**Mỗi dịch vụ đều có kho mã nguồn riêng.**

**Mỗi dịch vụ đều có quy trình triển khai tự động riêng.**

**Dịch vụ nhỏ, đơn giản, đáng tin cậy, dễ bảo trì**

Phát triển FTGO

Đội ngũ quản lý đơn hàng

Đường ống triển khai

JenkinsDịch vụ ClOrder

Đội ngũ quản lý nhà hàng

Dịch vụ đặt hàngkho lưu trữ mã nguồn

Đường ống triển khai

JenkinsClDịch vụ nhà hàng

Đội ngũ quản lý giao hàng

Dịch vụ nhà hàngkho lưu trữ mã nguồn

Đường ống triển khai

JenkinsDịch vụ giao hàng

Dịch vụ giao hàngkho lưu trữ mã nguồn

Sản xuất

**Hình 1.8 Ứng dụng FTGO dựa trên dịch vụ vi mô bao gồm một tập hợp các dịch vụ được kết nối lỏng lẻo.Mỗi nhóm phát triển, thử nghiệm và triển khai dịch vụ của mình một cách độc lập.**

**SDỊCH VỤ CÓ THỂ MỞ RỘNG ĐỘC LẬP**

Mỗi dịch vụ trong kiến ​​trúc dịch vụ vi mô có thể được mở rộng độc lập với các dịch vụ khác bằng cách sử dụng sao chép trục X và phân vùng trục Z. Hơn nữa, mỗi dịch vụ có thể được triển khai trên phần cứng phù hợp nhất với yêu cầu tài nguyên của nó. Điều này khá khác so với khi sử dụng kiến ​​trúc đơn khối, trong đó các thành phần có yêu cầu tài nguyên khác nhau rất nhiều—ví dụ, sử dụng nhiều CPU so với sử dụng nhiều bộ nhớ—phải được triển khai cùng nhau.

**BCÁCH LY LỖI ETTER**

Kiến trúc microservice có khả năng cô lập lỗi tốt hơn. Ví dụ, rò rỉ bộ nhớ trong một dịch vụ chỉ ảnh hưởng đến dịch vụ đó. Các dịch vụ khác sẽ tiếp tục xử lý các yêu cầu bình thường. Ngược lại, một thành phần hoạt động không tốt của kiến ​​trúc monolithic sẽ làm sập toàn bộ hệ thống.

**EASILY THỬ NGHIỆM VÀ ÁP DỤNG CÔNG NGHỆ MỚI**

Cuối cùng nhưng không kém phần quan trọng, kiến ​​trúc microservice loại bỏ mọi cam kết dài hạn đối với một ngăn xếp công nghệ. Về nguyên tắc, khi phát triển một dịch vụ mới, các nhà phát triển được tự do lựa chọn bất kỳ ngôn ngữ và khuôn khổ nào phù hợp nhất với dịch vụ đó.

Trong nhiều tổ chức, việc hạn chế các lựa chọn là điều hợp lý, nhưng điều quan trọng là bạn không bị ràng buộc bởi các quyết định trong quá khứ.

Hơn nữa, vì các dịch vụ nhỏ, nên việc viết lại chúng bằng các ngôn ngữ và công nghệ tốt hơn trở nên thực tế. Nếu thử nghiệm công nghệ mới không thành công, bạn có thể bỏ công việc đó mà không gây rủi ro cho toàn bộ dự án. Điều này khá khác so với khi sử dụng kiến ​​trúc đơn khối, khi các lựa chọn công nghệ ban đầu của bạn hạn chế nghiêm trọng khả năng sử dụng các ngôn ngữ và khuôn khổ khác nhau trong tương lai.

###### Nhược điểm của kiến ​​trúc microservice

Chắc chắn, không có công nghệ nào là giải pháp hoàn hảo, và kiến ​​trúc microservice có một số nhược điểm và vấn đề đáng kể. Thực tế, phần lớn cuốn sách này nói về cách giải quyết những nhược điểm và vấn đề này. Khi bạn đọc về những thách thức, đừng lo lắng. Sau đó trong cuốn sách này, tôi sẽ mô tả các cách giải quyết chúng.

Sau đây là những nhược điểm và vấn đề chính của kiến ​​trúc vi dịch vụ:

* + - * Tìm đúng bộ dịch vụlà một thách thức.
      * Hệ thống phân tán rất phức tạp, khiến cho việc phát triển, thử nghiệm và triển khai trở nên khó khăn.
      * Triển khai các tính năngbao gồm nhiều dịch vụ đòi hỏi sự phối hợp cẩn thận.
      * Quyết định khi nàoáp dụng kiến ​​trúc microservice là rất khó. Chúng ta hãy cùng xem xét từng kiến ​​trúc một.

**FTÌM KIẾM CÁC DỊCH VỤ ĐÚNG LÀ MỘT THÁCH THỨC**

Một thách thức khi sử dụng kiến ​​trúc microservice là không có thuật toán cụ thể, được xác định rõ ràng để phân tích một hệ thống thành các dịch vụ. Cũng giống như nhiều quá trình phát triển phần mềm, đây là một nghệ thuật. Tệ hơn nữa, nếu bạn phân tích một hệ thống không đúng cách, bạn sẽ xây dựng một khối đơn phân tán, một hệ thống bao gồm các dịch vụ được ghép nối phải được triển khai cùng nhau. Một khối đơn phân tán có những nhược điểm của cả kiến ​​trúc monolithic và kiến ​​trúc microservice.

**DHỆ THỐNG PHÂN PHỐI RẤT PHỨC TẠP**

Một vấn đề khác khi sử dụng kiến ​​trúc microservice là các nhà phát triển phải giải quyết thêm sự phức tạp của việc tạo ra một hệ thống phân tán. Các dịch vụ phải sử dụng cơ chế giao tiếp giữa các tiến trình. Điều này phức tạp hơn một lệnh gọi phương thức đơn giản. Hơn nữa, một dịch vụ phải được thiết kế để xử lý lỗi một phần và giải quyết tình trạng dịch vụ từ xa không khả dụng hoặc có độ trễ cao.

Việc triển khai các trường hợp sử dụng trải dài trên nhiều dịch vụ đòi hỏi phải sử dụng các kỹ thuật không quen thuộc. Mỗi dịch vụ có cơ sở dữ liệu riêng, khiến việc triển khai các giao dịch và truy vấn trải dài trên nhiều dịch vụ trở thành một thách thức. Như đã mô tả trong chương 4, một ứng dụng dựa trên dịch vụ vi mô phải sử dụng cái được gọi là sagas để duy trì tính nhất quán của dữ liệu trên các dịch vụ. Chương 7 giải thích rằng một ứng dụng dựa trên dịch vụ vi mô không thể truy xuất dữ liệu từ nhiều dịch vụ bằng các truy vấn đơn giản. Thay vào đó, ứng dụng đó phải triển khai các truy vấn bằng cách sử dụng API composition hoặc chế độ xem CQRS.

IDE và các công cụ phát triển khác tập trung vào việc xây dựng các ứng dụng đơn khối và không cung cấp hỗ trợ rõ ràng cho việc phát triển các ứng dụng phân tán. Viết các bài kiểm tra tự động liên quan đến nhiều dịch vụ là một thách thức. Đây đều là những vấn đề cụ thể đối với kiến ​​trúc dịch vụ vi mô. Do đó, các nhà phát triển của tổ chức bạn phải có kỹ năng phát triển và cung cấp phần mềm tinh vi để sử dụng thành công các dịch vụ vi mô.

Kiến trúc microservice cũng đưa vào sự phức tạp đáng kể trong hoạt động. Nhiều bộ phận chuyển động hơn—nhiều trường hợp của các loại dịch vụ khác nhau—phải được quản lý trong sản xuất. Để triển khai thành công microservice, bạn cần mức độ tự động hóa cao. Bạn phải sử dụng các công nghệ như sau:

* Triển khai tự độngcông cụ, như Netflix Spinnaker
* Một PaaS có sẵn, như Pivotal Cloud Foundry hoặc Red Hat OpenShift
* Nền tảng điều phối Docker, như Docker Swarmhoặc Kubernetes Tôi sẽ mô tả chi tiết hơn về các tùy chọn triển khai trong chương 12.

**DSỬ DỤNG CÁC TÍNH NĂNG TRÊN NHIỀU DỊCH VỤ CẦN PHỐI HỢP CẨN THẬN**

Một thách thức khác khi sử dụng kiến ​​trúc microservice là việc triển khai các tính năng trải rộng trên nhiều dịch vụ đòi hỏi sự phối hợp chặt chẽ giữa các nhóm phát triển khác nhau. Bạn phải tạo một kế hoạch triển khai để sắp xếp các triển khai dịch vụ dựa trên sự phụ thuộc giữa các dịch vụ. Điều đó khá khác so với kiến ​​trúc monolithic, nơi bạn có thể dễ dàng triển khai các bản cập nhật cho nhiều thành phần một cách nguyên tử.

**DQUYẾT ĐỊNH THỜI ĐIỂM NÀO ĐỂ NHẬN NUÔI LÀ KHÓ KHĂN**

Một vấn đề khác khi sử dụng kiến ​​trúc vi dịch vụ là quyết định thời điểm nào trongvòng đời của ứng dụng bạn nên sử dụng kiến ​​trúc này. Khi phát triển phiên bản đầu tiên của ứng dụng, bạn thường không gặp phải những vấn đề mà kiến ​​trúc này giải quyết. Hơn nữa, sử dụng kiến ​​trúc phân tán phức tạp sẽ làm chậm quá trình phát triển. Đó có thể là một tình thế tiến thoái lưỡng nan lớn đối với các công ty khởi nghiệp, nơi mà vấn đề lớn nhất thường là làm thế nào để phát triển nhanh chóng mô hình kinh doanh và ứng dụng đi kèm. Sử dụng kiến ​​trúc vi dịch vụ khiến việc lặp lại nhanh chóng trở nên khó khăn hơn nhiều. Một công ty khởi nghiệp gần như chắc chắn nên bắt đầu bằng một ứng dụng đơn khối.

Tuy nhiên, sau đó, khi vấn đề là cách xử lý độ phức tạp, thì đó là lúc việc phân tích chức năng ứng dụng thành một tập hợp các dịch vụ siêu nhỏ là hợp lý. Bạn có thể thấy việc tái cấu trúc khó khăn vì các phụ thuộc phức tạp. Chương 13 sẽ đề cập đến các chiến lược để tái cấu trúc một ứng dụng đơn khối thành các dịch vụ siêu nhỏ.

Như bạn có thể thấy, kiến ​​trúc vi dịch vụ mang lại nhiều lợi ích, nhưng cũng có một sốnhững nhược điểm đáng kể. Vì những vấn đề này, việc áp dụng kiến ​​trúc vi dịch vụ không nên được thực hiện một cách dễ dàng. Nhưng đối với các ứng dụng phức tạp, chẳng hạn như ứng dụng web hướng đến người tiêu dùng hoặc ứng dụng SaaS, thì đây thường là lựa chọn đúng đắn. Các trang web nổi tiếng như eBay ([www.slideshare.net/RandyShoup/the-ebay-architecture-striking-a-](http://www.slideshare.net/RandyShoup/the-ebay-architecture-striking-a-balance-between-site-stability-feature-velocity-performance-and-cost) [sự cân bằng giữa tính ổn định của trang web, tính năng, tốc độ, hiệu suất và chi phí](http://www.slideshare.net/RandyShoup/the-ebay-architecture-striking-a-balance-between-site-stability-feature-velocity-performance-and-cost)),[Amazon.com](http://Amazon.com/)Groupon và Gilt đều đã phát triển từ kiến ​​trúc đơn khối sang kiến ​​trúc dịch vụ vi mô.

Bạn phải giải quyết nhiều vấn đềcác vấn đề về thiết kế và kiến ​​trúc khi sử dụng kiến ​​trúc vi dịch vụ. Hơn nữa, nhiều vấn đề trong số này có nhiều giải pháp, mỗi giải pháp có một tập hợp các đánh đổi khác nhau. Không có một giải pháp hoàn hảo duy nhất nào. Để giúp bạn đưa ra quyết định, tôi đã tạo ra ngôn ngữ mẫu kiến ​​trúc vi dịch vụ. Tôi tham chiếu ngôn ngữ mẫu này trong suốt phần còn lại của cuốn sách khi tôi hướng dẫn bạn về kiến ​​trúc vi dịch vụ. Hãy cùng xem ngôn ngữ mẫu là gì và tại sao nó hữu ích.

#### Ngôn ngữ mẫu kiến ​​trúc Microservice

Kiến trúc và thiết kế đều liên quan đến việc đưa ra quyết định. Bạn cần quyết định xem kiến ​​trúc monolithic hay microservice phù hợp nhất với ứng dụng của mình. Khi đưa ra những quyết định này, bạn phải cân nhắc rất nhiều sự đánh đổi. Nếu bạn chọn kiến ​​trúc microservice, bạn sẽ cần giải quyết rất nhiều vấn đề.

Một cách tốt để mô tả các tùy chọn kiến ​​trúc và thiết kế khác nhau và cải thiện việc ra quyết định là sử dụng ngôn ngữ mẫu. Trước tiên, hãy xem lý do tại sao chúng ta cần các mẫu và ngôn ngữ mẫu, sau đó chúng ta sẽ tham quan ngôn ngữ mẫu kiến ​​trúc Microservice.

###### Kiến trúc microservice không phải là bạcviên đạn

Quay trở lại năm 1986, Fred Brooks,tác giả của The Mythical Man-Month (Addison-Wesley Profes- sional, 1995), đã nói rằng trong kỹ thuật phần mềm, không có viên đạn bạc nào cả. Điều đó có nghĩa là không có kỹ thuật hoặc công nghệ nào nếu áp dụng sẽ giúp bạn tăng gấp mười lần năng suất. Tuy nhiên, nhiều thập kỷ sau, các nhà phát triển vẫn đang tranh luận sôi nổi về viên đạn bạc yêu thích của họ, hoàn toàn tin rằng công nghệ yêu thích của họ sẽ giúp họ tăng năng suất đáng kể.

Rất nhiều lập luận theo sự phân đôi giữa hút và đá ([http://nealford.com/memeagora/](http://nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html) [2009/08/05/suck-rock-dichotomy.html](http://nealford.com/memeagora/2009/08/05/suck-rock-dichotomy.html)), một thuật ngữ do Neal Ford đặt ra để mô tả cách mọi thứ trong thế giới phần mềm hoặc là tệ hoặc là tuyệt vời, không có điểm trung gian. Những lập luận này có cấu trúc như sau: nếu bạn làm X, thì một chú chó con sẽ chết, do đó bạn phải làm Y. Ví dụ, lập trình đồng bộ so với lập trình phản ứng, hướng đối tượng so với chức năng, Java so với JavaScript, REST so với nhắn tin. Tất nhiên, thực tế phức tạp hơn nhiều. Mọi công nghệ đều có nhược điểm và hạn chế mà những người ủng hộ thường bỏ qua. Do đó, việc áp dụng một công nghệ thường tuân theo chu kỳ cường điệu của Gartner (<https://en.wikipedia.org/wiki/Hype_cycle>), trong đó một công nghệ mới nổi trải qua năm giai đoạn, bao gồm đỉnh điểm của kỳ vọng quá mức (thật tuyệt vời), tiếp theo là đáy của sự vỡ mộng (thật tệ), và kết thúc bằng sự ổn định của năng suất (bây giờ chúng ta đã hiểu được sự đánh đổi và thời điểm nên sử dụng nó).

Các dịch vụ vi mô không miễn nhiễm với hiện tượng viên đạn bạc. Kiến trúc này có phù hợp với ứng dụng của bạn hay không phụ thuộc vào nhiều yếu tố. Do đó, lời khuyên không nên luôn sử dụng kiến ​​trúc dịch vụ vi mô là một lời khuyên tồi, nhưng lời khuyên không bao giờ sử dụng nó cũng là một lời khuyên tồi. Cũng như nhiều thứ khác, điều này còn tùy thuộc.

Lý do cơ bản cho những lập luận phân cực và cường điệu này về công nghệ làrằng con người chủ yếu bị thúc đẩy bởi cảm xúc của họ. Jonathan Haidt, trong tác phẩm tuyệt vời của mình

cuốn sách The Righteous Mind:*Tại sao những người tốt lại bị chia rẽ bởi chính trị và tôn giáo*(Vintage, 2013), sử dụng phép ẩn dụ về một con voi và người cưỡi voi để mô tả cách thức hoạt động của tâm trí con người. Con voi đại diện cho phần cảm xúc của bộ não con người. Nó đưa ra hầu hết các quyết định. Người cưỡi voi đại diện cho phần lý trí của bộ não. Đôi khi nó có thể ảnh hưởng đến con voi, nhưng chủ yếu là cung cấp lý do biện minh cho các quyết định của con voi.

Chúng ta—cộng đồng phát triển phần mềm—cần vượt qua bản chất cảm xúc của mình và tìm ra cách tốt hơn để thảo luận và áp dụng công nghệ. Một cách tuyệt vời để thảo luận và mô tả công nghệ là sử dụng định dạng mẫu, vì nó khách quan. Khi mô tả một công nghệ theo định dạng mẫu, ví dụ, bạn phải mô tả những nhược điểm. Hãy cùng xem xét định dạng mẫu.

###### Các mẫu và ngôn ngữ mẫu

Mẫu là giải pháp có thể tái sử dụng cho một vấn đề xảy ra trong một bối cảnh cụ thể. Đây là ý tưởng có nguồn gốc từ kiến ​​trúc thực tế và đã được chứng minh là hữu ích trong thiết kế và kiến ​​trúc phần mềm. Khái niệm mẫu được tạo ra bởi Christopher Alexander, một kiến ​​trúc sư thực tế. Ông cũng tạo ra khái niệm ngôn ngữ mẫu, một tập hợp các mẫu có liên quan để giải quyết các vấn đề trong một phạm vi cụ thể. Cuốn sách Ngôn ngữ mẫu: Thị trấn, Tòa nhà, Xây dựng (Nhà xuất bản Đại học Oxford, 1977) của ông mô tả một ngôn ngữ mẫu cho kiến ​​trúc bao gồm 253 mẫu. Các mẫu bao gồm từ các giải pháp cho các vấn đề cấp cao, chẳng hạn như xác định vị trí của một thành phố ("Tiếp cận nguồn nước"), đến các vấn đề cấp thấp, chẳng hạn như cách thiết kế một căn phòng ("Ánh sáng ở hai bên của mỗi phòng"). Mỗi mẫu này giải quyết một vấn đề bằng cách sắp xếp các đối tượng vật lý có phạm vi từ thành phố đến cửa sổ.

Các bài viết của Christopher Alexander đã truyền cảm hứng cho cộng đồng phần mềm áp dụng khái niệm về các mẫu và ngôn ngữ mẫu. Cuốn sách Design Patterns: Elements of Reus-able Object-Oriented Software (Addison-Wesley Professional, 1994), của Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides là một bộ sưu tập các mẫu thiết kế hướng đối tượng. Cuốn sách đã phổ biến các mẫu trong số các nhà phát triển phần mềm. Kể từ giữa những năm 1990, các nhà phát triển phần mềm đã ghi lại nhiều mẫu phần mềm. Một mẫu phần mềm giải quyết một vấn đề về kiến ​​trúc hoặc thiết kế phần mềm bằng cách xác định một tập hợp các thành phần phần mềm cộng tác.

Ví dụ, hãy tưởng tượng rằng bạn đang xây dựng một ứng dụng ngân hàng phải hỗ trợ nhiều chính sách thấu chi khác nhau. Mỗi chính sách xác định giới hạn về số dư của một tài khoản và phí tính cho một tài khoản thấu chi. Bạn có thể giải quyết vấn đề này bằng cách sử dụng mẫu Strategy, đây là một mẫu nổi tiếng trong sách Design Patterns kinh điển. Giải pháp được xác định bởi mẫu Strategy bao gồm ba phần:

* + - * Một chiến lượcgiao diện được gọi làThấu chibao gồm thuật toán thấu chi
      * Một hoặc nhiều lớp chiến lược cụ thể, một lớp cho mỗi bối cảnh cụ thể
      * CácTài khoảnlớp sử dụng thuật toán

Mẫu Strategy là mẫu thiết kế hướng đối tượng, do đó các thành phần của giải pháp là các lớp. Sau trong phần này, tôi sẽ mô tả các mẫu thiết kế cấp cao, trong đó giải pháp bao gồm các dịch vụ cộng tác.

Một lý do tại sao các mẫu có giá trị là vì một mẫu phải mô tả ngữ cảnh mà nó áp dụng. Ý tưởng rằng một giải pháp là cụ thể cho một ngữ cảnh cụ thể và có thể không hoạt động tốt trong các ngữ cảnh khác là một cải tiến so với cách công nghệ thường được thảo luận. Ví dụ, một giải pháp giải quyết vấn đề ở quy mô của Netflix có thể không phải là cách tiếp cận tốt nhất cho một ứng dụng có ít người dùng hơn.

Tuy nhiên, giá trị của một mẫu vượt xa việc yêu cầu bạn xem xét bối cảnh của một vấn đề. Nó buộc bạn phải mô tả các khía cạnh quan trọng khác nhưng thường bị bỏ qua của một giải pháp. Một cấu trúc mẫu thường được sử dụng bao gồm ba phần đặc biệt có giá trị:

* + - * + Lực lượng
        + Kết quảbối cảnh
        + Các mẫu liên quan

Chúng ta hãy xem xét từng cái một, bắt đầuvới lực lượng.

**FORCE: TNHỮNG VẤN ĐỀ BẠN PHẢI XỬ LÝ KHI GIẢI QUYẾT MỘT VẤN ĐỀ**

Phần lực của một mẫu mô tả các lực (vấn đề) mà bạn phải giải quyết khi giải quyết một vấn đề trong một bối cảnh nhất định. Các lực có thể xung đột, do đó có thể không thể giải quyết tất cả các lực đó. Lực nào quan trọng hơn tùy thuộc vào bối cảnh. Bạn phải ưu tiên giải quyết một số lực hơn các lực khác. Ví dụ, mã phải dễ hiểu và có hiệu suất tốt. Mã được viết theo phong cách phản ứng có hiệu suất tốt hơn mã đồng bộ, nhưng thường khó hiểu hơn. Việc liệt kê rõ ràng các lực rất hữu ích vì nó làm rõ những vấn đề nào cần được giải quyết.

**RBỐI CẢNH KẾT QUẢ: THẬU QUẢ CỦA VIỆC ÁP DỤNG MỘT MẪU**

Phần ngữ cảnh kết quả của một mẫu mô tả hậu quả của việc áp dụng mẫu. Nó bao gồm ba phần:

* + - * + *Những lợi ích*—Những lợi ích của mô hình, bao gồm các lực đã được giải quyết
        + *Nhược điểm*—Những hạn chế của mô hình, bao gồm các lực chưa được giải quyết
        + *Vấn đề*—Những vấn đề mới đã được đưa ra khi áp dụng mô hình

Bối cảnh kết quả cung cấp góc nhìn hoàn thiện hơn và ít thiên vị hơn về giải pháp, giúp đưa ra quyết định thiết kế tốt hơn.

**RMẪU VUI VẺ: TNĂM LOẠI MỐI QUAN HỆ KHÁC NHAU**

Phần mẫu liên quan của một mẫu mô tả mối quan hệgiữa mẫu và các mẫu khác. Có năm loại mối quan hệ giữa các mẫu:

* + - * + *Người tiền nhiệm*—Mẫu tiền nhiệm là mẫu thúc đẩy nhu cầu về mẫu này. Ví dụ, mẫu kiến ​​trúc Microservice là mẫu tiền nhiệm của các mẫu còn lại trong ngôn ngữ mẫu, ngoại trừ mẫu kiến ​​trúc monolithic.
        + *Người kế nhiệm*—Một mẫu giải quyết vấn đề đã được đưa ra bởi mẫu này. Ví dụ, nếu bạn áp dụng mẫu kiến ​​trúc Microservice, bạn phải

sau đó áp dụng nhiều mẫu kế thừa, bao gồm mẫu khám phá dịch vụ và mẫu ngắt mạch.

* + - * *Thay thế*—Một mẫu cung cấp giải pháp thay thế cho mẫu này. Ví dụ, mẫu kiến ​​trúc Monolithic và mẫu kiến ​​trúc Microservice là những cách thay thế để thiết kế ứng dụng. Bạn chọn một trong hai.
      * *Tổng quát hóa*—Một mô hình là giải pháp chung cho một vấn đề. Ví dụ, trong chương 12, bạn sẽ tìm hiểu về các triển khai khác nhau của mô hình Dịch vụ đơn trên mỗi máy chủ.
      * *Chuyên môn hóa*—Một dạng chuyên biệt của một mẫu cụ thể. Ví dụ, trong chương 12, bạn sẽ học rằng mẫu Triển khai dịch vụ dưới dạng vùng chứa là một dạng chuyên biệt của Dịch vụ đơn lẻ trên mỗi máy chủ.

Ngoài ra, bạn có thể sắp xếp các mẫu giải quyết các vấn đề trong một lĩnh vực vấn đề cụ thể thành các nhóm. Mô tả rõ ràng về các mẫu liên quan cung cấp hướng dẫn có giá trị về cách giải quyết hiệu quả một vấn đề cụ thể. Hình 1.9 cho thấy mối quan hệ giữa các mẫu được thể hiện trực quan như thế nào.

Chìa khóa

Người tiền nhiệm

Thay thếMỘT

Tổng quan

Người kế nhiệm

Thay thếB Cụ thể

Kiến trúc nguyên khối



**Mẫu**

Triển khai

Kiến trúc vi dịch vụ

|  |  |
| --- | --- |
| Đơndịch vụ cho mỗi máy chủ | |
|  |  |
| Dịch vụ theo container | |

**Vấn đềkhu vực**

**Hình 1.9 Biểu diễn trực quan các loại mối quan hệ khác nhau giữa các mẫu: a*người kế nhiệm*mẫu giải quyết một vấn đề được tạo ra bằng cách áp dụng*người tiền nhiệm*mẫu; hai hoặc nhiều mẫu có thể được*thay thế*giải pháp cho cùng một vấn đề; một mô hình có thể là một*chuyên môn hóa*của một mô hình khác; và các mô hình giải quyết các vấn đề trong cùng một lĩnh vực có thể được nhóm lại hoặc*tổng quát*.**

Các loại mối quan hệ khác nhau giữa các mẫu được thể hiện trong hình 1.9 được biểu diễn như sau:

* + - * Biểu thị mối quan hệ tiền nhiệm-người kế nhiệm
      * Các mô hình là giải pháp thay thế cho cùng một vấn đề
      * Chỉ ra rằng một mẫu là sự chuyên môn hóa của một mẫu khác
      * Các mẫu áp dụng cho một khu vực vấn đề cụ thể

Một tập hợp các mẫu liên quan thông qua các mối quan hệ này đôi khi tạo thành cái được gọi là ngôn ngữ mẫu. Các mẫu trong ngôn ngữ mẫu hoạt động cùng nhau để giải quyết các vấn đề trong một miền cụ thể. Cụ thể, tôi đã tạo ra ngôn ngữ mẫu kiến ​​trúc Microservice. Đây là tập hợp các kiến ​​trúc phần mềm và mẫu thiết kế có liên quan với nhau cho các microservice. Chúng ta hãy cùng xem ngôn ngữ mẫu này.

###### Tổng quan về ngôn ngữ mẫu kiến ​​trúc Microservice

Ngôn ngữ mẫu kiến ​​trúc Microservice là tập hợp các mẫu giúp bạn thiết kế ứng dụng bằng kiến ​​trúc microservice. Hình 1.10 cho thấy cấu trúc cấp cao của ngôn ngữ mẫu. Ngôn ngữ mẫu đầu tiên giúp bạn quyết định có nên sử dụng kiến ​​trúc microservice hay không. Ngôn ngữ này mô tả kiến ​​trúc đơn khối và kiến ​​trúc microservice, cùng với các lợi ích và hạn chế của chúng. Sau đó, nếu kiến ​​trúc microservice phù hợp với ứng dụng của bạn, ngôn ngữ mẫu sẽ giúp bạn sử dụng hiệu quả bằng cách giải quyết nhiều vấn đề về kiến ​​trúc và thiết kế.

Ngôn ngữ mẫu bao gồm một số nhóm mẫu. Bên trái trong hình 1.10là nhóm các mẫu kiến ​​trúc ứng dụng, mẫu kiến ​​trúc Monolithic và mẫu kiến ​​trúc Microservice. Đó là những mẫu mà chúng ta đã thảo luận

Chìa khóa Mẫu ứng dụng

Người tiền nhiệm

Thay thếMỘT

Tổng quan

Người kế nhiệm

Phương án BCụ thể

Cơ sở dữ liệu

Phân hủy ngành kiến ​​​​trúc

Duy trì

tính nhất quán của dữ liệu

Kiểm tra

Truy vấn

Các mẫu cơ sở hạ tầng ứng dụng

Tin nhắn giao dịch

Khả năng quan sát

Bảo vệ

Giao tiếpphong cáchĐộ tin cậy

Mẫu cơ sở hạ tầng

Khám phá

Triển khai

Bên ngoài

Giao diện lập trình ứng dụng (API)

Mẫu giao tiếp

Các mẫu dịch vụ vi mô

Kiến trúc ứng dụng

Kiến trúc vi dịch vụ

Kiến trúc nguyên khối

Khu vực có vấn đề

Mối quan tâm xuyên suốt

**Hình 1.10 Góc nhìn cấp cao về ngôn ngữ mẫu kiến ​​trúc Microservice cho thấy các lĩnh vực vấn đề khác nhau mà các mẫu giải quyết. Bên trái là các mẫu kiến ​​trúc ứng dụng: Kiến trúc đơn khối và Kiến trúc Microservice. Tất cả các nhóm mẫu khác giải quyết các vấn đề phát sinh từ việc lựa chọnMẫu kiến ​​trúc vi dịch vụ.**

trong chương này. Phần còn lại của ngôn ngữ mẫu bao gồm các nhóm mẫu là giải pháp cho các vấn đề được đưa ra bằng cách sử dụng mẫu kiến ​​trúc Microservice.

Các mẫu cũng được chia thành ba lớp:

* *Mẫu cơ sở hạ tầng*—Những điều này giải quyết các vấn đề chủ yếu là vấn đề về cơ sở hạ tầngbên ngoài sự phát triển.
* *Cơ sở hạ tầng ứng dụng*—Đây là những vấn đề về cơ sở hạ tầng cũngphát triển tác động.
* *Mẫu ứng dụng*—Những giải pháp này giải quyết các vấn đề mà các nhà phát triển gặp phải.

Các mẫu này được nhóm lại với nhau dựa trên loại vấn đề mà chúng giải quyết. Hãy cùng xem xét các nhóm mẫu chính.

**PCÁC BIỂU MỤC ĐỂ PHÂN TÍCH ỨNG DỤNG THÀNH CÁC DỊCH VỤ**

Quyết định cách phân tích một hệ thống thành một tập hợp các dịch vụ là một nghệ thuật, nhưng có một số chiến lược có thể giúp ích. Hai mẫu phân tích được hiển thị trong hình 1.11 là các chiến lược khác nhau mà bạn có thể sử dụng để xác định kiến ​​trúc ứng dụng của mình.

Chìa khóa



Người tiền nhiệm

Phương án A

Tổng quan

Người kế nhiệm

Phương án BCụ thể

Khu vực có vấn đề

Phân tích theo năng lực kinh doanh

**Hình 1.11 Có hai mô hình phân tách: Phân tách theo khả năng kinh doanh, tổ chức các dịch vụ xung quanh các khả năng kinh doanh và Phân tách theo miền phụ, tổ chức các dịch vụ xung quanh các miền phụ thiết kế theo miền (DDD).**

Phân hủytheo tên miền phụ

Chương 2 mô tả chi tiết các mô hình này.

**CCÁC MẪU GIAO TIẾP**

Một ứng dụng được xây dựng bằng kiến ​​trúc microservice là một hệ thống phân tán. Do đó, giao tiếp giữa các tiến trình (IPC) là một phần quan trọng của kiến ​​trúc microservice. Bạn phải đưa ra nhiều quyết định về kiến ​​trúc và thiết kế về cách các dịch vụ của bạn giao tiếp với nhau và với thế giới bên ngoài. Hình 1.12 cho thấy các mẫu giao tiếp, được tổ chức thành năm nhóm:

* *Phong cách giao tiếp*—Bạn nên sử dụng cơ chế IPC nào?
* *Khám phá*—Làm thế nào để một khách hàng củamột dịch vụ xác định địa chỉ IP của một phiên bản dịch vụ để, ví dụ, thực hiện một yêu cầu HTTP?
* *Độ tin cậy*—Làm sao bạn có thể đảm bảorằng việc giao tiếp giữa các dịch vụ là đáng tin cậy ngay cả khi các dịch vụ có thể không khả dụng?
* *Tin nhắn giao dịch*—Bạn nên tích hợp việc gửi tin nhắn và xuất bản sự kiện với các giao dịch cơ sở dữ liệu cập nhật dữ liệu kinh doanh như thế nào?
* *API bên ngoài*—Khách hàng của bạn thế nào?ứng dụng giao tiếp với các dịch vụ?

Chìa khóa

Giao dịchhộp thư đi

Giao dịchkhai thác gỗ

Nhà xuất bản thăm dò ý kiến

Tin nhắn giao dịch



Người tiền nhiệm

Phương án A

Tổng quan

Người kế nhiệm

Phương án BCụ thể

Khu vực có vấn đề





Phong cách giao tiếp

Cụ thể theo miền

Xathủ tục gọi

Nhắn tin

Độ tin cậy

Bộ ngắt mạch

Khám phá

Đăng ký của bên thứ 3

Khám phá phía máy chủ

Đăng ký dịch vụ

Tự đăng ký

Khám phá phía máy khách

API bên ngoài

Backend cho frontend

Cổng API

**Hình 1.12 Năm nhóm mô hình giao tiếp**

Chương 3 xem xét bốn nhóm mẫu đầu tiên: phong cách giao tiếp, khám phá, độ tin cậy và nhắn tin giao dịch. Chương 8 xem xét các mẫu API bên ngoài.

**DMẪU NHẤT QUÁN ATA ĐỂ TRIỂN KHAI QUẢN LÝ GIAO DỊCH**

Như đã đề cập trước đó, để đảm bảo sự liên kết lỏng lẻo, mỗi dịch vụ đều có cơ sở dữ liệu riêng. Thật không may, việc có một cơ sở dữ liệu cho mỗi dịch vụ lại gây ra một số vấn đề đáng kể. Tôi mô tả trong chương 4 rằng cách tiếp cận truyền thống là sử dụng các giao dịch phân tán (2PC) không phải là lựa chọn khả thi cho một ứng dụng hiện đại. Thay vào đó, một ứng dụng cần duy trì tính nhất quán của dữ liệu bằng cách sử dụng mẫu Saga. Hình 1.13 cho thấy các mẫu liên quan đến dữ liệu.

Các chương 4, 5 và 6 mô tả các mô hình này chi tiết hơn.

**PCÁC HƯỚNG DẪN TRUY VẤN DỮ LIỆU TRONG KIẾN TRÚC VI DỊCH VỤ**

Một vấn đề khác khi sử dụng cơ sở dữ liệu cho mỗi dịch vụ là một số truy vấn cần phải kết hợp dữ liệu thuộc sở hữu của nhiều dịch vụ. Dữ liệu của một dịch vụ chỉ có thể truy cập thông qua API của dịch vụ đó, do đó bạn không thể sử dụng các truy vấn phân tán đối với cơ sở dữ liệu của dịch vụ đó. Hình 1.14 cho thấy một số mẫu bạn có thể sử dụng để triển khai các truy vấn.

Chìa khóa

Truyện dài

Sự kiện miền



Người tiền nhiệm

Phương án A

Tổng quan

Người kế nhiệm

Phương án BCụ thể

Khu vực có vấn đề

Cơ sở dữ liệu theo dịch vụ

Tổng hợp

Nguồn sự kiện

**Hình 1.13 Vì mỗi dịch vụ có cơ sở dữ liệu riêng nên bạn phải sử dụng mẫu Saga để duy trìtính nhất quán của dữ liệu trên các dịch vụ.**

Chìa khóa

Giao diện lập trình ứng dụng (API)

thành phần

Cơ sở dữ liệu theo dịch vụ



Người tiền nhiệm

Phương án A

Tổng quan

Người kế nhiệm

Phương án BCụ thể

Khu vực có vấn đề

**Hình 1.14 Vì mỗi dịch vụ có cơ sở dữ liệu riêng nên bạn phải sử dụng một trong các mẫu truy vấn để truy xuất dữ liệu nằm rải rác trên nhiều dịch vụ.**

CQRS

Đôi khi bạn có thể sử dụng mẫu thành phần API, mẫu này gọi API của một hoặc nhiều dịch vụ và tổng hợp kết quả. Những lần khác, bạn phải sử dụng mẫu phân tách trách nhiệm truy vấn lệnh (CQRS), mẫu này duy trì một hoặc nhiều dịch vụ dễ dàng hơn.bản sao dữ liệu được truy vấn. Chương 7 xem xét các cách khác nhau để triển khai truy vấn.

**SMẪU TRIỂN KHAI DỊCH VỤ**

Triển khai một ứng dụng đơn khối không phải lúc nào cũng dễ dàng, nhưng nó đơn giản theo nghĩa là chỉ có một ứng dụng duy nhất để triển khai. Bạn phải chạy nhiều phiên bản của ứng dụng đằng sau bộ cân bằng tải.

So sánh, việc triển khai một ứng dụng dựa trên dịch vụ vi mô phức tạp hơn nhiều. Có thể có hàng chục hoặc hàng trăm dịch vụ được viết bằng nhiều ngôn ngữ và khuôn khổ khác nhau. Có nhiều bộ phận chuyển động hơn cần được quản lý. Hình 1.15 cho thấy các mẫu triển khai.

Cách triển khai ứng dụng theo cách truyền thống và thường là thủ công theo định dạng đóng gói ngôn ngữ cụ thể, ví dụ như tệp WAR, không mở rộng để hỗ trợ kiến ​​trúc dịch vụ vi mô. Bạn cần một cơ sở hạ tầng triển khai tự động hóa cao. Lý tưởng nhất là bạn nên sử dụng một nền tảng triển khai cung cấp cho nhà phát triển một giao diện người dùng đơn giản (dòng lệnh hoặc GUI) để triển khai và quản lý các dịch vụ của họ. Nền tảng triển khai thường sẽ dựa trên máy ảo (VM), vùng chứa hoặc công nghệ không máy chủ. Chương 12 sẽ xem xét các tùy chọn triển khai khác nhau.

Chìa khóa

**Cách tiếp cận truyền thống là triển khai các dịch vụ bằng cách sử dụng gói ngôn ngữ cụ thể của chúng, chẳng hạn như tệp WAR**



Người tiền nhiệm

Phương án A

Tổng quan

Người kế nhiệm

Phương án BCụ thể

Khu vực có vấn đề

**Một cách tiếp cận hiện đại, bao gồm công nghệ của một dịch vụ**

**Một cách tiếp cận hiện đại, chạy mã của bạn mà không cần bạn phải lo lắng về việc quản lý cơ sở hạ tầng**

Triển khai không cần máy chủ

Nhiềudịch vụ cho mỗi máy chủ

|  |  |
| --- | --- |
| Đơndịch vụ cho mỗi máy chủ | |
|  |  |
| Dịch vụ trên mỗi VM | |

**Nền tảng tự động, tự phục vụ để triển khai và quản lý dịch vụ**

Dịch vụnền tảng triển khai

Dịch vụ theo container

**Hình 1.15 Một số mô hình triển khai dịch vụ vi mô. Cách tiếp cận truyền thống là triển khai dịch vụ theo định dạng đóng gói ngôn ngữ cụ thể. Có hai cách tiếp cận hiện đại để triển khai dịch vụ. Cách đầu tiên triển khai dịch vụ dưới dạng VM hoặc container. Cách thứ hai là cách tiếp cận không có máy chủ. Bạn chỉ cần tải mã dịch vụ lên và nền tảng không có máy chủ sẽ chạy mã đó. Bạn nên sử dụng nền tảng triển khai dịch vụ, đây là nền tảng tự phục vụ, tự động để triển khai và quản lýdịch vụ.**

**ỒCÁC MẪU KHẢ NĂNG KIỂM SOÁT CUNG CẤP THÔNG TIN CHI TIẾT VỀ HÀNH VI ỨNG DỤNG**

Một phần quan trọng của việc vận hành ứng dụng là hiểu hành vi thời gian chạy của nó và khắc phục sự cố như yêu cầu không thành công và độ trễ cao. Mặc dù hiểu vàxử lý sự cố ứng dụng đơn khối không phải lúc nào cũng dễ dàng, nhưng sẽ hữu ích nếu các yêu cầu được xử lý theo cách đơn giản, trực tiếp. Mỗi yêu cầu đến được cân bằng tải cho một phiên bản ứng dụng cụ thể, thực hiện một vài cuộc gọi đến cơ sở dữ liệu và trả về phản hồi. Ví dụ, nếu bạn cần hiểu cách xử lý một yêu cầu cụ thể, bạn hãy xem tệp nhật ký của phiên bản ứng dụng đã xử lý yêu cầu đó.

Ngược lại, việc hiểu và chẩn đoán các vấn đề trong kiến ​​trúc vi dịch vụ phức tạp hơn nhiều. Một yêu cầu có thể bị chuyển qua lại giữa nhiều dịch vụ trước khi phản hồi cuối cùng được trả về cho máy khách. Do đó, không có một tệp nhật ký nào để kiểm tra. Tương tự như vậy, các vấn đề về độ trễ khó chẩn đoán hơn vì có nhiều nghi phạm.

Bạn có thể sử dụng các mẫu sau để thiết kế các dịch vụ có thể quan sát được:

* + *API kiểm tra sức khỏe*— Hiển thị điểm cuối trả về tình trạng hoạt động của dịch vụ.
  + *Tổng hợp nhật ký*—Ghi nhật ký hoạt động dịch vụ và viết nhật ký vào máy chủ ghi nhật ký tập trung, cung cấp chức năng tìm kiếm và cảnh báo.
* *Phân phối theo dõi*—Gán một ID duy nhất cho mỗi yêu cầu bên ngoài và theo dõi các yêu cầu khi chúng di chuyển giữa các dịch vụ.
* *Theo dõi ngoại lệ*—Báo cáo ngoại lệ cho dịch vụ theo dõi ngoại lệ, dịch vụ này sẽ loại bỏ các ngoại lệ trùng lặp, cảnh báo cho nhà phát triển và theo dõi quá trình giải quyết từng ngoại lệ.
* *Số liệu ứng dụng*-Duy trìcác số liệu, chẳng hạn như bộ đếm và đồng hồ đo, và hiển thị chúng cho máy chủ số liệu.
* *Nhật ký kiểm toán*—Nhật kýhành động của người dùng.

Chương 11 mô tả các mô hình này chi tiết hơn.

**PCÁC BIỆN PHÁP CHO VIỆC KIỂM TRA TỰ ĐỘNG CÁC DỊCH VỤ**

Kiến trúc microservice giúp các dịch vụ riêng lẻ dễ kiểm tra hơn vì chúng nhỏ hơn nhiều so với ứng dụng đơn khối. Tuy nhiên, đồng thời, điều quan trọng là phải kiểm tra xem các dịch vụ khác nhau có hoạt động cùng nhau hay không trong khi tránh sử dụng các bài kiểm tra đầu cuối phức tạp, chậm và dễ vỡ để kiểm tra nhiều dịch vụ cùng nhau. Sau đây là các mẫu để đơn giản hóa việc kiểm tra bằng cách kiểm tra các dịch vụ riêng lẻ:

* *Kiểm tra hợp đồng do người tiêu dùng thúc đẩy*—Xác minh rằng dịch vụ đáp ứng được mong đợi của khách hàng.
* *Kiểm tra hợp đồng phía người tiêu dùng*—Xác minh rằng máy khách của dịch vụ có thể giao tiếp với dịch vụ đó.
* *Kiểm tra thành phần dịch vụ*—Kiểm tra dịch vụ một cách riêng biệt.

Chương 9 và 10 mô tả các mô hình thử nghiệm này chi tiết hơn.

**P****CÁC LƯU Ý KHI XỬ LÝ CHỮ THẬP-CẮT BỎ MỐI QUAN TÂM**

Trong kiến ​​trúc microservice, có rất nhiều mối quan tâm mà mọi dịch vụ phải triển khai, bao gồm các mẫu quan sát và mẫu khám phá. Nó cũng phải triển khai mẫu Externalized Configuration, mẫu này cung cấp các tham số cấu hình như thông tin xác thực cơ sở dữ liệu cho một dịch vụ khi chạy. Khi phát triển một dịch vụ mới, sẽ rất tốn thời gian để triển khai lại các mối quan tâm này từ đầu. Một cách tiếp cận tốt hơn nhiều là áp dụng mẫu Microservice Chassis và xây dựng các dịch vụ trên một khuôn khổ xử lý các mối quan tâm này. Chương 11 mô tả các mẫu này chi tiết hơn.

**SMẪU CHỨNG KHOÁN**

Trong kiến ​​trúc vi dịch vụ, người dùng thườngđược xác thực bởi cổng API. Sau đó, nó phải truyền thông tin về người dùng, chẳng hạn như danh tính và vai trò, đến các dịch vụ mà nó gọi. Một giải pháp phổ biến là áp dụng mẫu mã thông báo truy cập. Cổng API truyền mã thông báo truy cập, chẳng hạn như JWT (Mã thông báo web JSON), đến các dịch vụ, có thể xác thực mã thông báo và lấy thông tin về người dùng. Chương 11 thảo luận chi tiết hơn về mẫu mã thông báo truy cập.

Không có gì ngạc nhiên khi các mẫu trong ngôn ngữ mẫu kiến ​​trúc Microservice tập trung vào việc giải quyết các vấn đề về kiến ​​trúc và thiết kế. Bạn chắc chắn cần đúng

kiến trúc để phát triển phần mềm thành công, nhưng đó không phải là mối quan tâm duy nhất. Bạn cũng phải xem xét quy trình và tổ chức.

#### Ngoài các dịch vụ vi mô: Quy trình và tổ chức

Đối với một ứng dụng lớn, phức tạp, kiến ​​trúc microservice thường là lựa chọn tốt nhất. Nhưng ngoài việc có kiến ​​trúc phù hợp, phát triển phần mềm thành công còn đòi hỏi bạn phải có tổ chức, quy trình phát triển và phân phối. Hình 1.16 cho thấy mối quan hệ giữa quy trình, tổ chức và kiến ​​trúc.

**Quá trình:**

DevOps/giao hàng liên tục/triển khai

Cho phép

Cho phép

Nhanh chóng, thường xuyên,

và cung cấp phần mềm đáng tin cậy

**Tổ chức:**

Các nhóm nhỏ, tự chủ, đa chức năng

Cho phép

**Ngành kiến ​​​​trúc:**

Kiến trúc vi dịch vụ

**Hình 1.16Việc cung cấp nhanh chóng, thường xuyên và đáng tin cậy các ứng dụng lớn, phức tạp đòi hỏi sự kết hợp của DevOps, bao gồm phân phối/triển khai liên tục, nhỏ, tự chủcác nhóm và kiến ​​trúc dịch vụ vi mô.**

Tôi đã mô tả kiến ​​trúc microservice rồi. Chúng ta hãy xem xét tổ chức và quy trình.

###### Tổ chức phát triển và cung cấp phần mềm

Thành công chắc chắn có nghĩa là kỹ thuậtnhóm sẽ phát triển. Một mặt, đó là điều tốt vì nhiều nhà phát triển hơn có thể hoàn thành nhiều việc hơn. Vấn đề với các nhóm lớn là, như Fred Brooks đã viết trong The Mythical Man-Month, chi phí truyền thông của một nhóm có quy mô N là O(N 2). Nếu nhóm quá lớn, nó sẽ trở nên kém hiệu quả, do chi phí truyền thông. Ví dụ, hãy tưởng tượng cố gắng thực hiện một cuộc họp hàng ngày với 20 người.

Giải pháp là tái cấu trúc một nhóm lớn thành một nhóm gồm nhiều nhóm. Mỗi nhóm đều nhỏ, bao gồm không quá 8–12 người. Nhóm có sứ mệnh hướng đến doanh nghiệp được xác định rõ ràng: phát triển và có thể vận hành một hoặc nhiều dịch vụ triển khai một tính năng hoặc khả năng kinh doanh. Nhóm có chức năng chéo và có thể phát triển, thử nghiệm và triển khai các dịch vụ của mình mà không cần phải thường xuyên giao tiếp hoặc phối hợp với các nhóm khác.

**Conway ngược lạiđiều động**

Để cung cấp phần mềm hiệu quả khi sử dụng kiến ​​trúc vi dịch vụ, bạn cần tính đến định luật Conway ([https://en.wikipedia.org/wiki/Conway%27s](https://en.wikipedia.org/wiki/Conway%27s_law)

[\_pháp luật](https://en.wikipedia.org/wiki/Conway%27s_law)), trong đó nêu rõ như sau:

*Các tổ chức thiết kế hệ thống… bị hạn chế phải đưa ra các thiết kế sao chép cấu trúc truyền thông của chính các tổ chức này.*

Melvin Conway

Nói cách khác, kiến ​​trúc ứng dụng của bạn phản ánh cấu trúc của tổ chức.tion đã phát triển nó. Do đó, điều quan trọng là áp dụng luật Conway theo chiều ngược lại ([www.thoughtworks.com/radar/techniques/inverse-conway-maneuver](http://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver)) và thiết kế tổ chức của bạn sao cho cấu trúc của nó phản ánh kiến ​​trúc dịch vụ vi mô của bạn. Bằng cách đó, bạn đảm bảo rằng các nhóm phát triển của mình được liên kết lỏng lẻo như các dịch vụ.

Tốc độ của nhóm các nhóm cao hơn đáng kể so với một nhóm lớn duy nhất. Như đã mô tả trước đó trong phần 1.5.1, kiến ​​trúc dịch vụ vi mô đóng vai trò quan trọng trong việc cho phép các nhóm tự chủ. Mỗi nhóm có thể phát triển, triển khai và mở rộng dịch vụ của mình mà không cần phối hợp với các nhóm khác. Hơn nữa, rất rõ ràng khi một dịch vụ không đáp ứng SLA của mình thì cần liên hệ với ai.

Hơn nữa, tổ chức phát triển có khả năng mở rộng hơn nhiều. Bạn phát triển tổ chức bằng cách thêm nhóm. Nếu một nhóm duy nhất trở nên quá lớn, bạn sẽ chia nhóm đó và dịch vụ hoặc các dịch vụ liên quan. Vì các nhóm được liên kết lỏng lẻo, bạn tránh được chi phí giao tiếp của một nhóm lớn. Do đó, bạn có thể thêm người mà không ảnh hưởng đến năng suất.

###### Quy trình phát triển và cung cấp phần mềm

Sử dụng kiến ​​trúc microservice với quy trình phát triển thác nước giống như lái xe Ferrari do ngựa kéo—bạn lãng phí hầu hết lợi ích khi sử dụng microservice. Nếu bạn muốn phát triển ứng dụng với kiến ​​trúc microservice, điều cần thiết là bạn phải áp dụng các phương pháp triển khai và phát triển nhanh như Scrum hoặc Kanban. Tốt hơn nữa, bạn nên thực hành phân phối/triển khai liên tục, đây là một phần của DevOps.

Jez khiêm tốn (<https://continuousdelivery.com/>) định nghĩa phân phối liên tục như sau:

*Phân phối liên tục là khả năng nhận được các thay đổi của tất cả các loại—bao gồmcác tính năng mới, thay đổi cấu hình, sửa lỗi và thử nghiệm—vào sản xuất hoặc đến tay người dùng, một cách an toàn và nhanh chóng theo cách bền vững.*

Một đặc điểm chính của phân phối liên tục là phần mềm luôn có thể phát hành. Nó dựa vào mức độ tự động hóa cao, bao gồm cả thử nghiệm tự động. Triển khai liên tục đưa phân phối liên tục tiến thêm một bước nữa trong thực hành triển khai tự động mã có thể phát hành vào sản xuất. Các tổ chức có hiệu suất cao

thực hành triển khai liên tục triển khai nhiều lần mỗi ngày vào sản xuất, có ít sự cố sản xuất hơn và phục hồi nhanh chóng sau bất kỳ sự cố nào xảy ra (<https://puppet.com/resources/whitepaper/state-of-devops-report>). Như đã mô tả trước đó trong phần 1.5.1, kiến ​​trúc vi dịch vụ hỗ trợ trực tiếp việc phân phối/triển khai liên tục.

**Di chuyển nhanh mà không làm hỏng đồ vật**

Mục tiêu của việc phân phối/triển khai liên tục (và nói chung là DevOps) là phân phối phần mềm nhanh chóng nhưng đáng tin cậy. Bốn số liệu hữu ích để đánh giá quá trình phát triển phần mềmCác biện pháp như sau:

* *Tần suất triển khai*—Tần suất triển khai phần mềm vào sản xuất
* *Thời gian dẫn*—Thời gian từ nhà phát triển kiểm tra thay đổi đóthay đổi đang được triển khai
* *Thời gian trung bình để**hồi phục*—Thời gian phục hồi sau sự cố sản xuất
* *Thay đổi tỷ lệ thất bại*—Tỷ lệ phần trăm các thay đổi dẫn đến vấn đề sản xuất

Trong một tổ chức truyền thống, tần suất triển khai làthấp và thời gian chờ cao. Các nhà phát triển và nhân viên vận hành căng thẳng thường thức khuya để sửa các sự cố vào phút chót trong thời gian bảo trì. Ngược lại, một tổ chức DevOps phát hành phần mềm thường xuyên, thường là nhiều lần mỗi ngày, với ít sự cố sản xuất hơn. Ví dụ, Amazon triển khai các thay đổi vào sản xuất mỗi

11,6 giây vào năm 2014 ([www.youtube.com/watch?v=dxk8b9rSKOo](http://www.youtube.com/watch?v=dxk8b9rSKOo)), và Netflix có thời gian chuẩn bị là 16 phút cho một thành phần phần mềm ([https://medium.com/netflix-](https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15) [techblog/cách-chúng-tôi-xây-dựng-mã-tại-netflix-c5d9bd727f15](https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15)).

###### Mặt con người trong việc áp dụng các dịch vụ vi mô

Việc áp dụng kiến ​​trúc microservice sẽ thay đổi kiến ​​trúc, tổ chức và quy trình phát triển của bạn. Tuy nhiên, cuối cùng, nó sẽ thay đổi môi trường làm việc của con người, những người, như đã đề cập trước đó, là những sinh vật giàu cảm xúc. Nếu bị bỏ qua, cảm xúc của họ có thể khiến việc áp dụng microservice trở nên khó khăn. Mary và các nhà lãnh đạo FTGO khác sẽ phải vật lộn để thay đổi cách FTGO phát triển phần mềm.

Cuốn sách bán chạy nhất Quản lý quá trình chuyển đổi (Da Capo Lifelong Books, 2017,<https://wmbridges.com/books>) của William và Susan Bridges giới thiệu khái niệm về quá trình chuyển đổi, đề cập đến quá trình mọi người phản ứng về mặt cảm xúc với sự thay đổi. Nó mô tả một Mô hình chuyển đổi ba giai đoạn:

**1***Kết thúc, mất mát và buông bỏ*—Giai đoạn biến động cảm xúc và kháng cự khi mọi người phải đối mặt với một sự thay đổi buộc họ phải rời khỏi vùng an toàn của mình. Họ thường than thở về sự mất mát của cách làm việc cũ. Ví dụ, khi mọi người tổ chức lại thành các nhóm chức năng chéo, họ nhớ những người đồng đội cũ của mình. Tương tự như vậy, một nhóm mô hình hóa dữ liệu sở hữu mô hình dữ liệu toàn cầu sẽ bị đe dọa bởi ý tưởng mỗi dịch vụ có mô hình dữ liệu riêng.

**2***Khu vực trung lập*—Giai đoạn trung gian giữa cách làm việc cũ và mới, khi mọi người thường bối rối. Họ thường phải vật lộn để học cách làm việc mới.

**3***Sự khởi đầu mới*—Giai đoạn cuối cùng khi mọi người nhiệt tình đón nhận cách làm việc mới và bắt đầu trải nghiệm những lợi ích.

Cuốn sách mô tả cách tốt nhất để quản lý từng giai đoạn chuyển đổi và tăng khả năng thực hiện thành công thay đổi. FTGO chắc chắn đang phải chịu đựng địa ngục đơn khối và cần phải di chuyển sang kiến ​​trúc dịch vụ vi mô. Nó cũng phải thay đổi tổ chức và quy trình phát triển của mình. Tuy nhiên, để FTGO thực hiện thành công điều này, nó phải tính đến mô hình chuyển đổi và xem xét cảm xúc của mọi người.

Ở chương tiếp theo, bạn sẽ tìm hiểu về mục tiêu của kiến ​​trúc phần mềm và cách phân tích ứng dụng thành các dịch vụ.

#### Bản tóm tắt

* + - * Mô hình kiến ​​trúc Monolithic cấu trúc ứng dụng như một triển khai duy nhấtđơn vị có khả năng.
      * Mô hình kiến ​​trúc Microservice phân tách một hệ thống thành một tập hợp các dịch vụ có thể triển khai độc lập, mỗi dịch vụ có cơ sở dữ liệu riêng.
      * Kiến trúc nguyên khối là một kiến ​​trúc tốtlựa chọn cho các ứng dụng đơn giản, nhưng kiến ​​trúc dịch vụ vi mô thường là lựa chọn tốt hơn cho các ứng dụng lớn, phức tạp.
      * Kiến trúc vi dịch vụ giúp tăng tốc độ phát triển phần mềm bằng cách cho phép các nhóm nhỏ, tự chủ làm việc song song.
      * Kiến trúc vi dịch vụ không phải là giải pháp hoàn hảo, nó có nhiều nhược điểm đáng kể, bao gồm cả tính phức tạp.
      * Ngôn ngữ mẫu kiến ​​trúc Microservice là tập hợp các mẫu giúp bạn thiết kế ứng dụng bằng kiến ​​trúc microservice. Nó giúp bạn quyết định có nên sử dụng kiến ​​trúc microservice hay không và nếu bạn chọn kiến ​​trúc microservice, ngôn ngữ mẫu sẽ giúp bạn áp dụng hiệu quả.
      * Bạn cần nhiều hơn là chỉ kiến ​​trúc microservice để đẩy nhanh việc cung cấp phần mềm. Phát triển phần mềm thành công cũng đòi hỏi DevOps và các nhóm nhỏ, tự chủ.
      * Đừng quên khía cạnh con người khi áp dụng dịch vụ vi mô. Bạn cần cân nhắc đến cảm xúc của nhân viên để chuyển đổi thành công sang kiến ​​trúc dịch vụ vi mô.

*Chiến lược phân hủy*

***Chương này bao gồm***

* Hiểu về kiến ​​trúc phần mềm và lý do tại sao nóquan trọng
* Phân tích ứng dụng thành các dịch vụ bằng cách áp dụng các mẫu phân tích Phân tích theo khả năng kinh doanh và Phân tích theo miền phụ
* Sử dụng khái niệm ngữ cảnh bị giới hạn từ miềnthiết kế hướng đích (DDD) để giải quyết dữ liệu và phân tích dễ dàng hơn

Đôi khi bạn phải cẩn thận với những gì bạn mong muốn. Sau một nỗ lực vận động hành lang mạnh mẽ, Mary cuối cùng đã thuyết phục được doanh nghiệp rằng việc di chuyển sang kiến ​​trúc vi dịch vụ là điều đúng đắn cần làm. Cảm thấy vừa phấn khích vừa lo lắng, Mary đã có một cuộc họp kéo dài cả buổi sáng với các kiến ​​trúc sư của mình để thảo luận về việc nên bắt đầu từ đâu. Trong cuộc thảo luận, rõ ràng là một số khía cạnh của ngôn ngữ mẫu kiến ​​trúc vi dịch vụ, chẳng hạn như triển khai và khám phá dịch vụ, là mới và không quen thuộc, nhưng lại đơn giản. Thách thức chính, là bản chất của kiến ​​trúc vi dịch vụ, là phân tách chức năng của ứng dụng thành các dịch vụ. Khía cạnh đầu tiên và quan trọng nhất của kiến ​​trúc là,

**33**

do đó, định nghĩa về các dịch vụ. Khi họ đứng quanh bảng trắng, nhóm FTGO tự hỏi chính xác phải làm điều đó như thế nào!

Trong chương này, bạn sẽ học cách xác định kiến ​​trúc dịch vụ vi mô cho một ứng dụng. Tôi mô tả các chiến lược để phân tích ứng dụng thành các dịch vụ. Bạn sẽ học được rằng các dịch vụ được tổ chức xung quanh các mối quan tâm kinh doanh hơn là các mối quan tâm kỹ thuật. Tôi cũng chỉ ra cách sử dụng các ý tưởng từ thiết kế theo miền (DDD) để loại bỏ các lớp god, là các lớp được sử dụng trong toàn bộ ứng dụng và gây ra các phụ thuộc rối rắm ngăn cản việc phân tích.

Tôi bắt đầu chương này bằng cách định nghĩa kiến ​​trúc microservice theo các khái niệm về kiến ​​trúc phần mềm. Sau đó, tôi mô tả quy trình định nghĩa kiến ​​trúc microservice cho một ứng dụng bắt đầu từ các yêu cầu của ứng dụng đó. Tôi thảo luận về các chiến lược để phân tích ứng dụng thành một tập hợp các dịch vụ, các trở ngại đối với ứng dụng đó và cách khắc phục chúng. Chúng ta hãy bắt đầu bằng cách xem xét khái niệm về kiến ​​trúc phần mềm.

#### Kiến trúc vi dịch vụ thực chất là gì?

Chương 1 mô tả cách ý tưởng chính của kiến ​​trúc microservice là phân tích chức năng. Thay vì phát triển một ứng dụng lớn, bạn cấu trúc ứng dụng như một tập hợp các dịch vụ. Một mặt, việc mô tả kiến ​​trúc microservice như một loại phân tích chức năng là hữu ích. Nhưng mặt khác, nó để lại một số câu hỏi chưa được trả lời, bao gồm kiến ​​trúc microservice liên quan đến các khái niệm rộng hơn về kiến ​​trúc phần mềm như thế nào? Dịch vụ là gì? Và quy mô của dịch vụ quan trọng như thế nào?

Để trả lời những câu hỏi đó, chúng ta cần lùi lại một bước và xem xét ý nghĩa của kiến ​​trúc phần mềm. Kiến trúc của một ứng dụng phần mềm là cấu trúc cấp cao của nó, bao gồm các thành phần cấu thành và sự phụ thuộc giữa các thành phần đó. Như bạn sẽ thấy trong phần này, kiến ​​trúc của một ứng dụng là đa chiều, vì vậy có nhiều cách để mô tả nó. Lý do kiến ​​trúc quan trọng là vì nó xác định các thuộc tính hoặc khả năng chất lượng phần mềm của ứng dụng. Theo truyền thống, mục tiêu của kiến ​​trúc là khả năng mở rộng, độ tin cậy và bảo mật. Nhưng ngày nay, điều quan trọng là kiến ​​trúc cũng cho phép phân phối phần mềm nhanh chóng và an toàn. Bạn sẽ biết rằng kiến ​​trúc vi dịch vụ là một kiểu kiến ​​trúc mang lại cho ứng dụng khả năng bảo trì, khả năng kiểm tra và khả năng triển khai cao.

Tôi bắt đầu phần này bằng cách mô tả khái niệm về kiến ​​trúc phần mềm và lý do tại sao nó quan trọng. Tiếp theo, tôi thảo luận về ý tưởng về phong cách kiến ​​trúc. Sau đó, tôi định nghĩa kiến ​​trúc dịch vụ vi mô là một phong cách kiến ​​trúc cụ thể. Chúng ta hãy bắt đầu bằng cách xem xét khái niệm kiến ​​trúc phần mềm.

###### Kiến trúc phần mềm là gì và tại sao nó lại quan trọng?

Kiến trúc rõ ràng là quan trọng. Có ít nhất hai hội nghị dành riêng cho chủ đề này: Hội nghị Kiến trúc Phần mềm O'Reilly ([https://conferences.oreilly.com/](https://conferences.oreilly.com/software-architecture) [kiến trúc phần mềm](https://conferences.oreilly.com/software-architecture)) và hội nghị SATURN ([https://resource.sei.cmu.edu/](https://resources.sei.cmu.edu/news-events/events/saturn/) [tin tức-sự kiện/sự kiện/sao thổ/](https://resources.sei.cmu.edu/news-events/events/saturn/)). Nhiều nhà phát triển có mục tiêu trở thành kiến ​​trúc sư. Nhưng kiến ​​trúc là gì và tại sao nó lại quan trọng?

Để trả lời câu hỏi đó, trước tiên tôi định nghĩa thuật ngữ kiến ​​trúc phần mềm có nghĩa là gì. Sau đó, tôi thảo luận về cách kiến ​​trúc của ứng dụng là đa chiều và được mô tả tốt nhất bằng cách sử dụng một tập hợp các chế độ xem hoặc bản thiết kế. Sau đó, tôi mô tả rằng kiến ​​trúc phần mềm quan trọng vì tác động của nó đến các thuộc tính chất lượng phần mềm của ứng dụng.

**MỘTĐỊNH NGHĨA KIẾN TRÚC PHẦN MỀM**

Có rất nhiều định nghĩa về kiến ​​trúc phần mềm. Ví dụ, hãy xem[https://](https://en.wikiquote.org/wiki/Software_architecture) [vi.wikiquote.org/wiki/Software\_architecture](https://en.wikiquote.org/wiki/Software_architecture)để đọc một số trong số chúng. Định nghĩa yêu thích của tôi đến từ Len Bass và các đồng nghiệp tại Viện Kỹ thuật Phần mềm ([www.sei.cmu.edu](http://www.sei.cmu.edu/)), người đóng vai trò chủ chốt trong việc thiết lập kiến ​​trúc phần mềm như một chuyên ngành. Họ định nghĩa kiến ​​trúc phần mềm như sau:

*Kiến trúc phần mềm của một hệ thống máy tính là tập hợp các cấu trúc cần thiết để lý giải về hệ thống, bao gồm các thành phần phần mềm, mối quan hệ giữa chúng và các thuộc tính của cả hai.*

Tài liệu về kiến ​​trúc phần mềm của Bass và cộng sự

Rõ ràng đó là một định nghĩa khá trừu tượng. Nhưng bản chất của nó là kiến ​​trúc của ứng dụng là sự phân chia thành các phần (các thành phần) và các mối quan hệ (các mối quan hệ) giữa các phần đó. Phân chia quan trọng vì một số lý do:

* Nó tạo điều kiện cho việc phân chia lao động và kiến ​​thức. Nó cho phép nhiều người (hoặc nhiều nhóm) có thể có kiến ​​thức chuyên môn làm việc hiệu quả cùng nhau trên một ứng dụng.
* Nó xác định cách các thành phần phần mềm tương tác với nhau.

Chính sự phân chia thành các phần và mối quan hệ giữa các phần đó quyết định tính khả thi của ứng dụng.

**TANH TA4+1XEM MÔ HÌNH KIẾN TRÚC PHẦN MỀM**

Cụ thể hơn, kiến ​​trúc của một ứng dụng có thể được xem xét từ nhiều góc độ, giống như cách kiến ​​trúc của một tòa nhà có thể được xem xét từ góc độ kết cấu, hệ thống ống nước, điện và các góc độ khác. Phillip Krutchen đã viết một bài báo kinh điển mô tả mô hình quan điểm 4+1 của kiến ​​trúc phần mềm, “Architectural Blueprints—The '4+1' View Model of Software Architecture” ([www.cs.ubc.ca/~gregor/teaching/](http://www.cs.ubc.ca/~gregor/teaching/papers/4%2B1view-architecture.pdf) [giấy tờ/4+1view-architecture.pdf](http://www.cs.ubc.ca/~gregor/teaching/papers/4%2B1view-architecture.pdf)). Mô hình 4+1, được thể hiện trong Hình 2.1, định nghĩa bốn góc nhìn khác nhau về kiến ​​trúc phần mềm. Mỗi góc nhìn mô tả một khía cạnh cụ thể của kiến ​​trúc và bao gồm một tập hợp các thành phần phần mềm và mối quan hệ giữa chúng.

Mục đích của mỗi chế độ xem như sau:

* *Quan điểm logic*—Cáccác thành phần phần mềm do các nhà phát triển tạo ra. Trong các ngôn ngữ hướng đối tượng, các thành phần này là các lớp và các gói. Các mối quan hệ giữa chúng là các mối quan hệ giữa các lớp và các gói, bao gồm cả sự kế thừa, liên kết và phụ thuộc.
* *Quan điểm thực hiện*—Đầu ra của hệ thống xây dựng. Chế độ xem này bao gồm các mô-đun, biểu diễn mã đóng gói và các thành phần có thể thực thi

**Những gì các nhà phát triển tạo raCác yếu tố:Các lớp học và góiMối quan hệ:Các mối quan hệ**

**giữa họ**

**Những gì được tạo ra bởi hệ thống xây dựngCác yếu tố:Các mô-đun, (tệp JAR) và**

**thành phần (tệp WARhoặc các tệp thực thi)**

**Mối quan hệ:Sự phụ thuộc của họ**

Quan điểm logic

Các tình huống

Quan điểm thực hiện

**Làm cho các góc nhìn trở nên sinh động.**

Quá trình xem

Triển khai xem

**Các thành phần đang chạyCác yếu tố:Các quy trìnhMối quan hệ:Liên quá trình**

**giao tiếp**

**Các tiến trình chạy trên “máy móc”Các yếu tố:Máy móc và quy trìnhMối quan hệ:Mạng lưới**

**Hình 2.1 Mô hình dạng xem 4+1 mô tả kiến ​​trúc của ứng dụng bằng cách sử dụng bốn dạng xem, cùng với các kịch bản cho thấy cách các thành phần trong mỗi dạng xem cộng tác để xử lýyêu cầu.**

hoặc các đơn vị có thể triển khai bao gồm một hoặc nhiều mô-đun. Trong Java, mô-đun là tệp JAR và thành phần thường là tệp WAR hoặc tệp JAR có thể thực thi. Mối quan hệ giữa chúng bao gồm mối quan hệ phụ thuộc giữa các mô-đun và mối quan hệ thành phần giữa các thành phần và mô-đun.

* *Quá trình xem*—Các thành phần khi chạy. Mỗi phần tử là một tiến trình và mối quan hệ giữa các tiến trình thể hiện sự giao tiếp giữa các tiến trình.
* *Triển khai*—Các quá trình diễn ra như thế nàođược ánh xạ tới máy móc. Các thành phần trong chế độ xem này bao gồm máy móc (vật lý hoặc ảo) và các quy trình. Mối quan hệ giữa các máy móc thể hiện mạng lưới. Chế độ xem này cũng mô tả mối quan hệ giữa các quy trình và máy móc.

Ngoài bốn chế độ xem này, còn có các kịch bản—+1 trong mô hình 4+1—làm hoạt hình các chế độ xem. Mỗi kịch bản mô tả cách các thành phần kiến ​​trúc khác nhau trong một chế độ xem cụ thể cộng tác với nhau để xử lý một yêu cầu. Ví dụ, một kịch bản trong chế độ xem logic cho thấy cách các lớp cộng tác với nhau. Tương tự như vậy, một kịch bản trong chế độ xem quy trình cho thấy cách các quy trình cộng tác với nhau.

Mô hình xem 4+1 là một cách tuyệt vời để mô tả kiến ​​trúc của ứng dụng. Mỗi chế độ xem mô tả một khía cạnh quan trọng của kiến ​​trúc và các kịch bản

minh họa cách các thành phần của chế độ xem hợp tác với nhau. Bây giờ chúng ta hãy xem tại sao kiến ​​trúc lại quan trọng.

**TKIẾN TRÚC HY QUAN TRỌNG**

Một ứng dụng có hai loại yêu cầu. Loại đầu tiên bao gồm các yêu cầu chức năng, xác định những gì ứng dụng phải làm. Chúng thường ở dạng các trường hợp sử dụng hoặc câu chuyện của người dùng. Kiến trúc không liên quan nhiều đến các yêu cầu chức năng. Bạn có thể triển khai các yêu cầu chức năng với hầu hết mọi kiến ​​trúc, ngay cả một quả cầu bùn lớn.

Kiến trúc quan trọng vì nó cho phép ứng dụng đáp ứng loại yêu cầu thứ hai: yêu cầu về chất lượng dịch vụ. Những yêu cầu này cũng được gọi là các thuộc tính chất lượng và được gọi là -ilities. Các yêu cầu về chất lượng dịch vụ xác định các phẩm chất thời gian chạy như khả năng mở rộng và độ tin cậy. Chúng cũng xác định các phẩm chất thời gian phát triển bao gồm khả năng bảo trì, khả năng kiểm tra và khả năng triển khai. Kiến trúc bạn chọn cho ứng dụng của mình sẽ xác định mức độ đáp ứng các yêu cầu về chất lượng này.

###### Tổng quan về phong cách kiến ​​trúc

Trong thế giới vật lý, kiến ​​trúc của một tòa nhà thườngtheo một phong cách cụ thể, chẳng hạn như phong cách Victoria, American Craftsman hoặc Art Deco. Mỗi phong cách là một gói các quyết định thiết kế hạn chế các đặc điểm và vật liệu xây dựng của tòa nhà. Khái niệm về phong cách kiến ​​trúc cũng áp dụng cho phần mềm. David Garlan và Mary Shaw (Giới thiệu về Kiến trúc phần mềm, tháng 1 năm 1994,[https://www.cs.cmu.edu/afs/cs/project/](https://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf) [có thể/ftp/intro\_softarch/intro\_softarch.pdf](https://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf)), những người tiên phong trong lĩnh vực kiến ​​trúc phần mềm, định nghĩa phong cách kiến ​​trúc như sau:

*Sau đó, một phong cách kiến ​​trúc xác định một họ các hệ thống như vậy theo một mô hình tổ chức cấu trúc. Cụ thể hơn, một phong cách kiến ​​trúc xác định vốn từ vựng của các thành phần và kết nối có thể được sử dụng trong các trường hợp của phong cách đó, cùng với một tập hợp các ràng buộc về cách chúng có thể được kết hợp.*

Một phong cách kiến ​​trúc cụ thể cung cấp một bảng giới hạn các thành phần (thành phần) và mối quan hệ (bộ kết nối) mà từ đó bạn có thể xác định chế độ xem kiến ​​trúc ứng dụng của mình. Một ứng dụng thường sử dụng sự kết hợp của các phong cách kiến ​​trúc. Ví dụ, sau trong phần này, tôi sẽ mô tả cách kiến ​​trúc nguyên khối là một phong cách kiến ​​trúc cấu trúc chế độ xem triển khai thành một thành phần duy nhất (có thể thực thi/triển khai). Kiến trúc dịch vụ vi mô cấu trúc ứng dụng thành một tập hợp các dịch vụ được ghép nối lỏng lẻo.

**TKIẾN TRÚC PHONG CÁCH TẦNG**

Ví dụ kinh điển về phong cách kiến ​​trúc là kiến ​​trúc phân lớp. Kiến trúc phân lớp tổ chức các thành phần phần mềm thành các lớp. Mỗi lớp có một tập hợp trách nhiệm được xác định rõ ràng. Kiến trúc phân lớp cũng hạn chế sự phụ thuộc giữa các lớp. Một lớp chỉ có thể phụ thuộc vào lớp ngay bên dưới nó (nếu phân lớp nghiêm ngặt) hoặc bất kỳ lớp nào bên dưới nó.

Bạn có thể áp dụng kiến ​​trúc phân lớp cho bất kỳ chế độ xem nào trong bốn chế độ xem đã thảo luận trước đó. Kiến trúc ba tầng phổ biến là kiến ​​trúc phân lớp được áp dụng cho chế độ xem logic. Nó tổ chức các lớp của ứng dụng thành các tầng hoặc lớp sau:

* *Lớp trình bày*—Chứa mã thực hiện giao diện người dùng hoặc API bên ngoài
* *Lớp logic kinh doanh*—Chứa logic kinh doanh
* *Lớp bền bỉ*—Thực hiện logic tương tác với cơ sở dữ liệu

Kiến trúc nhiều lớp là một ví dụ tuyệt vời về phong cách kiến ​​trúc, nhưng nó cũng có một số nhược điểm đáng kể:

* *Lớp trình bày đơn*—Điều này không có nghĩa là một ứng dụng có thể được nhiều hệ thống khác nhau sử dụng.
* *Lớp duy trì đơn*—Điều này không phản ánh thực tế là một ứng dụng có khả năng tương tác với nhiều hơn một cơ sở dữ liệu duy nhất.
* *Xác định lớp logic kinh doanh phụ thuộc vàotrên lớp bền bỉ*—Về mặt lý thuyết, sự phụ thuộc này ngăn cản bạn kiểm tra logic kinh doanh mà không có cơ sở dữ liệu.

Ngoài ra, kiến ​​trúc phân lớp trình bày sai lệch các phụ thuộc trong một ứng dụng được thiết kế tốt. Logic kinh doanh thường định nghĩa một giao diện hoặc một kho lưu trữ các giao diện xác định các phương pháp truy cập dữ liệu. Tầng lưu trữ bền bỉ xác định các lớp DAO triển khai các giao diện kho lưu trữ. Nói cách khác, các phụ thuộc ngược lại với những gì được mô tả bởi kiến ​​trúc phân lớp.

Hãy cùng xem xét một kiến ​​trúc thay thế có thể khắc phục những nhược điểm này: kiến ​​trúc lục giác.

**MỘTVỀ KIẾN TRÚC LỤC GIÁC**

*Kiến trúc lục giác*là một giải pháp thay thế cho phong cách kiến ​​trúc phân lớp. Như hình 2.2 cho thấy, phong cách kiến ​​trúc lục giác sắp xếp chế độ xem logic theo cách đặt logic kinh doanh vào trung tâm. Thay vì lớp trình bày, ứng dụng có một hoặc nhiều bộ điều hợp đến xử lý các yêu cầu từ bên ngoài bằng cách gọi logic kinh doanh. Tương tự như vậy, thay vì một tầng lưu trữ dữ liệu, ứng dụng có một hoặc nhiều bộ điều hợp đi được logic kinh doanh gọi và gọi các ứng dụng bên ngoài. Một đặc điểm và lợi ích chính của kiến ​​trúc này là logic kinh doanh không phụ thuộc vào các bộ điều hợp. Thay vào đó, chúng phụ thuộc vào nó.

Logic kinh doanh có một hoặc nhiều cổng. Một cổng xác định một tập hợp các hoạt động và là cách logic kinh doanh tương tác với những gì bên ngoài nó. Ví dụ, trong Java, một cổng thường là một giao diện Java. Có hai loại cổng: cổng vào và cổng ra. Một cổng vào là một API được logic kinh doanh phơi bày, cho phép nó được gọi bởi các ứng dụng bên ngoài. Một ví dụ về một cổng vào là một giao diện dịch vụ, xác định các phương thức công khai của một dịch vụ. Một cổng ra là cách logic kinh doanh gọi các hệ thống bên ngoài. Một ví dụ về một cổng ra là một giao diện kho lưu trữ, xác định một tập hợp các hoạt động truy cập dữ liệu.

**Bộ chuyển đổi đầu vào Bộ chuyển đổi đầu vào**

Một số lớp điều khiển



Trình duyệt

Người tiêu dùng tin nhắn

Dịch vụ ăn uống

Logic kinh doanh

Người môi giới tin nhắn

Nhắn tingiao diện

**Cảng nhập cảnh**

Tin nhắnnhà sản xuất

Kho lưu trữgiao diện

**Bộ chuyển đổi đầu ra**

ĐẠO

Cơ sở dữ liệu

**Bộ chuyển đổi đầu raCổng ra**

**Hình 2.2 Một ví dụ về kiến ​​trúc lục giác, bao gồm logic nghiệp vụ và một hoặc nhiều bộ điều hợp giao tiếp với các hệ thống bên ngoài. Logic nghiệp vụ có một hoặc nhiều cổng. Bộ điều hợp đầu vào, xử lý các yêu cầu từ các hệ thống bên ngoài, gọi một cổng đầu vào. Mộtbộ điều hợp đầu ra thực hiện một cổng đầu ra và gọi một hệ thống bên ngoài.**

Bao quanh logic nghiệp vụ là các bộ điều hợp. Cũng giống như các cổng, có hai loại bộ điều hợp: inbound và outbound. Bộ điều hợp inbound xử lý các yêu cầu từ thế giới bên ngoài bằng cách gọi một cổng inbound. Một ví dụ về bộ điều hợp inbound là Spring MVC Controller triển khai một tập hợp các điểm cuối REST hoặc một tập hợp các trang web. Một ví dụ khác là một máy khách môi giới tin nhắn đăng ký tin nhắn. Nhiều bộ điều hợp inbound có thể gọi cùng một cổng inbound.

Bộ điều hợp gửi đi triển khai một cổng gửi đi và xử lý các yêu cầu từ logic nghiệp vụ bằng cách gọi một ứng dụng hoặc dịch vụ bên ngoài. Một ví dụ về bộ điều hợp gửi đi là lớp đối tượng truy cập dữ liệu (DAO) triển khai các hoạt động để truy cập cơ sở dữ liệu. Một ví dụ khác là lớp proxy gọi một dịch vụ từ xa. Bộ điều hợp gửi đi cũng có thể xuất bản các sự kiện.

Một lợi ích quan trọng của phong cách kiến ​​trúc lục giác là nó tách rời logic nghiệp vụ khỏi logic trình bày và truy cập dữ liệu trong bộ điều hợp. Logic nghiệp vụ không phụ thuộc vào logic trình bày hoặc logic truy cập dữ liệu.

Do sự tách biệt này, việc kiểm tra logic kinh doanh một cách riêng biệt dễ dàng hơn nhiều. Một lợi ích khác là nó phản ánh chính xác hơn kiến ​​trúc của một ứng dụng hiện đại. Logic kinh doanh có thể được gọi thông qua nhiều bộ điều hợp, mỗi bộ điều hợp triển khai một API hoặc UI cụ thể. Logic kinh doanh cũng có thể gọi nhiều bộ điều hợp, mỗi bộ điều hợp gọi một hệ thống bên ngoài khác nhau. Kiến trúc lục giác là một cách tuyệt vời để mô tả kiến ​​trúc của từng dịch vụ trong kiến ​​trúc dịch vụ vi mô.

Kiến trúc phân lớp và kiến ​​trúc lục giác đều là ví dụ về phong cách kiến ​​trúc. Mỗi phong cách xác định các khối xây dựng của một kiến ​​trúc và áp đặt các ràng buộc lên mối quan hệ giữa chúng. Kiến trúc lục giác và kiến ​​trúc phân lớp, dưới dạng kiến ​​trúc ba tầng, tổ chức chế độ xem logic. Bây giờ chúng ta hãy định nghĩa kiến ​​trúc dịch vụ vi mô là một phong cách kiến ​​trúc tổ chức chế độ xem triển khai.

###### Kiến trúc microservice là một phong cách kiến ​​trúc

Tôi đã thảo luận về mô hình xem 4+1 và các phong cách kiến ​​trúc, vì vậy bây giờ tôi có thể định nghĩa kiến ​​trúc monolithic và microservice. Cả hai đều là phong cách kiến ​​trúc. Kiến trúc monolithic là một phong cách kiến ​​trúc cấu trúc chế độ xem triển khai thành một thành phần duy nhất: một tệp thực thi hoặc tệp WAR duy nhất. Định nghĩa này không nói gì về các chế độ xem khác. Ví dụ, một ứng dụng monolithic có thể có chế độ xem logic được tổ chức theo các dòng của kiến ​​trúc lục giác.

**Mẫu: Kiến trúc nguyên khối**

Cấu trúc ứng dụng như một thành phần có thể thực thi/triển khai duy nhất. Xem[http://](http://microservices.io/patterns/monolithic.html) [microservices.io/patterns/monolithic.html.](http://microservices.io/patterns/monolithic.html)

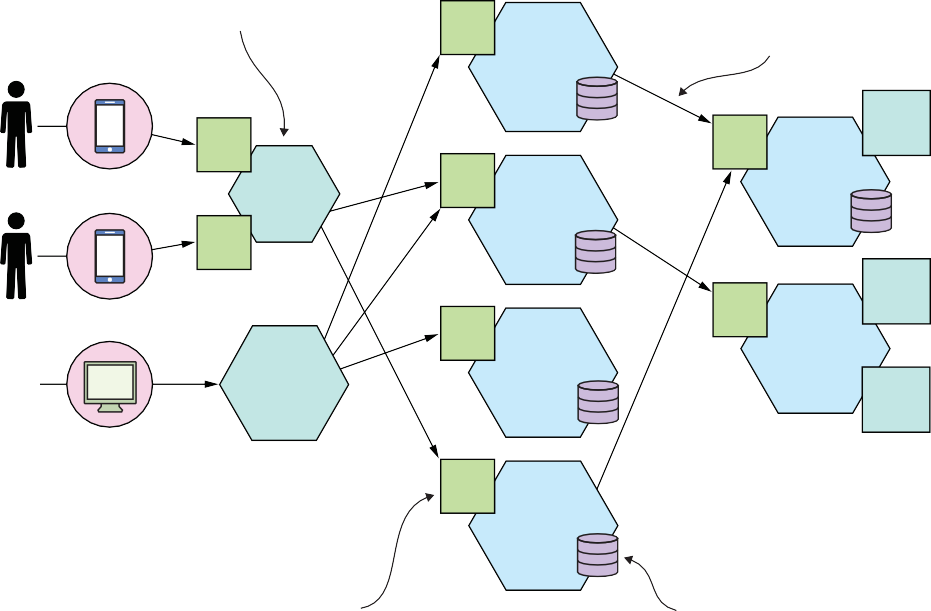
Kiến trúc vi dịch vụ cũng là một phong cách kiến ​​trúc. Nó cấu trúc chế độ xem triển khai thành một tập hợp nhiều thành phần: tệp thực thi hoặc tệp WAR. Các thành phần là các dịch vụ và các trình kết nối là các giao thức truyền thông cho phép các dịch vụ đó cộng tác. Mỗi dịch vụ có kiến ​​trúc chế độ xem logic riêng, thường là kiến ​​trúc lục giác. Hình 2.3 cho thấy một kiến ​​trúc vi dịch vụ có thể có cho ứng dụng FTGO. Các dịch vụ trong kiến ​​trúc này tương ứng với các khả năng kinh doanh, chẳng hạn như Quản lý đơn hàng và Quản lý nhà hàng.

**Mẫu: Kiến trúc vi dịch vụ**

Cấu trúc ứng dụng như một tập hợp các thành phần được liên kết lỏng lẻo, có thể triển khai độc lậpdịch vụ. Xem[http://microservices.io/patterns/microservices.html.](http://microservices.io/patterns/microservices.html)

Sau trong chương này, tôi sẽ mô tả ý nghĩa của khả năng kinh doanh. Các kết nối giữa các dịch vụ được triển khai bằng các cơ chế giao tiếp giữa các tiến trình như REST API và nhắn tin không đồng bộ. Chương 3 thảo luận chi tiết hơn về giao tiếp giữa các tiến trình.

**API Gateway định tuyến các yêu cầu từ ứng dụng di động đến các dịch vụ.**



NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Dịch vụ đặt hàng

**Các dịch vụ tương ứng với năng lực kinh doanh/ tên miền phụ DDD**

Người chuyển phát nhanh

Người tiêu dùng

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Giao diện lập trình ứng dụng (API)

Cổng vào

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Nhà hàngDịch vụ

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Kế toánDịch vụ

Thông báo

Bộ chuyển đổi sọc

Bộ chuyển đổi Twilio

Nhà hàng

Nhà hàngGiao diện người dùng web

Phòng bếp

Dịch vụ

Dịch vụ

AmazonSES

Bộ chuyển đổi

NGHỈ NGƠIGiao diện lập trình ứng dụng (API)

Vận chuyển

Dịch vụ

**Dịch vụ cóAPI.A dữ liệu của dịch vụ là riêng tư.**

**Hình 2.3 Kiến trúc dịch vụ vi mô có thể có cho ứng dụng FTGO. Nó bao gồm nhiềudịch vụ.**

Một hạn chế chính do kiến ​​trúc microservice áp đặt là các dịch vụ được kết hợp lỏng lẻo. Do đó, có những hạn chế về cách các dịch vụ cộng tác. Để giải thích những hạn chế đó, tôi sẽ cố gắng định nghĩa thuật ngữ dịch vụ, mô tả ý nghĩa của việc kết hợp lỏng lẻo và cho bạn biết lý do tại sao điều này lại quan trọng.

**TMŨ LÀ MỘT DỊCH VỤ?**

Dịch vụ là một thành phần phần mềm độc lập, có thể triển khai độc lập, thực hiện một số chức năng hữu ích. Hình 2.4 cho thấy chế độ xem bên ngoài của một dịch vụ, trong đóVí dụ này là Order Service. Một dịch vụ có một API cung cấp cho khách hàng của mình quyền truy cập vào chức năng của nó. Có hai loại hoạt động: lệnh và truy vấn. API bao gồm các lệnh, truy vấn và sự kiện. Một lệnh, chẳng hạn như createOrder(), thực hiện các hành động và cập nhật dữ liệu. Một truy vấn, chẳng hạn như findOrderById(), truy xuất dữ liệu. Một dịch vụ cũng công bố các sự kiện, chẳng hạn như OrderCreated, được khách hàng của nó sử dụng.

API của dịch vụ đóng gói triển khai nội bộ của nó. Không giống như trong khối đơn, nhà phát triển không thể viết mã bỏ qua API của nó. Do đó, kiến ​​trúc dịch vụ vi mô thực thi tính mô-đun của ứng dụng.

Mỗi dịch vụ trong kiến ​​trúc dịch vụ vi mô có kiến ​​trúc riêng và có khả năng là ngăn xếp công nghệ. Nhưng một dịch vụ điển hình có kiến ​​trúc lục giác. API của nó được triển khai bởi các bộ điều hợp tương tác với logic kinh doanh của dịch vụ. Các hoạt động

**Xác định các hoạt động**



API dịch vụ

Triệu hồi

Dịch vụ đặt hàng

Đăng ký sự kiện

Đơn hàng đã tạo Đơn hàng đã hủy

Nhà xuất bản sự kiện đặt hàng

Dịch vụ đặt hàng khách hàng

Lệnh:

tạo đơn hàng()

...

Thắc mắc:

tìmOrderbyId()

...

**Xuất bản các sự kiện khi dữ liệu thay đổi**

**Hình 2.4 Một dịch vụ có API đóng gói việc triển khai. API định nghĩa các hoạt động được khách hàng gọi. Có hai loại hoạt động: lệnh cập nhật dữ liệu và truy vấn truy xuất dữ liệu. Khi dữ liệu của nó thay đổi, một dịch vụ sẽ công bố các sự kiện mà khách hàng có thể đăng ký.**

bộ chuyển đổi gọi doanh nghiệplogic và bộ điều hợp sự kiện sẽ công bố các sự kiện do logic nghiệp vụ phát ra.

Sau đó trong chương 12, khi tôi thảo luận về các công nghệ triển khai, bạn sẽ thấy rằng chế độ xem triển khai của một dịch vụ có thể có nhiều dạng. Thành phần có thể là một quy trình độc lập, một ứng dụng web hoặc gói OSGI chạy trong một vùng chứa hoặc một chức năng đám mây không có máy chủ. Tuy nhiên, một yêu cầu thiết yếu là dịch vụ phải có API và có thể triển khai độc lập.

**TMŨ LỎNGKHỚP NỐI?**

Một đặc điểm quan trọng của kiến ​​trúc vi dịch vụ là các dịch vụ được kết hợp lỏng lẻo (<https://en.wikipedia.org/wiki/Loose_coupling>). Tất cả tương tác với mộtdịch vụ diễn ra thông qua API của nó, đóng gói các chi tiết triển khai của nó. Điều này cho phép việc triển khai dịch vụ thay đổi mà không ảnh hưởng đến các máy khách của nó. Các dịch vụ được ghép nối lỏng lẻo là chìa khóa để cải thiện các thuộc tính thời gian phát triển của ứng dụng, bao gồm khả năng bảo trì và khả năng kiểm tra của nó. Chúng dễ hiểu, dễ thay đổi và dễ kiểm tra hơn nhiều.

Yêu cầu đối với các dịch vụ phải được kết nối lỏng lẻo và chỉ cộng tác thông qua API cấm các dịch vụ giao tiếp qua cơ sở dữ liệu. Bạn phải xử lý dữ liệu liên tục của dịch vụ giống như các trường của một lớp và giữ chúng ở chế độ riêng tư. Việc giữ dữ liệu ở chế độ riêng tư cho phép nhà phát triển thay đổi lược đồ cơ sở dữ liệu của dịch vụ mà không cần phải

dành thời gian phối hợp với các nhà phát triển đang làm việc trên các dịch vụ khác. Không chia sẻ bảng cơ sở dữ liệu cũng cải thiện khả năng cô lập thời gian chạy. Ví dụ, nó đảm bảo rằng một dịch vụ không thể giữ khóa cơ sở dữ liệu chặn dịch vụ khác. Tuy nhiên, sau này, bạn sẽ biết rằng một nhược điểm của việc không chia sẻ cơ sở dữ liệu là việc duy trì tính nhất quán của dữ liệu và truy vấn giữa các dịch vụ phức tạp hơn.

**TVAI TRÒ CỦA THƯ VIỆN CHUNG**

Các nhà phát triển thường đóng gói chức năng trong một thư viện (module) để có thể được nhiều ứng dụng sử dụng lại mà không cần sao chép mã. Rốt cuộc, chúng ta sẽ ra sao nếu không có kho lưu trữ Maven hoặc npm? Bạn có thể bị cám dỗ sử dụng cả thư viện dùng chung trong kiến ​​trúc vi dịch vụ. Trên bề mặt, có vẻ như đây là một cách tốt để giảm trùng lặp mã trong các dịch vụ của bạn. Nhưng bạn cần đảm bảo rằng bạn không vô tình giới thiệu sự kết hợp giữa các dịch vụ của mình.

Ví dụ, hãy tưởng tượng rằng nhiều dịch vụ cần cập nhật đối tượng kinh doanh Order. Một cách tiếp cận là đóng gói chức năng đó thành một thư viện được nhiều dịch vụ sử dụng. Một mặt, sử dụng thư viện sẽ loại bỏ việc trùng lặp mã. Mặt khác, hãy xem xét điều gì xảy ra khi các yêu cầu thay đổi theo cách ảnh hưởng đến đối tượng kinh doanh Order. Bạn sẽ cần phải đồng thời xây dựng lại và triển khai lại các dịch vụ đó. Một cách tiếp cận tốt hơn nhiều là triển khai chức năng có khả năng thay đổi, chẳng hạn như Quản lý đơn hàng, thành một dịch vụ.

Bạn nên cố gắng sử dụng các thư viện cho chức năng không có khả năng thay đổi. Ví dụ, trong một ứng dụng thông thường, không có ý nghĩa gì khi mọi dịch vụ đều triển khai một lớp Money chung. Thay vào đó, bạn nên tạo một thư viện được các dịch vụ sử dụng.

**TKÍCH THƯỚC CỦA MỘT DỊCH VỤ ĐA SỐ LÀ KHÔNG QUAN TRỌNG**

Một vấn đề với thuật ngữ microservice là điều đầu tiên bạn nghe thấy là micro. Điều này cho thấy một dịch vụ phải rất nhỏ. Điều này cũng đúng với các thuật ngữ dựa trên kích thước khác như miniservice hoặc nanoservice. Trên thực tế, kích thước không phải là một số liệu hữu ích.

Một mục tiêu tốt hơn nhiều là xác định một dịch vụ được thiết kế tốt là một dịch vụ có khả năng được phát triển bởi một nhóm nhỏ với thời gian chuẩn bị tối thiểu và với sự hợp tác tối thiểu với các nhóm khác. Về lý thuyết, một nhóm có thể chỉ chịu trách nhiệm cho một dịch vụ duy nhất, do đó dịch vụ đó không phải là dịch vụ vi mô. Ngược lại, nếu một dịch vụ yêu cầu một nhóm lớn hoặc mất nhiều thời gian để thử nghiệm, thì có lẽ nên tách nhóm và dịch vụ. Hoặc nếu bạn liên tục cần thay đổi một dịch vụ do các dịch vụ khác thay đổi hoặc nếu nó kích hoạt các thay đổi trong các dịch vụ khác, thì đó là dấu hiệu cho thấy nó không được kết hợp lỏng lẻo. Bạn thậm chí có thể đã xây dựng một khối phân tán.

Kiến trúc vi dịch vụ cấu trúc một ứng dụng như một tập hợp các dịch vụ nhỏ, được kết nối lỏng lẻo. Do đó, nó cải thiện các thuộc tính thời gian phát triển—khả năng bảo trì, khả năng kiểm tra, khả năng triển khai, v.v.—và cho phép một tổ chức phát triển phần mềm tốt hơn nhanh hơn. Nó cũng cải thiện khả năng mở rộng của ứng dụng, mặc dù đó không phải là mục tiêu chính. Để phát triển kiến ​​trúc vi dịch vụ cho ứng dụng của bạn, bạn cần xác định các dịch vụ và xác định cách chúng cộng tác. Hãy cùng xem cách thực hiện điều đó.

#### Xác định kiến ​​trúc dịch vụ vi mô của ứng dụng

Chúng ta nên định nghĩa kiến ​​trúc vi dịch vụ như thế nào? Cũng như với bất kỳ nỗ lực phát triển phần mềm nào, điểm khởi đầu là các yêu cầu đã viết, hy vọng là các chuyên gia trong lĩnh vực và có thể là một ứng dụng hiện có. Giống như nhiều hoạt động phát triển phần mềm khác, việc định nghĩa kiến ​​trúc mang tính nghệ thuật hơn là khoa học. Phần này mô tả một quy trình đơn giản gồm ba bước, được thể hiện trong hình 2.5, để định nghĩa kiến ​​trúc của ứng dụng. Tuy nhiên, điều quan trọng cần nhớ là đây không phải là quy trình bạn có thể tuân theo một cách máy móc. Quy trình này có thể mang tính lặp đi lặp lại và đòi hỏi nhiều sự sáng tạo.

**Điểm khởi đầu là các yêu cầu, chẳng hạn như câu chuyện của người dùng.**

**Hoạt động của hệ thống thể hiện một yêu cầu bên ngoài.**

Bước 1: Xác định hệ thốnghoạt động

tạo đơn hàng()

FTGO

chấp nhận đơn hàng()

Yêu cầu chức năng

Là một nhà hàng

Tôi muốn chấp nhận đơn hàng để tôi có thể ...

Là một người tiêu dùng

Tôi muốn đặt hàng để có thể ...

Bước 2: Xác địnhdịch vụBước 3: Xác định API dịch vụ và cộng tác

FTGO

Dịch vụ đặt hàng

Nhà hàngDịch vụ

Dịch vụ nhà bếp

...

|  |  |  |
| --- | --- | --- |
| tạo đơn hàng() | FTGO  Đặt hàng xác minh đơn hàng()  Dịch vụ  Nhà hàng  Vé te() Dịch vụ | |
| Lặp lại  sáng tạo |
| chấp nhận đơn hàng() |  | Dịch vụ nhà bếp |

tạo đơn hàng()

chấp nhận đơn hàng()

**Hình 2.5 Quy trình ba bước để xác định kiến ​​trúc dịch vụ vi mô của ứng dụng**

Một ứng dụng tồn tại để xử lý các yêu cầu, vì vậy bước đầu tiên trong việc xác định kiến ​​trúc của nó là chắt lọc các yêu cầu của ứng dụng thành các yêu cầu chính. Nhưng thay vì mô tả các yêu cầu theo các công nghệ IPC cụ thể như REST hoặc nhắn tin, tôi sử dụng

khái niệm trừu tượng hơn về hoạt động của hệ thống. Hoạt động của hệ thống là sự trừu tượng hóa của một yêu cầu mà ứng dụng phải xử lý. Đó có thể là lệnh, cập nhật dữ liệu hoặc truy vấn, truy xuất dữ liệu. Hành vi của mỗi lệnh được xác định theo mô hình miền trừu tượng, cũng bắt nguồn từ các yêu cầu. Các hoạt động của hệ thống trở thành các kịch bản kiến ​​trúc minh họa cách các dịch vụ cộng tác.

Bước thứ hai trong quy trình là xác định sự phân rã thành các dịch vụ. Có một số chiến lược để lựa chọn. Một chiến lược có nguồn gốc từ ngành kiến ​​trúc doanh nghiệp là xác định các dịch vụ tương ứng với các khả năng kinh doanh. Một chiến lược khác là tổ chức các dịch vụ xung quanh các miền con thiết kế theo miền. Kết quả cuối cùng là các dịch vụ được tổ chức xung quanh các khái niệm kinh doanh thay vì các khái niệm kỹ thuật.

Bước thứ ba trong việc xác định kiến ​​trúc của ứng dụng là xác định API của từng dịch vụ. Để làm được điều đó, bạn chỉ định từng hoạt động hệ thống được xác định trong bước đầu tiên cho một dịch vụ. Một dịch vụ có thể tự triển khai một hoạt động hoàn toàn. Ngoài ra, nó có thể cần phải cộng tác với các dịch vụ khác. Trong trường hợp đó, bạn xác định cách các dịch vụ cộng tác, điều này thường yêu cầu các dịch vụ hỗ trợ các hoạt động bổ sung. Bạn cũng sẽ cần quyết định cơ chế IPC nào mà tôi mô tả trong chương 3 để triển khai API của từng dịch vụ.

Có một số trở ngại đối với quá trình phân tách. Trở ngại đầu tiên là độ trễ mạng. Bạn có thể phát hiện ra rằng một quá trình phân tách cụ thể sẽ không thực tế do có quá nhiều vòng lặp giữa các dịch vụ. Một trở ngại khác đối với quá trình phân tách là giao tiếp đồng bộ giữa các dịch vụ làm giảm tính khả dụng. Bạn có thể cần sử dụng khái niệm về các dịch vụ độc lập, được mô tả trong chương 3. Trở ngại thứ ba là yêu cầu duy trì tính nhất quán của dữ liệu trên các dịch vụ. Bạn thường sẽ cần sử dụng sagas, được thảo luận trong chương 4. Trở ngại thứ tư và cũng là trở ngại cuối cùng đối với quá trình phân tách là cái gọi là god class, được sử dụng trong toàn bộ ứng dụng. May mắn thay, bạn có thể sử dụng các khái niệm từ thiết kế theo miền để loại bỏ god class.

Phần này trước tiên mô tả cách xác định hoạt động của ứng dụng. Sau đó, chúng ta sẽ xem xét các chiến lược và hướng dẫn để phân tích ứng dụng thành các dịch vụ, cũng như các trở ngại đối với quá trình phân tích và cách giải quyết chúng. Cuối cùng, tôi sẽ mô tả cách xác định API của từng dịch vụ.

###### Xác định các hoạt động của hệ thống

Bước đầu tiên trong việc xác định kiến ​​trúc của ứng dụng là xác định các hoạt động của hệ thống. Điểm khởi đầu là các yêu cầu của ứng dụng, bao gồm các câu chuyện của người dùng và các kịch bản người dùng liên quan (lưu ý rằng chúng khác với các kịch bản kiến ​​trúc). Các hoạt động của hệ thống được xác định và xác định bằng quy trình hai bước được thể hiện trong hình 2.6. Quy trình này lấy cảm hứng từ quy trình thiết kế hướng đối tượng được đề cập trong cuốn sách Applying UML and Patterns (Prentice Hall, 2004) của Craig Larman (xem[www.craiglarman.com/wiki/index.php?title=Book\_Applying\_UML\_and\_Patterns](http://www.craiglarman.com/wiki/index.php?title=Book_Applying_UML_and_Patterns) để biết chi tiết). Bước đầu tiên tạo ra mô hình miền cấp cao bao gồm các lớp chính

cung cấp một vốn từ vựng để mô tả các hoạt động của hệ thống. Bước thứ hai xác định các hoạt động của hệ thống và mô tả hành vi của từng hoạt động theo mô hình miền.

Bước chânBước 1 2

Mô hình miền cấp cao

Đặt hàng

Nhà hàng

Mô hình miềnxuất phát từ yêu cầu

Yêu cầu chức năng

tạo đơn hàng()

Là một nhà hàng

Tôi muốn chấp nhận đơn hàng để tôi có thể ...

FTGO

Là một người tiêu dùng

Tôi muốn đặt hàng để có thể ...

chấp nhận đơn hàng()

Bản đồ đến

Vận chuyển

Hoạt động của hệ thống được định nghĩa theo mô hình miền.

**Hình 2.6 Các hoạt động của hệ thống được bắt nguồn từ các yêu cầu của ứng dụng bằng quy trình hai bước. Bước đầu tiên là tạo mô hình miền cấp cao. Bước thứ hai là xác định các hoạt động của hệ thống, được xác địnhvề mặt mô hình miền.**

Mô hình miền làbắt nguồn chủ yếu từ danh từ của các câu chuyện người dùng, và các hoạt động của hệ thống chủ yếu bắt nguồn từ động từ. Bạn cũng có thể định nghĩa mô hình miền bằng một kỹ thuật gọi là Event Storming, mà tôi sẽ nói đến trong chương 5. Hành vi của mỗi hoạt động của hệ thống được mô tả theo tác động của nó lên một hoặc nhiều đối tượng miền và mối quan hệ giữa chúng. Một hoạt động của hệ thống có thể tạo, cập nhật hoặc xóa các đối tượng miền, cũng như tạo hoặc hủy các mối quan hệ giữa chúng.

Chúng ta hãy xem cách xác định mô hình miền cấp cao. Sau đó, tôi sẽ xác định các hoạt động của hệ thống theo mô hình miền.

**CĐỌC MỘT CAO-MÔ HÌNH MIỀN CẤP ĐỘ**

Bước đầu tiên trong quá trình xác định các hoạt động của hệ thống là phác thảo một mô hình miền cấp cao cho ứng dụng. Lưu ý rằng mô hình miền này đơn giản hơn nhiều so với những gì cuối cùng sẽ được triển khai. Ứng dụng thậm chí sẽ không có một mô hình miền duy nhất vì, như bạn sẽ sớm biết, mỗi dịch vụ đều có mô hình miền riêng. Mặc dù là một sự đơn giản hóa đáng kể, một mô hình miền cấp cao vẫn hữu ích ở giai đoạn này vì nó xác định vốn từ vựng để mô tả hành vi của các hoạt động của hệ thống.

Một mô hình miền được tạo ra bằng cách sử dụng các kỹ thuật tiêu chuẩn như phân tích danh từ trong các câu chuyện và kịch bản và nói chuyện với các chuyên gia về miền. Ví dụ, hãy xem xét:

Câu chuyện Đặt hàng. Chúng ta có thể mở rộng câu chuyện đó thành nhiều kịch bản người dùng bao gồm cả câu chuyện này:

Cho một người tiêu dùng Và một nhà hàng

Và địa chỉ/thời gian giao hàng mà nhà hàng đó có thể phục vụ Và tổng đơn hàng đáp ứng mức tối thiểu của nhà hàng

Khi người tiêu dùng đặt hàng cho nhà hàng Sau đó thẻ tín dụng của người tiêu dùng được ủy quyền

Và một đơn hàng được tạo trong trạng thái PENDING\_ACCEPTANCE Và đơn hàng được liên kết với người tiêu dùng

Và thứ tự được liên kết với nhà hàng

Các danh từ trong kịch bản người dùng này gợi ý về sự tồn tại của nhiều lớp khác nhau, bao gồm

Người tiêu dùng,Đặt hàng,Nhà hàng, VàThẻ tín dụng.

Tương tự như vậy, câu chuyện Accept Order có thể được mở rộng thànhmột kịch bản như thế này:

Với đơn hàng đang ở trạng thái PENDING\_ACCEPTANCE và có đơn vị chuyển phát nhanh sẵn sàng giao đơn hàng

Khi một nhà hàng chấp nhận một đơn hàng với lời hứa sẽ chuẩn bị vào một thời điểm cụ thể

Sau đó trạng thái của đơn hàng được thay đổi thành ĐÃ CHẤP NHẬN

Và lệnh promiseByTime được cập nhật theo thời gian đã hứaVà người chuyển phát nhanh được giao nhiệm vụ giao hàng

Kịch bản này cho thấy sự tồn tại của các lớp Courier và Delivery. Kết quả cuối cùng sau một vài lần lặp lại phân tích sẽ là một mô hình miền bao gồm, không có gì ngạc nhiên, các lớp đó và các lớp khác, chẳng hạn như MenuItem và Address. Hình 2.7 là sơ đồ lớp hiển thị các lớp chính.

Được giao cho

Được đặt bởi

Vì

Trả tiền bằng cách sử dụng

Đã thanh toán bằng

phố1 phố2thành phố tiểu bang mã bưu chính

Địa chỉ

tên giá

Mục Menu

Số lượng

Mục đơn hàng

Thời gian giao hàng

Thông tin giao hàng

thẻ tín dụngId

...

Thông tin thanh toán

vĩ độ kinh độ

Vị trí

có sẵn

...

Người chuyển phát nhanh

tên

...

Nhà hàng

tình trạng

...

Đặt hàng

Người tiêu dùng

**Hình 2.7 Các lớp chính trong mô hình miền FTGO**

Trách nhiệm của mỗi lớp như sau:

* Người tiêu dùng—Người tiêu dùng đặt hàng.
* Đặt hàng—Một đơn hàng được đặt bởi người tiêu dùng. Nó mô tả đơn hàng và theo dõitình trạng của nó.
* Mục đơn hàng—Một mục của mộtĐặt hàng.
* Thông tin giao hàng—Thời gian và địa điểm giao hàng.
* Nhà hàng—Một nhà hàng chuẩn bị các đơn hàng để giao đến tay người tiêu dùng.
* Mục Menu—Một món trong thực đơn của nhà hàng.
* Người chuyển phát nhanh—Một đơn vị chuyển phát nhanh giao hàng cho người tiêu dùng. Theo dõi tình trạng sẵn có của đơn vị chuyển phát nhanh và vị trí hiện tại của họ.
* Địa chỉ—Địa chỉ của mộtNgười tiêu dùnghoặc mộtNhà hàng.
* Vị trí—Vĩ độ và kinh độ của mộtNgười chuyển phát nhanh.

Một sơ đồ lớp như trong hình 2.7 minh họa một khía cạnh của kiến ​​trúc ứng dụng. Nhưng nó không hơn gì một bức tranh đẹp nếu không có các kịch bản để làm cho nó hoạt hình. Bước tiếp theo là xác định các hoạt động của hệ thống, tương ứng với các kịch bản kiến ​​trúc.

**DXÁC ĐỊNH HOẠT ĐỘNG CỦA HỆ THỐNG**

Sau khi bạn đã xác định mô hình miền cấp cao, bước tiếp theo là xác định các yêu cầumà ứng dụng phải xử lý. Chi tiết về UI nằm ngoài phạm vi của cuốn sách này, nhưng bạn có thể hình dung rằng trong mỗi kịch bản người dùng, UI sẽ đưa ra yêu cầu đến logic nghiệp vụ phụ trợ để truy xuất và cập nhật dữ liệu. FTGO chủ yếu là một ứng dụng web, nghĩa là hầu hết các yêu cầu đều dựa trên HTTP, nhưng có khả năng một số máy khách có thể sử dụng tin nhắn. Do đó, thay vì cam kết với một giao thức cụ thể, việc sử dụng khái niệm trừu tượng hơn về hoạt động của hệ thống để biểu diễn các yêu cầu là hợp lý.

Có hai loại hoạt động hệ thống:

* *Lệnh*—Các hoạt động hệ thống tạo, cập nhật và xóa dữ liệu
* *Truy vấn*—Các hoạt động hệ thống đọc (truy vấn) dữ liệu

Cuối cùng, các hoạt động hệ thống này sẽ tương ứng với các điểm cuối REST, RPC hoặc nhắn tin, nhưng hiện tại, việc nghĩ về chúng một cách trừu tượng là hữu ích. Trước tiên, hãy xác định một số lệnh.

Một điểm khởi đầu tốt để xác định lệnh hệ thống là phân tích các động từ trong các câu chuyện và kịch bản của người dùng. Ví dụ, hãy xem xét câu chuyện Place Order. Câu chuyện này cho thấy rõ ràng rằng hệ thống phải cung cấp thao tác Create Order. Nhiều câu chuyện khác riêng lẻ được ánh xạ trực tiếp đến các lệnh hệ thống. Bảng 2.1 liệt kê một số lệnh hệ thống chính.

**Bảng 2.1 Các lệnh hệ thống chính cho ứng dụng FTGO**

|  |  |  |  |
| --- | --- | --- | --- |
| **Diễn viên** | **Câu chuyện** | **Yêu cầu** | **Sự miêu tả** |
| Nhà hàng tiêu dùng | Tạo đơn hàng Chấp nhận đơn hàng | tạo đơn hàng() chấp nhận đơn hàng() | Tạo ramột lệnh  Chỉ ra rằng nhà hàng đã chấp nhận đơn hàng và cam kết sẽ chuẩn bị vào thời gian đã chỉ định |

**Bảng 2.1 Các lệnh hệ thống chính cho ứng dụng FTGO*(tiếp theo)***

|  |  |  |  |
| --- | --- | --- | --- |
| **Diễn viên** | **Câu chuyện** | **Yêu cầu** | **Sự miêu tả** |
| Nhà hàng | Đặt hàng đã sẵn sàngđể đón | lưu ýOrderReadyForPickup() | Chỉ ra rằng đơn hàng đã sẵn sàngnhặt lên |
| Người chuyển phát nhanh | Cập nhật vị trí | noteUpdatedLocation() | Cập nhật vị trí hiện tại củangười chuyển phát nhanh |
| Người chuyển phát nhanh | Đã nhận hàng | lưu ýDeliveryPickedUp() | Chỉ ra rằng người chuyển phát nhanh đãđã nhận đơn hàng |
| Người chuyển phát nhanh | Giao hàng đã được giao | lưu ýDeliveryDelivered() | Chỉ ra rằng người chuyển phát đã giao hàngđã thực hiện lệnh |

Một lệnh có một đặc tả định nghĩa các tham số, giá trị trả về và hành vi của nó theo các lớp mô hình miền. Đặc tả hành vi bao gồm các điều kiện tiên quyết phải đúng khi thao tác được gọi và các điều kiện hậu quyết phải đúng sau khi thao tác được gọi. Ví dụ, đây là đặc tả của thao tác hệ thống createOrder():

|  |  |
| --- | --- |
| Hoạt động | tạo đơn hàng(ID người tiêu dùng, phương thức thanh toán, địa chỉ giao hàng, thời gian giao hàng, |
|  | ID nhà hàng, các mục đơn hàng) |
| Trả lại | Mã đơn hàng, … |
| Điều kiện tiên quyết | * Người tiêu dùng tồn tại và có thể đặt hàng. |
|  | * Các mục hàng tương ứng với các mục trong thực đơn của nhà hàng. |
|  | * Địa chỉ và thời gian giao hàng có thể được nhà hàng cung cấp. |
| Điều kiện sau | * Thẻ tín dụng của người tiêu dùng đã được chấp thuận thanh toán tổng giá trị đơn hàng. |
|  | * Một đơn hàng đã được tạo ra trongĐANG CHỜ CHẤP NHẬNtình trạng. |

Các điều kiện tiên quyết phản ánh các điều kiện đã cho trong kịch bản người dùng Place Order được mô tả trước đó. Các điều kiện hậu kỳ phản ánh các thens từ kịch bản. Khi một hoạt động hệ thống được gọi, nó sẽ xác minh các điều kiện tiên quyết và thực hiện các hành động cần thiết để làm cho các điều kiện hậu kỳ trở thành đúng.

Đây là thông số kỹ thuật củachấp nhận đơn hàng()hoạt động của hệ thống:

|  |  |
| --- | --- |
| Hoạt động | acceptOrder(restaurantId, orderId, readyByTime) |
| Trả lại | — |
| Điều kiện tiên quyết | * Cácđơn hàng.trạng tháilàĐANG CHỜ CHẤP NHẬN. * Sẽ có nhân viên chuyển phát nhanh đến giao hàng. |
| Điều kiện sau | * Cácđơn hàng.trạng tháiđã được thay đổi thànhĐÃ CHẤP NHẬN. * Cácđặt hàng.readyByTimeđã được thay đổi thànhsẵn sàng theo thời gian. * Người chuyển phát nhanh được giao nhiệm vụ giao đơn hàng. |

Các điều kiện trước và sau của nó phản ánh kịch bản của người dùng trước đó.

Hầu hết các hoạt động hệ thống có liên quan đến kiến ​​trúc là các lệnh. Tuy nhiên, đôi khi, các truy vấn, truy xuất dữ liệu, cũng quan trọng.

Bên cạnh việc triển khai lệnh, một ứng dụng cũng phải triển khai truy vấn. Các truy vấn cung cấp cho UI thông tin mà người dùng cần để đưa ra quyết định. Ở giai đoạn này, chúng tôi không có thiết kế UI cụ thể nào cho ứng dụng FTGO, nhưng hãy xem xét, ví dụ, luồng khi người tiêu dùng đặt hàng:

**1** Người dùng nhập địa chỉ và thời gian giao hàng.

**2** Hệ thống hiển thị các nhà hàng còn trống.

**3** Người dùng chọn nhà hàng.

**4** Hệ thống hiển thị menu.

**5** Người dùng chọn mục và thanh toán.

**6**Hệ thống tạo ra trật tự.

Kịch bản người dùng này gợi ý các truy vấn sau:

* tìmNhà hàng có sẵn(Địa chỉ giao hàng, Thời gian giao hàng)—Lấy các nhà hàng có thể giao hàng đến địa chỉ giao hàng đã chỉ định vào thời gian đã chỉ định
* tìm Menu Nhà Hàng(id)—Lấy thông tin về một nhà hàng bao gồm các mục trong thực đơn

Trong hai truy vấn,findAvailableRestaurants() có lẽ là quan trọng nhất về mặt kiến ​​trúc. Đây là truy vấn phức tạp liên quan đến tìm kiếm địa lý. Thành phần tìm kiếm địa lý của truy vấn bao gồm việc tìm tất cả các điểm—nhà hàng—gần một vị trí—địa chỉ giao hàng. Nó cũng lọc ra những nhà hàng đóng cửa khi đơn hàng cần được chuẩn bị và nhận. Hơn nữa, hiệu suất là rất quan trọng, vì truy vấn này được thực hiện bất cứ khi nào người tiêu dùng muốn đặt hàng.

Mô hình miền cấp cao và các hoạt động hệ thống nắm bắt những gì ứng dụng thực hiện. Chúng giúp thúc đẩy việc xác định kiến ​​trúc của ứng dụng. Hành vi của mỗi hoạt động hệ thống được mô tả theo mô hình miền. Mỗi hoạt động hệ thống quan trọng đại diện cho một kịch bản có ý nghĩa về mặt kiến ​​trúc, là một phần của mô tả về kiến ​​trúc.

Sau khi các hoạt động của hệ thống đã được xác định, bước tiếp theo là xác định các dịch vụ của ứng dụng. Như đã đề cập trước đó, không có quy trình cơ học nào cần tuân theo. Tuy nhiên, có nhiều chiến lược phân tích khác nhau mà bạn có thể sử dụng. Mỗi chiến lược giải quyết vấn đề theo một góc nhìn khác nhau và sử dụng thuật ngữ riêng của mình. Nhưng với tất cả các chiến lược, kết quả cuối cùng đều giống nhau: một kiến ​​trúc bao gồm các dịch vụ chủ yếu được tổ chức xung quanh các khái niệm kinh doanh hơn là các khái niệm kỹ thuật.

Hãy cùng xem xét chiến lược đầu tiên, chiến lược này định nghĩa các dịch vụ tương ứng với năng lực kinh doanh.

###### Xác định dịch vụ bằng cách áp dụng Phân tích theo doanh nghiệpmẫu khả năng

Một chiến lược để tạo ra kiến ​​trúc vi dịch vụ là phân tích theo khả năng kinh doanh. Một khái niệm từ mô hình kiến ​​trúc kinh doanh, khả năng kinh doanh là một cái gì đó mà một doanh nghiệp thực hiện để tạo ra giá trị. Bộ khả năng cho một doanh nghiệp nhất định phụ thuộc vào loại hình kinh doanh. Ví dụ, khả năng của một công ty bảo hiểm thường bao gồm Bảo hiểm, Quản lý khiếu nại, Thanh toán, Tuân thủ, v.v. Khả năng của một cửa hàng trực tuyến bao gồm Quản lý đơn hàng, Quản lý hàng tồn kho, Vận chuyển, v.v.

**Mẫu: Phân tích theo năng lực kinh doanh**

Xác định các dịch vụ tương ứng với khả năng kinh doanh. Xem[http://microservices.io/](http://microservices.io/patterns/decomposition/decompose-by-business-capability.html) [mẫu/phân hủy/phân hủy theo năng lực kinh doanh.html](http://microservices.io/patterns/decomposition/decompose-by-business-capability.html).

**BNĂNG LỰC KINH DOANH XÁC ĐỊNH NHỮNG GÌ MỘT TỔ CHỨC LÀM**

MỘTnăng lực kinh doanh của tổ chức nắm bắt được hoạt động kinh doanh của tổ chức. Chúng thường ổn định, trái ngược với cách một tổ chức tiến hành hoạt động kinh doanh của mình, thay đổi theo thời gian, đôi khi là rất nhiều. Điều đó đặc biệt đúng ngày nay, với việc sử dụng công nghệ ngày càng tăng nhanh để tự động hóa nhiều quy trình kinh doanh. Ví dụ, cách đây không lâu, bạn đã gửi séc tại ngân hàng của mình bằng cách giao chúng cho nhân viên thu ngân. Sau đó, có thể gửi séc bằng máy ATM. Ngày nay, bạn có thể gửi hầu hết các séc một cách thuận tiện bằng điện thoại thông minh của mình. Như bạn có thể thấy, năng lực kinh doanh séc gửi tiền vẫn ổn định, nhưng cách thức thực hiện đã thay đổi đáng kể.

**TÔIXÁC ĐỊNHNĂNG LỰC KINH DOANH**

Năng lực kinh doanh của một tổ chức được xác định bằng cách phân tích mục đích, cấu trúc và quy trình kinh doanh của tổ chức đó. Mỗi năng lực kinh doanh có thể được coi là một dịch vụ, ngoại trừ việc nó hướng đến kinh doanh hơn là kỹ thuật. Đặc điểm kỹ thuật của nó bao gồm nhiều thành phần khác nhau, bao gồm đầu vào, đầu ra và các thỏa thuận cấp độ dịch vụ. Ví dụ, đầu vào cho năng lực bảo hiểm là đơn đăng ký của người tiêu dùng và đầu ra bao gồm phê duyệt và giá.

Một khả năng kinh doanh thường tập trung vào một đối tượng kinh doanh cụ thể. Ví dụ, đối tượng kinh doanh Claim là trọng tâm của khả năng quản lý Claim. Một khả năng thường có thể được phân tách thành các khả năng phụ. Ví dụ, khả năng quản lý Claim có một số khả năng phụ, bao gồm Quản lý thông tin Claim, Xem xét Claim và Quản lý thanh toán Claim.

Không khó để hình dung rằng năng lực kinh doanh của FTGO bao gồm những điều sau:

* Quản lý nhà cung cấp
  + *Quản lý chuyển phát nhanh*—Quản lýthông tin chuyển phát nhanh
  + *Quản lý thông tin nhà hàng*—Quản lý thực đơn nhà hàng và các thông tin khác, bao gồm địa điểm và giờ mở cửa
    - * Quản lý người tiêu dùng—Quản lý thông tin về người tiêu dùng
      * Tiếp nhận và thực hiện đơn hàng
        + *Quản lý đơn hàng*—Cho phép người tiêu dùng tạo và quản lý đơn hàng
        + *Quản lý đơn hàng nhà hàng*—Quản lý việc chuẩn bị đơn hàng tại nhà hàng
        + Hậu cần
        + *Quản lý tính khả dụng của dịch vụ chuyển phát nhanh*—Quản lý tính khả dụng theo thời gian thực của người chuyển phát để giao hàng
        + *Quản lý giao hàng*—Giao hàng cho người tiêu dùng
      * Kế toán
        + *Kế toán tiêu dùng*—Quản lý việc thanh toán của người tiêu dùng
        + *Kế toán nhà hàng*—Quản lý thanh toán cho nhà hàng
        + *Kế toán chuyển phát nhanh*—Quản lý thanh toán cho người chuyển phát nhanh
      * …

Các khả năng cấp cao nhất bao gồm Quản lý nhà cung cấp, Quản lý người tiêu dùng, Tiếp nhận và thực hiện đơn hàng, và Kế toán. Có thể sẽ có nhiều khả năng cấp cao khác, bao gồm các khả năng liên quan đến tiếp thị. Hầu hết các khả năng cấp cao nhất được phân chia thành các khả năng phụ. Ví dụ, Tiếp nhận và thực hiện đơn hàng được phân chia thành năm khả năng phụ.

Một khía cạnh thú vị của hệ thống phân cấp năng lực này là có ba năng lực liên quan đến nhà hàng: Quản lý thông tin nhà hàng, Quản lý đơn hàng nhà hàng và Kế toán nhà hàng. Đó là vì chúng đại diện cho ba khía cạnh rất khác nhau của hoạt động nhà hàng.

Tiếp theo chúng ta sẽ xem cách sử dụng khả năng kinh doanh để xác định dịch vụ.

**FROM NĂNG LỰC KINH DOANH ĐẾN DỊCH VỤ**

Sau khi xác định được các khả năng kinh doanh, bạn sẽ xác định một dịch vụ cho từng khả năng hoặc nhóm các khả năng liên quan. Hình 2.8 cho thấy ánh xạ từ các khả năng sang các dịch vụ cho ứng dụng FTGO. Một số khả năng cấp cao nhất, chẳng hạn như khả năng Kế toán, được ánh xạ sang các dịch vụ. Trong các trường hợp khác, các khả năng phụ được ánh xạ sang các dịch vụ.

Quyết định về mức độ nào của hệ thống phân cấp năng lực để ánh xạ tới các dịch vụ, vì có phần chủ quan. Lý do biện minh cho việc ánh xạ cụ thể này của tôi như sau:

* + - * Tôi đã ánh xạ các năng lực phụ của Quản lý nhà cung cấp thành hai dịch vụ, vì Nhà hàng và Giao hàng là những loại nhà cung cấp rất khác nhau.
      * Tôi đã lập bản đồ khả năng tiếp nhận và thực hiện đơn hàng thành ba dịch vụ, mỗi dịch vụ chịu trách nhiệm cho các giai đoạn khác nhau của quy trình. Tôi đã kết hợp khả năng quản lý tính khả dụng của dịch vụ chuyển phát nhanh và khả năng quản lý giao hàng và lập bản đồ chúng thành một dịch vụ duy nhất vì chúng có mối liên hệ chặt chẽ với nhau.
      * Tôi đã ánh xạ khả năng Kế toán vào dịch vụ riêng của nó vì các loại kế toán khác nhau có vẻ tương tự nhau.

Dịch vụ

Quản lý chuyển phát nhanh

Phân cấp năng lực

Quản lý nhà cung cấp

Nhà hàngquản lý thông tin

Quản lý đơn hàng

**Người chuyển phát nhanh và nhà hànglà những loại nhà cung cấp rất khác nhau**

**=> các dịch vụ khác nhau.**

Dịch vụ nhà hàng

Dịch vụ chuyển phát nhanh

**Ba dịch vụ khác nhau xử lý các giai đoạn khác nhau của việc tiếp nhận và thực hiện đơn hàng**

Dịch vụ nhà bếp

Dịch vụ đặt hàng

Dịch vụ khách hàng

Hậu cần

Người chuyển phát nhanhquản lý khả dụng

Dịch vụ giao hàng

Tiếp nhận và thực hiện đơn hàng

Quản lý người tiêu dùng

Quản lý phiếu đặt hàng nhà hàng

Quản lý giao hàng

Kế toán

Kế toán tiêu dùng

**Hãy xử lý việc thanh toán và lập hóa đơn giống nhau ngay bây giờ.**

Kế toán nhà hàng

Dịch vụ kế toán

Kế toán chuyển phát nhanh

**Hình 2.8 Ánh xạ năng lực kinh doanh FTGO với các dịch vụ. Năng lực ở nhiều cấp độ khác nhau củahệ thống phân cấp năng lực được ánh xạ tới các dịch vụ.**

Sau này, có thể tách riêng phần thanh toán (của Nhà hàng và Người chuyển phát nhanh) và phần lập hóa đơn (của Người tiêu dùng).

Một lợi ích chính của việc tổ chức các dịch vụ xung quanh các khả năng là vì chúng ổn định, nên kiến ​​trúc kết quả cũng sẽ tương đối ổn định. Các thành phần riêng lẻ của kiến ​​trúc có thể phát triển khi khía cạnh cách thức kinh doanh thay đổi, nhưng kiến ​​trúc vẫn không thay đổi.

Nói như vậy, điều quan trọng cần nhớ là các dịch vụ được hiển thị trong hình 2.8 chỉ là nỗ lực đầu tiên trong việc xác định kiến ​​trúc. Chúng có thể phát triển theo thời gian khi chúng ta tìm hiểu thêm về miền ứng dụng. Đặc biệt, một bước quan trọng trong quá trình xác định kiến ​​trúc là điều tra cách các dịch vụ cộng tác trong từng dịch vụ kiến ​​trúc chính. Ví dụ, bạn có thể phát hiện ra rằng một phân tích cụ thể không hiệu quả do giao tiếp giữa các tiến trình quá mức và bạn phải kết hợp các dịch vụ. Ngược lại, một dịch vụ có thể trở nên phức tạp hơn

điểm mà việc chia thành nhiều dịch vụ trở nên đáng giá. Hơn nữa, trong phần 2.2.5, tôi mô tả một số trở ngại đối với việc phân tích có thể khiến bạn phải xem xét lại quyết định của mình.

Chúng ta hãy cùng xem nhémột cách khác để phân tích ứng dụng dựa trên thiết kế theo miền.

###### Xác định các dịch vụ bằng cách áp dụng Phân tích theomẫu miền phụ

DDD, như được mô tả trong cuốn sách tuyệt vời Domain-driven design của Eric Evans (Addison-Wesley Professional, 2003), là một phương pháp tiếp cận để xây dựng các ứng dụng phần mềm phức tạp tập trung vào việc phát triển mô hình miền hướng đối tượng. Chế độ miền nắm bắt kiến ​​thức về một miền dưới dạng có thể được sử dụng để giải quyết các vấn đề trong miền đó. Nó định nghĩa từ vựng được nhóm sử dụng, thứ mà DDD gọi là Ngôn ngữ phổ biến. Mô hình miền được phản ánh chặt chẽ trong thiết kế và triển khai ứng dụng. DDD có hai khái niệm cực kỳ hữu ích khi áp dụng kiến ​​trúc vi dịch vụ: miền phụ và ngữ cảnh bị giới hạn.

**Mẫu: Phân tích theo miền phụ**

Xác định các dịch vụ tương ứng với các miền phụ DDD. Xem[http://microservices.io](http://microservices.io/patterns/decomposition/decompose-by-subdomain.html)

[/patterns/decomposition/decompose-by-subdomain.html](http://microservices.io/patterns/decomposition/decompose-by-subdomain.html).

DDD khá khác so với cách tiếp cận truyền thống đối với mô hình hóa doanh nghiệp, vốn tạo ra một mô hình duy nhất cho toàn bộ doanh nghiệp. Trong một mô hình như vậy, ví dụ, sẽ có một định nghĩa duy nhất cho mỗi thực thể kinh doanh, chẳng hạn như khách hàng, đơn hàng, v.v. Vấn đề với loại mô hình hóa này là việc khiến các bộ phận khác nhau của một tổ chức đồng ý về một mô hình duy nhất là một nhiệm vụ to lớn. Ngoài ra, điều đó có nghĩa là theo quan điểm của một bộ phận nhất định của tổ chức, mô hình quá phức tạp so với nhu cầu của họ. Hơn nữa, mô hình miền có thể gây nhầm lẫn vì các bộ phận khác nhau của tổ chức có thể sử dụng cùng một thuật ngữ cho các khái niệm khác nhau hoặc các thuật ngữ khác nhau cho cùng một khái niệm. DDD tránh những vấn đề này bằng cách xác định nhiều mô hình miền, mỗi mô hình có phạm vi rõ ràng.

DDD định nghĩa một mô hình miền riêng cho mỗi miền phụ. Một miền phụ là một phần của miền, thuật ngữ của DDD dùng để chỉ không gian vấn đề của ứng dụng. Các miền phụ được xác định bằng cách sử dụng cùng một phương pháp như xác định năng lực kinh doanh: phân tích doanh nghiệp và xác định các lĩnh vực chuyên môn khác nhau. Kết quả cuối cùng rất có thể là các miền phụ tương tự như năng lực kinh doanh. Các ví dụ về miền phụ trong FTGO bao gồm Nhận đơn hàng, Quản lý đơn hàng, Quản lý bếp, Giao hàng và Tài chính. Như bạn có thể thấy, các miền phụ này rất giống với các năng lực kinh doanh được mô tả trước đó.

DDD gọi phạm vi của một mô hình miền là một bối cảnh bị giới hạn. Một ngữ cảnh bị giới hạn bao gồm các hiện vật mã thực hiện mô hình. Khi sử dụng kiến ​​trúc vi dịch vụ, mỗi ngữ cảnh bị giới hạn là một dịch vụ hoặc có thể là một tập hợp các dịch vụ. Chúng ta có thể tạo ra một kiến ​​trúc vi dịch vụ bằng cách áp dụng DDD và xác định một dịch vụ cho mỗi miền phụ. Hình 2.9 cho thấy cách các miền phụ ánh xạ tới các dịch vụ, mỗi miền có mô hình miền riêng.

Tên miền FTGO

Bản đồ đến

Nhận đơn hàng

miền phụ

Bản đồ đến

Vận chuyển

miền phụ

Phòng bếp

miền phụ

Bản đồ đến

....

miền phụ

Bản đồ đến

Kế toán

miền phụ

Bản đồ đến

Dịch vụ kế toán

Mô hình miền kế toán

.... Dịch vụ

Dịch vụ nhà bếp

Mô hình miền bếp

Dịch vụ giao hàng

Mô hình miền phân phối

Dịch vụ đặt hàng

Mô hình miền đặt hàng

**Hình 2.9 Từ miền phụ đến dịch vụ: mỗi miền phụ của miền ứng dụng FTGOđược ánh xạ tới một dịch vụ có mô hình miền riêng.**

DDD và kiến ​​trúc dịch vụ vi mô gần như hoàn toàn phù hợp. Khái niệm DDD về miền phụ và ngữ cảnh bị giới hạn ánh xạ tốt đến các dịch vụ trong kiến ​​trúc dịch vụ vi mô. Ngoài ra, khái niệm về các nhóm tự chủ sở hữu các dịch vụ của kiến ​​trúc dịch vụ vi mô hoàn toàn phù hợp với khái niệm của DDD về mỗi mô hình miền do một nhóm duy nhất sở hữu và phát triển. Thậm chí tốt hơn, như tôi sẽ mô tả sau trong phần này, khái niệm về miền phụ với mô hình miền riêng của nó là một cách tuyệt vời để loại bỏ các lớp thần thánh và do đó giúp phân tách dễ dàng hơn.

Phân tích theo miền phụ và Phân tích theo khả năng kinh doanh là hai mô hình chính để xác định kiến ​​trúc dịch vụ vi mô của ứng dụng. Tuy nhiên, có một số hướng dẫn hữu ích để phân tích có nguồn gốc từ thiết kế hướng đối tượng. Chúng ta hãy cùng xem xét chúng.

###### Hướng dẫn phân hủy

Cho đến nay trong chương này, chúng ta đã xem xét các cách chính để xác định kiến ​​trúc vi dịch vụ. Chúng ta cũng có thể điều chỉnh và sử dụng một số nguyên tắc từ thiết kế hướng đối tượng khi áp dụng mô hình kiến ​​trúc vi dịch vụ. Các nguyên tắc này được Robert C. Martin tạo ra và được mô tả trong cuốn sách kinh điển của ông Designing Object Oriented C++ Applications Using The Booch*Phương pháp*(Prentice Hall, 1995). Nguyên tắc đầu tiênlà Nguyên tắc trách nhiệm đơn (SRP), để xác định trách nhiệm của một lớp. Nguyên tắc thứ hai là Nguyên tắc đóng chung (CCP), để tổ chức các lớp thành các gói. Chúng ta hãy xem xét các nguyên tắc này và xem cách chúng có thể được áp dụng vào kiến ​​trúc dịch vụ vi mô.

**STiếng AnhRTRÁCH NHIỆMPNGUYÊN TẮC**

Một trong những mục tiêu chính của kiến ​​trúc và thiết kế phần mềm là xác định trách nhiệm của từng thành phần phần mềm. Nguyên tắc trách nhiệm đơn lẻ như sau:

*Mỗi lớp chỉ nên có một lý do để thay đổi.*

Robert C. Martin

Mỗi trách nhiệm mà một lớp có là một lý do tiềm năng để lớp đó thay đổi. Nếu một lớp có nhiều trách nhiệm thay đổi độc lập, lớp đó sẽ không ổn định. Bằng cách tuân theo SRP, bạn định nghĩa các lớp mà mỗi lớp có một trách nhiệm duy nhất và do đó có một lý do duy nhất để thay đổi.

Chúng ta có thể áp dụng SRP khi xác định kiến ​​trúc dịch vụ vi mô và tạo ra các dịch vụ nhỏ, gắn kết, mỗi dịch vụ có một trách nhiệm duy nhất. Điều này sẽ làm giảm kích thước của các dịch vụ và tăng tính ổn định của chúng. Kiến trúc FTGO mới là một ví dụ về SRP trong hành động. Mỗi khía cạnh của việc đưa thực phẩm đến tay người tiêu dùng—nhận đơn hàng, chuẩn bị đơn hàng và giao hàng—đều là trách nhiệm của một dịch vụ riêng biệt.

**COMMONCMẤT MÁTPNGUYÊN TẮC**

Nguyên tắc hữu ích khác là Nguyên tắc đóng chung:

*Các lớp trong một gói phải được đóng lại với nhau để chống lại cùng một loại thay đổi.thay đổi ảnh hưởng đến một gói sẽ ảnh hưởng đến tất cả các lớp trong gói đó.*

Robert C. Martin

Ý tưởng là nếu hai lớp thay đổi theo cùng một bước vì cùng một lý do cơ bản, thì chúng thuộc về cùng một gói. Ví dụ, có lẽ các lớp đó triển khai một khía cạnh khác nhau của một quy tắc kinh doanh cụ thể. Mục tiêu là khi quy tắc kinh doanh đó thay đổi, các nhà phát triển chỉ cần thay đổi mã trong một số ít gói (lý tưởng nhất là chỉ một). Việc tuân thủ CCP cải thiện đáng kể khả năng bảo trì của một ứng dụng.

Chúng ta có thể áp dụng CCP khi tạo kiến ​​trúc dịch vụ vi mô và đóng gói các thành phần thay đổi vì cùng một lý do vào cùng một dịch vụ. Làm như vậy sẽ giảm thiểu

số lượng dịch vụ cần thay đổi và triển khai khi một số yêu cầu thay đổi. Lý tưởng nhất là một thay đổi chỉ ảnh hưởng đến một nhóm và một dịch vụ duy nhất. CCP là thuốc giải cho mô hình phản khối phân tán.

SRP và CCP là 2 trong số 11 nguyên tắc được phát triển bởiBob Martin. Chúng đặc biệt hữu ích khi phát triển kiến ​​trúc vi dịch vụ. Chín nguyên tắc còn lại được sử dụng khi thiết kế các lớp và gói. Để biết thêm thông tin về SRP, CCP và các nguyên tắc OOD khác, hãy xem bài viết “Các nguyên tắc của thiết kế hướng đối tượng” trên trang web của Bob Martin ([http://butunclebob.com/ArticleS.UncleBob](http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod)

[.Nguyên lý của Ood](http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod)).

Phân tích theo năng lực kinh doanh vàtheo miền phụ cùng với SRP và CCP là những kỹ thuật tốt để phân tích ứng dụng thành các dịch vụ. Để áp dụng chúng và phát triển thành công kiến ​​trúc dịch vụ vi mô, bạn phải giải quyết một số vấn đề về quản lý giao dịch và giao tiếp giữa các quy trình.

###### Những trở ngại khi phân tích ứng dụng thành các dịch vụ

Trên bề mặt, chiến lược tạo kiến ​​trúc dịch vụ vi mô bằng cách xác định các dịch vụ tương ứng với năng lực kinh doanh hoặc miền phụ có vẻ đơn giản. Tuy nhiên, bạn có thể gặp phải một số trở ngại:

* + - * Độ trễ mạng
      * Giảm khả năng sử dụng do đồng bộgiao tiếp
      * Duy trì tính nhất quán của dữ liệu trên các dịch vụ
      * Có được mộtquan điểm nhất quán về dữ liệu
      * Các lớp của Chúa ngăn chặn sự phân hủy

Hãy cùng xem xét từng trở ngại, bắt đầu với độ trễ mạng.

**NĐỘ TRỄ ETWORK**

*Độ trễ mạng*là mối quan tâm thường trực trong hệ thống phân tán. Bạn có thể phát hiện ra rằng một phân tách cụ thể thành các dịch vụ dẫn đến một số lượng lớn các chuyến khứ hồi giữa hai dịch vụ. Đôi khi, bạn có thể giảm độ trễ xuống mức có thể chấp nhận được bằng cách triển khai API hàng loạt để truy xuất nhiều đối tượng trong một chuyến khứ hồi duy nhất. Nhưng trong các tình huống khác, giải pháp là kết hợp các dịch vụ, thay thế IPC đắt tiền bằng các lệnh gọi phương thức hoặc hàm ở cấp độ ngôn ngữ.

**SGIAO TIẾP LIÊN TIẾP KHÔNG ĐỒNG BỘGIẢM SỰ CÓ SẴN**

Một vấn đề khác là làm thế nào để triển khai giao tiếp giữa các dịch vụ theo cách không làm giảm tính khả dụng. Ví dụ, cách đơn giản nhất để triển khaihoạt động createOrder() là để Order Service đồng bộ gọi các dịch vụ khác bằng REST. Nhược điểm của việc sử dụng giao thức như REST là nó làm giảm tính khả dụng của Order Service. Nó sẽ không thể tạo đơn hàng nếu bất kỳ dịch vụ nào khác không khả dụng. Đôi khi đây là một sự đánh đổi đáng giá, nhưng trong chương 3, bạn sẽ biết rằng sử dụng nhắn tin không đồng bộ, loại bỏ sự kết hợp chặt chẽ và cải thiện tính khả dụng, thường là lựa chọn tốt hơn.

**TôiĐẢM BẢO SỰ NHẤT QUÁN CỦA DỮ LIỆU TRÊN CÁC DỊCH VỤ**

Một thách thức khác là duy trì tính nhất quán của dữ liệu trên nhiều dịch vụ. Một số hoạt động của hệ thống cần cập nhật dữ liệu trong nhiều dịch vụ. Ví dụ, khi một nhà hàng chấp nhậnmột đơn hàng, các bản cập nhật phải diễn ra trong cả Dịch vụ Bếpvà Dịch vụ giao hàng. Dịch vụ nhà bếp thay đổi trạng thái của Vé. Dịch vụ giao hàng lên lịch giao đơn hàng. Cả hai bản cập nhật này phải được thực hiện một cách nguyên tử.

Giải pháp truyền thống là sử dụng cơ chế quản lý giao dịch phân tán, dựa trên cam kết, hai giai đoạn. Nhưng như bạn sẽ thấy trong chương 4, đây không phải là lựa chọn tốt cho các ứng dụng hiện đại và bạn phải sử dụng một cách tiếp cận rất khác để quản lý giao dịch, một saga. Một saga là một chuỗi các giao dịch cục bộ được phối hợp bằng cách sử dụng tin nhắn. Saga phức tạp hơn các giao dịch ACID truyền thống nhưng chúng hoạt động tốt trong nhiều tình huống. Một hạn chế của saga là chúng cuối cùng là nhất quán. Nếu bạn cần cập nhật một số dữ liệu một cách nguyên tử, thì dữ liệu đó phải nằm trong một dịch vụ duy nhất, điều này có thể là trở ngại cho việc phân tách.

**ỒĐẠT ĐƯỢC MỘT CÁI NHÌN NHẤT QUÁN VỀ DỮ LIỆU**

Một trở ngại khác đối với quá trình phân tách là không thể có được chế độ xem dữ liệu thực sự nhất quán trên nhiều cơ sở dữ liệu. Trong một ứng dụng đơn khối, các thuộc tính của giao dịch ACID đảm bảo rằng một truy vấn sẽ trả về chế độ xem nhất quán của cơ sở dữ liệu. Ngược lại, trong kiến ​​trúc dịch vụ vi mô, mặc dù cơ sở dữ liệu của mỗi dịch vụ là nhất quán, bạn không thể có được chế độ xem dữ liệu nhất quán toàn cầu. Nếu bạn cần chế độ xem nhất quán của một số dữ liệu, thì dữ liệu đó phải nằm trong một dịch vụ duy nhất, điều này có thể ngăn chặn quá trình phân tách. May mắn thay, trong thực tế, điều này hiếm khi là vấn đề.

**GĐẠI HỌCCÁC LỚP NGĂN CHẶN SỰ PHÂN HỦY**

Một trở ngại khác đối với sự phân hủy là sự tồn tại của cái gọi là các lớp thần thánh. Các lớp thần thánh

là các lớp phình to được sử dụng trong toàn bộ ứng dụng([http://wiki.c2.com/](http://wiki.c2.com/?GodClass)

[?Lớp học của Chúa](http://wiki.c2.com/?GodClass)). Một lớp god thường triển khai logic nghiệp vụ cho nhiều khía cạnh khác nhau của ứng dụng. Nó thường có một số lượng lớn các trường được ánh xạ vào một bảng cơ sở dữ liệu với nhiều cột. Hầu hết các ứng dụng đều có ít nhất một trong những lớp này, mỗi lớp đại diện cho một khái niệm cốt lõi của miền: tài khoản trong ngân hàng, đơn đặt hàng trong thương mại điện tử, chính sách trong bảo hiểm, v.v. Vì một lớp god kết hợp trạng thái và hành vi cho nhiều khía cạnh khác nhau của ứng dụng, nên đây là một trở ngại không thể vượt qua đối với việc chia tách bất kỳ logic nghiệp vụ nào sử dụng nó thành các dịch vụ.

Lớp Order là một ví dụ tuyệt vời về một lớp god trong ứng dụng FTGO. Điều đó không có gì đáng ngạc nhiên—suy cho cùng, mục đích của FTGO là giao đơn đặt hàng thực phẩm cho khách hàng. Hầu hết các phần của hệ thống đều liên quan đến đơn đặt hàng. Nếu ứng dụng FTGO có một mô hình miền duy nhất, lớp Order sẽ là một lớp rất lớn. Nó sẽ có trạng thái và hành vi tương ứng với nhiều phần khác nhau của ứng dụng. Hình 2.10 cho thấy cấu trúc của lớp này sẽ được tạo bằng các kỹ thuật mô hình hóa truyền thống.

Như bạn có thể thấy, lớp Order có các trường và phương thức tương ứng với xử lý đơn hàng, quản lý đơn hàng nhà hàng, giao hàng và thanh toán. Lớp này cũng có một mô hình trạng thái phức tạp, do thực tế là một mô hình phải mô tả các chuyển đổi trạng thái

<<giao hàng>> gánCourier() notePickedUp() noteDelivered()

<<nhà hàng>> chấp nhận() từ chối()

lưu ýReadyForPickup()

<<orderTaking>> tạo() hủy()

<<thanh toán>> transactionid

<<giao hàng>> thời gian nhận hàng

Tổng đơn hàngThời gian giao hàngTrạng thái

Đặt hàng

Mục đơn hàng

Thông tin thanh toán

Nhà hàng

Người tiêu dùng

Người chuyển phát nhanh

Địa chỉ

**Hình 2.10 CácĐặt hàngTầng lớp thượng lưu phải đảm nhiệm rất nhiều trách nhiệm.**

từ các phần khác nhau của ứng dụng. Ở dạng hiện tại, lớp này làm cho nó cực kỳkhó để phân chia mã thành các dịch vụ.

Một giải pháp là đóng gói lớp Order thành một thư viện và tạo một cơ sở dữ liệu Order trung tâm. Tất cả các dịch vụ xử lý các đơn hàng đều sử dụng thư viện này và truy cập vào cơ sở dữ liệu access. Vấn đề với cách tiếp cận này là nó vi phạm một trong những nguyên tắc chính của kiến ​​trúc microservice và dẫn đến sự kết hợp chặt chẽ không mong muốn. Ví dụ, bất kỳ thay đổi nào đối với lược đồ Order đều yêu cầu các nhóm phải cập nhật mã của họ theo từng bước.

Một giải pháp khác là đóng gói cơ sở dữ liệu Order trong Order Service, được các dịch vụ khác gọi đến để truy xuất và cập nhật các đơn hàng. Vấn đề với thiết kế đó là Order Service sẽ là một dịch vụ dữ liệu với mô hình miền thiếu sức sống chứa ít hoặc không có logic kinh doanh. Cả hai tùy chọn này đều không hấp dẫn, nhưng may mắn thay, DDD cung cấp một giải pháp.

Một cách tiếp cận tốt hơn nhiều là áp dụng DDD và xử lý từng dịch vụ như một miền phụ riêng biệt với mô hình miền riêng. Điều này có nghĩa là mỗi dịch vụ trong ứng dụng FTGO có liên quan đến đơn hàng đều có mô hình miền riêng với phiên bản lớp Đơn hàng của nó. Một ví dụ tuyệt vời về lợi ích của nhiều mod miềnels là Dịch vụ giao hàng. Quan điểm của nó về Đơn hàng, được thể hiện trong hình 2.11, cực kỳ đơn giản: địa chỉ nhận hàng, thời gian nhận hàng, địa chỉ giao hàng và thời gian giao hàng. Hơn nữa, thay vì gọi là Đơn hàng, Dịch vụ giao hàng sử dụng tên gọi phù hợp hơn là Giao hàng.

Địa điểm đón

Địa chỉ

|  |  |
| --- | --- |
| Vận chuyển | Địa điểm giao hàng |
| trạng thái theo lịch trìnhThời gian nhận hàngThời gian giao hàng theo lịch trình | Được giao cho |

Người chuyển phát nhanh

**Hình 2.11 CácDịch vụ giao hàngmô hình miền**

CácDịch vụ giao hàngkhông quan tâmtrong bất kỳ thuộc tính nào khác của một đơn hàng.

CácDịch vụ nhà bếpcũng có một cái nhìn đơn giản hơn nhiều về một đơn hàng. Phiên bản của nóĐặt hàngđược gọi là mộtVé. Như hình 2.12 cho thấy, mộtVéchỉ đơn giản bao gồm một trạng thái,yêu cầuThời gian giao hàng, Mộtchuẩn bị theo thời gianvà danh sách các mục cho biết nhà hàng cần chuẩn bị những gì. Không quan tâm đến người tiêu dùng, thanh toán, giao hàng, v.v.

số lượng mặt hàng

Mục Vé

trạng thái yêu cầuThời gian giao hàngđã chuẩn bịTheoThời gian

Vé

**Hình 2.12 CácDịch vụ nhà bếpmô hình miền**

Dịch vụ Order có chế độ xem phức tạp nhất về một đơn hàng, được hiển thị trong hình 2.13. Mặc dù có khá nhiều trường và phương thức, nhưng nó vẫn đơn giản hơn nhiều so với phiên bản gốc.

Đặt hàng

trạng thái đơn hàngTổng thời gian giao hàng

...

Mục đơn hàng

Nhà hàng

Người tiêu dùng

Thông tin thanh toán

Địa chỉ

**Hình 2.13 CácDịch vụ đặt hàngmô hình miền**

Lớp Order trong mỗi mô hình miền đại diện cho các khía cạnh khác nhau của cùng một thực thể kinh doanh Order. Ứng dụng FTGO phải duy trì tính nhất quán giữa các đối tượng khác nhau này trong các dịch vụ khác nhau. Ví dụ, sau khi Order Service đã ủy quyền

thẻ tín dụng của người tiêu dùng, nó phải kích hoạt việc tạo ra Vé trong BếpDịch vụ. Tương tự, nếu nhà hàng từ chối đơn hàng qua Dịch vụ nhà bếp, đơn hàng phải được hủy trong dịch vụ Dịch vụ đặt hàng và khách hàng được ghi có trong dịch vụ thanh toán. Trong chương 4, bạn sẽ học cách duy trì tính nhất quán giữa các dịch vụ, sử dụng các sagas cơ chế hướng sự kiện đã đề cập trước đó.

Ngoài việc tạo ra các thách thức về mặt kỹ thuật, việc có nhiều mô hình miền cũng tác động đến việc triển khai trải nghiệm người dùng. Một ứng dụng phải chuyển đổi giữa trải nghiệm người dùng, là mô hình miền của riêng nó, và các mô hình miền của từng dịch vụ. Ví dụ, trong ứng dụng FTGO, trạng thái Đơn hàng được hiển thị cho người tiêu dùng được lấy từ thông tin Đơn hàng được lưu trữ trong nhiều dịch vụ. Bản dịch này thường được xử lý bởi cổng API, được thảo luận trong chương 8. Bất chấp những thách thức này, điều cần thiết là bạn phải xác định và loại bỏ các lớp thần thánh khi định nghĩa kiến ​​trúc vi dịch vụ.

Bây giờ chúng ta sẽ xem cách xác định API dịch vụ.

###### Xác định API dịch vụ

Cho đến nay, chúng ta có danh sách các hoạt động hệ thống và danh sách các dịch vụ tiềm năng. Bước tiếp theo là xác định API của từng dịch vụ: các hoạt động và sự kiện của nó. Một hoạt động API dịch vụ tồn tại vì một trong hai lý do sau: một số hoạt động tương ứng với các hoạt động hệ thống. Chúng được gọi bởi các máy khách bên ngoài và có thể là các dịch vụ khác. Các hoạt động khác tồn tại để hỗ trợ sự cộng tác giữa các dịch vụ. Các hoạt động này chỉ được gọi bởi các dịch vụ khác.

Một dịch vụ phát hành các sự kiện chủ yếu để cho phép nó cộng tác với các dịch vụ khác. Chương 4 mô tả cách các sự kiện có thể được sử dụng để triển khai sagas, duy trì tính nhất quán của dữ liệu trên các dịch vụ. Và chương 7 thảo luận về cách các sự kiện có thể được sử dụng để cập nhật chế độ xem CQRS, hỗ trợ truy vấn hiệu quả. Một ứng dụng cũng có thể sử dụng các sự kiện để thông báo cho các máy khách bên ngoài. Ví dụ, nó có thể sử dụng WebSockets để phân phối các sự kiện đến trình duyệt.

Điểm khởi đầu để xác định API dịch vụ là ánh xạ từng hoạt động hệ thống với một dịch vụ. Sau đó, chúng ta quyết định xem một dịch vụ có cần cộng tác với những dịch vụ khác để triển khai hoạt động hệ thống hay không. Nếu cần cộng tác, chúng ta sẽ xác định những API mà các dịch vụ khác đó phải cung cấp để hỗ trợ cho sự cộng tác. Hãy bắt đầu bằng cách xem cách gán các hoạt động hệ thống cho các dịch vụ.

**MỘTKÝ HOẠT ĐỘNG HỆ THỐNG CHO CÁC DỊCH VỤ**

Bước đầu tiên là quyết định dịch vụ nào là điểm vào ban đầu cho một yêu cầu. Nhiều hoạt động hệ thống được ánh xạ gọn gàng tới một dịch vụ, nhưng đôi khi việc ánh xạ không rõ ràng.Ví dụ, hãy xem xét hoạt động noteUpdatedLocation(), hoạt động này cập nhật vị trí của người chuyển phát nhanh. Một mặt, vì nó liên quan đến người chuyển phát nhanh, nên hoạt động này phải được chỉ định cho dịch vụ Chuyển phát nhanh. Mặt khác, Dịch vụ giao hàng cần vị trí của người chuyển phát nhanh. Trong trường hợp này, việc chỉ định một hoạt động cho một dịch vụ cần thông tin do hoạt động cung cấp là lựa chọn tốt hơn. Trong các tình huống khác,

có thể có ý nghĩa khi chỉ định một hoạt độngcho dịch vụ có thông tin cần thiết để xử lý nó.

Bảng 2.2 hiển thị những dịch vụ nào trong ứng dụng FTGO chịu trách nhiệm cho những hoạt động nào.

**Bảng 2.2 Ánh xạ các hoạt động của hệ thống với các dịch vụ trong ứng dụng FTGO**

|  |  |
| --- | --- |
| **Dịch vụ** | **Hoạt động** |
| Dịch vụ khách hàng | tạoConsumer() |
| Dịch vụ đặt hàng | tạo đơn hàng() |
| Dịch vụ nhà hàng | tìmNhà hàng có sẵn() |
| Dịch vụ nhà bếp | * chấp nhận đơn hàng() * lưu ýOrderReadyForPickup() |
| Dịch vụ giao hàng | * noteUpdatedLocation() * lưu ýDeliveryPickedUp() * lưu ýDeliveryDelivered() |

Sau khi đã chỉ định các hoạt động cho các dịch vụ, bước tiếp theo là quyết định cách thức các dịch vụhợp tác để xử lý từng hoạt động của hệ thống.

**DXÁC ĐỊNH CÁCGiao diện lập trình ứng dụng (API)S YÊU CẦU ĐỂ HỖ TRỢ SỰ HỢP TÁC GIỮA CÁC DỊCH VỤ**

Một số hoạt động của hệ thống được xử lý hoàn toàn bởi một dịch vụ duy nhất. Ví dụ, trong ứng dụng FTGO,Dịch vụ khách hàngxử lýtạoConsumer()hoạt độnghoàn toàn tự nó. Nhưng các hoạt động hệ thống kháctrải dài trên nhiều dịch vụ. Dữ liệu cần thiết để xử lý một trong những yêu cầu này có thể, ví dụ, nằm rải rác trên nhiều dịch vụ. Ví dụ, để triển khaitạo đơn hàng()hoạt động, cácDịch vụ đặt hàngphải sử dụng các dịch vụ sau để xác minh các điều kiện tiên quyết của nó vàlàm cho các điều kiện sau trở thành đúng:

* + - * Dịch vụ khách hàng-Xác minhrằng người tiêu dùng có thể đặt hàng và nhận đượcthông tin thanh toán.
      * Dịch vụ nhà hàng—Xác thực các mục đơn hàng, xác minh rằng việc giao hàngđịa chỉ/thời gian nằm trong khu vực phục vụ của nhà hàng, xác minh số lượng đơn hàng tối thiểu đã đạt yêu cầu và lấy giá cho các mặt hàng trong đơn hàng.
      * Dịch vụ nhà bếp—Tạo raVé.
      * Dịch vụ kế toán—Ủy quyền sử dụng thẻ tín dụng của người tiêu dùng.

Tương tự như vậy, đểthực hiệnchấp nhận đơn hàng()hoạt động hệ thống,Dịch vụ nhà bếpphải triệu tậpDịch vụ giao hàngđể lên lịch cho người chuyển phát nhanh giao đơn hàng.Bảng 2.3 hiển thị các dịch vụ, API đã sửa đổi của chúng và những người cộng tác. Để xác định đầy đủ các API dịch vụ, bạn cần phân tích từng hoạt động của hệ thống và xác định sự cộng tác nào là cần thiết.

**Bảng 2.3 Các dịch vụ, API đã sửa đổi của họ và những người cộng tác của họ**

|  |  |  |
| --- | --- | --- |
| **Dịch vụ** | **Hoạt động** | **Cộng tác viên** |
| Dịch vụ khách hàng | xác minhConsumerDetails() | — |
| Dịch vụ đặt hàng | tạo đơn hàng() | * Dịch vụ khách hàng |
|  |  | xác minhConsumerDetails() |
|  |  | * Dịch vụ nhà hàng |
|  |  | xác minh chi tiết đơn hàng() |
|  |  | * Dịch vụ nhà bếp |
|  |  | tạoTicket() |
|  |  | * Dịch vụ kế toán |
|  |  | ủy quyền Thẻ() |
| Nhà hàng | * tìmNhà hàng có sẵn() | — |
| Dịch vụ | * xác minh chi tiết đơn hàng() |  |
| Dịch vụ nhà bếp | * tạoTicket() | * Dịch vụ giao hàng |
|  | * chấp nhận đơn hàng() | lịch trìnhDelivery() |
|  | * lưu ýOrderReadyForPickup() |  |
| Dịch vụ giao hàng | * lịch trìnhDelivery() | — |
|  | * noteUpdatedLocation() |  |
|  | * lưu ýDeliveryPickedUp() |  |
|  | * lưu ýDeliveryDelivered() |  |
| Kế toán | * ủy quyền Thẻ() | — |
| Dịch vụ |  |  |

Cho đến nay, chúng ta đã xác định được các dịch vụ và hoạt động mà mỗi dịch vụ triển khai. Nhưng điều quan trọng cần nhớ là kiến ​​trúc mà chúng ta phác thảo ra rất trừu tượng. Chúng ta không chọn bất kỳ công nghệ IPC cụ thể nào. Hơn nữa, mặc dù thuật ngữ hoạt động gợi ý một số loại cơ chế IPC dựa trên yêu cầu/phản hồi đồng bộ, bạn sẽ thấy rằng nhắn tin không đồng bộ đóng một vai trò quan trọng. Trong suốt cuốn sách này, tôi mô tả các khái niệm về kiến ​​trúc và thiết kế ảnh hưởng đến cách các dịch vụ này cộng tác.

Chương 3 mô tả các công nghệ IPC cụ thể, bao gồm các cơ chế giao tiếp đồng bộ như REST và nhắn tin không đồng bộ bằng cách sử dụng một môi giới tin nhắn. Tôi thảo luận về cách giao tiếp đồng bộ có thể tác động đến tính khả dụng và giới thiệu khái niệm về dịch vụ độc lập, không gọi các dịch vụ khác một cách đồng bộ. Một cách để triển khai dịch vụ độc lập là sử dụng mẫu CQRS, được đề cập trong chương 7.Dịch vụ đặt hàngcó thể, ví dụ, duy trì một bản sao của dữ liệu sở hữubởiDịch vụ nhà hàngđể loại bỏ nhu cầu của nó đồng bộ gọiDịch vụ nhà hàngđể xác thực một đơn hàng. Nó giữ cho bản sao được cập nhật bằng cách đăng ký các sự kiện đã xuất bản qua cái Dịch vụ nhà hàngbất cứ khi nào Nó cập nhậtdữ liệu của nó.

Chương 4 giới thiệu khái niệm saga và cách sử dụng tin nhắn không đồng bộ để phối hợp các dịch vụ tham gia vào saga. Cũng như cập nhật đáng tin cậy

dữ liệu phân tán trên nhiều dịch vụ, saga cũng là một cách để triển khai một dịch vụ độc lập. Ví dụ, tôi mô tả cáchtạo đơn hàng()hoạt độngđược thực hiện bằng cách sử dụng một saga, trong đó gọi các dịch vụ nhưDịch vụ khách hàng,Dịch vụ nhà bếp,VàDịch vụ kế toánsử dụng tin nhắn không đồng bộ.

Chương 8 mô tả khái niệm về cổng API, nơi đưa API ra cho các máy khách bên ngoài. Cổng API có thể triển khai hoạt động truy vấn bằng cách sử dụng mẫu thành phần API, được mô tả trong chương 7, thay vì chỉ định tuyến đến dịch vụ. Logic trong cổng API thu thập dữ liệu cần thiết cho truy vấn bằng cách gọi nhiều dịch vụ và kết hợp các kết quả. Trong tình huống này, hoạt động hệ thống được gán cho cổng API thay vìdịch vụ. Các dịch vụ cần triển khai các hoạt động truy vấn cần thiết của cổng API.

#### Bản tóm tắt

* + - * Kiến trúc xác định khả năng của ứng dụng, bao gồm khả năng bảo trì, khả năng kiểm tra và khả năng triển khai, ảnh hưởng trực tiếp đến tốc độ phát triển.
      * Kiến trúc vi dịch vụ là một kiểu kiến ​​trúc mang lại cho ứng dụng khả năng bảo trì, khả năng kiểm tra và khả năng triển khai cao.
      * Các dịch vụ trong kiến ​​trúc vi dịch vụ được tổ chức xung quanh các mối quan tâm kinh doanh—năng lực kinh doanh hoặc miền phụ—thay vì các mối quan tâm kỹ thuật.
      * Có hai mô hình phân hủy:
        + Phân tích theo năng lực kinh doanh, có nguồn gốc từ kiến ​​trúc kinh doanh
        + Phân tích theo miền phụ, dựa trên các khái niệm từ thiết kế theo miền
      * Bạn có thể loại bỏ các lớp god, gây ra sự phụ thuộc phức tạp ngăn cản việc phân tách, bằng cách áp dụng DDD và xác định một mô hình miền riêng cho mỗi dịch vụ.

*Liên tiến trìnhgiao tiếp trong*

*một kiến ​​trúc dịch vụ vi mô*

***Chương này bao gồm***

* Áp dụng các mẫu giao tiếp: Gọi thủ tục từ xa, Bộ ngắt mạch, Khám phá phía máy khách, Tự đăng ký, Khám phá phía máy chủ, Đăng ký của bên thứ ba, Nhắn tin không đồng bộ, Hộp thư đi giao dịch, Theo dõi nhật ký giao dịch,Nhà xuất bản thăm dò ý kiến
* Tầm quan trọng của giao tiếp giữa các tiến trình trongmột kiến ​​trúc dịch vụ vi mô
* Định nghĩa và phát triểnAPI
* Các tùy chọn giao tiếp liên tiến trình khác nhauvà sự đánh đổi của họ
* Lợi ích của các dịch vụ giao tiếp bằng cách sử dụngnhắn tin không đồng bộ
* Gửi tin nhắn đáng tin cậy như một phần của giao dịch cơ sở dữ liệu

Mary và nhóm của cô ấy, giống như hầu hết các nhà phát triển khác, có một số kinh nghiệm với các cơ chế giao tiếp giữa các tiến trình (IPC). Ứng dụng FTGO có một REST API được sử dụng bởi các ứng dụng di động và JavaScript phía trình duyệt. Nó cũng sử dụng nhiều

**65**

dịch vụ đám mây, chẳng hạn như dịch vụ nhắn tin Twilio và dịch vụ thanh toán Stripe. Nhưng trong một ứng dụng đơn khối như FTGO, các mô-đun gọi nhau thông qua phương thức hoặc hàm cấp độ ngôn ngữ. Các nhà phát triển FTGO thường không cần phải nghĩ đến IPC trừ khi họ đang làm việc trên REST API hoặc các mô-đun tích hợp với các dịch vụ đám mây.

Ngược lại, như bạn đã thấy trong chương 2, kiến ​​trúc vi dịch vụ cấu trúc một ứng dụng như một tập hợp các dịch vụ. Các dịch vụ đó thường phải cộng tác để xử lý một yêu cầu. Vì các phiên bản dịch vụ thường là các quy trình chạy trên nhiều máy, nên chúng phải tương tác bằng IPC. Nó đóng vai trò quan trọng hơn nhiều trong kiến ​​trúc vi dịch vụ so với trong ứng dụng đơn khối. Do đó, khi họ di chuyển ứng dụng của mình sang vi dịch vụ, Mary và những nhà phát triển FTGO còn lại sẽ cần dành nhiều thời gian hơn để suy nghĩ về IPC.

Không thiếu các cơ chế IPC để lựa chọn. Ngày nay, lựa chọn hợp thời là REST (với JSON). Tuy nhiên, điều quan trọng là phải nhớ rằng không có giải pháp nào là tuyệt đối. Bạn phải cân nhắc kỹ lưỡng các lựa chọn. Chương này khám phá nhiều tùy chọn IPC khác nhau, bao gồm REST và nhắn tin, và thảo luận về các sự đánh đổi.

Việc lựa chọn cơ chế IPC là một quyết định quan trọng về mặt kiến ​​trúc. Nó có thể tác động đến tính khả dụng của ứng dụng. Hơn nữa, như tôi đã giải thích trong chương này và chương tiếp theo, IPC thậm chí còn giao thoa với quản lý giao dịch. Tôi ủng hộ một kiến ​​trúc bao gồm các dịch vụ được ghép nối lỏng lẻo giao tiếp với nhau bằng cách sử dụng tin nhắn không đồng bộ. Các giao thức đồng bộ như REST chủ yếu được sử dụng để giao tiếp với các ứng dụng khác.

Tôi bắt đầu chương này bằng phần tổng quan về giao tiếp giữa các tiến trình trong kiến ​​trúc vi dịch vụ. Tiếp theo, tôi mô tả IPC dựa trên lệnh gọi thủ tục từ xa, trong đó REST là ví dụ phổ biến nhất. Tôi đề cập đến các chủ đề quan trọng bao gồm khám phá dịch vụ và cách xử lý lỗi một phần. Sau đó, tôi mô tả IPC dựa trên nhắn tin không đồng bộ. Tôi cũng nói về việc mở rộng quy mô người tiêu dùng trong khi vẫn giữ nguyên thứ tự tin nhắn, xử lý đúng các tin nhắn trùng lặp và nhắn tin giao dịch. Cuối cùng, tôi trình bày khái niệm về các dịch vụ độc lập xử lý các yêu cầu đồng bộ mà không cần giao tiếp với các dịch vụ khác để cải thiện tính khả dụng.

#### Tổng quan về giao tiếp giữa các tiến trình trong mộtKiến trúc dịch vụ vi mô

Có rất nhiều công nghệ IPC khác nhau để lựa chọn. Các dịch vụ có thể sử dụng cơ chế giao tiếp dựa trên yêu cầu/phản hồi đồng bộ, chẳng hạn như REST dựa trên HTTP hoặc gRPC. Ngoài ra, chúng có thể sử dụng cơ chế giao tiếp dựa trên tin nhắn không đồng bộ như AMQP hoặc STOMP. Ngoài ra còn có nhiều định dạng tin nhắn khác nhau. Các dịch vụ có thể sử dụng các định dạng dựa trên văn bản, dễ đọc như JSON hoặc XML. Ngoài ra, chúng có thể sử dụng định dạng nhị phân hiệu quả hơn như Avro hoặc Protocol Buffers.

Trước khi đi vào chi tiết về các công nghệ cụ thể, tôi muốn nêu ra một số vấn đề thiết kế mà bạn nên cân nhắc. Tôi bắt đầu phần này bằng một cuộc thảo luận về tương tác

styles, là một cách độc lập với công nghệ để mô tả cách khách hàng và dịch vụ tương tác. Tiếp theo, tôi thảo luận về tầm quan trọng của việc xác định chính xác API trong kiến ​​trúc dịch vụ vi mô, bao gồm khái niệm thiết kế API-first. Sau đó, tôi thảo luận về chủ đề quan trọng về sự tiến hóa của API. Cuối cùng, tôi thảo luận về các tùy chọn khác nhau cho định dạng tin nhắn và cách chúng có thể xác định mức độ dễ dàng tiến hóa của API. Hãy bắt đầu bằng cách xem xét các phong cách tương tác.

###### Phong cách tương tác

Sẽ rất hữu ích nếu trước tiên bạn nghĩ về phong cách tương tác giữa một dịch vụ và các máy khách của nó trước khi chọn cơ chế IPC cho API của dịch vụ. Việc suy nghĩ trước về phong cách tương tác sẽ giúp bạn tập trung vào các yêu cầu và tránh sa lầy vào các chi tiết của một công nghệ IPC cụ thể. Ngoài ra, như đã mô tả trong phần 3.4, lựa chọn phong cách tương tác sẽ ảnh hưởng đến tính khả dụng của ứng dụng của bạn. Hơn nữa, như bạn sẽ thấy trong chương 9 và 10, nó giúp bạn chọn chiến lược kiểm thử tích hợp phù hợp.

Có nhiều loại tương tác dịch vụ khách hàngphong cách. Như bảng 3.1 cho thấy, chúng có thể được phân loại thành hai chiều. Chiều thứ nhất là tương tác là một-một hay một-nhiều:

* + - * *Một-một*—Mỗi yêu cầu của khách hàng chỉ được xử lý bởi một dịch vụ.
      * *Một-nhiều*—Mỗi yêu cầu được xử lý bởi nhiều dịch vụ.

Chiều thứ hai là liệu tương tác có đồng bộ hay không đồng bộ:

* + - * *Đồng bộ*—Khách hàng mong đợi phản hồi kịp thời từ dịch vụ và thậm chí có thể chặn trong khi chờ đợi.
      * *Không đồng bộ*—Máy khách không chặn và phản hồi, nếu có, không nhất thiết phải được gửi ngay lập tức.

**Bảng 3.1 Các phong cách tương tác khác nhau có thể được mô tả theo hai chiều: một-một so với một-đối-mộtnhiều và đồng bộ so với không đồng bộ.**

|  |  |  |
| --- | --- | --- |
|  | **một-một** | **một-nhiều** |
| Đồng bộ Không đồng bộ | Yêu cầu/phản hồi  Yêu cầu/phản hồi không đồng bộ Thông báo một chiều | —  Xuất bản/đăng ký Xuất bản/phản hồi không đồng bộ |

Sau đây là những khác biệtcác loại tương tác một-một:

* + - * *Yêu cầu/phản hồi*—Một máy khách dịch vụ đưa ra yêu cầu đến một dịch vụ và chờ phản hồi. Máy khách mong đợi phản hồi sẽ đến kịp thời. Nó có thể bị chặn trong khi chờ đợi. Đây là một kiểu tương tác thường dẫn đến các dịch vụ được kết nối chặt chẽ.
      * *Yêu cầu/phản hồi không đồng bộ*—Một máy khách dịch vụ gửi yêu cầu đến một dịch vụ, dịch vụ này trả lời không đồng bộ. Máy khách không chặn trong khi chờ đợi, vì dịch vụ có thể không gửi phản hồi trong một thời gian dài.
* *Thông báo một chiều*—Một máy khách dịch vụ gửi yêu cầu đến một dịch vụ nhưng không mong đợi hoặc không nhận được phản hồi.

Điều quan trọng cần nhớ là kiểu tương tác yêu cầu/phản hồi đồng bộ chủ yếu là trực giao với các công nghệ IPC. Ví dụ, một dịch vụ có thể tương tác với một dịch vụ khác bằng cách sử dụng tương tác kiểu yêu cầu/phản hồi với REST hoặc tin nhắn. Ngay cả khi hai dịch vụ đang giao tiếp bằng một nhà môi giới tin nhắn, dịch vụ khách hàng có thể bị chặn khi chờ phản hồi. Điều đó không nhất thiết có nghĩa là chúng được ghép nối lỏng lẻo. Đó là điều tôi sẽ xem xét lại sau trong chương này khi thảo luận về tác động của giao tiếp giữa các dịch vụ đối với tính khả dụng.

Sau đây là các loại tương tác một-nhiều khác nhau:

* *Xuất bản/đăng ký*—Một máy khách gửi một tin nhắn thông báo, tin nhắn này sẽ được sử dụng bởi không hoặc nhiều dịch vụ quan tâm.
* *Xuất bản/phản hồi không đồng bộ*—Khách hàng gửi tin nhắn yêu cầu và chờ trong một khoảng thời gian nhất định để nhận được phản hồi từ các dịch vụ quan tâm.

Mỗi dịch vụ thường sẽ sử dụng kết hợp các kiểu tương tác này. Nhiều dịch vụ trong ứng dụng FTGO có cả API đồng bộ và không đồng bộ cho các hoạt động và nhiều dịch vụ cũng xuất bản sự kiện.

Hãy cùng xem cách xác định API của dịch vụ.

###### Định nghĩa API trong kiến ​​trúc dịch vụ vi mô

API hoặc giao diện là trung tâm của phát triển phần mềm. Một ứng dụng bao gồmcủa các mô-đun. Mỗi mô-đun có một giao diện xác định tập hợp các hoạt động mà các máy khách của mô-đun có thể gọi. Một giao diện được thiết kế tốt sẽ phơi bày chức năng hữu ích trong khi ẩn việc triển khai. Nó cho phép việc triển khai thay đổi mà không ảnh hưởng đến máy khách.

Trong một ứng dụng đơn khối, giao diện thường được chỉ định bằng cách sử dụng cấu trúc ngôn ngữ lập trình như giao diện Java. Giao diện Java chỉ định một tập hợp các phương thức mà máy khách có thể gọi. Lớp triển khai được ẩn khỏi máy khách. Hơn nữa, vì Java là ngôn ngữ được gõ tĩnh, nếu giao diện thay đổi thành không tương thích với máy khách, ứng dụng sẽ không biên dịch được.

API và giao diện đều quan trọng như nhau trong kiến ​​trúc microservice. API của dịch vụ là hợp đồng giữa dịch vụ và máy khách của dịch vụ đó. Như đã mô tả trong chương 2, API của dịch vụ bao gồm các hoạt động mà máy khách có thể gọi và các sự kiện do dịch vụ công bố. Một hoạt động có tên, tham số và kiểu trả về. Một sự kiện có kiểu và một tập hợp các trường và được công bố tới kênh tin nhắn như đã mô tả trong phần 3.3.

Thách thức là API dịch vụ không được định nghĩa bằng cách sử dụng một cấu trúc ngôn ngữ lập trình đơn giản. Theo định nghĩa, dịch vụ và các máy khách của nó không được biên dịch cùng nhau. Nếu một phiên bản mới của dịch vụ được triển khai với một API không tương thích, sẽ không có lỗi biên dịch. Thay vào đó, sẽ có lỗi thời gian chạy.

Bất kể bạn chọn cơ chế IPC nào, điều quan trọng là phải xác định chính xác API của dịch vụ bằng một số loại ngôn ngữ định nghĩa giao diện (IDL). Hơn nữa, có những lý lẽ chính đáng để sử dụng phương pháp tiếp cận API-first để xác định dịch vụ (xem[www](http://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10)

[.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/](http://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10) [2015/07/10](http://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10)để biết thêm). Trước tiên, bạn viết định nghĩa giao diện. Sau đó, bạn xem xét định nghĩa giao diện với các nhà phát triển khách hàng. Chỉ sau khi lặp lại định nghĩa API, bạn mới triển khai dịch vụ. Thực hiện thiết kế trước này sẽ tăng cơ hội xây dựng dịch vụ đáp ứng nhu cầu của khách hàng.

**Thiết kế API-first là điều cần thiết**

Ngay cả trong các dự án nhỏ, tôi đã thấy các vấn đề xảy ra vì các thành phần không thống nhất về API. Ví dụ, trong một dự án, nhà phát triển Java backend và nhà phát triển frontend AngularJS đều nói rằng họ đã hoàn thành quá trình phát triển. Tuy nhiên, ứng dụng không hoạt động. REST và WebSocket API mà ứng dụng frontend sử dụng để giao tiếp với backend được định nghĩa kém. Kết quả là, hai ứng dụng không thể giao tiếp!

Bản chất của định nghĩa API phụ thuộc vào cơ chế IPC mà bạn đang sử dụng. Ví dụ, nếu bạn đang sử dụng tin nhắn, API bao gồm các kênh tin nhắn, các loại tin nhắn và các định dạng tin nhắn. Nếu bạn đang sử dụng HTTP, API bao gồm các URL, các động từ HTTP và các định dạng yêu cầu và phản hồi. Sau trong chương này, tôi sẽ giải thích cách định nghĩa API.

API của một dịch vụ hiếm khi được thiết lập cố định. Nó có thể sẽ phát triển theo thời gian. Hãy cùng xem cách thực hiện điều đó và xem xét các vấn đề bạn sẽ gặp phải.

###### API đang phát triển

API luôn thay đổi theo thời gian khi các tính năng mới được thêm vào, các tính năng hiện có được thay đổi và (có thể) các tính năng cũ bị xóa. Trong một ứng dụng đơn khối, việc thay đổi API và cập nhật tất cả các trình gọi là tương đối đơn giản. Nếu bạn đang sử dụng ngôn ngữ được gõ tĩnh, trình biên dịch sẽ giúp bằng cách đưa ra danh sách các lỗi biên dịch. Thách thức duy nhất có thể là phạm vi thay đổi. Có thể mất nhiều thời gian để thay đổi một API được sử dụng rộng rãi.

Trong một ứng dụng dựa trên dịch vụ vi mô, việc thay đổi API của dịch vụ khó hơn nhiều. Các máy khách của dịch vụ là các dịch vụ khác, thường được phát triển bởi các nhóm khác. Các máy khách thậm chí có thể là các ứng dụng khác bên ngoài tổ chức. Bạn thường không thể buộc tất cả các máy khách nâng cấp cùng lúc với dịch vụ. Ngoài ra, vì các ứng dụng hiện đại thường không bao giờ ngừng hoạt động để bảo trì, nên bạn thường sẽ thực hiện nâng cấp liên tục dịch vụ của mình, do đó cả phiên bản cũ và mới của dịch vụ sẽ chạy đồng thời.

Điều quan trọng là phải có một chiến lược để giải quyết những vấn đề nàythách thức. Cách bạn xử lý thay đổi đối với API phụ thuộc vào bản chất của thay đổi.

**BạnPHIÊN BẢN SE NGỮ NGHĨA**

Đặc tả phiên bản ngữ nghĩa ([http://semver.org](http://semver.org/)) là hướng dẫn hữu ích về API phiên bản. Đây là một tập hợp các quy tắc chỉ định cách sử dụng và tăng số phiên bản. Phiên bản ngữ nghĩa ban đầu được dùng để phiên bản hóa các gói phần mềm, nhưng bạn có thể sử dụng nó để phiên bản hóa API trong hệ thống phân tán.

Đặc tả Phiên bản ngữ nghĩa (Semvers) yêu cầu số phiên bản phải bao gồm ba phần: MAJOR.MINOR.PATCH. Bạn phải tăng từng phần của số phiên bản như sau:

* LỚN LAO—Khi bạn thực hiện một thay đổi không tương thích với API
* NGƯỜI VỊ THÀNH NIÊN—Khi bạn thực hiện các cải tiến tương thích ngược với API
* VÁ—Khi bạn thực hiện sửa lỗi tương thích ngược

Có một vài nơi bạn có thể sử dụng số phiên bản trong API. Nếu bạn đang triển khai REST API, bạn có thể, như đã đề cập bên dưới, sử dụng phiên bản chính làm phần tử đầu tiên của đường dẫn URL. Ngoài ra, nếu bạn đang triển khai dịch vụ sử dụng tin nhắn, bạn có thể bao gồm số phiên bản trong các tin nhắn mà dịch vụ đó xuất bản. Mục tiêu là phiên bản API đúng cách và phát triển chúng theo cách được kiểm soát. Hãy cùng xem cách xử lý các thay đổi nhỏ và lớn.

**TôiĐẶT HÀNG NHỎ,LÙI LẠI-THAY ĐỔI TƯƠNG THÍCH**

Lý tưởng nhất là bạn nên cố gắng chỉ thực hiện những thay đổi tương thích ngược. Những thay đổi tương thích ngược là những thay đổi bổ sung vào API:

* Thêm các thuộc tính tùy chọn vào yêu cầu
* Thêm thuộc tính vào phản hồi
* Thêmhoạt động mới

Nếu bạn chỉ thực hiện những thay đổi như thế này, các khách hàng cũ sẽ làm việc với các dịch vụ mới hơn, với điều kiện là họ tuân thủ nguyên tắc Mạnh mẽ ([https://en.wikipedia.org/wiki/](https://en.wikipedia.org/wiki/Robustness_principle) [Nguyên tắc\_độ\_mạnh\_mẽ](https://en.wikipedia.org/wiki/Robustness_principle)), trong đó nêu: “Hãy thận trọng trong những gì bạn làm, hãy rộng lượng trong những gì bạn chấp nhận từ người khác”. Các dịch vụ nên cung cấp các giá trị mặc định cho các thuộc tính yêu cầu bị thiếu. Tương tự như vậy, các máy khách nên bỏ qua bất kỳ thuộc tính phản hồi bổ sung nào. Để điều này không gây đau đớn, các máy khách và dịch vụ phải sử dụng định dạng yêu cầu và phản hồi hỗ trợ nguyên tắc Robustness. Sau này trong phần này, tôi sẽ mô tả cách các định dạng dựa trên văn bản như JSON và XML thường giúp phát triển API dễ dàng hơn.

**TôiĐẠI HỌC,ĐANG PHÁ VỠTHAY ĐỔI**

Đôi khi bạn phải thực hiện những thay đổi lớn, không tương thích với API. Vì bạn không thể buộc khách hàng nâng cấp ngay lập tức, nên một dịch vụ phải đồng thời hỗ trợ phiên bản cũ và mới của API trong một khoảng thời gian. Nếu bạn đang sử dụng cơ chế IPC dựa trên HTTP, chẳng hạn như REST, một cách tiếp cận là nhúng số phiên bản chính vào URL. Ví dụ: đường dẫn phiên bản 1 được thêm tiền tố '/v1/…' và đường dẫn phiên bản 2 được thêm tiền tố '/v2/…'.

Một lựa chọn khác là sử dụng cơ chế đàm phán nội dung của HTTP và bao gồm số phiên bản trong loại MIME. Ví dụ, một máy khách sẽ yêu cầu phiên bản 1.x củamột Lệnh sử dụng yêu cầu như thế này:

NHẬN /orders/xyz HTTP/1.1

Chấp nhận: application/vnd.example.resource+json; phiên bản=1

...

Yêu cầu này cho biếtDịch vụ đặt hàngrằng khách hàng mong đợi một phiên bản1.xphản ứng.

Để hỗ trợ nhiều phiên bản của một API, các bộ điều hợp của dịch vụ triển khai các API sẽ chứa logic chuyển đổi giữa các phiên bản cũ và mới. Ngoài ra, như đã mô tả trong chương 8, cổng API gần như chắc chắn sẽ sử dụng các API có phiên bản. Nó thậm chí có thể phải hỗ trợ nhiều phiên bản cũ hơn của một API.

Bây giờ chúng ta sẽ xem xét vấn đề định dạng tin nhắn, việc lựa chọn định dạng này có thể ảnh hưởng đến mức độ dễ dàng phát triển API.

###### Định dạng tin nhắn

Bản chất của IPC là trao đổi tin nhắn. Tin nhắn thường chứa dữ liệu, do đó, một quyết định thiết kế quan trọng là định dạng của dữ liệu đó. Việc lựa chọn định dạng tin nhắn có thể ảnh hưởng đến hiệu quả của IPC, khả năng sử dụng API và khả năng phát triển của nó. Nếu bạn đang sử dụng hệ thống nhắn tin hoặc giao thức như HTTP, bạn sẽ được chọn định dạng tin nhắn của mình. Một số cơ chế IPC—chẳng hạn như gRPC, mà bạn sẽ tìm hiểu ngay sau đây—có thể quyết định định dạng tin nhắn. Trong cả hai trường hợp, điều cần thiết là phải sử dụng định dạng tin nhắn đa ngôn ngữ. Ngay cả khi bạn đang viết các dịch vụ siêu nhỏ của mình bằng một ngôn ngữ duy nhất hiện nay, thì rất có thể bạn sẽ sử dụng các ngôn ngữ khác trong tương lai. Ví dụ, bạn không nên sử dụng tuần tự hóa Java.

Có hai loại định dạng tin nhắn chính: văn bản và nhị phân. Chúng ta hãy xem xét từng loại.

**TPHỤ LỤC-DỰA TRÊNĐỊNH DẠNG TIN NHẮN**

Thể loại đầu tiên là các định dạng dựa trên văn bản như JSON và XML. Một lợi thế của các định dạng này là chúng không chỉ dễ đọc đối với con người mà còn tự mô tả. Một thông điệp JSON là một tập hợp các thuộc tính được đặt tên. Tương tự như vậy, một thông điệp XML về cơ bản là một tập hợp các phần tử và giá trị được đặt tên. Định dạng này cho phép người dùng thông điệp chọn ra các giá trị quan tâm và bỏ qua phần còn lại. Do đó, nhiều thay đổi đối với lược đồ thông điệp có thể dễ dàng tương thích ngược.

Cấu trúc của tài liệu XML được chỉ định bởi một lược đồ XML ([www.w3.org/](http://www.w3.org/XML/Schema) [XML/Sơ đồ](http://www.w3.org/XML/Schema)). Theo thời gian, cộng đồng nhà phát triển đã nhận ra rằng JSON cũng cần một cơ chế tương tự. Một lựa chọn phổ biến là sử dụng tiêu chuẩn JSON Schema ([http://json-schema.org](http://json-schema.org/)). Một lược đồ JSON xác định tên và loại thuộc tính của tin nhắn và liệu chúng là tùy chọn hay bắt buộc. Cũng như là tài liệu hữu ích, một lược đồ JSON có thể được một ứng dụng sử dụng để xác thực các tin nhắn đến.

Một nhược điểm của việc sử dụng định dạng tin nhắn dạng văn bản là các tin nhắn có xu hướng dài dòng, đặc biệt là XML. Mỗi tin nhắn đều có chi phí chứa tên của

các thuộc tính ngoài các giá trị của chúng. Một nhược điểm khác là chi phí phân tích cú pháp văn bản, đặc biệt là khi các thông điệp có kích thước lớn. Do đó, nếu hiệu quả và hiệu suất là quan trọng, bạn có thể cân nhắc sử dụng định dạng nhị phân.

**BĐỊNH DẠNG TIN NHẮN INARY**

Có một số định dạng nhị phân khác nhau để lựa chọn. Các định dạng phổ biến bao gồm Protocol Buffers (<https://developers.google.com/protocol-buffers/docs/overview>) và Avro ([https://avro.apache.org](https://avro.apache.org/)). Cả hai định dạng đều cung cấp IDL được gõ để xác định cấu trúc tin nhắn của bạn. Sau đó, trình biên dịch sẽ tạo mã tuần tự hóa và hủy tuần tự hóa tin nhắn. Bạn buộc phải áp dụng phương pháp tiếp cận API-first để thiết kế dịch vụ! Hơn nữa, nếu bạn viết ứng dụng khách của mình bằng ngôn ngữ được gõ tĩnh, trình biên dịch sẽ kiểm tra xem ứng dụng có sử dụng API đúng không.

Một điểm khác biệt giữa hai định dạng nhị phân này là Protocol Buffers sử dụng các trường được gắn thẻ, trong khi người dùng Avro cần biết lược đồ để diễn giải các thông báo. Do đó, việc xử lý sự tiến hóa của API dễ dàng hơn với Protocol Buffers so với Avro. Bài đăng trên blog này ([http://martin.kleppmann.com/2012/12/05/schema-](http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html) [tiến hóa-trong-giao-thức-avro-bộ-đệm-thrift.html](http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html)) là sự so sánh tuyệt vời giữa Thrift, Protocol Buffers và Avro.

Bây giờ chúng ta đã xem xét các định dạng tin nhắn, hãy cùng xem xét các cơ chế IPC cụ thể truyền tải tin nhắn, bắt đầu với mô hình Gọi thủ tục từ xa (RPI).

#### Giao tiếp bằng Remote đồng bộmẫu gọi thủ tục

Khi sử dụng cơ chế IPC dựa trên lệnh gọi thủ tục từ xa, máy khách gửi yêu cầu đến dịch vụ và dịch vụ xử lý yêu cầu và gửi lại phản hồi. Một số máy khách có thể chặn chờ phản hồi và một số máy khách khác có thể có kiến ​​trúc phản ứng, không chặn. Nhưng không giống như khi sử dụng tin nhắn, máy khách cho rằng phản hồi sẽ đến kịp thời.

Hình 3.1 cho thấy cách RPI hoạt động. Logic nghiệp vụ trong máy khách gọi một giao diện proxy, được triển khai bởi lớp bộ điều hợp proxy RPI. Proxy RPI gửi yêu cầu đến dịch vụ. Yêu cầu được xử lý bởi lớp bộ điều hợp máy chủ RPI, lớp này gọi logic nghiệp vụ của dịch vụ thông qua một giao diện. Sau đó, lớp này gửi lại phản hồi cho proxy RPI, lớp này trả về kết quả cho logic nghiệp vụ của máy khách.

**Mẫu: Gọi thủ tục từ xa**

Một máy khách gọi một dịch vụ bằng cách sử dụng giao thức dựa trên lệnh gọi thủ tục từ xa đồng bộ, chẳng hạn như REST ([http://microservices.io/patterns/communication-style/](http://microservices.io/patterns/communication-style/messaging.html) [tin nhắn.html](http://microservices.io/patterns/communication-style/messaging.html)).

Giao diện proxy thường đóng gói giao thức truyền thông cơ bản. Có rất nhiều giao thức để lựa chọn. Trong phần này, tôi mô tả REST và

Dịch vụ khách hàng



Logic kinh doanh

Logic kinh doanh

gọi

Lời yêu cầu

Hồi đáp

RPI

máy chủ

RPI

người đại diện

**Người ủy nhiệmgiao diệnDịch vụ giao diện**

**Hình 3.1 Logic kinh doanh của khách hàng gọi một giao diện được triển khai bởi*Máy chủ RPI*lớp bộ điều hợp.*Lớp proxy RPI*đưa ra yêu cầu cho dịch vụ.*Máy chủ RPI*lớp bộ chuyển đổixử lý yêu cầu bằng cách gọi logic kinh doanh của dịch vụ.**

gRPC. Tôi sẽ trình bày cách cải thiện tính khả dụng của các dịch vụ của bạn bằng cách xử lý đúng lỗi một phần và giải thích lý do tại sao ứng dụng dựa trên dịch vụ vi mô sử dụng RPI phải sử dụng cơ chế khám phá dịch vụ.

Trước tiên chúng ta hãy cùng xem xét REST.

###### Sử dụng REST

Ngày nay, việc phát triển API theo phong cách RESTful đang trở nên thời thượng ([https://vi.wikipedia](https://en.wikipedia.org/wiki/Representational_state_transfer)

[.org/wiki/Chuyển\_chuyển ...](https://en.wikipedia.org/wiki/Representational_state_transfer)). REST là một cơ chế IPC (gần như luôn luôn) sử dụng HTTP. Roy Fielding, người tạo ra REST, định nghĩa REST như sau:

*REST cung cấp một tập hợp các ràng buộc về kiến ​​trúc, khi được áp dụng như mộttoàn bộ, nhấn mạnh khả năng mở rộng của các tương tác thành phần, tính tổng quát của các giao diện, triển khai độc lập các thành phần và các thành phần trung gian để giảm độ trễ tương tác, thực thi bảo mật,và đóng gói các hệ thống cũ.*

[www.ics.uci.edu/~fielding/pubs/dissertation/top.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)

Một khái niệm chính trong REST là một tài nguyên, thường biểu diễn một đối tượng kinh doanh duy nhất, chẳng hạn như Khách hàng hoặc Sản phẩm, hoặc một tập hợp các đối tượng kinh doanh. REST sử dụng các động từ HTTP để thao tác các tài nguyên, được tham chiếu bằng URL. Ví dụ, một yêu cầu GET trả về biểu diễn của một tài nguyên, thường ở dạng tài liệu XML hoặc đối tượng JSON, mặc dù các định dạng khác như nhị phân có thể được sử dụng. Một yêu cầu POST tạo một tài nguyên mới và một yêu cầu PUTyêu cầu cập nhật một tài nguyên. Ví dụ, Dịch vụ đơn hàng có điểm cuối POST /orders để tạo Đơn hàng và điểm cuối GET /orders/{orderId} để truy xuất Đơn hàng.

Nhiều nhà phát triển tuyên bố của họAPI dựa trên HTTP là RESTful. Nhưng như Roy Fielding mô tả trong bài đăng trên blog, không phải tất cả chúng đều như vậy ([http://roy.gbiv.com/untangled/](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven) [2008/rest-apis-must-be-hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)). Để hiểu lý do, chúng ta hãy xem xét mô hình trưởng thành REST.

**TANH TANGHỈ NGƠIMÔ HÌNH TRƯỞNG THÀNH**

Leonard Richardson (không liên quan đến tác giả của bạn) định nghĩa một mô hình trưởng thành rất hữu ích cho REST (<http://martinfowler.com/articles/richardsonMaturityModel.html>)bao gồm các cấp độ sau:

* *Cấp độ 0*—Khách hàng củadịch vụ cấp độ 0 gọi dịch vụ bằng cách thực hiện các yêu cầu HTTP POST đến điểm cuối URL duy nhất của nó. Mỗi yêu cầu chỉ định hành động cần thực hiện, mục tiêu của hành động (ví dụ: đối tượng kinh doanh) và bất kỳ tham số nào.
* *Cấp độ 1*—Dịch vụ cấp độ 1 hỗ trợ ý tưởng về tài nguyên. Để thực hiện hành động trên tài nguyên, máy khách thực hiện yêu cầu POST chỉ định hành động cần thực hiện và bất kỳ tham số nào.
* *Cấp độ 2*—Dịch vụ cấp độ 2 sử dụng động từ HTTP để thực hiện các hành động: GET để truy xuất, POST để tạo và PUT để cập nhật. Các tham số truy vấn yêu cầu và nội dung, nếu có, sẽ chỉ định các tham số của hành động. Điều này cho phép các dịch vụ sử dụng cơ sở hạ tầng web như lưu trữ đệm cho các yêu cầu GET.
* *Cấp độ 3*—Thiết kế của dịch vụ cấp độ 3 dựa trên nguyên tắc HATEOAS (Hypertext As The Engine Of Application State) có tên gọi rất khủng khiếp. Ý tưởng cơ bản là biểu diễn của một tài nguyên được trả về bởi yêu cầu GET chứa các liên kết để thực hiện các hành động trên tài nguyên đó. Ví dụ, khách hàng có thể hủy đơn hàng bằng cách sử dụng liên kết trong biểu diễn được trả về bởi yêu cầu GET đã truy xuất đơn hàng. Các lợi ích của HATEOAS bao gồm không còn phải kết nối cứng các URL vào mã máy khách ([www.infoq.com/news/2009/04/](http://www.infoq.com/news/2009/04/hateoas-restful-api-advantages) [hateoas-restful-api-lợi thế](http://www.infoq.com/news/2009/04/hateoas-restful-api-advantages)).

Tôi khuyến khích bạn xem xét các API REST tại tổ chức của mình để xem chúng tương ứng với cấp độ nào.

**SCHÍNH XÁCGiao diện lập trình ứng dụng RESTS**

Như đã đề cập trước đó trong phần 3.1, bạn phải xác định API của mình bằng giao diệnngôn ngữ định nghĩa (IDL). Không giống như các giao thức truyền thông cũ hơn như CORBA và SOAP, REST ban đầu không có IDL. May mắn thay, cộng đồng nhà phát triển đã khám phá lại giá trị của IDL đối với API RESTful. IDL REST phổ biến nhất là Đặc tả API mở ([www.openapis.org](http://www.openapis.org/)), phát triển từ dự án nguồn mở Swagger. Dự án Swagger là một bộ công cụ để phát triển và lập tài liệu cho REST API. Nó bao gồm các công cụ tạo ra các stub máy khách và bộ khung máy chủ từ định nghĩa giao diện.

**TTHÁCH THỨC LẤY NHIỀU TÀI NGUYÊN TRONG MỘT YÊU CẦU DUY NHẤT**

Tài nguyên REST thường được định hướng xung quanh các đối tượng kinh doanh, chẳng hạn như Người tiêu dùng và

Đặt hàng. Do đó, một vấn đề phổ biến khi thiết kế REST API là làm thế nào để

cho phép khách hàng truy xuất nhiều đối tượng liên quan trong một yêu cầu duy nhất. Ví dụ,Hãy tưởng tượng rằng một máy khách REST muốn lấy một Đơn hàng và Người tiêu dùng của Đơn hàng. Một API REST thuần túy sẽ yêu cầu máy khách thực hiện ít nhất hai yêu cầu, một cho Đơn hàng và một cho Người tiêu dùng của nó. Một kịch bản phức tạp hơn sẽ yêu cầu nhiều lượt khứ hồi hơn và gặp phải độ trễ quá mức.

Một giải pháp cho vấn đề này là sử dụng API để cho phép máy khách truy xuất các tài nguyên liên quan khi nhận được một tài nguyên. Ví dụ, máy khách có thể truy xuất một Đơn hàng và Người tiêu dùng của nó bằng cách sử dụng NHẬN /orders/order-id-1345?expand=consumer. Các truy vấn tham số chỉ định các tài nguyên liên quan để trả về với Order. Cách tiếp cận này hoạt động tốt trong nhiều tình huống nhưng thường không đủ cho các tình huống phức tạp hơn. Nó cũng có khả năng tốn thời gian để triển khai. Điều này đã dẫn đến sự phổ biến ngày càng tăng của các công nghệ API thay thế như GraphQL ([http://graphql.org](http://graphql.org/)) và Netflix Falcor (<http://netflix.github.io/falcor/>), được thiết kếđể hỗ trợ việc truy xuất dữ liệu hiệu quả.

**TTHÁCH THỨC CỦA CÁC HOẠT ĐỘNG LẬP BẢN ĐỒ ĐẾNGiao thức HTTPĐỘNG TỪ**

Một vấn đề thiết kế REST API phổ biến khác là cách ánh xạ các hoạt động bạn muốn thực hiện trên một đối tượng kinh doanh thành một động từ HTTP. REST API nên sử dụng PUT để cập nhật, nhưng có thể có nhiều cách để cập nhật một đơn hàng, bao gồm hủy đơn hàng, sửa đổi đơn hàng, v.v. Ngoài ra, một bản cập nhật có thể không phải là idempotent, đây là yêu cầu để sử dụng PUT. Một giải pháp là xác định một tài nguyên phụ để cập nhật mộtkhía cạnh cụ thể của một nguồn tài nguyên. Ví dụ, Dịch vụ Đặt hàng có POST /orders/

{orderId}/hủyđiểm cuối để hủy đơn hàng vàPOST /đơn hàng/{orderId}/

sửa đổi điểm cuối để sửa đổi đơn hàng. Một giải pháp khác là chỉ định một động từ làm tham số truy vấn URL. Đáng buồn thay, không có giải pháp nào thực sự RESTful.

Vấn đề này với các hoạt động ánh xạ tới các động từ HTTP đã dẫn đến sự phổ biến ngày càng tăng của các giải pháp thay thế cho REST, chẳng hạn như gPRC, được thảo luận ngắn gọn trong phần 3.2.2. Nhưng trước tiên hãy xem xét những lợi ích và hạn chế của REST.

**BLỢI ÍCH VÀ NHƯỢC ĐIỂM CỦANGHỈ NGƠI**

Có rất nhiều lợi ích khi sử dụng REST:

* + Thật đơn giản và quen thuộc.
  + Bạn có thể kiểm tra API HTTP từ trong trình duyệt bằng cách sử dụng plugin Postman chẳng hạn hoặc từ dòng lệnh bằng curl (giả sử sử dụng JSON hoặc một số định dạng văn bản khác).
  + Nó hỗ trợ trực tiếp kiểu giao tiếp yêu cầu/phản hồi.
  + Tất nhiên, HTTP thân thiện với tường lửa.
  + Nó không yêu cầu một nhà môi giới trung gian, giúp đơn giản hóa kiến ​​trúc của hệ thống.

Có một số nhược điểm khi sử dụng REST:

* + Nó chỉ hỗ trợ kiểu giao tiếp yêu cầu/phản hồi.
  + Giảm tính khả dụng. Vì máy khách và dịch vụ giao tiếp trực tiếp mà không cần trung gian để lưu trữ tin nhắn nên cả hai đều phải chạy trong suốt thời gian trao đổi.
* Khách hàng phải biết vị trí (URL) của các phiên bản dịch vụ. Như đã mô tả trong phần 3.2.4, đây là một vấn đề không tầm thường trong một ứng dụng hiện đại. Khách hàng phảisử dụng cái được gọi là cơ chế khám phá dịch vụ để xác định vị trí các phiên bản dịch vụ.
* Việc tìm nạp nhiều tài nguyên trong một yêu cầu duy nhất là một thách thức.
* Đôi khi rất khó để ánh xạ nhiều hoạt động cập nhật vào các động từ HTTP.

Bất chấp những nhược điểm này, REST dường như là tiêu chuẩn thực tế cho API, mặc dù có một vài lựa chọn thay thế thú vị. Ví dụ, GraphQL triển khai việc truy xuất dữ liệu linh hoạt, hiệu quả. Chương 8 thảo luận về GraphQL và đề cập đến mô hình cổng API.

gRPC là một giải pháp thay thế khác cho REST. Hãy cùng xem cách thức hoạt động của nó.

###### Sử dụng gRPC

Như đã đề cập trong phần trước, một thách thức khi sử dụng REST là vì HTTP chỉ cung cấp một số lượng động từ hạn chế nên không phải lúc nào cũng dễ dàng thiết kế một REST API hỗ trợ nhiều hoạt động cập nhật. Một công nghệ IPC tránh được vấn đề này là gRPC ([www.grpc.io](http://www.grpc.io/)), một khuôn khổ để viết các máy khách và máy chủ đa ngôn ngữ (xem <https://en.wikipedia.org/wiki/Remote_procedure_call> để biết thêm). gRPC là một giao thức dựa trên tin nhắn nhị phân và điều này có nghĩa là—như đã đề cập trước đó trong phần thảo luận về các định dạng tin nhắn nhị phân—bạn buộc phải áp dụng phương pháp tiếp cận API-first đối với thiết kế dịch vụ. Bạn định nghĩa các API gRPC của mình bằng cách sử dụng IDL dựa trên Protocol Buffers, đây là cơ chế trung lập về ngôn ngữ của Google để tuần tự hóa dữ liệu có cấu trúc. Bạn sử dụng trình biên dịch Protocol Buffer để tạo các stub phía máy khách và skel- eton phía máy chủ. Trình biên dịch có thể tạo mã cho nhiều ngôn ngữ khác nhau, bao gồm Java, C#, NodeJS và GoLang. Máy khách và máy chủ trao đổi tin nhắn nhị phân theo định dạng Protocol Buffers bằng HTTP/2.

API gRPC bao gồm một hoặc nhiều dịch vụ và định nghĩa tin nhắn yêu cầu/phản hồi. Định nghĩa dịch vụ tương tự như giao diện Java và là tập hợp các phương thức được gõ mạnh. Cũng như hỗ trợ RPC yêu cầu/phản hồi đơn giản, gRPC hỗ trợ RPC phát trực tuyến. Máy chủ có thể trả lời bằng luồng tin nhắn cho máy khách. Ngoài ra, máy khách có thể gửi luồng tin nhắn đến máy chủ.

gRPC sử dụng Protocol Buffers làm định dạng tin nhắn. Protocol Buffers, như đã đề cập trước đó, là một định dạng nhị phân hiệu quả, nhỏ gọn. Đây là một định dạng được gắn thẻ. Mỗi trường của tin nhắn Protocol Buffers được đánh số và có mã loại. Người nhận tin nhắn có thể trích xuất các trường mà họ cần và bỏ qua các trường mà họ không nhận ra. Do đó, gRPC cho phép API phát triển trong khi vẫn tương thích ngược.

Danh sách 3.1 hiển thị một đoạn trích của API gRPC choDịch vụ đặt hàng. Nó định nghĩa một số phương pháp, bao gồmtạo đơn hàng(). Phương pháp này mất mộtTạoYêu cầu đặt hàngnhư một tham số và trả về mộtTạo đơn hàngTrả lời.

**Danh sách 3.1 Một đoạn trích của API gRPC choDịch vụ đặt hàng**

dịch vụ OrderService {

rpc createOrder(CreateOrderRequest) trả về (CreateOrderReply) {}

rpc cancelOrder(CancelOrderRequest) trả về (CancelOrderReply) {} rpc revisionOrder(ReviseOrderRequest) trả về (ReviseOrderReply) {}

...

}

tin nhắn CreateOrderRequest { int64 restaurantId = 1; int64 consumerId = 2;

lặp lại LineItem lineItems = 3;

...

}

tin nhắn LineItem { chuỗi menuItemId = 1; int32 số lượng = 2;

}

tin nhắn CreateOrderReply { int64 orderId = 1;

}

...

TạoYêu cầu đặt hàngVàTạo đơn hàngTrả lờiđược đánh máytin nhắn. Ví dụ,Tạo-OrderRequesttin nhắn có mộtnhà hàngIdtrường loạiint64. Giá trị thẻ của trường là 1.

gRPC có một số lợi ích:

* Việc thiết kế một API có nhiều hoạt động cập nhật là điều khá đơn giản.
* Nó có cơ chế IPC hiệu quả và nhỏ gọn, đặc biệt khi trao đổi các tin nhắn lớn.
* Truyền phát hai chiều cho phép sử dụng cả phương thức giao tiếp RPI và nhắn tin.
* Nó cho phép khả năng tương tác giữa máy khách và các dịch vụ được viết bằng nhiều ngôn ngữ khác nhau.

gRPC cũng có một số nhược điểm:

* Các máy khách JavaScript phải làm việc nhiều hơn để sử dụng API dựa trên gRPC so với API dựa trên REST/JSON.
* Tường lửa cũ hơn có thể không hỗ trợ HTTP/2.

gRPC là một giải pháp thay thế hấp dẫn cho REST, nhưng giống như REST, nó là một cơ chế giao tiếp đồng bộ, do đó nó cũng gặp phải vấn đề về lỗi một phần. Chúng ta hãy cùng xem đó là gì và cách xử lý.

###### Xử lý lỗi một phần bằng cách sử dụng Mạchmẫu ngắt

Trong một hệ thống phân tán, bất cứ khi nào một dịch vụ thực hiện yêu cầu đồng bộ đến một dịch vụ khác, luôn có nguy cơ xảy ra lỗi một phần. Vì máy khách và dịch vụ là các quy trình riêng biệt, nên dịch vụ có thể không phản hồi kịp thời yêu cầu của máy khách. Dịch vụ có thể ngừng hoạt động do lỗi hoặc để bảo trì. Hoặc dịch vụ có thể bị quá tải và phản hồi cực kỳ chậm đối với các yêu cầu.

Do máy khách bị chặn chờ phản hồi nên nguy cơ lỗi có thể lan sang các máy khách khác và gây ra tình trạng ngừng hoạt động.

**Mẫu: Cầu dao điện**

Một proxy RPI ngay lập tức từ chối các lệnh gọi trong một khoảng thời gian chờ sau khi số lần lỗi liên tiếp vượt quá ngưỡng đã chỉ định. Xem[http://dịch vụ vi mô](http://microservices.io/patterns/reliability/circuit-breaker.html)

[.io/patterns/reliability/circuit-breaker.html](http://microservices.io/patterns/reliability/circuit-breaker.html).

Ví dụ, hãy xem xét kịch bản được thể hiện trong hình 3.2, trong đóDịch vụ Order không phản hồi. Một máy khách di động thực hiện yêu cầu REST tới cổng API, như đã thảo luận trong chương 8, là điểm vào ứng dụng cho máy khách API. APIcổng chuyển tiếp yêu cầu đến Dịch vụ đặt hàng không phản hồi.

**Dịch vụ từ xa không phản hồi**

Giao diện lập trình ứng dụng (API)

cổng vào

BÀI ĐĂNG/ĐƠN HÀNG

BÀI ĐĂNG/ĐƠN HÀNG

Dịch vụ đặt hàng proxy

Tạo điểm cuối đơn hàng

Di độngứng dụng

Dịch vụ đặt hàng

**Hình 3.2 Cổng API phải tự bảo vệ mình khỏi các dịch vụ không phản hồi, chẳng hạn nhưĐặt hàngDịch vụ.**

Một triển khai ngây thơ của OrderServiceProxy sẽ chặn vô thời hạn, chờ phản hồi. Điều đó không chỉ dẫn đến trải nghiệm người dùng kém mà trong nhiều ứng dụng, nó còn tiêu tốn một tài nguyên quý giá, chẳng hạn như một luồng. Cuối cùng, cổng API sẽ hết tài nguyên và không thể xử lý các yêu cầu. Toàn bộ API sẽ không khả dụng.

Điều quan trọng là bạn phải thiết kế các dịch vụ của mình để ngăn chặn các lỗi cục bộ lan rộng khắp ứng dụng. Giải pháp có hai phần:

* Bạn phải sử dụng proxy RPI thiết kế, chẳng hạn nhưDịch vụ đặt hàngProxy, để xử lý các dịch vụ từ xa không phản hồi.
* Bạn cần quyết định cách phục hồi sau khi dịch vụ từ xa bị lỗi. Trước tiên, chúng ta sẽ xem cách viết proxy RPI mạnh mẽ.

**DPHÁT TRIỂN MẠNH MẼ RPIPROXY**

Bất cứ khi nào một dịch vụ đồng bộ gọi một dịch vụ khác, nó phải tự bảo vệ mình bằng cách sử dụng phương pháp được mô tả bởi Netflix ([http://techblog.netflix.com/2012/02/fault-](http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html) [dung-dung-trong-khối-lượng-lớn.html](http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html)). Cách tiếp cận này bao gồm sự kết hợp của các cơ chế sau:

* + *Hết thời gian chờ mạng*—Không bao giờ chặn vô thời hạn và luôn sử dụng thời gian chờ khi chờ phản hồi. Sử dụng thời gian chờ đảm bảo rằng tài nguyên không bao giờ bị ràng buộc vô thời hạn.
  + *Giới hạn số lượng yêu cầu chưa hoàn thành từ khách hàng đến một dịch vụ*—Áp đặt một trênbị ràng buộc bởi số lượng yêu cầu chưa giải quyết mà khách hàng có thể thực hiện đối với một dịch vụ cụ thể. Nếu đã đạt đến giới hạn, có lẽ việc thực hiện thêm yêu cầu là vô nghĩa và những nỗ lực đó sẽ thất bại ngay lập tức.
  + *Mẫu cầu dao điện*—Theo dõi số lượng yêu cầu thành công và không thành công, và nếu tỷ lệ lỗi vượt quá một ngưỡng nào đó, hãy ngắt cầu dao để các lần thử tiếp theo sẽ thất bại ngay lập tức. Một số lượng lớn các yêu cầu không thành công cho thấy dịch vụ không khả dụng và việc gửi thêm yêu cầu là vô nghĩa. Sau một khoảng thời gian chờ, máy khách nên thử lại và nếu thành công, hãy đóng cầu dao.

Netflix Hystrix (<https://github.com/Netflix/Hystrix>) là một thư viện mã nguồn mở triển khai các mẫu này và các mẫu khác. Nếu bạn đang sử dụng JVM, bạn chắc chắn nên cân nhắc sử dụng Hystrix khi triển khai proxy RPI. Và nếu bạn đang chạy trong môi trường không phải JVM, bạn nên sử dụng một thư viện tương đương. Ví dụ, thư viện Polly rất phổ biến trong cộng đồng .NET (<https://github.com/App-vNext/Polly>).

**RPHỦ TRỌNGTỪ MỘT DỊCH VỤ KHÔNG CÓ SẴN**

Sử dụng thư viện như Hystrix chỉ là một phần của giải pháp. Bạn cũng phải quyết định từng trường hợp cụ thể về cách dịch vụ của bạn sẽ phục hồi từ dịch vụ từ xa không phản hồi. Một tùy chọn là dịch vụ chỉ cần trả về lỗi cho máy khách của mình. Ví dụ, cách tiếp cận này có ý nghĩa đối với tình huống được hiển thị trong hình 3.2, trong đó yêu cầu tạo Đơn hàng không thành công. Tùy chọn duy nhất là cổng API trả về lỗi cho máy khách di động.

Trong các trường hợp khác, việc trả về giá trị dự phòng, chẳng hạn như giá trị mặc định hoặc phản hồi được lưu trong bộ nhớ đệm, có thể có ý nghĩa. Ví dụ, chương 7 mô tả cách cổng API có thể triển khai hoạt động truy vấn findOrder() bằng cách sử dụng thành phần APImẫu. Như hình 3.3 cho thấy, việc triển khai điểm cuối GET /orders/{orderId} của nó sẽ gọi một số dịch vụ, bao gồm Dịch vụ đặt hàng, Dịch vụ nhà bếp và Dịch vụ giao hàng, rồi kết hợp các kết quả.

Có khả năng là dữ liệu của mỗi dịch vụ không quan trọng như nhau đối với khách hàng. Dữ liệutừ Dịch vụ Đặt hàng là điều cần thiết. Nếu dịch vụ này không khả dụng, cổng API sẽ trả về phiên bản dữ liệu được lưu trong bộ nhớ đệm hoặc lỗi. Dữ liệu từ các dịch vụ khác ít quan trọng hơn. Ví dụ, máy khách có thể hiển thị thông tin hữu ích cho người dùng ngay cả khi trạng thái giao hàng không khả dụng. Nếu Dịch vụ Giao hàng không khả dụng,

**Làm thế nào để xử lý từng dịch vụ không phản hồi?**

**Không phản hồi**

**dịch vụ**

Giao diện lập trình ứng dụng (API)

cổng vào

NHẬN/đơn hàng/xyz

NHẬN/vé?orderId=xyz

Nhận/đơn hàng/xyz

NHẬN/giao hàng?orderId-xyz

...

Dịch vụngười đại diện

Di độngứng dụng

Dịch vụ đặt hàng

Vận chuyểnDịch vụ proxy

Nhận điểm cuối đơn hàng

Dịch vụ nhà bếpngười đại diện

Dịch vụ đặt hàng proxy

...

Dịch vụ

Vận chuyểnDịch vụ

Dịch vụ nhà bếp

**Hình 3.3 Cổng API triển khaiNHẬN /đơn hàng/{orderId}điểm cuối sử dụng API composition. Nó gọi một số dịch vụ, tổng hợp phản hồi của chúng và gửi phản hồi đến ứng dụng di động. Mã triển khai điểm cuối phải có chiến lược xử lý lỗicủa mỗi dịch vụ mà nó gọi.**

cổng API phải trả về phiên bản dữ liệu được lưu trong bộ nhớ đệm hoặc loại bỏ dữ liệu đó khỏi phản hồi.

Điều cần thiết là bạn phải thiết kế các dịch vụ của mình để xử lý lỗi một phần, nhưng đó không phải là vấn đề duy nhất bạn cần giải quyết khi sử dụng RPI. Một vấn đề khác là để một dịch vụ có thể gọi một dịch vụ khác bằng RPI, nó cần biết vị trí mạng của một phiên bản dịch vụ. Trên bề mặt, điều này nghe có vẻ đơn giản, nhưng trên thực tế, đây là một thách thứcvấn đề. Bạn phải sử dụng cơ chế khám phá dịch vụ. Hãy cùng xem cơ chế đó hoạt động như thế nào.

###### Sử dụng dịch vụ khám phá

Giả sử bạn đang viết một số mã lệnh gọi một dịch vụ có REST API. Để thực hiện một yêu cầu, mã lệnh của bạn cần biết vị trí mạng (địa chỉ IP và cổng) của một phiên bản dịch vụ. Trong một ứng dụng truyền thống chạy trên phần cứng vật lý, vị trí mạng của các phiên bản dịch vụ thường là tĩnh. Ví dụ, mã lệnh của bạn có thể đọc vị trí mạng từ một tệp cấu hình thỉnh thoảng được cập nhật. Nhưng trong một ứng dụng vi dịch vụ hiện đại dựa trên đám mây, thường thì không đơn giản như vậy. Như được thể hiện trong hình 3.4, một ứng dụng hiện đại năng động hơn nhiều.

Các trường hợp dịch vụ có các vị trí mạng được chỉ định động. Hơn nữa, tập hợp các trường hợp dịch vụ thay đổi động do tự động mở rộng, lỗi và nâng cấp. Do đó, mã máy khách của bạn phải sử dụng khám phá dịch vụ.

**IP được gán động**

**Được tạo ra và phá hủy một cách năng động**

10.2

|  |  |  |
| --- | --- | --- |
| 32.23.1 | Dịch vụ đặt hàng | |
|  | Dịch vụ thể hiện 1 |
| 32.23.2 |  |
|  | |
|  | Dịch vụ thể hiện 2 |
| 32.23.3 |  |
|  | |
|  | Dịch vụ thể hiện 3 |
|  |  |
|  | |



10.2

10.2

?

Dịch vụ

khách hàng

**Hình 3.4 Các phiên bản dịch vụ có địa chỉ IP được gán động.**

**ỒTỔNG QUANKHÁM PHÁ DỊCH VỤ**

Như bạn vừa thấy, bạn không thể cấu hình tĩnh một máy khách với các địa chỉ IP của các dịch vụ. Thay vào đó, một ứng dụng phải sử dụng một cơ chế khám phá dịch vụ động. Về mặt khái niệm, khám phá dịch vụ khá đơn giản: thành phần chính của nó là một sổ đăng ký dịch vụ, là cơ sở dữ liệu về các vị trí mạng của các phiên bản dịch vụ của ứng dụng.

Cơ chế khám phá dịch vụ cập nhật sổ đăng ký dịch vụ khi các phiên bản dịch vụ bắt đầu và dừng. Khi máy khách gọi một dịch vụ, cơ chế khám phá dịch vụ sẽ truy vấn sổ đăng ký dịch vụ để lấy danh sách các phiên bản dịch vụ khả dụng và định tuyến yêu cầu đến một trong số chúng.

Có hai cách chính để triển khai khám phá dịch vụ:

* Các dịch vụ và khách hàng của họ tương tác trực tiếp với cơ quan đăng ký dịch vụ.
* Cơ sở hạ tầng triển khai xử lý việc khám phá dịch vụ. (Tôi sẽ nói thêm về điều đó trong chương 12.)

Chúng ta hãy xem xét từng lựa chọn.

**MỘTÁP DỤNG ỨNG DỤNG-MẪU PHÁT HIỆN DỊCH VỤ CẤP ĐỘ**

Một cách để triển khai khám phá dịch vụ là để các dịch vụ của ứng dụng và các máy khách của chúng tương tác với sổ đăng ký dịch vụ. Hình 3.5 cho thấy cách thức hoạt động của điều này. Một phiên bản dịch vụ đăng ký vị trí mạng của nó với sổ đăng ký dịch vụ. Một máy khách dịch vụ gọi một dịch vụ bằng cách đầu tiên truy vấn sổ đăng ký dịch vụ để có được danh sách các phiên bản dịch vụ. Sau đó, nó gửi yêu cầu đến một trong những phiên bản đó.

|  |  |
| --- | --- |
| Dịch vụ | Địa chỉ IP |
| dịch vụ đặt hàng | 10.232.23.1 |
| dịch vụ đặt hàng | 10.232.23.2 |
| dịch vụ đặt hàng | 10.232.23.3 |
| ... | ... |

**Hình 3.5 Sổ đăng ký dịch vụ theo dõi các phiên bản dịch vụ. Khách hàng truy vấn dịch vụđăng ký để tìm vị trí mạng của các phiên bản dịch vụ khả dụng.**



Dịch vụ đặt hàng

10.232.23.1

**Khám phá phía máy khách**

Dịch vụ

ví dụ 1

Đăng ký("order-service", "10.232.23.1")

10.232.23.2

Yêu cầu cân bằng tải

Dịch vụ

trường hợp 2

10.232.23.1

10.232.23.2

10.232.23.3

Truy vấn ("order-service")

10.232.23.3

API truy vấn

API đăng ký

Đăng ký dịch vụ

**Mẫu tự đăng ký**

Dịch vụ khách hàng

Thư viện khám phá dịch vụ

Máy khách RPC/rest

Thư viện khám phá dịch vụ

Dịch vụ thể hiện 3

Thư viện khám phá dịch vụ

Thư viện khám phá dịch vụ

Cách tiếp cận này để khám phá dịch vụ là sự kết hợp của hai mẫu. Mẫu đầu tiên là mẫu Tự đăng ký. Một phiên bản dịch vụ gọi API đăng ký của sổ đăng ký dịch vụ để đăng ký vị trí mạng của nó. Nó cũng có thể cung cấp URL kiểm tra tình trạng, được mô tả chi tiết hơn trong chương 11. URL kiểm tra tình trạng là điểm cuối API mà sổ đăng ký dịch vụ gọi định kỳ để xác minh rằng phiên bản dịch vụ đang hoạt động bình thường và sẵn sàng xử lý các yêu cầu. Sổ đăng ký dịch vụ có thể yêu cầu phiên bản dịch vụ gọi định kỳ API "nhịp tim" để ngăn không cho đăng ký của nó hết hạn.

**Mẫu: Tự đăng ký**

Một phiên bản dịch vụ tự đăng ký với sổ đăng ký dịch vụ. Xem[http://microser-](http://microservices.io/patterns/self-registration.html) [Vices.io/patterns/self-registration.html](http://microservices.io/patterns/self-registration.html).

Mẫu thứ hai là khám phá phía máy kháchmẫu. Khi một máy khách dịch vụ muốn gọi một dịch vụ, nó sẽ truy vấn sổ đăng ký dịch vụ để lấy danh sách các phiên bản của dịch vụ. Để cải thiện hiệu suất, máy khách có thể lưu trữ đệm các phiên bản dịch vụ. Máy khách dịch vụ

sau đó sử dụng cân bằng tảithuật toán, chẳng hạn như vòng tròn hoặc ngẫu nhiên, để chọn một phiên bản dịch vụ. Sau đó, nó đưa ra yêu cầu đến một phiên bản dịch vụ được chọn.

**Mẫu: Khám phá phía máy khách**

Một máy khách dịch vụ sẽ lấy danh sách các phiên bản dịch vụ khả dụng từ sổ đăng ký dịch vụ.istry và cân bằng tải trên chúng. Xem[http://microservices.io/patterns/client-](http://microservices.io/patterns/client-side-discovery.html) [khám phá bên cạnh.html](http://microservices.io/patterns/client-side-discovery.html).

Khám phá dịch vụ cấp ứng dụng đã được Netflix và Pivotal phổ biến. Netflix đã phát triển và mã nguồn mở một số thành phần: Eureka, một sổ đăng ký dịch vụ có tính khả dụng cao, máy khách Eureka Java và Ribbon, một máy khách HTTP tinh vi hỗ trợ máy khách Eureka. Pivotal đã phát triển Spring Cloud, một khuôn khổ dựa trên Spring giúp sử dụng các thành phần Netflix dễ dàng đáng kể. Các dịch vụ dựa trên Spring Cloud tự động đăng ký với Eureka và các máy khách dựa trên Spring Cloud tự động sử dụng Eureka để khám phá dịch vụ.

Một lợi ích của khám phá dịch vụ cấp ứng dụng là nó xử lý tình huống khi các dịch vụ được triển khai trên nhiều nền tảng triển khai. Ví dụ, hãy tưởng tượng bạn chỉ triển khai một số dịch vụ trên Kubernetes, được thảo luận trong chương 12 và phần còn lại đang chạy trong môi trường cũ. Ví dụ, khám phá dịch vụ cấp ứng dụng bằng Eureka hoạt động trên cả hai môi trường, trong khi khám phá dịch vụ dựa trên Kubernetes chỉ hoạt động trong Kubernetes.

Một nhược điểm của khám phá dịch vụ cấp ứng dụng là bạn cần một thư viện khám phá dịch vụ cho mọi ngôn ngữ—và có thể là cả khuôn khổ—mà bạn sử dụng. Spring Cloud chỉ giúp các nhà phát triển Spring. Nếu bạn đang sử dụng một số khuôn khổ Java khác hoặc một ngôn ngữ không phải JVM như NodeJS hoặc GoLang, bạn phải tìm một số khuôn khổ khám phá dịch vụ khác. Một nhược điểm khác của khám phá dịch vụ cấp ứng dụng là bạn chịu trách nhiệm thiết lập và quản lý sổ đăng ký dịch vụ, đây là một sự sao nhãng. Do đó, thường thì tốt hơn làsử dụng cơ chế khám phá dịch vụ được cung cấp bởi cơ sở hạ tầng triển khai.

**MỘTĐÓNG BỘ NỀN TẢNG-DỊCH VỤ CUNG CẤP CÁC MẪU KHÁM PHÁ**

Sau đó trong chương 12, bạn sẽ biết rằng nhiều nền tảng triển khai hiện đại như Docker và Kubernetes có cơ chế đăng ký dịch vụ và khám phá dịch vụ tích hợp sẵn. Nền tảng triển khai cung cấp cho mỗi dịch vụ một tên DNS, một địa chỉ IP ảo (VIP) và một tên DNS phân giải thành địa chỉ VIP. Một máy khách dịch vụ đưa ra yêu cầu đến tên DNS/VIP và nền tảng triển khai sẽ tự động định tuyến yêu cầu đến một trong các phiên bản dịch vụ khả dụng. Do đó, việc đăng ký dịch vụ, khám phá dịch vụ và định tuyến yêu cầu hoàn toàn được xử lý bởi nền tảng triển khai. Hình 3.6 cho thấy cách thức hoạt động của việc này.

Việc triển khainền tảng bao gồm một sổ đăng ký dịch vụ theo dõi địa chỉ IP củacác dịch vụ được triển khai. Trong ví dụ này, một khách hàng truy cập Dịch vụ đơn hàng bằng cách sử dụng

|  |  |
| --- | --- |
| Dịch vụ | Địa chỉ IP |
| dịch vụ đặt hàng | 10.232.23.1 |
| dịch vụ đặt hàng | 10.232.23.2 |
| dịch vụ đặt hàng | 10.232.23.3 |
| ... | ... |

**Dịch vụ IP ảođịa chỉ (VIP)**

**Tên dịch vụ DNS phân giải thành dịch vụ VIP**

Dịch vụ đặt hàng

10.232.23.1

Dịch vụ

khách hàng

10.232.23.2

LẤY[http://order-service/...](http://order-service/)

10.232.23.3

10.232.24.99

Quan sát

Đăng ký dịch vụ

Truy vấn

Cập nhật

Nền tảng triển khai

Người đăng ký

Bộ định tuyến nền tảng

Dịch vụ thể hiện 3

Dịch vụ thể hiện 2

Dịch vụ thể hiện 1

Máy khách RPC/rest

**Khám phá phía máy chủ**

**Đăng ký bên thứ 3**

**Hình 3.6 Nền tảng chịu trách nhiệm đăng ký dịch vụ, khám phá và định tuyến yêu cầu. Các trường hợp dịch vụ được đăng ký với sổ đăng ký dịch vụ bởi*người đăng ký*. Mỗi dịch vụ có một vị trí mạng,tên DNS/địa chỉ IP ảo. Máy khách gửi yêu cầu đến vị trí mạng của dịch vụ. Bộ định tuyến truy vấn sổ đăng ký dịch vụ và cân bằng tải các yêu cầu trên các phiên bản dịch vụ khả dụng.**

Tên DNS order-service, giải quyết thành địa chỉ IP ảo 10.1.3.4. Nền tảng triển khai tự động cân bằng tải các yêu cầu trên ba phiên bản của Order Service.

Cách tiếp cận này là sự kết hợp của hai mô hình:

* *Mẫu đăng ký của bên thứ 3*—Thay vì một dịch vụ tự đăng ký với cơ quan đăng ký dịch vụ, bên thứ ba được gọi là cơ quan đăng ký, thường là một phần của nền tảng triển khai, sẽ xử lý việc đăng ký.
* *Mẫu khám phá phía máy chủ*—Thay vì máy khách truy vấn sổ đăng ký dịch vụ, nó sẽ gửi yêu cầu đến tên DNS, sau đó chuyển đến bộ định tuyến yêu cầu truy vấn sổ đăng ký dịch vụ và cân bằng tải các yêu cầu.

**Mẫu:Đăng ký bên thứ 3**

Các phiên bản dịch vụ được tự động đăng ký với cơ quan đăng ký dịch vụ bởi bên thứ ba.Nhìn thấy<http://microservices.io/patterns/3rd-party-registration.html>.

**Mẫu: Phía máy chủkhám phá**

Một máy khách gửi yêu cầu đến bộ định tuyến, bộ định tuyến này chịu trách nhiệm về dịch vụkhám phá. Xem<http://microservices.io/patterns/server-side-discovery.html>.

Lợi ích chính của khám phá dịch vụ do nền tảng cung cấp là mọi khía cạnh của khám phá dịch vụ đều được xử lý hoàn toàn bởi nền tảng triển khai. Cả dịch vụ và máy khách đều không chứa bất kỳ mã khám phá dịch vụ nào. Do đó, cơ chế khám phá dịch vụ luôn sẵn sàng cho tất cả các dịch vụ và máy khách bất kể chúng được viết bằng ngôn ngữ hay khuôn khổ nào.

Một nhược điểm của khám phá dịch vụ do nền tảng cung cấp là nó chỉ hỗ trợ khám phá các dịch vụ đã được triển khai bằng nền tảng. Ví dụ, như đã đề cập trước đó khi mô tả khám phá cấp ứng dụng, khám phá dựa trên Kubernetes chỉ hoạt động đối với các dịch vụ chạy trên Kubernetes. Bất chấp hạn chế này, tôi khuyên bạn nên sử dụng khám phá dịch vụ do nền tảng cung cấp bất cứ khi nào có thể.

Bây giờ chúng ta đã xem xét IPC đồng bộ sử dụng REST hoặc gRPC, chúng ta hãy cùng xem xét giải pháp thay thế: giao tiếp không đồng bộ, dựa trên tin nhắn.

#### Giao tiếp bằng cách sử dụng tin nhắn không đồng bộmẫu

Khi sử dụng tin nhắn, các dịch vụ giao tiếp bằng cách trao đổi tin nhắn không đồng bộ. Một ứng dụng dựa trên tin nhắn thường sử dụng một nhà môi giới tin nhắn, đóng vai trò là trung gian giữa các dịch vụ, mặc dù một tùy chọn khác là sử dụng kiến ​​trúc không có nhà môi giới, trong đó các dịch vụ giao tiếp trực tiếp với nhau. Một máy khách dịch vụ đưa ra yêu cầu đến một dịch vụ bằng cách gửi cho dịch vụ đó một tin nhắn. Nếu phiên bản dịch vụ được mong đợi trả lời, nó sẽ thực hiện bằng cách gửi một tin nhắn riêng trở lại máy khách. Vì giao tiếp là không đồng bộ, máy khách không chặn chờ trả lời. Thay vào đó, máy khách được viết với giả định rằng phản hồi sẽ không được nhận ngay lập tức.

**Mẫu: Nhắn tin**

Một máy khách gọi một dịch vụ bằng cách sử dụng tin nhắn không đồng bộ. Xem[http://dịch vụ vi mô](http://microservices.io/patterns/communication-style/messaging.html)

[.io/patterns/communication-style/messaging.html](http://microservices.io/patterns/communication-style/messaging.html).

Tôi bắt đầu phần này bằng phần tổng quan về nhắn tin. Tôi chỉ cách mô tảmột kiến ​​trúc nhắn tin độc lập với công nghệ nhắn tin. Tiếp theo tôi so sánh và đối chiếu

kiến trúc brokerless và broker-based và mô tả các tiêu chí để lựa chọn một message broker. Sau đó, tôi thảo luận về một số chủ đề quan trọng, bao gồm việc mở rộng quy mô người tiêu dùng trong khi vẫn giữ nguyên thứ tự tin nhắn, phát hiện và loại bỏ các tin nhắn trùng lặp, và gửi và nhận tin nhắn như một phần của giao dịch cơ sở dữ liệu. Chúng ta hãy bắt đầu bằng cách xem cách thức hoạt động của tin nhắn.

###### Tổng quan về tin nhắn

Một mô hình hữu ích về nhắn tin được định nghĩa trong cuốn sách Enterprise Integration Patterns (Addison-Wesley Professional, 2003) của Gregor Hohpe và Bobby Woolf. Trong mô hình này, các tin nhắn được trao đổi qua các kênh tin nhắn. Người gửi (một ứng dụng hoặc dịch vụ) viết tin nhắn vào một kênh và người nhận (một ứng dụng hoặc dịch vụ) đọc tin nhắn từ một kênh. Chúng ta hãy xem xét các tin nhắn và sau đó xem xét các kênh.

**MỘTVỀ TIN NHẮN**

Một tin nhắn bao gồm phần tiêu đề và phần nội dung tin nhắn ([www.enterpriseintegrationpatterns](http://www.enterpriseintegrationpatterns.com/Message.html)

[.com/Tin nhắn.html](http://www.enterpriseintegrationpatterns.com/Message.html)). Tiêu đề là tập hợp các cặp tên-giá trị, siêu dữ liệu mô tả dữ liệu đang được gửi. Ngoài các cặp tên-giá trị do người gửi tin nhắn cung cấp, tiêu đề tin nhắn còn chứa các cặp tên-giá trị, chẳng hạn như ID tin nhắn duy nhất do người gửi hoặc cơ sở hạ tầng tin nhắn tạo ra và địa chỉ trả về tùy chọn, chỉ định kênh tin nhắn mà phản hồi sẽ được ghi vào. Nội dung tin nhắn là dữ liệu đang được gửi, ở định dạng văn bản hoặc nhị phân.

Có một số loại khác nhautin nhắn:

* + - * *Tài liệu*—Một thông điệp chung chỉ chứa dữ liệu. Người nhận quyết định cách diễn giải thông điệp đó. Phản hồi cho một lệnh là một ví dụ về thông điệp tài liệu.
      * *Yêu cầu*—Một thông báo tương đương với yêu cầu RPC. Nó chỉ định thao tác cần gọi và các tham số của thao tác đó.
      * *Sự kiện*—Một thông điệp cho biết có điều gì đó đáng chú ý đã xảy ra ở người gửi.Một sự kiện thường là một sự kiện miền, biểu thị sự thay đổi trạng thái của một đối tượng miền nhưĐặt hàng, hoặc mộtKhách hàng.

Phương pháp tiếp cận kiến ​​trúc vi dịch vụ được mô tả trong cuốn sách này sử dụng rộng rãi các lệnh và sự kiện.

Bây giờ chúng ta hãy xem xét các kênh, cơ chế mà các dịch vụ sử dụng để giao tiếp.

**MỘTVỀ KÊNH TIN NHẮN**

Như hình 3.7 cho thấy, tin nhắnđược trao đổi qua các kênh ([www.enterpriseintegra-](http://www.enterpriseintegrationpatterns.com/MessageChannel.html) [tionpotypes.com/MessageChannel.html](http://www.enterpriseintegrationpatterns.com/MessageChannel.html)). Logic nghiệp vụ trong người gửi gọi một giao diện cổng gửi, đóng gói cơ chế giao tiếp cơ bản. Cổng gửi được triển khai bởi một lớp bộ điều hợp người gửi tin nhắn, lớp này gửi tin nhắn đến người nhận qua một kênh tin nhắn. Kênh tin nhắn là một sự trừu tượng của cơ sở hạ tầng nhắn tin. Một lớp bộ điều hợp trình xử lý tin nhắn trong người nhận được gọi để xử lý tin nhắn. Nó gọi một giao diện cổng nhận được triển khai bởi người tiêu dùng

Người gửiNgười nhận



**Cổng gửi**

**Cổng nhận**

Việc kinh doanh

lý luận

Nhắn tin

cơ sở hạ tầng

Logic kinh doanh

gọi

Tin nhắn

Gửi

Nhận được

gọi

Tiêu đề

Thân hình

Tin nhắn

kênh

Tin nhắnngười gửi

Tin nhắnngười xử lý

Dịch vụ

**Hình 3.7 Logic kinh doanhtrong người gửi gọi một giao diện cổng gửi, được triển khai bởi một bộ điều hợp người gửi tin nhắn. Người gửi tin nhắn gửi tin nhắn đến người nhận qua một kênh tin nhắn. Kênh tin nhắn là một sự trừu tượng của cơ sở hạ tầng nhắn tin. Một bộ điều hợp xử lý tin nhắn trong người nhận được gọi để xử lýtin nhắn. Nó gọi giao diện cổng nhận được triển khai bởi logic nghiệp vụ của người nhận.**

logic kinh doanh. Bất kỳ số lượng người gửi nào cũng có thể gửi tin nhắn đến một kênh. Tương tự như vậy, bất kỳ số lượng người nhận nào cũng có thể nhận tin nhắn từ một kênh.

Có hai loại kênh: điểm-đến-điểm ([www.enterpriseintegrationpatterns](http://www.enterpriseintegrationpatterns.com/PointToPointChannel.html)

[.com/PointToPointChannel.html](http://www.enterpriseintegrationpatterns.com/PointToPointChannel.html)) Vàxuất bản-đăng ký ([www.enterpriseintegration-](http://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html) [mẫu.com/PublishSubscribeChannel.html](http://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html)):

* + - * + Kênh điểm-điểm truyền tải thông điệp đến đúng một trong những người tiêu dùng đang đọc từ kênh. Các dịch vụ sử dụng kênh điểm-điểm cho các kiểu tương tác một-một được mô tả trước đó. Ví dụ, thông điệp lệnh thường được gửi qua kênh điểm-điểm.
        + Kênh publish-subscribe phân phối từng tin nhắn đến tất cả người tiêu dùng được kết nối. Các dịch vụ sử dụng kênh publish-subscribe cho các kiểu tương tác một-nhiều được mô tả trước đó. Ví dụ, một tin nhắn sự kiện thường được gửi qua kênh publish-subscribe.

###### Thực hiện các phong cách tương tác bằng cách sử dụng tin nhắn

Một trong những tính năng có giá trị của nhắn tin là nó đủ linh hoạt để hỗ trợ tất cả các kiểu tương tác được mô tả trong phần 3.1.1. Một số kiểu tương tác được triển khai trực tiếp bằng nhắn tin. Những kiểu khác phải được triển khai trên đầu tin nhắn.

Hãy cùng xem cách triển khai từng kiểu tương tác, bắt đầu bằng yêu cầu/phản hồivà yêu cầu/phản hồi không đồng bộ.

**TÔIYÊU CẦU THỰC HIỆN/PHẢN HỒI VÀ YÊU CẦU KHÔNG ĐỒNG BỘ/PHẢN ỨNG**

Khi máy khách và dịch vụ tương tác bằng cách sử dụng yêu cầu/phản hồi hoặc yêu cầu/phản hồi không đồng bộ, máy khách sẽ gửi yêu cầu và dịch vụ sẽ gửi lại phản hồi.

sự khác biệt giữa hai kiểu tương tác là với yêu cầu/phản hồi, khách hàng mong đợi dịch vụ phản hồi ngay lập tức, trong khi với yêu cầu/phản hồi không đồng bộ thì không có kỳ vọng như vậy. Nhắn tin vốn không đồng bộ, do đó chỉ cung cấp yêu cầu/phản hồi không đồng bộ. Nhưng khách hàng có thể chặn cho đến khi nhận được phản hồi.

Máy khách và dịch vụ triển khai tương tác theo kiểu yêu cầu/phản hồi không đồng bộ bằng cách trao đổi một cặp tin nhắn. Như hình 3.8 cho thấy, máy khách gửi tin nhắn lệnh, trong đó chỉ định thao tác cần thực hiện và các tham số, đến kênh nhắn tin điểm-đến-điểm do dịch vụ sở hữu. Dịch vụ xử lý các yêu cầu và gửi tin nhắn trả lời, trong đó có kết quả, đến kênh điểm-đến-điểm do máy khách sở hữu.

**Máy khách gửi tin nhắn có chứa msgId và kênh trả lời.**



Lời yêu cầu

Yêu cầu kênh

Gửi

MessageId: msgId Địa chỉ trả về: ReplyKênh

Thân hình

Đọc

Chỉ định

Đọc

Hồi đáp

Gửi

CorrelationId:msgId

Thân hình

Kênh trả lời

Dịch vụ

Khách hàng

**Dịch vụ gửi phản hồi đến kênh phản hồi được chỉ định. Phản hồi chứa một relativeId, là msgId của yêu cầu.**

**Hình 3.8 Triển khai yêu cầu/phản hồi không đồng bộ bằng cách bao gồm kênh trả lời và mã định danh tin nhắn trong tin nhắn yêu cầu. Người nhận xử lý tin nhắn và gửi phản hồi đếnkênh trả lời được chỉ định.**

Máy khách phải cho dịch vụ biết nơi gửi tin nhắn trả lời và phải khớp tin nhắn trả lời với yêu cầu. May mắn thay, việc giải quyết hai vấn đề này không quá khó. Máy khách gửi tin nhắn lệnh có tiêu đề kênh trả lời. Máy chủ viết tin nhắn trả lời.cây xô thơm, có chứa một mối tương quan*nhận dạng*có cùng giá trị với mã định danh tin nhắn, đến kênh trả lời. Máy khách sử dụng mã định danh tương quan để khớp tin nhắn trả lời với yêu cầu.

Vì máy khách và dịch vụ giao tiếp bằng tin nhắn, nên tương tác này về bản chất là không đồng bộ. Về lý thuyết, máy khách nhắn tin có thể chặn cho đến khi nhận được phản hồi, nhưng trên thực tế, máy khách sẽ xử lý phản hồi theo cách không đồng bộ. Hơn nữa, phản hồi thường được xử lý bởi bất kỳ phiên bản nào của máy khách.

**TÔITHỰC HIỆN MỘT-THÔNG BÁO ĐƯỜNG ĐI**

Việc triển khai thông báo một chiều rất đơn giản khi sử dụng tin nhắn không đồng bộ. Máy khách gửi tin nhắn, thường là tin nhắn lệnh, đến kênh điểm-đến-điểm do dịch vụ sở hữu. Dịch vụ đăng ký kênh và xử lý tin nhắn. Nó không gửi lại phản hồi.

**TÔITHỰC HIỆN XUẤT BẢN/ĐẶT MUA**

Nhắn tin có hỗ trợ tích hợp cho kiểu tương tác publish/subscribe. Một máy khách sẽ xuất bản một tin nhắn đến một kênh publish-subscribe được nhiều người dùng đọc. Như đã mô tả trong chương 4 và 5, các dịch vụ sử dụng publish/subscribe để xuất bản các sự kiện miền, biểu diễn các thay đổi đối với các đối tượng miền. Dịch vụ xuất bản các sự kiện miền sở hữu một kênh publish-subscribe, có tên bắt nguồn từlớp miền. Ví dụ, Dịch vụ Đặt hàng xuất bản các sự kiện Đặt hàng tới một kênh Đặt hàng và Dịch vụ Giao hàng xuất bản các sự kiện Giao hàng tới một kênh Giao hàng. Một dịch vụ quan tâm đến các sự kiện của một đối tượng miền cụ thể chỉ phải đăng ký kênh thích hợp.

**TÔITHỰC HIỆN XUẤT BẢN/PHẢN HỒI KHÔNG ĐỒNG BỘ**

Kiểu tương tác phản hồi publish/async là kiểu tương tác cấp cao hơn được triển khai bằng cách kết hợp các yếu tố publish/subscribe và request/response. Một client sẽ publish một thông báo chỉ định tiêu đề kênh trả lời tới một kênh publish-subscribe. Một consumer sẽ viết một thông báo trả lời có chứa id tương quan tới kênh trả lời. Client sẽ thu thập các phản hồi bằng cách sử dụng id tương quan để khớp các thông báo trả lời với yêu cầu.

Mỗi dịch vụ trong ứng dụng của bạn có API không đồng bộ sẽ sử dụng một hoặc nhiều kỹ thuật triển khai này. Một dịch vụ có API không đồng bộ để gọi các hoạt động sẽ có một kênh tin nhắn cho các yêu cầu. Tương tự như vậy, một dịch vụ phát hành sự kiện sẽ phát hành chúng đến một kênh tin nhắn sự kiện.

Như đã mô tả trong phần 3.1.2, điều quan trọng là phải viết một đặc tả API cho một dịch vụ. Hãy cùng xem cách thực hiện điều đó cho một API không đồng bộ.

###### Tạo thông số kỹ thuật API cho API dịch vụ dựa trên tin nhắn

Đặc điểm kỹ thuậtđối với API không đồng bộ của dịch vụ, như hình 3.9 cho thấy, phải chỉ định tên của các kênh tin nhắn, các loại tin nhắn được trao đổi qua mỗi kênh và định dạng của chúng. Bạn cũng phải mô tả định dạng của các tin nhắn bằng một tiêu chuẩn như JSON, XML hoặc Protobuf. Nhưng không giống như REST và Open API, không có tiêu chuẩn nào được áp dụng rộng rãi để ghi lại các kênh và các loại tin nhắn. Thay vào đó, bạn cần phải viết một tài liệu không chính thức.

API không đồng bộ của dịch vụ bao gồm các hoạt động, được khách hàng gọi và các sự kiện, được dịch vụ công bố. Chúng được ghi lại theo nhiều cách khác nhau. Hãy cùng xem xét từng hoạt động, bắt đầu với các hoạt động.



API dịch vụ

Lệnh

CCC

«Kênh chỉ huy»

Yêu cầu

truy vấnGiao diện lập trình ứng dụng (API)

Trả lời

Dịch vụ

«Kênh trả lời»

RRR

Sự kiện

«Kênh sự kiện»

Nhà xuất bản sự kiện

**Hình 3.9 API không đồng bộ của dịch vụ bao gồm các kênh tin nhắn và lệnh, trả lời vàcác loại tin nhắn sự kiện.**

**DHOẠT ĐỘNG KHÔNG ĐỒNG BỘ TẠO RA**

Các hoạt động của dịch vụ có thể được gọi bằng một trong hai kiểu tương tác khác nhau:

* + - * *API theo kiểu phản hồi yêu cầu/không đồng bộ*—Bao gồm kênh tin nhắn lệnh của dịch vụ, các loại và định dạng của các loại tin nhắn lệnh mà dịch vụ chấp nhận, cũng như các loại và định dạng của các tin nhắn trả lời được dịch vụ gửi đi.
      * *Kiểu thông báo một chiềuGiao diện lập trình ứng dụng (API)*—Điều này bao gồm kênh tin nhắn lệnh của dịch vụ và các loại và định dạng của các loại tin nhắn lệnh mà dịch vụ chấp nhận.

Một dịch vụ có thể sử dụng cùng một kênh yêu cầu cho cả yêu cầu/phản hồi không đồng bộ và thông báo một chiều.

**DSỰ KIỆN ĐÃ XUẤT BẢN**

Một dịch vụ cũng có thể xuất bản sự kiện bằng cách sử dụng kiểu tương tác xuất bản/đăng ký. Đặc điểm kỹ thuật của kiểu API này bao gồm kênh sự kiện và các loại và định dạng của các thông báo sự kiện được dịch vụ xuất bản tới kênh.

Mô hình tin nhắn và kênh của tin nhắn là một sự trừu tượng tuyệt vời và là một cách tốt để thiết kế API không đồng bộ của dịch vụ. Nhưng để triển khai một dịch vụ, bạn cần chọn một công nghệ nhắn tin và xác định cách triển khai thiết kế của mình bằng cách sử dụng các khả năng của nó. Hãy cùng xem xét những gì liên quan.

###### Sử dụng một nhà môi giới tin nhắn

Một ứng dụng dựa trên tin nhắn thường sử dụng một môi giới tin nhắn, một dịch vụ cơ sở hạ tầng mà dịch vụ giao tiếp thông qua đó. Nhưng kiến ​​trúc dựa trên môi giới không phải là kiến ​​trúc tin nhắn duy nhất. Bạn cũng có thể sử dụng kiến ​​trúc tin nhắn không dựa trên môi giới, trong đó các dịch vụ giao tiếp trực tiếp với nhau. Hai cách tiếp cận, được thể hiện trong hình 3.10, có những sự đánh đổi khác nhau, nhưng thường thì kiến ​​trúc dựa trên môi giới là cách tiếp cận tốt hơn.

Không có môi giớikiến trúcKiến trúc dựa trên môi giới

So với



Dịch vụ

Dịch vụ

Dịch vụ

Dịch vụ

Dịch vụ

Dịch vụ

Người môi giới tin nhắn

**Hình 3.10 Các dịch vụ trong kiến ​​trúc không có môi giới giao tiếp trực tiếp, trong khi các dịch vụtrong kiến ​​trúc dựa trên môi giới giao tiếp thông qua một môi giới tin nhắn.**

Cuốn sách này tập trung vào kiến ​​trúc dựa trên môi giới, nhưng bạn cũng nên xem qua kiến ​​trúc không có môi giới, vì có thể có những trường hợp bạn thấy kiến ​​trúc này hữu ích.

**BTIN NHẮN ROKERLESS**

Trong kiến ​​trúc không có môi giới, các dịch vụ có thể trao đổi tin nhắn trực tiếp. ZeroMQ ([http://](http://zeromq.org/) [zeromq.org](http://zeromq.org/)) là một công nghệ nhắn tin không cần môi giới phổ biến. Nó vừa là một đặc điểm kỹ thuật vừa là một tập hợp các thư viện cho các ngôn ngữ khác nhau. Nó hỗ trợ nhiều phương thức vận chuyển, bao gồm TCP, socket miền kiểu UNIX và đa hướng.

Kiến trúc không có môi giới có một số lợi ích:

* Cho phép lưu lượng mạng nhẹ hơn và độ trễ tốt hơn, vì tin nhắn được chuyển trực tiếp từ người gửi đến người nhận, thay vì phải chuyển từ người gửi đến trung gian tin nhắn và từ đó đến người nhận.
* Loại bỏ khả năng môi giới tin nhắn là nút thắt cổ chai về hiệu suất hoặc là điểm lỗi duy nhất
* Có ít tính phức tạp trong hoạt động hơn vì không có nhà môi giới tin nhắn nào để thiết lập và duy trì

Mặc dù những lợi ích này có vẻ hấp dẫn, nhưng nhắn tin không qua trung gian cũng có những nhược điểm đáng kể:

* Các dịch vụ cần biết về vị trí của nhau và do đó phải sử dụng một trong các cơ chế khám phá được mô tả trước đó trong phần 3.2.4.
* Nó cung cấp khả năng sẵn có giảm, bởi vìcả người gửi và người nhận tin nhắn đều phải có mặt khi tin nhắn đang được trao đổi.
* Cơ chế thực hiện,chẳng hạn như đảm bảo giao hàng, thì khó khăn hơn.

Trên thực tế, một số nhược điểm này, chẳng hạn như tính khả dụng giảm và nhu cầu khám phá dịch vụ, cũng giống như khi sử dụng phản hồi/phản hồi đồng bộ.

Do những hạn chế này, hầu hết các ứng dụng doanh nghiệp đều sử dụng kiến ​​trúc dựa trên môi giới tin nhắn. Hãy cùng xem cách thức hoạt động của nó.

**ỒTỔNG QUANCỦA NGƯỜI MÔI GIỚI-TIN NHẮN DỰA TRÊN**

Message broker là một trung gian mà tất cả các tin nhắn đều được truyền qua. Người gửi viết tin nhắn cho message broker, và message broker chuyển tin nhắn đó đến người nhận. Một lợi ích quan trọng khi sử dụng message broker là người gửi không cần biết vị trí mạng của người dùng. Một lợi ích khác là message broker sẽ lưu trữ tin nhắn cho đến khi người dùng có thể xử lý chúng.

Có nhiều message broker để lựa chọn. Ví dụ về các message broker mã nguồn mở phổ biến bao gồm:

* + - * Hoạt độngMQ ([http://activemq.apache.org](http://activemq.apache.org/))
      * ThỏMQ ([https://www.rabbitmq.com](https://www.rabbitmq.com/))
      * Apache Kafka ([http://kafka.apache.org](http://kafka.apache.org/))

Ngoài ra còn có các dịch vụ nhắn tin dựa trên đám mây, chẳng hạn như AWS Kinesis ([https://aws.amazon](https://aws.amazon.com/kinesis/)

[.com/kinesis/](https://aws.amazon.com/kinesis/)) và AWS SQS (<https://aws.amazon.com/sqs/>).

Khi lựa chọn nhà môi giới tin nhắn, bạn cần cân nhắc nhiều yếu tố, bao gồm:

* + - * *Ngôn ngữ lập trình được hỗ trợ*—Có lẽ bạn nên chọn mộthỗ trợ nhiều ngôn ngữ lập trình khác nhau.
      * *Tiêu chuẩn nhắn tin được hỗ trợ*—Liệu trình môi giới tin nhắn có hỗ trợ bất kỳ tiêu chuẩn nào không, chẳng hạn như AMQP và STOMP, hay nó là độc quyền?
      * *Đặt hàng tin nhắn*—Liệu trình môi giới tin nhắn có duy trì thứ tự của tin nhắn không?
      * *Đảm bảo giao hàng*—Người môi giới đưa ra những đảm bảo giao hàng nào?
      * *Sự kiên trì*—Tin nhắn có được lưu vào đĩa và có thể tồn tại sau khi môi giới bị sập không?
      * *Độ bền*—Nếu người dùng kết nối lại với nhà môi giới tin nhắn, liệu họ có nhận được các tin nhắn đã được gửi khi bị ngắt kết nối không?
      * *Khả năng mở rộng*—Mức độ mở rộng của trình môi giới tin nhắn là bao nhiêu?
      * *Độ trễ*—Độ trễ đầu cuối là bao nhiêu?
      * *Người tiêu dùng cạnh tranh*—Liệu nhà môi giới tin nhắn có hỗ trợ người tiêu dùng cạnh tranh không?

Mỗi nhà môi giới tạo ra những điều khác nhausự đánh đổi. Ví dụ, một broker có độ trễ rất thấp có thể không bảo toàn thứ tự, không đảm bảo gửi tin nhắn và chỉ lưu trữ tin nhắn trong bộ nhớ. Một broker tin nhắn đảm bảo gửi tin nhắn và lưu trữ tin nhắn đáng tin cậy trên đĩa có thể có độ trễ cao hơn. Loại broker tin nhắn nào phù hợp nhất tùy thuộc vào yêu cầu của ứng dụng của bạn. Thậm chí có thể các phần khác nhau của ứng dụng của bạn sẽ có các yêu cầu nhắn tin khác nhau.

Tuy nhiên, có khả năng là thứ tự tin nhắn và khả năng mở rộng là điều cần thiết. Bây giờ chúng ta hãy xem cách triển khai các kênh tin nhắn bằng cách sử dụng một môi giới tin nhắn.

**TÔITRIỂN KHAI KÊNH TIN NHẮN BẰNG MESSAGE BROKER**

Mỗi message broker triển khai khái niệm kênh message theo một cách khác nhau. Như bảng 3.2 cho thấy, các message broker JMS như ActiveMQ có hàng đợi và chủ đề. Các message broker dựa trên AMQP như RabbitMQ có các exchange và hàng đợi. Apache Kafka có các chủ đề, AWS Kinesis có các luồng và AWS SQS có các hàng đợi. Hơn nữa, một số message broker cung cấp khả năng nhắn tin linh hoạt hơn so với sự trừu tượng hóa message và channels được mô tả trong chương này.

**Bảng 3.2 Mỗi nhà môi giới tin nhắn triển khai khái niệm kênh tin nhắn theo một cách khác nhau.**

|  |  |  |
| --- | --- | --- |
| **Người môi giới tin nhắn** | **Kênh điểm-điểm** | **Kênh đăng ký-xuất bản** |
| JMS | Hàng đợi | Đề tài |
| Apache Kafka | Đề tài | Đề tài |
| Các nhà môi giới dựa trên AMQP, chẳng hạn như RabbitMQ | Trao đổi + Hàng đợi | Trao đổi Fanout và mộthàng đợi cho mỗi người tiêu dùng |
| AWS Kinesis | Suối | Suối |
| AWS SQS | Hàng đợi | — |

Hầu như tất cả các nhà môi giới tin nhắn được mô tả ở đây đều hỗ trợ cả điểm tới điểm và xuất bản.đăng ký kênh. Ngoại lệ duy nhất là AWS SQS, chỉ hỗ trợ các kênh điểm-đến-điểm.

Bây giờ chúng ta hãy xem xét những lợi ích và hạn chế của nhắn tin dựa trên môi giới.

**BLỢI ÍCH VÀ NHƯỢC ĐIỂM CỦA BROKER-TIN NHẮN DỰA TRÊN**

Có nhiều lợi thế khi sử dụng môi giớinhắn tin:

* + - * + *Khớp nối lỏng lẻo*—Một máy khách thực hiện yêu cầu bằng cách chỉ cần gửi tin nhắn đến kênh thích hợp. Máy khách hoàn toàn không biết về các trường hợp dịch vụ. Nó không cần sử dụng cơ chế khám phá để xác định vị trí của một trường hợp dịch vụ.
        + *Bộ đệm tin nhắn*—Môi giới tin nhắn đệm tin nhắn cho đến khi chúng có thể được xử lý. Với giao thức yêu cầu/phản hồi đồng bộ như HTTP, cả máy khách và dịch vụ phải khả dụng trong suốt thời gian trao đổi. Tuy nhiên, với tin nhắn, tin nhắn sẽ xếp hàng cho đến khi chúng có thể được người tiêu dùng xử lý. Điều này có nghĩa là, ví dụ, một cửa hàng trực tuyến có thể chấp nhận đơn đặt hàng từ khách hàng ngay cả khi hệ thống thực hiện đơn hàng chậm hoặc không khả dụng. Các tin nhắn sẽ chỉ xếp hàng cho đến khi chúng có thể được xử lý.
        + *Giao tiếp linh hoạt*—Nhắn tin hỗ trợ tất cả các kiểu tương tác được mô tả trước đó.
        + *Giao tiếp giữa các tiến trình rõ ràng*—Cơ chế dựa trên RPCcố gắng làm cho việc gọi một dịch vụ từ xa trông giống như việc gọi một dịch vụ cục bộ. Nhưng do các định luật vật lý và khả năng xảy ra lỗi một phần, trên thực tế chúng khá khác nhau.

Việc nhắn tin làm cho những khác biệt này trở nên rất rõ ràng, do đó các nhà phát triển sẽ không bị ru ngủ trong cảm giác an toàn sai lầm.

Có một số nhược điểmsử dụng tin nhắn:

* + - * *Nút thắt hiệu suất tiềm ẩn*—Córủi ro là message broker có thể là nút thắt cổ chai về hiệu suất. May mắn thay, nhiều message broker hiện đại được thiết kế để có khả năng mở rộng cao.
      * *Điểm lỗi đơn lẻ tiềm ẩn*—Điều cần thiết là message broker phải có tính khả dụng cao—nếu không, độ tin cậy của hệ thống sẽ bị ảnh hưởng. May mắn thay, hầu hết các broker hiện đại đều được thiết kế để có tính khả dụng cao.
      * *Sự phức tạp trong hoạt động bổ sung*—Hệ thống nhắn tin là một thành phần hệ thống khác phải được cài đặt, cấu hình và vận hành.

Hãy cùng xem xét một số vấn đề thiết kế mà bạn có thể gặp phải.

###### Người nhận cạnh tranh và thứ tự tin nhắn

Một thách thức là làm thế nào để mở rộng các bộ thu tin nhắn trong khi vẫn giữ nguyên thứ tự tin nhắn. Yêu cầu chung là phải có nhiều phiên bản của một dịch vụ để xử lý tin nhắn đồng thời. Hơn nữa, ngay cả một phiên bản dịch vụ duy nhất cũng có thể sử dụng luồng để xử lý đồng thời nhiều tin nhắn. Sử dụng nhiều luồng và phiên bản dịch vụ để xử lý đồng thời tin nhắn sẽ làm tăng thông lượng của ứng dụng. Nhưng thách thức với việc xử lý tin nhắn đồng thời là đảm bảo rằng mỗi tin nhắn được xử lý một lần và theo thứ tự.

Ví dụ, hãy tưởng tượng rằng có ba trường hợp dịch vụ đọc từcùng một điểm tới điểmkênh và người gửi sẽ xuất bản các thông báo sự kiện Order Created, Order Updated và Order Cancelled theo trình tự. Một triển khai nhắn tin đơn giản có thể đồng thời gửi từng thông báo đến một người nhận khác nhau. Do sự chậm trễ do sự cố mạng hoặc thu gom rác, các thông báo có thể được xử lý không theo thứ tự, dẫn đến hành vi lạ. Về mặt lý thuyết, một phiên bản dịch vụ có thể xử lý thông báo Order Cancelled trước khi một dịch vụ khác xử lý thông báo Order Created! Một giải pháp phổ biến, được sử dụng bởi các nhà môi giới tin nhắn hiện đại như Apache Kafka và AWS Kinesis, là sử dụng các kênh được phân mảnh (phân vùng). Hình 3.11 cho thấy cách thức hoạt động của giải pháp này.

Giải pháp này gồm có ba phần:

**1** Một kênh phân mảnh bao gồm hai hoặc nhiều phân mảnh, mỗi phân mảnh hoạt động như một kênh.

**2**Người gửi chỉ định một khóa phân mảnh trong tiêu đề của tin nhắn, thường là một chuỗi hoặc chuỗi byte tùy ý. Người môi giới tin nhắn sử dụng khóa phân mảnh để gán tin nhắn cho một phân mảnh/phân vùng cụ thể. Ví dụ, nó có thể chọn phân mảnh bằng cách tính toán hàm băm của khóa phân mảnh theo số lượng phân mảnh.

**3**Trình môi giới tin nhắn nhóm nhiều phiên bản của một máy thu lại với nhau và coi chúng như cùng một máy thu logic. Ví dụ, Apache Kafka sử dụng thuật ngữ nhóm người tiêu dùng. Trình môi giới tin nhắn gán mỗi phân đoạn cho một máy thu duy nhất. Nó gán lại các phân đoạn khi máy thu khởi động và tắt.



**Các tuyến đường dựa trên hàm băm của khóa phân đoạn**

Bộ thu logic A

Tạo đơn hàng

lời yêu cầu

Kênh

Khóa phân mảnh:orderId

Người gửi

Phân công mảnh vỡ

Mảnh vỡ 1

Bộ định tuyến

Người nhận

Người nhận

...

Mảnh vỡ ...

Người nhận A thể hiện2

Mảnh vỡ 0

Người nhận A thể hiện1

**Hình 3.11 Mở rộng quy mô người tiêu dùng trong khi vẫn giữ nguyên thứ tự tin nhắn bằng cách sử dụng kênh tin nhắn được phân mảnh (phân vùng). Người gửi bao gồm khóa phân mảnh trong tin nhắn. Người môi giới tin nhắn ghi tin nhắn vào một phân mảnh được xác định bởi khóa phân mảnh. Người môi giới tin nhắn gán từng phân vùng cho một phiên bản của người nhận được sao chép.**

Trong ví dụ này, mỗi thông báo sự kiện Order có orderId là khóa phân đoạn của nó. Mỗi sự kiện cho một đơn hàng cụ thể được xuất bản đến cùng một phân đoạn, được đọc bởi một người dùng duy nhấtVí dụ. Do đó, các tin nhắn này được đảm bảo sẽ được xử lý theo thứ tự.

###### Xử lý tin nhắn trùng lặp

Một thách thức khác mà bạn phải giải quyết khi sử dụng tin nhắn là xử lý các tin nhắn trùng lặp. Một nhà môi giới tin nhắn lý tưởng nhất là chỉ gửi mỗi tin nhắn một lần, nhưng đảm bảo tin nhắn chính xác một lần thường quá tốn kém. Thay vào đó, hầu hết các nhà môi giới tin nhắn hứa sẽ gửi một tin nhắn ít nhất một lần.

Khi hệ thống hoạt động bình thường, một message broker đảm bảo việc phân phối ít nhất một lần sẽ chỉ phân phối mỗi tin nhắn một lần. Nhưng lỗi của máy khách, mạng hoặc message broker có thể dẫn đến việc một tin nhắn được phân phối nhiều lần. Giả sử một máy khách gặp sự cố sau khi xử lý tin nhắn và cập nhật cơ sở dữ liệu của nó—nhưng trước khi xác nhận tin nhắn. Message broker sẽ phân phối lại tin nhắn chưa được xác nhận, hoặc cho máy khách đó khi nó khởi động lại hoặc cho một bản sao khác của máy khách.

Lý tưởng nhất là bạn nên sử dụng một nhà môi giới tin nhắn có thể duy trì thứ tự khi phân phối lại.tin nhắn. Hãy tưởng tượng rằng khách hàng xử lý mộtĐã tạo đơn hàngsự kiện theo sau bởi mộtĐơn hàng đã hủysự kiện cho cùng mộtĐặt hàngvà bằng cách nào đóĐã tạo đơn hàngsự kiện không được xác nhận. Nhà môi giới tin nhắn nên chuyển lại cả haiĐặt hàng Cre-đã ănVàĐơn hàng đã hủysự kiện. Nếu nó chỉ cung cấp lạiĐã tạo đơn hàng, khách hàng có thể hoàn tác việc hủy bỏĐặt hàng.

Có một số cách khác nhau để xử lý tin nhắn trùng lặp:

* + - * Viết trình xử lý tin nhắn có tính bất biến.
      * Theo dõi tin nhắn và loại bỏ tin nhắn trùng lặp. Hãy cùng xem xét từng tùy chọn.

**TVIẾTTRÌNH XỬ LÝ TIN NHẮN BẤT CƯỜNG**

Nếu logic ứng dụng xử lý tin nhắn là idempotent, thì các tin nhắn trùng lặp là vô hại. Logic ứng dụng là idempotent nếu việc gọi nó nhiều lần với cùng các giá trị đầu vào không có tác dụng bổ sung nào. Ví dụ, việc hủy một đơn hàng đã hủy là một hoạt động idempotent. Tương tự như vậy là việc tạo một đơn hàng với ID do khách hàng cung cấp. Một trình xử lý tin nhắn idempotent có thể được thực hiện an toàn nhiều lần, với điều kiện là trình môi giới tin nhắn duy trì thứ tự khi phân phối lại tin nhắn.

Thật không may, logic ứng dụng thường không phải là idempotent. Hoặc bạn có thể đang sử dụng một message broker không giữ nguyên thứ tự khi gửi lại tin nhắn. Tin nhắn trùng lặp hoặc không theo thứ tự có thể gây ra lỗi. Trong trường hợp này, bạn phải viết trình xử lý tin nhắn theo dõi tin nhắn và loại bỏ tin nhắn trùng lặp.

**TSỬA CHỮA TIN NHẮN VÀ LOẠI BỎ CÁC TIN NHẮN TRÙNG LẶP**

Ví dụ, hãy xem xét một trình xử lý tin nhắn cho phép thẻ tín dụng của người tiêu dùng. Nó phải cho phép thẻ đúng một lần cho mỗi đơn hàng. Ví dụ về logic ứng dụng này có hiệu ứng khác nhau mỗi lần được gọi. Nếu các tin nhắn trùng lặp khiến trình xử lý tin nhắn thực thi logic này nhiều lần, ứng dụng sẽ hoạt động không chính xác. Trình xử lý tin nhắn thực thi loại logic ứng dụng này phải trở nên idem-power bằng cách phát hiện và loại bỏ các tin nhắn trùng lặp.

Một giải pháp đơn giản là người dùng tin nhắn theo dõi các tin nhắn đã xử lý bằng cách sử dụng id tin nhắn và loại bỏ bất kỳ bản sao nào. Ví dụ, nó có thể lưu trữ id tin nhắn của mỗi tin nhắn mà nó đã sử dụng trong một bảng cơ sở dữ liệu. Hình 3.12 cho thấy cách thực hiện việc này bằng cách sử dụng một bảng chuyên dụng.

**INSERT sẽ không thành công đối với các tin nhắn trùng lặp.**

Giao dịch

TIN\_NHẮN\_ĐÃ\_XỬ\_LÝbàn

CHÈN

Bảng ứng dụng

CẬP NHẬT

...

...

xyz

Mã số MSG



Tin nhắn

Mã số: xyz

Người tiêu dùng

**Hình 3.12 Người dùng phát hiện và loại bỏ các tin nhắn trùng lặp bằng cách ghi lại ID của các tin nhắn đã xử lý trong bảng cơ sở dữ liệu. Nếu một tin nhắn đã được xử lý trước đó, INSERT vào bảng PROCESSED\_MESSAGES sẽ không thành công.**

Khi người tiêu dùng xử lý một tin nhắn, nó sẽ ghi lại tin nhắn đóid trong bảng cơ sở dữ liệu như một phần của giao dịch tạo và cập nhật các thực thể kinh doanh. Trong ví dụ này,người tiêu dùng chèn một hàng chứa id tin nhắn vào bảng PROCESSED\_MESSAGES. Nếu mộttin nhắn trùng lặp, lệnh INSERT sẽ không thành công và người dùng có thể hủy tin nhắn.

Một tùy chọn khác là trình xử lý tin nhắn ghi lại id tin nhắn trong bảng ứng dụng thay vì bảng chuyên dụng. Cách tiếp cận này đặc biệt hữu ích khi sử dụng cơ sở dữ liệu NoSQL có mô hình giao dịch hạn chế, do đó không hỗ trợ cập nhật hai bảng như một phần của giao dịch cơ sở dữ liệu. Chương 7 trình bày một ví dụ về cách tiếp cận này.

###### Tin nhắn giao dịch

Một dịch vụ thường cần phải xuất bản các thông báo như một phần của giao dịch cập nhật cơ sở dữ liệu. Ví dụ, trong suốt cuốn sách này, bạn sẽ thấy các ví dụ về các dịch vụ xuất bản các sự kiện miền bất cứ khi nào chúng tạo hoặc cập nhật các thực thể kinh doanh. Cả việc cập nhật cơ sở dữ liệu và việc gửi thông báo phải diễn ra trong một giao dịch. Nếu không, một dịch vụ có thể cập nhật cơ sở dữ liệu rồi sập, ví dụ, trước khi gửi thông báo. Nếu dịch vụ không thực hiện hai hoạt động này một cách nguyên tử, lỗi có thể khiến hệ thống ở trạng thái không nhất quán.

Giải pháp truyền thống là sử dụng giao dịch phân tán trải dài trên cơ sở dữ liệu và message broker. Nhưng như bạn sẽ tìm hiểu trong chương 4, giao dịch phân tán không phải là lựa chọn tốt cho các ứng dụng hiện đại. Hơn nữa, nhiều broker hiện đại như Apache Kafka không hỗ trợ giao dịch phân tán.

Do đó, ứng dụng phải sử dụng một cơ chế khác để xuất bản tin nhắn một cách đáng tin cậy. Hãy cùng xem cách thức hoạt động của nó.

**BạnHÁT MỘT BẢNG CƠ SỞ DỮ LIỆU NHƯ MỘT HÀNG ĐỢI TIN NHẮN**

Hãy tưởng tượng rằng ứng dụng của bạn đang sử dụng cơ sở dữ liệu quan hệ. Một cách đơn giản để xuất bản tin nhắn một cách đáng tin cậy là áp dụng mẫu Transactional outbox. Mẫu này sử dụng bảng cơ sở dữ liệu làm hàng đợi tin nhắn tạm thời. Như hình 3.13 cho thấy, một dịch vụ gửi tin nhắn có bảng cơ sở dữ liệu OUTBOX. Là một phần của cơ sở dữ liệu

|  |  |  |
| --- | --- | --- |
|  |  |  |

Xuất bản



Giao dịch

CHÈN, CẬP NHẬT, XÓA

CHÈN

ĐẶT HÀNGbàn HỘP THƯ RAbàn

Đọc

HỘP THƯ RA

bàn

Cơ sở dữ liệu

...

...

Dịch vụ đặt hàng



Tin nhắntiếp sức

Tin nhắnngười môi giới

**Hình 3.13 Một dịch vụ đáng tin cậy xuất bản một thông điệp bằng cách chèn nó vào mộtHỘP THƯ RAbảng như một phần của giao dịch cập nhật cơ sở dữ liệu.Chuyển tiếp tin nhắnđọcHỘP THƯ RAbảng và xuất bản các thông điệp tới mộtngười môi giới tin nhắn.**

giao dịch tạo, cập nhật và xóa các đối tượng kinh doanh, dịch vụ gửi tin nhắn bằng cách chèn chúng vào bảng OUTBOX. Tính nguyên tử được đảm bảo vì đây là giao dịch ACID cục bộ.

Bảng OUTBOX hoạt động như một hàng đợi tin nhắn tạm thời. MessageRelay là một thành phần đọc bảng OUTBOX và xuất bản tin nhắn đến một nhà môi giới tin nhắn.

**Mẫu: Hộp thư đi giao dịch**

Xuất bản một sự kiện hoặc tin nhắn như một phần của giao dịch cơ sở dữ liệu bằng cách lưu nó trong mộtOUT-BOXtrong cơ sở dữ liệu. Xem[http://microservices.io/patterns/data/transactional-out-](http://microservices.io/patterns/data/transactional-outbox.html) [hộp.html](http://microservices.io/patterns/data/transactional-outbox.html).

Bạn có thể sử dụng một cách tiếp cận tương tự với một số cơ sở dữ liệu NoSQL. Mỗi thực thể kinh doanh được lưu trữ dưới dạng bản ghi trong cơ sở dữ liệu có một thuộc tính là danh sách các thông báo cần được công bố. Khi một dịch vụ cập nhật một thực thể trong cơ sở dữ liệu, nó sẽ thêm một thông báo vào danh sách đó. Đây là nguyên tử vì nó được thực hiện bằng một hoạt động cơ sở dữ liệu duy nhất. Tuy nhiên, thách thức là tìm hiệu quả các thực thể kinh doanh có sự kiện và công bố chúng.

Có một vài khác nhaucách di chuyển tin nhắn từ cơ sở dữ liệu đến nhà môi giới tin nhắn. Chúng ta sẽ xem xét từng cách.

**PXUẤT BẢN SỰ KIỆN BẰNG CÁCH SỬ DỤNGPMẪU NHÀ XUẤT BẢN OLLING**

Nếu ứng dụng sử dụng cơ sở dữ liệu quan hệ, một cách rất đơn giản để xuất bản tin nhắnđược chèn vào bảng OUTBOX là để MessageRelay thăm dò bảng để tìm các tin nhắn chưa được công bố. Nó định kỳ truy vấn bảng:

CHỌN \* TỪ HỘP THƯ ĐI ĐƯỢC ĐẶT HÀNG BỞI ... TĂNG DẦN

Tiếp theo, MessageRelayxuất bản những thông điệp đó cho message broker, gửi một thông điệp đến kênh thông điệp đích của nó. Cuối cùng, nó xóa những thông điệp đó khỏi bảng OUTBOX:

BẮT ĐẦU

XÓA TỪ HỘP THƯ ĐI NƠI ID trong(.)

LÀM

**Mẫu: Nhà xuất bản thăm dò ý kiến**

Xuất bản tin nhắn bằng cách thăm dò hộp thư đi trong cơ sở dữ liệu. Xem[http://microser-](http://microservices.io/patterns/data/polling-publisher.html) [Vices.io/patterns/data/polling-publisher.html](http://microservices.io/patterns/data/polling-publisher.html).

Thăm dò cơ sở dữ liệu là một cách tiếp cận đơn giản, hoạt động khá tốt ở quy mô nhỏ. Nhược điểm là việc thăm dò cơ sở dữ liệu thường xuyên có thể tốn kém. Ngoài ra, việc bạn có thể sử dụng cách tiếp cận này với cơ sở dữ liệu NoSQL hay không phụ thuộc vào khả năng truy vấn của nó. Đó là vì thay vì truy vấn bảng OUTBOX, ứng dụng phải truy vấn

các thực thể kinh doanh và có thể hoặc không thể thực hiện hiệu quả. Do những nhược điểm và hạn chế này, thường thì tốt hơn—và trong một số trường hợp là cần thiết—sử dụng phương pháp tinh vi và hiệu quả hơn là theo dõi nhật ký giao dịch cơ sở dữ liệu.

**PSỰ KIỆN ĐƯỢC XUẤT BẢN BẰNG CÁCH ÁP DỤNGTMẪU ĐUÔI GỖ TẠO RANSACTION**

Một giải pháp tinh vi là MessageRelay theo dõi nhật ký giao dịch cơ sở dữ liệu (còn gọi là nhật ký cam kết). Mỗi bản cập nhật đã cam kết do ứng dụng thực hiện được biểu diễn dưới dạng một mục trong nhật ký giao dịch của cơ sở dữ liệu. Một trình khai thác nhật ký giao dịch có thể đọc nhật ký giao dịch và công bố mỗi thay đổi dưới dạng một thông báo cho trình môi giới thông báo. Hình 3.14 cho thấy cách tiếp cận này hoạt động như thế nào.

Xuất bản

CHÈN VÀO HỘP THƯ ĐI ...

Cơ sở dữ liệu

Thay đổi

Nhật ký giao dịchthợ mỏ

Nhật ký giao dịch

Dịch vụ đặt hàng

HỘP THƯ RAbàn

Tin nhắnngười môi giới

**Đã cam kết chèn vào**

**bảng OUTBOX được ghi lại trong nhật ký giao dịch của cơ sở dữ liệu.**

**Đọc nhật ký giao dịch**

**Hình 3.14 Một dịch vụ xuất bản các tin nhắn được chèn vàoHỘP THƯ RAbảng theo khai thácnhật ký giao dịch của cơ sở dữ liệu.**

Transaction Log Miner đọc các mục nhập nhật ký giao dịch. Nó chuyển đổi từng mục nhập nhật ký có liên quan tương ứng với một thông điệp được chèn vào thành một thông điệp và xuất bản thông điệp đó cho message broker. Phương pháp này có thể được sử dụng để xuất bản các thông điệp được ghi vào bảng OUTBOX trong RDBMS hoặc các thông điệp được thêm vào các bản ghi trong cơ sở dữ liệu NoSQL.

**Mẫu: Theo dõi nhật ký giao dịch**

Xuất bản những thay đổi đã thực hiệnvào cơ sở dữ liệu bằng cách theo dõi nhật ký giao dịch. Xem[http://micro-](http://microservices.io/patterns/data/transaction-log-tailing.html) [services.io/patterns/data/transaction-log-tailing.html](http://microservices.io/patterns/data/transaction-log-tailing.html).

Có một số ví dụ về cách tiếp cận này đang được sử dụng:

* *Debezi*([http://debezium.io](http://debezium.io/))—Một dự án nguồn mở công bố những thay đổi trong cơ sở dữ liệu tới trình môi giới tin nhắn Apache Kafka.
* *Dữ liệu LinkedIn*(<https://github.com/linkedin/databus>)—Một dự án nguồn mởect khai thác nhật ký giao dịch Oracle và công bố các thay đổi dưới dạng sự kiện. LinkedIn sử dụng Databus để đồng bộ hóa nhiều kho dữ liệu phái sinh với hệ thống lưu trữ.
* *Luồng DynamoDB*([http://docs.aws.amazon.com/amazondynamodb/latest/](http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html) [hướng dẫn dành cho nhà phát triển/Streams.html](http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html))—Luồng DynamoDB chứa chuỗi thay đổi theo thứ tự thời gian (tạo, cập nhật và xóa) được thực hiện đối với các mục trong bảng DynamoDB trong 24 giờ qua. Ứng dụng có thể đọc những thay đổi đó từ luồng và, ví dụ, xuất bản chúng dưới dạng sự kiện.
* *Xe điện Eventuate*(<https://github.com/eventuate-tram/eventuate-tram-core>)-Của bạnthư viện nhắn tin giao dịch nguồn mở của riêng tác giả sử dụng giao thức binlog MySQL, Postgres WAL hoặc thăm dò để đọc các thay đổi được thực hiện cho mộtHỘP THƯ RAvà xuất bản chúng lên Apache Kafka.

Mặc dù cách tiếp cận này khá mơ hồ, nhưng nó hoạt động rất tốt. Thách thức là việc triển khai nó đòi hỏi một số nỗ lực phát triển. Ví dụ, bạn có thể viết mã cấp thấp gọi các API dành riêng cho cơ sở dữ liệu. Ngoài ra, bạn có thể sử dụng một khuôn khổ nguồn mở như Debezium để xuất bản các thay đổi do ứng dụng thực hiện đối với MySQL, Postgres hoặc MongoDB đối với Apache Kafka. Nhược điểm khi sử dụng Debezium là trọng tâm của nó là nắm bắt các thay đổi ở cấp cơ sở dữ liệu và các API để gửi và nhận tin nhắn nằm ngoài phạm vi của nó. Đó là lý do tại sao tôi đã tạo ra khuôn khổ Eventuate Tram, cung cấp các API nhắn tin cũng như theo dõi và thăm dò giao dịch.

###### Thư viện và khuôn khổ cho nhắn tin

Một dịch vụ cần sử dụng thư viện để gửi và nhận tin nhắn. Một cách tiếp cận là sử dụng thư viện máy khách của message broker, mặc dù có một số vấn đề khi sử dụng trực tiếp thư viện như vậy:

* + - * Thư viện khách hàng kết hợp logic kinh doanh để xuất bản tin nhắn tới tin nhắnAPI môi giới.
      * Thư viện máy khách của message broker thường ở cấp độ thấp và yêu cầu nhiều dòng mã để gửi hoặc nhận tin nhắn. Là một nhà phát triển, bạn không muốn viết mã boilerplate nhiều lần. Ngoài ra, với tư cách là tác giả của cuốn sách này, tôi không muốn mã ví dụ lộn xộn với boilerplate cấp độ thấp.
      * Thư viện máy khách thường chỉ cung cấp cơ chế cơ bản để gửi và nhận tin nhắn và không hỗ trợ các kiểu tương tác cấp cao hơn.

Một cách tiếp cận tốt hơn là sử dụng một thư viện hoặc khuôn khổ cấp cao hơn ẩn các chi tiết cấp thấp và hỗ trợ trực tiếp các kiểu tương tác cấp cao hơn. Để đơn giản, các ví dụ trong cuốn sách này sử dụng khuôn khổ Eventuate Tram của tôi. Nó có một API đơn giản, dễ hiểu giúp ẩn đi sự phức tạp khi sử dụng message broker. Bên cạnh một API

để gửi và nhận tin nhắn, Eventuate Tram cũng hỗ trợ các kiểu tương tác cấp cao hơn như yêu cầu/phản hồi không đồng bộ và xuất bản sự kiện miền.

**Cái gì cơ!? Tại sao lại là khung Eventuate?**

Các mẫu mã trong cuốn sách này sử dụng các khuôn khổ Eventuate nguồn mở mà tôi đã phát triểnoped cho giao dịch nhắn tin, sự kiện tìm nguồn và sagas. Tôi chọn sử dụng các khuôn khổ của mình vì, không giống như với, chẳng hạn, tiêm phụ thuộc và khuôn khổ Spring, không có khuôn khổ nào được áp dụng rộng rãi cho nhiều tính năng mà kiến ​​trúc microservice yêu cầu. Nếu không có khuôn khổ Eventuate Tram, nhiều ví dụ sẽ phải sử dụng trực tiếp các API nhắn tin cấp thấp, khiến chúng phức tạp hơn nhiều và làm lu mờ các khái niệm quan trọng. Hoặc họ sẽ sử dụng một khuôn khổ không được áp dụng rộng rãi, điều này cũng sẽ gây ra sự chỉ trích.

Thay vào đó, các ví dụ sử dụng các khung Eventuate Tram, có API đơn giản, dễ hiểu, ẩn các chi tiết triển khai. Bạn có thể sử dụng các khung này trong ứng dụng của mình. Ngoài ra, bạn có thể nghiên cứu các khung Eventuate Tram và tự triển khai lại các khái niệm.

Eventuate Tram cũng thực hiệnhai cơ chế quan trọng:

* + - * + *Tin nhắn giao dịch*—Nó xuất bản các thông điệp như là một phần của giao dịch cơ sở dữ liệu.
        + *Phát hiện tin nhắn trùng lặp*—CácNgười dùng tin nhắn Eventuate Tram phát hiện và loại bỏ các tin nhắn trùng lặp, điều này rất cần thiết để đảm bảo rằng người dùng xử lý tin nhắn đúng một lần, như đã thảo luận trong phần 3.3.6.

Chúng ta hãy cùng xem xét EventuateAPI xe điện.

**BTIN NHẮN ASIC**

API nhắn tin cơ bản bao gồm hai giao diện Java:Nhà sản xuất tin nhắnVàTin nhắn-Người tiêu dùng. Một dịch vụ sản xuất sử dụngNhà sản xuất tin nhắngiao diện để xuất bản tin nhắn đến các kênh tin nhắn. Sau đây là ví dụ về việc sử dụng giao diện này:

Nhà sản xuất tin nhắnmessageProducer = ...; Chuỗi kênh = ...;

Tải trọng chuỗi = ...;

messageProducer.send(đích đến, MessageBuilder.withPayload(tải trọng).build())

Một dịch vụ tiêu dùng sử dụngTin nhắnNgười tiêu dùngGiao diện để đăng ký nhận tin nhắn:

MessageConsumer messageConsumer;

messageConsumer.subscribe(subscriberId,Collections.singleton(đích), tin nhắn -> { ... })

Nhà sản xuất tin nhắnVàTin nhắnNgười tiêu dùnglà nền tảng của cái cấp độ cao hơn API cho yêu cầu/phản hồi không đồng bộ và xuất bản sự kiện miền.

Chúng ta hãy nói về cách đăng tải và đăng ký sự kiện.

**DXUẤT BẢN SỰ KIỆN OMAIN**

Eventuate Tram có API để xuất bản và sử dụng các sự kiện miền. Chương 5 giải thích rằng các sự kiện miền là các sự kiện được phát ra bởi một tổng hợp (đối tượng kinh doanh) khi nó được tạo, cập nhật hoặc xóa. Một dịch vụ xuất bản một sự kiện miềnsử dụng giao diện DomainEventPublisher. Đây là một ví dụ:

Nhà xuất bản sự kiện miềndomainEventPublisher; Chuỗi accountId = ...;

DomainEvent domainEvent = new AccountDebited(...);

domainEventPublisher.publish("Tài khoản",accountId, Collections.singletonList(domainEvent));

Một dịch vụ sử dụng các sự kiện miền bằng cách sử dụngcáiTrình phân phối sự kiện miền. Sau đây là một ví dụ:

DomainEventHandlers domainEventHandlers = DomainEventHandlersBuilder

.forAggregateType("Đặt hàng")

.onEvent(AccountDebited.class, domainEvent -> { ... })

.xây dựng();

DomainEventDispatcher mới ("eventDispatcherId",trình xử lý sự kiện miền,tin nhắnNgười tiêu dùng);

Sự kiện không phải là mẫu nhắn tin cấp cao duy nhất được Eventuate Tram hỗ trợ. Nó cũng hỗ trợ nhắn tin dựa trên lệnh/trả lời.

**COMAND/RTỪNG-TIN NHẮN DỰA TRÊN**

Một máy khách có thể gửi một thông điệp lệnh đến một dịch vụ bằng cách sử dụng giao diện CommandProducer. Ví dụ

LệnhProducer lệnhProducer = ...;

Bản đồ<Chuỗi, Chuỗi> extraMessageHeaders = Collections.emptyMap(); Chuỗi commandId = commandProducer.send("CustomerCommandChannel",

DoSomethingCommand() mới,

"Trả lời kênh",extraMessageHeaders);

Một dịch vụ sử dụng các thông điệp lệnh bằng cách sử dụngLệnhDispatcherlớp học.Chỉ huy-Điều phối viênsử dụngTin nhắnNgười tiêu dùnggiao diện để đăng ký các sự kiện được chỉ định. Nó dis-vá từng thông báo lệnh vào phương thức xử lý thích hợp. Sau đây là một ví dụ:

CommandHandlers commandHandlers =CommandHandlersBuilder

.fromChannel(lệnhKênh)

.onMessage(DoSomethingCommand.class, (lệnh) -

> { ... ; trả về vớiThành công(); })

.xây dựng();

Bộ điều phối CommandDispatcher = new CommandDispatcher("subscribeId", commandHandlers, messageConsumer, messageProducer);

Trong suốt cuốn sách này, bạn sẽ thấy các ví dụ mã sử dụng các API này để gửi và nhận tin nhắn.

Như bạn đã thấy, khuôn khổ Eventuate Tramtriển khai nhắn tin giao dịch cho các ứng dụng Java. Nó cung cấp API cấp thấp để gửi và nhận tin nhắn theo giao dịch. Nó cũng cung cấp API cấp cao hơn để xuất bản và sử dụng các sự kiện miền và để gửi và xử lý lệnh.

Bây giờ chúng ta hãy xem xét phương pháp thiết kế dịch vụ sử dụng nhắn tin không đồng bộ để cải thiện tính khả dụng.

#### Sử dụng nhắn tin không đồng bộ để cải thiện tính khả dụng

Như bạn đã thấy, mộtnhiều cơ chế IPC có những sự đánh đổi khác nhau. Một sự đánh đổi cụ thể là cách lựa chọn cơ chế IPC của bạn tác động đến tính khả dụng. Trong phần này, bạn sẽ tìm hiểu rằng giao tiếp đồng bộ với các dịch vụ khác như một phần của việc xử lý yêu cầu làm giảm tính khả dụng của ứng dụng. Do đó, bạn nên thiết kế các dịch vụ của mình để sử dụng nhắn tin không đồng bộ bất cứ khi nào có thể.

Trước tiên chúng ta hãy xem xét vấn đề liên quan đến giao tiếp đồng bộ và cách nó tác động đến tính khả dụng.

###### Giao tiếp đồng bộ làm giảm tính khả dụng

REST là một cơ chế IPC cực kỳ phổ biến. Bạn có thể bị cám dỗ sử dụng nó để liên kếtgiao tiếp dịch vụ. Tuy nhiên, vấn đề với REST là nó là một giao thức đồng bộ: máy khách HTTP phải đợi dịch vụ gửi phản hồi. Bất cứ khi nào các dịch vụ giao tiếp bằng giao thức đồng bộ, tính khả dụng của ứng dụng sẽ giảm.

Để xem lý do tại sao, hãy xem xét kịch bản được hiển thị trong hình 3.15.Dịch vụ đặt hàngcó một REST API để tạo ra mộtĐặt hàng. Nó gợi lênDịch vụ khách hàngvàNhà hàngDịch vụđể xác thựcĐặt hàng. Cả hai dịch vụ đó đều có REST API.

NHẬN/người tiêu dùng/id

Dịch vụ đặt hàng

Người tiêu dùngDịch vụ

BÀI ĐĂNG/ĐƠN HÀNG

Khách hàng

NHẬN/nhà hàng/id

Nhà hàngDịch vụ

**Hình 3.15Dịch vụ đặt hànggọi các dịch vụ khác bằng REST. Rất đơn giản, nhưng nó yêu cầu tất cả các dịch vụ phải khả dụng đồng thời, làm giảm tính khả dụng của API.**

Trình tự các bướcđể tạo một đơn hàng như sau:

**1**Khách hàng thực hiện HTTPBƯU KIỆN/đơn hàngyêu cầu đếnDịch vụ đặt hàng.

**2** Dịch vụ đặt hànglấy thông tin người tiêu dùng bằng cách thực hiện HTTP LẤY

/người tiêu dùng/idyêu cầu đếnDịch vụ khách hàng.

**3** Dịch vụ đặt hànglấy thông tin nhà hàng bằng cách thực hiện HTTP LẤY

/nhà hàng/idyêu cầu đếnNhà hàngDịch vụ.

**4**Chức năng ghi nhận đơn hàng xác thực yêu cầu bằng cách sử dụng thông tin của người tiêu dùng và nhà hàng.

**5** Nhận đơn hàngtạo một Đơn hàng.

**6** Nhận đơn hànggửi phản hồi HTTP tới máy khách.

Vì các dịch vụ này sử dụng HTTP, tất cả chúng phải đồng thời khả dụng để ứng dụng FTGO xử lý yêu cầu CreateOrder. Ứng dụng FTGO không thể tạo đơn hàng nếu bất kỳ một trong ba dịch vụ này ngừng hoạt động. Về mặt toán học, tính khả dụng của một hoạt động hệ thống là sản phẩm của tính khả dụng củadịch vụ được gọi bởi hoạt động đó. Nếu Order Service và hai dịch vụ mà nó gọi có sẵn 99,5%, thì tính khả dụng chung là 99,5%3 = 98,5%, ít hơn đáng kể. Mỗi dịch vụ bổ sung tham gia xử lý yêu cầu làm giảm tính khả dụng hơn nữa.

Vấn đề này không chỉ dành riêng cho giao tiếp dựa trên REST. Tính khả dụng bị giảm bất cứ khi nào một dịch vụ chỉ có thể phản hồi cho máy khách của mình sau khi nhận được phản hồi từ một dịch vụ khác. Vấn đề này tồn tại ngay cả khi các dịch vụ giao tiếp bằng tương tác theo kiểu yêu cầu/phản hồi qua tin nhắn không đồng bộ. Ví dụ, tính khả dụngcủa LệnhDịch vụ sẽ bị giảm nếu gửi tin nhắn đến Dịch vụ người tiêu dùng thông qua một trung gian tin nhắn rồi chờ phản hồi.

Nếu bạn muốn tối đa hóa tính khả dụng, bạn phải giảm thiểu lượng giao tiếp đồng bộ. Hãy cùng xem cách thực hiện điều đó.

###### Loại bỏ tương tác đồng bộ

Có một số cách khác nhau để giảm lượng giao tiếp đồng bộ với các dịch vụ khác trong khi xử lý các yêu cầu đồng bộ. Một giải pháp là tránh hoàn toàn vấn đề bằng cách xác định các dịch vụ chỉ có API không đồng bộ. Tuy nhiên, điều đó không phải lúc nào cũng khả thi. Ví dụ, API công khai thường là RESTful. Do đó, đôi khi các dịch vụ được yêu cầu phải có API đồng bộ.

May mắn thay, có nhiều cách để xử lý các yêu cầu đồng bộ mà không cần thực hiện các yêu cầu đồng bộ. Hãy cùng thảo luận về các tùy chọn.

**BạnSE CÁC KIỂU TƯƠNG TÁC KHÔNG ĐỒNG BỘ**

Lý tưởng nhất là tất cả các tương tác nên được thực hiện bằng cách sử dụng các kiểu tương tác không đồng bộ được mô tả trước đó trong chương này. Ví dụ, giả sử một máy khách của ứng dụng FTGO sử dụng kiểu tương tác yêu cầu không đồng bộ/phản hồi không đồng bộ để tạo đơn hàng. Máy khách tạo đơn hàng bằng cách gửi tin nhắn yêu cầu đến Dịch vụ đơn hàng.

Dịch vụ này sau đó trao đổi tin nhắn không đồng bộ với các dịch vụ khác và cuối cùng gửi tin nhắn trả lời cho máy khách. Hình 3.16 cho thấy thiết kế.

Tạo đơn hànglời yêu cầu

Khách hàng

Yêu cầu đặt hàngkênh

Yêu cầu của người tiêu dùngkênh

Trả lời của khách hàngkênh

Tạo đơn hàngphản ứng

Yêu cầu nhà hàngkênh

**Hình 3.16 Ứng dụng FTGO có tính khả dụng cao hơn nếu các dịch vụ của nó giao tiếp bằng cách sử dụng giao thức không đồng bộnhắn tin thay vì gọi điện đồng bộ.**

Đặt hàngKênh trả lời dịch vụ

Nhà hàngDịch vụ

Dịch vụ đặt hàng

Người tiêu dùngDịch vụ

Máy khách và dịch vụ giao tiếp không đồng bộ bằng cách gửi tin nhắn qua kênh nhắn tin. Không có người tham gia nào trong tương tác này bị chặn chờ phản hồi.

Kiến trúc như vậy sẽ cực kỳ bền bỉ, vì message broker sẽ lưu trữ tin nhắn cho đến khi chúng có thể được sử dụng. Tuy nhiên, vấn đề là các dịch vụ thường có API bên ngoài sử dụng giao thức đồng bộ như REST, do đó nó phải phản hồi các yêu cầu ngay lập tức.

Nếu một dịch vụ có API đồng bộ, một cách để cải thiện tính khả dụng là sao chép dữ liệu. Hãy cùng xem cách thức hoạt động của nó.

**RTÍCH HỢP DỮ LIỆU**

Một cách để giảm thiểu các yêu cầu đồng bộ trong quá trình xử lý yêu cầu là sao chép dữ liệu. Một dịch vụ duy trì một bản sao của dữ liệu mà nó cần khi xử lý các yêu cầu. Nó giữ cho bản sao được cập nhật bằng cách đăng ký các sự kiện do các dịch vụ sở hữu dữ liệu công bố. Ví dụ: Order Service có thể duy trì một bản sao của dữ liệu do Consumer Service và Restaurant Service sở hữu. Điều này sẽ cho phép Order Service xử lý yêu cầu tạo đơn hàng mà không cần phải tương tác với các dịch vụ đó. Hình 3.17 cho thấy thiết kế.

Dịch vụ khách hàngVàDịch vụ nhà hàngcông bố sự kiện bất cứ khi nào dữ liệu của họthay đổi.Dịch vụ đặt hàngđăng ký những sự kiện đó và cập nhật bản sao của nó.

Trong một số trường hợp, sao chép dữ liệu là một cách tiếp cận hữu ích. Ví dụ, chương 5 mô tả cách Order Service sao chép dữ liệu từ Restaurant Service để có thể xác thực và định giá các mục trong thực đơn. Một nhược điểm của việc sao chép là đôi khi nó có thể yêu cầu sao chép một lượng lớn dữ liệu, điều này không hiệu quả. Ví dụ, Order Service có thể không thực tế khi duy trì một bản sao dữ liệu do Consumer Service sở hữu, do số lượng người tiêu dùng lớn. Một nhược điểm khác của

Sự kiện tiêu dùngkênh



Cơ sở dữ liệu dịch vụ tiêu dùng

«bảng» NGƯỜI TIÊU DÙNG

Người tiêu dùngDịch vụ

tạo đơn hàng()

Dịch vụ đặt hàng

Sự kiện nhà hàngkênh

Cơ sở dữ liệu dịch vụ nhà hàng

«bàn» NHÀ HÀNG

Nhà hàngDịch vụ

**Các dịch vụ sẽ công bố sự kiện khi dữ liệu của chúng thay đổi.**

Cơ sở dữ liệu dịch vụ đặt hàng

"bàn"ĐƠN HÀNG

«bàn» NHÀ HÀNG

«bảng» NGƯỜI TIÊU DÙNG

**Dữ liệu được sao chép cho phép Order Service xử lý yêu cầu createOrder() mà không cần gọi đồng bộ các dịch vụ.**

**Hình 3.17 Dịch vụ đặt hàngmang tính độc lập vì nó có bản sao dữ liệu của người tiêu dùng và nhà hàng.**

sao chép là nó khônggiải quyết vấn đề về cách dịch vụ cập nhật dữ liệu thuộc sở hữu của các dịch vụ khác.

Một cách để giải quyết vấn đề đó là dịch vụ trì hoãn tương tác với các dịch vụ khác cho đến khi nó phản hồi cho máy khách. Tiếp theo chúng ta sẽ xem cách thức hoạt động của nó.

**FXỬ LÝ INISH SAU KHI TRẢ LỜI PHẢN HỒI**

Một cách khác để loại bỏ giao tiếp đồng bộ trong quá trình xử lý yêu cầu là sử dụng dịch vụ để xử lý yêu cầu như sau:

**1** Xác thực yêu cầu chỉ bằng dữ liệu có sẵn tại địa phương.

**2** Cập nhật cơ sở dữ liệu, bao gồm việc chèn tin nhắn vào bảng OUTBOX.

**3** Trả về phản hồi cho máy khách.

Trong khi xử lý yêu cầu, dịch vụ không tương tác đồng bộ với bất kỳ dịch vụ nào khác. Thay vào đó, nó gửi tin nhắn không đồng bộ đến các dịch vụ khác. Cách tiếp cận này đảm bảo rằng các dịch vụ được kết nối lỏng lẻo. Như bạn sẽ tìm hiểu trong chương tiếp theo, điều này thường được triển khai bằng cách sử dụng saga.

Ví dụ, nếu Order Service sử dụng cách tiếp cận này, nó sẽ tạo một đơn hàng ở trạng thái PENDING và sau đó xác thực đơn hàng đó một cách không đồng bộ bằng cách trao đổi tin nhắn với các dịch vụ khác. Hình 3.18 cho thấy điều gì xảy ra khi hoạt động createOrder() được gọi. Trình tự các sự kiện như sau:

**1**Dịch vụ đặt hàngtạo một đơn hàng trong mộtCHƯA GIẢI QUYẾTtình trạng.

**2**Đặt hàngDịch vụtrả về phản hồi cho máy khách có chứa ID đơn hàng.

**3** Dịch vụ đặt hànggửi mộtXác thựcThông tin người tiêu dùngtin nhắn đếnDịch vụ khách hàng.

Chìa khóa

Đồng bộ Không đồng bộ

...

Khách hàng

tạo đơn hàng

tạo nênđặt hàng

tạo đơn hàng

Xác thựcThông tin người tiêu dùng

cập nhậtđặt hàng

OrderDetailsĐã xác thực

cập nhậtđặt hàng

...

Người tiêu dùng đã xác thực

Xác nhận chi tiết đơn hàng

**Không đồng bộ**

Dịch vụ nhà hàng

Dịch vụ khách hàng

Dịch vụ đặt hàng

**Đồng bộ**

**Hình 3.18Dịch vụ đặt hàngtạo một lệnh mà không cần gọi bất kỳ dịch vụ nào khác. Sau đó, nó xác thực lệnh mới được tạo một cách không đồng bộĐặt hàngbằng cách trao đổi tin nhắn với các dịch vụ khác, bao gồmDịch vụ khách hàngVàDịch vụ nhà hàng.**

**4** Dịch vụ đặt hànggửi mộtXác nhận chi tiết đơn hàngtin nhắn đếnDịch vụ nhà hàng.

**5**Dịch vụ khách hàngnhận được mộtXác thựcThông tin người tiêu dùngtin nhắn, xác minh con-người tiêu dùng có thể đặt hàng và gửi mộtNgười tiêu dùng đã xác thựctin nhắn đếnDịch vụ đặt hàng.

**6**Dịch vụ nhà hàngnhận được mộtXác nhận chi tiết đơn hàngtin nhắn, xác minh mục menu là hợp lệ và nhà hàng có thể giao hàng đến địa chỉ giao hàng của đơn hàng và gửiOrderDetailsĐã xác thựctin nhắn đếnDịch vụ đặt hàng.

**7**Dịch vụ đặt hàngnhận đượcNgười tiêu dùng đã xác thựcVàOrderDetailsĐã xác thựcvà thay đổi trạng thái của lệnh thànhĐÃ XÁC NHẬN.

**8** …

Dịch vụ đặt hàngcó thể nhận đượcNgười tiêu dùng đã xác thựcVàOrderDetailsĐã xác thựctin nhắn theo thứ tự nào. Nó theo dõi tin nhắn nào nó nhận được đầu tiên bằng cách thay đổitrạng thái của đơn hàng. Nếu nó nhận đượcNgười tiêu dùng đã xác thựcĐầu tiên, nó thay đổi trạng thái của lệnh thànhNGƯỜI TIÊU DÙNG ĐÃ XÁC THỰC, trong khi nếu nó nhận đượcOrderDetailsĐã xác thựctin nhắn đầu tiên, nó thay đổi trạng thái của nó thànhĐƠN\_HÀNH\_CHI\_THÔNG\_TIN\_ĐÃ\_XÁC\_NHẬN.Dịch vụ đặt hàngthay đổitình trạng củaĐặt hàngĐẾNĐÃ XÁC NHẬNkhi nó nhận được tin nhắn khác.

Sau khi Đơn hàng được xác thực, Dịch vụ đơn hàng sẽ hoàn tất phần còn lại của quy trình tạo đơn hàng, được thảo luận trong chương tiếp theo. Điểm hay của cách tiếp cận này làrằng ngay cả khi Dịch vụ tiêu dùng ngừng hoạt động, ví dụ, Dịch vụ đặt hàng vẫntạo đơn hàng và phản hồi cho khách hàng. Cuối cùng, Consumer Service sẽ quay lại và xử lý mọi tin nhắn đã xếp hàng và đơn hàng sẽ được xác thực.

Một nhược điểm của dịch vụ phản hồi trước khi xử lý đầy đủ yêu cầu là nó làm cho khách hàng phức tạp hơn. Ví dụ, Order Service đưa ra các đảm bảo tối thiểu về trạng thái của đơn hàng mới tạo khi trả về phản hồi. Nó tạo đơn hàng và trả về ngay trước khi xác thực đơn hàng và ủy quyền thẻ tín dụng của người tiêu dùng. Do đó, để khách hàng biết được đơn hàngđã được tạo thành công, hoặc là nó phải thăm dò định kỳ hoặc Order Service phải gửi cho nó một thông báo. Nghe có vẻ phức tạp, nhưng trong nhiều tình huống, đây là cách tiếp cận được ưa chuộng—đặc biệt là vì nó cũng giải quyết các vấn đề quản lý giao dịch phân tán mà tôi thảo luận trong chương tiếp theo.Ví dụ, trong chương 4 và 5, tôimô tả cách Dịch vụ đặt hàng sử dụng phương pháp này.

#### Bản tóm tắt

* Kiến trúc vi dịch vụ là kiến ​​trúc phân tán, do đó giao tiếp giữa các quy trình đóng vai trò quan trọng.
* Điều cần thiết là phải quản lý cẩn thận sự tiến hóa của API của dịch vụ. Các thay đổi tương thích ngược là dễ thực hiện nhất vì chúng không ảnh hưởng đến khách hàng. Nếu bạn thực hiện thay đổi đột ngột đối với API của dịch vụ, thì thường sẽ cần hỗ trợ cả phiên bản cũ và mới cho đến khi khách hàng của dịch vụ được nâng cấp.
* Có nhiều công nghệ IPC, mỗi công nghệ có những sự đánh đổi khác nhau. Một quyết định thiết kế quan trọng là chọn mẫu gọi thủ tục từ xa đồng bộ hoặc mẫu Nhắn tin không đồng bộ. Các giao thức dựa trên lệnh gọi thủ tục từ xa đồng bộ, chẳng hạn như REST, là dễ sử dụng nhất. Nhưng lý tưởng nhất là các dịch vụ nên giao tiếp bằng cách sử dụng nhắn tin không đồng bộ để tăng tính khả dụng.
* Để ngăn chặn lỗi lan truyền qua hệ thống, một máy khách dịch vụ sử dụng giao thức đồng bộ phải được thiết kế để xử lý lỗi cục bộ, tức là khi dịch vụ được gọi ngừng hoạt động hoặc có độ trễ cao. Cụ thể, nó phải sử dụng thời gian chờ khi thực hiện yêu cầu, giới hạn số lượng yêu cầu đang chờ xử lý và sử dụng mẫu Circuit breaker để tránh thực hiện cuộc gọi đến dịch vụ đang gặp sự cố.
* Kiến trúc sử dụng giao thức đồng bộ phải bao gồm cơ chế khám phá dịch vụ để máy khách có thể xác định vị trí mạng của một phiên bản dịch vụ. Cách tiếp cận đơn giản nhất là sử dụng cơ chế khám phá dịch vụ do nền tảng triển khai triển khai: khám phá phía máy chủ và các mẫu đăng ký của bên thứ 3. Nhưng một cách tiếp cận thay thế là triển khai khám phá dịch vụ ở cấp ứng dụng: khám phá phía máy khách và tự đăng ký

***Bản tóm tắt* 109**

mẫu. Việc này tốn nhiều công sức hơn, nhưng nó xử lý được tình huống khi các dịch vụ đang chạy trên nhiều nền tảng triển khai.

* + Một cách tốt để thiết kế kiến ​​trúc dựa trên tin nhắn là sử dụng mô hình tin nhắn và kênh, mô hình này tóm tắt các chi tiết của hệ thống tin nhắn cơ bản. Sau đó, bạn có thể ánh xạ thiết kế đó vào một cơ sở hạ tầng tin nhắn cụ thể, thường dựa trên môi giới tin nhắn.
  + Một thách thức chính khi sử dụng tin nhắn là cập nhật cơ sở dữ liệu và xuất bản tin nhắn theo cách nguyên tử. Một giải pháp tốt là sử dụng mẫu Transactional outbox và trước tiên ghi tin nhắn vào cơ sở dữ liệu như một phần của giao dịch cơ sở dữ liệu. Sau đó, một quy trình riêng biệt sẽ truy xuất tin nhắn từ cơ sở dữ liệu bằng mẫu Polling publisher hoặc mẫu Transaction log tailing và xuất bản tin nhắn đó cho message broker.

*Quản lý giao dịch*

*với saga*

***Chương này bao gồm***

* Tại sao các giao dịch phân tán không phù hợp vớiứng dụng hiện đại
* Sử dụng mô hình Saga để duy trì tính nhất quán của dữ liệu trong kiến ​​trúc dịch vụ vi mô
* Phối hợp các saga bằng vũ đạo và dàn nhạc
* Sử dụng các biện pháp đối phó để giải quyếtthiếu sự cô lập

Khi Mary bắt đầu nghiên cứu kiến ​​trúc dịch vụ vi mô, một trong những mối quan tâm lớn nhất của cô là cách triển khai các giao dịch trải dài trên nhiều dịch vụ. Giao dịch là thành phần thiết yếu của mọi ứng dụng doanh nghiệp. Nếu không có giao dịch, sẽ không thể duy trì tính nhất quán của dữ liệu.

Các giao dịch ACID (Atomicity, Consistency, Isolation, Durability) đơn giản hóa đáng kể công việc của nhà phát triển bằng cách tạo ra ảo giác rằng mỗi giao dịch có quyền truy cập độc quyền vào dữ liệu. Trong kiến ​​trúc dịch vụ vi mô, các giao dịch nằm trong một dịch vụ duy nhất vẫn có thể sử dụng các giao dịch ACID. Tuy nhiên, thách thức nằm ở việc triển khai các giao dịch cho các hoạt động cập nhật dữ liệu thuộc sở hữu của nhiều dịch vụ.

**110**

Ví dụ, như đã mô tả trong chương 2,tạo đơn hàng()hoạt động bao gồm nhiều dịch vụ, bao gồmDịch vụ đặt hàng,Dịch vụ nhà bếp, VàDịch vụ kế toán. Các hoạt động như thế này cần có cơ chế quản lý giao dịch hoạt động trên nhiều dịch vụ.

Mary phát hiện ra rằng, như đã đề cập trong chương 2, cách tiếp cận truyền thống đối với quản lý giao dịch phân tán không phải là lựa chọn tốt cho các ứng dụng hiện đại. Thay vì giao dịch ACID, một hoạt động trải dài trên các dịch vụ phải sử dụng cái được gọi là saga, một chuỗi giao dịch cục bộ được điều khiển bởi thông điệp, để duy trì tính nhất quán của dữ liệu. Một thách thức với saga là chúng là ACD (Atomicity, Consistency, Durability). Chúng thiếu tính năng cô lập của các giao dịch ACID truyền thống. Do đó, một ứng dụng phải sử dụng cái được gọi là biện pháp đối phó, các kỹ thuật thiết kế ngăn ngừa hoặc giảm tác động của các bất thường đồng thời do thiếu sự cô lập.

Theo nhiều cách, trở ngại lớn nhất mà Mary và các nhà phát triển FTGO sẽ phải đối mặt khi áp dụng các dịch vụ vi mô là chuyển từ một cơ sở dữ liệu duy nhất với các giao dịch ACID sang kiến ​​trúc đa cơ sở dữ liệu với các saga ACD. Họ đã quen với sự đơn giản của mô hình giao dịch ACID. Nhưng trên thực tế, ngay cả các ứng dụng đơn khối như ứng dụng FTGO thường không sử dụng các giao dịch ACID theo sách giáo khoa. Ví dụ, nhiều ứng dụng sử dụng mức độ cô lập giao dịch thấp hơn để cải thiện hiệu suất. Ngoài ra, nhiều quy trình kinh doanh quan trọng, chẳng hạn như chuyển tiền giữa các tài khoản tại các ngân hàng khác nhau, cuối cùng cũng nhất quán. Ngay cả Starbucks cũng không sử dụng cam kết hai giai đoạn ([www.enterpriseintegrationpatterns.com/ramblings/18\_starbucks.html](http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html)).

Tôi bắt đầu chương này bằng cách xem xét những thách thức của quản lý giao dịch trong kiến ​​trúc vi dịch vụ và giải thích lý do tại sao cách tiếp cận truyền thống đối với quản lý giao dịch phân tán không phải là một lựa chọn. Tiếp theo, tôi giải thích cách duy trì tính nhất quán của dữ liệu bằng saga. Sau đó, tôi xem xét hai cách khác nhau để điều phối saga: biên đạo, trong đó những người tham gia trao đổi các sự kiện mà không có điểm kiểm soát tập trung và phối hợp, trong đó một bộ điều khiển tập trung cho những người tham gia saga biết phải thực hiện thao tác nào. Tôi thảo luận về cách sử dụng các biện pháp đối phó để ngăn ngừa hoặc giảm tác động của các bất thường đồng thời do thiếu sự cô lập giữa các saga. Cuối cùng, tôi mô tả việc triển khai một saga ví dụ.

Chúng ta hãy bắt đầu bằng cách xem xét thách thức trong việc quản lý giao dịch trong kiến ​​trúc dịch vụ vi mô.

#### Quản lý giao dịch trong một dịch vụ vi môngành kiến ​​​​trúc

Hầu như mọi yêu cầu được xử lý bởi một ứng dụng doanh nghiệp đều được thực hiện trong một giao dịch cơ sở dữ liệu. Các nhà phát triển ứng dụng doanh nghiệp sử dụng các khuôn khổ và thư viện giúp đơn giản hóa việc quản lý giao dịch. Một số khuôn khổ và thư viện cung cấp API theo chương trình để bắt đầu, cam kết và hoàn tác các giao dịch một cách rõ ràng. Các khuôn khổ khác, chẳng hạn như khuôn khổ Spring, cung cấp một cơ chế khai báo. Spring cung cấp chú thích @Transactional sắp xếp để các lệnh gọi phương thức được

được thực hiện tự động trong một giao dịch. Do đó, việc viết logic kinh doanh giao dịch rất đơn giản.

Hoặc, nói chính xác hơn, quản lý giao dịch rất đơn giản trong một ứng dụng đơn khối truy cập vào một cơ sở dữ liệu duy nhất. Quản lý giao dịch khó khăn hơn trong một ứng dụng đơn khối phức tạp sử dụng nhiều cơ sở dữ liệu và môi giới tin nhắn. Và trong kiến ​​trúc vi dịch vụ, các giao dịch trải dài trên nhiều dịch vụ, mỗi dịch vụ có cơ sở dữ liệu riêng. Trong tình huống này, ứng dụng phải sử dụng một cơ chế phức tạp hơn để quản lý các giao dịch. Như bạn sẽ thấy, cách tiếp cận truyền thống là sử dụng các giao dịch phân tán không phải là một lựa chọn khả thi cho các ứng dụng hiện đại. Thay vào đó, một ứng dụng dựa trên vi dịch vụ phải sử dụng sagas.

Trước khi tôi giải thích về saga, trước tiên chúng ta hãy xem tại sao quản lý giao dịch lại là một thách thức trong kiến ​​trúc vi dịch vụ.

###### Nhu cầu về các giao dịch phân tán trong một dịch vụ vi môngành kiến ​​​​trúc

Hãy tưởng tượng rằng bạn là nhà phát triển FTGO chịu trách nhiệm triển khai hoạt động hệ thống create-Order(). Như đã mô tả trong chương 2, hoạt động này phải xác minh rằng người tiêu dùng có thể đặt hàng, xác minh thông tin chi tiết về đơn hàng, ủy quyền thẻ tín dụng của người tiêu dùng và tạo Đơn hàng trong cơ sở dữ liệu. Việc triển khai hoạt động này trong ứng dụng FTGO nguyên khối tương đối đơn giản. Tất cả dữ liệu cần thiết để xác thực đơn hàng đều có thể truy cập dễ dàng. Hơn nữa, bạn có thể sử dụng giao dịch ACIDđể đảm bảo tính nhất quán của dữ liệu. Bạn có thể sử dụng chú thích @Transactional của Spring trên phương thức dịch vụ createOrder().

Ngược lại, việc triển khai cùng một hoạt động trong kiến ​​trúc dịch vụ vi mô phức tạp hơn nhiều. Như hình 4.1 cho thấy, dữ liệu cần thiết nằm rải rác xung quanh nhiều dịch vụ.tạo đơn hàng()hoạt động truy cập dữ liệu trong nhiều dịch vụ. Nó đọc dữ liệu từDịch vụ khách hàngvà cập nhật dữ liệu trongDịch vụ đặt hàng,Dịch vụ nhà bếp, VàDịch vụ kế toán.

Vì mỗi dịch vụ đều có cơ sở dữ liệu riêng nên bạn cần sử dụng cơ chế để duy trì tính nhất quán của dữ liệu trên các cơ sở dữ liệu đó.

###### Rắc rối với các giao dịch phân tán

Cách tiếp cận truyền thống để duy trì tính nhất quán của dữ liệu trên nhiều dịch vụ, cơ sở dữ liệu hoặc nhà môi giới tin nhắn là sử dụng các giao dịch phân tán. Tiêu chuẩn thực tế choquản lý giao dịch phân tán là Mô hình Xử lý giao dịch phân tán (DTP) X/Open(X/Mở XA—xem<https://en.wikipedia.org/wiki/X/Open_XA>).XA sử dụng cam kết hai giai đoạn (2PC) để đảm bảo rằng tất cả những người tham gia giao dịch đều cam kết hoặc khôi phục. Một ngăn xếp công nghệ tuân thủ XA bao gồm các cơ sở dữ liệu và trình môi giới tin nhắn tuân thủ XA, trình điều khiển cơ sở dữ liệu và API nhắn tin, và một cơ chế giao tiếp giữa các tiến trình truyền bá ID giao dịch toàn cục XA. Hầu hết các cơ sở dữ liệu SQL đều tuân thủ XA, cũng như một số trình môi giới tin nhắn. Ví dụ, các ứng dụng Java EE có thể sử dụng JTA để thực hiện các giao dịch phân tán.

**Hoạt động createOrder() đọc từ Consumer Service và cập nhật dữ liệu**

**Dịch vụ đặt hàng, Dịch vụ bếp và Dịch vụ kế toán.**

tạo đơn hàng()

Dịch vụ đặt hàng

Yêu cầu tính nhất quán của dữ liệu

Đọc

Viết

Viết

Bộ điều khiển đơn hàng

Tài khoản

Vé

Người tiêu dùng

Đặt hàng

Người tiêu dùngDịch vụ

Dịch vụ nhà bếp

Dịch vụ kế toán

**Hình 4.1tạo đơn hàng()hoạt động cập nhật dữ liệu trong một số dịch vụ. Nó phải sử dụng mộtcơ chế duy trì tính nhất quán của dữ liệu trên các dịch vụ đó.**

Nghe có vẻ đơn giản, nhưng có nhiều vấn đề với giao dịch phân tán. Một vấn đề là nhiều công nghệ hiện đại, bao gồm cả cơ sở dữ liệu NoSQL như MongoDB và Cassandra, không hỗ trợ chúng. Ngoài ra, các giao dịch phân tán không được hỗ trợ bởi các trình môi giới tin nhắn hiện đại như RabbitMQ và Apache Kafka. Do đó, nếu bạn khăng khăng sử dụng giao dịch phân tán, bạn không thể sử dụng nhiều công nghệ hiện đại.

Một vấn đề khác với các giao dịch phân tán là chúng là một dạng IPC đồng bộ, làm giảm tính khả dụng. Để một giao dịch phân tán có thể cam kết, tất cả các dịch vụ tham gia phải khả dụng. Như đã mô tả trong chương 3, tính khả dụng là tích của tính khả dụng của tất cả những người tham gia giao dịch. Nếu một giao dịch phân tán bao gồm hai dịch vụ có 99,5% khả dụng, thì tính khả dụng chung là 99%, tức là ít hơn đáng kể. Mỗi dịch vụ bổ sung liên quan đến một giao dịch phân tán làm giảm thêm tính khả dụng. Thậm chí còn có định lý CAP của Eric Brewer, trong đó nêu rằng một hệ thống chỉ có thể có hai trong ba thuộc tính sau:

tính nhất quán, tính khả dụng và dung sai phân vùng ([https://en.wikipedia.org/wiki/CAP](https://en.wikipedia.org/wiki/CAP_theorem)

[\_định lý](https://en.wikipedia.org/wiki/CAP_theorem)). Ngày nay, các kiến ​​trúc sư thích có một hệ thống có sẵn hơn là một hệ thống nhất quán.

Trên bề mặt, các giao dịch phân tán rất hấp dẫn. Theo quan điểm của nhà phát triển, chúng có cùng mô hình lập trình với các giao dịch cục bộ. Nhưng do những vấn đề đã đề cập cho đến nay, các giao dịch phân tán không phải là công nghệ khả thi cho các ứng dụng hiện đại. Chương 3 mô tả cách gửi tin nhắn như một phần của giao dịch cơ sở dữ liệu mà không sử dụng các giao dịch phân tán. Để giải quyết vấn đề phức tạp hơn là duy trì tính nhất quán của dữ liệu trong kiến ​​trúc dịch vụ vi mô, ứng dụng phải sử dụng một cơ chế khác dựa trên khái niệm về các dịch vụ không đồng bộ, liên kết lỏng lẻo. Đây chính là lúc sagas xuất hiện.

###### Sử dụng mẫu Saga để duy trì tính nhất quán của dữ liệu

*Truyện dài*là các cơ chế duy trì tính nhất quán của dữ liệu trong kiến ​​trúc vi dịch vụ mà không cần sử dụng các giao dịch phân tán. Bạn định nghĩa một saga cho mỗi lệnh hệ thống cần cập nhật dữ liệu trong nhiều dịch vụ. Một saga là một chuỗi các giao dịch cục bộ. Mỗi giao dịch cục bộ cập nhật dữ liệu trong một dịch vụ duy nhất bằng cách sử dụng các thư viện và khuôn khổ giao dịch ACID quen thuộc đã đề cập trước đó.

**Mẫu: Saga**

Duy trì tính nhất quán của dữ liệu trên các dịch vụ bằng cách sử dụng một chuỗi các giao dịch cục bộ được phối hợp bằng cách sử dụng nhắn tin không đồng bộ. Xem[http://microservices.io/](http://microservices.io/patterns/data/saga.html) [mẫu/dữ liệu/saga.html](http://microservices.io/patterns/data/saga.html).

Hoạt động của hệ thống khởi tạo bước đầu tiên của saga. Việc hoàn thành một giao dịch cục bộ sẽ kích hoạt việc thực hiện giao dịch cục bộ tiếp theo. Sau đó, trong phần 4.2, bạn sẽ thấy cách phối hợp các bước được triển khai bằng cách sử dụng tin nhắn không đồng bộ. Một lợi ích quan trọng của tin nhắn không đồng bộ là nó đảm bảo rằng tất cả các bước của saga đều được thực hiện, ngay cả khi một hoặc nhiều người tham gia saga tạm thời không có mặt.

Sagas khác với giao dịch ACID ở một số điểm quan trọng. Như tôi mô tả trongchi tiết trong phần 4.3, chúng thiếu thuộc tính cô lập của các giao dịch ACID. Ngoài ra, vì mỗi giao dịch cục bộ cam kết các thay đổi của nó, nên một saga phải được khôi phục bằng cách sử dụng các giao dịch bù trừ. Tôi sẽ nói thêm về các giao dịch bù trừ sau trong phần này. Chúng ta hãy xem một saga ví dụ.

**MỘTN VÍ DỤ SAGA: TANH TACTỔNG KẾTỒSỰ KIỆN RDER**

Saga mẫu được sử dụng trong toàn bộ chương này là Create Order Saga, được hiển thị trong hình 4.2. Order Service triển khai thao tác createOrder() bằng saga này. Giao dịch cục bộ đầu tiên của saga được khởi tạo bằng yêu cầu bên ngoài để tạo một đơn hàng. Năm giao dịch cục bộ khác được kích hoạt khi hoàn tất giao dịch trước đó.

Tạo đơn hàng

Txn:1

|  |  |  |
| --- | --- | --- |
|  | Txn:1 | |
| Tạo đơn hàng | |  |

**Hình 4.2 Tạo mộtĐặt hàngsử dụng một saga.tạo đơn hàng()hoạt động được thực hiện bởi mộtsaga bao gồm các giao dịch cục bộ trong một số dịch vụ.**

Truyện dài

Dịch vụ kế toán

Dịch vụ nhà bếp

Dịch vụ khách hàng

Dịch vụ đặt hàng

|  |  |  |
| --- | --- | --- |
|  | Txn:2 | |
| Xác minh người tiêu dùng | |  |

|  |  |  |
| --- | --- | --- |
|  | Txn:3 | |
| Tạo vé | |  |

|  |  |  |
| --- | --- | --- |
|  | Txn:4 | |
| Thẻ ủy quyền | |  |

|  |  |  |
| --- | --- | --- |
|  | Txn:5 | |
| Phê duyệt vé | |  |

|  |  |  |
| --- | --- | --- |
|  | Txn:6 | |
| Phê duyệt đơn hàng | |  |

Câu chuyện này bao gồm các giao dịch địa phương sau đây:

**1** Dịch vụ đặt hàng—Tạo mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.

**2** Dịch vụ khách hàng—Xác minh rằng người tiêu dùng có thể đặt hàng.

**3** Dịch vụ nhà bếp—Xác thực thông tin chi tiết về đơn hàng và tạoVétrongTẠO NÊN

\_CHƯA GIẢI QUYẾT.

**4** Dịch vụ kế toán—Ủy quyền thẻ tín dụng của người tiêu dùng.

**5** Dịch vụ nhà bếp—Thay đổi trạng thái củaVéĐẾNĐANG CHỜ CHẤP NHẬN.

**6** Dịch vụ đặt hàng—Thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNH.

Sau đó, trong phần 4.2, tôi mô tả cách các dịch vụ tham gia vào một saga giao tiếp bằng cách sử dụng tin nhắn không đồng bộ. Một dịch vụ sẽ công bố một tin nhắn khi một giao dịch cục bộ hoàn tất. Tin nhắn này sau đó sẽ kích hoạt bước tiếp theo trong saga. Việc sử dụng tin nhắn không chỉ đảm bảo những người tham gia saga được kết nối lỏng lẻo mà còn đảm bảo rằng một saga hoàn tất. Đó là bởi vì nếu người nhận tin nhắn tạm thời không có mặt, thì bộ trung gian tin nhắn sẽ lưu trữ tin nhắn cho đến khi có thể gửi được.

Trên bề mặt, saga có vẻ đơn giản, nhưng có một vài thách thức khi sử dụng chúng. Một thách thức là thiếu sự cô lập giữa các saga. Phần 4.3 mô tả cách xử lý vấn đề này. Một thách thức khác là khôi phục các thay đổi khi xảy ra lỗi. Chúng ta hãy cùng xem cách thực hiện.

**SAGAS SỬ DỤNG CÁC GIAO DỊCH BÙ ĐỔI ĐỂ HOÀN LẠI CÁC THAY ĐỔI**

Một tính năng tuyệt vời của các giao dịch ACID truyền thống là logic nghiệp vụ có thể dễ dàng khôi phục một giao dịch nếu phát hiện ra vi phạm quy tắc nghiệp vụ. Nó thực thi một câu lệnh ROLL-BACK và cơ sở dữ liệu sẽ hoàn tác tất cả các thay đổi đã thực hiện cho đến nay. Thật không may, sagas không thể tự động khôi phục, vì mỗi bước đều cam kết các thay đổi của nó với cơ sở dữ liệu cục bộ. Điều này có nghĩa là, ví dụ, nếu ủy quyền của thẻ tín dụngkhông thành công trong bước thứ tư của Create Order Saga, ứng dụng FTGO phải rõ ràng

hoàn tác các thay đổi được thực hiện bởi ba bước đầu tiên. Bạn phải viết những gì được gọi là giao dịch bù trừ.

Giả sử giao dịch thứ (n + 1) của một saga bị lỗi. Các tác động của n giao dịch trước đó phải được hoàn tác. Về mặt khái niệm, mỗi bước trong số các bước đó, Ti, có một giao dịch bù tương ứng, Ci, hoàn tác các tác động của Ti. Để hoàn tác các tác động của n bước đầu tiên đó, saga phải thực hiện từng Ci theo thứ tự ngược lại. Trình tự các bước là T1 … Tn, Cn … C1, như thể hiện trong hình 4.3. Trong ví dụ này, Tn+1 bị lỗi, yêu cầu các bước T1 … Tn phải được hoàn tác.

**Những thay đổi được thực hiện bởi T1...TNđã được cam kết.**

Truyện dài

**Các giao dịch bù trừ sẽ hoàn tác các thay đổi được thực hiện bởi T1...TN.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| T1 | ... | T*N* | T*N*+1THẤT ​​BẠI | C*N* | ... | C1 |

**Hình 4.3 Khi một bước của saga không thành công do vi phạm quy tắc kinh doanh, saga phải được xác định rõ rànghoàn tác các bản cập nhật được thực hiện ở các bước trước đó bằng cách thực hiện các giao dịch bù trừ.**

Câu chuyện thực hiện các giao dịch bù trừ theo thứ tự ngược lại với các giao dịch chuyển tiếp: Cn … C1. Cơ chế sắp xếp thứ tự Cis không khác gì so với sắp xếp thứ tự Tis. Việc hoàn thành Ci phải kích hoạt việc thực hiện Ci-1.

Ví dụ, hãy xem xét Create Order Saga. Saga này có thể thất bại vì nhiều lý do:

* Thông tin người tiêu dùng không hợp lệ hoặc người tiêu dùng không được phép tạo đơn hàng.
* Thông tin nhà hàng không hợp lệ hoặc nhà hàng không thể chấp nhậnđơn đặt hàng.
* Việc ủy ​​quyền thẻ tín dụng của người tiêu dùng không thành công.

Nếu một giao dịch cục bộ không thành công, cơ chế phối hợp của saga phải thực hiện các giao dịch bù trừ từ chối Đơn hàng và có thể là cả Vé. Bảng 4.1 hiển thị các giao dịch bù trừ cho từng bước của Saga Tạo đơn hàng. Điều quan trọng cần lưu ý là không phải tất cả các bước đều cần giao dịch bù trừ. Các bước chỉ đọc, chẳng hạn như verify- ConsumerDetails(), không cần giao dịch bù trừ. Các bước như authorizeCreditCard() theo sau là các bước luôn thành công cũng vậy.

Phần 4.3 thảo luận về cách ba bước đầu tiên củaTạo đơn hàng Sagađược gọi là các giao dịch có thể bù trừ vì chúng được theo sau bởi các bước có thể thất bại, cách bước thứ tư được gọi là giao dịch trục của saga vì nó được theo sau bởi các bước

**Bảng 4.1 Các giao dịch bù trừ choTạo đơn hàng Saga**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bước chân** | **Dịch vụ** | **Giao dịch** | **Giao dịch bù trừ** |
| 1 | Dịch vụ đặt hàng | tạo đơn hàng() | từ chối đơn hàng() |
| 2 | Dịch vụ khách hàng | xác minhConsumerDetails() | — |
| 3 | Dịch vụ nhà bếp | tạoTicket() | từ chốiTicket() |
| 4 | Dịch vụ kế toán | ủy quyền thẻ tín dụng() | — |
| 5 | Dịch vụ nhà bếp | phê duyệtVé() | — |
| 6 | Dịch vụ đặt hàng | chấp thuận đơn hàng() | — |

không bao giờ thất bại và hai bước cuối cùng được gọi là giao dịch có thể thử lại vì chúng luôn thành công.

Để xem cách sử dụng các giao dịch bù trừ, hãy tưởng tượng một kịch bản trong đó việc ủy ​​quyền thẻ tín dụng của người tiêu dùng không thành công. Trong kịch bản này, saga thực hiện các giao dịch cục bộ sau:

**1** Dịch vụ đặt hàng—Tạo mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.

**2** Dịch vụ khách hàng—Xác minh rằng người tiêu dùng có thể đặt hàng.

**3**Dịch vụ nhà bếp—Xác thực thông tin chi tiết đơn hàngvà tạo ra mộtVétrongTẠO NÊN

\_CHƯA GIẢI QUYẾTtình trạng.

**4** Dịch vụ kế toán—Ủy quyền thẻ tín dụng của người tiêu dùng nhưng không thành công.

**5** Dịch vụ nhà bếp—Thay đổi trạng thái củaVéĐẾNTẠO\_BỊ TỪ CHỐI.

**6** Dịch vụ đặt hàng—Thay đổi trạng thái củaĐặt hàngĐẾNVẬT BỊ LOẠI BỎ.

Bước thứ năm và thứ sáu là các giao dịch bù trừ hoàn tác các bản cập nhật được thực hiện bởi Kitchen Service và Order Service. Logic phối hợp của một saga chịu trách nhiệm sắp xếpviệc thực hiện các giao dịch chuyển tiếp và bù trừ. Hãy cùng xem cách thức hoạt động của nó.

#### Phối hợp các saga

Việc triển khai saga bao gồm logic phối hợp các bước của saga. Khi một saga được khởi tạo bởi lệnh hệ thống, logic phối hợp phải chọn và yêu cầu người tham gia saga đầu tiên thực hiện một giao dịch cục bộ. Khi giao dịch đó hoàn tất, phối hợp trình tự của saga sẽ chọn và gọi người tham gia saga tiếp theo. Quá trình này tiếp tục cho đến khi saga thực hiện tất cả các bước. Nếu bất kỳ giao dịch cục bộ nào không thành công, saga phải thực hiện các giao dịch bù trừ theo thứ tự ngược lại. Có một vài cách khác nhau để cấu trúc logic phối hợp của saga:

* + - *Biên đạo múa*—Phân phối việc ra quyết định và sắp xếp giữa những người tham gia saga. Họ chủ yếu giao tiếp bằng cách trao đổi các sự kiện.
* *Sự phối hợp*—Tập trung logic phối hợp của saga vào lớp saga orchestrator. Một saga orchestrator gửi tin nhắn lệnh đến những người tham gia saga để cho họ biết phải thực hiện những thao tác nào.

Hãy cùng xem xét từng lựa chọn, bắt đầu với vũ đạo.

###### Dựa trên vũ đạo truyện dài

Một cách bạn có thể thực hiện một saga là sử dụng vũ đạo. Khi sử dụng vũ đạo, không có điều phối viên trung tâm nào chỉ bảo những người tham gia saga phải làm gì. Thay vào đó, những người tham gia saga đăng ký các sự kiện của nhau và phản hồi theo đó. Để chỉ ra cách hoạt động của saga dựa trên vũ đạo, trước tiên tôi sẽ mô tả một ví dụ. Sau đó, tôi sẽ thảo luận về một vài vấn đề thiết kế mà bạn phải giải quyết. Sau đó, tôi sẽ thảo luận về những lợi ích và hạn chế của việc sử dụng vũ đạo.

**TÔITHỰC HIỆNCTỔNG KẾTỒRDER SAGA SỬ DỤNG BIÊN ĐẠO**

Hình 4.4 cho thấy thiết kế của phiên bản dựa trên vũ đạo của Create Order Saga. Những người tham gia giao tiếp bằng cách trao đổi các sự kiện. Mỗi người tham gia, bắt đầu với Order Service, cập nhật cơ sở dữ liệu của mình và công bố một sự kiện kích hoạt người tham gia tiếp theo.

Chìa khóa



Người môi giới tin nhắn

Xuất bản

Đặt mua

Đặt hàng sự kiện

1

Đã tạo đơn hàng

2

Người tiêu dùng đã xác minh

Sự kiện tiêu dùng

3

Vé đã được tạo

6

4

7

Sự kiện vé

5 giờ

5b

5

Thẻ tín dụng được chấp thuận

Sự kiện thẻ tín dụng

1. tạo đơn hàng()

7. phê duyệt đơn hàng()

Dịch vụ đặt hàng

4. tạoPendingAuthorization()

6. ủy quyền Thẻ()

Dịch vụ kế toán

Dịch vụ nhà bếp

3. tạoTicket()

6. phê duyệtTicket()

Dịch vụ khách hàng

2. xác minhConsumerDetails()

**Hình 4.4 Triển khaiTạo đơn hàng Sagasử dụng vũ đạo. Những người tham gia saga giao tiếp bằngtrao đổi sự kiện.**

Con đường hạnh phúc xuyên suốt câu chuyện này như sau:

**1**Dịch vụ đặt hàngtạo ra mộtĐặt hàngtrongĐANG CHỜ PHÊ DUYỆTnhà nước và công bố mộtĐơn hàngĐã tạosự kiện.

**2**Dịch vụ khách hàngtiêu thụĐơn hàngĐã tạosự kiện, xác minh rằng người tiêu dùng có thể đặt hàng và công bốNgười tiêu dùng đã xác minhsự kiện.

**3** Dịch vụ nhà bếptiêu thụĐơn hàngĐã tạosự kiện, xác nhậnĐặt hàng, tạo ra mộtVétrong mộtĐANG TẠO\_ĐANG CHỜnhà nước và công bốVéĐã tạosự kiện.

**4**Dịch vụ kế toántiêu thụĐơn hàngĐã tạosự kiện và tạo ra mộtỦy quyền thẻ tín dụngtrong mộtCHƯA GIẢI QUYẾTtình trạng.

**5**Dịch vụ kế toántiêu thụVéĐã tạoVàNgười tiêu dùng đã xác minhsự kiện, tính phí thẻ tín dụng của người tiêu dùng và công bốThẻ tín dụng-Được ủy quyềnsự kiện.

**6**Dịch vụ nhà bếptiêu thụThẻ tín dụngĐược ủy quyềnsự kiện và thay đổi trạng thái củaVéĐẾNĐANG CHỜ CHẤP NHẬN.

**7**Dịch vụ đặt hàngnhận đượcThẻ tín dụngĐược ủy quyềnsự kiện, thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNHvà xuất bản mộtĐơn hàng đã được phê duyệtsự kiện.

Saga Create Order cũng phải xử lý tình huống mà người tham gia saga từ chối Order và công bố một số loại sự kiện lỗi. Ví dụ, việc ủy ​​quyền thẻ tín dụng của người tiêu dùng có thể không thành công. Saga phải thực hiện các giao dịch bù trừ để hoàn tác những gì đã thực hiện. Hình 4.5 cho thấy luồng sự kiện khi AccountingService không thể ủy quyền thẻ tín dụng của người tiêu dùng.

Trình tự các sự kiện như sau:

**1** Dịch vụ đặt hàngtạo ra mộtĐặt hàngtrongĐANG CHỜ PHÊ DUYỆTnhà nước và công bố mộtĐơn hàngĐã tạosự kiện.

**2**Dịch vụ khách hàngtiêu thụĐơn hàngĐã tạosự kiện, xác minh rằng người tiêu dùng có thể đặt hàng và công bốNgười tiêu dùng đã xác minhsự kiện.

**3** Dịch vụ nhà bếptiêu thụĐơn hàngĐã tạosự kiện, xác nhậnĐặt hàng, tạo ra mộtVétrong mộtĐANG TẠO\_ĐANG CHỜnhà nước và công bốVéĐã tạosự kiện.

**4**Dịch vụ kế toántiêu thụĐơn hàngĐã tạosự kiện và tạo ra mộtỦy quyền thẻ tín dụngtrong mộtCHƯA GIẢI QUYẾTtình trạng.

**5**Dịch vụ kế toántiêu thụVéĐã tạoVàNgười tiêu dùng đã xác minhsự kiện, tính phí thẻ tín dụng của người tiêu dùng và công bố mộtXác thực thẻ tín dụng không thành côngsự kiện.

**6**Dịch vụ nhà bếptiêu thụXác thực thẻ tín dụng không thành côngsự kiện vàthay đổi trạng thái củaVéĐẾNVẬT BỊ LOẠI BỎ.

**7** Đặt hàngDịch vụtiêu thụXác thực thẻ tín dụng không thành côngsự kiện vàthay đổi trạng thái củaĐặt hàngĐẾNVẬT BỊ LOẠI BỎ.

Như bạn có thể thấy, những người tham gia saga dựa trên vũ đạo tương tác bằng cách sử dụng publish/subscribe. Chúng ta hãy xem xét kỹ hơn một số vấn đề bạn cần cân nhắc khi triển khai giao tiếp dựa trên publish/subscribe cho saga của mình.

Chìa khóa



Người môi giới tin nhắn

Xuất bản

Đặt mua

Đặt hàng sự kiện

1

Đã tạo đơn hàng

2

Người tiêu dùng đã xác minh

Sự kiện tiêu dùng

3

Vé đã được tạo

6

4

7

Sự kiện vé

5 giờ

5b

5

Xác thực thẻ tín dụng không thành công

Sự kiện thẻ tín dụng

1. tạo đơn hàng()

7. từ chối đơn hàng()

Dịch vụ đặt hàng

4. tạoPendingAuthorization()

6. ủy quyền Thẻ()

Dịch vụ kế toán

Dịch vụ nhà bếp

3. tạoTicket()

6. từ chốiTicket()

Dịch vụ khách hàng

2. xác minhConsumerDetails()

**Hình 4.5Trình tự các sự kiện trongTạo đơn hàng Sagakhi sự cho phép của tín dụng của người tiêu dùngthẻ bị lỗi.Dịch vụ kế toánxuất bảnXác thực thẻ tín dụng không thành côngsự kiện, gây raDịch vụ nhà bếpđể từ chốiVé, VàDịch vụ đặt hàngđể từ chốiĐặt hàng.**

**RSỰ KIỆN ĐỦ ĐIỀU KIỆN-TRUYỀN THÔNG DỰA TRÊN**

Có một số vấn đề liên quan đến giao tiếp giữa các dịch vụ mà bạn phải cân nhắc khi triển khai saga dựa trên biên đạo. Vấn đề đầu tiên là đảm bảo rằng người tham gia saga cập nhật cơ sở dữ liệu của mình và xuất bản sự kiện như một phần của giao dịch cơ sở dữ liệu. Mỗi bước của saga dựa trên biên đạo sẽ cập nhật cơ sở dữ liệu và xuất bản sự kiện. Ví dụ, trong Saga Tạo đơn hàng, Dịch vụ nhà bếp nhận được sự kiện Đã xác minh của người tiêu dùng, tạo Vé và xuất bản sự kiện Đã tạo vé. Điều cần thiết là việc cập nhật cơ sở dữ liệu và xuất bản sự kiện phải diễn ra nguyên tử. Do đó, để giao tiếp đáng tin cậy, những người tham gia saga phải sử dụng tin nhắn giao dịch, được mô tả trong chương 3.

Vấn đề thứ hai bạn cần cân nhắc là đảm bảo rằng người tham gia saga phải có khả năng ánh xạ từng sự kiện mà họ nhận được vào dữ liệu của riêng họ. Ví dụ, khi OrderDịch vụ nhận được sự kiện được ủy quyền bằng thẻ tín dụng, nó phải có khả năng tra cứu Đơn hàng tương ứng. Giải pháp là để người tham gia saga công bố các sự kiện có chứa ID tương quan, là dữ liệu cho phép những người tham gia khác thực hiện ánh xạ.

Ví dụ, những người tham giaTạo đơn hàng Sagacó thể sử dụngMã đơn hàngnhư một ID tương quan được truyền từ người tham gia này sang người tham gia khác.Dịch vụ kế toánquán rượu-lishes mộtThẻ tín dụng được ủy quyềnsự kiện có chứaMã đơn hàngtừVé- Đã tạosự kiện. KhiDịch vụ đặt hàngnhận được mộtThẻ tín dụng được ủy quyềnsự kiện, nó sử dụngcáiMã đơn hàngđể lấy lại tương ứngĐặt hàng. Tương tự như vậy,Dịch vụ nhà bếpsử dụngMã đơn hàngtừ sự kiện đó để lấy lại tương ứngVé.

**BLỢI ÍCH VÀ NHƯỢC ĐIỂM CỦA BIÊN ĐẠO-SAGAS DỰA TRÊN**

Các câu chuyện sử thi dựa trên vũ đạo có một số lợi ích:

* *Sự đơn giản*—Các dịch vụ sẽ công bố các sự kiện khi chúng tạo, cập nhật hoặc xóa các đối tượng kinh doanh.
* *Khớp nối lỏng lẻo*—Những người tham gia đăng ký sự kiện và không cóhiểu biết trực tiếp về nhau.

Và có một số nhược điểm:

* *Khó hiểu hơn*—Không giống như với dàn nhạc, không có một nơi duy nhấttrong mã định nghĩa saga. Thay vào đó, biên đạo phân phối việc triển khai saga giữa các dịch vụ. Do đó, đôi khi nhà phát triển khó hiểu được cách hoạt động của một saga nhất định.
* *Sự phụ thuộc tuần hoàn giữa các dịch vụ*—Những người tham gia saga đăng ký vào các sự kiện của nhau, điều này thường tạo ra sự phụ thuộc tuần hoàn. Ví dụ, nếu bạn xem xét kỹ hình 4.4, bạn sẽ thấy có sự phụ thuộc tuần hoàn, chẳng hạn nhưDịch vụ đặt hàngDịch vụ kế toánDịch vụ đặt hàng. Mặc dù điều này không hẳn là vấn đề, nhưng sự phụ thuộc tuần hoàn được coi là vấn đề thiết kế.
* *Rủi ro của sự kết hợp chặt chẽ*—Mỗi người tham gia saga cần phải đăng ký tất cả các sự kiệnảnh hưởng đến họ. Ví dụ,Dịch vụ kế toánphải đăng ký tất cả các sự kiện khiến thẻ tín dụng của người tiêu dùng bị tính phí hoặc hoàn lại tiền. Do đó, có nguy cơ là nó sẽ cần được cập nhật đồng bộ với vòng đời đơn hàng được triển khai bởiDịch vụ đặt hàng.

Biên đạo có thể hiệu quả với các saga đơn giản, nhưng vì những nhược điểm này, thường thì sử dụng phối khí sẽ tốt hơn cho các saga phức tạp hơn. Hãy cùng xem cách phối khí hoạt động.

###### Các saga dựa trên sự phối hợp

Orchestration là một cách khác để triển khai saga. Khi sử dụng orchestration, bạn định nghĩa một lớp orchestrator có trách nhiệm duy nhất là cho những người tham gia saga biết phải làm gì. Saga orchestrator giao tiếp với những người tham gia bằng cách sử dụng tương tác command/async reply-style. Để thực hiện một bước saga, nó sẽ gửi một thông báo lệnh đến một người tham gia để cho biết thao tác nào cần thực hiện. Sau khi người tham gia saga đã thực hiện thao tác, nó sẽ gửi một thông báo trả lời đến orchestrator. Sau đó, orchestrator sẽ xử lý thông báo và xác định bước saga nào cần thực hiện tiếp theo.





5

Để chỉ ra cách hoạt động của saga dựa trên dàn nhạc, trước tiên tôi sẽ mô tả một ví dụ. Sau đó, tôi sẽ mô tả cách mô hình hóa saga dựa trên dàn nhạc như các máy trạng thái. Tôi sẽ thảo luận về cách sử dụng tin nhắn giao dịch để đảm bảo giao tiếp đáng tin cậy giữa người dàn nhạc saga và những người tham gia saga. Sau đó, tôi sẽ mô tả những lợi ích và hạn chế của việc sử dụng saga dựa trên dàn nhạc.

**TÔITHỰC HIỆNCTỔNG KẾTỒRDER SAGA SỬ DỤNG ORCHESTATION**

Hình 4.6 cho thấy thiết kế của phiên bản dựa trên dàn nhạc của CreateĐặt hàngSaga. Saga được dàn dựng bởi lớp CreateOrderSaga, lớp này gọi những người tham gia saga bằng cách sử dụng yêu cầu/phản hồi không đồng bộ. Lớp này theo dõi quá trình và gửi tin nhắn lệnh đến những người tham gia saga, chẳng hạn như Kitchen Service và Consumer Service. Lớp CreateOrderSaga đọc tin nhắn trả lời từ kênh trả lời của nó và sau đó xác định bước tiếp theo, nếu có, trong saga.



Tin nhắn lệnhTrả lời tin nhắn



Dịch vụ đặt hàng

Chìa khóa

Người môi giới tin nhắn

1

Xác minh người tiêu dùng

Dịch vụ khách hàng

yêu cầu kênh

2

Người tiêu dùng đã xác minh

4

3

Tạo nên Tạo nên

đặt hàngtruyện dài vé

người điều phối

7

Phê duyệt đơn hàng nhà hàng

Tạo đơn hàng

kênh trả lời saga

Vé đã được tạo

Dịch vụ nhà bếp

yêu cầu kênh

Ủy quyền

thẻ

6

Thẻ được ủy quyền

8

Chấp thuậnđặt hàng

Dịch vụ kế toán

yêu cầu kênh

Dịch vụ đặt hàng

yêu cầu kênh

Dịch vụ kế toán

Dịch vụ nhà bếp

Dịch vụ khách hàng

**Hình 4.6 Triển khaiTạo đơn hàng Sagasử dụng phối khí.Dịch vụ đặt hàngtriển khai một bộ điều phối saga, gọi những người tham gia saga bằng cách sử dụng yêu cầu không đồng bộ/phản ứng.**

Dịch vụ đặt hàngđầu tiên tạo ra mộtĐặt hàngvà mộtTạo đơn hàng Sagangười điều phối. Sau đó, luồng cho con đường hạnh phúc như sau:

**1**Người dàn dựng saga gửi mộtXác minh người tiêu dùnglệnh choDịch vụ khách hàng.

**2** Dịch vụ khách hàngtrả lờivới mộtNgười tiêu dùng đã xác minhtin nhắn.

**3**Người dàn dựng saga gửi mộtTạo Vélệnh choDịch vụ nhà bếp.

**4** Dịch vụ nhà bếptrả lời với mộtVé đã được tạotin nhắn.

**5** Người dàn dựng saga gửi mộtThẻ ủy quyềntin nhắn đếnDịch vụ kế toán.

**6** Dịch vụ kế toántrả lời với mộtThẻ được ủy quyềntin nhắn.

**7**Người dàn dựng saga gửi mộtPhê duyệt vélệnh choDịch vụ nhà bếp.

**8** Người dàn dựng saga gửi mộtPhê duyệt đơn hànglệnh choDịch vụ đặt hàng.

Lưu ý rằng ở bước cuối cùng, người điều phối saga sẽ gửi một thông điệp lệnh đếnĐặt hàngDịch vụ, mặc dù nó là một thành phần củaDịch vụ đặt hàng. Về nguyên tắc,Tạo đơn hàng Sagacó thể chấp thuậnĐặt hàngbằng cách cập nhật trực tiếp. Nhưng để nhất quán, câu chuyện xử lýDịch vụ đặt hàngchỉ là một người tham gia bình thường.

Các sơ đồ như hình 4.6 mỗi sơ đồ mô tả một kịch bản cho một câu chuyện dài, nhưng một câu chuyện dài có khả năngcó nhiều kịch bản. Ví dụ, Create Order Saga có bốn kịch bản. Ngoài con đường hạnh phúc, saga có thể thất bại do lỗi trong Consumer Service, Kitchen Service hoặc Accounting Service. Do đó, việc mô hình hóa saga như một máy trạng thái là hữu ích, vì nó mô tả tất cả các kịch bản có thể xảy ra.

**TôiNHỮNG NGƯỜI ĐÀN ÔNG SAGA ODELING NHƯ NHỮNG CƠ QUAN NHÀ NƯỚC**

Một cách tốt để mô hình hóa một saga orchestrator là như một máy trạng thái. Một máy trạng thái bao gồm một tập hợp các trạng thái và một tập hợp các chuyển đổi giữa các trạng thái được kích hoạt bởi các sự kiện. Mỗi chuyển đổi có thể có một hành động, đối với một saga là việc gọi một người tham gia saga. Các chuyển đổi giữa các trạng thái được kích hoạt bằng cách hoàn thành một giao dịch cục bộ do một người tham gia saga thực hiện. Trạng thái hiện tại và kết quả cụ thể của giao dịch cục bộ xác định quá trình chuyển đổi trạng thái và hành động nào, nếu có, cần thực hiện. Ngoài ra còn có các chiến lược kiểm tra hiệu quả cho các máy trạng thái. Do đó, việc sử dụng một mô hình máy trạng thái giúp thiết kế, triển khai và kiểm tra saga dễ dàng hơn.

Hình 4.7 cho thấy mô hình máy trạng thái cho Create Order Saga. Máy trạng thái này bao gồm nhiều trạng thái, bao gồm:

* Xác minh người tiêu dùng—Trạng thái ban đầu. Khi ở trạng thái này, saga đang chờ đợiDịch vụ khách hàngđể xác minh rằng người tiêu dùng có thể đặt hàng.
* Tạo Vé—Câu chuyện đang chờ đợi câu trả lờiTạo Véyêu cầu.
* Thẻ ủy quyền—Đang chờ đợiDịch vụ kế toánđể ủy quyền cho thẻ tín dụng của người tiêu dùng.
* Đơn hàng đã được chấp thuận—Trạng thái cuối cùng cho thấy câu chuyện đã kết thúc thành công.
* Đơn hàng bị từ chối—Trạng thái cuối cùng chỉ ra rằngĐặt hàngđã bị một trong những người tham gia từ chối.

/Gửi VerifyConsumer

ConsumerVerificationFailed/ gửi RejectOrder



ConsumerVerified/ gửi CreateRestaurantOrder



Đơn hàng bị từ chối

Tạo vé

Từ chối đơn hàng

Xác minh người tiêu dùng

Đã tạo ticket/gửi AuthorizeCard

Thẻ ủy quyền

Thẻ được ủy quyền/gửi ApproveTicket

Phê duyệt vé

Vé đã được chấp thuận/gửi ApproveOrder

Xác thực thẻ không thành công/ gửi RejectTicket

Tạo ticket không thành công/gửi RejectOrder

Từ chối vé

Đang phê duyệt đơn hàng

Đơn hàng đã được chấp thuận

**Hình 4.7 Mô hình máy trạng thái choTạo đơn hàng Saga**

Đơn hàng đã được chấp thuận

Máy trạng thái cũng xác định nhiều trạng thái chuyển đổi. Ví dụ, trạng tháimáy chuyển đổi từTạo Vétrạng thái cho một trong haiThẻ ủy quyềnhoặcĐơn hàng bị từ chốitrạng thái. Nó chuyển sangThẻ ủy quyềntrạng thái khi nhận được phản hồi thành công choTạo Vélệnh. Ngoài ra, nếuPhòng bếpDịch vụkhông thể tạo raVé, máy trạng thái chuyển sangĐơn hàng bị từ chốitình trạng.

Hành động ban đầu của máy trạng thái là gửiXác minh người tiêu dùnglệnh choDịch vụ khách hàng. Phản hồi từDịch vụ khách hàngkích hoạt trạng thái chuyển tiếp tiếp theo. Nếu người tiêu dùng đã được xác minh thành công, saga sẽ tạo raVévà chuyển giaovị trí đểTạo Vétrạng thái. Nhưng nếu xác minh người tiêu dùng không thành công, saga sẽ từ chốiĐặt hàngvà chuyển tiếp đếnTừ chối đơn hàngtrạng thái. Máy trạng thái trải qua nhiều quá trình chuyển đổi trạng thái khác nhau, được điều khiển bởi phản hồi từ những người tham gia saga, cho đến khi đạt đến trạng thái cuối cùng củaĐơn hàng đã được chấp thuậnhoặcĐơn hàng bị từ chối.

**SAGA ORCHESTRATION VÀ GIAO DỊCHTIN NHẮN**

Mỗi bước của một saga dựa trên dàn nhạc bao gồm một dịch vụ cập nhật cơ sở dữ liệuvà xuất bản một thông điệp. Ví dụ, Order Service duy trì một Order và một Create Order Saga orchestrator và gửi một thông điệp đến người tham gia saga đầu tiên. Một người tham gia saga, chẳng hạn như Kitchen Service, xử lý một thông điệp lệnh bằng cách cập nhật cơ sở dữ liệu của mình và gửi một thông điệp trả lời. Order Service xử lý thông điệp trả lời của người tham gia bằng cách cập nhật trạng thái của saga orchestrator và gửi một thông điệp lệnh đến người tham gia saga tiếp theo. Như đã mô tả trong chương 3, một dịch vụ phải sử dụng nhắn tin giao dịch để cập nhật cơ sở dữ liệu và xuất bản thông điệp một cách nguyên tử. Sau đó trong phần 4.4, tôi sẽ mô tả chi tiết hơn về việc triển khai Create Order Saga orchestrator, bao gồm cách nó sử dụng nhắn tin giao dịch.

Hãy cùng xem xét những lợi ích và hạn chế của việc sử dụng Saga Orchestration.

**BLỢI ÍCH VÀ NHƯỢC ĐIỂM CỦA ORCHESTRATION-SAGAS DỰA TRÊN**

Các câu chuyện sử thi theo hình thức dàn nhạc có một số lợi ích:

* *Phụ thuộc đơn giản hơn*—Một lợi ích của việc phối hợp lànó không giới thiệu các phụ thuộc tuần hoàn. Người điều phối saga triệu hồi những người tham gia saga, nhưng những người tham gia không triệu hồi người điều phối. Do đó, người điều phối phụ thuộc vào những người tham gia nhưng không ngược lại, và do đó không có sự phụ thuộc tuần hoàn.
* *Ít liên kết*—Mỗi dịch vụ đều triển khai một API được người điều phối gọi đến, do đó không cần biết về các sự kiện do những người tham gia saga công bố.
* *Cải thiện việc tách biệt các mối quan tâmvà đơn giản hóa logic kinh doanh*—Logic phối hợp saga được bản địa hóa trong bộ điều phối saga. Các đối tượng miền đơn giản hơn và không có kiến ​​thức về saga mà chúng tham gia. Ví dụ, khi sử dụng phối hợp,Đặt hànglớp không có kiến ​​thức về bất kỳ sagas nào, vì vậy nó có mô hình máy trạng thái đơn giản hơn. Trong quá trình thực hiệnTạo đơn hàng Saga, nó chuyển tiếp trực tiếp từĐANG CHỜ PHÊ DUYỆTtrạng thái đến TÁN THÀNHtrạng thái. CácĐặt hànglớp không có bất kỳ trạng thái trung gian nào tương ứng với các bước của saga. Do đó, công việc kinh doanh đơn giản hơn nhiều.

Orchestration cũng có một nhược điểm: rủi ro tập trung quá nhiều logic kinh doanh vào orchestrator. Điều này dẫn đến một thiết kế mà orchestrator thông minh sẽ chỉ cho các dịch vụ ngu ngốc biết phải làm những hoạt động nào. May mắn thay, bạn có thể tránh được vấn đề này bằng cách thiết kế orchestrator chỉ chịu trách nhiệm về trình tự và không chứa bất kỳ logic kinh doanh nào khác.

Tôi khuyên bạn nên sử dụng orchestration cho tất cả các saga trừ những saga đơn giản nhất. Việc triển khai logic phối hợp cho saga của bạn chỉ là một trong những vấn đề thiết kế mà bạn cần giải quyết. Một vấn đề khác, có lẽ là thách thức lớn nhất mà bạn sẽ phải đối mặt khi sử dụng saga, là xử lý tình trạng thiếu cô lập. Hãy cùng xem xét vấn đề đó và cách giải quyết.

#### Xử lý tình trạng thiếu cô lập

Chữ I trong ACID là viết tắt của isolation (cô lập). Thuộc tính cô lập của các giao dịch ACID đảm bảo rằng kết quả của việc thực hiện nhiều giao dịch đồng thời giống như khi chúng được thực hiện theo một thứ tự tuần tự nào đó. Cơ sở dữ liệu tạo ra ảo giác rằng mỗi giao dịch ACID có quyền truy cập độc quyền vào dữ liệu. Isolation giúp viết logic nghiệp vụ thực hiện đồng thời dễ dàng hơn nhiều.

Thách thức khi sử dụng saga là chúng thiếu tính chất cô lập của các giao dịch ACID. Đó là vì các bản cập nhật được thực hiện bởi mỗi giao dịch cục bộ của saga sẽ ngay lập tức hiển thị với các saga khác sau khi giao dịch đó được xác nhận. Hành vi này có thể gây ra hai vấn đề. Đầu tiên, các saga khác có thể thay đổi dữ liệu mà saga truy cập trong khi nó đang thực thi. Và các saga khác có thể đọc dữ liệu của nó trước khi saga hoàn tất các bản cập nhật của nó và do đó có thể bị phơi bày với dữ liệu không nhất quán. Trên thực tế, bạn có thể coi một saga là ACD:

* + - *Nguyên tử*—Việc triển khai saga đảm bảo rằng tất cả các giao dịch được thực hiện hoặc tất cả các thay đổi được hoàn tác.
    - *Sự nhất quán*—Tính toàn vẹn tham chiếu trong một dịch vụ được xử lý bởi cơ sở dữ liệu cục bộ. Tính toàn vẹn tham chiếu giữa các dịch vụ được xử lý bởi các dịch vụ.
    - *Độ bền*—Được xử lý bởi cơ sở dữ liệu cục bộ.

Sự thiếu cô lập này có khả năng gây ra những gì mà tài liệu cơ sở dữ liệu gọi là bất thường. Bất thường là khi một giao dịch đọc hoặc ghi dữ liệu theo cách mà nó sẽ không làm nếu các giao dịch được thực hiện từng cái một. Khi bất thường xảy ra, kết quả của việc thực hiện các saga đồng thời sẽ khác so với khi chúng được thực hiện tuần tự.

Trên bề mặt, việc thiếu sự cô lập nghe có vẻ không khả thi. Nhưng trên thực tế, các nhà phát triển thường chấp nhận giảm sự cô lập để đổi lấy hiệu suất cao hơn. RDBMS cho phép bạn chỉ định mức độ cô lập cho mỗi giao dịch ([https://dev.mysql](https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html)

[.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html](https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html)). Mức cô lập mặc định thường là mức cô lập yếu hơn mức cô lập hoàn toàn, còn được gọi là giao dịch có thể tuần tự hóa. Giao dịch cơ sở dữ liệu trong thế giới thực thường khác với định nghĩa trong sách giáo khoa về giao dịch ACID.

Phần tiếp theo thảo luận về một tập hợp các chiến lược thiết kế saga giải quyết vấn đề thiếu cô lập. Các chiến lược này được gọi là biện pháp đối phó. Một số biện pháp đối phó triển khai cô lập ở cấp ứng dụng. Các biện pháp đối phó khác làm giảm rủi ro kinh doanh do thiếu cô lập. Bằng cách sử dụng các biện pháp đối phó, bạn có thể viết logic kinh doanh dựa trên saga hoạt động chính xác.

Tôi sẽ bắt đầu phần này bằng cách mô tảnhững bất thường do thiếu sự cô lập gây ra. Sau đó, tôi sẽ nói về các biện pháp đối phó có thể loại bỏ những bất thường đó hoặc giảm thiểu rủi ro kinh doanh của họ.

###### Tổng quan về các bất thường

Việc thiếu sự cô lập có thể gây ra ba điều bất thường sau:

* + - * *Đã mất bản cập nhật*—Một saga ghi đè mà không đọc những thay đổi do saga khác thực hiện.
      * *Đọc bẩn*—Một giao dịch hoặc saga sẽ đọc các bản cập nhật được thực hiện bởi một saga chưa hoàn tất các bản cập nhật đó.
      * *Đọc mờ/không lặp lại*—Hai bước khác nhau của một saga đọc cùng một dữ liệu và đưa ra kết quả khác nhau vì một saga khác đã thực hiện cập nhật.

Cả ba bất thường đều có thể xảy ra, nhưng hai bất thường đầu tiên là phổ biến nhất và khó khăn nhất. Chúng ta hãy xem xét hai loại bất thường đó, bắt đầu với các bản cập nhật bị mất.

**LCẬP NHẬT OST**

Một sự bất thường về mất bản cập nhật xảy ra khi một saga ghi đè lên bản cập nhật do saga khác thực hiện. Ví dụ, hãy xem xét tình huống sau:

**1**Bước đầu tiên củaTạo đơn hàng Sagatạo ra mộtĐặt hàng.

**2** Trong khi câu chuyện đó đang diễn ra,Hủy đơn hàng Sagahủy bỏĐặt hàng.

**3**Bước cuối cùng củaTạo đơn hàng Sagachấp thuậnĐặt hàng.

Trong trường hợp này, Create Order Saga bỏ qua bản cập nhật do Cancel Order Saga thực hiện và ghi đè lên. Do đó, ứng dụng FTGO sẽ gửi đơn hàng mà khách hàng đã hủy. Sau trong phần này, tôi sẽ chỉ cách ngăn chặn việc mất bản cập nhật.

**DIRTY ĐỌC**

Đọc bẩn xảy ra khi một saga đọc dữ liệu đang được saga khác cập nhật. Ví dụ, hãy xem xét phiên bản kho ứng dụng FTGO nơi người tiêu dùng có hạn mức tín dụng. Trong ứng dụng này, một saga hủy đơn hàng bao gồm các giao dịch sau:

* + - * Dịch vụ khách hàng-Tăngtín dụng có sẵn.
      * Dịch vụ đặt hàng—Thay đổi trạng thái củaĐặt hàngđể hủy bỏ.
      * Dịch vụ giao hàng—Hủy việc giao hàng.

Hãy tưởng tượng một kịch bản xen kẽ việc thực hiệnHủy đơn hàngVàTạo đơn hàng Sagas, vàHủy đơn hàng Sagađược khôi phục vì đã quá muộn để hủy giao hàng. Có thể là trình tự các giao dịch gọiDịch vụ khách hàngnhư sau:

**1**Hủy đơn hàngTruyện dài—Tăng lượng tín dụng có sẵn.

**2**Tạo đơn hàng Saga-Giảm bớttín dụng có sẵn.

**3** Hủy đơn hàng Saga—Một giao dịch bù trừ làm giảm lượng tín dụng khả dụng.

Trong trường hợp này, Create Order Saga sẽ đọc sai hạn mức tín dụng khả dụng, cho phép người tiêu dùng đặt hàng vượt quá hạn mức tín dụng của họ. Có khả năng đây là rủi ro không thể chấp nhận được đối với doanh nghiệp.

Hãy cùng xem cách ngăn chặn hiện tượng này và các loại bất thường khác ảnh hưởng đến ứng dụng.

###### Các biện pháp đối phó để xử lý tình trạng thiếu cô lập

Mô hình giao dịch saga là ACD và việc thiếu sự cô lập của nó có thể dẫn đến các bất thường khiến ứng dụng hoạt động không bình thường. Trách nhiệm của nhà phát triển là viết saga theo cách ngăn ngừa các bất thường hoặc giảm thiểu tác động của chúng đối với doanh nghiệp. Điều này có vẻ như là một nhiệm vụ khó khăn, nhưng bạn đã thấy một ví dụ vềchiến lược màngăn ngừa các bất thường. Việc sử dụng các trạng thái \*\_PENDING của Lệnh, chẳng hạn như APPROVAL

\_PENDING, là một ví dụ về một chiến lược như vậy. Các Saga cập nhật Order, chẳng hạn như

Tạo đơn hàng Saga, bắt đầu bằng cách thiết lập trạng thái của mộtĐặt hàngĐẾN\*\_CHƯA GIẢI QUYẾT. Các\*\_CHƯA GIẢI QUYẾTtrạng thái cho biết các giao dịch khác rằngĐặt hàngđang được cập nhật bằng một câu chuyện và phải hành động phù hợp.

Việc sử dụng trạng thái \*\_PENDING của Order là một ví dụ về những gì mà bài báo năm 1998 “Thuộc tính ACID ngữ nghĩa trong nhiều cơ sở dữ liệu sử dụng các lệnh gọi thủ tục từ xa và truyền bá cập nhật” của Lars Frank và Torben U. Zahle gọi là biện pháp đối phó khóa ngữ nghĩa ([https://dl.acm.org/cite.cfm?id=284472.284478](https://dl.acm.org/citation.cfm?id=284472.284478)). Bài báo mô tảcách xử lý tình trạng thiếu cô lập giao dịch trong các kiến ​​trúc đa cơ sở dữ liệu không sử dụng giao dịch phân tán. Nhiều ý tưởng của bài viết này hữu ích khi thiết kế saga. Bài viết mô tả một tập hợp các biện pháp đối phó để xử lý các bất thường do thiếu cô lập, nhằm ngăn ngừa một hoặc nhiều bất thường hoặc giảm thiểu tác động của chúng đối với doanh nghiệp. Các biện pháp đối phó được mô tả trong bài viết này như sau:

* + - * *Khóa ngữ nghĩa*—Khóa cấp ứng dụng.
      * *Cập nhật giao hoán*—Thiết kế các hoạt động cập nhật có thể thực hiện theo bất kỳ thứ tự nào.
      * *Quan điểm bi quan*—Sắp xếp lại các bước của một câu chuyện để giảm thiểu rủi ro kinh doanh.
      * *Đọc lại giá trị*—Ngăn chặn việc ghi bẩn bằng cách đọc lại dữ liệu để xác minh rằng dữ liệu không thay đổi trước khi ghi đè.
      * *Tập tin phiên bản*—Ghi lại các cập nhật vào bản ghi để có thể sắp xếp lại chúng.
      * *Theo giá trị*—Sử dụng rủi ro kinh doanh của mỗi yêu cầu để lựa chọn cơ chế đồng thời một cách linh hoạt.

Ở phần sau của phần này, tôi sẽ mô tả từng biện pháp đối phó, nhưng trước tiên tôi muốn giới thiệu một số thuật ngữ để mô tả cấu trúc của một câu chuyện, hữu ích khi thảo luận về các biện pháp đối phó.

**TCẤU TRÚC CỦA MỘT SAGA**

Bài báo về biện pháp đối phó được đề cập trong phần cuối cùng định nghĩa một mô hình hữu ích cho cấu trúc của một saga. Trong mô hình này, được thể hiện trong hình 4.8, một saga bao gồm ba loại giao dịch:

* + - * *Giao dịch có thể bù trừ*—Các giao dịch có khả năng có thể được hoàn lại bằng cách sử dụngmột giao dịch bù trừ.
      * *Giao dịch trục*—Điểm đi/không đi trong một saga. Nếu giao dịch trục cam kết, saga sẽ chạy cho đến khi hoàn tất. Giao dịch trục có thể là giao dịch không thể bồi thường hoặc không thể thử lại. Ngoài ra, nó có thể là giao dịch cuối cùng có thể bồi thường hoặc giao dịch đầu tiên có thể thử lại.
        + *Giao dịch có thể truy xuất*—Các giao dịch theo sau giao dịch trục và được bảo đảm-mong muốn thành công.

**Có thể đền bù giao dịch:**

**Phải hỗ trợ quay lại**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bước chân | Dịch vụ | Giao dịch | Giao dịch bồi thường |  |
|  | 1 | Dịch vụ đặt hàng | tạo đơn hàng() | từ chối đơn hàng() |  |
| 2 | Dịch vụ khách hàng | xác minhConsumerDetails() | - |  |
| 3 | Dịch vụ nhà bếp | tạoTicket() | từ chốiTicket() |  |
|  | 4 | Dịch vụ kế toán | ủy quyền thẻ tín dụng() | - |  |
|  | 5 | Dịch vụ đặt món ăn nhà hàng | chấp thuậnRestaurantOrder() | - |  |
| 6 | Dịch vụ đặt hàng | chấp thuận đơn hàng() | - |  |

**Giao dịch trục:**

**Câu chuyện của đi/không đi giao dịch. Nếu thành công thì câu chuyện sẽ đi đến hồi kết.**

**Giao dịch có thể thu hồi: Đảm bảo hoàn tất**

**Hình 4.8 Một saga bao gồm ba loại giao dịch khác nhau: giao dịch có thể bù trừ, có thể được hoàn nguyên, do đó có giao dịch bù trừ, giao dịch trục, là điểm đi/không đi của saga và giao dịch có thể thử lại, là giao dịch không cần phảiđược khôi phục và đảm bảo hoàn tất.**

TrongTạo đơn hàng Saga, cáctạo đơn hàng(),xác minhConsumerDetails(), Vàtạo nên-Vé()các bước là các giao dịch có thể đền bù.tạo đơn hàng()Vàtạo-Vé()các giao dịch có các giao dịch bù trừ hoàn tác các bản cập nhật của chúng.xác minhConsumerDetails()giao dịch chỉ đọc, do đó không cần phải bù trừgiao dịch. Cácủy quyền thẻ tín dụng()giao dịch là trục xoay của câu chuyện nàygiao dịch. Nếu thẻ tín dụng của người tiêu dùng có thể được chấp thuận, thì câu chuyện này chắc chắn sẽ hoàn thành.phê duyệtVé()Vàchấp thuận đơn hàng()các bước là các giao dịch có thể thử lại theo sau giao dịch trục.

Sự khác biệt giữa các giao dịch có thể bồi thường và các giao dịch có thể thử lại đặc biệt quan trọng. Như bạn sẽ thấy, mỗi loại giao dịch đóng một vai trò khác nhau trong các biện pháp đối phó. Chương 13 nêu rằng khi di chuyển sang các dịch vụ vi mô, đôi khi khối đơn phải tham gia vào các sagas và sẽ đơn giản hơn đáng kể nếu khối đơn chỉ cần thực hiện các giao dịch có thể thử lại.

Bây giờ chúng ta hãy xem xét từng biện pháp đối phó, bắt đầu với biện pháp đối phó khóa ngữ nghĩa.

**CTHỜI HẠN:SKHÓA EMANTIC**

Khi sử dụng biện pháp đối phó khóa ngữ nghĩa, giao dịch có thể bù trừ của saga sẽ đặt cờ trong bất kỳ bản ghi nào mà nó tạo hoặc cập nhật. Cờ chỉ ra rằng bản ghi

không được cam kết và có khả năng thay đổi. Cờ có thể là khóa ngăn các giao dịch khác truy cập vào bản ghi hoặc cảnh báo cho biết các giao dịch khác nên xử lý bản ghi đó một cách đáng ngờ. Nó được xóa bằng giao dịch có thể thử lại—saga đang hoàn tất thành công—hoặc bằng giao dịch bù trừ: saga đang quay lại.

CácTrật tự.trạng tháitrường là một ví dụ tuyệt vời về khóa ngữ nghĩa.\*\_CHƯA GIẢI QUYẾTcác tiểu bang, chẳng hạn nhưĐANG CHỜ PHÊ DUYỆTVàĐANG XEM XÉT, thực hiện một khóa ngữ nghĩa. Họ nói với các saga khác rằng truy cập vào mộtĐặt hàngrằng một saga đang trong quá trình cập nhậtĐặt hàng.Ví dụ, bước đầu tiên củaTạo đơn hàng Saga, là một giao dịch có thể bù trừ, tạo ra mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtrạng thái. Bước cuối cùng củaTạo đơn hàng Saga, là một giao dịch có thể truy xuất lại, thay đổi trường thànhTÁN THÀNH. Một giao dịch bù trừ thay đổi trường thànhVẬT BỊ LOẠI BỎ.

Quản lý khóa chỉ là một nửa vấn đề. Bạn cũng cần quyết định từng trường hợp cụ thể về cách một saga nên xử lý một bản ghi đã bị khóa. Hãy xem xét, ví dụví dụ, lệnh hệ thống cancelOrder(). Một máy khách có thể gọi thao tác này để hủy một Đơn hàng đang ở trạng thái APPROVAL\_PENDING.

Có một số cách khác nhau để xử lý tình huống này. Một tùy chọn là lệnh hệ thống cancel-Order() sẽ thất bại và yêu cầu máy khách thử lại sau. Lợi ích chính của cách tiếp cận này là nó dễ triển khai. Tuy nhiên, nhược điểm là nó làm cho máy khách phức tạp hơn vì phải triển khai logic thử lại.

Một tùy chọn khác là cancelOrder() sẽ chặn cho đến khi khóa được giải phóng. Một lợi ích của việc sử dụng khóa ngữ nghĩa là về cơ bản chúng tái tạo sự cô lập do các giao dịch ACID cung cấp. Các Saga cập nhật cùng một bản ghi được tuần tự hóa, giúp giảm đáng kể công sức lập trình. Một lợi ích khác là chúng loại bỏ gánh nặng thử lại khỏi máy khách. Nhược điểm là ứng dụng phải quản lý khóa. Nó cũng phải triển khai thuật toán phát hiện bế tắc thực hiện khôi phục một saga để phá vỡ bế tắc và thực hiện lại.

**CTHỜI HẠN: CCẬP NHẬT OMUTATIC**

Một biện pháp đối phó đơn giản là thiết kế các hoạt động cập nhật có tính giao hoán. Các hoạt động có tính giao hoán nếu chúng có thể được thực hiện theo bất kỳ thứ tự nào. MộtCác hoạt động ghi nợ() và ghi có() của tài khoản có tính giao hoán (nếu bạn bỏ qua các lần kiểm tra thấu chi). Biện pháp đối phó này hữu ích vì nó loại bỏ các bản cập nhật bị mất.

Ví dụ, hãy xem xét một kịch bản trong đó một câu chuyện cầnđược hoàn lại sau khi giao dịch có thể đền bù đã ghi nợ (hoặc ghi có) vào tài khoản. Giao dịch đền bù có thể chỉ cần ghi có (hoặc ghi nợ) vào tài khoản để hoàn tác bản cập nhật. Không có khả năng ghi đè lên các bản cập nhật do các saga khác thực hiện.

**CTHỜI HẠN: PQUAN ĐIỂM ESSIMISTIC**

Một cách khác để giải quyết tình trạng thiếu cô lập là biện pháp đối phó theo quan điểm bi quan. Nó sắp xếp lại các bước của một saga để giảm thiểu rủi ro kinh doanh do đọc bẩn. Ví dụ, hãy xem xét kịch bản trước đó được sử dụng để mô tả sự bất thường của đọc bẩn. Trong kịch bản đó, Create Order Saga đã thực hiện đọc bẩn tín dụng khả dụng và tạo ra

đơn hàng vượt quá hạn mức tín dụng tiêu dùng. Để giảm thiểu rủi ro xảy ra điều đó,biện pháp đối phó này sẽ sắp xếp lại Saga Hủy Lệnh:

**1** Dịch vụ đặt hàng—Thay đổi trạng thái củaĐặt hàngđể hủy bỏ.

**2** Dịch vụ giao hàng—Hủy việc giao hàng.

**3**Dịch vụ khách hàng-Tăngtín dụng có sẵn.

Trong phiên bản được sắp xếp lại của câu chuyện này, số tín dụng khả dụng được tăng lên trong một giao dịch có thể thử lại, giúp loại bỏ khả năng đọc sai.

**CTHỜI HẠN: RGIÁ TRỊ EREAD**

Biện pháp đối phó giá trị đọc lại ngăn chặn việc mất các bản cập nhật. Một saga sử dụng biện pháp đối phó này sẽ đọc lại một bản ghi trước khi cập nhật, xác minh rằng nó không thay đổi, sau đó cập nhật bản ghi. Nếu bản ghi đã thay đổi, saga sẽ hủy bỏ và có thể khởi động lại. Biện pháp đối phó này là một dạng của mẫu Optimistic Offline Lock ([https://martinfowler](https://martinfowler.com/eaaCatalog/optimisticOfflineLock.html)

[.com/eaaCatalog/optimisticOfflineLock.html](https://martinfowler.com/eaaCatalog/optimisticOfflineLock.html)).

Saga Create Order có thểsử dụng biện pháp đối phó này để xử lý tình huống trong đó Lệnh bị hủy trong khi đang trong quá trình phê duyệt. Giao dịch phê duyệt Lệnh xác minh rằng Lệnh không thay đổi kể từ khi tạo trước đó trong saga. Nếu không thay đổi, giao dịch sẽ phê duyệt Lệnh. Nhưng nếu Lệnh đã bị hủy, giao dịch sẽ hủy bỏ saga, khiến các giao dịch bù trừ của nó được thực hiện.

**CTHỜI HẠN:TỆP PHIÊN BẢN**

Biện pháp đối phó tệp phiên bản được đặt tên như vậy vì nó ghi lại các hoạt động được thực hiện trên một bản ghi để có thể sắp xếp lại chúng. Đây là một cách để biến các hoạt động không giao hoán thành các hoạt động giao hoán. Để xem biện pháp đối phó này hoạt động như thế nào, hãyxem xét một kịch bản trong đó Create Order Saga thực hiện đồng thời với Cancel Order Saga. Trừ khi các saga sử dụng biện pháp đối phó khóa ngữ nghĩa, có thể Cancel Order Saga sẽ hủy quyền ủy quyền thẻ tín dụng của người tiêu dùng trước khi Create Order Saga ủy quyền thẻ.

Một cách choDịch vụ kế toánđể xử lý các yêu cầu không theo thứ tự này là để nó ghi lại các hoạt động khi chúng đến và sau đó thực hiện chúng theo đúng thứ tự. Trong trường hợp này, trước tiên nó sẽ ghi lạiHủy bỏ ủy quyềnyêu cầu. Sau đó, khiDịch vụ kế toánnhận được sau đóThẻ ủy quyềnyêu cầu, nó sẽ nhận thấyrằng nó đã nhận đượcHủy bỏ ủy quyềnyêu cầu và bỏ qua việc ủy ​​quyền thẻ tín dụng.

**CTHỜI HẠN:BGIÁ TRỊ Y**

Biện pháp đối phó cuối cùng là biện pháp đối phó theo giá trị. Đây là chiến lược để lựa chọn cơ chế đồng thời dựa trên rủi ro kinh doanh. Một ứng dụng sử dụng biện pháp đối phó này sử dụng các thuộc tính của từng yêu cầu để quyết định giữa việc sử dụng sagas và các giao dịch phân tán. Nó thực hiện các yêu cầu rủi ro thấp bằng sagas, có thể áp dụng các biện pháp đối phó được mô tả trong phần trước. Nhưng nó thực hiện các yêu cầu rủi ro cao liên quan đến, ví dụ, số tiền lớn, bằng cách sử dụng các giao dịch phân tán.

Chiến lược này cho phép ứng dụng có thể cân nhắc linh hoạt giữa rủi ro kinh doanh, tính khả dụng và khả năng mở rộng.

Có khả năng bạn sẽ cần sử dụng một hoặc nhiều biện pháp đối phó này khi triển khai sagas trong ứng dụng của mình. Hãy cùng xem xét thiết kế và triển khai chi tiết của Create Order Saga, sử dụng biện pháp đối phó khóa ngữ nghĩa.

#### Thiết kế của Dịch vụ Đặt hàng vàSaga Tạo Lệnh

Bây giờ chúng ta đã xem xét các vấn đề thiết kế và triển khai saga khác nhau, hãy cùng xem mộtVí dụ. Hình 4.9 cho thấy thiết kế của Order Service. Logic kinh doanh của dịch vụ bao gồm các lớp logic kinh doanh truyền thống, chẳng hạn như Order Service và Order

**Xử lý các lệnh được gửi bởi Create Order Saga đến Order Service**

**Xử lý phản hồi từ những người tham gia saga**



tạo đơn hàng() hủy đơn hàng()

...

Yêu cầu dịch vụ đặt hàng

CreateOrderSagaTrả lời

Kế toánDịch vụYêu cầu

Yêu cầu dịch vụ của người tiêu dùng

Yêu cầu dịch vụ nhà bếp

Nhà xuất bản tin nhắn lệnh

lưu() tìmById()

...

Kho lưu trữ đơn hàng

createOrder() hủy đơn hàng() chấp thuận đơn hàng() từ chối đơn hàng()

...

Dịch vụ đặt hàng

Bộ điều khiển dịch vụ đặt hàng

Proxy dịch vụ nhà bếp

Đặt hàng

Proxy dịch vụ đặt hàng

Tạo đơn hàng Saga

Trả lời người tiêu dùng

Trình xử lý lệnh đặt hàng

**Xác định API nhắn tin của Dịch vụ đặt hàng nhà hàng**

**Gửilệnh cho người tham gia saga**

**Người biên soạn cho Create Order Saga**

**Hình 4.9 Thiết kế củaDịch vụ đặt hàngvà những câu chuyện của nó**

thực thể. Ngoài ra còn có các lớp dàn nhạc saga, bao gồmTạo đơn hàngSagalớp học, nơi dàn dựngTạo đơn hàng Saga. Ngoài ra, bởi vìDịch vụ đặt hàngtham gia vào nósaga riêng của nó, nó có mộtTrình xử lý lệnhlớp bộ điều hợp xử lý các thông điệp lệnh bằng cách gọiDịch vụ đặt hàng.

Một số phần của Order Service có vẻ quen thuộc. Như trong một ứng dụng truyền thống, cốt lõi của logic kinh doanh được triển khai bởi các lớp OrderService, Order và Order-Repository. Trong chương này, tôi sẽ mô tả ngắn gọn các lớp này. Tôi sẽ mô tả chúng chi tiết hơn trong chương 5.

Những gì ít quen thuộc hơn vềDịch vụ đặt hànglà các lớp liên quan đến saga. Dịch vụ này vừa là người dàn dựng saga vừa là người tham gia saga.Dịch vụ đặt hàngcó một số dàn nhạc saga-trators, chẳng hạn nhưTạo đơn hàngSaga. Những người dàn dựng saga gửi tin nhắn lệnh đến mộtngười tham gia saga sử dụng lớp proxy của người tham gia saga, chẳng hạn nhưDịch vụ nhà bếpProxyVàDịch vụ đặt hàngProxy.Proxy của người tham gia saga định nghĩa API nhắn tin của người tham gia saga.Dịch vụ đặt hàngcũng có mộtTrình xử lý lệnhlớp, xử lý com-tin nhắn lệnhđược gửi bởi sagas đếnDịch vụ đặt hàng.

Chúng ta hãy xem xét chi tiết hơn về thiết kế, bắt đầu với lớp OrderService.

###### Lớp OrderService

Lớp OrderService là một dịch vụ miền được gọi bởi lớp API của dịch vụ. Lớp này chịu trách nhiệm tạo và quản lý các đơn hàng. Hình 4.10 cho thấy OrderService và một số cộng tác viên của nó. OrderService tạo và cập nhật Orders, gọi OrderRepository để duy trì Orders và tạo saga, chẳng hạn như CreateOrderSaga, bằng cách sử dụng SagaManager. Lớp SagaManager là một trong những lớp do khung Eventuate Tram Saga cung cấp, đây là một khung để viết các trình biên soạn và người tham gia saga, và sẽ được thảo luận một chút sau trong phần này.

tạo nên(..)

Quản lý Saga

lưu() tìmMột()

...

Kho lưu trữ đơn hàng

tạo đơn hàng()

...

Dịch vụ đặt hàng

Đặt hàng

Tạo đơn hàng Saga

**Hình 4.10Dịch vụ đặt hàngtạo và cập nhậtĐơn hàng, gọi làKho lưu trữ đơn hàngđể kiên trìĐơn hàngvà tạo ra các câu chuyện dài, bao gồmcáiTạo đơn hàngSaga.**

Tôi sẽ thảo luận về lớp này chi tiết hơn trong chương 5. Bây giờ, chúng ta hãy tập trung vàotạo- Order()phương pháp. Danh sách sau đây cho thấyDịch vụ đặt hàng'Stạo đơn hàng()phương pháp.Phương pháp này đầu tiên tạo ra mộtĐặt hàngvà sau đó tạo ra mộtTạo đơn hàngSagađể xác nhận đơn hàng.

**Liệt kê 4.1Dịch vụ đặt hànglớp và nótạo đơn hàng()phương pháp**

@Giao dịch

lớp công khai OrderService {

@Autowired

**Đảm bảo rằng các phương thức dịch vụ mang tính giao dịch.**

riêng tưSagaManager<CreateOrderSagaState> createOrderSagaManager;

@Autowired

OrderRepository riêng tư orderRepository;

@Autowired

riêng tư DomainEventPublisher eventPublisher;

công khai Order createOrder(OrderDetails orderDetails) {

...

**Tạo ra**

**Đặt hàng.**

ResultWithEvents<Order> orderAndEvents = Order.createOrder(...); Đơn hàng order = orderAndEvents.result;

orderRepository.save(thứ tự);

eventPublisher.publish(Lớp đơn hàng,

**Duy trì trật tự**

**trong cơ sở dữ liệu.**

**Xuất bản tên miền**

Long.toString(order.getId()),orderAndEvents.events);

**sự kiện.**

Dữ liệu CreateOrderSagaState =

mới CreateOrderSagaState(order.getId(), orderDetails); createOrderSagaManager.create(dữ liệu, Order.class, order.getId());

trả lại đơn hàng;

}

**Tạo CreateOrderSaga.**

...

}

Cáctạo đơn hàng()phương pháp tạo ra mộtĐặt hàngbằng cách gọi phương thức nhà máyĐặt hàng

.createOrder(). Sau đó nó vẫn tồn tạiĐặt hàngsử dụngKho lưu trữ đơn hàng, là kho lưu trữ dựa trên JPA. Nó tạo raTạo đơn hàngSagabằng cách gọiSagaManager. tạo(),đi quaMộtTạo đơn hàng SagaStatechứa ID của mục mới được lưuĐặt hàngvàChi tiết đơn hàng. CácQuản lý Sagakhởi tạo trình điều phối saga, khiến nó gửi tin nhắn lệnh đến người tham gia saga đầu tiên và lưu trình điều phối saga trong cơ sở dữ liệu.

Chúng ta hãy nhìn xemtạiTạo đơn hàngSagavà các lớp liên quan.

**SagaManager xử lýduy trì một saga, gửi các tin nhắn lệnh mà nó tạo ra, đăng ký để trả lời tin nhắn và sử dụng saga để xử lý các câu trả lời.**



**Trạng thái của một saga**

this.sagaĐịnh nghĩa=bước chân()

.withCompensation(...)

.bước chân()

.invokeNgười tham gia(...)

.bước chân()

.invokeNgười tham gia(...)

.onTrả lời(...)

.withCompensation(...)

Tạo ra

Triệu hồi

**Mô tả một kênh tin nhắn**

Quản lý

**Siêu lớp trừu tượng cho các nhà điều phối saga**

tạo nên()

...

**Mô tả các bước của một câu chuyện dài**

Sử dụng

Tạo đơn hàngSaga

trả lời

Xe điện Eventuate

SagaDefinition lấySagaDefinition()

Quản lý Saga

«giao diện»Đơn giảnSaga

Những câu chuyện về xe điện

Thực hiện

Proxy dịch vụ đặt hàng

Tạo đơn hàng Saga

orderId chi tiết đơn hàng

...

Tạo đơn hàng SagaState

Cơ sở dữ liệu đơn hàng

«bảng» SAGA\_INSTANCE

Dịch vụ nhà bếpProxy

Định nghĩa Saga

LệnhEndpoint

Dịch vụ đặt hàng

**Lưu trữ trạng thái của các phiên bản saga**

Dịch vụ đặt hàngyêu cầu

**Hình 4.11 Dịch vụ đặt hàngsaga của,chẳng hạn nhưTạo đơn hàng Saga, được thực hiệnsử dụng khuôn khổ Eventuate Tram Saga.**

###### Việc thực hiện Create Order Saga

Hình 4.11 cho thấy các lớp thực hiện Create Order Saga. Trách nhiệm của mỗi lớp như sau:

* + - * Tạo đơn hàngSaga—Một lớp đơn lẻ xác định máy trạng thái của saga. Nógọi làTạo đơn hàng SagaStateđể tạo tin nhắn lệnh và gửi chúng đến những người tham gia bằng cách sử dụng các kênh tin nhắn được chỉ định bởi các lớp proxy của người tham gia saga, chẳng hạn nhưDịch vụ nhà bếpProxy.
* Tạo đơn hàng SagaState—Trạng thái liên tục của một saga, tạo ra các thông điệp lệnh.
* *Các lớp proxy của người tham gia Saga, chẳng hạn như*Dịch vụ nhà bếpProxy—Mỗi lớp proxy xác định API nhắn tin của người tham gia saga, bao gồm kênh lệnh, loại tin nhắn lệnh và loại trả lời.

Các lớp học này được viết bằng cách sử dụng khuôn khổ Eventuate Tram Saga.

Khung Eventuate Tram Saga cung cấp một ngôn ngữ dành riêng cho miền (DSL) để xác định máy trạng thái của saga. Nó thực thi máy trạng thái của saga và trao đổi tin nhắn với những người tham gia saga bằng khung Eventuate Tram. Khung này cũng lưu trữ trạng thái của saga trong cơ sở dữ liệu.

Chúng ta hãy xem xét kỹ hơn việc thực hiệnTạo đơn hàng Saga, bắt đầu bằngTạo đơn hàngSagalớp học.

**TANH TACTỔNG KẾTỒĐẠI HỌCSNHẠC SĨ AGA**

Lớp CreateOrderSaga triển khai máy trạng thái được hiển thị trước đó trong hình 4.7. Lớp này triển khai SimpleSaga, một giao diện cơ sở cho sagas. Trái tim của lớp CreateOrderSaga là định nghĩa saga được hiển thị trong danh sách sau. Nó sử dụng DSL do khung Eventuate Tram Saga cung cấp để xác định các bước của Create Order Saga.

**Liệt kê 4.2 Định nghĩa củaTạo đơn hàngSaga**

lớp công khai CreateOrderSaga triển khai SimpleSaga<CreateOrderSagaState> {SagaDefinition<CreateOrderSagaState> sagaDefinition riêng tư;

công khai CreateOrderSaga(OrderServiceProxy orderService,

ConsumerServiceProxy consumerService, KitchenServiceProxy kitchenService, AccountingServiceProxy accountingService) {

this.sagaĐịnh nghĩa =

bước chân()

.withCompensation(dịch vụ đặt hàng. từ chối,

(CreateOrderSagaState::makeRejectOrderCommand)

.bước chân()

.invokeParticipant(consumerService.validateOrder, CreateOrderSagaState::makeValidateOrderByConsumerCommand)

.bước chân()

.invokeParticipant(kitchenService.create, CreateOrderSagaState::makeCreateTicketCommand)

.onReply(CreateTicketReply.class, CreateOrderSagaState::handleCreateTicketReply)

.withCompensation(kitchenService.cancel, CreateOrderSagaState::makeCancelCreateTicketCommand)

.bước chân()

.invokeParticipant(accountingService.authorize, CreateOrderSagaState::makeAuthorizeCommand)

.bước chân()

.invokeParticipant(kitchenService.confirmCreate, CreateOrderSagaState::makeConfirmCreateTicketCommand)

.bước chân()

.invokeParticipant(orderService.approve,

(CreateOrderSagaState::makeApproveOrderCommand)

.xây dựng();

}

@Ghi đè

công khai SagaDefinition<CreateOrderSagaState> getSagaDefinition() {trả về sagaDefinition;

}

CácTạo đơn hàngSaga' constructor tạo ra định nghĩa saga và lưu trữ nó trongsagaĐịnh nghĩacánh đồng.lấySagaDefinition()phương thức trả về định nghĩa saga.Để xem cáchTạo đơn hàngSagatác phẩm, chúng ta hãy xem định nghĩa của bước thứ ba của saga, được hiển thị trong danh sách sau. Bước này của saga gợi lênDịch vụ nhà bếpđể tạo ra mộtVé. Giao dịch bù trừ của nó hủy bỏ điều đóVé. Cácbước chân(),invokeNgười tham gia(),trênTrả lời(), VàvớiBù trừ()phương pháp là

một phần của DSL được cung cấp bởi EventuateTruyện về xe điện.

**Liệt kê 4.3 Định nghĩa bước thứ ba của saga**

lớp công khai CreateOrderSaga ...

**Gọi handleCreateTicketReply() khi nhận được phản hồi thành công.**

công khai CreateOrderSaga(..., KitchenServiceProxy kitchenService,

...) {

...

.bước chân()

.invokeParticipant(kitchenService.create,

**Xác định giao dịch chuyển tiếp.**

CreateOrderSagaState::makeCreateTicketCommand)

.onReply(CreateTicketReply. lớp,

CreateOrderSagaState::handleCreateTicketReply)

.withCompensation(kitchenService.cancel, CreateOrderSagaState::makeCancelCreateTicketCommand)

...

;

**Xác định sự bù trừ**

**giao dịch.**

Cuộc gọi đếninvokeNgười tham gia()định nghĩa giao dịch chuyển tiếp. Nó tạo raTạo nên-Vélệnh tin nhắn bằng cách gọiCreateOrderSagaState.makeCreateTicket-Yêu cầu()và gửi nó đến kênh được chỉ định bởiDịch vụ nhà bếp.create.Cuộc gọi đếntrênTrả lời()chỉ rõ rằngCreateOrderSagaState.handleCreateTicketReply()nên được gọi khi nhận được phản hồi thành công từDịch vụ nhà bếp. Phương pháp này lưu trữ dữ liệu trả vềMã số vétrongTạo đơn hàng SagaState. Cuộc gọi đếnvớiBù trừ()định nghĩa giao dịch bù trừ. Nó tạo ra mộtLệnh RejectTicketlệnh tin nhắnbằng cách gọiCreateOrderSagaState.makeCancelCreate- Ticket()và gửi nó đến kênh được chỉ định bởiDịch vụ nhà bếp.create.

Các bước khác của saga được định nghĩa theo cách tương tự. CreateOrder-SagaState tạo ra từng tin nhắn, được saga gửi đến điểm cuối nhắn tin

được xác định bởi mộtDịch vụ nhà bếpProxy. Chúng ta hãy cùng xem xét từng cái mộtcủa những lớp đó, bắt đầu bằngTạo đơn hàng SagaState.

**TANH TACTỔNG KẾTỒĐẠI HỌCSAGASLỚP TATE**

Lớp CreateOrderSagaState, được hiển thị trong danh sách sau, biểu diễn trạng thái của một phiên bản saga. Một phiên bản của lớp này được tạo bởi OrderService và được lưu trong cơ sở dữ liệu bởi khung Eventuate Tram Saga. Trách nhiệm chính của nó là tạo các thông báo được gửi đến những người tham gia saga.

**Liệt kê 4.4Tạo đơn hàng SagaStatelưu trữ trạng thái của một phiên bản saga**

lớp công khai CreateOrderSagaState { riêng tư Long orderId;

riêng tư OrderDetails orderDetails; riêng tư long ticketId;

công khai Long getOrderId() { trả về orderId;

}

riêng tư CreateOrderSagaState() {

}

**Được gọi bởi OrderService để khởi tạo CreateOrderSagaState**

công khai CreateOrderSagaState(Long orderId, OrderDetails orderDetails) { this.orderId = orderId;

this.orderDetails = orderDetails;

}

Tạo Vé makeCreateTicketCommand() {

**Tạo một thông báo lệnh CreateTicket**

trả về CreateTicket mới(getOrderDetails().getRestaurantId(), getOrderId(), makeTicketDetails(getOrderDetails()));

}

void handleCreateTicketReply(CreateTicketReply reply) { logger.debug("getTicketId {}", reply.getTicketId());

đặtTicketId(trả lời.getTicketId());

}

CancelCreateTicket makeCancelCreateTicketCommand() { trả về CancelCreateTicket mới(getOrderId());

}

...

**Lưu ID của Ticket mới tạo**

**Tạo thông báo lệnh CancelCreateTicket**

CácTạo đơn hàngSagagọi làTạo đơn hàng SagaStateđể tạo các thông điệp lệnh. Nó gửi các thông điệp lệnh đó đến các điểm cuối được xác định bởiSaga- Người tham giaProxylớp học. Chúng ta hãy cùng xem xét một trong những lớp học đó:KitchenService- Proxy.

**TANH TAKNGỨASDỊCH VỤPROXYLỚP HỌC**

CácDịch vụ nhà bếpProxylớp, được hiển thị trong danh sách 4.5, xác định các điểm cuối của thông báo lệnh choDịch vụ nhà bếp. Có ba điểm cuối:

* + tạo nên—Tạo ra mộtVé
  + xác nhậnTạo—Xác nhận việc tạo ra
  + Hủy bỏ—Hủy bỏ mộtVé

Mỗi CommandEndpoint chỉ định loại lệnh, kênh đích của tin nhắn lệnh và loại phản hồi dự kiến.

**Liệt kê 4.5 Dịch vụ nhà bếpProxyxác định các điểm cuối của thông báo lệnh cho**

**Dịch vụ nhà bếp**

lớp công khai KitchenServiceProxy {

công khai cuối cùng CommandEndpoint<CreateTicket> tạo = CommandEndpointBuilder

.forCommand(CreateTicket. lớp)

.vớiKênh(

KitchenServiceChannels.kitchenServiceChannel)

.withReply(CreateTicketReply.lớp)

.xây dựng();

công khai cuối cùng CommandEndpoint<ConfirmCreateTicket> confirmCreate = CommandEndpointBuilder

.forCommand(Xác nhậnTạoTicket.lớp)

.vớiKênh(

KitchenServiceChannels.kitchenServiceChannel)

.withReply(Lớp thành công)

.xây dựng();

công khai cuối cùng CommandEndpoint<CancelCreateTicket> hủy = CommandEndpointBuilder

.forCommand(HủyTạoTicket.lớp)

.vớiKênh(

KitchenServiceChannels.kitchenServiceChannel)

.withReply(Lớp thành công)

.xây dựng();

}

Các lớp proxy, chẳng hạn như KitchenServiceProxy, không hoàn toàn cần thiết. Một saga có thể chỉ cần gửi tin nhắn lệnh trực tiếp đến những người tham gia. Nhưng các lớp proxy có hai lợi ích quan trọng. Đầu tiên, một lớp proxy định nghĩa các điểm cuối được gõ tĩnh, giúp giảm khả năng saga gửi tin nhắn sai đến một dịch vụ. Thứ hai, một lớp proxy là một API được định nghĩa rõ ràng để gọi một dịch vụ giúp mã dễ hiểu và dễ kiểm tra hơn. Ví dụ, chương 10 mô tả cách viết các bài kiểm tra cho KitchenServiceProxy để xác minh rằng Order Service gọi đúng Kitchen Service. Nếu không có KitchenServiceProxy, sẽ không thể viết một bài kiểm tra có phạm vi hẹp như vậy.

**TANH TAETHỬ THÁCHTĐẬPSKHUNG AGA**

Eventuate Tram Saga, được thể hiện trong hình 4.12, là một khuôn khổ để viết cả người dàn dựng saga và người tham gia saga. Nó sử dụng khả năng nhắn tin giao dịch của Eventuate Tram, được thảo luận trong chương 3.

**SagaManager xử lýduy trì một saga, gửi các tin nhắn lệnh mà nó tạo ra, đăng ký nhận tin nhắn trả lời và gọi saga để xử lý các câu trả lời.**

**Siêu lớp trừu tượng cho các nhà điều phối saga**

**Mô tả một kênh tin nhắn**

**Tuyến lệnh tin nhắn đến trình xử lý tin nhắn**

Người tham gia

Sự phối hợp

Khung truyện tram saga của sự kiện

Đơn giảnSaga

SagaDefinition lấySagaDefinition()

LệnhEndpoint

tạo(sagaState)

...

Quản lý Saga

SagaCommand HandlersBuilder

Lệnh SagaNgười điều phối

Sử dụng

Định nghĩa Saga

Cơ sở dữ liệu đơn hàng

«bảng» SAGA\_INSTANCE

Gửi và nhận

Xe điện Eventuate

Kênh

**Lưu trữ trạng thái của các phiên bản saga**

**Mô tả các bước của một câu chuyện dài**

**Hình 4.12 Eventuate Tram Saga là một khuôn khổ để viết cả người dàn dựng saga và sagangười tham gia.**

Gói dàn dựng saga là phần phức tạp nhất của khung. Nó cung cấp SimpleSaga, một giao diện cơ sở cho saga, và một lớp SagaManager, tạo và quản lý các thể hiện saga. SagaManager xử lý việc duy trì saga, gửi các thông báo lệnh mà nó tạo ra, đăng ký các thông báo trả lời và gọi saga để xử lý các phản hồi. Hình 4.13 cho thấy trình tự các sự kiện khi OrderService tạo một saga. Trình tự các sự kiện như sau:

**1** Dịch vụ đặt hàngtạo raTạo đơn hàng SagaState.

**2** Nó tạo ra một trường hợp của một saga bằng cách gọiQuản lý Saga.

**3** CácQuản lý Sagathực hiện bước đầu tiên của định nghĩa saga.

**4** CácTạo đơn hàng SagaStateđược gọi để tạo ra một thông điệp lệnh.

Định nghĩa Saga Sự kiện xe điện

mới()

tạo(sagaState)

getSagaDefinition() thực hiệnFirstStep(sagaState)

makeValidateOrderByConsumerCommand()

sendMessage(lệnh) saveSagaInstance(sagaState)

Quản lý Saga

Tạo đơn hàngSaga

Dịch vụ đặt hàng

Cơ sở dữ liệu

**Hình 4.13 Trình tự các sự kiện khiDịch vụ đặt hàngtạo ra một trường hợp củaTạo đơn hàng Saga**

**5** CácQuản lý Sagagửi tin nhắn lệnh đến người tham gia saga (

Dịch vụ khách hàng).

**6** CácQuản lý Sagalưu phiên bản saga vào cơ sở dữ liệu.

Hình 4.14 cho thấy trình tự các sự kiện khi SagaManager nhận được phản hồi từ

Người tiêu dùngDịch vụ.

Tạo đơn hàng SagaState

Định nghĩa Saga

Quản lý Saga

|  |  |  |
| --- | --- | --- |
| Sự kiện xe điện | |  |
|  | xử lý tin nhắn() | |
| gửi tin nhắn(yêu cầu) | |

Cơ sở dữ liệu

Tạo đơn hàngSaga

loadSagaInstance() getSagaDefinition() thực hiệnFirstStep(sagaState)

saveSagaInstance(tiểu bang hiền triết)

makeValidateOrderByConsumerCommand()

Tạo đơn hàng SagaState

**Hình 4.14 Trình tự các sự kiện khiQuản lý Saganhận được tin nhắn trả lời từ người tham gia saga**

Trình tự các sự kiện như sau:

**1**Sự kiện Tram triệu hồiQuản lý Sagavới câu trả lời từDịch vụ khách hàng.

**2** Quản lý Sagalấy phiên bản saga từ cơ sở dữ liệu.

**3** Quản lý Sagathực hiện bước tiếp theo của định nghĩa saga.

**4** Tạo đơn hàng SagaStateđược gọi để tạo ra một thông điệp lệnh.

**5**Quản lý Sagagửi tin nhắn lệnh đến người tham gia saga đã chỉ định (Dịch vụ nhà bếp).

**6** Quản lý Sagalưu phiên bản cập nhật saga vào cơ sở dữ liệu.

Nếu một người tham gia saga thất bại, SagaManager sẽ thực hiệncác giao dịch bù trừ theo thứ tự ngược lại.

Phần còn lại của khung Eventuate Tram Saga là gói người tham gia saga. Nó cung cấp các lớp SagaCommandHandlersBuilder và SagaCommandDispatcher để viết người tham gia saga. Các lớp này định tuyến các thông điệp lệnh đến các phương thức handler, gọi logic nghiệp vụ của người tham gia saga và tạo ra các thông điệp trả lời. Chúng ta hãy xem cách các lớp này được Order Service sử dụng.

###### Lớp OrderCommandHandlers

Dịch vụ đặt hàngtham gia vào các sagas của riêng mình. Ví dụ,Tạo đơn hàngSagagọiDịch vụ đặt hàngđể chấp thuận hoặc từ chối mộtĐặt hàng. CácTrình xử lý lệnhlớp, được hiển thị trong hình 4.15, định nghĩa các phương thức xử lý cho các thông điệp lệnh được gửi bởi các saga này.

Mỗi phương thức xử lý gọiDịch vụ đặt hàngđể cập nhật mộtĐặt hàngvà tạo một tin nhắn trả lời.SagaCommandDispatcherlớp định tuyến các thông điệp lệnh đến phương thức xử lý thích hợp và gửi phản hồi.

**Tuyến lệnh gửi tin nhắn đến trình xử lý và gửi phản hồi**

Đọc

Triệu hồi

Sử dụng

Người điều phối SagaCommand

Những câu chuyện về xe điện

Gửi

Xe điện Eventuate

|  |  |
| --- | --- |
| Trình xử lý lệnh | |
| chấp thuận đơn hàng() từ chối đơn hàng()  ... | |
|  | Triệu hồi |

Dịch vụ đặt hàngyêu cầu

chấp thuận đơn hàng() từ chối đơn hàng()

...

Dịch vụ đặt hàng

Tạo đơn hàngSagatrả lời

**Hình 4.15Trình xử lý lệnhthực hiện trình xử lý lệnh cho các lệnh đó làđược gửi bởi các khác nhauDịch vụ đặt hàngtruyện dài.**

Danh sách sau đây cho thấyTrình xử lý lệnhlớp học. NólệnhHandlers()phương pháp ánh xạ các loại tin nhắn lệnh tới các phương pháp xử lý. Mỗi phương pháp xử lý lấy một tin nhắn lệnh làm tham số, gọiDịch vụ đặt hàngvà trả về tin nhắn trả lời.

**Liệt kê 4.6 Trình xử lý lệnh choDịch vụ đặt hàng**

lớp công khai OrderCommandHandlers {

@Autowired

riêng tư OrderService orderService;

công khai CommandHandlers commandHandlers() { trả về SagaCommandHandlersBuilder

.fromChannel("Dịch vụ đặt hàng")

**Chuyển hướng từng thông báo lệnh đến phương thức xử lý thích hợp.**

.onMessage(Lớp ApproveOrderCommand,này::phê duyệt đơn hàng)

.onMessage(Lệnh từ chối đơn hàng,this::rejectOrder)

...

.xây dựng();

}

công khai Message approveOrder(CommandMessage<ApproveOrderCommand> cm) { dài orderId = cm.getCommand().getOrderId();

orderService.approveOrder(orderId);trả về vớiThành công();

**Thay đổi trạng thái của Order thành**

} **Trả về một cái chung chung**

**tin nhắn thành công.**

**được ủy quyền.**

công khai Message rejectOrder(CommandMessage<RejectOrderCommand> cm) { dài orderId = cm.getCommand().getOrderId();

orderService.rejectOrder(orderId);trả về vớiThành công();

}

**Thay đổi trạng thái của Đơn hàng thành đã từ chối.**

Cácchấp thuận đơn hàng()Vàtừ chối đơn hàng()phương pháp cập nhật các chỉ địnhĐặt hàngbằng cách triệu hồiDịch vụ đặt hàng. Các dịch vụ khác tham gia vào sagas có các lớp xử lý lệnh tương tự cập nhật các đối tượng miền của chúng.

###### Lớp OrderServiceConfiguration

CácDịch vụ đặt hàngsử dụng khung Spring. Danh sách sau đây là một đoạn trích củaCấu hình dịch vụ đặt hànglớp, đó là một@Cấu hìnhlớp khởi tạo và kết nối Spring với nhau@Đậu.

**Liệt kê 4.7Cấu hình dịch vụ đặt hànglà một mùa xuân@Cấu hình**

**lớp định nghĩa Spring@ĐậuchoDịch vụ đặt hàng.**

@Cấu hình

lớp công khai OrderServiceConfiguration {

@Đậu

công khai OrderService orderService(RestaurantRepository restaurantRepository,

...

SagaManager<CreateOrderSagaState>tạoOrderSagaManager,

...) {

trả về OrderService mới(restaurantRepository,

...

tạoOrderSagaManager

...);

}

@Đậu

công khai SagaManager<CreateOrderSagaState>createOrderSagaManager(CreateOrderSaga là saga) {

trả về SagaManagerImpl<>(saga) mới;

}

@Đậu

công khai CreateOrderSaga createOrderSaga(OrderServiceProxy orderService,

ConsumerServiceProxy người tiêu dùngDịch vụ,

...) {

trả về CreateOrderSaga mới(orderService, consumerService, ...);

}

@Đậu

công khai OrderCommandHandlers orderCommandHandlers() { trả về OrderCommandHandlers() mới;

}

@Đậu

công khai SagaCommandDispatcher orderCommandHandlersDispatcher(OrderCommandHan dlers orderCommandHandlers) {

trả về SagaCommandDispatcher mới ("orderService", orderCommandHandlers.comma ndHandlers());

}

@Đậu

công khai KitchenServiceProxy kitchenServiceProxy() { trả về KitchenServiceProxy mới();

}

@Đậu

công khai OrderServiceProxy orderServiceProxy() { trả về OrderServiceProxy mới();

}

...

}

Lớp này định nghĩa một số Spring@Đậubao gồmdịch vụ đặt hàng,createOrder-SagaManager,tạoOrderSaga,lệnhCommandHandlers, VàorderCommandHandlers-Người điều phối. Nó cũng định nghĩa Spring@Đậucho các lớp proxy khác nhau, bao gồmDịch vụ nhà bếpProxyVàDịch vụ đặt hàngProxy.

***Bản tóm tắt* 145**

Tạo đơn hàngSagachỉ là một trongDịch vụ đặt hàngnhiều saga của nó. Nhiều hệ thống khác của nócác hoạt động tem cũng sử dụng sagas. Ví dụ,hủy đơn hàng()hoạt động sử dụng mộtHủy đơn hàng Sagavàsửa lại đơn hàng()hoạt động sử dụng mộtSửa đổi Order Saga. Do đó, mặc dù nhiều dịch vụ có API bên ngoài sử dụng giao thức đồng bộ, chẳng hạn như REST hoặc gRPC, nhưng một lượng lớn giao tiếp giữa các dịch vụ sẽ sử dụng nhắn tin không đồng bộ.

Như bạn có thể thấy, quản lý giao dịch và một số khía cạnh của thiết kế logic kinh doanh khá khác nhau trong kiến ​​trúc microservice. May mắn thay, saga orchestrator thường là các máy trạng thái khá đơn giản và bạn có thể sử dụng một khung saga để đơn giản hóa mã của mình. Tuy nhiên, quản lý giao dịch chắc chắn phức tạp hơn trong kiến ​​trúc monolithic. Nhưng đó thường là một cái giá nhỏ phải trả cho những lợi ích to lớn của microservice.

#### Bản tóm tắt

* Một số hoạt động hệ thống cần cập nhật dữ liệu phân tán trên nhiều dịch vụ. Các giao dịch phân tán truyền thống dựa trên XA/2PC không phù hợp với các ứng dụng hiện đại. Một cách tiếp cận tốt hơn là sử dụng mô hình Saga. Một saga là chuỗi các giao dịch cục bộ được phối hợp bằng cách sử dụng tin nhắn. Mỗi giao dịch cục bộ cập nhật dữ liệu trong một dịch vụ duy nhất. Vì mỗi giao dịch cục bộ cam kết các thay đổi của nó, nếu một saga phải khôi phục do vi phạm quy tắc kinh doanh, nó phải thực hiện các giao dịch bù trừ để hoàn tác rõ ràng các thay đổi.
* Bạn có thể sử dụng biên đạo hoặc phối hợp để phối hợp các bước của một saga. Trong một saga dựa trên biên đạo, một giao dịch cục bộ sẽ công bố các sự kiện kích hoạt những người tham gia khác thực hiện các giao dịch cục bộ. Trong một saga dựa trên phối hợp, một bộ phối hợp saga tập trung sẽ gửi các thông điệp lệnh đến những người tham gia yêu cầu họ thực hiện các giao dịch cục bộ. Bạn có thể đơn giản hóa quá trình phát triển và thử nghiệm bằng cách mô hình hóa các bộ phối hợp saga như các máy trạng thái. Các saga đơn giản có thể sử dụng biên đạo, nhưng phối hợp thường là một cách tiếp cận tốt hơn cho các saga phức tạp.
* Việc thiết kế logic kinh doanh dựa trên saga có thể là một thách thức vì,không giống như các giao dịch ACID, sagas không bị cô lập với nhau. Bạn thường phải sử dụng các biện pháp đối phó, là các chiến lược thiết kế ngăn ngừa các bất thường đồng thời do mô hình giao dịch ACD gây ra. Một ứng dụng thậm chí có thể cần sử dụng khóa để đơn giản hóa logic kinh doanh, mặc dù điều đó có nguy cơ gây ra bế tắc.

*Thiết kế logic kinh doanh trong*

*một kiến ​​trúc dịch vụ vi mô*

***Chương này bao gồm***

* Áp dụng các mô hình tổ chức logic kinh doanh:Mẫu kịch bản giao dịch và mẫu mô hình miền
* Thiết kế logic kinh doanh theo hướng miềnthiết kế (DDD) mẫu tổng hợp
* Áp dụng mẫu sự kiện Domain trongKiến trúc dịch vụ vi mô

Trái tim của một ứng dụng doanh nghiệp là logic kinh doanh, nơi triển khai các quy tắc kinh doanh. Việc phát triển logic kinh doanh phức tạp luôn là một thách thức. Logic kinh doanh của ứng dụng FTGO triển khai một số logic kinh doanh khá phức tạp, đặc biệt là đối với quản lý đơn hàng và quản lý giao hàng. Mary đã khuyến khích nhóm của mình áp dụng các nguyên tắc thiết kế hướng đối tượng, vì theo kinh nghiệm của cô, đây là cách tốt nhất để triển khai logic kinh doanh phức tạp. Một số logic kinh doanh sử dụng mẫu tập lệnh Phiên âm thủ tục. Nhưng phần lớn logic kinh doanh của ứng dụng FTGO được triển khai trong mô hình miền hướng đối tượng được ánh xạ vào cơ sở dữ liệu bằng JPA.

Việc phát triển logic kinh doanh phức tạp thậm chí còn khó khăn hơn trong kiến ​​trúc dịch vụ vi mô, nơi logic kinh doanh được phân bổ trên nhiều dịch vụ. Bạn cần

**146**

giải quyết hai thách thức chính. Đầu tiên, một mô hình miền điển hình là một mạng lưới rối rắm của các lớp được kết nối với nhau. Mặc dù đây không phải là vấn đề trong một ứng dụng đơn khối, nhưng trong kiến ​​trúc dịch vụ vi mô, nơi các lớp nằm rải rác xung quanh các dịch vụ khác nhau, bạn cần loại bỏ các tham chiếu đối tượng nếu không sẽ mở rộng ranh giới dịch vụ. Thách thức thứ hai là thiết kế logic kinh doanh hoạt động trong các ràng buộc quản lý giao dịch của kiến ​​trúc dịch vụ vi mô. Logic kinh doanh của bạn có thể sử dụng các giao dịch ACID trong các dịch vụ, nhưng như đã mô tả trong chương 4, nó phải sử dụng mẫu Saga để duy trì tính nhất quán của dữ liệu trên các dịch vụ.

May mắn thay, chúng ta có thể giải quyết những vấn đề này bằng cách sử dụng mẫu Aggregate từ DDD. Mẫu Aggregate cấu trúc logic kinh doanh của dịch vụ như một tập hợp các tổng hợp. Tổng hợp là một cụm các đối tượng có thể được coi là một đơn vị. Có hai lý do tại sao tổng hợp hữu ích khi phát triển logic kinh doanh trong kiến ​​trúc dịch vụ vi mô:

* Tổng hợp tránh mọi khả năng tham chiếu đối tượng vượt ra ngoài ranh giới dịch vụ, vì tham chiếu liên tổng hợp là giá trị khóa chính chứ không phải tham chiếu đối tượng.
* Vì một giao dịch chỉ có thể tạo hoặc cập nhật một tổng hợp duy nhất nên các tổng hợp này phù hợp với các ràng buộc của mô hình giao dịch vi dịch vụ.

Kết quả là, một giao dịch ACID được đảm bảo nằm trong mộtdịch vụ.

Tôi bắt đầu chương này bằng cách mô tả các cách khác nhau để tổ chức logic kinh doanh: mẫu tập lệnh Phiên âm và mẫu mô hình Miền. Tiếp theo, tôi giới thiệu khái niệm về tổng hợp DDD và giải thích lý do tại sao nó là một khối xây dựng tốt cho logic kinh doanh của dịch vụ. Sau đó, tôi mô tả các sự kiện mẫu sự kiện Miền và giải thích lý do tại sao việc xuất bản sự kiện lại hữu ích cho dịch vụ. Tôi kết thúc chương này bằng một vài ví dụ về logic kinh doanh từ Kitchen Service và Order Service.

Bây giờ chúng ta hãy xem xét các mô hình tổ chức logic kinh doanh.

#### Các mô hình tổ chức logic kinh doanh

Hình 5.1 cho thấy kiến ​​trúc của một dịch vụ điển hình. Như đã mô tả trong chương 2, logic nghiệp vụ là cốt lõi của kiến ​​trúc lục giác. Bao quanh logic nghiệp vụ là các bộ điều hợp inbound và outbound. Bộ điều hợp inbound xử lý các yêu cầu từ máy khách và gọi logic nghiệp vụ. Bộ điều hợp outbound, được logic nghiệp vụ gọi, gọi các dịch vụ và ứng dụng khác.

Dịch vụ này bao gồm logic kinh doanh và các bộ điều hợp sau:

* + - Bộ điều hợp REST API—Bộ điều hợp đầu vào triển khai REST API để gọi logic kinh doanh
    - Trình xử lý lệnh-MỘTbộ điều hợp đến sử dụng các tin nhắn lệnh từ kênh tin nhắn và gọi logic kinh doanh
    - Bộ điều hợp cơ sở dữ liệu—Một bộ điều hợp đầu ra được logic kinh doanh gọi để truy cập cơ sở dữ liệu
    - Bộ điều hợp xuất bản sự kiện miền—Một bộ điều hợp ra ngoài phát hành các sự kiệncho một nhà môi giới tin nhắn



**Bộ chuyển đổi đầu vào**

BÀI ĐĂNG/ĐƠN HÀNG

NHẬN/đặt hàng/Id

Đặt hàng

Yêu cầu dịch vụ

Dịch vụ đặt hàng

logic kinh doanh

Đặt hàng sự kiện

**Bộ chuyển đổi đầu ra**

Bộ điều hợp cơ sở dữ liệu

Bộ điều hợp nhà xuất bản sự kiện miền

Giao diện lập trình ứng dụng REST

Cơ sở dữ liệu đơn hàng

Trình xử lý lệnh đặt hàng

**Hình 5.1Dịch vụ đặt hàngcó kiến ​​trúc lục giác. Nó bao gồm logic kinh doanh và một hoặc nhiều bộ điều hợp giao tiếp với các ứng dụng bên ngoài và các dịch vụ khác.**

Logic kinh doanh thường là phần phức tạp nhất của dịch vụ. Khi phát triển logic kinh doanh, bạn nên có ý thức tổ chức logic kinh doanh của mình theo cách phù hợp nhất với ứng dụng của bạn. Sau cùng, tôi chắc chắn rằng bạn đã từng trải qua sự thất vọng khi phải duy trì mã có cấu trúc kém của người khác. Hầu hết các ứng dụng doanh nghiệp đều được viết bằng ngôn ngữ hướng đối tượng như Java, vì vậy chúng bao gồm các lớp và phương thức. Nhưng việc sử dụng ngôn ngữ hướng đối tượng không đảm bảo rằng logic kinh doanh có thiết kế hướng đối tượng. Quyết định quan trọng mà bạn phải đưa ra khi phát triển logic kinh doanh là sử dụng phương pháp tiếp cận hướng đối tượng hay phương pháp tiếp cận theo thủ tục. Có hai mô hình chính để tổ chức

logic kinh doanh: mô hình tập lệnh Giao dịch theo thủ tục và mô hình Miền hướng đối tượng.

###### Thiết kế logic kinh doanh bằng cách sử dụng mẫu tập lệnh Giao dịch

Mặc dù tôi là người ủng hộ mạnh mẽ cách tiếp cận hướng đối tượng, nhưng có một số tình huống mà cách tiếp cận này là quá mức cần thiết, chẳng hạn như khi bạn đang phát triển logic kinh doanh đơn giản. Trong tình huống như vậy, cách tiếp cận tốt hơn là viết mã thủ tục và sử dụng những gì trong sách Patterns*của Kiến trúc ứng dụng doanh nghiệp*bởi Martin Fowler (Addison-Wesley Professional, 2002)gọi mẫu tập lệnh Giao dịch. Thay vì thực hiện bất kỳ thiết kế hướng đối tượng nào, bạn viết một phương thức được gọi là tập lệnh giao dịch để xử lý từng yêu cầu từ tầng trình bày. Như hình 5.2 cho thấy, một đặc điểm quan trọng của phương pháp này là các lớp triển khai hành vi tách biệt với các lớp lưu trữ trạng thái.

Các lớp họcvới

hành vi

**Hình 5.2 Tổ chức logic kinh doanhdưới dạng tập lệnh giao dịch. Trong thiết kế dựa trên tập lệnh giao dịch thông thường, một tập hợp các lớp thực hiện hành vi và một tập hợp khác lưu trữ trạng thái. Các tập lệnh giao dịch được tổ chức thành các lớp thường không có trạng thái. Các tập lệnh sử dụng các lớp dữ liệu, thường không có hành vi.**

orderId đơn hàng

...

Đặt hàng

lưu(Đặt hàng) tìmOrderById()

...

Đặt hàngDao

createOrder() sửa lạiOrder() hủyOrder()

...

Dịch vụ đặt hàng

Các lớp học

với trạng thái

Khi sử dụng mẫu tập lệnh Giao dịch, các tập lệnh thường nằm trong dịch vụCác lớp, trong ví dụ này là lớp OrderService. Một lớp dịch vụ có một phương thức cho mỗi yêu cầu/hoạt động hệ thống. Phương thức này triển khai logic nghiệp vụ cho yêu cầu đó. Nó truy cập cơ sở dữ liệu bằng các đối tượng truy cập dữ liệu (DAO), chẳng hạn như OrderDao. Các đối tượng dữ liệu, trong ví dụ này là lớp Order, là dữ liệu thuần túy với ít hoặc không có hành vi.

**Mẫu: Kịch bản giao dịch**

Tổ chức logic kinh doanh như một tập hợp các tập lệnh giao dịch thủ tục, một chomỗi loại yêu cầu.

Phong cách thiết kế này mang tính thủ tục cao và dựa vào một số ít khả năng của ngôn ngữ lập trình hướng đối tượng (OOP). Đây là những gì bạn sẽ tạo ra nếu bạn đang viết ứng dụng bằng C hoặc một ngôn ngữ không phải OOP khác. Tuy nhiên, bạn không nên

xấu hổ khi sử dụng thiết kế thủ tục khi thích hợp. Cách tiếp cận này hoạt động tốt đối với logic kinh doanh đơn giản.Nhược điểm là đây không phải là cách tốt để triển khai logic kinh doanh phức tạp.

###### Thiết kế logic kinh doanh bằng cách sử dụng mô hình Domain

Sự đơn giản của thủ tụcCách tiếp cận có thể khá hấp dẫn. Bạn có thể viết mã mà không cần phải cân nhắc cẩn thận cách tổ chức các lớp. Vấn đề là nếu logic kinh doanh của bạn trở nên phức tạp, bạn có thể kết thúc với mã là cơn ác mộng để duy trì. Trên thực tế, giống như cách một ứng dụng đơn khối có thói quen liên tục phát triển, các tập lệnh giao dịch cũng có cùng vấn đề. Do đó, trừ khi bạn đang viết một ứng dụng cực kỳ đơn giản, bạn nên chống lại sự cám dỗ viết mã thủ tục và thay vào đó áp dụng mô hình mô hình miền và phát triển một thiết kế hướng đối tượng.

**Mẫu: Mô hình miền**

Tổ chức logic kinh doanh như một đối tượngmô hình bao gồm các lớp có trạng thái và hành vi.

Trong thiết kế hướng đối tượng, logic nghiệp vụ bao gồm một mô hình đối tượng, một mạng lưới các lớp tương đối nhỏ. Các lớp này thường tương ứng trực tiếp với các khái niệm từ miền vấn đề. Trong thiết kế như vậy, một số lớp chỉ có trạng thái hoặc hành vi, nhưng nhiều lớp chứa cả hai, đây là đặc điểm của một lớp được thiết kế tốt. Hình 5.3 cho thấy một ví dụ về mẫu mô hình miền.

Một số lớp chỉ cóhành vi. Một số lớp chỉ có trạng thái.

Sử dụng

tìmOrderById()

...

Kho lưu trữ đơn hàng

Thời gian giao hàng Địa chỉ giao hàng

Thông tin giao hàng

createOrder() sửa lạiOrder() hủyOrder()

...

Dịch vụ đặt hàng

Nhiều lớp có trạng thái và hành vi.

|  |
| --- |
| Đặt hàng |
| «riêng tư» orderId orderLineItems  ... |
| sửa đổi() hủy bỏ()  «tĩnh» tạo() |

**Hình 5.3 Tổ chức logic kinh doanh như một mô hình miền. Phần lớnlogic kinh doanh bao gồm các lớp có trạng thái và hành vi.**

Giống như mô hình tập lệnh Giao dịch, một OrderServicelớp có một phương thức cho mỗi yêu cầu/hoạt động hệ thống. Nhưng khi sử dụng mô hình mô hình Miền, các phương thức dịch vụ thường đơn giản. Đó là vì một phương thức dịch vụ hầu như luôn chuyển giao cho các đối tượng miền liên tục, chứa phần lớn logic nghiệp vụ. Ví dụ, một phương thức dịch vụ có thể tải một đối tượng miền từ cơ sở dữ liệu và gọi một trong các phương thức của nó. Trong ví dụ này, lớp Order có cả trạng thái và hành vi. Hơn nữa, trạng thái của nó là riêng tư và chỉ có thể được truy cập gián tiếp thông qua các phương thức của nó.

Sử dụng thiết kế hướng đối tượng có một số lợi ích. Đầu tiên, thiết kế dễ hiểu và dễ bảo trì. Thay vì bao gồm một lớp lớn thực hiện mọi thứ, nó bao gồm một số lớp nhỏ, mỗi lớp có một số ít trách nhiệm. Ngoài ra, các lớp như Account, BankingTransaction vàOverdraftPolicy phản ánh chặt chẽ thế giới thực, giúp dễ hiểu hơn về vai trò của chúng trong thiết kế. Thứ hai, thiết kế hướng đối tượng của chúng ta dễ kiểm tra hơn: mỗi lớp có thể và nên được kiểm tra độc lập. Cuối cùng, thiết kế hướng đối tượng dễ mở rộng hơn vì nó có thể sử dụng các mẫu thiết kế nổi tiếng, chẳng hạn như mẫu Strategy và mẫu Template method, định nghĩa các cách mở rộng một thành phần mà không cần sửa đổi mã.

Mô hình miền hoạt động tốt, nhưng có một số vấn đề với cách tiếp cận này, đặc biệt là trong dịch vụ vi môkiến trúc. Để giải quyết những vấn đề đó, bạn cần sử dụng một cải tiến của OOD được gọi là DDD.

###### Về thiết kế theo miền

DDD, được mô tả trong cuốn sách Domain-Driven Design của Eric Evans (Addison-Wesley Professional, 2003), là sự cải tiến của OOD và là một phương pháp để phát triển logic kinh doanh phức tạp. Tôi đã giới thiệu DDD trong chương 2 khi thảo luận về tính hữu ích của các miền phụ DDD khi phân tích một ứng dụng thành các dịch vụ. Khi sử dụng DDD, mỗi dịch vụ có mô hình miền riêng, giúp tránh được các vấn đề của một mô hình miền đơn lẻ trên toàn ứng dụng. Các miền phụ và khái niệm liên quan đến Bounded Context là hai trong số các mô hình DDD chiến lược.

DDD cũng có một số mẫu chiến thuật là các khối xây dựng cho các mô hình miền. Mỗi mẫu là một vai trò mà một lớp đóng trong một mô hình miền và xác định các đặc điểm của lớp. Các khối xây dựng đã được các nhà phát triển áp dụng rộng rãi bao gồm:

* + - * *Thực thể*—Một đối tượng có danh tính cố định. Hai thực thể có các thuộc tính có cùng giá trị vẫn là các đối tượng khác nhau. Trong ứng dụng Java EE, các lớpđược lưu trữ bằng JPA@Thực thểthường là các thực thể DDD.
      * *Đối tượng giá trị*—Một đối tượng là một tập hợp các giá trị. Hai đối tượng giá trị có các thuộc tính có cùng giá trị có thể được sử dụng thay thế cho nhau. Một ví dụ về một đối tượng giá trị làTiền bạclớp, bao gồm một loại tiền tệ và một số tiền.
      * *Nhà máy*—Một đối tượng hoặc phương thức thực hiện logic tạo đối tượng quá phức tạp để có thể thực hiện trực tiếp bằng một hàm tạo. Nó cũng có thể ẩn các đối tượng cụ thể

các lớp được khởi tạo. Một nhà máy có thể được triển khai như một phương thức tĩnh của một lớp.

* *Kho lưu trữ*-MỘTđối tượng cung cấp quyền truy cập vào các thực thể bền bỉ và đóng gói cơ chế truy cập cơ sở dữ liệu.
* *Dịch vụ*—Một đối tượng thực hiện logic kinh doanh không thuộc về một thực thể hoặc một đối tượng giá trị.

Những khối xây dựng này được nhiều nhà phát triển sử dụng. Một số được hỗ trợ bởi các khung như JPA và khung Spring. Có một khối xây dựng nữa thường bị bỏ qua (kể cả tôi!) ngoại trừ những người theo chủ nghĩa thuần túy DDD: tổng hợp. Hóa ra, tổng hợp là một khái niệm cực kỳ hữu ích khi phát triển các dịch vụ siêu nhỏ. Trước tiên, chúng ta hãy xem xét một số vấn đề tinh tế với OOD cổ điển được giải quyết bằng cách sử dụng tổng hợp.

#### Thiết kế một mô hình miền sử dụngMẫu tổng hợp DDD

Trong thiết kế hướng đối tượng truyền thống, mô hình miền là tập hợp các lớp và mối quan hệ giữa các lớp. Các lớp thường được tổ chức thành các gói. Ví dụ, hình 5.4 cho thấy một phần của mô hình miền cho ứng dụng FTGO. Đây là mô hình miền điển hình bao gồm một mạng lưới các lớp được kết nối với nhau.

Được giao cho

Được đặt bởi

Vì

Trả tiền bằng cách sử dụng

Đã thanh toán bằng

phố1 phố2thành phố tiểu bang mã bưu chính

Địa chỉ

tên giá

Mục Menu

Số lượng

Mục đơn hàng

Thời gian giao hàng

Thông tin giao hàng

thẻ tín dụngId

...

Thông tin thanh toán

vĩ độ kinh độ

Vị trí

có sẵn

...

Người chuyển phát nhanh

tên

...

Nhà hàng

tình trạng

...

Đặt hàng

Người tiêu dùng

**Hình 5.4 Một mô hình miền truyền thống là một mạng lưới các lớp được kết nối với nhau. Nó không chỉ định rõ ràng ranh giới của các đối tượng kinh doanh, chẳng hạn nhưNgười tiêu dùngVàĐặt hàng.**

Ví dụ này có một số lớp tương ứng với các đối tượng kinh doanh: Consumer, Order, Restaurant và Courier. Nhưng điều thú vị là ranh giới rõ ràng của từng đối tượng kinh doanh lại không có trong loại mô hình miền truyền thống này. Nó không chỉ định, ví dụ

Ví dụ, lớp nào là một phần của đối tượng kinh doanh Order. Việc thiếu ranh giới này đôi khi có thể gây ra vấn đề, đặc biệt là trong kiến ​​trúc vi dịch vụ.

Tôi bắt đầu phần này bằng một ví dụ về vấn đề do thiếu ranh giới rõ ràng. Tiếp theo, tôi mô tả khái niệm về tổng hợp và cách tổng hợp có ranh giới rõ ràng. Sau đó, tôi mô tả các quy tắc mà tổng hợp phải tuân theo và cách chúng làm cho tổng hợp phù hợp với kiến ​​trúc dịch vụ vi mô. Sau đó, tôi mô tả cách cẩn thận lựa chọn ranh giới của tổng hợp và lý do tại sao điều đó lại quan trọng. Cuối cùng, tôi thảo luận về cách thiết kế logic kinh doanh bằng cách sử dụng tổng hợp. Trước tiên, chúng ta hãy xem xét các vấn đề do ranh giới mờ gây ra.

###### Vấn đề với ranh giới mờ

Ví dụ, hãy tưởng tượng rằng bạn muốn thực hiện một thao tác, chẳng hạn như tải hoặc xóa, trên một đối tượng kinh doanh Order. Điều đó có nghĩa chính xác là gì? Phạm vi của một thao tác là gì? Bạn chắc chắn sẽ tải hoặc xóa đối tượng Order. Nhưng trên thực tế, Order không chỉ đơn thuần là đối tượng Order. Ngoài ra còn có các mục dòng order, thông tin thanh toán, v.v. Hình 5.4 để lại ranh giới của một đối tượng miền cho trực giác của nhà phát triển.

Bên cạnh sự mơ hồ về mặt khái niệm, việc thiếu ranh giới rõ ràng gây ra vấn đề khi cập nhật đối tượng kinh doanh. Một đối tượng kinh doanh thông thường có các bất biến, các quy tắc kinh doanh phải được thực thi mọi lúc. Ví dụ, Đơn hàng có số lượng đơn hàng tối thiểu. Ứng dụng FTGO phải đảm bảo rằng bất kỳ nỗ lực nào để cập nhật đơn hàng đều không vi phạm bất biến như số lượng đơn hàng tối thiểu. Thách thức là để thực thi các bất biến, bạn phải thiết kế logic kinh doanh của mình một cách cẩn thận.

Ví dụ, hãy xem cách đảm bảo đơn hàng tối thiểu được đáp ứng khi nhiều người tiêu dùng làm việc cùng nhau để tạo một đơn hàng. Hai người tiêu dùng—Sam và Mary—làm việc cùng nhau trên một đơn hàng và đồng thời quyết định rằng đơn hàng vượt quá ngân sách của họ. Sam giảm số lượng samosa, và Mary giảm số lượng bánh mì naan. Theo quan điểm của ứng dụng, cả hai người tiêu dùng đều lấy đơn hàng và các mục hàng của đơn hàng đó từ cơ sở dữ liệu. Sau đó, cả hai người tiêu dùng đều cập nhật một mục hàng để giảm chi phí của đơn hàng. Theo quan điểm của mỗi người tiêu dùng, đơn hàng tối thiểu được bảo toàn. Sau đây là trình tự các giao dịch cơ sở dữ liệu.

|  |  |
| --- | --- |
| Người tiêu dùng - Mary | Người tiêu dùng - Sam |
| BẮT ĐẦU TXN | BẮT ĐẦU TXN |
| CHỌN ORDER\_TOTAL TỪ ORDER NƠI ORDER ID = X | CHỌN ORDER\_TOTAL TỪ ORDER NƠI ORDER ID = X |
| CHỌN \* TỪ ORDER\_LINE\_ITEM NƠI ORDER\_ID = X  ... KẾT THÚC TXN | CHỌN \* TỪ ORDER\_LINE\_ITEM NƠI ORDER\_ID = X  ... KẾT THÚC TXN |
| Xác minh mức tối thiểu được đáp ứng |  |

|  |  |
| --- | --- |
| BẮT ĐẦU TXN |  |
| CẬP NHẬT ORDER\_LINE\_ITEM  ĐẶT PHIÊN BẢN=..., SỐ LƯỢNG=...  NƠI VERSION = <phiên bản đã tải> VÀ ID = ... |  |
| KẾT THÚC TXN |  |
|  | Xác minh mức tối thiểu được đáp ứng |
|  | BẮT ĐẦU TXN |
|  | CẬP NHẬT ORDER\_LINE\_ITEM  ĐẶT PHIÊN BẢN=..., SỐ LƯỢNG=...  NƠI VERSION = <phiên bản đã tải> VÀ ID = ... |
|  | KẾT THÚC TXN |

Mỗi người tiêu dùng thay đổi một mục hàng bằng cách sử dụng một chuỗi gồm hai giao dịch. Giao dịch đầu tiên tải đơn hàng và các mục hàng của nó. Giao diện người dùng xác minh rằng đơn hàng tối thiểu được đáp ứng trước khi thực hiện giao dịch thứ hai. Giao dịch thứ hai cập nhật số lượng mục hàng bằng cách sử dụng kiểm tra khóa ngoại tuyến lạc quan để xác minh rằng dòng đơn hàng không thay đổi kể từ khi được giao dịch đầu tiên tải.

Trong trường hợp này, Sam giảm tổng đơn hàng đi $Xvà Mary giảm nó đi $Y. Do đó, Order không còn hiệu lực nữa, mặc dù ứng dụng đã xác minh rằng order vẫn đáp ứng được mức tối thiểu của order sau mỗi lần cập nhật của người tiêu dùng. Như bạn thấy, việc cập nhật trực tiếp một phần của đối tượng kinh doanh có thể dẫn đến vi phạm các quy tắc kinh doanh. Tổng hợp DDD được thiết kế để giải quyết vấn đề này.

###### Các tập hợp có ranh giới rõ ràng

Một tập hợp là một cụm các đối tượng miền trong một ranh giới có thể được coi là một đơn vị. Nó bao gồm một thực thể gốc và có thể là một hoặc nhiều thực thể và đối tượng giá trị khác. Nhiều đối tượng kinh doanh được mô hình hóa như các tập hợp. Ví dụ, trong chương 2, chúng tôi đã tạo ra một mô hình miền thô bằng cách phân tích các danh từ được sử dụng trong các yêu cầu và bởi các chuyên gia miền. Nhiều danh từ trong số này, chẳng hạn như Order, Consumer và Restau- rant, là các tập hợp.

**Mẫu: Tổng hợp**

Tổ chức mô hình miền như một tập hợp các tổng hợp, mỗi tổng hợp là một đồ thị các đối tượng có thể được coi như một đơn vị.

Hình 5.5 cho thấyĐặt hàngtổng hợp và ranh giới của nó. MộtĐặt hàngtổng hợp bao gồm mộtĐặt hàngthực thể, một hoặc nhiềuMục đơn hàngcác đối tượng giá trị và các đối tượng giá trị khác như giao hàngĐịa chỉVàThông tin thanh toán.

Tổng hợp người tiêu dùng

Tổng hợp đơn hàng

«đối tượng giá trị» Thông tin thanh toán

«gốc tổng hợp" Đặt hàng

«đối tượng giá trị» Thông tin giao hàng

Số lượng

«đối tượng giá trị» Mục đơn hàng

«gốc tổng hợp" Người tiêu dùng

«gốc tổng hợp" Nhà hàng

...

Tổng hợp nhà hàng

«đối tượng giá trị» Thông tin thanh toán

«đối tượng giá trị» Thông tin giao hàng

**Hình 5.5 Việc cấu trúc mô hình miền thành một tập hợp các tổng hợp sẽ làm cho các ranh giới trở nên rõ ràng.**

Các tập hợp phân tách một mô hình miền thành các khối, mỗi khối dễ hiểu hơn. Chúng cũng làm rõ phạm vi của các hoạt động như tải, cập nhật và xóa. Các hoạt động này tác động lên toàn bộ tập hợp chứ không phải lên các phần của tập hợp. Một tập hợp thường được tải toàn bộ từ cơ sở dữ liệu, do đó tránh được mọi sự phức tạp của tải lười biếng. Việc xóa một tập hợp sẽ xóa tất cả các đối tượng của nó khỏi cơ sở dữ liệu.

**MỘTTỔNG HỢPLÀ RÀNG BUỘC NHẤT QUÁN**

Cập nhật toàn bộ tổng hợp thay vì các phần của nó giải quyết các vấn đề về tính nhất quán, chẳng hạn như ví dụ được mô tả trước đó. Các hoạt động cập nhật được gọi trên gốc tổng hợp, thực thi các bất biến. Ngoài ra, tính đồng thời được xử lý bằng cách khóa gốc tổng hợp bằng cách sử dụng, ví dụ, số phiên bản hoặc khóa cấp cơ sở dữ liệu. Ví dụ, thay vì cập nhật trực tiếp số lượng của các mục hàng, máy khách phải gọi một phương thức trên gốc của tổng hợp Đơn hàng, thực thi các bất biến như số lượng đơn hàng tối thiểu. Tuy nhiên, lưu ý rằng cách tiếp cận này không yêu cầu toàn bộ tổng hợp phải được cập nhật trong cơ sở dữ liệu. Ví dụ, một ứng dụng có thể chỉ cập nhật các hàng tương ứngphản hồi đối tượng Order và OrderLineItem đã cập nhật.

**TÔIXÁC ĐỊNH CÁC TỔNG HỢP LÀ CHÌA KHÓA**

Trong DDD, một phần quan trọng trong việc thiết kế mô hình miền là xác định các tập hợp, ranh giới và gốc của chúng. Chi tiết về cấu trúc bên trong của các tập hợp là thứ yếu. Tuy nhiên, lợi ích của các tập hợp vượt xa việc mô-đun hóa một mô hình miền. Đó là vì các tập hợp phải tuân theo một số quy tắc nhất định.

###### Quy tắc tổng hợp

DDD yêu cầu các tổng hợp phải tuân theo một bộ quy tắc. Nhữngcác quy tắc đảm bảo rằng tổng hợp là một đơn vị độc lập có thể thực thi các bất biến của nó. Hãy cùng xem xét từng quy tắc.

**RULE#1: RCHỈ THAM KHẢO GỐC TỔNG HỢP**

CácVí dụ trước minh họa những nguy cơ khi cập nhật OrderLineItems trực tiếp. Mục tiêu của quy tắc tổng hợp đầu tiên là loại bỏ vấn đề này. Quy tắc này yêu cầu thực thể gốc phải là phần duy nhất của tổng hợp có thể được các lớp bên ngoài tổng hợp tham chiếu. Máy khách chỉ có thể cập nhật tổng hợp bằng cách gọi phương thức trên gốc tổng hợp.

Ví dụ, một dịch vụ sử dụng một kho lưu trữ để tải một tổng hợp từ cơ sở dữ liệu và lấy tham chiếu đến gốc tổng hợp. Nó cập nhật một tổng hợp bằng cách gọi một phương thức trên gốc tổng hợp. Quy tắc này đảm bảo rằng tổng hợp có thể thực thi bất biến của nó.

**RULE#2: TôiNTER-THAM CHIẾU TỔNG HỢP PHẢI SỬ DỤNG KHÓA CHÍNH**

Một quy tắc khác là các tổng hợp tham chiếu lẫn nhau theo danh tính (ví dụ: khóa chính) thay vì tham chiếu đối tượng. Ví dụ, như hình 5.6 cho thấy, một tham chiếu Orderkiểm tra Consumer của nó bằng cách sử dụng consumerId thay vì tham chiếu đến đối tượng Consumer. Tương tự, một Order tham chiếu đến Restaurant bằng cách sử dụng restaurantId.

Tổng hợp người tiêu dùng

Tổng hợp đơn hàng

Thông tin thanh toán

consumerId restaurantId

...

«gốc tổng hợp" Đặt hàng

Thông tin giao hàng

Số lượng

Mục đơn hàng

...

«gốc tổng hợp" Người tiêu dùng

«gốc tổng hợp" Nhà hàng

...

Tổng hợp nhà hàng

Thông tin thanh toán

Thông tin giao hàng

**Hình 5.6 Tham chiếu giữatổng hợp được thực hiện theo khóa chính thay vì theo tham chiếu đối tượng.Đặt hàngtổng hợp có ID củaNgười tiêu dùngVàNhà hàngtổng hợp. Trong một tổng hợp, các đối tượng có tham chiếu đến nhau.**

Cách tiếp cận này khá khác so với mô hình hóa đối tượng truyền thống, coi khóa ngoại trong mô hình miền là mùi thiết kế. Nó có một số lợi ích. Việc sử dụng danh tính thay vì tham chiếu đối tượng có nghĩa là các tổng hợp được kết hợp lỏng lẻo. Nó đảm bảo rằng ranh giới tổng hợp giữa các tổng hợp được xác định rõ ràng và tránh vô tình cập nhật một tổng hợp khác. Ngoài ra, nếu một tổng hợp là một phần của dịch vụ khác, thì không có vấn đề gì về tham chiếu đối tượng bao trùm các dịch vụ.

Cách tiếp cận này cũng đơn giản hóa tính bền bỉ vì tổng hợp là đơn vị lưu trữ. Nó giúp lưu trữ tổng hợp dễ dàng hơn trong cơ sở dữ liệu NoSQL như MongoDB. Nó cũng

loại bỏ nhu cầu tải lười minh bạch và các vấn đề liên quan. Việc mở rộng cơ sở dữ liệu bằng cách phân mảnh tổng hợp tương đối đơn giản.

**RULE#3: ỒGIAO DỊCH NE TẠO HOẶC CẬP NHẬT MỘT TỔNG HỢP**

Một quy tắc khác mà các tổng hợp phải tuân thủ là một giao dịch chỉ có thể tạo hoặc cập nhật một tổng hợp duy nhất. Khi tôi lần đầu tiên đọc về nó cách đây nhiều năm, quy tắc này không có ý nghĩa gì! Vào thời điểm đó, tôi đang phát triển các ứng dụng monolithic truyền thống sử dụng RDBMS, do đó các giao dịch có thể cập nhật nhiều tổng hợp. Ngày nay, ràng buộc này hoàn hảo cho kiến ​​trúc dịch vụ vi mô. Nó đảm bảo rằng một giao dịch được chứa trong một dịch vụ. Ràng buộc này cũng phù hợp với mô hình giao dịch hạn chế của hầu hết các cơ sở dữ liệu NoSQL.

Quy tắc này làm phức tạp hơn việc triển khai các hoạt động cần tạo hoặc cập nhật nhiều tổng hợp. Nhưng đây chính xác là vấn đề mà saga (được mô tả trong chương 4) được thiết kế để giải quyết. Mỗi bước của saga tạo hoặc cập nhật chính xác một tổng hợp. Hình 5.7 cho thấy cách thức hoạt động của nó.

Truyện dài

Dịch vụ A

Dịch vụ B

Giao dịch địa phương 1

Giao dịch địa phương 2

Giao dịch địa phương 3

Tạo/cập nhật

Tạo/cập nhật

Tạo/cập nhật

Tổng hợp Z

Tổng hợp Y

Tổng hợp X

**Hình 5.7 Một giao dịch chỉ có thể tạo hoặc cập nhật một tổng hợp duy nhất, do đó một ứng dụng sử dụng một sagađể cập nhật nhiều tổng hợp. Mỗi bước của saga tạo ra hoặc cập nhật một tổng hợp.**

Trong ví dụ này, saga bao gồm ba giao dịch. Giao dịch đầu tiên cập nhật tổng hợp X trong dịch vụ A. Hai giao dịch còn lại đều trong dịch vụ B. Một giao dịch cập nhật tổng hợp X và giao dịch còn lại cập nhật tổng hợp Y.

Một cách tiếp cận thay thế để duy trì tính nhất quán giữa nhiều tổng hợp trong một dịch vụ duy nhất là gian lận và cập nhật nhiều tổng hợp trong một giao dịch. Ví dụ, dịch vụ B có thể cập nhật tổng hợp Y và Z trong một giao dịch duy nhất. Điều này chỉ khả thi khi sử dụng cơ sở dữ liệu, chẳng hạn như RDBMS, hỗ trợ mô hình giao dịch phong phú. Nếu bạn đang sử dụng cơ sở dữ liệu NoSQL chỉ có các giao dịch đơn giản, không có tùy chọn nào khác ngoài việc sử dụng sagas.

Hay là có? Hóa ra là ranh giới tổng hợp không phải là bất di bất dịch. Khi phát triển một mô hình miền, bạn có thể chọn ranh giới nằm ở đâu. Nhưng giống như một cường quốc thực dân thế kỷ 20 vẽ ranh giới quốc gia, bạn cần phải cẩn thận.

###### Độ chi tiết tổng hợp

Khi phát triển một mô hình miền, một quyết định quan trọng mà bạn phải đưa ra là làm cho mỗi tổng hợp lớn đến mức nào. Một mặt, tổng hợp lý tưởng nhất là nên nhỏ. Vì các bản cập nhật cho mỗi tổng hợp được tuần tự hóa, các tổng hợp chi tiết hơn sẽ làm tăng số lượng yêu cầu đồng thời mà ứng dụng có thể xử lý, cải thiện khả năng mở rộng. Nó cũng sẽ cải thiện trải nghiệm của người dùng vì nó làm giảm khả năng hai người dùng cố gắng cập nhật xung đột của cùng một tổng hợp. Mặt khác, vì tổng hợp là phạm vi giao dịch, bạn có thể cần phải xác định một tổng hợp lớn hơn để làm cho một bản cập nhật cụ thể trở thành nguyên tử.

Ví dụ, trước đó tôi đã đề cập đến cách trong mô hình miền của ứng dụng FTGOOrder và Consumer là các tập hợp riêng biệt. Một thiết kế thay thế là biến Order thành một phần của tập hợp Consumer. Hình 5.8 cho thấy thiết kế thay thế này.

Tổng hợp người tiêu dùng

Thông tin thanh toán

Thông tin thanh toán

...

nhà hàngId

...

Thông tin giao hàng

<<gốc tổng hợp>>

Người tiêu dùng

Đặt hàng

Thông tin giao hàng

Số lượng

Mục đơn hàng

<<gốc tổng hợp>>

Nhà hàng

...

Tổng hợp nhà hàng

**Hình 5.8 Một thiết kế thay thế xác định mộtKhách hàngtổng hợp có chứaKhách hàngVàĐặt hànglớp. Thiết kế này cho phép một ứng dụng cập nhật nguyên tử mộtNgười tiêu dùngvà một hoặc nhiềucủa nóĐơn hàng.**

Một lợi ích của tổng hợp Consumer lớn hơn này là ứng dụng có thể cập nhật nguyên tử một Consumer và một hoặc nhiều Orders của nó. Một nhược điểm của cách tiếp cận này là nó làm giảm khả năng mở rộng. Các giao dịch cập nhật các orders khác nhau cho cùng một khách hàng sẽ được tuần tự hóa. Tương tự như vậy, hai người dùng sẽ xung đột nếu họ cố gắng chỉnh sửa các orders khác nhau cho cùng một khách hàng.

Một nhược điểm khác của cách tiếp cận này trong kiến ​​trúc vi dịch vụ là nó là mộttrở ngại cho việc phân tích. Ví dụ, logic kinh doanh cho Orders và Consumers phải được đặt chung trong cùng một dịch vụ, điều này làm cho dịch vụ lớn hơn. Do những vấn đề này, tốt nhất là tạo ra các tổng hợp càng chi tiết càng tốt.

###### Thiết kế logic kinh doanh với tổng hợp

Trong một dịch vụ (vi mô) điển hình, phần lớn logic kinh doanh bao gồm các tổng hợp. Phần còn lại của logic kinh doanh nằm trong các dịch vụ miền và sagas. Các sagas sắp xếp các chuỗi giao dịch cục bộ để thực thi tính nhất quán của dữ liệu. Các dịch vụ là điểm vào logic kinh doanh và được gọi bởi các bộ điều hợp đến. Một dịch vụ sử dụng một kho lưu trữ để truy xuất các tổng hợp từ cơ sở dữ liệu hoặc lưu các tổng hợp vào cơ sở dữ liệu. Mỗi kho lưu trữ được triển khai bởi một bộ điều hợp đi truy cập vào cơ sở dữ liệu. Hình 5.9 cho thấy thiết kế dựa trên tổng hợp của logic kinh doanh cho Dịch vụ đơn hàng.

Lãnh địa

nhà xuất bản sự kiện

voidSave(Order) Hoặc findOne(id)

...

«kho lưu trữ»

Kho lưu trữ đơn hàng

số lượng menuTên mục

«đối tượng giá trị»

Mục đơn hàng

nhận dạng

...

«tổng hợp»

Đặt hàng

createOrder() sửa lạiOrder() hủyOrder()

"dịch vụ"

Dịch vụ đặt hàng

Giao diện lập trình ứng dụng REST

Bộ điều hợp cơ sở dữ liệu

«truyện dài»Sửa đổi Order Saga

«truyện dài»Tạo đơn hàng Saga

Trình xử lý lệnh đặt hàng

**Hình 5.9 Một thiết kế dựa trên tổng hợp choDịch vụ đặt hànglogic kinh doanh**

Logic kinh doanh bao gồmĐặt hàngtổng hợp, cácDịch vụ đặt hànglớp dịch vụ,Kho lưu trữ đơn hàngvà một hoặc nhiều saga.Dịch vụ đặt hànggọi làĐặt hàng-Kho lưu trữđể lưu và tảiĐơn hàng. Đối với các yêu cầu đơn giản mang tính cục bộ đối với dịch vụ,

dịch vụ cập nhật tổng hợp Đơn hàng. Nếu yêu cầu cập nhật trải dài trên nhiều dịch vụ,OrderService cũng sẽ tạo ra một saga như được mô tả trong chương 4.

Chúng ta sẽ xem xét mã—nhưng trước tiên, hãy cùng xem xét một khái niệm có liên quan chặt chẽ đến tổng hợp: sự kiện miền.

#### Xuất bản sự kiện miền

Merriam-Webster(<https://www.merriam-webster.com/dictionary/event>) liệt kê một số định nghĩa của từ sự kiện, bao gồm:

**1** Một cái gì đó xảy ra

**2** Một sự kiện đáng chú ý

**3** Một dịp hoặc hoạt động xã hội

**4**Một sự kiện y tế bất lợi hoặc gây tổn hại, một cơn đau tim hoặc các sự kiện tim khác

Trong ngữ cảnh của DDD, một sự kiện miền là một điều gì đó đã xảy ra với một aggreg-gate. Nó được biểu diễn bởi một lớp trong mô hình miền. Một sự kiện thường biểu diễn một sự thay đổi trạng thái. Ví dụ, hãy xem xét một tổng hợp Order trong ứng dụng FTGO. Các sự kiện thay đổi trạng thái của nó bao gồm Order Created, Order Cancelled, Order Shipped, v.v. Một tổng hợp Order có thể, nếu có người tiêu dùng quan tâm, công bố một trong các sự kiện mỗi khi nó trải qua quá trình chuyển đổi trạng thái.

**Mẫu: Sự kiện miền**

Tổng hợp sẽ công bố sự kiện miền khi sự kiện đó được tạo hoặc trải qua một số thay đổi quan trọng khác.

###### Tại sao phải công bố các sự kiện thay đổi?

Sự kiện miền hữu ích vì các bên khác—người dùng, các ứng dụng khác hoặc các thành phần khác trong cùng một ứng dụng—thường quan tâm đến việc biết về các thay đổi trạng thái của tổng hợp. Sau đây là một số ví dụ về tình huống:

* + - * Duy trì tính nhất quán của dữ liệu trêndịch vụ sử dụng các truyện kể dựa trên vũ đạo, được mô tả trong chương 4.
      * Thông báo cho dịch vụ duy trì bản sao rằng dữ liệu nguồn đã thay đổi. Phương pháp này được gọi là Phân tách trách nhiệm truy vấn lệnh (CQRS) và được mô tả trong chương 7.
      * Thông báo một ứng dụng khác thông qua một đăng kýwebhook hoặc thông qua một môi giới tin nhắn để kích hoạt bước tiếp theo trong quy trình kinh doanh.
      * Thông báo cho một thành phần khác của cùng một ứng dụng để, ví dụ, gửi tin nhắn WebSocket đến trình duyệt của người dùng hoặc cập nhật cơ sở dữ liệu văn bản như ElasticSearch.
      * Gửi thông báo—tin nhắn văn bản hoặc email—cho người dùng để thông báo rằng đơn hàng của họ đã được chuyển đi, đơn thuốc theo toa của họ đã sẵn sàng để nhận hoặc chuyến bay của họ bị hoãn.
        + Giám sát các sự kiện miền để xác minh rằng ứng dụngđang hoạt động đúng.
        + Phân tích các sự kiện để mô hình hóa hành vi của người dùng.

Yếu tố kích hoạt thông báo trong tất cả các tình huống này là sự thay đổi trạng thái của tổng hợp trong cơ sở dữ liệu của ứng dụng.

###### Tên miền là gìsự kiện?

Một sự kiện miền là một lớp có tên được hình thành bằng cách sử dụng động từ phân từ quá khứ. Nó có các thuộc tính truyền đạt sự kiện một cách có ý nghĩa. Mỗi thuộc tính là một giá trị nguyên thủy hoặc mộtđối tượng giá trị. Ví dụ, lớp sự kiện OrderCreated có thuộc tính orderId.

Một sự kiện miền thường cũng có siêu dữ liệu, chẳng hạn như ID sự kiện và dấu thời gian.

Nó cũng có thể có danh tính của người dùng đã thực hiện thay đổi, vì điều đó hữu ích cho việc kiểm toán. Siêu dữ liệu có thể là một phần của đối tượng sự kiện, có thể được định nghĩa trong siêu lớp. Ngoài ra, siêu dữ liệu sự kiện có thể nằm trong đối tượng envelope bao bọc đối tượng sự kiện. ID của tổng hợp phát ra sự kiện cũng có thể là một phần của envelope chứ không phải là thuộc tính sự kiện rõ ràng.

CácĐơn hàngĐã tạoevent là một ví dụ về sự kiện miền. Nó không có bất kỳ trường nào, vì ID của Order là một phần của phong bì sự kiện. Danh sách sau đây hiển thịĐơn hàngĐã tạolớp sự kiện vàMiềnSự kiệnPhong bìlớp học.

**Liệt kê 5.1Đơn hàngĐã tạosự kiện vàMiềnSự kiệnPhong bìlớp học**

giao diện DomainEvent {}

giao diện OrderDomainEvent mở rộng DomainEvent {} lớp OrderCreated triển khai OrderDomainEvent {} lớp DomainEventEnvelope<T mở rộng DomainEvent> {

riêng tư String aggregateType;

Đối tượng riêng aggregateId; sự kiện T riêng;

...

}

**Sự kiện củasiêu dữ liệu**

Giao diện DomainEvent là một điểm đánh dấugiao diện xác định một lớp là một miềnsự kiện. OrderDomainEvent là giao diện đánh dấu cho các sự kiện, chẳng hạn như OrderCreated,được xuất bản bởi tổng hợp Order. DomainEventEnvelope là một lớp chứa siêu dữ liệu sự kiện và đối tượng sự kiện. Đây là một lớp chung được tham số hóa theo loại sự kiện miền.

###### Sự kiện làm phong phú

Ví dụ, hãy tưởng tượng rằng bạn đang viết một trình tiêu thụ sự kiện xử lý Đơn hàngsự kiện. Lớp sự kiện OrderCreated được hiển thị trước đó nắm bắt được bản chất của những gì đã xảy ra. Nhưng người dùng sự kiện của bạn có thể cần thông tin chi tiết về đơn hàng khi xử lý

Sự kiện OrderCreated. Một tùy chọn là để nó lấy thông tin đó từ Order-Service. Nhược điểm của người dùng sự kiện truy vấn dịch vụ cho tổng hợp là nó phát sinh chi phí chung của yêu cầu dịch vụ.

Một cách tiếp cận thay thế được gọi là làm giàu sự kiện là để các sự kiện chứa thông tin mà người tiêu dùng cần. Nó đơn giản hóa người tiêu dùng sự kiện vì họ không còn cần phải yêu cầu dữ liệu đó từ dịch vụ đã xuất bản sự kiện. Trong sự kiện OrderCreated, tổng hợp Order có thể làm giàu sự kiện bằng cách bao gồm các chi tiết đơn hàng. Danh sách sau đây hiển thị sự kiện đã được làm giàu.

**Liệt kê 5.2 Làm giàuĐơn hàngĐã tạosự kiện**

lớp OrderCreated triển khai OrderEvent { private List<OrderLineItem> lineItems;

riêng tư DeliveryInformation deliveryInformation; riêng tư PaymentInformation paymentInformation; riêng tư long restaurantId;

chuỗi riêng tư restaurantName;

...

**Dữ liệu mà người tiêu dùng thường cần**

}

Bởi vì phiên bản này củaĐơn hàngĐã tạosự kiện chứachi tiết đơn hàng, một sự kiệnngười tiêu dùng, chẳng hạn nhưDịch vụ Lịch sử đơn hàng(được thảo luận trong chương 7) không còn cần phải lấy dữ liệu đó khi xử lýĐơn hàngĐã tạosự kiện.

Mặc dù sự kiện làm giàu đơn giản hóa người dùng, nhược điểm là nó có nguy cơ làm cho các lớp sự kiện kém ổn định hơn. Một lớp sự kiện có khả năng cần phải thay đổi bất cứ khi nào các yêu cầu của người dùng thay đổi. Điều này có thể làm giảm khả năng bảo trì vì loại thay đổi này có thể tác động đến nhiều phần của ứng dụng. Việc đáp ứng mọi người dùng cũng có thể là một nỗ lực vô ích. May mắn thay, trong nhiều tình huống, khá rõ ràng những thuộc tính nào cần đưa vào một sự kiện.

Bây giờ chúng ta đã tìm hiểu những kiến ​​thức cơ bản về sự kiện miền, hãy cùng xem cách khám phá chúng.

###### Xác định sự kiện miền

Có một số chiến lược khác nhau để xác định các sự kiện miền. Thông thường, các yêu cầu sẽ mô tả các tình huống cần thông báo. Các yêu cầu có thể bao gồm ngôn ngữ như "Khi X xảy ra thì thực hiện Y". Ví dụ, một yêu cầu trong ứng dụng FTGO là "Khi có Đơn hàng được đặt, hãy gửi email cho người tiêu dùng". Yêu cầu thông báo cho thấy sự tồn tại của một sự kiện miền.

Một cách tiếp cận khác, ngày càng phổ biến, là sử dụng event storming. Event storming là một định dạng hội thảo lấy sự kiện làm trung tâm để hiểu một lĩnh vực phức tạp. Nó bao gồm việc tập hợp các chuyên gia trong lĩnh vực trong một căn phòng, nhiều ghi chú dán và một bề mặt rất lớn—bảng trắng hoặc cuộn giấy—để dán ghi chú. Kết quả của event storming là một mô hình miền lấy sự kiện làm trung tâm bao gồm các tổng hợp và sự kiện.

Sự kiện bão tápbao gồm ba bước chính:

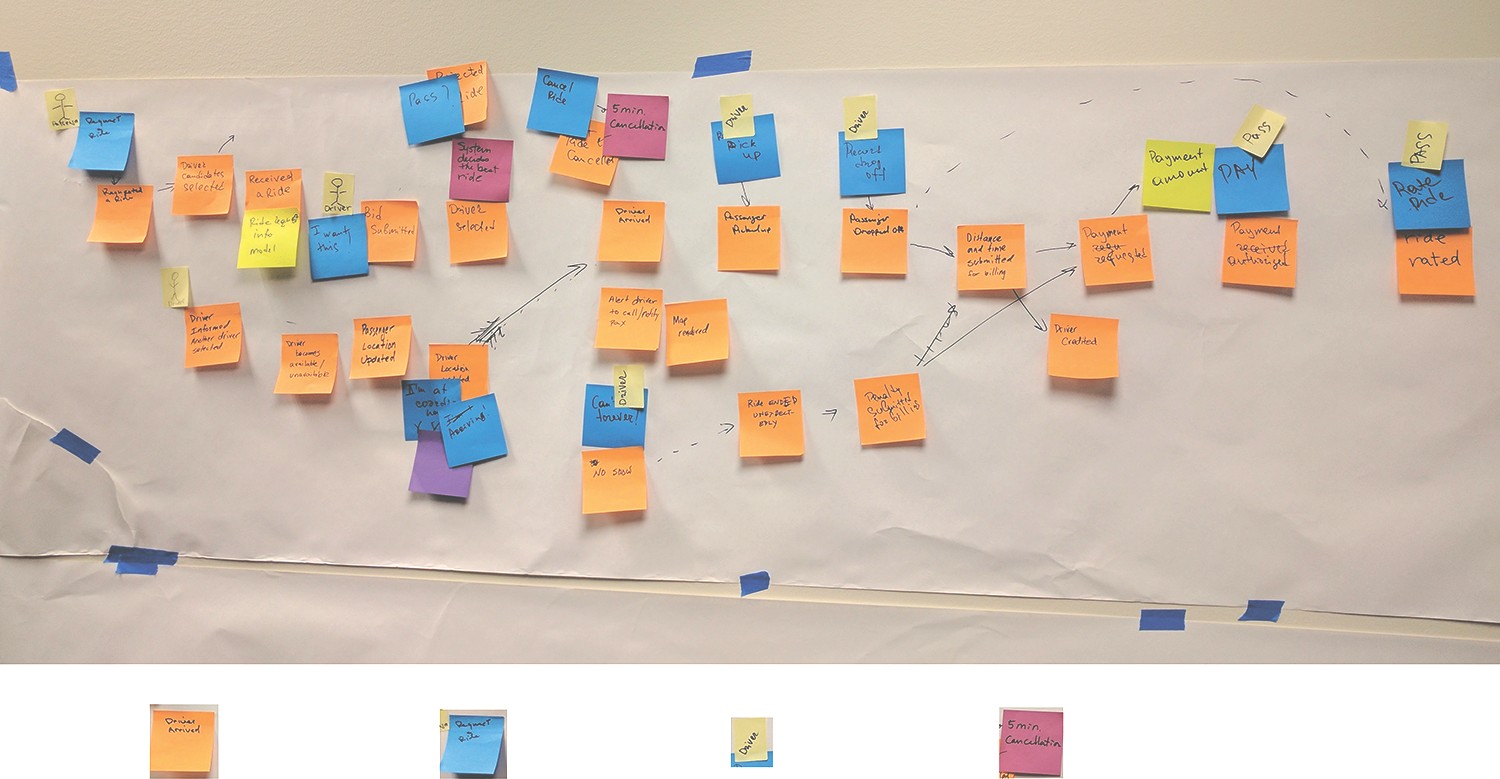
**1***Sự kiện động não*—Yêu cầu các chuyên gia về miền đưa ra ý tưởng về các sự kiện miền. Các sự kiện miền được thể hiện bằng các ghi chú màu cam được trình bày theo dòng thời gian sơ bộ trên bề mặt mô hình.

**2***Xác định các sự kiện kích hoạt*—Yêu cầu các chuyên gia trong lĩnh vực này xác định nguyên nhân gây ra mỗi sự kiện, đó là một trong những điều sau:

* Hành động của người dùng, được thể hiện dưới dạng lệnh bằng ghi chú màu xanh
* Hệ thống bên ngoài,được biểu thị bằng một tờ giấy nhớ màu tím
* Một sự kiện miền khác
* Thời gian trôi qua

**3***Xác định tổng hợp*—Yêu cầu các chuyên gia trong lĩnh vực này xác địnhtổng hợp tiêu thụ mỗi lệnh và phát ra sự kiện tương ứng. Tổng hợp được biểu thị bằng các ghi chú màu vàng.

Hình 5.10 cho thấykết quả của hội thảo event-storming. Chỉ trong vài giờ, những người tham gia đã xác định được nhiều sự kiện, lệnh và tổng hợp miền. Đây là bước đầu tiên tốt trong quá trình tạo mô hình miền.



Sự kiện

Yêu cầu

Tổng hợp

Chính sách

**Hình 5.10 Kết quả của một hội thảo về sự kiện bão táp kéo dài vài giờ.Các ghi chú dán là các sự kiện được trình bày theo dòng thời gian; các lệnh thể hiện hành động của người dùng; vàtập hợp, phát ra các sự kiện để đáp ứng một lệnh.**

Bão sự kiện là một kỹ thuật hữu ích để nhanh chóng tạo ra mô hình miền.

Bây giờ chúng ta đã tìm hiểu những điều cơ bản về sự kiện miền, hãy cùng xemvề cơ chế tạo ra và xuất bản chúng.

###### Tạo và xuất bản sự kiện miền

Giao tiếp bằng tên miềnevents là một dạng nhắn tin không đồng bộ, được thảo luận trong chương 3. Nhưng trước khi logic kinh doanh có thể xuất bản chúng tới một nhà môi giới tin nhắn, trước tiên nó phải tạo chúng. Hãy cùng xem cách thực hiện điều đó.

**GTẠO RA SỰ KIỆN MIỀN**

Về mặt khái niệm, các sự kiện miền được xuất bản bởi các tổng hợp. Một tổng hợp biết khi nào trạng thái của nó thay đổi và do đó biết sự kiện nào cần xuất bản. Một tổng hợp có thể gọi trực tiếp một API nhắn tin. Nhược điểm của cách tiếp cận này là vì các tổng hợp không thể sử dụng tiêm phụ thuộc, API nhắn tin sẽ cần được truyền xung quanh như một đối số phương thức. Điều đó sẽ đan xen các mối quan tâm về cơ sở hạ tầng và logic kinh doanh, điều này cực kỳ không mong muốn.

Một cách tiếp cận tốt hơn là phân chia trách nhiệm giữa tổng hợp và dịch vụ (hoặc lớp tương đương) gọi nó. Các dịch vụ có thể sử dụng tiêm phụ thuộc để có được tham chiếu đến API nhắn tin, dễ dàng xuất bản các sự kiện. Tổng hợp tạo ra các sự kiện bất cứ khi nào trạng thái của nó thay đổi và trả về dịch vụ. Có một vài cách khác nhau mà tổng hợp có thể trả về các sự kiện trở lại dịch vụ. Một tùy chọn là giá trị trả về của phương thức tổng hợp bao gồm danh sách các sự kiện. Ví dụ:danh sách sau đây cho thấy cách phương thức accept() của tổng hợp Ticket có thể trả về một Ticket-AcceptedEvent cho người gọi nó.

**Liệt kê 5.3Vétổng hợp củachấp nhận()phương pháp**

lớp công cộng Vé {

công khai List<DomainEvent> chấp nhận(ZonedDateTime readyBy) {

...

this.acceptTime = ZonedDateTime.now(); this.readyBy = readyBy;

trả về singletonList(new TicketAcceptedEvent(readyBy));

}

}

**Cập nhật Vé**



**Trả về mộtsự kiện**

Dịch vụ này gọi phương thức gốc tổng hợp, sau đó xuất bản các sự kiện. Ví dụ: danh sách sau đây cho thấy cách KitchenService gọi Ticket.accept() và xuất bản các sự kiện.

**Liệt kê 5.4 Dịch vụ nhà bếpgọiVé.chấp nhận()**

lớp công khai KitchenService { @Autowired

TicketRepository riêng tư ticketRepository;

@Autowired

riêng tư DomainEventPublisher domainEventPublisher;

public void accept(long ticketId, ZonedDateTime readyBy) { Vé ticket =

ticketRepository.findById(ticketId)

.orElseThrow(() ->

mới TicketNotFoundException(ticketId)); Danh sách<DomainEvent> sự kiện = ticket.accept(readyBy); domainEventPublisher.publish(Ticket.class, orderId, sự kiện);

**Xuất bảnsự kiện miền**

}

Cácchấp nhận()phương pháp đầu tiên gọiKho lưu trữ véđể tảiVétừcơ sở dữ liệu. Sau đó nó cập nhậtVébằng cách gọichấp nhận().Dịch vụ nhà bếpsau đó xuất bảnsự kiện lishes trả về bởiVébằng cách gọiDomainEventPublisher.publish(), được mô tả ngắn gọn.

Cách tiếp cận này khá đơn giản. Các phương thức nếu không có kiểu trả về void thì giờ đây sẽ trả về List<Event>. Nhược điểm tiềm ẩn duy nhất là kiểu trả về của các phương thức không phải void giờ đây phức tạp hơn. Chúng phải trả về một đối tượng chứa giá trị trả về ban đầu và List<Event>. Bạn sẽ sớm thấy một ví dụ về phương thức như vậy. Một tùy chọn khác là để gốc tổng hợp tích lũy các sự kiện trong một trường. Sau đó, dịch vụ sẽ truy xuất các sự kiện và xuất bản chúng. Ví dụ, danh sách sau

hiển thị một biến thể của lớp Ticket hoạt động theo cách này.

**Liệt kê 5.5Vémở rộng một siêu lớp, ghi lại các sự kiện miền**

lớp công khai Ticket mở rộng AbstractAggregateRoot { công khai void accept(ZonedDateTime readyBy) {

...

this.acceptTime = ZonedDateTime.now(); this.readyBy = readyBy;

registerDomainEvent(mớiSự kiện chấp nhận vé (readyBy));

}

}

Vémở rộng Tóm tắtTổng hợpRoot,trong đó định nghĩa mộtđăng kýDomainEvent()phương pháp ghi lại sự kiện. Một dịch vụ sẽ gọiTóm tắt AggregateRoot.get- DomainEvents()để lấy lại những sự kiện đó.

Tôi thích phương án đầu tiên: phương pháp trả về sự kiện cho dịch vụ. Nhưng tích lũy sự kiện trong gốc tổng hợp cũng là một phương án khả thi. Trên thực tế, Spring Data Ingalls phát hành train ([https://spring.io/blog/2017/01/30/what-s-new-in-](https://spring.io/blog/2017/01/30/what-s-new-in-spring-data-release-ingalls) [spring-data-release-ingalls](https://spring.io/blog/2017/01/30/what-s-new-in-spring-data-release-ingalls)) triển khai một cơ chế tự động xuất bản các sự kiện tới Spring ApplicationContext. Nhược điểm chính là để giảm trùng lặp mã, các gốc tổng hợp nên mở rộng một siêu lớp như AbstractAggregate-Root, điều này có thể xung đột với yêu cầu mở rộng một siêu lớp khác. Một vấn đề khác là mặc dù các phương thức của gốc tổng hợp dễ dàng gọi register-DomainEvent(), các phương thức trong các lớp khác trong tổng hợp sẽ thấy khó khăn. Chúng chủ yếu có thể cần phải truyền các sự kiện cho gốc tổng hợp theo một cách nào đó.

**HCÁCH ĐỂ XUẤT BẢN CÁC SỰ KIỆN MIỀN MỘT CÁCH ĐÁNG TIN CẬY?**

Chương 3 nói về cách gửi tin nhắn đáng tin cậy như một phần của giao dịch cơ sở dữ liệu cục bộ. Các sự kiện miền không khác gì. Một dịch vụ phải sử dụng tin nhắn giao dịch để xuất bản các sự kiện nhằm đảm bảo rằng chúng được xuất bản như một phần của giao dịch cập nhật tổng hợp trong cơ sở dữ liệu. Khung Eventuate Tram, được mô tả trong chương 3, triển khai một cơ chế như vậy. Nó chèn các sự kiện vào bảng OUTBOX như một phần của giao dịch ACID cập nhật cơ sở dữ liệu. Sau khi giao dịch được xác nhận, các sự kiện đã được chèn vào bảng OUTBOX sau đó được xuất bản tới trình môi giới tin nhắn.

Khung Tram cung cấp giao diện DomainEventPublisher, được hiển thị trong danh sách sau. Nó định nghĩa một số phương thức publish() quá tải lấy loại tổng hợp và ID làm tham số, cùng với danh sách các sự kiện miền.

**Danh sách 5.6 Khung xe điện EventuateNhà xuất bản sự kiện miềngiao diện**

giao diện công khai DomainEventPublisher {

void publish(String aggregateType, Object aggregateId, List<DomainEvent> domainEvents);

Nó sử dụng giao diện MessageProducer của khung Eventuate Tram để xuất bản các sự kiện đó theo giao dịch.

Một dịch vụ có thể gọi trực tiếp trình xuất bản DomainEventPublisher. Nhưng một nhược điểm của việc làm như vậy là nó không đảm bảo rằng một dịch vụ chỉ xuất bản các sự kiện hợp lệ. Ví dụ, KitchenService chỉ nên xuất bản các sự kiện triển khai Ticket-DomainEvent, là giao diện đánh dấu cho các sự kiện tổng hợp Ticket. Một tùy chọn tốt hơn là các dịch vụ triển khai một lớp con của AbstractAggregateDomainEvent-Publisher, được hiển thị trong danh sách 5.7. AbstractAggregateDomainEventPublisher là một lớp trừu tượng cung cấp giao diện an toàn về kiểu để xuất bản các sự kiện miền. Đây là một lớp chung có hai tham số kiểu, A, kiểu tổng hợp và E, kiểu giao diện đánh dấu cho các sự kiện miền. Một dịch vụ xuất bản các sự kiện bằng cách gọi phương thức publish(), có hai tham số: một tổng hợp kiểu A và một danh sách các sự kiện kiểu E.

**Liệt kê 5.7 Siêu lớp trừu tượng của các nhà xuất bản sự kiện miền an toàn kiểu**

lớp trừu tượng công khai AbstractAggregateDomainEventPublisher<A, E mở rộng Doma inEvent> {

riêng tư Function<A, Object> idSupplier; riêng tư DomainEventPublisher eventPublisher; riêng tư Class<A> aggregateType;

được bảo vệ AbstractAggregateDomainEventPublisher(DomainEventPublisher sự kiệnPublisher,

Lớp<A> aggregateType, Hàm<A, Đối tượng> idSupplier) {

this.eventPublisher = eventPublisher; this.aggregateType = aggregateType;

this.idSupplier = idSupplier;

}

public void publish(A aggregate, List<E> events) { eventPublisher.publish(aggregateType, idSupplier.apply(aggregate),

(Danh sách<DomainEvent>) sự kiện);

}

}

Cácxuất bản()phương pháp lấy ID của tổng hợp và gọiNhà xuất bản sự kiện miền

. xuất bản(). Danh sách sau đây cho thấyNhà xuất bản sự kiện TicketDomain,trong đó xuất bản các sự kiện miền choVétổng hợp.

**Liệt kê 5.8 Một giao diện an toàn kiểu để xuất bảnVésự kiện miền tổng hợp**

lớp công khai TicketDomainEventPublisher mở rộng AbstractAggregateDomainEventPublisher<Ticket, TicketDomainEvent> {

công khai TicketDomainEventPublisher(DomainEventPublisher eventPublisher) { siêu(eventPublisher, Ticket.class, Ticket::getId);

}

}

Lớp này chỉ xuất bản các sự kiện là một lớp con củaSự kiện TicketDomain.

Bây giờ chúng ta đã xem cách xuất bản các sự kiện miền, hãy cùng xem cách sử dụng chúng.

###### Tiêu thụ sự kiện miền

Sự kiện miền cuối cùng được xuất bản dưới dạng tin nhắn đến một nhà môi giới tin nhắn, chẳng hạn như Apache Kafka. Người dùng có thể sử dụng API máy khách của nhà môi giới trực tiếp. Nhưng sẽ thuận tiện hơn khi sử dụng API cấp cao hơn như Domain-EventDispatcher, được mô tả trong chương 3. DomainEventDispatcher phân phối các sự kiện miền đến phương thức xử lý thích hợp. Liệt kê 5.9 hiển thị một lớp xử lý sự kiện ví dụ. KitchenServiceEventConsumer đăng ký các sự kiện do Restaurant Service công bố bất cứ khi nào thực đơn của nhà hàng được cập nhật. Nó chịu trách nhiệm giữ cho bản sao dữ liệu của Kitchen Service được cập nhật.

**Liệt kê 5.9 Gửi sự kiện đến các phương thức xử lý sự kiện**

lớp công khai KitchenServiceEventConsumer {@Autowired

Nhà hàng riêngDịch vụ nhà hàngDịch vụ;

công khai DomainEventHandlers domainEventHandlers() {trả về DomainEventHandlersBuilder

**Sự kiện bản đồđến trình xử lý sự kiện**

.forAggregateType("net.chrisrichardson.ftgo.restaurantservice.Restaurant")

.onEvent(RestaurantMenuRevised.class, this::reviseMenu)

.xây dựng();

}

công khai void revisionMenu(DomainEventEnvelope<RestaurantMenuRevised> de) {id dài = Long.parseLong(de.getAggregateId());

RestaurantMenu revedMenu = de.getEvent().getRevisedMenu();

restaurantService.reviseMenu(id, revisionMenu);

} **Một trình xử lý sự kiện cho**

**Nhà hàngMenuĐã sửa đổi**

} **sự kiện**

Cácsửa lạiMenu()phương pháp xử lýNhà hàngMenuĐã sửa đổisự kiện. Nó gọinhà hàng- Service.reviseMenu(),cập nhật thực đơn của nhà hàng. Phương pháp đó trả về danh sách các sự kiện miền, được trình xử lý sự kiện công bố.

Bây giờ chúng ta đã xem xét các tổng hợp và sự kiện miền, đã đến lúc xem xét một số ví dụ về logic kinh doanh được triển khai bằng cách sử dụng các tổng hợp.

#### Logic kinh doanh dịch vụ nhà bếp

Ví dụ đầu tiên là Dịch vụ nhà bếp, cho phép nhà hàng quản lý đơn hàng. Hai tổng hợp chính trong dịch vụ này là tổng hợp Nhà hàng và Vé. Tổng hợp Nhà hàng biết thực đơn và giờ mở cửa của nhà hàng và có thể xác thực đơn hàng. Vé là đơn hàng mà nhà hàng phải chuẩn bị để chuyển phát nhanh đến lấy. Hình 5.11 cho thấy các tổng hợp này và các phần chính khác của logic kinh doanh của dịch vụ, cũng như các bộ điều hợp của dịch vụ.

Ngoài các tổng hợp, các bộ phận chính khác củaDịch vụ nhà bếplogic kinh doanh của 's làDịch vụ nhà bếp,Kho lưu trữ vé, VàNhà hàngRepository.Bếp-Dịch vụlà mục nhập logic kinh doanh. Nó xác định các phương pháp để tạo và cập nhậtNhà hàngVàVétổng hợp.Kho lưu trữ véVàNhà hàngRepositoryxác định các phương pháp để duy trìVéVàNhà hàngtương ứng.

CácDịch vụ nhà bếpdịch vụ có ba bộ điều hợp đến:

* + - Giao diện lập trình ứng dụng REST—API REST được gọi bởi giao diện người dùng được sử dụng bởi các công nhântạinhà hàng. Nó gợi lênDịch vụ nhà bếpđể tạo và cập nhậtVé.
    - KitchenServiceCommandHandler—API dựa trên yêu cầu/phản hồi không đồng bộ được sagas gọi. Nó gọiDịch vụ nhà bếpđể tạo và cập nhậtVé.
    - KitchenServiceEventConsumer—Đăng ký các sự kiện được xuất bản bởiDịch vụ nhà hàng. Nó gợi lênDịch vụ nhà bếpđể tạo và cập nhậtNhà hàng.

Dịch vụ này cũng có hai bộ điều hợp đầu ra:

* + - Bộ chuyển đổi DB—Thực hiệnKho lưu trữ vévàNhà hàngRepository

giao diện và truy cậpcơ sở dữ liệu.

* + - DomainEventPublishingAdapter—Thực hiệnNhà xuất bản sự kiện miềngiao diện vàxuất bảnVésự kiện miền.



Nhà hàng

chấp nhận từ chối chuẩn bị

sẵn sàng để nhậnnhặt được



Tạo vé

Xác nhận tạo ticket

Nhà hàng được tạo ra

Thực đơn nhà hàng đã được sửa đổi

Dịch vụ nhà bếp

kênh lệnh

Sự kiện nhà hàng

kênh

Sự kiện vé

kênh

Dịch vụ nhà bếp

cơ sở dữ liệu

Bộ điều hợp cơ sở dữ liệu

Bộ điều hợp xuất bản sự kiện miền

«kho lưu trữ» Kho lưu trữ nhà hàng

nhà hàng «tổng hợp»

Lãnh địanhà xuất bản sự kiện

«kho lưu trữ» Kho lưu trữ vé

Vé «tổng hợp»

Dịch vụ nhà bếp

Dịch vụ nhà bếp

Sự kiệnNgười tiêu dùng

Dịch vụ nhà bếp

Bộ xử lý lệnh

Giao diện lập trình ứng dụng REST

**Hình 5.11 Thiết kế củaDịch vụ nhà bếp**

Chúng ta hãy xem xét kỹ hơn thiết kế củaDịch vụ nhà bếp, bắt đầu bằngVé

tổng hợp.

###### 5.4.1 Tổng hợp vé

Vé là mộtcủa các tổng hợp của Kitchen Service. Như đã mô tả trong chương 2, khi nói về khái niệm Bounded Context, tổng hợp này thể hiện quan điểm của bếp nhà hàng về một đơn hàng. Nó không chứa thông tin về người tiêu dùng, chẳng hạn như danh tính của họ, thông tin giao hàng hoặc chi tiết thanh toán. Nó tập trung vào việc cho phép bếp nhà hàng chuẩn bị Đơn hàng để nhận. Hơn nữa, Kitchen-Service không tạo ID duy nhất cho tổng hợp này. Thay vào đó, nó sử dụng ID do OrderService cung cấp.

Chúng ta hãyĐầu tiên hãy xem cấu trúc của lớp này và sau đó chúng ta sẽ kiểm tra các phương thức của nó.

**SCẤU TRÚC CỦATLỚP ICKET**

Danh sách sau đây hiển thị một đoạn trích của mã choLớp này. Lớp Ticket tương tự như lớp miền truyền thống. Sự khác biệt chính là các tham chiếu đến các tập hợp khác được thực hiện theo khóa chính.

**Liệt kê 5.10 Một phần củaVélớp, là một thực thể JPA**

@Entity(bảng="vé")lớp công cộng Vé {

@Nhận dạng

private Long id;

trạng thái TicketState riêng tư; trạng thái Long restaurantId riêng tư;

@ElementCollection @CollectionTable(name="ticket\_line\_items") danh sách riêng tư<TicketLineItem> lineItems;

riêng tư ZonedDateTime readyBy; riêng tư ZonedDateTime acceptTime; riêng tư ZonedDateTime PreparingTime; riêng tư ZonedDateTimepickUpTime;

riêng tư ZonedDateTime readyForPickupTime;

...

Lớp này được lưu trữ với JPA và được ánh xạ tớiVÉbảng. Cácnhà hàngIdlĩnh vực là mộtDàithay vì một tham chiếu đối tượng đến mộtNhà hàng. Cácsẵn sàngBởitrường lưu trữ ước tính thời điểm đơn hàng sẽ sẵn sàng để nhận.Vélớp có một số trường theo dõi lịch sử của đơn hàng, bao gồmchấp nhậnThời gian,chuẩn bị- Thời gian, Vàgiờ đón. Chúng ta hãy xem xét các phương thức của lớp này.

**BHÀNH VI CỦATTỔNG HỢP ICKET**

Tổng hợp Vé xác định một sốphương pháp. Như bạn đã thấy trước đó, nó có phương pháp create() tĩnh, là phương pháp nhà máy tạo ra một Vé. Ngoài ra còn có một số phương pháp được gọi khi nhà hàng cập nhật trạng thái của đơn hàng:

* chấp nhận()—Nhà hàng đã chấp nhận đơn hàng.
* chuẩn bị()—Nhà hàng đã bắt đầu chuẩn bị đơn hàng, nghĩa là đơn hàng không thể thay đổi hoặc hủy nữa.
* readyForPickup()—Giờ đây có thể nhận đơn hàng. Danh sách sau đây cho thấy một số phương pháp của nó.

**Liệt kê 5.11 Một sốVé'Sphương pháp**

lớp công cộng Vé {

công khai tĩnh ResultWithAggregateEvents<Ticket, TicketDomainEvent>create(ID dài, chi tiết TicketDetails) {

trả về ResultWithAggregateEvents<>(newTicket(id, chi tiết), new TicketCreatedEvent(id, chi tiết));

}

công khai Danh sách<TicketPreparationStartedEvent>chuẩn bị() { chuyển đổi (trạng thái) {

trường hợp được chấp nhận:

trạng thái này = TicketState.CHUẨN BỊ; thời gian chuẩn bị = ZonedDateTime.now();

trả về singletonList(new TicketPreparationStartedEvent());mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

công khai List<TicketDomainEvent>hủy() { chuyển đổi (trạng thái) {

trường hợp ĐÃ TẠO:

trường hợp được chấp nhận:

this.state = TicketState.CANCELLED;

trả về singletonList(new TicketCancelled());trường hợp SẴN SÀNG ĐỂ NHẬN HÀNG:

ném ngoại lệ TicketCannotBeCancelledException mới();

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

Phương thức create() tạo ra một Ticket. Phương thức Preparing() được gọi khi nhà hàng bắt đầu chuẩn bị đơn hàng. Phương thức này thay đổi trạng thái của đơn hàng thành PREPARING, ghi lại thời gian và công bố sự kiện. Phương thức cancel() được gọi khi người dùng cố gắng hủy đơn hàng. Nếu được phép hủy, phương thức này sẽ thay đổi trạng thái của đơn hàng và trả về một sự kiện. Nếu không, phương thức này sẽ ném ra một ngoại lệ. Các phương thức này được gọi để phản hồi các yêu cầu REST API cũng như các sự kiện và thông báo lệnh. Hãy cùng xem các lớp gọi phương thức tổng hợp.

**TANH TAKNGỨASDỊCH VỤ MIỀN DỊCH VỤ**

KitchenService được gọi bởi các bộ điều hợp đến của dịch vụ. Nó định nghĩa nhiều phương thức khác nhau để thay đổi trạng thái của một đơn hàng, bao gồm accept(), reject(), Preparing() và các phương thức khác. Mỗi phương thức tải tổng hợp được chỉ định, gọi phương thức tương ứng trên gốc tổng hợp và xuất bản bất kỳ sự kiện miền nào. Danh sách sau đây hiển thị phương thức accept() của nó.

**Danh sách 5.12 Dịch vụchấp nhận()phương pháp cập nhậtVé**

lớp công khai KitchenService {

@Autowired

TicketRepository riêng tư ticketRepository;

@Autowired

riêng tư TicketDomainEventPublisher domainEventPublisher;

public void accept(long ticketId, ZonedDateTime readyBy) { Vé ticket =

ticketRepository.findById(ticketId)

.orElseThrow(() ->

mới TicketNotFoundException(ticketId));Danh sách<TicketDomainEvent> sự kiện = ticket.accept(readyBy); domainEventPublisher.publish(vé, sự kiện);

}

}

**Xuất bản sự kiện miền**

Phương thức accept() được gọi khi nhà hàng chấp nhận một đơn hàng mới. Nó có hai tham số:

* Mã đơn hàng—ID của lệnh cần chấp nhận
* sẵn sàngBởi—Thời gian ước tính khi đơn hàng sẽ sẵn sàng để nhận

Phương pháp này lấy tổng hợp Ticket và gọi phương thức accept() của nó. Nó công bố bất kỳ sự kiện nào được tạo ra.

Bây giờ chúng ta hãy xem lớp xử lýlệnh không đồng bộ.

**TANH TAKNGỨASDỊCH VỤCOMANDHANDLERLỚP HỌC**

Lớp KitchenServiceCommandHandler là một bộ điều hợp chịu trách nhiệm xử lýtin nhắn lệnh được gửi bởi các sagas khác nhau được Order Service triển khai. Lớp này định nghĩa một phương thức xử lý cho mỗi lệnh, gọi KitchenService để tạo hoặc cập nhật một Vé. Danh sách sau đây hiển thị một đoạn trích của lớp này.

**Liệt kê 5.13 Xử lý các tin nhắn lệnh được gửi bởitruyện dài**

lớp công khai KitchenServiceCommandHandler {

@Autowired

Dịch vụ bếp riêng Dịch vụ bếp riêng;

công khai CommandHandlers commandHandlers() { trả về CommandHandlersBuilder

.fromChannel("Dịch vụ đặt hàng")

**Bản đồ lệnh tin nhắn đến trình xử lý tin nhắn**

.onMessage(CreateTicket. lớp,này::createTicket)

.onMessage(Xác nhậnTạoTicket. lớp,này::xác nhậnTạoTicket)

.onMessage(HủyTạoTicket.lớp,này::cancelCreateTicket)

.xây dựng();

}

Tin nhắn riêng createTicket(CommandMessage<CreateTicket>

cm) {

Lệnh CreateTicket = cm.getCommand();

dài restaurantId = lệnh.getRestaurantId(); dài ticketId = lệnh.getOrderId(); TicketDetails ticketDetails =

lệnh.getTicketDetails();

thử {

Vé ticket = kitchenService.createTicket(restaurantId,

**Gọi KitchenService để tạo Ticket**

ticketId, ticketDetails);

CreateTicketReply trả lời =

mớiCreateTicketReply(ticket.getId()); trả về vớiThành công(trả lời);

} catch (RestaurantDetailsVerificationException e) { return withFailure();

**Gửi lại phản hồi thành công**

} **Gửi lại một**

} **trả lời thất bại**

Tin nhắn riêng tư xác nhậnCreateTicket (CommandMessage<ConfirmCreateTicket> cm) {

Long ticketId = cm.getCommand().getTicketId(); kitchenService.confirmCreateTicket(ticketId); return withSuccess();

**Xác nhận đơn hàng**

}

...

Tất cả các phương thức xử lý lệnh đều gọi KitchenServicevà trả lời thành công hoặc thất bại.

Bây giờ bạn đã thấy logic kinh doanh cho một dịch vụ tương đối đơn giản, chúng ta sẽ xem xétmột ví dụ phức tạp hơn: Dịch vụ đặt hàng.

#### Logic kinh doanh dịch vụ đặt hàng

Như đã đề cập trong các chương trước, OrderDịch vụ cung cấp API để tạo, cập nhật và hủy đơn hàng. API này chủ yếu được người tiêu dùng gọi. Hình 5.12 cho thấy thiết kế cấp cao của dịch vụ. Tổng hợp đơn hàng là tổng hợp trung tâmCổng của Order Service. Nhưng cũng có một tổng hợp Restaurant, là bản sao một phần dữ liệu do Restaurant Service sở hữu. Nó cho phép Order Service xác thực và định giá các mục hàng của Order.

Ngoài ra còn cóĐặt hàngVàNhà hàngtổng hợp, logic kinh doanh bao gồmDịch vụ đặt hàng,Kho lưu trữ đơn hàng,Nhà hàngRepositoryvà nhiều truyện cổ tích khác nhưcáiTạo đơn hàngSagađược mô tả trong chương 4.Dịch vụ đặt hànglà điểm vào chính trong logic kinh doanh và xác định các phương pháp để tạo và cập nhậtĐơn hàng



Người tiêu dùng

createOrder() hủy đơn hàng() sửa lại đơn hàng()

Sự kiện nhà hàngkênh



Sự kiện vé

kênh

Dịch vụ đặt hàng

kênh lệnh

Dịch vụ khách hàng

kênh lệnh

Dịch vụ nhà bếp

kênh lệnh

Dịch vụ kế toán

kênh lệnh

Tạo đơn hàng saga

kênh trả lời

Hủy đơn hàng saga

kênh trả lời

Sửa đổi lệnh saga

kênh trả lời

Bộ điều hợp cơ sở dữ liệu

«kho lưu trữ» OrderRepository

Nhà sản xuất lệnh

Lệnh gửi đibộ chuyển đổi tin nhắn

Dịch vụ đặt hàng

Bộ chuyển đổi tin nhắn SagaReply

«kho lưu trữ» Kho lưu trữ nhà hàng

«tổng hợp» Đặt hàng

Nhà hàng «tổng hợp»

«truyện dài»

\*Đặt hàngSaga

Nhà xuất bản sự kiện miền

Trình xử lý lệnh đặt hàng

Sự kiện miềnbộ chuyển đổi xuất bản

Sự kiện đặt hàngngười tiêu dùng

Giao diện lập trình ứng dụng REST

Dịch vụ đặt hàngcơ sở dữ liệu

**Hình 5.12 Thiết kế củaDịch vụ đặt hàng. Nó có REST API để quản lý đơn hàng. Nó trao đổi tin nhắnvà các sự kiện với các dịch vụ khác thông qua nhiều kênh tin nhắn.**

VàNhà hàng.Kho lưu trữ đơn hàngđịnh nghĩa các phương pháp để duy trìĐơn hàng, VàNhà hàngRepositorycó phương pháp để duy trìNhà hàng.Dịch vụ đặt hàngcó một số bộ điều hợp đầu vào:

* Giao diện lập trình ứng dụng REST—API REST được gọi bởi giao diện người dùng được sử dụngbởi người tiêu dùng. NógọiDịch vụ đặt hàngđể tạo và cập nhậtĐơn hàng.
  + Đơn hàngSự kiệnNgười tiêu dùng—Đăng ký các sự kiện được xuất bản bởiDịch vụ nhà hàng. NógọiDịch vụ đặt hàngđể tạo và cập nhật bản sao của nóNhà hàng.
  + Trình xử lý lệnh—API dựa trên yêu cầu/phản hồi không đồng bộ được sagas gọi. Nó gọiDịch vụ đặt hàngđể cập nhậtĐơn hàng.
  + Bộ chuyển đổi trả lời Saga—Đăng ký kênh trả lời saga và gọi saga. Dịch vụ này cũng có một số bộ điều hợp gửi đi:
  + Bộ chuyển đổi DB—Thực hiệnKho lưu trữ đơn hànggiao diện và truy cậpĐặt hàngDịch vụcơ sở dữ liệu
  + DomainEventPublishingAdapter—Thực hiệnNhà xuất bản sự kiện miềngiao diện vàxuất bảnĐặt hàngsự kiện miền
  + Bộ chuyển đổi tin nhắn OutboundCommand—Thực hiệncáiLệnhPublishergiao diện và gửi tin nhắn lệnh cho những người tham gia saga

Trước tiên chúng ta hãy xem xét kỹ hơnĐặt hàngtổng hợp và sau đó kiểm traDịch vụ đặt hàng.

###### Tổng hợp đơn hàng

Tổng hợp đơn hàng biểu thị đơn hàng do người tiêu dùng đặt. Trước tiên, chúng ta sẽ xem xét cấu trúc của tổng hợp đơn hàng và sau đó kiểm tra các phương thức của nó.

**TCẤU TRÚC CỦAỒĐÁ TỔNG HỢP RDER**

Hình 5.13 cho thấy cấu trúc củaĐặt hàngtổng hợp. CácĐặt hànglớp là gốc rễ củacáiĐặt hàngtổng hợp. CácĐặt hàngtổng hợp cũng bao gồm các đối tượng giá trị nhưĐơn hàng- Dòng mục,Thông tin giao hàng, VàThông tin thanh toán.

Đặt hàng tối thiểu

số lượng

«đối tượng giá trị» Tiền

phương thức thanh toánId

«đối tượng giá trị» PaymentInfo

trạng thái consumerId restaurantId

...

«tổng hợp» Đặt hàng

Thời gian giao hàng

«đối tượng giá trị» DeliveryInfo

phố1 phố2 thành phố tiểu bang mã bưu chính

«đối tượng giá trị» Địa chỉ

|  |  |
| --- | --- |
| «đối tượng giá trị» OrderLineItem | Giá |
| số lượng menuTên mục |
|  |

**Hình 5.13 Thiết kế củaĐặt hàngtổng hợp, bao gồmĐặt hànggốc tổng hợpvà nhiều đối tượng giá trị khác nhau.**

CácĐặt hànglớp có một bộ sưu tậpMục hàng. Bởi vìĐặt hàng'SNgười tiêu dùngVàNhà hànglà các tập hợp khác, nó tham chiếu chúng theo giá trị khóa chính.Đặt hànglớp học có mộtThông tin giao hànglớp, lưu trữ địa chỉ giao hàng và thời gian giao hàng mong muốn, vàThông tin thanh toán, nơi lưu trữ thông tin thanh toán. Danh sách sau đây hiển thị mã.

**Liệt kê 5.14Đặt hànglớp và các trường của nó**

@Entity @Table(name="orders") @Access(AccessType.FIELD) lớp công khai Order {

@Id @GeneratedValue private Mã số dài;

@Phiên bản

Phiên bản dài riêng tư;

trạng thái OrderState riêng tư; trạng thái ConsumerId dài riêng tư; trạng thái RestaurantId dài riêng tư;

@Nhúng

riêng tư OrderLineItems orderLineItems;

@Nhúng

riêng tư DeliveryInformation deliveryInformation;

@Nhúng

thông tin thanh toán riêng tư thông tin thanh toán;

@Nhúng

Lệnh chuyển tiền riêngTối thiểu;

Lớp này được lưu trữ với JPA và được ánh xạvào bảng ORDERS. Trường id làkhóa chính. Trường phiên bản được sử dụng để khóa lạc quan. Trạng thái của Đơn hàng được biểu diễn bằng phép liệt kê OrderState. Các trường DeliveryInformation và Payment- Information được ánh xạ bằng chú thích @Embedded và được lưu trữ dưới dạng các cột của bảng ORDERS. Trường orderLineItems là một đối tượng nhúng chứa các mục dòng đơn hàng. Tổng hợp Order bao gồm nhiều hơn là chỉ các trường. Nó cũng triển khai logic kinh doanh, có thể được mô tả bởi một máy trạng thái. Chúng ta hãy xem xét máy trạng thái.

**TANH TAỒMÁY TRẠNG THÁI TỔNG HỢP RDER**

Để tạo hoặc cập nhật đơn hàng, Dịch vụ đơn hàng phải cộng tác với các dịch vụ kháctệ nạn sử dụng saga. OrderService hoặc bước đầu tiên của saga sẽ gọi phương thức Order để xác minh rằng thao tác có thể được thực hiện và thay đổi trạng thái của Order thành trạng thái đang chờ xử lý. Trạng thái đang chờ xử lý, như đã giải thích trong chương 4, là một ví dụ về

một biện pháp đối phó khóa ngữ nghĩa, giúp đảm bảo rằng các saga được cô lập với nhau. Cuối cùng, sau khi saga đã gọi các dịch vụ tham gia, nó sẽ cập nhậtĐặt hàngđể phản ánh kết quả. Ví dụ, như được mô tả trong chương 4,Tạo đơn hàng Sagacó nhiều dịch vụ dành cho người tham gia, bao gồmDịch vụ khách hàng, Dịch vụ kế toán, VàDịch vụ nhà bếp.Dịch vụ đặt hàngđầu tiên tạo ra mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtrạng thái, và sau đó thay đổi trạng thái của nó thànhTÁN THÀNHhoặcVẬT BỊ LOẠI BỎ. Hành vi của mộtĐặt hàngcó thể được mô hình hóa như máy trạng thái được hiển thị trong hình 5.14.



ĐÃ HỦY

HỦY\_ĐANG\_ĐANG\_XỬ LÝ

**Trạng thái ban đầu**

hủyĐã xác nhận

hủyBị từ chốihủy

được ủy quyền



TÁN THÀNH

ĐANG CHỜ PHÊ DUYỆT

...

vật bị loại bỏ

ôn lại

sửa đổiĐã xác nhậnsửa lạiBị từ chối

VẬT BỊ LOẠI BỎ

ĐANG XEM XÉT

**Hình 5.14 Một phần của mô hình máy trạng thái củaĐặt hàngtổng hợp**



Tương tự như vậy, các hoạt động Dịch vụ Đặt hàng khác như revision() và cancel() thay đổi đầu tiênLệnh sang trạng thái chờ và sử dụng saga để xác minh rằng thao tác có thể được thực hiện. Sau đó, khi saga đã xác minh rằng thao tác có thể được thực hiện, nó sẽ thay đổi các chuyển đổi Lệnh sang một số trạng thái khác phản ánh kết quả thành công của thao tác. Nếu xác minh thao tác không thành công, Lệnh sẽ trở lại trạng thái trước đó. Ví dụ, thao tác cancel() đầu tiên chuyển Lệnh sangHỦY\_ĐANG\_ĐANG\_XỬ LÝtrạng thái. Nếu lệnh có thể bị hủy, Cancel Order Saga sẽ thay đổi trạng thái của Lệnh thành trạng thái CANCELLED. Ngược lại, nếu thao tác cancel() bị từ chối vì, ví dụ, đã quá muộn để hủy lệnh, thì Lệnh sẽ chuyển trở lại trạng thái APPROVED.

Bây giờ chúng ta hãy xem cách tổng hợp Order triển khai máy trạng thái này.

**TANH TAỒĐÁ TỔNG HỢP RDER'S PHƯƠNG PHÁP**

Lớp Order có một số nhóm phương thức, mỗi nhóm tương ứng với một saga. Trong mỗi nhóm, một phương thức được gọi khi bắt đầu saga và các phương thức khác được gọi khi kết thúc. Trước tiên, tôi sẽ thảo luận về logic nghiệp vụ tạo ra một Order. Sauchúng ta sẽ xem cách một Order được cập nhật. Danh sách sau đây hiển thị các phương thức của Order được gọi trong quá trình tạo Order.

**Liệt kê 5.15 Các phương thức được gọi trong quá trình tạo đơn hàng**

lớp công khai Order { ...

public static ResultWithDomainEvents<Order, OrderDomainEvent> createOrder(long consumerId, Nhà hàng nhà hàng,

Danh sách<OrderLineItem> orderLineItems) { Đơn hàng order = new Order(consumerId, restaurant.getId(), orderLineItems); Danh sách<OrderDomainEvent> sự kiện = singletonList(new OrderCreatedEvent(

OrderDetails mới (consumerId, restaurant.getId(), orderLineItems, order.getOrderTotal()),

nhà hàng.getName()));

trả về ResultWithDomainEvents<>(thứ tự, sự kiện);

}

công khai Order(OrderDetails orderDetails) {

này.orderLineItems = new OrderLineItems(orderDetails.getLineItems()); này.orderMinimum = orderDetails.getOrderMinimum();

this.state = ĐANG CHỜ PHÊ DUYỆT;

}

...

công khai List<DomainEvent> noteApproved() { chuyển đổi (trạng thái) {

trường hợp CHỜ PHÊ DUYỆT:

this.state = ĐÃ DUYỆT;

trả về singletonList(new OrderAuthorized());

...

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

công khai List<DomainEvent> noteRejected() { chuyển đổi (trạng thái) {

trường hợp CHỜ PHÊ DUYỆT:

this.state = BỊ TỪ CHỐI;

trả về singletonList(new OrderRejected());

...

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

Cáctạo đơn hàng()phương pháp là một phương pháp nhà máy tĩnh tạo ra một Đơn hàng và xuất bản mộtSự kiện OrderCreated. CácSự kiện OrderCreatedđược làm giàu với các chi tiết củaĐặt hàng, bao gồm các mục dòng, tổng số tiền, ID nhà hàng vàtên nhà hàng. Chương 7 thảo luận về cáchDịch vụ Lịch sử đơn hàngsử dụngĐặt hàngcác sự kiện, bao gồmSự kiện OrderCreated, để duy trì một bản sao dễ dàng truy vấn củaĐơn hàng.

Trạng thái ban đầu củaĐặt hànglàĐANG CHỜ PHÊ DUYỆT. KhiTạo đơn hàngSagahoàn thành, nó sẽ gọilưu ýĐã phê duyệt()hoặcnoteRejected(). Cáclưu ý- Đã phê duyệt()phương pháp được gọikhi thẻ tín dụng của người tiêu dùng đã được ủy quyền thành công.noteRejected()phương pháp được gọi khi một trong các dịch vụ từ chối lệnh hoặc ủy quyền không thành công. Như bạn có thể thấy,tình trạngcủaĐặt hàngtổng hợp xác định hành vi của hầu hết các phương pháp của nó. Giống nhưVétổng hợp, nó cũng phát ra các sự kiện.

Ngoài ratạo đơn hàng(), cácĐặt hànglớp định nghĩa một số phương pháp cập nhật. Ví dụ,Sửa đổi Order Sagasửa đổi một đơn đặt hàngbằng cách đầu tiên triệu hồiôn lại()phương pháp và sau đó, khi đã xác minh được rằng bản sửa đổi có thể được thực hiện, nó sẽ gọixác nhận-Đã sửa đổi()phương pháp. Danh sách sau đây hiển thị các phương pháp này.

**Liệt kê 5.16Đặt hàngphương pháp để sửa đổi mộtĐặt hàng**

Lớp Đặt hàng ...

Danh sách công khai<OrderDomainEvent> sửa lại(OrderRevisionorderRevision) { chuyển đổi (trạng thái) {

trường hợp ĐÃ ĐƯỢC PHÊ DUYỆT: LineItemQuantityChange thay đổi =

orderLineItems.lineItemQuantityChange(orderRevision); nếu (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) {

ném OrderMinimumNotMetException mới();

}

trạng thái này= ĐANG CHỜ XEM XÉT;

trả về singletonList(new OrderRevisionProposed(orderRevision, change.currentOrderTotal, change.newOrderTotal));

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

công khai List<OrderDomainEvent> xác nhậnRevision(OrderRevisionorderRevision) { chuyển đổi (trạng thái) {

trường hợp REVISION\_PENDING:

LineItemQuantityChange licd = orderLineItems.lineItemQuantityChange(orderRevision);

đơn hàngSửa đổi

.getDeliveryInformation()

.ifPresent(newDi -> this.deliveryInformation = newDi);

nếu (!orderRevision.getRevisedLineItemQuantities().isEmpty()) {orderLineItems.updateLineItems(orderRevision);

}

this.state = ĐÃ DUYỆT;

trả về singletonList(new OrderRevised(orderRevision,

licd.currentOrderTotal, licd.newOrderTotal));

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

}

Cácôn lại()phương pháp được gọi để bắt đầu sửa đổi một lệnh. Trong số những thứ khác, nó xác minh rằng lệnh đã sửa đổi sẽ không vi phạm mức tối thiểu của lệnh và thay đổi trạng thái của lệnh thànhĐANG XEM XÉT. Một lầnSửa đổi Order Sagađã cập nhật thành côngDịch vụ nhà bếpVàKế toánDịch vụ, sau đó nó gọixác nhậnSửa đổi()để hoàn tất việc sửa đổi.

Các phương thức này được OrderService gọi. Chúng ta hãy xem xét lớp đó.

###### Lớp OrderService

CácDịch vụ đặt hànglớp định nghĩa các phương thức để tạo và cập nhậtĐơn hàng. Đây là điểm vào chính vào logic kinh doanh và được kích hoạt bởi nhiều bộ điều hợp đầu vào, chẳng hạn nhưGiao diện lập trình ứng dụng REST. Hầu hết các phương pháp của nó tạo ra một câu chuyện để dàn dựng việc tạo ra và cập nhậtĐặt hàngtổng hợp. Kết quả là, dịch vụ nàyphức tạp hơnDịch vụ nhà bếplớp đã thảo luận trước đó. Danh sách sau đây hiển thị một đoạn trích của lớp này.Dịch vụ đặt hàngđược tiêm với nhiều phụ thuộc khác nhau, bao gồmKho lưu trữ đơn hàng,Nhà xuất bản sự kiện miền đặt hàngvà một số trình quản lý saga. Nó định nghĩa một số phương pháp, bao gồmtạo đơn hàng()Vàsửa lại đơn hàng().

**Liệt kê 5.17Dịch vụ đặt hànglớp có các phương thức để tạo và quản lý đơn hàng**

@Giao dịch

lớp công khai OrderService {

@Autowired

OrderRepository riêng tư orderRepository;

@Autowired

SagaManager riêng tư<CreateOrderSagaState, CreateOrderSagaState> createOrderSagaManager;

@Autowired

SagaManager riêng tư<ReviseOrderSagaState, ReviseOrderSagaData> revisionOrderSagaManagement;

@Autowired

riêng tư OrderDomainEventPublisher orderAggregateEventPublisher; công khai Order createOrder(OrderDetails orderDetails) {

Nhà hàng nhà hàng = restaurantRepository.findById(restaurantId)

.hoặcElseThrow(() -

> mới RestaurantNotFoundException(restaurantId));

Danh sách<OrderLineItem> orderLineItems = makeOrderLineItems(lineItems, nhà hàng);

**Tạo tổng hợp đơn hàng**

ResultWithDomainEvents<Order, OrderDomainEvent> orderAndEvents = Order.createOrder(consumerId, restaurant, orderLineItems);

Thứ tự thứ tự = orderAndEvents.result;orderRepository.save(thứ tự); 

**Duy trì thứ tự trong cơ sở dữ liệu**

**Xuất bảnsự kiện miền**

);

orderAggregateEventPublisher.publish(thứ tự,orderAndEvents.sự kiện

Chi tiết đơn hàngChi tiết đơn hàng =

mới OrderDetails(consumerId, restaurantId, orderLineItems, order.getOrderTotal());

Dữ liệu CreateOrderSagaState = new CreateOrderSagaState(order.getId(), orderDetails);

createOrderSagaManager.create(dữ liệu,Lớp lệnh, lệnh.getId());

trả lại đơn hàng;

}

**Tạo ra Tạo**

**Đặt hàngTruyện dài**

công khai Order revisionOrder(Long orderId, Long expectedVersion,

OrderRevision orderRevision) {

công khai Order revisionOrder(long orderId, OrderRevision orderRevision) { Order order = orderRepository.findById(orderId)

.orElseThrow(() -> new OrderNotFoundException(orderId)); ReviseOrderSagaData sagaData =

mới ReviseOrderSagaData(order.getConsumerId(), orderId, null, orderRevision);

**Lấy lại đơn hàng**

revisionOrderSagaManager.create(sagaData);trả lại đơn hàng;

}

}

**Tạo ra Revise Order Saga**

Cáctạo đơn hàng()phương pháp đầu tiên tạo ra và duy trì mộtĐặt hàngtổng hợp. Sau đó, nó xuất bản các sự kiện miền được phát ra bởi tổng hợp. Cuối cùng, nó tạo ra mộtCreateOrder-Saga. Cácsửa lại đơn hàng()lấy lạiĐặt hàngvà sau đó tạo ra mộtSửa đổi OrderSaga.

Theo nhiều cách, logic nghiệp vụ cho ứng dụng dựa trên dịch vụ vi mô không khác nhiều so với ứng dụng đơn khối. Nó bao gồm các lớp như dịch vụ, thực thể được JPA hỗ trợ và kho lưu trữ. Tuy nhiên, vẫn có một số điểm khác biệt. Mô hình miền được tổ chức dưới dạng một tập hợp các tập hợp DDD áp đặt nhiều ràng buộc thiết kế khác nhau. Không giống như trong mô hình đối tượng truyền thống, các tham chiếu giữa các lớp trong các tập hợp khác nhau được tính theo giá trị khóa chính chứ không phải tham chiếu đối tượng. Ngoài ra, một giao dịch chỉ có thể tạo hoặc cập nhật một tập hợp duy nhất. Việc các tập hợp công bố các sự kiện miền khi trạng thái của chúng thay đổi cũng rất hữu ích.

Một sự khác biệt lớn nữa là các dịch vụ thường sử dụng sagas để duy trì tính nhất quán của dữ liệu.sự bền vững trên nhiều dịch vụ. Ví dụ, Dịch vụ nhà bếp chỉ tham gia vào saga, không khởi tạo chúng. Ngược lại, Dịch vụ đặt hàng dựa nhiều vào saga khi

tạo và cập nhật đơn hàng. Đó là bởi vì Đơn hàng phải được thống nhất về mặt giao dịchlều với dữ liệu thuộc sở hữu của các dịch vụ khác. Do đó, hầu hết các phương thức OrderService tạo ra một saga thay vì cập nhật trực tiếp một Order.

Chương này đã đề cập đến cách triển khai logic kinh doanh bằng cách sử dụng phương pháp truyền thống để duy trì. Điều đó liên quan đến việc tích hợp nhắn tin và xuất bản sự kiện với quản lý giao dịch cơ sở dữ liệu. Mã xuất bản sự kiện được đan xen với logic kinh doanh. Chương tiếp theo sẽ xem xét về nguồn sự kiện, một phương pháp tiếp cận lấy sự kiện làm trung tâm để viết logic kinh doanh, trong đó việc tạo sự kiện là một phần không thể thiếu của logic kinh doanh thay vì được gắn vào.

#### Bản tóm tắt

* Mẫu kịch bản giao dịch thủ tục thường là một cách tốt để triển khai logic kinh doanh đơn giản. Nhưng khi triển khai logic kinh doanh phức tạp, bạn nên cân nhắc sử dụng mẫu mô hình miền hướng đối tượng.
* Một cách tốt để tổ chức logic kinh doanh của dịch vụ là tập hợp các DDD aggregates. DDD aggregate hữu ích vì chúng mô-đun hóa mô hình miền, loại bỏ khả năng tham chiếu đối tượng giữa các dịch vụ và đảm bảo rằng mỗi giao dịch ACID nằm trong một dịch vụ.
* Một tổng hợp phải công bố các sự kiện miền khi được tạo hoặc cập nhật. Các sự kiện miền có nhiều cách sử dụng khác nhau. Chương 4 thảo luận về cách chúng có thể triển khai các saga dựa trên vũ đạo. Và, trong chương 7, tôi sẽ nói về cách sử dụng các sự kiện miền để cập nhật dữ liệu được sao chép. Người đăng ký sự kiện miền cũng có thể thông báo cho người dùng và các ứng dụng khác, và công bố các thông báo WebSocket tới trình duyệt của người dùng.

*Phát triển logic kinh doanh với nguồn sự kiện*

***Chương này bao gồm***

* Sử dụng mô hình Nguồn sự kiện để phát triểnlogic kinh doanh
* Triển khai một cửa hàng sự kiện
* Tích hợp saga và sự kiệnlogic kinh doanh dựa trên nguồn cung ứng
* Triển khai các trình điều phối saga bằng cách sử dụng sự kiệnnguồn cung ứng

Mary thích ý tưởng được mô tả trong chương 5 về việc cấu trúc logic kinh doanh như một tập hợp các tổng hợp DDD xuất bản các sự kiện miền. Cô ấy có thể hình dung việc sử dụng các sự kiện đó cực kỳ hữu ích trong kiến ​​trúc dịch vụ vi mô. Mary đã lên kế hoạch sử dụng các sự kiện để triển khai các saga dựa trên biên đạo, duy trì tính nhất quán của dữ liệu trên các dịch vụ và được mô tả trong chương 4. Cô ấy cũng mong đợi sử dụng các chế độ xem CQRS, các bản sao hỗ trợ truy vấn hiệu quả được mô tả trong chương 7.

Tuy nhiên, cô ấy lo lắng rằng logic xuất bản sự kiện có thể dễ bị lỗi. Một mặt, logic xuất bản sự kiện khá đơn giản. Mỗi phương thức của tổng hợp khởi tạo hoặc thay đổi trạng thái của tổng hợp trả về một danh sách các sự kiện. Sau đó, dịch vụ miền sẽ xuất bản các sự kiện đó. Nhưng mặt khác

**183**

Mặt khác, logic xuất bản sự kiện được gắn chặt vào logic nghiệp vụ. Logic nghiệp vụ vẫn tiếp tục hoạt động ngay cả khi nhà phát triển quên xuất bản sự kiện. Mary lo ngại rằng cách xuất bản sự kiện này có thể là nguồn gốc của lỗi.

Nhiều năm trước, Mary đã tìm hiểu về event sourcing, một cách viết logic kinh doanh và duy trì các đối tượng miền theo sự kiện. Vào thời điểm đó, cô ấy bị hấp dẫn bởi nhiều lợi ích của nó, bao gồm cách nó lưu giữ toàn bộ lịch sử các thay đổi đối với một tổng hợp, nhưng nó vẫn là một sự tò mò. Với tầm quan trọng của các sự kiện miền trong kiến ​​trúc vi dịch vụ, giờ đây cô ấy tự hỏi liệu có đáng để khám phá việc sử dụng event sourcing trong ứng dụng FTGO hay không. Rốt cuộc, event sourcing loại bỏ một nguồn lỗi lập trình bằng cách đảm bảo rằng một sự kiện sẽ được công bố bất cứ khi nào một tổng hợp được tạo hoặc cập nhật.

Tôi bắt đầu chương này bằng cách mô tả cách thức hoạt động của event sourcing và cách bạn có thể sử dụng nó để viết logic kinh doanh. Tôi mô tả cách event sourcing lưu trữ từng tổng hợp như một chuỗi sự kiện trong cái được gọi là event store. Tôi thảo luận về những lợi ích và hạn chế của event sourcing và đề cập đến cách triển khai event store. Tôi mô tả một khuôn khổ đơn giản để viết logic kinh doanh dựa trên event sourcing. Sau đó, tôi thảo luận về cách event sourcing là nền tảng tốt để triển khai sagas. Chúng ta hãy bắt đầu bằng cách xem xét cách phát triển logic kinh doanh với event sourcing.

#### Phát triển logic kinh doanh bằng cách sử dụng nguồn sự kiện

Event sourcing là một cách khác để cấu trúc logic kinh doanh và lưu trữ aggre-gates. Nó lưu trữ một tổng hợp dưới dạng một chuỗi các sự kiện. Mỗi sự kiện biểu diễn một thay đổi trạng thái của tổng hợp. Một ứng dụng tạo lại trạng thái hiện tại của một tổng hợp bằng cách phát lại các sự kiện.

**Mẫu: Nguồn sự kiện**

Duy trì tổng hợp dưới dạng một chuỗi các sự kiện miền biểu diễn các thay đổi trạng thái. Xem<http://microservices.io/patterns/data/event-sourcing.html>.

Event sourcing có một số lợi ích quan trọng. Ví dụ, nó bảo toàn lịch sử của các tổng hợp, rất có giá trị cho mục đích kiểm toán và quản lý. Và nó xuất bản các sự kiện miền một cách đáng tin cậy, đặc biệt hữu ích trong kiến ​​trúc vi dịch vụ. Event sourcing cũng có những nhược điểm. Nó liên quan đến đường cong học tập, vì đây là một cách khác để viết logic kinh doanh của bạn. Ngoài ra, việc truy vấn kho sự kiện thường khó khăn, đòi hỏi bạn phải sử dụng mẫu CQRS, được mô tả trong chương 7.

Tôi bắt đầu phần này bằng cách mô tả những hạn chế của sự bền bỉ truyền thống. Sau đó, tôi mô tả chi tiết về event sourcing và nói về cách nó khắc phục những hạn chế đó. Sau đó, tôi chỉ cách triển khai tổng hợp Order bằng event sourcing. Cuối cùng, tôi mô tả những lợi ích và hạn chế của event sourcing.

Trước tiên chúng ta hãy xem xét những hạn chế của phương pháp tiếp cận truyền thống đối với tính bền bỉ.

###### Rắc rối với sự kiên trì truyền thống

Cách tiếp cận truyền thốngđể duy trì ánh xạ các lớp vào các bảng cơ sở dữ liệu, các trường của các lớp đó vào các cột bảng và các thể hiện của các lớp đó vào các hàng trong các bảng đó. Ví dụ, hình 6.1 cho thấy cách tổng hợp Order, được mô tả trong chương 5, được ánh xạ vào bảng ORDER. Các OrderLineItem của nó được ánh xạ vào bảng ORDER\_LINE\_ITEM.

ĐẶT HÀNGbàn

«lớp» Đặt hàng

|  |  |  |  |
| --- | --- | --- | --- |
| NHẬN DẠNG | KHÁCH HÀNG\_ID | TỔNG CỘNG ĐẶT HÀNG | ... |
|  |  |  |  |
| 1234 | khách hàng-abc | 1234,56 | ... |

MẶT HÀNG\_DÒNG\_ĐẶT HÀNGbàn

«lớp» OrderLineItem

|  |  |  |  |
| --- | --- | --- | --- |
| NHẬN DẠNG | ĐƠN\_HÀNH | SỐ LƯỢNG | ... |
|  |  |  |  |
| 567 | 1234 | 2 | ... |

**Hình 6.1 Phương pháp tiếp cận truyền thống để duy trì ánh xạ các lớp thành các bảng và các đối tượng thành các hàng trongnhững cái bàn đó.**

Ứng dụng duy trì một thể hiện thứ tự dưới dạng các hàng trong các bảng ORDER và ORDER\_LINE\_ITEM. Nó có thể thực hiện điều đó bằng cách sử dụng một khuôn khổ ORM như JPA hoặc một khuôn khổ cấp thấp hơn như MyBATIS.

Cách tiếp cận này rõ ràng là hiệu quả vì hầu hết các ứng dụng doanh nghiệp đều lưu trữ dữ liệu theo cách này. Nhưng nó có một số nhược điểm và hạn chế:

* Sự không phù hợp trở kháng quan hệ đối tượng.
* Thiếu lịch sử tổng hợp.
* Việc triển khai ghi nhật ký kiểm tra rất tẻ nhạt và dễ xảy ra lỗi.
* Việc xuất bản sự kiện được gắn chặt vào logic kinh doanh.

Chúng ta hãy xem xét từng vấn đề này, bắt đầu với vấn đề không khớp trở kháng quan hệ đối tượng.

**ỒĐỐI TƯỢNG-RSỰ KHÔNG PHÙ HỢP TRỞ KHÁNG ĐIỆN**

Một vấn đề lâu đời là cái gọi là vấn đề không khớp trở kháng Đối tượng-Quan hệ. Có một sự không khớp khái niệm cơ bản giữa lược đồ quan hệ dạng bảng và cấu trúc đồ thị của một mô hình miền phong phú với các mối quan hệ phức tạp của nó. Một số khía cạnh của vấn đề này được phản ánh trong các cuộc tranh luận phân cực về tính phù hợp của các khuôn khổ ánh xạ Đối tượng/Quan hệ (ORM). Ví dụ, Ted Neward đã nói rằng "Ánh xạ Đối tượng-Quan hệ là Việt Nam của Khoa học Máy tính" ([http://blogs](http://blogs.tedneward.com/post/the-vietnam-of-computer-science/)

[.tedneward.com/post/the-vietnam-of-computer-science/](http://blogs.tedneward.com/post/the-vietnam-of-computer-science/)). Công bằng mà nói, tôi đã sử dụng

Hibernate thành công để phát triển các ứng dụng mà lược đồ cơ sở dữ liệu được lấy từ mô hình đối tượng. Nhưng vấn đề sâu xa hơn những hạn chế của bất kỳ khuôn khổ ORM cụ thể nào.

**LACK CỦA LỊCH SỬ TỔNG HỢP**

Một hạn chế khác của tính bền bỉ truyền thống là nó chỉ lưu trữ trạng thái hiện tại của một tổng hợp. Khi một tổng hợp đã được cập nhật, trạng thái trước đó của nó sẽ bị mất. Nếu một ứng dụng phải lưu giữ lịch sử của một tổng hợp, có thể là vì mục đích quản lý, thì các nhà phát triển phải tự triển khai cơ chế này. Việc triển khai cơ chế lịch sử tổng hợp tốn nhiều thời gian và liên quan đến việc sao chép mã phải được đồng bộ hóa với logic kinh doanh.

**TÔIVIỆC THỰC HIỆN VIỆC GHI CHÉP KIỂM TOÁN RẤT TỒI TỆ VÀ DỄ GẶP LỖI**

Một vấn đề khác là nhật ký kiểm toán. Nhiều ứng dụng phải duy trì nhật ký kiểm toán để theo dõi những người dùng nào đã thay đổi tổng hợp. Một số ứng dụng yêu cầu kiểm toán vì mục đích bảo mật hoặc quản lý. Trong các ứng dụng khác, lịch sử hành động của người dùng là một tính năng quan trọng. Ví dụ, trình theo dõi sự cố và ứng dụng quản lý tác vụ như Asana và JIRA hiển thị lịch sử thay đổi đối với các tác vụ và sự cố. Thách thức của việc triển khai kiểm toán là ngoài việc tốn thời gian, mã nhật ký kiểm toán và logic kinh doanh có thể khác nhau, dẫn đến lỗi.

**EVENT PUBLISHING ĐƯỢC GẮN VÀO LOGIC KINH DOANH**

Một hạn chế khác của tính bền bỉ truyền thống là nó thường không hỗ trợ việc xuất bản các sự kiện miền. Các sự kiện miền, được thảo luận trong chương 5, là các sự kiện được xuất bản bởimột tổng hợp khi trạng thái của nó thay đổi. Chúng là một cơ chế hữu ích để đồng bộ hóa dữ liệu và gửi thông báo trong kiến ​​trúc vi dịch vụ. Một số khuôn khổ ORM, chẳng hạn như Hibernate, có thể gọi các lệnh gọi lại do ứng dụng cung cấp khi các đối tượng dữ liệu thay đổi. Nhưng không có hỗ trợ nào cho việc tự động xuất bản các thông báo như một phần của giao dịch cập nhật dữ liệu. Do đó, giống như lịch sử và kiểm toán, các nhà phát triển phải bổ sung logic tạo sự kiện, có nguy cơ không được đồng bộ hóa với logic nghiệp vụ. May mắn thay, có một giải pháp cho những vấn đề này: nguồn sự kiện.

###### Tổng quan về nguồn sự kiện

Nguồn sự kiện là một kỹ thuật lấy sự kiện làm trung tâmđể triển khai logic kinh doanh và duy trì tổng hợp. Tổng hợp được lưu trữ trong cơ sở dữ liệu dưới dạng một loạt các sự kiện. Mỗi sự kiện biểu thị một thay đổi trạng thái của tổng hợp. Logic kinh doanh của tổng hợp được cấu trúc xung quanh yêu cầu tạo và sử dụng các sự kiện này. Chúng ta hãy xem cách thức hoạt động.

**EVENT SOURCING DUY TRÌ TỔNG HỢP SỬ DỤNG SỰ KIỆN**

Trước đó, trong phần 6.1.1, tôi đã thảo luận về cách thức persistence truyền thống ánh xạ các tổng hợp thành các bảng, các trường của chúng thành các cột và các thể hiện của chúng thành các hàng. Event sourcing là một cách tiếp cận rất khác để persistence các tổng hợp dựa trên khái niệm về các sự kiện miền. Nó lưu trữ từng tổng hợp dưới dạng một chuỗi các sự kiện trong cơ sở dữ liệu, được gọi là kho lưu trữ sự kiện.

Hãy xem xét, ví dụ,Đặt hàngtổng hợp. Như hình 6.2 cho thấy, thay vì lưu trữmỗiĐặt hàngnhư một hàng trong mộtĐẶT HÀNGbảng, sự kiện nguồn vẫn tồn tại mỗiĐặt hàngtổng hợp thành một hoặc nhiều hàng trong mộtSỰ KIỆNbảng. Mỗi hàng là một sự kiện miền, chẳng hạn nhưĐã tạo đơn hàng,Đơn hàng đã được chấp thuận,Đơn hàng đã được vận chuyển, và vân vân.

**Sự kiện độc đáoIDCác loại củasự kiệnXác định cáitổng hợp sự kiện nối tiếp,**

**chẳng hạn như JSON**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sự kiện\_id | loại sự kiện | loại thực thể | thực thể\_id | dữ liệu sự kiện |
| 102 | Đã tạo đơn hàng | Đặt hàng | 101 | {...} |
| 103 | Đặt hàng  Tán thành | Đặt hàng | 101 | {...} |
| 104 | Đặt hàng  Đã vận chuyển | Đặt hàng | 101 | {...} |
| 105 | Đặt hàng  Đã giao hàng | Đặt hàng | 101 | {...} |
| ... | ... | ... | ... | ... |

Bảng SỰ KIỆN

**Hình 6.2 Nguồn sự kiện duy trì mỗi tổng hợp như một chuỗi sự kiện. Một ứng dụng dựa trên RDBMS có thể, ví dụ, lưu trữ các sự kiện trong mộtSỰ KIỆNbàn.**

Khi một ứng dụng tạo hoặc cập nhật một tổng hợp, nó sẽ chèn các sự kiện do tổng hợp phát ra vào bảng EVENTS. Một ứng dụng tải một tổng hợp từ kho sự kiện bằng cách truy xuất các sự kiện của nó và phát lại chúng. Cụ thể, việc tải một tổng hợp bao gồm ba bước sau:

**1** Tải các sự kiện cho tổng hợp.

**2** Tạo một thể hiện tổng hợp bằng cách sử dụng hàm tạo mặc định của nó.

**3** Lặp lại các sự kiện, gọiáp dụng().

Ví dụ, khuôn khổ Eventuate Client, được đề cập sau trong phần 6.2.2, sử dụng mãtương tự như sau để tái tạo một tổng hợp:

Lớp aggregateClass = ...;

Tổng hợp tổng hợp = aggregateClass.newInstance(); cho (Sự kiện sự kiện: sự kiện) {

aggregate = aggregate.applyEvent(sự kiện);

}

// sử dụng tổng hợp...

Nó tạo ra một thể hiện của lớp và lặp lại các sự kiện, gọi aggre-Phương thức applyEvent() của gate. Nếu bạn quen thuộc với lập trình hàm, bạn có thể nhận ra đây là thao tác fold hoặc reduce.

Có thể lạ và không quen khi tái tạo trạng thái trong bộ nhớ của một aggregate bằng cách tải các sự kiện và phát lại các sự kiện. Nhưng theo một số cách, nó không khác nhiều so với cách một khung ORM như JPA hoặc Hibernate tải một thực thể. Một khung ORM tải một đối tượng bằng cách thực thi một hoặc nhiều câu lệnh SELECT để truy xuất trạng thái hiện tại đã lưu, khởi tạo các đối tượng bằng các hàm tạo mặc định của chúng. Nó sử dụng phản xạ để khởi tạo các đối tượng đó. Điểm khác biệt về nguồn sự kiện là việc tái tạo trạng thái trong bộ nhớ được thực hiện bằng các sự kiện.

Bây giờ chúng ta hãy xem xét các yêu cầu về sự kiện nguồn trên các sự kiện miền.

**EVENTS ĐẠI DIỆNTHAY ĐỔI TRẠNG THÁI**

Chương 5 định nghĩa sự kiện miền là cơ chế thông báo cho người đăng ký về những thay đổi đối với tổng hợp. Sự kiện có thể chứa dữ liệu tối thiểu, chẳng hạn như chỉ ID tổng hợp hoặc có thể được làm giàu để chứa dữ liệu hữu ích cho người tiêu dùng thông thường. Ví dụ:Order Service có thể xuất bản sự kiện OrderCreated khi một đơn hàng được tạo. Sự kiện OrderCreated chỉ có thể chứa orderId. Ngoài ra, sự kiện có thể chứa toàn bộ đơn hàng để người dùng của sự kiện đó không phải lấy dữ liệu từ Order Service. Việc các sự kiện có được xuất bản hay không và những sự kiện đó chứa gì được thúc đẩy bởi nhu cầu của người dùng. Tuy nhiên, với nguồn sự kiện, chủ yếu là tổng hợp xác định các sự kiện và cấu trúc của chúng.

Sự kiện không phải là tùy chọn khi sử dụng nguồn sự kiện. Mỗi thay đổi trạng thái của một tổng hợp, bao gồm cả việc tạo ra nó, được biểu diễn bằng một sự kiện miền. Bất cứ khi nào trạng thái của tổng hợp thay đổi, nó phải phát ra một sự kiện. Ví dụ, một tổng hợp Order phải phát ra mộtSự kiện OrderCreated khi nó được tạo và sự kiện Order\* khi nó được cập nhật. Đây là yêu cầu nghiêm ngặt hơn nhiều so với trước đây, khi tổng hợp chỉ phát ra các sự kiện mà người tiêu dùng quan tâm.

Hơn nữa, một sự kiện phải chứa dữ liệu mà tổng hợp cần để thực hiện chuyển đổi trạng thái. Trạng thái của tổng hợp bao gồm các giá trị của các trường của các đối tượng tạo nên tổng hợp. Một thay đổi trạng thái có thể đơn giản như thay đổi giá trị của trường của một đối tượng, chẳng hạn như Order.state. Ngoài ra, một thay đổi trạng thái có thể bao gồm việc thêm hoặc xóa các đối tượng, chẳng hạn như sửa đổi các mục dòng của Đơn hàng.

Giả sử, như hình 6.3 cho thấy, trạng thái hiện tại của tổng hợp là S và trạng thái mới là S'. Một sự kiện E biểu diễn sự thay đổi trạng thái phải chứa dữ liệu như vậyrằng khi một Order ở trạng thái S, việc gọi order.apply(E) sẽ cập nhật Order thành trạng thái S'. Trong phần tiếp theo, bạn sẽ thấy apply() là một phương thức thực hiện thay đổi trạng thái được biểu thị bằng một sự kiện.

Một số sự kiện, chẳng hạn như sự kiện Đơn hàng đã giao,chứa ít hoặc không có dữ liệu và chỉđại diện cho quá trình chuyển đổi trạng thái. Phương thức apply() xử lý sự kiện Order Shipped bằng cách thay đổi trường trạng thái của Order thành SHIPPED. Tuy nhiên, các sự kiện khác lại chứa rất nhiều dữ liệu. Ví dụ, sự kiện OrderCreated phải chứa tất cả dữ liệu cần thiết theo phương thức apply() để khởi tạo Order, bao gồm các mục hàng, thông tin thanh toán, thông tin giao hàng, v.v. Vì các sự kiện được sử dụng để duy trì tổng hợp, nên bạn không còn tùy chọn sử dụng sự kiện OrderCreated tối thiểu có chứa orderId.

Sự kiện

«tổng hợp» Đặt hàng

S

«tổng hợp» Đặt hàng

S'

áp dụng()



**Đối tượng và giá trị trường**

**Cập nhật các đối tượng và giá trị trường**

**Hình 6.3 Áp dụng sự kiệnEkhiĐặt hàngđang ở trạng tháiSphải thay đổiĐặt hàngtrạng thái đểS'. Sự kiện phải chứa dữ liệu cần thiết để thực hiệnthay đổi trạng thái.**

**MỘTPHƯƠNG PHÁP TỔNG HỢP LÀ TẤT CẢ VỀ CÁC SỰ KIỆN**

Logic nghiệp vụ xử lý yêu cầu cập nhật tổng hợp bằng cách gọi phương thức lệnh trên gốc tổng hợp. Trong ứng dụng truyền thống, phương thức lệnh thường xác thực các đối số của nó và sau đó cập nhật một hoặc nhiều trường của tổng hợp. Phương thức lệnh trong ứng dụng dựa trên nguồn sự kiện hoạt động vì chúng phải tạo ra các sự kiện. Như hình 6.4 cho thấy, kết quả của việc gọi phương thức lệnh của tổng hợp là một chuỗi các sự kiện biểu diễn các thay đổi trạng thái cần thực hiện. Các sự kiện này được lưu trong cơ sở dữ liệu và áp dụng cho tổng hợp để cập nhật trạng thái của nó.

Sự kiện

«tổng hợp» Đặt hàng

S

Tiến trình(lệnh)



Sự kiện

«tổng hợp» Đặt hàng

S

«tổng hợp» Đặt hàng

S'

áp dụng()



**Hình 6.4 Xử lýlệnh tạo ra sự kiện mà không thay đổi trạng thái của tổng hợp. Tổng hợp được cập nhật bằng cách áp dụng sự kiện.**

Yêu cầu tạo sự kiện và áp dụng chúng đòi hỏi phải tái cấu trúc—mặc dù là cơ học—của logic kinh doanh. Event sourcing tái cấu trúc một phương thức lệnh thành hai hoặc nhiều phương thức. Phương thức đầu tiên lấy tham số đối tượng lệnh, biểu diễn yêu cầu và xác định những thay đổi trạng thái nào cần được thực hiện. Nó xác thực các đối số của mình và không thay đổi trạng thái của tổng hợp, trả về danh sách các sự kiện biểu diễn các thay đổi trạng thái. Phương thức này thường ném một ngoại lệ nếu lệnh không thể được thực hiện.

Các phương pháp khác đều lấy một loại sự kiện cụ thể làm tham số và cập nhật tổng hợp. Có một trong những phương pháp này cho mỗi sự kiện. Điều quan trọng cần lưu ý là các phương pháp này không thể lỗi, vì một sự kiện biểu thị một thay đổi trạng thái đã xảy ra. Mỗi phương pháp cập nhật tổng hợp dựa trên sự kiện.

Khung Eventuate Client, một khung nguồn sự kiện được mô tả trongchi tiết trong phần 6.2.2, đặt tên cho các phương thức này là process() và apply(). Một phương thức process() lấy một đối tượng lệnh, chứa các đối số của yêu cầu cập nhật, làm tham số và trả về danh sách các sự kiện. Một phương thức apply() lấy một sự kiện làm tham số và trả về void. Một tập hợp sẽ định nghĩa nhiều phiên bản quá tải của các phương thức này: một phương thức process() cho mỗi lớp lệnh và một phương thức apply() cho mỗi loại sự kiện do tập hợp phát ra. Hình 6.5 hiển thị một ví dụ.

lớp công khai Order {

công khai List<DomainEvent> revision(OrderRevision orderRevision) { chuyển đổi (trạng thái) {

this.state = ĐANG CHỜ XEM XÉT;trở lại …;

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

trường hợp được ủy quyền:

LineItemQuantityChange thay đổi = orderLineItems.lineItemQuantityChange(orderRevision);

nếu (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) { ném new OrderMinimumNotMetException();

}

trả về singletonList(

OrderRevisionProposed( mới)

orderRevision, change.currentOrderTotal, change.newOrderTotal));

lớp công khai Order {

public void apply(OrderRevisionProposed event) {this.state = ĐANG CHỜ XEM XÉT;

}

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

lớp công khai Order {

public List<Event> process(ReviseOrder command) { OrderRevision orderRevision = command.getOrderRevision(); switch (state) {

trường hợp được ủy quyền:

LineItemQuantityChange thay đổi = orderLineItems.lineItemQuantityChange(orderRevision);

nếu (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) { ném new OrderMinimumNotMetException();

}



**Trả về các sự kiện mà không cập nhậtĐơn hàngÁp dụng sự kiện để cập nhật Đơn hàng**

**Hình 6.5 Nguồn sự kiện chia tách một phương pháp cập nhật tổng hợp thành mộtquá trình()phương pháp, lấy một lệnh và trả về các sự kiện, và một hoặc nhiềuáp dụng()phương pháp, lấy một sự kiện và cập nhậttổng hợp.**

Trong ví dụ này,sửa lại đơn hàng()phương pháp được thay thế bằng mộtquá trình()phương pháp và mộtáp dụng()phương pháp. Cácquá trình()phương pháp mất mộtSửa đổi đơn hànglệnh như một tham số. Lớp lệnh này được định nghĩa bằng cách áp dụng tái cấu trúc Đối tượng tham số Giới thiệu (<https://refactoring.com/catalog/introduceParameterObject.html>)đếnsửa đổi- Order()phương pháp. Cácquá trình()phương pháp trả về mộtOrderRevisionĐề xuấtsự kiện, hoặc ném một ngoại lệ nếu quá muộn để sửa đổiĐặt hànghoặc nếu bản sửa đổi được đề xuấtsion không đáp ứng được đơn hàng tối thiểu.áp dụng()phương pháp choOrderRevision- Đề xuấtsự kiện thay đổi trạng thái củaĐặt hàngĐẾNĐANG XEM XÉT.

Tổng hợp được tạo ra bằng các bước sau:

**1** Khởi tạo gốc tổng hợp bằng cách sử dụng hàm tạo mặc định của nó.

**2** Gọiquá trình()để tạo ra các sự kiện mới.

**3** Cập nhật tổng hợp bằng cách lặp qua các sự kiện mới, gọi phương thức apply() của nó.

**4** Lưu các sự kiện mới vào cửa hàng sự kiện.

Một tổng hợp được cập nhật bằng cách sử dụng sau đâycác bước:

**1** Tải các sự kiện tổng hợp từ kho sự kiện.

**2** Khởi tạo gốc tổng hợp bằng cách sử dụng hàm tạo mặc định của nó.

**3** Lặp lại các sự kiện đã tải, gọi apply() trên gốc tổng hợp.

**4** Gọi nóquá trình()phương pháp tạo sự kiện mới.

**5** Cập nhật tổng hợp bằng cách lặp qua các sự kiện mới, gọi apply().

**6** Lưu các sự kiện mới vào cửa hàng sự kiện.

Để xem điều này trong thực tế, bây giờ chúng ta hãy xem nguồn sự kiệnphiên bản tổng hợp của Đơn hàng.

**ETÌM NGUỒN THÔNG GIÓ-DỰA TRÊNỒĐÁ TỔNG HỢP RDER**

Liệt kê 6.1 cho thấy các trường của tổng hợp Order và các phương thức chịu trách nhiệm tạo ra tổng hợp này. Phiên bản nguồn sự kiện của tổng hợp Order có một số điểm tương đồng với phiên bản dựa trên JPA được hiển thị trong chương 5. Các trường của nó gần như giống hệt nhau và nó phát ra các sự kiện tương tự. Điểm khác biệt là logic kinh doanh của nó được triển khai theo các lệnh xử lý phát ra các sự kiện và áp dụng các sự kiện đó, cập nhật trạng thái của nó. Mỗi phương thức tạo hoặc cập nhật tổng hợp dựa trên JPA, chẳng hạn như createOrder()và revisionOrder() được thay thế trong phiên bản nguồn sự kiện bằng các phương thức process() và apply().

**Liệt kê 6.1Đặt hàngcác trường tổng hợp và các phương thức của nó khởi tạo một thể hiện**

lớp công khai Order {

trạng thái OrderState riêng tư; trạng thái ConsumerId dài riêng tư; trạng thái RestaurantId dài riêng tư;

riêng tư OrderLineItems orderLineItems;

riêng tư Thông tin giao hàng thông tin giao hàng; riêng tư Thông tin thanh toán thông tin thanh toán thông tin; riêng tư Lệnh chuyển tiền Tối thiểu;

công khai Order() {

}

**Xác thực lệnh và trả về OrderCreatedEvent**

public List<Event> process(lệnh CreateOrderCommand){

... xác thực lệnh ...

trở lạisự kiện (OrderCreatedEvent mới (command.getOrderDetails()));

}

public void apply(OrderCreatedEvent event) { OrderDetails orderDetails = event.getOrderDetails();

này.orderLineItems = new OrderLineItems(orderDetails.getLineItems()); này.orderMinimum = orderDetails.getOrderMinimum();

this.state = ĐANG CHỜ PHÊ DUYỆT;

} **Áp dụng OrderCreatedEvent bằng**

**khởi tạo các trường của Order.**

Các trường của lớp này tương tự như các trường của lớp dựa trên JPAĐặt hàng. Sự khác biệt duy nhất là tổng hợpnhận dạngkhông được lưu trữ trong tổng hợp.Đặt hàngphương pháp của 's khákhác nhau.tạo đơn hàng()phương pháp nhà máy đã được thay thế bằngquá trình()Vàáp dụng()phương pháp. Cácquá trình()phương pháp mất mộtTạo đơn hànglệnh và phát ra mộtĐơn hàngĐã tạosự kiện.áp dụng()phương pháp lấyĐơn hàngĐã tạovà khởi tạocác lĩnh vực củaĐặt hàng.

Bây giờ chúng ta sẽ xem xét logic kinh doanh phức tạp hơn một chút để sửa đổi đơn hàng. Trước đây, logic kinh doanh này bao gồm ba phương pháp:sửa lại đơn hàng(),xác nhận-Ôn tập(), Vàtừ chốiSửa đổi(). Phiên bản nguồn sự kiện thay thế ba phương pháp này bằng baquá trình()phương pháp và một sốáp dụng()phương pháp. Danh sách sau đây hiển thị phiên bản nguồn sự kiện củasửa lại đơn hàng()Vàxác nhậnSửa đổi().

**Liệt kê 6.2quá trình()Vàáp dụng()phương pháp sửa đổi mộtĐặt hàngtổng hợp**

lớp công khai Order {

public List<Event> process(ReviseOrder command) { OrderRevision orderRevision = command.getOrderRevision(); switch (state) {

trường hợp ĐÃ ĐƯỢC PHÊ DUYỆT: LineItemQuantityChange thay đổi =

**Xác minh rằng Đơn hàng có thể được sửa đổi và đơn hàng đã sửa đổi đáp ứng được yêu cầu tối thiểu của đơn hàng.**

orderLineItems.lineItemQuantityChange(orderRevision); nếu (change.newOrderTotal.isGreaterThanOrEqual(orderMinimum)) {

ném OrderMinimumNotMetException mới();

}

trả về singletonList(new OrderRevisionProposed(orderRevision,

thay đổi.currentOrderTotal,thay đổi.newOrderTotal));

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

**Thay đổi trạng thái của Order**

public void apply(OrderRevisionProposed event) { this.state = REVISION\_PENDING;

**đến REVISION\_PENDING.**

}

public List<Event> process(lệnh ConfirmReviseOrder) { OrderRevision orderRevision = lệnh.getOrderRevision();

chuyển đổi (trạng thái) {

trường hợp REVISION\_PENDING:

LineItemQuantityChange licd = orderLineItems.lineItemQuantityChange(orderRevision);

trả về singletonList(new OrderRevised(orderRevision, licd.currentOrderTotal, licd.newOrderTotal));

**Xác minh rằng bản sửa đổi có thể được xác nhận và trả về sự kiện Đơn hàng đã sửa đổi.**

mặc định:

ném UnsupportedStateTransitionException mới (trạng thái);

}

}

**Sửa đổi**

public void apply(sự kiện OrderRevised) {

OrderRevision orderRevision = event.getOrderRevision();

**Đặt hàng.**

nếu (!orderRevision.getRevisedLineItemQuantities().isEmpty()) { orderLineItems.updateLineItems(orderRevision);

}

this.state = ĐÃ DUYỆT;

}

Như bạn có thể thấy, mỗi phương pháp đã được thay thế bằng mộtquá trình()phương pháp và một hoặc nhiềuáp dụng()phương pháp. Cácsửa lại đơn hàng()phương pháp đã được thay thế bằngquá trình (ReviseOrder)Vàáp dụng(OrderRevisionProposed). Tương tự như vậy,xác nhậnSửa đổi()đã được thay thế bởiquá trình (Xác nhận Sửa đổi Đơn hàng)Vàáp dụng(OrderRevised).

###### Xử lý các bản cập nhật đồng thời bằng cách sử dụng khóa lạc quan

Không phải là hiếm khi hai hoặc nhiều yêu cầu cùng cập nhật một tổng hợp. Một ứng dụng sử dụng tính bền bỉ truyền thống thường sử dụng khóa lạc quan để ngăn một giao dịch ghi đè lên các thay đổi của giao dịch khác. Khóa lạc quan thường sử dụng cột phiên bản để phát hiện xem tổng hợp có thay đổi kể từ khi được đọc hay không. Ứng dụng ánh xạ gốc tổng hợp vào một bảng có cột VERSION, được tăng lên bất cứ khi nào tổng hợp được cập nhật. Ứng dụng cập nhật tổng hợp bằng cách sử dụng câu lệnh UPDATE như sau:

CẬP NHẬT AGGREGATE\_ROOT\_TABLE ĐẶT PHIÊN BẢN = PHIÊN BẢN+1 ...

WHERE VERSION = <phiên bản gốc>

Câu lệnh UPDATE này sẽ chỉ thành công nếu phiên bản không thay đổi kể từ khi ứng dụng đọc tổng hợp. Nếu hai giao dịch đọc cùng một tổng hợp, giao dịch đầu tiên cập nhật tổng hợp sẽ thành công. Giao dịch thứ hai sẽ thất bại vì số phiên bản đã thay đổi, do đó, nó sẽ không vô tình ghi đè lên các thay đổi của giao dịch đầu tiên.

Một kho sự kiện cũng có thể sử dụng khóa lạc quan để xử lý các bản cập nhật đồng thời. Mỗi phiên bản tổng hợp có một phiên bản được đọc cùng với các sự kiện. Khi ứng dụng chèn các sự kiện, kho sự kiện sẽ xác minh rằng phiên bản không thay đổi. Một

cách tiếp cận là sử dụng số sự kiện làm số phiên bản. Ngoài ra, như bạn sẽ thấy bên dưới trong phần 6.2, một kho sự kiện có thể duy trì một số phiên bản rõ ràng.

###### Sự kiện tìm nguồn và xuất bản sự kiện

Nói một cách chính xác, event sourcing lưu trữ các tập hợp dưới dạng sự kiện và tái tạo trạng thái hiện tại của một tập hợp từ các sự kiện đó. Bạn cũng có thể sử dụng event sourcing như một cơ chế xuất bản sự kiện đáng tin cậy. Việc lưu một sự kiện trong kho sự kiện là một hoạt động nguyên tử vốn có. Chúng ta cần triển khai một cơ chế để phân phối tất cả các sự kiện đã lưu trữ cho những người dùng quan tâm.

Chương 3 mô tả một số cơ chế khác nhau—thăm dò vàtransaction log tailing—để xuất bản các thông báo được chèn vào cơ sở dữ liệu như một phần của giao dịch. Một ứng dụng dựa trên nguồn sự kiện có thể xuất bản các sự kiện bằng một trong các cơ chế này. Sự khác biệt chính là nó lưu trữ vĩnh viễn các sự kiện trong bảng EVENTS thay vì lưu tạm thời các sự kiện trong bảng OUTBOX rồi xóa chúng. Chúng ta hãy xem xét từng cách tiếp cận, bắt đầu bằng việc thăm dò.

**BạnSING POLLING ĐỂ XUẤT BẢN SỰ KIỆN**

Nếu các sự kiện được lưu trữ trong bảng EVENTS được hiển thị trong hình 6.6, một nhà xuất bản sự kiện có thể thăm dò bảng để tìm các sự kiện mới bằng cách thực hiện một câu lệnh SELECT và xuất bản các sự kiện tới một nhà môi giới tin nhắn. Thách thức là xác định sự kiện nào là mới. Ví dụ, hãy tưởng tượng rằng eventId đang tăng đơn điệu. Cách tiếp cận hấp dẫn bề ngoài là để nhà xuất bản sự kiện ghi lại eventId cuối cùng mà nó đã xử lý. Nósau đó sẽ truy xuất các sự kiện mới bằng truy vấn như thế này: SELECT \* FROM EVENTS where event\_id > ? ORDER BY event\_id ASC.

Vấn đề với cách tiếp cận này là các giao dịch có thể cam kết theo thứ tự khác với thứ tự mà chúng tạo ra sự kiện. Do đó, người xuất bản sự kiện có thể vô tình bỏ qua một sự kiện. Hình 6.6 cho thấy một kịch bản như vậy.

**Cam kết cuối cùng**

**Lấy lại sự kiện 1020**

**Bỏ qua sự kiện1010 vì 1010 <= sự kiện 1020**

CHỌN \* TỪ SỰ KIỆN NƠI EVENT\_ID > 1020...

LÀM

CHỌN \* TỪ SỰ KIỆN NƠI EVENT\_ID > ....

LÀM

CHÈN sự kiện với EVENT\_ID = 1020

CHÈN sự kiện với EVENT\_ID = 1010

BẮT ĐẦU

BẮT ĐẦU

Giao dịch B

Giao dịchMỘT

**Hình 6.6 Một kịch bản trong đó một sự kiện bị bỏ qua vì giao dịch của nó*MỘT*cam kết sau khi giao dịch*B*. Thăm dò ý kiến ​​thấysự kiệnId=1020và sau đó bỏ quasự kiệnId=1010.**

TRONGkịch bản này, Giao dịch A chèn một sự kiện có EVENT\_ID là 1010. Tiếp theo, giao dịch B chèn một sự kiện có EVENT\_ID là 1020 rồi xác nhận. Nếu nhà xuất bản sự kiện bây giờ truy vấn bảng EVENTS, nó sẽ tìm thấy sự kiện 1020. Sau đó, sau khi giao dịch A xác nhận và sự kiện 1010 trở nên hiển thị, nhà xuất bản sự kiện sẽ bỏ qua sự kiện đó.

Một giải pháp cho vấn đề này là thêm một cột bổ sung vào bảng EVENTS để theo dõi xem sự kiện đã được công bố hay chưa. Sau đó, nhà xuất bản sự kiện sẽ sử dụng quy trình sau:

**1**Tìm các sự kiện chưa được công bố bằng cách thực hiện câu lệnh SELECT này:CHỌN \* TỪ CÁC SỰ KIỆN nơi ĐÃ XUẤT BẢN = 0 SẮP XẾP THEO event\_id ASC.

**2** Xuất bản sự kiện tới trình môi giới tin nhắn.

**3**Đánh dấu các sự kiện đã được công bố:CẬP NHẬT SỰ KIỆN ĐÃ XUẤT BẢN = 1NƠI EVENT\_ID trong.

Cách tiếp cận này ngăn chặn nhà xuất bản sự kiện bỏ qua các sự kiện.

**BạnSING TRANSACTION LOG TAILING ĐỂ XUẤT BẢN SỰ KIỆN MỘT CÁCH ĐÁNG TIN CẬY**

Các kho sự kiện phức tạp hơn sử dụng chức năng theo dõi nhật ký giao dịch, như chương 3 mô tả, chức năng này đảm bảo rằng các sự kiện sẽ được công bố và cũng hiệu suất và khả năng mở rộng cao hơn. Ví dụ, Eventuate Local, một kho sự kiện nguồn mở, sử dụng phương pháp này. Nó đọc các sự kiện được chèn vào bảng EVENTS từ nhật ký giao dịch cơ sở dữ liệu và công bố chúng cho trình môi giới tin nhắn. Phần 6.2 thảo luận về cách Eventuate Local hoạt động chi tiết hơn.

###### Sử dụng ảnh chụp nhanh để cải thiện hiệu suất

Một tổng hợp đơn hàng cótương đối ít chuyển đổi trạng thái, vì vậy nó chỉ có một số lượng nhỏ các sự kiện. Sẽ hiệu quả khi truy vấn kho sự kiện cho các sự kiện đó và xây dựng lại tổng hợp Order. Tuy nhiên, các tổng hợp tồn tại lâu dài có thể có một số lượng lớn các sự kiện. Ví dụ: tổng hợp Account có khả năng có một số lượng lớn các sự kiện. Theo thời gian, việc tải và gấp các sự kiện đó sẽ ngày càng kém hiệu quả.

Một giải pháp phổ biến là duy trì định kỳ một ảnh chụp nhanh về trạng thái của tổng hợp. Hình 6.7 cho thấy một ví dụ về việc sử dụng ảnh chụp nhanh. Ứng dụng khôi phục trạng thái của

**Ứng dụng chỉ một cần lấy lại ảnh chụp nhanh và các sự kiện xảy ra sau đó.**



Sự kiện1Sự kiện 2Sự kiện ...Sự kiện *N*

Sự kiện*N*+1 Sự kiện*N*+2

Phiên bản chụp nhanh*N*

**Hình 6.7 Sử dụng snapshot cải thiện hiệu suất bằng cách loại bỏ nhu cầu tải tất cả các sự kiện. Ứng dụng chỉ cần tải snapshot và các sự kiện xảy ra sau đó.**

tổng hợp bằng cách tải ảnh chụp nhanh gần đây nhất và chỉ những sự kiện đã xảy ra kể từ khi ảnh chụp nhanh được tạo.

Trong ví dụ này, phiên bản snapshot là N. Ứng dụng chỉ cần tải snapshot và hai sự kiện theo sau để khôi phục trạng thái của tổng hợp. N sự kiện trước đó không được tải từ kho lưu trữ sự kiện.

Khi khôi phục trạng thái của tổng hợp từ ảnh chụp nhanh, trước tiên ứng dụng sẽ tạo một phiên bản tổng hợp từsnapshot và sau đó lặp lại các sự kiện, áp dụng chúng. Ví dụ, khung Eventuate Client, được mô tả trong phần 6.2.2, sử dụng mã tương tự như sau để tái tạo tổng hợp:

Lớp aggregateClass = ...; Ảnh chụp nhanh snapshot = ...;

Tổng hợp tổng hợp = tái tạoFromSnapshot(aggregateClass, snapshot); cho (Sự kiện sự kiện: sự kiện) {

aggregate = aggregate.applyEvent(sự kiện);

}

// sử dụng tổng hợp...

Khi sử dụng snapshot, thể hiện tổng hợp được tạo lại từ snapshot thay vì được tạo bằng cách sử dụng constructor mặc định của nó. Nếu một aggregate có cấu trúc đơn giản, dễ tuần tự hóa, snapshot có thể là, ví dụ, tuần tự hóa JSON của nó. Các aggregate phức tạp hơn có thể được snapshot bằng cách sử dụng mẫu Memento ([https://vi.wikipedia](https://en.wikipedia.org/wiki/Memento_pattern)

[.org/wiki/Memento\_pattern](https://en.wikipedia.org/wiki/Memento_pattern)).

Tổng hợp khách hàng trong ví dụ về cửa hàng trực tuyến có cấu trúc rất đơn giản: thông tin khách hàng, hạn mức tín dụng và hạn mức tín dụng của họ. Một snap-ảnh chụp của Khách hàng là chuỗi hóa JSON về trạng thái của Khách hàng. Hình 6.8 cho thấy cách tạo lại Khách hàng từ ảnh chụp nhanh tương ứng với trạng thái của Khách hàng kể từ sự kiện #103. Dịch vụ Khách hàng cần tải ảnh chụp nhanh và các sự kiện đã xảy ra sau sự kiện #103.

SỰ KIỆN CHỤP ẢNH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sự kiện\_id | loại sự kiện | loại thực thể | thực thể\_id | dữ liệu sự kiện |
| ... | ... | ... | ... | ... |
| 103 | ... | Khách hàng | 101 | {...} |
| 104 | Tín dụng được giữ lại | Khách hàng | 101 | {...} |
| 105 | Địa chỉ đã thay đổi | Khách hàng | 101 | {...} |
| 106 | Tín dụng được giữ lại | Khách hàng | 101 | {...} |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | sự kiện\_id | loại thực thể | sự kiện\_id | dữ liệu ảnh chụp nhanh |
| ... | ... | ... | ... |
| 103 | Khách hàng | 101 | {tên: "..." , ...} |
|  |
| ... | ... | ... | ... |
| ... | ... | ... | ... |

**Hình 6.8Dịch vụ khách hàngtái tạoKhách hàngbằng cách hủy tuần tự hóa JSON của ảnh chụp nhanh và sau đóđang tải và áp dụng các sự kiện #104 đến #106.**

Dịch vụ khách hàng tạo lại Khách hàng bằng cách hủy tuần tự hóa JSON của ảnh chụp nhanh, sau đó tải và áp dụng các sự kiện từ #104 đến #106.

###### Xử lý tin nhắn bất biến

Các dịch vụ thường sử dụng tin nhắn từ các ứng dụng khác hoặc các dịch vụ khác. Ví dụ, một dịch vụ có thể sử dụng các sự kiện miền do các tổng hợp công bố hoặc tin nhắn lệnh do một saga orchestrator gửi. Như đã mô tả trong chương 3, một vấn đề quan trọng khi phát triển một người tiêu dùng tin nhắn là đảm bảo rằng nó là idempotent, vì một message-broker có thể gửi cùng một tin nhắn nhiều lần.

Người dùng tin nhắn là idempotent nếu nó có thể được gọi một cách an toàn với cùng một tin nhắn nhiều lần. Ví dụ, khung Eventuate Tram triển khai xử lý tin nhắn idempotent bằng cách phát hiện và loại bỏ các tin nhắn trùng lặp. Nó ghi lạiid của các thông điệp đã xử lý trong bảng PROCESSED\_MESSAGES như một phần của giao dịch ACID cục bộ được logic nghiệp vụ sử dụng để tạo hoặc cập nhật tổng hợp. Nếu ID của một thông điệp nằm trong bảng PROCESSED\_MESSAGES, thì đó là bản sao và có thể bị loại bỏ. Logic nghiệp vụ dựa trên nguồn sự kiện phải triển khai một cơ chế tương đương. Cách thực hiện tùy thuộc vào việc kho sự kiện sử dụng RDBMS hay cơ sở dữ liệu NoSQL.

**TÔIXỬ LÝ TIN NHẮN YẾU TỐ VỚI MỘTRDBMS-CỬA HÀNG SỰ KIỆN DỰA TRÊN**

Nếu một ứng dụng sử dụng kho lưu trữ sự kiện dựa trên RDBMS, nó có thể sử dụng phương pháp giống hệt nhau để phát hiện và loại bỏ các tin nhắn trùng lặp. Nó chèn ID tin nhắn vào PROCESSED

Bảng \_MESSAGES là một phần của giao dịchchèn các sự kiện vào bảng EVENTS.

**TÔIXỬ LÝ TIN NHẮN YẾU TỐ KHI SỬ DỤNGNỒSQL-CỬA HÀNG SỰ KIỆN DỰA TRÊN**

Một kho lưu trữ sự kiện dựa trên NoSQL, có mô hình giao dịch hạn chế, phải sử dụng một cơ chế khác để triển khai xử lý tin nhắn bất biến. Một người dùng tin nhắn phải lưu trữ các sự kiện một cách nguyên tử và ghi lại ID tin nhắn. May mắn thay, có một giải pháp đơn giản. Một người dùng tin nhắn lưu trữ ID của tin nhắn trong các sự kiện được tạo ra trong khi xử lý nó. Nó phát hiện các bản sao bằng cách xác minh rằng không có sự kiện nào của aggregate chứa ID tin nhắn.

Một thách thức khi sử dụng cách tiếp cận này là việc xử lý tin nhắn có thể không tạo ra bất kỳ sự kiện nào. Việc thiếu sự kiện có nghĩa là không có hồ sơ nào về việc tin nhắn đã được xử lý. Việc phân phối lại và xử lý lại cùng một tin nhắn sau đó có thể dẫn đến hành vi không chính xác. Ví dụ, hãy xem xét tình huống sau:

**1** Tin nhắn A được xử lý nhưng không cập nhật tổng hợp.

**2** Tin nhắn B được xử lý và người dùng tin nhắn cập nhật tổng hợp.

**3**Tin nhắn A được gửi lại và vì không có hồ sơ nào cho thấy tin nhắn này đã được xử lý nên người dùng tin nhắn sẽ cập nhật tổng hợp.

**4** Tin nhắn B được xử lý lại….

Trong trường hợp này, việc truyền lại các sự kiện sẽ dẫn đến kết quả khác và có thể là sai.

Một cách để tránh vấn đề này là luôn công bố một sự kiện. Nếu tổng hợp không phát ra sự kiện, ứng dụng sẽ lưu một sự kiện giả chỉ để ghi lại ID tin nhắn. Người dùng sự kiện phải bỏ qua các sự kiện giả này.

###### Sự kiện miền đang phát triển

Event sourcing, ít nhất là về mặt khái niệm, lưu trữ sự kiện mãi mãi—đây là con dao hai lưỡi. Một mặt, nó cung cấp cho ứng dụng nhật ký kiểm tra các thay đổi được đảm bảo là chính xác. Nó cũng cho phép ứng dụng tái tạo trạng thái lịch sử của tổng hợp. Mặt khác, nó tạo ra một thách thức, vì cấu trúc của các sự kiện thường thay đổi theo thời gian.

Một ứng dụng có khả năng phải xử lý nhiều phiên bản sự kiện. Ví dụ, một dịch vụ tải tổng hợp Đơn hàng có khả năng cần phải gấp nhiều phiên bản sự kiện. Tương tự, một người đăng ký sự kiện có khả năng thấy nhiều phiên bản.

Đầu tiên chúng ta hãy xem xét những cách khác nhaurằng các sự kiện có thể thay đổi, và sau đó tôi sẽ mô tả một cách tiếp cận thường được sử dụng để xử lý những thay đổi.

**ESƠ ĐỒ THÔNG GIÓ TIẾN HÓA**

Về mặt khái niệm, một ứng dụng tìm nguồn sự kiện có một lược đồ được tổ chức thành ba cấp độ:

* + - * Bao gồm một hoặc nhiều tập hợp
      * Xác định các sự kiện mà mỗi tổng hợp phát ra
      * Xác định cấu trúc của các sự kiện

Bảng 6.1 hiển thị các loại thay đổi khác nhau có thể xảy ra ở mỗi cấp độ.

**Bảng 6.1 Những cách khác nhau mà các sự kiện của ứng dụng có thể phát triển**

|  |  |  |
| --- | --- | --- |
| **Mức độ** | **Thay đổi** | **Tương thích ngược** |
| Sơ đồ | Xác định một loại tổng hợp mới | Đúng |
| Xóa tổng hợp | Xóa một tổng hợp hiện có | KHÔNG |
| Đổi tên tổng hợp | Thay đổi tên của một tổng hợpkiểu | KHÔNG |
| Tổng hợp | Thêm một loại sự kiện mới | Đúng |
| Xóa sự kiện | Xóa một loại sự kiện | KHÔNG |
| Đổi tên sự kiện | Thay đổi tên của một sự kiệnkiểu | KHÔNG |
| Sự kiện | Thêm một trường mới | Đúng |
| Xóa trường | Xóa một trường | KHÔNG |
| Đổi tên trường | Đổi tên một trường | KHÔNG |
| Thay đổi loại trường | Thay đổi loại trường | KHÔNG |

Những thay đổi này diễn ra tự nhiên khi mô hình miền của dịch vụ phát triển theo thời gian—ví dụ, khi các yêu cầu của dịch vụ thay đổi hoặc khi các nhà phát triển của dịch vụ có được cái nhìn sâu sắc hơn về miền và cải thiện mô hình miền. Ở cấp độ lược đồ, các nhà phát triển thêm, xóa và đổi tên các lớp tổng hợp. Ở cấp độ tổng hợp, các loại sự kiện

phát ra bởi một tổng hợp cụ thể có thể thay đổi. Nhà phát triển có thể thay đổi cấu trúc của loại sự kiện bằng cách thêm, xóa và thay đổi tên hoặc loại của trường.

May mắn thay, nhiều loại thay đổi này là những thay đổi tương thích ngược. Ví dụ, việc thêm một trường vào một sự kiện không có khả năng ảnh hưởng đến người dùng. Người dùng bỏ qua các trường không xác định. Tuy nhiên, những thay đổi khác không tương thích ngược. Ví dụ, việc thay đổi tên của một sự kiện hoặc tên của một trường yêu cầu người dùng của loại sự kiện đó phải thay đổi.

**TôiGIÀU CÓSƠ ĐỒ THAY ĐỔI THÔNG QUA UCPASTING**

Trong thế giới cơ sở dữ liệu SQL, các thay đổi đối với lược đồ cơ sở dữ liệu thường được xử lý bằng cách sử dụng di chuyển lược đồ. Mỗi thay đổi lược đồ được biểu diễn bằng một di chuyển, một tập lệnh SQL thay đổi lược đồ và di chuyển dữ liệu sang lược đồ mới. Di chuyển lược đồ được lưu trữ trong hệ thống kiểm soát phiên bản và được áp dụng cho cơ sở dữ liệu bằng một công cụ như Flyway.

Một ứng dụng nguồn sự kiện có thể sử dụng một cách tiếp cận tương tự để xử lý các thay đổi không tương thích ngược. Nhưng thay vì di chuyển các sự kiện sang phiên bản lược đồ mới tại chỗ, các khuôn khổ nguồn sự kiện chuyển đổi các sự kiện khi chúng được tải từ kho sự kiện. Một thành phần thường được gọi là upcaster cập nhật các sự kiện riêng lẻ từ phiên bản cũ sang phiên bản mới hơn. Do đó, mã ứng dụng chỉ xử lý lược đồ sự kiện hiện tại.

Bây giờ chúng ta đã xem xét cách thức hoạt động của nguồn sự kiện, hãy cùng xem xét những lợi ích và hạn chế của nó.

###### Lợi ích của việc tìm nguồn sự kiện

Nguồn sự kiện có cả hai lợi íchvà nhược điểm. Những lợi ích bao gồm:

* + - * Xuất bản các sự kiện miền một cách đáng tin cậy
      * Lưu giữ lịch sử của các tập hợp
      * Hầu hết tránh được vấn đề không khớp trở kháng O/R
      * Cung cấp cho các nhà phát triển một cỗ máy thời gian Hãy cùng xem xét từng lợi ích chi tiết hơn.

**RELIABLY XUẤT BẢN CÁC SỰ KIỆN MIỀN**

Một lợi ích chính của việc tìm nguồn sự kiện là nó sẽ xuất bản các sự kiện một cách đáng tin cậy bất cứ khi nào trạng thái của tổng hợp thay đổi. Đó là nền tảng tốt cho kiến ​​trúc vi dịch vụ theo sự kiện. Ngoài ra, vì mỗi sự kiện có thể lưu trữ danh tính của người dùng đã thực hiện thay đổi, nên việc tìm nguồn sự kiện cung cấp nhật ký kiểm tra được đảm bảo là chính xác. Luồng sự kiện có thể được sử dụng cho nhiều mục đích khác nhau, bao gồm thông báo cho người dùng, tích hợp ứng dụng, phân tích và giám sát.

**PDỰ TRỮ LỊCH SỬ CỦA CÁC ĐÁNH GIÁ**

Một lợi ích khác của việc tìm nguồn sự kiện là nó lưu trữ toàn bộ lịch sử của mỗi tổng hợp. Bạn có thể dễ dàng triển khai các truy vấn thời gian để truy xuất trạng thái trước đó của một tổng hợp. Để xác định trạng thái của một tổng hợp tại một thời điểm nhất định, bạn gấp các sự kiện

đã xảy ra cho đến thời điểm đó. Ví dụ, rất dễ để tính toán tín dụng khả dụng của một khách hàng tại một thời điểm nào đó trong quá khứ.

**TôiOSTLY TRÁNHHOẶCVẤN ĐỀ KHÔNG PHÙ HỢP TRỞ KHÁNG**

Nguồn sự kiện duy trì các sự kiện thay vì tổng hợp chúng. Sự kiện thường có cấu trúc đơn giản, dễ tuần tự hóa. Như đã đề cập trước đó, một dịch vụ có thể chụp nhanh một tổng hợp phức tạp bằng cách tuần tự hóa một bản ghi nhớ về trạng thái của nó, điều này bổ sung một mức độ gián tiếp giữa một tổng hợp và biểu diễn tuần tự hóa của nó.

**PROVIDES NHÀ PHÁT TRIỂN VỚI CỖ MÁY THỜI GIAN**

Event sourcing lưu trữ lịch sử mọi thứ đã xảy ra trong suốt vòng đời của một ứng dụng. Hãy tưởng tượng rằng các nhà phát triển FTGO cần triển khai một yêu cầu mới cho những khách hàng đã thêm một mặt hàng vào giỏ hàng của họ rồi xóa nó đi. Một ứng dụng truyền thống sẽ không lưu giữ thông tin này, do đó chỉ có thể tiếp thị cho những khách hàng thêm và xóa các mặt hàng sau khi tính năng được triển khai. Ngược lại, một ứng dụng dựa trên event sourcing có thể tiếp thị ngay lập tức cho những khách hàng đã thực hiện việc này trong quá khứ. Nó giống như event sourcing cung cấp cho các nhà phát triển một cỗ máy thời gian để du hành về quá khứ và triển khai các yêu cầu không lường trước được.

###### Nhược điểm của việc tìm nguồn sự kiện

Nguồn sự kiện không phải là giải pháp hoàn hảo. Nó có những nhược điểm sau:

* + - * Nó có một mô hình lập trình khác với đường cong học tập khác nhau.
      * Nó có độ phức tạp của một ứng dụng dựa trên tin nhắn.
      * Các sự kiện diễn biến có thể rất phức tạp.
      * Việc xóa dữ liệu rất khó khăn.
      * Truy vấn kho sự kiện là một thách thức. Hãy cùng xem xét từng nhược điểm.

**DMÔ HÌNH LẬP TRÌNH KHÁC NHAU CÓ ĐƯỜNG CONG HỌC TẬP**

Đây là một mô hình lập trình khác biệt và không quen thuộc, và điều đó có nghĩa là một đường cong học tập. Để một ứng dụng hiện có sử dụng event sourcing, bạn phải viết lại logic nghiệp vụ của ứng dụng đó. May mắn thay, đó là một sự chuyển đổi khá cơ học mà bạn có thể thực hiện khi di chuyển ứng dụng của mình sang các dịch vụ siêu nhỏ.

**CSự phức tạp của một thông điệp-DỰA TRÊNỨNG DỤNG**

Một nhược điểm khác của việc tìm nguồn sự kiện là các nhà môi giới tin nhắn thường đảm bảo việc phân phối ít nhất một lần. Các trình xử lý sự kiện không phải là idempotent phải phát hiện và loại bỏ các sự kiện trùng lặp. Khung tìm nguồn sự kiện có thể giúp bằng cách chỉ định cho mỗi sự kiện một ID tăng dần theo đơn điệu. Sau đó, trình xử lý sự kiện có thể phát hiện các sự kiện trùng lặp bằng cách theo dõi ID sự kiện được nhìn thấy nhiều nhất. Điều này thậm chí xảy ra tự động khi trình xử lý sự kiện cập nhật tổng hợp.

**ECÁC SỰ KIỆN DIỄN RA CÓ THỂ KHÓ KHĂN**

Với nguồn sự kiện, lược đồ sự kiện (và ảnh chụp nhanh!) sẽ phát triển theo thời gian. Vì sự kiện được lưu trữ mãi mãi, nên các tổng hợp có khả năng cần phải gấp các sự kiện tương ứng với nhiều phiên bản lược đồ. Có một rủi ro thực sự là các tổng hợp có thể trở nên phình to với mã để xử lý tất cả các phiên bản khác nhau. Như đã đề cập trong phần 6.1.7, một giải pháp tốt cho vấn đề này là nâng cấp sự kiện lên phiên bản mới nhất khi chúng được tải từ kho sự kiện. Cách tiếp cận này tách mã nâng cấp sự kiện khỏi tổng hợp, giúp đơn giản hóa các tổng hợp vì chúng chỉ cần áp dụng phiên bản mới nhất của sự kiện.

**DXÓA DỮ LIỆUTHẬT KHÓ KHĂN**

Bởi vì một trong những mục tiêu của nguồn sự kiện là bảo toàn lịch sử của các tổng hợp, nên nó cố tình lưu trữ dữ liệu mãi mãi. Cách truyền thống để xóa dữ liệu khi sử dụng nguồn sự kiện là thực hiện xóa mềm. Một ứng dụng xóa một tổng hợp bằng cách đặt*đã xóa*cờ. Tổng hợp thường sẽ phát ra sự kiện Đã xóa, thông báo cho bất kỳ người tiêu dùng nào quan tâm. Bất kỳ mã nào truy cập tổng hợp đó đều có thể kiểm tra cờ và hành động theo đó.

Sử dụng chức năng xóa mềm có hiệu quả với nhiều loại dữ liệu. Tuy nhiên, một thách thức là tuân thủ Quy định bảo vệ dữ liệu chung (GDPR), một quy định về bảo vệ dữ liệu và quyền riêng tư của Châu Âu cấp cho cá nhân quyền xóa ([https://](https://gdpr-info.eu/art-17-gdpr/) [gdpr-info.eu/nghệ thuật-17-gdpr/](https://gdpr-info.eu/art-17-gdpr/)). Một ứng dụng phải có khả năng quên thông tin cá nhân của người dùng, chẳng hạn như địa chỉ email của họ. Vấn đề với ứng dụng dựa trên nguồn sự kiện là địa chỉ email có thể được lưu trữ trong sự kiện AccountCreated hoặc được sử dụng làm khóa chính của tổng hợp. Ứng dụng bằng cách nào đó phải quên thông tin về người dùng mà không xóa các sự kiện.

Mã hóa là một cơ chế bạn có thể sử dụng để giải quyết vấn đề này. Mỗi người dùng có một khóa mã hóa, được lưu trữ trong một bảng cơ sở dữ liệu riêng biệt. Ứng dụng sử dụng khóa mã hóa đó để mã hóa bất kỳ sự kiện nào chứa thông tin cá nhân của người dùng trước khi lưu trữ chúng trong kho sự kiện. Khi người dùng yêu cầu xóa, ứng dụng sẽ xóa bản ghi khóa mã hóa khỏi bảng cơ sở dữ liệu. Thông tin cá nhân của người dùng thực sự bị xóa, vì các sự kiện không còn có thể được giải mã nữa.

Mã hóa sự kiện giải quyết hầu hết các vấn đề liên quan đến việc xóa thông tin cá nhân của người dùng. Nhưng nếu một số khía cạnh của thông tin cá nhân của người dùng, chẳng hạn như địa chỉ email, được sử dụng làm ID tổng hợp, thì việc xóa khóa mã hóa có thể không đủ. Ví dụ, phần 6.2 mô tả một kho lưu trữ sự kiện có bảng thực thể có khóa chính là ID tổng hợp. Một giải pháp cho vấn đề này là sử dụng kỹ thuật giả danh, thay thế địa chỉ email bằng mã thông báo UUID và sử dụng mã thông báo đó làm ID tổng hợp. Ứng dụng lưu trữ mối liên kết giữa mã thông báo UUID và địa chỉ email trong bảng cơ sở dữ liệu. Khi người dùng yêu cầu xóa, ứng dụng sẽ xóa hàng cho địa chỉ email của họ khỏi bảng đó. Điều này ngăn ứng dụng ánh xạ UUID trở lại địa chỉ email.

**HỏiSỬ DỤNG CỬA HÀNG SỰ KIỆN LÀ MỘT THÁCH THỨC**

Hãy tưởng tượng bạn cần tìm những khách hàng đã sử dụng hết hạn mức tín dụng. Bởi vìkhông có cột nào chứa tín dụng, bạn không thể viết SELECT \* FROM CUSTOMER WHERE CREDIT\_LIMIT = 0. Thay vào đó, bạn phải sử dụng truy vấn phức tạp hơn và có khả năng kém hiệu quả hơn có SELECT lồng nhau để tính toán hạn mức tín dụng bằng cách gấp các sự kiện đặt tín dụng ban đầu và điều chỉnh nó. Tệ hơn nữa, kho lưu trữ sự kiện dựa trên NoSQL thường chỉ hỗ trợ tra cứu dựa trên khóa chính. Do đó, bạn phải triển khai các truy vấn bằng cách sử dụng phương pháp CQRS được mô tả trong chương 7.

#### Triển khai một cửa hàng sự kiện

Một ứng dụng sử dụng nguồn sự kiện lưu trữ các sự kiện của nó trong một kho sự kiện. Kho sự kiện là sự kết hợp giữa cơ sở dữ liệu và một nhà môi giới tin nhắn. Nó hoạt động như một cơ sở dữ liệu vì nó có một API để chèn và truy xuất các sự kiện của tổng hợp theo khóa chính. Và nó hoạt động như một nhà môi giới tin nhắn vì nó có một API để đăng ký các sự kiện.

Có một số cách khác nhau để triển khai một kho sự kiện. Một lựa chọn là triển khai kho sự kiện và khung tìm nguồn sự kiện của riêng bạn. Ví dụ, bạn có thể duy trì các sự kiện trong RDBMS. Một cách đơn giản, mặc dù hiệu suất thấp, để xuất bản sự kiện là để người đăng ký thăm dò bảng EVENTS để biết các sự kiện. Nhưng, như đã lưu ý trong phần 6.1.4, một thách thức là đảm bảo rằng người đăng ký xử lý tất cả các sự kiện theo thứ tự.

Một lựa chọn khác là sử dụng kho sự kiện chuyên dụng, thường cung cấp một bộ tính năng phong phú và hiệu suất cũng như khả năng mở rộng tốt hơn. Có một số lựa chọn sau:

* *Cửa hàng sự kiện*—Một kho sự kiện nguồn mở dựa trên .NET được phát triển bởi Greg Young, một người tiên phong trong lĩnh vực tìm nguồn sự kiện ([https://eventstore.org](https://eventstore.org/)).
* *Thời gian*—Một khuôn khổ dịch vụ vi mô được phát triển bởi Lightbend, công ty trước đây được gọi là Typesafe ([www.lightbend.com/lagom-framework](http://www.lightbend.com/lagom-framework)).
* *Sợi trục*—Một khuôn khổ Java nguồn mở để phát triển các ứng dụng hướng sự kiện sử dụng nguồn sự kiện và CQRS ([www.axonframework.org](http://www.axonframework.org/)).
* *Sự kiện xảy ra*—Được phát triển bởi công ty khởi nghiệp của tôi, Eventuate ([http://eventuate.io](http://eventuate.io/)). Cóhai phiên bản của Eventuate: Eventuate SaaS, một dịch vụ đám mây và Eventuate Local, một dự án nguồn mở dựa trên Apache Kafka/RDBMS.

Mặc dù các khuôn khổ này khác nhau về chi tiết, nhưng các khái niệm cốt lõi vẫn giống nhau. Vì Eventuate là khuôn khổ mà tôi quen thuộc nhất, nên đó là khuôn khổ mà tôi sẽ đề cập ở đây. Nó có kiến ​​trúc đơn giản, dễ hiểu, minh họa các khái niệm về nguồn sự kiện. Bạn có thể sử dụng nó trong các ứng dụng của mình, tự triển khai lại các khái niệm hoặc áp dụng những gì bạn học được ở đây để xây dựng các ứng dụng với một trong các khuôn khổ nguồn sự kiện khác.

Tôi bắt đầu các phần sau bằng cách mô tả cách hoạt động của kho sự kiện Eventuate Local. Sau đó, tôi mô tả khung Eventuate Client cho Java, một khung dễ sử dụng để viết logic kinh doanh dựa trên nguồn sự kiện sử dụng kho sự kiện Eventuate Local.

###### Cửa hàng sự kiện Eventuate Local hoạt động như thế nào

Eventuate Local là một kho lưu trữ sự kiện nguồn mở. Hình 6.9 cho thấy kiến ​​trúc. Sự kiện được lưu trữ trong cơ sở dữ liệu, chẳng hạn như MySQL. Các ứng dụng chèn và truy xuất các sự kiện tổng hợp theo khóa chính. Các ứng dụng sử dụng sự kiện từ một nhà môi giới tin nhắn, chẳng hạn như Apache Kafka. Một cơ chế theo dõi nhật ký giao dịch truyền các sự kiện từ cơ sở dữ liệu đến nhà môi giới tin nhắn.

**Lưu trữ các sự kiện**



Sự kiện chuyển tiếp

Cơ sở dữ liệu sự kiện

SỰ KIỆN

Môi giới sự kiện

THỰC THỂ

ẢNH CHỤP

**Xuất bản các sự kiện được lưu trữ trong cơ sở dữ liệu tới nhà môi giới tin nhắn**

Ứng dụng

Sự kiện chuyển tiếp

Chủ đề khách hàng

Đặt hàngđề tài

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sự kiện\_id | loại sự kiện | loại thực thể | thực thể\_id | dữ liệu sự kiện |
| 102 | Đã tạo đơn hàng | Đặt hàng | 101 | {...} |
| 103 | Đơn hàng đã được chấp thuận | Đặt hàng | 101 | {...} |
| ... | ... | ... | ... | ... |

|  |  |  |  |
| --- | --- | --- | --- |
| loại thực thể | thực thể\_id | phiên bản thực thể | ... |
| ... | ... | ... | ... |

|  |  |  |  |
| --- | --- | --- | --- |
| loại thực thể | thực thể\_id | phiên bản thực thể | ... |
| ... | ... | ... | ... |

**Hình 6.9Kiến trúc của Eventuate Local. Nó bao gồm một cơ sở dữ liệu sự kiện (như MySQL) lưu trữ các sự kiện, một nhà môi giới sự kiện (như Apache Kafka) phân phối các sự kiện cho người đăng ký và một rơle sự kiện xuất bản các sự kiện được lưu trữ trong cơ sở dữ liệu sự kiện cho nhà môi giới sự kiện.**

Hãy cùng xem xét các thành phần Eventuate Local khác nhau, bắt đầu với lược đồ cơ sở dữ liệu.

**TSƠ ĐỒ CỦAETHỬ THÁCHLOCAL'CƠ SỞ DỮ LIỆU SỰ KIỆN**

Cơ sở dữ liệu sự kiện bao gồm ba bảng:

* sự kiện—Lưu trữ các sự kiện
* các thực thể—Một hàng cho mỗi thực thể
* ảnh chụp nhanh—Lưu trữ ảnh chụp nhanh

Bảng trung tâm là bảng sự kiện. Cấu trúc của bảng này rất giống với bảng được hiển thị trong hình 6.2. Sau đây là định nghĩa của nó:

tạo sự kiện bảng (

event\_id varchar(1000) KHÓA CHÍNH, event\_type varchar(1000), event\_data varchar(1000) KHÔNG NULL, entity\_type VARCHAR(1000) KHÔNG NULL, entity\_id VARCHAR(1000) KHÔNG NULL, triggering\_event VARCHAR(1000)

);

Cột triggering\_event được sử dụng để phát hiện các sự kiện/tin nhắn trùng lặp. Nó lưu trữ ID của tin nhắn/sự kiện mà quá trình xử lý tạo ra sự kiện này.

Bảng thực thể lưu trữ phiên bản hiện tại của mỗi thực thể. Nó được sử dụng để triển khai khóa lạc quan. Sau đây là định nghĩa của bảng này:

tạo bảng thực thể ( entity\_type VARCHAR(1000), entity\_id VARCHAR(1000),

entity\_version VARCHAR(1000) KHÔNG NULL, KHÓA CHÍNH(entity\_type, entity\_id)

);

Khi một thực thể được tạo ra, một hàng được chèn vào bảng này. Mỗi lần một thực thể đượcđã cập nhật, cột entity\_version được cập nhật.

Bảng snapshot lưu trữ snapshot của từng thực thể. Sau đây là định nghĩa của

bảng này:

tạo ảnh chụp nhanh bảng ( entity\_type VARCHAR(1000), entity\_id VARCHAR(1000), entity\_version VARCHAR(1000),

snapshot\_type VARCHAR(1000) KHÔNG NULL, snapshot\_json VARCHAR(1000) KHÔNG NULL, triggering\_events VARCHAR(1000),

KHÓA CHÍNH(entity\_type, entity\_id, entity\_version)

)

Cácloại thực thểVàthực thể\_idcác cột chỉ địnhthực thể của snapshot.ảnh chụp nhanh

Cột \_json là biểu diễn tuần tự của snapshot và snapshot\_type là kiểu của snapshot. entity\_version chỉ định phiên bản của thực thể mà snapshot này là.

Ba hoạt động được hỗ trợ bởi lược đồ này làtìm thấy(),tạo nên(), Vàcập nhật(). Cáctìm thấy()hoạt động truy vấnảnh chụp nhanhbảng để lấy ảnh chụp nhanh mới nhất, nếu có. Nếu ảnh chụp nhanh tồn tại,tìm thấy()hoạt động truy vấnsự kiệnbảng đểtìm tất cả các sự kiện cósự kiện\_idlớn hơn ảnh chụp nhanhphiên bản thực thể. Nếu không thì,tìm thấy()lấy tất cả các sự kiện cho thực thể được chỉ định.tìm thấy()hoạt động cũng truy vấnthực thểbảng để lấy phiên bản hiện tại của thực thể.

Cáctạo nên()hoạt động chèn một hàng vàocáithực thểbảng và chèn các sự kiệnvàosự kiệnbảng. Cáccập nhật()hoạt động chèn các sự kiện vàosự kiệnbàn. Nó

cũng thực hiện kiểm tra khóa lạc quan bằng cách cập nhật phiên bản thực thể trong

các thực thểbảng sử dụng nàyCẬP NHẬTtuyên bố:

CẬP NHẬT thực thể ĐẶT entity\_version = ?

Ở ĐÂUE entity\_type = ? và entity\_id=? và entity\_version = ?

Tuyên bố này xác minh rằngphiên bản không thay đổi kể từ khi nó được lấy ra bởi find()hoạt động. Nó cũng cập nhật entity\_version lên phiên bản mới. Hoạt động update() thực hiện các cập nhật này trong một giao dịch để đảm bảo tính nguyên tử.

Bây giờ chúng ta đã xem cách Eventuate Local lưu trữ các sự kiện và ảnh chụp nhanh của tổng hợp, hãy cùng xem cách máy khách đăng ký các sự kiện bằng trình môi giới sự kiện của Eventuate Local.

**CSỰ KIỆN TIÊU DÙNG BẰNG CÁCH ĐĂNG KÝETHỬ THÁCHLOCAL'S SỰ KIỆN MÔI GIỚI**

Các dịch vụ sử dụng sự kiện bằng cách đăng ký với event broker, được triển khai bằng Apache Kafka. Event broker có một chủ đề cho mỗi loại tổng hợp. Như đã mô tả trong chương 3, một chủ đề là một kênh tin nhắn được phân vùng. Điều này cho phép người dùng mở rộng theo chiều ngang trong khi vẫn giữ nguyên thứ tự tin nhắn. ID tổng hợp được sử dụng làm khóa phân vùng, giúp giữ nguyên thứ tự các sự kiện được xuất bản bởi một tổng hợp nhất định. Để sử dụng các sự kiện của một tổng hợp, một dịch vụ đăng ký với chủ đề của tổng hợp.

Bây giờ chúng ta hãy xem xét chuyển tiếp sự kiện—chất kết dính giữa cơ sở dữ liệu sự kiện và trình môi giới sự kiện.

**TANH TAETHỬ THÁCHLOCAL EVENT RELAY TRUYỀN TẢI CÁC SỰ KIỆN TỪ CƠ SỞ DỮ LIỆU ĐẾN MESSAGE BROKER**

Chuyển tiếp sự kiện truyền các sự kiện được chèn vào cơ sở dữ liệu sự kiện đến trình môi giới sự kiện. Nó sử dụng chức năng theo dõi nhật ký giao dịch bất cứ khi nào có thể và thăm dò các cơ sở dữ liệu khác. Ví dụ, phiên bản MySQL của chuyển tiếp sự kiện sử dụng giao thức sao chép chủ/tớ MySQL. Chuyển tiếp sự kiện kết nối với máy chủ MySQL như thể nó là một máy chủ tớ và đọc binlog MySQL, một bản ghi các bản cập nhật được thực hiện cho cơ sở dữ liệu. Các mục chèn vào bảng EVENTS, tương ứng với các sự kiện, được xuất bản tới chủ đề Apache Kafka thích hợp. Chuyển tiếp sự kiện bỏ qua mọi loại thay đổi khác.

Chuyển tiếp sự kiện được triển khai như một quy trình độc lập. Để khởi động lại đúng cách, nó sẽ lưu định kỳ vị trí hiện tại trong binlog—tên tệp và độ lệch—trong một chủ đề Apache Kafka đặc biệt. Khi khởi động, đầu tiên nó sẽ truy xuất vị trí được ghi lại gần đây nhất từ ​​chủ đề. Sau đó, chuyển tiếp sự kiện bắt đầu đọc binlog MySQL từ vị trí đó.

Cơ sở dữ liệu sự kiện, bộ trung gian tin nhắn và chuyển tiếp sự kiện tạo nên kho lưu trữ sự kiện.

Bây giờ chúng ta hãy xem xét khuôn khổ mà ứng dụng Java sử dụng để truy cập kho sự kiện.

###### Khung máy khách Eventuate cho Java

Khung máy khách Eventuate cho phép các nhà phát triển viết các ứng dụng dựa trên nguồn sự kiện sử dụng kho sự kiện Eventuate Local. Khung, được hiển thị trong hình 6.10, cung cấp nền tảng để phát triển các tổng hợp, dịch vụ và trình xử lý sự kiện dựa trên nguồn sự kiện.

**Các lớp trừu tượng và giao diện mà các lớp ứng dụng mở rộng hoặc triển khai**



«lệnh»«sự kiện»

Tạo đơn hàngĐơn hàngĐã tạo

tín dụngĐã đặt trước()

tạo đơn hàng()

xử lý() áp dụng()

Trình xử lý sự kiện OrderService

«giao diện» OrderEvent

«giao diện» OrderCommand

Đặt hàng

Dịch vụ đặt hàng

Dịch vụ đặt hàng

cứu()

tìm() cập nhật()

Kho lưu trữ tổng hợp

Khung khách hàng Eventuate

«chú thích» Người đăng ký sự kiện

Sự kiện «giao diện»

Lệnh «giao diện»

«trừu tượng» ReflectiveMutableCommand

Xử lýTổng hợp

**Hình 6.10 Các lớp và giao diện chính được cung cấp bởi khung máy khách Eventuate cho Java**

Khung cung cấp các lớp cơ sở cho các tập hợp, lệnh và sự kiện. Cócũng là một lớp AggregateRepository cung cấp chức năng CRUD. Và khung có API để đăng ký sự kiện.

Chúng ta hãy cùng xem xét sơ qua từng loại được thể hiện trong hình 6.10.

**DĐỊNH NGHĨA CÁC TỔNG HỢP VỚIRHIỆU QUẢTôiBẢNG SỬ DỤNGCOMANDPXAYMỘTLỚP TỔNG HỢP**

ReflectiveMutableCommandProcessingAggregatelà lớp cơ sở cho các tập hợp. Đây là một lớp chung có hai tham số kiểu: tham số đầu tiên là lớp tập hợp cụ thể và tham số thứ hai là siêu lớp của các lớp lệnh của tập hợp. Như tên khá dài của nó gợi ý, nó sử dụng phản xạ để phân phối lệnh và sự kiện đến phương thức thích hợp. Các lệnh được phân phối đến phương thức process() và các sự kiện đến phương thức apply().

CácĐặt hànglớp bạn đã thấy trước đó mở rộngReflectiveMutableCommandProcessing-

Tổng hợp. Danh sách sau đây cho thấy Đặt hànglớp học.

**Liệt kê 6.3 Phiên bản Eventuate củaĐặt hànglớp học**

lớp công khai Order mở rộng ReflectiveMutableCommandProcessingAggregate<Order, OrderCommand> {

công khai List<Event> process(lệnh CreateOrderCommand) { ... } công khai void apply(sự kiện OrderCreatedEvent) { ... }

...

}

Hai tham số kiểu được truyền tớiReflectiveMutableCommandProcessingAggregate

làĐặt hàngVàLệnhCommand, là giao diện cơ sở choĐặt hànglệnh của 's.

**DĐỊNH NGHĨA TỔNG HỢPLỆNH**

Các lớp lệnh của tổng hợp phải mở rộng giao diện cơ sở dành riêng cho tổng hợp, giao diện này phải mở rộng giao diện Command. Ví dụ, các lệnh của tổng hợp Order mở rộng OrderCommand:

giao diện công cộng OrderCommand mở rộng Command {

}

lớp công khai CreateOrderCommand triển khai OrderCommand { ... }

CácLệnhCommandgiao diện mở rộngYêu cầuvàTạo lệnh đặt hànglớp lệnh mở rộngLệnhCommand.

**DXÁC ĐỊNH CÁC SỰ KIỆN MIỀN**

Các lớp sự kiện của một tổng hợp phải mở rộng giao diện Sự kiện, là một giao diện đánh dấu không có phương thức. Cũng hữu ích khi định nghĩa một giao diện cơ sở chung, mở rộng Sự kiện cho tất cả các lớp sự kiện của một tổng hợp. Ví dụ, đây là định nghĩa của sự kiện OrderCreated:

giao diện OrderEvent mở rộng Event {

}

lớp công khai OrderCreated mở rộng OrderEvent { ... }

CácĐơn hàngĐã tạolớp sự kiện mở rộngSự kiện đặt hàng, là giao diện cơ sở cho

Đặt hàngcác lớp sự kiện tổng hợp.Sự kiện đặt hànggiao diện mở rộngSự kiện.

**CĐỌC,TÌM KIẾM,VÀ CẬP NHẬT TỔNG HỢP VỚIMỘTTỔNG HỢPRLỚP HỌC KÝ ỨC**

Khung cung cấp một số cách để tạo, tìm và cập nhật tổng hợp. Sim-Cách tiếp cận tốt nhất mà tôi mô tả ở đây là sử dụng AggregateRepository. Aggregate- Repository là một lớp chung được tham số hóa bởi lớp tổng hợp và lớp lệnh cơ sở của tổng hợp. Nó cung cấp ba phương thức quá tải:

* cứu()—Tạo ra một tổng hợp
* tìm thấy()—Tìm một tổng hợp
* cập nhật()—Cập nhật tổng hợp

Các phương thức save() và update() đặc biệt tiện lợi vìchúng đóng gói mã mẫu cần thiết để tạo và cập nhật tổng hợp. Ví dụ, save() lấy một đối tượng lệnh làm tham số và thực hiện các bước sau:

**1** Khởi tạo tổng hợp bằng cách sử dụng hàm tạo mặc định của nó

**2** Triệu hồiquá trình()để xử lý lệnh

**3** Áp dụng các sự kiện được tạo ra bằng cách gọi apply()

**4** Lưu các sự kiện được tạo trong kho sự kiện

Phương thức update() cũng tương tự. Nó có hai tham số, một ID tổng hợp và một lệnh, và thực hiện các bước sau:

**1** Lấy tổng hợp từ kho sự kiện

**2** Triệu hồiquá trình()để xử lý lệnh

**3** Áp dụng các sự kiện được tạo ra bằng cách gọi apply()

**4** Lưu các sự kiện được tạo trong kho sự kiện

CácKho lưu trữ tổng hợplớp chủ yếu được sử dụng bởi các dịch vụ, tạo và cập nhật các tập hợp để đáp ứng các yêu cầu bên ngoài. Ví dụ, danh sách sau đây cho thấy cáchDịch vụ đặt hàngsử dụng mộtKho lưu trữ tổng hợpđể tạo ra mộtĐặt hàng.

**Liệt kê 6.4 Dịch vụ đặt hàngsử dụng mộtKho lưu trữ tổng hợp**

lớp công khai OrderService{

riêng tư AggregateRepository<Order, OrderCommand> orderRepository;

công khai OrderService(AggregateRepository<Order, OrderCommand> orderRepository)

{

this.orderRepository = orderRepository;

}

công khai EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {trả về orderRepository.save(new CreateOrder(orderDetails));

}

}

Dịch vụ đặt hàngđược tiêm mộtKho lưu trữ tổng hợpvìĐơn hàng. Của nótạo nên()

phương pháp gọiLưu trữ tổng hợp ()với mộtTạo đơn hàngyêu cầu.

**SĐĂNG KÝ VÀO DOMAIN EVENTS**

Khung Eventuate Client cũng cung cấp API để viết trình xử lý sự kiện. Liệt kê 6.5 hiển thị trình xử lý sự kiện cho các sự kiện CreditReserved. Chú thích @EventSubscriber chỉ định ID của đăng ký bền vững. Các sự kiện được xuất bản khi người đăng ký không chạy sẽ được phân phối khi nó khởi động. @EventHandler-Chú thích phương thức xác định phương thức creditReserved() là trình xử lý sự kiện.

**Liệt kê 6.5 Một trình xử lý sự kiện choSự kiện OrderCreated**

@EventSubscriber(id="orderServiceEventHandlers")lớp công khai OrderServiceEventHandlers {

@EventHandlerPhương pháp

public void creditReserved(EventHandlerContext<CreditReserved> ctx) { Sự kiện CreditReserved = ctx.getEvent();

...

}

Trình xử lý sự kiện có tham số kiểu EventHandlerContext, chứa sự kiện và siêu dữ liệu của sự kiện đó.

Bây giờ chúng ta đã xem cách viết logic kinh doanh dựa trên nguồn sự kiện bằng cách sử dụng khuôn khổ máy khách Eventuate, hãy cùng xem cách sử dụng logic kinh doanh dựa trên nguồn sự kiện với sagas.

#### Sử dụng saga và sự kiện cùng nhau

Hãy tưởng tượng bạn đã triển khai một hoặc nhiều dịch vụ bằng cách sử dụng nguồn sự kiện. Bạn có thể đã viết các dịch vụ tương tự như dịch vụ được hiển thị trong danh sách 6.4. Nhưng nếu bạn đã đọc chương 4, bạn sẽ biết rằng các dịch vụ thường cần khởi tạo và tham gia vào saga, chuỗi các giao dịch cục bộ được sử dụng để duy trì tính nhất quán của dữ liệu trên các dịch vụ. Ví dụ: Order Service sử dụng saga để xác thực một Order. Kitchen Service, Consumer Service,và Dịch vụ Kế toán tham gia vào câu chuyện đó. Do đó, bạn phải tích hợp các câu chuyện và logic kinh doanh dựa trên sự kiện.

Nguồn sự kiện giúp dễ dàng sử dụng saga dựa trên biên đạo. Những người tham gia trao đổi các sự kiện miền do tổng hợp của họ phát ra. Tổng hợp của mỗi người tham gia xử lý các sự kiện bằng cách xử lý lệnh và phát ra các sự kiện mới. Bạn cần viết các tổng hợp và các lớp trình xử lý sự kiện, cập nhật các tổng hợp.

Nhưng việc tích hợp logic kinh doanh dựa trên sự kiện với saga dựa trên dàn dựng có thể khó khăn hơn. Đó là vì khái niệm giao dịch của cửa hàng sự kiện có thể khá hạn chế. Khi sử dụng một số cửa hàng sự kiện, ứng dụng chỉ có thể tạo hoặc cập nhật một tổng hợp duy nhất và xuất bản sự kiện kết quả. Nhưng mỗi bước của saga bao gồm một số hành động phải được thực hiện một cách nguyên tử:

* *Sáng tạo Saga*—Một dịch vụ khởi tạo một saga phải tạo hoặc cập nhật một tổng hợp và tạo ra trình điều phối saga. Ví dụ,Dịch vụ đặt hàng'Stạo đơn hàng()phương pháp phải tạo ra mộtĐặt hàngtổng hợp và mộtTạo đơn hàngSaga.
* *Dàn dựng Saga*—Người điều phối saga phải tiếp nhận các phản hồi một cách nguyên tử, cập nhật trạng thái của nó và gửi tin nhắn lệnh.
* *Những người tham gia Saga*—Những người tham gia Saga, chẳng hạn nhưDịch vụ nhà bếpVàDịch vụ đặt hàng, phải tiêu thụ tin nhắn một cách nguyên tử, phát hiện và loại bỏ các tin nhắn trùng lặp, tạo hoặc cập nhật tổng hợp và gửi tin nhắn trả lời.

Do sự không phù hợp giữa các yêu cầu này và khả năng giao dịch của kho sự kiện, việc tích hợp các câu chuyện dựa trên dàn dựng và nguồn sự kiện có thể tạo ra một số thách thức thú vị.

Một yếu tố quan trọng trong việc xác định mức độ dễ dàng tích hợp nguồn sự kiện và điều phối-saga dựa trên là liệu kho sự kiện có sử dụng RDBMS hay cơ sở dữ liệu NoSQL hay không. Khung saga Eventuate Tram được mô tả trong chương 4 và khung nhắn tin Tram cơ bản được mô tả trong chương 3 dựa trên các giao dịch ACID linh hoạt do RDBMS cung cấp. Người điều phối saga và những người tham gia saga sử dụng các giao dịch ACID để cập nhật cơ sở dữ liệu và trao đổi tin nhắn của họ một cách nguyên tử. Nếu ứng dụng sử dụng kho sự kiện dựa trên RDBMS, chẳng hạn như Eventuate Local, thì nó có thể gian lận và gọi

Khung saga Eventuate Tram và cập nhật kho sự kiện trong giao dịch ACID. Nhưng nếu kho sự kiện sử dụng cơ sở dữ liệu NoSQL, không thể tham gia vào cùng một giao dịch như khung saga Eventuate Tram, thì nó sẽ phải áp dụng một cách tiếp cận khác.

Hãy cùng xem xét kỹ hơn một số tình huống và vấn đề khác nhau mà bạn cần giải quyết:

* Thực hiện các saga dựa trên vũ đạo
* Tạo một dàn nhạc dựa trêntruyện dài
* Triển khai một người tham gia saga dựa trên nguồn sự kiện
* Triển khai các trình điều phối saga bằng cách sử dụng nguồn sự kiện

Chúng ta sẽ bắt đầu bằng cách xem xét cách triển khai các câu chuyện dựa trên vũ đạo bằng cách sử dụng nguồn sự kiện.

###### Thực hiện các saga dựa trên vũ đạo bằng cách sử dụng nguồn sự kiện

Bản chất hướng sự kiện của nguồn sự kiện làm cho việc triển khai các saga dựa trên biên đạo trở nên khá đơn giản. Khi một tổng hợp được cập nhật, nó sẽ phát ra một sự kiện. Một trình xử lý sự kiện cho một tổng hợp khác có thể sử dụng sự kiện đó và cập nhật tổng hợp của nó. Khung nguồn sự kiện tự động làm cho mỗi trình xử lý sự kiện trở nên bất biến.

Ví dụ, chương 4 thảo luận về cách thực hiệnTạo đơn hàng Sagasử dụng vũ đạo.Dịch vụ tiêu dùng,Dịch vụ nhà bếp, VàDịch vụ kế toánđăng ký đểDịch vụ đặt hàngsự kiện và ngược lại. Mỗi dịch vụ có một trình xử lý sự kiện tương tự như trình xử lý được hiển thị trong danh sách 6.5. Trình xử lý sự kiện cập nhật aggre-gate tương ứng, phát ra một sự kiện khác.

Nguồn sự kiện và saga dựa trên biên đạo phối hợp rất tốt với nhau. Nguồn sự kiện cung cấp các cơ chế mà saga cần, bao gồm IPC dựa trên tin nhắn, loại bỏ trùng lặp tin nhắn và cập nhật nguyên tử trạng thái và gửi tin nhắn. Mặc dù đơn giản, saga dựa trên biên đạo có một số nhược điểm. Tôi sẽ nói về một số nhược điểm trong chương 4, nhưng có một nhược điểm cụ thể đối với nguồn sự kiện.

Vấn đề với việc sử dụng sự kiện cho biên đạo saga là các sự kiện hiện có mục đích kép. Nguồn sự kiện sử dụng sự kiện để biểu diễn các thay đổi trạng thái, nhưng sử dụng sự kiện cho biên đạo saga yêu cầu tổng hợp để phát ra sự kiện ngay cả khi không có thay đổi trạng thái. Ví dụ, nếu việc cập nhật tổng hợp sẽ vi phạm quy tắc kinh doanh, thì tổng hợp phải phát ra sự kiện để báo cáo lỗi. Một vấn đề thậm chí còn tệ hơn là khi người tham gia saga không thể tạo tổng hợp. Không có tổng hợp nào có thể phát ra sự kiện lỗi.

Do những vấn đề như thế này, tốt nhất là triển khai các saga phức tạp hơn bằng cách sử dụng orchestration. Các phần sau đây giải thích cách tích hợp các saga dựa trên orchestration và event sourcing. Như bạn sẽ thấy, nó liên quan đến việc giải quyết một số vấn đề thú vị.

Trước tiên chúng ta hãy xem cách thức một phương thức dịch vụ như thế nàoOrderService.createOrder()tạo ra một người dàn dựng câu chuyện.

###### Tạo ra một câu chuyện dựa trên dàn nhạc

Saga orchestrators được tạo ra bởi một số phương thức dịch vụ. Các phương thức dịch vụ khác, chẳng hạn như OrderService.createOrder(), thực hiện hai việc: tạo hoặc cập nhật tổng hợp và tạo saga orchestrator. Dịch vụ phải thực hiện cả hai hành động theo cách đảm bảo rằng nếu thực hiện hành động đầu tiên, thì hành động thứ hai cuối cùng sẽ được thực hiện. Cách dịch vụ đảm bảo rằng cả hai hành động này được thực hiện phụ thuộc vào loại kho sự kiện mà nó sử dụng.

**CĐỌC MỘT SAGA ORCHESTRATOR KHI SỬ DỤNGRDBMS-CỬA HÀNG SỰ KIỆN DỰA TRÊN**

Nếu một dịch vụ sử dụng RDBMS dựa trêncửa hàng sự kiện, nó có thể cập nhật cửa hàng sự kiện và tạo ra một bộ điều phối saga trong cùng một giao dịch ACID. Ví dụ, hãy tưởng tượng rằngOrderService sử dụng Eventuate Local và khung saga Eventuate Tram. Phương thức createOrder() của nó sẽ trông như thế này:

lớp OrderService

@Autowired

**Đảm bảo createOrder() được thực thi trong giao dịch cơ sở dữ liệu.**

SagaManager riêng tư<CreateOrderSagaState> createOrderSagaManager;

@Giao dịch

công khai EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {EntityWithIdAndVersion<Đặt hàng> đặt hàng =

}

...

orderRepository.save(mới CreateOrder(orderDetails));

Dữ liệu CreateOrderSagaState =

mới CreateOrderSagaState(order.getId(), orderDetails);

createOrderSagaManager.create(dữ liệu, Order.class, order.getId());trả lại đơn hàng;

**Tạo đơn hàng**

**tổng hợp.**

**Tạo CreateOrderSaga.**

Đây là sự kết hợp của OrderService trong danh sách 6.4 và OrderService được mô tả trong chương 4. Vì Eventuate Local sử dụng RDBMS, nên nó có thể tham gia vào cùng một giao dịch ACID như khung Eventuate Tram saga. Nhưng nếu một dịch vụ sử dụng kho lưu trữ sự kiện dựa trên NoSQL, thì việc tạo một saga orchestrator không đơn giản như vậy.

**CĐỌC MỘT SAGA ORCHESTRATOR KHI SỬ DỤNGNỒSQL-CỬA HÀNG SỰ KIỆN DỰA TRÊN**

Một dịch vụ sử dụng kho sự kiện dựa trên NoSQL rất có thể sẽ không thể cập nhật kho sự kiện và tạo ra một saga orchestrator một cách nguyên tử. Khung dàn dựng saga có thể sử dụng một cơ sở dữ liệu hoàn toàn khác. Ngay cả khi sử dụng cùng một cơ sở dữ liệu NoSQL, ứng dụng sẽ không thể tạo hoặc cập nhật hai đối tượng khác nhau một cách nguyên tử do mô hình giao dịch hạn chế của cơ sở dữ liệu NoSQL. Thay vào đó, một dịch vụ phải có trình xử lý sự kiện tạo ra saga orchestrator để phản hồi sự kiện miền do tổng hợp phát ra.

Vì ví dụ, nhân vật 6.11 chương trình Làm sao Dịch vụ đặt hàngtạo ra Một Tạo đơn hàngSaga

sử dụng trình xử lý sự kiện choĐơn hàngĐã tạosự kiện.Dịch vụ đặt hàngđầu tiên tạo ra một

Tổng hợp đơn hàng và lưu trữ trong kho sự kiện. Kho sự kiện xuất bản Đơn hàng-Sự kiện Created, được xử lý bởi trình xử lý sự kiện. Trình xử lý sự kiện gọi khung saga Eventuate Tram để tạo CreateOrderSaga.

Dịch vụ đặt hàng



Trình xử lý sự kiện OrderCreated

Đặt hàng

**Lưu trữ sự kiện OrderCreated.**

Đã kiên trìBẰNG

Đơn hàngĐã tạo

Tạo đơn hàngSaga

**Tạo CreateOrderSaga để phản hồi sự kiện OrderCreated.**

Cửa hàng sự kiện

Đơn hàngĐã tạo

**Hình 6.11 Sử dụng trình xử lý sự kiện để tạo một saga đáng tin cậy sau khi một dịch vụ tạo ra một sự kiệntổng hợp dựa trên nguồn cung ứng**

Một vấn đề cần lưu ý khi viết trình xử lý sự kiện tạo ra saga orchestrator là nó phải xử lý các sự kiện trùng lặp. Việc phân phối tin nhắn ít nhất một lần có nghĩa là trình xử lý sự kiện tạo ra saga có thể được gọi nhiều lần. Điều quan trọng là phải đảm bảo rằng chỉ có một phiên bản saga được tạo.

Một cách tiếp cận đơn giản là lấy ID của saga từ một thuộc tính duy nhất của sự kiện. Có một vài lựa chọn khác nhau. Một là sử dụng ID của aggregate phát ra sự kiện làm ID của saga. Cách này hiệu quả đối với saga được tạo ra để phản hồi các sự kiện tạo tổng hợp.

Một lựa chọn khác là sử dụng ID sự kiện làm ID saga. Vì ID sự kiện là duy nhất, điều này sẽ đảm bảo rằng ID saga là duy nhất. Nếu một sự kiện trùng lặp, nỗ lực tạo saga của trình xử lý sự kiện sẽ thất bại vì ID đã tồn tại. Tùy chọn này hữu ích khi nhiều phiên bản của cùng một saga có thể tồn tại cho một phiên bản tổng hợp nhất định.

Một dịch vụ sử dụng kho lưu trữ sự kiện dựa trên RDBMS cũng có thể sử dụng cùng một phương pháp tiếp cận theo sự kiện để tạo sagas. Một lợi ích của phương pháp tiếp cận này là nó thúc đẩy sự kết hợp lỏng lẻovì các dịch vụ như OrderService không còn khởi tạo saga một cách rõ ràng nữa.

Bây giờ chúng ta đã xem xét cách tạo ra một trình điều phối saga đáng tin cậy, hãy cùng xem cách

các dịch vụ dựa trên nguồn sự kiện có thể tham gia vào các câu chuyện dựa trên dàn dựng.

###### Triển khai một người tham gia saga dựa trên nguồn sự kiện

Hãy tưởng tượng rằng bạn đã sử dụng event sourcing để triển khai một dịch vụ cần tham gia vào một saga dựa trên orchestration. Không có gì ngạc nhiên, nếu dịch vụ của bạn sử dụng một kho lưu trữ sự kiện dựa trên RDBMS như Eventuate Local, bạn có thể dễ dàng đảm bảo rằng nó xử lý nguyên tử các thông báo lệnh saga và gửi phản hồi. Nó có thể cập nhật kho lưu trữ sự kiện như một phần của giao dịch ACID do khung Eventuate Tram khởi tạo. Nhưng bạn phải sử dụng một cách tiếp cận hoàn toàn khác nếu dịch vụ của bạn sử dụng một kho lưu trữ sự kiện không thể tham gia vào cùng một giao dịch như khung Eventuate Tram.

Bạn phải giải quyết một số vấn đề khác nhau:

* + - * Xử lý tin nhắn lệnh bất biến
      * Gửi tin nhắn trả lời một cách nguyên tử

Trước tiên chúng ta hãy xem cách triển khai trình xử lý tin nhắn lệnh bất biến.

**TÔIXỬ LÝ TIN NHẮN LỆNH DEMPOTENT**

Vấn đề đầu tiên cần giải quyết là làm thế nào một người tham gia saga dựa trên sự kiện có thể phát hiện và loại bỏ các tin nhắn trùng lặp để triển khai xử lý tin nhắn lệnh idempotent. May mắn thay, đây là một vấn đề dễ giải quyết bằng cách sử dụng cơ chế xử lý tin nhắn idempotent được mô tả trước đó. Một người tham gia saga ghi lại ID tin nhắn trong các sự kiện được tạo khi xử lý tin nhắn. Trước khi cập nhật tổng hợp, người tham gia saga xác minh rằng họ chưa xử lý tin nhắn trước đó bằng cách tìm ID tin nhắn trong các sự kiện.

**MỘTTOMICALLY GỬI TIN NHẮN TRẢ LỜI**

Vấn đề thứ hai cần giải quyết là làm thế nào một người tham gia saga dựa trên nguồn sự kiện có thể gửi phản hồi một cách nguyên tử. Về nguyên tắc, một saga orchestrator có thể đăng ký các sự kiện do một tổng hợp phát ra, nhưng có hai vấn đề với cách tiếp cận này. Đầu tiên là lệnh saga có thể không thực sự thay đổi trạng thái của một tổng hợp. Trong kịch bản này, tổng hợp sẽ không phát ra sự kiện, vì vậy sẽ không có phản hồi nào được gửi đến saga orchestrator. Vấn đề thứ hai là cách tiếp cận này yêu cầu saga orchestrator phải xử lý những người tham gia saga sử dụng nguồn sự kiện khác với những người không sử dụng. Đó là bởi vì để nhận được các sự kiện miền, saga orchestrator phải đăng ký kênh sự kiện của tổng hợp ngoài kênh phản hồi của riêng nó.

Một cách tiếp cận tốt hơn là người tham gia saga tiếp tục gửi tin nhắn trả lời đến kênh trả lời của người điều phối saga. Nhưng thay vì gửi tin nhắn trả lời trực tiếp, người tham gia saga sử dụng quy trình hai bước:

**1**Khi trình xử lý lệnh saga tạo hoặc cập nhật một tổng hợp, nó sẽ sắp xếp chosự kiện giả SagaReplyRequested được lưu trong kho sự kiện cùng với các sự kiện thực tế do tổng hợp phát ra.

**2**Trình xử lý sự kiện cho sự kiện giả SagaReplyRequested sử dụng dữ liệu có trong sự kiện để xây dựng thông điệp trả lời, sau đó ghi vào kênh trả lời của trình điều phối saga.

Hãy xem một ví dụ để thấy cách thức hoạt động của nó.

**EVÍ DỤTÌM NGUỒN SỰ KIỆN-NGƯỜI THAM GIA SAGA DỰA TRÊN**

Ví dụ này xem xétDịch vụ kế toán, mộtcủa những người tham giaTạo đơn hàng Saga. Hình 6.12 cho thấy cáchDịch vụ kế toánxử lýLệnh ủy quyềnđược gửi bởi saga.Dịch vụ kế toánđược triển khai bằng cách sử dụng khung Eventuate Saga. Khung Eventuate Saga là một khung nguồn mở để viết saga sử dụng nguồn sự kiện. Nó được xây dựng trên khung Eventuate Client.

**Gửi lệnh đến dịch vụ kế toán.**



Dịch vụ kế toán

Tài khoản

Dịch vụ đặt hàng

Ủy quyền

yêu cầu

ủy quyền()

Tạo nên

đặt hàngtruyện dài

Tài khoản

kênh lệnh

Ủy quyền

hồi đáp

Tạo đơn hàng saga

kênh trả lời

SagaTrả lời

yêu cầu EventHandler

SagaTrả lờiYêu cầu

API sự kiện

Người điều phối sự kiện

**Xử lý sự kiện yêu cầu SagaReply và gửi phản hồi.**

Tài khoảnĐược ủy quyền

SagaTrả lờiYêu cầu

**Emit SagaSự kiện được yêu cầu trả lời.**

Cửa hàng sự kiện

Khung truyện Eventuate

Truyện dàingười điều phối lệnh

Cho phép trình xử lý lệnh tài khoản

Kho lưu trữ tổng hợp

**Cho phép tài khoản.**



|  |
| --- |
| Tài khoản đã tạo |
| .... |
| Tài khoảnĐược ủy quyền |

**Hình 6.12 Cách thức tìm nguồn sự kiện dựa trênDịch vụ kế toántham gia vàoTạo nênĐặt hàng Saga**

Hình này cho thấy cáchTạo đơn hàng SagaVàDịch vụ kế toántương tác. CácTrình tự các sự kiện như sau:

**1**Tạo đơn hàng Sagagửi mộtỦy quyền tài khoảnlệnh choKế toán-Dịch vụthông qua kênh nhắn tin. Khung Eventuate SagaSagaCommand- Người điều phốigọiKế toánDịch vụCommandHandlerđể xử lý lệnh tin nhắn.

**2** Kế toánDịch vụCommandHandlergửi lệnh đến địa chỉ đã chỉ định

Tài khoảntổng hợp.

**3** Tổng hợp phát rahai sự kiện,Tài khoảnĐược ủy quyềnVàSagaReplyRequested-Sự kiện.

**4**SagaReplyRequestedEventHandlertay cầmSự kiện SagaReplyRequestedbằng cách gửi-gửi tin nhắn trả lờiTạo đơn hàngSaga.

CácKế toánDịch vụCommandHandlerđược hiển thị trong danh sách sau đây xử lýỦy quyền tài khoảnlệnh tin nhắn bằng cách gọiAggregateRepository.cập nhật()ĐẾNcập nhậtTài khoảntổng hợp.

**Liệt kê 6.6 Xử lý các tin nhắn lệnh được gửi bởitruyện dài**

lớp công khai AccountingServiceCommandHandler {

@Autowired

private AggregateRepository<Tài khoản, AccountCommand> accountRepository;

public void authorize(CommandMessage<AuthorizeCommand> cm) { AuthorizeCommand lệnh = cm.getCommand(); accountRepository.update(command.getOrderId(),

lệnh, trả lời(cm)

.catching(AccountDisabledException. lớp,

() -> withFailure(tài khoản mớiDisabledReply()))

.xây dựng());

}

...

Cácủy quyền()phương pháp gọi mộtKho lưu trữ tổng hợpđể cập nhậtTài khoảntổng hợp. Lập luận thứ ba chocập nhật(), đó làCập nhậtTùy chọn, được tính bằng biểu thức này:

trả lời(cm)

.catching(AccountDisabledException. lớp,

() -> withFailure(tài khoản mớiDisabledReply()))

.xây dựng()

Những cái nàyCập nhậtTùy chọncấu hìnhcập nhật()phương pháp thực hiện như sau:

**1**Sử dụng id tin nhắn làm khóa idempotency để đảm bảo tin nhắn được xử lý đúng một lần. Như đã đề cập trước đó, khung Eventuate lưu trữ khóa idempotency trong tất cả các sự kiện được tạo, cho phép nó phát hiện và bỏ qua các nỗ lực trùng lặp để cập nhật tổng hợp.

**2** Thêm vàoMộtSự kiện SagaReplyRequestedsự kiện giả vào danh sách các sự kiện được lưu trong kho sự kiện. KhiSagaReplyRequestedEventHandlernhận đượcSagaReply- RequestedEventsự kiện giả, nó gửi một phản hồi tớiTạo đơn hàngSagakênh trả lời của.

**3**Gửi mộtTài khoản đã bị vô hiệu hóaTrả lờithay vì trả lời lỗi mặc định khi tổng hợp đưa ra mộtTài khoản bị vô hiệu hóa.

Bây giờ chúng ta đã xem cách triển khai những người tham gia saga bằng cách sử dụng nguồn sự kiện, hãy cùng tìm hiểu cách triển khai trình điều phối saga.

###### Triển khai các trình điều phối saga bằng cách sử dụng nguồn sự kiện

Cho đến nay trong phần này, tôi đã mô tả cách các dịch vụ dựa trên nguồn sự kiện có thể khởi tạo và tham gia vào saga. Bạn cũng có thể sử dụng nguồn sự kiện để triển khai các nhà tổ chức saga. Điều này sẽ cho phép bạn phát triển các ứng dụng hoàn toàn dựa trên một kho sự kiện.

Có ba vấn đề thiết kế chính mà bạn phải giải quyết khi triển khai trình biên soạn saga:

**1** Làm sao bạn có thể duy trì vai trò là người dàn dựng câu chuyện?

**2**Làm thế nào bạn có thể thay đổi trạng thái của bộ điều phối và gửi tin nhắn lệnh?

**3**Làm sao bạn có thể đảm bảo rằng trình biên soạn saga xử lý tin nhắn trả lời đúng một lần?

Chương 4 thảo luận về cách triển khai một trình điều phối saga dựa trên RDBMS. Hãyxem cách giải quyết những vấn đề này khi sử dụng nguồn sự kiện.

**PSỬ DỤNG SỰ KIỆN SOURCING**

Một saga orchestrator có vòng đời rất đơn giản. Đầu tiên, nó được tạo ra. Sau đó, nó được cập nhật để phản hồi lại các phản hồi từ những người tham gia saga. Do đó, chúng ta có thể duy trì một saga bằng cách sử dụng các sự kiện sau:

* + - * SagaOrchestratorĐã tạo—Người biên soạn câu chuyện đã được tạo ra.
      * SagaOrchestratorĐã cập nhật—Người biên soạn truyện đã được cập nhật.

Một người dàn dựng saga phát ra mộtSagaOrchestratorĐã tạosự kiện khi nó được tạo ra và mộtSagaOrchestratorĐã cập nhậtsự kiện khi nó đã được cập nhật. Các sự kiện này chứa dữ liệu cần thiết để tạo lại trạng thái của saga orchestrator. Ví dụ, các sự kiện choTạo đơn hàngSaga, được mô tả trong chương 4, sẽ chứa một chuỗi được tuần tự hóa (ví dụ: JSON)Tạo đơn hàng SagaState.

**SKẾT THÚC TIN NHẮN LỆNH MỘT CÁCH ĐÁNG TIN CẬY**

Một vấn đề thiết kế quan trọng khác là làm thế nào để cập nhật trạng thái của saga và gửi lệnh một cách nguyên tử. Như đã mô tả trong chương 4, việc triển khai saga dựa trên Eventuate Tram thực hiện điều này bằng cách cập nhật bộ điều phối và chèn thông báo lệnh vào bảng thông báo như một phần của cùng một giao dịch. Một ứng dụng sử dụng

Kho lưu trữ sự kiện dựa trên RDBMS, chẳng hạn như Eventuate Local, có thể sử dụng cùng một cách tiếp cận. Một ứng dụng sử dụng kho lưu trữ sự kiện dựa trên NoSQL, chẳng hạn như Eventuate SaaS, có thể sử dụng cách tiếp cận tương tự, mặc dù có mô hình giao dịch rất hạn chế.

Bí quyết là phải duy trì SagaCommandEvent, biểu thị lệnh cần gửi.Sau đó, trình xử lý sự kiện sẽ đăng ký SagaCommandEvents và gửi từng thông báo lệnh đến kênh thích hợp. Hình 6.13 cho thấy cách thức hoạt động của nó.

Dịch vụTin nhắn người môi giới

Gửi lệnh

Trình xử lý sự kiện SagaCommand

«saga» Tạo đơn hàngSaga

Kênh lệnh dịch vụ

1. **Phát ra SagaCommandEventcho mỗi lệnh được gửi.**

Đã tồn tại như

 Sự kiện SagaCommand

1. **Xử lý SagaCommandEvent bằng cách gửi lệnh.**

Cửa hàng sự kiện

|  |
| --- |
| Sự kiện SagaCreated |
| Sự kiện SagaCommand |
| Sự kiện SagaUpdated |
| Sự kiện SagaCommand |

**Hình 6.13 Cách một người điều phối saga dựa trên nguồn sự kiện gửi lệnh đến những người tham gia saga**

Người dàn dựng saga sử dụng mộtquy trình hai bước để gửi lệnh:

**1**Saga orchestrator phát ra SagaCommandEvent cho mỗi lệnh mà nó muốn gửi. SagaCommandEvent chứa tất cả dữ liệu cần thiết để gửi lệnh, chẳng hạn như kênh đích và đối tượng lệnh. Các sự kiện này được lưu trong kho lưu trữ sự kiện.

**2**Trình xử lý sự kiện xử lý các SagaCommandEvent này và gửi tin nhắn lệnh đến kênh tin nhắn đích.

Phương pháp hai bước này đảm bảo rằng lệnh sẽ được gửi ít nhất một lần.

Bởi vì kho sự kiện cung cấp giao hàng ít nhất một lần, trình xử lý sự kiện có thể được gọi nhiều lần với cùng một sự kiện. Điều đó sẽ khiến trình xử lý sự kiện choSagaCommandEvents để gửi tin nhắn lệnh trùng lặp. Tuy nhiên, may mắn thay, người tham gia saga có thể dễ dàng phát hiện và loại bỏ các lệnh trùng lặp bằng cách sử dụng sau

cơ chế. ID của SagaCommandEvent, được đảm bảo là duy nhất, được sử dụng làm ID của tin nhắn lệnh. Do đó, các tin nhắn trùng lặp sẽ có cùng ID. Người tham gia saga nhận được tin nhắn lệnh trùng lặp sẽ loại bỏ tin nhắn đó bằng cơ chế được mô tả trước đó.

**PROCESSING TRẢ LỜI ĐÚNG MỘT LẦN**

Người điều phối saga cũng cần phát hiện và loại bỏ các tin nhắn trả lời trùng lặp,mà nó có thể thực hiện bằng cách sử dụng cơ chế được mô tả trước đó. Bộ điều phối lưu trữ ID của tin nhắn trả lời trong các sự kiện mà nó phát ra khi xử lý tin nhắn trả lời. Sau đó, nó có thể dễ dàng xác định xem tin nhắn có phải là bản sao hay không.

Như bạn có thể thấy, event sourcing là nền tảng tốt để triển khai sagas. Đây là lợi ích bổ sung cho các lợi ích khác của event sourcing, bao gồm khả năng tạo sự kiện đáng tin cậy bất cứ khi nào dữ liệu thay đổi, ghi nhật ký kiểm tra đáng tin cậy và khả năng thực hiện các truy vấn tạm thời. Tuy nhiên, event sourcing không phải là giải pháp hoàn hảo. Nó liên quan đến một đường cong học tập đáng kể. Việc phát triển lược đồ sự kiện không phải lúc nào cũng đơn giản. Nhưng bất chấp những nhược điểm này, event sourcing có vai trò chính trong kiến ​​trúc vi dịch vụ. Trong chương tiếp theo, chúng ta sẽ chuyển hướng và xem xét cách giải quyết một thách thức quản lý dữ liệu phân tán khác trong kiến ​​trúc vi dịch vụ: truy vấn. Tôi sẽ mô tả cách triển khai các truy vấn truy xuất dữ liệu nằm rải rác trên nhiều dịch vụ.

#### Bản tóm tắt

* Event sourcing lưu trữ một tổng hợp dưới dạng một chuỗi các sự kiện. Mỗi sự kiện đại diện cho việc tạo tổng hợp hoặc thay đổi trạng thái. Một ứng dụng tạo lại trạng thái của tổng hợp bằng cách phát lại các sự kiện. Event sourcing lưu giữ lịch sử của một đối tượng miền, cung cấp nhật ký kiểm tra chính xác và xuất bản các sự kiện miền một cách đáng tin cậy.
* Ảnh chụp nhanh cải thiện hiệu suất bằng cách giảm số lượng sự kiện phải phát lại.
* Sự kiện được lưu trữ trong một kho sự kiện, một sự kết hợp giữa cơ sở dữ liệu và một nhà môi giới tin nhắn. Khi một dịch vụ lưu một sự kiện trong kho sự kiện, nó sẽ phân phối sự kiện đó cho người đăng ký.
* Eventuate Local là một kho sự kiện nguồn mở dựa trên MySQL và Apache Kafka. Các nhà phát triển sử dụng khung máy khách Eventuate để viết các tập hợp và trình xử lý sự kiện.
* Một thách thức khi sử dụng nguồn sự kiện là xử lý sự tiến hóa của các sự kiện. Một ứng dụng có khả năng phải xử lý nhiều phiên bản sự kiện khi phát lại các sự kiện. Một giải pháp tốt là sử dụng upcasting, nâng cấp các sự kiện lên phiên bản mới nhất khi chúng được tải từ kho sự kiện.
* Việc xóa dữ liệu trong ứng dụng tìm nguồn sự kiện rất khó khăn. Một ứng dụngphải sử dụng các kỹ thuật như mã hóa và ẩn danh để tuân thủ các quy định như GDPR của Liên minh Châu Âu, yêu cầu phải có ứng dụng để xóa dữ liệu của cá nhân.

***Bản tóm tắt* 219**

* + Nguồn sự kiện là một cách đơn giản để triển khai các saga dựa trên biên đạo. Các dịch vụ có trình xử lý sự kiện lắng nghe các sự kiện được tổng hợp dựa trên nguồn sự kiện công bố.
  + Nguồn sự kiện là một cách tốt để triển khai saga orchestrators. Do đó, bạn có thể viết các ứng dụng chỉ sử dụng một kho sự kiện.

*Triển khai truy vấn trong kiến ​​trúc dịch vụ vi mô*

***Chương này bao gồm***

* Những thách thức của việc truy vấn dữ liệu trong một dịch vụ vi môngành kiến ​​​​trúc
* Khi nào và làm thế nào để thực hiện truy vấnsử dụng mẫu thành phần API
* Khi nào và làm thế nào để thực hiện truy vấnsử dụng mô hình phân tách trách nhiệm truy vấn lệnh (CQRS)

Mary và nhóm của cô ấy mới bắt đầu cảm thấy thoải máivới ý tưởng sử dụng sagas để duy trì tính nhất quán của dữ liệu. Sau đó, họ phát hiện ra rằng quản lý giao dịch không phải là thách thức duy nhất liên quan đến dữ liệu phân tán mà họ phải lo lắng khi di chuyển ứng dụng FTGO sang các dịch vụ vi mô. Họ cũng phải tìm ra cách triển khai các truy vấn.

Để hỗ trợ UI, ứng dụng FTGO triển khai nhiều hoạt động truy vấn khác nhau. Việc triển khai các truy vấn này trong ứng dụng đơn khối hiện có tương đối đơn giản, vì nó có một cơ sở dữ liệu duy nhất. Phần lớn, tất cả những gì các nhà phát triển FTGO cần làm là viết các câu lệnh SQL SELECT và xác định các chỉ mục cần thiết. Như Mary đã phát hiện ra, việc viết các truy vấn trong kiến ​​trúc dịch vụ vi mô là một thách thức. Các truy vấn thường cần phải truy xuất dữ liệu bị phân tán

**220**

giữa các cơ sở dữ liệu do nhiều dịch vụ sở hữu. Tuy nhiên, bạn không thể sử dụng cơ chế truy vấn phân tán truyền thống, vì ngay cả khi khả thi về mặt kỹ thuật, nó vẫn vi phạm tính đóng gói.

Ví dụ, hãy xem xét các hoạt động truy vấn cho ứng dụng FTGO được mô tả trong chương 2. Một số truy vấn lấy dữ liệu chỉ thuộc sở hữu của một dịch vụ. Việc tìm kiếmVí dụ, truy vấn ConsumerProfile() trả về dữ liệu từ Consumer Service. Nhưng các hoạt động truy vấn FTGO khác, chẳng hạn như findOrder() và findOrderHistory(), trả về dữ liệu thuộc sở hữu của nhiều dịch vụ. Việc triển khai các hoạt động truy vấn này không đơn giản như vậy.

Có hai mô hình khác nhau để triển khai các hoạt động truy vấn trong kiến ​​trúc dịch vụ vi mô:

* + *Mẫu thành phần API*—Đây là cách tiếp cận đơn giản nhất và nên được sử dụng bất cứ khi nào có thể. Nó hoạt động bằng cách khiến các máy khách của dịch vụ sở hữu dữ liệu chịu trách nhiệm gọi các dịch vụ và kết hợp các kết quả.
  + *Mẫu phân tách trách nhiệm truy vấn lệnh (CQRS)*—Mô hình này mạnh hơn mô hình thành phần API, nhưng cũng phức tạp hơn. Nó duy trì một hoặc nhiều cơ sở dữ liệu dạng xem có mục đích duy nhất là hỗ trợ truy vấn.

Sau khi thảo luận về hai mẫu này, tôi sẽ nói về cách thiết kế chế độ xem CQRS, tiếp theo là triển khai chế độ xem ví dụ. Chúng ta hãy bắt đầu bằng cách xem xét mẫu thành phần API.

#### Truy vấn bằng cách sử dụng mẫu thành phần API

Ứng dụng FTGO triển khai nhiều hoạt động truy vấn. Một số truy vấn, như đã đề cập trước đó, truy xuất dữ liệu từ một dịch vụ duy nhất. Việc triển khai các truy vấn này thường đơn giản—mặc dù sau này trong chương này, khi tôi trình bày về mẫu CQRS, bạn sẽ thấy các ví dụ về các truy vấn dịch vụ đơn lẻ khó triển khai.

Ngoài ra còn có các truy vấn lấy dữ liệu từ nhiều dịch vụ. Trong phần này, tôi mô tả hoạt động truy vấn findOrder(), đây là ví dụ về truy vấn lấy dữ liệu từ nhiều dịch vụ. Tôi giải thích những thách thức thường nảy sinh khi triển khai loại truy vấn này trong kiến ​​trúc dịch vụ vi mô. Sau đó, tôi mô tả mẫu thành phần API và chỉ ra cách bạn có thể sử dụng nó để triển khai các truy vấn như findOrder().

###### Hoạt động truy vấn findOrder()

Hoạt động findOrder() lấy một đơn hàng theo khóa chính của nó. Nó lấy orderId làm tham số và trả về một đối tượng OrderDetails, chứa thông tin về đơn hàng. Như thể hiện trong hình 7.1, hoạt động này được gọi bởi một mô-đun giao diện, chẳng hạn như thiết bị di động hoặc ứng dụng web, triển khai chế độ xem Trạng thái đơn hàng.

Thông tin hiển thị bởi chế độ xem Trạng thái đơn hàng bao gồm thông tin cơ bản về đơn hàng, bao gồm trạng thái của đơn hàng, trạng thái thanh toán, trạng thái của đơn hàng từ

**Dữ liệu từ nhiềudịch vụDi động thiết bị hoặc ứng dụng web**



Chi tiết đơn hàng findOrder(orderId)

Ứng dụng FTGO

Trạng thái đơn hàng

Giao diện FTGO

Dịch vụ kế toán

Hóa đơn

số hiệu:343-45611

Mã đơn hàng:3492-2323tình trạng: ĐÃ THANH TOÁN

Dịch vụ giao hàng

Vận chuyển

số hiệu:45-4545

Mã đơn hàng:3492-2323trạng thái:ĐANG LÊN ĐƯỜNG dự kiến:6:25 chiều

Dịch vụ nhà bếp

Vé

số hiệu:3492-2323

trạng thái:ĐÃ CHUẨN BỊ

Dịch vụ đặt hàng

Đặt hàng

số hiệu:3492-2323

nhà hàng:Ajanta

Xem trạng thái đơn hàng

Mã đơn hàng: 3492-2323 Nhà hàng: Ajanta

Tình trạng: Đang trên đường ETA: 6:25 chiều

Thanh toán: Đã thanh toán

**Hình 7.1Cáctìm kiếmOrder()hoạt động được gọi bởi một giao diện FTGOmô-đun và trả vềchi tiết của mộtĐặt hàng.**

quan điểm của nhà hàng và tình trạng giao hàng, bao gồm vị trí và thời gian giao hàng ước tính nếu đang vận chuyển.

Vì dữ liệu của nó nằm trong một cơ sở dữ liệu duy nhất, ứng dụng FTGO nguyên khối có thể dễ dàng truy xuất thông tin chi tiết về đơn hàng bằng cách thực thi một câu lệnh SELECT duy nhất nối các bảng khác nhau. Ngược lại, trong phiên bản dựa trên dịch vụ vi mô của ứng dụng FTGO, dữ liệu được phân tán xung quanh các dịch vụ sau:

* Dịch vụ đặt hàng—Thông tin đơn hàng cơ bản, bao gồm các chi tiếtvà trạng thái
* Dịch vụ nhà bếp—Tình trạng đơn hàng từ nhà hànggóc nhìn và thời gian ước tính nó sẽ sẵn sàng để nhận
* Dịch vụ giao hàng—Trạng thái giao hàng của đơn hàng, thông tin giao hàng ước tính và vị trí hiện tại của đơn hàng
* Dịch vụ kế toán—Trạng thái thanh toán của đơn hàng

Bất kỳ khách hàng nào cần thông tin chi tiết về đơn hàng đều phải hỏi tất cả các dịch vụ này.

###### Tổng quan về mô hình thành phần API

Một cách để thực hiện các hoạt động truy vấn, chẳng hạn như findOrder(),rằng việc truy xuất dữ liệu do nhiều dịch vụ sở hữu là sử dụng mô hình thành phần API. Mô hình này triển khai một

hoạt động truy vấn bằng cách gọi các dịch vụ sở hữu dữ liệu và kết hợp các kết quả. Hình 7.2 cho thấy cấu trúc của mô hình này. Nó có hai loại người tham gia:

* + - * *Một nhà soạn thảo API*—Điều này thực hiện hoạt động truy vấn bằng cách truy vấn các dịch vụ của nhà cung cấp.
      * *Một dịch vụ nhà cung cấp*—Đây là dịch vụ sở hữu một số dữ liệu mà truy vấn trả về.

**Thực hiện thao tác truy vấn bằng cách gọi các nhà cung cấp và kết hợp các kết quả.**



Nhà soạn nhạc API

truy vấn()

**Các dịch vụ sở hữu dữ liệu**

truy vấnA()

truy vấnB()

truy vấnC()

Nhà cung cấp dịch vụ A

Nhà cung cấp dịch vụ B

Nhà cung cấp dịch vụ C

Cơ sở dữ liệu A

Cơ sở dữ liệu B

Cơ sở dữ liệu C

**Hình 7.2 Mẫu thành phần API bao gồm một API composer và hai hoặc nhiều dịch vụ nhà cung cấp. API composer triển khai truy vấn bằng cách truy vấn các nhà cung cấp và kết hợp các kết quả.**

Hình 7.2 hiển thị ba nhà cung cấpdịch vụ. API composer triển khai truy vấn bằng cách lấy dữ liệu từ các dịch vụ của nhà cung cấp và kết hợp các kết quả. API composer có thể là một máy khách, chẳng hạn như ứng dụng web, cần dữ liệu để hiển thị trang web. Ngoài ra, nó có thể là một dịch vụ, chẳng hạn như cổng API và biến thể Backends cho frontends được mô tả trong chương 8, hiển thị thao tác truy vấn dưới dạng điểm cuối API.

**Mẫu: Thành phần API**

Triển khai truy vấn lấy dữ liệu từ nhiều dịch vụ bằng cách truy vấn từng dịch vụ thông qua API của dịch vụ đó và kết hợp các kết quả. Xem[http://microservices.io/patterns/data/api-](http://microservices.io/patterns/data/api-composition.html) [thành phần.html](http://microservices.io/patterns/data/api-composition.html).

Việc bạn có thể sử dụng mẫu này để triển khai một hoạt động truy vấn cụ thể hay không phụ thuộc vào một số yếu tố, bao gồm cách phân vùng dữ liệu, khả năng của các API do các dịch vụ sở hữu dữ liệu cung cấp và khả năng của các cơ sở dữ liệu do các dịch vụ sử dụng. Ví dụ, ngay cả khi các dịch vụ Nhà cung cấp có API để truy xuất

dữ liệu bắt buộc, trình tổng hợp có thể cần thực hiện một phép nối trong bộ nhớ không hiệu quả của các tập dữ liệu lớn. Sau đó, bạn sẽ thấy các ví dụ về các hoạt động truy vấn không thể triển khai bằng cách sử dụng mẫu này. Tuy nhiên, may mắn thay, có nhiều trường hợp mà mẫu này có thể áp dụng. Để xem nó hoạt động như thế nào, chúng ta sẽ xem một ví dụ.

###### Triển khai hoạt động truy vấn findOrder() bằng APImẫu thành phần

Hoạt động truy vấn findOrder() tương ứng với mộttruy vấn equi-join dựa trên khóa chính. Có thể mong đợi rằng mỗi dịch vụ Nhà cung cấp đều có điểm cuối API để truy xuất dữ liệu cần thiết theo orderId. Do đó, hoạt động truy vấn findOrder() là ứng cử viên tuyệt vời để được triển khai theo mẫu thành phần API. API composer gọi bốn dịch vụ và kết hợp các kết quả lại với nhau.Hình 7.3 cho thấy thiết kế của TìmTrình soạn đơn hàng.



NHẬN/đặt hàng/{orderId}

NHẬN/đơn hàng/

{mã đơn hàng}

NHẬN/vé?Mã đơn hàng =

{mã đơn hàng}

NHẬN/giao hàng?Mã đơn hàng =

{mã đơn hàng}

NHẬN/phí?Mã đơn hàng =

{mã đơn hàng}

Dịch vụ đặt hàng

Dịch vụ nhà bếp

Dịch vụ giao hàng

Dịch vụ kế toán

Tìm đơn hàngNhà soạn nhạc

«tổng hợp»Thù lao

«tổng hợp»Vận chuyển

«tổng hợp» RestaurantOrder

«tổng hợp»Đặt hàng

**Hình 7.3 Thực hiệntìm kiếmOrder()sử dụng mẫu thành phần API**

Trong ví dụ này, API composer là một dịch vụ hiển thị truy vấn dưới dạng điểm cuối REST. Các dịch vụ Provider cũng triển khai REST API. Nhưng khái niệm này giống nhau nếu các dịch vụ sử dụng một số giao thức truyền thông liên tiến trình khác, chẳng hạn như gRPC, thay vì HTTP. Find Order Composer triển khai một điểm cuối REST GET /order/{orderId}. Nó gọi bốn dịch vụ và kết hợp các phản hồi bằng orderId. Mỗi dịch vụ Provider triển khai một điểm cuối REST trả về phản hồi tương ứng với mộttổng hợp. OrderService lấy phiên bản Đơn hàng của mình theo khóa chính và các dịch vụ khác sử dụng orderId làm khóa ngoại để lấy tổng hợp của chúng.

Như bạn thấy, mẫu thành phần API khá đơn giản. Hãy cùng xem xét một số vấn đề thiết kế mà bạn phải giải quyết khi áp dụng mẫu này.

###### Các vấn đề thiết kế thành phần API

Khikhi sử dụng mẫu này, bạn phải giải quyết một số vấn đề về thiết kế:

* + - * Quyết định thành phần nào trong kiến ​​trúc của bạn là trình soạn thảo API của hoạt động truy vấn
      * Cách viết logic tổng hợp hiệu quả Hãy cùng xem xét từng vấn đề.

**THO ĐÓNG VAI TRÒ CỦAGiao diện lập trình ứng dụng (API)NHẠC SĨ?**

Một quyết định mà bạn phải đưa ra là ai sẽ đóng vai trò là API composer của hoạt động truy vấn. Bạn có ba lựa chọn. Lựa chọn đầu tiên, được hiển thị trong hình 7.4, là để một máy khách của các dịch vụ trở thành API composer.

**Hình 7.4 Triển khai thành phần API trong máy khách. Máy khách truy vấn các dịch vụ của nhà cung cấp để lấy dữ liệu.**

Khách hàng, chẳng hạn như ứng dụng web

Vận chuyển

Dịch vụ

Dịch vụ kế toán

Dịch vụ nhà bếp

Dịch vụ đặt hàng

Nhà soạn nhạc API

Một máy khách giao diện như một ứng dụng web, thực hiệnTrạng thái đơn hàng và đang chạy trên cùng một mạng LAN, có thể truy xuất hiệu quả thông tin chi tiết về đơn hàng bằng cách sử dụng mẫu này. Nhưng như bạn sẽ tìm hiểu trong chương 8, tùy chọn này có thể không thực tế đối với các máy khách nằm ngoài tường lửa và truy cập dịch vụ qua mạng chậm hơn.

Tùy chọn thứ hai, được hiển thị trong hình 7.5, dành cho cổng API, nơi triển khai API bên ngoài của ứng dụng, đóng vai trò là trình biên soạn API cho hoạt động truy vấn.

Tùy chọn này có ý nghĩa nếu thao tác truy vấn là một phần của API bên ngoài của ứng dụng. Thay vì định tuyến yêu cầu đến một dịch vụ khác, API Gateway triển khai logic thành phần API. Cách tiếp cận này cho phép máy khách, chẳng hạn như thiết bị di động, chạy bên ngoài tường lửa truy xuất dữ liệu hiệu quả từ nhiều dịch vụ chỉ bằng một lệnh gọi API. Tôi thảo luận về API Gateway trong chương 8.

Lựa chọn thứ ba, được thể hiện trong hình 7.6, là triển khai trình soạn thảo API như một dịch vụ độc lập.

|  |  |
| --- | --- |
| Khách hàng bên ngoài, chẳng hạn như ứng dụng di động | |
|  | tìm kiếmOrder() |

**Hình 7.5Triển khai thành phần API trong cổng API. API truy vấn các dịch vụ của nhà cung cấp để lấy dữ liệu, kết hợp các kết quả và trả về phản hồi cho máy khách.**

Cổng API

Vận chuyển

Dịch vụ

Dịch vụ kế toán

Dịch vụ nhà bếp

Dịch vụ đặt hàng

Nhà soạn nhạc API

**Hình 7.6 Triển khai một hoạt động truy vấn được sử dụng bởi nhiềukhách hàng và dịch vụ như một dịch vụ độc lập.**

tìm kiếmOrder()

Tìm dịch vụ đặt hàng

Vận chuyển

Dịch vụ

Khách hàng

Dịch vụ kế toán

Dịch vụ nhà bếp

Dịch vụ đặt hàng

Nhà soạn nhạc API

Bạn nên sử dụng tùy chọn này cho một hoạt động truy vấnđược sử dụng nội bộ bởi nhiều dịch vụ. Hoạt động này cũng có thể được sử dụng cho các hoạt động truy vấn có thể truy cập bên ngoài mà logic tổng hợp của chúng quá phức tạp để trở thành một phần của cổng API.

**Giao diện lập trình ứng dụng (API)NHẠC SĨ NÊN SỬ DỤNG MÔ HÌNH LẬP TRÌNH PHẢN ỨNG**

Khi phát triển một hệ thống phân tán, việc giảm thiểu độ trễ luôn là mối quan tâm thường trực. Bất cứ khi nào có thể, một API composer nên gọi các dịch vụ nhà cung cấp song song để giảm thiểu thời gian phản hồi cho một hoạt động truy vấn. Ví dụ, Find Order Aggregator nên gọi bốn dịch vụ đồng thời vì không có sự phụ thuộc nào giữa các cuộc gọi. Tuy nhiên, đôi khi, một API composer cần kết quả của một dịch vụ Nhà cung cấp để gọi một dịch vụ khác. Trong trường hợp này, nó sẽ cần phải gọi một số dịch vụ nhà cung cấp—nhưng hy vọng là không phải tất cả—theo trình tự.

Logic để thực hiện hiệu quả hỗn hợp các lệnh gọi dịch vụ tuần tự và song song có thể phức tạp. Để một API composer có thể bảo trì cũng như có hiệu suất và khả năng mở rộng, nó phải sử dụng thiết kế phản ứng dựa trên Java Completable-Future, RxJava observable hoặc một số tương đương khác trừu tượng. Tôi sẽ thảo luận thêm về chủ đề này trong chương 8 khi tôi trình bày về mô hình cổng API.

###### Lợi ích và hạn chế của mô hình thành phần API

Mẫu này là một cách đơn giản và trực quan để triển khai các hoạt động truy vấn trong kiến ​​trúc dịch vụ vi mô. Nhưng nó có một số nhược điểm:

* + - * Tăng chi phí chung
      * Nguy cơ giảm khả năng cung cấp
      * Thiếu sự nhất quán trong dữ liệu giao dịch Chúng ta hãy cùng xem xét chúng.

**TÔITĂNG CHI PHÍ ...**

Một nhược điểm của mô hình này là chi phí phải gọi nhiều dịch vụ và truy vấn nhiều cơ sở dữ liệu. Trong một ứng dụng đơn khối, máy khách có thể truy xuất dữ liệu bằng một yêu cầu duy nhất, thường sẽ thực hiện một truy vấn cơ sở dữ liệu duy nhất. So sánh, sử dụng mô hình thành phần API liên quan đến nhiều yêu cầu và truy vấn cơ sở dữ liệu. Do đó, cần nhiều tài nguyên mạng và tính toán hơn, làm tăng chi phí chạy ứng dụng.

**RISK GIẢM GIÁSỰ CÓ SẴN**

Một nhược điểm khác của mô hình này là tính khả dụng giảm. Như đã mô tả trong chương 3, tính khả dụng của một hoạt động giảm theo số lượng dịch vụ liên quan. Vì việc triển khai một hoạt động truy vấn liên quan đến ít nhất ba dịch vụ—API composer và ít nhất hai dịch vụ nhà cung cấp—nên tính khả dụng của nó sẽ ít hơn đáng kể so với một dịch vụ duy nhất. Ví dụ, nếu tính khả dụng của một dịch vụ riêng lẻ là 99,5%, thì tính khả dụng của điểm cuối findOrder(), gọi bốn dịch vụ nhà cung cấp, là 99,5%(4+1) = 97,5%!

Có một số chiến lược bạn có thể sử dụng để cải thiện tính khả dụng. Chiến lược đầu tiên là để API composer trả về dữ liệu đã lưu trong bộ nhớ đệm trước đó khi dịch vụ Nhà cung cấp

không khả dụng. Đôi khi, API composer lưu trữ dữ liệu được trả về bởi dịch vụ Nhà cung cấp để cải thiện hiệu suất. Nó cũng có thể sử dụng bộ nhớ đệm này để cải thiện tính khả dụng. Nếu nhà cung cấp không khả dụng, API composer có thể trả về dữ liệu từ bộ nhớ đệm, mặc dù dữ liệu có thể đã cũ.

Một chiến lược khác để cải thiện tính khả dụng là trình soạn thảo API trả về incom-dữ liệu đầy đủ. Ví dụ, hãy tưởng tượng rằng Kitchen Service tạm thời không khả dụng. API Composer cho hoạt động truy vấn findOrder() có thể bỏ qua dữ liệu của dịch vụ đó khỏi phản hồi, vì UI vẫn có thể hiển thị thông tin hữu ích. Bạn sẽ thấy thêm chi tiết về thiết kế API, bộ nhớ đệm và độ tin cậy trong chương 8.

**LXÁC NHẬN SỰ NHẤT QUÁN CỦA DỮ LIỆU GIAO DỊCH**

Một nhược điểm khác của mô hình thành phần API là thiếu tính nhất quán của dữ liệu. Một ứng dụng đơn khối thường thực hiện một hoạt động truy vấn bằng cách sử dụng một giao dịch cơ sở dữ liệu duy nhất. Các giao dịch ACID—tuân theo các điều khoản chi tiết về mức độ cô lập—đảm bảo rằng một ứng dụng có chế độ xem dữ liệu nhất quán, ngay cả khi nó thực hiện nhiều truy vấn cơ sở dữ liệu. Ngược lại, mô hình thành phần API thực hiện nhiều truy vấn cơ sở dữ liệu trên nhiều cơ sở dữ liệu. Do đó, có nguy cơ là một hoạt động truy vấn sẽ trả về dữ liệu không nhất quán.

Ví dụ, mộtĐặt hànglấy lại từDịch vụ đặt hàngcó thể ở trongĐÃ HỦYtrạng thái, trong khi tương ứngVélấy lại từDịch vụ nhà bếpcó thể vẫn chưa bị hủy. API composer phải giải quyết sự khác biệt này, làm tăng độ phức tạp của mã. Tệ hơn nữa, API composer không phải lúc nào cũng có thể phát hiện dữ liệu không nhất quán và sẽ trả về cho máy khách.

Bất chấp những nhược điểm này, mẫu API composition cực kỳ hữu ích. Bạn có thể sử dụng nó để triển khai nhiều hoạt động truy vấn. Nhưng có một số hoạt động truy vấn không thể triển khai hiệu quả bằng mẫu này. Ví dụ, một hoạt động truy vấn có thể yêu cầu API composer thực hiện lệnh nối trong bộ nhớ của các tập dữ liệu lớn.

Thường thì tốt hơn là triển khai các loại hoạt động truy vấn này bằng cách sử dụng mẫu CQRS. Hãy cùng xem cách mẫu này hoạt động.

#### Sử dụng mẫu CQRS

Nhiều ứng dụng doanh nghiệp sử dụng RDBMS làm hệ thống lưu trữ giao dịch vàcơ sở dữ liệu tìm kiếm văn bản, chẳng hạn như Elasticsearch hoặc Solr, cho các truy vấn tìm kiếm văn bản. Một số ứng dụng giữ cho các cơ sở dữ liệu được đồng bộ hóa bằng cách ghi vào cả hai cùng lúc. Những ứng dụng khác sao chép dữ liệu định kỳ từ RDBMS vào công cụ tìm kiếm văn bản. Các ứng dụng có kiến ​​trúc này tận dụng thế mạnh của nhiều cơ sở dữ liệu: các thuộc tính giao dịch của RDBMS và khả năng truy vấn của cơ sở dữ liệu văn bản.

**Mẫu: Phân chia trách nhiệm truy vấn lệnh**

Triển khai truy vấn cần dữ liệu từ nhiều dịch vụ bằng cách sử dụng các sự kiện để duy trìchỉ đọcxem sao chép dữ liệu từ các dịch vụ. Xem[http://dịch vụ vi mô](http://microservices.io/patterns/data/cqrs.html)

[.io/mẫu/dữ liệu/cqrs.html](http://microservices.io/patterns/data/cqrs.html).

CQRS là một khái quát của loại kiến ​​trúc này. Nó duy trì một hoặc nhiều cơ sở dữ liệu dạng xem—không chỉ là cơ sở dữ liệu tìm kiếm văn bản—thực hiện một hoặc nhiều truy vấn của ứng dụng. Để hiểu lý do tại sao điều này hữu ích, chúng ta sẽ xem xét một số truy vấn không thể thực hiện hiệu quả bằng cách sử dụng mẫu thành phần API. Tôi sẽ giải thích cách CQRS hoạt động và sau đó nói về những lợi ích và hạn chế của CQRS. Hãy cùng xem xét khi nào bạn cần sử dụng CQRS.

###### Động lực sử dụng CQRS

Mẫu API composition là một cách tốt để triển khai nhiều truy vấn phải lấy dữ liệu từ nhiều dịch vụ. Thật không may, đây chỉ là giải pháp một phần cho vấn đề truy vấn trong kiến ​​trúc microservice. Đó là vì có nhiều truy vấn dịch vụ mà mẫu API composition không thể triển khai hiệu quả.

Hơn nữa, cũng có những truy vấn dịch vụ đơn lẻ khó triển khai. Có lẽ cơ sở dữ liệu của dịch vụ không hỗ trợ hiệu quả truy vấn. Hoặc đôi khi, việc một dịch vụ triển khai truy vấn truy xuất dữ liệu thuộc sở hữu của một dịch vụ khác là hợp lý. Hãy cùng xem xét những vấn đề này, bắt đầu bằng truy vấn đa dịch vụ không thể triển khai hiệu quả bằng cách sử dụng API composition.

**TÔITHỰC HIỆN TÌM KIẾMỒĐẠI HỌCHCÂU CHUYỆN()HOẠT ĐỘNG TRUY VẤN**

Hoạt động findOrderHistory() lấy lại lịch sử đơn hàng của người tiêu dùng. Nó có một số tham số:

* + - * người tiêu dùngId—Xác định người tiêu dùng
      * phân trang—Trang kết quả trả về
      * lọc—Tiêu chí lọc, bao gồm độ tuổi tối đa của đơn hàng cần trả lại, trạng thái đơn hàng tùy chọn và các từ khóa tùy chọn khớp với tên nhà hàng và các mục trong thực đơn

Hoạt động truy vấn này trả về một đối tượng OrderHistory có chứa tóm tắt các đơn hàng khớp được sắp xếp theo độ tuổi tăng dần. Nó được gọi bởi mô-đun triển khaichế độ xem Lịch sử đơn hàng. Chế độ xem này hiển thị tóm tắt của từng đơn hàng, bao gồm số đơn hàng, trạng thái đơn hàng, tổng đơn hàng và thời gian giao hàng ước tính.

Trên bề mặt, thao tác này tương tự như thao tác truy vấn findOrder(). Điểm khác biệt duy nhất là nó trả về nhiều đơn hàng thay vì chỉ một. Có vẻ như trình soạn thảo API chỉ phải thực hiện cùng một truy vấn đối với mỗi dịch vụ Nhà cung cấp và kết hợp các kết quả. Thật không may, nó không đơn giản như vậy.

Đó là bởi vì không phải tất cả các dịch vụ đều lưu trữ các thuộc tính được sử dụng để lọc hoặcphân loại. Ví dụ, một trong nhữngtìmLịch sử đơn hàng()tiêu chí lọc của hoạt động là một từ khóa khớp với một mục menu. Chỉ có hai dịch vụ,Dịch vụ đặt hàngVàDịch vụ nhà bếp, lưu trữ mộtĐặt hàng'các mục menu. KhôngDịch vụ giao hàngcũng khôngDịch vụ kế toánlưu trữ các mục menu, do đó không thể lọc dữ liệu của chúng bằng từ khóa này. Tương tự như vậy, khôngDịch vụ nhà bếpcũng khôngDịch vụ giao hàngcó thể sắp xếp theoNgày tạo đơn hàngthuộc tính.

Có hai cách mà một API composer có thể giải quyết vấn đề này. Một giải pháp là API composer thực hiện một phép nối trong bộ nhớ, như thể hiện trong hình 7.7. Nó lấy tất cả các đơn đặt hàng chongười tiêu dùng từDịch vụ giao hàngVàDịch vụ kế toánvà thực hiện một sự kết hợp với các lệnh được lấy từDịch vụ đặt hàngVàDịch vụ nhà bếp.



NHẬN/giao hàng?người tiêu dùngId=

NHẬN/phí?người tiêu dùngId=

Dịch vụ giao hàng

Dịch vụ kế toán

NHẬN/đặt hàng?consumerId=&từ khóa=

NHẬN/đơn hàng? consumerId=&từ khóa=

NHẬN/vé?consumerId= &từ khóa=

Dịch vụ đặt hàng

Dịch vụ nhà bếp

Tìm thấyđơn đặt hàng nhà soạn nhạc

«tổng hợp» RestaurantOrder

«tổng hợp» Đặt hàng

Phí «tổng hợp»

Giao hàng «tổng hợp»

**Các dịch vụ này không lưu trữ dữ liệu cần thiết cho tìm kiếm theo từ khóa, do đó sẽ trả về tất cả đơn đặt hàng của người tiêu dùng.**

**Hình 7.7 Thành phần API không thể truy xuất hiệu quả các đơn đặt hàng của người tiêu dùng vì một số nhà cung cấp, chẳng hạn nhưDịch vụ giao hàng, không lưu trữ các thuộc tính được sử dụng để lọc.**

Nhược điểm của cách tiếp cận này là nó có khả năng yêu cầu trình soạn thảo API phải truy xuất và kết hợp các tập dữ liệu lớn, điều này không hiệu quả.

Giải pháp khác là để trình soạn thảo API lấy các đơn hàng khớp từ OrderDịch vụ và Dịch vụ nhà bếp sau đó yêu cầu đơn hàng từ các dịch vụ khác theo ID. Nhưng điều này chỉ khả thi nếu các dịch vụ đó có API lấy hàng loạt. Yêu cầu đơn hàng riêng lẻ có thể sẽ không hiệu quả vì lưu lượng mạng quá lớn.

Các truy vấn như findOrderHistory() yêu cầu API composer sao chép chức năng của công cụ thực thi truy vấn của RDBMS. Một mặt, điều này có khả năng chuyển công việc từ cơ sở dữ liệu ít có khả năng mở rộng sang ứng dụng có khả năng mở rộng hơn. Mặt khác, nó kém hiệu quả hơn. Ngoài ra, các nhà phát triển nên viết chức năng kinh doanh, không phải là công cụ thực thi truy vấn.

Tiếp theo tôi sẽ chỉ cho bạn cách áp dụng mẫu CQRS và sử dụng kho dữ liệu riêng biệt,được thiết kế để triển khai hiệu quả hoạt động truy vấn findOrderHistory().

Nhưng trước tiên, hãy xem một ví dụ về hoạt động truy vấn khó triển khai mặc dù nó nằm trong một dịch vụ duy nhất.

**MỘTTHỬ THÁCH TRẢ LỜI DỊCH VỤ ĐƠN LẺ:TÌM THẤYMỘTCÓ SẴNRNHÀ HÀNG()**

Như bạn vừa thấy, việc triển khai các truy vấn lấy dữ liệu từ nhiều dịch vụ có thể là một thách thức. Nhưng ngay cả các truy vấn cục bộ đối với một dịch vụ duy nhất cũng có thể khó triển khai. Có một vài lý do tại sao điều này có thể xảy ra. Một là vì, như đã thảo luận ngắn gọn, đôi khi không phù hợp để dịch vụ sở hữu dữ liệu triển khai truy vấn. Lý do khác là đôi khi cơ sở dữ liệu (hoặc mô hình dữ liệu) của dịch vụ không hỗ trợ hiệu quả truy vấn.

Ví dụ, hãy xem xét thao tác truy vấn findAvailableRestaurants(). Truy vấn này tìm các nhà hàng có thể giao đến một địa chỉ nhất định tại một thời điểm nhất định. Trọng tâm của truy vấn này là tìm kiếm theo không gian địa lý (dựa trên vị trí) các nhà hàng nằm trong một khoảng cách nhất định so với địa chỉ giao hàng. Đây là một phần quan trọng của quy trình đặt hàng và được gọi bởi mô-đun UI hiển thị các nhà hàng có sẵn.

Thách thức chính khi thực hiện hoạt động truy vấn này là thực hiện hiệu quảkhoa họctruy vấn không gian địa lý. Cách bạn triển khai truy vấn findAvailableRestaurants() phụ thuộc vào khả năng của cơ sở dữ liệu lưu trữ các nhà hàng. Ví dụ, rất dễ dàng để triển khai truy vấn findAvailableRestaurants() bằng MongoDB hoặc các tiện ích mở rộng không gian địa lý Postgres và MySQL. Các cơ sở dữ liệu này hỗ trợ các kiểu dữ liệu, chỉ mục và truy vấn không gian địa lý. Khi sử dụng một trong các cơ sở dữ liệu này, Restaurant Service sẽ lưu trữ một Nhà hàng dưới dạng bản ghi cơ sở dữ liệu có thuộc tính vị trí. Nó tìm các nhà hàng khả dụng bằng truy vấn không gian địa lý được tối ưu hóa bằng chỉ mục không gian địa lý trên thuộc tính vị trí.

Nếu ứng dụng FTGO lưu trữ các nhà hàng trong một số loại cơ sở dữ liệu khác, hãy triển khai

việc đề cập đến truy vấn findAvailableRestaurant() khó khăn hơn. Nó phải duy trì bản sao dữ liệu nhà hàng ở dạng được thiết kế để hỗ trợ truy vấn không gian địa lý. Ví dụ, ứng dụng có thể sử dụng Thư viện lập chỉ mục không gian địa lý cho DynamoDB (<https://github.com/awslabs/dynamodb-geo>) sử dụng bảng như một chỉ mục không gian địa lý. Ngoài ra, ứng dụng có thể lưu trữ bản sao dữ liệu nhà hàng trong một loại cơ sở dữ liệu hoàn toàn khác, một tình huống rất giống với việc sử dụng cơ sở dữ liệu tìm kiếm văn bản cho các truy vấn văn bản.

Thách thức khi sử dụng bản sao là phải cập nhật chúng bất cứ khi nào dữ liệu gốc thay đổi. Như bạn sẽ tìm hiểu bên dưới, CQRS giải quyết vấn đề đồng bộ hóa bản sao.

**TANH ẤY CẦNĐỂ TÁCH BIỆT MỐI QUAN TÂM**

Một lý do khác khiến việc triển khai các truy vấn dịch vụ đơn lẻ trở nên khó khăn là đôi khi dịch vụ sở hữu dữ liệu không phải là dịch vụ triển khai truy vấn.Phương thức findAvailableRestaurants()hoạt động truy vấn lấy dữ liệu thuộc sở hữu của Dịch vụ nhà hàng. Dịch vụ này cho phép chủ nhà hàng quản lý hồ sơ nhà hàng và các mục menu của họ. Nó lưu trữ nhiều thuộc tính khác nhau của một nhà hàng, bao gồm tên, địa chỉ, ẩm thực, thực đơn và giờ mở cửa. Với điều kiện là dịch vụ này sở hữu

dữ liệu, ít nhất là trên bề mặt, thì việc triển khai hoạt động truy vấn này là hợp lý. Nhưng quyền sở hữu dữ liệu không phải là yếu tố duy nhất cần xem xét.

Bạn cũng phải tính đến nhu cầu tách biệt các mối quan tâm và tránh quá tải các dịch vụ với quá nhiều trách nhiệm. Ví dụ, trách nhiệm chínhcủa nhóm phát triển Dịch vụ nhà hàng là cho phép các nhà quản lý nhà hàng duy trì nhà hàng của họ. Điều đó khá khác so với việc triển khai truy vấn quan trọng, khối lượng lớn. Hơn nữa, nếu họ chịu trách nhiệm cho hoạt động truy vấn findAvailable- Restaurants(), nhóm sẽ liên tục sống trong nỗi sợ triển khai thay đổi ngăn cản người tiêu dùng đặt hàng.

Có lý khi Restaurant Service chỉ cung cấp dữ liệu nhà hàng cho một dịch vụ khác triển khai hoạt động truy vấn findAvailableRestaurants() và rất có thể thuộc sở hữu của nhóm Order Service. Cũng giống như hoạt động truy vấn findOrderHistory() và khi cần duy trì chỉ mục không gian địa lý, có yêu cầu duy trì bản sao nhất quán cuối cùng của một số dữ liệu để triển khai truy vấn. Hãy cùng xem cách thực hiện điều đó bằng CQRS.

###### Tổng quancủa CQRS

Các ví dụ được mô tả trong phần 7.2.1 đã nêu bật ba vấn đề thường gặpgặp phải khi triển khai truy vấn trong kiến ​​trúc vi dịch vụ:

* + - * Sử dụng mô hình thành phần API để truy xuất dữ liệu nằm rải rác trên nhiều dịch vụ dẫn đến việc kết hợp trong bộ nhớ tốn kém và kém hiệu quả.
      * Dịch vụ sở hữu dữ liệu lưu trữ dữ liệu trong biểu mẫu hoặc cơ sở dữ liệu không hỗ trợ hiệu quả truy vấn cần thiết.
      * Nhu cầu phân tách các mối quan tâm có nghĩa là dịch vụ sở hữu dữ liệu không phải là dịch vụ phải triển khai hoạt động truy vấn.

Giải pháp cho cả ba vấn đề này là sử dụng mô hình CQRS.

**CQRSTÁCH CÁC LỆNH KHỎI CÁC TRUY VẤN**

Phân tách trách nhiệm truy vấn lệnh, như tên gọi của nó, là tất cả về phân tách hoặc tách biệt các mối quan tâm. Như hình 7.8 cho thấy, nó chia tách một mô hình dữ liệu liên tục và các mô-đun sử dụng nó thành hai phần: phía lệnh và phía truy vấn. Các mô-đun phía lệnh và mô hình dữ liệu triển khai các hoạt động tạo, cập nhật và xóa (viết tắt là CUD—ví dụ: HTTP POST, PUT và DELETE). Các mô-đun phía truy vấn và mô hình dữ liệu triển khai các truy vấn (chẳng hạn như HTTP GET). Phía truy vấn giữ cho mô hình dữ liệu của nó được đồng bộ hóa với mô hình dữ liệu phía lệnh bằng cách đăng ký các sự kiện do phía lệnh công bố.

Cả phiên bản không phải CQRS và phiên bản CQRS của dịch vụ đều có API bao gồm nhiều hoạt động CRUD khác nhau. Trong dịch vụ không dựa trên CQRS, các hoạt động đó thường được triển khai bởi một mô hình miền được ánh xạ tới cơ sở dữ liệu. Để có hiệu suất, một số truy vấn có thể bỏ qua mô hình miền và truy cập trực tiếp vào cơ sở dữ liệu. Một mô hình dữ liệu liên tục duy nhất hỗ trợ cả lệnh và truy vấn.

Không phải CQRSCQRS



Hoạt động CRUD

Dịch vụ

CRUD

R

Cơ sở dữ liệu

Bỏ qua truy vấn

Mô hình miền

Tổng hợp

Tổng hợp



Hoạt động CRUD

Dịch vụ

CUD

R

Mô hình lệnh/miền Sự kiện Mô hình truy vấn

Truy vấn cơ sở dữ liệu

Cơ sở dữ liệu phía lệnh

Tổng hợp

Trình xử lý sự kiện

Tổng hợp

**Cơ sở dữ liệu duy nhất cho tất cả CRUD**

**Một cơ sở dữ liệu để tạo, cập nhật và xóa. Một cơ sở dữ liệu riêng cho các truy vấn. Nó được cập nhật bằng cách sử dụng các sự kiện được công bố bất cứ khi nào cơ sở dữ liệu phía lệnh thay đổi.**

**Hình 7.8 Bên trái là phiên bản không phải CQRS của dịch vụ và bên phải là phiên bản CQRS. CQRS tái cấu trúc một dịch vụ thành các mô-đun phía lệnh và phía truy vấn, có cáccơ sở dữ liệu.**

Trong dịch vụ dựa trên CQRS, mô hình miền phía lệnh xử lý các hoạt động CRUD và được ánh xạ vào cơ sở dữ liệu riêng của nó. Nó cũng có thể xử lý các truy vấn đơn giản, chẳng hạn như các truy vấn không tham gia, dựa trên khóa chính. Phía lệnh sẽ công bố các sự kiện miền bất cứ khi nào dữ liệu của nó thay đổi. Các sự kiện này có thể được công bố bằng cách sử dụng một khuôn khổ như Eventuate Tram hoặc sử dụng nguồn sự kiện.

Một mô hình truy vấn riêng xử lý các truy vấn không tầm thường. Nó đơn giản hơn nhiều so với phía lệnh vì nó không chịu trách nhiệm triển khai các quy tắc kinh doanh. Phía truy vấn sử dụng bất kỳ loại cơ sở dữ liệu nào có ý nghĩa đối với các truy vấn mà nó phải hỗ trợ. Phía truy vấn có trình xử lý sự kiện đăng ký các sự kiện miền và cập nhật cơ sở dữ liệu hoặc các cơ sở dữ liệu. Thậm chí có thể có nhiều mô hình truy vấn, một mô hình cho mỗi loại truy vấn.

**CQRSVÀ TRUY VẤN-DỊCH VỤ CHỈ**

Không chỉ có thể áp dụng CQRS trong một dịch vụ, mà bạn còn có thể sử dụng mẫu này để xác định các dịch vụ truy vấn. Một dịch vụ truy vấn có một API chỉ bao gồm các hoạt động truy vấn—không có hoạt động lệnh. Nó triển khai các hoạt động truy vấn bằng cách truy vấn một cơ sở dữ liệu mà nó luôn cập nhật bằng cách đăng ký các sự kiện do một hoặc nhiều dịch vụ khác công bố. Một dịch vụ phía truy vấn là một cách tốt để triển khai một chế độ xem được xây dựng bởi

đăng ký các sự kiện do nhiều dịch vụ công bố. Kiểu xem này không thuộc về bất kỳ dịch vụ cụ thể nào, do đó, việc triển khai nó như một dịch vụ độc lập là hợp lý. AVí dụ tốt về dịch vụ như vậy là Order History Service, đây là dịch vụ truy vấn triển khai hoạt động truy vấn findOrderHistory(). Như hình 7.9 cho thấy, dịch vụ này đăng ký các sự kiện do một số dịch vụ công bố, bao gồm Order Service, Delivery Service, v.v.

Đặt hàng sự kiện



Dịch vụ đặt hàng



Sự kiện vé

Dịch vụ nhà bếp

Trình xử lý sự kiện

tìmLịch sử đơn hàng() tìmLịch sử đơn hàng()

Vận chuyểnsự kiện



Dịch vụ giao hàng

Lịch sử đơn hàngDịch vụ

Kế toánsự kiện

Cơ sở dữ liệu xem lịch sử đơn hàng



Dịch vụ kế toán

**Hình 7.9 Thiết kế củaDịch vụ Lịch sử đơn hàng, là một dịch vụ phía truy vấn. Nó thực hiệntìmLịch sử đơn hàng()hoạt động truy vấn bằng cách truy vấn cơ sở dữ liệu, trong đóNó duy trì bằng cách đăng ký các sự kiện được nhiều dịch vụ khác nhau công bố.**

Dịch vụ Lịch sử đơn hàngcó trình xử lý sự kiện đăng ký các sự kiện được xuất bản bởi một số dịch vụ và cập nhậtLịch sử đơn hàng Xem cơ sở dữ liệu. Tôi mô tả việc triển khai dịch vụ này chi tiết hơn ở phần 7.4.

Dịch vụ truy vấn cũng là một cách tốt để triển khai chế độ xem sao chép dữ liệu do một dịch vụ duy nhất sở hữu nhưng do nhu cầu tách biệt các mối quan tâm không phải là một phần của dịch vụ đó.Ví dụ,Các nhà phát triển FTGO có thể định nghĩa một Dịch vụ Nhà hàng khả dụng, thực hiện thao tác truy vấn findAvailableRestaurants() được mô tả trước đó. Nó đăng ký các sự kiện do Dịch vụ Nhà hàng công bố và cập nhật cơ sở dữ liệu được thiết kế cho các truy vấn không gian địa lý hiệu quả.

Theo nhiều cách, CQRS là một sự kiện tổng quát hóa của cách tiếp cận phổ biến là sử dụng RDBMS làm hệ thống bản ghi và công cụ tìm kiếm văn bản, chẳng hạn như Elasticsearch, để xử lý các truy vấn văn bản. Điểm khác biệt là CQRS sử dụng phạm vi cơ sở dữ liệu rộng hơn

các loại—không chỉ là một công cụ tìm kiếm văn bản. Ngoài ra, các chế độ xem phía truy vấn CQRS được cập nhậtgần như theo thời gian thực bằng cách đăng ký sự kiện.

Bây giờ chúng ta hãy xem xét những lợi ích và hạn chế của CQRS.

###### Lợi ích của CQRS

CQRS có cả ưu điểm và nhược điểm. Các ưu điểm như sau:

* + - * Cho phép triển khai hiệu quả các truy vấn trong kiến ​​trúc dịch vụ vi mô
      * Cho phép thực hiện hiệu quả các truy vấn đa dạng
      * Cho phép truy vấn trong ứng dụng dựa trên nguồn sự kiện
      * Cải thiện việc tách biệt các mối quan tâm

**ENABLES VIỆC TRIỂN KHAI HIỆU QUẢ CÁC TRUY VẤN TRONG KIẾN TRÚC VI DỊCH VỤ**

Một lợi ích của mẫu CQRS là nó thực hiện hiệu quả các truy vấn truy xuất dữ liệu thuộc sở hữu của nhiều dịch vụ. Như đã mô tả trước đó, việc sử dụng mẫu thành phần API để thực hiện các truy vấn đôi khi dẫn đến việc kết hợp trong bộ nhớ tốn kém và không hiệu quả của các tập dữ liệu lớn. Đối với các truy vấn đó, hiệu quả hơn là sử dụng chế độ xem CQRS dễ truy vấn kết hợp trước dữ liệu từ hai hoặc nhiều dịch vụ.

**ENABLES VIỆC TRIỂN KHAI HIỆU QUẢ CÁC TRUY VẤN ĐA DẠNG**

Một lợi ích khác của CQRS là nó cho phép một ứng dụng hoặc dịch vụ triển khai hiệu quả một tập hợp các truy vấn đa dạng. Việc cố gắng hỗ trợ tất cả các truy vấn bằng một mô hình dữ liệu liên tục duy nhất thường là một thách thức và trong một số trường hợp là không thể. Một số cơ sở dữ liệu NoSQL có khả năng truy vấn rất hạn chế. Ngay cả khi một cơ sở dữ liệu có phần mở rộng để hỗ trợ một loại truy vấn cụ thể, việc sử dụng một cơ sở dữ liệu chuyên biệt thường hiệu quả hơn. Mẫu CQRS tránh được những hạn chế của một kho dữ liệu duy nhất bằng cách xác định một hoặc nhiều chế độ xem, mỗi chế độ xem triển khai hiệu quả các truy vấn cụ thể.

**ENABLES QUERYING TRONG MỘT SỰ KIỆN SOURCING-ỨNG DỤNG DỰA TRÊN**

CQRS cũng khắc phục được một hạn chế lớn của việc tìm nguồn sự kiện. Một kho sự kiện chỉ hỗ trợ các truy vấn dựa trên khóa chính. Mẫu CQRS giải quyết hạn chế này bằng cách định nghĩa một hoặc nhiều chế độ xem của các tổng hợp, được cập nhật liên tục, bằng cách đăng ký vào các luồng sự kiện được xuất bản bởi các tổng hợp dựa trên tìm nguồn sự kiện. Do đó, một ứng dụng dựa trên tìm nguồn sự kiện luôn sử dụng CQRS.

**TÔIMCẢI THIỆN SỰ TÁCH BIỆT CỦA MỐI QUAN TÂM**

Một lợi ích khác của CQRS là nó tách biệt các mối quan tâm. Một mô hình miền và mô hình dữ liệu liên tục tương ứng của nó không xử lý cả lệnh và truy vấn. Mẫu CQRS định nghĩa các mô-đun mã và lược đồ cơ sở dữ liệu riêng biệt cho phía lệnh và phía truy vấn của một dịch vụ. Bằng cách tách biệt các mối quan tâm, phía lệnh và phía truy vấn có thể đơn giản hơn và dễ bảo trì hơn.

Hơn nữa, CQRS cho phép dịch vụ triển khai truy vấn khác với dịch vụ sở hữu dữ liệu. Ví dụ, trước đó tôi đã mô tả cách mà mặc dù Dịch vụ nhà hàng sở hữu dữ liệu được truy vấn bởi hoạt động truy vấn findAvailableRestaurants, nhưng việc một dịch vụ khác triển khai một truy vấn quan trọng như vậy là hợp lý,

truy vấn khối lượng lớn. Dịch vụ truy vấn CQRS duy trì chế độ xem bằng cách đăng ký các sự kiện do dịch vụ hoặc các dịch vụ sở hữu dữ liệu công bố.

###### Những nhược điểm của CQRS

Mặc dù CQRS có nhiều lợi ích nhưng nó cũng có những nhược điểm đáng kể:

* + - * Kiến trúc phức tạp hơn
      * Xử lý độ trễ sao chép

Hãy cùng xem xét những nhược điểm này, bắt đầu với độ phức tạp ngày càng tăng.

**TôiKIẾN TRÚC PHỨC HỢP ORE**

Một nhược điểm của CQRS là nó làm tăng thêm tính phức tạp. Các nhà phát triển phải viết các dịch vụ phía truy vấn để cập nhật và truy vấn các chế độ xem. Ngoài ra còn có sự phức tạp về mặt vận hành khi quản lý và vận hành các kho dữ liệu bổ sung. Hơn nữa, một ứng dụng có thể sử dụng các loại cơ sở dữ liệu khác nhau, điều này làm tăng thêm tính phức tạp cho cả nhà phát triển và hoạt động.

**DEALING VỚI ĐỘ TRỄ SAO CHÉP**

Một nhược điểm khác của CQRS là xử lý "độ trễ" giữa chế độ xem phía lệnh và phía truy vấn. Như bạn có thể mong đợi, có độ trễ giữa thời điểm phía lệnh công bố sự kiện và thời điểm sự kiện đó được xử lý bởi phía truy vấn và chế độ xem được cập nhật. Một ứng dụng khách hàng cập nhật tổng hợp và sau đó ngay lập tức truy vấn chế độ xem có thể thấy phiên bản trước đó của tổng hợp. Nó thường phải được viết theo cách tránh phơi bày những sự không nhất quán tiềm ẩn này cho người dùng.

Một giải pháp là API phía lệnh và phía truy vấn cung cấp cho máy khách thông tin phiên bản cho phép máy khách biết rằng phía truy vấn đã lỗi thời. Máy khách có thể thăm dò chế độ xem phía truy vấn cho đến khi chế độ này được cập nhật. Tôi sẽ thảo luận về cách API dịch vụ có thể cho phép máy khách thực hiện việc này.

Một ứng dụng UI như ứng dụng di động gốc hoặc ứng dụng JavaScript trang đơn có thể xử lý độ trễ sao chép bằng cách cập nhật mô hình cục bộ của nó sau khi lệnh thành công mà không cần đưa ra truy vấn. Ví dụ, nó có thể cập nhật mô hình của mình bằng dữ liệu do lệnh trả về. Hy vọng rằng khi hành động của người dùng kích hoạt truy vấn, chế độ xem sẽ được cập nhật. Một nhược điểm của cách tiếp cận này là mã UI có thể cần phải sao chép mã phía máy chủ để cập nhật mô hình của nó.

Như bạn có thể thấy, CQRS có cả lợi ích và hạn chế. Như đã đề cậptrước đó, bạn nên sử dụng thành phần API bất cứ khi nào có thể và chỉ sử dụng CQRS khi cần thiết.

Bây giờ bạn đã thấy được những lợi ích và hạn chế của CQRS, chúng ta hãy cùng xem cách thiết kế chế độ xem CQRS.

#### Thiết kế chế độ xem CQRS

Một mô-đun xem CQRS có một API bao gồm một hoạt động truy vấn nữa. Nó thực hiện các hoạt động truy vấn này bằng cách truy vấn một cơ sở dữ liệu mà nó duy trì bằng cách đăng ký các sự kiện được một hoặc nhiều dịch vụ công bố. Như hình 7.10 cho thấy, một mô-đun xem bao gồm một cơ sở dữ liệu xem và ba mô-đun con.

tìm thấy...()



...

Sự kiện

Trình xử lý sự kiện

API truy vấn

Mô-đun xem CQRS

**Thực hiện dữ liệulogic truy cập**

cập nhật()

Truy cập dữ liệu

truy vấn()

**Hình 7.10 Thiết kế của mô-đun chế độ xem CQRS. Trình xử lý sự kiện cập nhật cơ sở dữ liệu chế độ xem, được truy vấn bởi mô-đun API truy vấn.**

Xem cơ sở dữ liệu

Mô-đun truy cập dữ liệu triển khai logic truy cập cơ sở dữ liệu. Các trình xử lý sự kiện và mô-đun API truy vấn sử dụng mô-đun truy cập dữ liệu để cập nhật và truy vấn cơ sở dữ liệu. Mô-đun trình xử lý sự kiện đăng ký các sự kiện và cập nhật cơ sở dữ liệu. Mô-đun API truy vấn triển khai API truy vấn.

Bạn phải đưa ra một số quyết định thiết kế quan trọng khi phát triển mô-đun xem:

* Bạn phải chọn cơ sở dữ liệu và thiết kế lược đồ.
* Khi thiết kế mô-đun truy cập dữ liệu, bạn phải giải quyết nhiều vấn đề khác nhau, bao gồm đảm bảo các bản cập nhật có tính bất biến và xử lý các bản cập nhật đồng thời.
* Khi triển khai chế độ xem mới trong ứng dụng hiện có hoặc thay đổi lược đồ của ứng dụng hiện có, bạn phải triển khai cơ chế để xây dựng hoặc xây dựng lại chế độ xem một cách hiệu quả.
* Bạn phải quyết định cách cho phép máy khách của chế độ xem xử lý độ trễ sao chép đã mô tả trước đó.

Chúng ta hãy cùng xem xét từng vấn đề này.

###### Chọn kho dữ liệu xem

Một quyết định thiết kế chính là lựa chọn cơ sở dữ liệu và thiết kế lược đồ. Mục đích chính của cơ sở dữ liệu và mô hình dữ liệu là triển khai hiệu quả các hoạt động truy vấn của mô-đun chế độ xem. Các đặc điểm của các truy vấn đó là yếu tố chính cần xem xét khi lựa chọn cơ sở dữ liệu. Nhưng cơ sở dữ liệu cũng phải triển khai hiệu quả các hoạt động cập nhật do trình xử lý sự kiện thực hiện.

**SQLso với. NỒSQLCƠ SỞ DỮ LIỆU**

Cách đây không lâu, có một loại cơ sở dữ liệu thống trị tất cả: RDBMS dựa trên SQL. Tuy nhiên, khi Web ngày càng phổ biến, nhiều công ty phát hiện ra rằng RDBMS không thể đáp ứng được các yêu cầu về quy mô web của họ. Điều đó dẫn đến việc tạo ra

cái gọi là cơ sở dữ liệu NoSQL. Cơ sở dữ liệu NoSQL thường có dạng giao dịch hạn chế và khả năng truy vấn ít chung hơn. Đối với một số trường hợp sử dụng nhất định, các cơ sở dữ liệu này có một số lợi thế nhất định so với cơ sở dữ liệu SQL, bao gồm mô hình dữ liệu linh hoạt hơn và hiệu suất cũng như khả năng mở rộng tốt hơn.

Cơ sở dữ liệu NoSQL thường là một lựa chọn tốt choChế độ xem CQRS, có thể tận dụng điểm mạnh và bỏ qua điểm yếu của nó. Chế độ xem CQRS được hưởng lợi từ mô hình dữ liệu phong phú hơn và hiệu suất của cơ sở dữ liệu NoSQL. Nó không bị ảnh hưởng bởi các hạn chế của cơ sở dữ liệu NoSQL, vì nó chỉ sử dụng các giao dịch đơn giản và thực hiện một tập hợp các truy vấn cố định.

Nói như vậy, đôi khi việc triển khai chế độ xem CQRS bằng cơ sở dữ liệu SQL là hợp lý. Một RDBMS hiện đại chạy trên phần cứng hiện đại có hiệu suất tuyệt vời. Các nhà phát triển, quản trị viên cơ sở dữ liệu và hoạt động CNTT nói chung quen thuộc hơn nhiều với cơ sở dữ liệu SQL so với cơ sở dữ liệu NoSQL. Như đã đề cập trước đó, cơ sở dữ liệu SQL thường có phần mở rộng cho các tính năng không quan hệ, chẳng hạn như kiểu dữ liệu không gian địa lý và truy vấn. Ngoài ra, chế độ xem CQRS có thể cần sử dụng cơ sở dữ liệu SQL để hỗ trợ công cụ báo cáo.

Như bạn có thể thấy trong bảng 7.1, có rất nhiều tùy chọn khác nhau để lựa chọn. Và để làm cho sự lựa chọn trở nên phức tạp hơn, sự khác biệt giữa các loại cơ sở dữ liệu khác nhau đang bắt đầu mờ nhạt. Ví dụ, MySQL, là một RDBMS, có hỗ trợ tuyệt vời cho JSON, đây là một trong những điểm mạnh của MongoDB, một cơ sở dữ liệu hướng tài liệu theo kiểu JSON.

**Bảng 7.1 Lưu trữ chế độ xem truy vấn**

|  |  |  |
| --- | --- | --- |
| **Nếu bạn cần** | **Sử dụng** | **Ví dụ** |
| Tra cứu đối tượng JSON dựa trên PK | Kho lưu trữ tài liệu như MongoDB hoặc DynamoDB hoặc kho lưu trữ giá trị khóa như Redis | Thực hiện lịch sử đơn hàng theo chínhtạo một tài liệu MongoDB chứa thông tin của từng khách hàng. |
| Tra cứu JSON dựa trên truy vấncác đối tượng | Một kho lưu trữ tài liệu như MongoDB hoặc DynamoDB | Triển khai chế độ xem khách hàng bằng cách sử dụngMongoDB hoặc DynamoDB. |
| Truy vấn văn bản | Một công cụ tìm kiếm văn bản như Elastic-search | Thực hiện tìm kiếm văn bản cho các đơn hàng bằng cách duy trì Elas theo đơn hàngtài liệu ticsearch. |
| Truy vấn đồ thị | Một cơ sở dữ liệu đồ thị như Neo4j | Thực hiện phát hiện gian lận bằng cách duy trì biểu đồ tùy chỉnhđơn đặt hàng, đơn đặt hàng và dữ liệu khác. |
| Báo cáo SQL truyền thống/BI | Một RDBMS | Báo cáo và phân tích kinh doanh tiêu chuẩn. |

Bây giờ tôi đã thảo luận về các loại cơ sở dữ liệu khác nhau mà bạn có thể sử dụng để triển khai chế độ xem CQRS, hãy cùng xem xét vấn đề làm thế nào để cập nhật chế độ xem hiệu quả.

**SHỖ TRỢ HOẠT ĐỘNG CẬP NHẬT**

Bên cạnh việc thực hiện hiệu quả các truy vấn, mô hình dữ liệu xem cũng phải thực hiện hiệu quả các hoạt động cập nhật được thực hiện bởi trình xử lý sự kiện. Thông thường, một sự kiện

trình xử lý sẽ cập nhật hoặc xóa một bản ghi trong cơ sở dữ liệu dạng xem bằng khóa chính của nó. Ví dụ, tôi sẽ sớm mô tả thiết kế của dạng xem CQRS cho truy vấn findOrderHistory(). Nó lưu trữ mỗi Đơn hàng dưới dạng bản ghi cơ sở dữ liệu bằng cách sử dụng orderId làm khóa chính.Khi chế độ xem này nhận được sự kiện từ Order Service, nó có thể cập nhật trực tiếp bản ghi tương ứng.

Tuy nhiên, đôi khi, nó sẽ cần cập nhật hoặc xóa một bản ghi bằng cách sử dụng khóa ngoại tương đương. Ví dụ, hãy xem xét trình xử lý sự kiện cho sự kiện Delivery\*. Nếu có sự tương ứng một-một giữa Delivery và Order,thì Delivery.id có thể giống với Order.id. Nếu đúng như vậy thì trình xử lý sự kiện Delivery\* có thể dễ dàng cập nhật bản ghi cơ sở dữ liệu của đơn hàng.

Nhưng giả sử một Vận chuyểncó khóa chính riêng hoặc có mối quan hệ một-nhiềutàu giữa mộtĐặt hàngvà mộtVận chuyển. Một sốVận chuyển\*các sự kiện, chẳng hạn nhưGiao hàng- Đã tạosự kiện, sẽ chứaMã đơn hàng. Nhưng các sự kiện khác, chẳng hạn nhưGiao hàngĐã nhậnsự kiện, có thể không. Trong trường hợp này, một trình xử lý sự kiện choGiao hàngĐã nhậnsẽ cần phải cập nhật hồ sơ đơn hàng bằng cách sử dụnggiao hàngIdtương đương với khóa ngoại.

Một số loại cơ sở dữ liệu hỗ trợ hiệu quả các hoạt động cập nhật dựa trên khóa ngoại. Ví dụ, nếu bạn đang sử dụng RDBMS hoặc MongoDB, bạn tạo một chỉ mục trên các cột cần thiết. Tuy nhiên, các bản cập nhật không dựa trên khóa chính không đơn giản khi sử dụng các cơ sở dữ liệu NOSQL khác. Ứng dụng sẽ cần duy trì một số loại ánh xạ cụ thể của cơ sở dữ liệu từ khóa ngoại sang khóa chính để xác định bản ghi nào cần cập nhật. Ví dụ, một ứng dụng sử dụng DynamoDB, chỉ hỗ trợ các bản cập nhật và xóa dựa trên khóa chính, trước tiên phải truy vấn chỉ mục phụ DynamoDB (sẽ thảo luận sau) để xác định khóa chính của các mục cần cập nhật hoặc xóa.

###### Thiết kế mô-đun truy cập dữ liệu

Trình xử lý sự kiện và mô-đun API truy vấn không truy cập trực tiếp vào kho dữ liệu. Thay vào đó, chúng sử dụng mô-đun truy cập dữ liệu, bao gồm một đối tượng truy cập dữ liệu (DAO) và các lớp trợ giúp của nó. DAO có một số trách nhiệm. Nó triển khai các hoạt động cập nhật được gọi bởi trình xử lý sự kiện và các hoạt động truy vấn được gọi bởi mô-đun truy vấn. DAO ánh xạ giữa các kiểu dữ liệu được sử dụng bởi mã cấp cao hơn và API cơ sở dữ liệu. Nó cũng phải xử lý các bản cập nhật đồng thời và đảm bảo rằng các bản cập nhật là idempotent.

Hãy cùng xem xét những vấn đề này, bắt đầu bằng cách xử lý các bản cập nhật đồng thời.

**HSỰ ĐỒNG THỜI CỦA ANDLING**

Đôi khi, DAO phải xử lý khả năng cập nhật đồng thời nhiều bản ghi cơ sở dữ liệu. Nếu một chế độ xem đăng ký các sự kiện do một loại tổng hợp duy nhất xuất bản, sẽ không có bất kỳ vấn đề đồng thời nào. Đó là vì các sự kiện do một thể hiện tổng hợp cụ thể xuất bản được xử lý tuần tự. Do đó, một bản ghi tương ứng với một thể hiện tổng hợp sẽ không được cập nhật đồng thời. Nhưng nếu một chế độ xem đăng ký các sự kiện do nhiều loại tổng hợp xuất bản, thì có thể nhiều trình xử lý sự kiện sẽ cập nhật cùng một bản ghi đồng thời.

Ví dụ, trình xử lý sự kiện cho sự kiện Order\* có thể được gọi cùng lúc với trình xử lý sự kiện cho sự kiện Delivery\* cho cùng một đơn hàng. Cả hai trình xử lý sự kiện sau đó đồng thời gọi DAO để cập nhật bản ghi cơ sở dữ liệu cho Đơn hàng đó. DAO phải được viết theo cách đảm bảo rằng tình huống này được xử lý chính xác. Nó không được phép một bản cập nhật ghi đè lên bản cập nhật khác. Nếu DAO triển khai các bản cập nhật bằng cách đọc bản ghi rồi ghi bản ghi đã cập nhật, thì nó phải sử dụng khóa bi quan hoặc khóa lạc quan. Trong phần tiếp theo, bạn sẽ thấy ví dụ về DAO xử lý các bản cập nhật đồng thời bằng cách cập nhật các bản ghi cơ sở dữ liệu mà không cần đọc chúng trước.

**TÔITRÌNH XỬ LÝ SỰ KIỆN DEMPOTENT**

Như đã đề cập trong chương 3, một trình xử lý sự kiện có thể được gọi với cùng một sự kiện nhiều hơn một lần. Nhìn chung, đây không phải là vấn đề nếu trình xử lý sự kiện phía truy vấn là idempotent. Trình xử lý sự kiện là idempotent nếu việc xử lý các sự kiện trùng lặp dẫn đến kết quả chính xác. Trong trường hợp xấu nhất, kho dữ liệu chế độ xem sẽ tạm thời lỗi thời. Ví dụ, trình xử lý sự kiện duy trì chế độ xem Lịch sử đơn hàng có thể được gọi với chuỗi sự kiện (thừa nhận là không thể xảy ra) được hiển thị trong hình 7.11: Delivery- PickedUp, DeliveryDelivered, DeliveryPickedUp và DeliveryDelivered. Sau khi phân phối các sự kiện DeliveryPickedUp và DeliveryDelivered lần đầu tiên, trình môi giới tin nhắn, có thể do lỗi mạng, bắt đầu phân phối các sự kiện từmột thời điểm trước đó, và do đó cung cấp lại DeliveryPickedUp và DeliveryDelivered.

Thời gian

Đã nhận hàng



Giao hàng đã được giao



Đã nhận hàng



**Tạm thời đã lỗi thời**

Giao hàng đã được giao



**Hình 7.11Giao hàngĐã nhậnVàGiao hàngĐã giao hàngsự kiện được giaohai lần, khiến cho trạng thái đơn hàng đang xem tạm thời lỗi thời.**

Lịch sử đơn hàngXem

OrderId: 123 Trạng thái: ĐÃ GIAO HÀNG

OrderId: 123 Trạng thái: ĐÃ NHẬN

OrderId: 123 Trạng thái: ĐÃ GIAO HÀNG

OrderId: 123 Trạng thái: ĐÃ NHẬN

Sau khi trình xử lý sự kiện xử lý sự kiện DeliveryPickedUp thứ hai, chế độ xem Lịch sử đơn hàng tạm thời chứa trạng thái lỗi thời của Đơn hàng cho đến khi Giao hàng-Đã giao được xử lý. Nếu hành vi này không mong muốn, thì trình xử lý sự kiện phải phát hiện và loại bỏ các sự kiện trùng lặp, giống như trình xử lý sự kiện không có tính bất biến.

Trình xử lý sự kiện không phải là idempotent nếu các sự kiện trùng lặp dẫn đến kết quả không chính xác. Ví dụ, trình xử lý sự kiện tăng số dư của tài khoản ngân hàng không phải là idempotent. Trình xử lý sự kiện không phải là idempotent phải, như đã giải thích trong chương 3, phát hiện và loại bỏ các sự kiện trùng lặp bằng cách ghi lại ID của các sự kiện mà nó đã xử lý trong kho dữ liệu chế độ xem.

Để đáng tin cậy, trình xử lý sự kiện phải ghi lại ID sự kiện và cập nhật kho dữ liệu một cách nguyên tử. Cách thực hiện tùy thuộc vào loại cơ sở dữ liệu. Nếu kho dữ liệu xem là cơ sở dữ liệu SQL, trình xử lý sự kiện có thể chèn các sự kiện đã xử lý vàoBảng PROCESSED\_EVENTS là một phần của giao dịch cập nhật chế độ xem. Nhưng nếu kho dữ liệu chế độ xem là cơ sở dữ liệu NoSQL có mô hình giao dịch hạn chế, trình xử lý sự kiện phải lưu sự kiện trong "bản ghi" kho dữ liệu (ví dụ: tài liệu MongoDB hoặc mục bảng DynamoDB) mà nó cập nhật.

Điều quan trọng cần lưu ý là trình xử lý sự kiện không cần ghi lại ID của mọi sự kiện. Nếu, như trường hợp của Eventuate, các sự kiện có ID tăng dần theo đơn điệu, thì mỗi bản ghi chỉ cần lưu trữ max(eventId) được nhận từ một thể hiện tổng hợp nhất định. Hơn nữa, nếu bản ghi tương ứng với một thể hiện tổng hợp duy nhất, thì trình xử lý sự kiện chỉ cần ghi lại max(eventId). Chỉ những bản ghi biểu diễn các liên kết của các sự kiện từ nhiều tổng hợp mới phải chứa một bản đồ từ [loại tổng hợp, id tổng hợp] đến max(eventId).

Ví dụ, bạn sẽ sớm thấy rằng triển khai DynamoDB của chế độ xem Lịch sử đơn hàng chứa các mục có thuộc tính để theo dõi các sự kiện trông như thế này:

{...

}

"Order3949384394-039434903": "0000015e0c6fc18f-0242ac1100e50002", "Delivery3949384394-039434903": "0000015e0c6fc264-0242ac1100e50002",

Góc nhìn này là sự kết hợp của các sự kiện được xuất bản bởi nhiều dịch vụ khác nhau. Tên của mỗi dịch vụ nàythuộc tính theo dõi sự kiện là «aggregateType»«aggregateId» và giá trị là eventId. Sau này, tôi sẽ mô tả cách thức hoạt động này chi tiết hơn.

**ENABLING MỘT ỨNG DỤNG KHÁCH HÀNG ĐỂ SỬ DỤNG MỘT CÁI NHÌN NHẤT QUÁN CUỐI CÙNG**

Như tôi đã nói trước đó, một vấn đề khi sử dụng CQRS là một máy khách cập nhật phía lệnh và sau đó thực thi ngay một truy vấn có thể không thấy bản cập nhật của chính nó. Cuối cùng, chế độ xem sẽ nhất quán do độ trễ không thể tránh khỏi của cơ sở hạ tầng nhắn tin. API mô-đun lệnh và truy vấn có thể cho phép máy khách phát hiện ra sự không nhất quán bằng cách sử dụng phương pháp sau. Một hoạt động phía lệnh trả về một mã thông báo chứa ID của sự kiện đã xuất bản cho máy khách. Sau đó, máy khách chuyển mã thông báo cho một hoạt động truy vấn, hoạt động này trả về lỗi nếu chế độ xem chưa được sự kiện đó cập nhật. Một mô-đun chế độ xem có thể triển khai cơ chế này bằng cách sử dụng sự kiện trùng lặp

cơ chế phát hiện.

###### Thêm và cập nhật chế độ xem CQRS

Các chế độ xem CQRS sẽ được thêm và cập nhật trong suốt vòng đời của ứng dụng. Đôi khi bạn cần thêm chế độ xem mới để hỗ trợ truy vấn mới. Vào những thời điểm khác, bạn có thể cần tạo lại chế độ xem vì lược đồ đã thay đổi hoặc bạn cần sửa lỗi trong mã cập nhật chế độ xem.

Việc thêm và cập nhật chế độ xem về mặt khái niệm khá đơn giản. Để tạo chế độ xem mới, bạn phát triển mô-đun phía truy vấn, thiết lập kho dữ liệu và triển khai dịch vụ. Truy vấn

trình xử lý sự kiện của mô-đun phụ xử lý tất cả các sự kiện và cuối cùng chế độ xem sẽ được cập nhật. Tương tự như vậy, việc cập nhật chế độ xem hiện có cũng đơn giản về mặt khái niệm: bạn thay đổi trình xử lý sự kiện và xây dựng lại chế độ xem từ đầu. Tuy nhiên, vấn đề là cách tiếp cận này không có khả năng hoạt động trong thực tế. Hãy cùng xem xét các vấn đề.

**BULDCQRSLƯỢT XEM SỬ DỤNG SỰ KIỆN ĐÃ LƯU TR�**

Một vấn đề là các message broker không thể lưu trữ message vô thời hạn. Các message broker truyền thống như RabbitMQ xóa message sau khi nó được consumer xử lý. Ngay cả các broker hiện đại hơn như Apache Kafka, lưu giữ message trong một khoảng thời gian lưu giữ có thể cấu hình, cũng không có ý định lưu trữ các sự kiện vô thời hạn. Do đó, không thể xây dựng một view chỉ bằng cách đọc tất cả các sự kiện cần thiết từ message broker. Thay vào đó, một ứng dụng cũng phải đọc các sự kiện cũ hơn đã được lưu trữ trong, ví dụ, AWS S3. Bạn có thể thực hiện điều này bằng cách sử dụng công nghệ dữ liệu lớn có khả năng mở rộng như Apache Spark.

**BULDCQRSLƯỢT XEM TĂNG DẦN**

Một vấn đề khác với việc tạo chế độ xem là thời gian và tài nguyên cần thiết để xử lý tất cả các sự kiện tiếp tục tăng theo thời gian. Cuối cùng, việc tạo chế độ xem sẽ trở nên quá chậm và tốn kém. Giải pháp là sử dụng thuật toán gia tăng hai bước. Bước đầu tiên định kỳ tính toán ảnh chụp nhanh của mỗi phiên bản tổng hợp dựa trên ảnh chụp nhanh trước đó và các sự kiện đã xảy ra kể từ khi ảnh chụp nhanh đó được tạo. Bước thứ hai tạo chế độ xem bằng cách sử dụng ảnh chụp nhanh và bất kỳ sự kiện nào sau đó.

#### Triển khai chế độ xem CQRS với AWS DynamoDB

Bây giờ chúng ta đã xem xét các vấn đề thiết kế khác nhau mà bạn phải giải quyết khi sử dụng CQRS, hãy xem xét một ví dụ. Phần này mô tả cách triển khai CQRSxem cho hoạt động findOrderHistory() bằng DynamoDB. AWS DynamoDB là cơ sở dữ liệu NoSQL có thể mở rộng, có sẵn dưới dạng dịch vụ trên đám mây Amazon. Mô hình dữ liệu DynamoDB bao gồm các bảng chứa các mục, giống như các đối tượng JSON, là tập hợp các cặp tên-giá trị phân cấp. AWS DynamoDB là cơ sở dữ liệu được quản lý hoàn toàn và bạn có thể tăng hoặc giảm dung lượng thông lượng của một bảng một cách linh hoạt.

Chế độ xem CQRS cho findOrderHistory() sử dụng các sự kiện từ nhiều dịch vụ, do đó, nó được triển khai như một Dịch vụ xem đơn hàng độc lập. Dịch vụ này có một API triển khai hai hoạt động: findOrderHistory() và findOrder(). Mặc dù findOrder() có thể được triển khai bằng cách sử dụng thành phần API, nhưng chế độ xem này cung cấp hoạt động này miễn phí. Hình 7.12 cho thấy thiết kế của dịch vụ. Dịch vụ lịch sử đơn hàng được cấu trúc như một tập hợp các mô-đun, mỗi mô-đun triển khai một trách nhiệm cụ thể để đơn giản hóa quá trình phát triển và thử nghiệm. Trách nhiệm của từng mô-đun như sau:

* Trình xử lý sự kiện OrderHistory—Đăng ký các sự kiện do các dịch vụ khác nhau công bố và kêu gọiLịch sử đơn hàngDAO
* API OrderHistoryQuery*mô-đun*—Triển khai các điểm cuối REST được mô tả trước đó
  + Lịch sử đơn hàngTruy cập dữ liệu—Chứa đựngLịch sử đơn hàngDAO, trong đó xác địnhphương pháp cập nhật và truy vấnftgo-order-lịch-sửBảng DynamoDB và các lớp trợ giúp của nó
  + ftgo-order-lịch-sử*Bảng DynamoDB*—Bảng lưu trữ các đơn hàng

Đặt hàng giao hàng

...

sự kiện

Trình xử lý sự kiện OrderHistory

Truy vấn OrderHistory

Dịch vụ Lịch sử đơn hàng

tìmLịch sử đơn hàng()tìm kiếmĐặt hàng

Cập nhật truy vấn

Lịch sử đơn hàngTruy cập dữ liệu

Lịch sử đơn hàngDAO

<Bảng DynamoDB> ftgo-order-history

**Hình 7.12 Thiết kế củaLịch sử đơn hàngDịch vụ.OrderHistory- EventHandlerscập nhật cơ sở dữ liệu để phản hồi các sự kiện.Truy vấn Lịch sử đơn hàngmô-đun thực hiện các hoạt động truy vấn bằng truy vấn-ing cơ sở dữ liệu. Hai mô-đun này sử dụngOrderHistory- Truy cập dữ liệumô-đun để truy cập cơ sở dữ liệu.**



Hãy cùng xem xét chi tiết hơn về thiết kế của trình xử lý sự kiện, DAO và bảng DynamoDB.

###### Mô-đun OrderHistoryEventHandlers

Mô-đun này bao gồm các trình xử lý sự kiện sử dụng các sự kiện và cập nhật bảng DynamoDB. Như danh sách sau đây cho thấy, các trình xử lý sự kiện là phương pháp đơn giảnods. Mỗi phương thức là một dòng lệnh gọi phương thức OrderHistoryDao với các đối số bắt nguồn từ sự kiện.

**Liệt kê 7.1 Trình xử lý sự kiện gọiLịch sử đơn hàngDao**

lớp công khai OrderHistoryEventHandlers { riêng OrderHistoryDao orderHistoryDao;

public OrderHistoryEventHandlers(OrderHistoryDao orderHistoryDao) { this.orderHistoryDao = orderHistoryDao;

}

công khai void handleOrderCreated(DomainEventEnvelope<OrderCreated> dee) { orderHistoryDao.addOrder(makeOrder(dee.getAggregateId(), dee.getEvent()),

makeSourceEvent(dee));

}

lệnh riêng tư makeOrder(String orderId, OrderCreatedEvent sự kiện) {

...

}

công khai vô hiệuhandleDeliveryPickedUp(DomainEventEnvelope<DeliveryPickedUp>

dee) { orderHistoryDao.notePickedUp(dee.getEvent().getOrderId(),

makeSourceEvent(dee));

}

...

Mỗi trình xử lý sự kiện có một tham số duy nhất thuộc loạiMiềnSự kiệnPhong bì, chứa sự kiện và một số siêu dữ liệu mô tả sự kiện. Ví dụ,xử lýOrderCreated()phương pháp được gọi để xử lý mộtĐơn hàngĐã tạosự kiện. Nó gọiorderHistoryDao.addOrder()để tạo ra mộtĐặt hàngtrong cơ sở dữ liệu. Tương tự như vậy,xử lýGiao hàngĐã nhận()phương pháp được gọi để xử lý mộtGiao hàngĐã nhậnsự kiện. Nó gọiorderHistoryDao.notePickedUp()để cập nhật trạng thái củaĐặt hàngtrong cơ sở dữ liệu.

Cả hai phương pháp đều gọi phương pháp trợ giúp makeSourceEvent(), phương pháp này xây dựng một SourceEvent chứa loại và ID của tổng hợp đã phát ra sự kiện vàID sự kiện. Trong phần tiếp theo, bạn sẽ thấy OrderHistoryDao sử dụng SourceEvent để đảm bảo các hoạt động cập nhật là bất biến.

Bây giờ chúng ta hãy xem xét thiết kế của bảng DynamoDB và sau đó kiểm tra

Lịch sử đơn hàngDao.

###### Mô hình hóa dữ liệu và thiết kế truy vấn với DynamoDB

Giống như nhiều cơ sở dữ liệu NoSQL, DynamoDB có các hoạt động truy cập dữ liệu kém mạnh hơn nhiều so với các hoạt động được cung cấp bởi RDBMS. Do đó, bạn phải thiết kế cẩn thận cách lưu trữ dữ liệu. Đặc biệt, các truy vấn thường quyết định thiết kế của lược đồ. Chúng ta cần giải quyết một số vấn đề về thiết kế:

* + - * Thiết kếftgo-order-lịch-sửbàn
      * Xác định một chỉ số chotìmLịch sử đơn hàngtruy vấn
        + Thực hiệncáitìmLịch sử đơn hàngtruy vấn
        + Phân trang kết quả truy vấn
        + Đang cập nhật đơn hàng
        + Phát hiện sự kiện trùng lặpChúng ta sẽ xem xét từng cái một.

**DTHIẾT KẾ FTGO-ĐẶT HÀNG-BẢNG LỊCH SỬ**

Mô hình lưu trữ DynamoDB bao gồm các bảng, chứa các mục và chỉ mục, cung cấp các cách thay thế để truy cập các mục của bảng (sẽ được thảo luận sau). Một mục là một tập hợp các thuộc tính được đặt tên. Một giá trị thuộc tính là một giá trị vô hướng như một chuỗi, một tập hợp các chuỗi đa giá trị hoặc một tập hợp các thuộc tính được đặt tên. Mặc dù một mục tương đương với một hàng trong RDBMS, nhưng nó linh hoạt hơn nhiều và có thể lưu trữ toàn bộ một tập hợp.

Tính linh hoạt này cho phép OrderHistoryDataAccessmodule để lưu trữ mỗi Order dưới dạng một mục duy nhất trong bảng DynamoDB có tên là ftgo-order-history. Mỗi trường của lớp Order được ánh xạ tới một thuộc tính item, như thể hiện trong hình 7.13. Các trường đơn giản như orderCreationTime và status được ánh xạ tới các thuộc tính item có giá trị đơn. Trường lineItems được ánh xạ tới một thuộc tính là danh sách các bản đồ, một bản đồ cho mỗi dòng thời gian. Nó có thể được coi là một mảng JSON của các đối tượng.

bảng ftgo-order-history

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Khóa chính | | |  | | | | |
|  | Mã đơn hàng |  | người tiêu dùngId | Thời gian tạo đơn hàng | trạng thái | dòng mục | ... |
|  |  |  |  |  |  | [{...}. |  |
|  |  |  | |
| ... | xyz-abc | | 22939283232 | TẠO | {...},  ....] | ... |
|  | ... | ... | | ... | ... | .... | ... |

**Hình 7.13 Cấu trúc sơ bộ của DynamoDBLịch sử đơn hàngbàn**

Một phần quan trọng của định nghĩa bảng là khóa chính của nó. Ứng dụng DynamoDB chèn, cập nhật và truy xuất các mục của bảng theo khóa chính. Có vẻ nhưcó ý nghĩa đối với khóa chính là orderId. Điều này cho phép Order History Service chèn, cập nhật và truy xuất đơn hàng theo orderId. Nhưng trước khi hoàn tất quyết định này, trước tiên chúng ta hãy khám phá cách khóa chính của bảng tác động đến các loại hoạt động truy cập dữ liệu mà nó hỗ trợ.

**DĐỊNH NGHĨA MỘT CHỈ SỐ CHO VIỆC TÌM KIẾMỒĐẠI HỌCHCÂU HỎI CÂU CHUYỆN**

Định nghĩa bảng này hỗ trợ đọc và ghi dựa trên khóa chính của Orders. Nhưng nókhông hỗ trợ truy vấn như findOrderHistory() trả về nhiều đơn hàng khớp nhau được sắp xếp theo độ tuổi tăng dần. Đó là vì, như bạn sẽ thấy sau trong phần này, truy vấn này sử dụng thao tác DynamoDB query(), yêu cầu bảng phải có

khóa chính tổng hợp bao gồm hai thuộc tính vô hướng. Thuộc tính đầu tiên là khóa phân vùng. Khóa phân vùng được gọi như vậy vì tính năng chia tỷ lệ trục Z của DynamoDB (được mô tả trong chương 1) sử dụng khóa này để chọn phân vùng lưu trữ của một mục. Thuộc tính thứ hai là khóa sắp xếp. Một hoạt động query() trả về các mục có khóa phân vùng được chỉ định, có khóa sắp xếp trong phạm vi được chỉ định và khớp với biểu thức bộ lọc tùy chọn. Nó trả về các mục theo thứ tự được chỉ định bởi khóa sắp xếp.

CáctìmLịch sử đơn hàng()hoạt động truy vấn trả về các đơn hàng của người tiêu dùng được sắp xếp theo độ tuổi tăng dần. Do đó, nó yêu cầu một khóa chính cóngười tiêu dùngIdnhư là par-chìa khóa tiêu đề vàNgày tạo đơn hàngnhư là khóa sắp xếp. Nhưng nó không có ý nghĩa đối với(consumerId, ngày tạo đơn hàng)là khóa chính củaftgo-order-lịch-sửbảng, vì nó không phải là duy nhất.

Giải pháp là dùng findOrderHistory() để truy vấn những gì DynamoDB gọi là chỉ mục phụ trên bảng ftgo-order-history. Chỉ mục này có (consumerId, orderCreation- Date) là khóa không duy nhất. Giống như chỉ mục RDBMS, chỉ mục DynamoDB được tự động cập nhật bất cứ khi nào bảng của nó được cập nhật. Nhưng không giống như chỉ mục RDBMS thông thường, chỉ mục DynamoDB có thể có các thuộc tính không phải khóa. Các thuộc tính không phải khóa cải thiện hiệu suất vì chúng được truy vấn trả về, do đó ứng dụng không phải lấy chúng từ bảng. Ngoài ra, như bạn sẽ sớm thấy, chúng có thể được sử dụng để lọc. Hình 7.14 cho thấy cấu trúc của bảng và chỉ mục này.

Chỉ số là một phần của định nghĩaftgo-order-lịch-sửbảng và được gọi là

ftgo-order-history-by-consumer-id-and-creation-time. Các thuộc tính của chỉ mục

ftgo-order-history-by-consumer-id-and-creation-time chỉ số thứ cấp toàn cầu

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Khóa chính | |  | | |
| người tiêu dùngId | Thời gian tạo đơn hàng | Mã đơn hàng | trạng thái | ... |
|  |  | cde-fgh | TẠO | ... |
| xyz-abc | 22939283232 |
| ... | ... | ... | ... | ... |

bảng ftgo-order-history

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Khóa chính | | |  | | | | |
|  | Mã đơn hàng |  | người tiêu dùngId | Thời gian tạo đơn hàng | trạng thái | dòng mục | ... |
|  |  |  |  |  |  | [{...}. |  |
|  |  |  | |
| cde-fgh | xyz-abc | | 22939283232 | TẠO | {...},  ....] | ... |
|  | ... | ... | | ... | ... | .... | ... |

**Hình 7.14 Thiết kế củaLịch sử đơn hàngbảng và chỉ mục**

bao gồmcác thuộc tính khóa chính,người tiêu dùngIdVàThời gian tạo đơn hàngvà các thuộc tính không phải khóa, bao gồmMã đơn hàngVàtrạng thái.

Các ftgo-order-history-by-consumer-id-and-creation-timechỉ số cho phépcáiLịch sử đơn hàngDaoDynamoDbđể thu thập hiệu quả các đơn hàng của người tiêu dùng được sắp xếp theo độ tuổi tăng dần.

Bây giờ chúng ta hãy xem cách chỉ lấy những đơn hàng khớp với tiêu chí lọc.

**TÔITHỰC HIỆN TÌM KIẾMỒĐẠI HỌCHCÂU HỎI CÂU CHUYỆN**

CáctìmLịch sử đơn hàng()hoạt động truy vấn có mộtlọctham số chỉ định tiêu chí tìm kiếm. Một tiêu chí lọc là độ tuổi tối đa của các đơn hàng cần trả về. Điều này dễ thực hiện vì DynamoDBTruy vấnbiểu thức điều kiện khóa của hoạt động hỗ trợ giới hạn phạm vi trên khóa sắp xếp. Các tiêu chí lọc khác tương ứng với các thuộc tính không phải khóa và có thể được triển khai bằng cách sử dụng biểu thức lọc, là biểu thức Boolean. DynamoDBTruy vấnhoạt động chỉ trả về những mục đáp ứng bộ lọc biểu thức. Ví dụ, để tìmĐơn hàngđó làĐÃ HỦY, cácOrderHistoryDao-Máy phát điệnsử dụng một truy vấnsự biểu lộTrạng thái đơn hàng = :Trạng thái đơn hàng, Ở đâu:trạng thái đơn hànglà tham số giữ chỗ.

Tiêu chí lọc từ khóa khó thực hiện hơn. Nó chọn các đơn hàng có tên nhà hàng hoặc các mục menu khớp với một trong các từ khóa được chỉ định.Lịch sử đơn hàngDaoDynamoDbcho phép tìm kiếm từ khóa bằng cách mã hóa tên nhà hàng và các mục menu và lưu trữ tập hợp các từ khóa trong một thuộc tính có giá trị được gọi làtừ khóa. Nó tìm các đơn hàng phù hợp với các từ khóa bằng cách sử dụng biểu thức bộ lọc sử dụng cái chứa()chức năng, vì ví dụ chứa(từ khóa, :từ khóa1) HOẶC chứa(từ khóa, :từ khóa2), Ở đâu:từ khóa1Và:từ khóa2là chỗ giữ chỗ cho các từ khóa đã chỉ định.

**PXEM XÉT KẾT QUẢ TRUY VẤN**

Một số người tiêu dùng sẽ có một số lượng lớn đơn đặt hàng. Do đó, điều đó có ý nghĩa đối vớiHoạt động truy vấn findOrderHistory() để sử dụng phân trang. Hoạt động truy vấn DynamoDB có tham số pageSize, chỉ định số lượng mục tối đa cần trả về. Nếu có nhiều mục hơn, kết quả của truy vấn có thuộc tính Last-EvaluatedKey không phải null. DAO có thể truy xuất trang mục tiếp theo bằng cách gọi truy vấn với tham số exclusiveStartKey được đặt thành LastEvaluatedKey.

Như bạn có thể thấy, DynamoDB không hỗ trợ phân trang theo vị trí. Hậu quả làDịch vụ Lịch sử Đơn hàng trả về một mã thông báo phân trang mờ cho máy khách của mình. Máy khách sử dụng mã thông báo phân trang này để yêu cầu trang kết quả tiếp theo.

Bây giờ tôi đã mô tả cách truy vấn DynamoDB để lấy đơn hàng, hãy cùng xem cách chèn và cập nhật chúng.

**BạnLỆNH PDATING**

DynamoDB hỗ trợ hai thao tác để thêm và cập nhật mục:Đặt mục()VàCập nhật mục(). CácĐặt mục()hoạt động tạo hoặc thay thế toàn bộ một mục bằng khóa chính của nó. Về lý thuyết,Lịch sử đơn hàngDaoDynamoDbcó thể sử dụng thao tác này để chèn

và cập nhật lệnh. Tuy nhiên, một thách thức khi sử dụng PutItem() là đảm bảo các bản cập nhật đồng thời cho cùng một mục được xử lý chính xác.

Ví dụ, hãy xem xét tình huống trong đó hai trình xử lý sự kiện đồng thờicố gắng cập nhật cùng một mục. Mỗi trình xử lý sự kiện gọi OrderHistoryDaoDynamoDb để tải mục từ DynamoDB, thay đổi mục đó trong bộ nhớ và cập nhật mục đó trong DynamoDB bằng PutItem(). Một trình xử lý sự kiện có khả năng ghi đè lên thay đổi do trình xử lý sự kiện kia thực hiện. OrderHistoryDaoDynamoDb có thể ngăn chặn các bản cập nhật bị mất bằng cách sử dụng cơ chế khóa lạc quan của DynamoDB. Nhưng một cách tiếp cận thậm chí còn đơn giản và hiệu quả hơn là sử dụng thao tác UpdateItem().

Hoạt động UpdateItem() cập nhật các thuộc tính riêng lẻ của mục, tạo mục nếu cần thiết. Vì các trình xử lý sự kiện khác nhau cập nhật các thuộc tính khác nhau củaOrder item, sử dụng UpdateItem có ý nghĩa. Hoạt động này cũng hiệu quả hơn vì không cần phải lấy order từ bảng trước.

Một thách thức khi cập nhật cơ sở dữ liệu để phản hồi các sự kiện là, như đã đề cập trước đó, phát hiện và loại bỏ các sự kiện trùng lặp. Hãy cùng xem cách thực hiện điều đó khi sử dụng DynamoDB.

**DXÓA CÁC SỰ KIỆN TRÙNG LẶP**

Tất cả các trình xử lý sự kiện của Order History Service đều là idempotent. Mỗi trình xử lý đặt một hoặc nhiều thuộc tính của mục Order. Do đó, Order History Service có thể bỏ qua vấn đề về các sự kiện trùng lặp. Tuy nhiên, nhược điểm của việc bỏ qua vấn đề này là mục Order đôi khi sẽ tạm thời lỗi thời. Đó là vì trình xử lý sự kiện nhận được sự kiện trùng lặp sẽ đặt các thuộc tính của mục Order thành các giá trị trước đó. Mục Order sẽ không có các giá trị chính xác cho đến khi các sự kiện sau được phân phối lại.

Như đã mô tả trước đó, một cách để ngăn dữ liệu trở nên lỗi thời làphát hiện và loại bỏ các sự kiện trùng lặp. OrderHistoryDaoDynamoDb có thể phát hiện các sự kiện trùng lặp bằng cách ghi lại trong mỗi mục các sự kiện khiến mục đó được cập nhật. Sau đó, nó có thể sử dụng cơ chế cập nhật có điều kiện của thao tác UpdateItem() để chỉ cập nhật một mục nếu sự kiện không phải là sự kiện trùng lặp.

Cập nhật có điều kiện chỉ được thực hiện nếu biểu thức điều kiện là đúng. Biểu thức điều kiện kiểm tra xem thuộc tính có tồn tại hay có giá trị cụ thể hay không.Đặt hàng-Lịch sửDaoDynamoDbDAO có thể theo dõi các sự kiện nhận được từ mỗi phiên bản tổng hợp bằng cách sử dụng một thuộc tính được gọi là«aggregateType» «aggregateId»của aigiá trị là ID sự kiện nhận được cao nhất. Một sự kiện là trùng lặp nếu thuộc tính tồn tại và giá trị của nó nhỏ hơn hoặc bằng ID sự kiện.Lịch sử đơn hàngDaoDynamoDbDAO sử dụng biểu thức điều kiện này:

thuộc tính\_không\_tồn\_tại(«aggregateType»«aggregateId»)HOẶC «aggregateType» «aggregateId» < :eventId

Biểu thức điều kiện chỉ cho phép cập nhật nếu thuộc tính không tồn tại hoặc

eventId lớn hơnhơn ID sự kiện được xử lý cuối cùng.

Ví dụ, giả sử một trình xử lý sự kiện nhận được mộtGiao hàngNhận hàngsự kiện có ID là123323-343434từ mộtVận chuyểntổng hợp có ID là3949384394-039434903. Tên của thuộc tính theo dõi làGiao hàng3949384394-039434903.Trình xử lý sự kiện sẽ coi sự kiện là trùng lặp nếu giá trị của thuộc tính này làlớn hơn hoặc bằng123323-343434. Cáctruy vấn()hoạt động được gọi bởi trình xử lý sự kiện cập nhậtĐặt hàngmục sử dụng biểu thức điều kiện này:

thuộc tính\_không\_tồn\_tại(Giao\_hàng3949384394-039434903)HOẶC Giao hàng3949384394-039434903 < :eventId

Bây giờ tôi đã mô tả mô hình dữ liệu DynamoDB và thiết kế truy vấn, chúng ta hãy xem xétLịch sử đơn hàngDaoDynamoDb, định nghĩa các phương pháp cập nhật và truy vấnftgo-order-lịch-sửbàn.

###### Lịch sử đơn hàngDaoDynamoDblớp học

CácLịch sử đơn hàngDaoDynamoDblớp thực hiện các phương thức đọc và ghi các mục trongftgo-order-lịch-sửbảng. Các phương pháp cập nhật của nó được gọi bởiLịch sử đơn hàng-Trình xử lý sự kiệnvà các phương thức truy vấn của nó được gọi bởiAPI OrderHistoryQuery. Chúng ta hãy xem xét một số phương pháp ví dụ, bắt đầu vớithêm đơn hàng()phương pháp.

**TANH ẤY THÊMỒĐẠI HỌC()PHƯƠNG PHÁP**

Cácthêm đơn hàng()phương pháp, được hiển thị trong danh sách 7.2, thêm một thứ tự vàoftgo-lịch sử đơn hàngbảng. Nó có hai tham số:đặt hàngVànguồnSự kiện.Cácđặt hàngtham số làĐặt hàngđể thêm vào, được lấy từĐơn hàngĐã tạosự kiện.nguồnSự kiệntham số chứasự kiệnIDvà loại và ID của tổng hợp phát ra sự kiện. Nó được sử dụng để triển khai bản cập nhật có điều kiện.

**Liệt kê 7.2thêm đơn hàng()phương pháp thêm hoặc cập nhật mộtĐặt hàng**

lớp công khai OrderHistoryDaoDynamoDb ...@Ghi đè

**Khóa chính của mục Order cần cập nhật**

public boolean addOrder(Order order, Optional<SourceEvent> eventSource) {UpdateItemSpec thông số kỹ thuật = new UpdateItemSpec()

.withPrimaryKey("orderId", order.getOrderId())

**Biểu thức cập nhật cập nhật các thuộc tính**

**Các giá trị của chỗ giữ chỗ trong biểu thức cập nhật**

.withUpdateExpression("ĐẶT orderStatus = :orderStatus, " + "creationDate = :cd, consumerId = :consumerId, lineItems =" +

" :lineItems, từ khóa= :keywords, restaurantName="+":tên nhà hàng")

.withValueMap(Bản đồ mới()

.add(":orderStatus", order.getStatus().toString())

.add(":cd", thứ tự.getCreationDate().getMillis())

.add(":consumerId", thứ tự.getConsumerId())

.add(":lineItems", mapLineItems(order.getLineItems()))

.add(":từ khóa", mapKeywords(thứ tự))

.add(":tên nhà hàng", order.getTên Nhà hàng())

.bản đồ())

.withReturnValues(Giá trị trả về. KHÔNG CÓ);

trả về idempotentUpdate(spec, eventSource);

}

Phương thức addOrder() tạo UpdateSpec, là một phần của AWS SDK và mô tả hoạt động cập nhật. Sau khi tạo UpdateSpec, nó gọi idempotent- Update(), một phương thức trợ giúp thực hiện cập nhật sau khi thêm một biểu thức điều kiện bảo vệ chống lại các bản cập nhật trùng lặp.

**TLƯU ÝPGhê quáBạnP()PHƯƠNG PHÁP**

CácnotePickedUp()phương pháp, được hiển thị trong danh sách 7.3, được gọi bởi trình xử lý sự kiện choGiao hàngĐã nhậnsự kiện. Nó thay đổitình trạng giao hàngcủaĐặt hàngmục đểĐÃ NHẬN.

**Liệt kê 7.3notePickedUp()phương pháp thay đổi trạng thái đơn hàng thànhĐÃ NHẬN**

lớp công khai OrderHistoryDaoDynamoDb ... @Override

public void notePickedUp(String orderId, Optional<SourceEvent> eventSource) {

UpdateItemSpec thông số kỹ thuật = new UpdateItemSpec()

.withPrimaryKey("mã đơn hàng", Mã đơn hàng)

.withUpdateExpression("ĐẶT #deliveryStatus = :deliveryStatus")

.withNameMap(Bộ sưu tập.singletonMap("#deliveryStatus",TRƯỜNG\_TRẠNG\_THÁI\_GIAO\_HÀNG))

.withValueMap(Bộ sưu tập.singletonMap(":deliveryStatus",Trạng thái giao hàng.PICKED\_UP.toString()))

.withReturnValues(Giá trị trả về. KHÔNG CÓ);idempotentUpdate(spec, eventSource);

}

Phương pháp này tương tự nhưthêm đơn hàng(). Nó tạo ra mộtCập nhậtItemSpecvà triệu hồi

idempotentUpdate(). Chúng ta hãy nhìn vàoidempotentUpdate()phương pháp.

**TNgười bất lựcBạnNGÀY 08/09/2019()PHƯƠNG PHÁP**

Danh sách sau đây hiển thị phương thức idempotentUpdate(), phương thức này cập nhật mục sau khi có thể thêm biểu thức điều kiện vào UpdateItemSpec để bảo vệ chống lại các bản cập nhật trùng lặp.

**Liệt kê 7.4idempotentUpdate()phương pháp bỏ qua các sự kiện trùng lặp**

lớp công khai OrderHistoryDaoDynamoDb ...

boolean riêng tư idempotentUpdate(UpdateItemSpec spec, Optional<SourceEvent> eventSource) {

thử {

table.updateItem(eventSource.map(es-> es.addDuplicateDetection(đặc tả))

.orElse(đặc tả));trả về giá trị đúng;

} bắt (ConditionalCheckFailedException e) {

// Không làm gì cả trả về false;

}

}

NếunguồnSự kiệnđược cung cấp,idempotentUpdate()gọiSourceEvent.add- DuplicateDetection()để thêm vàoCập nhậtItemSpecbiểu thức điều kiện đã được mô tả trước đó.idempotentUpdate()phương pháp bắt và bỏ quaNgoại lệ Kiểm tra không thành công,được ném bởicập nhật mục()nếu sự kiện trùng lặp.

Bây giờ chúng ta đã thấy đoạn mã cập nhật bảng, hãy cùng xem phương thức truy vấn.

**TANH TA TÌM THẤYỒĐẠI HỌCHCÂU CHUYỆN()PHƯƠNG PHÁP**

CáctìmLịch sử đơn hàng()phương pháp, được hiển thị trong danh sách 7.5, lấy lại các đơn đặt hàng của người tiêu dùng bằng cách truy vấnftgo-order-lịch-sửbảng sử dụngftgo-order-history-by-consumer-id-và-thời-gian-tạochỉ số thứ cấp. Nó có hai tham số:người tiêu dùngIdchỉ định người tiêu dùng, vàlọcchỉ định tiêu chí tìm kiếm. Phương pháp này tạo raTruy vấn-Spec—cái nào, giống nhưCập nhậtSpec, là một phần củaAWS SDK—từ các tham số của nó, truy vấn chỉ mục và chuyển đổi các mục được trả về thành mộtLịch sử đơn hàngsự vật.

**Liệt kê 7.5tìmLịch sử đơn hàng()phương pháp lấy lại các đơn hàng khớp lệnh của người tiêu dùng**

lớp công khai OrderHistoryDaoDynamoDb ... @Override

public OrderHistory findOrderHistory(String ConsumerId, OrderHistoryFilter

lọc) {

Đặc tả QuerySpec = new QuerySpec()

.withScanIndexForward(sai)

.withHashKey("consumerId",consumerId)

**Chỉ định rằng truy vấn phải trả về các đơn hàng theo thứ tự tăng dần độ tuổi**

.withRangeKeyCondition(mớiRangeKeyCondition("ngày tạo")

.gt(bộ lọc.getSince().getMillis()));

bộ lọc.getStartKeyToken().ifPresent(mã thông báo -> spec.withExclusiveStartKey(toStartingPrimaryKey(mã thông báo)));

Bản đồ<Chuỗi, Đối tượng> valuesMap = new HashMap<>();

**Độ tuổi tối đa của các đơn đặt hàng**

**trở lại**

Chuỗi filterExpression = Expressions.and(keywordFilterExpression(valuesMap, filter.getKeywords()),

trạng tháiFilterExpression(valuesMap,bộ lọc. getStatus()));

nếu (!valuesMap.isEmpty()) spec.withValueMap(valuesMap);

nếu (StringUtils.isNotBlank(filterExpression)) { spec.withFilterExpression(filterExpression);

}

**Xây dựng biểu thức bộ lọc và bản đồ giá trị giữ chỗ từ OrderHistoryFilter.**

**Giới hạn số lượng**

bộ lọc. getPageSize().ifPresent(spec::withMaxResultSize);ItemCollection<QueryOutcome> kết quả = index.query(spec);

trả về OrderHistory mới(

StreamSupport.stream(result.spliterator(),SAI)

**của kết quả nếungười gọi đã chỉ định kích thước trang.**

.map(this::toOrder)

.collect(toList()), Tùy chọn.ofNullable(kết quả

.getLastLowLevelResult()

.getQueryResult().getLastEvaluatedKey())

.map(this::toStartKeyToken));

}

**Tạo một Đơn hàng từ một mục được trả về bởi truy vấn.**

Sau khi xây dựng QuerySpec, phương thức này sẽ thực hiện truy vấn và xây dựng Order-History, chứa danh sách các Đơn hàng từ các mục được trả về.

CáctìmLịch sử đơn hàng()phương pháp thực hiện phân trang bằng cách tuần tự hóa giá trị được trả về bởilấy LastEvaluatedKey()thành một mã thông báo JSON. Nếu một máy khách chỉ định một mã thông báo bắt đầu trongBộ lọc Lịch sử đơn hàng, sau đótìmLịch sử đơn hàng()tuần tự hóa nó và gọivới ExclusiveStartKey()để cài đặt phím bắt đầu.

Như bạn thấy, bạn phải giải quyết nhiều vấn đề khi triển khai chế độ xem CQRS, bao gồm chọn cơ sở dữ liệu, thiết kế mô hình dữ liệu triển khai hiệu quả các bản cập nhật và truy vấn, xử lý các bản cập nhật đồng thời và xử lý các sự kiện trùng lặp. Phần phức tạp duy nhất của mã là DAO, vì nó phải xử lý đồng thời đúng cách và đảm bảo rằng các bản cập nhật là bất biến.

#### Bản tóm tắt

* Việc triển khai các truy vấn lấy dữ liệu từ nhiều dịch vụ là một thách thức vì dữ liệu của mỗi dịch vụ là riêng tư.
* Có hai cách để triển khai các loại truy vấn này: mô hình thành phần API và mô hình phân tách trách nhiệm truy vấn lệnh (CQRS).
* Mô hình thành phần API, thu thập dữ liệu từ nhiều dịch vụ, là cách đơn giản nhất để triển khai truy vấn và nên được sử dụng bất cứ khi nào có thể.
* Một hạn chế của mô hình thành phần API là một số truy vấn phức tạp yêu cầuviệc kết nối trong bộ nhớ không hiệu quả của các tập dữ liệu lớn.
* Mẫu CQRS, triển khai các truy vấn bằng cách sử dụng cơ sở dữ liệu dạng xem, mạnh hơn nhưng phức tạp hơn khi triển khai.
* Mô-đun chế độ xem CQRS phải xử lý các bản cập nhật đồng thời cũng như phát hiện và phân tíchsự kiện trùng lặp thẻ.
* CQRS cải thiện khả năng phân tách mối quan tâm bằng cách cho phép dịch vụ triển khai truy vấn trả về dữ liệu thuộc sở hữu của một dịch vụ khác.
* Khách hàng phải xử lý tính nhất quán cuối cùng của chế độ xem CQRS.

*Mẫu API bên ngoài*

***Chương này bao gồm***

* Thách thứcthiết kế các API hỗ trợ nhiều nhóm khách hàng khác nhau
* Áp dụng API Gateway và Backend cho frontendmẫu hình
* Thiết kế và triển khai cổng API
* Sử dụng lập trình phản ứng để đơn giản hóa APIthành phần
* Triển khai cổng API bằng GraphQL

Ứng dụng FTGO, giống như nhiều ứng dụng khác, có một REST API. Các máy khách của nó bao gồm các ứng dụng di động FTGO, JavaScript chạy trong trình duyệt và các ứng dụng do các đối tác phát triển. Trong một kiến ​​trúc đơn khối như vậy, API được cung cấp cho máy khách là API của đơn khối. Nhưng khi nhóm FTGO bắt đầu triển khai các dịch vụ vi mô, sẽ không còn một API nào nữa, vì mỗi dịch vụ đều có API riêng. Mary và nhóm của cô ấy phải quyết định loại API nào mà ứng dụng FTGO hiện sẽ cung cấp cho máy khách. Ví dụ, máy khách có nên biết về sự tồn tại của các dịch vụ và gửi yêu cầu trực tiếp đến chúng không?

**253**

Nhiệm vụ thiết kế API bên ngoài của ứng dụng được thực hiện thậm chíthách thức hơn do tính đa dạng của các máy khách. Các máy khách khác nhau thường yêu cầu dữ liệu khác nhau. Giao diện người dùng dựa trên trình duyệt trên máy tính để bàn thường hiển thị nhiều thông tin hơn nhiều so với ứng dụng di động. Ngoài ra, các máy khách khác nhau truy cập các dịch vụ qua các loại mạng khác nhau. Các máy khách trong tường lửa sử dụng mạng LAN hiệu suất cao và các máy khách bên ngoài tường lửa sử dụng internet hoặc mạng di động, sẽ có hiệu suất thấp hơn. Do đó, như bạn sẽ biết, thường không hợp lý khi có một API duy nhất, phù hợp với tất cả.

Chương này bắt đầu bằng việc mô tả nhiều vấn đề thiết kế API bên ngoài. Sau đó, tôi mô tả các mẫu API bên ngoài. Tôi đề cập đến mẫu API gateway và sau đó là mẫu Back-end cho frontend. Sau đó, tôi thảo luận về cách thiết kế và triển khai API gateway. Tôi xem xét các tùy chọn khác nhau có sẵn, bao gồm các sản phẩm và khuôn khổ API gateway có sẵn để bạn tự phát triển. Tôi mô tả thiết kế và triển khai API gateway được xây dựng bằng khuôn khổ Spring Cloud Gateway. Tôi cũng mô tả cách xây dựng API gateway bằng GraphQL, một khuôn khổ cung cấp ngôn ngữ truy vấn dựa trên đồ thị.

#### Các vấn đề thiết kế API bên ngoài

Để khám phá những vấn đề khác nhau liên quan đến API, chúng ta hãy xem xét ứng dụng FTGO.Như hình 8.1 cho thấy, các dịch vụ của ứng dụng này được nhiều loại máy khách sử dụng. Bốn loại máy khách sử dụng API của dịch vụ:

* + - Các ứng dụng web, chẳng hạn nhưỨng dụng web dành cho người tiêu dùng, triển khai giao diện người dùng dựa trên trình duyệt cho người tiêu dùng,Ứng dụng web nhà hàng, thực hiện-ments giao diện người dùng dựa trên trình duyệt cho nhà hàng vàỨng dụng web quản trị, thực hiện giao diện người dùng quản trị nội bộ
    - Ứng dụng JavaScriptchạy trong trình duyệt
    - Ứng dụng di động, một dành cho người tiêu dùng và một dành cho người chuyển phát nhanh
    - Các ứng dụng được viết bởi các nhà phát triển bên thứ ba

Các ứng dụng web chạy bên trong tường lửa, do đó chúng truy cập các dịch vụ qua mạng LAN băng thông cao, độ trễ thấp. Các máy khách khác chạy bên ngoài tường lửa, do đó chúng truy cập các dịch vụ qua mạng internet băng thông thấp, độ trễ cao hoặc mạng di động.

Một cách tiếp cận thiết kế API là để khách hàng gọi trực tiếp các dịch vụ. Trên bề mặt, điều này nghe có vẻ khá đơn giản—sau cùng, đó là cách khách hàng gọi API của một ứng dụng đơn khối. Nhưng cách tiếp cận này hiếm khi được sử dụng trong kiến ​​trúc vi dịch vụ vì những nhược điểm sau:

* + - Các API dịch vụ chi tiết yêu cầu khách hàng phải thực hiện nhiều yêu cầu để truy xuất dữ liệu họ cần, điều này không hiệu quả và có thể dẫn đến trải nghiệm người dùng kém.
    - Việc thiếu đóng gói do khách hàng biết về từng dịch vụ và API của dịch vụ đó khiến việc thay đổi kiến ​​trúc và API trở nên khó khăn.
    - Các dịch vụ có thể sử dụng cơ chế IPC không thuận tiện hoặc không thực tế để khách hàng sử dụng, đặc biệt là những khách hàng nằm ngoài tường lửa.

Tường lửa

Trang web

yêu cầu

Giao diện lập trình ứng dụng (API)

yêu cầu

iPhone/

Ứng dụng Android

Giao diện lập trình ứng dụng (API)

yêu cầu

Giao diện lập trình ứng dụng (API)

yêu cầu

Ứng dụng web

Ứng dụng của bên thứ 3

Dịch vụ phụ trợ

Dịch vụ nhà bếp

Dịch vụ giao hàng

Dịch vụ khách hàng

Dịch vụ đặt hàng

Trình duyệt HTML

Ứng dụng JavaScript

Hiệu suất thấp hơninternet

Hiệu suất cao hơnMạng LAN

**Hình 8.1 Ứng dụng FTGOdịch vụ và khách hàng của họ. Có một số loại khách hàng khác nhau. Một số nằm bên trong tường lửa, và một số khác nằm bên ngoài. Những người bên ngoài tường lửa truy cập các dịch vụ qua mạng internet/di động có hiệu suất thấp hơn. Những khách hàng bên trong tường lửa sử dụng mộthiệu suất LAN.**

Để tìm hiểu thêm về những nhược điểm này, chúng ta hãy cùng xemvề cách ứng dụng di động FTGO dành cho người tiêu dùng thu thập dữ liệu từ các dịch vụ.

###### Các vấn đề thiết kế API cho máy khách di động FTGO

Người tiêu dùng sử dụng ứng dụng di động FTGO để đặt và quản lý đơn hàng của họ. Hãy tưởng tượngbạn đang phát triển chế độ xem Xem đơn hàng của máy khách di động, chế độ này hiển thị một đơn hàng. Như đã mô tả trong chương 7, thông tin được hiển thị bởi chế độ xem này bao gồm thông tin đơn hàng cơ bản, bao gồm trạng thái của đơn hàng, trạng thái thanh toán, trạng thái của đơn hàng theo quan điểm của nhà hàng và trạng thái giao hàng, bao gồm vị trí và thời gian giao hàng ước tính nếu đang vận chuyển.

Phiên bản đơn khối của ứng dụng FTGO có điểm cuối API trả về chi tiết đơn hàng. Máy khách di động lấy thông tin cần thiết bằng cách thực hiện một yêu cầu duy nhất. Ngược lại, trong phiên bản dịch vụ vi mô của ứng dụng FTGO, chi tiết đơn hàng, như đã mô tả trước đó, được phân tán trên nhiều dịch vụ, bao gồm:

* + - * Dịch vụ đặt hàng—Thông tin đơn hàng cơ bản, bao gồm các chi tiếtvà trạng thái
      * Dịch vụ nhà bếp—Trạng thái đơn hàng theo quan điểm của nhà hàng và thời gian ước tính để nhận hàng
      * Dịch vụ giao hàng—Trạng thái giao hàng của đơn hàng, thời gian giao hàng ước tính và vị trí hiện tại của đơn hàng
      * Dịch vụ kế toán—Trạng thái thanh toán của đơn hàng

Nếu máy khách di động gọi trực tiếp các dịch vụ, thì như hình 8.2 cho thấy, máy khách phải thực hiện nhiều cuộc gọi để lấy dữ liệu này.

**Yêu cầu một API**

internet

lấy chi tiết đơn hàng()

Ứng dụng tiêu dùng iPhone/ Android

Tường lửa

Ứng dụng FTGO nguyên khối

Tường lửa

lấy đơn hàng()

internet

lấyDelivery()

lấyBill()

lấy Vé()

Ứng dụng tiêu dùng iPhone/ Android

Dịch vụ phụ trợ FTGO

Dịch vụ nhà bếp

Dịch vụ kế toán

Dịch vụ giao hàng

Dịch vụ đặt hàng

**Nhiều lệnh gọi API được yêu cầu**

**Hình 8.2 Một khách hàng có thể lấy thông tin chi tiết về đơn hàng từ ứng dụng FTGO nguyên khối chỉ bằng một yêu cầu. Nhưng khách hàng phải thực hiện nhiều yêu cầu để lấy cùng một thông tin trong mộtkiến trúc dịch vụ vi mô.**

Trong thiết kế này, ứng dụng di động đóng vai trò là trình soạn thảo API. Nó gọi nhiều dịch vụ và kết hợp các kết quả. Mặc dù cách tiếp cận này có vẻ hợp lý, nhưng nó có một số vấn đề nghiêm trọng.

**PTRẢI NGHIỆM NGƯỜI DÙNG OOR DO KHÁCH HÀNG THỰC HIỆN NHIỀU YÊU CẦU**

Vấn đề đầu tiên là ứng dụng di động đôi khi phải thực hiện nhiều yêu cầu để truy xuất dữ liệu mà nó muốn hiển thị cho người dùng. Tương tác tán gẫu giữa ứng dụng và các dịch vụ có thể khiến ứng dụng có vẻ không phản hồi, đặc biệt là khi sử dụng internet hoặc mạng di động. Internet có băng thông thấp hơn nhiều và độ trễ cao hơn nhiều so với mạng LAN, và mạng di động thậm chí còn tệ hơn. Độ trễ của mạng di động (và internet) thường lớn hơn 100 lần so với mạng LAN.

Độ trễ cao hơn có thể không phải là vấn đề khi truy xuất thông tin chi tiết về đơn hàng, vì ứng dụng di động giảm thiểu độ trễ bằng cách thực hiện các yêu cầu đồng thời. Thời gian phản hồi chung không lớn hơn thời gian phản hồi của một yêu cầu duy nhất. Nhưng trong các trường hợp khác, khách hàng có thể cần thực hiện các yêu cầu theo trình tự, điều này sẽ dẫn đến trải nghiệm người dùng kém.

Hơn nữa, trải nghiệm người dùng kém do độ trễ mạng không phải là vấn đề duy nhất với API chatty. Nó đòi hỏi nhà phát triển di động phải viết mã biên soạn API có khả năng phức tạp. Công việc này làm sao nhãng nhiệm vụ chính của họ là tạo ra trải nghiệm người dùng tuyệt vời. Ngoài ra, vì mỗi yêu cầu mạng đều tiêu tốn năng lượng, API chatty làm hao pin thiết bị di động nhanh hơn.

**LACK OF ENCAPSULATION YÊU CẦU CÁC NHÀ PHÁT TRIỂN FRONTEND PHẢI THAY ĐỔI MÃ CỦA HỌ TRONG LOCKSTEP VỚI BACKEND**

Một nhược điểm khác của ứng dụng di động truy cập trực tiếp vàodịch vụ là sự thiếu đóng gói. Khi một ứng dụng phát triển, các nhà phát triển dịch vụ đôi khi thay đổi API theo cách phá vỡ các máy khách hiện có. Họ thậm chí có thể thay đổi cách hệ thống được phân tách thành các dịch vụ. Các nhà phát triển có thể thêm các dịch vụ mới và tách hoặc hợp nhất các dịch vụ hiện có. Nhưng nếu kiến ​​thức về các dịch vụ được đưa vào ứng dụng di động, có thể khó thay đổi API của các dịch vụ.

Không giống như khi cập nhật ứng dụng phía máy chủ, phải mất hàng giờ hoặc thậm chí nhiều ngày để triển khai phiên bản mới của ứng dụng di động. Apple hoặc Google phải chấp thuận bản nâng cấp và cho phép tải xuống. Người dùng có thể không tải xuống bản nâng cấp ngay lập tức—nếu có. Và bạn có thể không muốn buộc những người dùng miễn cưỡng phải nâng cấp. Chiến lược đưa API dịch vụ ra thiết bị di động tạo ra một trở ngại đáng kể cho việc phát triển các API đó.

**SDỊCH VỤ CÓ THỂ SỬ DỤNG KHÁCH HÀNG-KHÔNG THÂN THIỆNIPCCƠ CHẾ**

Một thách thức khác với ứng dụng di động gọi trực tiếp các dịch vụ là một số dịch vụ có thể sử dụng các giao thức mà máy khách không dễ dàng sử dụng. Các ứng dụng máy khách chạy bên ngoài tường lửa thường sử dụng các giao thức như HTTP và WebSockets. Nhưng như đã mô tả trong chương 3, các nhà phát triển dịch vụ có nhiều giao thức để lựa chọn—không chỉ HTTP. Một số dịch vụ của ứng dụng có thể sử dụng gRPC, trong khi những dịch vụ khác có thể sử dụng giao thức nhắn tin AMQP. Các loại giao thức này hoạt động tốt

bên trong, nhưng có thể không dễ dàng được sử dụng bởi máy khách di động. Một số thậm chí không thân thiện với tường lửa.

###### Các vấn đề thiết kế API cho các loại khách hàng khác

Tôi đã chọn ứng dụng di động vì đây là cách tuyệt vời để chứng minh những nhược điểm của cli-ents truy cập trực tiếp vào các dịch vụ. Nhưng các vấn đề phát sinh do việc phơi bày các dịch vụ cho máy khách không chỉ dành riêng cho máy khách di động. Các loại máy khách khác, đặc biệt là những máy khách nằm ngoài tường lửa, cũng gặp phải những vấn đề này. Như đã mô tả trước đó, các dịch vụ của ứng dụng FTGO được sử dụng bởi các ứng dụng web, ứng dụng JavaScript dựa trên trình duyệt và các ứng dụng của bên thứ ba. Chúng ta hãy xem xét các vấn đề về thiết kế API với các máy khách này.

**Giao diện lập trình ứng dụng (API)VẤN ĐỀ THIẾT KẾ CHO ỨNG DỤNG WEB**

Phía máy chủ truyền thốngcác ứng dụng web, xử lý các yêu cầu HTTP từ trình duyệt và trả về các trang HTML, chạy trong tường lửa và truy cập các dịch vụ qua mạng LAN. Băng thông mạng và độ trễ không phải là trở ngại đối với việc triển khai thành phần API trong ứng dụng web. Ngoài ra, các ứng dụng web có thể sử dụng các giao thức không thân thiện với web để truy cập các dịch vụ. Các nhóm phát triển ứng dụng web là một phần của cùng một tổ chức và thường làm việc chặt chẽ với các nhóm viết dịch vụ phụ trợ, do đó, ứng dụng web có thể dễ dàng được cập nhật bất cứ khi nào các dịch vụ phụ trợ thay đổi. Do đó, ứng dụng web có thể truy cập trực tiếp vào các dịch vụ phụ trợ.

**Giao diện lập trình ứng dụng (API)THIẾT KẾVẤN ĐỀ CHO TRÌNH DUYỆT-DỰA TRÊN JAVASCRIPTỨNG DỤNG**

Các ứng dụng trình duyệt hiện đại sử dụng một lượng JavaScript nhất định. Ngay cả khi HTML chủ yếu được tạo ra bởi ứng dụng web phía máy chủ, JavaScript chạy trong trình duyệt thường sẽ gọi các dịch vụ. Ví dụ, tất cả các ứng dụng web FTGOcations—Consumer, Restaurant và Admin—chứa JavaScript gọi các dịch vụ back-end. Ví dụ, ứng dụng web Consumer làm mới động trang Chi tiết đơn hàng bằng JavaScript gọi các API dịch vụ.

Một mặt, các ứng dụng JavaScript dựa trên trình duyệt dễ cập nhật khi API dịch vụ thay đổi. Mặt khác, các ứng dụng JavaScript truy cập các dịch vụ qua internet có cùng vấn đề về độ trễ mạng như các ứng dụng di động. Tệ hơn nữa, các giao diện người dùng dựa trên trình duyệt, đặc biệt là các giao diện dành cho máy tính để bàn, thường phức tạp hơn và cần tạo ra nhiều dịch vụ hơn so với các ứng dụng di động.Rất có thể các ứng dụng dành cho Người tiêu dùng và Nhà hàng, những ứng dụng truy cập dịch vụ qua Internet, sẽ không thể tạo ra các API dịch vụ một cách hiệu quả.

**DKÝ TÊNGiao diện lập trình ứng dụng (API)S CHO THỨ BA-ĐƠN XIN BỮA TIỆC**

FTGO, giống như nhiều tổ chức khác, cung cấp API cho các nhà phát triển bên thứ ba. Các nhà phát triển có thể sử dụng API FTGO để viết các ứng dụng đặt và quản lý đơn hàng. Các ứng dụng bên thứ ba này truy cập API qua internet, do đó, việc tạo thành API có thể không hiệu quả. Nhưng việc tạo thành API không hiệu quả là một vấn đề tương đối nhỏ so với thách thức lớn hơn nhiều trong việc thiết kế API

được các ứng dụng của bên thứ ba sử dụng. Đó là vì các nhà phát triển bên thứ ba cần một API ổn định.

Rất ít tổ chức có thể buộc các nhà phát triển bên thứ ba nâng cấp lên API mới. Các tổ chức có API không ổn định có nguy cơ mất nhà phát triển vào tay đối thủ cạnh tranh. Do đó, bạn phải quản lý cẩn thận quá trình phát triển của API được các nhà phát triển bên thứ ba sử dụng. Bạn thường phải duy trì các phiên bản cũ trong một thời gian dài—có thể là mãi mãi.

Yêu cầu này là gánh nặng lớn đối với một tổ chức. Không thực tế khi để các nhà phát triển dịch vụ phụ trợ chịu trách nhiệm duy trì khả năng tương thích ngược lâu dài. Thay vì cung cấp dịch vụ trực tiếp cho các nhà phát triển bên thứ ba, các tổ chức nên có một API công khai riêng do một nhóm riêng phát triển. Như bạn sẽ tìm hiểu sau, API công khai được triển khai bởi một thành phần kiến ​​trúc được gọi là cổng API. Hãy cùng xem cổng API hoạt động như thế nào.

#### Mẫu cổng API

Như bạn vừa thấy, có rất nhiều nhược điểm khi các dịch vụ truy cập trực tiếp vào các dịch vụ. Thường thì không thực tế khi khách hàng thực hiện biên soạn API qua internet. Việc thiếu đóng gói khiến các nhà phát triển khó thay đổi phân tách dịch vụ và API. Đôi khi các dịch vụ sử dụng các giao thức truyền thông không phù hợp bên ngoài tường lửa. Do đó, một cách tiếp cận tốt hơn nhiều là sử dụng cổng API.

**Mẫu: Cổng API**

Triển khai một dịch vụ là điểm vào ứng dụng dựa trên dịch vụ vi mô từ các máy khách API bên ngoài. Xem<http://microservices.io/patterns/apigateway.html>.

Cổng API là một dịch vụđiểm vào ứng dụng từ thế giới bên ngoài. Nó chịu trách nhiệm định tuyến yêu cầu, biên soạn API và các chức năng khác, chẳng hạn như xác thực. Phần này đề cập đến mô hình cổng API. Tôi thảo luận về lợi ích và hạn chế của nó và mô tả các vấn đề thiết kế khác nhau mà bạn phải giải quyết khi phát triển cổng API.

###### Tổng quan về mô hình cổng API

Mục 8.1.1 mô tả những nhược điểm của máy khách, chẳng hạn như ứng dụng di động FTGO, thực hiện nhiều yêu cầu để hiển thị thông tin cho người dùng. Một cách tiếp cận tốt hơn nhiều là máy khách thực hiện một yêu cầu duy nhất đến cổng API, một dịch vụ đóng vai trò là điểm nhập duy nhất cho các yêu cầu API vào ứng dụng từ bên ngoài tường lửa. Nó tương tự như mô hình Facade từ thiết kế hướng đối tượng. Giống như một mặt tiền, cổng API đóng gói kiến ​​trúc bên trong của ứng dụng và cung cấp API cho máy khách của nó. Nó cũng có thể có các trách nhiệm khác, chẳng hạn như xác thực, giám sát,

và giới hạn tốc độ. Hình 8.3 cho thấy mối quan hệ giữa máy khách, cổng API và các dịch vụ.

Trang web

yêu cầu

Giao diện lập trình ứng dụng (API)

yêu cầu

yêu cầu

Giao diện lập trình ứng dụng (API)

cổng vào

Ứng dụng web

Giao diện lập trình ứng dụng (API)

Giao diện lập trình ứng dụng (API)

yêu cầu

Ứng dụng iPhone/ Android

Tường lửa

Dịch vụ phụ trợ

Dịch vụ giao hàng

Dịch vụ khách hàng

Dịch vụ đặt hàng

Trình duyệt HTML

Ứng dụng JavaScript

Ứng dụng của bên thứ 3

Hiệu suất thấp hơninternet

Hiệu suất cao hơnMạng LAN

**Hình 8.3 Cổng API là điểm vào duy nhất vào ứng dụng cho các cuộc gọi API từ bên ngoàitường lửa.**

Cổng API chịu trách nhiệm định tuyến yêu cầu, biên soạn API và biên dịch giao thức. Tất cả các yêu cầu API từ máy khách bên ngoài trước tiên sẽ đến cổng API, nơi định tuyến một số yêu cầu đến dịch vụ phù hợp. Cổng API xử lý các yêu cầu khác bằng cách sử dụng mẫu biên soạn API và bằng cách gọi nhiều dịch vụ và tổng hợp kết quả. Nó cũng có thể biên dịch giữa các giao thức thân thiện với máy khách như HTTP và WebSockets và các giao thức không thân thiện với máy khách được các dịch vụ sử dụng.

**RLỘ TRÌNH EQUEST**

Một trong những chức năng chính của API gateway là định tuyến yêu cầu. API gateway triển khai một số hoạt động API bằng cách định tuyến yêu cầu đến dịch vụ tương ứng. Khi nhận được yêu cầu, API gateway sẽ tham khảo bản đồ định tuyến chỉ định dịch vụ nào để định tuyến yêu cầu đến. Ví dụ, bản đồ định tuyến có thể ánh xạ phương thức HTTP và đường dẫn đến URL HTTP của dịch vụ. Chức năng này giống hệt với các tính năng proxy ngược do máy chủ web như NGINX cung cấp.

**APICBẤT NGỜ**

Cổng API thường làm nhiều hơn là chỉ đơn giản là đảo ngược proxy. Nó cũng có thể triển khai một số hoạt động API bằng cách sử dụng thành phần API. Cổng API FTGO, để kiểm traple, triển khai hoạt động API Get Order Details bằng cách sử dụng API composition. Như hình 8.4 cho thấy, ứng dụng di động thực hiện một yêu cầu đến cổng API, cổng này sẽ lấy thông tin chi tiết về đơn hàng từ nhiều dịch vụ.

Cổng API FTGO cung cấp mộtAPI cho phép khách hàng di động truy xuất dữ liệu họ cần chỉ bằng một yêu cầu. Ví dụ, khách hàng di độngtạo một getOrderDetails() duy nhấtyêu cầu đến cổng API.

**Nhiều lệnh gọi API được yêu cầu**

Tường lửa

internet

lấy đơn hàng() lấy giao hàng()

lấy Hóa đơn() lấy Vé()

Ứng dụng tiêu dùng iPhone/ Android

Dịch vụ phụ trợ FTGO

Dịch vụ nhà bếp

Dịch vụ kế toán

Dịch vụ giao hàng

Dịch vụ đặt hàng

Mạng LAN

lấy đơn hàng()

internet

lấyDelivery()

lấy chi tiết đơn hàng()

lấyBill()

lấy Vé()

Ứng dụng tiêu dùng iPhone/ Android

Tường lửa

Giao diện lập trình ứng dụng (API)

cổng vào

Dịch vụ phụ trợ FTGO

Dịch vụ nhà bếp

Dịch vụ kế toán

Dịch vụ giao hàng

Dịch vụ đặt hàng

**Mạng hiệu suất thấp hơn**

**Yêu cầu một lệnh gọi API**

**Mạng hiệu suất cao hơn**

**Hình 8.4 Cổng API thường thực hiện thành phần API, cho phép một máy khách như vậynhư một thiết bị di động để truy xuất dữ liệu hiệu quả bằng một yêu cầu API duy nhất.**

**PROTOCOLDỊCH THUẬT**

Cổng API cũng có thể thực hiện dịch giao thức. Nó có thể cung cấp API RESTful cho các máy khách bên ngoài, mặc dù các dịch vụ ứng dụng sử dụng hỗn hợp các giao thức bên trong, bao gồm REST và gRPC. Khi cần, việc triển khai một số hoạt động API sẽ dịch giữa API RESTful bên ngoài và API dựa trên gRPC bên trong.

**TANH TAGiao diện lập trình ứng dụng (API)GATEWAY CUNG CẤP CHO MỖI KHÁCH HÀNG CÁC KHÁCH HÀNG-****CỤ THỂGiao diện lập trình ứng dụng (API)**

Cổng API có thể cung cấp một API duy nhất phù hợp với tất cả (OSFA). Vấn đề với một API duy nhất là các máy khách khác nhau thường có các yêu cầu khác nhau. Ví dụ, một ứng dụng của bên thứ ba có thể yêu cầu hoạt động API Lấy thông tin chi tiết đơn hàng để trả về thông tin chi tiết đơn hàng đầy đủ, trong khi máy khách di động chỉ cần một tập hợp con dữ liệu. Một cách để giải quyết vấn đề này là cung cấp cho máy khách tùy chọn chỉ định trong yêu cầu những trường nào và các đối tượng liên quan mà máy chủ sẽ trả về. Cách tiếp cận này phù hợp với một API công khai phải phục vụ nhiều ứng dụng của bên thứ ba, nhưng thường không cung cấp cho máy khách quyền kiểm soát mà họ cần.

Một cách tiếp cận tốt hơn là API Gateway cung cấp cho mỗi máy khách API riêng của nó. Ví dụ, FTGO API Gateway có thể cung cấp cho máy khách di động FTGO một API được thiết kế riêng để đáp ứng các yêu cầu của nó. Nó thậm chí có thể có các API khác nhau cho các ứng dụng di động Android và iPhone. API Gateway cũng sẽ triển khai một API công khai để các nhà phát triển bên thứ ba sử dụng. Sau này, tôi sẽ mô tả mẫu Backends cho frontends đưa khái niệm về API-per-client này đi xa hơn nữa bằng cách xác định một API Gateway riêng cho mỗi máy khách.

**TÔITRIỂN KHAI CÁC CHỨC NĂNG EDGE**

Mặc dù trách nhiệm chính của cổng API là định tuyến và biên soạn API, nhưng nó cũng có thể triển khai những gì được gọi là hàm biên. Hàm biên, như tên gọi của nó, là hàm xử lý yêu cầu được triển khai ở biên của ứng dụng. Ví dụ về các hàm biên mà ứng dụng có thể triển khai bao gồm:

* *Xác thực*—Xác minh danh tính của khách hàng đưa ra yêu cầu.
* *Ủy quyền*—Xác minh rằng khách hàngđược phép thực hiện hoạt động cụ thể đó.
* *Giới hạn tỷ lệ*—Hạn chế số lượng yêu cầu mỗi giây từ một máy khách cụ thể và/hoặc từ tất cả máy khách.
* *Bộ nhớ đệm*—Lưu trữ phản hồi để giảm số lượng yêu cầu gửi đến các dịch vụ.
* *Bộ sưu tập số liệu*—Thu thập số liệu trên APIsử dụng cho mục đích phân tích thanh toán.
* *Yêu cầu ghi nhật ký*—Yêu cầu ghi nhật ký.

Có ba nơi khác nhau trong ứng dụng của bạn mà bạn có thể triển khai các chức năng biên này. Đầu tiên, bạn có thể triển khai chúng trong các dịch vụ phụ trợ. Điều này có thể hợp lý đối với một số chức năng, chẳng hạn như lưu trữ đệm, thu thập số liệu và có thể là ủy quyền. Nhưng nhìn chung sẽ an toàn hơn nếu ứng dụng xác thực các yêu cầu trên biên trước khi chúng đến được các dịch vụ.

Tùy chọn thứ hai là triển khai các chức năng biên này trong một dịch vụ biên nằm ở thượng nguồn của API Gateway. Dịch vụ biên là điểm liên hệ đầu tiên cho một máy khách bên ngoài. Nó xác thực yêu cầu và thực hiện các xử lý biên khác trước khi chuyển yêu cầu đến API Gateway.

Một lợi ích quan trọng của việc sử dụng dịch vụ biên chuyên dụng là nó tách biệt các mối quan tâm. Cổng API tập trung vào định tuyến và biên soạn API. Một lợi ích khác là nó tập trung trách nhiệm cho các chức năng biên quan trọng như xác thực. Điều đó đặc biệt có giá trị khi một ứng dụng có nhiều cổng API có thể được viết bằng nhiều ngôn ngữ và khuôn khổ khác nhau. Tôi sẽ nói thêm về điều đó sau. Nhược điểm của cách tiếp cận này là nó làm tăng độ trễ mạng do có thêm bước nhảy. Nó cũng làm tăng thêm độ phức tạp của ứng dụng.

Do đó, thường thuận tiện khi sử dụng tùy chọn thứ ba và triển khai các chức năng biên này, đặc biệt là ủy quyền, trong chính API gateway. Có ít hơn một bước nhảy mạng, giúp cải thiện độ trễ. Cũng có ít bộ phận chuyển động hơn, giúp giảm độ phức tạp. Chương 11 mô tả cách API gateway và các dịch vụ cộng tác để triển khai bảo mật.

**Giao diện lập trình ứng dụng (API)KIẾN TRÚC CỔNG**

Cổng API có kiến ​​trúc mô-đun theo lớp. Kiến trúc của nó, được thể hiện trong hình 8.5, bao gồm hai lớp: lớp API và một lớp chung. Lớp API bao gồm một hoặc nhiều mô-đun API độc lập. Mỗi mô-đun API triển khai một API cho một

Lớp API

API công khai

API trình duyệt

API di động

Cổng API

Ứng dụng của bên thứ 3

Trình duyệtỨng dụng JavaScript

Khách hàng di động

Lớp chung

**Hình 8.5Cổng API có kiến ​​trúc mô-đun nhiều lớp. API cho mỗi máy khách được triển khai bởi một mô-đun riêng biệt. Lớp chung triển khai chức năng chung cho tất cả các API, chẳng hạn như xác thực.**

khách hàng cụ thể. Lớp chung triển khai chức năng chia sẻ, bao gồm các chức năng biên như xác thực.

Trong ví dụ này, cổng API có ba mô-đun API:

* *API di động*—Triển khai API cho ứng dụng di động FTGO
* *API trình duyệt*—Triển khai API cho ứng dụng JavaScript đang chạy trên trình duyệt
* *API công khai*—Triển khai API cho các nhà phát triển bên thứ ba

Một mô-đun API triển khai từng hoạt động API theo một trong hai cách. Một số hoạt động API ánh xạ trực tiếp đến hoạt động API dịch vụ đơn lẻ. Một mô-đun API triển khai các hoạt động này bằng cách định tuyến các yêu cầu đến hoạt động API dịch vụ tương ứng. Nó có thể định tuyến các yêu cầu bằng cách sử dụng một mô-đun định tuyến chung đọc tệp cấu hình mô tả các quy tắc định tuyến.

Một mô-đun API triển khai các hoạt động API khác, phức tạp hơn bằng cách sử dụng API com-composition. Việc triển khai hoạt động API này bao gồm mã tùy chỉnh. Mỗi triển khai hoạt động API xử lý các yêu cầu bằng cách gọi nhiều dịch vụ và kết hợp các kết quả.

**Giao diện lập trình ứng dụng (API)MÔ HÌNH SỞ HỮU GATEWAY**

Một câu hỏi quan trọng mà bạn phải trả lời là ai chịu trách nhiệm phát triển cổng API và hoạt động của nó? Có một số tùy chọn khác nhau. Một là để một nhóm riêng chịu trách nhiệm về API gateway. Nhược điểm của nó là nó tương tự như SOA, trong đó nhóm Enterprise Service Bus (ESB) chịu trách nhiệm cho mọi hoạt động phát triển ESB. Nếu một nhà phát triển làm việc trên ứng dụng di động cần truy cập vào một dịch vụ cụ thể, họ phải gửi yêu cầu đến nhóm API gateway và đợi họ công khai API. Kiểu nút thắt cổ chai tập trung này trong tổ chức rất trái ngược với triết lý của kiến ​​trúc vi dịch vụ, vốn thúc đẩy các nhóm tự chủ liên kết lỏng lẻo.

Một cách tiếp cận tốt hơn, được Netflix thúc đẩy, là để các nhóm khách hàng—nhóm API di động, web và công khai—sở hữu mô-đun API để hiển thị API của họ. Nhóm cổng API chịu trách nhiệm phát triển mô-đun Chung và các khía cạnh vận hành của cổng. Mô hình sở hữu này, được thể hiện trong hình 8.6, trao cho các nhóm quyền kiểm soát API của họ.

Khi một nhóm cần thay đổi API của họ, họ sẽ kiểm tra các thay đổi trong kho lưu trữ nguồn cho API Gateway. Để hoạt động tốt, đường ống triển khai API Gateway phải được tự động hóa hoàn toàn. Nếu không, các nhóm khách hàng thường sẽ bị chặn khi chờ nhóm API Gateway triển khai phiên bản mới.

**BạnHÁTBACKENDS CHO MẪU FRONTENDS**

Một mối quan tâm với API gateway là trách nhiệm đối với nó bị mờ nhạt. Nhiều nhóm đóng góp vào cùng một cơ sở mã. Một nhóm API gateway chịu trách nhiệm cho hoạt động của nó. Mặc dù không tệ như SOA ESB, nhưng sự mờ nhạt về trách nhiệm này trái ngược với triết lý kiến ​​trúc vi dịch vụ là "nếu bạn xây dựng nó, bạn sở hữu nó".



Đội ngũ khách hàng di động

Nhóm khách hàng trình duyệt

Nhóm API công khai

Khách hàng di động

Trình duyệtJavaScript

ứng dụng

Ứng dụng của bên thứ 3

Sở hữu

Sở hữu

Sở hữu

Cổng API

Lớp API

API di động

API trình duyệt

API công khai

Sở hữu

Lớp chung

Nhóm cổng API



**Hình 8.6 Một nhóm khách hàng sở hữu mô-đun API của họ. Khi họ thay đổi khách hàng, họ có thể thay đổi APImô-đun và không yêu cầu nhóm cổng API thực hiện thay đổi.**

Giải pháp là có một cổng API cho mỗi máy khách, cái gọi là mô hình Backends for front-ends (BFF), được tiên phong bởi Phil Calçado (<http://philcalcado.com/>) và các đồng nghiệp của anh ấy tại SoundCloud. Như hình 8.7 cho thấy, mỗi mô-đun API trở thành cổng API độc lập do một nhóm khách hàng duy nhất phát triển và vận hành.

**Mẫu: Backend cho frontend**

Triển khai một cổng API riêng cho từng loại khách hàng. Xem[http://dịch vụ vi mô](http://microservices.io/patterns/apigateway.html)

[.io/patterns/apigateway.html](http://microservices.io/patterns/apigateway.html).



Di độngnhóm khách hàngNhóm khách hàng trình duyệtCông khai Nhóm API

Sở hữu

API di động

Lớp API

API di độngcổng vào

Khách hàng di động

Lớp chung

Sở hữu

API trình duyệt

Lớp API

API trình duyệtcổng vào

Trình duyệtỨng dụng JavaScript

Lớp chung

Sở hữu

API công khai

Lớp API

API công khaicổng vào

Ứng dụng của bên thứ 3

Lớp chung

**Hình 8.7 Mẫu Backends for frontends định nghĩa một cổng API riêng cho mỗi máy khách. MỗiNhóm khách hàng sở hữu cổng API của họ. Nhóm cổng API sở hữu lớp chung.**

Nhóm API công khai sở hữu và vận hành cổng API của họ, nhóm di động sở hữu và vận hành cổng của họ, v.v. Về lý thuyết, các cổng API khác nhau có thể được phát triển bằng cách sử dụng các ngăn xếp công nghệ khác nhau. Nhưng điều đó có nguy cơ trùng lặp mã cho chức năng chung, chẳng hạn như mã triển khai các chức năng biên. Lý tưởng nhất là tất cả các cổng API đều sử dụng cùng một ngăn xếp công nghệ. Chức năng chung là một thư viện được chia sẻ do nhóm cổng API triển khai.

Bên cạnh việc xác định rõ ràng trách nhiệm, mô hình BFF còn có những lợi ích khác. Các mô-đun API được tách biệt với nhau, giúp cải thiện độ tin cậy. Một API hoạt động không tốt không dễ dàng ảnh hưởng đến các API khác. Nó cũng cải thiện khả năng quan sát, vì các mô-đun API khác nhau là các quy trình khác nhau. Một lợi ích khác của mô hình BFF là mỗi API có thể mở rộng độc lập. Mô hình BFF cũng giảm thời gian khởi động vì mỗi cổng API là một ứng dụng nhỏ hơn, đơn giản hơn.

###### Lợi ích và hạn chế của cổng API

Như bạn có thể mong đợi, mô hình cổng API có cả ưu điểm và nhược điểm.

**BLỢI ÍCH CỦA MỘTGiao diện lập trình ứng dụng (API)CỔNG**

Một lợi ích chính của việc sử dụng API Gateway là nó đóng gói cấu trúc bên trong của ứng dụng. Thay vì phải gọi các dịch vụ cụ thể, các máy khách sẽ nói chuyện với cổng. API Gateway cung cấp cho mỗi máy khách một API dành riêng cho máy khách, giúp giảm số lần khứ hồi giữa máy khách và ứng dụng. Nó cũng đơn giản hóa mã máy khách.

**DRAWBACKS CỦA MỘTGiao diện lập trình ứng dụng (API)CỔNG**

Mẫu API gateway cũng có một số nhược điểm. Đây là một thành phần có tính khả dụng cao khác phải được phát triển, triển khai và quản lý. Cũng có nguy cơ API gateway trở thành nút thắt cổ chai trong phát triển. Các nhà phát triển phải cập nhật API gateway để có thể đưa API của dịch vụ của họ ra ngoài. Điều quan trọng là quy trình cập nhật API gateway phải nhẹ nhất có thể. Nếu không, các nhà phát triển sẽ buộc phải xếp hàng chờ để cập nhật API gateway. Tuy nhiên, bất chấp những nhược điểm này, đối với hầu hết các ứng dụng trong thế giới thực, việc sử dụng API gateway là hợp lý. Nếu cần, bạn có thể sử dụng mẫu Backends for frontends để cho phép các nhóm phát triển và triển khai API của họ một cách độc lập.

###### Netflix là một ví dụ về cổng API

Một ví dụ tuyệt vời về cổng API là Netflix API. Dịch vụ phát trực tuyến Netflix có sẵn trên hàng trăm loại thiết bị khác nhau bao gồm tivi, đầu phát Blu-ray, điện thoại thông minh và nhiều tiện ích khác. Ban đầu, Netflix đã cố gắng có một API theo phong cách phù hợp với tất cả cho dịch vụ phát trực tuyến của mình ([www.programmableweb.com/news/](http://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15) [tại sao-nghỉ-ngơi-giữ-cho-tôi-đêm-đêm/2012/05/15](http://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15)). Nhưng công ty sớm phát hiện ra rằng điều đó không hiệu quả vì có nhiều loại thiết bị khác nhau và nhu cầu khác nhau của chúng. Ngày nay, Netflix sử dụng một cổng API triển khai API riêng cho từng thiết bị. Nhóm thiết bị khách hàng phát triển và sở hữu việc triển khai API.

Trong phiên bản đầu tiên của API Gateway, mỗi nhóm khách hàng đã triển khai API của họ bằng các tập lệnh Groovy thực hiện định tuyến và biên soạn API. Mỗi tập lệnh gọi một hoặc nhiều API dịch vụ bằng các thư viện máy khách Java do các nhóm dịch vụ cung cấp. Một mặt, điều này hoạt động tốt và các nhà phát triển máy khách đã viết hàng nghìn tập lệnh. Netflix API Gateway xử lý hàng tỷ yêu cầu mỗi ngày và trung bình mỗi cuộc gọi API sẽ chuyển đến sáu hoặc bảy dịch vụ phụ trợ. Mặt khác, Netflix thấy kiến ​​trúc đơn khối này hơi cồng kềnh.

Kết quả là, Netflix hiện đang chuyển sang kiến ​​trúc cổng API tương tự như mô hình Backends cho frontends. Trong kiến ​​trúc mới này, các nhóm khách hàng viết các mô-đun API bằng NodeJS. Mỗi mô-đun API chạy vùng chứa Docker riêng của nó, nhưng các tập lệnh không gọi trực tiếp các dịch vụ. Thay vào đó, chúng gọi một "cổng API" thứ hai, nơi hiển thị các API dịch vụ bằng Netflix Falcor. Netflix Falcor là một công nghệ API thực hiện thành phần API động, khai báo và cho phép khách hàng gọi nhiều

dịch vụ sử dụng một yêu cầu duy nhất. Kiến trúc mới này có một số lợi ích. Các mô-đun API được tách biệt với nhau, giúp cải thiện độ tin cậy và khả năng quan sát, và mô-đun API của máy khách có thể mở rộng độc lập.

###### Các vấn đề thiết kế cổng API

Bây giờ chúng ta đã xem xét mô hình cổng API và những lợi ích cũng như hạn chế của nó, hãy cùng xem xét các vấn đề thiết kế cổng API khác nhau. Có một số vấn đề cần xem xét khi thiết kế cổng API:

* + - * Hiệu suất và khả năng mở rộng
      * Viết mã có thể bảo trì bằng cách sử dụng phản ứngtrừu tượng lập trình
      * Xử lý lỗi một phần
      * Trở thành một công dân tốt trong kiến ​​trúc ứng dụng Chúng ta sẽ xem xét từng yếu tố.

**PHIỆU SUẤT VÀKHẢ NĂNG MỞ RỘNG**

Cổng API là cửa trước của ứng dụng. Tất cả các yêu cầu bên ngoài trước tiên phải đi qua cổng. Mặc dù hầu hết các công ty không hoạt động ở quy mô như Netflix, nơi xử lý hàng tỷ yêu cầu mỗi ngày, nhưng hiệu suất và khả năng mở rộng của cổng API thường rất quan trọng. Một quyết định thiết kế quan trọng ảnh hưởng đến hiệu suất và khả năng mở rộng là liệu cổng API có nên sử dụng I/O đồng bộ hay không đồng bộ.

Trong mô hình I/O đồng bộ, mỗi kết nối mạng được xử lý bởi một luồng chuyên dụng. Đây là một mô hình lập trình đơn giản và hoạt động khá tốt. Ví dụ, đây là cơ sở của khuôn khổ servlet Java EE được sử dụng rộng rãi, mặc dù khuôn khổ này cung cấp tùy chọn hoàn thành yêu cầu một cách không đồng bộ. Tuy nhiên, một hạn chế của I/O đồng bộ là các luồng hệ điều hành rất nặng, do đó có giới hạn về số lượng luồng và do đó là các kết nối đồng thời mà một cổng API có thể có.

Cách tiếp cận khác là sử dụng mô hình I/O không đồng bộ (không chặn). Trong mô hình này, một luồng vòng lặp sự kiện đơn lẻ sẽ phân phối các yêu cầu I/O đến các trình xử lý sự kiện. Bạn có nhiều công nghệ I/O không đồng bộ để lựa chọn. Trên JVM, bạn có thể sử dụng một trong các khuôn khổ dựa trên NIO như Netty, Vertx, Spring Reactor hoặc JBoss Undertow. Một tùy chọn không phải JVM phổ biến là NodeJS, một nền tảng được xây dựng trên công cụ JavaScript của Chrome.

Nonblocking I/O có khả năng mở rộng hơn nhiều vì nó không có chi phí sử dụng nhiều luồng. Tuy nhiên, nhược điểm là mô hình lập trình dựa trên callback không đồng bộ phức tạp hơn nhiều. Mã khó viết, khó hiểu và khó gỡ lỗi hơn. Trình xử lý sự kiện phải trả về nhanh để tránh chặn luồng vòng lặp sự kiện.

Ngoài ra, việc sử dụng I/O không chặn có lợi ích tổng thể có ý nghĩa hay không phụ thuộc vào đặc điểm của logic xử lý yêu cầu của cổng API. Netflix đã có kết quả hỗn hợp khi viết lại Zuul, máy chủ biên của mình, để sử dụng NIO (xem[https://medium.com/netflix-](https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c) [techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c](https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c)).

Một mặt, như bạn mong đợi, việc sử dụng NIO đã giảm chi phí cho mỗi kết nối mạng, do thực tế là không còn luồng chuyên dụng cho từng kết nối nữa. Ngoài ra, một cụm Zuul chạy logic chuyên sâu về I/O—như định tuyến yêu cầu—có thông lượng tăng 25% và giảm 25% mức sử dụng CPU. Mặt khác, một cụm Zuul chạy logic chuyên sâu về CPU—như giải mã và nén—không cho thấy cải thiện nào.

**BạnTRÍCH DẪN LẬP TRÌNH PHẢN ỨNG SE**

Như đã đề cập trước đó, thành phần API bao gồm việc gọi nhiều dịch vụ phụ trợ. Một số yêu cầu dịch vụ phụ trợ hoàn toàn phụ thuộc vào các tham số của yêu cầu máy khách. Những yêu cầu khác có thể phụ thuộc vào kết quả của các yêu cầu dịch vụ khác. Một cách tiếp cận là sử dụng phương thức xử lý điểm cuối API để gọi các dịch vụ theo thứ tự được xác định bởi các phụ thuộc. Ví dụ, danh sách sau đây hiển thị trình xử lý cho yêu cầu findOrder() được viết theo cách này. Nó gọi từng dịch vụ trong bốn dịch vụ, lần lượt từng dịch vụ.

**Liệt kê 8.1 Lấy thông tin chi tiết về đơn hàng bằng cách gọi các dịch vụ phụ trợ theo trình tự**

@Bộ điều khiển nghỉ ngơi

lớp công khai OrderDetailsController { @RequestMapping("/order/{orderId}")

công khai OrderDetails getOrderDetails(@PathVariable String orderId) { OrderInfo orderInfo = orderService.findOrderById(orderId);

TicketInfo ticketInfo = dịch vụ nhà bếp

.findTicketByOrderId(mã đơn hàng);

DeliveryInfo deliveryInfo = dịch vụ giao hàng

.findDeliveryByOrderId(mã đơn hàng);

BillInfo billInfo = dịch vụ kế toán

.findBillByOrderId(mã đơn hàng);

OrderDetails orderDetails = OrderDetails.makeOrderDetails(orderInfo, ticketInfo,

thông tin giao hàng, thông tin hóa đơn);

trả lại đơn hàngChi tiết;

}

...

Nhược điểm của việc gọi các dịch vụ theo trình tự là thời gian phản hồi là tổng thời gian phản hồi của dịch vụ. Để giảm thiểu thời gian phản hồi, logic thành phần nên, bất cứ khi nào có thể, gọi các dịch vụ đồng thời. Trong ví dụ này, không có sự phụ thuộc giữa các cuộc gọi dịch vụ. Tất cả các dịch vụ nên được gọi đồng thời, điều này làm giảm đáng kể thời gian phản hồi. Thách thức là viết mã đồng thời có thể bảo trì được.

Điều này là do cách truyền thống để viết mã đồng thời, có thể mở rộng là sử dụng lệnh gọi lại. I/O không đồng bộ, theo sự kiện vốn dựa trên lệnh gọi lại. Ngay cả một Servlet

API dựa trên API composer gọi các dịch vụ đồng thời thường sử dụng lệnh gọi lại. Nócó thể thực hiện các yêu cầu đồng thời bằng cách gọi ExecutorService.submitCallable(). Vấn đề ở đây là phương thức này trả về Future, có API chặn. Một cách tiếp cận có khả năng mở rộng hơn là để một API composer gọi ExecutorService.submit (Runnable) và để mỗi Runnable gọi một lệnh gọi lại với kết quả của yêu cầu. Lệnh gọi lại tích lũy kết quả và sau khi nhận được tất cả các kết quả, nó sẽ gửi lại phản hồi cho máy khách.

Viết mã API composition bằng cách sử dụng phương pháp callback bất đồng bộ truyền thống sẽ nhanh chóng đưa bạn đến địa ngục callback. Mã sẽ bị rối, khó hiểu và dễ xảy ra lỗi, đặc biệt là khi composition yêu cầu kết hợp các yêu cầu song song và tuần tự. Một cách tiếp cận tốt hơn nhiều là viết mã API composition theo phong cách khai báo bằng cách sử dụng phương pháp tiếp cận phản ứng. Ví dụ về các trừu tượng phản ứng cho JVM bao gồm:

* + - * Java 8CompletableFutures
      * Dự án lò phản ứngĐơn âm
      * RxJava (Phần mở rộng phản ứng cho Java)Các quan sát được, được Netflix tạo ra đặc biệt để giải quyết vấn đề này trong cổng API của mình
      * thang độTương lai

Cổng API dựa trên NodeJS sẽ sử dụng JavaScript promises hoặc RxJS, là phần mở rộng phản ứng cho JavaScript. Sử dụng một trong những trừu tượng phản ứng này sẽ cho phép bạn viết mã đồng thời đơn giản và dễ hiểu. Sau trong chương này, tôi sẽ trình bày một ví dụ về phong cách mã hóa này bằng Project Reactor Monos và phiên bản 5 của Spring Framework.

**HANDLING THẤT BẠI MỘT PHẦN**

Ngoài khả năng mở rộng, cổng API cũng phải đáng tin cậy. Một cách để đạt được độ tin cậy là chạy nhiều phiên bản của cổng sau bộ cân bằng tải. Nếu một phiên bản bị lỗi, bộ cân bằng tải sẽ định tuyến các yêu cầu đến các phiên bản khác.

Một cách khác để đảm bảo rằng một API Gateway đáng tin cậy là xử lý đúng các yêu cầu không thành công và các yêu cầu có độ trễ cao không thể chấp nhận được. Khi một API Gateway gọi một dịch vụ, luôn có khả năng dịch vụ đó chậm hoặc không khả dụng. Một API Gateway có thể phải đợi rất lâu, thậm chí là vô thời hạn, để có phản hồi, điều này tiêu tốn tài nguyên và ngăn không cho API Gateway gửi phản hồi đến máy khách của mình. Một yêu cầu chưa giải quyết đến một dịch vụ không thành công thậm chí có thể tiêu tốn một tài nguyên quý giá, hạn chế như một luồng và cuối cùng dẫn đến việc API Gateway không thể xử lý bất kỳ yêu cầu nào khác. Giải pháp, như đã mô tả trong chương 3, là API Gateway sử dụng mẫu Circuit breaker khi gọi các dịch vụ.

**BTrở thành công dân tốt trong kiến ​​trúc**

Trong chương 3, tôi đã mô tả các mẫu để khám phá dịch vụ và trong chương 11, tôi đề cập đến các mẫu để quan sát. Các mẫu khám phá dịch vụ cho phép một máy khách dịch vụ, chẳng hạn như cổng API, xác định vị trí mạng của một phiên bản dịch vụ để có thể gọi phiên bản đó. Các mẫu quan sát cho phép các nhà phát triển theo dõi

hành vi của một ứng dụng và khắc phục sự cố. Một cổng API, nhưcác dịch vụ khác trong kiến ​​trúc phải triển khai các mẫu đã được chọn cho kiến ​​trúc.

#### Triển khai cổng API

Bây giờ chúng ta hãy xem cách triển khai cổng API. Như đã đề cập trước đó, trách nhiệm của cổng API như sau:

* + - *Yêu cầu định tuyến*—Định tuyến các yêu cầu đến các dịch vụ bằng các tiêu chí như phương thức yêu cầu HTTP và đường dẫn. Cổng API phải định tuyến bằng phương thức yêu cầu HTTP khi ứng dụng có một hoặc nhiều dịch vụ truy vấn CQRS. Như đã thảo luận trong chương 7, trong kiến ​​trúc như vậy, các lệnh và truy vấn được xử lý bởi các dịch vụ riêng biệt.
    - *Thành phần API*—Thực hiện mộtLẤYĐiểm cuối REST sử dụngmẫu thành phần API, được mô tả trong chương 7. Trình xử lý yêu cầu kết hợp các kết quả của việc gọi nhiều dịch vụ.
    - *Các hàm cạnh*—Đáng chú ý nhất trong số này là tính xác thực.
    - *Biên dịch giao thức*—Chuyển đổi giữa các giao thức thân thiện với máy khách và các giao thức không thân thiện với máy khách được các dịch vụ sử dụng.
    - Là một công dân tốt trong kiến ​​trúc của ứng dụng. Có một số cách khác nhau để triển khai cổng API:
    - *Sử dụng sản phẩm/dịch vụ cổng API có sẵn*—Tùy chọn này yêu cầu ít hoặc không yêu cầu

phát triển nhưng ít linh hoạt nhất. Ví dụ, một cổng API có sẵn thường không hỗ trợ thành phần API

* + - *Phát triển cổng API của riêng bạn bằng cách sử dụnghoặc là một khuôn khổ cổng API hoặc một khuôn khổ web-làm việc như điểm khởi đầu*—Đây là cách tiếp cận linh hoạt nhất, mặc dù nó đòi hỏi một số nỗ lực phát triển.

Hãy cùng xem xét các tùy chọn này, bắt đầu bằng việc sử dụng sản phẩm hoặc dịch vụ cổng API có sẵn.

###### Sử dụng sản phẩm/dịch vụ cổng API có sẵn

Một số dịch vụ và sản phẩm tự cung cấp triển khai các tính năng cổng API. HãyĐầu tiên hãy xem qua một số dịch vụ do AWS cung cấp. Sau đó, tôi sẽ thảo luận về một số sản phẩm mà bạn có thể tải xuống, cấu hình và tự chạy.

**Giao diện lập trình ứng dụng AWSCỔNG**

Cổng API AWS, một trong nhiều dịch vụ do Amazon Web Services cung cấp, là một dịch vụ triển khai và quản lý API. API cổng API AWS là một tập hợp các tài nguyên REST, mỗi tài nguyên hỗ trợ một hoặc nhiều phương thức HTTP. Bạn cấu hình APIcổng để định tuyến từng (Phương pháp, Tài nguyên) đến một dịch vụ phụ trợ. Một dịch vụ phụ trợ là một Hàm AWS Lambda, được mô tả sau trong chương 12, một dịch vụ HTTP do ứng dụng xác định hoặc một dịch vụ AWS. Nếu cần, bạn có thể cấu hình API

cổng để chuyển đổi yêu cầu và phản hồi bằng cơ chế dựa trên mẫu. Cổng API AWS cũng có thể xác thực yêu cầu.

Cổng API AWS đáp ứng một số yêu cầu đối với cổng API mà tôi đã liệt kê trước đó. Cổng API do AWS cung cấp, do đó bạn không chịu trách nhiệm về cài đặt và vận hành. Bạn cấu hình cổng API và AWS xử lý mọi thứ khác, bao gồm cả việc mở rộng quy mô.

Thật không may, AWS API Gateway có một số nhược điểm và hạn chế khiến nó không đáp ứng được các yêu cầu khác. Nó không hỗ trợ API composition, vì vậy bạn cần triển khai API composition trong các dịch vụ phụ trợ. AWS API Gateway chỉ hỗ trợ HTTP(S) với trọng tâm là JSON. Nó chỉ hỗ trợ mô hình khám phá phía máy chủ, được mô tả trong chương 3. Một ứng dụng thường sẽ sử dụng AWS Elastic Load Balancer để cân bằng tải các yêu cầu trên một tập hợp các phiên bản EC2 hoặc vùng chứa ECS. Bất chấp những hạn chế này, trừ khi bạn cần APIVề mặt cấu thành, cổng API AWS là một triển khai tốt của mô hình cổng API.

**AWSAỨNG DỤNGLOADBNGƯỜI ĐÀN BÀ**

Một dịch vụ AWS khác cung cấp chức năng giống như cổng API là AWS Application Load Balancer, đây là bộ cân bằng tải cho HTTP, HTTPS, WebSocket và HTTP/2 (<https://aws.amazon.com/blogs/aws/new-aws-application-load-balancer/>). Khi cấu hình Application Load Balancer, bạn sẽ xác định các quy tắc định tuyến để định tuyến các yêu cầu đến các dịch vụ phụ trợ, các dịch vụ này phải chạy trên các phiên bản AWS EC2.

Giống như AWS API Gateway, AWS Application Load Balancer đáp ứng một số yêu cầu đối với API Gateway. Nó triển khai chức năng định tuyến cơ bản. Nó được lưu trữ, do đó bạn không chịu trách nhiệm cài đặt hoặc vận hành. Thật không may, nó khá hạn chế. Nó không triển khai định tuyến dựa trên phương thức HTTP. Nó cũng không triển khai thành phần API hoặc xác thực. Do đó, AWS Application Load Balancer không đáp ứng các yêu cầu đối với API Gateway.

**BạnHÁT MỘTGiao diện lập trình ứng dụng (API)SẢN PHẨM GATEWAY**

Một lựa chọn khác là sử dụng sản phẩm cổng API như Kong hoặc Traefik. Đây là các gói nguồn mở mà bạn tự cài đặt và vận hành. Kong dựa trên máy chủ HTTP NGINX và Traefik được viết bằng GoLang. Cả hai sản phẩm đều cho phép bạn cấu hình các quy tắc định tuyến linh hoạt sử dụng phương thức HTTP, tiêu đề và đường dẫn để chọn dịch vụ phụ trợ. Kong cho phép bạn cấu hình các plugin triển khai các chức năng biên như xác thực. Traefik thậm chí có thể tích hợp với một số sổ đăng ký dịch vụ, được mô tả trong chương 3.

Mặc dù các sản phẩm này triển khai các chức năng biên và khả năng định tuyến mạnh mẽ, chúng vẫn có một số nhược điểm. Bạn phải tự cài đặt, cấu hình và vận hành chúng. Chúng không hỗ trợ thành phần API. Và nếu bạn muốn cổng API thực hiện thành phần API, bạn phải phát triển cổng API của riêng mình.

###### Phát triển cổng API của riêng bạn

Việc phát triển một cổng API không thực sự khó khăn. Về cơ bản, đó là một ứng dụng webproxy yêu cầu đến các dịch vụ khác. Bạn có thể xây dựng một proxy bằng cách sử dụng khuôn khổ web yêu thích của mình. Tuy nhiên, có hai vấn đề thiết kế chính mà bạn cần giải quyết:

* + - * Triển khai cơ chế xác định các quy tắc định tuyến nhằm giảm thiểu mã hóa phức tạp
      * Triển khai đúng hành vi proxy HTTP, bao gồm cách xử lý tiêu đề HTTP

Do đó, điểm khởi đầu tốt hơn để phát triển cổng API là sử dụng một khuôn khổ được thiết kế cho mục đích đó. Chức năng tích hợp của nó làm giảm đáng kể lượng mã bạn cần viết.

Chúng ta sẽ xem xét Netflix Zuul, một dự án nguồn mở của Netflix, và sau đó xem xét Spring Cloud Gateway, một dự án nguồn mở của Pivotal.

**BạnHÁTNETFLIXZUUL**

Netflix đã phát triển khuôn khổ Zuul để triển khai các chức năng biên như định tuyến, giới hạn tốc độ và xác thực (<https://github.com/Netflix/zuul>). Khung Zuul sử dụng khái niệm bộ lọc, bộ chặn yêu cầu có thể tái sử dụng tương tự như bộ lọc servlet hoặc phần mềm trung gian NodeJS Express. Zuul xử lý yêu cầu HTTP bằng cách lắp ráp một chuỗi các bộ lọc có thể áp dụng sau đó chuyển đổi yêu cầu, gọi các dịch vụ phụ trợ và chuyển đổi phản hồi trước khi gửi lại cho máy khách. Mặc dù bạn có thể sử dụng Zuul trực tiếp, nhưng sử dụng Spring Cloud Zuul, một dự án nguồn mở từ Pivotal, dễ dàng hơn nhiều. Spring Cloud Zuul xây dựng trên Zuul và thông qua quy ước-trên-cấu hình giúp phát triển máy chủ dựa trên Zuul dễ dàng đáng kể.

Zuul xử lý chức năng định tuyến và cạnh. Bạn có thể mở rộng Zuul bằng cách định nghĩa bộ điều khiển Spring MVC triển khai thành phần API. Nhưng một hạn chế lớn của Zuul là nó chỉ có thể triển khai định tuyến dựa trên đường dẫn. Ví dụ, nó không có khả năngđịnh tuyến GET /orders đến một dịch vụ và POST /orders đến một dịch vụ khác. Do đó, Zuul không hỗ trợ kiến ​​trúc truy vấn được mô tả trong chương 7.

**MỘTVỀSInCỒN ÀOGATEWAY**

Không có tùy chọn nào tôi đã mô tả cho đến nay đáp ứng được tất cả các yêu cầu. Trên thực tế, tôi đã từ bỏ việc tìm kiếm một khuôn khổ cổng API và đã bắt đầu phát triển một cổng API dựa trên Spring MVC. Nhưng sau đó tôi đã khám phá ra dự án Spring Cloud Gate-way (<https://cloud.spring.io/spring-cloud-gateway/>). Đây là một khuôn khổ cổng API được xây dựng trên một số khuôn khổ, bao gồm Spring Framework 5, Spring Boot 2 và Spring Webflux, là một khuôn khổ web phản ứng là một phần của Spring Framework 5 và được xây dựng trên Project Reactor. Project Reactor là một khuôn khổ phản ứng dựa trên NIO cho JVM cung cấp sự trừu tượng Mono được sử dụng sau một chút trong chương này.

Spring Cloud Gateway cung cấp một cách đơn giản nhưng toàn diện để thực hiện những việc sau:

* Chuyển hướng yêu cầu đến các dịch vụ phụ trợ.
* Triển khai trình xử lý yêu cầu thực hiện tổng hợp API.
* Xử lý các chức năng tiên tiến như xác thực.

Hình 8.8 hiển thị các phần chính của cổng API được xây dựng bằng cách sử dụng khuôn khổ này.



Đơn hàng«Gói API»

mono<ServerResponse> getOrderDetails(ServerRequest){

...

}

Gói proxy từ xa

tìmOrderById()

«proxy» Dịch vụ đặt hàng

lấy chi tiết đơn hàng()

Người xử lý đơn hàng

NHẬN/đơn hàng/{orderId}

=>

Trình xử lý đơn hàng::getOrderDetails

«@Bean» orderHandlerRouting

«@Bean» đặt hàngProxyRouting

đơn hàng\*

=>

[http://dịch vụ đặt hàng](http://orderservice/)

«Cấu hình Spring»Cấu hình đơn hàng

tĩnh void main(String[]args){

...

}

«@SpringBootApplication»Ứng dụng ApiGateway

mono<Thông tin đơn hàng> tìm đơn hàng theo ID()(ID đơn hàng){

...WebClient

.lấy()

[.url("http://dịch-vụ-đặt-hàng/..."}](http://order-service/)

}

|  |  |
| --- | --- |
| «người đại diện»  .... | «proxy» Dịch vụ giao hàng |
|  | tìmGiao hàng theo đơn hàng() |

Cổng đám mây mùa xuân

Mùa xuân 5

Mùa xuân webFlux

Dự án lò phản ứng

**Hình 8.8 Kiến trúc của một cổng API được xây dựng bằng Spring Cloud Gateway**

Cổng API bao gồm các gói sau:

* ApiGatewayChính*bưu kiện*—Xác định chương trình chính cho cổng API.
* *Một hoặc nhiều gói API*—Gói API triển khai một tập hợp các điểm cuối API. Ví dụ,Đơn hànggói thực hiệnĐặt hàng-điểm cuối API liên quan.
* *Gói proxy*—Bao gồm các lớp proxy được các gói API sử dụng để gọi các dịch vụ.

Cấu hình đơn hànglớp định nghĩa các bean Spring chịu trách nhiệm định tuyến các yêu cầu liên quan đến Order. Một quy tắc định tuyến có thể khớp với một số kết hợp củaPhương thức HTTP, tiêu đề và đường dẫn. @Bean orderProxyRoutes định nghĩa các quy tắc ánh xạ các hoạt động API tới các URL dịch vụ phụ trợ. Ví dụ, nó định tuyến các đường dẫn bắt đầu bằng /orders tới Order Service.

CácorderHandlers @Beanxác định các quy tắc ghi đè lên các quy tắc được xác định bởiđặt hàng- ProxyRoutes. Các quy tắc này ánh xạ các hoạt động API với các phương thức xử lý, tương đương với các phương thức điều khiển Spring MVC của Spring WebFlux. Ví dụ:Người xử lý đơn hànglập bản đồ hoạt độngNHẬN /đơn hàng/{orderId}đếnOrderHandlers::get- OrderDetails()phương pháp.

CácNgười xử lý đơn hànglớp thực hiện nhiều phương thức xử lý yêu cầu khác nhau, chẳng hạn nhưOrderHandlers::getOrderDetails().Phương pháp này sử dụng API composition để lấy thông tin chi tiết về đơn hàng (đã mô tả trước đó). Các phương pháp xử lý gọi các dịch vụ phụ trợ bằng cách sử dụng các lớp proxy từ xa, chẳng hạn nhưDịch vụ đặt hàng. Lớp này định nghĩa các phương thức để gọiDịch vụ đặt hàng.

Chúng ta hãy xem xét mã, bắt đầu bằngCấu hình đơn hànglớp học.

**TANH TAỒĐẠI HỌCCLỚP HÌNH ẢNH**

CácCấu hình đơn hànglớp, được hiển thị trong danh sách 8.2, là Spring@Cấu hìnhlớp học.Nó định nghĩa mùa xuân@Đậuthực hiện/đơn hàngđiểm cuối. Cácđặt hàng- ProxyRoutingVàorderHandlerRouting @Beanssử dụng định tuyến Spring WebFluxDSL để xác định định tuyến yêu cầu.orderHandlers @Beantriển khai trình xử lý yêu cầu thực hiện biên soạn API.

**Liệt kê 8.2 Mùa xuân@Đậuthực hiện/đơn hàngđiểm cuối**

@Configuration @EnableConfigurationProperties(OrderDestinations.class) lớp công khai OrderConfiguration {

@Đậu

public RouteLocator orderProxyRouting(OrderDestinationorderDestinations) { trả về Routes.locator()

.route("đơn hàng")

.uri(orderDestinations.orderServiceUrl)

.predicate(đường dẫn("/đơn hàng").hoặc(đường dẫn("/đơn hàng/\*")))

.Và()

}

@Đậu

...

.xây dựng();

**Theo mặc định, định tuyến tất cả các yêu cầu có đường dẫn bắt đầu bằng /orders đến URL orderDestinations.orderServiceUrl.**

**Định tuyến GET**

public RouterFunction<ServerResponse> orderHandlerRouting(OrderHandlers orderHandlers) {

trả về RouterFunctions.route(GET("/orders/{orderId}"),

orderHandlers::getOrderDetails);

**/orders/{orderId} tới orderHandlers:: getOrderDetails.**

}

@Đậu

public OrderHandlers orderHandlers(OrderServicedịch vụ đặt hàng,

Dịch vụ nhà bếp Dịch vụ nhà bếp, Dịch vụ giao hàng Dịch vụ giao hàng, Dịch vụ kế toán Dịch vụ kế toán) {

trả về OrderHandlers mới (orderService,dịch vụ nhà bếp,

dịch vụ giao hàng, dịch vụ kế toán);

}

**@Bean, thực hiệncái**

} **phong tục xử lý yêu cầu lý luận**

Đặt hàngĐiểm đến, được hiển thị trong danh sách sau đây, là một mùa xuân@Configuration- Thuộc tínhlớp cho phép cấu hình bên ngoài các URL dịch vụ phụ trợ.

**Liệt kê 8.3 Cấu hình bên ngoài của URL dịch vụ phụ trợ**

@ConfigurationProperties(tiền tố = "order.destinations") lớp công khai OrderDestinations {

@KhôngNull

công khai String orderServiceUrl;

công khai String getOrderServiceUrl() { trả về orderServiceUrl;

}

công khai void setOrderServiceUrl(String orderServiceUrl) { this.orderServiceUrl = orderServiceUrl;

}

...

}

Ví dụ, bạn có thể chỉ địnhĐịa chỉ URLcủaDịch vụ đặt hànghoặc là nhưđặt hàng

.destinations.orderServiceUrlthuộc tính trong một tập tin thuộc tính hoặc như một hệ điều hànhbiến môi trường tem,ĐẶT\_ĐỊA\_TÍCH\_ĐẶT\_DỊCH\_VỤ\_URL.

**TANH TAỒĐẠI HỌCHLỚP ANDLERS**

Lớp OrderHandlers, được hiển thị trong danh sách sau, định nghĩa các phương thức xử lý yêu cầu triển khai hành vi tùy chỉnh, bao gồm cả thành phần API. Ví dụ, phương thức getOrder- Details() thực hiện thành phần API để truy xuất thông tin về đơn hàng. Lớp này được tiêm một số lớp proxy thực hiện yêu cầu đến các dịch vụ phụ trợ.

**Liệt kê 8.4Người xử lý đơn hànglớp thực hiện logic xử lý yêu cầu tùy chỉnh.**

lớp công khai OrderHandlers { riêng tư OrderService orderService;

Dịch vụ bếp riêng Dịch vụ bếp riêng;

Dịch vụ giao hàng riêngDịch vụ giao hàng; Dịch vụ kế toán riêngDịch vụ kế toán;

public OrderHandlers(OrderService orderService,

Dịch vụ nhà bếp Dịch vụ nhà bếp, Dịch vụ giao hàng Dịch vụ giao hàng, Dịch vụ kế toán Dịch vụ kế toán) {

this.orderService = dịch vụ đặt hàng; this.kitchenService = dịch vụ nhà bếp; this.deliveryService = dịch vụ giao hàng; this.accountingService = dịch vụ kế toán;

}

công khai Mono<ServerResponse> getOrderDetails(ServerRequest serverRequest) { Chuỗi orderId = serverRequest.pathVariable("orderId");

Mono<OrderInfo> orderInfo = orderService.findOrderById(orderId);

Mono<Tùy chọn<Thông tin vé>> ticketInfo = kitchenService

.findTicketByOrderId(mã đơn hàng)

.map(Tùy chọn::of)

.onErrorReturn(Tùy chọn. empty());

Mono<Tùy chọn<Thông tin giao hàng>> deliveryInfo = deliveryService

.findDeliveryByOrderId(orderId)

.map(Tùy chọn::of)

.onErrorReturn(Tùy chọn. empty());

**Chuyển đổi TicketInfo thành Optional<TicketInfo>.**

**Nếu việc gọi dịch vụ không thành công, hãy trả về Optional.empty().**

Mono<Tùy chọn<BillInfo>> billInfo = accountingService

.findBillByOrderId(mã đơn hàng)

.map(Tùy chọn::of)

.onErrorReturn(Tùy chọn. empty());

Đơn vị <Tuple4 <Thông tin đơn hàng,Tùy chọn<TicketInfo>,

**Kết hợp bốn giá trị thành một giá trị duy nhất, Tuple4.**

<Thông tin giao hàng> tùy chọn, <Thông tin hóa đơn>>> tùy chọn kết hợp = Mono.when(orderInfo, ticketInfo, deliveryInfo, billInfo);

Mono<OrderDetails> orderDetails = combined.map(OrderDetails::makeOrderDetails);

**Chuyển đổi Tuple4 thành OrderDetails.**

trả về orderDetails.flatMap(person -> ServerResponse.ok()

.contentType(MediaType.APPLICATION\_JSON)

.body(fromObject(người)));

} **Biến đổi**

**Chi tiết đơn hàng vào trong**

} **một ServerResponse.**

Phương thức getOrderDetails() triển khai thành phần API để lấy đơn hàngchi tiết. Nó được viết theo phong cách có thể mở rộng, phản ứng bằng cách sử dụng trừu tượng Mono, đượcdo Project Reactor cung cấp. Mono, một loại Java 8 CompletableFuture phong phú hơn, chứa kết quả của một hoạt động bất đồng bộ là giá trị hoặc ngoại lệ. Nó có API phong phú để chuyển đổi và kết hợp các giá trị được trả về bởi các hoạt động bất đồng bộ. Bạn có thể sử dụng Monos để viết mã đồng thời theo một phong cách

đơn giản và dễ hiểu. Trong ví dụ này, phương thức getOrderDetails()gọi bốn dịch vụ song song và kết hợp các kết quả để tạo ra một đối tượng Chi tiết đơn hàng.

Phương thức getOrderDetails() lấy ServerRequest, là biểu diễn của Spring Web-Flux cho yêu cầu HTTP, làm tham số và thực hiện các thao tác sau:

**1**Nó trích xuấtcáiMã đơn hàngtừ con đường.

**2**Nó gọi bốn dịch vụ một cách không đồng bộ thông qua proxy của chúng, trả vềĐơn âm.Để cải thiện tính khả dụng,lấy chi tiết đơn hàng()xử lý kết quả của tất cả các dịch vụ ngoại trừDịch vụ đặt hàngnhư tùy chọn. Nếu mộtBệnh tăng bạch cầu đơn nhânđược trả về bởi một dịch vụ tùy chọn chứa một ngoại lệ, lệnh gọi đếnonErrorReturn()biến đổi nó thành mộtBệnh tăng bạch cầu đơn nhânchứa một cái gì đó trống rỗngKhông bắt buộc.

**3** Nó kết hợp các kết quả một cách không đồng bộ bằng cách sử dụng Mono.when(), trả về một

Đơn<Tuple4>chứa bốn giá trị.

**4**Nó biến đổiĐơn<Tuple4>vào mộtMono<Chi tiết đơn hàng>bằng cách gọiĐặt hàng-Chi tiết::makeOrderDetails.

**5** Nó biến đổiChi tiết đơn hàngvào mộtPhản hồi của máy chủ, cái màlà biểu diễn Spring WebFlux của phản hồi JSON/HTTP.

Như bạncó thể thấy, vì getOrderDetails() sử dụng Monos, nó đồng thời gọi các dịch vụ và kết hợp các kết quả mà không sử dụng các lệnh gọi lại lộn xộn, khó đọc. Chúng ta hãy xem xét một trong các proxy dịch vụ trả về kết quả của lệnh gọi API dịch vụ được gói trong Mono.

**TANH TAỒĐẠI HỌCSLỚP DỊCH VỤ**

CácDịch vụ đặt hànglớp, được hiển thị trong danh sách sau, là một proxy từ xa choDịch vụ đặt hàng. Nó gợi lênDịch vụ đặt hàngsử dụng mộtWebClient, là ứng dụng khách HTTP phản ứng Spring Web-Flux.

**Liệt kê 8.5Dịch vụ đặt hànglớp—một proxy từ xa choĐặt hàngDịch vụ**

@Dịch vụ

lớp công khai OrderService {

riêng tư OrderDestinations orderDestinations; máy khách WebClient riêng tư;

công khai OrderService(OrderDestinations orderDestinations, WebClient khách hàng)

{

this.orderDestinations = orderDestinations; this.client = khách hàng;

}

công khai Mono<OrderInfo> findOrderById(String orderId) { Mono<ClientResponse> phản hồi = khách hàng

.lấy()

**Gọi dịch vụ.**

}

.uri(orderDestinations.orderServiceUrl + "/đơn hàng/{orderId}", orderId)

.trao đổi();

trả về response.flatMap(resp -> resp.bodyToMono(OrderInfo.class));

}

**Chuyển đổi nội dung phản hồi thành OrderInfo.**

Cáctìm kiếmOrder()phương pháp lấy lạiThông tin đơn hàngcho một đơn đặt hàng. Nó sử dụngWebClientđể làmyêu cầu HTTP đếnDịch vụ đặt hàngvà hủy tuần tự hóa phản hồi JSONđến mộtThông tin đơn hàng.WebClientcó API phản ứng và phản hồi được gói trong mộtBệnh tăng bạch cầu đơn nhân. Cáctìm kiếmOrder()phương pháp sử dụngBản đồ phẳng()để chuyển đổiĐơn <Phản hồi của khách hàng>vào mộtMono<Thông tin đơn hàng>. Như tên gọi của nó,bodyToMono()phương pháp trả về nội dung phản hồi dưới dạngBệnh tăng bạch cầu đơn nhân.

**TANH TAMỘTTÍNH TOÁNGATEWAYMỘTLỚP ỨNG DỤNG**

Lớp ApiGatewayApplication, được hiển thị trong danh sách sau, triển khai phương thức main() của API gateway. Đây là lớp chính Spring Boot chuẩn.

**Liệt kê 8.6chủ yếu()phương pháp cho cổng API**

@SpringBootConfiguration @EnableAutoConfiguration @EnableGateway @Import(OrdersConfiguration.class) lớp công khai ApiGatewayApplication {

public static void main(String[] args) { SpringApplication.run(ApiGatewayApplication.class, args);

}

}

Chú thích @EnableGateway nhập cấu hình Spring cho khung Spring Cloud Gateway.

Spring Cloud Gateway là một khuôn khổ tuyệt vời để triển khai một cổng API. Nó cho phép bạn cấu hình proxy cơ bản bằng cách sử dụng một DSL quy tắc định tuyến đơn giản, súc tích. Cũng dễ dàng định tuyến các yêu cầu đến các phương thức xử lý thực hiện biên dịch giao thức và biên dịch API. Spring Cloud Gateway được xây dựng bằng cách sử dụng các khuôn khổ Spring Framework 5 và Project Reactor có khả năng mở rộng, phản ứng. Nhưng có một tùy chọn hấp dẫn khác để phát triển cổng API của riêng bạn: GraphQL, một khuôn khổ cung cấp ngôn ngữ truy vấn dựa trên đồ thị. Hãy cùng xem cách thức hoạt động của nó.

###### Triển khai cổng API bằng GraphQL

Hãy tưởng tượng rằng bạn chịu trách nhiệm triển khai GET của API Gateway của FTGO

điểm cuối /orders/{orderId},trả về chi tiết đơn hàng. Trên bề mặt, việc triển khai điểm cuối này có vẻ đơn giản. Nhưng như đã mô tả trong phần 8.1, điểm cuối này truy xuất dữ liệu từ nhiều dịch vụ. Do đó, bạn cần sử dụng

Mẫu thành phần API và viết mã để gọi các dịch vụ và kết hợp các kết quả.

Một thách thức khác, đã đề cập trước đó, là các máy khách khác nhau cần dữ liệu hơi khác nhau. Ví dụ, không giống như ứng dụng di động, ứng dụng SPA trên máy tính để bàn hiển thị xếp hạng của bạn cho đơn hàng. Một cách để tùy chỉnh dữ liệu được trả về bởi điểm cuối, như được mô tả trong chương 3, là cung cấp cho máy khách khả năng chỉ định dữ liệu họ cần. Ví dụ, điểm cuối có thể hỗ trợ các tham số truy vấn như tham số mở rộng, chỉ định các tài nguyên liên quan để trả về và tham số trường, chỉ định các trường của từng tài nguyên để trả về. Tùy chọn khác là xác định nhiều phiên bản của điểm cuối này như một phần của việc áp dụng mẫu Backends cho frontends. Đây là rất nhiều công việc chỉ đối với một trong nhiều điểm cuối API mà API Gateway của FTGO cần triển khai.

Việc triển khai một cổng API với một API REST hỗ trợ tốt nhiều tập hợp máy khách khác nhau rất tốn thời gian. Do đó, bạn có thể cân nhắc sử dụng một khuôn khổ API dựa trên đồ thị, chẳng hạn như GraphQL, được thiết kế để hỗ trợ việc truy xuất dữ liệu hiệu quả. Ý tưởng chính với các khuôn khổ API dựa trên đồ thị là, như hình 8.9 cho thấy, API của máy chủ bao gồm một lược đồ dựa trên đồ thị. Lược đồ dựa trên đồ thị xác định một tập hợp các nút (kiểu), có các thuộc tính (trường) và mối quan hệ với các nút khác. Máy khách truy xuất dữ liệu bằng cách thực hiện một truy vấn chỉ định dữ liệu cần thiết theo các nút của đồ thị và các thuộc tính cũng như mối quan hệ của chúng. Do đó, máy khách có thể truy xuất dữ liệu cần thiết trong một lần khứ hồi duy nhất đến cổng API.

Truy vấn

Sơ đồ đồ thị

Truy vấn

Truy vấn

Đặt hàng

Sơ đồ

=>

Bản đồ dịch vụ

Truy vấn

Truy vấn

Khách hàng

Khung API dựa trên đồ thị

Cổng API

Dịch vụ giao hàng

Vận chuyển

Dịch vụ nhà hàng

Nhà hàng

Dịch vụ đặt hàng

Đặt hàng

Dịch vụ khách hàng

Người tiêu dùng

Vận chuyển

Nhà hàng

Người tiêu dùng

**Hình 8.9 API của cổng API bao gồm một lược đồ dựa trên đồ thị được ánh xạ tớidịch vụ. Một máy khách đưa ra truy vấn để lấy nhiều nút đồ thị. Khung API dựa trên đồ thị thực hiện truy vấn bằng cách lấy dữ liệu từ một hoặc nhiều dịch vụ.**

Công nghệ API dựa trên đồ thị có một số lợi ích quan trọng. Nó cho phép khách hàng kiểm soát dữ liệu được trả về. Do đó, việc phát triển một API duy nhất linh hoạt

đủ để hỗ trợ nhiều khách hàng khác nhau trở nên khả thi. Một lợi ích khác là mặc dùAPI linh hoạt hơn nhiều, cách tiếp cận này làm giảm đáng kể nỗ lực phát triển. Đó là vì bạn viết mã phía máy chủ bằng cách sử dụng một khung thực thi truy vấn được thiết kế để hỗ trợ biên soạn và dự báo API. Giống như thể, thay vì buộc khách hàng truy xuất dữ liệu thông qua các thủ tục được lưu trữ mà bạn cần viết và duy trì, bạn để họ thực hiện các truy vấn đối với cơ sở dữ liệu cơ bản.

**Công nghệ API theo lược đồ**

Hai công nghệ API dựa trên đồ thị phổ biến nhất là GraphQL ([http://graphql.org](http://graphql.org/)) và Netflix Falcor (<http://netflix.github.io/falcor/>).Netflix Falcor mô hình hóa phía máy chủdữ liệu dưới dạng biểu đồ đối tượng JSON ảo. Máy khách Falcor truy xuất dữ liệu từ máy chủ Falcor bằng cách thực hiện truy vấn truy xuất các thuộc tính của đối tượng JSON đó. Máy khách cũng có thể cập nhật các thuộc tính. Trong máy chủ Falcor, các thuộc tính của biểu đồ đối tượng được ánh xạ tới các nguồn dữ liệu phụ trợ, chẳng hạn như các dịch vụ có REST API. Máy chủ xử lý yêu cầu đặt hoặc lấy các thuộc tính bằng cách gọi một hoặc nhiều nguồn dữ liệu phụ trợ.

GraphQL, được phát triển bởi Facebook và phát hành vào năm 2015, là một công nghệ API dựa trên đồ thị phổ biến khác. Công nghệ này mô hình hóa dữ liệu phía máy chủ dưới dạng đồ thị của các đối tượng có các trường và tham chiếu đến các đối tượng khác. Đồ thị đối tượng được ánh xạ tới các nguồn dữ liệu phụ trợ. Các máy khách GraphQL có thể thực hiện các truy vấn để truy xuất dữ liệu và các đột biến để tạo và cập nhật dữ liệu. Không giống như Netflix Falcor, là một triển khai, GraphQL là một tiêu chuẩn, với các máy khách và máy chủ có sẵn cho nhiều ngôn ngữ, bao gồm NodeJS, Java và Scala.

Apollo GraphQL là một triển khai JavaScript/NodeJS phổ biến ([www.apollographql](http://www.apollographql.com/)

[.com](http://www.apollographql.com/)). Đây là nền tảng bao gồm máy chủ và máy khách GraphQL. Apollo GraphQL triển khai một số tiện ích mở rộng mạnh mẽ cho đặc tả GraphQL, chẳng hạn như đăng ký đẩy dữ liệu đã thay đổi đến máy khách.

Phần này nói về cách phát triển cổng API bằng Apollo GraphQL. Tôi chỉ đề cập đến một số tính năng chính của GraphQL và Apollo GraphQL. Để biết thêm thông tin, bạn nên tham khảo tài liệu GraphQL và Apollo GraphQL.

Cổng API dựa trên GraphQL, được hiển thị trong hình 8.10, được viết bằng JavaScript sử dụng khung web NodeJS Express và máy chủ Apollo GraphQL. Các phần chính của thiết kế như sau:

* *Sơ đồ GraphQL*—Sơ đồ GraphQL xác định mô hình dữ liệu phía máy chủ và các truy vấn mà nó hỗ trợ.
* *Các hàm giải quyết*—Các hàm giải quyết ánh xạ các thành phần của lược đồ tới nhiều dịch vụ phụ trợ khác nhau.
* *Các lớp proxy*—Các lớp proxy gọi ứng dụng FTGOdịch vụ.

Ngoài ra còn có một lượng nhỏ mã keo tích hợp máy chủ GraphQL với khung web Express. Hãy cùng xem xét từng phần, bắt đầu với lược đồ GraphQL.



Apollo

đồ thịQLkhách hàng

[http://.../graphql?query={](http://./graphql?query)đơn hàng(consumerId:1){orde rId,nhà hàng{id}}}

Cổng API FTGO

hàm resolveOrder(\_. {orderId}, ngữ cảnh){

trở vềNngữ cảnh.orderServiceProxy.tìm kiếmOrder(orderI d);

}

loại Query{ orders(consumerId:Int!): [Order] order(orderId : int!): Order

người tiêu dùng(consumerId : int!): Người tiêu dùng

}

loại Order { orderId: ID, consumerId: Int, consumer: Consumer

nhà hàng: Nhà hàng deliveryInfo : DeliveryInfo

...

const giải quyết = {Truy vấn: {

lệnh: giải quyếtĐơn hàng,thứ tự: giải quyếtThứ tự,

...

},

Đặt hàng:{

người tiêu dùng: resolveOrderNgười tiêu dùng, nhà hàng: resolveOrderRestaurant, deliveryInfo: resolveOrderDeliveryInfo

},

...

;

gọi

gọi

gọi

gọi

Giao hàngDịch vụProxy

Nhà hàngDịch vụProxy

Dịch vụ đặt hàngProxy

Dịch vụ khách hàngProxy

Khung web Express

Dịch vụ giao hàng

Dịch vụ nhà hàng

Dịch vụ đặt hàng

Dịch vụ khách hàng

chức năng giải quyếtOrderDeliveryInfo({orderId}, các đối số, ngữ cảnh) { trả về ngữ cảnh.deliveryServiceProxy.findDeliveryF hoặcOrder(orderId)

}

Công cụ GraphQL của Apollo

**Hình 8.10 Thiết kế của Cổng API FTGO dựa trên GraphQL**

**DĐỊNH NGHĨA AGRAPHQLSƠ ĐỒ**

GraphQL API tập trung vào một lược đồ, bao gồm một tập hợp các kiểu định nghĩa cấu trúc của mô hình dữ liệu phía máy chủ và các hoạt động, chẳng hạn như truy vấn, mà máy khách có thể thực hiện. GraphQL có một số loại kiểu khác nhau. Mã ví dụ trong phần này chỉ sử dụng hai loại kiểu: kiểu đối tượng, là

cách chính để xác định mô hình dữ liệu và enum, tương tự như Java enum. Một kiểu đối tượng có tên và một tập hợp các trường được đặt tên, có kiểu. Một trường có thể là một kiểu vô hướng, chẳng hạn như số, chuỗi hoặc enum; một danh sách các kiểu vô hướng; một tham chiếu đến một kiểu đối tượng khác; hoặc một tập hợp các tham chiếu đến một kiểu đối tượng khác. Mặc dù giống với một trường của một lớp hướng đối tượng truyền thống, một trường GraphQL về mặt khái niệm là một hàm trả về một giá trị. Nó có thể có các đối số, cho phép một máy khách GraphQL tùy chỉnh dữ liệu mà hàm trả về.

GraphQL cũng sử dụng các trường để định nghĩa các truy vấn được hỗ trợ bởi lược đồ. Bạn định nghĩa các truy vấn của lược đồ bằng cách khai báo một kiểu đối tượng, theo quy ước được gọi là Truy vấn. Mỗi trường của đối tượng Truy vấn là một truy vấn được đặt tên, có một tập hợp các tham số tùy chọn và một kiểu trả về. Tôi thấy cách định nghĩa truy vấn này hơi khó hiểu khi lần đầu tiên gặp phải, nhưng nó giúp ghi nhớ rằng trường GraphQL là một hàm. Nó sẽ trở nên rõ ràng hơn khi chúng ta xem cách các trường được kết nối với các nguồn dữ liệu phụ trợ.

Danh sách sau đây cho thấymột phần của lược đồ cho cổng API FTGO dựa trên GraphQL. Nó định nghĩa một số loại đối tượng. Hầu hết các loại đối tượng tương ứng vớiCác thực thể Consumer, Order và Restaurant của ứng dụng FTGO. Nó cũng có một đối tượng Querykiểu định nghĩa các truy vấn của lược đồ.

**Liệt kê 8.7 Sơ đồ GraphQL cho cổng API FTGO**

loại Truy vấn {

đơn hàng(consumerId : Int!): [Đặt hàng] đơn hàng(orderId : Int!): Đặt hàng người tiêu dùng(consumerId : Int!): Người tiêu dùng

**Xác định các truy vấn mà máy khách có thể thực hiện**

}

loại Người tiêu dùng { id: ID

firstName: Chuỗi lastName: Chuỗi orders: [Order]

}

loại Order { orderId: ID, consumerId : Int, consumer: Consumer

nhà hàng: Nhà hàng

**ID duy nhất cho Người tiêu dùng**

**Một người tiêu dùng códanh sách các đơn hàng.**

deliveryInfo : Thông tin giao hàng

...

}

loại Nhà hàng { id: ID

tên: Chuỗi

...

}

loại DeliveryInfo { trạng thái : DeliveryStatus

estimateDeliveryTime : Int assignedCourier :String

}

enum DeliveryStatus { CHUẨN BỊ SẴN SÀNG ĐỂ NHẬN HÀNG ĐÃ NHẬN HÀNG

ĐÃ GIAO HÀNG

}

Mặc dù có cú pháp khác nhau, các kiểu đối tượng Consumer, Order, Restaurant và Delivery-Info có cấu trúc tương tự như các lớp Java tương ứng. Một điểm khác biệt là kiểu ID, biểu diễn một mã định danh duy nhất.

Sơ đồ này định nghĩa ba truy vấn:

* đơn hàng()—Trả vềĐơn hàngcho những gì đã chỉ địnhNgười tiêu dùng
* đặt hàng()—Trả về giá trị đã chỉ địnhĐặt hàng
* người tiêu dùng()—Trả về giá trị đã chỉ địnhNgười tiêu dùng

Các truy vấn này có vẻ không khác gì các điểm cuối REST tương đương, nhưng GraphQL cung cấp cho máy khách quyền kiểm soát dữ liệu được trả về rất lớn. Để hiểu lý do, chúng ta hãy xem cách máy khách thực hiện các truy vấn GraphQL.

**EXẾP HÀNHGRAPHQLTHẮC MẮC**

Lợi ích chính của việc sử dụng GraphQL là ngôn ngữ truy vấn của nó cung cấp cho máy khách quyền kiểm soát đáng kinh ngạc đối với dữ liệu được trả về. Máy khách thực hiện truy vấn bằng cách gửi yêu cầu có chứa tài liệu truy vấn đến máy chủ. Trong trường hợp đơn giản, tài liệu truy vấn chỉ định tên truy vấn, giá trị đối số và các trường của kết quảđối tượng để trả về. Đây là một truy vấn đơn giản để lấy firstName và lastName của người tiêu dùng có ID cụ thể:

truy vấn { người tiêu dùng(consumerId:1)

{

Tênhọ

}

}

**Chỉ định truy vấn được gọi là người tiêu dùng, truy xuất một người tiêu dùng**

**Các trường của Người tiêu dùng để trả về**

Truy vấn này trả về các trường của Người tiêu dùng đã chỉ định.

Sau đây là truy vấn phức tạp hơn trả về người tiêu dùng, đơn hàng của họ và ID cũng như tên nhà hàng của mỗi đơn hàng:

truy vấn {

người tiêu dùng(consumerId:1) { id

Tênhọ

đơn hàng { orderId nhà hàng {

tên id

}

deliveryInfo { ước tínhDeliveryTime tên

}

}

}

}

Truy vấn này yêu cầu máy chủ trả về nhiều hơn là chỉ các trường của Consumer. Nó truy xuất Orders của Consumer và restaurant của mỗi Order. Như bạn có thể thấy, một máy khách GraphQL có thể chỉ định chính xác dữ liệu cần trả về, bao gồm các trường của các đối tượng có liên quan gián tiếp.

Ngôn ngữ truy vấn linh hoạt hơn so với vẻ bề ngoài ban đầu của nó. Đó là vì truy vấn là một trường của đối tượng Query và một tài liệu truy vấn chỉ định trường nào trong số các trường đó mà máy chủ sẽ trả về. Những ví dụ đơn giản này truy xuất một trường duy nhất, nhưng một tài liệu truy vấn có thể thực hiện nhiều truy vấn bằng cách chỉ định nhiều trường. Đối với mỗi trường, tài liệu truy vấn cung cấp các đối số của trường và chỉ định những trường nào của đối tượng kết quả mà nó quan tâm. Sau đây là một truy vấn truy xuất hai người dùng khác nhau:

truy vấn {

c1: người tiêu dùng (consumerId:1) { id, firstName, lastName} c2: người tiêu dùng (consumerId:2) { id, firstName, lastName}

}

Trong tài liệu truy vấn này, c1 và c2 là những gì GraphQL gọi là alias. Chúng được sử dụng để phân biệt giữa hai Consumer trong kết quả, nếu không thì cả hai đều được gọi là consumer. Ví dụ này truy xuất hai đối tượng cùng loại, nhưng một máy khách có thể truy xuất một số đối tượng có loại khác nhau.

Một lược đồ GraphQL định nghĩahình dạng của dữ liệu và các truy vấn được hỗ trợ. Để hữu ích, nó phải được kết nối với nguồn dữ liệu. Hãy cùng xem cách thực hiện điều đó.

**CKẾT NỐI SƠ ĐỒ VỚI DỮ LIỆU**

Khi máy chủ GraphQL thực hiện một truy vấn, nó phải truy xuất dữ liệu được yêu cầu từ một hoặc nhiều kho dữ liệu. Trong trường hợp ứng dụng FTGO, máy chủ GraphQL phải gọi các API của các dịch vụ sở hữu dữ liệu. Bạn liên kết một lược đồ GraphQL với các nguồn dữ liệu bằng cách đính kèm các hàm phân giải vào các trường của các kiểu đối tượng được lược đồ xác định. Máy chủ GraphQL triển khai mẫu thành phần API bằng cách gọi các hàm phân giải để truy xuất dữ liệu, trước tiên là cho truy vấn cấp cao nhất, sau đó là đệ quy cho các trường của đối tượng hoặc các đối tượng kết quả.

Chi tiết về cách các hàm phân giải được liên kết với lược đồ phụ thuộc vào máy chủ GraphQL bạn đang sử dụng. Liệt kê 8.8 cho thấy cách xác định các trình phân giải

khi sử dụng máy chủ Apollo GraphQL. Bạn tạo một đối tượng JavaScript lồng nhau kép. Mỗi thuộc tính cấp cao nhất tương ứng với một loại đối tượng, chẳng hạn như Query và Order. Mỗi thuộc tính cấp hai, chẳng hạn như Order.consumer, xác định hàm giải quyết của trường.

**Liệt kê 8.8 Đính kèm các hàm giải quyết vào các trường của lược đồ GraphQL**

const resolvers = { Truy vấn: {

đơn hàng: giải quyếtĐơn hàng, người tiêu dùng: giải quyếtNgười tiêu dùng, đơn hàng: giải quyếtĐơn hàng

},

Đặt hàng: {

người tiêu dùng: giải quyếtOrderConsumer,

**Bộ giải quyết cho truy vấn đơn hàng**

**Bộ giải quyết cho trường người tiêu dùng của Đơn hàng**

...

};

nhà hàng: giải quyếtĐặt hàngNhà hàng, giao hàngThông tin: giải quyếtĐặt hàngGiao hàngThông tin

Một hàm giải quyết có ba tham số:

* *Sự vật*—Đối với trường truy vấn cấp cao nhất, chẳng hạn nhưgiải quyếtĐơn hàng, sự vậtlà Một đối tượng gốc thường bị bỏ qua bởi hàm giải quyết. Nếu không,sự vậtlà giá trị được trả về bởi trình giải quyết cho đối tượng cha. Ví dụ, trình giải quyếtchức năng choĐặt hàng.người tiêu dùngtrường được truyền giá trị trả về bởiĐặt hànghàm giải quyết của.
* *Đối số truy vấn*—Những thông tin này được cung cấp trong tài liệu truy vấn.
* *Bối cảnh*—Trạng thái toàn cục của quá trình thực hiện truy vấn mà tất cả các trình phân giải có thể truy cập. Ví dụ, nó được sử dụng để truyền thông tin người dùng và các phụ thuộc cho các trình phân giải.

Một hàm giải quyết có thể gọi một dịch vụ duy nhất hoặc nó có thể triển khai mẫu thành phần API và truy xuất dữ liệu từ nhiều dịch vụ. Máy chủ Apollo GraphQLhàm resolver trả về một Promise, đó là phiên bản của JavaScriptcủa Java's Completable- Future. Lời hứa chứa đối tượng (hoặc danh sách các đối tượng) mà hàm giải quyết đã lấy từ kho dữ liệu. Công cụ GraphQL bao gồm giá trị trả về trong đối tượng kết quả.

Hãy xem một vài ví dụ. Đây là hàm resolveOrders(), là trình giải quyết cho truy vấn orders:

hàm resolveOrders(\_, { consumerId }, context) { trả về context.orderServiceProxy.findOrders(consumerId);

}

Chức năng này có đượcDịch vụ đặt hàngProxytừbối cảnhvà gọi nó để lấy đơn hàng của người tiêu dùng. Nó bỏ qua tham số đầu tiên của nó. Nó truyềnngười tiêu dùngIdtranh luận-ment, được cung cấp bởi tài liệu truy vấn, đểOrderServiceProxy.tìm kiếm đơn hàng(). Cáctìm đơn hàng()phương pháp lấy lại đơn đặt hàng của người tiêu dùng từLịch sử đơn hàngDịch vụ.



Đây làgiải quyếtĐặt hàngNhà hàng()chức năng, là trình giải quyết cho

Đặt hàng.nhà hàngtrường lấy thông tin nhà hàng của đơn hàng:

hàm resolveOrderRestaurant({restaurantId}, args, context) { trả về context.restaurantServiceProxy.findRestaurant(restaurantId);

}

Tham số đầu tiên của nó làĐặt hàng. Nó gợi lênRestaurantServiceProxy.tìmNhà hàng()

vớiĐặt hàng'Snhà hàngId, được cung cấp bởigiải quyếtĐơn hàng().

GraphQL sử dụng thuật toán đệ quy đểthực thi các hàm giải quyết. Đầu tiên, nó thực thi hàm giải quyết cho truy vấn cấp cao nhất được chỉ định bởi tài liệu Truy vấn. Tiếp theo, đối với mỗi đối tượng được truy vấn trả về, nó lặp qua các trường được chỉ định trong tài liệu Truy vấn. Nếu một trường có trình giải quyết, nó sẽ gọi trình giải quyết với đối tượng và các đối số từ tài liệu Truy vấn. Sau đó, nó đệ quy trên đối tượng hoặc các đối tượng được trả về bởi trình giải quyết đó.

Hình 8.11 cho thấy cách thuật toán này thực hiện truy vấn lấy thông tin đơn hàng của người tiêu dùng và thông tin giao hàng của từng đơn hàng cũng như nhà hàng. Đầu tiên, công cụ GraphQLgọigiải quyếtNgười tiêu dùng(), trong đó lấy lạiNgười tiêu dùng. Tiếp theo, nó gọigiải quyết- ConsumerOrders(), là trình giải quyết cho cái Người tiêu dùng.đơn hàngcánh đồng cái đó trả về các đơn đặt hàng của người tiêu dùng. Sau đó, công cụ GraphQL lặp lại thông quaĐơn hàng, gọi các trình giải quyết choĐặt hàng.nhà hàngVàĐơn hàng.giao hàngThông tincánh đồng.

**Sơ đồ Tài liệu truy vấn**

Đối số truy vấn được chuyển đến trình giải quyết

truy vấn{

người tiêu dùng(consumerId:1){nhận dạng

Tênhọđơn đặt hàng{

orderId nhà hàng{

tên id

}

deliveryInfo{ ước tính thời gian giao hàng tên

}

}

}

}

**Các hàm giải quyết**

loại Query{ consumer(consumerId:int!): Consumer

}

loại Đặt hàng {

...

nhà hàng: Nhà hàng deliveryInfo : DeliveryInfo

...

người tiêu dùng = giải quyếtNgười tiêu dùng(..., 1)

đơn hàng = giải quyếtConsumerOrders(người tiêu dùng)

giải quyếtOrderRestaurant(đơn hàng, ...)

giải quyếtOrderDeliveryInfo(đơn hàng)

**Hình 8.11 GraphQL thực hiện truy vấn bằng cách đệ quy gọi các hàm giải quyết cho các trường được chỉ định trong tài liệu Truy vấn. Đầu tiên, nó thực hiện trình giải quyết cho truy vấn, sau đó đệ quy gọi các trình giải quyết chocác trường trong hệ thống phân cấp đối tượng kết quả.**

Kết quả của việc thực hiện các trình giải quyết là một đối tượng Consumer được điền dữ liệu đã truy xuấttừ nhiều dịch vụ.

Bây giờ chúng ta hãy xem cách tối ưu hóa việc thực thi các trình phân giải bằng cách sử dụng xử lý hàng loạtvà lưu trữ đệm.

**ỒTẢI PTIMIZINGSỬ DỤNG BATCHING VÀ Caching**

GraphQL có khả năng thực thi một số lượng lớn trình phân giảikhi thực hiện truy vấn. Vì máy chủ GraphQL thực hiện từng trình phân giải độc lập, nên có nguy cơ hiệu suất kém do quá nhiều vòng lặp đến các dịch vụ. Ví dụ, hãy xem xét truy vấn truy xuất người tiêu dùng, đơn hàng của họ và nhà hàng của đơn hàng. Nếu có N đơn hàng, thì một triển khai đơn giản sẽ thực hiện một cuộc gọi đến Dịch vụ người tiêu dùng, một cuộc gọi đến Dịch vụ lịch sử đơn hàng và sau đó là N cuộc gọi đến Dịch vụ nhà hàng.Mặc dù công cụ GraphQL thường sẽ thực hiện các cuộc gọi đến Restaurant Service song song, vẫn có nguy cơ hiệu suất kém. May mắn thay, bạn có thể sử dụng một số kỹ thuật để cải thiện hiệu suất.

Một tối ưu hóa quan trọng là sử dụng kết hợp xử lý hàng loạt phía máy chủ vàbộ nhớ đệm. Việc xử lý hàng loạt biến N lệnh gọi đến một dịch vụ, chẳng hạn như Dịch vụ nhà hàng, thành một lệnh gọi duy nhất truy xuất một loạt N đối tượng. Bộ nhớ đệm sử dụng lại kết quả của lần truy xuất trước đó của cùng một đối tượng để tránh thực hiện lệnh gọi trùng lặp không cần thiết. Sự kết hợp của việc xử lý hàng loạt và bộ nhớ đệm làm giảm đáng kể số lần khứ hồi đến các dịch vụ phụ trợ.

Một máy chủ GraphQL dựa trên NodeJS có thểsử dụng mô-đun DataLoader để triển khai việc tạo hàng loạt và lưu trữ đệm (<https://github.com/facebook/dataloader>). Nó hợp nhất các tải xảy ra trong một lần thực thi duy nhất của vòng lặp sự kiện và gọi hàm tải hàng loạt mà bạn cung cấp. Nó cũng lưu trữ các lệnh gọi để loại bỏ các tải trùng lặp. Danh sách sau đây cho thấy cách RestaurantServiceProxy có thể sử dụng DataLoader. Phương thức findRestaurant() tải một Nhà hàng thông qua DataLoader.

**Liệt kê 8.9 Sử dụngBộ nạp dữ liệuđể tối ưu hóa các cuộc gọi đếnNhà hàngDịch vụ**

const DataLoader = yêu cầu('dataloader'); lớp RestaurantServiceProxy {

người xây dựng() {

this.dataLoader =

Trình tải dữ liệu mới (nhà hàngIds =>

**Tạo một DataLoader sử dụng batchFindRestaurants() làm hàm tải hàng loạt.**

này.batchFindRestaurants(restaurantIds));

}

**Tải Nhà hàng được chỉ định**

findRestaurant(restaurantId) {

trở lạithis.dataLoader.load(restaurantId);

**thông qua DataLoader.**

}

}

batchFindRestaurants(restaurantIds){

...

}

**Tải một loạt Nhà hàng.**

Nhà hàngDịch vụProxyvà do đó,Bộ nạp dữ liệuđược tạo ra cho mỗi yêu cầu, vì vậykhông có khả năng nàoBộ nạp dữ liệutrộn lẫn dữ liệu của nhiều người dùng khác nhau.

Bây giờ chúng ta hãy xem cách tích hợp công cụ GraphQL với một khuôn khổ web để máy khách có thể sử dụng.

**TÔITÍCH HỢPMỘTBÒGRAPHQLMÁY CHỦ VỚIEXPRESS**

Máy chủ Apollo GraphQL thực hiện các truy vấn GraphQL. Để máy khách có thể gọi nó, bạn cần tích hợp nó với một khuôn khổ web. Máy chủ Apollo GraphQL hỗ trợ một số khuôn khổ web, bao gồm Express, một khuôn khổ web NodeJS phổ biến.

Danh sách 8.10 cho thấy cách sử dụng máy chủ Apollo GraphQL trong ứng dụng Expresstion. Chức năng chính là graphqlExpress, được cung cấp bởi mô-đun apollo-server-express. Nó xây dựng một trình xử lý yêu cầu Express thực thi các truy vấn GraphQL theo một lược đồ. Ví dụ này cấu hình Express để định tuyến các yêu cầu đến GET

Các điểm cuối /graphql và POST /graphql của trình xử lý yêu cầu GraphQL này. Nó cũng tạo ra một ngữ cảnh GraphQL chứa các proxy, giúp chúng có sẵn cho các trình phân giải.

**Liệt kê 8.10 Tích hợp máy chủ GraphQL với khung web Express**

const {graphqlExpress} = yêu cầu ("apollo-server-express");

const typeDefs = gql` kiểu Truy vấn {

lệnh: giải quyếtĐơn hàng,

...

}

loại Người tiêu dùng {

...

const resolvers = { Truy vấn: {

...

}

}

**Xác định lược đồ GraphQL.**

**Xác định trình giải quyết.**

**Kết hợp lược đồ với các trình phân giải để tạo ra một tệp thực thi**

const schema = makeExecutableSchema({ typeDefs, resolvers });

**lược đồ.**

const ứng dụng = express();

chức năng makeContextWithDependencies(req) {

const orderServiceProxy = new OrderServiceProxy();

**Đưa kho lưu trữ vào ngữ cảnh để trình giải quyết có thể sử dụng.**

const consumerServiceProxy = new ConsumerServiceProxy(); const restaurantServiceProxy = new RestaurantServiceProxy();

...

trả về {orderServiceProxy, consumerServiceProxy, restaurantServiceProxy, ...};

} **Tạo trình xử lý yêu cầu nhanh**

**thực hiện các truy vấn GraphQL**

hàm makeGraphQLHandler() { trả về graphqlExpress(req => {

**so với lược đồ thực thi.**

});

}

trả về {schema: schema, context: makeContextWithDependencies(req)}

ứng dụng.post('/graphql', bodyParser.json(), makeGraphQLHandler());

ứng dụng.get('/graphql',makeGraphQLHandler());ứng dụng.listen(PORT);

**Tuyến đường POST /graphql và GET**

**/graphql điểm cuối đến**

**Máy chủ GraphQL.**

Ví dụ này không xử lý các mối quan tâm như bảo mật, nhưng những mối quan tâm đó sẽ dễ dàng triển khai. Ví dụ, cổng API có thể xác thực người dùng bằng Passport, một khuôn khổ bảo mật NodeJS được mô tả trong chương 11. makeContext-Hàm WithDependencies() sẽ truyền thông tin người dùng đến từng hàm tạo của kho lưu trữ để chúng có thể truyền bá thông tin người dùng đến các dịch vụ.

Bây giờ chúng ta hãy xem cách một khách hàngcó thể gọi máy chủ này để thực thi các truy vấn GraphQL.

**TVIẾT AGRAPHQLKHÁCH HÀNG**

Có một vài cách khác nhau mà ứng dụng khách có thể gọi máy chủ GraphQL. Vì máy chủ GraphQL có API dựa trên HTTP, nên ứng dụng kháchcó thểsử dụng thư viện HTTP để thực hiện các yêu cầu, chẳng hạn như GET http://localhost:3000/ graphql?query={orders(consumerId:1){orderId,restaurant{id}}}'. Tuy nhiên, sử dụng thư viện máy khách GraphQL dễ hơn, thư viện này xử lý việc định dạng đúng các yêu cầu và thường cung cấp các tính năng như lưu trữ đệm phía máy khách.

Danh sách sau đây hiển thị lớp FtgoGraphQLClient, là một máy khách dựa trên GraphQL đơn giản cho ứng dụng FTGO. Hàm tạo của nó khởi tạo Apollo-Client, được cung cấp bởi thư viện máy khách Apollo GraphQL. Lớp FtgoGraphQL-Client định nghĩa phương thức findConsumer() sử dụng máy khách để truy xuất tên của người tiêu dùng.

**Liệt kê 8.11 Sử dụng máy khách Apollo GraphQL để thực hiện các truy vấn**

lớp FtgoGraphQLClient {

người xây dựng(...) {

this.client = new ApolloClient({ ... });

}

tìmConsumer(consumerId) { trả về this.client.query({



biến: { cid: consumerId}, truy vấn: gql`

truy vấn foo($cid : Int!) { người tiêu dùng(consumerId: $cid) {

id tên họ

**Cung cấp giá trị của $cid.**

**Định nghĩa $cid là biến có kiểu Int.**

**Đặt giá trị của tham số truy vấn consumerid thành $cid.**

}

} `,

})

}

}

Lớp FtgoGraphQLClient có thể định nghĩa nhiều phương thức truy vấn khác nhau, chẳng hạn như find- Consumer(). Mỗi phương thức thực hiện một truy vấn để lấy chính xác dữ liệu mà máy khách cần.

***Bản tóm tắt* 291**

Phần này hầu như không đề cập đến khả năng của GraphQL. Tôi hy vọng mình đã chứng minh được rằng GraphQL là một giải pháp thay thế rất hấp dẫn cho một API gateway REST truyền thống hơn. Nó cho phép bạn triển khai một API đủ linh hoạt để hỗ trợ nhiều loại máy khách khác nhau. Do đó, bạn nên cân nhắc sử dụng GraphQL để triển khai API gateway của mình.

#### Bản tóm tắt

* + Các máy khách bên ngoài của ứng dụng của bạn thường truy cập các dịch vụ của ứng dụng thông qua cổng API. Cổng API cung cấp cho mỗi máy khách một API tùy chỉnh. Nó chịu trách nhiệm định tuyến yêu cầu, biên soạn API, biên dịch giao thức và triển khai các chức năng biên như xác thực.
  + Ứng dụng của bạn có thể có một cổng API duy nhất hoặc có thể sử dụng mẫu Backends for frontends, mẫu này định nghĩa một cổng API cho từng loại máy khách. Ưu điểm chính của mẫu Backends for frontends là nó mang lại cho nhóm máy khách quyền tự chủ lớn hơn, vì họ phát triển, triển khai và vận hành cổng API của riêng mình.
  + Có nhiều công nghệ bạn có thể sử dụng để triển khai API gateway, bao gồm các sản phẩm API gateway có sẵn. Ngoài ra, bạn có thể phát triển API gateway của riêng mình bằng cách sử dụng một khuôn khổ.
  + Spring Cloud Gateway làmột khuôn khổ tốt, dễ sử dụng để phát triển một cổng API. Nó định tuyến các yêu cầu bằng bất kỳ thuộc tính yêu cầu nào, bao gồm phương thức và đường dẫn. Spring Cloud Gateway có thể định tuyến một yêu cầu trực tiếp đến một dịch vụ phụ trợ hoặc đến một phương thức xử lý tùy chỉnh. Nó được xây dựng bằng các khuôn khổ Spring Framework 5 và Project Reactor có khả năng mở rộng, phản ứng. Bạn có thể viết các trình xử lý yêu cầu tùy chỉnh của mình theo phong cách phản ứng bằng cách sử dụng, ví dụ, ProjectLò phản ứngBệnh tăng bạch cầu đơn nhânsự trừu tượng.
  + GraphQL, một khuôn khổ cung cấp ngôn ngữ truy vấn dựa trên đồ thị, là một nền tảng tuyệt vời khác để phát triển API Gateway. Bạn viết một lược đồ hướng đồ thị để mô tả mô hình dữ liệu phía máy chủ và các truy vấn được hỗ trợ của nó. Sau đó, bạn ánh xạ lược đồ đó với các dịch vụ của mình bằng cách viết các trình phân giải, truy xuất dữ liệu. Các máy khách dựa trên GraphQL thực hiện các truy vấn đối với lược đồ chỉ định chính xác dữ liệu mà máy chủ sẽ trả về. Do đó, một cổng API dựa trên GraphQL có thể hỗ trợ nhiều máy khách khác nhau.

*Kiểm tra các dịch vụ vi mô:*

*Phần 1*

***Chương này bao gồm***

* Các chiến lược thử nghiệm hiệu quả cho các dịch vụ vi mô
* Sử dụng mocks và stubs để kiểm tra một phần tử phần mềm một cách riêng biệt
* Sử dụng kim tự tháp thử nghiệm để xác định nơi cầntập trung nỗ lực thử nghiệm
* Kiểm thử đơn vị các lớp bên trong một dịch vụ

FTGO, giống như nhiều tổ chức khác, đã áp dụng cách tiếp cận truyền thống để thử nghiệm. Kiểm thử chủ yếu là hoạt động diễn ra sau khi phát triển. Các nhà phát triển FTGO ném mã của họ qua tường cho nhóm QA, những người xác minh rằng phần mềm hoạt động như mong đợi. Hơn nữa, hầu hết các thử nghiệm của họ được thực hiện thủ công. Thật đáng buồn, cách tiếp cận thử nghiệm này đã bị hỏng—vì hai lý do:

* + *Kiểm tra thủ công cực kỳ kém hiệu quả*—Bạn không bao giờ nên yêu cầu con người làm những gì máy móc có thể làm tốt hơn. So với máy móc, con người chậm hơn và không thể làm việc 24/7. Bạn sẽ không thể cung cấp phần mềm nhanh chóng và an toàn nếu bạn dựa vào thử nghiệm thủ công. Điều cần thiết là bạn phải viết các thử nghiệm tự động.
  + *Việc thử nghiệm được thực hiện quá muộn trong quá trình giao hàng*—Chắc chắn có một vai trò cho các bài kiểm tra phê bình một ứng dụng sau khi nó được viết, nhưng kinh nghiệm đã chỉ ra rằng các bài kiểm tra đó là không đủ. Một cách tiếp cận tốt hơn nhiều là các nhà phát triển

**292**

**293**

viết các bài kiểm tra tự động như một phần của quá trình phát triển. Nó cải thiện năng suất của họ vì, ví dụ, họ sẽ có các bài kiểm tra cung cấp phản hồi ngay lập tức trong khi chỉnh sửa mã.

Về vấn đề này, FTGO là một tổ chức khá điển hình. Báo cáo Xu hướng thử nghiệm của Sauce Labs năm 2018 vẽ nên một bức tranh khá ảm đạm về tình trạng tự động hóa thử nghiệm ([https://](https://saucelabs.com/resources/white-papers/testing-trends-for-2018) [Saulabs.com/resources/white-papers/testing-trends-for-2018](https://saucelabs.com/resources/white-papers/testing-trends-for-2018)). Nó mô tả cách chỉ26% các tổ chức chủ yếu là tự động hóa và chỉ có 3% là hoàn toàn tự động! Việc phụ thuộc vào thử nghiệm thủ công không phải do thiếu công cụ và khuôn khổ.

Ví dụ, JUnit, một khuôn khổ kiểm thử Java phổ biến, được phát hành lần đầu tiên vào năm 1998. Lý do cho việc thiếu các bài kiểm thử tự động chủ yếu là do văn hóa: "Kiểm thử là công việc của QA", "Đó không phải là cách sử dụng thời gian tốt nhất của nhà phát triển", v.v. Việc phát triển một bộ kiểm thử chạy nhanh nhưng hiệu quả và có thể bảo trì là một thách thức cũng không giúp ích gì. Và, một ứng dụng lớn, đơn khối điển hình cực kỳ khó kiểm thử.

Một động lực chính để sử dụng kiến ​​trúc microservice là, như đã mô tả trong chương 2, cải thiện khả năng kiểm thử. Tuy nhiên, đồng thời, tính phức tạp của kiến ​​trúc microservice đòi hỏi bạn phải viết các bài kiểm tra tự động. Hơn nữa, một số khía cạnh của việc kiểm thử microservice rất khó khăn. Đó là vì chúng ta cần xác minh rằng các dịch vụ có thể tương tác chính xác trong khi giảm thiểu số lượng các bài kiểm tra đầu cuối chậm, phức tạp và không đáng tin cậy khi khởi chạy nhiều dịch vụ.

Chương này là chương đầu tiên trong hai chương về thử nghiệm. Đây là phần giới thiệu về thử nghiệm. Chương 10 đề cập đến các khái niệm thử nghiệm nâng cao hơn. Hai chương này dài, nhưng khi kết hợp lại, chúng đề cập đến các ý tưởng và kỹ thuật thử nghiệm thiết yếu đối với phát triển phần mềm hiện đại nói chung và kiến ​​trúc vi dịch vụ nói riêng.

Tôi bắt đầu chương này bằng cách mô tả các chiến lược kiểm thử hiệu quả cho ứng dụng dựa trên microservices. Các chiến lược này cho phép bạn tự tin rằng phần mềm của mình hoạt động, đồng thời giảm thiểu độ phức tạp của bài kiểm thử và thời gian thực hiện. Sau đó, tôi mô tả cách viết một loại kiểm thử cụ thể cho các dịch vụ của bạn: kiểm thử đơn vị. Chương 10 đề cập đến các loại kiểm thử khác: tích hợp, thành phần và đầu cuối.

Chúng ta hãy bắt đầu bằng cách xem xét các chiến lược thử nghiệm cho dịch vụ vi mô.

**Tại sao cần giới thiệu về thử nghiệm?**

Bạn có thể thắc mắc tại sao chương này lại bao gồm phần giới thiệu về các khái niệm kiểm thử cơ bản. Nếu bạn đã quen thuộc với các khái niệm như kim tự tháp kiểm thử và các loại kiểm thử khác nhau, hãy thoải mái đọc lướt qua chương này và chuyển sang chương tiếp theo, tập trung vào các chủ đề kiểm thử dành riêng cho dịch vụ vi mô. Nhưng dựa trên kinh nghiệm tư vấn và đào tạo khách hàng trên toàn thế giới, một điểm yếu cơ bản của nhiều tổ chức phát triển phần mềm là thiếu kiểm thử tự động. Đó là bởi vì nếu bạn muốn cung cấp phần mềm nhanh chóng và đáng tin cậy, thì*hoàn toàn cần thiết*để thực hiện thử nghiệm tự động.Đó là cách duy nhất để có một đoạn ngắn*thời gian dẫn*, đó là thời gian cần thiết để đưa mã đã cam kết vào sản xuất. Có lẽ quan trọng hơn nữa, thử nghiệm tự động là điều cần thiếtvì nó buộc bạn phải phát triển một ứng dụng có thể kiểm tra được. Thường rất khó để đưa thử nghiệm tự động vào một ứng dụng vốn đã lớn và phức tạp. Nói cách khác,Con đường nhanh nhất dẫn đến địa ngục đơn điệu là không viết các bài kiểm tra tự động.

#### Chiến lược thử nghiệm cho kiến ​​trúc vi dịch vụ

Giả sử bạn đã thực hiện thay đổi cho ứng dụng FTGODịch vụ đặt hàng. Tất nhiên, bước tiếp theo là bạn chạy mã của mình và xác minh rằng thay đổi hoạt động chính xác. Một lựa chọn là kiểm tra thay đổi theo cách thủ công. Đầu tiên, bạn chạyDịch vụ đặt hàngvà tất cả phụ thuộc vào nócác dịch vụ cơ sở hạ tầng như cơ sở dữ liệu và các dịch vụ ứng dụng khác. Sau đó, bạn "kiểm tra" dịch vụ bằng cách gọi API của dịch vụ hoặc sử dụng giao diện người dùng của ứng dụng FTGO. Nhược điểm của cách tiếp cận này là nó là một cách chậm và thủ công để kiểm tra mã của bạn.

Một lựa chọn tốt hơn nhiều là có các bài kiểm tra tự động mà bạn có thể chạy trong quá trình phát triển. Quy trình phát triển của bạn nên là: chỉnh sửa mã, chạy thử nghiệm (lý tưởng nhất là chỉ với một lần nhấn phím), lặp lại. Các bài kiểm tra chạy nhanh sẽ nhanh chóng cho bạn biết liệu các thay đổi của bạn có hoạt động trong vòng vài giây hay không. Nhưng làm thế nào để bạn viết các bài kiểm tra chạy nhanh? Và chúng có đủ hay bạn cần các bài kiểm tra toàn diện hơn? Đây là loại câu hỏi mà tôi trả lời trong phần này và các phần khác trong chương này.

Tôi bắt đầu phần này bằng phần tổng quan về các khái niệm kiểm thử tự động quan trọng. Chúng ta sẽ xem xét mục đích của việc kiểm thử và cấu trúc của một bài kiểm thử thông thường. Tôi sẽ đề cập đến các loại kiểm thử khác nhau mà bạn sẽ cần phải viết. Tôi cũng mô tả kim tự tháp kiểm thử, cung cấp hướng dẫn có giá trị về nơi bạn nên tập trung nỗ lực kiểm thử của mình. Sau khi đề cập đến các khái niệm kiểm thử, tôi thảo luận về các chiến lược để kiểm thử các dịch vụ siêu nhỏ. Chúng ta sẽ xem xét các thách thức riêng biệt của việc kiểm thử các ứng dụng có kiến ​​trúc dịch vụ siêu nhỏ. Tôi mô tả các kỹ thuật bạn có thể sử dụng để viết các bài kiểm thử đơn giản hơn và nhanh hơn, nhưng vẫn hiệu quả, cho các dịch vụ siêu nhỏ của mình.

Chúng ta hãy cùng xem xét các khái niệm thử nghiệm.

###### Tổng quan về thử nghiệm

Trong chương này, tôi tập trung vào thử nghiệm tự động và tôi sử dụng thuật ngữ thử nghiệm như một cách viết tắt cho thử nghiệm tự động. Wikipedia định nghĩa trường hợp thử nghiệm hoặc thử nghiệm như sau:

*Một trường hợp thử nghiệm là một tập hợp các đầu vào thử nghiệm, điều kiện thực hiện và kết quả mong đợi được phát triển cho một mục tiêu cụ thể, chẳng hạn như để thực hiện một đường dẫn chương trình cụ thể hoặc để xác minh sự tuân thủcó yêu cầu cụ thể.*

<https://en.wikipedia.org/wiki/Test_case>

Nói cách khác, mục đích của một bài kiểm tra, như hình 9.1 cho thấy, là để xác minh hành vi của Hệ thống đang được kiểm tra (SUT). Trong định nghĩa này, hệ thống là một thuật ngữ hoa mỹ có nghĩa làphần tử phần mềm đang được thử nghiệm. Nó có thể là thứ gì đó nhỏ như một lớp, lớn như toàn bộ ứng dụng hoặc thứ gì đó ở giữa, chẳng hạn như một nhóm các lớp hoặc một dịch vụ riêng lẻ. Một tập hợp các bài kiểm tra liên quan tạo thành một bộ kiểm tra.

Trước tiên, hãy xem khái niệm về bài kiểm tra tự động. Sau đó, tôi sẽ thảo luận về các loại bài kiểm tra khác nhau mà bạn cần viết. Sau đó, tôi sẽ thảo luận về kim tự tháp kiểm tra, mô tả tỷ lệ tương đối của các loại bài kiểm tra khác nhau mà bạn nên viết.

**Hình 9.1 Mục tiêu của thử nghiệm là xác minh hành vi của hệ thốngđang được thử nghiệm. Một SUT có thể nhỏ như một lớp hoặc lớn như toàn bộ một ứng dụng.**

Xác minh hành vi của

Bài kiểm trabộ

Bài kiểm tra

Hệ thốngĐang kiểm tra (SUT)

**TKIỂM TRA TỰ ĐỘNG RITING**

Các bài kiểm tra tự động thường được viết bằng cách sử dụng một khuôn khổ kiểm tra. JUnit, choVí dụ, là một khuôn khổ kiểm thử Java phổ biến. Hình 9.2 cho thấy cấu trúc của một bài kiểm thử tự động. Mỗi bài kiểm thử được triển khai bởi một phương pháp kiểm thử, thuộc về một lớp kiểm thử.

Đồ đạc

ĐẠI HỌC

Lớp kiểm tra

Bài kiểm traphương pháp

Thực hiện

Bài kiểm traphương pháp

Bài kiểm traphương pháp

Người chạy thử nghiệm

|  |  |  |
| --- | --- | --- |
| Cài đặt |  | |
|  |  |
| Thực hiện |
|  |  |
| Xác minh |
| Tháo dỡ |
|  | |

**Hình 9.2 Mỗi bài kiểm tra tự động được triển khai bằng một phương pháp kiểm tra, thuộc về một lớp kiểm tra. Một bài kiểm tra bao gồm bốn giai đoạn:*cài đặt*, khởi tạo bộ thử nghiệm, là mọi thứ cần thiết để chạy thử nghiệm;*thực hiện*, điều này sẽ gọi SUT;*xác minh*, xác minh kết quả của bài kiểm tra; và*tháo dỡ*, giúp làm sạch thiết bị thử nghiệm.**

Một bài kiểm tra tự động thường bao gồm bốn giai đoạn ([http://xunitpatterns.com/](http://xunitpatterns.com/Four%20Phase%20Test.html) [Four%20Phase%20Test.html](http://xunitpatterns.com/Four%20Phase%20Test.html)):

**1***Cài đặt*—Khởi tạo đồ gá thử nghiệm, bao gồm SUT và các phần phụ thuộc của nó, đến trạng thái ban đầu mong muốn. Ví dụ, tạo lớp đang thử nghiệm và khởi tạo nó đến trạng thái cần thiết để nó thể hiện hành vi mong muốn.

**2** *Bài tập*—Gọi SUT—ví dụ, gọi một phương thức trên lớp đang được kiểm tra.

**3***Xác minh*—Đưa ra các khẳng định về kết quả của lệnh gọi và trạng thái của SUT. Ví dụ, xác minh giá trị trả về của phương thức và trạng thái mới của lớp đang được kiểm tra.

**4***Tháo dỡ*—Dọn dẹp đồ gá thử nghiệm, nếu cần. Nhiều thử nghiệm bỏ qua giai đoạn này, nhưng một số loại thử nghiệm cơ sở dữ liệu sẽ, ví dụ, khôi phục giao dịch được khởi tạo bởi giai đoạn thiết lập.

Để giảm trùng lặp mã và đơn giản hóa các bài kiểm tra, một lớp kiểm tra có thể có các phương thức thiết lập được chạy trước một phương thức kiểm tra và các phương thức hủy bỏ được chạy sau đó. Một bộ kiểm tra là một tập hợp các lớp kiểm tra. Các bài kiểm tra được thực hiện bởi một trình chạy kiểm tra.

**TESTING SỬ DỤNG MOCKS VÀ STUBS**

Một SUT thường có sự phụ thuộc. Vấn đề với sự phụ thuộc là chúng có thểgấp lạivà làm chậm các bài kiểm tra. Ví dụ, lớp OrderController gọi Order-Service, về cơ bản phụ thuộc vào nhiều dịch vụ ứng dụng và dịch vụ cơ sở hạ tầng khác. Sẽ không thực tế khi kiểm tra lớp OrderController bằng cách chạy một phần lớn hệ thống. Chúng ta cần một cách để kiểm tra SUT một cách riêng biệt.

Giải pháp, như hình 9.3 cho thấy, là thay thế các phụ thuộc của SUT bằng các đối tượng kiểm thử kép. Đối tượng kiểm thử kép là một đối tượng mô phỏng hành vi của phụ thuộc.

Kiểm tra

Hệ thốngĐang kiểm tra (SUT)

Chậm, phức tạpBài kiểm tra

Sự phụ thuộc

Thay thế bằng

Kiểm tra

Hệ thốngĐang kiểm tra (SUT)

Nhanh hơn, đơn giản hơnBài kiểm tra

Kiểm tra gấp đôi

**Hình 9.3 Thay thế một phụ thuộc bằng một bản sao thử nghiệm cho phép SUTđược thử nghiệm riêng biệt. Thử nghiệm này đơn giản hơn và nhanh hơn.**

Có hai loại test double: stubs và mocks. Các thuật ngữ stubs và mocks thường được sử dụng thay thế cho nhau, mặc dù chúng có hành vi hơi khác nhau. Stub là test double trả về giá trị cho SUT. Mock là test double mà test sử dụng để xác minh rằng SUT gọi đúng dependency. Ngoài ra, mock thường là stub.

Sau này trong chương này, bạn sẽ thấy các ví dụcủa các bản sao thử nghiệm đang hoạt động. Ví dụ,phần 9.2.5 cho thấy cách kiểm tra OrderControllerlớp riêng biệt bằng cách sử dụng một đối tượng kiểm tra cho lớp OrderService. Trong ví dụ đó, đối tượng kiểm tra OrderService được triển khai bằng Mockito, một khuôn khổ đối tượng giả phổ biến cho Java. Chương 10 cho thấy cách kiểm tra Order Service bằng cách sử dụng các đối tượng kiểm tra cho các dịch vụ khác mà nó gọi. Các đối tượng kiểm tra đó phản hồi các thông báo lệnh được gửi bởi Order Service.

Bây giờ chúng ta hãy xem xét các loại thử nghiệm khác nhau.

**TCÁC LOẠI KIỂM TRA KHÁC NHAU**

Có nhiều loại thử nghiệm khác nhau. Một số thử nghiệm, chẳng hạn như thử nghiệm hiệu suất và thử nghiệm khả năng sử dụng, xác minh rằng ứng dụng đáp ứng các yêu cầu về chất lượng dịch vụ. Trong chương này, tôi tập trung vào các thử nghiệm tự động xác minh các khía cạnh chức năng của ứng dụng hoặc dịch vụ. Tôi mô tả cách viết bốn loại thử nghiệm khác nhau:

* *Kiểm tra đơn vị*—Kiểm tra một phần nhỏ của dịch vụ, chẳng hạn như một lớp.
* *Kiểm tra tích hợp*—Xác minh rằng dịch vụ có thể tương tác với các dịch vụ cơ sở hạ tầng như cơ sở dữ liệu và các dịch vụ ứng dụng khác.
* *Kiểm tra thành phần*—Kiểm tra chấp nhận cho từng dịch vụ riêng lẻ.
* *Kiểm tra đầu cuối*—Kiểm tra chấp nhận toàn bộ ứng dụng.

Chúng khác nhau chủ yếu ở phạm vi. Ở một đầu của quang phổ là các bài kiểm tra đơn vị, xác minh hành vi của phần tử chương trình có ý nghĩa nhỏ nhất. Đối với ngôn ngữ hướng đối tượng như Java, đó là một lớp. Ở đầu kia của quang phổ là các bài kiểm tra đầu cuối, xác minh hành vi của toàn bộ ứng dụng. Ở giữa là các bài kiểm tra thành phần, kiểm tra các dịch vụ riêng lẻ. Các bài kiểm tra tích hợp, như bạn sẽ thấy trong chương tiếp theo, có phạm vi tương đối nhỏ, nhưng chúng phức tạp hơn các bài kiểm tra đơn vị thuần túy. Phạm vi chỉ là một cách để mô tả các bài kiểm tra. Một cách khác là sử dụng góc phần tư kiểm tra.

**Kiểm tra đơn vị thời gian biên dịch**

Kiểm thử là một phần không thể thiếu của quá trình phát triển. Quy trình phát triển hiện đại là chỉnh sửa mã, sau đó chạy thử nghiệm. Hơn nữa, nếu bạn là người thực hành Phát triển theo hướng kiểm thử (TDD), bạn sẽ phát triển một tính năng mới hoặc sửa lỗi bằng cách đầu tiên là viết một bài kiểm tra không thành công rồi viết mã để làm cho nó thành công. Ngay cả khi bạn không phải là người tuân thủ TDD, một cách tuyệt vờisửa lỗi là viết một bài kiểm tra để tái tạo lỗi và sau đó viết mã để sửa lỗi đó.

Các bài kiểm tra mà bạn chạy như một phần của quy trình làm việc này được gọi là*thời gian biên dịch*kiểm tra. Trong một IDE hiện đại, chẳng hạn như IntelliJ IDEA hoặc Eclipse, bạn thường không biên dịch mã của mình thành một bước riêng biệt. Thay vào đó, bạn sử dụng một lần nhấn phím để biên dịch mã và chạy các bài kiểm tra. Để duy trì luồng, các bài kiểm tra này cần được thực hiện nhanh chóng—lý tưởng nhất là không quá vài giây.

**BạnHÁT BẢNG KIỂM TRA ĐỂ PHÂN LOẠI CÁC BÀI KIỂM TRA**

Một cách tốt để phân loại các bài kiểm tra là bảng kiểm tra của Brian Marick ([www.exampler.com/old-](http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1) [blog/2003/08/21/#agile-testing-project-1](http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1)). Phần tư thử nghiệm, được hiển thị trong hình 9.4, phân loại các thử nghiệm theo hai chiều:

* *Cho dù bài kiểm tra hướng đến kinh doanh hay công nghệ*—Một bài kiểm tra hướng đến doanh nghiệp được mô tả bằng thuật ngữ của chuyên gia trong lĩnh vực, trong khi một bài kiểm tra hướng đến công nghệ được mô tả bằng thuật ngữ của nhà phát triển và việc triển khai.
* *Cho dù mục tiêu của bài kiểm tra là hỗ trợ lập trình hay phê bình ứng dụng*—Các nhà phát triển sử dụng các bài kiểm tra hỗ trợ lập trình như một phần công việc hàng ngày của họ. Các bài kiểm tra phê bình ứng dụng nhằm mục đích xác định các lĩnh vực cần cải thiện.

Đối mặt với doanh nghiệp

Hỗ trợ lập trình

Dự án phê bình

|  |  |
| --- | --- |
| Q2 TỰ ĐỘNG  Kiểm tra chức năng/chấp nhận | Q3 HƯỚNG DẪN SỬ DỤNG  Kiểm thử thăm dò, kiểm thử khả năng sử dụng |
| Q1 TỰ ĐỘNG | Q4 THỦ CÔNG/TỰ ĐỘNG |
| Đơn vị, tích hợp, thành phần | Kiểm tra chấp nhận phi chức năng: hiệu suất  và nhiều hơn nữa |

Công nghệ đối mặt

**Hình 9.4 Góc phần tư thử nghiệm phân loại các thử nghiệm theo hai chiều. Chiều thứ nhất là liệu một thử nghiệm có hướng đến doanh nghiệp hay hướng đến công nghệ hay không. Chiều thứ hai làcho dù mục đích của bài kiểm tra là hỗ trợ lập trình hay phê bình ứng dụng.**

Phần kiểm tra xác định bốn loại bài kiểm tra khác nhau:

* + - * *Câu hỏi 1*—Hỗ trợ lập trình/công nghệ đối mặt: kiểm tra đơn vị và tích hợp
      * *Quý 2*—Hỗ trợ lập trình/đối mặt với doanh nghiệp: kiểm tra thành phần và đầu cuối
      * *Quý 3*—Đánh giá ứng dụng/kinh doanh đối mặt: khả năng sử dụng và thử nghiệm khám phá
      * *Quý 4*—Đánh giá ứng dụng/công nghệ đang đối mặt: các bài kiểm tra chấp nhận không có chức năng nhưnhư các bài kiểm tra hiệu suất

Bảng kiểm tra không phải là cách duy nhất để tổ chức các bài kiểm tra. Ngoài ra còn có kim tự tháp kiểm tra, cung cấp hướng dẫn về số lượng bài kiểm tra của từng loại cần viết.

**BạnHÁT KIM TỰ THÍCH KIỂM TRA NHƯ MỘT HƯỚNG DẪN TẬP TRUNG NỖ LỰC KIỂM TRA CỦA BẠN**

Chúng ta phải viết các loại kiểm thử khác nhau để có thể tự tin rằng ứng dụng của chúng ta hoạt động. Tuy nhiên, thách thức là thời gian thực hiện và độ phức tạp của một bài kiểm thử tăng theo phạm vi của nó. Ngoài ra, phạm vi của một bài kiểm thử càng lớn và càng có nhiều phần chuyển động thì độ tin cậy càng kém. Các bài kiểm thử không đáng tin cậy cũng tệ như không có bài kiểm thử nào, vì nếu bạn không thể tin tưởng một bài kiểm thử, bạn có thể bỏ qua các lỗi.

Ở một đầu của quang phổ là các bài kiểm tra đơn vị cho các lớp riêng lẻ. Chúng nhanh chóng thực hiện, dễ viết và đáng tin cậy. Ở đầu kia của quang phổ là các bài kiểm tra đầu cuối cho toàn bộ ứng dụng. Những bài kiểm tra này có xu hướng chậm, khó viết và thường không đáng tin cậy vì tính phức tạp của chúng. Vì chúng tôi không có ngân sách không giới hạn cho việc phát triển và thử nghiệm, chúng tôi muốn tập trung vào việc viết các bài kiểm tra có phạm vi nhỏ mà không ảnh hưởng đến hiệu quả của bộ kiểm tra.

Kim tự tháp thử nghiệm, được thể hiện trong hình 9.5, là một hướng dẫn tốt ([https://martinfowler.com/](https://martinfowler.com/bliki/TestPyramid.html) [bliki/TestPyramid.html](https://martinfowler.com/bliki/TestPyramid.html)). Ở đáy kim tự tháp là các bài kiểm tra đơn vị nhanh, đơn giản và đáng tin cậy. Ở đỉnh kim tự tháp là các bài kiểm tra đầu cuối chậm, phức tạp và dễ vỡ. Giống như kim tự tháp thực phẩm của USDA, mặc dù hữu ích hơn và ít gây tranh cãi hơn ([https://vi](https://en.wikipedia.org/wiki/History_of_USDA_nutrition_guides)

[.wikipedia.org/wiki/History\_of\_USDA\_nutrition\_guides](https://en.wikipedia.org/wiki/History_of_USDA_nutrition_guides)),Kim tự tháp thử nghiệm mô tả tỷ lệ tương đối của từng loại thử nghiệm.

Ý tưởng chính của kim tự tháp thử nghiệm làkhi chúng ta di chuyển lên kim tự tháp, chúng ta nên viết ngày càng ít bài kiểm tra. Chúng ta nên viết nhiều bài kiểm tra đơn vị và rất ít bài kiểm tra đầu cuối.

Chậm, giòn, tốn kém

Kiểm tra chấp nhận cho một ứng dụng

Sự chấp nhậnkiểm tra cho một dịch vụ

Xác minh rằng một dịch vụ giao tiếp với các phụ thuộc của nó

Từ đầu đến cuối

Thành phần

Tích hợp

Kiểm tralogic kinh doanh

Đơn vị

Nhanh chóng, đáng tin cậy, giá rẻ

**Hình 9.5 Kim tự tháp kiểm tra mô tả tỷ lệ tương đối của từng loại bài kiểm tra mà bạn cần viết. Khi bạn di chuyển lên kim tự tháp, bạn nên viết ngày càng ít bài kiểm tra hơn.**

Như bạn sẽ thấy trong chương này, tôi mô tả một chiến lược nhấn mạnh vào việc kiểm tra các phần của dịch vụ. Nó thậm chí còn giảm thiểu số lượng các bài kiểm tra thành phần, kiểm tra toàn bộ dịch vụ.Rõ ràng là cách kiểm tra từng dịch vụ siêu nhỏ như Consumer Service, không phụ thuộc vào bất kỳ dịch vụ nào khác. Nhưng còn các dịch vụ như Order Service, phụ thuộc vào nhiều dịch vụ khác thì sao? Và làm sao chúng ta có thể tự tin rằng ứng dụng hoạt động như một tổng thể? Đây là thách thức chính của việc kiểm tra các ứng dụng có kiến ​​trúc microservice. Độ phức tạp của việc kiểm tra đã chuyển từ các dịch vụ riêng lẻ sang các tương tác giữa chúng. Hãy cùng xem cách giải quyết vấn đề này

vấn đề.

###### Thách thức của việc thử nghiệm các dịch vụ vi mô

Giao tiếp giữa các tiến trình đóng vai trò quan trọng hơn nhiều trong ứng dụng dựa trên dịch vụ vi mô so với ứng dụng đơn khối. Ứng dụng đơn khối có thể giao tiếp với một số dịch vụ và máy khách bên ngoài. Ví dụ, phiên bản đơn khối của ứng dụng FTGO sử dụng một số dịch vụ web của bên thứ ba, chẳng hạn như Stripe để thanh toán, Twilio để nhắn tin và Amazon SES để gửi email, có API ổn định. Bất kỳ tương tác nào giữa các mô-đun của ứng dụng đều thông qua API dựa trên ngôn ngữ lập trình. Giao tiếp giữa các tiến trình nằm rất xa ứng dụng.

Ngược lại, giao tiếp giữa các tiến trình là trọng tâm của kiến ​​trúc microservice. Ứng dụng dựa trên microservice là hệ thống phân tán. Các nhóm liên tục phát triển dịch vụ và cải tiến API của họ. Điều cần thiết là các nhà phát triển dịch vụ phải viết các bài kiểm tra để xác minh rằng dịch vụ của họ tương tác với các phụ thuộc và máy khách của nó.

Như đã mô tả trong chương 3, các dịch vụ giao tiếp với nhau bằng nhiều kiểu tương tác và cơ chế IPC. Một số dịch vụ sử dụng tương tác kiểu yêu cầu/phản hồi được triển khai bằng giao thức đồng bộ, chẳng hạn như REST hoặc gRPC.

Các dịch vụ khác tương tác thông qua yêu cầu/trả lời không đồng bộ hoặc xuất bản/đăng ký bằng cách sử dụng tin nhắn không đồng bộ. Ví dụ, hình 9.6 cho thấy cách một số dịch vụ trong ứng dụng FTGO giao tiếp. Mỗi mũi tên chỉ từ dịch vụ người dùng đến dịch vụ nhà sản xuất.

Chìa khóa

Triệu hồidịch vụ sử dụng HTTP

Giao diện lập trình ứng dụng (API)

cổng vào



Khách hàng REST

Dịch vụ REST

Người đăng ký E

Sự kiện miền

nhà xuất bản

(Tin nhắn lệnh)

người yêu cầu Người trả lời

C



Vận chuyển

Dịch vụ

E

E

C

E

Đăng ký

đặt hàng\* sự kiện

E

C

Dịch vụ đặt hàng

Dịch vụ nhà hàng

Dịch vụ nhà bếp

Dịch vụ Lịch sử đơn hàng

Dịch vụ khách hàng

C

Order Service saga gửi lệnh đến nhiều dịch vụ khác nhau.

Dịch vụ kế toán

**Hình 9.6 Một số giao tiếp giữa các dịch vụ trong ứng dụng FTGO. Mỗi mũi tên chỉtừ dịch vụ tiêu dùng sang dịch vụ sản xuất.**

Mũi tên chỉ hướng phụ thuộc, từ người dùng API đến nhà cung cấp API. Các giả định mà người dùng đưa ra về API phụ thuộc vào bản chất của tương tác:

* *Khách hàng REST**dịch vụ*—Cổng API định tuyến các yêu cầu đến các dịch vụ và triển khai thành phần API.
* *Người tiêu dùng sự kiện miền**nhà xuất bản*—Dịch vụ Lịch sử đơn hàngtiêu thụ các sự kiện pub-được viết bởiDịch vụ đặt hàng.
* *Tin nhắn lệnh người yêu cầu*  *người trả lời*—Dịch vụ đặt hànggửi yêu cầu tin nhắnđến nhiều dịch vụ khác nhau và sử dụng các phản hồi.

Mỗi tương tác giữa một cặp dịch vụ đại diện cho một thỏa thuận hoặc hợp đồng giữa hai dịch vụ. Ví dụ, Order History Service và Order Service phải đồng ý về cấu trúc thông báo sự kiện và kênh mà chúng được xuất bản. Tương tự, API gateway và các dịch vụ phải đồng ý về các điểm cuối REST API. Và Order Service và mỗi dịch vụ mà nó gọi bằng yêu cầu/phản hồi không đồng bộ phải đồng ý về kênh lệnh và định dạng của lệnh và thông báo trả lời.

Là nhà phát triển dịch vụ, bạn cần phải tự tin rằng các dịch vụ bạn sử dụng có API ổn định. Tương tự như vậy, bạn không muốn vô tình thực hiện các thay đổi đột ngộtvới API của dịch vụ của bạn. Ví dụ, nếu bạn đang làm việc trên Order Service, bạn muốn chắc chắn rằng các nhà phát triển các phụ thuộc của dịch vụ của bạn, chẳng hạn như Consumer Service và Kitchen Service, không thay đổi API của họ theo cách không tương thích với dịch vụ của bạn. Tương tự như vậy, bạn phải đảm bảo rằng bạn không thay đổi API của Order Services theo cách phá vỡ API Gateway hoặc Order History Service.

Một cách để xác minh rằng hai dịch vụ có thể tương tác là chạy cả hai dịch vụ, gọi một API kích hoạt giao tiếp và xác minh rằng nó có kết quả mong đợi. Điều này chắc chắn sẽ phát hiện ra các vấn đề tích hợp, nhưng về cơ bản, đây là một đầu cuối. Bài kiểm tra có thể sẽ cần chạy nhiều phụ thuộc chuyển tiếp khác của các dịch vụ đó. Một bài kiểm tra cũng có thể cần gọi chức năng phức tạp, cấp cao như logic kinh doanh, ngay cả khi mục tiêu của nó là kiểm tra IPC cấp độ tương đối thấp. Tốt nhất là tránh viết các bài kiểm tra đầu cuối như thế này. Bằng cách nào đó, chúng ta cần viết các bài kiểm tra nhanh hơn, đơn giản hơn và đáng tin cậy hơn, lý tưởng nhất là kiểm tradịch vụ riêng lẻ. Giải pháp là sử dụng cái được gọi là thử nghiệm hợp đồng do người tiêu dùng thúc đẩy.

**CNGƯỜI TIÊU DÙNG-KIỂM TRA HỢP ĐỒNG LÁI XE**

Hãy tưởng tượng rằngbạn là thành viên của nhóm phát triểnCổng API, được mô tảtrong chương 8.Cổng API'SDịch vụ đặt hàngProxygọi nhiều điểm cuối REST khác nhau, bao gồm-trongNHẬN /đơn hàng/{orderId}điểm cuối. Nó là điều cần thiếtrằng chúng tôi viết các bài kiểm tra để xác minh rằngCổng APIVàDịch vụ đặt hàngđồng ý về một API. Theo thuật ngữ thử nghiệm hợp đồng của người tiêu dùng, hai dịch vụ tham gia vào mối quan hệ giữa người tiêu dùng và nhà cung cấp.Cổng APIlà người tiêu dùng vàDịch vụ đặt hànglà một nhà cung cấp. Một thử nghiệm hợp đồng tiêu dùng là một sự tích hợpkiểm tra cho một nhà cung cấp, chẳng hạn nhưDịch vụ đặt hàng, xác minh rằng API của nó phù hợp với kỳ vọng của người tiêu dùng, chẳng hạn nhưCổng API.

Kiểm tra hợp đồng của người tiêu dùng tập trung vào việc xác minh rằng "hình dạng" của API của nhà cung cấp đáp ứng được kỳ vọng của người tiêu dùng. Đối với điểm cuối REST, kiểm tra hợp đồng xác minh rằng nhà cung cấp triển khai điểm cuối

* + Có phương thức và đường dẫn HTTP mong đợi
  + Chấp nhận các tiêu đề mong đợi, nếu có
  + Chấp nhận một cơ quan yêu cầu, nếu có
  + Trả về phản hồi với mã trạng thái, tiêu đề và nội dung mong đợi

Điều quan trọng cần nhớ là các bài kiểm tra hợp đồng không kiểm tra kỹ lưỡng logic kinh doanh của nhà cung cấp. Đó là công việc của các bài kiểm tra đơn vị. Sau này, bạn sẽ thấy rằng các bài kiểm tra hợp đồng của người dùng cho REST API thực chất là các bài kiểm tra bộ điều khiển giả.

Nhóm phát triển người tiêu dùng viết một bộ kiểm thử hợp đồng và thêm nó (ví dụ, thông qua yêu cầu kéo) vào bộ kiểm thử của nhà cung cấp. Các nhà phát triển của các dịch vụ kháccác tệ nạn gọi Order Service cũng đóng góp một bộ kiểm thử, như thể hiện trong hình 9.7. Mỗi bộ kiểm thử sẽ kiểm thử các khía cạnh của API Order Service có liên quan đến từng người dùng. Ví dụ, bộ kiểm thử cho Order History Service xác minh rằng Order Service công bố các sự kiện dự kiến.



Nhóm cổng API

Đội ngũ dịch vụ Lịch sử đơn hàng

... Đội ngũ dịch vụ

Viết

Viết

Viết

Kiểm traKiểm tra

Kiểm tra

Đường ống triển khai dịch vụ đặt hàng

... Dịch vụ - Bộ kiểm tra hợp đồng dịch vụ đặt hàng

Dịch vụ Lịch sử Đơn hàng - Bộ kiểm tra hợp đồng Dịch vụ Đơn hàng

Cổng API - Bộ kiểm tra hợp đồng dịch vụ đơn hàng

Dịch vụ đặt hàng

**Hình 9.7 Mỗi nhóm phát triển một dịch vụ tiêu thụDịch vụ đặt hàngAPI của 's đóng góp một bộ kiểm thử hợp đồng. Bộ kiểm thử xác minh rằng API phù hợp với kỳ vọng của người tiêu dùng. Bộ kiểm thử này, cùng với các bộ kiểm thử do các nhóm khác đóng góp, được điều hành bởiDịch vụ đặt hàng'Sđường ống triển khai.**

Các bộ kiểm thử này được thực hiện bởi đường ống triển khai cho Dịch vụ Đặt hàng. Nếu một thử nghiệm hợp đồng người tiêu dùng không thành công, lỗi đó sẽ cho nhà sản xuất biếtnhóm rằng họ đã thực hiện một thay đổi đột phá đối với API. Họ phải sửa API hoặc nói chuyện với nhóm người tiêu dùng.

**Mẫu: Kiểm tra hợp đồng do người tiêu dùng thúc đẩy**

Xác minh rằng một dịch vụ đáp ứng được kỳ vọng của khách hàng Xem[http://microser-](http://microservices.io/patterns/testing/service-integration-contract-test.html) [Vices.io/patterns/testing/service-integration-contract-test.html](http://microservices.io/patterns/testing/service-integration-contract-test.html).

Các thử nghiệm hợp đồng do người tiêu dùng điều khiển thường sử dụng thử nghiệm bằng ví dụ. Tương tác giữa người tiêu dùng và nhà cung cấp được xác định bằng một tập hợp các ví dụ, được gọi là hợp đồng. Mỗi hợp đồng bao gồm các thông điệp ví dụ được trao đổi trong một tương tác.

Ví dụ, hợp đồng cho REST API bao gồm một ví dụ về yêu cầu và phản hồi HTTP. Trên bề mặt, có vẻ tốt hơn khi định nghĩa tương tác bằng cách sử dụng các lược đồ được viết bằng, ví dụ, lược đồ OpenAPI hoặc JSON. Nhưng hóa ra lược đồ không hữu ích khi viết các bài kiểm tra. Một bài kiểm tra có thể xác thực phản hồi bằng cách sử dụng lược đồ nhưng vẫn cần phải gọi nhà cung cấp bằng một yêu cầu ví dụ.

Hơn nữa, các bài kiểm tra người dùng cũng cần các phản hồi mẫu. Đó là bởi vì mặc dù trọng tâm của thử nghiệm hợp đồng do người dùng điều khiển là kiểm tra nhà cung cấp, các hợp đồng cũng được sử dụng để xác minh rằng người dùng tuân thủ hợp đồng. Ví dụ, một bài kiểm tra hợp đồng phía người dùng cho máy khách REST sử dụng hợp đồng để cấu hình dịch vụ HTTP stub xác minh rằng yêu cầu HTTP khớp với yêu cầu của hợp đồng và gửi lại phản hồi HTTP của hợp đồng. Kiểm tra cả hai phía tương tác đảm bảo rằng người dùng và nhà cung cấp đồng ý về API. Sau này chúng ta sẽ xem các ví dụ về cách viết loại thử nghiệm này, nhưng trước tiên hãy xem cách viết các bài kiểm tra hợp đồng người dùng bằng Spring Cloud Contract.

**Mẫu: Kiểm tra hợp đồng phía người tiêu dùng**

Xác minh rằng máy khách của dịch vụ có thể giao tiếp với dịch vụ. Xem[https://](https://microservices.io/patterns/testing/consumer-side-contract-test.html) [microservices.io/patterns/testing/consumer-side-contract-test.html](https://microservices.io/patterns/testing/consumer-side-contract-test.html).

**TDỊCH VỤ ESTING SỬ DỤNGSInCỒN ÀOCTRÌNH TỰA**

Hai khuôn khổ thử nghiệm hợp đồng phổ biến là Spring Cloud Contract ([https://đám mây](https://cloud.spring.io/spring-cloud-contract/)

[.spring.io/spring-cloud-contract/](https://cloud.spring.io/spring-cloud-contract/)), là một khuôn khổ thử nghiệm hợp đồng của người tiêu dùng dành cho các ứng dụng Spring và họ khuôn khổ Pact ([https://github.com/pact-](https://github.com/pact-foundation) [sự thành lập](https://github.com/pact-foundation)), hỗ trợ nhiều ngôn ngữ khác nhau. Ứng dụng FTGO là ứng dụng dựa trên nền tảng Spring, vì vậy trong chương này tôi sẽ mô tả cách sử dụng Spring Cloud Contract. Nó cung cấp ngôn ngữ Groovy dành riêng cho miền (DSL) để viết hợp đồng. Mỗi hợp đồng là một ví dụ cụ thể về tương tác giữa người dùng và nhà cung cấp, chẳng hạn như yêu cầu và phản hồi HTTP. Mã Spring Cloud Contract tạo các bài kiểm tra hợp đồng cho nhà cung cấp. Nó cũng cấu hình các bản mô phỏng, chẳng hạn như máy chủ HTTP mô phỏng, cho các bài kiểm tra tích hợp của người dùng.

Nói ví dụ, bạn đang làm việc trênCổng APIvà muốn viết một người tiêu dùngkiểm tra hợp đồng choDịch vụ đặt hàngHình 9.8 cho thấy quá trình này, đòi hỏi bạn phải cộng tác vớiDịch vụ đặt hàngđội. Bạn viết hợp đồng xác định cáchCổng API tương tác vớiDịch vụ đặt hàng. CácDịch vụ đặt hàngnhóm sử dụng những hợp đồng này để thử nghiệmDịch vụ đặt hàng, và bạnsử dụng chúng để thử nghiệmCổng API. Trình tự các bước như sau:

**1**Bạn viết một hoặc nhiều hợp đồng, chẳng hạn như hợp đồng được hiển thị trong danh sách 9.1. Mỗi hợp đồngđường dẫnbao gồm một yêu cầu HTTP mà API Gateway có thể gửi đến Order Service và một phản hồi HTTP dự kiến. Bạn cung cấp hợp đồng, có thể thông qua yêu cầu kéo Git, cho nhóm Order Service.

**2**CácDịch vụ đặt hàngkiểm tra nhómDịch vụ đặt hàngsử dụng các thử nghiệm hợp đồng của người tiêu dùng mà mã Spring Cloud Contract tạo ra từ các hợp đồng.



Nhóm cổng API



Viết

Mã được tạo ra từ

Phát triểnKiểm tra



Kho lưu trữ Maven

Kiểm tra

Phát triển

ĐọcXuất bản

Kiểm tra tích hợp cổng API

Dịch vụ đặt hàng

Cổng API

Hợp đồng đã công bố



Đặt hàngKiểm tra hợp đồng người tiêu dùng dịch vụ

Contract.make { yêu cầu {..}

phản ứng {...}

}

}

Đội ngũ dịch vụ đặt hàng

**Hình 9.8Cổng APInhóm viết hợp đồng.Dịch vụ đặt hàngnhóm sử dụng những hợp đồng đó để thử nghiệmDịch vụ đặt hàngvà xuất bản chúng vào một kho lưu trữ.Cổng APInhóm sử dụng các hợp đồng đã công bố để thử nghiệmCổng API.**



**3**CácDịch vụ đặt hàngnhóm công bố các hợp đồng đã được thử nghiệmDịch vụ đặt hàngvào kho lưu trữ Maven.

**4** Bạn sử dụng các hợp đồng đã công bố để viết các bài kiểm tra choCổng API.

Bởi vì bạn kiểm tra API Gateway bằng cách sử dụng các hợp đồng đã công bố, bạn có thể tự tin rằngnó hoạt động với việc triển khaiDịch vụ đặt hàng.

Hợp đồng là phần chính của chiến lược thử nghiệm này. Danh sách sau đây cho thấy một ví dụ về Spring Cloud Contract. Nó bao gồm một yêu cầu HTTP và một phản hồi HTTP.

**Danh sách 9.1 Một hợp đồng mô tả cách thứcCổng APIgọiĐặt hàngDịch vụ**

org.springframework.cloud.contract.spec.Contract.make{ lời yêu cầu {

phương pháp 'GET'

url '/orders/1223232'

**Phương pháp và đường dẫn của yêu cầu HTTP**

}

phản ứng {

trạng thái 200 tiêu đề {

**Mã trạng thái, tiêu đề và nội dung của phản hồi HTTP**

header('Kiểu-Nội-dung':'ứng dụng/json;charset=UTF-8')

}

thân hình("{ ... }")

}

}

Phần tử yêu cầu là một yêu cầu HTTP cho điểm cuối REST GET /orders/

{orderId}. Phần tử phản hồi là phản hồi HTTP mô tả Order mà API Gateway mong đợi. Các hợp đồng Groovy là một phần trong cơ sở mã của nhà cung cấp. Mỗi nhóm người tiêu dùng viết các hợp đồng mô tả cách dịch vụ của họ tương tác với nhà cung cấp và cung cấp chúng, có thể thông qua yêu cầu kéo Git, cho nhóm nhà cung cấp. Nhóm nhà cung cấp chịu trách nhiệm đóng gói các hợp đồng dưới dạng JAR và xuất bản chúng lên kho lưu trữ Maven. Các bài kiểm tra phía người tiêu dùng tải xuống JAR từ kho lưu trữ.

Yêu cầu và phản hồi của mỗi hợp đồng đóng vai trò kép là dữ liệu thử nghiệm và chỉ định hành vi mong đợi. Trong thử nghiệm phía người dùng, hợp đồng được sử dụng để cấu hình một stub, tương tự như đối tượng giả Mockito và mô phỏng hành vi của Order Service. Nó cho phép API Gateway được thử nghiệm mà không cần chạy Order Service. Trong thử nghiệm phía nhà cung cấp, lớp thử nghiệm được tạo ra sẽ gọi nhà cung cấp với yêu cầu của hợp đồng và xác minh rằng nó trả về phản hồi khớp với phản hồi của hợp đồng. Chương tiếp theo thảo luận về chi tiết về cách sử dụng Spring Cloud Contract, nhưng bây giờ chúng ta sẽ xem xét cách sử dụng thử nghiệm hợp đồng người dùng cho API nhắn tin.

**CKIỂM TRA HỢP ĐỒNG NGƯỜI TIÊU DÙNG CHO TIN NHẮNGiao diện lập trình ứng dụng (API)S**

Máy khách REST không phải là loại người tiêu dùng duy nhất có kỳ vọng vào API của nhà cung cấp. Các dịch vụ đăng ký sự kiện miền và sử dụng giao tiếp dựa trên yêu cầu/phản hồi không đồng bộ cũng là người tiêu dùng. Chúng sử dụng API nhắn tin của một số dịch vụ khác và đưa ra giả định về bản chất của API đó. Chúng ta cũng phải viết các bài kiểm tra hợp đồng người tiêu dùng cho các dịch vụ này.

Spring Cloud Contract cũng cung cấp hỗ trợ cho việc kiểm thử các tương tác dựa trên tin nhắn. Cấu trúc của hợp đồng và cách sử dụng hợp đồng trong các bài kiểm tra phụ thuộc vào loại tương tác. Hợp đồng cho việc xuất bản sự kiện miền bao gồm một sự kiện miền mẫu. Bài kiểm tra nhà cung cấp khiến nhà cung cấp phát ra một sự kiện và xác minh rằng sự kiện này khớp với sự kiện của hợp đồng. Bài kiểm tra người dùng xác minh rằng người dùng có thể xử lý sự kiện đó. Trong chương tiếp theo, tôi sẽ mô tả một bài kiểm tra mẫu.

Hợp đồng cho tương tác yêu cầu/phản hồi không đồng bộ tương tự như hợp đồng HTTP. Nó bao gồm một thông báo yêu cầu và một thông báo phản hồi. Một bài kiểm tra nhà cung cấp sẽ gọi API bằng thông báo yêu cầu của hợp đồng và xác minh rằng phản hồi khớp với phản hồi của hợp đồng. Một bài kiểm tra người dùng sử dụng hợp đồng để cấu hình một stub subscriber, lắng nghe thông báo yêu cầu của hợp đồng và trả lời bằng phản hồi đã chỉ định. Chương tiếp theo sẽ thảo luận về một bài kiểm tra ví dụ. Nhưng trước tiên chúng ta sẽ xem xét đường ống triển khai, chạy các bài kiểm tra này và các bài kiểm tra khác.

###### Đường ống triển khai

Mỗi dịch vụ đều có một đường ống triển khai. Cuốn sách của Jez Humble, Giao hàng liên tục (Addison-Wesley, 2010) mô tả đường ống triển khai là quy trình tự động đưa mã từ máy tính để bàn của nhà phát triển vào sản xuất. Như hình 9.9 cho thấy, nó bao gồm

của một loạt các giai đoạn thực hiện bộ kiểm thử, tiếp theo là một giai đoạn phát hành hoặc triển khai dịch vụ. Lý tưởng nhất là nó hoàn toàn tự động, nhưng nó có thể chứa các bước thủ công. Một đường ống triển khai thường được triển khai bằng cách sử dụng máy chủ Tích hợp liên tục (CI), chẳng hạn như Jenkins.

Nhanhnhận xét Phản hồi chậm

Không sản xuấtsẵn sàng

Sản xuấtsẵn sàng

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Đường ống triển khai | | | | | | | | | |
| Cam kết trướckiểm tra |  |  | | Làm |  | Tích hợp |  | Thành phần |  | Giai đoạn triển khai |  |
|  |  |  | kiểm tra  sân khấu |  | kiểm tra  sân khấu |  | kiểm tra  sân khấu |  |  |
|  | |  | | | | | | | | |

Môi trường sản xuất

**Hình 9.9 Một ví dụ về đường ống triển khai choDịch vụ đặt hàng. Nó bao gồm một loạt các giai đoạn. Các bài kiểm tra trước khi cam kết được chạy bởi nhà phát triển trước khi cam kết mã của họ. Các giai đoạn còn lạiđược thực hiện bởi một công cụ tự động, chẳng hạn như máy chủ Jenkins CI.**

Khi mã chạy qua đường ống, các bộ kiểm thử sẽ kiểm thử ngày càng kỹ lưỡng hơn trong các môi trường giống môi trường sản xuất hơn. Đồng thời, thời gian thực hiện của mỗi bộ kiểm thử thường tăng lên. Ý tưởng là cung cấp phản hồi về lỗi kiểm thử càng nhanh càng tốt.

Ví dụ về quy trình triển khai được hiển thị trong hình 9.9 bao gồm các giai đoạn sau:

* *Giai đoạn kiểm tra trước khi cam kết*—Chạy các bài kiểm tra đơn vị. Điều này được thực hiện bởi nhà phát triển trước khi cam kết các thay đổi của họ.
* *Giai đoạn kiểm tra cam kết*—Biên dịch dịch vụ, chạy thử nghiệm đơn vị và thực hiện phân tích mã tĩnh.
* *Giai đoạn kiểm tra tích hợp*—Chạy thử nghiệm tích hợp.
* *Giai đoạn kiểm tra thành phần*—Chạy thử nghiệm thành phần cho dịch vụ.
* *Giai đoạn triển khai*—Triển khaiđưa dịch vụ vào sản xuất.

Máy chủ CI chạy giai đoạn cam kết khi nhà phát triển cam kết thay đổi. Nó thực hiện cực kỳ nhanh, do đó cung cấp phản hồi nhanh về cam kết. Các giai đoạn sau mất nhiều thời gian hơn để chạy, cung cấp phản hồi ít tức thời hơn. Nếu tất cả các bài kiểm tra đều vượt qua, giai đoạn cuối cùng là khi đường ống này triển khai nó vào sản xuất.

Trong ví dụ này, đường ống triển khai được tự động hóa hoàn toàn từ cam kết đến triển khai. Tuy nhiên, có những tình huống yêu cầu các bước thủ công. Ví dụ, bạn có thể cần một giai đoạn thử nghiệm thủ công, chẳng hạn như môi trường dàn dựng. Trong trường hợp như vậy, mã sẽ tiến triển đến giai đoạn tiếp theo khi người kiểm tra nhấp vào nút để chỉ ra rằng nó đã thành công. Ngoài ra, một đường ống triển khai cho một cơ sở tại chỗ

sản phẩm sẽ phát hành phiên bản mới của dịch vụ. Sau đó, các dịch vụ đã phát hành sẽ được đóng gói thành bản phát hành sản phẩm và chuyển đến khách hàng.

Bây giờ chúng ta đã xem xét tổ chức của đường ống triển khai và khi nó thực hiện các loại thử nghiệm khác nhau, hãyđi đến cuối kim tự tháp thử nghiệm và xem cách viết các bài kiểm tra đơn vị cho một dịch vụ.

#### Viết các bài kiểm tra đơn vị cho một dịch vụ

Hãy tưởng tượng rằng bạn muốn viết một bài kiểm tra để xác minh rằng Đơn hàng của ứng dụng FTGOService tính toán chính xác tổng phụ của một Order. Bạn có thể viết các bài kiểm tra chạy Order Service, gọi REST API của nó để tạo một Order và kiểm tra xem phản hồi HTTP có chứa các giá trị mong đợi hay không. Nhược điểm của cách tiếp cận này là không chỉ bài kiểm tra phức tạp mà còn chậm. Nếu các bài kiểm tra này là các bài kiểm tra thời gian biên dịch cho lớp Order, bạn sẽ mất rất nhiều thời gian chờ đợi để nó hoàn tất. Một cách tiếp cận hiệu quả hơn nhiều là viết các bài kiểm tra đơn vị cho lớp Order.

Như hình 9.10 cho thấy, các bài kiểm tra đơn vị là cấp độ thấp nhất của kim tự tháp kiểm tra. Chúng là các bài kiểm tra hướng đến công nghệ hỗ trợ phát triển. Một bài kiểm tra đơn vị xác minh rằng một đơn vị, là một phần rất nhỏ của một dịch vụ, hoạt động chính xác. Một đơn vị thường là một lớp, vì vậy mục tiêu của bài kiểm tra đơn vị là xác minh rằng nó hoạt động như mong đợi.

Kiểm tra

Từ đầu đến cuối

Thành phần

Tích hợp

Đơn vị

Lớp học

Sự phụ thuộc của stub/mock 2

Phụ thuộc stub/mock 1

Kiểm tra đơn vị đơn lẻ

Phụ thuộc 1

Sự phụ thuộc của stub/mock

...

Kiểm tra

**Hình 9.10Các bài kiểm tra đơn vị là cơ sở của kim tự tháp. Chúng chạy nhanh,dễ viết và đáng tin cậy.Một bài kiểm tra đơn vị riêng lẻ kiểm tra một lớp riêng biệt, sử dụngmocks hoặc stubs cho các phụ thuộc của nó. Một bài kiểm tra đơn vị xã hội sẽ kiểm tra một lớp và các phụ thuộc của nó.**

Lớp học

Sự phụ thuộc

...

Phụ thuộc 2

Kiểm tra đơn vị xã hội

Có hai loại kiểm thử đơn vị (<https://martinfowler.com/bliki/UnitTest.html>):

* *Kiểm tra đơn vị đơn lẻ*—Kiểm tra một lớp riêng biệt bằng cách sử dụng các đối tượng giả cho các phụ thuộc của lớp
* *Bài kiểm tra đơn vị xã hội*—Kiểm tra một lớp và các phụ thuộc của nó

Trách nhiệm của lớp và vai trò của nó trong kiến ​​trúc xác định loại kiểm tra nào sẽ sử dụng. Hình 9.11 cho thấy kiến ​​trúc lục giác của một dịch vụ điển hình và loại kiểm tra đơn vị mà bạn thường sử dụng cho từng loại lớp. Các lớp điều khiển và dịch vụ thường được kiểm tra bằng các kiểm tra đơn vị đơn lẻ. Các đối tượng miền, chẳng hạn như các thực thể và các đối tượng giá trị, thường được kiểm tra bằng các kiểm tra đơn vị xã hội.



POST/cái gì đó GET/cái gì đó/id

«Kênh tin nhắn»

Logic miền

«Kênh tin nhắn»

Kho lưu trữ

Bộ điều hợp cơ sở dữ liệu

Đi ra ngoàibộ chuyển đổi tin nhắn

Bộ điều khiển

Cơ sở dữ liệu

Truyện dài

Đối tượng giá trị

Thực thể

Bài kiểm tra đơn vị xã hội

Kiểm tra đơn vị đơn lẻ

Dịch vụ

Bộ điều hợp tin nhắn đến

Kiểm tra đơn vị đơn lẻ

**Hình 9.11 Trách nhiệm của một lớp sẽ quyết định xem nên sử dụng bài kiểm tra đơn lẻ hay bài kiểm tra giao lưu.**

Chiến lược kiểm tra điển hình cho mỗi lớp như sau:

* + Các thực thể, chẳng hạn nhưĐặt hàng, như được mô tả trong chương 5, là các đối tượng có bản sắc cố định, được thử nghiệm bằng các bài kiểm tra đơn vị xã hội.
  + Các đối tượng giá trị, chẳng hạn nhưTiền bạc, như được mô tả trong chương 5, là các đối tượng là tập hợp các giá trị, được kiểm tra bằng các bài kiểm tra đơn vị xã hội.
  + Những câu chuyện như thế nàyTạo đơn hàngSaga, như được mô tả trong chương 4, duy trì tính nhất quán của dữ liệu trên các dịch vụ, được thử nghiệm bằng các bài kiểm tra đơn vị xã hội.
  + Dịch vụ tên miền, chẳng hạn nhưDịch vụ đặt hàng, như được mô tả trong chương 5, là các lớp triển khai logic kinh doanh không thuộc về các thực thể hoặc đối tượng giá trị, được kiểm tra bằng các bài kiểm tra đơn vị riêng lẻ.
  + Bộ điều khiển,chẳng hạn nhưBộ điều khiển đơn hàng, xử lý các yêu cầu HTTP, được thử nghiệm bằng các bài kiểm tra đơn vị riêng lẻ.
  + Đến và đicổng nhắn tin được kiểm tra bằng các bài kiểm tra đơn vị riêng lẻ. Chúng ta hãy bắt đầu bằng cách xem cách kiểm tra các thực thể.

###### Phát triển các bài kiểm tra đơn vị cho các thực thể

Danh sách sau đây hiển thị một đoạn trích của lớp OrderTest, lớp này triển khai đơn vịkiểm tra cho thực thể Order. Lớp có phương thức @Before setUp() tạo Order trước khi chạy mỗi thử nghiệm. Phương thức @Test của nó có thể khởi tạo Order thêm nữa, gọi một trongcác phương thức của nó, sau đó đưa ra khẳng định về giá trị trả về và trạng thái của Order.

**Liệt kê 9.2 Một bài kiểm tra đơn vị chạy nhanh, đơn giản choĐặt hàngthực thể**

lớp công khai OrderTest {

riêng tư ResultWithEvents<Order> createResult; riêng tư Order order;

@Trước

public void setUp() ném ra ngoại lệ {

createResult = Order.createOrder(CONSUMER\_ID, AJANTA\_ID, GÀ\_VINDALOO

\_DÒNG\_MỤC);

thứ tự = createResult.result;

}

@Bài kiểm tra

công khai void shouldCalculateTotal() { assertEquals(CHICKEN\_VINDALOO\_PRICE.multiply(CHICKEN\_VINDALOO\_QUANTITY),

đặt hàng.getOrderTotal());

}

...

}

Các@Test nên tính tổng()phương pháp xác minh rằngĐặt hàng.getOrderTotal()

trả về giá trị mong đợi. Các bài kiểm tra đơn vị kiểm tra kỹ lưỡng logic kinh doanh. Chúng là

các bài kiểm tra đơn vị xã hội cho lớp Order và các phụ thuộc của nó. Bạn có thể sử dụng chúng như các bài kiểm tra thời gian biên dịch vì chúng thực thi cực kỳ nhanh. Lớp Order dựa vào đối tượng giá trị Money, vì vậy điều quan trọng là phải kiểm tra lớp đó. Hãy cùng xem cách thực hiện.

###### Viết các bài kiểm tra đơn vị cho các đối tượng giá trị

Các đối tượng giá trị không thể thay đổi, do đó chúng có xu hướng dễ kiểm tra. Bạn không phải lo lắng về các tác dụng phụ. Một bài kiểm tra cho một đối tượng giá trị thường tạo ra một đối tượng giá trị ở trạng thái cụ thể, gọi một trong các phương thức của nó và đưa ra các khẳng định về giá trị trả về. Liệt kê 9.3 cho thấy các bài kiểm tra cho đối tượng giá trị Money, đây là một lớp đơn giản biểu diễn một giá trị tiền. Các bài kiểm tra này xác minh hành vi của các phương thức của lớp Money,bao gồm add(), thêm hai đối tượng Money và multiply(), nhân một đối tượng Money với một số nguyên. Chúng là các bài kiểm tra đơn độc vì lớp Money không phụ thuộc vào bất kỳ lớp ứng dụng nào khác.

**Liệt kê 9.3 Một bài kiểm tra đơn giản, chạy nhanh choTiền bạcđối tượng giá trị**

lớp công khai MoneyTest {

số nguyên cuối cùng riêng tư M1\_AMOUNT = 10; số nguyên cuối cùng riêng tư M2\_AMOUNT = 15;

Tiền riêng m1 = Tiền mới(M1\_AMOUNT); Tiền riêng m2 = Tiền mới(M2\_AMOUNT);

@Bài kiểm tra

công khai void shouldAdd() {

**Xác minh rằng hai đối tượng Tiền có thể được cộng lại với nhau.**

assertEquals(Tiền mới(M1\_AMOUNT + M2\_AMOUNT), m1.add(m2));

}

**Xác minh rằng một khoản tiền**

@Bài kiểm tra

công khai void shouldMultiply() { int multiplier = 12;

**đối tượng có thể được nhân với một số nguyên.**

assertEquals(Tiền mới(M2\_AMOUNT \* hệ số nhân), m2.multiply(hệ số nhân));

}

...

}

Các thực thể và đối tượng giá trị là các khối xây dựng logic kinh doanh của dịch vụ. Nhưng một số logic kinh doanh cũng nằm trong các sagas và dịch vụ của dịch vụ. Hãy cùng xem cách kiểm tra chúng.

###### Phát triển các bài kiểm tra đơn vị cho sagas

Một saga, chẳng hạn như lớp CreateOrderSaga, triển khai logic kinh doanh quan trọng, do đó cần phải được kiểm tra. Đây là một đối tượng liên tục gửi tin nhắn lệnh đến những người tham gia saga và xử lý các phản hồi của họ. Như đã mô tả trong chương 4, CreateOrderSaga trao đổi tin nhắn lệnh/phản hồi với một số dịch vụ, chẳng hạn như Consumer Service và Kitchen Service. Một bài kiểm tra cho lớp này tạo ra một saga và xác minh rằng nó gửi

trình tự dự kiến ​​của các thông điệp gửi đến những người tham gia saga. Một bài kiểm tra bạn cần viết là cho happy path. Bạn cũng phải viết các bài kiểm tra cho các tình huống khác nhau khi saga quay lại vì một người tham gia saga đã gửi lại thông báo lỗi.

Một cách tiếp cận là viết các bài kiểm tra sử dụng cơ sở dữ liệu thực và môi giới tin nhắn cùng với các stub để mô phỏng những người tham gia saga khác nhau. Ví dụ, một stub choDịch vụ tiêu dùng sẽđăng ký kênh lệnh consumerService và gửi lại tin nhắn trả lời mong muốn. Nhưng các bài kiểm tra được viết bằng cách tiếp cận này sẽ khá chậm. Một cách tiếp cận hiệu quả hơn nhiều là viết các bài kiểm tra mô phỏng các lớp tương tác với cơ sở dữ liệu và môi giới tin nhắn. Theo cách đó, chúng ta có thể tập trung vào việc kiểm tra trách nhiệm cốt lõi của saga.

Liệt kê 9.4 cho thấy một bài kiểm tra cho CreateOrderSaga. Đây là một bài kiểm tra đơn vị xã hội kiểm tra lớp saga và các phụ thuộc của nó. Nó được viết bằng cách sử dụng khuôn khổ kiểm tra Eventuate Tram Saga (<https://github.com/eventuate-tram/eventuate-tram-sagas>). Khung này cung cấp một DSL dễ sử dụng, tóm tắt các chi tiết tương tác với saga. Với DSL này, bạn có thể tạo saga và xác minh rằng nó gửi đúng thông điệp lệnh. Bên dưới, khung kiểm tra Saga cấu hình khung Saga với các bản mô phỏng cho cơ sở dữ liệu và cơ sở hạ tầng nhắn tin.

**Liệt kê 9.4 Một bài kiểm tra đơn vị chạy nhanh, đơn giản choTạo đơn hàngSaga**

lớp công khai CreateOrderSagaTest { @Test

công khai void shouldCreateOrder() {

được cho()

**Tạo nên câu chuyện.**

**Xác minh rằng nó gửi**

.saga(CreateOrderSaga mới(kitchenServiceProxy),mới CreateOrderSagaState(ORDER\_ID,

CHI TIẾT ĐẶT HÀNG GÀ VINDALOO)).

**tin nhắn ValidateOrderBy- Consumer gửi đến Dịch vụ Người tiêu dùng.**

trông chờ().

lệnh(mới Xác thực đơn hàng theo người tiêu dùng(ID\_NGƯỜI TIÊU DÙNG, ID\_ĐƠN\_HÀNG, TỔNG\_ĐƠN\_HÀNG\_GÀ)).

tới (ConsumerServiceChannels.consumerServiceChannel).

vàGiven().

trả lời thành công().trông chờ().

**Gửi phản hồi thành công cho tin nhắn đó.**

}

@Bài kiểm tra

lệnh(CreateTicket mới(AJANTA\_ID, ORDER\_ID, null)). đến(KitchenServiceChannels.kitchenServiceChannel);

**Xác minh rằng nó gửi tin nhắn CreateTicket đến Kitchen Service.**

public void shouldRejectOrderDueToConsumerVerificationFailed() { đã cho()

.saga(CreateOrderSaga mới(kitchenServiceProxy),mới CreateOrderSagaState(ORDER\_ID,

CHI TIẾT ĐẶT HÀNG GÀ VINDALOO)).

trông chờ().

lệnh(mới Xác thực đơn hàng theo người tiêu dùng(ID\_NGƯỜI TIÊU DÙNG, ID\_ĐƠN\_HÀNG, TỔNG\_ĐƠN\_HÀNG\_GÀ)).

tới (ConsumerServiceChannels.consumerServiceChannel).vàGiven().

trả lời lỗi().trông chờ().

lệnh(rejectOrderCommand(ORDER\_ID) mới).đến(OrderServiceChannels.orderServiceChannel);

} **Xác minh rằng saga gửi**

**Một Lệnh từ chối đơn hàng**

} **tin nhắn đến Dịch vụ đặt hàng.**

**Gửi phản hồi không thành công cho biết Bộ phận dịch vụ khách hàng đã từ chối đơn hàng.**

Các@Test nênTạo đơn hàng()phương pháp kiểm tra con đường hạnh phúc.@Test nên- RejectOrderDueToConsumerVerificationFailed()phương pháp kiểm tra kịch bản trong đóDịch vụ khách hàngtừ chối đơn hàng. Nó xác minh rằngTạo đơn hàngSagagửi mộtTừ chối-LệnhCommandđể bù đắp cho việc người tiêu dùng bị từ chối.CreateOrder-SagaTestlớp có các phương thứcđể kiểm tra các tình huống thất bại khác.

Bây giờ chúng ta hãy xem cách kiểm tra dịch vụ tên miền.

###### Viết các bài kiểm tra đơn vị cho các dịch vụ miền

Phần lớn logic kinh doanh của dịch vụ được thực hiện bởi các thực thể, đối tượng giá trị,và sagas. Các lớp dịch vụ miền, chẳng hạn như lớp OrderService, triển khai phần còn lại. Lớp này là một lớp dịch vụ miền điển hình. Các phương thức của nó gọi các thực thể và kho lưu trữ và xuất bản các sự kiện miền. Một cách hiệu quả để kiểm tra loại lớp này là sử dụng một bài kiểm tra đơn vị chủ yếu là đơn độc, mô phỏng các phụ thuộc như kho lưu trữ và lớp nhắn tin.

Liệt kê 9.5 hiển thị lớp OrderServiceTest, kiểm tra OrderService. Nó định nghĩa các bài kiểm tra đơn vị riêng lẻ, sử dụng các bản mô phỏng Mockito cho các phụ thuộc của dịch vụ. Mỗi bài kiểm tra triển khai các giai đoạn kiểm tra như sau:

**1** *Cài đặt*—Cấu hình các đối tượng giả cho các phụ thuộc của dịch vụ

**2** *Thực hiện*—Gọi một phương thức dịch vụ

**3***Xác minh*—Xác minh rằng giá trị được trả về bởi phương thức dịch vụ là chính xác và các phụ thuộc đã được gọi đúng cách

**Liệt kê 9.5 Một bài kiểm tra đơn vị chạy nhanh, đơn giản choDịch vụ đặt hànglớp học**

lớp công khai OrderServiceTest { riêng tư OrderService orderService;

OrderRepository riêng tư orderRepository;

riêng tư DomainEventPublisher eventPublisher; riêng tư RestaurantRepository restaurantRepository;

riêng tưSagaManager<CreateOrderSagaState> createOrderSagaManager; riêng tư SagaManager<CancelOrderSagaData> cancelOrderSagaManager; riêng tư SagaManager<ReviseOrderSagaData> revisionOrderSagaManager;

@Trước

công khai void thiết lập() {

orderRepository = mock(OrderRepository. class); eventPublisher = mock(DomainEventPublisher. class);

restaurantRepository = mock(RestaurantRepository. class);

**Tạo bản mô phỏng Mockito cho các phụ thuộc của OrderService.**

createOrderSagaManager = mô phỏng(SagaManager.class); cancelOrderSagaManager = mô phỏng(SagaManager.class); revisionOrderSagaManager = mô phỏng(SagaManager.class);

orderService = new OrderService(orderRepository, eventPublisher, restaurantRepository, createOrderSagaManager,

cancelOrderSagaManager,sửa đổiOrderSagaManager);

} **Tạo một OrderService được tiêm**

**với các phụ thuộc giả định.**

@Bài kiểm tra

public void shouldCreateOrder() { khi(restaurantRepository

**Cấu hình RestaurantRepository.findById() để trả về nhà hàng Ajanta.**

.findById(AJANTA\_ID)).thenReturn(Tùy chọn.của(AJANTA\_RESTAURANT\_);khi(orderRepository.save(any(Order.class))).sau đó(gọi -> {

Thứ tự thứ tự = (Thứ tự) invocation.getArguments()[0];

đơn hàng.setId(ORDER\_ID);trả lại đơn hàng;

});

**Cấu hình OrderRepository.save()**

**để thiết lập ID của đơn hàng.**

**Gọi OrderService**

**.tạo nên().**

Đặt hàng order = orderService.createOrder(CONSUMER\_ID,

AJANTA\_ID, GÀ\_VINDALOO\_MENU\_MÓN\_VÀ\_SỐ LƯỢNG);

**Xác minh rằng**

**Xác minh rằngDịch vụ đặt hàng**

xác minh(orderRepository).lưu(same(order));

xác minh(eventPublisher).publish(Order.class,ORDER\_ID, danh sách đơn lẻ(

**Dịch vụ đặt hàng lưu Đơn hàng mới tạo vào cơ sở dữ liệu.**

**đã công bố một Lệnh-**

mớiOrderCreatedEvent(CHICKEN\_VINDALOO\_ORDER\_DETAILS)));

**Sự kiện đã tạo.**

xác minh(createOrderSagaManager)

.create(CreateOrderSagaState mới(ORDER\_ID, CHICKEN\_VINDALOO\_ORDER\_DETAILS),

Lớp đơn hàng, ORDER\_ID);

}

**Xác minh rằng Order-Service đã tạo CreateOrderSaga.**

}

Cáccài đặt()phương pháp tạo ra mộtDịch vụ đặt hàngđược tiêm với các phụ thuộc giả.@Test nênTạo đơn hàng()phương pháp xác minh rằngOrderService.createOrder()gọiKho lưu trữ đơn hàngđể lưu những gì mới tạo raĐặt hàng, xuất bản mộtOrderCreated-Sự kiệnvà tạo ra mộtTạo đơn hàngSaga.

Bây giờ chúng ta đã biết cách kiểm tra đơn vị các lớp logic miền, hãy cùng xem cách kiểm tra đơn vị các bộ điều hợp tương tác với các hệ thống bên ngoài.

###### Phát triển các bài kiểm tra đơn vị cho bộ điều khiển

Các dịch vụ, chẳng hạn như Order Service, thường có một hoặc nhiều bộ điều khiển xử lý các yêu cầu HTTP từ các dịch vụ khác và cổng API. Một lớp bộ điều khiển bao gồm một tập hợp các phương thức xử lý yêu cầu. Mỗi phương thức triển khai một điểm cuối REST API. Các tham số của phương thức biểu diễn các giá trị từ yêu cầu HTTP, chẳng hạn như các biến đường dẫn. Nó thường gọi một dịch vụ miền hoặc một kho lưu trữ và trả về một đối tượng phản hồi.

Bộ điều khiển đơn hàng, ví dụ, gọiDịch vụ đặt hàngVàKho lưu trữ đơn hàng. Một chiến lược thử nghiệm hiệu quả cho bộ điều khiển là các bài kiểm tra đơn vị riêng biệt để mô phỏng các dịch vụ và kho lưu trữ.

Bạn có thể viết một lớp kiểm thử tương tự như lớp OrderServiceTest để khởi tạo một lớp điều khiển và gọi các phương thức của lớp đó. Nhưng cách tiếp cận này không kiểm thử một số chức năng quan trọng, chẳng hạn như định tuyến yêu cầu. Sẽ hiệu quả hơn nhiều nếu sử dụng một khuôn khổ kiểm thử MVC giả, chẳng hạn như Spring Mock Mvc, là một phần của Spring Framework hoặc Rest Assured Mock MVC, được xây dựng trên Spring Mock Mvc. Các bài kiểm thử được viết bằng một trong những khuôn khổ này sẽ tạo ra những gì có vẻ là các yêu cầu HTTP và đưa ra các khẳng định về các phản hồi HTTP. Các khuôn khổ này cho phép bạn kiểm thử định tuyến yêu cầu HTTP và chuyển đổi các đối tượng Java sang và từ JSON mà không cần phải thực hiện các cuộc gọi mạng thực sự. Dưới lớp vỏ, Spring Mock Mvc khởi tạo vừa đủ các lớp Spring MVC để thực hiện điều này.

**Đây có thực sự là các bài kiểm tra đơn vị không?**

Bởi vì các bài kiểm tra này sử dụng Spring Framework, bạn có thể cho rằng chúng không phải là đơn vịkiểm tra. Chúng chắc chắn nặng hơn đơn vịcác bài kiểm tra mà tôi đã mô tả cho đến nay. Tài liệu Spring Mock Mvc gọi những bài kiểm tra này là các bài kiểm tra tích hợp ngoài vùng chứa servlet ([https://docs.spring.io/spring/docs/current/spring-framework-reference/](https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-vs-end-to-end-integration-tests) [testing.html#spring-mvc-test-vs-end-to-end-integration-tests](https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-vs-end-to-end-integration-tests)). Nhưng hãy yên tâm MockMVC mô tả các thử nghiệm này như các thử nghiệm đơn vị ([https://github.com/rest-asured/rest-](https://github.com/rest-assured/rest-assured/wiki/Usage#spring-mock-mvc-module) [đảm bảo/wiki/Sử dụng#spring-mock-mvc-module](https://github.com/rest-assured/rest-assured/wiki/Usage#spring-mock-mvc-module)). Bất kể cuộc tranh luận về thuật ngữVề mặt khoa học, đây là những bài kiểm tra quan trọng cần phải viết.

Danh sách 9.6 hiển thịlớp OrderControllerTest, lớp này kiểm tra Order Service's Order-Controller. Nó định nghĩa các bài kiểm tra đơn vị riêng lẻ sử dụng mocks cho các phụ thuộc của OrderController. Nó được viết bằng Rest Assured Mock MVC, cung cấp một DSL đơn giản trừu tượng hóa các chi tiết tương tác với bộ điều khiển. Rest Assured giúp dễ dàng gửi yêu cầu HTTP mock đến bộ điều khiển và xác minh phản hồi. OrderController- Test tạo một bộ điều khiển được tiêm mocks Mockito cho OrderService và OrderRepository. Mỗi bài kiểm tra cấu hình mocks, tạo một yêu cầu HTTP, xác minh rằng phản hồi là chính xác và có thể xác minh rằng bộ điều khiển đã gọi mocks.

**Liệt kê 9.6 Một bài kiểm tra đơn vị chạy nhanh, đơn giản choBộ điều khiển đơn hànglớp học**

lớp công khai OrderControllerTest {

riêng tư OrderService orderService; riêng tư OrderRepository orderRepository;

@Trước

public void setUp() throws Exception { orderService = mock(OrderService.class); orderRepository = mock(OrderRepository.class);

**Tạo bản mô phỏng cho các phụ thuộc của OrderController.**

orderController = new OrderController(orderService, orderRepository);

}

@Bài kiểm tra

công khai void shouldFindOrder() {

khi(orderRepository.findById(1L))

.thenReturn(Tùy chọn.của(CHICKEN\_VINDALOO\_ORDER\_);

đã cho().standaloneSetup(configureControllers(

**Cấu hình OrderRepository giả để trả về một Đơn hàng.**

**Cấu hình OrderController.**

**Làm một**

**Giao thức HTTP**

khi().

OrderController mới(orderService,orderRepository))).

**lời yêu cầu.**

sau đó().

lấy("/đơn hàng/1").

**Xác minh mã trạng thái phản hồi.**

**Xác minh các thành phần của phản hồi JSON**

**thân hình.**

Mã trạng thái(200).

thân("orderId",

bằng(mớiDài(OrderDetailsMother.ORDER\_ID).intValue())). body("state",

bằng(OrderDetailsMother.CHICKEN\_VINDALOO\_ORDER\_STATE.name())).thân("tổng đơn hàng",

bằng(CHICKEN\_VINDALOO\_ORDER\_TOTAL.asString()))

;

}

@Bài kiểm tra

công khai void shouldFindNotOrder() { ... }

Bộ điều khiển StandaloneMockMvcBuilder riêng tư (Bộ điều khiển đối tượng...) { ... }

}

Phương thức kiểm tra shouldFindOrder() đầu tiên cấu hình OrderRepository mock để trả về một Order. Sau đó, nó thực hiện một yêu cầu HTTP để lấy order. Cuối cùng, nó kiểm tra xem yêu cầu có thành công không và nội dung phản hồi có chứa dữ liệu mong đợi không.

Bộ điều khiển không phải là bộ điều hợp duy nhất xử lý các yêu cầu từ hệ thống bên ngoài.

Ngoài ra còn có trình xử lý sự kiện/thông báo, vì vậy chúng ta hãy nói về cách kiểm tra đơn vị những trình xử lý đó.

###### Viết các bài kiểm tra đơn vị cho trình xử lý sự kiện và thông báo

Các dịch vụ thường xử lý các thông báo được gửi bởi các hệ thống bên ngoài. Ví dụ, Order Service có OrderEventConsumer, là một bộ điều hợp thông báo xử lý các sự kiện miền do các dịch vụ khác công bố. Giống như bộ điều khiển, bộ điều hợp thông báo có xu hướng là các lớp đơn giản gọi các dịch vụ miền. Mỗi phương thức của bộ điều hợp thông báo thường gọi một phương thức dịch vụ với dữ liệu từ thông báo hoặc sự kiện.

Chúng ta có thể kiểm tra bộ điều hợp tin nhắn đơn vị bằng cách sử dụng phương pháp tương tự như phương pháp chúng ta sử dụng cho bộ điều khiển kiểm tra đơn vị. Mỗi lần kiểm tra sẽ tạo phiên bản bộ điều hợp tin nhắn, gửi tin nhắn đến kênh và xác minh rằng bản mô phỏng dịch vụ đã được gọi đúng cách. Đằng sau

Tuy nhiên, cơ sở hạ tầng nhắn tin đã bị cắt ngắn, do đó không có trình môi giới tin nhắn nào tham gia. Hãy cùng xem cách kiểm tra lớp OrderEventConsumer.

Danh sách 9.7 hiển thị một phần củaĐơn hàngSự kiệnNgười tiêu dùngKiểm tralớp học, kiểm traĐặt hàng-

EventConsumer. Nó xác minh rằng OrderEventConsumer định tuyến từng sự kiện đến phương thức xử lý thích hợp và gọi đúng OrderService. Bài kiểm tra sử dụng khung Eventuate Tram Mock Messaging, cung cấp DSL dễ sử dụng để viết các bài kiểm tra tin nhắn giả sử dụng cùng định dạng given-when-then như RestĐã đảm bảo. Mỗi lần kiểm tra sẽ khởi tạo OrderEventConsumer được tiêm với một bản dịch vụ Order-Service giả, phát hành một sự kiện miền và xác minh rằng OrderEventConsumer gọi đúng bản dịch vụ giả.

**Liệt kê 9.7 Một bài kiểm tra đơn vị chạy nhanh choĐơn hàngSự kiệnNgười tiêu dùnglớp học**

lớp công khai OrderEventConsumerTest{

riêng tư OrderService orderService;

riêng tư OrderEventConsumer orderEventConsumer;

@Trước

public void setUp() ném ngoại lệ { orderService = mock(OrderService.class);

**Khởi tạoOrderEventConsumer với các phụ thuộc được mô phỏng.**

orderEventConsumer = new OrderEventConsumer(orderService);

}

@Bài kiểm tra

công khai void shouldCreateMenu(){

được cho().

**Cấu hình trình xử lý miền OrderEventConsumer.**

eventHandlers(orderEventConsumer.domainEventHandlers()).

khi().

**Xuất bản một nhà hàng-**

**Tạo**

tổng hợp("net.chrisrichardson.ftgo.restaurantservice.domain.Restaurant",Mã số AJANTA).

xuất bản(nhà hàng mới được tạo(AJANTA\_RESTAURANT\_NAME,

Nhà hàngMother.AJANTA\_RESTAURANT\_MENU))

sau đó().

**sự kiện.**

xác minh(() -> { xác minh(orderService)

.createMenu(ID\_AJANTA,

**Xác minh rằng OrderEventConsumer đã gọi OrderService.createMenu().**

mới RestaurantMenu(RestaurantMother.AJANTA\_RESTAURANT\_MENU\_ITEMS));

})

;

}

}

Cáccài đặt()phương pháp tạo ra mộtĐơn hàngSự kiệnNgười tiêu dùngtiêm một chất giảĐặt hàng-Dịch vụ. CácnênCreateMenu()phương pháp xuất bản một Nhà hàngĐã tạosự kiện và xác minh rằngĐơn hàngSự kiệnNgười tiêu dùngđược triệu tậpOrderService.createMenu(). CácĐơn hàngSự kiệnNgười tiêu dùngKiểm tralớp và các lớp kiểm tra đơn vị khác thực thi cực kỳ nhanh chóng.Các bài kiểm tra đơn vị chỉ chạy trong vài giây.

***Bản tóm tắt* 317**

Nhưng các bài kiểm tra đơn vị không xác minh rằng một dịch vụ, chẳng hạn như Order Service, tương tác đúng cách với các dịch vụ khác. Ví dụ, các bài kiểm tra đơn vị không xác minh rằng một Order có thể được lưu trong MySQL. Chúng cũng không xác minh rằng CreateOrderSaga gửi các thông điệp lệnh theo đúng định dạng đến đúng kênh thông điệp. Và chúng không xác minh rằng sự kiện RestaurantCreated được xử lý bởi OrderEventConsumer có cùng cấu trúc với sự kiện được công bố bởi Restaurant Service. Để xác minh rằng một dịch vụ tương tác đúng cách với các dịch vụ khác, chúng ta phải viết các bài kiểm tra tích hợp. Chúng ta cũng cần viết các bài kiểm tra thành phần để kiểm tra toàn bộ dịch vụ một cách riêng biệt. Chương tiếp theo sẽ thảo luận về cách tiến hành các loại bài kiểm tra đó, cũng như các bài kiểm tra đầu cuối.

#### Bản tóm tắt

* Kiểm thử tự động là nền tảng chính của việc phân phối phần mềm nhanh chóng và an toàn. Hơn nữa, do tính phức tạp vốn có của nó, để tận dụng tối đa kiến ​​trúc microservice, bạn phải tự động hóa các bài kiểm tra của mình.
* Mục đích của một bài kiểm tra là để xác minh hành vi của hệ thống đang được kiểm tra (SUT). Trong định nghĩa này, hệ thống là một thuật ngữ hoa mỹ có nghĩa là thành phần phần mềm đang được kiểm tra. Nó có thể là thứ gì đó nhỏ như một lớp, lớn như toàn bộ ứng dụng hoặc thứ gì đó ở giữa, chẳng hạn như một nhóm các lớp hoặc một dịch vụ riêng lẻ. Một tập hợp các bài kiểm tra liên quan tạo thành một bộ kiểm tra.
* Một cách tốt để đơn giản hóa và tăng tốc một bài kiểm tra là sử dụng test double. Test double là một đối tượng mô phỏng hành vi của sự phụ thuộc của SUT. Có hai loại test double: stub và mock. Stub là một test double trả về giá trị cho SUT. Mock là một test double mà một bài kiểm tra sử dụng để xác minh rằng SUT gọi đúng một sự phụ thuộc.
* Sử dụng kim tự tháp thử nghiệm để xác định nơi tập trung nỗ lực thử nghiệm cho các dịch vụ của bạn. Phần lớn các thử nghiệm của bạn phải là các thử nghiệm đơn vị nhanh, đáng tin cậy và dễ viết. Bạn phải giảm thiểu số lượng các thử nghiệm đầu cuối, vì chúng chậm, dễ vỡ và tốn thời gian để viết.

*Kiểm tra các dịch vụ vi mô:*

*Phần 2*

***Chương này bao gồm***

* Kỹ thuật thử nghiệm dịch vụđang cách ly
* Sử dụng thử nghiệm hợp đồng do người tiêu dùng thúc đẩy để viết các bài kiểm tra xác minh dịch vụ liên ngành một cách nhanh chóng nhưng đáng tin cậygiao tiếp
* Khi nào và làm thế nào để thực hiện thử nghiệm đầu cuối củaứng dụng

Chương này xây dựng dựa trên chương trước, chương này giới thiệu các khái niệm kiểm thử, bao gồm kim tự tháp kiểm thử. Kim tự tháp kiểm thử mô tả tỷ lệ tương đối của các loại kiểm thử khác nhau mà bạn nên viết. Chương trước mô tả cách viết các bài kiểm thử đơn vị, nằm ở phần cơ sở của kim tự tháp kiểm thử. Trong chương này, chúng ta tiếp tục đi lên kim tự tháp kiểm thử.

Chương này bắt đầu với cách viết các bài kiểm tra tích hợp, là cấp độ cao hơn các bài kiểm tra đơn vị trong kim tự tháp kiểm tra. Các bài kiểm tra tích hợp xác minh rằng một dịch vụ có thể tương tác đúng cách với các dịch vụ cơ sở hạ tầng, chẳng hạn như cơ sở dữ liệu và các dịch vụ ứng dụng khác. Tiếp theo, tôi sẽ trình bày các bài kiểm tra thành phần, là các bài kiểm tra chấp nhận cho các dịch vụ. Một bài kiểm tra thành phần kiểm tra một dịch vụ riêng lẻ bằng cách sử dụng các stub cho các phụ thuộc của nó. Sau đó, tôi sẽ mô tả cách viết các bài kiểm tra đầu cuối, kiểm tra một nhóm các dịch vụ hoặc

**318**

toàn bộ ứng dụng. Các bài kiểm tra đầu cuối nằm ở đầu kim tự tháp kiểm tra và do đó, nên được sử dụng một cách tiết kiệm.

Chúng ta hãy bắt đầu bằng cách xem cách viếtkiểm tra tích hợp.

#### Viết các bài kiểm tra tích hợp

Các dịch vụ thường tương tác với các dịch vụ khác. Ví dụ, Order Service, như hình 10.1 cho thấy, tương tác với một số dịch vụ. API REST của nó được API Gateway sử dụng và các sự kiện miền của nó được các dịch vụ, bao gồm Order History Service, sử dụng. Order Service sử dụng một số dịch vụ khác. Nó lưu trữ Orders trong MySQL. Nó cũng gửi lệnh đến và sử dụng các phản hồi từ một số dịch vụ khác, chẳng hạn như Kitchen Service.

Bài kiểm tra

Giao diện lập trình ứng dụng (API)

cổng vào

Người tiêu dùng

Dịch vụ đặt hàng proxy

Nhà cung cấp

Huyền thoại

Lớp đang kiểm tra

Sự kiện

kênh

Nhà cung cấp

Lịch sử đơn hàng

Dịch vụ

Đặt hàng

trình xử lý sự kiện lịch sử

Nhà cung cấp

Đặt hàng

Dịch vụ

Phòng bếp

Dịch vụngười đại diện

Người tiêu dùng Yêu cầukênh

Kênh trả lời

Phòng bếp

Dịch vụ

Phòng bếp

Lệnh dịch vụ

Nhà cung cấp

người xử lý

Bài kiểm tra

Đặt hàng Kho lưu trữ

Bài kiểm tra

Bài kiểm tra

Nhà xuất bản sự kiện tổng hợp đơn hàng

Cơ sở dữ liệu

Bộ điều khiển đơn hàng

**Hình 10.1Các bài kiểm tra tích hợp phải xác minh rằng một dịch vụ có thể giao tiếp với các máy khách và các phụ thuộc của nó. Nhưng thay vì kiểm tra toàn bộ các dịch vụ, chiến lược là kiểm tra các lớp bộ điều hợp riêng lẻ thực hiện giao tiếp.**

Để tin tưởng rằng một dịch vụ như Order Service hoạt động như mong đợi, chúng ta phải viết các bài kiểm tra để xác minh rằng dịch vụ có thể tương tác đúng với các dịch vụ cơ sở hạ tầng và các dịch vụ ứng dụng khác. Một cách tiếp cận là khởi chạy tất cả các dịch vụ và kiểm tra chúng thông qua API của chúng. Tuy nhiên, đây là những gì được gọi là kiểm tra đầu cuối, chậm, dễ hỏng và tốn kém. Như đã giải thích trong phần 10.3, có một vai trò cho đầu cuối

đôi khi phải thử nghiệm, nhưng nó nằm ở đỉnh của kim tự tháp thử nghiệm, vì vậy bạn muốn giảm thiểu số lượng các thử nghiệm đầu cuối.

Một chiến lược hiệu quả hơn nhiều là viết những gì được gọi là kiểm tra tích hợp. Như hình 10.2 cho thấy, kiểm tra tích hợp là lớp trên các kiểm tra đơn vị trong kim tự tháp kiểm tra. Chúng xác minh rằng một dịch vụ có thể tương tác đúng cách với các dịch vụ cơ sở hạ tầng và các dịch vụ khác. Nhưng không giống như các kiểm tra đầu cuối, chúng không khởi chạy các dịch vụ. Thay vào đó, chúng tôi sử dụng một vài chiến lược giúp đơn giản hóa đáng kể các bài kiểm tra mà không ảnh hưởng đến hiệu quả của chúng.

**Hình 10.2 Kiểm thử tích hợp là lớp trên các kiểm thử đơn vị. Chúng xác minh rằng một dịch vụ có thể giao tiếp với các phụ thuộc của nó, bao gồm các dịch vụ cơ sở hạ tầng, chẳng hạn như cơ sở dữ liệu và các dịch vụ ứng dụng.**

Từ đầu đến cuối

Thành phần

Tích hợp

Đơn vị

Chiến lược đầu tiên là kiểm tra từng bộ điều hợp của dịch vụ, cùng với, có lẽ, các lớp hỗ trợ của bộ điều hợp. Ví dụ, trong phần 10.1.1, bạn sẽ thấy một bài kiểm tra tính bền bỉ của JPA xác minh rằng Orders được duy trì đúng cách. Thay vì kiểm tra tính bền bỉ thông qua API của Order Service, nó trực tiếp kiểm tra lớp OrderRepository. Tương tự như vậy, trongphần 10.1.3 bạn sẽ thấy một bài kiểm tra xác minh rằng Order Service xuất bản các sự kiện miền có cấu trúc đúng bằng cách kiểm tra lớp OrderDomainEventPublisher. Lợi ích của việc chỉ kiểm tra một số ít lớp thay vì toàn bộ dịch vụ là các bài kiểm tra đơn giản hơn và nhanh hơn đáng kể.

Chiến lược thứ hai để đơn giản hóa các bài kiểm tra tích hợp xác minh tương tác giữa các dịch vụ ứng dụng là sử dụng hợp đồng, được thảo luận trong chương 9. Hợp đồng là một ví dụ cụ thể về tương tác giữa một cặp dịch vụ. Như bảng 10.1 cho thấy, cấu trúc của hợp đồng phụ thuộc vào loại tương tác giữa các dịch vụ.

**Bảng 10.1 Cấu trúc của hợp đồng phụ thuộc vào loại tương tác giữa các dịch vụ.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Phong cách tương tác** | **Người tiêu dùng** | **Nhà cung cấp** | **Hợp đồng** |
| Dựa trên REST, yêu cầu/phản hồi | Cổng API | Dịch vụ đặt hàng | Yêu cầu HTTP vàphản ứng |
| Xuất bản/đăng ký | Dịch vụ Lịch sử đơn hàng | Dịch vụ đặt hàng | Sự kiện miền |
| Yêu cầu/phản hồi không đồng bộ | Dịch vụ đặt hàng | Dịch vụ nhà bếp | Tin nhắn lệnh và tin nhắn trả lời |

Hợp đồng bao gồm một thông báo, trong trường hợp tương tác theo kiểu xuất bản/đăng ký, hoặc hai thông báo, trong trường hợp tương tác theo kiểu yêu cầu/phản hồi và yêu cầu/phản hồi không đồng bộ.

Các hợp đồng được sử dụng để kiểm tra cả người dùng và nhà cung cấp, đảm bảo rằng họ đồng ý về API. Chúng được sử dụng theo những cách hơi khác nhau tùy thuộc vào việc bạn đang kiểm tra người dùng hay nhà cung cấp:

* *Kiểm tra phía người tiêu dùng*—Đây là những thử nghiệm dành cho người tiêu dùngbộ điều hợp. Họ sử dụng các hợp đồng để cấu hình các đoạn mã mô phỏng nhà cung cấp, cho phép bạn viết các bài kiểm tra tích hợp cho người dùng mà không cần nhà cung cấp đang chạy.
* *Kiểm tra phía nhà cung cấp*—Đây là các bài kiểm tra cho bộ điều hợp của nhà cung cấp. Chúng sử dụng các hợp đồng để kiểm tra các bộ điều hợp bằng cách sử dụng các bản mô phỏng cho các phụ thuộc của bộ điều hợp.

Ở phần sau của phần này, tôi sẽ mô tả các ví dụ về các loại thử nghiệm này, nhưng trước tiên chúng ta hãy xem cách viết các thử nghiệm về tính bền bỉ.

###### Kiểm tra tích hợp tính bền bỉ

Các dịch vụ thường lưu trữ dữ liệu trong cơ sở dữ liệu. Ví dụ, Order Service lưu trữ các tập hợp, chẳng hạn như Order, trong MySQL bằng JPA. Tương tự, Order History Service duy trì chế độ xem CQRS trong AWS DynamoDB. Các bài kiểm tra đơn vị mà chúng tôi đã viết trước đó chỉ kiểm tra các đối tượng trong bộ nhớ. Để tin tưởng rằng một dịch vụ hoạt động chính xác, chúng tôi phải viết các bài kiểm tra tích hợp tính bền bỉ, xác minh rằng logic truy cập cơ sở dữ liệu của dịch vụ hoạt động như mong đợi. Trong trường hợp của Order Service, điều này có nghĩa là kiểm tra các kho lưu trữ JPA, chẳng hạn như OrderRepository.

Mỗi giai đoạn của thử nghiệm tích hợp tính bền bỉ diễn ra như sau:

* + - * *Cài đặt*—Thiết lập cơ sở dữ liệu bằng cách tạo lược đồ cơ sở dữ liệu và khởi tạo nó ở trạng thái đã biết. Nó cũng có thể bắt đầu một giao dịch cơ sở dữ liệu.
      * *Thực hiện*—Thực hiện thao tác cơ sở dữ liệu.
      * *Xác minh*—Đưa ra khẳng định về trạng thái của cơ sở dữ liệu và các đối tượng được lấy từ cơ sở dữ liệu.
      * *Tháo dỡ*—Giai đoạn tùy chọn có thể hoàn tác các thay đổi được thực hiện đối với cơ sở dữ liệu bằng cách, ví dụ, khôi phục giao dịch đã được bắt đầu bởi giai đoạn thiết lập.

Danh sách 10.1 hiển thị một thử nghiệm tích hợp liên tục choOrder aggregate và Order-Repository. Ngoài việc dựa vào JPA để tạo lược đồ cơ sở dữ liệu, các bài kiểm tra tích hợp tính bền bỉ không đưa ra bất kỳ giả định nào về trạng thái của cơ sở dữ liệu. Do đó, các bài kiểm tra không cần phải khôi phục các thay đổi mà chúng thực hiện đối với cơ sở dữ liệu, điều này tránh được các vấn đề với ORM lưu trữ dữ liệu thay đổi trong bộ nhớ đệm.

**Liệt kê 10.1 Một bài kiểm tra tích hợp xác minh rằng mộtĐặt hàngcó thể được duy trì**

@RunWith(Lớp SpringRunner)

@SpringBootTest(các lớp = OrderJpaTestConfiguration.class) lớp công khai OrderJpaTest {

@Autowired

OrderRepository riêng tư orderRepository;

@Autowired

riêng tư TransactionTemplate transactionTemplate;

@Bài kiểm tra

công khai void shouldSaveAndLoadOrder() {

OrderId dài = transactionTemplate.execute((ts) -> { Order order =

Đơn hàng mới(CONSUMER\_ID, AJANTA\_ID, CHICKEN\_VINDALOO\_LINE\_ITEMS);

orderRepository.save(thứ tự);trả về lệnh.getId();

});

transactionTemplate.execute((ts) -> {

Đặt hàng = orderRepository.findById(orderId).get();

assertEquals(OrderState.APPROVAL\_PENDING, order.getState()); assertEquals(AJANTA\_ID, order.getRestaurantId()); assertEquals(CONSUMER\_ID, order.getConsumerId().longValue()); assertEquals(CHICKEN\_VINDALOO\_LINE\_ITEMS, order.getLineItems()); trả về null;

});

}

}

Phương thức kiểm tra shouldSaveAndLoadOrder() thực hiện hai giao dịch. Giao dịch đầu tiên lưu Order mới tạo trong cơ sở dữ liệu. Giao dịch thứ hai tải Order và xác minh rằng các trường của nó được khởi tạo đúng cách.

Một vấn đề bạn cần giải quyết là cách cung cấp cơ sở dữ liệu được sử dụng trong các thử nghiệm tích hợp liên tục. Một giải pháp hiệu quả để chạy một phiên bản của cơ sở dữ liệu trong quá trình thử nghiệm là sử dụng Docker. Phần 10.2 mô tả cách sử dụng plugin Docker Compose Gradle để tự động chạy các dịch vụ trong quá trình thử nghiệm thành phần. Bạn có thể sử dụng một cách tiếp cận tương tự để chạy MySQL, ví dụ, trong quá trình thử nghiệm tích hợp liên tục.

Cơ sở dữ liệu chỉ là một trong những dịch vụ bên ngoài mà một dịch vụ tương tácvới. Bây giờ chúng ta hãy xem cách viết các bài kiểm tra tích hợp cho giao tiếp giữa các dịch vụ ứng dụng, bắt đầu với REST.

###### Kiểm thử tích hợp theo phong cách yêu cầu/phản hồi dựa trên RESTtương tác

REST là cơ chế giao tiếp liên dịch vụ được sử dụng rộng rãi. Máy khách REST và dịch vụ REST phải thống nhất về REST API, bao gồm các điểm cuối REST và cấu trúc của các phần yêu cầu và phản hồi. Máy khách phải gửi yêu cầu HTTP đến điểm cuối chính xác và dịch vụ phải gửi lại phản hồi mà máy khách mong đợi.

Ví dụ, chương 8 mô tả cách ứng dụng FTGOCổng APIthực hiện các cuộc gọi REST API ĐẾN nhiều dịch vụ, bao gồm Dịch vụ tiêu dùng, Dịch vụ đặt hàng, VàDịch vụ giao hàng.CácDịch vụ đặt hàng'SNHẬN /đơn hàng/{orderId}điểm cuối là mộtcủa các điểm cuối được gọi bởiCổng API. Để có thể tự tin rằngCổng APIVàDịch vụ đặt hàngcó thể giao tiếp mà không cần sử dụng thử nghiệm đầu cuối, chúng ta cần viết các thử nghiệm tích hợp.

Như đã nêu trong chương trước, một chiến lược kiểm thử tích hợp tốt là sử dụng các thử nghiệm hợp đồng do người tiêu dùng điều khiển. Tương tác giữa API Gateway và GET

/đơn hàng/{orderId}có thể được mô tả bằng cách sử dụng một tập hợp các hợp đồng dựa trên HTTP. Mỗi con-

tract bao gồm một yêu cầu HTTP và một phản hồi HTTP. Các hợp đồng được sử dụng để kiểm tra

Cổng APIVàDịch vụ đặt hàng.

Hình 10.3 cho thấy cách sử dụng Spring Cloud Contract để kiểm tra các tương tác dựa trên REST. Các bài kiểm tra tích hợp API Gateway phía người dùng sử dụng các hợp đồng để cấu hình máy chủ HTTP stub mô phỏng hành vi của Order Service. Yêu cầu của hợp đồng chỉ định một yêu cầu HTTP từ API gateway và phản hồi của hợp đồng chỉ định phản hồi mà stub gửi lại cho API gateway. Spring Cloud Contract sử dụng các hợp đồng để tạo mã các bài kiểm tra tích hợp Order Service phía nhà cung cấp, kiểm tra các bộ điều khiển bằng Spring Mock MVC hoặc Rest Assured Mock MVC. Yêu cầu của hợp đồng chỉ định yêu cầu HTTP cần thực hiện với bộ điều khiển và phản hồi của hợp đồng chỉ định phản hồi mong đợi của bộ điều khiển.

Phía người tiêu dùngKiểm tra dịch vụ OrderServiceProxygọiDịch vụ đặt hàngProxy, đã được cấu hình để thực hiện các yêu cầu HTTP tới WireMock. WireMock là một công cụ để hiệu quảchế nhạo máy chủ HTTP một cách hiệu quả—trong thử nghiệm này, nó mô phỏngDịch vụ đặt hàng.Mây mùa xuân

Kiểm tra tích hợp phía người tiêu dùng cho cổng API

Kiểm tra tích hợp phía nhà cung cấp cho Dịch vụ đặt hàng

Kiểm tra



lớp HttpTest mở rộng BaseHttp {

}

lớp trừu tượng BaseHttp {

@Trước

công khai void setup() { RestAssuredMockMvc

.standaloneSetup(...);

}

}

Hợp đồng.thực hiện {lời yêu cầu {..}

phản ứng{...}

}

}

Cấu hình

Dịch vụ đặt hàngNgười ủy nhiệm

Máy chủ HTTP stub dựa trên Wiremock

Sử dụng

Dịch vụ đặt hàngKiểm tra Proxy

|  |  |
| --- | --- |
|  | Đọc |
| Mây mùa xuânHợp đồng | |

Giao thức HTTP

Tạo ra

Kiểm tra

Cấu hình

Bộ điều khiển đơn hàng

**Hình 10.3 Các hợp đồng được sử dụng để xác minh rằng các lớp bộ điều hợp ở cả hai bên của**

**Giao tiếp dựa trên REST giữaCổng APIVàDịch vụ đặt hàngtuân thủ hợp đồng. Các thử nghiệm phía người tiêu dùng xác minh rằngDịch vụ đặt hàngProxygọiDịch vụ đặt hàngđúng. Các thử nghiệm phía nhà cung cấp xác minh rằngBộ điều khiển đơn hàngtriển khai các điểm cuối của REST API một cách chính xác.**

Contract quản lý WireMock và cấu hình nó để phản hồi các yêu cầu HTTP được xác định bởi các hợp đồng.

Về phía nhà cung cấp, Spring Cloud Contract tạo ra một lớp thử nghiệm được gọi làKiểm tra Http,sử dụng Rest Assured Mock MVC để kiểm traDịch vụ đặt hàngbộ điều khiển của. Các lớp kiểm tra nhưKiểm tra Httpphải mở rộng một lớp cơ sở viết tay. Trong ví dụ này, lớp cơ sởCơ sởHttpkhởi tạoBộ điều khiển đơn hàngđược tiêm với các phụ thuộc và cuộc gọi giảRestAssuredMockMvc.standaloneSetup()để cấu hình Spring MVC.

Chúng ta hãy xem xét kỹ hơn cách thức hoạt động của nó, bắt đầu bằngví dụ hợp đồng.

**MỘTN VÍ DỤ HỢP ĐỒNG CHO AGiao diện lập trình ứng dụng REST**

Hợp đồng REST, chẳng hạn như hợp đồng được hiển thị trong danh sách 10.2, chỉ định yêu cầu HTTP, được gửi bởi máy khách REST và phản hồi HTTP, mà máy khách mong đợi nhận được từ máy chủ REST. Yêu cầu của hợp đồng chỉ định phương thức HTTP, đường dẫn và tiêu đề tùy chọn. Phản hồi của hợp đồng chỉ định mã trạng thái HTTP, tiêu đề tùy chọn và, khi thích hợp, nội dung mong đợi.

**Danh sách 10.2 Một hợp đồng mô tả tương tác theo kiểu yêu cầu/phản hồi dựa trên HTTP**

org.springframework.cloud.contract.spec.Contract.make{ lời yêu cầu {

phương pháp 'GET'

url '/orders/1223232'

}

phản ứng {

trạng thái 200 tiêu đề {

header('Kiểu-Nội-dung':'ứng dụng/json;charset=UTF-8')

}

thân('''{"orderId" : "1223232", "state" : "ĐANG CHỜ PHÊ DUYỆT"}''')

}

}

Hợp đồng cụ thể này mô tảmột nỗ lực thành công của API Gateway để lấy lại mộtĐặt hàng từ Order Service. Bây giờ chúng ta hãy xem cách sử dụng hợp đồng này để viết các bài kiểm tra tích hợp, bắt đầu với các bài kiểm tra cho Order Service.

**CNGƯỜI TIÊU DÙNG-KIỂM TRA TÍCH HỢP ... ĐỒNG ĐƯỢC ĐIỀU KHIỂN CHOỒĐẠI HỌCSDỊCH VỤ**

Các bài kiểm tra tích hợp hợp đồng do người dùng điều khiển cho Order Service xác minh rằng API của nó đáp ứng được kỳ vọng của khách hàng. Liệt kê 10.3 hiển thị HttpBase, là lớp cơ sở cho lớp kiểm tra được tạo ra bởi Spring Cloud Contract. Nó chịu trách nhiệm cho giai đoạn thiết lập của bài kiểm tra. Nó tạo ra các bộ điều khiển được tiêm với các phụ thuộc giả và cấu hình các bản giả đó để trả về các giá trị khiến bộ điều khiển tạo ra phản hồi mong đợi.

**Danh sách 10.3 Lớp cơ sở trừu tượng cho các bài kiểm tra được tạo ra bởi Spring Cloud Contract**

lớp trừu tượng công khaiCơ sở dữ liệu Http

Bộ điều khiển StandaloneMockMvcBuilder riêng tư (Bộ điều khiển đối tượng...) {

...

trả về MockMvcBuilders.standaloneSetup(bộ điều khiển)

.setMessageConverters(...);

}

**Tạo OrderRepository**

@Trước

công khai void thiết lập() {

**được tiêm thuốc giả.**

OrderService orderService = mock(OrderService.class); Kho lưu trữ đơn hàng orderRepository = mock(OrderRepository.class);

Bộ điều khiển đơn hàng =

mới OrderController(orderService, orderRepository);

khi(orderRepository.findById(1223232L))

.thenReturn(Tùy chọn.của(OrderDetailsMother.CHICKEN\_VINDALOO\_ORDER));

...

RestAssuredMockMvc.standaloneSetup(bộ điều khiển(orderController));

} **Cấu hình Spring MVC với**

} **Người điều khiển đơn hàng.**

**Cấu hình OrderResponse để trả về một Order khi findById() được gọi với orderId được chỉ định trong hợp đồng.**

Lập luận1223232Lđiều đó đã được chuyển đến chế nhạoKho lưu trữ đơn hàng'StìmTheoId()phương pháp phù hợp vớiMã đơn hàngđược chỉ định trong hợp đồng được hiển thị trong danh sách 10.3. Bài kiểm tra này xác minh rằngDịch vụ đặt hàngcó mộtNHẬN /đơn hàng/{orderId}điểm cuối phù hợp với mong đợi của khách hàng.

Chúng ta hãy cùng xem xét bài kiểm tra máy khách tương ứng.

**CNGƯỜI TIÊU DÙNG-KIỂM TRA TÍCH HỢP BÊN CHOAPIGATEWAY'SỒ****ĐẠI HỌCSDỊCH VỤPROXY**

Cổng API'SDịch vụ đặt hàngProxygọi làNHẬN /đơn hàng/{orderId}điểm cuối. Danh sách-ing 10.4 cho thấyKiểm tra tích hợp OrderServiceProxylớp kiểm tra, xác minh rằngnó phù hợp với các hợp đồng. Lớp này được chú thích bằng@AutoConfigureStubRunner, được cung cấp bởi Spring Cloud Contract. Nó yêu cầu Spring Cloud Contract chạy máy chủ Wire-Mock trên một cổng ngẫu nhiên và cấu hình nó bằng các hợp đồng đã chỉ định.Đặt hàng- ServiceProxyIntegrationTestcấu hìnhDịch vụ đặt hàngProxyđể gửi yêu cầu tới cổng WireMock.

**Liệt kê 10.4 Một thử nghiệm tích hợp phía người tiêu dùng choCổng API'SDịch vụ đặt hàngProxy**

**Nhận được ngẫu nhiêncổng được chỉ định mà WireMock đang chạy.**

@RunWith(SpringRunner.class) @SpringBootTest(lớp=TestConfiguration.class,

**Yêu cầu Spring Cloud Contract cấu hình WireMock với các hợp đồng của Order Service.**

webEnvironment= SpringBootTest.WebEnvironment.NONE)@AutoConfigureStubRunner(id =

{"net.chrisrichardson.ftgo.contracts:ftgo-order-service-contracts"},workOffline = sai)

@BẩnBốiCảnh

lớp công khai OrderServiceProxyIntegrationTest { @Value("${stubrunner.runningstubs.ftgo-order-service-contracts.port}")

cổng int riêng tư;

riêng tư OrderDestinations orderDestinations; riêng tư OrderServiceProxy orderService;

@Trước

public void setUp() ném ra ngoại lệ { orderDestinations = new OrderDestinations();

**Tạo một OrderServiceProxy được cấu hình để thực hiện các yêu cầu**

**đến WireMock.**

Chuỗi orderServiceUrl = "[http://localhost:"](http://localhost/)+ cổng; orderDestinations.setOrderServiceUrl(orderServiceUrl); orderService = new OrderServiceProxy(orderDestinations,

WebClient.create());

}

@Bài kiểm tra

công khai void shouldVerifyExistingCustomer() {

Kết quả OrderInfo = orderService.findOrderById("1223232").block(); assertEquals("1223232", result.getOrderId()); assertEquals("ĐANG CHỜ PHÊ DUYỆT", result.getState());

}

@Test(dự kiến ​​= OrderNotFoundException.class) public void shouldFailToFindMissingOrder() {

orderService.findOrderById("555").block();

}

}

Mỗi phương pháp thử nghiệm gọiDịch vụ đặt hàngProxyvà xác minh rằng nó trả về các giá trị chính xác hoặc đưa ra ngoại lệ mong đợi.nênXác minhHiện tại-Khách hàng()phương pháp thử nghiệm xác minh rằngtìmOrderById()trả về các giá trị bằng với các giá trị được chỉ định trong phản hồi của hợp đồng.shouldFailToFindMissingOrder()cố gắng để lấy lại một cái không tồn tạiĐặt hàngvà xác minh rằngDịch vụ đặt hàngProxyném mộtĐặt hàng-NotFoundException. Kiểm tra cả máy khách REST và dịch vụ REST bằng cùng một hợp đồng để đảm bảo rằng chúng đồng ý về API.

Bây giờ chúng ta hãy xem cách thực hiện thử nghiệm tương tự cho các dịch vụ tương tác bằng cách sử dụng tin nhắn.

###### TRONGtích hợp thử nghiệmxuất bản/suphong cách bscribeTRONGtương tác

Các dịch vụ thường công bố các sự kiện miền được một hoặc nhiều dịch vụ khác sử dụng. Kiểm thử tích hợp phải xác minh rằng nhà xuất bản và người tiêu dùng của họ đồng ý về kênh tin nhắn và cấu trúc của các sự kiện miền. Ví dụ, Order Servicecông bố các sự kiện Order\* bất cứ khi nào nó tạo hoặc cập nhật tổng hợp Order. Order History Service là một trong những người tiêu thụ các sự kiện đó. Do đó, chúng ta phải viết các bài kiểm tra để xác minh rằng các dịch vụ này có thể tương tác.

Hình 10.4 cho thấy cách tiếp cận để kiểm tra tích hợp tương tác publish/subscribe. Nó khá giống với cách tiếp cận được sử dụng để kiểm tra tương tác REST. Như trước đây, các tương tác được xác định bởi một tập hợp các hợp đồng. Điểm khác biệt là mỗi hợp đồng chỉ định một sự kiện miền.

**Lớp họcdướiBài kiểm tra Lớp dướiBài kiểm tra**

Kiểm tra tích hợp phía người tiêu dùng cho Dịch vụ Lịch sử đơn hàng

|  |  |
| --- | --- |
| Kiểm tra sự kiện OrderHistory EventHandlers | |
|  | Kiểm tra |

Sử dụng



Tích hợp phía nhà cung cấp

Bài kiểm trađể Đặt dịch vụ

Triệu hồi

Xuất bản tới

Mã số

tạo ra

Kiểm tra

Đọc từ

lớp MessageTest mở rộng MessagingBase{@Bài kiểm tra

công khai void validate\_orderCreatedEvent(){lệnh đã tạo();

...

}

}

kênh mong đợi

đặt

Gọi hàm kích hoạtxác minh rằng tin nhắn đầu ra được công bố tới

Đặt hàng tên miền EventPublisher

Kênh

Mây mùa xuânhợp đồng

lớp MessagingBase{ void orderCreated(){

}

hợp đồng.làm{

nhãn 'orderCreatedEvent'đầu vào{

được kích hoạt bởi('orderCreated()')

}

đầu raMessage{...}

}

Trình xử lý sự kiện OrderHistory

Kích hoạt 'orderCreatedEvent'

Đọc từ

Kênh

Xuất bản tới

Cấu hình

Tin nhắn gốc

**Hình 10.4 Các hợp đồng được sử dụng để kiểm tra cả hai mặt của tương tác xuất bản/đăng ký. Các thử nghiệm phía nhà cung cấp xác minh rằngNhà xuất bản sự kiện miền đặt hàngcông bố các sự kiện xác nhận hợp đồng. Các thử nghiệm phía người tiêu dùng xác minh rằngTrình xử lý sự kiện OrderHistorytiêu thụ các sự kiện ví dụ từ**

**hợp đồng.**

Mỗi thử nghiệm phía người tiêu dùng sẽ công bố sự kiện được chỉ định bởi hợp đồng và xác minh rằng

Trình xử lý sự kiện OrderHistorygọi các phụ thuộc giả định của nó một cách chính xác.

Về phía nhà cung cấp, Spring Cloud Contract tạo mã các lớp kiểm tra mở rộng MessagingBase, là một siêu lớp trừu tượng được viết thủ công. Mỗi phương thức kiểm tra gọi một phương thức hook được định nghĩa bởi MessagingBase, được mong đợi sẽ kích hoạt việc xuất bản một sự kiện của dịch vụ. Trong ví dụ này, mỗi phương thức hook gọiOrderDomainEventPublisher, chịu trách nhiệm xuất bản tổng hợp đơn hàngsự kiện. Sau đó, phương pháp kiểm tra xác minh rằng OrderDomainEventPublisher đã xuất bản sự kiện mong đợi. Hãy xem chi tiết về cách thức hoạt động của các bài kiểm tra này, bắt đầu với hợp đồng.

**THỢP ĐỒNG XUẤT BẢNỒĐẠI HỌCCSỰ KIỆN ĐÃ DIỄN RA**

Danh sách 10.5 hiển thị hợp đồng cho sự kiện OrderCreated. Nó chỉ định kênh của sự kiện, cùng với phần thân và tiêu đề tin nhắn dự kiến.

**Danh sách 10.5 Một hợp đồng cho kiểu tương tác xuất bản/đăng ký**

hợp đồng gói; org.springframework.cloud.contract.spec.Contract.make {

nhãn 'orderCreatedEvent'

đầu vào {

**Được sử dụng bởi thử nghiệm người tiêu dùng để kích hoạt sự kiện được công bố**

được kích hoạt bởi('orderCreated()')

}

**Được gọi bởi nhà cung cấp thử nghiệm được tạo ra bởi mã**

**Một sự kiện tên miền được tạo ra**

đầu raMessage { đã gửi tới('net.chrisrichardson.ftgo.orderservice.domain.Order') nội dung('''{"orderDetails":{"lineItems":[{"quantity":5,"menuItemId":"1",

"tên":"Gà Vindaloo","giá":"12,34","tổng":"61,70"}],"orderTotal:"61.70","restaurantId":1,

"consumerId":1511300065921},"orderState":"ĐANG CHỜ XÉT DUYỆT"}''')

tiêu đề {

tiêu đề('sự kiện-tổng hợp-kiểu',

'net.chrisrichardson.ftgo.orderservice.domain.Order')tiêu đề('sự kiện-tổng hợp-id', '1')

}

}

}

Hợp đồng cũng có hai yếu tố quan trọng khác:

* nhãn—được sử dụng bởi một thử nghiệm của người tiêu dùng để kích hoạt việc công bố sự kiệnbởi Spring Contact
* được kích hoạtBởi—têncủa phương thức siêu lớp được phương thức kiểm tra được tạo ra gọi để kích hoạt việc xuất bản sự kiện

Hãy cùng xem cách sử dụng hợp đồng, bắt đầu với thử nghiệm phía nhà cung cấp cho Đơn hàng-Dịch vụ.

**CNGƯỜI TIÊU DÙNG-ĐƯỢC LÁI XEKIỂM TRA HỢP ĐỒNG CHOỒĐẠI HỌCSDỊCH VỤ**

Kiểm tra phía nhà cung cấp cho Order Service là một kiểm tra tích hợp hợp đồng do người tiêu dùng điều khiển khác. Kiểm tra này xác minh rằng OrderDomainEventPublisher, chịu trách nhiệm xuất bản các sự kiện miền tổng hợp Order, xuất bản các sự kiện phù hợp với kỳ vọng của khách hàng. Liệt kê 10.6 hiển thị MessagingBase, là lớp cơ sở cho các lớp kiểm tra do Spring Cloud Contract tạo ra. Nó chịu trách nhiệm cấu hình lớp OrderDomainEventPublisher để sử dụng tin nhắn trong bộ nhớ gốc cây. Nó Mà còn định nghĩa các phương thức, chẳng hạn như orderCreated(), được các bài kiểm tra được tạo ra gọi để kích hoạt việc xuất bản sự kiện.

**Danh sách 10.6 Lớp cơ sở trừu tượng cho các bài kiểm tra phía nhà cung cấp Spring Cloud Contract**

@RunWith(Lớp SpringRunner)

@SpringBootTest(lớp = MessagingBase.TestConfiguration.class, webEnvironment = SpringBootTest.WebEnvironment.NONE)

@AutoConfigureMessageVerifier

lớp trừu tượng công khai MessagingBase {

@Cấu hình @EnableAutoConfiguration

@Import({EventuateContractVerifierConfiguration.class,TramEventsPublisherConfiguration.class, TramInMemoryConfiguration.class})

lớp tĩnh công khai TestConfiguration {

@Đậu

công khai OrderDomainEventPublisher OrderDomainEventPublisher(DomainEventPublisher eventPublisher) {

trả về OrderDomainEventPublisher(eventPublisher) mới;

}

} **orderCreated() được gọi bởi một**

**lớp kiểm tra do mã tạo ra để công bố sự kiện.**

@Autowired

riêng tư OrderDomainEventPublisherNhà xuất bản sự kiện OrderDomain;

được bảo vệ void orderCreated() { OrderDomainEventPublisher.publish(CHICKEN\_VINDALOO\_ORDER,

singletonList(mới OrderCreatedEvent(CHICKEN\_VINDALOO\_ORDER\_DETAILS)

));

}

}

Bài kiểm tra nàylớp cấu hình OrderDomainEventPublisher với các stub nhắn tin trong bộ nhớ. orderCreated() được gọi bởi phương thức kiểm tra được tạo từ hợp đồng được hiển thị trước đó trong danh sách 10.5. Nó gọi OrderDomainEventPublisher để xuất bản sự kiện OrderCreated. Phương thức kiểm tra cố gắng nhận sự kiện này và sau đó xác minh rằng nó khớp với sự kiện được chỉ định trong hợp đồng. Bây giờ chúng ta hãy xem các bài kiểm tra phía người tiêu dùng tương ứng.

**CNGƯỜI TIÊU DÙNG-KIỂM TRA HỢP ĐỒNG PHỤCHOỒĐẠI HỌCHCÂU CHUYỆNSDỊCH VỤ**

Dịch vụ Lịch sử đơn hàngtiêu thụ các sự kiện được xuất bản bởiDịch vụ đặt hàng. Như tôi đã mô tả trong chương 7, lớp bộ điều hợp xử lý các sự kiện này làOrderHistoryEvent-Người xử lýlớp. Trình xử lý sự kiện của nó gọiLịch sử đơn hàngDaođể cập nhật chế độ xem CQRS. Liệt kê 10.7 hiển thị thử nghiệm tích hợp phía người tiêu dùng. Nó tạo ra mộtOrderHistoryEvent- Trình xử lýtiêm một chất giảLịch sử đơn hàngDao. Mỗi phương pháp kiểm tra đầu tiên sẽ gọi SpringĐám mây để công bố sự kiện được xác định trong hợp đồng và sau đó xác minh rằngOrderHistory- EventHandlersgọiLịch sử đơn hàngDaomột cách chính xác.

**Liệt kê 10.7 Kiểm tra tích hợp phía người tiêu dùng choTrình xử lý sự kiện OrderHistory**

**lớp học**

@RunWith(Lớp SpringRunner)

@SpringBootTest(các lớp= OrderHistoryEventHandlersTest.TestConfiguration.class,webEnvironment= SpringBootTest.WebEnvironment.NONE)

@AutoConfigureStubRunner(id =

{"net.chrisrichardson.ftgo.contracts:ftgo-order-service-contracts"},workOffline = sai)

@BẩnBốiCảnh

lớp công khai OrderHistoryEventHandlersTest {

@Cấu hình @EnableAutoConfiguration

@Import({OrderHistoryServiceMessagingConfiguration.class,TramCommandProducerConfiguration.class, TramInMemoryConfiguration.class, EventuateContractVerifierConfiguration.class})

lớp tĩnh công khai TestConfiguration {

@Đậu

public OrderHistoryDao orderHistoryDao() { return mock(OrderHistoryDao.class);

}

}

**Tạo một OrderHistoryDao giả để đưa vào OrderHistory-EventHandlers.**

**Kích hoạt orderCreatedEvent**

@Bài kiểm tra

public void shouldHandleOrderCreatedEvent() ném ... {stubFinder.trigger("sự kiện tạo đơn hàng");

sau cùng(()-> {

**stub, phát ra sự kiện OrderCreated.**

});

}

xác minh(orderHistoryDao).addOrder(bất kỳ(Order.class), bất kỳ(Optional.class));

**Xác minh rằng OrderHistoryEventHandlers đã được gọi orderHistoryDao.addOrder().**

CácshouldHandleOrderCreatedEvent()phương pháp kiểm tra cho biết Spring Cloud Contractcông bốĐơn hàngĐã tạosự kiện. Sau đó, nó xác minh rằngTrình xử lý sự kiện OrderHistoryđược triệu tậporderHistoryDao.addOrder(). Kiểm tra cả publisher và consumer của sự kiện miền bằng cùng một hợp đồng đảm bảo rằng chúng đồng ý về API. Bây giờ chúng ta hãy xem cách thực hiện kiểm tra tích hợp các dịch vụ tương tác bằng cách sử dụng yêu cầu/phản hồi không đồng bộ.

###### Kiểm tra hợp đồng tích hợp cho yêu cầu/phản hồi không đồng bộtương tác

Xuất bản/đăng ký không phải là kiểu tương tác duy nhất dựa trên tin nhắn. Các dịch vụ cũng tương tác bằng cách sử dụng yêu cầu/phản hồi không đồng bộ. Ví dụ, trong chương 4 chúng ta đã thấyDịch vụ Đặt hàng triển khai các saga gửi tin nhắn lệnh đến nhiều dịch vụ khác nhau, chẳng hạn như Dịch vụ Nhà bếp, và xử lý các tin nhắn trả lời.

Hai bên trong tương tác yêu cầu/phản hồi không đồng bộ là bên yêu cầu, là dịch vụ gửi lệnh và bên trả lời, là dịch vụ xử lý lệnh và gửi lại phản hồi. Họ phải thống nhất về tên kênh tin nhắn lệnh và cấu trúc của tin nhắn lệnh và phản hồi. Hãy cùng xem cách viết các bài kiểm tra tích hợp cho tương tác yêu cầu/phản hồi không đồng bộ.

Hình 10.5 cho thấy cách kiểm tra tương tác giữa Order Service và Kitchen Service. Cách tiếp cận để kiểm tra tích hợp tương tác yêu cầu/phản hồi không đồng bộ khá giống với cách tiếp cận được sử dụng để kiểm tra tương tác REST. Tương tác giữa các dịch vụ được xác định bởi một tập hợp các hợp đồng. Điểm khác biệt là hợp đồng chỉ định một thông báo đầu vào và một thông báo đầu ra thay vì một yêu cầu và phản hồi HTTP.

Kiểm tra tích hợp phía người tiêu dùng cho Dịch vụ nhà bếp



Đọc

Kiểm tra

Cấu hình

Bên ngoài

Nhận được từ

Gửi

lệnh trả lời

Mã số

tạo ra

Gửi đến

Gửi

yêu cầuTin nhắn

Đọc repl

Cấu hình

lớp MessageTest mở rộng BaseMessaging{

}

xác minh rằng câu trả lời phù hợp

thông điệp đầu ra của hợp đồng

Mây mùa xuânhợp đồng

Gửi tin nhắn đầu vào và

nds

lớp trừu tượng BaseMessaging{

void setUp(){...}

Tin nhắn gốc

Kênh trả lời

Kênh trả lời

Yêu cầukênh

Dịch vụ nhà bếpNgười ủy nhiệm

Yêu cầukênh

KitchenService Proxy IntegrationTest

Hợp đồng.make { inputMessage{...}

Thông điệp đầu ra{...}

}

Kiểm tra tích hợp phía nhà cung cấp cho Dịch vụ nhà bếp

GửiĐẾN

Triệu hồi

Đọc

yêu cầu

Bộ điều khiển lệnh KitchenService

«giả» KitchenService

Nhận được

từ và

**Hình 10.5 Các hợp đồng được sử dụng để kiểm tra các lớp bộ điều hợp triển khai mỗi bên của tương tác yêu cầu/phản hồi không đồng bộ. Các thử nghiệm phía nhà cung cấp xác minh rằngKitchenServiceCommandHandlerxử lý lệnh và gửi lại phản hồi. Các thử nghiệm phía người tiêu dùng xác minhDịch vụ nhà bếpProxygửi lệnhtuân thủ hợp đồng và xử lý các câu trả lời mẫu từ hợp đồng.**

Kiểm tra phía người tiêu dùng xác minh rằng lớp proxy tin nhắn lệnh gửi tin nhắn lệnh có cấu trúc đúng và xử lý đúng tin nhắn trả lời. Trong ví dụ này, KitchenServiceProxyTest kiểm tra KitchenServiceProxy. Nó sử dụng Spring Cloud Contract để cấu hình các đoạn mã tin nhắn xác minh rằng tin nhắn lệnh khớp với tin nhắn đầu vào của hợp đồng và trả lời bằng tin nhắn đầu ra tương ứng.

Các bài kiểm tra phía nhà cung cấp được tạo ra bằng mã bởi Spring Cloud Contract. Mỗi phương pháp kiểm tra tương ứng với một hợp đồng. Nó gửi tin nhắn đầu vào của hợp đồng dưới dạng tin nhắn lệnh và xác minh rằng tin nhắn trả lời khớp với tin nhắn đầu ra của hợp đồng. Hãy xem xét các chi tiết, bắt đầu với hợp đồng.

**EVÍ DỤYÊU CẦU KHÔNG ĐỒNG BỘ/HỢP ĐỒNG PHẢN HỒI**

Liệt kê 10.8 cho thấy hợp đồng cho một tương tác. Nó bao gồm một tin nhắn đầu vào và một tin nhắn đầu ra. Cả hai tin nhắn đều chỉ định một kênh tin nhắn, nội dung tin nhắn và tiêu đề tin nhắn. Quy ước đặt tên là từ góc nhìn của nhà cung cấp. Phần tử messageFrom của tin nhắn đầu vào chỉ định kênh mà tin nhắn được đọc từ đó.

Tương tự như vậy, phần tử sentTo của tin nhắn đầu ra chỉ định kênh mà phản hồi sẽ được gửi tới.

**Liệt kê 10.8 Hợp đồng mô tả cách thứcDịch vụ đặt hànggọi không đồng bộ**

**Dịch vụ nhà bếp**

hợp đồng trọn gói;

org.springframework.cloud.contract.spec.Contract.make {nhãn 'createTicket'

đầu vào {

tin nhắnTừ('kitchenService')

**Tin nhắn lệnh được gửi bởi Order Service đến kitchenService**

**kênh**

messageBody('''{"orderId":1,"restaurantId":1,"ticketDetails":{...}}''')Tiêu đề tin nhắn {

tiêu đề('command\_type','net.chrisrichardson...CreateTicket')tiêu đề('command\_saga\_type','net.chrisrichardson...CreateOrderSaga') tiêu đề('command\_saga\_id',$(người tiêu dùng(regex('[0-9a-f]{16}-[0-9a-f]

{16}'))))

header('command\_reply\_to','net.chrisrichardson...CreateOrderSaga-Reply')

} **Tin nhắn trả lời đã được gửi**

} **bởi Dịch vụ nhà bếp**

outputMessage { sentTo('net.chrisrichardson...CreateOrderSaga-reply') body([

Mã vé: 1

])

tiêu đề {

tiêu đề('reply\_type', 'net.chrisrichardson...CreateTicketReply') tiêu đề('reply\_outcome-type', 'THÀNH CÔNG')

}

}

}

Trong ví dụ hợp đồng này, thông báo đầu vào là lệnh CreateTicket được gửi đến kênh kitchenService. Thông báo đầu ra là phản hồi thành công được gửi đến kênh phản hồi của CreateOrderSaga. Hãy cùng xem cách sử dụng hợp đồng này trong các bài kiểm tra, bắt đầu-cùng với các thử nghiệm phía người tiêu dùng cho Dịch vụ đặt hàng.

**CNGƯỜI TIÊU DÙNG-KIỂM TRA TÍCH HỢP HỢP ĐỒNG PHỤ CHO YÊU CẦU KHÔNG ĐỒNG BỘ/TƯƠNG TÁC PHẢN HỒI**

Chiến lược viết bài kiểm tra tích hợp phía người dùng cho tương tác yêu cầu/phản hồi không đồng bộ tương tự như kiểm tra máy khách REST. Bài kiểm tra này gọi proxy nhắn tin của dịch vụ và xác minh hai khía cạnh về hành vi của nó. Đầu tiên, nó xác minh rằng proxy nhắn tin gửi tin nhắn lệnh tuân thủ hợp đồng. Thứ hai, nó xác minh rằng proxy xử lý đúng tin nhắn trả lời.

Danh sách 10.9 cho thấy phía người tiêu dùngkiểm tra tích hợp choDịch vụ nhà bếpProxy, đó là proxy nhắn tin được sử dụng bởiDịch vụ đặt hàngđể triệu hồiDịch vụ nhà bếp. Mỗikiểm tra gửi một tin nhắn lệnh bằng cách sử dụngDịch vụ nhà bếpProxyvà xác minh rằng nó trả về kết quả mong đợi. Nó sử dụng Spring Cloud Contract để cấu hình các stub nhắn tin cho

Dịch vụ nhà bếptìm hợp đồng có thông điệp đầu vào khớp với thông điệp lệnh và gửi thông điệp đầu ra dưới dạng phản hồi. Các bài kiểm tra sử dụng thông điệp trong bộ nhớ để đơn giản và nhanh hơn.

**Danh sách 10.9 Kiểm tra tích hợp hợp đồng phía người tiêu dùng choDịch vụ đặt hàng**

@RunWith(Lớp SpringRunner)@SpringBootTest(lớp=

KitchenServiceProxyIntegrationTest.TestConfiguration.class, webEnvironment= SpringBootTest.WebEnvironment.NONE)

@AutoConfigureStubRunner(id =

{"net.chrisrichardson.ftgo.contracts:ftgo-kitchen-service-contracts"},workOffline = sai)

@BẩnBốiCảnh

lớp công khai KitchenServiceProxyIntegrationTest {

@Cấu hình @EnableAutoConfiguration

@Import({TramCommandProducerConfiguration.class,Lớp TramInMemoryConfiguration,

EventuateContractVerifierConfiguration.class})lớp tĩnh công khai TestConfiguration { ... }

@Autowired

riêng tư SagaMessagingTestHelper sagaMessagingTestHelper;

@Autowired

riêng tư KitchenServiceProxy kitchenServiceProxy;

@Bài kiểm tra

public void shouldSuccessfullyCreateTicket() { Lệnh CreateTicket = new CreateTicket(AJANTA\_ID,

OrderDetailsMother.ORDER\_ID,

TicketDetails mới(Collections.singletonList(

TicketLineItem mới(CHICKEN\_VINDALOO\_MENU\_ITEM\_ID,

GÀ\_VINDALOO, SỐ LƯỢNG GÀ\_VINDALOO))))

Chuỗi sagaType = CreateOrderSaga.class.getName();

**Cấu hình dịch vụ nhà bếp stub để phản hồi tin nhắn.**

**Gửi lệnh và chờ phản hồi.**

CreateTicketReply trả lời = sagaMessagingTestHelper

**Xác minh**

**hồi đáp.**

ly);

.sendAndReceiveCommand(kitchenServiceProxy.create,yêu cầu,

CreateTicketReply.class, sagaType); assertEquals(new CreateTicketReply(OrderDetailsMother.ORDER\_ID), rep

}

}

CácshouldSuccessfullyCreateTicket()phương pháp thử nghiệm gửi mộtTạoTickettin nhắn lệnh và xác minh rằng phản hồi có chứa dữ liệu mong đợi. Nó sử dụngSaga-Trợ lý kiểm tra tin nhắn, là lớp trợ giúp kiểm tra có chức năng gửi và nhận tin nhắn đồng bộ.

Bây giờ chúng ta hãy xem cách viết các bài kiểm tra tích hợp phía nhà cung cấp.

**TNHÀ CUNG CẤP RITING-BÊN,NGƯỜI TIÊU DÙNG-KIỂM TRA HỢP ĐỒNG ĐƯỢC ĐIỀU KHIỂN CHO YÊU CẦU KHÔNG ĐỒNG BỘ/TƯƠNG TÁC PHẢN HỒI**

Kiểm tra tích hợp phía nhà cung cấp phải xác minh rằng nhà cung cấp xử lý thông báo lệnh bằng cách gửi phản hồi chính xác. Spring Cloud Contract tạo các lớp kiểm tra có phương pháp kiểm tra cho mỗi hợp đồng. Mỗi phương pháp kiểm tra gửi thông báo đầu vào của hợp đồng và xác minh rằng phản hồi khớp với thông báo đầu ra của hợp đồng.

Các bài kiểm tra tích hợp phía nhà cung cấp choDịch vụ nhà bếpBài kiểm traKitchenService-CommandHandler. CácKitchenServiceCommandHandlerlớp xử lý một thông điệp bằng cách gọiDịch vụ nhà bếp. Danh sách sau đây cho thấyTóm tắtKitchenService-ConsumerContractTestlớp, là lớp cơ sở cho các bài kiểm tra do Spring Cloud Contract tạo ra. Nó tạo ra mộtKitchenServiceCommandHandlertiêm một chất giảDịch vụ nhà bếp.

**Liệt kê 10.10 Siêu lớp của các bài kiểm tra hợp đồng do người dùng điều khiển, phía nhà cung cấp choPhòng bếpDịch vụ**

@RunWith(Lớp SpringRunner)@SpringBootTest(lớp =

Tóm tắtKitchenServiceConsumerContractTest.TestConfiguration.class,webEnvironment = SpringBootTest.WebEnvironment.NONE)

@AutoConfigureMessageVerifier

lớp trừu tượng công khai AbstractKitchenServiceConsumerContractTest {

@Configuration @Import(RestaurantMessageHandlersConfiguration.class) lớp tĩnh công khai TestConfiguration {

...

@Đậu

công khai KitchenService kitchenService() { trả về mock(KitchenService.class);

}

}

**Ghi đè định nghĩa của kitchenService @Bean bằng một bản nháp**

@Autowired

Dịch vụ bếp riêng Dịch vụ bếp riêng;

@Trước

public void setup() { reset(kitchenService); khi(kitchenService

.createTicket(eq(1L), eq(1L),

**Cấu hình bản mô phỏng để trả về các giá trị khớp với thông báo đầu ra của hợp đồng**

bất kỳ(TicketDetails.class)))

.thenReturn(vé mới(1L, 1L,

mới TicketDetails(Collections.emptyList())));

}

}

KitchenServiceCommandHandlergọiDịch vụ nhà bếpvới các đối số được lấy từ thông điệp đầu vào của hợp đồng và tạo ra thông điệp trả lời được lấy từtừ giá trị trả về. Lớp kiểm tracài đặt()phương pháp cấu hình mockBếp-Dịch vụđể trả về các giá trị phù hợp với thông điệp đầu ra của hợp đồng

Kiểm thử tích hợp và kiểm thử đơn vị xác minh hành vi của từng phần riêng lẻ của một dịch vụ. Kiểm thử tích hợp xác minh rằng các dịch vụ có thể giao tiếp với máy khách và các phụ thuộc của chúng. Kiểm thử đơn vị xác minh rằng logic của dịch vụ là đúng. Không loại kiểm thử nào chạy toàn bộ dịch vụ. Để xác minh rằng một dịch vụ như một tổng thể hoạt động, chúng ta sẽ di chuyển lên kim tự tháp và xem cách viết các kiểm thử thành phần.

#### Phát triển các bài kiểm tra thành phần

Cho đến nay, chúng ta đã xem xét cách kiểm tra từng lớp và nhóm lớp riêng lẻ. Nhưng hãy tưởng tượngine rằng bây giờ chúng ta muốn xác minh rằng Order Service hoạt động như mong đợi. Nói cách khác, chúng ta muốn viết các bài kiểm tra chấp nhận của dịch vụ, coi nó như một hộp đen và xác minh hành vi của nó thông qua API của nó. Một cách tiếp cận là viết những gì về cơ bản là các bài kiểm tra đầu cuối và triển khai Order Service cùng tất cả các phụ thuộc chuyển tiếp của nó. Như bạn đã biết, đó là một cách chậm, dễ hỏng và tốn kém để kiểm tra một dịch vụ.

**Mẫu: Kiểm tra thành phần dịch vụ**

Kiểm tra một dịch vụ riêng biệt. Xem[http://microservices.io/patterns/testing/service-](http://microservices.io/patterns/testing/service-component-test.html) [thành phần-test.html](http://microservices.io/patterns/testing/service-component-test.html).

Một cách tốt hơn nhiều để viết các bài kiểm tra chấp nhận cho một dịch vụ là sử dụng kiểm tra thành phần. Như hình 10.6 cho thấy, các bài kiểm tra thành phần được kẹp giữa các bài kiểm tra tích hợp và các bài kiểm tra đầu cuối. Kiểm tra thành phần xác minh hành vi của một dịch vụ một cách riêng biệt. Nó thay thế các phụ thuộc của một dịch vụ bằng các stub mô phỏng hành vi của chúng. Nó thậm chí có thể sử dụng các phiên bản trong bộ nhớ của các dịch vụ cơ sở hạ tầng như cơ sở dữ liệu. Do đó, các bài kiểm tra thành phần dễ viết hơn nhiều và chạy nhanh hơn.

Tôi bắt đầu bằng cách mô tả ngắn gọn cách sử dụng DSL thử nghiệm có tên là Gherkin để viếtcác bài kiểm tra chấp nhận cho các dịch vụ, chẳng hạn như Dịch vụ đặt hàng. Sau đó, tôi thảo luận về các vấn đề thiết kế thử nghiệm thành phần khác nhau. Sau đó, tôi trình bày cách viết các bài kiểm tra chấp nhận cho Dịch vụ đặt hàng.

Hãy cùng xem cách viết bài kiểm tra chấp nhận bằng Gherkin.

Kiểm tra

Từ đầu đến cuối

Thành phần

Tích hợp

Đơn vị

Dịch vụ

Kiểm tra thành phần

Phụ thuộc stub 2

Phụ thuộc stub 1

**Hình 10.6 Một thử nghiệm thành phần kiểm tra một dịch vụ một cách riêng biệt. Nó thường sử dụng các stub cho dịch vụsự phụ thuộc.**

Phụ thuộc stub

...

###### Xác định các bài kiểm tra chấp nhận

Kiểm thử chấp nhận là các kiểm thử hướng đến doanh nghiệp đối với một thành phần phần mềm. Chúng mô tả hành vi mong muốn có thể nhìn thấy bên ngoài theo quan điểm của khách hàng của thành phần đó thay vì theo quan điểm của việc triển khai nội bộ. Các kiểm thử này bắt nguồn từ người dùngcâu chuyện hoặc trường hợp sử dụng. Ví dụ, một trong những câu chuyện chính cho Dịch vụ đặt hàng là câu chuyện Đặt hàng:

Với tư cách là người tiêu dùng của Dịch vụ Đặt hàngTôi có thể đặt hàng

Chúng ta có thể mở rộng câu chuyện này thành các kịch bản nhưsau đây:

Cho một người tiêu dùng hợp lệ

Được cung cấp bằng thẻ tín dụng hợp lệ

Cho rằng nhà hàng đang chấp nhận đơn đặt hàng

Khi tôi đặt hàng Gà Vindaloo tại Ajanta thì đơn hàng đó phải được DUYỆT

Và sự kiện OrderAuthorized phải được công bố

Kịch bản này mô tả hành vi mong muốn của Order Service theo API của nó.

Mỗi kịch bản xác định một bài kiểm tra chấp nhận. Các giá trị cho trước tương ứng với giai đoạn thiết lập của bài kiểm tra, when ánh xạ đến giai đoạn thực hiện và then và and đến giai đoạn xác minh. Sau đó, bạn sẽ thấy một bài kiểm tra cho kịch bản này thực hiện các thao tác sau:

**1** Tạo ra mộtĐặt hàngbằng cách triệu hồiPOST / đơn hàngđiểm cuối

**2**Xác minh trạng thái củaĐặt hàngbằng cách triệu hồiNHẬN /đơn hàng/{orderId}điểm cuối

**3**Xác minh rằngDịch vụ đặt hàngđã xuất bản mộtĐặt hàngĐược ủy quyềnsự kiện bằng cách đăng ký kênh tin nhắn thích hợp

Chúng ta có thể dịch từng kịch bản thành mã Java. Tuy nhiên, một lựa chọn dễ hơn là viết các bài kiểm tra chấp nhận bằng DSL như Gherkin.

###### Viết bài kiểm tra chấp nhận bằng Gherkin

Viết các bài kiểm tra chấp nhận trong Java là một thách thức. Có nguy cơ là các kịch bản và các bài kiểm tra Java sẽ khác nhau. Cũng có sự ngắt kết nối giữa các kịch bản cấp cao và các bài kiểm tra Java, bao gồm các chi tiết triển khai cấp thấp. Ngoài ra, có nguy cơ là một kịch bản thiếu độ chính xác hoặc mơ hồ và không thể dịch thành mã Java. Một cách tiếp cận tốt hơn nhiều là loại bỏ bước dịch thủ công và viết các kịch bản có thể thực thi.

Gherkin là DSL để viết các thông số kỹ thuật thực thi. Khi sử dụng Gherkin, bạn định nghĩa các bài kiểm tra chấp nhận của mình bằng các kịch bản giống tiếng Anh, chẳng hạn như kịch bản được hiển thị trước đó. Sau đó, bạn thực hiện các thông số kỹ thuật bằng Cucumber, một khuôn khổ tự động hóa thử nghiệm cho Gherkin. Gherkin và Cucumber loại bỏ nhu cầu dịch thủ công các kịch bản thành mã có thể chạy được.

Đặc tả Gherkin cho một dịch vụ như Order Service bao gồm một tập hợp các tính năng. Mỗi tính năng được mô tả bằng một tập hợp các kịch bản như kịch bản bạn đã thấy trước đó. Một kịch bản có cấu trúc given-when-then. Các given là các điều kiện tiên quyết, when là hành động hoặc sự kiện xảy ra, và then/and là kết quả mong đợi.

Ví dụ, hành vi mong muốn củaDịch vụ đặt hàngđược xác định bởi một số tính năng, bao gồmĐặt hàng,Hủy đơn hàng, VàSửa đổi đơn hàng. Danh sách 10.11 là một đoạn trích củacáiĐặt hàngTính năng này bao gồm một số yếu tố:

* + - * *Tên*—Đối với tính năng này, tên làĐặt hàng.
      * *Tóm tắt thông số kỹ thuật*—Điều này mô tả lý do tại sao tính năng này tồn tại. Đối với tính năng này, tóm tắt thông số kỹ thuật là câu chuyện của người dùng.
      * *Các tình huống*—Đơn hàng đã được ủy quyềnVàĐơn hàng bị từ chối do thẻ tín dụng đã hết hạn.

**Liệt kê 10.11 Định nghĩa của GherkinĐặt hàngtính năng và một số kịch bản của nó**

Tính năng: Đặt hàng

Là người tiêu dùng của Dịch vụ Đặt hàng, tôi có thể đặt hàng

Kịch bản: Đơn hàng được chấp thuận Với một người tiêu dùng hợp lệ

Được cung cấp bằng thẻ tín dụng hợp lệ

Cho rằng nhà hàng đang chấp nhận đơn đặt hàng

Khi tôi đặt hàng Gà Vindaloo tại Ajanta thì đơn hàng đó phải được DUYỆT

Và sự kiện OrderAuthorized phải được công bố

Kịch bản: Đơn hàng bị từ chối do thẻ tín dụng đã hết hạn Với một người tiêu dùng hợp lệ

Được cung cấp khi sử dụng thẻ tín dụng đã hết hạn Được cung cấp khi nhà hàng đang chấp nhận đơn đặt hàng

Khi tôi đặt hàng Gà Vindaloo tại Ajanta thì đơn hàng đó phải bị TỪ CHỐI

Và sự kiện OrderRejected sẽ được công bố

...

Trong cả hai trường hợp, người tiêu dùng đều cố gắng đặt hàng. Trong trường hợp đầu tiên, họ thành công. Trong trường hợp thứ hai, đơn hàng bị từ chối vì thẻ tín dụng của người tiêu dùng đã hết hạn. Để biết thêm thông tin về Gherkin, hãy xem cuốn sách Writing Great Specifi-*cations: Sử dụng Specification by Example và Gherkin*của Kamil Nicieja (Manning, 2017).

**EXẾP HÀNHGTHÔNG SỐ KỸ THUẬT HERKIN SỬ DỤNGCDưa chuột**

Cucumber là một khuôn khổ kiểm thử tự động thực hiện các bài kiểm thử được viết bằng Gherkin. Nó có sẵn trong nhiều ngôn ngữ, bao gồm cả Java. Khi sử dụng Cucumber cho Java, bạn viết một lớp định nghĩa bước, chẳng hạn như lớp được hiển thị trong danh sách 10.12. Một lớp định nghĩa bước bao gồm các phương thức định nghĩa ý nghĩa của từng bước given-then-when. Mỗi phương thức định nghĩa bước được chú thích bằng @Given, @When, @Then hoặc @And. Mỗi chú thích này có một phần tử giá trị là một biểu thức chính quy, mà Cucumber khớp với các bước.

**Danh sách 10.12 Lớp định nghĩa bước Java giúp các kịch bản Gherkin có thể thực thi được.**

lớp công khai StepDefinitions ... {

...

@Given("Một người tiêu dùng hợp lệ")

công khai void useConsumer() { ... }

@Given("sử dụng thẻ tín dụng(.?) (.\*)")

public void useCreditCard(String bỏ qua, String creditCard) { ... }

@When("Tôi đặt hàng Gà Vindaloo tại Ajanta") public void placeOrder() { ... }

@Then("thứ tự phải là (.\*)")

công khai void theOrderShouldBe(String desiredOrderState) { ... }

@And("một sự kiện (.\*) nên được công bố")

công khai void verifyEventPublished(String expectedEventClass) { ... }

}

Mỗi loại phương pháp là một phần của giai đoạn thử nghiệm cụ thể:

* @Được cho—Giai đoạn thiết lập
* @Khi—Giai đoạn thực hiện
* @Sau đó*Và*@Và—Giai đoạn xác minh

Sau đó trong phần 10.2.4, khi tôi mô tả lớp này chi tiết hơn, bạn sẽ thấy rằng nhiềucủa những phương pháp này thực hiện các cuộc gọi REST tớiDịch vụ đặt hàng. Ví dụ,Đặt hàng()phương pháp tạo raĐặt hàngbằng cách triệu hồiPOST / đơn hàngĐiểm cuối REST.the- OrderShouldBe()phương pháp xác minh trạng thái của lệnh bằng cách gọiNHẬN /đơn hàng/

{mã đơn hàng}.

Nhưng trước khi đi sâu vào chi tiết về cách viết các lớp bước, chúng ta hãy cùng tìm hiểu một số vấn đề thiết kế với các bài kiểm tra thành phần.

###### Thiết kế các bài kiểm tra thành phần

Hãy tưởng tượng bạn đang thực hiệncác bài kiểm tra thành phần cho Dịch vụ Đặt hàng. Phần 10.2.2 cho thấy cách chỉ định hành vi mong muốn bằng Gherkin và thực hiện nó bằng Cucumber. Nhưng trước khi một bài kiểm tra thành phần có thể thực hiện các kịch bản Gherkin, trước tiên nó phải chạyOrder Service và thiết lập các phụ thuộc của dịch vụ. Bạn cần kiểm tra Order Service một cách riêng biệt, vì vậy, kiểm tra thành phần phải cấu hình stub cho một số dịch vụ, bao gồm Kitchen Service. Nó cũng cần thiết lập cơ sở dữ liệu và cơ sở hạ tầng nhắn tin. Có một số tùy chọn khác nhau đánh đổi tính hiện thực với tốc độ và sự đơn giản.

**TÔIN-KIỂM TRA THÀNH PHẦN QUÁ TRÌNH**

Một lựa chọn là viết các bài kiểm tra thành phần đang xử lý. Một bài kiểm tra thành phần đang xử lý chạy dịch vụ với các stub và mock trong bộ nhớ cho các phụ thuộc của nó. Ví dụ: bạn có thể viết một bài kiểm tra thành phần cho dịch vụ dựa trên Spring Boot bằng cách sử dụng khung kiểm tra Spring Boot. Một lớp kiểm tra, được chú thích bằng @SpringBootTest, chạy dịch vụ trong cùng một JVM với bài kiểm tra. Nó sử dụng dependency injection để cấu hình dịch vụ để sử dụng mock và stub. Ví dụ: một bài kiểm tra cho Order Service sẽ cấu hình nó để sử dụng cơ sở dữ liệu JDBC trong bộ nhớ, chẳng hạn như H2, HSQLDB hoặc Derby và stub trong bộ nhớ cho Eventuate Tram. Các bài kiểm tra đang xử lý đơn giản hơn để viết và nhanh hơn, nhưng có nhược điểm là không kiểm tra được dịch vụ có thể triển khai.

**ỒĐại học-CỦA-KIỂM TRA THÀNH PHẦN QUÁ TRÌNH**

Một cách tiếp cận thực tế hơn là đóng gói dịch vụ theo định dạng sẵn sàng sản xuất và chạy nó như một quy trình riêng biệt. Ví dụ, chương 12 giải thích rằng việc đóng gói dịch vụ dưới dạng hình ảnh vùng chứa Docker ngày càng phổ biến. Một thử nghiệm thành phần ngoài quy trình sử dụng các dịch vụ cơ sở hạ tầng thực, chẳng hạn như cơ sở dữ liệu và trình môi giới tin nhắn, nhưng sử dụng các stub cho bất kỳ sự phụ thuộc nào là dịch vụ ứng dụng. Ví dụ, một thử nghiệm ngoài quy trìnhkiểm tra thành phần cho FTGO Order Service sẽ sử dụng MySQL và Apache Kafka, và stub cho các dịch vụ bao gồm Consumer Service và Accounting Service. Vì Order Service tương tác với các dịch vụ đó bằng cách sử dụng tin nhắn, các stub này sẽ sử dụng tin nhắn từ Apache Kafka và gửi lại tin nhắn trả lời.

Một lợi ích chính của việc kiểm thử thành phần ngoài quy trình là nó cải thiện phạm vi kiểm thử, vì những gì đang được kiểm thử gần hơn nhiều với những gì đang được triển khai. Nhược điểm là loại kiểm thử này phức tạp hơn khi viết, chậm hơn khi thực hiện và có khả năng dễ hỏng hơn so với kiểm thử thành phần trong quy trình. Bạn cũng phải tìm ra cách stub các dịch vụ ứng dụng. Hãy cùng xem cách thực hiện điều đó.

**HCÁCH THỨC ĐỂ GỬI DỊCH VỤ VÀO RA-CỦA-KIỂM TRA THÀNH PHẦN QUÁ TRÌNH**

Dịch vụ đang được thử nghiệm thường gọi các phụ thuộc bằng cách sử dụng các kiểu tương tác liên quan đến việc gửi lại phản hồi. Ví dụ, Order Service sử dụng yêu cầu/phản hồi và gửi tin nhắn lệnh đến nhiều dịch vụ khác nhau. API Gateway sử dụng HTTP, là kiểu tương tác yêu cầu/phản hồi. Một bài kiểm tra ngoài quy trình phải cấu hình stub cho các loại phụ thuộc này, xử lý các yêu cầu và gửi lại phản hồi. Một tùy chọn là sử dụng Spring Cloud Contract, mà chúng ta đã xem xét trước đó trong phần 10.1 khi thảo luận về các bài kiểm tra tích hợp. Chúng ta có thể viết các hợp đồng cấu hình

stubs cho các bài kiểm tra thành phần. Tuy nhiên, một điều cần cân nhắc là có khả năng các hợp đồng này, không giống như các hợp đồng được sử dụng để tích hợp, chỉ được sử dụng bởi các bài kiểm tra thành phần. Một nhược điểm khác khi sử dụng Spring Cloud Contract để kiểm tra thành phần là vì trọng tâm của nó là kiểm tra hợp đồng của người tiêu dùng, nên nó có cách tiếp cận hơi nặng nề. Các tệp JAR chứa các hợp đồng phải được triển khai trong kho lưu trữ Maven thay vì chỉ nằm trên classpath. Việc xử lý các tương tác liên quan đến các giá trị được tạo động cũng rất khó khăn. Do đó, một tùy chọn đơn giản hơn là

cấu hình stubstừ bên trong bài kiểm tra.

Ví dụ, một bài kiểm tra có thể cấu hình một stub HTTP bằng cách sử dụng WireMock stubbing DSL. Tương tự như vậy, một bài kiểm tra cho một dịch vụ sử dụng nhắn tin Eventuate Tram có thể cấu hình các stub nhắn tin. Sau này trong phần này, tôi sẽ trình bày một thư viện Java dễ sử dụng thực hiện việc này.

Bây giờ chúng ta đã xem xét cách thiết kế các bài kiểm tra thành phần, hãy cùng xem xét cáchviết các bài kiểm tra thành phần cho Dịch vụ đặt hàng FTGO.

###### Viết các bài kiểm tra thành phần cho FTGO Order Service

Như bạn đã thấy trước đó trong phần này, có một số cách khác nhau để triển khai các bài kiểm tra thành phần. Phần này mô tả các bài kiểm tra thành phần cho Order Service sử dụng chiến lược out-of-process để kiểm tra dịch vụ đang chạy dưới dạng vùng chứa Docker. Bạn sẽ thấy cách các bài kiểm tra sử dụng plugin Gradle để khởi động và dừng vùng chứa Docker. Tôi thảo luận về cách sử dụng Cucumber để thực hiện các kịch bản dựa trên Gherkin xác định hành vi mong muốn cho Order Service.

Hình 10.7 cho thấy thiết kế của các bài kiểm tra thành phần choDịch vụ đặt hàng.Đặt hàng-

Kiểm tra thành phần dịch vụlà lớp kiểm tra đóchạy Dưa chuột:

@RunWith(Lớp Dưa Chuột)

@CucumberOptions(tính năng= "src/component-test/resources/features") lớp công khai OrderServiceComponentTest {

}

Nó có chú thích @CucumberOptions chỉ định nơi tìm Gherkincác tệp tính năng. Nó cũng được chú thích bằng @RunWith(Cucumber.class), cho biết JUNIT sử dụng trình chạy thử nghiệm Cucumber. Nhưng không giống như lớp thử nghiệm dựa trên JUNIT thông thường, nó không có bất kỳ phương thức thử nghiệm nào. Thay vào đó, nó định nghĩa các thử nghiệm bằng cách đọc các tính năng Gherkin và sử dụng lớp OrderServiceComponentTestStepDefinitions để thực thi chúng.

Sử dụng Cucumber với khung thử nghiệm Spring Boot yêu cầu một chút khác thườngstructure. Mặc dù không phải là lớp kiểm thử, OrderServiceComponentTestStepDefinitions vẫn được chú thích bằng @ContextConfiguration, là một phần của khung Spring Testing. Nó tạo ra Spring ApplicationContext, định nghĩa các thành phần Spring khác nhau, bao gồm cả các đoạn mã thông báo. Hãy xem chi tiết về các định nghĩa bước.

src/kiểm tra thành phần/tài nguyên/createorder.tính năng

Là người tiêu dùng của Dịch vụ Đặt hàng, tôi có thể tạo một đơn hàng

**Kịch bản:**Đơn hàng đã được ủy quyền

**Được cho**Mộtngười tiêu dùng hợp lệ

**Được cho**sử dụng một**có hiệu lực**thẻ tín dụng

Chạy

Chạy

Sử dụng

Docker-compose.yml

**dịch vụ đặt hàng ftgo:xây dựng: .**

**cổng:**

**-"8082:8080"**

Chạy

xây dựng.gradle

dockerCompose {

...

}

Kafka

MySQL

Dịch vụ đặt hàngthùng chứa docker

Gọi REST API

|  |  |  |  |
| --- | --- | --- | --- |
|  | Đọc | | |
| Dịch vụ đặt hàngKiểm tra thành phần | | Sử dụng | Dịch vụ đặt hàngBước thành phần  Định nghĩa |
|  |

Đọc lệnhvà gửi trả lời

Đọc sự kiện

Được viết bằng khuôn khổ thử nghiệm Cucumber

**Hình 10.7 Các bài kiểm tra thành phần choDịch vụ đặt hàngsử dụng khuôn khổ thử nghiệm Cucumber để thực hiện các kịch bản thử nghiệm được viết bằng Gherkin thử nghiệm chấp nhận DSL. Các thử nghiệm sử dụng Docker để chạyDịch vụ đặt hàngcùng với các dịch vụ cơ sở hạ tầng như Apache Kafka và MySQL.**

**TANH TAỒĐẠI HỌCSDỊCH VỤCPHẦN MỞ RỘNGTESTSTEPDLỚP ĐỊNH NGHĨA**

CácOrderServiceComponentTestStepDefinitionslớp là trái tim của các bài kiểm tra. Lớp này xác định ý nghĩa của từng bước trongDịch vụ đặt hàngcác bài kiểm tra thành phần của. Các danh sách lowing cho thấysử dụngCreditCard()phương pháp, xác định ý nghĩa củaĐược cung cấp bằng cách sử dụng … thẻ tín dụngbước chân.

**Liệt kê 10.13 Các@GivenuseCreditCard()phương pháp xác định ý nghĩa của**

**Được cung cấp bằng cách sử dụng … thẻ tín dụngbước chân.**

@ContextConfiguration(các lớp = OrderServiceComponentTestStepDefinitions.TestConfiguration.class)

lớp công khai OrderServiceComponentTestStepDefinitions {

...

@Autowired

được bảo vệSagaParticipantStubManager sagaParticipantStubManager;

@Given("sử dụng thẻ tín dụng(.?) (.\*)")

public void useCreditCard(String bỏ qua, String creditCard) { nếu (creditCard.equals("hợp lệ"))

sagaNgười tham giaStubQuản lý

.forChannel("Dịch vụ kế toán")

.when(AuthorizeCommand.class).replyWithSuccess();else if (creditCard.equals("không hợp lệ"))

sagaNgười tham giaStubQuản lý

**Gửi phản hồi thành công.**

**Gửi phản hồi lỗi.**

khác

.forChannel("Dịch vụ kế toán")

.when(AuthorizeCommand.class).replyWithFailure();

fail("Không biết phải làm gì với thẻ tín dụng này");

}

Phương pháp này sử dụngSagaNgười tham giaStubQuản lýlớp, một lớp trợ giúp kiểm tra tạo ra các bản nháp cho những người tham gia saga.sử dụngCreditCard()phương pháp sử dụng nó để cấu hìnhDịch vụ kế toántrả lời bằng thông báo thành công hoặc thất bại, tùy thuộc vào thẻ tín dụng được chỉ định.

Danh sách sau đây cho thấyĐặt hàng()phương pháp, xác địnhKhi tôi đặt hàng Gà Vindaloo tại Ajantabước. Nó gọiDịch vụ đặt hàngREST API để tạoĐặt hàngvà lưu phản hồi để xác thực ở bước sau.

**Liệt kê 10.14Đặt hàng()phương pháp xác địnhKhi tôi đặt hàng Gà Vindaloo tại Ajantabước chân.**

@ContextConfiguration(các lớp = OrderServiceComponentTestStepDefinitions.TestConfiguration.class)

lớp công khai OrderServiceComponentTestStepDefinitions {

cổng int riêng tư = 8082;

Chuỗi riêng tư máy chủ = System.getenv("DOCKER\_HOST\_IP");

được bảo vệ String baseUrl(Đường dẫn chuỗi) {

trả về String.format("http://%s:%s%s", máy chủ, cổng, đường dẫn);

}

Phản hồi riêng tư phản hồi;

@When("Tôi đặt hàng Gà Vindaloo tại Ajanta") public void placeOrder() {

**Gọi API REST của dịch vụ đặt hàng**

phản hồi = đã cho().

thân(mớiCreateOrderRequest(consumerId,

**để tạo ra đơn hàng**

Nhà hàngMother.AJANTA\_ID,Collections.singletonList(mới CreateOrderRequest.LineItem(

Nhà hàngMẹ.GÀ\_VINDALOO\_MENU\_ITEM\_ID, Chi tiết đơn hàngMẹ.GÀ\_VINDALOO\_QUANTITY)))).

contentType("ứng dụng/json").khi().post(baseUrl("/orders"));

}

Phương thức trợ giúp baseUrl() trả về URL của dịch vụ đặt hàng.

Danh sách 10.15 cho thấytheOrderShouldBe()phương pháp, xác định ý nghĩa củaKhi đó thứ tự sẽ là …bước. Nó xác minh rằngĐặt hàngđã được tạo thành công và ở trạng thái mong đợi.

**Liệt kê 10.15@ThentheOrderShouldBe()phương pháp xác minh yêu cầu HTTP đã đượcthành công.**

@ContextConfiguration(các lớp = OrderServiceComponentTestStepDefinitions.TestConfiguration.class)

lớp công khai OrderServiceComponentTestStepDefinitions {

@Then("thứ tự phải là (.\*)")

công khai void theOrderShouldBe(Chuỗi desiredOrderState) {

Số nguyên orderId =

this.response.then().statusCode(200).extract().path("orderId");

khẳng địnhKhôngNull(Id đơn hàng);cuối cùng(() -> {

Trạng thái chuỗi = given().

khi().

lấy(baseUrl("/đơn hàng/" + orderId)).sau đó().

Mã trạng thái(200)

.trích xuất().

đường dẫn("trạng thái"); khẳng địnhEquals(trạng thái mong muốn, trạng thái);

});

**Xác minh rằng Đơn hàng đã được tạo thành công.**

**Xác minh trạng thái củaĐặt hàng.**

}

]

Khẳng định về trạng thái mong đợi được gói gọn trong lệnh gọi đến finally(), thực thi khẳng định này nhiều lần.

Danh sách sau đây cho thấyxác minhEventPublished()phương pháp, xác địnhVà một sự kiện … nên được công bốBước này xác minh rằng sự kiện miền mong đợi đã được công bố.

**Liệt kê 10.16 Lớp định nghĩa bước Cucumber choDịch vụ đặt hàngthành phầnkiểm tra**

@ContextConfiguration(các lớp = OrderServiceComponentTestStepDefinitions.TestConfiguration.class)

lớp công khai OrderServiceComponentTestStepDefinitions {

@Autowired

được bảo vệ MessageTracker messageTracker;

@And("một sự kiện (.\*) nên được công bố")

public void verifyEventPublished(String expectedEventClass) ném ClassNot FoundException {

messageTracker.assertDomainEventPublished("net.chrisrichardson.ftgo.orderdịch vụ.tên miền.Đặt hàng",

(Lớp<DomainEvent>)Lớp.forName("net.chrisrichardson.ftgo.orderdịch vụ.tên miền." + expectedEventClass));

}

....

}

Cácxác minhEventPublished()phương pháp sử dụngTrình theo dõi tin nhắnlớp, một lớp trợ giúp kiểm tra ghi lại các sự kiện đã được công bố trong quá trình kiểm tra. Lớp nàyVàSagaNgười tham giaStubQuản lýđược thực hiện bởiKiểm tra cấu hình @Cấu hìnhlớp học.

Bây giờ chúng ta đã xem xét các định nghĩa bước, hãy cùng xem cách chạy các thử nghiệm thành phần.

**RTHỬ NGHIỆM THÀNH PHẦN**

Bởi vì các thử nghiệm này tương đối chậm, chúng tôi không muốn chạy chúng như một phần của./gradlewBài kiểm tra. Thay vào đó, chúng tôi sẽ đặt mã kiểm tra vào mộtsrc/thành phần-kiểm tra/javathư mụcvà chạy chúng bằng cách sử dụng./gradlew thành phần kiểm tra.Hãy nhìn vàoftgo-order-service/xây dựng.gradletệp để xem cấu hình Gradle.

Các bài kiểm tra sử dụng Docker để chạy Order Service và các phụ thuộc của nó. Như đã mô tả trong chương 12, một container Docker là một cơ chế ảo hóa hệ điều hành nhẹ cho phép bạn triển khai một phiên bản dịch vụ trong một hộp cát bị cô lập. Docker Compose là một công cụ cực kỳ hữu ích mà bạn có thể sử dụng để định nghĩa một tập hợp các container và khởi động và dừng chúng như một đơn vị. Ứng dụng FTGO có một tệp docker-compose trong thư mục gốc để định nghĩa các container cho tất cả các dịch vụ và dịch vụ cơ sở hạ tầng.

Chúng ta có thể sử dụng plugin Gradle Docker Compose để chạy các container trước khi thực hiện các thử nghiệm và dừng các container sau khi các thử nghiệm hoàn tất:

áp dụng plugin: 'docker-compose'

dockerCompose.isRequiredBy(thành phần Test)componentTest.dependsOn(lắp ráp)

dockerCompose {

startedServices = [ 'ftgo-order-service']

}

Đoạn mã trước của cấu hình Gradle thực hiện hai việc. Đầu tiên, nó cấu hình plugin Gradle Docker Compose để chạy trước khi kiểm tra thành phần và khởi động Order Service cùng với các dịch vụ cơ sở hạ tầng mà nó được cấu hình để phụ thuộcon. Thứ hai, nó cấu hình componentTest để phụ thuộc vào assemble để file JAR mà Docker image yêu cầu được build trước. Với điều đó, chúng ta có thể chạy các bài kiểm tra thành phần này bằng các lệnh sau:

./gradlew :ftgo-order-service:componentTest

***Viết từ đầu đến cuốikiểm tra* 345**

Những lệnh này mất vài phút để thực hiện và thực hiện các hành động sau:

**1** Xây dựngDịch vụ đặt hàng.

**2**Chạy dịch vụ và các dịch vụ cơ sở hạ tầng của nó.

**3** Chạy thử nghiệm.

**4** Dừng các dịch vụ đang chạy.

Bây giờ chúng ta đã xem xét cách kiểm tra một dịch vụ riêng lẻ, chúng ta sẽ xem cách kiểm tra toàn bộ ứng dụng.

#### Viết các bài kiểm tra đầu cuối

Kiểm thử thành phần kiểm thử từng dịch vụ riêng biệt. Tuy nhiên, kiểm thử đầu cuối kiểm thử toàn bộ ứng dụng. Như hình 10.8 cho thấy, kiểm thử đầu cuối là đỉnh của kim tự tháp kiểm thử. Đó là vì những loại kiểm thử này—hãy nói theo tôi ngay—chậm, dễ vỡ và tốn thời gian để phát triển.

Kiểm tra

Từ đầu đến cuối

Thành phần

Tích hợp

Đơn vị

Dịch vụ

Kiểm tra đầu cuối

Dịch vụ 2

Dịch vụ 1

**Hình 10.8 Các bài kiểm tra đầu cuối nằm ở đầu kim tự tháp kiểm tra. Chúng chậm, giòn và mất thời giantiêu thụ để phát triển. Bạn nên giảm thiểu số lượng các bài kiểm tra đầu cuối.**

Dịch vụ ...

Các bài kiểm tra đầu cuối có một số lượng lớn các bộ phận chuyển động. Bạn phải triển khai nhiều dịch vụ và các dịch vụ cơ sở hạ tầng hỗ trợ của chúng. Do đó, các bài kiểm tra đầu cuối chậm. Ngoài ra, nếu bài kiểm tra của bạn cần triển khai một số lượng lớn các dịch vụ, rất có thể một trong số chúng sẽ không triển khai được, khiến các bài kiểm tra không đáng tin cậy. Do đó, bạn nên giảm thiểu số lượng các bài kiểm tra đầu cuối.

###### Thiết kế các bài kiểm tra đầu cuối

Như tôi đã giải thích, tốt nhất là viết càng ít các bài kiểm tra này càng tốt. Một chiến lược tốt là viết các bài kiểm tra hành trình của người dùng. Bài kiểm tra hành trình của người dùng tương ứng với hành trình của người dùng qua hệ thống. Ví dụ, thay vì kiểm tra tạo đơn hàng, sửa đơn hàng và hủy đơn hàng riêng biệt, bạn có thể viết một bài kiểm tra duy nhất thực hiện cả ba. Cách tiếp cận này làm giảm đáng kể số lượng bài kiểm tra bạn phải viết và rút ngắn thời gian thực hiện bài kiểm tra.

###### Viết các bài kiểm tra đầu cuối

Các bài kiểm tra đầu cuối, giống như các bài kiểm tra chấp nhận được đề cập trong phần 10.2, là các bài kiểm tra hướng đến doanh nghiệp. Việc viết chúng trong DSL cấp cao mà những người trong doanh nghiệp hiểu được là hợp lý. Ví dụ, bạn có thể viết các bài kiểm tra đầu cuối bằng Gherkin và thực hiện chúng bằng Cucumber. Danh sách sau đây cho thấy một ví dụ về một bài kiểm tra như vậy. Nó tương tự như các bài kiểm tra chấp nhận mà chúng ta đã xem trước đó. Sự khác biệt chính là thay vì một Then duy nhất, bài kiểm tra này có nhiều hành động.

**Danh sách 10.17 Một đặc tả dựa trên Gherkin về hành trình của người dùng**

Tính năng: Đặt Sửa đổi và Hủy bỏ

Với tư cách là người tiêu dùng của Dịch vụ Đặt hàng

Tôi có thể đặt hàng, sửa đổi và hủy đơn hàng

Kịch bản: Đơn hàng được tạo, sửa đổi và hủy Với một người tiêu dùng hợp lệ

Được cung cấp bằng thẻ tín dụng hợp lệ

Cho rằng nhà hàng đang chấp nhận đơn đặt hàng

Khi tôi đặt hàng Gà Vindaloo tại Ajanta thì đơn hàng đó phải được DUYỆT

Khi đó tổng đơn hàng phải là 16,33

Và khi tôi sửa đổi thứ tự bằng cách thêm 2 chiếc samosa rau

**Tạo đơn hàng.**

**Sửa đổi đơn hàng.**

Khi đó tổng đơn hàng phải là 20,97 Và khi tôi hủy đơn hàng

Sau đó đơn hàng phải được HỦY

**Hủy đơn hàng.**

Kịch bản này đặt lệnh, sửa đổi rồi hủy lệnh. Hãy cùng xem cách chạy lệnh này.

###### Chạy thử nghiệm đầu cuối

Các bài kiểm tra đầu cuối phải chạy toàn bộ ứng dụng, bao gồm mọi dịch vụ cơ sở hạ tầng bắt buộc. Như bạn đã thấy ở phần trước trong phần 10.2, plugin Gradle Docker Compose cung cấp một cách thuận tiện để thực hiện việc này. Tuy nhiên, thay vì chạy một dịch vụ ứng dụng duy nhất, tệp Docker Compose sẽ chạy tất cả các dịch vụ của ứng dụng.

Bây giờ chúng ta đã xem xét các khía cạnh khác nhau của việc thiết kế và viết các bài kiểm tra đầu cuối, hãy cùng xem một ví dụ về bài kiểm tra đầu cuối.

Mô-đun ftgo-end-to-end-test triển khai các bài kiểm tra đầu cuối cho ứng dụng FTGO. Việc triển khai bài kiểm tra đầu cuối khá giống với việc triển khai các bài kiểm tra thành phần đã thảo luận trước đó trong phần 10.2. Các bài kiểm tra này được viết bằng Gherkin và được thực hiện bằng Cucumber. Plugin Gradle Docker Compose chạy các container trước khi các bài kiểm tra chạy. Phải mất khoảng bốn đến năm phút để khởi động các container và chạy các bài kiểm tra.

Có vẻ như không phải là một thời gian dài, nhưng đây là một ứng dụng tương đối đơn giản chỉ với một số ít container và thử nghiệm. Hãy tưởng tượng nếu có hàng trăm container và nhiều thử nghiệm hơn nữa. Các thử nghiệm có thể mất khá nhiều thời gian. Do đó, tốt nhất là tập trung vào việc viết các thử nghiệm ở dưới cùng của kim tự tháp.

***Bản tóm tắt* 347**

#### Bản tóm tắt

* Sử dụng hợp đồng, là các thông điệp ví dụ, để thúc đẩy việc kiểm tra tương tác giữa các dịch vụ. Thay vì viết các bài kiểm tra chạy chậm chạy cả hai dịch vụ và các phụ thuộc chuyển tiếp của chúng, hãy viết các bài kiểm tra xác minh rằng bộ điều hợp của cả hai dịch vụ đều tuân thủ hợp đồng.
* Viết các bài kiểm tra thành phần để xác minh hành vi của dịch vụ thông qua API của nó. Bạn nên đơn giản hóa và tăng tốc các bài kiểm tra thành phần bằng cách kiểm tra dịch vụ một cách riêng biệt, sử dụng stub cho các phụ thuộc của nó.
* Viết các bài kiểm tra hành trình người dùng để giảm thiểu số lượng các bài kiểm tra đầu cuối, vốn chậm, dễ vỡ và tốn thời gian. Bài kiểm tra hành trình người dùng mô phỏng hành trình của người dùng thông qua ứng dụng và xác minh hành vi cấp cao của một phần tương đối lớn chức năng của ứng dụng. Vì có ít bài kiểm tra, nên lượng chi phí chung cho mỗi bài kiểm tra, chẳng hạn như thiết lập bài kiểm tra, được giảm thiểu, giúp tăng tốc các bài kiểm tra.

*Phát triển các dịch vụ sẵn sàng sản xuất*

***Chương này bao gồm:***

* Phát triển các dịch vụ an toàn
* Áp dụngMẫu cấu hình bên ngoài
* Áp dụng các mô hình quan sát:
  + Kiểm tra sức khỏeGiao diện lập trình ứng dụng (API)
  + Tổng hợp nhật ký
  + Phân phối theo dõi
  + Theo dõi ngoại lệ
  + Số liệu ứng dụng
  + Nhật ký kiểm toán
* Đơn giản hóa việc phát triển các dịch vụ bằng cácháp dụngMẫu khung gầm dịch vụ vi mô

Mary và nhóm của cô ấy cảm thấy rằng họ đã thành thạo việc phân tích dịch vụ, giao tiếp giữa các dịch vụ, quản lý giao dịch, truy vấn và thiết kế logic kinh doanh, và thử nghiệm. Họ tự tin rằng họ có thể phát triển các dịch vụ đáp ứng các yêu cầu chức năng của họ. Nhưng để một dịch vụ sẵn sàng triển khai vào sản xuất, họ cần đảm bảo rằng nó cũng sẽ đáp ứng ba thuộc tính chất lượng cực kỳ quan trọng: bảo mật, khả năng cấu hình và khả năng quan sát.

**348**

Thuộc tính chất lượng đầu tiên là bảo mật ứng dụng. Điều cần thiết là phải phát triển các ứng dụng bảo mật, trừ khi bạn muốn công ty của mình trở thành tiêu đề cho một vụ vi phạm dữ liệu. May mắn thay, hầu hết các khía cạnh bảo mật trong kiến ​​trúc vi dịch vụ không khác gì so với ứng dụng đơn khối. Nhóm FTGO biết rằng phần lớn những gì họ đã học được trong nhiều năm phát triển khối đơn khối cũng áp dụng cho các dịch vụ vi mô. Nhưng kiến ​​trúc vi dịch vụ buộc bạn phải triển khai một số khía cạnh của bảo mật cấp ứng dụng theo cách khác. Ví dụ: bạn cần triển khai một cơ chế để truyền danh tính của người dùng từ dịch vụ này sang dịch vụ khác.

Thuộc tính chất lượng thứ hai bạn phải giải quyết là khả năng cấu hình dịch vụ.Một dịch vụ thường sử dụng một hoặc nhiều dịch vụ bên ngoài, chẳng hạn như message broker và cơ sở dữ liệu. Vị trí mạng và thông tin xác thực của mỗi dịch vụ bên ngoài thường phụ thuộc vào môi trường mà dịch vụ đang chạy. Bạn không thể kết nối cứng các thuộc tính cấu hình vào dịch vụ. Thay vào đó, bạn phải sử dụng cơ chế cấu hình bên ngoài cung cấp cho dịch vụ các thuộc tính cấu hình khi chạy.

Thuộc tính chất lượng thứ ba là khả năng quan sát. Nhóm FTGO đã triển khai giám sát và ghi nhật ký cho ứng dụng hiện tại. Nhưng kiến ​​trúc vi dịch vụ là một hệ thống phân tán và điều đó đặt ra một số thách thức bổ sung. Mọi yêu cầu đều được xử lý bởi cổng API và ít nhất một dịch vụ. Ví dụ, hãy tưởng tượng rằng bạn đang cố gắng xác định dịch vụ nào trong sáu dịch vụ đang gây ra sự cố độ trễ. Hoặc hãy tưởng tượng rằng bạn đang cố gắng hiểu cách xử lý yêu cầu khi các mục nhật ký nằm rải rác trên năm dịch vụ khác nhau. Để dễ hiểu hơn về hành vi của ứng dụng và khắc phục sự cố, bạn phải triển khai một số mẫu khả năng quan sát.

Tôi bắt đầu chương này bằng cách mô tả cách triển khai bảo mật trong kiến ​​trúc microservice. Tiếp theo, tôi thảo luận về cách thiết kế các dịch vụ có thể cấu hình được. Tôi đề cập đến một vài cơ chế cấu hình dịch vụ khác nhau. Sau đó, tôi nói về cách làm cho các dịch vụ của bạn dễ hiểu hơn và khắc phục sự cố bằng cách sử dụng các mẫu quan sát. Tôi kết thúc chương này bằng cách chỉ ra cách đơn giản hóa việc triển khai những mối quan tâm này và các mối quan tâm khác bằng cách phát triển các dịch vụ của bạn trên một khuôn khổ khung microservice.

Trước tiên chúng ta hãy xem xét về bảo mật.

#### Phát triển các dịch vụ an toàn

An ninh mạng đã trở thành vấn đề quan trọng đối với mọi tổ chức. Hầu như ngày nào cũng có những tiêu đề về cách tin tặc đánh cắp dữ liệu của công ty. Để phát triển phần mềm an toàn và tránh xa các tiêu đề, một tổ chức cần giải quyết nhiều vấn đề bảo mật khác nhau, bao gồm bảo mật vật lý của phần cứng, mã hóa dữ liệu khi truyền và khi lưu trữ, xác thực và ủy quyền, cũng như chính sách vá lỗ hổng phần mềm. Hầu hết các vấn đề này đều giống nhau bất kể bạn đang sử dụng kiến ​​trúc đơn khối hay kiến ​​trúc vi dịch vụ. Phần này tập trung vào cách kiến ​​trúc vi dịch vụ tác động đến bảo mật ở cấp ứng dụng.

Nhà phát triển ứng dụng chủ yếu chịu trách nhiệm triển khai bốn khía cạnh khác nhau của bảo mật:

* + - *Xác thực*—Xác minh danh tính của ứng dụng hoặc con người (hay còn gọi là người đứng đầu) đang cố gắng truy cập ứng dụng. Ví dụ: ứng dụng thường xác minh thông tin xác thực của người đứng đầu, chẳng hạn như ID người dùng và mật khẩu hoặc khóa API và bí mật của ứng dụng.
    - *Ủy quyền*—Xác minh rằng người đứng đầu được phép thực hiện thao tác được yêu cầu trên dữ liệu đã chỉ định. Các ứng dụng thường sử dụng kết hợp bảo mật dựa trên vai trò và danh sách kiểm soát truy cập (ACL). Bảo mật dựa trên vai trò chỉ định cho mỗi người dùng một hoặc nhiều vai trò cấp cho họ quyền gọi các thao tác cụ thể. ACL cấp cho người dùng hoặc vai trò quyền thực hiện thao tác trên một đối tượng kinh doanh cụ thể hoặc tổng hợp.
    - *Kiểm toán*—Theo dõi các hoạt động mà người đứng đầu thực hiện để phát hiện các vấn đề bảo mật, hỗ trợ khách hàng và thực thi việc tuân thủ.
    - *Giao tiếp giữa các tiến trình an toàn*—Lý tưởng nhất là,tất cả giao tiếp vào và ra khỏi dịch vụ phải qua Giao thức bảo mật lớp truyền tải (TLS). Giao tiếp giữa các dịch vụ thậm chí có thể cần sử dụng xác thực.

Tôi mô tả chi tiết về việc kiểm toán trong phần 11.3 và đề cập đến việc bảo mật thông tin liên lạc giữa các dịch vụ khi thảo luận về các mạng lưới dịch vụ trong phần 11.4.1. Phần này tập trung vào việc triển khai xác thực và ủy quyền.

Tôi bắt đầu bằng cách mô tả cách bảo mật được triển khai trong ứng dụng FTGO monolith. Sau đó, tôi mô tả những thách thức khi triển khai bảo mật trong kiến ​​trúc microservice và cách các kỹ thuật hoạt động tốt trong kiến ​​trúc monolithic không thể được sử dụng trong kiến ​​trúc microservice. Sau đó, tôi sẽ trình bày cách triển khai bảo mật trong kiến ​​trúc microservice.

Chúng ta hãy bắt đầu bằng cách xem xét cách ứng dụng FTGO độc lập xử lý vấn đề bảo mật.

###### Tổng quan về bảo mật trong ứng dụng đơn khối truyền thống

Ứng dụng FTGO có nhiều loại người dùng, bao gồm người tiêu dùng, người giao hàng và nhân viên nhà hàng. Họ truy cập ứng dụng bằng các ứng dụng web dựa trên trình duyệt và ứng dụng di động. Tất cả người dùng FTGO phải đăng nhập để truy cập ứng dụng. Hình 11.1 cho thấy cách các máy khách của ứng dụng FTGO nguyên khối xác thực và đưa ra yêu cầu.

Khi người dùng đăng nhập bằng ID người dùng và mật khẩu của họ, máy khách sẽ thực hiện yêu cầu POST chứa thông tin xác thực của người dùng tới ứng dụng FTGO. Ứng dụng FTGO sẽ xác minh thông tin xác thực và trả về mã thông báo phiên cho máy khách. Máy khách sẽ bao gồm mã thông báo phiên trong mỗi yêu cầu tiếp theo tới ứng dụng FTGO.

Hình 11.2 cho thấy góc nhìn tổng quan về cách ứng dụng FTGO triển khai bảo mật. Ứng dụng FTGO được viết bằng Java và sử dụng khung Spring Security, nhưng tôi sẽ mô tả thiết kế bằng các thuật ngữ chung áp dụng cho các khung khác, chẳng hạn như Passport cho NodeJS.

**Đăng nhập để lấy mã thông báo phiên, đây là một cookie.**



Người giao hàng nhà hàng tiêu dùng



NHẬN /orders/order-xyz Cookie: JSESSIONID=...

**Bao gồm cookie mã thông báo phiên, dùng để xác định người dùng, trong các yêu cầu tiếp theo.**

HTTP/1.1 200 Đồng ý

Đặt cookie: JSESSIONID=...

...

POST /đăng nhập

id=...mật khẩu=...

Trình duyệt hoặc ứng dụng di động

FTGO

ứng dụng

**Hình 11.1 Đầu tiên, máy khách của ứng dụng FTGO đăng nhập để lấy mã thông báo phiên, thường là cookie. Máy khách bao gồm mã thông báo phiên trong mỗi yêu cầu tiếp theo mà nó thực hiện với ứng dụng.**

**Sử dụng một khuôn khổ bảo mật**

Việc triển khai xác thực và ủy quyền đúng cách là một thách thức. Tốt nhất là sử dụng một khuôn khổ bảo mật đã được chứng minh. Sử dụng khuôn khổ nào tùy thuộc vào ứng dụng của bạncông nghệ của tion. Một số khuôn khổ phổ biến bao gồm:

* *Bảo mật mùa xuân*(<https://projects.spring.io/spring-security/>)—Một khuôn khổ phổ biến cho các ứng dụng Java. Đây là một khuôn khổ tinh vi xử lý xác thựcxác nhận và ủy quyền.
* *Apache Shiro*([https://shiro.apache.org](https://shiro.apache.org/))—Một nền tảng Java khác.
* *Hộ chiếu*([http://www.passportjs.org](http://www.passportjs.org/))—Một khuôn khổ bảo mật phổ biến cho NodeJScác ứng dụng tập trung vào xác thực.

Một phần quan trọng của kiến ​​trúc bảo mật là phiên, nơi lưu trữ ID và vai trò của người đứng đầu. Ứng dụng FTGO là ứng dụng Java EE truyền thống, vì vậy phiên làmột phiên HttpSession trong bộ nhớ. Một phiên được xác định bằng một mã thông báo phiên, mà máy khách bao gồm trong mỗi yêu cầu. Nó thường là một mã thông báo mờ đục như một số ngẫu nhiên mạnh về mặt mật mã. Mã thông báo phiên của ứng dụng FTGO là một cookie HTTP có tên là JSESSIONID.

Phần quan trọng khác của việc triển khai bảo mật là bối cảnh bảo mật, trong đó

lưu trữ thông tin về người dùng thực hiện yêu cầu hiện tại. Spring Security

framework sử dụng phương pháp Java EE chuẩn để lưu trữ ngữ cảnh bảo mật trong một biến tĩnh cục bộ của luồng, có thể dễ dàng truy cập được bằng bất kỳ mã nào được gọi đếnbỏ qua yêu cầu. Trình xử lý yêu cầu có thể gọi SecurityContextHolder.getContext()

.getXác thực()để có được thông tin về người dùng hiện tại, chẳng hạn như

danh tính và vai trò. Ngược lại, khuôn khổ Passport lưu trữ bối cảnh bảo mật dưới dạngthuộc tính người dùng của yêu cầu.

**Đăng nhập bằng ID người dùng**

**Vàmật khẩu. Trả về cookie phiên.**



từ cơ sở dữ liệu

Jane

Khởi tạo

Người sử dụng

cơ sở dữ liệu

Thiết lập

Đọc

Thiết lập

Đọc

NHẬN /orders/order-xyz Cookie: JSESSIONID=...

HTTP/1.1 200 Đồng ý

Đặt cookie: JSESSIONID=...

...

POST /đăng nhập

userId-Jane&mật khẩu=..

Trình xử lý đăng nhập

Lấy thông tin người dùng

Ứng dụng FTGO

Bộ chặn bảo mật dựa trên phiên

UserId: jane quy tắc: [CONSUMER]

...

Bối cảnh bảo mật

UserId: jane quy tắc: [CONSUMER]

...

Phiên họp

Khách hàng dựa trên đăng nhập

Trình xử lý yêu cầu chi tiết đơn hàng

**Cung cấp cookie phiên**

**Hình 11.2 Khi một máy khách của ứng dụng FTGO thực hiện yêu cầu đăng nhập,Trình xử lý đăng nhậpxác thực người dùng, khởi tạo thông tin người dùng phiên và trả về cookie mã thông báo phiên, cookie này xác định một cách an toànphiên. Tiếp theo, khi máy khách thực hiện yêu cầu có chứa mã thông báo phiên,****SessionBasedSecurity- Interceptorlấy thông tin người dùng từ phiên được chỉ định và thiết lập ngữ cảnh bảo mật. Trình xử lý yêu cầu, chẳng hạn như****Trình xử lý yêu cầu chi tiết đơn hàng, lấy thông tin người dùng từ bảo mậtbối cảnh.**

Trình tự các sự kiện được thể hiện trong Hình 11.2 như sau:

**1** Khách hàng gửi yêu cầu đăng nhập vào ứng dụng FTGO.

**2**Yêu cầu đăng nhập được xử lý bởi LoginHandler, xác minh thông tin đăng nhập, tạothực hiện phiên và lưu trữ thông tin về người đứng đầu trong phiên.

**3** Trình xử lý đăng nhậptrả về mã thông báo phiên cho máy khách.

**4** Máy khách bao gồm mã thông báo phiên trong các yêu cầu gọi hoạt động.

**5** Các yêu cầu này trước tiên được xử lý bởi SessionBasedSecurityInterceptor. Bộ chặn xác thực từng yêu cầu bằng cách xác minh mã thông báo phiên và thiết lập ngữ cảnh bảo mật. Ngữ cảnh bảo mật mô tả người đứng đầu và các vai trò của người đó.

**6**Trình xử lý yêu cầu sử dụng ngữ cảnh bảo mật để xác định xem có cho phép người dùng thực hiện thao tác được yêu cầu và lấy danh tính của họ hay không.

Ứng dụng FTGO sử dụng quyền hạn dựa trên vai trò. Nó định nghĩa một số vai trò tương ứng với các loại người dùng khác nhau, bao gồm CONSUMER, RESTAURANT, COURIER và ADMIN. Nó sử dụng cơ chế bảo mật khai báo của Spring Security để hạn chế quyền truy cập vào URL và phương thức dịch vụ cho các vai trò cụ thể. Các vai trò cũng được đan xen vào logic kinh doanh. Ví dụ, người tiêu dùng chỉ có thể truy cập đơn hàng của họ, trong khi người quản trị có thể truy cập tất cả các đơn hàng.

Thiết kế bảo mật được sử dụng bởi ứng dụng FTGO nguyên khối chỉ là một cách có thể để triển khai bảo mật. Ví dụ, một nhược điểm của việc sử dụng phiên trong bộ nhớ là nó yêu cầu tất cả các yêu cầu cho một phiên cụ thể phải được định tuyến đến cùng một phiên bản ứng dụng. Yêu cầu này làm phức tạp việc cân bằng tải và hoạt động. Ví dụ, bạn phải triển khai cơ chế thoát phiên chờ tất cả các phiên hết hạn trước khi tắt phiên bản ứng dụng. Một cách tiếp cận thay thế, tránh được những vấn đề này, là lưu trữ phiên trong cơ sở dữ liệu.

Đôi khi bạn có thể loại bỏ hoàn toàn phiên phía máy chủ. Ví dụ, nhiều ứng dụng có các máy khách API cung cấp thông tin xác thực của chúng, chẳng hạn như khóa API và bí mật, trong mọi yêu cầu. Do đó, không cần phải duy trì phiên phía máy chủ. Ngoài ra, ứng dụng có thể lưu trữ trạng thái phiên trong mã thông báo phiên. Sau trong phần này, tôi sẽ mô tả một cách sử dụng mã thông báo phiên để lưu trữ trạng thái phiên. Nhưng chúng ta hãy bắt đầu bằng cách xem xét những thách thức khi triển khai bảo mật trong kiến ​​trúc dịch vụ vi mô.

###### Triển khai bảo mật trong kiến ​​trúc vi dịch vụ

Kiến trúc dịch vụ vi mô là kiến ​​trúc phân tán. Mỗi yêu cầu bên ngoài được xử lý bởi cổng API và ít nhất một dịch vụ. Ví dụ, hãy xem xét get-Truy vấn OrderDetails(), được thảo luận trong chương 8. Cổng API xử lý truy vấn này bằng cách gọi một số dịch vụ, bao gồm Order Service, Kitchen Service và Accounting Service. Mỗi dịch vụ phải triển khai một số khía cạnh bảo mật. Ví dụ, Order Service chỉ được phép người tiêu dùng xem đơn hàng của họ, điều này đòi hỏi sự kết hợp giữa xác thực và ủy quyền. Để triển khai bảo mật trong kiến ​​trúc microservice, chúng ta cần xác định ai chịu trách nhiệm xác thực người dùng và ai chịu trách nhiệm ủy quyền.

Một thách thức khi triển khai bảo mật trong ứng dụng microservices là chúng ta không thể sao chép thiết kế từ ứng dụng monolithic. Đó là vì hai khía cạnh của kiến ​​trúc bảo mật của ứng dụng monolithic không phải là điểm khởi đầu cho kiến ​​trúc microservices:

* + - * *Bối cảnh bảo mật trong bộ nhớ*—Sử dụng ngữ cảnh bảo mật trong bộ nhớ, chẳng hạn như luồng-cục bộ, để truyền xung quanh danh tính người dùng. Các dịch vụ không thể chia sẻ bộ nhớ, vì vậy chúng không thể sử dụng ngữ cảnh bảo mật trong bộ nhớ, chẳng hạn như luồng cục bộ, để truyền xung quanh

danh tính người dùng. Trong kiến ​​trúc vi dịch vụ, chúng ta cần một cơ chế khác để truyền danh tính người dùng từ dịch vụ này sang dịch vụ khác.

* *Phiên họp tập trung*—Vì ngữ cảnh bảo mật trong bộ nhớ không có ý nghĩa, nên phiên trong bộ nhớ cũng vậy. Về lý thuyết, nhiều dịch vụ có thể truy cập phiên dựa trên cơ sở dữ liệu, ngoại trừ việc nó sẽ vi phạm nguyên tắc liên kết lỏng lẻo. Chúng ta cần một cơ chế phiên khác trong kiến ​​trúc dịch vụ vi mô.

Chúng ta hãy bắt đầu khám phá bảo mật trong kiến ​​trúc vi dịch vụ bằng cách xem cách xử lý xác thực.

**HXÁC THỰC ANDLING TRONGGiao diện lập trình ứng dụng (API)CỔNG**

Có một số cách khác nhau để xử lý xác thực. Một tùy chọn là để các dịch vụ riêng lẻ xác thực người dùng. Vấn đề với cách tiếp cận này là nó cho phép các yêu cầu chưa xác thực vào mạng nội bộ. Nó phụ thuộc vào mọi nhóm phát triển triển khai bảo mật đúng cách trong tất cả các dịch vụ của họ. Do đó, có nguy cơ đáng kể về một ứng dụng chứa lỗ hổng bảo mật.

Một vấn đề khác khi triển khai xác thực trong các dịch vụ là các máy khách khác nhau xác thực theo những cách khác nhau. Các máy khách Pure API cung cấp thông tin xác thực với mỗi yêu cầu bằng cách sử dụng, ví dụ, xác thực cơ bản. Các máy khách khác có thể đăng nhập trước rồi cung cấp mã thông báo phiên với mỗi yêu cầu. Chúng tôi muốn tránh yêu cầu các dịch vụ xử lý một tập hợp các cơ chế xác thực đa dạng.

Một cách tiếp cận tốt hơn là API gateway xác thực yêu cầu trước khi chuyển tiếp yêu cầu đó đến các dịch vụ. Việc tập trung xác thực API trong API gateway có lợi thế là chỉ có một nơi để xác thực đúng. Do đó, khả năng xảy ra lỗ hổng bảo mật sẽ nhỏ hơn nhiều. Một lợi ích khác là chỉ có API gateway phải xử lý nhiều cơ chế xác thực khác nhau. Nó ẩn sự phức tạp này khỏi các dịch vụ.

Hình 11.3 cho thấy cách tiếp cận này hoạt động như thế nào. Khách hàng xác thực với cổng API. Khách hàng API bao gồm thông tin xác thực trong mỗi yêu cầu. Khách hàng dựa trên đăng nhập POST thông tin xác thực của người dùng đến xác thực của cổng API và nhận được mã thông báo phiên. Sau khi cổng API đã xác thực một yêu cầu, nó sẽ gọi một hoặc nhiều dịch vụ.

**Mẫu:Mã thông báo truy cập**

Cổng API truyền một mã thông báo chứa thông tin về người dùng, chẳng hạn như danh tính và vai trò của họ, tới các dịch vụ mà nó gọi. Xem[http://microservices.io/](http://microservices.io/patterns/security/access-token.html) [mẫu/bảo mật/mã thông báo truy cập.html](http://microservices.io/patterns/security/access-token.html).

Một dịch vụ được API gateway gọi cần biết người thực hiện yêu cầu. Nó cũng phải xác minh rằng yêu cầu đã được xác thực. Giải pháp là API gateway sẽ bao gồm một mã thông báo trong mỗi yêu cầu dịch vụ. Dịch vụ sử dụng mã thông báo để xác thực yêu cầu và lấy thông tin về người thực hiện. API gateway cũng có thể cung cấp cùng một mã thông báo cho các máy khách hướng phiên để sử dụng làm mã thông báo phiên.

**Các máy khách API cung cấp thông tin xác thực trong tiêu đề Ủy quyền.**



**Truyền mã thông báo cho các dịch vụ để họ có thể xác định và cấp quyền cho người dùng.**

Đặt hàng

Dịch vụ

NHẬN /đơn hàng/1

...MÃ\_THÔNG\_TIN\_BẢO\_MẬT...

HTTP/1.1 200 Đồng ý

...MÃ\_THÔNG\_TIN\_BẢO\_MẬT...

NHẬN /đơn hàng/1

...MÃ\_THÔNG\_TIN\_BẢO\_MẬT...

POST /đăng nhập

id=...mật khẩu=...

Cổng API

Khách hàng dựa trên đăng nhập

Khách hàng API

NHẬN /đơn hàng/1

Quyền hạn: ...GIẤY CHỨNG NHẬN...

...

Bộ chặn xác thực

**Đăng nhập khách hàng đầu tiên có được**

**một mã thông báo bảo mật.**

**Bao gồm mã thông báo bảo mật**

**trong mỗi yêu cầu.**

**Hình 11.3 Cổng API xác thực các yêu cầu từ máy khách và bao gồm mã thông báo bảo mật trong các yêu cầu mà nó thực hiện đối với các dịch vụ. Các dịch vụ sử dụng mã thông báo để lấy thông tin về người đứng đầu. Cổng API có thểcũng sử dụng mã thông báo bảo mật làm mã thông báo phiên.**

Trình tự các sự kiện dành cho máy khách API như sau:

**1** Khách hàng đưa ra yêu cầu có chứa thông tin xác thực.

**2**Cổng API xác thực thông tin đăng nhập, tạo mã thông báo bảo mật và chuyển mã thông báo đó tới dịch vụ hoặc các dịch vụ.

Trình tự các sự kiện dành cho máy khách đăng nhập như sau:

**1** Khách hàng gửi yêu cầu đăng nhập có chứa thông tin xác thực.

**2** Cổng API trả về mã thông báo bảo mật.

**3** Máy khách bao gồm mã thông báo bảo mật trong các yêu cầu gọi hoạt động.

**4**Cổng API xác thực mã thông báo bảo mật và chuyển tiếp nó đến dịch vụ hoặc các dịch vụ.

Ở phần sau của chương này, tôi sẽ mô tả cách triển khai mã thông báo, nhưng trước tiên chúng ta hãy xem xét khía cạnh chính khác của bảo mật: ủy quyền.

**HỦY QUYỀN ANDLING**

Xác thực thông tin xác thực của khách hàng là quan trọng nhưng không đủ. Một ứng dụng cũng phải triển khai cơ chế ủy quyền để xác minh rằng khách hàng được phép thực hiện thao tác được yêu cầu. Ví dụ, trong ứng dụng FTGOtruy vấn getOrderDetails() chỉ có thể được gọi bởi người tiêu dùng đã Đặt hàng (một ví dụ về bảo mật dựa trên phiên bản) và nhân viên dịch vụ khách hàng đang hỗ trợ người tiêu dùng.

Một nơi để thực hiện ủy quyền là cổng API. Ví dụ, nó có thể,hạn chế quyền truy cập vào GET /orders/{orderId} chỉ dành cho người dùng là người tiêu dùng và đại lý dịch vụ khách hàng. Nếu người dùng không được phép truy cập vào một đường dẫn cụ thể, cổng API có thể từ chối yêu cầu trước khi chuyển tiếp yêu cầu đó đến dịch vụ. Cũng giống như xác thực, việc tập trung ủy quyền trong cổng API giúp giảm nguy cơ lỗ hổng bảo mật. Bạn có thể triển khai ủy quyền trong cổng API bằng cách sử dụng một khuôn khổ bảo mật, chẳng hạn như Spring Security.

Một nhược điểm của việc triển khai ủy quyền trong API Gateway là nó có nguy cơ liên kết API Gateway với các dịch vụ, yêu cầu chúng phải được cập nhật đồng bộ. Hơn nữa, API Gateway thường chỉ có thể triển khai quyền truy cập dựa trên vai trò vào các đường dẫn URL. Nhìn chung, API Gateway không thực tế khi triển khai ACL kiểm soát quyền truy cập vào từng đối tượng miền riêng lẻ, vì điều đó đòi hỏi phải có kiến ​​thức chi tiết về logic miền của dịch vụ.

Nơi khác để triển khai ủy quyền là trong các dịch vụ. Một dịch vụ có thể triển khai ủy quyền dựa trên vai trò cho URL và cho các phương thức dịch vụ. Nó cũng có thể triển khai ACL để quản lý quyền truy cập vào các tổng hợp. Ví dụ, Order Service có thể triển khai cơ chế ủy quyền dựa trên vai trò và ACL để kiểm soát quyền truy cập vào các đơn hàng. Các dịch vụ khác trong ứng dụng FTGO triển khai logic ủy quyền tương tự.

**BạnHÁTJWTS ĐỂ CHUYỂN NHẬN DẠNG VÀ VAI TRÒ CỦA NGƯỜ**

Khi triển khai bảo mật trong một dịch vụ vi môkiến trúc, bạn cần quyết định loại mã thông báo nào mà cổng API nên sử dụng để truyền thông tin người dùng đến các dịch vụ. Có hai loại mã thông báo để lựa chọn. Một tùy chọn là sử dụng mã thông báo mờ đục, thường là UUID. Nhược điểm của mã thông báo mờ đục là chúng làm giảm hiệu suất và tính khả dụng và tăng độ trễ. Đó là vì người nhận mã thông báo như vậy phải thực hiện cuộc gọi RPC đồng bộ đến dịch vụ bảo mật để xác thực mã thông báo và truy xuất thông tin người dùng.

Một cách tiếp cận thay thế, loại bỏ việc gọi đến dịch vụ bảo mật, là sử dụng một mã thông báo trong suốt chứa thông tin về người dùng. Một tiêu chuẩn phổ biến như vậy cho các mã thông báo trong suốt là JSON Web Token (JWT). JWT là cách tiêu chuẩn để biểu diễn các khiếu nại một cách an toàn, chẳng hạn như danh tính và vai trò của người dùng, giữa hai bên. JWT có một pay-load, là một đối tượng JSON chứa thông tin về người dùng, chẳng hạn như danh tính và vai trò của họ, và các siêu dữ liệu khác, chẳng hạn như ngày hết hạn. Nó được ký bằng một bí mật mà chỉ người tạo JWT biết, chẳng hạn như cổng API và người nhận JWT, chẳng hạn như dịch vụ. Bí mật đảm bảo rằng bên thứ ba có ác ý không thể làm giả hoặc can thiệp vào JWT.

Một vấn đề với JWT là vì một token là độc lập, nên nó không thể thu hồi được. Theo thiết kế, một dịch vụ sẽ thực hiện thao tác yêu cầu sau khi xác minh chữ ký và ngày hết hạn của JWT. Do đó, không có cách thực tế nào để thu hồi một JWT riêng lẻ đã rơi vào tay bên thứ ba có ác ý. Giải pháp là phát hành JWT với thời gian hết hạn ngắn, vì điều đó hạn chế những gì bên có ác ý có thể làm. Tuy nhiên, một nhược điểm của JWT tồn tại trong thời gian ngắn là ứng dụng phải liên tục phát hành lại JWT để duy trì phiên hoạt động. May mắn thay, đây là một trong nhiều giao thức được giải quyết bằng tiêu chuẩn bảo mật gọi là OAuth 2.0. Hãy cùng xem cách thức hoạt động của nó.

**BạnHÁTOAĐại học2.0TRONG KIẾN TRÚC VI DỊCH VỤ**

Giả sử bạn muốn triển khai Dịch vụ người dùng cho ứng dụng FTGO quản lý cơ sở dữ liệu người dùng chứa thông tin người dùng, chẳng hạn như thông tin xác thực và vai trò. Cổng API gọi Dịch vụ người dùng để xác thực yêu cầu của khách hàng và lấy JWT. Bạn có thể thiết kế API Dịch vụ người dùng và triển khai nó bằng cách sử dụng khung web yêu thích của mình. Nhưng đó là chức năng chung không dành riêng cho ứng dụng FTGO—phát triển một dịch vụ như vậy sẽ không phải là cách sử dụng hiệu quả các nguồn lực phát triển.

May mắn thay, bạn không cần phải phát triển cơ sở hạ tầng bảo mật như thế này.Bạn có thể sử dụng một dịch vụ hoặc khuôn khổ có sẵn triển khai một tiêu chuẩn gọi là OAuth 2.0. OAuth 2.0 là một giao thức ủy quyền ban đầu được thiết kế để cho phép người dùng dịch vụ đám mây công cộng, chẳng hạn như GitHub hoặc Google, cấp cho ứng dụng của bên thứ ba quyền truy cập vào thông tin của mình mà không tiết lộ mật khẩu. Ví dụ: OAuth 2.0 là cơ chếcho phép bạn cấp quyền truy cập an toàn vào kho lưu trữ GitHub của bạn cho dịch vụ Tích hợp liên tục (CI) dựa trên đám mây của bên thứ ba.

Mặc dù mục tiêu ban đầu của OAuth 2.0 là cấp quyền truy cập vào các dịch vụ đám mây công cộng, bạn cũng có thể sử dụng nó để xác thực và cấp quyền trong ứng dụng của mình. Hãy cùng xem nhanh cách kiến ​​trúc dịch vụ vi mô có thể sử dụng OAuth 2.0.

**Giới thiệu về OAuth 2.0**

OAuth 2.0 là một chủ đề phức tạp. Trong chương này, tôi chỉ có thể cung cấp một cái nhìn tổng quan ngắn gọn vàmô tả cách nó có thể được sử dụng trong kiến ​​trúc microservice. Để biết thêm thông tin về OAuth 2.0, hãy xem sách trực tuyến*Máy chủ OAuth 2.0*bởi Aaron Parecki ([www.oauth.com](http://www.oauth.com/)). Chương 7 của*Spring Microservices trong hành động*(Manning, 2017) cũng đề cập đến chủ đề này ([https://livebook.manning.com/#!/book/spring-microservices-in-](https://livebook.manning.com/%23!/book/spring-microservices-in-action/chapter-7/) [hành động/chương-7/](https://livebook.manning.com/%23!/book/spring-microservices-in-action/chapter-7/)).

Các khái niệm chính trong OAuth 2.0 như sau:

* + Máy chủ ủy quyền—Cung cấp API để xác thực người dùng và lấy mã thông báo truy cập và mã thông báo làm mới. Spring OAuth là một ví dụ tuyệt vời về một khuôn khổ để xây dựng máy chủ ủy quyền OAuth 2.0.
  + Mã thông báo truy cập—Một mã thông báo cấp quyền truy cập vàoTài nguyênMáy chủ. Định dạng của mã thông báo truy cập phụ thuộc vào việc triển khai. Nhưng một số triển khai, chẳng hạn như Spring OAuth, sử dụng JWT.
* Làm mới mã thông báo—Một mã thông báo tồn tại lâu dài nhưng có thể thu hồi đượcKhách hàngsử dụng để có được một cái mớiMã thông báo truy cập.
* Máy chủ tài nguyên—Một dịch vụ sử dụng mã thông báo truy cập để cấp quyền truy cập. Trong kiến ​​trúc dịch vụ vi mô, các dịch vụ là máy chủ tài nguyên.
* Khách hàng—Một khách hàng muốntruy cập mộtMáy chủ tài nguyên. Trong kiến ​​trúc vi dịch vụ,Cổng APIlà ứng dụng khách OAuth 2.0.

Sau trong phần này, tôi sẽ mô tả cách hỗ trợ máy khách dựa trên đăng nhập. Nhưng trước tiên, hãy nói về cách xác thực máy khách API.

Hình 11.4 cho thấy cách API Gateway xác thực yêu cầu từ máy khách API. API Gateway xác thực máy khách API bằng cách gửi yêu cầu đến máy chủ ủy quyền OAuth 2.0, máy chủ này trả về mã thông báo truy cập. Sau đó, API Gateway thực hiện một hoặc nhiều yêu cầu có chứa mã thông báo truy cập đến các dịch vụ.

Trình tự các sự kiện được thể hiện trong hình 11.4 như sau:

**1** Khách hàng đưa ra yêu cầu, cung cấp thông tin xác thực của mình bằng phương pháp xác thực cơ bản.

**2**Cổng API thực hiện yêu cầu cấp mật khẩu OAuth 2.0 ([www.oauth.com/](http://www.oauth.com/oauth2-servers/access-tokens/password-grant/) [oauth2-servers/mã thông báo truy cập/cấp mật khẩu/](http://www.oauth.com/oauth2-servers/access-tokens/password-grant/))đến máy chủ xác thực OAuth 2.0.

**Yêu cầu cấp mật khẩu**

Mùa xuân OAuth2Máy chủ xác thực



HTTP/1.1200 Được

...

{

"access\_token": "Mã truy cập"

...

}

Cổng API

POST/oauth/token userid=...&password=...

Cơ sở dữ liệu người dùng

Khách hàng API

GET /orders/1 Quyền hạn: Cơ bản...

....



Dịch vụ đặt hàng

NHẬN /đơn hàng/1

Quyền hạn: Bearer AccessToken

**Chứa ID người dùng và vai trò của họ**

**Hình 11.4 Cổng API xác thực máy khách API bằng cách gửi yêu cầu Cấp mật khẩu đến máy chủ xác thực OAuth 2.0. Máy chủ trả về mã thông báo truy cập, cổng API chuyển mã thông báo này đến các dịch vụ. Dịch vụ xác minh chữ ký của mã thông báo và trích xuất thông tin về người dùng, bao gồm danh tính và vai trò của họ.**

**3**Máy chủ xác thực xác thực thông tin đăng nhập của máy khách API và trả về mã thông báo truy cập và mã thông báo làm mới.

**4**Cổng API bao gồm mã thông báo truy cập trong các yêu cầu mà nó thực hiện đối với các dịch vụ. Một dịch vụ xác thực mã thông báo truy cập và sử dụng nó để ủy quyền cho yêu cầu.

Cổng API dựa trên OAuth 2.0 có thể xác thực các máy khách hướng phiên bằng cách sử dụng mã thông báo truy cập OAuth 2.0 làm mã thông báo phiên. Hơn nữa, khi mã thông báo truy cập hết hạn, nó có thể lấy mã thông báo truy cập mới bằng cách sử dụng mã thông báo làm mới. Hình 11.5 cho thấy cách cổng API có thể sử dụng OAuth 2.0 để xử lý các máy khách hướng phiên. Một API cli-ent khởi tạo phiên bằng cách POST thông tin xác thực của nó đến điểm cuối /login của API gateway. API gateway trả về một mã thông báo truy cập và một mã thông báo làm mới cho máy khách. Sau đó, máy khách API cung cấp cả hai mã thông báo khi thực hiện yêu cầu đến API gateway.

**Yêu cầu cấp mật khẩu**



Người sử dụng

cơ sở dữ liệu

Cổng API

Đặt hàng

Dịch vụ

NHẬN/đơn hàng/1

Bánh quy:access\_token=...;làm mới\_token...

HTTP/1.1 200ĐƯỢC RỒI

...

{

"access\_token": "Mã truy cập"

...

}

POST/đăng nhập userId=...&password=...

POST/oauth/token userid=...&password=...

Mùa xuân OAuth2Máy chủ xác thực

NHẬN /đơn hàng/1

Quyền hạn: Bearer AccessToken

Dựa trên đăng nhậpkhách hàng

HTTP/1.1 200 Đồng ý

Đặt-Cookie: access\_token=... Đặt-Cookie:refresh\_token=...

Bộ chặn xác thực phiên

Trình xử lý đăng nhập

**Hình 11.5 Một máy khách đăng nhập bằng cách POST thông tin xác thực của mình tới cổng API. Cổng API xác thực thông tin xác thực bằng máy chủ xác thực OAuth 2.0 và trả về mã thông báo truy cập và mã thông báo làm mới dưới dạng cookie.Máy khách sẽ bao gồm các mã thông báo này trong các yêu cầu gửi tới cổng API.**

Trình tự các sự kiện như sau:

**1** Máy khách dựa trên thông tin đăng nhập sẽ POST thông tin đăng nhập của mình tới cổng API.

**2**Trình xử lý đăng nhập của cổng API thực hiện yêu cầu cấp mật khẩu OAuth 2.0 ([www.oauth.com/oauth2-servers/access-tokens/password-grant/](http://www.oauth.com/oauth2-servers/access-tokens/password-grant/))đến OAuth

Xác thực 2.0máy chủ.

**3**Máy chủ xác thực xác thực thông tin đăng nhập của máy khách và trả về mã thông báo truy cập và mã thông báo làm mới.

**4**Cổng API trả về mã thông báo truy cập và làm mới cho máy khách, ví dụ như dưới dạng cookie.

**5**Máy khách bao gồm mã thông báo truy cập và làm mới trong các yêu cầu gửi tới cổng API.

**6**Cổng API củaBộ chặn xác thực phiênxác thực mã thông báo truy cập và đưa mã này vào các yêu cầu gửi tới dịch vụ.

Nếu mã thông báo truy cập đã hết hạn hoặc sắp hết hạn, cổng API sẽ lấy mã thông báo truy cập mới bằng cách thực hiện yêu cầu cấp quyền làm mới OAuth 2.0 ([www.oauth.com/](http://www.oauth.com/oauth2-servers/access-tokens/refreshing-access-tokens/) [oauth2-servers/mã thông báo truy cập/làm mới mã thông báo truy cập/](http://www.oauth.com/oauth2-servers/access-tokens/refreshing-access-tokens/)), chứa mã thông báo làm mới, đến máy chủ ủy quyền. Nếu mã thông báo làm mới chưa hết hạn hoặc chưa bị thu hồi, máy chủ ủy quyền sẽ trả về mã thông báo truy cập mới. API Gateway sẽ chuyển mã thông báo truy cập mới đến các dịch vụ và trả về cho máy khách.

Một lợi ích quan trọng khi sử dụng OAuth 2.0 là đây là tiêu chuẩn bảo mật đã được chứng minh.Sử dụng Máy chủ xác thực OAuth 2.0 có sẵn nghĩa là bạn không phải tốn thời gian phát minh lại bánh xe hoặc có nguy cơ phát triển một thiết kế không an toàn. Nhưng OAuth

2.0 không phải là cách duy nhất để triển khai bảo mật trong kiến ​​trúc microservice. Bất kể bạn sử dụng cách tiếp cận nào, ba ý tưởng chính như sau:

* Cổng API có trách nhiệm xác thực máy khách.
* Cổng API và các dịch vụ sử dụng mã thông báo minh bạch, chẳng hạn như JWT, để truyền thông tin về bên chủ thể.
* Một dịch vụ sử dụng mã thông báo để lấy danh tính và vai trò của người đứng đầu.

Bây giờ chúng ta đã xem xét cách bảo mật dịch vụ, hãy cùng xem cách cấu hình chúng.

#### Thiết kế các dịch vụ có thể cấu hình

Hãy tưởng tượng rằng bạn chịu trách nhiệm cho Dịch vụ Lịch sử Đơn hàng. Như hình 11.6 cho thấy, dịch vụ này sử dụng các sự kiện từ Apache Kafka và đọc và ghi các mục bảng AWS DynamoDB. Để dịch vụ này chạy, nó cần nhiều thuộc tính cấu hình khác nhau, bao gồm vị trí mạng của Apache Kafka và thông tin xác thực và vị trí mạng cho AWS DynamoDB.

Giá trị của các thuộc tính cấu hình này phụ thuộc vào môi trường mà dịch vụ đang chạy. Ví dụ, môi trường nhà phát triển và môi trường sản xuất sẽ sử dụng các môi giới Apache Kafka khác nhau và thông tin xác thực AWS khác nhau. Không hợp lý khi cố định các giá trị thuộc tính cấu hình của một môi trường cụ thể vào dịch vụ có thể triển khai vì điều đó sẽ yêu cầu phải xây dựng lại cho từng môi trường. Thay vào đó, một dịch vụ nên được xây dựng một lần theo đường ống triển khai và triển khai vào nhiều môi trường.

Cũng không có ý nghĩa gì khi kết nối cứng các tập hợp thuộc tính cấu hình khác nhau vào mã nguồn và sử dụng, ví dụ, cơ chế hồ sơ của Spring Framework để

**Môi trường cụ thể cấu hình**

Apache Kafka

bootstrap.servers=kafka1:9092

..

«Kênh sự kiện đặt hàng»

Người tiêu dùng Apache Kafka

Dịch vụ Lịch sử đơn hàng

AWS DynamoDB

«Kênh sự kiện giao hàng»

aws.access.key.id=... aws.secret.access.key=... aws.region=...

Bộ điều hợp DynamoDB

**Cấu hình cụ thể theo môi trường**

**Hình 11.6Dịch vụ Lịch sử đơn hàngsử dụng Apache Kafka và AWS DynamoDB. Nó cần phải làđược cấu hình với vị trí mạng, thông tin đăng nhập, v.v. của từng dịch vụ.**

chọn bộ thích hợp khi chạy. Đó là vì làm như vậy sẽ tạo ra lỗ hổng bảo mật và giới hạn nơi có thể triển khai. Ngoài ra, dữ liệu nhạy cảm như thông tin xác thực nên được lưu trữ an toàn bằng cơ chế lưu trữ bí mật, chẳng hạn như Hashicorp Vault ([www.vaultproject.io](http://www.vaultproject.io/)) hoặc AWS Parameter Store ([https://docs.aws](https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html)

[.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html](https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html)). Thay vào đó, bạn nên cung cấp các thuộc tính cấu hình phù hợp cho dịch vụ khi chạy bằng cách sử dụng mẫu cấu hình Externalized.

**Mẫu: Cấu hình bên ngoài**

Cung cấp các giá trị thuộc tính cấu hình, chẳng hạn như thông tin xác thực cơ sở dữ liệu và vị trí mạng, cho một dịch vụ khi chạy. Xem[http://microservices.io/patterns/externalized-](http://microservices.io/patterns/externalized-configuration.html) [cấu hình.html](http://microservices.io/patterns/externalized-configuration.html).

Cơ chế cấu hình bên ngoài cung cấp các giá trị thuộc tính cấu hình cho một phiên bản dịch vụ khi chạy. Có hai cách tiếp cận chính:

* *Mô hình đẩy*—Cơ sở hạ tầng triển khai truyền các thuộc tính cấu hình cho phiên bản dịch vụ bằng cách sử dụng, ví dụ, các biến môi trường của hệ điều hành hoặc tệp cấu hình.
* *Mô hình kéo*—Thể hiện dịch vụ đọc các thuộc tính cấu hình của nó từ máy chủ cấu hình.

Chúng ta sẽ xem xét từng cách tiếp cận, bắt đầu với mô hình đẩy.

###### Sử dụng cấu hình bên ngoài dựa trên đẩy

Mô hình đẩy dựa vào sự cộng tác của môi trường triển khai và dịch vụ. Môi trường triển khai cung cấp các thuộc tính cấu hình khi tạo một phiên bản dịch vụ. Nó có thể, như hình 11.7 cho thấy, truyền các thuộc tính cấu hình dưới dạng biến môi trường. Ngoài ra, môi trường triển khai có thể cung cấp các thuộc tính cấu hình bằng cách sử dụng tệp cấu hình. Sau đó, phiên bản dịch vụ sẽ đọc các thuộc tính cấu hình khi khởi động.

Cấu hình

Quá trình

BOOTSTRAP\_SERVERS=kafka1:9092 AWS\_ACCESS\_KEY\_ID= AWS\_SECRET\_ACCESS\_KEY=...

AWS\_REGION=...

....

Đọc

Dịch vụ Lịch sử đơn hàng

ví dụ

Biến môi trường

Cơ sở hạ tầng triển khai

Tạo ra

**Hình 11.7 Khi cơ sở hạ tầng triển khai tạo ra một phiên bản củaDịch vụ Lịch sử đơn hàng, nó thiết lập các biến môi trường chứa cấu hình bên ngoài.Dịch vụ Lịch sử đơn hàngđọc các biến môi trường đó.**

Môi trường triển khai và dịch vụ phải thống nhất về cách cung cấp các thuộc tính cấu hình. Cơ chế chính xác phụ thuộc vào môi trường triển khai cụ thể. Ví dụ, chương 12 mô tả cách bạn có thể chỉ định các biến môi trường của một vùng chứa Docker.

Hãy tưởng tượng rằng bạn đã quyết định cung cấp thuộc tính cấu hình bên ngoàigiá trị sử dụng biến môi trường. Ứng dụng của bạn có thể gọi System.getenv() để lấy giá trị của chúng. Nhưng nếu bạn là nhà phát triển Java, có khả năng bạn đang sử dụng một khuôn khổ cung cấp cơ chế thuận tiện hơn. Các dịch vụ FTGO được xây dựng bằng Spring Boot, có cơ chế cấu hình bên ngoài cực kỳ linh hoạt, có thể truy xuất các thuộc tính cấu hình từ nhiều nguồn khác nhau với các quy tắc ưu tiên được xác định rõ ràng ([https://docs.spring.io/spring-boot/docs/current/reference/html/boot-](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html) [features-external-config.html](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html)). Hãy cùng xem nó hoạt động như thế nào.

Spring Boot đọc thuộc tính từ nhiều nguồn khác nhau.Tôi thấy các nguồn sau đây hữu ích trong kiến ​​trúc vi dịch vụ:

**1** Đối số dòng lệnh

**2**ỨNG DỤNG SPRING\_JSON, một biến môi trường hệ điều hành hoặc thuộc tính hệ thống JVM có chứa JSON

**3** Thuộc tính hệ thống JVM

**4**Môi trường hệ điều hànhbiến số

**5** Một tập tin cấu hình trong thư mục hiện tại

Giá trị thuộc tính cụ thể từ một nguồn trước đó trong danh sách này sẽ ghi đè lên cùng một thuộc tính từ một nguồn sau đó trong danh sách này. Ví dụ, các biến môi trường hệ điều hành ghi đè lên các thuộc tính được đọc từ tệp cấu hình.

Spring Boot cung cấp các thuộc tính này cho Spring FrameworkApplicationContext. Ví dụ, một dịch vụ có thể lấy giá trị của một thuộc tính bằng cách sử dụng chú thích @Value:

lớp công khai OrderHistoryDynamoDBConfiguration {

@Giá trị("${aws.region}")Chuỗi riêng tư awsRegion;

Spring Framework khởi tạo trường awsRegion thành giá trị của thuộc tính aws.region. Thuộc tính này được đọc từ một trong các nguồn được liệt kê trước đó, chẳng hạn như tệp cấu hình hoặc từ biến môi trường AWS\_REGION.

Mô hình đẩy là một cơ chế hiệu quả và được sử dụng rộng rãi để cấu hình một dịch vụ. Tuy nhiên, một hạn chế là việc cấu hình lại một dịch vụ đang chạy có thể là một thách thức, nếu không muốn nói là không thể. Cơ sở hạ tầng triển khai có thể không cho phép bạn thay đổi cấu hình bên ngoài của một dịch vụ đang chạy mà không cần khởi động lại dịch vụ đó. Ví dụ, bạn không thể thay đổi các biến môi trường của một quy trình đang chạy. Một hạn chế khác là có nguy cơ các giá trị thuộc tính cấu hình bị phân tán trong suốt quá trình định nghĩa nhiều dịch vụ. Do đó, bạn có thể cân nhắc sử dụng mô hình dựa trên kéo. Hãy cùng xem cách thức hoạt động của mô hình này.

###### Sử dụng cấu hình bên ngoài dựa trên kéo

Trong mô hình kéo, một phiên bản dịch vụ đọc các thuộc tính cấu hình của nó từ một cấu hìnhmáy chủ tion. Hình 11.8 cho thấy cách thức hoạt động. Khi khởi động, một phiên bản dịch vụ truy vấn dịch vụ cấu hình để biết cấu hình của nó. Các thuộc tính cấu hình để truy cập máy chủ cấu hình, chẳng hạn như vị trí mạng của nó, được cung cấp cho phiên bản dịch vụ thông qua cơ chế cấu hình dựa trên đẩy, chẳng hạn như các biến môi trường.

Có nhiều cách khác nhau để triển khai máy chủ cấu hình, bao gồm:

* + - * Hệ thống kiểm soát phiên bản như Git
      * Cơ sở dữ liệu SQL và NoSQL
      * Các máy chủ cấu hình chuyên biệt, chẳng hạn như Spring Cloud Config Server, Hashicorp Vault, là nơi lưu trữ dữ liệu nhạy cảm như thông tin đăng nhập và AWS Parameter Store

Quá trình

Cấu hình

URL\_SERVER\_CONFIG=...

Đặt hàng

Dịch vụ Lịch sửví dụ

getConfiguration(“orderHistoryService”)

BOOTSTRAP\_SERVERS=kafka1:9092 AWS\_ACCESS\_KEY\_ID= AWS\_SECRET\_ACCESS\_KEY=...

AWS\_REGION=...

....

Tạo ra

Cơ sở hạ tầng triển khai

Máy chủ cấu hình

Biến môi trường

**Hình 11.8 Khi khởi động, một phiên bản dịch vụlấy các thuộc tính cấu hình của nó từ máy chủ cấu hình. Cơ sở hạ tầng triển khai cung cấp các thuộc tính cấu hình để truy cập máy chủ cấu hình.**

Dự án Spring Cloud Config là một ví dụ điển hình về một khuôn khổ dựa trên máy chủ cấu hình. Nó bao gồm một máy chủ và một máy khách. Máy chủ hỗ trợ nhiều chương trình phụ trợ để lưu trữ các thuộc tính cấu hình, bao gồm hệ thống kiểm soát phiên bản, cơ sở dữ liệu và Hashicorp Vault. Máy khách truy xuất các thuộc tính cấu hình từ máy chủ và đưa chúng vào Spring ApplicationContext.

Sử dụng máy chủ cấu hình có một số lợi ích:

* *Tập trungcấu hình*—Tất cả các thuộc tính cấu hình được lưu trữ ở một nơi, giúp quản lý dễ dàng hơn. Hơn nữa, để loại bỏ các thuộc tính cấu hình trùng lặp, một số triển khai cho phép bạn xác định các mặc định toàn cục, có thể ghi đè trên cơ sở từng dịch vụ.
* *Giải mã minh bạch dữ liệu nhạy cảm*—Mã hóa dữ liệu nhạy cảm như thông tin xác thực cơ sở dữ liệu là biện pháp bảo mật tốt nhất. Tuy nhiên, một thách thức khi sử dụng mã hóa là thường thì phiên bản dịch vụ cần giải mã chúng, nghĩa là cần có khóa mã hóa. Một số triển khai máy chủ cấu hình tự động giải mã các thuộc tính trước khi trả về dịch vụ.
* *Cấu hình lại động*—Một dịch vụ có khả năng phát hiện thuộc tính được cập nhậtgiá trị bằng cách thăm dò và tự cấu hình lại.

Nhược điểm chính của việc sử dụng máy chủ cấu hình là trừ khi được cơ sở hạ tầng cung cấp, nếu không thì đó là một phần cơ sở hạ tầng khác cần được thiết lập và bảo trì. May mắn thay, có nhiều khuôn khổ mã nguồn mở, chẳng hạn như Spring Cloud Config, giúp chạy máy chủ cấu hình dễ dàng hơn.

Bây giờ chúng ta đã xem xét cách thiết kế các dịch vụ có thể cấu hình, hãy nói vềcách thiết kế các dịch vụ có thể quan sát được.

#### Thiết kếdịch vụ có thể quan sát được

Giả sử bạn đã triển khai ứng dụng FTGO vào sản xuất. Bạn có thể muốn biết ứng dụng đang làm gì: yêu cầu mỗi giây, sử dụng tài nguyên và

v.v. Bạn cũng cần được cảnh báo nếu có vấn đề, chẳng hạn như trường hợp dịch vụ bị lỗi hoặc đĩa đầy—tốt nhất là trước khi nó ảnh hưởng đến người dùng. Và nếu có vấn đề, bạn cần có khả năng khắc phục sự cố và xác định nguyên nhân gốc rễ.

Nhiều khía cạnh của việc quản lý ứng dụng trong sản xuất nằm ngoài phạm vi của nhà phát triển, chẳng hạn như giám sát tính khả dụng và sử dụng phần cứng. Đây rõ ràng là trách nhiệm của hoạt động. Nhưng có một số mẫu mà bạn, với tư cách là nhà phát triển dịch vụ, phải triển khai để giúp dịch vụ của bạn dễ quản lý và khắc phục sự cố hơn. Các mẫu này, được hiển thị trong hình 11.9, cho thấy hành vi và tình trạng của một phiên bản dịch vụ. Chúng cho phép hệ thống giám sát theo dõi và trực quan hóa trạng thái của dịch vụ và tạo cảnh báo khi có sự cố. Các mẫu này cũng giúp khắc phục sự cố dễ dàng hơn.

Bạn có thể sử dụng các mẫu sau để thiết kế các dịch vụ có thể quan sát được:

* + - *API kiểm tra sức khỏe*— Hiển thị điểm cuối trả về tình trạng hoạt động của dịch vụ.
    - *Tổng hợp nhật ký*—Ghi nhật ký hoạt động dịch vụ và viết nhật ký vào máy chủ ghi nhật ký tập trung, cung cấp chức năng tìm kiếm và cảnh báo.

Chìa khóa

Kiểm tra sức khỏe

Mẫu API

Phân phối

mẫu theo dõi

Triệu hồi

Sức khỏe

kiểm traGiao diện lập trình ứng dụng (API)

Số liệu

người xuất khẩu

Có thể quan sát được

Dịch vụ

Ứng dụng

mẫu số liệu

Kiểm toán

Ghi nhật ký cơ sở dữ liệu

bộ chuyển đổi bộ chuyển đổi

Kiểm toán

mẫu ghi nhật ký

Tổng hợp nhật ký

mẫu

Ngoại lệ

mẫu theo dõi

Tệp nhật ký

Đường ống tổng hợp nhật ký

Ngoại lệphóng viên

Người gọi kiểm tra sức khỏe, chẳng hạn như dịch vụ giám sát

Phân phốitheo dõi xuất khẩu

Ngoại lệDịch vụ theo dõi

Kiểm toán cơ sở dữ liệu

Ghi nhật kýMáy chủ

Dịch vụ số liệu

Phân phốimáy chủ theo dõi

Mẫu

Trách nhiệm hoạt động

Trách nhiệm của nhà phát triển

Người tham gia mẫu

**Hình 11.9 Các mẫu quan sát cho phép các nhà phát triển và hoạt động hiểu được hành vi của ứng dụng và khắc phục sự cố. Các nhà phát triển có trách nhiệm đảm bảo rằng các dịch vụ của họ có thể quan sát được. Các hoạt động có trách nhiệm về cơ sở hạ tầng thu thập thông tin do các dịch vụ tiết lộ.**

* *Phân phối theo dõi*—Gán một ID duy nhất cho mỗi yêu cầu bên ngoài và theo dõi các yêu cầu khi chúng di chuyển giữa các dịch vụ.
* *Theo dõi ngoại lệ*—Báo cáo ngoại lệ cho dịch vụ theo dõi ngoại lệ, dịch vụ này sẽ loại bỏ các ngoại lệ trùng lặp, cảnh báo cho nhà phát triển và theo dõi quá trình giải quyết từng ngoại lệ.
* *Số liệu ứng dụng*—Các dịch vụ duy trì các số liệu, chẳng hạn như bộ đếmvà các thước đo, sau đó đưa chúng tới máy chủ số liệu.
* *Nhật ký kiểm toán*—Nhật kýhành động của người dùng.

Một đặc điểm riêng biệt của hầu hết các mẫu này là mỗi mẫu đều có một thành phần nhà phát triển và một thành phần hoạt động. Ví dụ, hãy xem xét mẫu API kiểm tra tình trạng. Nhà phát triển chịu trách nhiệm đảm bảo rằng dịch vụ của họ triển khai điểm cuối kiểm tra tình trạng. Hoạt động chịu trách nhiệm về hệ thống giám sát định kỳ gọi API kiểm tra tình trạng. Tương tự như vậy, đối với mẫu Tổng hợp nhật ký, nhà phát triển chịu trách nhiệm đảm bảo rằng dịch vụ của họ ghi lại thông tin hữu ích, trong khi hoạt động chịu trách nhiệm tổng hợp nhật ký.

Chúng ta hãy xem xét từng mẫu này, bắt đầu với mẫu API kiểm tra tình trạng.

###### Sử dụng Sức khỏekiểm tra mẫu API

Đôi khi một dịch vụ có thể đang chạy nhưng không thể xử lý các yêu cầu. Ví dụ, một phiên bản dịch vụ mới bắt đầu có thể không sẵn sàng chấp nhận các yêu cầu. Ví dụ, Dịch vụ Người dùng FTGO mất khoảng 10 giây để khởi tạo các bộ điều hợp cơ sở dữ liệu và nhắn tin. Sẽ vô nghĩa nếu cơ sở hạ tầng triển khai định tuyến các yêu cầu HTTP đến một phiên bản dịch vụ cho đến khi phiên bản đó sẵn sàng xử lý chúng.

Ngoài ra, một phiên bản dịch vụ có thể bị lỗi mà không kết thúc. Ví dụ, một lỗi có thể khiến một phiên bản của Consumer Service hết kết nối cơ sở dữ liệu và không thể truy cập cơ sở dữ liệu. Cơ sở hạ tầng triển khai không nên định tuyến các yêu cầu đến một phiên bản dịch vụ đã bị lỗi nhưng vẫn đang chạy. Và nếu phiên bản dịch vụ không phục hồi, cơ sở hạ tầng triển khai phải chấm dứt phiên bản đó và tạo một phiên bản mới.

**Mẫu: API kiểm tra tình trạng sức khỏe**

Một dịch vụ hiển thị điểm cuối API kiểm tra tình trạng sức khỏe, chẳng hạn nhưNHẬN /sức khỏe, trả về tình trạng sức khỏe của dịch vụ. Xem[http://microservices.io/patterns/observability/health-](http://microservices.io/patterns/observability/health-check-api.html) [kiểm tra-api.html](http://microservices.io/patterns/observability/health-check-api.html).

Một phiên bản dịch vụ cần có khả năng cho cơ sở hạ tầng triển khai biết liệu nó có thể xử lý các yêu cầu hay không. Một giải pháp tốt là để một dịch vụ triển khai điểm cuối kiểm tra tình trạng, được hiển thị trong hình 11.10. Thư viện Java Spring Boot Actuator,ví dụ, triển khai điểm cuối GET /actuator/health, trả về 200 nếu và chỉ nếu dịch vụ hoạt động tốt, và trả về 503 nếu không. Tương tự như vậy, HealthChecks .NET

thư viện triển khai điểm cuối GET /hc ([https://docs.microsoft.com/en-us/dotnet/](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/monitor-app-health) [chuẩn/kiến trúc dịch vụ vi mô/triển khai ứng dụng có khả năng phục hồi/ứng dụng giám sát-](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/monitor-app-health) [sức khỏe](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/monitor-app-health)). Cơ sở hạ tầng triển khai định kỳ gọi điểm cuối này để xác địnhtình trạng của dịch vụ và thực hiện hành động thích hợp nếu tình trạng không ổn định.

**Kiểm tra kết nối của dịch vụ với các dịch vụ cơ sở hạ tầng**

**Ví dụ: hệ thống giám sát, sổ đăng ký dịch vụ và các hệ thống khác**



Sức khỏekiểm tra

điểm cuối

Triệu hồi

Kiểm tra

Dịch vụ

Kiểm tra

Bộ điều hợp cơ sở dữ liệu

Bộ chuyển đổi tin nhắn

Sức khỏekiểm tra trình xử lý yêu cầu

Sức khỏekiểm tra người gọi

MySQL

Người môi giới tin nhắn

**Hình 11.10 Một dịch vụ triển khai điểm cuối kiểm tra tình trạng sức khỏe, được gọi định kỳ bởicơ sở hạ tầng triển khai để xác định tình trạng của phiên bản dịch vụ.**

Trình xử lý yêu cầu kiểm tra sức khỏe thường kiểm tra các kết nối của phiên bản dịch vụ với các dịch vụ bên ngoài. Ví dụ, nó có thể thực hiện truy vấn kiểm tra đối với cơ sở dữ liệu. Nếu tất cảcác thử nghiệm thành công, Health Check Request Handler trả về phản hồi lành mạnh, chẳng hạn như mã trạng thái HTTP 200. Nếu bất kỳ thử nghiệm nào trong số chúng không thành công, nó sẽ trả về phản hồi không lành mạnh, chẳng hạn như mã trạng thái HTTP 500.

Health Check Request Handler có thể chỉ trả về phản hồi HTTP trống với mã trạng thái phù hợp. Hoặc có thể trả về mô tả chi tiết về tình trạng của từng bộ điều hợp. Thông tin chi tiết hữu ích cho việc khắc phục sự cố. Nhưng vì nó có thể chứa thông tin nhạy cảm, một số khuôn khổ, chẳng hạn như Spring Boot Actuator, cho phép bạn định cấu hình mức độ chi tiết trong phản hồi điểm cuối về tình trạng.

Có hai vấn đề bạn cần cân nhắc khi sử dụng kiểm tra tình trạng. Đầu tiên là việc triển khai điểm cuối, phải báo cáo lại về tình trạng của phiên bản dịch vụ. Vấn đề thứ hai là cách cấu hình cơ sở hạ tầng triển khai để gọi điểm cuối kiểm tra tình trạng. Trước tiên, hãy xem cách triển khai điểm cuối.

**TÔITHỰC HIỆN ĐIỂM CUỐI KIỂM TRA SỨC KHỎE**

Mã thực hiện điểm cuối kiểm tra tình trạng sức khỏe phải xác định tình trạng sức khỏe của phiên bản dịch vụ. Một cách tiếp cận đơn giản là xác minh rằng phiên bản dịch vụ có thể truy cập các dịch vụ cơ sở hạ tầng bên ngoài của nó. Cách thực hiện điều này phụ thuộc vào

dịch vụ cơ sở hạ tầng. Ví dụ, mã kiểm tra tình trạng có thể xác minh rằng nó được kết nối với RDBMS bằng cách lấy kết nối cơ sở dữ liệu và thực hiện truy vấn thử nghiệm. Một cách tiếp cận phức tạp hơn là thực hiện giao dịch tổng hợp mô phỏng việc gọi API của dịch vụ bởi máy khách. Loại kiểm tra tình trạng này kỹ lưỡng hơn, nhưng có khả năng tốn nhiều thời gian hơn để triển khai và mất nhiều thời gian hơn để thực hiện.

Một ví dụ tuyệt vời về thư viện kiểm tra tình trạng là Spring Boot Actuator. Như đã đề cập trước đó, nó triển khai điểm cuối /actuator/health. Mã triển khai điểm cuối này trả về kết quả thực hiện một tập hợp các kiểm tra tình trạng. Bằng cách sử dụng quy ước trên cấu hình, Spring Boot Actuator triển khai một tập hợp các kiểm tra tình trạng hợp lý dựa trên các dịch vụ cơ sở hạ tầng mà dịch vụ sử dụng. Ví dụ, nếu một dịch vụ sử dụng JDBC DataSource, Spring Boot Actuator sẽ cấu hình một kiểm tra tình trạng thực hiện truy vấn thử nghiệm. Tương tự, nếu dịch vụ sử dụng RabbitMQ message broker, nó sẽ tự động cấu hình một kiểm tra tình trạng để xác minh rằng máy chủ RabbitMQ đang hoạt động.

Bạn cũng có thể tùy chỉnh hành vi này bằng cách triển khai các kiểm tra sức khỏe bổ sung cho dịch vụ của mình. Bạn triển khai kiểm tra sức khỏe tùy chỉnh bằng cách xác định một lớp thực hiệnđịnh nghĩa giao diện HealthIndicator. Giao diện này định nghĩa phương thức health(), được gọi bởi việc triển khai điểm cuối /actuator/health. Nó trả về kết quả kiểm tra tình trạng sức khỏe.

**TÔIGỌI ĐIỂM CUỐI KIỂM TRA SỨC KHỎE**

Điểm cuối kiểm tra tình trạng sức khỏe không có nhiều tác dụng nếu không ai gọi đến. Khi bạn triển khai dịch vụ, bạn phải cấu hình cơ sở hạ tầng triển khai để gọi điểm cuối. Cách bạn thực hiện tùy thuộc vào các chi tiết cụ thể của cơ sở hạ tầng triển khai của bạn. Ví dụ, như mô tả trong chương 3, bạn có thể cấu hình một số sổ đăng ký dịch vụ, chẳng hạn như Netflix Eureka, để gọi điểm cuối kiểm tra tình trạng sức khỏe nhằm xác định xem lưu lượng có nên được định tuyến đến phiên bản dịch vụ hay không. Chương 12 thảo luận về cách cấu hình Docker và Kubernetes để gọi điểm cuối kiểm tra tình trạng.

###### Áp dụng mô hình tổng hợp Nhật ký

Nhật ký là một công cụ khắc phục sự cố có giá trịcông cụ. Nếu bạn muốn biết ứng dụng của mình có vấn đề gì, một nơi tốt để bắt đầu là các tệp nhật ký. Nhưng sử dụng nhật ký trong kiến ​​trúc dịch vụ vi mô là một thách thức. Ví dụ, hãy tưởng tượng bạn đang gỡ lỗi một vấn đề vớigetOrderDetails() query. Như đã mô tả trong chương 8, ứng dụng FTGO triển khai truy vấn này bằng cách sử dụng API composition. Do đó, các mục nhập nhật ký bạn cần được phân tán trên các tệp nhật ký của cổng API và một số dịch vụ, bao gồm Order Service và Kitchen Service.

**Mẫu: Tổng hợp nhật ký**

Tổng hợp nhật ký của tất cả các dịch vụ trong cơ sở dữ liệu tập trung hỗ trợ tìm kiếm và cảnh báo. Xem[http://microservices.io/patterns/observability/application-logging](http://microservices.io/patterns/observability/application-logging.html)

[.html](http://microservices.io/patterns/observability/application-logging.html).

Giải pháp là sử dụng tổng hợp nhật ký. Như hình 11.11 cho thấy, đường ống tổng hợp nhật ký gửi nhật ký của tất cả các phiên bản dịch vụ đến máy chủ ghi nhật ký tập trung. Sau khi nhật ký được máy chủ ghi nhật ký lưu trữ, bạn có thể xem, tìm kiếm và phân tích chúng. Bạn cũng có thể cấu hình cảnh báo được kích hoạt khi một số thông báo nhất định xuất hiện trong nhật ký.



Người sử dụng

Dịch vụ B phiên bản 1

Nhật ký

Ghi nhật kýthư viện

Xem

Máy chủ ghi nhật ký

Dịch vụMột trường hợp 2

Ghi nhật kýthư viện

**Hình 11.11 Cơ sở hạ tầng tổng hợp nhật ký chuyển nhật ký của từng phiên bản dịch vụ đến máy chủ ghi nhật ký tập trung. Người dùng có thể xem và tìm kiếm nhật ký. Họ cũng có thể thiết lập cảnh báo, được kích hoạt khi mục nhật ký khớp với tiêu chí tìm kiếm.**

Dịch vụ A thể hiện 1

Ghi nhật kýthư viện

Đường ống nhật ký

Nhật ký

Nhật ký

Thông báo

Đường ống ghi nhật ký và máy chủ thường là trách nhiệm của hoạt động. Nhưng các nhà phát triển dịch vụ chịu trách nhiệm viết các dịch vụ tạo ra các bản ghi hữu ích. Trước tiên, hãy xem cách một dịch vụ tạo ra một bản ghi.

**HOW DỊCH VỤ TẠO RA MỘT NHẬT KÝ**

Là một nhà phát triển dịch vụ, có một vài vấn đề bạn cần cân nhắc. Đầu tiên, bạn cần quyết định sử dụng thư viện ghi nhật ký nào. Vấn đề thứ hai là viết mục nhật ký ở đâu. Trước tiên, hãy xem xét thư viện ghi nhật ký.

Hầu hết các ngôn ngữ lập trình đều có một hoặc nhiều thư viện ghi nhật ký giúp dễ dàng tạo các mục nhật ký có cấu trúc chính xác. Ví dụ, ba thư viện ghi nhật ký Java phổ biến là Logback, log4j và JUL (java.util.logging). Ngoài ra còn có SLF4J, đây là API mặt tiền ghi nhật ký cho nhiều khung ghi nhật ký khác nhau. Tương tự, Log4JS là một khung ghi nhật ký phổ biến cho NodeJS. Một cách hợp lý để sử dụng ghi nhật ký là rải rác các lệnh gọi đến một trong các thư viện ghi nhật ký này trong mã dịch vụ của bạn. Nhưng nếu bạn có các yêu cầu ghi nhật ký nghiêm ngặt mà thư viện ghi nhật ký không thể thực thi, bạn có thể cần phải xác định API ghi nhật ký của riêng mình để bao bọc một thư viện ghi nhật ký.

Bạn cũng cần quyết định nơi ghi nhật ký. Theo truyền thống, bạn sẽ cấu hình khung ghi nhật ký để ghi vào tệp nhật ký ở vị trí được biết đến trong hệ thống tệp. Nhưng với các công nghệ triển khai hiện đại hơn, chẳng hạn như container và không có máy chủ,

được mô tả trong chương 12, đây thường không phải là cách tiếp cận tốt nhất. Trong một số môi trường, chẳng hạn như AWS Lambda, thậm chí không có hệ thống tệp "vĩnh viễn" để ghi nhật ký vào! Thay vào đó, dịch vụ của bạn sẽ ghi vào stdout. Cơ sở hạ tầng triển khai sau đó sẽ quyết định phải làm gì với đầu ra của dịch vụ của bạn.

**TCƠ SỞ HẠ TẦNG TỔNG HỢP LOG**

Cơ sở hạ tầng ghi nhật ký chịu trách nhiệm tổng hợp các bản ghi, lưu trữ chúng và cho phép người dùng tìm kiếm chúng. Một cơ sở hạ tầng ghi nhật ký phổ biến là ngăn xếp ELK. ELK bao gồm ba sản phẩm nguồn mở:

* *Tìm kiếm đàn hồi*—Một cơ sở dữ liệu NoSQL hướng đến tìm kiếm văn bản được sử dụng làm máy chủ ghi nhật ký
* *Nhật ký*—Một đường ống nhật ký tổng hợp các nhật ký dịch vụ và ghi chúng vào Elasticsearch
* *Kibana*—Một công cụ trực quan cho Elasticsearch

Các đường ống nhật ký nguồn mở khác bao gồm Fluentd và Apache Flume. Ví dụ về máy chủ ghi nhật ký bao gồm các dịch vụ đám mây, chẳng hạn như AWS CloudWatch Logs, cũng như nhiều dịch vụ thương mại khác. Tổng hợp nhật ký là một công cụ gỡ lỗi hữu ích trong kiến ​​trúc dịch vụ vi mô.

Bây giờ chúng ta hãy xem xét việc theo dõi phân tán,đây là một cách khác để hiểu hành vi của ứng dụng dựa trên dịch vụ vi mô.

###### Sử dụng mẫu theo dõi phân tán

Hãy tưởng tượng bạn là một nhà phát triển FTGO đang điều tra lý do tại sao truy vấn getOrderDetails() lại chậm lại. Bạn đã loại trừ khả năng sự cố là do sự cố mạng bên ngoài. Độ trễ tăng lên có thể do cổng API hoặc một trong các dịch vụ mà nó đã gọi gây ra. Một tùy chọn là xem thời gian phản hồi trung bình của từng dịch vụ. Vấn đề với tùy chọn này là nó là giá trị trung bình trên các yêu cầu chứ không phải là sự cố thời gian cho một yêu cầu riêng lẻ. Thêm vào đó, các tình huống phức tạp hơn có thể liên quan đến nhiều lệnh gọi dịch vụ lồng nhau. Bạn thậm chí có thể không quen thuộc với tất cả các dịch vụ. Do đó, việc khắc phục sự cố và chẩn đoán các loại sự cố hiệu suất này trong kiến ​​trúc dịch vụ vi mô có thể rất khó khăn.

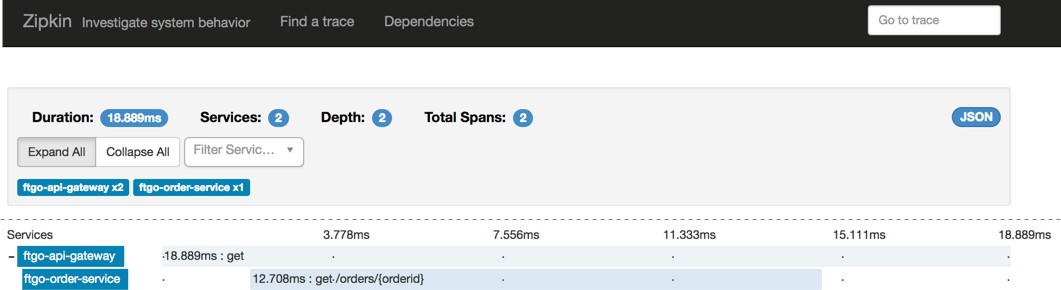
**Mẫu: Theo dõi phân tán**

Chỉ định cho mỗi yêu cầu bên ngoài một ID duy nhất và ghi lại cách nó chảy qua hệ thốngtừ dịch vụ này sang dịch vụ khác trong một máy chủ tập trung cung cấp khả năng trực quan hóa và phân tích. Xem<http://microservices.io/patterns/observability/distributed-tracing.html>.

Một cách tốt để có được cái nhìn sâu sắc về ứng dụng của bạnđang làm là sử dụng theo dõi phân tán. Theo dõi phân tán tương tự như một trình lập hồ sơ hiệu suất trong một ứng dụng đơn khối. Nó ghi lại thông tin (ví dụ, thời gian bắt đầu và thời gian kết thúc) về cây các cuộc gọi dịch vụ được thực hiện khi xử lý một yêu cầu. Sau đó, bạn có thể xem các dịch vụ

tương tác trong quá trình xử lý các yêu cầu bên ngoài, bao gồm cả việc phân tích thời gian dành cho việc gì.

Hình 11.12 cho thấy một ví dụ về cách máy chủ theo dõi phân tán hiển thị những gì xảy ra khi cổng API xử lý một yêu cầu. Nó hiển thị yêu cầu đếnCổng API và yêu cầu mà cổng gửi đến Order Service. Đối với mỗi yêu cầu, máy chủ theo dõi phân tán sẽ hiển thị hoạt động được thực hiện và thời gian của yêu cầu.



Cha mẹkhoảngTrẻ em dấu vết

**Hình 11.12 Máy chủ Zipkin cho thấy cách ứng dụng FTGO xử lý yêu cầu được định tuyến bởi cổng API tớiDịch vụ đặt hàng. Mỗi yêu cầu được biểu diễn bằng một dấu vết. Một dấu vết là một tập hợp các khoảng. Mỗi khoảng, có thể chứa các khoảng con, là lệnh gọi một dịch vụ. Tùy thuộc vào mức độ chi tiết được thu thập, một khoảng cũng có thể biểu diễn lệnh gọi một hoạt động bên trong một dịch vụ.**

Hình 11.12 cho thấy cái mà trong thuật ngữ theo dõi phân tán được gọi là dấu vết. Một dấu vết biểu thị một yêu cầu bên ngoài và bao gồm một hoặc nhiều khoảng. Một khoảng biểu thị một hoạt động và các thuộc tính chính của nó là tên hoạt động, dấu thời gian bắt đầu và thời gian kết thúc. Một khoảng có thể có một hoặc nhiều khoảng con, biểu thị các hoạt động lồng nhau. Ví dụ, một khoảng cấp cao nhất có thể biểu thị lệnh gọi cổng API, như trường hợp trong hình 11.12. Các khoảng con của nó biểu thị lệnh gọi dịch vụ của cổng API.

Một tác dụng phụ có giá trị của việc theo dõi phân tán là nó gán một ID duy nhất cho mỗi yêu cầu bên ngoài. Một dịch vụ có thể bao gồm ID yêu cầu trong các mục nhật ký của nó. Khi kết hợp với tổng hợp nhật ký, ID yêu cầu cho phép bạn dễ dàng tìm thấy tất cả các mục nhật ký cho một yêu cầu bên ngoài cụ thể. Ví dụ, đây là một mục nhật ký mẫu từ Order Service:

2018-03-04 17:38:12.032 GỠ LỖI [ftgo-order-

dịch vụ,8d8fdc37be104cc6,8d8fdc37be104cc6,sai]

7 --- [nio-8080-exec-6]org.hibernate.SQL:

chọn order0\_.id là id1\_3\_0\_, order0\_.consumer\_id là consumer2\_3\_0\_, order 0\_.city là city3\_3\_0\_,

order0\_.delivery\_state là delivery4\_3\_0\_, order0\_.street1 là street5\_3\_0\_, order0\_.street2 là street6\_3\_0\_, order0\_.zip là zip7\_3\_0\_,

order0\_.delivery\_time là delivery8\_3\_0\_, order0\_.a

Các[ftgo-order-service,8d8fdc37be104cc6,8d8fdc37be104cc6,sai]một phần của mục nhập nhật ký (Bối cảnh chẩn đoán được ánh xạ SLF4J—xem[www.slf4j.org/manual.html](http://www.slf4j.org/manual.html)) chứa thông tin từ cơ sở hạ tầng theo dõi phân tán. Nó bao gồm bốn giá trị:

* + - * ftgo-order-service—Tên của ứng dụng
      * 8d8fdc37be104cc6—Cáctheo dõiId
      * 8d8fdc37be104cc6—CácMã span
      * SAI—Chỉ ra rằng khoảng thời gian này không được xuất sang máy chủ theo dõi phân tán

Nếu bạn tìm kiếm nhật ký cho 8d8fdc37be104cc6, bạn sẽ tìm thấy tất cả các mục nhật ký cho yêu cầu đó.Hình 11.13 cho thấy cách thức hoạt động của việc theo dõi phân tán. Có hai phần để phân phối

theo dõi: một thư viện công cụ, được sử dụng bởi mỗi dịch vụ và mộtmáy chủ theo dõi. Thư viện công cụ quản lý các dấu vết và khoảng cách. Nó cũng thêm

Theo dõi XYZ

Khoảng cách DEF: Thứ tựDịch vụ

Span ABC: Cổng API

NHẬN/đơn hàng/1 HTTP/1.1

....

NHẬN/đơn hàng/1HTTP/1.1

X-B3-TraceId: XYZ

X-B3-ParentSpanId: ABC

APIĐặt hàng

Cổng dịch vụ

Dịch vụ: Cổng API

TraceId: XYZ ParentSpan: KHÔNG CÓ Span: ABC

Chuyên chở

Dịch vụ: Đặt dịch vụ

TraceId: XYZ ParentSpan: ABC Khoảng cách: DEF

Thiết bị đo lườngthư viện

Thiết bị đo lườngthư viện

Người sử dụng



Lượt xem dấu vết

Cơ sở dữ liệu theo dõi

|  |  |
| --- | --- |
| Máy chủ theo dõi phân tán | |
|  |  |

**Hình 11.13 Mỗi dịch vụ (bao gồm cả cổng API) sử dụng một thư viện công cụ. Thư viện công cụ gán một ID cho mỗi yêu cầu bên ngoài, truyền bá trạng thái theo dõi giữadịch vụ và báo cáo kéo dài đến máy chủ theo dõi phân tán.**

theo dõi thông tin, chẳng hạn như ID theo dõi hiện tại và ID khoảng thời gian cha, đến các yêu cầu gửi đi. Ví dụ, một tiêu chuẩn chung để truyền bá thông tin theo dõi là tiêu chuẩn B3 (<https://github.com/openzipkin/b3-propagation>), sử dụng đầu-chẳng hạn như X-B3-TraceId và X-B3-ParentSpanId.Thư viện công cụ cũng báo cáo các dấu vết cho máy chủ theo dõi phân tán. Máy chủ theo dõi phân tán lưu trữ các dấu vết và cung cấp giao diện người dùng để trực quan hóa chúng.

Chúng ta hãy xem xét thư viện công cụ và máy chủ theo dõi phân phối, bắt đầu từ thư viện.

**BạnHÁT MỘT THƯ VIỆN NHẠC CỤ**

Thư viện công cụ xây dựng cây các khoảng và gửi chúng đến máy chủ theo dõi phân tán. Mã dịch vụ có thể gọi trực tiếp thư viện công cụ, nhưng điều đó sẽ đan xen logic công cụ với logic kinh doanh và các logic khác. Một cách tiếp cận sạch hơn là sử dụng các bộ chặn hoặc lập trình hướng khía cạnh (AOP).

Một ví dụ tuyệt vời về một khuôn khổ dựa trên AOP là Spring Cloud Sleuth. Nó sử dụng cơ chế AOP của Spring Framework để tự động tích hợp theo dõi phân tán vào dịch vụ. Do đó, bạn phải thêm Spring Cloud Sleuth làm một phần phụ thuộc của dự án. Dịch vụ của bạn không cần phải gọi API theo dõi phân tán ngoại trừ trong những trường hợp không được Spring Cloud Sleuth xử lý.

**MỘTVỀ MÁY CHỦ THEO DÕI PHÂN TÁN**

Thư viện công cụ gửi các khoảng đến một máy chủ theo dõi phân tán. Máy chủ theo dõi phân tán sẽ ghép các khoảng lại với nhau để tạo thành các dấu vết hoàn chỉnh và lưu trữ chúng trong cơ sở dữ liệu. Một máy chủ theo dõi phân tán phổ biến là Open Zipkin. Zipkin ban đầu được phát triển bởi Twitter. Các dịch vụ có thể phân phối các khoảng đến Zipkin bằng HTTP hoặc một môi giới tin nhắn. Zipkin lưu trữ các dấu vết trong một backend lưu trữ, là một cơ sở dữ liệu SQL hoặc NoSQL. Nó có một giao diện người dùng hiển thị các dấu vết, nhưđược hiển thị trước đó trong hình 11.12. AWS X-ray là một ví dụ khác về máy chủ theo dõi phân tán.

###### Áp dụng mẫu số liệu ứng dụng

Một phần quan trọng của môi trường sản xuất là giám sát và cảnh báo. Như hình 11.14 cho thấy, hệ thống giám sát thu thập các số liệu, cung cấp thông tin quan trọng về tình trạng của ứng dụng, từ mọi phần của ngăn xếp công nghệ. Các số liệu bao gồm từ các số liệu cấp cơ sở hạ tầng, chẳng hạn như CPU, bộ nhớ và sử dụng đĩa, đến các số liệu cấp ứng dụng, chẳng hạn như độ trễ yêu cầu dịch vụ và số lượng yêu cầu được thực hiện. Ví dụ, Order Service thu thập các số liệu về số lượng đơn hàng đã đặt, đã phê duyệt và đã từ chối. Các số liệu được thu thập bởi một dịch vụ số liệu, cung cấp khả năng trực quan hóa và cảnh báo.

**Mẫu: Số liệu ứng dụng**

Các dịch vụ báo cáo số liệu cho một máy chủ trung tâm cung cấp khả năng tổng hợp, trực quan hóa,và cảnh báo.



Người sử dụng

Mã ứng dụng

Khung ứng dụngThời gian chạy ngôn ngữ

Dịch vụ thể hiện

Cơ sở hạ tầng triển khai

Thư viện số liệu

XemThông báo

Hình ảnh hóa

Cảnh báo

Dịch vụ số liệu

Thu thập số liệu

Mẫu số liệu:

tên=cpu\_percent giá trị=68 dấu thời gian=34938934893 kích thước:

máy=node1

...

Cơ sở dữ liệu số liệu

**Hình 11.14 Các số liệu ở mọi cấp độ của ngăn xếp được thu thập và lưu trữ trong một dịch vụ số liệu,cung cấp khả năng trực quan hóa và cảnh báo.**

Các số liệu được lấy mẫu định kỳ. Một mẫu số liệu có ba thuộc tính sau:

* *Tên*—Tên của số liệu, chẳng hạn nhưjvm\_memory\_max\_byteshoặcđơn hàng đã đặt
* *Giá trị*—Một giá trị số
* *Dấu thời gian*—Thời gian lấy mẫu

Ngoài ra, một số hệ thống giám sát hỗ trợ khái niệm về kích thước, đó làcặp tên-giá trị tùy ý. Ví dụ,jvm\_memory\_max\_bytesđược báo cáo với kích thướccác sions nhưkhu vực="đống",id="PS Eden Space"Vàkhu vực="đống",id="PS thế hệ cũ". Các chiều thường được sử dụng để cung cấp thông tin bổ sung, chẳng hạn như tên máy hoặc tên dịch vụ hoặc mã định danh phiên bản dịch vụ. Hệ thống giám sát thường tổng hợp (tổng hoặc trung bình) các mẫu số liệu theo một hoặc nhiều chiều.

Nhiều khía cạnh của việc giám sát là trách nhiệm của hoạt động. Nhưng một nhà phát triển dịch vụ chịu trách nhiệm cho hai khía cạnh của số liệu. Đầu tiên, họ phải thiết lập dịch vụ của mình để thu thập số liệu về hành vi của dịch vụ. Thứ hai, họ phải đưa các số liệu dịch vụ đó, cùng với số liệu từ JVM và khung ứng dụng, ra máy chủ số liệu.

Trước tiên, chúng ta hãy xem cách một dịch vụ thu thập số liệu.

**CDỊCH VỤ THU THẬP-ĐO LƯỜNG CẤP ĐỘ**

Lượng công việc bạn cần thực hiện để thu thập số liệu phụ thuộc vào các khuôn khổ mà ứng dụng của bạn sử dụng và số liệu bạn muốn thu thập. Ví dụ, một dịch vụ dựa trên Spring Boot có thể thu thập (và hiển thị) các số liệu cơ bản, chẳng hạn như số liệu JVM, bằng cách bao gồm

thư viện Micrometer Metrics như một phần phụ thuộc và sử dụng một vài dòng cấu hình. Cấu hình tự động của Spring Boot sẽ đảm nhiệm việc cấu hình thư viện số liệu và hiển thị số liệu. Một dịch vụ chỉ cần sử dụng API Micrometer Metrics trực tiếp nếu nó thu thập số liệu cụ thể của ứng dụng.

Danh sách sau đây cho thấy cách OrderService có thể thu thập số liệu về số lượng đơn hàng đã đặt, đã chấp thuận và đã từ chối. Nó sử dụng MeterRegistry, là Micrometer Metrics do giao diện cung cấp, để thu thập số liệu tùy chỉnh. Mỗi phương pháp sẽ tăng một bộ đếm có tên thích hợp.

**Liệt kê 11.1Dịch vụ đặt hàngtheo dõi số lượng đơn hàng đã đặt, đã được chấp thuận vàvật bị loại bỏ.**

lớp công khai OrderService {

@Autowired

riêng tư MeterRegistry meterRegistry;

công khai Order createOrder(...) {

...

**Các số liệu của MicrometAPI thư viện để quản lý các đồng hồ đo ứng dụng cụ thể**

**Tăng bộ đếm đơn hàng được đặt**

meterRegistry.counter("đơn\_hàng\_đặt").tăng();trả lại đơn hàng;

}

**khi một lệnh đã cóđã được đặt thành công**

public void approveOrder(orderId dài) {

...

meterRegistry.counter("đơn hàng đã được phê duyệt").tăng();

}

public void rejectOrder(orderId dài) {

...

meterRegistry.counter("đơn hàng bị từ chối").tăng();

}

**DCUNG CẤP SỐ LIỆU CHO DỊCH VỤ SỐ LIỆU**

**Tăng bộ đếm approvedOrders khi đơn hàng đã được chấp thuận**

**Tăng bộ đếm rejectedOrders khi một đơn hàng đã bị từ chối**

Một dịch vụ cung cấp số liệu cho Metrics Service theo một trong hai cách: đẩy hoặc kéo. Với mô hình đẩy, một phiên bản dịch vụ gửi số liệu đến Metrics Service bằng cách gọi API. Ví dụ, số liệu AWS Cloudwatch triển khai mô hình đẩy.

Với mô hình kéo, Metrics Service (hoặc tác nhân của nó chạy cục bộ) sẽ gọi API dịch vụ để lấy số liệu từ phiên bản dịch vụ. Prometheus, một hệ thống giám sát và cảnh báo nguồn mở phổ biến, sử dụng mô hình kéo.

Ứng dụng FTGODịch vụ đặt hàngsử dụngmicromet-registration-prometheusthư viện để tích hợp với Prometheus. Vì thư viện này nằm trên classpath, SpringKhởi động phơi bày mộtNHẬN /bộ truyền động/prometheusđiểm cuối, trả về số liệu theo định dạng mà Prometheus mong đợi. Số liệu tùy chỉnh từDịch vụ đặt hàngđược báo cáo như sau:

$ curl -v http://localhost:8080/actuator/prometheus | grep \_orders # TRỢ GIÚP tổng số đơn hàng đã đặt

# TYPE bộ đếm tổng số đơn hàng đã đặt

tổng\_số\_đơn\_hàng\_đặt{dịch\_vụ="ftgo-order-service",}1.0 # TRỢ GIÚP approved\_orders\_total

# TYPE approved\_orders\_total counterapproved\_orders\_total{dịch vụ="ftgo-order-service",}1.0

Cácđơn hàng đã đặtví dụ, bộ đếm được báo cáo là một số liệu có kiểuquầy tính tiền.

Máy chủ Prometheus định kỳ thăm dò điểm cuối này để lấy số liệu. Khi số liệu đã có trong Prometheus, bạn có thể xem chúng bằng Grafana, một công cụ trực quan hóa dữ liệu ([https://grafana.com](https://grafana.com/)). Bạn cũng có thể thiết lập cảnh báo cho các số liệu này, chẳng hạn như khitỷ lệ thay đổi của located\_orders\_total giảm xuống dưới một ngưỡng nào đó.

Số liệu ứng dụng cung cấp thông tin chi tiết có giá trị về hành vi của ứng dụng.

Cảnh báo được kích hoạt bởi số liệu cho phép bạn phản hồi nhanh chóng với sự cố sản xuất, có thể là trước khi nó ảnh hưởng đến người dùng. Bây giờ chúng ta hãy xem cách quan sát và phản hồi một nguồn cảnh báo khác: ngoại lệ.

###### Sử dụng mẫu theo dõi Ngoại lệ

Một dịch vụ hiếm khi ghi lại ngoại lệ và khi có, điều quan trọng là bạn phải xác định nguyên nhân gốc rễ. Ngoại lệ có thể là triệu chứng của lỗi hoặc lỗi lập trình. Cách truyền thống để xem ngoại lệ là xem trong nhật ký. Bạn thậm chí có thể cấu hình máy chủ ghi nhật ký để cảnh báo bạn nếu ngoại lệ xuất hiện trong tệp nhật ký. Tuy nhiên, có một số vấn đề với cách tiếp cận này:

* + - * Các tệp nhật ký được định hướng xung quanh một dòngmục nhật ký, trong khi ngoại lệ bao gồm nhiều dòng.
      * Không có cơ chế nào để theo dõi việc giải quyết các ngoại lệ xảy ra trong tệp nhật ký. Bạn sẽ phải sao chép/dán thủ công ngoại lệ vào trình theo dõi sự cố.
      * Có thể có những trường hợp ngoại lệ trùng lặp, nhưng không có cơ chế tự động nào xử lý chúng như một trường hợp.

**Mẫu: Theo dõi ngoại lệ**

Các dịch vụ báo cáo các ngoại lệ cho một dịch vụ trung tâm để loại bỏ các ngoại lệ trùng lặp, tạo cảnh báo và quản lý việc giải quyết các ngoại lệ. Xem[http://microservices.io/](http://microservices.io/patterns/observability/audit-logging.html) [mẫu/khả năng quan sát/kiểm toán-ghi nhật ký.html](http://microservices.io/patterns/observability/audit-logging.html).

Một cách tiếp cận tốt hơn là sử dụng dịch vụ theo dõi ngoại lệ. Như hình 11.15 cho thấy, bạn cấu hình dịch vụ của mình để báo cáo ngoại lệ cho dịch vụ theo dõi ngoại lệ thông qua, ví dụ, REST API. Dịch vụ theo dõi ngoại lệ loại bỏ trùng lặp ngoại lệ, tạo cảnh báo và quản lý việc giải quyết ngoại lệ.

Có một số cách để tích hợp dịch vụ theo dõi ngoại lệ vào ứng dụng của bạn. Dịch vụ của bạn có thể gọi trực tiếp API của dịch vụ theo dõi ngoại lệ. Một cách tiếp cận tốt hơn là sử dụng thư viện máy khách do dịch vụ theo dõi ngoại lệ cung cấp. Ví dụ, thư viện máy khách của HoneyBadger cung cấp một số cơ chế tích hợp dễ sử dụng, bao gồm Bộ lọc Servlet bắt và báo cáo ngoại lệ.



Người sử dụng

Xem & quản lý

Thông báo

Dịch vụ theo dõi ngoại lệ

Dịch vụ đặt hàng

Báo cáo ngoại lệ

Theo dõi ngoại lệthư viện khách hàng

POST/ngoại lệ

java.lang.NullPointerException

tại net.chrisrichardson.ftgo... tại net.chrisrichardson.ftgo... tại net.chrisrichardson.ftgo...

Cơ sở dữ liệu ngoại lệ

**Hình 11.15 Một dịch vụ báo cáo các ngoại lệ cho một dịch vụ theo dõi ngoại lệ, dịch vụ này loại bỏ các bản saongoại lệ và cảnh báo cho nhà phát triển. Nó có giao diện người dùng để xem và quản lý ngoại lệ.**

**Dịch vụ theo dõi ngoại lệ**

Có một số dịch vụ theo dõi ngoại lệ. Một số, chẳng hạn như Honeybadger ([www](http://www.honeybadger.io/)

[.honeybadger.io](http://www.honeybadger.io/)), hoàn toàn dựa trên đám mây. Những cái khác, chẳng hạn như Sentry.io ([https://sentry.io/](https://sentry.io/welcome/) [Chào mừng/](https://sentry.io/welcome/)), cũng có phiên bản mã nguồn mở mà bạn có thể triển khai trên cơ sở hạ tầng của riêng bạn. Các dịch vụ này nhận được ngoại lệ từ ứng dụng của bạn và tạo cảnh báo.Họ cung cấp một bảng điều khiển để xem các ngoại lệ và quản lý việc giải quyết chúng. Một dịch vụ theo dõi ngoại lệ thường cung cấp các thư viện máy khách bằng nhiều ngôn ngữ khác nhau.

Mẫu theo dõi ngoại lệ là một cách hữu ích để nhanh chóng xác định và phản hồi các vấn đề sản xuất.

Theo dõi hành vi của người dùng cũng rất quan trọng. Hãy cùng xem cách thực hiện điều đó.

###### Áp dụng mẫu ghi nhật ký kiểm toán

Mục đích của việc ghi nhật ký kiểm toán là ghi lại hành động của từng người dùng. Nhật ký kiểm toán thường được sử dụng để hỗ trợ khách hàng, đảm bảo tuân thủ và phát hiện hành vi đáng ngờ. Mỗi mục nhập nhật ký kiểm toán ghi lại danh tính của người dùng, hành động họ thực hiện và đối tượng kinh doanh. Một ứng dụng thường lưu trữ nhật ký kiểm toán trong bảng cơ sở dữ liệu.

**Mẫu: Ghi nhật ký kiểm tra**

Ghi lại các hành động của người dùng trong cơ sở dữ liệu để hỗ trợ khách hàng, đảm bảo tuân thủ và phát hiện hành vi đáng ngờ. Xem[http://microservices.io/patterns/](http://microservices.io/patterns/observability/audit-logging.html) [khả năng quan sát/kiểm toán-ghi nhật ký.html](http://microservices.io/patterns/observability/audit-logging.html).

Có một số cách khác nhauđể thực hiện ghi nhật ký kiểm tra:

* + - * Thêm mã ghi nhật ký kiểm tra vào logic kinh doanh.
      * Sử dụng lập trình hướng khía cạnh (AOP).
      * Sử dụng nguồn sự kiện. Hãy cùng xem xét từng tùy chọn.

**MỘTDD KIỂM TOÁN MÃ GHI CHÉP CHO LOGIC KINH DOANH**

Tùy chọn đầu tiên và dễ nhất là rải mã ghi nhật ký kiểm toán khắp logic kinh doanh của dịch vụ. Ví dụ, mỗi phương pháp dịch vụ có thể tạo mục nhập nhật ký kiểm toán và lưu vào cơ sở dữ liệu. Nhược điểm của phương pháp này là nó đan xen mã ghi nhật ký kiểm toán và logic kinh doanh, làm giảm khả năng bảo trì. Nhược điểm khác là nó có khả năng dễ xảy ra lỗi vì nó phụ thuộc vào việc nhà phát triển viết mã ghi nhật ký kiểm toán.

**BạnPHƯƠNG HƯỚNG ĐÔNG NAM-LẬP TRÌNH ĐỊNH HƯỚNG**

Tùy chọn thứ hai là sử dụng AOP. Bạn có thể sử dụng một khuôn khổ AOP, chẳng hạn như Spring AOP, để xác định lời khuyên tự động chặn mỗi cuộc gọi phương thức dịch vụ và duy trì mục nhập nhật ký kiểm toán. Đây là một cách tiếp cận đáng tin cậy hơn nhiều, vì nó tự động ghi lại mọi lệnh gọi phương thức dịch vụ. Nhược điểm chính của việc sử dụng AOP là lời khuyên chỉ có quyền truy cập vào tên phương thức và các đối số của nó, do đó có thể khó xác định đối tượng kinh doanh đang được thực hiện và tạo mục nhập nhật ký kiểm toán hướng đến doanh nghiệp.

**BạnSE SOURCING SỰ KIỆN**

Tùy chọn thứ ba và cuối cùng là triển khai logic kinh doanh của bạn bằng cách sử dụng event sourcing. Như đã đề cập trong chương 6, event sourcing tự động cung cấp nhật ký kiểm tra cho các hoạt động tạo và cập nhật. Bạn cần ghi lại danh tính của người dùng trong mỗi sự kiện. Tuy nhiên, một hạn chế khi sử dụng event sourcing là nó không ghi lại các truy vấn. Nếu dịch vụ của bạn phải tạo mục nhật ký cho các truy vấn, thì bạn cũng sẽ phải sử dụng một trong các tùy chọn khác.

#### Phát triển dịch vụ bằng cách sử dụng khung Microservice mẫu

Chương này đã mô tả nhiều mối quan tâm mà một dịch vụ phải triển khai, bao gồm số liệu, báo cáo ngoại lệ cho trình theo dõi ngoại lệ, ghi nhật ký và kiểm tra tình trạng, cấu hình bên ngoài và bảo mật. Hơn nữa, như đã mô tả trong chương 3, một dịch vụ cũng có thể cần xử lý khám phá dịch vụ và triển khai bộ ngắt mạch. Đó không phải là thứ bạn muốn thiết lập từ đầu mỗi khi triển khai một dịch vụ mới. Nếu bạn làm vậy, có khả năng sẽ mất nhiều ngày, thậm chí nhiều tuần, trước khi bạn viết dòng logic kinh doanh đầu tiên của mình.

**Mẫu: Khung máy vi dịch vụ**

Xây dựng các dịch vụ trên một khuôn khổ hoặc tập hợp các khuôn khổ xử lý các mối quan tâm xuyên suốt, chẳng hạn như theo dõi ngoại lệ, ghi nhật ký, kiểm tra tình trạng, cấu hình bên ngoài và theo dõi phân tán. Xem[http://microservices.io/patterns/microservice-](http://microservices.io/patterns/microservice-chassis.html) [khung gầm.html](http://microservices.io/patterns/microservice-chassis.html).

Một cách nhanh hơn nhiều để phát triển dịch vụ là xây dựng dịch vụ của bạn trên một dịch vụ siêu nhỏkhung gầm. Như hình 11.16 cho thấy, khung gầm microservice là một khuôn khổ hoặc một tập hợp các khuôn khổ xử lý những mối quan tâm này. Khi sử dụng khung gầm microservice, bạn sẽ viết ít mã, nếu có, để xử lý những mối quan tâm này.

**Hình 11.16 Khung máy vi dịch vụlà một khuôn khổ xử lý nhiều vấn đề, chẳng hạn như theo dõi ngoại lệ, ghi nhật ký, kiểm tra tình trạng, cấu hình bên ngoài và theo dõi phân tán.**

Dịch vụ

Khung gầm vi dịch vụ

Dịch vụmã số

|  |
| --- |
| Bộ ngắt mạch |
| Phân phối theo dõi |
| Ghi nhật ký |
| Cấu hình bên ngoài. |

|  |
| --- |
| Khám phá dịch vụ |
| Số liệu ứng dụng |
| Kiểm tra sức khỏe |
| ... |

Trong phần này, trước tiên tôi sẽ mô tả khái niệm về khung dịch vụ vi mô và đề xuất một số khung dịch vụ vi mô tuyệt vời. Sau đó, tôi sẽ giới thiệu khái niệm về lưới dịch vụ, tại thời điểm viết bài này đang nổi lên như một giải pháp thay thế hấp dẫn cho việc sử dụng các khung và thư viện.

Trước tiên chúng ta hãy xem xét ý tưởng về khung dịch vụ vi mô.

###### Sử dụng khung máy vi dịch vụ

Khung dịch vụ vi mô là một khuôn khổ hoặc một tập hợp các khuôn khổ xử lý nhiều vấn đề bao gồm:

* + - * Cấu hình bên ngoài
      * Kiểm tra sức khỏe
      * Số liệu ứng dụng
      * Khám phá dịch vụ
* Bộ ngắt mạch
* Phân phối theo dõi

Nó làm giảm đáng kể lượng mã bạn cần viết. Bạn thậm chí có thể không cần viết bất kỳ mã nào. Thay vào đó, bạn cấu hình khung dịch vụ vi mô để phù hợp với yêu cầu của mình. Khung dịch vụ vi mô cho phép bạn tập trung vào việc phát triển logic kinh doanh của dịch vụ.

Ứng dụng FTGO sử dụng Spring Boot và Spring Cloud làm khung gầm dịch vụ vi mô. Spring Boot cung cấp các chức năng như cấu hình bên ngoài. Spring Cloud cung cấp các chức năng như bộ ngắt mạch. Nó cũng triển khai khám phá dịch vụ phía máy khách, mặc dù ứng dụng FTGO dựa vào cơ sở hạ tầng để khám phá dịch vụ. Spring Boot và Spring Cloud không phải là khung gầm dịch vụ vi mô duy nhất. Ví dụ, nếu bạn đang viết dịch vụ trong GoLang, bạn có thể sử dụng Go Kit (<https://github.com/go-kit/kit>) hoặc Micro (<https://github.com/micro/micro>).

Một nhược điểm của việc sử dụng khung vi dịch vụ là bạn cần một khung cho mọi kết hợp ngôn ngữ/nền tảng mà bạn sử dụng để phát triển dịch vụ. May mắn thay, nhiều khả năng là nhiều chức năng được triển khai bởi khung vi dịch vụ sẽ được triển khai bởi cơ sở hạ tầng. Ví dụ, như đã mô tả trong chương 3, nhiều môi trường triển khai xử lý việc khám phá dịch vụ. Hơn nữa, nhiều chức năng liên quan đến mạng của khung vi dịch vụ sẽ được xử lý bởi cái được gọi là lưới dịch vụ, một lớp cơ sở hạ tầng chạy bên ngoài các dịch vụ.

###### Từ khung dịch vụ vi mô đến lưới dịch vụ

Khung dịch vụ vi mô là một cách tốt để triển khai nhiều mối quan tâm cắt ngang khác nhau, chẳng hạn như bộ ngắt mạch. Nhưng một trở ngại khi sử dụng khung dịch vụ vi mô là bạn cần một khung cho mỗi ngôn ngữ lập trình bạn sử dụng. Ví dụ, Spring Boot và Spring Cloud hữu ích nếu bạn là nhà phát triển Java/Spring, nhưng chúng không hữu ích nếu bạn muốn viết dịch vụ dựa trên NodeJS.

**Mẫu: Lưới dịch vụ**

Định tuyến tất cả lưu lượng mạng vào và ra khỏi các dịch vụ thông qua một lớp mạng thực hiện nhiều mối quan tâm khác nhau, bao gồm bộ ngắt mạch, theo dõi phân tán, khám phá dịch vụ, cân bằng tải và định tuyến lưu lượng dựa trên quy tắc. Xem[http://microservices.io/](http://microservices.io/patterns/deployment/service-mesh.html) [mẫu/triển khai/service-mesh.html](http://microservices.io/patterns/deployment/service-mesh.html).

Một giải pháp thay thế mới nổi giúp tránh được vấn đề này là triển khai một số chức năng này bên ngoài dịch vụ trong cái được gọi là lưới dịch vụ. Lưới dịch vụ là cơ sở hạ tầng mạng làm trung gian cho việc giao tiếp giữa một dịch vụ và các dịch vụ khác cũng như các ứng dụng bên ngoài. Như hình 11.17 cho thấy, tất cả lưu lượng mạng vào và ra khỏi một dịch vụ đều đi qua lưới dịch vụ. Nó triển khai nhiều mối quan tâm khác nhau bao gồm bộ ngắt mạch, theo dõi phân tán, khám phá dịch vụ, cân bằng tải và định tuyến lưu lượng dựa trên quy tắc. Lưới dịch vụ cũng có thể bảo mật giao tiếp giữa các quy trình bằng cách sử dụng

**Ít chức năng hơn**



Giao diện lập trình ứng dụng (API)

cổng vào

Đặt hàng

Dịch vụ

Nhà hàng

Dịch vụ

Khung gầm vi dịch vụ

Số liệu ứng dụng

...

Dịch vụ

lưới

**Chức năng được chuyển từ khung dịch vụ vi mô sang lưới dịch vụ**

Kiểm tra sức khỏe

Cấu hình bên ngoài.

Dịch vụ vi môkhung gầm

Phân phối theo dõi

Bộ ngắt mạch

Dịch vụ vi môkhung gầm

Dịch vụ vi môkhung gầm

Phân phối theo dõi

Ghi nhật ký

|  |  |  |  |
| --- | --- | --- | --- |
| Khám phá dịch vụ | |  | |
| Định tuyến giao thông thông minh | | |  |
| Cân bằng tải |  | | |
| Truyền thông an toàn | | | |

Cơ sở hạ tầng triển khai

**Hình 11.17 Tất cả lưu lượng mạng vào và ra khỏi một dịch vụ đều chảy qua lưới dịch vụ. Lưới dịch vụ triển khai nhiều chức năng khác nhau bao gồm bộ ngắt mạch, theo dõi phân tán, khám phá dịch vụ và cân bằng tải. Khung vi dịch vụ triển khai ít chức năng hơn. Nó cũng bảo mậtgiao tiếp giữa các tiến trình bằng cách sử dụng IPC dựa trên TLS giữa các dịch vụ.**

IPC dựa trên TLS giữa các dịch vụ. Do đó, bạn không còn cần phải triển khai những mối quan tâm cụ thể này trong các dịch vụ nữa.

Khi sử dụng lưới dịch vụ, khung dịch vụ vi mô đơn giản hơn nhiều. Nó chỉ cần triển khai các mối quan tâm được tích hợp chặt chẽ với mã ứng dụng, chẳng hạn như cấu hình bên ngoài và kiểm tra tình trạng. Khung dịch vụ vi mô phải hỗ trợ theo dõi phân tán bằng cách truyền bá thông tin theo dõi phân tán, chẳng hạn như tiêu đề chuẩn B3 mà tôi đã thảo luận trước đó trong phần 11.3.3.

**Trạng thái hiện tại của việc triển khai dịch vụ lưới**

Có nhiều triển khai dịch vụ lưới khác nhau, bao gồm:

* Istio ([https://istio.io](https://istio.io/))
* Liên kết ([https://linkerd.io](https://linkerd.io/))
* Ống dẫn ([https://conduit.io](https://conduit.io/))

Tính đến thời điểm viết bài, Linkerd là công nghệ trưởng thành nhất, với Istio và Conduit vẫn đang trong quá trình phát triển tích cực. Để biết thêm thông tin về công nghệ mới thú vị này, hãy xemxem tài liệu hướng dẫn của từng sản phẩm.

Khái niệm lưới dịch vụ là một ý tưởng cực kỳ hứa hẹn. Nó giải phóng nhà phát triển khỏi việc phải giải quyết nhiều mối quan tâm cắt ngang khác nhau. Ngoài ra, khả năng của lưới dịch vụ

route traffic cho phép bạn tách biệt việc triển khai khỏi việc phát hành. Nó cung cấp cho bạn khả năng triển khai phiên bản mới của dịch vụ vào sản xuất nhưng chỉ phát hành cho một số người dùng nhất định, chẳng hạn như người dùng thử nghiệm nội bộ. Chương 12 thảo luận thêm về khái niệm này khi mô tả cách triển khai dịch vụ bằng Kubernetes.

#### Bản tóm tắt

* + - * Điều quan trọng là dịch vụ phải triển khai được các yêu cầu chức năng của mình, nhưng cũng phải an toàn, có thể cấu hình và quan sát được.
      * Nhiều khía cạnh của bảo mật trong kiến ​​trúc vi dịch vụ không khác gì so với kiến ​​trúc đơn khối. Nhưng có một số khía cạnh của bảo mật ứng dụng cần phải khác biệt, bao gồm cách danh tính người dùng được truyền giữa cổng API và các dịch vụ và ai chịu trách nhiệm xác thực và ủy quyền. Một cách tiếp cận thường được sử dụng là để cổng API xác thực máy khách. Cổng API bao gồm một mã thông báo trong suốt, chẳng hạn như JWT, trong mỗi yêu cầu đến một dịch vụ. Mã thông báo chứa danh tính của người đứng đầu và vai trò của họ. Các dịch vụ sử dụng thông tin trong mã thông báo để ủy quyền truy cập vào tài nguyên. OAuth 2.0 là nền tảng tốt cho bảo mật trong kiến ​​trúc vi dịch vụ.
      * Một dịch vụ thường sử dụng một hoặc nhiều dịch vụ bên ngoài, chẳng hạn như các nhà môi giới tin nhắn và cơ sở dữ liệu. Vị trí mạng và thông tin xác thực của mỗi dịch vụ bên ngoài thường phụ thuộc vào môi trường mà dịch vụ đang chạy. Bạn phải áp dụng mẫu cấu hình Externalized và triển khai một cơ chế cung cấp cho dịch vụ các thuộc tính cấu hình khi chạy. Một cách tiếp cận thường được sử dụng là cơ sở hạ tầng triển khai cung cấp các thuộc tính đó thông qua các biến môi trường của hệ điều hành hoặc tệp thuộc tính khi tạo phiên bản dịch vụ. Một tùy chọn khác là phiên bản dịch vụ truy xuất cấu hình của nó từ máy chủ thuộc tính cấu hình.
      * Hoạt động và nhà phát triển chia sẻ trách nhiệm triển khai các mẫu khả năng quan sát. Hoạt động chịu trách nhiệm về cơ sở hạ tầng khả năng quan sát, chẳng hạn như máy chủ xử lý tổng hợp nhật ký, số liệu, theo dõi ngoại lệ và theo dõi phân tán. Nhà phát triển chịu trách nhiệm đảm bảo rằng các dịch vụ của họ có thể quan sát được. Các dịch vụ phải có điểm cuối API kiểm tra tình trạng, tạo mục nhật ký, thu thập và hiển thị số liệu, báo cáo ngoại lệ cho dịch vụ theo dõi ngoại lệ và triển khai theo dõi phân tán.
      * Để đơn giản hóa và tăng tốc quá trình phát triển, bạn nên phát triển các dịch vụ trên một khung dịch vụ vi mô. Khung dịch vụ vi mô là một khung hoặc một tập hợp các khung xử lý nhiều mối quan tâm cắt ngang, bao gồm cả những mối quan tâm được mô tả trong chương này. Tuy nhiên, theo thời gian, nhiều khả năng là nhiều chức năng liên quan đến mạng của khung dịch vụ vi mô sẽ di chuyển vào một lưới dịch vụ, một lớp phần mềm cơ sở hạ tầng mà tất cả lưu lượng mạng của dịch vụ đều đi qua.

*Triển khai các dịch vụ vi mô*

***Chương này bao gồm***

* Bốn mô hình triển khai chính,họ làm việc như thế nào,và lợi ích cũng như hạn chế của chúng:
  + Định dạng đóng gói theo ngôn ngữ cụ thể
  + Triển khai dịch vụ dưới dạng VM
  + Triển khai dịch vụ dưới dạng container
  + Triển khai không cần máy chủ
* Triển khai dịch vụ với Kubernetes
* Sử dụng lưới dịch vụ để tách biệt triển khai khỏi phát hành
* Triển khai dịch vụ với AWS Lambda
* Chọn một mô hình triển khai

Mary và nhóm của cô ấy tại FTGO gần như đã hoàn thành việc viết dịch vụ đầu tiên của họ. Mặc dùnó vẫn chưa hoàn thiện tính năng, nó đang chạy trên máy tính xách tay của nhà phát triển và máy chủ Jenkins CI. Nhưng thế vẫn chưa đủ. Phần mềm không có giá trị gì đối với FTGO cho đến khi nó chạy trong sản xuất và có sẵn cho người dùng. FTGO cần triển khai dịch vụ của họ vào sản xuất.

**383**

*Triển khai*là sự kết hợp của hai khái niệm có liên quan với nhau: quy trình và kiến ​​trúc. Quy trình triển khai bao gồm các bước mà con người phải thực hiện—nhà phát triển và vận hành—để đưa phần mềm vào sản xuất. Kiến trúc triển khai xác định cấu trúc của môi trường mà phần mềm đó chạy. Cả hai khía cạnh của việc triển khai đã thay đổi hoàn toàn kể từ khi tôi bắt đầu phát triển các ứng dụng Enterprise Java vào cuối những năm 1990. Quy trình thủ công của các nhà phát triển đưa mã qua tường để sản xuất đã trở nên tự động hóa cao. Như hình 12.1 cho thấy, môi trường sản xuất vật lý đã được thay thế bằng cơ sở hạ tầng điện toán ngày càng nhẹ và phù du.

Nhẹ,tạm thời, tự động

Ứng dụng

Thuộc vật chất

máy móc

|  |
| --- |
| Ứng dụng |
| Không có máy chủthời gian chạy |
| Cơ sở hạ tầng ẩn |
| Máy vật lý |

|  |
| --- |
| Ứng dụng |
| Thùng chứathời gian chạy |
| Máy ảo |
| Máy vật lý |

|  |
| --- |
| Ứng dụng |
| Máy ảo |
| Máy vật lý |

hạng nặng,vĩnh viễn, thủ công

Những năm 1990200620132014

Thời gian

AWS EC2

phát hành

Docker ban đầugiải phóng

AWS Lambdagiới thiệu

**Hình 12.1 Các máy móc vật lý nặng và tồn tại lâu dài đã được trừu tượng hóabằng những công nghệ ngày càng nhẹ và phù du.**

Quay trở lại những năm 1990, nếu bạn muốn triển khai một ứng dụng vào sản xuất, bước đầu tiên là ném ứng dụng của bạn cùng với một bộ hướng dẫn vận hành qua bức tường đến bộ phận vận hành. Ví dụ, bạn có thể gửi phiếu yêu cầu sự cố yêu cầu bộ phận vận hành triển khai ứng dụng. Bất cứ điều gì xảy ra tiếp theo đều hoàn toàn là trách nhiệm của bộ phận vận hành, trừ khi họ gặp phải sự cố mà họ cần bạn giúp khắc phục. Thông thường, bộ phận vận hành đã mua và cài đặt các máy chủ ứng dụng đắt tiền và nặng như WebLogic hoặc WebSphere. Sau đó, họ sẽ đăng nhập vào bảng điều khiển máy chủ ứng dụng và triển khai các ứng dụng của bạn. Họ sẽ yêu thương chăm sóc những máy đó, như thể chúng là thú cưng, cài đặt các bản vá và cập nhật phần mềm.

Vào giữa những năm 2000, các máy chủ ứng dụng đắt tiền đã được thay thế bằng các container web nhẹ, nguồn mở như Apache Tomcat và Jetty. Bạn vẫn có thể chạy nhiều ứng dụng trên mỗi container web, nhưng việc có một ứng dụng cho mỗi container web trở nên khả thi. Ngoài ra, máy ảo bắt đầu thay thế máy vật lý.

**385**

Nhưng máy móc vẫn được đối xửnhư những vật nuôi được yêu quý, và việc triển khai vẫn là một quá trình cơ bản thủ công.

Ngày nay, quy trình triển khai đã hoàn toàn khác. Thay vì giao mã cho một nhóm sản xuất riêng, việc áp dụng DevOps có nghĩa là nhóm phát triển cũng chịu trách nhiệm triển khai ứng dụng hoặc dịch vụ của họ. Trong một số tổ chức, hoạt động cung cấp cho các nhà phát triển một bảng điều khiển để triển khai mã của họ. Hoặc, tốt hơn nữa, sau khi các bài kiểm tra vượt qua, đường ống triển khai sẽ tự động triển khai mã vào sản xuất.

Các tài nguyên điện toán được sử dụng trong môi trường sản xuất cũng đã thay đổi hoàn toàn với các máy vật lý được trừu tượng hóa. Các máy ảo chạy trên đám mây tự động hóa cao, chẳng hạn như AWS, đã thay thế các máy vật lý và máy ảo giống như vật nuôi tồn tại lâu dài. Các máy ảo ngày nay là bất biến. Chúng được coi như gia súc dùng một lần thay vì vật nuôi và bị loại bỏ và tạo lại thay vì được cấu hình lại. Các container, một lớp trừu tượng nhẹ hơn nữa của các máy ảo, là một cách triển khai ứng dụng ngày càng phổ biến. Bạn cũng có thể sử dụng một nền tảng triển khai không có máy chủ nhẹ hơn nữa, chẳng hạn như AWS Lambda, cho nhiều trường hợp sử dụng.

Không phải ngẫu nhiên mà sự phát triển của các quy trình và kiến ​​trúc triển khai đãtrùng hợp với việc áp dụng ngày càng tăng của kiến ​​trúc dịch vụ vi mô. Một ứng dụng có thể có hàng chục hoặc hàng trăm dịch vụ được viết bằng nhiều ngôn ngữ và khung khác nhau. Vì mỗi dịch vụ là một ứng dụng nhỏ, điều đó có nghĩa là bạn có hàng chục hoặc hàng trăm ứng dụng đang được sản xuất. Ví dụ, việc quản trị viên hệ thống tự tay cấu hình máy chủ và dịch vụ không còn khả thi nữa. Nếu bạn muốn triển khai các dịch vụ vi mô ở quy mô lớn, bạn cần một quy trình triển khai và cơ sở hạ tầng được tự động hóa cao.

Hình 12.2 cho thấy góc nhìn cấp cao của môi trường sản xuất. Môi trường sản xuất cho phép các nhà phát triển cấu hình và quản lý các dịch vụ của họ, triển khai

Dịch vụ

MỘT

Quan sát và

khắc phục sự cốdịch vụ

Dịch vụ

B

Dịch vụ

C

Tiêu thụ

dịch vụ

Nhà phát triển

Cấu hình và quản lý dịch vụ

Dịch vụ

D

Cập nhật

dịch vụ

Triển khaiđường ống

đang

Lộ trình

Quản lý dịch vụ thời gian chạy

Giao diện quản lý dịch vụ

Giám sát

Bảng điều khiển

Cảnh báo



Người sử dụng

**Hình 12.2 Một góc nhìn đơn giản về môi trường sản xuất. Nó cung cấp bốn khả năng chính: quản lý dịch vụ cho phép các nhà phát triển triển khai và quản lý các dịch vụ của họ, quản lý thời gian chạy đảm bảo rằng các dịch vụ đang chạy, giám sát trực quan hóa hành vi dịch vụ và tạo cảnh báo,và yêu cầu định tuyến các yêu cầu từ người dùng đến các dịch vụ.**

đường ống để triển khai các phiên bản dịch vụ mới và người dùng truy cập vào chức năng được triển khai bởi các dịch vụ đó.

Môi trường sản xuất phải triển khai bốn khả năng chính:

* *Giao diện quản lý dịch vụ*—Cho phép các nhà phát triển tạo, cập nhật và cấu hình dịch vụ. Lý tưởng nhất là giao diện này là REST API được gọi bằng các công cụ triển khai dòng lệnh và GUI.
* *Quản lý dịch vụ thời gian chạy*—Cố gắng đảm bảo rằng số lượng dịch vụ mong muốn đang chạy mọi lúc. Nếu một dịch vụ bị sập hoặc không thể xử lý yêu cầu, môi trường sản xuất phải khởi động lại dịch vụ đó. Nếu một máy bị sập, môi trường sản xuất phải khởi động lại các dịch vụ đó trên một máy khác.
* *Giám sát*—Cung cấp cho các nhà phát triển thông tin chi tiết về những gì dịch vụ của họ đang làm, bao gồm các tệp nhật ký và số liệu. Nếu có vấn đề, môi trường sản xuất phải cảnh báo cho các nhà phát triển. Chương 11 mô tả giám sát, còn được gọi là khả năng quan sát.
* *Yêu cầu định tuyến*—Định tuyến yêu cầu từ người dùng đến các dịch vụ. Trong chương này, tôi thảo luận về bốn tùy chọn triển khai chính:
* Triển khai dịch vụdưới dạng các gói dành riêng cho từng ngôn ngữ, chẳng hạn như tệp Java JAR hoặc WAR.

Thật đáng để khám phá lựa chọn này, vì mặc dù tôi khuyên bạn nên sử dụng một trong các lựa chọn khác, nhưng nhược điểm của nó lại thúc đẩy bạn sử dụng các lựa chọn khác.

* Triển khai các dịch vụ dưới dạng máy ảo, giúp đơn giản hóa việc triển khai bằng cách đóng gói dịch vụ dưới dạng ảnh máy ảo bao gồm ngăn xếp công nghệ của dịch vụ.
* Triển khai các dịch vụ dưới dạng container, nhẹ hơn so với ảomáy móc. Tôi chỉ cách triển khai ứng dụng FTGODịch vụ nhà hàngsử dụng Kubernetes, một nền tảng điều phối Docker phổ biến.
* Triển khai các dịch vụ bằng cách sử dụng triển khai không có máy chủ, thậm chí còn hiện đại hơncontainer. Chúng ta sẽ xem cách triển khaiDịch vụ nhà hàngsử dụng AWS Lambda,một nền tảng không có máy chủ phổ biến.

Trước tiên, chúng ta hãy xem cách triển khai các dịch vụ dưới dạng các gói ngôn ngữ cụ thể.

#### Triển khai các dịch vụ bằng Ngôn ngữ cụ thểmẫu định dạng bao bì

Hãy tưởng tượng rằng bạn muốn triển khai Dịch vụ nhà hàng của ứng dụng FTGO, đây là ứng dụng Java dựa trên Spring Boot. Một cách để triển khai dịch vụ này là sử dụng Dịch vụ như một mẫu gói dành riêng cho ngôn ngữ. Khi sử dụng mẫu này, những gì được triển khai trong sản xuất và những gì được quản lý bởi thời gian chạy dịch vụ là một dịch vụ trong gói dành riêng cho ngôn ngữ của nó. Trong trường hợp của Dịch vụ nhà hàng, đó là tệp JAR có thể thực thi hoặc tệp WAR. Đối với các ngôn ngữ khác, chẳng hạn như NodeJS, dịch vụ là một thư mục chứa mã nguồn và các mô-đun. Đối với một số ngôn ngữ, chẳng hạn như GoLang, dịch vụ là một tệp thực thi dành riêng cho hệ điều hành.

**Mẫu: Định dạng đóng gói theo ngôn ngữ cụ thể**

Triển khai một gói ngôn ngữ cụ thể vào sản xuất. Xem[http://microservices.io/](http://microservices.io/patterns/deployment/language-specific-packaging.html) [mẫu/triển khai/gói ngôn ngữ cụ thể.html](http://microservices.io/patterns/deployment/language-specific-packaging.html).

Để triển khai Restaurant Service trên một máy, trước tiên bạn sẽ cài đặt runtime cần thiết, trong trường hợp này là JDK. Nếu đó là tệp WAR, bạn cũng cần cài đặt một web container như Apache Tomcat. Sau khi cấu hình máy, bạn sao chép gói vào máy và khởi động dịch vụ. Mỗi phiên bản dịch vụ chạy dưới dạng một quy trình JVM.

Lý tưởng nhất là bạn đã thiết lập đường ống triển khai của mình để tự động triển khai dịch vụ lên môi trường sản xuất, như thể hiện trong hình 12.3. Đường ống triển khai xây dựng tệp JAR hoặc tệp WAR có thể thực thi. Sau đó, nó sẽ gọi giao diện quản lý dịch vụ của môi trường sản xuất để triển khai phiên bản mới.

Xây dựngthời gian

Thời gian chạy

Dịch vụ thể hiện

Máy ảo JVM

quá trình

Máy ảo JVM

quá trình

Máy ảo JVM

quá trình

Máy móc

Máy móc

Sản xuất

Quản lý thời gian chạy dịch vụ

JDK/JRE

JDK/JRE

**Hình 12.3 Đường ống triển khai xây dựng một tệp JAR có thể thực thi và triển khai tệp đó vào sản xuất. Trong sản xuất, mỗi phiên bản dịch vụ là một JVM chạy trên máy đã cài đặt JDK hoặc JRE.**

Dịch vụmã số

Triển khaiđường ống

Tệp JAR/WAR có thể thực thi

Một thể hiện dịch vụ thường là một quy trình đơn lẻ nhưng đôi khi có thể là một nhóm các quy trình. Ví dụ, một thể hiện dịch vụ Java là một quy trình chạy JVM. Một dịch vụ NodeJS có thể tạo ra nhiều quy trình công nhân để xử lý các yêu cầu đồng thời. Một số ngôn ngữ hỗ trợ triển khai nhiều thể hiện dịch vụ trong cùng một quy trình.

Đôi khi bạn có thể triển khai một phiên bản dịch vụ duy nhất trên một máy, trong khi vẫn giữ nguyên tùy chọn triển khai nhiều phiên bản dịch vụ trên cùng một máy. Ví dụ, như hình 12.4 cho thấy, bạn có thể chạy nhiều JVM trên một máy. Mỗi JVM chạy một phiên bản dịch vụ duy nhất.

**Hình 12.4 Triển khai nhiều phiên bản dịch vụ trên cùng một máy. Chúng có thể là các phiên bản của cùng một dịch vụ hoặc các phiên bản của các dịch vụ khác nhau. Chi phí chung của hệ điều hành được chia sẻ giữa các phiên bản dịch vụ. Mỗi phiên bản dịch vụlà một quá trình riêng biệt, do đó có sự cô lập giữa chúng.**

Quá trình

Quá trình

Quá trình

Máy vật lý hoặc máy ảo

|  |
| --- |
| Dịch vụ thể hiện A |
| Mèo đực |
| Máy ảo JVM |

|  |
| --- |
| Dịch vụ thể hiện B |
| Mèo đực |
| Máy ảo JVM |

|  |
| --- |
| Phiên bản dịch vụ ... |
| Mèo đực |
| Máy ảo JVM |

Một số ngôn ngữ cũng cho phép bạn chạy nhiều phiên bản dịch vụ trong một quy trình duy nhất. Ví dụ, như hình 12.5 cho thấy, bạn có thể chạy nhiều dịch vụ Java trên một Apache Tomcat duy nhất.

**Hình 12.5 Triển khai nhiều phiên bản dịch vụ trên cùng một web container hoặc máy chủ ứng dụng. Chúng có thể là các phiên bản của cùng một dịch vụ hoặc các phiên bản của các dịch vụ khác nhau. Chi phí chung của hệ điều hành và thời gian chạy được chia sẻ giữa tất cả các phiên bản dịch vụ. Nhưng vì các phiên bản dịch vụ nằm trong cùng một quy trình nên không có sự cô lập giữa chúng.**

Quá trình

Máy vật lý hoặc máy ảo

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dịch vụ thể hiện A |  | Dịch vụ thể hiện B |  | Phiên bản dịch vụ ... |
| Mèo đực | | | | |
| Máy ảo JVM | | | | |

Cách tiếp cận này thường được sử dụng khi triển khai các ứng dụng trên các máy chủ ứng dụng truyền thống đắt tiền và nặng, chẳng hạn như WebLogic và WebSphere. Bạn cũng có thể đóng gói các dịch vụ dưới dạng các gói OSGI và chạy nhiều phiên bản dịch vụ trong mỗi vùng chứa OSGI.

Dịch vụ như một mẫu gói ngôn ngữ cụ thể có cả lợi ích và bất lợi. Trước tiên, hãy xem xét các lợi ích.

###### Lợi ích của Dịch vụ như một mẫu gói ngôn ngữ cụ thể

Dịch vụ dưới dạng mô hình gói dành riêng cho ngôn ngữ có một số lợi ích:

* + - * Triển khai nhanh chóng
      * Sử dụng tài nguyên hiệu quả, đặc biệt khi chạy nhiều phiên bản trên cùng một máy hoặc trong cùng một quy trình

Chúng ta hãy cùng xem xét từng cái một.

**FTRIỂN KHAI AST**

Một lợi ích chính của mô hình này là việc triển khai một phiên bản dịch vụ tương đối nhanh: bạn sao chép dịch vụ vào máy chủ và khởi động nó. Nếu dịch vụ được viết bằng Java, bạn sao chép tệp JAR hoặc WAR. Đối với các ngôn ngữ khác, chẳng hạn như NodeJS hoặc Ruby, bạn sao chép mã nguồn. Trong cả hai trường hợp, số byte được sao chép qua mạng tương đối nhỏ.

Ngoài ra, việc khởi động một dịch vụ hiếm khi tốn thời gian. Nếu dịch vụ là một quy trình riêng, bạn sẽ khởi động nó. Nếu không, nếu dịch vụ là một trong nhiều phiên bản đang chạy trong cùng một quy trình chứa, bạn sẽ triển khai động vào trong chứa hoặc khởi động lại chứa. Do không có chi phí chung, việc khởi động một dịch vụ thường diễn ra nhanh.

**ESỬ DỤNG NGUỒN LỰC HIỆU QUẢ**

Một lợi ích lớn khác của mô hình này là nó sử dụng tài nguyên tương đối hiệu quả. Nhiều phiên bản dịch vụ chia sẻ máy và hệ điều hành của nó. Thậm chí còn hiệu quả hơn nếu nhiều phiên bản dịch vụ chạy trong cùng một quy trình. Ví dụ, nhiều ứng dụng web có thể chia sẻ cùng một máy chủ Apache Tomcat và JVM.

###### Nhược điểm của Dịch vụ như một mẫu gói ngôn ngữ cụ thể

Mặc dù có sức hấp dẫn, nhưng Dịch vụ như một mô hình gói ngôn ngữ cụ thể có một số nhược điểm đáng kể:

* + - * Thiếu sự đóng góicủa ngăn xếp công nghệ.
      * Không có khả năng hạn chếtài nguyên được sử dụng bởi một phiên bản dịch vụ.
      * Thiếu tính cô lập khi chạy nhiều phiên bản dịch vụ trên cùng một máy.
      * Việc tự động xác định vị trí đặt các phiên bản dịch vụ là một thách thức. Hãy cùng xem xét từng nhược điểm.

**LACK CỦA VIỆC ĐÓNG GÓI CỦA CÔNG NGHỆ NGĂN NGẮN**

Nhóm vận hành phải biết các chi tiết cụ thể về cách triển khai từng dịch vụ. Mỗi dịch vụ cần một phiên bản thời gian chạy cụ thể. Ví dụ, một ứng dụng web Java cần các phiên bản cụ thể của Apache Tomcat và JDK. Nhóm vận hành phải cài đặt đúng phiên bản của từng gói phần mềm bắt buộc.

Tệ hơn nữa, các dịch vụ có thể được viết bằng nhiều ngôn ngữ và khung khác nhau. Chúng cũng có thể được viết bằng nhiều phiên bản của các ngôn ngữ và khung đó. Do đó, nhóm phát triển phải chia sẻ nhiều chi tiết với các hoạt động. Sự phức tạp này làm tăng nguy cơ xảy ra lỗi trong quá trình triển khai. Ví dụ, một máy có thể có phiên bản thời gian chạy ngôn ngữ không đúng.

**NO KHẢ NĂNG HẠN CHẾ CÁC NGUỒN LỰC ĐƯỢC TIÊU THỤ BỞI MỘT THỂ HIỆN DỊCH VỤ**

Một nhược điểm khác là bạn không thể hạn chế tài nguyên được sử dụng bởi một phiên bản dịch vụ. Một quy trình có khả năng sử dụng hết CPU hoặc bộ nhớ của máy, khiến các phiên bản dịch vụ khác và hệ điều hành thiếu tài nguyên. Ví dụ, điều này có thể xảy ra do lỗi.

**LXÁC NHẬN CÁCH LY KHI CHẠY NHIỀU TRƯỜNG HỢP DỊCH VỤ TRÊN CÙNG MỘT MÁY**

Vấn đề thậm chí còn tệ hơn khi chạy nhiều phiên bản trên cùng một máy. Việc thiếu sự cô lập có nghĩa là một phiên bản dịch vụ hoạt động không bình thường có thể ảnh hưởng đến các phiên bản dịch vụ khác. Do đó, ứng dụng có nguy cơ không đáng tin cậy, đặc biệt là khi chạy nhiều phiên bản dịch vụ trên cùng một máy.

**MỘTViệc tự động xác định nơi đặt các phiên bản dịch vụ là một thách thức**

Một thách thức khác khi chạy nhiều phiên bản dịch vụ trên cùng một máy là xác định vị trí của các phiên bản dịch vụ. Mỗi máy có một tập hợp tài nguyên cố định, CPU, bộ nhớ, v.v. và mỗi phiên bản dịch vụ cần một lượng tài nguyên nhất định. Điều quan trọng là phải chỉ định các phiên bản dịch vụ cho các máy theo cách sử dụng hiệu quả các máy mà không làm quá tải chúng. Như tôi đã giải thích ngắn gọn, các đám mây dựa trên VM và các khuôn khổ điều phối container xử lý việc này tự động. Khi triển khai các dịch vụ gốc, có khả năng bạn sẽ cần phải quyết định vị trí theo cách thủ công.

Như bạn có thể thấy, mặc dù quen thuộc, Service như một mẫu gói ngôn ngữ cụ thể có một số nhược điểm đáng kể. Bạn hiếm khi nên sử dụng cách tiếp cận này, ngoại trừ khi hiệu quả quan trọng hơn tất cả các mối quan tâm khác.

Bây giờ chúng ta hãy xem xét những cách triển khai dịch vụ hiện đại giúp tránh được những vấn đề này.

#### Triển khai các dịch vụ bằng cách sử dụng Dịch vụ như một máy chủ ảomẫu máy

Một lần nữa, hãy tưởng tượng bạn muốn triển khai FTGO Restaurant Service, ngoại trừ lần này là trên AWS EC2. Một tùy chọn là tạo và cấu hình một phiên bản EC2 và sao chép tệp thực thi hoặc tệp WAR vào đó. Mặc dù bạn sẽ nhận được một số lợi ích khi sử dụng đám mây, nhưng cách tiếp cận này gặp phải những nhược điểm được mô tả trong phần trước. Một cách tiếp cận tốt hơn, hiện đại hơn là đóng gói dịch vụ dưới dạng Amazon Machine Image (AMI), như thể hiện trong hình 12.6. Mỗi phiên bản dịch vụ là một phiên bản EC2 được tạo từ AMI đó. Các phiên bản EC2 thường được quản lý bởi nhóm AWS Auto Scaling, nhóm này cố gắng đảm bảo rằng số lượng phiên bản khỏe mạnh mong muốn luôn chạy.

**Mẫu: Triển khai dịch vụ dưới dạng VM**

Triển khai các dịch vụ được đóng gói dưới dạng hình ảnh VM vào sản xuất. Mỗi phiên bản dịch vụ là một VM. Xem<http://microservices.io/patterns/deployment/service-per-vm.html>.

Hình ảnh máy ảo được xây dựng bởi đường ống triển khai của dịch vụ. Đường ống triển khai, như hình 12.6 cho thấy, chạy trình xây dựng hình ảnh VM để tạo hình ảnh VM chứa mã dịch vụ và bất kỳ phần mềm nào cần thiết để chạy nó. Ví dụ, trình xây dựng VM cho dịch vụ FTGO cài đặt JDK và JAR có thể thực thi của dịch vụ. Trình xây dựng hình ảnh VM cấu hình máy hình ảnh VM để chạy ứng dụng khi VM khởi động, sử dụng hệ thống init của Linux, chẳng hạn như upstart.

***Triển khai dịch vụ bằng cách sử dụng Dịch vụ như một máy ảomẫu* 391**

Xây dựngthời gian

Dịch vụmã số

Ví dụ EC2

Dịch vụ

|  |  |  |
| --- | --- | --- |
|  |  | Tạo ra |
| Hình ảnh VMngười xây dựng |
|  |
| Đường ống triển khai | | |

Thời gian chạy

Yêu cầu

Nhóm tự động điều chỉnh

AMI (VM

hình ảnh)

Đã triển khaiBẰNG

Tải trọng đàn hồicân bằng

Ví dụ EC2

Dịch vụ

**Hình 12.6 Đường ống triển khai đóng gói một dịch vụ dưới dạng ảnh máy ảo, chẳng hạn như EC2 AMI, chứa mọi thứ cần thiết để chạy dịch vụ, bao gồm cả thời gian chạy ngôn ngữ. Khi chạy, mỗi phiên bản dịch vụ là một VM, chẳng hạn như phiên bản EC2, được khởi tạo từ ảnh đó. Một EC2 ElasticBộ cân bằng tải định tuyến các yêu cầu đến các phiên bản.**

Ví dụ EC2

Dịch vụ

Có nhiều công cụ khác nhau mà đường ống triển khai của bạn có thể sử dụng để xây dựng hình ảnh VM. Một công cụ ban đầu để tạo EC2 AMI là Aminator, do Netflix tạo ra, công ty này đã sử dụng công cụ này để triển khai dịch vụ phát trực tuyến video của mình trên AWS ([https://github.com/Netflix/](https://github.com/Netflix/aminator) [máy amin hóa](https://github.com/Netflix/aminator)). Một trình xây dựng hình ảnh VM hiện đại hơn là Packer, không giống như Aminator, hỗ trợ nhiều công nghệ ảo hóa khác nhau, bao gồm EC2, Digital Ocean, Virtual Box và VMware ([www.packer.io](http://www.packer.io/)). Để sử dụng Packer để tạo AMI, bạn phải viết một tệp cấu hình chỉ định hình ảnh cơ sở và một tập hợp các trình cung cấp cài đặt phần mềm và cấu hình AMI.

**Giới thiệu về Elastic Beanstalk**

Elastic Beanstalk, do AWS cung cấp, là một cách dễ dàng để triển khai dịch vụ của bạn bằng VM. Bạn tải mã của mình lên, chẳng hạn như tệp WAR, và Elastic Beanstalk triển khai nó dưới dạng một hoặc nhiều phiên bản EC2 được cân bằng tải và quản lý. Elastic Beanstalk có lẽ không được ưa chuộng như Kubernetes, nhưng đây là một cách dễ dàng để triển khai ứng dụng dựa trên dịch vụ vi mô trên EC2.

Điều thú vị là Elastic Beanstalk kết hợp các yếu tố của ba mô hình triển khaiđược mô tả trong chương này. Nó hỗ trợ một số định dạng đóng gói cho một số ngôn ngữ, bao gồm Java, Ruby và .NET. Nó triển khai ứng dụng dưới dạng VM, nhưng thay vì xây dựng AMI, nó sử dụng một hình ảnh cơ sở cài đặt ứng dụng khi khởi động.

***(tiếp theo)***

Elastic Beanstalk cũng có thể triển khai các container Docker. Mỗi phiên bản EC2 chạy một bộ sưu tập gồm một hoặc nhiều container. Không giống như một khuôn khổ dàn dựng Docker, được đề cập sau trong chương này, đơn vị mở rộng quy mô là phiên bản EC2 chứ không phải là một container.

Hãy cùng xem xét những lợi ích và hạn chế của việc sử dụng phương pháp này.

###### Lợi ích của việc triển khai dịch vụ dưới dạng VM

Mô hình Dịch vụ dưới dạng máy ảo có một số lợi ích:

* + - * Hình ảnh VM bao gồm ngăn xếp công nghệ.
      * Các trường hợp dịch vụ bị cô lập.
      * Sử dụng cơ sở hạ tầng đám mây trưởng thành. Chúng ta hãy xem xét từng cơ sở.

**TANH TAMáy ảoHÌNH ẢNH BAO GỒM CÔNG NGHỆ**

Một lợi ích quan trọng của mẫu này là hình ảnh VM chứa dịch vụ và tất cả các phụ thuộc của nó. Nó loại bỏ yêu cầu dễ xảy ra lỗi là phải cài đặt và thiết lập đúng phần mềm mà dịch vụ cần để chạy. Sau khi dịch vụ được đóng gói dưới dạng máy ảo, nó trở thành hộp đen đóng gói ngăn xếp công nghệ của dịch vụ của bạn. Hình ảnh VM có thể được triển khai ở bất kỳ đâu mà không cần sửa đổi. API để triển khai dịch vụ trở thành API quản lý VM. Việc triển khai trở nên đơn giản hơn nhiều và đáng tin cậy hơn.

**SCÁC TRƯỜNG HỢP DỊCH VỤ ĐƯỢC CÁCH RIÊNG BIỆT**

Một lợi ích chính của máy ảo là mỗi phiên bản dịch vụ chạy hoàn toàn biệt lập. Xét cho cùng, đó là một trong những mục tiêu chính của công nghệ máy ảo. Mỗi máy ảo có một lượng CPU và bộ nhớ cố định và không thể lấy cắp tài nguyên từ các dịch vụ khác.

**BạnĐÁM MÂY TRƯỞNG THÀNH SESCƠ SỞ HẠ TẦNG**

Một lợi ích khác của việc triển khai các dịch vụ vi mô của bạn dưới dạng máy ảo là bạn có thể tận dụng cơ sở hạ tầng đám mây tự động hóa cao, trưởng thành. Các đám mây công cộng như AWS cố gắng lên lịch cho các máy ảo trên các máy vật lý theo cách tránh làm quá tải máy. Chúng cũng cung cấp các tính năng có giá trị như cân bằng tải lưu lượng trên các máy ảo và tự động mở rộng quy mô.

###### Những hạn chế của việc triển khai dịch vụ dưới dạng VM

Mô hình Dịch vụ dưới dạng VM cũng có một số nhược điểm:

* + - * Sử dụng tài nguyên kém hiệu quả
      * Tương đốitriển khai chậm
      * Chi phí quản lý hệ thống Chúng ta hãy cùng xem xét từng nhược điểm.

**LESS-SỬ DỤNG HIỆU QUẢ NGUỒN LỰC**

Mỗi phiên bản dịch vụ có chi phí chung của toàn bộ máy ảo, bao gồm cả hệ điều hành của nó. Hơn nữa, một máy ảo IaaS công cộng thông thường cung cấp một tập hợp giới hạn các kích thước VM, do đó VM có thể sẽ không được sử dụng hết công suất. Điều này ít có khả năng là vấn đề đối với các dịch vụ dựa trên Java vì chúng tương đối nặng. Nhưng mô hình này có thể là một cách không hiệu quả để triển khai các dịch vụ NodeJS và GoLang nhẹ.

**RTRIỂN KHAI TƯƠNG ĐỐI CHẬM**

Việc xây dựng một hình ảnh VM thường mất một số phút do kích thước của VM. Có rất nhiều bit cần được di chuyển qua mạng. Ngoài ra, việc khởi tạo một VM từ một hình ảnh VM tốn thời gian vì, một lần nữa, lượng dữ liệu phải được di chuyển qua mạng. Hệ điều hành chạy bên trong VM cũng mất một thời gian để khởi động, mặc dù chậm là một thuật ngữ tương đối. Quá trình này, có thể mất vài phút, nhanh hơn nhiều so với quá trình triển khai truyền thống. Nhưng nó chậm hơn nhiều so với các mẫu triển khai nhẹ hơn mà bạn sẽ sớm đọc về.

**SQUẢN TRỊ HỆ THỐNGTRÊN KHÔNG**

Bạn chịu trách nhiệm vá hệ điều hành và thời gian chạy. Quản trị hệ thống có vẻ như là điều không thể tránh khỏi khi triển khai phần mềm, nhưng sau đó trong phần 12.5, tôi sẽ mô tảtriển khai không cần máy chủ, giúp loại bỏ loại quản trị hệ thống này.

Bây giờ chúng ta hãy xem xét một cách thay thế để triển khai các dịch vụ vi mô nhẹ hơn nhưng vẫn có nhiều lợi ích của máy ảo.

#### Triển khai các dịch vụ bằng cách sử dụng Dịch vụ nhưmột mẫu container

Container là cơ chế triển khai hiện đại và nhẹ hơn. Chúng là cơ chế ảo hóa cấp hệ điều hành. Một container, như hình 12.7 cho thấy, thường bao gồm một nhưng đôi khi là nhiều quy trình chạy trong hộp cát, giúp cô lập nó khỏi các container khác. Ví dụ, một container chạy dịch vụ Java thường bao gồm quy trình JVM.

Theo quan điểm của một tiến trình chạy trong một container, thì nó giống như đang chạy trên máy riêng của nó. Nó thường có địa chỉ IP riêng, giúp loại bỏ xung đột cổng. Ví dụ, tất cả các tiến trình Java đều có thể lắng nghe trên cổng 8080. Mỗi container cũng có hệ thống tệp gốc riêng. Thời gian chạy container sử dụng các cơ chế của hệ điều hành để cô lập các container với nhau. Ví dụ phổ biến nhất về thời gian chạy container là Docker, mặc dù có những ví dụ khác, chẳng hạn như Solaris Zones.

**Mẫu: Triển khai dịch vụ dưới dạng vùng chứa**

Triển khai các dịch vụ được đóng gói dưới dạng hình ảnh container vào sản xuất. Mỗi phiên bản dịch vụ là một container. Xem[http://microservices.io/patterns/deployment/service-per-container](http://microservices.io/patterns/deployment/service-per-container.html)

[.html](http://microservices.io/patterns/deployment/service-per-container.html).

**Mỗi container là một hộp cát giúp cô lập các quy trình.**

Máy móc

Hệ điều hành

Thời gian chạy của container, chẳng hạn nhưnhư Docker

Thùng chứa

Quy trình dịch vụ

Thùng chứa

Quy trình dịch vụ

Thùng chứa

Quy trình dịch vụ

**Được chia sẻ bởi tất cả các container**

**Hình 12.7 Một container bao gồm một hoặc nhiều quy trình chạy trong một hộp cát bị cô lập. Nhiều container thường chạy trên một máy duy nhất. Các container chia sẻ hệ điều hành.**

Khi bạn tạo một container, bạn có thể chỉ định CPU, tài nguyên bộ nhớ và tùy thuộc vàoing trên việc triển khai container, có lẽ là các tài nguyên I/O. Thời gian chạy container thực thi các giới hạn này và ngăn không cho container chiếm dụng tài nguyên của máy. Khi sử dụng một khuôn khổ phối hợp Docker như Kubernetes, điều đặc biệt quan trọng là phải chỉ định các tài nguyên của container. Đó là vì khuôn khổ phối hợp sử dụng các tài nguyên được container yêu cầu để chọn máy chạy container và do đó đảm bảo rằng các máy không bị quá tải.

Hình 12.8 cho thấy quá trình triển khai dịch vụ dưới dạng vùng chứa. Tại thời điểm xây dựng, đường ống triển khai sử dụng công cụ xây dựng hình ảnh vùng chứa, công cụ này đọc mã dịch vụ và mô tả về hình ảnh, để tạo hình ảnh vùng chứa và lưu trữ trong sổ đăng ký. Khi chạy, hình ảnh vùng chứa được kéo từ sổ đăng ký và được sử dụng để tạo vùng chứa.

Chúng ta hãy xem xét chi tiết hơn các bước xây dựng và thời gian chạy.

Xây dựngthời gianRuntime

Máy ảo

$ docker xây dựng ...

Đã triển khai

BẰNG

Dịch vụ

mã số

Tạo ra

Thùng chứa

đăng ký hình ảnh

Đã triển khai

BẰNG

Máy ảo

Đường ống triển khai

Công cụ xây dựng container

Thời gian chạy của container

Thùng chứa

Dịch vụ thể hiện

Người lái tàutài liệu

Hình ảnh container dịch vụ

Thời gian chạy của container

Thùng chứa

Dịch vụ thể hiện

Thùng chứa

Dịch vụ thể hiện

**Hình 12.8 Một dịch vụ được đóng gói như một hình ảnh container, được lưu trữ trong một sổ đăng ký. Khi chạy, dịch vụ bao gồm nhiều container được khởi tạo từ hình ảnh đó. Các container thường chạy trênmáy ảo. Một VM thường có thể chạy nhiều container.**

###### Triển khai dịch vụ bằng Docker

Để triển khai một dịch vụ dưới dạng một container, bạn phải đóng gói nó dưới dạng một hình ảnh container. Một hình ảnh container là một hình ảnh hệ thống tệp bao gồm ứng dụng và bất kỳ phần mềm nào cần thiết để chạy dịch vụ. Nó thường là một hệ thống tệp gốc Linux hoàn chỉnh, mặc dù các hình ảnh nhẹ hơn cũng được sử dụng. Ví dụ, để triển khai một dịch vụ dựa trên Spring Boot, bạn xây dựng một hình ảnh container chứa JAR thực thi của dịch vụ và phiên bản chính xác của JDK. Tương tự như vậy, để triển khai một ứng dụng web Java, bạn sẽ xây dựng một hình ảnh container chứa tệp WAR, Apache Tomcat và JDK.

**BULDING ADHÌNH ẢNH OCKER**

Bước đầu tiên trong việc xây dựng một hình ảnh là tạo một Dockerfile. Dockerfile mô tả cách xây dựng một hình ảnh container Docker. Nó chỉ định hình ảnh container cơ sở, một loạt các hướng dẫn để cài đặt phần mềm và cấu hình container, và lệnh shell để chạy khi container được tạo. Liệt kê 12.1 cho thấy Dockerfile được sử dụng đểxây dựng một hình ảnh cho Dịch vụ nhà hàng. Nó xây dựng một hình ảnh chứa tệp JAR thực thi của dịch vụ. Nó cấu hình container để chạy lệnh java -jar khi khởi động.

**Liệt kê 12.1Tệp Dockerđược sử dụng để xây dựngNhà hàngDịch vụ**

**Hình ảnh cơ sở**

TỪ openjdk:8u171-jre-alpine CHẠY apk --no-cache thêm curl

**Cài đặt curl để sử dụng khi kiểm tra sức khỏe.**

**Cấu hình Docker để chạy java -jar .. khi container**

CMD java ${JAVA\_OPTS} -jar ftgo-restaurant-service.jar KIỂM TRA SỨC KHỎE --start-period=30s --

**đã bắt đầu.**

khoảng thời gian = 5 giây CMD curl http://localhost:8080/actuator/health || thoát 1SAO CHÉP build/libs/ftgo-restaurant-service.jar .

**Sao chép JAR trong thư mục build của Gradle vào hình ảnh**

**Cấu hình Docker để**

**gọi điểm cuối kiểm tra tình trạng.**

Ảnh cơ sở openjdk:8u171-jre-alpine là ảnh Linux có dấu chân tối thiểu chứa JRE. Dockerfile sao chép JAR của dịch vụ vào ảnh và cấu hình ảnh để thực thi JAR khi khởi động. Nó cũng cấu hình Docker để định kỳ gọi điểm cuối kiểm tra tình trạng, được mô tả trong chương 11. Chỉ thị HEALTHCHECK yêu cầu gọi API điểm cuối kiểm tra tình trạng, được mô tả trong chương 11, cứ sau 5 giây sau độ trễ ban đầu là 30 giây, giúp dịch vụ có thời gian để bắt đầu.

Sau khi bạn đã viết Dockerfile, bạn có thể xây dựng hình ảnh. Sau đâydanh sách hiển thị các lệnh shell để xây dựng hình ảnh cho Dịch vụ nhà hàng. Tập lệnh xây dựng tệp JAR của dịch vụ và thực thi lệnh docker build để tạo hình ảnh.

**Liệt kê 12.2 Các lệnh shell được sử dụng để xây dựng hình ảnh container cho**

**Dịch vụ nhà hàng**

cd ftgo-restaurant-service

../gradlew lắp ráp

**Chuyển tới thư mục của dịch vụ.**

**Xây dựng JAR của dịch vụ.**

docker build -t ftgo-restaurant-service .

 **Xây dựng hình ảnh.**

Người khuân vácLệnh build có hai đối số: đối số -t chỉ định tên của hình ảnh và . chỉ định Docker gọi ngữ cảnh nào. Ngữ cảnh, trong ví dụ này là thư mục hiện tại, bao gồm Dockerfile và các tệp được sử dụng để xây dựng hình ảnh. Lệnh docker build tải ngữ cảnh lên daemon Docker, nơi xây dựng hình ảnh.

**PĐANG SỬ DỤNGDOCKER IMAGE ĐẾN MỘT ĐĂNG KÝ**

Bước cuối cùng của quá trình xây dựng là đẩy hình ảnh Docker mới được xây dựng vào cái được gọi là sổ đăng ký. Sổ đăng ký Docker tương đương với kho lưu trữ Java Maven cho các thư viện Java hoặc sổ đăng ký NodeJS npm cho các gói NodeJS. Docker hub là một ví dụ về sổ đăng ký Docker công khai và tương đương với Maven Central hoặc NpmJS.org. Nhưng đối với các ứng dụng của bạn, có thể bạn sẽ muốn sử dụng sổ đăng ký riêng do các dịch vụ cung cấp, chẳng hạn như sổ đăng ký Docker Cloud hoặc Sổ đăng ký AWS EC2 Container.

Bạn phải sử dụng hai lệnh Docker để đẩy một hình ảnh vào sổ đăng ký. Đầu tiên, bạn sử dụng lệnh docker tag để đặt tên cho hình ảnh có tiền tố là tên máy chủ

và cổng tùy chọn của sổ đăng ký. Tên hình ảnh cũng được thêm hậu tố là phiên bản, điều này sẽ quan trọng khi bạn tạo bản phát hành mới của dịch vụ. Ví dụ, nếu tên máy chủ của sổ đăng ký là registry.acme.com, bạn sẽ sử dụng lệnh này để gắn thẻ hình ảnh:

thẻ docker ftgo-restaurant-service registry.acme.com/ftgo-restaurant- service:1.0.0.RELEASE

Tiếp theo, bạn sử dụng lệnh docker push để tải hình ảnh được gắn thẻ đó lên sổ đăng ký:

đẩy dockerregister.acme.com/ftgo-restaurant-service:1.0.0.RELEASE

Lệnh này thường mất ít thời gian hơn nhiều so với bạn mong đợi. Đó là vì Docker image có cái gọi là hệ thống tệp nhiều lớp, cho phép Docker chỉ chuyển một phần của image qua mạng. Hệ điều hành của image, thời gian chạy Java và ứng dụng nằm trong các lớp riêng biệt. Docker chỉ cần chuyển những lớp không tồn tại trong đích. Do đó, việc chuyển image qua mạng khá nhanh khi Docker chỉ phải di chuyển các lớp của ứng dụng, vốn là một phần nhỏ của image.

Bây giờ chúng ta đã đẩy hình ảnh vào sổ đăng ký, hãy cùng xem cách tạo vùng chứa.

**RUNNING ADTHÙNG ĐỰNG OCKER**

Sau khi bạn đóng gói dịch vụ của mình dưới dạng hình ảnh container, bạn có thể tạo một hoặc nhiều container. Cơ sở hạ tầng container sẽ kéo hình ảnh từ sổ đăng ký vào máy chủ sản xuất. Sau đó, nó sẽ tạo một hoặc nhiều container từ hình ảnh đó. Mỗi container là một phiên bản của dịch vụ của bạn.

Như bạncó thể mong đợi, Docker cung cấp lệnh docker run để tạo và khởi động một container. Liệt kê 12.3 cho thấy cách sử dụng lệnh này để chạy Restaurant Service. Lệnh docker run có một số đối số, bao gồm hình ảnh container và thông số kỹ thuật của các biến môi trường để đặt trong container thời gian chạy. Chúng được sử dụng để truyền cấu hình bên ngoài, chẳng hạn như vị trí mạng của cơ sở dữ liệu và nhiều thông tin khác.

**Liệt kê 12.3 Sử dụngchạy dockerđể chạy một dịch vụ chứa trong container**

**Chạy nó như một**

chạy docker \

-d \

**nền daemon**

**Tên của container**

**Liên kết cổng 8080 củacontainer đến cảng 8082**

--tên ftgo-restaurant-service \

-p 8082:8080 \

**của máy chủ**

-e SPRING\_DATASOURCE\_URL=... -e SPRING\_DATASOURCE\_USERNAME=... \

-e MẬT KHẨU DỮ LIỆU SPRING=... \

register.acme.com/ftgo-restaurant-service:1.0.0.RELEASE

**Hình ảnh để chạy**

**Biến môi trường**

Lệnh docker run sẽ kéo hình ảnh từ sổ đăng ký nếu cần. Sau đó, nó tạo và khởi động container, chạy lệnh java -jar được chỉ định trong Dockerfile.

Sử dụng lệnh docker run có vẻ đơn giản, nhưng có một số vấn đề.lems. Một là docker run không phải là cách đáng tin cậy để triển khai dịch vụ, vì nó tạo ra một container chạy trên một máy duy nhất. Công cụ Docker cung cấp một số tính năng quản lý cơ bản, chẳng hạn như tự động khởi động lại container nếu chúng bị sập hoặc nếu máy được khởi động lại. Nhưng nó không xử lý được sự cố máy.

Một vấn đề khác là các dịch vụ thường không tồn tại riêng biệt. Chúng phụ thuộc vào các dịch vụ khác, chẳng hạn như cơ sở dữ liệu và trình môi giới tin nhắn. Sẽ rất tuyệt nếu triển khai hoặc hủy triển khai một dịch vụ và các phụ thuộc của nó như một đơn vị.

Một cách tiếp cận tốt hơn, đặc biệt hữu ích trong quá trình phát triển là sử dụng Docker Compose. Docker Compose là một công cụ cho phép bạn định nghĩa một cách khai báo một tập hợp các container bằng tệp YAML, sau đó bắt đầu và dừng các container đó như một nhóm. Hơn nữa, tệp YAML là một cách thuận tiện để chỉ định nhiều thuộc tính cấu hình bên ngoài. Để tìm hiểu thêm về Docker Compose, tôi khuyên bạn nên đọc Docker in Action của Jeff Nickoloff (Manning, 2016) và xem tệp docker-compose.yml trong mã ví dụ.

Tuy nhiên, vấn đề với Docker Compose là nó chỉ giới hạn ở một máy duy nhất.Để triển khai các dịch vụ một cách đáng tin cậy, bạn phải sử dụng một khuôn khổ điều phối Docker, chẳng hạn như Kubernetes, biến một tập hợp các máy thành một nhóm tài nguyên. Tôi mô tả cách sử dụng Kubernetes sau đó, trong phần 12.4. Trước tiên, chúng ta hãy xem xét những lợi ích và hạn chế của việc sử dụng container.

###### Lợi ích của việc triển khai dịch vụ dưới dạng container

Việc triển khai các dịch vụ dưới dạng container có một số lợi ích. Đầu tiên, container có nhiều lợi ích của máy ảo:

* + - * Đóng gói ngăn xếp công nghệ trong đó API để quản lý dịch vụ của bạn trở thành API chứa.
      * Các trường hợp dịch vụ bị cô lập.
      * Tài nguyên của các phiên bản dịch vụ bị hạn chế.

Nhưng không giống như máy ảo, container là công nghệ nhẹ. Container image thường được xây dựng nhanh. Ví dụ, trên máy tính xách tay của tôi, chỉ mất năm giây để đóng gói một ứng dụng Spring Boot thành một container image. Việc di chuyển một container image qua mạng, chẳng hạn như đến và đi từ sổ đăng ký container, cũng tương đối nhanh, chủ yếu là vì chỉ cần chuyển một tập hợp con các lớp của một image. Container cũng khởi động rất nhanh, vì không có quá trình khởi động hệ điều hành dài dòng. Khi một container khởi động, tất cả những gì chạy là dịch vụ.

###### Nhược điểm của việc triển khai dịch vụ dưới dạng container

Một nhược điểm đáng kể của container là bạn phải chịu trách nhiệm cho việc nâng cấp không phân biệt của việc quản lý hình ảnh container. Bạn phải vá hệ điều hành và thời gian chạy. Ngoài ra, trừ khi bạn đang sử dụng giải pháp container được lưu trữ như Google Container Engine hoặc AWS ECS, bạn phải quản lý cơ sở hạ tầng container và có thể là cơ sở hạ tầng VM mà nó chạy trên đó.

#### Triển khai ứng dụng FTGO với Kubernetes

Bây giờ chúng ta đã xem xét các container và sự đánh đổi của chúng, hãy xem cách triển khaiDịch vụ nhà hàng của ứng dụng FTGO sử dụng Kubernetes. Docker Compose, được mô tả trong phần 12.3.1, rất tuyệt vời cho việc phát triển và thử nghiệm. Nhưng để chạy các dịch vụ được chứa trong container một cách đáng tin cậy trong quá trình sản xuất, bạn cần sử dụng thời gian chạy container tinh vi hơn nhiều, chẳng hạn như Kubernetes. Kubernetes là một khuôn khổ điều phối Docker, một lớp phần mềm trên Docker biến một tập hợp các máy thành một nhóm tài nguyên duy nhất để chạy các dịch vụ. Nó cố gắng duy trì số lượng phiên bản mong muốn của mỗi dịch vụ đang chạy mọi lúc, ngay cả khi các phiên bản dịch vụ hoặc máy bị sập. Sự nhanh nhẹn của container kết hợp với sự tinh vi của Kubernetes là một cách hấp dẫn để triển khai các dịch vụ.

Trong phần này, trước tiên tôi sẽ giới thiệu tổng quan về Kubernetes, chức năng và kiến ​​trúc của nó. Sau đó, tôi sẽ chỉ cách triển khai dịch vụ bằng Kubernetes. Kubernetes là một chủ đề phức tạp và việc trình bày đầy đủ về nó nằm ngoài phạm vi của cuốn sách này, vì vậy tôi chỉ trình bày cách sử dụng Kubernetes theo góc nhìn của một nhà phát triển. Để biết thêm thông tin, tôi khuyên bạn nên đọc Kubernetes in Action của Marko Luksa (Manning, 2018).

###### Tổng quan về Kubernetes

Kubernetes là một khuôn khổ dàn dựng Docker. Một khuôn khổ dàn dựng Docker coi một tập hợp các máy chạy Docker như một nhóm tài nguyên. Bạn yêu cầu khuôn khổ dàn dựng Docker chạy N phiên bản dịch vụ của bạn và nó xử lý phần còn lại. Hình 12.9 cho thấy kiến ​​trúc của một khuôn khổ dàn dựng Docker.

Một khuôn khổ điều phối Docker,chẳng hạn như Kubernetes, có ba chức năng chính:

* + - * *Quản lý tài nguyên*—Xử lý một cụm máy như một nhóm CPU, bộ nhớ và khối lượng lưu trữ, biến tập hợp các máy thành một máy duy nhất.
      * *Lên lịch*—Chọn máy để chạy container của bạn. Theo mặc định, lập lịch sẽ xem xét các yêu cầu về tài nguyên của container và các tài nguyên khả dụng của từng nút. Nó cũng có thể triển khai affinity, tức là đặt các container trên cùng một nút và anti-affinity, tức là đặt các container trên các nút khác nhau.
      * *Quản lý dịch vụ*—Triển khai khái niệm về các dịch vụ được đặt tên và có phiên bản ánh xạ trực tiếp đến các dịch vụ trong kiến ​​trúc dịch vụ vi mô. Khung dàn dựng đảm bảo rằng số lượng phiên bản khỏe mạnh mong muốn luôn chạy. Nó cân bằng tải các yêu cầu trên các phiên bản đó. Khung dàn dựng thực hiện nâng cấp liên tục các dịch vụ và cho phép bạn quay lại phiên bản cũ.

SVCMỘT

Thùng chứa

SVCB

Thùng chứa

SVCC

Thùng chứa

|  |
| --- |
| Quản lý dịch vụ |
| Lên lịch |
| Quản lý tài nguyên |

**Hình 12.9 Một khung điều phối Docker biến một tập hợp các máy chạy Docker thành một cụm tài nguyên. Nó gán các container cho các máy. Khung cố gắng duy trì số lượng container khỏe mạnh mong muốn chạy mọi lúc.**

Khung điều phối Docker

Máy móc

Hoạt độnghệ thống

Người lái tàu

Máy móc

Hoạt độnghệ thống

Người lái tàu

Máy móc

Hoạt độnghệ thống

Người lái tàu

Các khuôn khổ phối hợp Docker là một cách ngày càng phổ biến để triển khai các ứng dụng. Docker Swarm là một phần của công cụ Docker, do đó dễ thiết lập và sử dụng. Kubernetes phức tạp hơn nhiều để thiết lập và quản lý, nhưng cũng tinh vi hơn nhiều. Tại thời điểm viết bài, Kubernetes có động lực to lớn, với một cộng đồng nguồn mở lớn. Chúng ta hãy xem xét kỹ hơn cách thức hoạt động của nó.

**KKIẾN TRÚC UBERNETES**

Kubernetes chạy trên một cụm máy. Hình 12.10 cho thấy kiến ​​trúc của một cụm Kubernetes. Mỗi máy trong một cụm Kubernetes là một máy chủ hoặc một nút. Một cụm thông thường có một số lượng nhỏ các máy chủ—có thể chỉ một—và nhiều nút. Một máy chủ chịu trách nhiệm quản lý cụm. Một nút là một công nhân chạy một hoặc nhiều pod. Một pod là đơn vị triển khai của Kubernetes và bao gồm một tập hợp các container.

Một bậc thầy điều hành một số thành phần, bao gồmsau đây:

* *Máy chủ API*—API REST để triển khai và quản lý các dịch vụ, được sử dụng bởi

dòng lệnh kubectlgiao diện, ví dụ.

* *v.v.*—Cơ sở dữ liệu NoSQL có khóa-giá trị lưu trữ dữ liệu cụm.



Nhà phát triển



Máy chủ API

Bộ điều khiển

sự quản lý

Cấu hình

lệnh

Ứng dụng

người sử dụng

Ứng dụngyêu cầu

Nút Kubernetes

Nút Kubernetes

SVC

Vỏ

SVC

Vỏ

KubectiCLI

Người lập lịch

Chủ nhân Kubernetes

Kube-proxy

Kubelet

Kube-proxy

Kubelet

v.v.

Triển khaiđường ống

**Hình 12.10 Một cụm Kubernetes bao gồm một máy chủ chính, quản lý cụm và các nút, chạy các dịch vụ. Các nhà phát triển và đường ống triển khai tương tác với Kubernetes thông qua máy chủ API, cùng với các phần mềm quản lý cụm khác chạy trên máy chủ chính. Các container ứng dụng chạy trên các nút. Mỗi nút chạy một Kubelet, quản lý container ứng dụng và một kube-proxy, định tuyến các yêu cầu ứng dụng đến các pod, trực tiếp dưới dạng proxy hoặc gián tiếp thông quacấu hình các quy tắc định tuyến iptables được tích hợp sẵn trong nhân Linux.**

* + *Người lập lịch*—Chọn một nút để chạy pod.
  + *Quản lý bộ điều khiển*—Chạy bộ điều khiển, đảm bảo rằng trạng thái của cụm khớp với trạng thái mong muốn. Ví dụ, một loại bộ điều khiển được gọi là bộ điều khiển sao chép đảm bảo rằng số lượng phiên bản mong muốn của một dịch vụ đang chạy bằng cách bắt đầu và kết thúc các phiên bản.

Một nút chạy nhiềucác thành phần, bao gồm:

* + *Kubelet*—Tạo và quản lý các pod đang chạy trên nút
  + *Kube-proxy*—Quản lý mạng, bao gồm cân bằng tải trên các nhóm
  + *Vỏ quả*—Các dịch vụ ứng dụng

Bây giờ chúng ta hãy xem các khái niệm chính của Kubernetesbạn sẽ cần phải nắm vững cách triển khai các dịch vụ trên Kubernetes.

**KTÔIKUBERNETESKHÁI NIỆM**

Như đã đề cập trong phần giới thiệu của phần này, Kubernetes khá phức tạp. Nhưng bạn có thể sử dụng Kubernetes một cách hiệu quả khi đã nắm vững một số khái niệm chính, được gọi là đối tượng. Kubernetes định nghĩa nhiều loại đối tượng. Theo quan điểm của nhà phát triển, các đối tượng quan trọng nhất là:

* *Vỏ*—Pod là đơn vị triển khai cơ bản trong Kubernetes. Nó bao gồm một hoặc nhiều container chia sẻ một địa chỉ IP và khối lượng lưu trữ. Pod cho một phiên bản dịch vụ thường bao gồm một container duy nhất, chẳng hạn như container chạy JVM. Nhưng trong một số trường hợp, pod chứa một hoặc nhiều container sidecar, triển khai các chức năng hỗ trợ. Ví dụ, máy chủ NGINX có thể có một sidecar định kỳ thực hiệnkéo gitđể tải xuống phiên bản mới nhất của trang web. Một pod là tạm thời, vì các container của pod hoặc nút mà nó đang chạy có thể bị sập.
* *Triển khai*—Một thông số kỹ thuật khai báo của một pod. Triển khai là một bộ điều khiển đảm bảo rằng số lượng phiên bản mong muốn của pod (phiên bản dịch vụ) đang chạy mọi lúc. Nó hỗ trợ phiên bản với các bản nâng cấp và khôi phục liên tục. Sau đó trong phần 12.4.2, bạn sẽ thấy rằng mỗi dịch vụ trong kiến ​​trúc dịch vụ vi mô là một triển khai Kubernetes.
* *Dịch vụ*—Cung cấp cho khách hàngcủa một dịch vụ ứng dụng có vị trí mạng tĩnh/ổn định. Đây là một dạng khám phá dịch vụ do cơ sở hạ tầng cung cấp, được mô tả trong chương 3. Một dịch vụ có địa chỉ IP và tên DNS phân giải thành địa chỉ IP đó và cân bằng tải lưu lượng TCP và UDP trên một hoặc nhiều pod. Địa chỉ IP và tên DNS chỉ có thể truy cập được trong Kubernetes. Sau đó, tôi sẽ mô tả cách cấu hình các dịch vụ có thể truy cập được từ bên ngoài cụm.
* *Bản đồ cấu hình*—Một tập hợp được đặt tên của các cặp tên-giá trị xác định cấu hình bên ngoài cho một hoặc nhiều dịch vụ ứng dụng (xem chương 11 để biết tổng quan về cấu hình bên ngoài). Định nghĩa về vùng chứa của pod có thể tham chiếu đến ConfigMap để xác định các biến môi trường của vùng chứa. Nó cũng có thể sử dụng ConfigMap để tạo các tệp cấu hình bên trong vùng chứa. Bạn có thể lưu trữ thông tin nhạy cảm, chẳng hạn như mật khẩu, dưới dạng ConfigMap được gọi là Bí mật.

Bây giờ chúng ta đã xem xét các khái niệm chính của Kubernetes, hãy cùng xem chúng hoạt động như thế nào bằng cách xem cách triển khai dịch vụ ứng dụng trên Kubernetes.

###### Triển khai dịch vụ Nhà hàng trên Kubernetes

Như đã đề cập trước đó, để triển khai một dịch vụ trên Kubernetes, bạn cần phải xác định một triển khai. Cách dễ nhất để tạo một đối tượng Kubernetes như triển khai là bằng cách viếttệp YAML. Liệt kê 12.4 là tệp YAML xác định việc triển khai cho Dịch vụ nhà hàng.Triển khai này chỉ định chạy hai bản sao của một pod. Pod chỉ có một container.

Định nghĩa container chỉ định hình ảnh Docker chạy cùng với các thuộc tính khác, chẳng hạn như giá trị của biến môi trường. Biến môi trường của container là cấu hình bên ngoài của dịch vụ. Chúng được Spring Boot đọc và có sẵn dưới dạng thuộc tính trong ngữ cảnh ứng dụng.

**Liệt kê 12.4 KubernetesTriển khaivìftgo-restaurant-service**

apiVersion: extensions/v1beta1 loại: Triển khai

siêu dữ liệu:

tên: ftgo-restaurant-service đặc điểm kỹ thuật:

**Chỉ định rằng điều nàylà một đối tượng thuộc loại Triển khai**

**Tên của triển khai**

bản sao: 2 mẫu:

siêu dữ liệu:

nhãn:

**Số lượng bản sao của pod**

**Cung cấp cho mỗi pod một nhãnđược gọi là ứng dụng có giá trị là**

ứng dụng: ftgo-restaurant-service đặc điểm kỹ thuật:

container:

- tên: ftgo-restaurant-service

**ftgo-restaurant-service**

**Đặc điểm kỹ thuật của pod, chỉ xác định một container**

hình ảnh: msapatterns/ftgo-restaurant-service:latest

imagePullPolicy: Luôn luôn

cổng:

* containerPort: 8080 tên: httpport

môi trường:

* + tên: JAVA\_OPTS

giá trị:"-Dsun.net.inetaddr.ttl=30"

* + tên: SPRING\_DATASOURCE\_URL

**Cảng container**

**Các biến môi trường của container được Spring Boot đọc**

giá trị:jdbc:mysql://ftgo-mysql/eventuate

* + tên: SPRING\_DATASOURCE\_USERNAME

giá trịTừ: secretKeyRef:

tên: ftgo-db-secret

chìa khóa: tên người dùng

* + tên: SPRING\_DATASOURCE\_PASSWORD

giá trịTừ: secretKeyRef:

tên: ftgo-db-secret khóa: mật khẩu

**Các giá trị nhạy cảm được lấy từ Kubernetes Secret có tên là ftgo-db-secret**

* + tên:SPRING\_DATASOURCE\_DRIVER\_CLASS\_NAME

giá trị: com.mysql.jdbc.Driver

* + tên: EVENTUATELOCAL\_KAFKA\_BOOTSTRAP\_SERVERS

giá trị: ftgo-kafka:9092

* + tên: EVENTUATELOCAL\_ZOOKEEPER\_CONNECTION\_STRING

giá trị: ftgo-zookeeper:2181

sự sống độngProbe:httpLấy:

đường dẫn: /actuator/health cổng: 8080

InitialDelayGiây: 60

giai đoạn Giây: 20 sẵn sàngThăm dò:

**Cấu hình Kubernetes để gọi điểm cuối kiểm tra tình trạng.**

httpLấy:

đường dẫn: /actuator/health cổng: 8080

InitialDelayGiây: 60

chu kỳGiây: 20

Định nghĩa triển khai này cấu hình Kubernetes để gọi điểm cuối kiểm tra tình trạng của Dịch vụ nhà hàng. Như đã mô tả trong chương 11, điểm cuối kiểm tra tình trạng cho phép Kubernetes xác định tình trạng của phiên bản dịch vụ. Kubernetes triển khai hai kiểm tra khác nhau. Kiểm tra đầu tiên là availabilityProbe, được sử dụng để xác định xem có nên định tuyến lưu lượng đến phiên bản dịch vụ hay không. Trong ví dụ này, Kubernetes gọi điểm cuối HTTP /actuator/health sau mỗi 20 giây sau độ trễ 30 giây ban đầu, giúp nó có cơ hội khởi tạo. Nếu một số (mặc định là 1) readyProbes liên tiếp thành công, Kubernetes coi dịch vụ là đã sẵn sàng, trong khi nếu một số (mặc định là 3) readyProbes liên tiếp không thành công, thì dịch vụ được coi là chưa sẵn sàng. Kubernetes sẽ chỉ định tuyến lưu lượng đến phiên bản dịch vụ khi readyProbe chỉ ra rằng dịch vụ đã sẵn sàng.

Kiểm tra sức khỏe thứ hai là livenessProbe. Nó được cấu hình theo cùng cách với availabilityProbe. Nhưng thay vì xác định xem lưu lượng có nên được định tuyến đến một phiên bản dịch vụ hay không, livenessProbe xác định xem Kubernetes có nên chấm dứt và khởi động lại phiên bản dịch vụ hay không. Nếu một số (mặc định là 3) liveness-Probe liên tiếp bị lỗi, Kubernetes sẽ chấm dứt và khởi động lại dịch vụ.

Sau khi bạn đã viết tệp YAML, bạn có thể tạo hoặc cập nhật triển khai bằng cáchsử dụng lệnh kubectl apply:

kubectl áp dụng -f ftgo-restaurant-service/src/deployment/kubernetes/ftgo- restaurant-service.yml

Lệnh này thực hiện yêu cầu tới máy chủ API Kubernetes dẫn đến việc tạo triển khai và các pod.

Để tạo triển khai này, trước tiên bạn phải tạo Kubernetes Secret có tên là

ftgo-db-bí mật. Một cách nhanh chóng và không an toàn để thực hiện điều đó là như sau:

kubectl tạo bí mật chung ftgo-db-secret \

--from-literal=tên người dùng=mysqluser--from-literal=mật khẩu=mysqlpw

Lệnh này tạo ra một bí mật chứa ID người dùng cơ sở dữ liệu và mật khẩu được chỉ định trên dòng lệnh. Xem tài liệu Kubernetes ([https://kuberneteS](https://kubernetes.io/docs/concepts/configuration/secret/#creating-your-own-secrets)

[.io/docs/concepts/configuration/secret/#creating-your-own-secrets](https://kubernetes.io/docs/concepts/configuration/secret/#creating-your-own-secrets)) để có cách tạo bí mật an toàn hơn.

**CĐỌC AKDỊCH VỤ UBERNETES**

Tại thời điểm này, các pod đang chạy và triển khai Kubernetes sẽ cố gắng hết sức để giữ cho chúng chạy. Vấn đề là các pod đã được gán địa chỉ IP động và do đó, không hữu ích cho máy khách muốn thực hiện yêu cầu HTTP. Như đã mô tả trong chương 3, giải pháp là sử dụng cơ chế khám phá dịch vụ.

Một cách tiếp cận là sử dụng cơ chế khám phá phía máy khách và cài đặt sổ đăng ký dịch vụ, chẳng hạn như Netflix OSS Eureka. May mắn thay, chúng ta có thể tránh làm điều đó bằng cách sử dụng cơ chế khám phá dịch vụ được tích hợp sẵn trong Kubernetes và xác định dịch vụ Kubernetes.

Dịch vụ là đối tượng Kubernetes cung cấp cho máy khách của một hoặc nhiều pod một điểm cuối ổn định. Nó có một địa chỉ IP và một tên DNS để giải quyết địa chỉ IP đó. Dịch vụ cân bằng tải lưu lượng đến địa chỉ IP đó trên các pod. Liệt kê 12.5hiển thị dịch vụ Kubernetes cho Dịch vụ nhà hàng. Dịch vụ này định tuyến lưu lượng truy cập từ http://ftgo-restaurant-service:8080 đến các nhóm được xác định bởi triển khai được hiển thị trong danh sách.

**Liệt kê 12.5 Định nghĩa YAML của dịch vụ Kubernetes cho**

**ftgo-restaurant-service**

apiVersion: v1 loại: Siêu dữ liệu dịch vụ:

tên: ftgo-restaurant-service đặc điểm kỹ thuật:

cổng:



**Sự phơi bày**

**Tên của dịch vụ, cũng là tên DNS**

- cổng: 8080

targetPort: 8080 bộ chọn:

**cảng**

**Cảng container**

**để định tuyến lưu lượng truy cập đến**

---

ứng dụng: ftgo-restaurant-service

**Chọn các container để định tuyến lưu lượng truy cập đến**

Phần chính của định nghĩa dịch vụ là bộ chọn, bộ chọn này chọn các pod mục tiêu. Nó chọn những pod có nhãn tên là app với giá trị ftgo-restaurant-service. NếuNếu bạn nhìn kỹ, bạn sẽ thấy rằng vùng chứa được định nghĩa trong danh sách 12.4 có nhãn như vậy.

Sau khi viết xong tệp YAML, bạn có thể tạo dịch vụ bằng lệnh này:

kubectl áp dụng -f ftgo-restaurant-service-service.yml

Bây giờ chúng ta đã tạo dịch vụ Kubernetes, bất kỳ máy khách nào của Restaurant Service đang chạy bên trong cụm Kubernetes đều có thể truy cập REST API của nó qua http:// ftgo-restaurant-service:8080. Sau đó, tôi sẽ thảo luận về cách nâng cấp các dịch vụ đang chạy, nhưng trước tiên hãy cùng xem cách làm cho các dịch vụ có thể truy cập được từ bên ngoài cụm Kubernetes.

###### Triển khai cổng API

Dịch vụ Kubernetes cho Restaurant Service, được hiển thị trong danh sách 12.5, chỉ có thể truy cập được từ bên trong cụm. Điều đó không phải là vấn đề đối với Restaurant Service, nhưng API Gateway thì sao? Vai trò của nó là định tuyến lưu lượng từ thế giới bên ngoài đến dịch vụ. Do đó, nó cần có thể truy cập được từ bên ngoài cụm. May mắn thay, một dịch vụ Kubernetes cũng hỗ trợ trường hợp sử dụng này. Dịch vụ mà chúng ta đã xem xét trước đó là dịch vụ ClusterIP, đây là dịch vụ mặc định, nhưng tuy nhiên, có hai loại dịch vụ khác: Node-Port và LoadBalancer.

Dịch vụ NodePort có thể truy cập thông qua một cổng toàn cụm trên tất cả các nút trong cụm. Bất kỳ lưu lượng nào đến cổng đó trên bất kỳ nút cụm nào đều được cân bằng tải đến các pod phụ trợ. Bạn phải chọn một cổng khả dụng trong phạm vi 30000–32767. Ví dụ: danh sách 12.6 hiển thị một dịch vụ định tuyến lưu lượng đến cổng 30000 của Consumer Service.

**Liệt kê 12.6 Định nghĩa YAML của mộtCổng Nodedịch vụ định tuyến lưu lượng đến cảng8082 củaDịch vụ khách hàng**

apiVersion: v1 loại: Siêu dữ liệu dịch vụ:

tên: ftgo-api-gateway đặc điểm kỹ thuật:

loại: Cổng NodePort:

- nodePort: 30000

cổng: 80

targetPort: 8080 bộ chọn:

ứng dụng: ftgo-api-gateway

**Chỉ định một loại NodePort**

**Cổng cụm rộng**

---

API Gateway nằm trong cụm sử dụng URL[http://ftgo-api-gateway](http://ftgo-api-gateway/)và bên ngoài URL http://<node-ip-address>:3000/, trong đó node-ip-address là địa chỉ IP của một trong các nút. Sau khi cấu hình dịch vụ NodePort, bạn có thể, ví dụ, cấu hình AWS Elastic Load Balancer (ELB) để cân bằng tải các yêu cầu từ internet trên các nút. Một lợi ích chính của phương pháp này là ELB hoàn toàn nằm trong tầm kiểm soát của bạn. Bạn có toàn quyền linh hoạt khi cấu hình nó.

Tuy nhiên, dịch vụ loại NodePort không phải là lựa chọn duy nhất. Bạn cũng có thể sử dụng dịch vụ Load-Balancer, dịch vụ này tự động cấu hình bộ cân bằng tải dành riêng cho đám mây. Bộ cân bằng tải sẽ là ELB nếu Kubernetes đang chạy trên AWS. Một lợi ích của loại dịch vụ này là bạn không còn phải cấu hình bộ cân bằng tải của riêng mình nữa. Tuy nhiên, nhược điểm là mặc dù Kubernetes cung cấp một số tùy chọn để cấu hình ELB, chẳng hạn như chứng chỉ SSL, nhưng bạn có ít quyền kiểm soát hơn đối với cấu hình của nó.

###### Triển khai không có thời gian chết

Hãy tưởng tượng bạn đã cập nhật Restaurant Service và muốn triển khai những thay đổi đó vào sản xuất. Cập nhật một dịch vụ đang chạy là một quy trình ba bước đơn giản khi sử dụng Kubernetes:

**1**Xây dựng một hình ảnh container mới và đẩy nó vào sổ đăng ký bằng cùng một quy trình được mô tả trước đó. Sự khác biệt duy nhất là hình ảnh sẽ được gắn thẻ bằng một thẻ phiên bản khác—ví dụ: ftgo-restaurant-service:1.1.0.RELEASE.

**2**Chỉnh sửa tệp YAML để triển khai dịch vụ sao cho nó tham chiếu đến hình ảnh mới.

**3** Cập nhật triển khai bằng cách sử dụngkubectl áp dụng -fyêu cầu.

Kubernetes sau đó sẽ thực hiện nâng cấp liên tục các pod. Nó sẽ tạo ra các pod chạy phiên bản 1.1.0.RELEASE và chấm dứt các pod chạy phiên bản

1.0.0.RELEASE. Điểm tuyệt vời về cách Kubernetes thực hiện điều này là nó không chấm dứt các pod cũ cho đến khi các pod thay thế của chúng sẵn sàng xử lý các yêu cầu. Nó sử dụng cơ chế availabilityProbe, một cơ chế kiểm tra tình trạng được mô tả trước đó trong phần này, để xác định xem một pod đã sẵn sàng hay chưa. Kết quả là, sẽ luôn có các pod khả dụng để xử lý các yêu cầu. Cuối cùng, giả sử các pod mới khởi động thành công, tất cả các pod của triển khai sẽ chạy phiên bản mới.

Nhưng nếu có vấn đề và các pod phiên bản 1.1.0.RELEASE không khởi động thì sao? Có thể có lỗi, chẳng hạn như tên ảnh container bị viết sai chính tả hoặc thiếu biến môi trường cho thuộc tính cấu hình mới. Nếu các pod không khởi động được, quá trình triển khai sẽ bị kẹt. Tại thời điểm đó, bạn có hai lựa chọn. Một lựa chọn là sửa tệp YAML và chạy lại kubectl apply -f để cập nhật quá trình triển khai. Lựa chọn còn lại là khôi phục quá trình triển khai.

Một triển khai duy trì lịch sử của những gì được gọi là triển khai. Mỗi lần bạn cập nhật triển khai, nó sẽ tạo ra một triển khai mới. Do đó, bạn có thể dễ dàng khôi phục triển khai về phiên bản trước đó bằng cách thực hiện lệnh sau:

kubectl rollout undo triển khai ftgo-restaurant-service

Kubernetes sau đó sẽ thay thế các pod đang chạy phiên bản1.1.0.PHÁT HÀNHvới các pod chạy phiên bản cũ hơn,1.0.0.PHÁT HÀNH.

Triển khai Kubernetes là một cách tốt để triển khai dịch vụ mà không có thời gian chết. Nhưng nếu lỗi chỉ xuất hiện sau khi pod đã sẵn sàng và nhận được lưu lượng sản xuất thì sao? Trong tình huống đó, Kubernetes sẽ tiếp tục triển khai các phiên bản mới, do đó, ngày càng nhiều người dùng sẽ bị ảnh hưởng. Mặc dù hệ thống giám sát của bạn hy vọng sẽ phát hiện ra sự cố và nhanh chóng khôi phục quá trình triển khai, nhưng bạn sẽ không tránh khỏi việc ảnh hưởng đến ít nhất một số người dùng. Để giải quyết vấn đề này và khiến việc triển khai phiên bản mới của dịch vụ trở nên đáng tin cậy hơn, chúng ta cần tách biệt việc triển khai, nghĩa là đưa dịch vụ vào hoạt động sản xuất, khỏi việc phát hành dịch vụ, nghĩa là làm cho dịch vụ khả dụng để xử lý lưu lượng sản xuất. Hãy cùng xem cách thực hiện điều đó bằng cách sử dụng lưới dịch vụ.

###### Sử dụng lưới dịch vụ để tách biệt triển khai khỏi phát hành

Cách truyền thống để triển khai phiên bản mới của dịch vụ là trước tiên thử nghiệm trong môi trường dàn dựng. Sau đó, khi đã vượt qua bài kiểm tra trong dàn dựng, bạn triển khai trong sản xuất bằng cách thực hiện nâng cấp liên tục thay thế các phiên bản cũ của dịch vụ bằng các phiên bản dịch vụ mới. Một mặt, như bạn vừa thấy, triển khai Kubernetes giúp thực hiện nâng cấp liên tục rất đơn giản. Mặt khác, cách tiếp cận này giả định rằng sau khi phiên bản dịch vụ đã vượt qua các bài kiểm tra trong môi trường dàn dựng, nó sẽ hoạt động trong sản xuất. Đáng buồn thay, điều này không phải lúc nào cũng đúng.

Một lý do là vì dàn dựng không có khả năng là một bản sao chính xác, nếu không vì lý do nào kháchơn là môi trường sản xuất có thể lớn hơn nhiều và xử lý nhiều lưu lượng hơn. Cũng tốn thời gian để giữ cho hai môi trường được đồng bộ hóa. Do sự khác biệt, có khả năng một số lỗi sẽ chỉ xuất hiện trong sản xuất. Và ngay cả khi đó là bản sao chính xác, bạn không thể đảm bảo rằng thử nghiệm sẽ phát hiện ra tất cả các lỗi.

Nhiều lắmcách đáng tin cậy hơn để triển khai phiên bản mới là tách biệt việc triển khai với việc phát hành:

* + - * *Triển khai*—Chạy trong môi trường sản xuất
      * *Phát hành một dịch vụ*—Cung cấp cho người dùng cuối

Sau đó, bạn triển khai dịch vụ vào sản xuất bằng các bước sau:

**1**Triển khai phiên bản mới vào sản xuất mà không cần chuyển bất kỳ yêu cầu nào từ người dùng cuối đến phiên bản đó.

**2** Kiểm tra trong quá trình sản xuất.

**3** Phát hành cho một số ít người dùng cuối.

**4**Phát hành dần dần cho số lượng người dùng ngày càng lớn cho đến khi có thể xử lý toàn bộ lưu lượng truy cập sản xuất.

**5**Nếu có bất kỳ vấn đề nào xảy ra, hãy quay lại phiên bản cũ—nếu không, khi bạn đã chắc chắn rằng phiên bản mới hoạt động chính xác, hãy xóa phiên bản cũ.

Trong điều kiện lý tưởng, các bước đó sẽ được thực hiện thông qua đường ống triển khai hoàn toàn tự động, giám sát cẩn thận dịch vụ mới triển khai để phát hiện lỗi.

Theo truyền thống, việc tách biệt các triển khai và bản phát hành theo cách này là một thách thức vì cần rất nhiều công sức để triển khai. Nhưng một trong những lợi ích của việc sử dụng lưới dịch vụ là việc sử dụng kiểu triển khai này dễ dàng hơn nhiều. Lưới dịch vụ, như đã mô tả trong chương 11, là cơ sở hạ tầng mạng lưới làm trung gian cho mọi giao tiếp giữa một dịch vụ và các dịch vụ khác cũng như các ứng dụng bên ngoài. Ngoài việc đảm nhận một số trách nhiệm của khuôn khổ khung dịch vụ vi mô, lưới dịch vụ còn cung cấp cân bằng tải dựa trên quy tắc và định tuyến lưu lượng cho phép bạn chạy an toàn nhiều phiên bản dịch vụ của mình cùng lúc. Sau trong phần này, bạn sẽ thấy rằng bạn có thể định tuyến người dùng thử nghiệm đến một phiên bản của dịch vụ và người dùng cuối đến một phiên bản khác, ví dụ.

Như đã mô tả trong chương 11, có một sốlưới dịch vụ để lựa chọn. Trong phần này, tôi sẽ chỉ cho bạn cách sử dụng Istio, một lưới dịch vụ mã nguồn mở phổ biến ban đầu được phát triển bởi Google, IBM và Lyft. Tôi bắt đầu bằng cách cung cấp tổng quan ngắn gọn về Istio và một số tính năng của nó. Tiếp theo, tôi sẽ mô tả cách triển khai ứng dụng bằng Istio. Sau đó, tôi sẽ chỉ cho bạn cách sử dụng khả năng định tuyến lưu lượng truy cập của nó để triển khai và phát hành bản nâng cấp cho dịch vụ.

**ỒTỔNG QUAN VỀTÔILƯỚI DỊCH VỤ STIO**

Trang web Istio mô tả Istio là “Nền tảng mở để kết nối, quản lý và bảo mật các dịch vụ vi mô” ([https://istio.io](https://istio.io/)). Đây là lớp mạng mà tất cả lưu lượng mạng của dịch vụ của bạn đều đi qua. Istio có một bộ tính năng phong phú được sắp xếp thành bốn danh mục chính:

* + - * *Quản lý giao thông*—Bao gồm khám phá dịch vụ, cân bằng tải, quy tắc định tuyến và bộ ngắt mạch
      * *Bảo vệ*—Bảo mật liên lạc giữa các dịch vụ bằng cách sử dụng Bảo mật lớp truyền tải (TLS)
        + *Đo từ xa*—Ghi lại số liệuvề lưu lượng mạng và triển khai theo dõi phân tán
        + *Thực thi chính sách*—Thực thi hạn ngạch và giới hạn tỷ lệ

Phần này tập trung vào khả năng quản lý lưu lượng truy cập của Istio.

Hình 12.11 cho thấy kiến ​​trúc của Istio. Nó bao gồm một mặt phẳng điều khiển và một mặt phẳng dữ liệu. Mặt phẳng điều khiển thực hiện các chức năng quản lý, bao gồm cấu hình mặt phẳng dữ liệu để định tuyến lưu lượng. Mặt phẳng dữ liệu bao gồm các proxy Envoy, một proxy cho mỗi phiên bản dịch vụ.

Hai thành phần chính của mặt phẳng điều khiển là Pilot và Mixer. Pilot trích xuất thông tin về các dịch vụ được triển khai từ cơ sở hạ tầng cơ bản. Ví dụ, khi chạy trên Kubernetes, Pilot sẽ truy xuất các dịch vụ và pod lành mạnh. Nó cấu hình các proxy Envoy để định tuyến lưu lượng theo các quy tắc định tuyến đã xác định.*Máy trộn*thu thập dữ liệu từ xa từ các proxy Envoy và thực thi các chính sách.

Chìa khóa

Đăng ký dịch vụ

Dịch vụ

Vỏ

Cổng APIthùng chứa

Vỏ

Chính sách yêu cầu cấu hình kiểm tra Telemetry

Vỏ

NHẬN/co

mer/1

o-dịch vụ tiêu dùng

GET/consu Máy chủ: ftg

người tiêu dùng/1

NHẬN/người tiêu dùng/1

Máy chủ: ftgo-consumer-service

Container dịch vụ tiêu dùng

NHẬN/người tiêu dùng/1

Giám sát cơ sở hạ tầng

Máy chủ số liệu

Máy chủ ghi nhật ký

IstioContainer sứ giả

Sứ giả Istiothùng chứa

Mặt phẳng dữ liệu Istio

Cấu hình Kiểm tra

Đo từ xa

Phi công

Máy trộn

Truy vấn cho các dịch vụ được triển khai

Máy bay điều khiển Istio

Kubernetes

**Hình 12.11 Istio bao gồm một mặt phẳng điều khiển, có các thành phần bao gồm Pilot và Mixer, và một mặt phẳng dữ liệu, bao gồm các máy chủ proxy Envoy. Pilot trích xuất thông tin về các dịch vụ được triển khai từ cơ sở hạ tầng cơ bản và cấu hình mặt phẳng dữ liệu. Mixer thực thi các chính sách như hạn ngạch và thu thập dữ liệu từ xa, báo cáo cho các máy chủ cơ sở hạ tầng giám sát. Các máy chủ proxy Envoy định tuyến lưu lượng vào và radịch vụ. Có một máy chủ proxy Envoy cho mỗi phiên bản dịch vụ.**

Proxy Istio Envoy là phiên bản sửa đổi của Envoy ([www.envoyproxy.io](http://www.envoyproxy.io/)). Đây là một proxy hiệu suất cao hỗ trợ nhiều giao thức, bao gồm TCP, các giao thức cấp thấp như HTTP và HTTPS và các giao thức cấp cao hơn. Nó cũng hiểu các giao thức MongoDB, Redis và DynamoDB. Envoy cũng hỗ trợ giao tiếp liên dịch vụ mạnh mẽ với các tính năng như bộ ngắt mạch, giới hạn tốc độ và thử lại tự động. Nó có thể bảo mật giao tiếp trong ứng dụng bằng cách sử dụng TLS cho giao tiếp liên Envoy.

Istio sử dụng Envoy như một sidecar, một tiến trình hoặc container chạy song song với phiên bản dịch vụ và triển khai các mối quan tâm cắt ngang. Khi chạy trên Kubernetes, proxy Envoy là một container trong pod của dịch vụ. Trong các môi trường khác không có khái niệm pod, Envoy chạy trong cùng một container với dịch vụ. Tất cả lưu lượng đến và đi từ một dịch vụ đều chảy qua proxy Envoy của nó, proxy này định tuyến lưu lượng theo các quy tắc định tuyến được cung cấp cho nó bởi mặt phẳng điều khiển. Ví dụ, dịch vụ trực tiếpGiao tiếp dịch vụ trở thành Dịch vụNguồn Sứ GiảSứ giả đích đếnDịch vụ.

**Mẫu: Xe đẩy**

Triển khai các mối quan tâm xuyên suốt trong một quy trình sidecar hoặc container chạy song song với phiên bản dịch vụ. Xem<http://microservices.io/patterns/deployment/sidecar.html>.

Istio được cấu hình bằng cách sử dụng các tệp cấu hình YAML theo phong cách Kubernetes. Nó có một lệnh-công cụ dòng gọi là istioctltương tự như kubectl. Bạn sử dụng istioctl để tạo, cập nhật và xóa các quy tắc và chính sách. Khi sử dụng Istio trên Kubernetes, bạn cũng có thể sử dụng kubectl.

Hãy cùng xem cách triển khai dịch vụ bằng Istio.

**DSỬ DỤNG DỊCH VỤ VỚITÔIĐỘI HỌC**

Việc triển khai một dịch vụ trên Istio khá đơn giản. Bạn định nghĩa một Dịch vụ Kubernetesvà Triển khai cho từng dịch vụ của ứng dụng của bạn. Liệt kê 12.7 hiển thị định nghĩa của Dịch vụ và Triển khai cho Dịch vụ Người tiêu dùng. Mặc dù gần giống với các định nghĩa tôi đã trình bày trước đó, nhưng có một vài điểm khác biệt. Đó là vì Istio có một vài yêu cầu đối với các dịch vụ và pod Kubernetes:

* + - * Cổng dịch vụ Kubernetes phải sử dụng quy ước đặt tên Istio là<nguyên mẫu-col>[-<hậu tố>], nơi giao thức làhttp,http2,grpc,mongo, hoặclàm lại. Nếu cổng không được đặt tên, Istio sẽ coi cổng đó là cổng TCP và sẽ không áp dụng định tuyến dựa trên quy tắc.
      * Một cái vỏ phải có mộtứng dụngnhãn nhưứng dụng: ftgo-consumer-service, xác định dịch vụ để hỗ trợ theo dõi phân tán Istio.
      * Để chạy nhiều phiên bản của một dịch vụ cùng lúc, tên của triển khai Kubernetes phải bao gồm phiên bản, chẳng hạn nhưftgo-consumer-dịch vụ-v1,ftgo-dịch-vụ-tiêu-dùng-v2, v.v. Các nhóm triển khai nêncó mộtphiên bảnnhãn, chẳng hạn nhưPhiên bản: v1, chỉ định phiên bản để Istio có thể định tuyến đến phiên bản cụ thể.

**Liệt kê 12.7 Triển khai dịch vụ khách hàng với Istio**

apiVersion: v1 loại: Siêu dữ liệu dịch vụ:

tên: ftgo-consumer-service đặc điểm kỹ thuật:

cổng:

- tên: http  cổng: 8080

targetPort: 8080 bộ chọn:

**Đã đặt têncảng**

ứng dụng: ftgo-consumer-service

---

apiVersion: extensions/v1beta1 loại: Triển khai

siêu dữ liệu:

tên: ftgo-consumer-service-v2 đặc điểm kỹ thuật:

bản sao: 1 mẫu:

siêu dữ liệu:nhãn:

ứng dụng: ftgo-consumer-service phiên bản: v2

đặc điểm kỹ thuật:

container:

**Triển khai theo phiên bản**

**Nhãn được đề xuất**

**Hình ảnh**

...

- hình ảnh: hình ảnh: ftgo-consumer-service:v2

**phiên bản**

Bây giờ, bạn có thể đang tự hỏi làm thế nào để chạy Envoy proxy container trong pod của dịch vụ. May mắn thay, Istio giúp việc đó trở nên dễ dàng đáng kể bằng cách tự động sửa đổi định nghĩa pod để bao gồm proxy Envoy. Có hai cách để thực hiện việc đó. Cách đầu tiên là sử dụng lệnh sidecar injection thủ công và chạy lệnh istioctl kube-inject:

istioctl kube-inject -f ftgo-consumer-service/src/deployment/kubernetes/ftgo- consumer-service.yml | kubectl áp dụng -f -

Lệnh này đọc tệp YAML của Kubernetes và xuất ra cấu hình đã sửa đổi có chứa proxy Envoy. Cấu hình đã sửa đổi sau đó được đưa vào kubectl apply.

Cách thứ hai để thêm xe đẩy Envoy vào vỏ là sử dụng hệ thống phun xe đẩy tự động.

*sự kiện*. Khi tính năng này được bật, bạn triển khai dịch vụ bằng cách sử dụng kubectl apply. Kubernetes tự động gọi Istio để sửa đổi định nghĩa pod để bao gồm proxy Envoy.

Nếu bạn mô tả nhóm dịch vụ của mình, bạn sẽ thấy rằng nó bao gồm nhiều thứ hơn là vùng chứa dịch vụ của bạn:

$ kubectl mô tả po ftgo-consumer-service-7db65b6f97-q9jprTên:ftgo-consumer-service-7db65b6f97-q9jpr

Không gian tên:mặc định

...

Init Container: istio-init:

**Khởi tạo pod**

Hình ảnh: docker.io/istio/proxy\_init:0.8.0

....

Container:

ftgo-dịch vụ tiêu dùng:

**Dịch vụthùng chứa**

Hình ảnh:msapaterns/ftgo-consumer-service:latest

...

istio-proxy:

Hình ảnh:docker.io/istio/proxyv2:0.8.0

...

**CácContainer sứ giả**

Bây giờ chúng ta đã triển khai dịch vụ, hãy cùng xem cách xác định quy tắc định tuyến.

**CQUY TẮC ĐỊNH TUYẾN ĐỂ ĐẾN V1PHIÊN BẢN**

Hãy tưởng tượng rằng bạn đã triển khai ftgo-consumer-service-v2. Trong trường hợp không có quy tắc định tuyến, Istio sẽ cân bằng tải các yêu cầu trên tất cả các phiên bản của một dịch vụ. Do đó, nó sẽ cân bằng tải trên các phiên bản 1 và 2 của ftgo-consumer-service, điều này làm mất đi mục đích sử dụng Istio. Để triển khai phiên bản mới một cách an toàn, bạn phải xác định quy tắc định tuyến định tuyến tất cả lưu lượng đến phiên bản v1 hiện tại.

**Các tuyến đường đến v1tập hợp con**



loại: VirtualServicesiêu dữ liệu:

tên:ftgo-consumer-serviceđặc điểm kỹ thuật:

chủ nhà:

-ftgo-dịch vụ tiêu dùnghttp:

**Tất cả lưu lượng truy cập được định tuyến đến v1**

siêu dữ liệu:nhãn:

**Xác định các tập hợp con v1và v2**

NHẬN/người tiêu dùng/1

-tuyến đường:

-điểm đến:

máy chủ: ftgo-consumer-servicetập hợp con: v1

trọng lượng: 100

ứng dụng: ftgo-consumer-service

**Phiên bản: v1**

Dịch vụ tiêu dùng v1 pod

loại: DestinationRulesiêu dữ liệu:

tên:ftgo-consumer-serviceđặc điểm kỹ thuật:

máy chủ: ftgo-consumer-service

Dịch vụ ảo

máy chủ:ftgo-người tiêu dùng-dịch vụ

Cổng API pod

tập hợp con:

-tên: v1nhãn:

Quy tắc đích

Phiên bản: v1

-tên: v2nhãn:

Phiên bản: v2

**Quy tắc định tuyến cho Dịch vụ Người tiêu dùng**

**Xác định các tập hợp con của các nhóm dịch vụ**

**Không có lưu lượng nào được định tuyến đến v2.**

siêu dữ liệu:

nhãn:

Dịch vụ tiêu dùng v2 pod

ứng dụng: ftgo-consumer-service

**Phiên bản: v2**

**Hình 12.12 Quy tắc định tuyến choDịch vụ khách hàng, định tuyến tất cả lưu lượng truy cập đến các pod v1. Nó bao gồm mộtDịch vụ ảo, những tuyến đường nàolưu lượng truy cập của nó đến tập hợp con v1 và mộtQuy tắc đích, định nghĩa tập hợp con v1 là các pod được gắn nhãn bằngphiên bản:v1. Sau khi bạn đã xác định quy tắc này, bạn có thể triển khai phiên bản mới một cách an toànmà không định tuyến bất kỳ lưu lượng truy cập nào đến đó ngay từ đầu.**

Hình 12.12 hiển thị quy tắc định tuyến choDịch vụ khách hàngđịnh tuyến tất cả lưu lượng truy cập đếnv1. Nó bao gồm hai đối tượng Istio: aDịch vụ ảovà mộtQuy tắc đích.

MỘTDịch vụ ảođịnh nghĩa cách định tuyến các yêu cầu cho một hoặc nhiều tên máy chủ. Trong nàyví dụ,Dịch vụ ảođịnh nghĩa các tuyến đường cho một tên máy chủ duy nhất:ftgo-người tiêu dùng-dịch vụ. Đây là định nghĩa củaDịch vụ ảovìDịch vụ khách hàng:

Phiên bản api:networking.istio.io/v1alpha3 loại: VirtualService

siêu dữ liệu:

tên: ftgo-consumer-service

đặc điểm kỹ thuật:

chủ nhà:

- ftgo-consumer-service http:

- tuyến đường:

- điểm đến:

**Áp dụng cho Dịch vụ Người tiêu dùng**

**Tuyến đường đến dịch vụ khách hàng**

máy chủ: ftgo-consumer-servicetập hợp con: v1

**Các v1tập hợp con**

Nó định tuyến tất cả các yêu cầu cho tập hợp con v1 của các pod của Consumer Service. Sau đó, tôi sẽ trình bày các ví dụ phức tạp hơn định tuyến dựa trên các yêu cầu HTTP và cân bằng tải trên nhiều đích có trọng số.

Ngoài raDịch vụ ảo, bạn cũng phải xác định mộtQuy tắc đích, định nghĩa một hoặc nhiều tập hợp con của pod cho một dịch vụ. Một tập hợp con của pod thường là một phiên bản dịch vụ. AQuy tắc đíchcũng có thể xác định chính sách giao thông, chẳng hạn như thuật toán cân bằng tải. Đây làQuy tắc đíchvìDịch vụ khách hàng:

Phiên bản api:networking.istio.io/v1alpha3 loại: DestinationRule

siêu dữ liệu:

tên: ftgo-consumer-service đặc điểm kỹ thuật:

máy chủ: ftgo-consumer-service

tập hợp con:

* tên: v1 nhãn:

Phiên bản: v1

* tên: v2 nhãn:

Phiên bản: v2

**Tên của tập hợp con**

**Bộ chọn pod cho tập hợp con**

Cái nàyQuy tắc đíchđịnh nghĩa hai tập hợp con của pod:v1Vàv2. Cácv1tập hợp con chọnvỏ có nhãnPhiên bản: v1. Cácv2tập hợp con chọn các pod có nhãnPhiên bản: v2. Sau khi bạn đã xác định các quy tắc này, Istio sẽ chỉ định tuyến các nhóm lưu lượng được gắn nhãnphiên bản:

v1. Bây giờ có thể triển khai v2 một cách an toàn.

**DPHIÊN BẢN SỬ DỤNG2CỦACNGƯỜI TIÊU DÙNGSDỊCH VỤ**

Đây là một đoạn trích của phiên bản 2Triển khaivìDịch vụ khách hàng:

apiVersion: extensions/v1beta1 loại: Triển khai

siêu dữ liệu:

tên: ftgo-consumer-service-v2 đặc điểm kỹ thuật:

bản sao: 1 mẫu:

siêu dữ liệu: nhãn:

ứng dụng: ftgo-consumer-service phiên bản: v2

**Phiên bản 2**

**Pod được dán nhãn với phiên bản**

...

Việc triển khai này được gọi làftgo-dịch-vụ-tiêu-dùng-v2. Nó dán nhãn các vỏ của nó vớiPhiên bản: v2. Sau khi tạo triển khai này, cả hai phiên bản củaftgo-dịch vụ-tiêu-dùngsẽđang chạy. Nhưng vì các quy tắc định tuyến, Istio sẽ không định tuyến bất kỳ lưu lượng nào đếnv2. Bây giờ bạn đã sẵn sàng để định tuyến một số lưu lượng thử nghiệm đếnv2.

**RKIỂM TRA LƯỢNG TRUY CẬP ĐẾN PHIÊN BẢN2**

Sau khi bạn triển khai phiên bản mới của dịch vụ, bước tiếp theo là kiểm tra nó. Giả sử các yêu cầu từ người dùng thử nghiệm có tiêu đề testuser. Chúng ta có thể nâng cao ftgo- consumer-service VirtualService để định tuyến các yêu cầu có tiêu đề này đến các phiên bản v2 bằng cách thực hiện thay đổi sau:

Phiên bản api:networking.istio.io/v1alpha3 loại: VirtualService

siêu dữ liệu:

tên: ftgo-consumer-service đặc điểm kỹ thuật:

chủ nhà:

- ftgo-consumer-service http:

- cuộc thi đấu:

- tiêu đề:

người dùng thử nghiệm: biểu thức chính quy: "^.+$"

**Phù hợp với mộttiêu đề testuser không trống**

tuyến đường:

- điểm đến:

máy chủ: ftgo-consumer-service tập hợp con: v2

- tuyến đường:

- điểm đến:

máy chủ: ftgo-consumer-servicetập hợp con: v1

**Các tuyến đường thử nghiệm người dùng đến v2**

**Định tuyến mọi người khác đến v1**

Ngoài tuyến đường mặc định ban đầu, VirtualService có một quy tắc định tuyến định tuyến các yêu cầu có tiêu đề testuser đến tập hợp con v2. Sau khi bạn đã cập nhậtrules, bây giờ bạn có thể kiểm tra Consumer Service. Sau đó, khi bạn cảm thấy tự tin rằng v2 đang hoạt động, bạn có thể định tuyến một số lưu lượng sản xuất đến đó. Hãy cùng xem cách thực hiện.

**RGIAO THÔNG SẢN XUẤT NGOÀI TRỜI ĐẾN PHIÊN BẢN2**

Sau khi bạn đã thử nghiệm một dịch vụ mới triển khai, bước tiếp theo là bắt đầu định tuyến lưu lượng sản xuất đến dịch vụ đó. Một chiến lược tốt là ban đầu chỉ định tuyến một lượng nhỏ lưu lượng. Ví dụ, đây là quy tắc định tuyến 95% lưu lượng đến v1 và 5% đến v2:

Phiên bản api:networking.istio.io/v1alpha3 loại: VirtualService

siêu dữ liệu:

tên: ftgo-consumer-service đặc điểm kỹ thuật:

chủ nhà:

* ftgo-dịch vụ-tiêu-dùnghttp:
  + tuyến đường:
    - điểm đến:

máy chủ: ftgo-consumer-service tập hợp con: v1

trọng lượng: 95

* + - điểm đến:

máy chủ: ftgo-consumer-service tập hợp con: v2

trọng lượng: 5

Khi bạn tin tưởng rằng dịch vụ có thể xử lý lưu lượng sản xuất, bạn có thể tăng dần lượng lưu lượng đến các pod phiên bản 2 cho đến khi đạt 100%. Tại thời điểm đó, Istio không định tuyến bất kỳ lưu lượng nào đến các pod v1. Bạn có thể để chúng chạy thêm một thời gian nữa trước khi xóa Triển khai phiên bản 1.

Bằng cách cho phép bạn dễ dàng tách biệt việc triển khai khỏi việc phát hành, Istio giúp việc triển khai trở nên dễ dàng

phiên bản mới của dịch vụ đáng tin cậy hơn nhiều. Tuy nhiên, tôi hầu như chưa khám phá hết khả năng của Istio. Tính đến thời điểm viết bài, phiên bản hiện tại của Istio là 0.8. Tôi rất mong chờ được chứng kiến ​​nó và các dịch vụ lưới khác phát triển và trở thành một phần tiêu chuẩn của môi trường sản xuất.

#### Triển khai dịch vụ bằng cách sử dụng triển khai Serverlessmẫu

Các mẫu đóng gói theo Ngôn ngữ cụ thể (phần 12.1), Dịch vụ dưới dạng VM (phần 12.2) và Dịch vụ dưới dạng vùng chứa (phần 12.3) đều khá khác nhau, nhưng chúng có chung một số đặc điểm. Đầu tiên là với cả ba mẫu, bạn phải hình dung trước một số tài nguyên điện toán—có thể là máy vật lý, máy ảo hoặc vùng chứa. Một số nền tảng triển khai triển khai tính năng tự động điều chỉnh, điều này sẽ điều chỉnh động số lượng VM hoặc vùng chứa dựa trên tải. Nhưng bạn sẽ luôn phải trả tiền cho một số VM hoặc vùng chứa, ngay cả khi chúng không hoạt động.

Một đặc điểm chung khác là bạn chịu trách nhiệm quản trị hệ thống. Nếu bạn đang chạy bất kỳ loại máy nào, bạn phải vá hệ điều hành. Trong trường hợp máy vật lý, điều này cũng bao gồm việc xếp chồng và xếp chồng. Bạn cũng chịu trách nhiệm quản lý thời gian chạy ngôn ngữ. Đây là một ví dụ về cái mà Amazon gọi là "nâng vật nặng không phân biệt". Kể từ những ngày đầu của máy tính, hệ thống

Quản trị là một trong những việc bạn cần làm. Nhưng hóa ra, có một giải pháp: không cần máy chủ.

###### Tổng quan về triển khai không cần máy chủ với AWS Lambda

Tại AWS Re:Invent 2014, Werner Vogels, CTO của Amazon, đã giới thiệu AWS Lambda với cụm từ tuyệt vời “phép màu xảy ra tại giao điểm của các chức năng, sự kiện và dữ liệu”. Như cụm từ này gợi ý, AWS Lambda ban đầu là để triển khai các dịch vụ theo sự kiện. Nó là “phép màu” vì, như bạn sẽ thấy, AWS Lambda là một ví dụ về công nghệ triển khai không cần máy chủ.

**Công nghệ triển khai không cần máy chủ**

Các đám mây công cộng chính đều cung cấp tùy chọn triển khai không cần máy chủ, mặc dù AWS Lambda là tiên tiến nhất. Google Cloud có các chức năng của Google Cloud, hiện đang trong giai đoạn thử nghiệm (<https://cloud.google.com/functions/>). Microsoft Azure có các chức năng Azure (<https://azure.microsoft.com/en-us/services/functions>).

Ngoài ra còn có các khuôn khổ không có máy chủ nguồn mở, chẳng hạn như Apache Openwhisk ([https://openwhisk.apache.org](https://openwhisk.apache.org/)) và Fission cho Kubernetes ([https://fission.io](https://fission.io/)), mà bạn có thể chạy trên cơ sở hạ tầng của riêng mình. Nhưng tôi không hoàn toàn tin tưởng vào giá trị của chúng.Bạn cần quản lý cơ sở hạ tầng chạy khung không máy chủ—điều này nghe có vẻ không chính xác*không có máy chủ*. Hơn nữa, như bạn sẽ thấy sau trong phần này, serverless cung cấp một mô hình lập trình bị hạn chế để đổi lấy việc quản trị hệ thống tối thiểu. Nếu bạn cần quản lý cơ sở hạ tầng, thì bạn có những hạn chế mà không có lợi ích.

AWS Lambda hỗ trợ Java, NodeJS, C#, GoLang và Python. Hàm lambda là dịch vụ không trạng thái. Nó thường xử lý các yêu cầu bằng cách gọi các dịch vụ AWS. Ví dụ, hàm lambda được gọi khi hình ảnh được tải lên thùng S3 có thể chèn một mục vào bảng IMAGES của DynamoDB và xuất bản thông báo tới Kinesis để kích hoạt xử lý hình ảnh. Hàm lambda cũng có thể gọi các dịch vụ web của bên thứ ba.

Để triển khai một dịch vụ, bạn đóng gói ứng dụng của mình dưới dạng tệp ZIP hoặc tệp JAR, tải lên AWS Lambda và chỉ định tên hàm để gọi để xử lý yêu cầu (còn gọi là sự kiện). AWS Lambda tự động chạy đủ số lượng phiên bản của dịch vụ vi mô của bạn để xử lý các yêu cầu đến. Bạn được tính phí cho mỗi yêu cầu dựa trên thời gian thực hiện và bộ nhớ đã sử dụng. Tất nhiên, vấn đề nằm ở chi tiết và sau này bạn sẽ thấy rằng AWS Lambda có những hạn chế. Nhưng quan niệm rằng bạn với tư cách là nhà phát triển hay bất kỳ ai trong tổ chức của bạn không cần phải lo lắng về bất kỳ khía cạnh nào của máy chủ, máy ảo hoặc vùng chứa là vô cùng mạnh mẽ.

**Mẫu: Triển khai không có máy chủ**

Triển khai các dịch vụ bằng cơ chế triển khai không cần máy chủ do đám mây công cộng cung cấp. Xem<http://microservices.io/patterns/deployment/serverless-deployment.html>.

###### Phát triển một hàm lambda

Không giống như khi sử dụng ba mẫu khác, bạn phải sử dụng một mô hình lập trình khác cho các hàm lambda của mình. Mã hàm lambda và cách đóng gói phụ thuộc vào ngôn ngữ lập trình. Hàm lambda Java là một lớp triển khai giao diện chung RequestHandler, được định nghĩa bởi thư viện lõi Java của AWS Lambda và được hiển thị trong danh sách sau. Giao diện này có hai tham số kiểu: I, là kiểu đầu vào và O, là kiểu đầu ra. Kiểu của I và O phụ thuộc vào loại yêu cầu cụ thể mà lambda xử lý.

**Liệt kê 12.8 Một hàm lambda Java là một lớp thực hiệnTrình xử lý yêu cầu**

**giao diện.**

giao diện công khai RequestHandler<I, O> {

public O handleRequest(I input, Context context);

}

Giao diện RequestHandlerđịnh nghĩa một phương thức handleRequest() duy nhất. Phương thức này có hai tham số, một đối tượng đầu vào và một ngữ cảnh, cung cấp quyền truy cập vào môi trường thực thi lambda, chẳng hạn như ID yêu cầu. Phương thức handleRequest() trả về một đối tượng đầu ra. Đối với các hàm lambda xử lý các yêu cầu HTTP được ủy quyền bởi AWS API Gateway, I và O lần lượt là APIGatewayProxyRequestEvent và APIGatewayProxyResponseEvent. Như bạn sẽ sớm thấy, các hàm xử lý khá giống với các servlet Java EE kiểu cũ.

Java lambda được đóng gói dưới dạng tệp ZIP hoặc tệp JAR. Tệp JAR là uber JAR (hoặc fat JAR) được tạo bởi, ví dụ, plugin Maven Shade. Tệp ZIP có các lớp trong thư mục gốc và các phụ thuộc JAR trong thư mục lib. Sau đó, tôi sẽ chỉ cho bạn cách một dự án Gradle có thể tạo tệp ZIP. Nhưng trước tiên, hãy xem các cách khác nhau để gọi hàm lambda.

###### Gọi hàm lambda

Có bốn cách để gọi hàm lambda:

* + - * Yêu cầu HTTP
      * Sự kiện được tạo ra bởi dịch vụ AWS
      * Các lệnh gọi theo lịch trình
      * Sử dụng trực tiếp lệnh gọi API Chúng ta hãy cùng xem xét từng lệnh.

**HCÁ TRĂNGiao thức HTTPYÊU CẦU**

Một cách để gọi hàm lambda là cấu hình AWS API Gateway để định tuyến các yêu cầu HTTP đến lambda của bạn. API Gateway sẽ hiển thị hàm lambda của bạn như một điểm cuối HTTPS. Nó hoạt động như một proxy HTTP, gọi hàm lambda bằng một đối tượng yêu cầu HTTP và mong đợi hàm lambda trả về một đối tượng phản hồi HTTP. Bằng cách sử dụng API Gateway với AWS Lambda, bạn có thể triển khai các dịch vụ RESTful dưới dạng các hàm lambda.

**HSỰ KIỆN ANDLINGĐƯỢC TẠO RA BỞIAWSDỊCH VỤ**

Cách thứ hai để gọi hàm lambda là cấu hình hàm lambda của bạn để xử lý các sự kiện do dịch vụ AWS tạo ra. Ví dụ về các sự kiện có thể kích hoạt hàm lambda bao gồm:

* Một đối tượng được tạo trong thùng S3.
* Một mục được tạo, cập nhật hoặc xóa trong bảng DynamoDB.
* Có thể đọc tin nhắn từ luồng Kinesis.
* Email được nhận qua dịch vụ email đơn giản.

Nhờ tích hợp với các dịch vụ AWS khác nên AWS Lambda hữu ích cho nhiều tác vụ khác nhau.

**DXÁC ĐỊNH CÁC HÀM LAMBDA ĐƯỢC LỊCH TRÌNH**

Một cách khác để gọi hàm lambda là sử dụng lịch trình giống như cron của Linux. Bạn có thể cấu hình hàm lambda của mình để được gọi định kỳ—ví dụ, mỗi phút,3 giờ hoặc 7 ngày. Ngoài ra, bạn có thể sử dụng biểu thức cron để chỉ định thời điểm AWS nên gọi lambda của bạn. Biểu thức cron mang lại cho bạn sự linh hoạt to lớn. Ví dụ, bạn có thể cấu hình lambda để gọi lúc 2:15 chiều từ Thứ Hai đến Thứ Sáu.

**TÔIGỌI HÀM LAMBDA BẰNG YÊU CẦU DỊCH VỤ WEB**

Cách thứ tư để gọi hàm lambda là ứng dụng của bạn gọi hàm đó bằng yêu cầu dịch vụ web. Yêu cầu dịch vụ web chỉ định tên hàm lambda và dữ liệu sự kiện đầu vào. Ứng dụng của bạn có thể gọi hàm lambda đồng bộ hoặc không đồng bộ. Nếu ứng dụng của bạn gọi hàm lambda đồng bộ, phản hồi HTTP của dịch vụ web sẽ chứa phản hồi của hàm lambda. Ngược lại, nếu ứng dụng gọi hàm lambda không đồng bộ, phản hồi của dịch vụ web sẽ chỉ ra liệu việc thực thi lambda có được khởi tạo thành công hay không.

###### Lợi ích của việc sử dụng hàm lambda

Việc triển khai các dịch vụ bằng hàm lambda có một số lợi ích:

* + - * *Được tích hợp với nhiều dịch vụ AWS*—Thật dễ dàng để viết lambda sử dụng các sự kiện do các dịch vụ AWS tạo ra, chẳng hạn như DynamoDB và Kinesis, và xử lý các yêu cầu HTTP thông qua AWS API Gateway.
      * *Loại bỏ nhiều tác vụ quản trị hệ thống*—Bạn không còn chịu trách nhiệm quản trị hệ thống cấp thấp nữa. Không có hệ điều hành hoặc thời gian chạy nào để vá. Do đó, bạn có thể tập trung vào việc phát triển ứng dụng của mình.
      * *Độ đàn hồi*—AWS Lambda chạy nhiều phiên bản của bạnứng dụng cần thiết để xử lý tải. Bạn không phải đối mặt với thách thức dự đoán dung lượng cần thiết hoặc có nguy cơ cung cấp quá mức hoặc thiếu hụt VM hoặc container.
      * *Giá dựa trên mức sử dụng*—Không giống như đám mây IaaS thông thường, tính phí theo phút hoặc giờ cho máy ảo hoặc vùng chứa ngay cả khi không hoạt động, AWS Lambda chỉ tính phí cho các tài nguyên được sử dụng trong khi xử lý từng yêu cầu.

###### Nhược điểm của việc sử dụng hàm lambda

Như bạn có thể thấy, AWS Lambda là một cách cực kỳ tiện lợi để triển khai dịch vụ, nhưng có một số nhược điểm và hạn chế đáng kể:

* + - * *Độ trễ đuôi dài*—Vì AWS Lambda chạy mã của bạn theo kiểu động, một số yêu cầu có độ trễ cao do thời gian AWS cung cấp phiên bản ứng dụng của bạn và thời gian ứng dụng khởi động. Điều này đặc biệt khó khăn khi chạy các dịch vụ dựa trên Java vì chúng thường mất ít nhất vài giây để khởi động. Ví dụ, hàm lambda mẫu được mô tả trong phần tiếp theo mất một thời gian để khởi động. Do đó, AWS Lambda có thể không phù hợp với các dịch vụ nhạy cảm với độ trễ.
      * *Mô hình lập trình dựa trên sự kiện/yêu cầu hạn chế*—AWS Lambda không có ý địnhđược sử dụng để triển khai các dịch vụ chạy dài hạn, chẳng hạn như dịch vụ sử dụng tin nhắn từ một nhà môi giới tin nhắn của bên thứ ba.

Do những nhược điểm và hạn chế này, AWS Lambda không phù hợp với tất cả các dịch vụ. Nhưng khi chọn mô hình triển khai, tôi khuyên bạn nên đánh giá trước xem triển khai không máy chủ có hỗ trợ các yêu cầu của dịch vụ của bạn hay không trước khi xem xét các giải pháp thay thế.

#### Triển khai dịch vụ RESTful bằng AWS Lambdavà AWS Gateway

Hãy cùng xem cách triển khai Restaurant Service bằng AWS Lambda. Đây là dịch vụ có REST API để tạo và quản lý nhà hàng. Ví dụ, dịch vụ này không có kết nối lâu dài với Apache Kafka, do đó, dịch vụ này phù hợp với AWS lambda. Hình 12.13 cho thấy kiến ​​trúc triển khai cho dịch vụ này. Dịch vụ bao gồm một số hàm lambda, mỗi hàm cho một điểm cuối REST. AWS API Gateway chịu trách nhiệm định tuyến các yêu cầu HTTP đến các hàm lambda.

Mỗi hàm lambda có một lớp xử lý yêu cầu.ftgo-create-nhà-hànghàm lambda gọiTạo Trình Xử Lý Yêu Cầu Nhà Hànglớp học vàftgo-tìm-nhà-hànghàm lambda gọiTìm Trình Xử Lý Yêu Cầu Nhà Hàng. Bởi vì các lớp trình xử lý yêu cầu này triển khai các khía cạnh có liên quan chặt chẽ của cùng một dịch vụ,chúng được đóng gói cùng nhau trong cùng một tệp ZIP,dịch vụ nhà hàng-aws-lambda

.zip. Hãy cùng xem xét thiết kế của dịch vụ, bao gồm các lớp xử lý đó.

###### Thiết kế phiên bản AWS Lambda của Restaurant Service

Kiến trúc của dịch vụ, được thể hiện trong hình 12.14, khá giống với kiến ​​trúc của dịch vụ truyền thống. Sự khác biệt chính là bộ điều khiển Spring MVC đã được thay thế bằng các lớp trình xử lý yêu cầu AWS Lambda. Phần còn lại của logic kinh doanh không thay đổi.

Dịch vụ bao gồm một bài thuyết trìnhtầng bao gồm các trình xử lý yêu cầu, được AWS Lambda gọi để xử lý các yêu cầu HTTP và một doanh nghiệp truyền thống

Cổng API



POST/nhà hàng

NHẬN/nhà hàng/

{mã nhà hàng}

AWS Lambda

chức năng

ftgo-create-nhà-hàng

ftgo-tìm-nhà-hàng

ftgo-...

ftgo-nhà hàng-service-aws-lambda.zip

"lớp học"

...

Trình xử lý yêu cầu

«lớp» Trình xử lý yêu cầu FindRestaurant

«lớp» Tạo Trình xử lý yêu cầu nhà hàng

**Hình 12.13 Triển khaiDịch vụ nhà hàngnhư các hàm AWS Lambda. AWS API Gateway định tuyến các yêu cầu HTTP đến các hàm AWS Lambda, được triển khai bởi các lớp trình xử lý yêu cầu được xác định bởiDịch vụ nhà hàng.**

POST/nhà hàngGET/nhà hàng/{restaurantId}

Lớp trình bày

...

Trình xử lý yêu cầu

Tìm Trình xử lý yêu cầu nhà hàng

Tạo Trình xử lý yêu cầu nhà hàng

Lớp truy cập dữ liệu và kinh doanh

**Hình 12.14 Thiết kế dựa trên AWS LambdaDịch vụ nhà hàng. Lớp trình bày bao gồm các lớp trình xử lý yêu cầu, thực hiện các hàm lambda. Chúng gọi tầng nghiệp vụ, được viết theo phong cách truyền thốngbao gồm một lớp dịch vụ, một thực thể và một kho lưu trữ.**

Nhà hàngRepository

Nhà hàngDịch vụ

Nhà hàng

tầng. Tầng kinh doanh bao gồmNhà hàngDịch vụ, cácNhà hàngThực thể JPA,Và

Nhà hàngRepository, đóng gói cơ sở dữ liệu.

Chúng ta hãy cùng xem xétTìm Trình Xử Lý Yêu Cầu Nhà Hànglớp học.

**TANH TAFĐỘC LẬPRNHÀ HÀNGRYÊU CẦUHLỚP ANDLER**

CácTìm Trình Xử Lý Yêu Cầu Nhà Hànglớp thực hiệnNHẬN /nhà hàng/

Điểm cuối {restaurantId}. Lớp này cùng với yêu cầu kháclớp xử lý là các lá của hệ thống phân cấp lớp được hiển thị trong hình 12.15. Gốc của hệ thống phân cấp là RequestHandler, là một phần của AWS SDK. Các lớp con trừu tượng của nó xử lý lỗi và inject các phụ thuộc.

|  |  |
| --- | --- |
| Trình xử lý yêu cầu | |
|  |  |
| Tóm tắt HttpHandler | |
|  |  |
| Tóm tắt Autowiring HttpRequest Handler | |
|  |  |
| Tìm Trình xử lý yêu cầu nhà hàng | |







**Hình 12.15 Thiết kế trình xử lý yêu cầuCác lớp siêu trừu tượng thực hiệntiêm phụ thuộc và xử lý lỗi.**

Tạo Trình xử lý yêu cầu nhà hàng

...

Trình xử lý yêu cầu

CácTóm tắtHttpHandlerlớp là lớp cơ sở trừu tượng cho trình xử lý yêu cầu HTTP. Nó bắt các ngoại lệ chưa được xử lý được ném ra trong quá trình xử lý yêu cầu và trả về một500 - lỗi máy chủ nội bộphản hồi. CácTóm tắtAutowiringHttpRequestHandlerlớp thực hiện tiêm phụ thuộc cho trình xử lý yêu cầu. Tôi sẽ mô tả những điều trừu tượng nàysiêu lớp trong thời gian ngắn, nhưng trước tiên chúng ta hãy xem mã choTìm Trình Xử Lý Yêu Cầu Nhà Hàng.Danh sách 12.9 hiển thị mã choTìm Trình Xử Lý Yêu Cầu Nhà Hànglớp học.Tìm Trình Xử Lý Yêu Cầu Nhà Hànglớp học có mộtxử lýHttpRequest()phương pháp, màmất mộtSự kiện APIGatewayProxyRequestđại diện cho một yêu cầu HTTP như một tham số. Nó gọiNhà hàngDịch vụđể tìm nhà hàng và trả về mộtAPIGateway-

Sự kiện ProxyResponsemô tả phản hồi HTTP.

**Liệt kê 12.9 Lớp trình xử lý choLẤY/nhà hàng/{mã nhà hàng}**

lớp công khai FindRestaurantRequestHandler

mở rộng AbstractAutowiringHttpRequestHandler {

@Autowired

Nhà hàng riêngDịch vụ nhà hàngDịch vụ;

@Ghi đè

Lớp được bảo vệ<?> getApplicationContextClass() { trả về CreateRestaurantRequestHandler.class;

**Lớp cấu hình Spring Java để sử dụng cho ngữ cảnh ứng dụng**

}

@Ghi đè

APIGatewayProxyResponseEvent được bảo vệ handleHttpRequest(APIGatewayProxyRequestEvent request, Context context) {

nhà hàng dàiId;thử {

restaurantId = Long.parseLong(request.getPathParameters()

.get("mã nhà hàng"));

} catch (NumberFormatException e) { return makeBadRequestResponse(bối cảnh);

}

**Trả lại400 - phản hồi yêu cầu không hợp lệ nếu restaurantId bị thiếu hoặc không hợp lệ**

Tùy chọn<Nhà hàng> possibleRestaurant = restaurantService.findById(nhà hàng

mã ant);

có thể trả lạiNhà hàng

.map(this::makeGetRestaurantResponse)

**Trả về nhà hàng hoặc phản hồi 404 - không tìm thấy**

.orElseGet(() -> makeRestaurantNotFoundResponse(bối cảnh,

nhà hàngId));

}

APIGatewayProxyResponseEvent riêng tư makeBadRequestResponse(Context context) {

...

}

APIGatewayProxyResponseEvent riêng tư

makeRestaurantNotFoundResponse(Bối cảnh ngữ cảnh, restaurantId dài) { ... }

APIGatewayProxyResponseEvent riêng tư

makeGetRestaurantResponse(Nhà hàng nhà hàng) { ... }

}

Như bạn có thể thấy, nó khá giống với một servlet, ngoại trừ việc thay vì mộtdịch vụ()phương pháp, trong đó có một Yêu cầu HttpServletVà trả lại Phản hồi HttpServlet, nó có mộtxử lýHttpRequest(), trong đó có mộtSự kiện APIGatewayProxyRequestvà trả vềSự kiện phản hồi Proxy của APIGateway.

Bây giờ chúng ta hãy xem xét siêu lớp của nó, nơi triển khai tính năng tiêm phụ thuộc.

**DTIÊM PHỤ THUỘC SỬ DỤNGMỘTTÓM TẮTMỘTUTWIRING (DÂY CHUYỀN)HTTPRYÊU CẦUHLỚP ANDLER**

Chức năng AWS Lambda không phải là ứng dụng web cũng không phải là ứng dụng cóchủ yếu()phương pháp. Nhưng sẽ thật đáng tiếc nếu không thể sử dụng các tính năng của SpringChiếc giày mà chúng ta đã quen thuộc.Tóm tắtAutowiringHttpRequestHandlerlớp, được hiển thị trong danh sách sau, thực hiện việc tiêm phụ thuộc cho trình xử lý yêu cầu.Nó tạo ra mộtỨng dụngBối cảnhsử dụngSpringApplication. chạy()Vàautowires các phụ thuộc trước khi xử lý yêu cầu đầu tiên. Các lớp con nhưFindRestaurant- RequestHandlerphải thực hiệnlấy ApplicationContextClass()phương pháp.

**Liệt kê 12.10 Một bản tóm tắtTrình xử lý yêu cầuthực hiện tiêm phụ thuộc**

lớp trừu tượng công khai AbstractAutowiringHttpRequestHandlermở rộng AbstractHttpHandler {

riêng tư tĩnh ConfigurableApplicationContext ctx;

riêng tư ReentrantReadWriteLockctxLock = khóa ReentrantReadWriteLock mới();

boolean riêng tư autowired = false;

được bảo vệ đồng bộ ApplicationContext getAppCtx() { ctxLock.writeLock().lock();

thử {

nếu (ctx == null) {

**Tạo ngữ cảnh ứng dụng Spring Boot chỉ một lần**

ctx = SpringApplication.run(getApplicationContextClass());

}

trả về ctx;

} cuối cùng { ctxLock.writeLock().unlock();

}

}

**Chèn các phụ thuộc vào trình xử lý yêu cầu bằng cách sử dụng autowiring trước khi xử lý**

@Ghi đè void được bảo vệ

**yêu cầu đầu tiên**

trước khi Xử lý (APIGatewayProxyRequestEventyêu cầu, Bối cảnh ngữ cảnh) { super.beforeHandling(yêu cầu, bối cảnh);

nếu (!autowired) { getAppCtx().getAutowireCapableBeanFactory().autowireBean(cái này);

autowired = đúng;

} **Trả về @Configuration**

} **lớp được sử dụng để tạo**

**Ứng dụngBối cảnh**

lớp trừu tượng được bảo vệ<?> getApplicationContextClass();

}

Lớp này ghi đè lêntrước khi xử lý()phương pháp được xác định bởiTóm tắtHttpHandler. Của nótrước khi xử lý()phương pháp tiêm các phụ thuộc bằng cách sử dụng autowiring trước khi xử lýyêu cầu đầu tiên.

**TANH TAMỘTTÓM TẮTHTTPHLỚP ANDLER**

Trình xử lý yêu cầu choDịch vụ nhà hàngcuối cùng mở rộngTóm tắtHttpHandler,được hiển thị trong danh sách 12.11. Lớp này thực hiệnRequestHandler<APIGatewayProxy- RequestEventVàSự kiện phản hồi của APIGatewayProxy>. Trách nhiệm chính của nó là phát hiện các ngoại lệ khi xử lý yêu cầu và đưa ra mã lỗi 500.

**Danh sách 12.11 MỘT tóm tắt Trình xử lý yêu cầucái đó bắt ngoại lệ Và trả lạiphản hồi HTTP 500**

lớp trừu tượng công khai AbstractHttpHandler triển khai RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

Nhật ký Logger riêng tư = LoggerFactory.getLogger(this.getClass());@Ghi đè

công khai APIGatewayProxyResponseEvent handleRequest(

Đầu vào APIGatewayProxyRequestEvent, Bối cảnh ngữ cảnh) {log.debug("Đã nhận được yêu cầu: {}", đầu vào);

thử {

beforeHandling(đầu vào, ngữ cảnh);

trả về handleHttpRequest(đầu vào, ngữ cảnh);

} catch (Ngoại lệ e) {

log.error("Lỗi khi xử lý yêu cầu id: {}", context.getAwsRequestId(), e);trả về buildErrorResponse(new AwsLambdaError(

"Lỗi máy chủ nội bộ","500",

ngữ cảnh.getAwsRequestId(),

"Lỗir xử lý yêu cầu: " + context.getAwsRequestId()+""

+ đầu vào.toString()));

}

}

được bảo vệ void beforeHandling(APIGatewayProxyRequestEvent yêu cầu,Bối cảnh bối cảnh) {

// không làm gì cả

}

trừu tượng được bảo vệ APIGatewayProxyResponseEvent handleHttpRequest(Yêu cầu APIGatewayProxyRequestEvent, Context context);

}

###### Đóng gói dịch vụ nhưTập tin ZIP

Trước khi dịch vụ có thể được triển khai, chúng ta phải đóng gói nó dưới dạng tệp ZIP. Chúng ta có thể dễ dàng xây dựng tệp ZIP bằng tác vụ Gradle sau:

nhiệm vụ buildZip(type: Zip) { từ compileJava

từ processResources vào('lib') {

từ configurations.runtime

}

}

Nhiệm vụ này xây dựng một ZIP với các lớp và tài nguyên ở cấp cao nhất và các phụ thuộc JAR trong thư mục lib.

Bây giờ chúng ta đã xây dựng tệp ZIP, hãy xem cách triển khaihàm lambda.

###### Triển khai các hàm lambda bằng cách sử dụng khung Serverless

Sử dụng các công cụ do AWS cung cấp để triển khai các hàm lambda và cấu hình API gateway khá là tẻ nhạt. May mắn thay, dự án mã nguồn mở Serverless giúp việc sử dụng các hàm lambda dễ dàng hơn rất nhiều. Khi sử dụng Serverless, bạn viết một tệp server-less.yml đơn giản để định nghĩa các hàm lambda và các điểm cuối RESTful của chúng. Sau đó, Serverless triển khai các hàm lambda và tạo và cấu hình một API gateway định tuyến các yêu cầu đến chúng.

Sau đây làdanh sách là một đoạn trích củakhông có máy chủ.ymltriển khaiDịch vụ nhà hàngdưới dạng lambda.

**Liệt kê 12.12không có máy chủ.ymltriển khaiDịch vụ nhà hàng.**

dịch vụ: ftgo-application-lambda

nhà cung cấp: tên: aws

thời gian chạy: java8 thời gian chờ: 35

vùng: ${env:AWS\_REGION} giai đoạn: dev

môi trường:

**Yêu cầu serverless triển khai trên AWS**

**Cung cấp cấu hình bên ngoài của dịch vụ thông qua các biến môi trường**

SPRING\_DATASOURCE\_DRIVER\_CLASS\_NAME:com.mysql.jdbc.Driver NGUỒN DỮ LIỆU SPRING: ...

SPRING\_DATASOURCE\_USERNAME: ... SPRING\_DATASOURCE\_PASSWORD: ...

bưu kiện:

**Tệp ZIP chứa các hàm lambda**

hiện vật: ftgo-restaurant-service-aws-lambda/build/distributions/ftgo-nhà hàng-service-aws-lambda.zip

**Định nghĩa hàm Lambda bao gồm trình xử lý**

chức năng:

tạo nhà hàng:

**chức năng và điểm cuối HTTP**

người xử lý:net.chrisrichardson.ftgo.restaurantservice.lambda

.CreateRestaurantRequestHandlersự kiện:

- http:

đường dẫn: nhà hàng phương pháp: bài đăng

tìm-nhà-hàng:

người xử lý:net.chrisrichardson.ftgo.restaurantservice.lambda

.Tìm Trình Xử Lý Yêu Cầu Nhà Hàngsự kiện:

- http:

đường dẫn: nhà hàng/{restaurantId}phương pháp: lấy

Bạnsau đó có thể sử dụngtriển khai không cần máy chủyêu cầu,mà đọcserverless.yml

tệp, triển khai các hàm lambda và cấu hình AWS API Gateway. Sau một thời gian ngắn

chờ đã, dịch vụ của bạn sẽ có thể truy cập được thông qua URL điểm cuối của cổng API. AWS Lambdasẽ cung cấp nhiều phiên bản của mỗi hàm lambda Restaurant Service cần thiết để hỗ trợ tải. Nếu bạn thay đổi mã, bạn có thể dễ dàng cập nhật lambda bằng cách xây dựng lại tệp ZIP và chạy lại triển khai không có máy chủ. Không liên quan đến máy chủ!

Sự tiến hóa của cơ sở hạ tầng là đáng chú ý. Cách đây không lâu, chúng ta đã triển khai thủ công các ứng dụng trên máy vật lý. Ngày nay, các đám mây công cộng tự động hóa cao cung cấp một loạt các tùy chọn triển khai ảo. Một tùy chọn là triển khai các dịch vụ dưới dạng máy ảo. Hoặc tốt hơn nữa, chúng ta có thể đóng gói các dịch vụ dưới dạng các container và triển khai chúng bằng các khuôn khổ phối hợp Docker tinh vi như Kubernetes. Đôi khi chúng ta thậm chí còn tránh nghĩ về cơ sở hạ tầng hoàn toàn và triển khai các dịch vụ dưới dạng các hàm lambda nhẹ, phù du.

#### Bản tóm tắt

* Bạn nên chọn mô hình triển khai nhẹ nhất hỗ trợ các yêu cầu của dịch vụ. Đánh giá các tùy chọn theo thứ tự sau: không có máy chủ, vùng chứa, máy ảo và các gói ngôn ngữ cụ thể.
* Triển khai không có máy chủ không phù hợp với mọi dịch vụ vì độ trễ dài và yêu cầu sử dụng mô hình lập trình dựa trên sự kiện/yêu cầu. Tuy nhiên, khi phù hợp, triển khai không có máy chủ là một lựa chọn cực kỳ hấp dẫn vì nó loại bỏ nhu cầu quản lý hệ điều hành và thời gian chạy, đồng thời cung cấp khả năng cung cấp đàn hồi tự động và giá dựa trên yêu cầu.
* Docker container, là công nghệ ảo hóa nhẹ, cấp hệ điều hành, linh hoạt hơn so với triển khai không cần máy chủ và có độ trễ có thể dự đoán được. Tốt nhất là sử dụng một khuôn khổ điều phối Docker như Kubernetes, quản lý các container trên một cụm máy. Nhược điểm của việc sử dụng container là bạn phải quản lý hệ điều hành và thời gian chạy và rất có thể là khuôn khổ điều phối Docker và các VM mà nó chạy trên đó.
* Tùy chọn triển khai thứ ba là triển khai dịch vụ của bạn dưới dạng máy ảo. Một mặt, máy ảo là tùy chọn triển khai nặng, do đó triển khai chậm hơn và có khả năng sử dụng nhiều tài nguyên hơn tùy chọn thứ hai. Mặt khác, các đám mây hiện đại như Amazon EC2 được tự động hóa cao và cung cấp một bộ tính năng phong phú. Do đó, đôi khi có thể dễ triển khai một ứng dụng nhỏ, đơn giản bằng máy ảo hơn là thiết lập khuôn khổ phối hợp Docker.
* Việc triển khai các dịch vụ của bạn dưới dạng các gói ngôn ngữ cụ thể thường là tốt nhất trừ khi bạn chỉ có một số lượng nhỏ các dịch vụ. Ví dụ, như đã mô tả trong chương 13, khi bắt đầu hành trình đến các dịch vụ vi mô, bạn có thể sẽ triển khai các dịch vụ bằng cùng một cơ chế mà bạn sử dụng cho ứng dụng đơn khối của mình, rất có thể là tùy chọn này. Bạn chỉ nên cân nhắc thiết lập

***Bản tóm tắt* 427**

xây dựng cơ sở hạ tầng triển khai phức tạp như Kubernetes sau khi bạn đã phát triển một số dịch vụ.

* + Một trong nhiều lợi ích của việc sử dụng lưới dịch vụ—một lớp mạng làm trung gian cho tất cả lưu lượng mạng vào và ra khỏi các dịch vụ—là nó cho phép bạn triển khai một dịch vụ trong sản xuất, thử nghiệm dịch vụ đó và chỉ sau đó mới định tuyến lưu lượng sản xuất đến dịch vụ đó. Việc tách biệt triển khai khỏi phát hành giúp cải thiện độ tin cậy khi triển khai các phiên bản dịch vụ mới.

*Tái cấu trúc thànhdịch vụ vi mô*

***Chương này bao gồm***

* Khi nào di chuyển một ứng dụng độc lập sangKiến trúc dịch vụ vi mô
* Tại sao sử dụng phương pháp gia tăng là điều cần thiết khi tái cấu trúc ứng dụng đơn khối thành dịch vụ vi mô
* Thực hiệncác tính năng mới như dịch vụ
* Trích xuất các dịch vụ từ khối đơn
* Tích hợp dịch vụ và khối đơn

Tôi hy vọng rằng cuốn sách này đã cung cấp cho bạn hiểu biết tốt về kiến ​​trúc microservice, lợi ích và hạn chế của nó, và khi nào nên sử dụng nó. Tuy nhiên, có một khả năng khá cao là bạn đang làm việc trên một ứng dụng đơn khối lớn, phức tạp. Trải nghiệm hàng ngày của bạn khi phát triển và triển khai ứng dụng của mình rất chậm và đau đớn. Microservice, có vẻ phù hợp với ứng dụng của bạn, có vẻ như là cõi niết bàn xa xôi. Giống như Mary và những người còn lại trong nhóm phát triển FTGO, bạn đang tự hỏi làm thế nào bạn có thể áp dụng kiến ​​trúc microservice?

May mắn thay, có những chiến lược bạn có thể sử dụng để thoát khỏi địa ngục đơn khối mà không cần phải viết lại ứng dụng của mình từ đầu. Bạn chuyển đổi dần dần

**428**

monolith của bạn thành các dịch vụ vi mô bằng cách phát triển cái được gọi là ứng dụng strangler. Ý tưởng về ứng dụng strangler xuất phát từ cây nho strangler, mọc trong rừng mưa bằng cách bao phủ và đôi khi giết chết cây. Ứng dụng strangler là ứng dụng mới bao gồm các dịch vụ vi mô mà bạn phát triển bằng cách triển khai chức năng mới dưới dạng dịch vụ và trích xuất các dịch vụ từ monolith. Theo thời gian, khi ứng dụng strangler triển khai ngày càng nhiều chức năng, nó sẽ co lại và cuối cùng giết chết monolith. Một lợi ích quan trọng của việc phát triển ứng dụng strangler là, không giống như việc viết lại big bang, nó mang lại giá trị cho doanh nghiệp sớm và thường xuyên.

Tôi bắt đầu chương này bằng cách mô tả động lực để tái cấu trúc một khối đơn thành kiến ​​trúc vi dịch vụ. Sau đó, tôi mô tả cách phát triển ứng dụng strangler bằng cách triển khai chức năng mới dưới dạng dịch vụ và trích xuất dịch vụ từ khối đơn. Tiếp theo, tôi đề cập đến nhiều chủ đề thiết kế khác nhau, bao gồm cách tích hợp khối đơn và dịch vụ, cách duy trì tính nhất quán của cơ sở dữ liệu trên toàn khối đơn và dịch vụ, và cách xử lý bảo mật. Tôi kết thúc chương bằng cách mô tả một sốdịch vụ ví dụ. Một dịch vụ là Delayed Order Service, triển khai chức năng hoàn toàn mới. Dịch vụ còn lại là Delivery Service, được trích xuất từ ​​khối đơn. Chúng ta hãy bắt đầu bằng cách xem xét khái niệm tái cấu trúc thành kiến ​​trúc dịch vụ vi mô.

#### Tổng quan về việc tái cấu trúc thành các dịch vụ vi mô

Hãy đặt mình vào vị trí của Mary. Bạn chịu trách nhiệm cho ứng dụng FTGO, một ứng dụng đơn khối lớn và cũ. Doanh nghiệp vô cùng thất vọng với việc kỹ thuật không thể cung cấp các tính năng một cách nhanh chóng và đáng tin cậy. FTGO dường như đang phải chịu đựng một trường hợp kinh điển của địa ngục đơn khối. Các dịch vụ vi mô dường như, ít nhất là trên bề mặt, là câu trả lời. Bạn có nên đề xuất chuyển hướng các nguồn lực phát triển khỏi việc phát triển tính năng để di chuyển sang kiến ​​trúc dịch vụ vi mô không?

Tôi bắt đầu phần này bằng cách thảo luận về lý do tại sao bạn nên cân nhắc việc tái cấu trúc thành các dịch vụ siêu nhỏ. Tôi cũng thảo luận về lý do tại sao điều quan trọng là phải chắc chắn rằng các vấn đề phát triển phần mềm của bạn là do bạn đang ở trong địa ngục đơn khối chứ không phải là do, ví dụ, một quy trình phát triển phần mềm kém. Sau đó, tôi mô tả các chiến lược để tái cấu trúc từng phần kiến ​​trúc đơn khối của bạn thành kiến ​​trúc dịch vụ siêu nhỏ. Tiếp theo, tôi thảo luận về tầm quan trọng của việc cung cấp các cải tiến sớm hơn và thường xuyên để duy trì sự hỗ trợ của doanh nghiệp. Sau đó, tôi mô tả lý do tại sao bạn nên tránh đầu tư vào cơ sở hạ tầng triển khai phức tạp cho đến khi bạn đã phát triển một vài dịch vụ. Cuối cùng, tôi mô tả các chiến lược khác nhau mà bạn có thể sử dụng để đưa các dịch vụ vào kiến ​​trúc của mình, bao gồm triển khai các tính năng mới dưới dạng dịch vụ và trích xuất các dịch vụ từ khối đơn khối.

###### Tại sao phải tái cấu trúc một khối thống nhất?

Kiến trúc microservice có nhiều lợi ích như đã mô tả trong chương 1. Kiến trúc này có khả năng bảo trì, khả năng kiểm tra và khả năng triển khai tốt hơn nhiều, do đó, nó đẩy nhanh quá trình phát triển. Kiến trúc microservice có khả năng mở rộng hơn và cải thiện khả năng cô lập lỗi. Việc phát triển ngăn xếp công nghệ của bạn cũng dễ dàng hơn nhiều. Nhưng việc tái cấu trúc một khối đơn

microservices là một nỗ lực quan trọng. Nó sẽ chuyển hướng nguồn lực khỏi việc phát triển tính năng mới. Do đó, có khả năng doanh nghiệp sẽ chỉ hỗ trợ việc áp dụng microservices nếu nó giải quyết được một vấn đề kinh doanh quan trọng.

Nếu bạn đang ở trong địa ngục đơn khối, có khả năng là bạn đã có ít nhất một vấn đề kinh doanh. Sau đây là một số ví dụ về các vấn đề kinh doanh do địa ngục đơn khối gây ra:

* + - * *Giao hàng chậm*—Ứng dụng khó hiểu, khó bảo trì và khó kiểm tra nên năng suất của nhà phát triển thấp. Kết quả là tổ chức không thể cạnh tranh hiệu quả và có nguy cơ bị đối thủ cạnh tranh vượt mặt.
      * *Bản phát hành phần mềm lỗi*—Việc thiếu khả năng kiểm tra có nghĩa là các bản phát hành phần mềm thường có lỗi. Điều này khiến khách hàng không hài lòng, dẫn đến mất khách hàng và giảm doanh thu.
      * *Khả năng mở rộng kém*—Việc mở rộng quy mô một ứng dụng đơn khối rất khó khăn vì nó kết hợp các mô-đun có yêu cầu tài nguyên rất khác nhau thành một thành phần có thể thực thi. Việc thiếu khả năng mở rộng quy mô có nghĩa là không thể hoặc quá tốn kém để mở rộng quy mô ứng dụng vượt quá một điểm nhất định. Do đó, ứng dụng không thể hỗ trợ các nhu cầu hiện tại hoặc dự đoán của doanh nghiệp.

Điều quan trọng là phải chắc chắn rằng những vấn đề này tồn tại vì bạn đã vượt ra khỏi kiến ​​trúc của mình. Một lý do phổ biến khiến việc phân phối chậm và phát hành lỗi là quy trình phát triển phần mềm kém. Ví dụ, nếu bạn vẫn đang dựa vào thử nghiệm thủ công, thì việc áp dụng thử nghiệm tự động một mình có thể tăng đáng kể tốc độ phát triển. Tương tự như vậy, đôi khi bạn có thể giải quyết các vấn đề về khả năng mở rộng mà không cần thay đổi kiến ​​trúc của mình. Trước tiên, bạn nên thử các giải pháp đơn giản hơn. Nếu, và chỉ nếu, bạn vẫn gặp vấn đề về phân phối phần mềm, thì bạn mới nên di chuyển sang kiến ​​trúc vi dịch vụ. Hãy cùng xem cách thực hiện điều đó.

###### Siết chặt khối đá nguyên khối

Quá trình chuyển đổi một ứng dụng độc lập thành các dịch vụ vi mô là một hình thức hiện đại hóa ứng dụng (<https://en.wikipedia.org/wiki/Software_modernization>). Hiện đại hóa ứng dụng là quá trình chuyển đổi ứng dụng cũ thành ứng dụng có kiến ​​trúc và công nghệ hiện đại. Các nhà phát triển đã hiện đại hóa ứng dụng trong nhiều thập kỷ. Kết quả là, có sự khôn ngoan tích lũy qua kinh nghiệm mà chúng ta có thể sử dụng khi tái cấu trúc ứng dụng thành kiến ​​trúc vi dịch vụ. Bài học quan trọng nhất rút ra trong nhiều năm là không nên viết lại một cách đột ngột.

Viết lại big bang là khi bạn phát triển một ứng dụng mới—trong trường hợp này là ứng dụng dựa trên microservices—từ đầu. Mặc dù bắt đầu từ đầu và bỏ lại cơ sở mã cũ nghe có vẻ hấp dẫn, nhưng điều này cực kỳ rủi ro và có khả năng sẽ dẫn đến thất bại. Bạn sẽ mất nhiều tháng, thậm chí nhiều năm, để sao chép chức năng hiện có và chỉ khi đó bạn mới có thể triển khai các tính năng mà doanh nghiệp cần ngày nay! Ngoài ra, dù sao thì bạn cũng sẽ cần phát triển ứng dụng cũ, điều này sẽ chuyển hướng nỗ lực khỏi việc viết lại và có nghĩa là bạn có mục tiêu liên tục thay đổi. Hơn nữa, điều đó có thể

rằng bạn sẽ lãng phí thời gian để triển khai lại các tính năng không còn cần thiết nữa. Như Martin Fowler đã nói, "điều duy nhất mà một bản viết lại Big Bang đảm bảo là một Big Bang!" ([www.randyshoup.com/evolutionary-architecture](http://www.randyshoup.com/evolutionary-architecture)).

Thay vì viết lại big bang, bạn nên, như hình 13.1 cho thấy, tái cấu trúc dần dần ứng dụng monolithic của mình. Bạn dần dần xây dựng một ứng dụng mới, được gọi là ứng dụng strangler. Nó bao gồm các dịch vụ vi mô chạy cùng với ứng dụng monolithic của bạn. Theo thời gian, lượng chức năng được triển khai bởi ứng dụng monolithic sẽ giảm dần cho đến khi nó biến mất hoàn toàn hoặc trở thành một dịch vụ vi mô khác. Chiến lược này giống như việc bảo dưỡng xe của bạn trong khi lái xe trên đường cao tốc với tốc độ 70 dặm/giờ. Nó đầy thách thức, nhưng ít rủi ro hơn nhiều so với việc cố gắng viết lại big bang.

**Ứng dụng siết cổ sẽ lớn dần theo thời gian.**

Ứng dụng bóp cổ

...

Dịch vụ

Dịch vụ

Dịch vụ

Dịch vụ

Dịch vụ

Dịch vụ

|  |
| --- |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |

|  |
| --- |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |

|  |
| --- |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |
| Dịch vụ |

Thời gian

Khối đá nguyên khối

Khối đá nguyên khối

Khối đá nguyên khối

Khối đá nguyên khối

...

Khối đá nguyên khối

**Khối đá nguyên khối co lại theo thời gian.**

**Hình 13.1 Khối đơn được thay thế dần dần bằng ứng dụng bóp nghẹt bao gồm các dịch vụ. Cuối cùng, khối đơn được thay thế hoàn toàn bằng ứng dụng bóp nghẹt hoặc trở thành một khối khácdịch vụ vi mô.**

Martin Fowler gọi chiến lược hiện đại hóa ứng dụng này là mô hình ứng dụng Strangler ([www.martinfowler.com/bliki/StranglerApplication.html](http://www.martinfowler.com/bliki/StranglerApplication.html)). Tên này xuất phát từ cây nho siết cổ (hoặc cây sung siết cổ—xem[https://en.wikipedia.org/wiki/](https://en.wikipedia.org/wiki/Strangler_fig) [Người bóp cổ](https://en.wikipedia.org/wiki/Strangler_fig)) được tìm thấy trong rừng mưa. Một cây dây leo siết cổ mọc xung quanh một cái cây trong

để có thể đón được ánh sáng mặt trời phía trên tán rừng. Cây thường chết vì bị dây leo giết chết hoặc chết vì già, để lại một dây leo hình cây.

**Mẫu: Ứng dụng Strangler**

Hiện đại hóa ứng dụng bằng cách phát triển dần dần một ứng dụng mới (strangler) xung quanh ứng dụng cũ. Xem[http://microservices.io/patterns/refactoring/](http://microservices.io/patterns/refactoring/strangler-application.html) [strangler-application.htmtôi](http://microservices.io/patterns/refactoring/strangler-application.html).

Quá trình tái cấu trúc thường mất nhiều tháng hoặc nhiều năm. Ví dụ, theo Steve Yegge ([https://plus.google.com/+RipRowan/posts/eVeouesvaVX](https://plus.google.com/%2BRipRowan/posts/eVeouesvaVX)) Amazon mất vài năm để tái cấu trúc khối đơn của mình. Trong trường hợp hệ thống rất lớn, bạn có thể không bao giờ hoàn thành quy trình. Ví dụ, bạn có thể đạt đến điểm mà bạn có các nhiệm vụ quan trọng hơn việc phá vỡ khối đơn, chẳng hạn như triển khai các tính năng tạo doanh thu. Nếukhối đá nguyên khối không phải là trở ngại cho sự phát triển liên tục, bạn cũng có thể để nguyên nó.

**DTHỂ HIỆN GIÁ TRỊ SỚM VÀ THƯỜNG XUYÊN**

Một lợi ích quan trọng của việc tái cấu trúc từng bước cho kiến ​​trúc vi dịch vụ là bạn sẽ thu được lợi nhuận ngay lập tức từ khoản đầu tư của mình. Điều đó rất khác so với việc viết lại big bang, vốn không mang lại bất kỳ lợi ích nào cho đến khi hoàn tất. Khi tái cấu trúc từng bước khối đơn, bạn có thể phát triển từng dịch vụ mới bằng cách sử dụng một ngăn xếp công nghệ mới và một quy trình phát triển và phân phối theo phong cách DevOps hiện đại, tốc độ cao. Do đó, tốc độ phân phối của nhóm bạn tăng dần theo thời gian.

Hơn nữa, bạn có thể di chuyển các khu vực có giá trị cao của ứng dụng sang các vùng siêu nhỏ.Vices trước. Ví dụ, hãy tưởng tượng bạn đang làm việc trên ứng dụng FTGO. Ví dụ, doanh nghiệp có thể quyết định rằng thuật toán lập lịch giao hàng là một lợi thế cạnh tranh quan trọng. Có khả năng là quản lý giao hàng sẽ là một lĩnh vực phát triển liên tục, đang diễn ra. Bằng cách trích xuất quản lý giao hàng thành một dịch vụ độc lập, nhóm quản lý giao hàng sẽ có thể làm việc độc lập với các nhà phát triển FTGO còn lại và tăng đáng kể tốc độ phát triển của họ. Họ sẽ có thể thường xuyên triển khai các phiên bản mới của thuật toán và đánh giá hiệu quả của chúng.

Một lợi ích khác của việc có thể cung cấp giá trị sớm hơn là nó giúp duy trì sự hỗ trợ của doanh nghiệp cho nỗ lực di chuyển. Sự hỗ trợ liên tục của họ là điều cần thiết, vì nỗ lực tái cấu trúc sẽ có nghĩa là ít thời gian hơn được dành cho việc phát triển các tính năng. Một số tổ chức gặp khó khăn trong việc loại bỏ nợ kỹ thuật vì những nỗ lực trước đây quá tham vọng và không mang lại nhiều lợi ích. Do đó, doanh nghiệp trở nên miễn cưỡng đầu tư vào các nỗ lực dọn dẹp tiếp theo. Bản chất gia tăng của việc tái cấu trúc thành các dịch vụ siêu nhỏ có nghĩa là nhóm phát triển có thể chứng minh giá trị sớm và thường xuyên.

**TôiINIMIZE THAY ĐỔI ĐỐI VỚI MONOLITH**

Một chủ đề thường gặp trong chương này là bạn nên tránh thực hiện những thay đổi rộng rãi đối với khối đơn khi di chuyển sang kiến ​​trúc dịch vụ vi mô. Điều này là không thể tránh khỏi

rằng bạn sẽ cần thực hiện một số thay đổi để hỗ trợ di chuyển sang dịch vụ. Phần 13.3.2 nói về cách monolith thường cần được sửa đổi để có thể tham gia vào các sagas duy trì tính nhất quán của dữ liệu trên monolith và các dịch vụ. Vấn đề với việc thực hiện các thay đổi rộng rãi đối với monolith là tốn thời gian, tốn kém và rủi ro. Rốt cuộc, đó có lẽ là lý do tại sao bạn muốn di chuyển sang các dịch vụ siêu nhỏ ngay từ đầu.

May mắn thay, có những chiến lược bạn có thể sử dụng để giảm phạm vi các thay đổi bạn cần thực hiện. Ví dụ, trong phần 13.2.3, tôi mô tả chiến lược sao chép dữ liệu từ một dịch vụ đã trích xuất trở lại cơ sở dữ liệu của khối đơn. Và trong phần 13.3.2, tôi chỉ ra cách bạn có thể sắp xếp cẩn thận quá trình trích xuất các dịch vụ để giảm tác động lên khối đơn. Bằng cách áp dụng các chiến lược này, bạn có thể giảm lượng công việc cần thiết để cấu trúc lại khối đơn.

**TCƠ SỞ HẠ TẦNG TRIỂN KHAI KỸ THUẬT: CóOU DON'T CẦN TẤT CẢ NHƯ VẬY**

Trong suốt cuốn sách này, tôi đã thảo luận rất nhiều về công nghệ mới sáng bóng, bao gồm các nền tảng triển khai như Kubernetes và AWS Lambda và các cơ chế khám phá dịch vụ. Bạn có thể bị cám dỗ bắt đầu di chuyển sang các dịch vụ siêu nhỏ bằng cách lựa chọn các công nghệ và xây dựng cơ sở hạ tầng đó. Bạn thậm chí có thể cảm thấy áp lực từ những người kinh doanh và từ nhà cung cấp PaaS thân thiện của mình để bắt đầu chi tiền cho loại cơ sở hạ tầng này.

Mặc dù có vẻ hấp dẫn khi xây dựng cơ sở hạ tầng này ngay từ đầu, tôi khuyên bạn chỉ nên đầu tư tối thiểu ngay từ đầu để phát triển nó. Điều duy nhất bạn không thể sống thiếu là một đường ống triển khai thực hiện thử nghiệm tự động. Ví dụ, nếu bạn chỉ có một số ít dịch vụ, bạn không cần một cơ sở hạ tầng triển khai và khả năng quan sát phức tạp. Ban đầu, bạn thậm chí có thể thoát khỏi việc chỉ sử dụng tệp cấu hình được mã hóa cứng để khám phá dịch vụ. Tôi khuyên bạn nên hoãn mọi quyết định về cơ sở hạ tầng kỹ thuật liên quan đến đầu tư đáng kể cho đến khi bạn có được kinh nghiệm thực tế với kiến ​​trúc dịch vụ vi mô. Chỉ khi bạn có một vài dịch vụ đang chạy, bạn mới có kinh nghiệm để lựa chọn công nghệ.

Bây giờ chúng ta hãy xem xét các chiến lược bạn có thể sử dụng để di chuyển sang kiến ​​trúc vi dịch vụ.

#### Các chiến lược để tái cấu trúc một khối đơn thành các dịch vụ vi mô

Có ba chiến lược chính để hạn chế khối đơn và thay thế dần bằng các dịch vụ vi mô:

**1** Triển khai các tính năng mới dưới dạng dịch vụ.

**2** Tách lớp trình bày và lớp nền.

**3** Phá vỡ khối thống nhất bằng cách trích xuất chức năng thành các dịch vụ.

Chiến lược đầu tiên ngăn khối đơn khối phát triển. Đây thường là cách nhanh chóng để chứng minh giá trị của các dịch vụ vi mô, giúp xây dựng hỗ trợ cho nỗ lực di chuyển. Hai chiến lược còn lại phá vỡ khối đơn khối. Khi tái cấu trúc khối đơn khối của bạn, đôi khi bạn có thể sử dụng chiến lược thứ hai, nhưng chắc chắn bạn sẽ sử dụng

chiến lược thứ ba, vì đây là cách chức năng được di chuyển từ khối đơn sang ứng dụng Strangler.

Hãy cùng xem xét từng chiến lược này, bắt đầu bằng việc triển khai các tính năng mới dưới dạng dịch vụ.

###### Triển khai các tính năng mới dưới dạng dịch vụ

Luật về hố nêu rằng “nếu bạn thấy mình đang ở trong hố, hãy ngừng đào” ([https://](https://en.m.wikipedia.org/wiki/Law_of_holes) [vi.m.wikipedia.org/wiki/Law\_of\_holes](https://en.m.wikipedia.org/wiki/Law_of_holes)). Đây là lời khuyên tuyệt vời để làm theo khi ứng dụng monolithic của bạn trở nên không thể quản lý được. Nói cách khác, nếu bạn có một ứng dụng monolithic lớn, phức tạp, đừng triển khai các tính năng mới bằng cách thêm mã vào monolith. Điều đó sẽ khiến monolith của bạn thậm chí còn lớn hơn và khó quản lý hơn. Thay vào đó, bạn nên triển khai các tính năng mới dưới dạng dịch vụ.

Đây là một cách tuyệt vời để bắt đầu di chuyển ứng dụng monolithic của bạn sang kiến ​​trúc microservice. Nó làm giảm tốc độ tăng trưởng của monolith. Nó đẩy nhanh quá trình phát triển các tính năng mới, vì bạn đang phát triển trong một cơ sở mã hoàn toàn mới. Nó cũng nhanh chóng chứng minh giá trị của việc áp dụng kiến ​​trúc microservice.

**TÔITÍCH HỢP DỊCH VỤ MỚI VỚI MONOLITH**

Hình 13.2 cho thấy kiến ​​trúc của ứng dụng sautriển khai một tính năng mới dưới dạng dịch vụ. Bên cạnh dịch vụ và khối đơn mới, kiến ​​trúc bao gồm hai thành phần khác tích hợp dịch vụ vào ứng dụng:

* + - * *Cổng API*—Định tuyến các yêu cầu về chức năng mới tới dịch vụ mới và định tuyến các yêu cầu cũ tới khối đơn nguyên.
      * *Mã keo tích hợp*—Tích hợp dịch vụ với khối đơn. Nó cho phép dịch vụ truy cập dữ liệu do khối đơn sở hữu và gọi chức năng do khối đơn triển khai.

Mã keo tích hợp không phải là một thành phần độc lập. Thay vào đó, nó bao gồm các bộ điều hợp trong khối đơn và dịch vụ sử dụng một hoặc nhiều giao tiếp giữa các tiến trìnhcơ chế. Đối vớiVí dụ, keo tích hợp cho Dịch vụ giao hàng trễ, được mô tả trong phần 13.4.1, sử dụng cả sự kiện REST và sự kiện miền. Dịch vụ này lấy thông tin hợp đồng của khách hàng từ khối đơn bằng cách gọi API REST. Khối đơn công bố sự kiện miền Đơn hàng để Dịch vụ giao hàng trễ có thể theo dõi trạng thái của Đơn hàng và phản hồi các đơn hàng không được giao đúng hạn. Phần 13.3.1 mô tả mã keo tích hợp chi tiết hơn.

**THEN SẼ TRIỂN KHAI MỘT TÍNH NĂNG MỚI NHƯ MỘT DỊCH VỤ**

Lý tưởng nhất là bạn nên triển khai mọi tính năng mới trong ứng dụng strangler thay vì trong monolith. Bạn sẽ triển khai một tính năng mới dưới dạng dịch vụ mới hoặc là một phần của dịch vụ hiện có. Theo cách này, bạn sẽ tránh phải động đến cơ sở mã monolith. Tuy nhiên, thật không may, không phải mọi tính năng mới đều có thể được triển khai dưới dạng dịch vụ.

Đó là vì bản chất của kiến ​​trúc dịch vụ vi mô là một tập hợp các dịch vụ được ghép nối lỏng lẻo được tổ chức xung quanh các khả năng kinh doanh. Ví dụ, một tính năng có thể quá nhỏ để trở thành một dịch vụ có ý nghĩa. Ví dụ, bạn có thể chỉ cần thêm

**Dịch vụ triển khai**



Tính năng cũ

Tính năng mới

**tính năng mới**

Đến

bộ chuyển đổi

Đến

bộ chuyển đổi

Đến

bộ chuyển đổi

Đi ra ngoài

bộ chuyển đổi

Khối đá nguyên khối

Tích hợp

keo dán

Cơ sở dữ liệu

bộ chuyển đổi

Cơ sở dữ liệu

bộ chuyển đổi

Khối đá nguyên khối

cơ sở dữ liệu

Dịch vụ

cơ sở dữ liệu

Thông báo

«tổng hợp»

Bộ điều hợp thuê bao sự kiện

Bộ điều hợp nhà xuất bản sự kiện

«tổng hợp»Đặt hàng

«tổng hợp» Dịch vụ giao hàng trễ



Cổng API

**Hình 13.2 Một tính năng mới được triển khai như một dịch vụ là một phần của ứng dụng strangler. Keo tích hợp tích hợp dịch vụ với khối đơn và bao gồm các bộ điều hợp triển khai API đồng bộ và không đồng bộ. Cổng API định tuyến các yêu cầu gọi chức năng mớiđến dịch vụ.**

ít trường và phương thức cho một lớp hiện có. Hoặc tính năng mới có thể được liên kết quá chặt chẽ với mã trong khối đơn. Nếu bạn cố gắng triển khai loại tính năng này dưới dạng dịch vụ, bạn thường thấy rằng hiệu suất sẽ bị ảnh hưởng do giao tiếp giữa các tiến trình quá mức. Bạn cũng có thể gặp sự cố khi duy trì tính nhất quán của dữ liệu. Nếu không thể triển khai tính năng mới dưới dạng dịch vụ, giải pháp thường là triển khai tính năng mới ban đầu trong khối đơn. Sau đó, bạn có thể trích xuất tính năng đó cùng với các tính năng liên quan khác vào dịch vụ riêng của chúng.

Việc triển khai các tính năng mới dưới dạng dịch vụ sẽ đẩy nhanh quá trình phát triển các tính năng đó. Đây là một cách tốt để nhanh chóng chứng minh giá trị của kiến ​​trúc vi dịch vụ. Nó cũng làm giảm tốc độ tăng trưởng của khối đơn. Nhưng cuối cùng, bạn cần phải chia nhỏ khối đơn bằng hai chiến lược khác. Bạn cần di chuyển chức năng sang ứng dụng strangler bằng cách trích xuất chức năng từ khối đơn thành các dịch vụ. Bạn cũng có thể cải thiện tốc độ phát triển bằng cách chia khối đơn theo chiều ngang. Hãy cùng xem cách thực hiện điều đó.

###### Trình bày riêng biệttầng từ phía sau

Một chiến lược để thu nhỏ ứng dụng đơn khối là tách lớp trình bày khỏi lớp logic kinh doanh và lớp truy cập dữ liệu. Một ứng dụng doanh nghiệp điển hình bao gồm các lớp sau:

* + - * *Logic trình bày*—Điều này bao gồm các mô-đun xử lý các yêu cầu HTTP và tạo racác trang HTML triển khai giao diện người dùng web. Trong một ứng dụng có giao diện người dùng phức tạp, tầng trình bày thường là một khối mã đáng kể.
      * *Logic kinh doanh*—Bao gồm các mô-đun triển khai các quy tắc kinh doanh, có thể phức tạp trong ứng dụng doanh nghiệp.
      * *Logic truy cập dữ liệu*—Bao gồm các mô-đun truy cập các dịch vụ cơ sở hạ tầng như cơ sở dữ liệu và trung gian tin nhắn.

Thường có sự tách biệt rõ ràng giữa logic trình bày và logic kinh doanh và truy cập dữ liệu. Tầng kinh doanh có API chi tiết bao gồm một hoặc nhiều mặt tiền đóng gói logic kinh doanh. API này là một đường nối tự nhiên mà bạn có thể chia khối đơn thành hai ứng dụng nhỏ hơn, như thể hiện trong hình 13.3.

**Khối đơn chứa logic trình bày và logic kinh doanh phụ trợ**

**Nhỏ hơn, độc lậpkhối logic trình bày có thể triển khai**

**Nhỏ hơn, độc lậpkhối đơn khối backend có thể triển khai**

**Hình 13.3**

**Một API có thể được gọi bởi bất kỳ dịch vụ nào trong tương lai**

**Chia tách giao diệntừ phía sau cho phép mỗi cái được triển khai độc lập. Nó cũng phơi bày**



Các trang HTML

Các trang HTML

Trang web

ứng dụng

Tách ra

NGHỈ NGƠI

khách hàng

NGHỈ NGƠI

Giao diện lập trình ứng dụng (API)

Logic kinh doanh

Logic kinh doanh

Cơ sở dữ liệubộ chuyển đổi

Cơ sở dữ liệubộ chuyển đổi

Trình duyệt

Trình duyệt

MySQL

MySQL

Ứng dụng web

**một API để gọi các dịch vụ.**

Một ứng dụng chứa lớp trình bày và ứng dụng kia chứa logic truy cập dữ liệu và kinh doanh. Sau khi tách, ứng dụng logic trình bày thực hiện các cuộc gọi từ xa đến ứng dụng logic kinh doanh.

Chia tách khối đá nguyên khối trong nàycách này có hai lợi ích chính. Nó cho phép bạn phát triển, triển khai và mở rộng hai ứng dụng độc lập với nhau. Đặc biệt, nó cho phép các nhà phát triển lớp trình bày lặp lại nhanh chóng trên giao diện người dùng và dễ dàng thực hiện thử nghiệm A/B, ví dụ, mà không cần phải triển khai phần phụ trợ. Một lợi ích khác của cách tiếp cận này là nó cho thấy một API từ xa có thể được gọi bởi các dịch vụ vi mô mà bạn phát triển sau này.

Nhưng chiến lược này chỉ là một giải pháp một phần. Rất có thể là ít nhất mộthoặc cả hai ứng dụng kết quả vẫn sẽ là một khối đơn lẻ không thể quản lý được. Bạn cần sử dụng chiến lược thứ ba để thay thế khối đơn lẻ bằng các dịch vụ.

###### Trích xuất khả năng kinh doanh thành dịch vụ

Việc triển khai các tính năng mới dưới dạng dịch vụ và tách ứng dụng web frontend khỏi backend sẽ chỉ đưa bạn đến một mức độ nào đó. Bạn vẫn sẽ phải thực hiện rất nhiều phát triển trong cơ sở mã monolithic. Nếu bạn muốn cải thiện đáng kể kiến ​​trúc ứng dụng của mình và tăng tốc độ phát triển, bạn cần phải tách monolith ra bằng cách di chuyển dần dần các khả năng kinh doanh từ monolith sang dịch vụ. Ví dụ, phần 13.5 mô tả cách trích xuất quản lý phân phốitừ khối FTGO thành Dịch vụ giao hàng mới. Khi bạn sử dụng chiến lược này, theo thời gian, số lượng khả năng kinh doanh được triển khai bởi các dịch vụ sẽ tăng lên và khối dần thu hẹp lại.

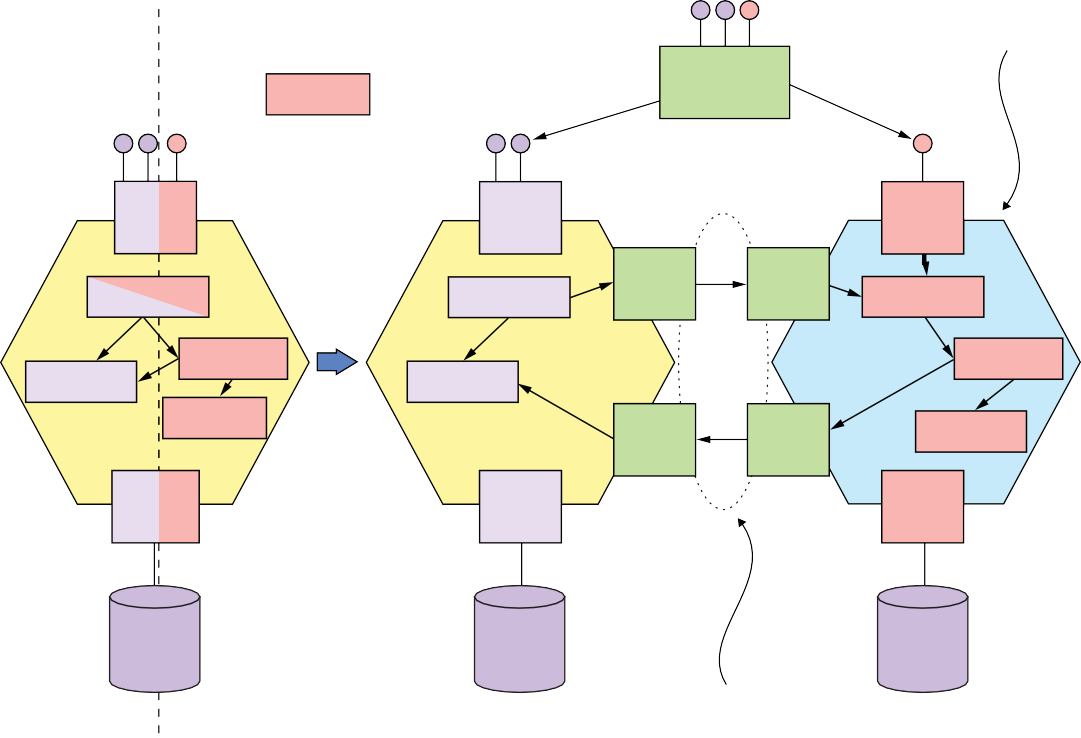
Chức năng bạn muốn trích xuất vào một dịch vụ là một lát cắt theo chiều dọc qua khối đơn. Lát cắt bao gồm các mục sau:

* + - * Bộ điều hợp đầu vào triển khai điểm cuối API
      * Logic miền
      * Bộ điều hợp đầu ra như logic truy cập cơ sở dữ liệu
      * Sơ đồ cơ sở dữ liệu của khối đơn

Như hình 13.4 cho thấy, mã này được trích xuất từ ​​khối đơn vàđược chuyển thành một dịch vụ độc lập. Một cổng API định tuyến các yêu cầu gọi khả năng kinh doanh đã trích xuất đến dịch vụ và định tuyến các yêu cầu khác đến khối đơn. Khối đơn và dịch vụ cộng tác thông qua mã keo tích hợp. Như đã mô tả trong phần 13.3.1, keo tích hợp bao gồm các bộ điều hợp trong dịch vụ và khối đơn sử dụng một hoặc nhiều cơ chế giao tiếp giữa các tiến trình (IPC).

Trích xuất dịch vụ là một thách thức. Bạn cần xác định cách chia mô hình miền của khối đơn thành hai mô hình miền riêng biệt, một trong số đó trở thành mô hình miền của dịch vụ. Bạn cần phá vỡ các phụ thuộc như tham chiếu đối tượng. Bạn thậm chí có thể cần chia các lớp để di chuyển chức năng vào dịch vụ. Bạn cũng cần cấu trúc lại cơ sở dữ liệu.

Việc trích xuất một dịch vụ thường tốn thời gian, đặc biệt là vì cơ sở mã của khối đơn có thể lộn xộn. Do đó, bạn cần phải suy nghĩ cẩn thận về việc



Khối đá nguyên khối

Mã để

trích xuất vào một dịch vụ

**Dịch vụ chứa mã đã trích xuất**

Cổng API

Đến

bộ chuyển đổi

Đến

bộ chuyển đổi

Đến

bộ chuyển đổi

«dịch vụ» Đặt hàng dịch vụ

«dịch vụ» Đặt hàng dịch vụ

Đến

bộ chuyển đổi

Đi ra ngoài

bộ chuyển đổi

«dịch vụ» Đặt hàng dịch vụ

«tổng hợp» Đặt hàng

Chuyển phát nhanh «tổng hợp»

«tổng hợp» Đặt hàng

Tích hợp

keo dán

Chuyển phát nhanh «tổng hợp»

Kế hoạch «tổng hợp»

Đến

bộ chuyển đổi

Đi ra ngoài

bộ chuyển đổi

Kế hoạch «tổng hợp»

Cơ sở dữ liệu

bộ chuyển đổi

Cơ sở dữ liệu

bộ chuyển đổi

Cơ sở dữ liệu

bộ chuyển đổi

Khối đá nguyên khối

cơ sở dữ liệu

Khối đá nguyên khối

cơ sở dữ liệu

Dịch vụ

cơ sở dữ liệu

**Mã keo tích hợp dịch vụ với khối đơn**

**Hình 13.4Tách khối đơn khối bằng cách trích xuất các dịch vụ. Bạn xác định một phần chức năng, bao gồm logic kinh doanh và bộ điều hợp, để trích xuất thành một dịch vụ. Bạn di chuyển mã đó vào dịch vụ. Mã mới trích xuấtdịch vụ và khối đơn khối cộng tác thông qua các API do keo tích hợp cung cấp.**

dịch vụ để trích xuất. Điều quan trọng là tập trung vào việc tái cấu trúc các phần của ứng dụng cung cấp nhiều giá trị. Trước khi trích xuất một dịch vụ, hãy tự hỏi lợi ích của việc làm đó là gì.

Ví dụ, việc trích xuất một dịch vụ triển khai chức năng quan trọng đối với doanh nghiệp và liên tục phát triển là điều đáng giá. Việc đầu tư công sức vào việc trích xuất các dịch vụ khi không có nhiều lợi ích từ việc làm đó là không có giá trị. Sau trong phần này, tôi sẽ mô tả một số chiến lược để xác định nội dung cần trích xuất và thời điểm trích xuất. Nhưng trước tiên, hãy cùng xem xét chi tiết hơn một số thách thức mà bạn sẽ phải đối mặt khi trích xuất một dịch vụ và cách giải quyết chúng.

Bạn sẽ gặp phải một số thách thức khi trích xuất một dịch vụ:

* Phân chia mô hình miền
* Tái cấu trúc cơ sở dữ liệu

Chúng ta hãy xem xét từng phương pháp, bắt đầu bằng việc phân tách mô hình miền.

**SCHIA GHÉP MÔ HÌNH MIỀN**

Để trích xuất một dịch vụ, bạn cần trích xuất mô hình miền của nó ra khỏi khối đơn.mô hình miền. Bạn sẽ cần phải thực hiện phẫu thuật lớn để chia tách các mô hình miền. Một thách thức bạn sẽ gặp phải là loại bỏ các tham chiếu đối tượng nếu không sẽ mở rộng ranh giới dịch vụ. Có thể các lớp vẫn còn trong khối đơn sẽ tham chiếu đến các lớp đã được chuyển đến dịch vụ hoặc ngược lại. Ví dụ, hãy tưởng tượng rằng, nhưhình 13.5 cho thấy, bạn trích xuất Order Service và kết quả là lớp Order của nó tham chiếu đến lớp Restaurant của khối đơn. Vì một thể hiện dịch vụ thường là một quy trình, nên không có ý nghĩa gì khi có các tham chiếu đối tượng vượt qua ranh giới dịch vụ. Bằng cách nào đó, bạn cần loại bỏ các loại tham chiếu đối tượng này.

**Dịch vụ trích xuất**

Khối FTGO

«Thực thể»Đặt hàng

Nhà hàng «Entity»

Khối FTGO

?

«Thực thể»Đặt hàng

Dịch vụ giao hàng

Nhà hàng

«Thực thể»

**Tham chiếu đối tượngvượt qua ranh giới dịch vụ**

**Hình 13.5Đặt hànglớp miền có tham chiếu đến mộtNhà hànglớp. Nếu chúng ta trích xuấtĐặt hàngvào một dịch vụ riêng biệt, chúng ta cần phải làm gì đó về tham chiếu của nó tớiNhà hàng,vì các tham chiếu đối tượng giữa các tiến trình không có ý nghĩa.**

Một cách tốt để giải quyết vấn đề này là suy nghĩ theo hướng tổng hợp DDD, được mô tả trong chương 5. Các tổng hợp tham chiếu lẫn nhau bằng khóa chính thay vì tham chiếu đối tượng.erences. Do đó, bạn sẽ nghĩ về các lớp Order và Restaurant như các tập hợp và, như hình 13.6 cho thấy, thay thế tham chiếu đến Restaurant trong lớp Order bằng trường restaurantId lưu trữ giá trị khóa chính.

Khối FTGO

?

«Thực thể»Đặt hàng

Dịch vụ giao hàng

Nhà hàng «Entity»

Dịch vụ giao hàng

«Thực thể»Đặt hàng

nhà hàngId

Khối FTGO

Nhà hàng «Entity»

**Tham chiếu đối tượng đóvượt qua ranh giới dịch vụ**

**Thay thế bằng khóa chính.**

**Hình 13.6Đặt hàngtham chiếu của lớp tớiNhà hàngđược thay thế bằngNhà hàng'Skhóa chính để loại bỏ đối tượng có thể vượt qua ranh giới quy trình.**

Một vấn đề với việc thay thế tham chiếu đối tượng bằng khóa chính là mặc dù đây là một thay đổi nhỏ đối với lớp, nhưng nó có khả năng có tác động lớn đến các máy khách của lớp, những máy khách mong đợi một tham chiếu đối tượng. Sau trong phần này, tôi sẽ mô tả cách giảm phạm vi thay đổi bằng cách sao chép dữ liệu giữa dịch vụ và khối đơn. Giao hàngVí dụ, Service có thể định nghĩa một lớp Restaurant là bản sao của lớp Restaurant của khối đơn.

Việc trích xuất một dịch vụ thường phức tạp hơn nhiều so với việc di chuyển toàn bộ các lớp vào một

dịch vụ. Một thách thức lớn hơn nữa khi chia tách mô hình miền là trích xuất chức năng được nhúng trong một lớp có các trách nhiệm khác. Vấn đề này thường xảy ra trong các lớp god, được mô tả trong chương 2, có quá nhiều trách nhiệm. Ví dụ, lớp Order là một trong những lớp god trong ứng dụng FTGO. Nó triển khai nhiều khả năng kinh doanh, bao gồm quản lý đơn hàng, quản lý giao hàng, v.v. Sau đó trong phần 13.5, tôi thảo luận về cách trích xuất quản lý giao hàng thành một dịch vụ liên quan đến việc trích xuất một lớp Delivery từLớp Order. Thực thể Delivery triển khai chức năng quản lý giao hàng trước đó được tích hợp với chức năng khác trong lớp Order.

**RPHÂN TÍCH CƠ SỞ DỮ LIỆU**

Việc tách một mô hình miền không chỉ liên quan đến việc thay đổi mã. Nhiều lớp trong một mô hình miền là liên tục. Các trường của chúng được ánh xạ tới một lược đồ cơ sở dữ liệu. Do đó, khi bạn trích xuất một dịch vụ từ khối đơn, bạn cũng đang di chuyển dữ liệu. Bạn cần di chuyển các bảng từ cơ sở dữ liệu của khối đơn sang cơ sở dữ liệu của dịch vụ.

Ngoài ra, khi bạn chia tách một thực thể, bạn cần chia tách bảng cơ sở dữ liệu tương ứng và di chuyển bảng mới đến dịch vụ. Ví dụ, khi trích xuất quản lý giao hàngthành một dịch vụ, bạn chia thực thể Order và trích xuất thực thể Delivery. Ở cấp độ cơ sở dữ liệu, bạn chia bảng ORDERS và định nghĩa một bảng DELIVERY mới. Sau đó, bạn di chuyển bảng DELIVERY đến dịch vụ.

Cuốn sách Refactoring Databases của Scott W. Ambler và Pramod J. Sadalage (Addison-Wesley, 2011) mô tả một tập hợp các refactoring cho một lược đồ cơ sở dữ liệu. Ví dụ, nó mô tả quá trình tái cấu trúc Split Table, chia một bảng thành hai hoặc nhiều bảng. Nhiều kỹ thuật trong cuốn sách đó hữu ích khi trích xuất các dịch vụ từ khối đơn. Một trong những kỹ thuật như vậy là ý tưởng sao chép dữ liệu để cho phép bạn cập nhật gia tăng các máy khách của cơ sở dữ liệu để sử dụng lược đồ mới. Chúng ta có thể điều chỉnh ý tưởng đó để giảm phạm vi các thay đổi mà bạn phải thực hiện đối với khối đơn khi trích xuất một dịch vụ.

**REPLICATE DỮ LIỆU ĐỂ TRÁNH NHỮNG THAY ĐỔI TRỘN RỘNG**

Như đã đề cập, việc trích xuất một dịch vụ yêu cầu bạn phải thay đổi mô hình miền của khối đơn. Ví dụ, bạn thay thế các tham chiếu đối tượng bằng khóa chính và các lớp phân tách. Những loại thay đổi này có thể lan tỏa khắp cơ sở mã và yêu cầu bạn phải thực hiện các thay đổi rộng rãi đối với khối đơn. Ví dụ, nếu bạn tách thực thể Order và trích xuất thực thể Delivery, bạn sẽ phải thay đổi mọi vị trí trong mã tham chiếu đến các trường đã được di chuyển. Việc thực hiện những loại thay đổi này có thể cực kỳ tốn thời gian và có thể trở thành rào cản lớn đối với việc chia nhỏ khối đơn.

Một cách tuyệt vời để trì hoãn và có thể tránh thực hiện những thay đổi tốn kém này là sử dụng một phương pháp tương tự như phương pháp được mô tả trong Refactoring Databases. Một trở ngại lớn đối với việc refactoring một cơ sở dữ liệu là thay đổi tất cả các máy khách của cơ sở dữ liệu đó để sử dụng lược đồ mới. Giải pháp được đề xuất trong cuốn sách là bảo toàn lược đồ gốc trong một thời gian chuyển tiếp và sử dụng các kích hoạt để đồng bộ hóa lược đồ gốc và lược đồ mới. Sau đó, bạn di chuyển các máy khách từ lược đồ cũ sang lược đồ mới theo thời gian.

Chúng ta có thể sử dụng cách tiếp cận tương tự khi trích xuất các dịch vụ từ khối đơn. Ví dụ, khi trích xuất thực thể Giao hàng, chúng ta để thực thể Đơn hàng hầu như không thay đổi trong thời gian chuyển tiếp. Như hình 13.7 cho thấy, chúng ta thực hiện liên quan đến giao hàngcác trường chỉ đọc và giữ chúng được cập nhật bằng cách sao chép dữ liệu từ Delivery Service trở lại khối đơn. Do đó, chúng ta chỉ cần tìm những vị trí trong mã của khối đơn cập nhật các trường đó và thay đổi chúng để gọi Delivery Service mới.

Bảo toàn cấu trúc của thực thể Order bằng cách sao chép dữ liệu từ Delivery

Dịch vụ làm giảm đáng kể lượng công việc chúng ta cần phải làm ngay lập tức.thời gian, chúng ta có thể di chuyển mã sử dụng các trường thực thể Order liên quan đến giao hàng hoặc các cột bảng ORDERS sang Delivery Service. Hơn nữa, có thể chúng ta không bao giờ cần

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ĐƠN\_HÀNH | ID NHÀ HÀNG | ... | THỜI GIAN ĐÃ LÊN LỊCH | THỜI GIAN GIAO HÀNG ĐÃ LÊN LỊCH |
| ... | ... | ... | ... | ... |

|  |  |
| --- | --- |
|  | «Thực thể»Đặt hàng |
| ...  người tiêu dùngId |
|  | thời gian nhận hàng theo lịch trìnhthời gian giao hàng theo lịch trình |
|  | ... |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ĐƠN\_HÀNH | ID NHÀ HÀNG | ... | THỜI GIAN ĐÃ LÊN LỊCH | THỜI GIAN GIAO HÀNG ĐÃ LÊN LỊCH |
| ... | ... | ... | ... | ... |

|  |  |  |  |
| --- | --- | --- | --- |
| ĐƠN\_HÀNH | ... | THỜI GIAN ĐÃ LÊN LỊCH | THỜI GIAN GIAO HÀNG ĐÃ LÊN LỊCH |
| ... | ... | ... | ... |

Sao chép dữ liệu từ Delivery Service sang khối FTGO.

Khối FTGO

ĐẶT HÀNGbàn

**Các trường liên quan đến việc phân phối chỉ đọc**

Trích xuất dịch vụ đơn hàng và di chuyển các cột từ

ĐẶT HÀNGbảng toa mới VẬN CHUYỂNbàn.

Khối FTGO

Dịch vụ giao hàng

ĐẶT HÀNGbàn

VẬN CHUYỂNbàn

...

orderId schedulePickupTime scheduleDeliveryTime

...

Giao hàng «Thực thể»

...

consumerId schedulePickupTime scheduleDeliveryTime

...

«Thực thể»Đặt hàng

**Hình 13.7 Giảm thiểu phạm vi thay đổi đối với khối FTGO bằng cách sao chép dữ liệu liên quan đến việc phân phối từ dữ liệu mới trích xuấtDịch vụ giao hàngquay lại cơ sở dữ liệu của khối.**

thực hiện thay đổi đó trong khối đơn. Nếu mã đó sau đó được trích xuất thành một chuỗiphó, sau đó dịch vụ có thể truy cậpDịch vụ giao hàng.

**TDỊCH VỤ MŨ ĐỂ TRÍCH XUẤT VÀ KHI NÀO**

Như tôi đã đề cập, việc chia nhỏ khối đơn khối tốn thời gian. Nó làm mất công sức triển khai các tính năng. Do đó, bạn phải quyết định cẩn thận trình tự trích xuất các dịch vụ. Bạn cần tập trung vào việc trích xuất các dịch vụ mang lại lợi ích lớn nhất. Hơn nữa, bạn muốn liên tục chứng minh cho doanh nghiệp thấy rằng việc di chuyển sang kiến ​​trúc vi dịch vụ có giá trị.

Trong bất kỳ hành trình nào, điều quan trọng là phải biết mình đang đi đâu. Một cách tốt để bắt đầu di chuyển sang các dịch vụ vi mô là với nỗ lực định nghĩa kiến ​​trúc có giới hạn thời gian. Bạn nên dành một khoảng thời gian ngắn, chẳng hạn như một vài tuần, để động não về kiến ​​trúc lý tưởng của mình và định nghĩa một tập hợp các dịch vụ. Điều này cung cấp cho bạn một đích đến để hướng tới. Tuy nhiên, điều quan trọng là phải nhớ rằng kiến ​​trúc này không phải là bất biến. Khi bạn phá vỡ khối thống nhất và tích lũy kinh nghiệm, bạn nên sửa đổi kiến ​​trúc để tính đến những gì bạn đã học được.

Sau khi xác định được đích đến gần đúng, bước tiếp theo là bắt đầu tách khối monolith. Có một số chiến lược khác nhau mà bạn có thể sử dụng để xác định trình tự trích xuất dịch vụ.

Một chiến lược là đóng băng hiệu quả quá trình phát triển khối đơn và trích xuất các dịch vụ theo yêu cầu. Thay vì triển khai các tính năng hoặc sửa lỗi trong khối đơn, bạn trích xuất dịch vụ hoặc các dịch vụ cần thiết và thay đổi chúng. Một lợi ích của cách tiếp cận này là nó buộc bạn phải chia nhỏ khối đơn. Một nhược điểm là việc trích xuất các dịch vụ được thúc đẩy bởi các yêu cầu ngắn hạn hơn là các nhu cầu dài hạn. Ví dụ, nó yêu cầu bạn trích xuất các dịch vụ ngay cả khi bạn đang thực hiện một thay đổi nhỏ đối với một phần tương đối ổn định của hệ thống. Do đó, bạn có nguy cơ làm rất nhiều việc mà chỉ thu được lợi ích tối thiểu.

Một chiến lược thay thế là một cách tiếp cận có kế hoạch hơn, trong đó bạn xếp hạng các mô-đun của ứng dụng theo lợi ích bạn dự đoán nhận được khi trích xuất chúng. Có một số lý do tại sao việc trích xuất một dịch vụ lại có lợi:

* *Tăng tốc phát triển*—Nếu lộ trình ứng dụng của bạn cho thấy một phần cụ thể của ứng dụng sẽ trải qua nhiều quá trình phát triển trong năm tới, thì việc chuyển đổi nó thành dịch vụ sẽ giúp đẩy nhanh quá trình phát triển.
* *Giải quyết vấn đề về hiệu suất, khả năng mở rộng hoặc độ tin cậy*-Nếu nhưmột phần cụ thể của ứng dụng của bạn có vấn đề về hiệu suất hoặc khả năng mở rộng hoặc không đáng tin cậy, thì việc chuyển đổi nó thành một dịch vụ là rất có giá trị.
* *Cho phép trích xuất một số dịch vụ khác*—Đôi khi việc trích xuất một dịch vụ sẽ đơn giản hóa việc trích xuất một dịch vụ khác, do sự phụ thuộc giữa các mô-đun.

Bạn có thể sử dụng các tiêu chí này để thêm các tác vụ tái cấu trúc vào danh sách tồn đọng của ứng dụng, được xếp hạng theo lợi ích mong đợi. Lợi ích của cách tiếp cận này là nó mang tính chiến lược hơn và phù hợp hơn nhiều với nhu cầu của doanh nghiệp. Trong quá trình lập kế hoạch sprint, bạn quyết định xem việc triển khai các tính năng hay trích xuất các dịch vụ có giá trị hơn.

#### Thiết kế cách thức dịch vụ và khối thống nhấthợp tác

Một dịch vụ hiếm khi độc lập. Nó thường cần phải cộng tác với khối đơn. Đôi khi một dịch vụ cần truy cập dữ liệu do khối đơn sở hữu hoặc gọi các hoạt động của khối đơn.Ví dụ, Giao hàng chậm trễDịch vụ, được mô tả chi tiết trong phần 13.4.1, yêu cầu quyền truy cập vào đơn hàng và thông tin liên hệ của khách hàng của khối đơn. Khối đơn cũng có thể cần truy cập dữ liệu do dịch vụ sở hữu hoặc gọi các hoạt động của nó. Ví dụ, sau đó trong phần 13.5, khi thảo luận về cách trích xuất quản lý giao hàng vào một dịch vụ, tôi mô tả cách khối đơn cần gọi Dịch vụ giao hàng.

Một mối quan tâm quan trọng là duy trì tính nhất quán của dữ liệu giữa dịch vụ và khối đơn. Cụ thể, khi bạn trích xuất một dịch vụ từ khối đơn, bạn luôn chia tách những gì ban đầu là giao dịch ACID. Bạn phải cẩn thận để đảm bảo tính nhất quán của dữ liệu vẫn được duy trì. Như đã mô tả sau trong phần này, đôi khi bạn sử dụng sagas để duy trì tính nhất quán của dữ liệu.

Tương tác giữa dịch vụ và khối đơn khối, như đã mô tả trước đó, được tạo điều kiện thuận lợi bằng mã keo tích hợp. Hình 13.8 cho thấy cấu trúc của keo tích hợp. Nó bao gồm các bộ điều hợp trong dịch vụ và khối đơn khối giao tiếp bằng một số loại cơ chế IPC. Tùy thuộc vào yêu cầu, dịch vụ và khối đơn khối có thể tương tác qua REST hoặc chúng có thể sử dụng tin nhắn. Chúng thậm chí có thể giao tiếp bằng nhiều cơ chế IPC.



Tích hợp

keo dán

Giao diện lập trình ứng dụng (API)

bộ chuyển đổi

Giao diện lập trình ứng dụng (API)

bộ chuyển đổi

Đi ra ngoài

bộ chuyển đổi

Đến

bộ chuyển đổi

Khối đá nguyên khối

Dịch vụ

Đến

bộ chuyển đổi

Đi ra ngoài

bộ chuyển đổi

**Hình 13.8 Khi di chuyển một khối đơn nguyên sang các dịch vụ vi mô, các dịch vụ và khối đơn nguyên thường cần truy cập dữ liệu của nhau. Tương tác này được tạo điều kiện thuận lợi bởi keo tích hợp, bao gồm các bộ điều hợp triển khai API.Một số API dựa trên tin nhắn. Một số API khác dựa trên RPI.**

Ví dụ, Delayed Delivery Service sử dụng cả REST và sự kiện miền. Nó lấy thông tin liên hệ của khách hàng từ khối đơn bằng REST. Nó theo dõi trạng thái của Đơn hàng bằng cách đăng ký các sự kiện miền do khối đơn công bố.

Trong phần này, trước tiên tôi sẽ mô tả thiết kế của keo tích hợp. Tôi sẽ nói về các vấn đề mà nó giải quyết và các tùy chọn triển khai khác nhau. Sau đó, tôi sẽ mô tả các chiến lược quản lý giao dịch, bao gồm cả việc sử dụng sagas. Tôi sẽ thảo luận về cách đôi khi yêu cầu duy trì tính nhất quán của dữ liệu sẽ thay đổi thứ tự bạn trích xuất các dịch vụ.

Trước tiên chúng ta hãy xem xét thiết kế của keo tích hợp.

###### Thiết kế keo tích hợp

Khi triển khai một tính năng dưới dạng dịch vụ hoặc trích xuất một dịch vụ từ khối đơn, bạn phải phát triển keo tích hợp cho phép một dịch vụ cộng tác với khối đơn. Nó bao gồm mã trong cả dịch vụ và khối đơn sử dụng một số loại cơ chế IPC. Cấu trúc của keo tích hợp phụ thuộc vào loại cơ chế IPC được sử dụng. Ví dụ, nếu dịch vụ gọi khối đơn bằng REST, thì keo tích hợp bao gồm một máy khách REST trong dịch vụ và bộ điều khiển web trong khối đơn. Ngoài ra, nếu khối đơn đăng ký các sự kiện miền do dịch vụ phát hành, thì keo tích hợp bao gồm một bộ điều hợp phát hành sự kiện trong dịch vụ và trình xử lý sự kiện trong khối đơn.

**DTHIẾT KẾ KEO TÍCH HỢPGiao diện lập trình ứng dụng (API)**

Bước đầu tiên trong việc thiết kế keo tích hợp là quyết định API nào mà nó cung cấp cho logic miền. Có một vài kiểu giao diện khác nhau để lựa chọn, tùy thuộc vào việc bạn đang truy vấn dữ liệu hay cập nhật dữ liệu. Giả sử bạn đang làm việcing trên Delayed Delivery Service, cần phải lấy thông tin liên hệ của khách hàng từ khối đơn. Logic kinh doanh của dịch vụ không cần biết cơ chế IPC mà keo tích hợp sử dụng để lấy thông tin. Do đó, cơ chế đó phải được đóng gói bằng một giao diện. Vì Delayed Delivery Service đang truy vấn dữ liệu, nên việc định nghĩa một CustomerContactInfoRepository là hợp lý:

giao diện CustomerContactInfoRepository {

CustomerContactInfo tìmCustomerContactInfo(long customerId)

}

Logic kinh doanh của dịch vụ có thểgọi API này mà không biết cách tích hợp keo lấy dữ liệu.

Hãy xem xét một dịch vụ khác.Hãy tưởng tượng rằng bạn đang trích xuất quản lý giao hàng từ khối FTGO. Khối cần gọi Dịch vụ giao hàng để lên lịch, lên lịch lại và hủy giao hàng. Một lần nữa, các chi tiết của cơ chế IPC cơ bản không quan trọng đối với logic kinh doanh và nên được đóng gói bằng một giao diện. Trong trường hợp này, khối phải gọi một hoạt động dịch vụ, do đó việc sử dụng kho lưu trữ là không hợp lý. Một cách tiếp cận tốt hơn là xác định một giao diện dịch vụ, chẳng hạn như sau:

giao diện DeliveryService { void scheduleDelivery(...); void rescheduleDelivery(...); void cancelDelivery(...);

}

Logic kinh doanh của khối đơn khối sẽ gọi API này mà không cần biết cách nó được tích hợp triển khai như thế nào.

Bây giờ chúng ta đã biết về thiết kế giao diện, hãy cùng xem xét các kiểu tương tác và cơ chế IPC.

**PICKING MỘT PHONG CÁCH TƯƠNG TÁC VÀIPCCƠ CHẾ**

Một quyết định thiết kế quan trọng mà bạn phải đưa ra khi thiết kế keo tích hợp là lựa chọn các kiểu tương tác và cơ chế IPC cho phép dịch vụ và khối đơn khối cộng tác. Như đã mô tả trong chương 3, có một số kiểu tương tác và cơ chế IPC để lựa chọn. Kiểu nào bạn nên sử dụng phụ thuộc vào những gì một bên —dịch vụ hoặc khối đơn khối — cần để truy vấn hoặc cập nhật bên kia.

Nếu một bên cần truy vấn dữ liệu do bên kia sở hữu, có một số tùy chọn. Một tùy chọn là, như hình 13.9 cho thấy, cho bộ điều hợp triển khai giao diện kho lưu trữ để gọi API của nhà cung cấp dữ liệu. API này thường sẽ sử dụng kiểu tương tác yêu cầu/phản hồi, chẳng hạn như REST hoặc gRPC. Ví dụ: DelayedDịch vụ giao hàng có thể truy xuất thông tin liên hệ của khách hàng bằng cách sử dụng REST API được triển khai bởi khối FTGO.



Bị trì hoãn

Vận chuyểnDịch vụ

FTGO

khối đá nguyên khối

Kho thông tin liên hệ khách hàng

NHẬN/khách hàng/{customerId} NGHỈ NGƠI

Giao diện lập trình ứng dụng (API)

Khối đá nguyên khối

cơ sở dữ liệu

NGHỈ NGƠI

khách hàng

**Hình 13.9 Bộ điều hợp thực hiệnKho thông tin liên hệ khách hànggiao diện gọiREST API của monolith để thu thập thông tin khách hàng.**

Trong ví dụ này,Dịch vụ giao hàng chậm trễLogic miền của 's lấy thông tin liên hệ của khách hàng bằng cách gọiKho thông tin liên hệ khách hànggiao diện. Việc thực hiện-Việc đề cập đến giao diện này sẽ gọi đến REST API của khối đơn khối.

Một lợi ích quan trọng của việc truy vấn dữ liệu bằng cách gọi API truy vấn là tính đơn giản của nó. Nhược điểm chính là nó có khả năng không hiệu quả. Người dùng có thể cần thực hiện một số lượng lớn yêu cầu. Nhà cung cấp có thể trả về một lượng lớn dữ liệu. Một nhược điểm khác là nó làm giảm tính khả dụng vì nó là IPC đồng bộ. Do đó, việc sử dụng API truy vấn có thể không thực tế.

Một cách tiếp cận thay thế là người dùng dữ liệu duy trì một bản sao của dữ liệu, như thể hiện trong hình 13.10. Bản sao về cơ bản là chế độ xem CQRS. Người dùng dữ liệu giữ bản sao được cập nhật bằng cách đăng ký các sự kiện miền do nhà cung cấp dữ liệu công bố.



Bị trì hoãn

Vận chuyểnDịch vụ

FTGO

khối đá nguyên khối

Thông tin liên hệ khách hàng

Kho lưu trữ Khách hàng

lãnh địasự kiện

truy vấn()

Sự kiện

người đăng ký

Sự kiện

nhà xuất bản

Cơ sở dữ liệu cập nhật()

bộ chuyển đổi

Khối đá nguyên khối

cơ sở dữ liệu

Kênh sự kiện khách hàng

Dịch vụ

cơ sở dữ liệu

**Hình 13.10 Keo tích hợp sao chép dữ liệu từ khối đơn nguyên sang dịch vụ. Khối đơn nguyên công bố các sự kiện miền và trình xử lý sự kiện do dịch vụ triển khai sẽ cập nhật cơ sở dữ liệu của dịch vụ.**

Sử dụng bản sao có một số lợi ích. Nó tránh được chi phí truy vấn liên tục đến nhà cung cấp dữ liệu. Thay vào đó, như đã thảo luận khi mô tả CQRS trong chương 7, bạn có thể thiết kế bản sao để hỗ trợ các truy vấn hiệu quả. Tuy nhiên, một nhược điểm của việc sử dụng bản sao là sự phức tạp trong việc duy trì nó. Một thách thức tiềm ẩn, như được mô tả sau trong phần này, là nhu cầu sửa đổi khối đơn để xuất bản các sự kiện miền.

Bây giờ chúng ta đã thảo luận về cách thực hiện truy vấn, hãy xem xét cách thực hiện cập nhật. Một thách thức khi thực hiện cập nhật là nhu cầu duy trì tính nhất quán của dữ liệu trên toàn bộ dịch vụ và khối đơn. Bên thực hiện yêu cầu cập nhật (người yêu cầu) đã cập nhật hoặc cần cập nhật cơ sở dữ liệu của mình. Vì vậy, điều cần thiết là cả hai bản cập nhật đều phải diễn ra. Giải pháp là dịch vụ và khối đơn giao tiếp bằng cách sử dụng tin nhắn giao dịch được triển khai bởi một khuôn khổ, chẳng hạn như Eventuate Tram. Trong các tình huống đơn giản, người yêu cầu có thể gửi tin nhắn thông báo hoặc công bố sự kiện để kích hoạt bản cập nhật. Trong các tình huống phức tạp hơn, người yêu cầu phải sử dụng saga để duy trì tính nhất quán của dữ liệu. Phần 13.3.2 thảo luận về những tác động của việc sử dụng saga.

**TÔITHỰC HIỆN MỘT CHƯƠNG TRÌNH CHỐNG-TẦNG THAM NHŨNG**

Hãy tưởng tượng bạn đang triển khai một tính năng mới như một dịch vụ hoàn toàn mới. Bạn không bị hạn chế bởi cơ sở mã của khối, do đó bạn có thể sử dụng các kỹ thuật phát triển hiện đại

chẳng hạn như DDD và phát triển một mô hình miền mới nguyên sơ. Ngoài ra, vì miền của khối FTGO được định nghĩa kém và hơi lỗi thời, nên có thể bạn sẽ mô hình hóa các khái niệm theo cách khác. Do đó, mô hình miền của dịch vụ của bạn sẽ có lớp khác nhautên, tên trường và giá trị trường. Ví dụ, Delayed Delivery Service có một thực thể Delivery với các trách nhiệm tập trung hẹp, trong khi FTGO monolith có một thực thể Order với số lượng trách nhiệm quá mức. Vì hai mô hình miền khác nhau, bạn phải triển khai cái mà DDD gọi là lớp chống hỏng (ACL) để dịch vụ có thể giao tiếp với monolith.

**Mẫu: Lớp chống tham nhũng**

Một lớp phần mềm chuyển đổi giữa hai mô hình miền khác nhau để ngăn chặn các khái niệm từ một mô hình làm ô nhiễm mô hình khác. Xem[https://microservices.io/](https://microservices.io/patterns/refactoring/anti-corruption-layer.html) [mẫu/tái cấu trúc/lớp chống tham nhũng.html](https://microservices.io/patterns/refactoring/anti-corruption-layer.html).

Mục tiêu của ACL là ngăn chặn mô hình miền của khối đơn nguyên cũ làm ô nhiễm mô hình miền của dịch vụ. Đó là một lớp mã dịch giữa cácmô hình miền. Ví dụ, như hình 13.11 cho thấy,Dịch vụ giao hàng chậm trễcó mộtKho thông tin liên hệ khách hànggiao diện, xác định mộttìmKhách hàngLiên hệ-Thông tin()phương pháp trả vềThông tin liên hệ khách hàng. Lớp thực hiệnKho thông tin liên hệ khách hànggiao diện phải dịch giữa ngôn ngữ phổ biến củaDịch vụ giao hàng chậm trễvà khối FTGO.



**Ngôn ngữ dịch vụ phổ biến**

Bị trì hoãn

Dịch vụ giao hàng

**Lớp chống tham nhũng**

FTGO

khối đá nguyên khối

Khách hàng Giao diện lập trình ứng dụng (API)

Thông tin liên hệ

Kho lưu trữ

NHẬN/người dùng/{userId}

NGHỈ NGƠI

Giao diện lập trình ứng dụng (API)

Khách hàng REST

Lớp đơn khối

Lớp dịch thuật

**Ngôn ngữ phổ biến của khối nguyên khối**

**Hình 13.11 Bộ điều hợp dịch vụ gọi khối đơn phải dịch chuyển giữa mô hình miền của dịch vụvà mô hình miền của khối đơn khối.**

Việc triển khai findCustomerContactInfo() sẽ gọi khối FTGO để lấy thông tin khách hàng và dịch phản hồi sang CustomerContact-Info. Trong ví dụ này, bản dịch khá đơn giản, nhưng trong các tình huống khác, nó có thể khá phức tạp và liên quan đến, ví dụ, các giá trị ánh xạ như mã trạng thái.

Một người đăng ký sự kiện, người sử dụng các sự kiện miền, cũng có một ACL. Các sự kiện miền là một phần của mô hình miền của nhà xuất bản. Một trình xử lý sự kiện phải dịch các sự kiện miền sang mô hình miền của người đăng ký. Ví dụ, như hình 13.12 cho thấy,Khối FTGO xuất bản các sự kiện miền Order. Delivery Service có trình xử lý sự kiện đăng ký các sự kiện đó.



**Ngôn ngữ dịch vụ phổ biến**

**Lớp chống tham nhũng**

FTGO

khối đá nguyên khối

Bị trì hoãn

Dịch vụ giao hàng

Đặt hàng

sự kiện

Kênh sự kiện

Khách hàng nhắn tin

Lớp dịch thuật

Trình xử lý sự kiện

Nhà xuất bản sự kiện

**Ngôn ngữ phổ biến của khối nguyên khối**

**Hình 13.12 Trình xử lý sự kiện phải dịch từ mô hình miền của nhà xuất bản sự kiện sang miền của người đăng kýngười mẫu.**

Trình xử lý sự kiện phải dịch các sự kiện miền từ ngôn ngữ miền của khối đơn sang ngôn ngữ của Dịch vụ phân phối. Nó có thể cần ánh xạ tên lớp và thuộc tính và có khả năng là các giá trị thuộc tính.

Không chỉ các dịch vụ sử dụng lớp chống tham nhũng. Một khối đơn khối cũng sử dụng ACL khi gọi dịch vụ và khi đăng ký các sự kiện miền do dịch vụ công bố. Ví dụ, khối đơn khối FTGO lên lịch phân phối bằng cách gửi thông báotin nhắn đến Delivery Service. Nó gửi thông báo bằng cách gọi một phương thức trên giao diện DeliveryService. Lớp triển khai dịch các tham số của nó thành một tin nhắn mà Delivery Service hiểu được.

**HOW THE MONOLITH XUẤT BẢN VÀ ĐĂNG KÝ CÁC SỰ KIỆN CỦA DOMAIN**

Sự kiện miền là một cơ chế cộng tác quan trọng. Một dịch vụ mới phát triển có thể dễ dàng xuất bản và sử dụng sự kiện. Nó có thể sử dụng một trong các cơ chế được mô tả trong chương 3, chẳng hạn như khung Eventuate Tram. Một dịch vụ thậm chí có thể xuất bản sự kiện bằng cách sử dụng nguồn sự kiện, được mô tả trong chương 6. Tuy nhiên, việc thay đổi khối đơn để xuất bản và sử dụng sự kiện có khả năng là một thách thức. Hãy cùng xem lý do tại sao.

Có một vài cách khác nhau mà một khối đơn có thể xuất bản các sự kiện miền. Một cách tiếp cận là sử dụng cùng một cơ chế xuất bản sự kiện miền được sử dụng bởi

dịch vụ. Bạn tìm thấy tất cả các vị trí trong mã thay đổi một thực thể cụ thể và chèn lệnh gọi đến API xuất bản sự kiện. Vấn đề với cách tiếp cận này là việc thay đổi một khối đơn không phải lúc nào cũng dễ dàng. Có thể mất thời gian và dễ xảy ra lỗi khi xác định vị trí tất cả các vị trí và chèn lệnh gọi để xuất bản sự kiện. Tệ hơn nữa, một số logic kinh doanh của khối đơn có thể bao gồm các thủ tục được lưu trữ không thể dễ dàng xuất bản sự kiện miền.

Một cách tiếp cận khác là công bố sự kiện miền ở cấp cơ sở dữ liệu. Ví dụ, bạn có thể sử dụng tailing logic giao dịch hoặc polling, được mô tả trong chương 3. Một lợi ích chính của việc sử dụng tailing giao dịch là bạn không phải thay đổi monolith. Nhược điểm chính của việc công bố sự kiện ở cấp cơ sở dữ liệu là thường khó xác định lý do cập nhật và công bố sự kiện kinh doanh cấp cao phù hợp. Do đó, dịch vụ thường sẽ công bố các sự kiện biểu thị các thay đổi đối với bảng chứ không phải các thực thể kinh doanh.

May mắn thay, thường thì khối đơn giản hơn khi đăng ký các sự kiện miền được xuất bản dưới dạng dịch vụ. Thông thường, bạn có thể viết trình xử lý sự kiện bằng một khuôn khổ, chẳng hạn như Eventuate Tram. Nhưng đôi khi, khối đơn thậm chí còn gặp khó khăn khi đăng ký các sự kiện. Ví dụ, khối đơn có thể được viết bằng ngôn ngữ không có máy khách môi giới tin nhắn. Trong tình huống đó, bạn cần viết một ứng dụng "trợ giúp" nhỏ đăng ký các sự kiện và cập nhật cơ sở dữ liệu của khối đơn trực tiếp.

Bây giờ chúng ta đã xem xét cách thiết kế chất kết dính tích hợp cho phép dịch vụ và khối đơn lẻ cộng tác, hãy cùng xem xét một thách thức khác mà bạn có thể gặp phải khi di chuyển sang dịch vụ vi mô: duy trì tính nhất quán của dữ liệu trên toàn bộ dịch vụ và khối đơn lẻ.

###### Duy trì tính nhất quán của dữ liệu trên một dịch vụ và một khối thống nhất

Khi bạn phát triển một dịch vụ, bạn có thể thấy khó khăn trong việc duy trì tính nhất quán của dữ liệu trên toàn bộ dịch vụ và khối đơn. Một hoạt động dịch vụ có thể cần cập nhật dữ liệu trong khối đơn hoặc một hoạt động khối đơn có thể cần cập nhật dữ liệu trong dịch vụ.vice. Ví dụ, hãy tưởng tượng bạn trích xuất Kitchen Service từ khối đơn. Bạn sẽ cần thay đổi các hoạt động quản lý đơn hàng của khối đơn, chẳng hạn như create-Order() và cancelOrder(), để sử dụng sagas nhằm giữ cho Ticket nhất quán với Order.

Tuy nhiên, vấn đề khi sử dụng sagas là khối đơn có thể không phải là người tham gia tự nguyện. Như đã mô tả trong chương 4, sagas phải sử dụng các giao dịch bù trừ để hoàn tác các thay đổi. Ví dụ, Create Order Saga bao gồm một giao dịch bù trừđánh dấu Đơn hàng bị từ chối nếu Đơn hàng bị Dịch vụ Nhà bếp từ chối. Vấn đề với việc bù trừ các giao dịch trong khối đơn là bạn có thể cần thực hiện nhiều thay đổi tốn thời gian đối với khối đơn để hỗ trợ chúng. Khối đơn cũng có thể cần triển khai các biện pháp đối phó để xử lý tình trạng thiếu cô lập giữa các saga. Chi phí cho những thay đổi mã này có thể là trở ngại lớn đối với việc trích xuất dịch vụ.

**Thuật ngữ chính của saga**

Tôi sẽ đề cập đến các câu chuyện trong chương 4. Sau đây là một số thuật ngữ chính:

* *Truyện dài*—Một chuỗi các giao dịch cục bộ được phối hợp thông qua nhắn tin không đồng bộ.
* *Giao dịch bù trừ*—Một giao dịch hoàn tác các bản cập nhật được thực hiện bởigiao dịch địa phương.
* *Biện pháp đối phó*—Một kỹ thuật thiết kế được sử dụng để xử lý tình trạng thiếu cô lậpgiữa các sử thi.
* *Khóa ngữ nghĩa*—Một biện pháp đối phó đặt cờ trong một bản ghi đang đượcđược cập nhật bởi một saga.
* *Giao dịch có thể đền bù*—Một giao dịchđiều đó cần một giao dịch bù trừ vì một trong những giao dịch tiếp theo trong câu chuyện có thể thất bại.
* *Giao dịch trục*—Một giao dịch là điểm thành công/không thành công của saga. Nếu thành công, saga sẽ chạy đến khi hoàn thành.
* *Giao dịch có thể thu hồi*—Một giao dịch theo sau giao dịch trục và được đảm bảo thành công.

May mắn thay, nhiều saga rất dễ triển khai. Như đã đề cập trong chương 4, nếu các giao dịch của khối đơn là giao dịch trục hoặc giao dịch có thể thử lại, thì việc triển khai saga sẽ rất dễ dàng. Bạn thậm chí có thể đơn giản hóa việc triển khai bằng cách sắp xếp cẩn thận trình tự trích xuất dịch vụ để các giao dịch của khối đơn không bao giờ cần phải bù trừ. Hoặc có thể tương đối khó để thay đổi khối đơn để hỗ trợ các giao dịch bù trừ. Để hiểu tại sao việc triển khai các giao dịch bù trừ trong khối đơn đôi khi lại khó khăn, chúng ta hãy xem xét một số ví dụ, bắt đầu bằng một ví dụ đặc biệt khó khăn.

**TTHÁCH THỨC THAY ĐỔI MONOLITH ĐỂ HỖ TRỢ CÁC GIAO DỊCH CÓ THỂ BỒI THƯỜNG**

Hãy cùng tìm hiểu sâu hơn về vấn đề bù trừ các giao dịch mà bạn sẽ cần giải quyết khi trích xuất Kitchen Service từ khối đơn. Việc tái cấu trúc này bao gồm việc chia táchThực thể đơn hàng và tạo thực thể Vé trong Dịch vụ nhà bếp. Nó tác động đến nhiềucác lệnh được triển khai bởi khối thống nhất, bao gồm createOrder().

Khối đơn vị triển khai lệnh createOrder() như một giao dịch ACID duy nhất bao gồm các bước sau:

**1** Xác thực thông tin chi tiết đơn hàng.

**2** Xác minh rằng người tiêu dùng có thể đặt hàng.

**3** Cho phép thẻ tín dụng của người tiêu dùng.

**4** Tạo mộtĐặt hàng.

Bạn cần thay thế giao dịch ACID này bằng một câu chuyện bao gồm các bước sau:

**1** Trong khối đá nguyên khối

* Tạo mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.
* Xác minh rằng người tiêu dùng có thể đặt hàng.

**2** TRONGcáiDịch vụ nhà bếp

* + Xác thực đơn hàngchi tiết.
  + Tạo mộtVétrongĐANG TẠO\_ĐANG CHỜtình trạng.

**3** Trong khối đá nguyên khối

* + Ủy quyềnthẻ tín dụng của người tiêu dùng.
  + Thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNH.

**4** TRONGDịch vụ nhà bếp

* + Thay đổi trạng thái củaVéĐẾNĐANG CHỜ CHẤP NHẬN.

Câu chuyện này tương tự nhưTạo đơn hàngSagađược mô tả trong chương 4. Nó bao gồm bốn giao dịch cục bộ, hai trong khối và hai trongDịch vụ nhà bếp. Đầu tiêngiao dịchtạo ra mộtĐặt hàngtrongĐANG CHỜ PHÊ DUYỆTtrạng thái. Giao dịch thứ hai tạo ra mộtVétrongĐANG TẠO\_ĐANG CHỜtrạng thái. Giao dịch thứ ba cho phépNgười tiêu dùngthẻ tín dụng và thay đổi trạng thái của đơn hàng thànhTÁN THÀNH. Thứ tưvà giao dịch cuối cùng thay đổi trạng thái củaVéĐẾNĐANG CHỜ CHẤP NHẬN.

Thách thức khi triển khai saga này là bước đầu tiên, tạo ra Order, phải có thể bù trừ được. Đó là vì giao dịch cục bộ thứ hai, xảy ra trong Kitchen Service, có thể không thành công và yêu cầu monolith phải hoàn tác các bản cập nhật được thực hiện bởi giao dịch cục bộ đầu tiên. Do đó, thực thể Order cần có APPROVAL\_PENDING, một biện pháp đối phó khóa ngữ nghĩa, được mô tả trong chương 4, cho biết Order đang trong quá trình được tạo.

CácVấn đề với việc giới thiệu trạng thái thực thể Order mới là nó có khả năng yêu cầu những thay đổi rộng rãi đối với khối đơn. Bạn có thể cần thay đổi mọi vị trí trong mã liên quan đến thực thể Order. Việc thực hiện những thay đổi rộng rãi như vậy đối với khối đơn tốn thời gian và không phải là khoản đầu tư tốt nhất cho các nguồn lực phát triển. Nó cũng có khả năng rủi ro, vì khối đơn thường khó kiểm tra.

**SAGAS DON'T LUÔN YÊU CẦU MONOLITH HỖ TRỢ CÁC GIAO DỊCH CÓ THỂ BỒI THƯỜNG**

Saga có tính đặc thù cao theo miền. Một số, chẳng hạn như cái mà chúng ta vừa xem xét, yêu cầu khối đơn để hỗ trợ các giao dịch bù trừ. Nhưng hoàn toàn có thể khi bạn trích xuất một dịch vụ, bạn có thể thiết kế các saga không yêu cầu khối đơn để triển khai các giao dịch bù trừ. Đó là vì khối đơn chỉ cần hỗ trợ các giao dịch bù trừ nếu các giao dịch theo sau giao dịch của khối đơn có thể thất bại. Nếu mỗi giao dịch của khối đơn là giao dịch trục hoặc giao dịch có thể thử lại, thì khối đơn không bao giờ cần thực hiện giao dịch bù trừ. Do đó, bạn chỉ cần thực hiện các thay đổi tối thiểu đối với khối đơn để hỗ trợ các saga.

Ví dụ, hãy tưởng tượng rằng thay vì trích xuấtDịch vụ nhà bếp, bạn trích xuấtDịch vụ đặt hàng. Việc tái cấu trúc này bao gồm việc chia táchĐặt hàngthực thể và tạo ra một thực thể thu gọnĐặt hàngthực thể trongDịch vụ đặt hàng. Nó cũng tác động đến nhiều lệnh, bao gồmtạo đơn hàng(), được di chuyển từ khối đá nguyên khối đếnDịch vụ đặt hàng. Để trích xuấtDịch vụ đặt hàng, bạn cần phải thay đổitạo đơn hàng()lệnh sử dụng saga, sử dụng các bước sau:

**1** Dịch vụ đặt hàng

* Tạo mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.

**2** Khối đá nguyên khối

* Xác minh rằng người tiêu dùng có thể đặt hàng.
* Xác thực thông tin chi tiết đơn hàng và tạoVé.
* Ủy quyềnthẻ tín dụng của người tiêu dùng.

**3** Dịch vụ đặt hàng

* Thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNH.

Truyện này bao gồm ba giao dịch cục bộ, một giao dịch trong khối và hai giao dịch trong OrderDịch vụ. Giao dịch đầu tiên, nằm trong Order Service, tạo ra một Order ở trạng thái APPROVAL\_PENDING. Giao dịch thứ hai, nằm trong monolith, xác minh rằng người tiêu dùng có thể đặt hàng, ủy quyền thẻ tín dụng của họ và tạo ra một Ticket. Giao dịch thứ ba, nằm trong Order Service, thay đổi trạng thái của Order thành APPROVED.

Giao dịch của khối đơn là giao dịch trục của saga—điểm không thể quay lại của saga. Nếu giao dịch của khối đơn hoàn tất, thì saga sẽ chạy cho đến khi hoàn tất. Chỉ có bước đầu tiên và bước thứ hai của saga này có thể thất bại. Giao dịch thứ ba không thể thất bại, vì vậy giao dịch thứ hai trong khối đơn không bao giờ cần phải khôi phục. Do đó, tất cả sự phức tạp của việc hỗ trợ các giao dịch có thể bù trừ đều nằm trong Order Service, có thể kiểm tra được nhiều hơn so với khối đơn.

Nếu tất cả các saga mà bạn cần viết khi trích xuất một dịch vụ đều có cấu trúc này

ture, bạn sẽ cần thực hiện ít thay đổi hơn nhiều đối với khối đơn. Hơn nữa, có thể sắp xếp cẩn thận việc trích xuất các dịch vụ để đảm bảo rằng các giao dịch của khối đơn là các giao dịch trục hoặc các giao dịch có thể thử lại. Hãy cùng xem cách thực hiện điều đó.

**SĐỂ TRÁNH VIỆC TRÍCH XUẤT CÁC DỊCH VỤ TRÁNH THỰC HIỆN CÁC GIAO DỊCH BỒI THƯỜNG TRONG MONOLITH**

Như chúng ta vừa thấy, việc trích xuất Dịch vụ Nhà bếp yêu cầu khối đơn phải triển khai các giao dịch bù trừ, trong khi việc trích xuất Dịch vụ Đơn hàng thì không. Điều này cho thấy thứ tự bạn trích xuất các dịch vụ rất quan trọng. Bằng cách sắp xếp cẩn thận việc trích xuất các dịch vụ, về mặt tiềm năng, bạn có thể tránh phải thực hiện các sửa đổi rộng rãi đối với khối đơn để hỗ trợ các giao dịch có thể bù trừ. Chúng ta có thể đảm bảo rằng các giao dịch của khối đơn là các giao dịch trục hoặc các giao dịch có thể thử lại. Ví dụ, nếu trước tiên chúng ta trích xuất Dịch vụ Đơn hàng từ khối đơn FTGO rồi sau đó trích xuất Dịch vụ Người tiêu dùng, thì việc trích xuất Dịch vụ Nhà bếp sẽ rất đơn giản. Chúng ta hãy xem xét kỹ hơn cách thực hiện điều đó.

Một khi chúng ta cóđã trích xuấtDịch vụ khách hàng, cáctạo đơn hàng()lệnh sử dụng câu chuyện sau:

**1** Dịch vụ đặt hàng: tạo ra mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.

**2** Dịch vụ khách hàng: xác minh rằng người tiêu dùng có thể đặt hàng.

**3** Khối đá nguyên khối

* + Xác thực thông tin chi tiết đơn hàng và tạoVé.
  + Ủy quyềnthẻ tín dụng của người tiêu dùng.

**4** Dịch vụ đặt hàng: thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNH.

Trong câu chuyện này, giao dịch của khối nguyên khối chính là giao dịch trục.Dịch vụ đặt hàng thực hiện giao dịch có thể bồi thường.

Hiện naymà chúng tôi đã trích xuấtDịch vụ khách hàng, chúng ta có thể trích xuấtDịch vụ nhà bếp. Nếu chúng tatrích xuất dịch vụ này,tạo đơn hàng()lệnh sử dụng câu chuyện sau:

**1** Dịch vụ đặt hàng: tạo ra mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.

**2** Dịch vụ khách hàng: xác minh rằng người tiêu dùng có thể đặt hàng.

**3** Dịch vụ nhà bếp: xác thực thông tin chi tiết về đơn hàng và tạo một ĐANG CHỜVé.

**4** Monolith: cho phép thẻ tín dụng của người tiêu dùng.

**5** Dịch vụ nhà bếp: thay đổi trạng thái củaVéĐẾNTÁN THÀNH.

**6** Dịch vụ đặt hàng: thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNH.

Trong câu chuyện này, giao dịch của khối đơn vẫn là giao dịch trục. Dịch vụ đặt hàng và

Dịch vụ nhà bếpthực hiện các giao dịch có thể bù trừ.

Chúng ta thậm chí có thể tiếp tục tái cấu trúc khối đơn bằng cách trích xuấtDịch vụ kế toán. Nếu chúng ta trích xuất dịch vụ này,tạo đơn hàng()lệnh sử dụng câu chuyện sau:

**1** Dịch vụ đặt hàng: tạo ra mộtĐặt hàngtrong mộtĐANG CHỜ PHÊ DUYỆTtình trạng.

**2** Dịch vụ khách hàng: xác minh rằng người tiêu dùng có thể đặt hàng.

**3** Dịch vụ nhà bếp: xác thực thông tin chi tiết về đơn hàng và tạo một ĐANG CHỜVé.

**4** Dịch vụ kế toán: cho phép thẻ tín dụng của người tiêu dùng.

**5** Dịch vụ nhà bếp: thay đổi trạng thái củaVéĐẾNTÁN THÀNH.

**6** Dịch vụ đặt hàng: thay đổi trạng thái củaĐặt hàngĐẾNTÁN THÀNH.

Như bạn có thể thấy, bằng cách sắp xếp cẩn thận các lần trích xuất, bạn có thể tránh sử dụng các saga đòi hỏi phải thực hiện các thay đổi phức tạp đối với khối đơn. Bây giờ chúng ta hãy xem cách xử lý bảo mật khi di chuyển sang kiến ​​trúc dịch vụ vi mô.

###### Xử lý xác thực và ủy quyền

Một vấn đề thiết kế khác mà bạn cần giải quyết khi tái cấu trúc ứng dụng monolithic thành kiến ​​trúc microservice là điều chỉnh cơ chế bảo mật của monolith để hỗ trợ các dịch vụ. Chương 11 mô tả cách xử lý bảo mật trong kiến ​​trúc microservice. Ứng dụng dựa trên microservice sử dụng các mã thông báo, chẳng hạn như mã thông báo Web JSON (JWT), để truyền xung quanh danh tính người dùng. Điều đó khá khác so với ứng dụng monolithic truyền thống thông thường sử dụng trạng thái phiên trong bộ nhớ và truyền xung quanh danh tính người dùng bằng cách sử dụng luồng cục bộ. Thách thức khi chuyển đổi ứng dụng monolithic thành kiến ​​trúc microservice là bạn cần hỗ trợ đồng thời cả cơ chế bảo mật dựa trên monolithic và JWT.

May mắn thay, có một cách đơn giản để giải quyết vấn đề này mà chỉ yêu cầu bạn thực hiện một thay đổi nhỏ đối với trình xử lý yêu cầu đăng nhập của khối đơn. Hình 13.13

cho thấy cách thức hoạt động này. Trình xử lý đăng nhập trả về một cookie bổ sung, trong ví dụ này tôi gọi là USERINFO, chứa thông tin người dùng, chẳng hạn như ID người dùng và vai trò. Trình duyệt bao gồm cookie đó trong mọi yêu cầu. Cổng API trích xuất thông tin từ cookie và bao gồm thông tin đó trong các yêu cầu HTTP mà nó thực hiện cho một dịch vụ. Do đó, mỗi dịch vụ đều có quyền truy cập vào thông tin người dùng cần thiết.

Truy vấn



Trình xử lý đăng nhập

Cơ sở dữ liệu người dùng

Cung cấp cookie phiên.

Đăng nhập bằng ID người dùng và mật khẩu.

POST/đăng nhập

Trả về cookie phiên.

|  |  |  |
| --- | --- | --- |
| Đăng nhập bằng người dùng | | Giao diện lập trình ứng dụng (API)  cổng vào |
| Ứng dụng SPA dựa trên trình duyệt | ID và mật khẩu.  POST/đăng nhập |
|  |
|  | |

HTTP/1.1 200 Đồng ý

Đặt cookie: JSESSIONID=... Đặt cookie: USERINFO=TOKEN

...

Khởi tạo

userId: xxx vai trò:[a, b, c]

...

**Chứa thông tin người dùng, chẳng hạn như ID và vai trò**

NHẬN/đơn hàng

Cookie: JSESSIONID=... Cookie: USERINFO=TOKEN

...

Cung cấp JWT.

GET/orders Ủy quyền: TOKEN

...

Khối FTGO

**Trong bộ nhớphiên họp**

Dịch vụ Lịch sử đơn hàng

Trình xử lý yêu cầu lịch sử đơn hàng

**Hình 13.13 Trình xử lý đăng nhập được nâng cao để thiết lậpTHÔNG TIN NGƯỜI DÙNGcookie, là JWT chứa thông tin người dùng.Cổng APIchuyển giaoTHÔNG TIN NGƯỜI DÙNGcookie vào tiêu đề ủy quyền khi nó gọi mộtdịch vụ.**

Trình tự các sự kiện như sau:

**1** Khách hàng gửi yêu cầu đăng nhập có chứa thông tin đăng nhập của người dùng.

**2** Cổng APIđịnh tuyến yêu cầu đăng nhập đến khối FTGO.

**3**Khối đơn khối trả về phản hồi có chứa cookie phiên JSESSIONID và cookie USERINFO, chứa thông tin người dùng, chẳng hạn như ID và vai trò.

**4**Khách hàng đưa ra yêu cầu, bao gồm cookie USERINFO, để thực hiện một hoạt động.

**5** Cổng APIxác nhậnTHÔNG TIN NGƯỜI DÙNGcookie và bao gồm nó trongỦy quyềntiêu đề của yêu cầu mà nó thực hiện cho dịch vụ. Dịch vụ xác thựcTHÔNG TIN NGƯỜI DÙNGmã thông báo và trích xuất thông tin người dùng.

Chúng ta hãy nhìn vàoTrình xử lý đăng nhậpVàCổng APIchi tiết hơn.

**TĐÁ MONOLITH'SLĐĂNG NHẬPHANDLER ĐẶTTHÔNG TIN NGƯỜI DÙNGBÁNH QUY**

LoginHandler xử lý POST thông tin đăng nhập của người dùng. Nó xác thực người dùng và lưu trữ thông tin về người dùng trong phiên. Nó thường được triển khai bởi

khuôn khổ bảo mật, chẳng hạn như Spring Security hoặc Passport cho NodeJS. Nếu ứng dụng được cấu hình để sử dụng phiên trong bộ nhớ mặc định, phản hồi HTTP sẽ đặt cookie phiên, chẳng hạn như JSESSIONID. Để hỗ trợ di chuyển sang các dịch vụ siêu nhỏ,LoginHandler cũng phải thiết lập cookie USERINFO có chứa JWT để mô tả người dùng.

**TANH TAGiao diện lập trình ứng dụng (API)BẢN ĐỒ CỔNG THÔNG TINTHÔNG TIN NGƯỜI DÙNGCOOKIE ĐẾNMỘTTIÊU ĐỀ ỦY QUYỀN**

Cổng API, như được mô tả trong chương8, chịu trách nhiệm định tuyến yêu cầu và biên soạn API. Nó xử lý từng yêu cầu bằng cách thực hiện một hoặc nhiều yêu cầu đến khối đơn và các dịch vụ. Khi cổng API gọi một dịch vụ, nó sẽ xác thực USERINFOcookie và chuyển nó đến dịch vụ trong tiêu đề Ủy quyền của yêu cầu HTTP. Bằng cách ánh xạ cookie đến tiêu đề Ủy quyền, cổng API đảm bảo rằng nó chuyển danh tính người dùng đến dịch vụ theo cách chuẩn không phụ thuộc vào loại máy khách.

Cuối cùng, chúng ta có thể sẽ trích xuấtđăng nhập và quản lý người dùng vào các dịch vụ. Nhưng như bạn có thể thấy, chỉ bằng cách thực hiện một thay đổi nhỏ đối với trình xử lý đăng nhập của khối, giờ đây các dịch vụ có thể truy cập thông tin người dùng. Điều này cho phép bạn tập trung vào việc phát triển các dịch vụ mang lại giá trị lớn nhất cho doanh nghiệp và trì hoãn việc trích xuất các dịch vụ ít giá trị hơn, chẳng hạn như quản lý người dùng.

Bây giờ chúng ta đã xem xét cách xử lý bảo mật khi tái cấu trúc thành các dịch vụ vi mô, hãy xem một ví dụ về việc triển khai tính năng mới dưới dạng dịch vụ.

#### Triển khai một tính năng mới dưới dạng dịch vụ: xử lýđơn hàng giao nhầm

Giả sử bạn được giao nhiệm vụ cải thiện cách FTGO xử lý các đơn hàng bị giao nhầmđơn hàng. Ngày càng có nhiều khách hàng phàn nàn về cách dịch vụ khách hàng xử lý các đơn hàng không được giao. Phần lớn các đơn hàng được giao đúng hạn, nhưng thỉnh thoảng các đơn hàng được giao trễ hoặc không được giao. Ví dụ, người giao hàng bị chậm trễ do giao thông xấu bất ngờ, vì vậy đơn hàng được nhận và giao muộn. Hoặc có thể khi người giao hàng đến nhà hàng, nhà hàng đã đóng cửa và không thể giao hàng được. Tệ hơn nữa, lần đầu tiên bộ phận dịch vụ khách hàng nghe về việc giao nhầm là khi họ nhận được email tức giận từ một khách hàng không hài lòng.

**Một câu chuyện có thật: Kem tôi mất**

Một tối thứ Bảy, tôi thấy lười và đã đặt hàng qua một ứng dụng giao đồ ăn nổi tiếng để được giao kem từ Smitten. Nó không bao giờ xuất hiện. Thông tin liên lạc duy nhất từ ​​công ty là một email vào sáng hôm sau nói rằng đơn hàng của tôi đã bị hủy. Tôi cũng nhận được một thư thoại từ một nhân viên dịch vụ khách hàng rất bối rối, rõ ràng là cô ấy không biết mình đang gọi về chuyện gì. Có lẽ cuộc gọi được thúc đẩy bởi một trong những dòng tweet của tôi mô tả những gì đã xảy ra. Rõ ràng là công ty giao hàng đã không thiết lập bất kỳ cơ chế nào để xử lý đúng cách những sai sót không thể tránh khỏi.

Nguyên nhân gốc rễ của nhiều vấn đề giao hàng này là thuật toán lập lịch giao hàng thô sơ được ứng dụng FTGO sử dụng. Một trình lập lịch tinh vi hơn đang được phát triển nhưng sẽ không hoàn thành trong vài tháng nữa. Giải pháp tạm thời là FTGO chủ động xử lý các đơn hàng bị chậm trễ hoặc bị hủy bằng cách xin lỗi khách hàng và trong một số trường hợp là bồi thường trước khi khách hàng phàn nàn.

Nhiệm vụ của bạn là triển khai một tính năng mới có thể thực hiện những chức năng sau:

**1** Thông báo cho khách hàng khi đơn hàng của họ không được giao đúng hạn.

**2**Thông báo cho khách hàng khi đơn hàng của họ không thể được giao vì không thể đến lấy trước khi nhà hàng đóng cửa.

**3**Thông báo cho bộ phận chăm sóc khách hàng khi đơn hàng không thể giao đúng hạn để họ có thể chủ động khắc phục tình hình bằng cách bồi thường cho khách hàng.

**4** Theo dõi số liệu thống kê giao hàng.

Tính năng mới này khá đơn giản. Mã mới phải theo dõi trạng thái của từng Đơn hàng và nếu Đơn hàng không thể được giao như đã hứa, mã phải thông báo cho khách hàng và bộ phận hỗ trợ khách hàng, ví dụ như bằng cách gửi email.

Nhưng làm thế nào—hay có lẽ chính xác hơn, ở đâu—nênbạn triển khai tính năng mới này? Một cách tiếp cận là triển khai một mô-đun mới trong khối đơn. Vấn đề ở đây là việc phát triển và thử nghiệm mã này sẽ khó khăn. Hơn nữa, cách tiếp cận này làm tăng kích thước của khối đơn và do đó khiến địa ngục khối đơn trở nên tồi tệ hơn. Hãy nhớ lại Luật Lỗ hổng từ trước: khi bạn đang ở trong một cái lỗ, tốt nhất là ngừng đào. Thay vì làm cho khối đơn lớn hơn, một cách tiếp cận tốt hơn nhiều là triển khai các tính năng mới này dưới dạng dịch vụ.

###### Thiết kế dịch vụ giao hàng chậm trễ

Chúng tôi sẽ triển khai tính năng này như một dịch vụ có tên làDịch vụ đặt hàng chậm trễ. Hình 13.14hiển thị kiến ​​trúc ứng dụng FTGO sau khi triển khai dịch vụ này. Ứng dụng bao gồm khối FTGO,Dịch vụ giao hàng chậm trễvà mộtCổng API.Dịch vụ giao hàng chậm trễcó một API định nghĩa một hoạt động truy vấn duy nhất được gọi làlấyDelayedOrder(), trả về các mục hiện đang bị trì hoãn hoặc chưa được giaocó thể ra lệnh.Cổng APItuyến đườnglấyDelayedOrder()yêu cầu dịch vụ và tất cả các yêu cầu khác đối với khối đơn. Keo tích hợp cung cấpDịch vụ đặt hàng chậm trễvới quyền truy cập vào dữ liệu của khối đơn.

CácDịch vụ đặt hàng chậm trễMô hình miền của 'bao gồm nhiều thực thể khác nhau, bao gồmThông báo đơn hàng bị trì hoãn,Đặt hàng, VàNhà hàng. Logic cốt lõi được thực hiện bởiDịch vụ đặt hàng chậm trễlớp. Nó được gọi định kỳ bởi một bộ đếm thời gian để tìm các đơn hàng sẽ không được giao đúng hạn. Nó thực hiện điều đó bằng cách truy vấnĐơn hàngVàNhà hàng. Nếu mộtĐặt hàngkhông thể giao hàng đúng hạn,Dịch vụ đặt hàng chậm trễthông báo cho người tiêu dùng và bộ phận dịch vụ khách hàng.

Delayed Order Service không sở hữu các thực thể Order và Restaurant. Thay vào đó, dữ liệu này được sao chép từ khối FTGO. Hơn nữa, dịch vụ này không lưu trữ thông tin liên hệ của khách hàng mà thay vào đó lấy thông tin đó từ khối.

|  |  |
| --- | --- |
| «thực thể»  Nhà hàng | |
|  |  |
| «entity» Giờ mở cửa | |

**Cần phải thiết kế.**

Lấy đơn hàng chậm trễ()

NGHỈ NGƠI

Giao diện lập trình ứng dụng (API)

NGHỈ NGƠI

Giao diện lập trình ứng dụng (API)

Tích hợp

keo dán

???

???

Bị trì hoãn

Dịch vụ đặt hàng

Khối đá nguyên khối

Gửi lời xin lỗi

thông báo.

Tạo trường hợp.

Hệ thống CRM

Thông báoDịch vụ

«kho lưu trữ»Kho thông tin liên hệ khách hàng

Thông báo «thực thể»

«khuôn mẫu»Đặt hàng

«Dịch vụ» Dịch vụ giao hàng chậm trễ

Cổng API

**Hình 13.14 Thiết kế củaDịch vụ giao hàng chậm trễ. Keo tích hợp cung cấpDịch vụ giao hàng chậm trễtruy cập vào dữ liệu do khối đơn vị sở hữu, chẳng hạn nhưĐặt hàngVàNhà hàngcác thực thể và khách hàngthông tin liên lạc.**

Chúng ta hãy xem xét thiết kế của keo tích hợp cung cấpDịch vụ đặt hàng chậm trễ

truy cập vào dữ liệu của khối đơn.

###### Thiết kế keo tích hợp cho Dịch vụ giao hàng chậm trễ

Mặc dù một dịch vụ triển khai một tính năng mới xác định các lớp thực thể riêng của nó, nhưng nóthường truy cập dữ liệu do khối đơn sở hữu. Delayed Delivery Service cũng không ngoại lệ. Nó có một thực thể DelayedOrderNotification, biểu thị thông báo mà nó đã gửi đến người tiêu dùng. Nhưng như tôi vừa đề cập, các thực thể Order và Restaurant của nó sao chép dữ liệu từ khối đơn FTGO. Nó cũng cần truy vấn thông tin liên hệ của người dùng để thông báo cho người dùng. Do đó, chúng ta cần triển khai keo tích hợp cho phép Delivery Service truy cập dữ liệu của khối đơn.

Hình 13.15 cho thấy thiết kế của keo tích hợp. FTGO monolith pub-các sự kiện miền Order và Restaurant. Dịch vụ giao hàng sử dụng các sự kiện nàyvà cập nhật các bản sao của các thực thể đó. Khối FTGO triển khai REST

điểm cuối để truy vấn thông tin liên lạc của khách hàng. Dịch vụ giao hàng gọi đây làđiểm cuối khi cần thông báo cho người dùng rằng đơn hàng của họ không thể được giao đúng hạn.

Đặt hàng sự kiện

Đặt hàng sự kiện



Nhà hàng

sự kiện

Khối đá nguyên khối

Sự kiện nhà hàng

Dịch vụ đặt hàng chậm trễ

lấyThông tin liên hệ khách hàng()

<Kho lưu trữ>Kho thông tin liên hệ khách hàng

Proxy thông tin liên hệ khách hàng

NGHỈ NGƠI

điểm cuối

Người đăng ký sự kiện

Nhà xuất bản sự kiện miền

**Hình 13.15 Keo tích hợp cung cấpDịch vụ giao hàng chậm trễvới quyền truy cập vào dữ liệu dokhối đá nguyên khối.**

Hãy cùng xem xét thiết kế của từng phần tích hợp, bắt đầu với REST API để lấy thông tin liên hệ của khách hàng.

**HỏiUERYING THÔNG TIN LIÊN HỆ KHÁCH HÀNG SỬ DỤNGCNGƯỜI SỬ DỤNGCLIÊN HỆTÔITổ chức phi lợi nhuậnRBẢNG KÝ**

Như đã mô tả trong phần 13.3.1, có một vài cách khác nhau mà một dịch vụ như vậyvì Delayed Delivery Service có thể đọc dữ liệu của khối đơn. Tùy chọn đơn giản nhất là Delayed Order Service truy xuất dữ liệu bằng API truy vấn của khối đơn. Cách tiếp cận này có ý nghĩa khi truy xuất thông tin liên hệ của Người dùng. Không có bất kỳ độ trễ hoặc vấn đề hiệu suất nào vì Delayed Delivery Service hiếm khi cần truy xuất thông tin liên hệ của người dùng và lượng dữ liệu khá nhỏ.

Kho thông tin liên hệ khách hànglà một giao diện cho phépGiao hàng chậm trễDịch vụđể lấy thông tin liên lạc của người tiêu dùng. Nó được thực hiện bởi mộtKhách hàng- ContactInfoProxy, lấy thông tin người dùng bằng cách gọi khối đơn nguyênlấyThông tin liên hệ khách hàng()Điểm cuối REST.

**PSỬ DỤNG VÀ TIÊU DÙNGỒRDER VÀRSỰ KIỆN MIỀN ESTAURANT**

Thật không may, Delayed Delivery Service không thực tế khi truy vấn monolith để biết trạng thái của tất cả các Đơn hàng và giờ mở cửa của Nhà hàng. Đó là vì việc truyền liên tục một lượng lớn dữ liệu qua mạng là không hiệu quả. Do đó, Delayed Delivery Service phải sử dụng tùy chọn thứ hai phức tạp hơn và duy trì bản sao của Đơn hàng và Nhà hàng bằng cách đăng ký các sự kiện do monolith công bố. Điều quan trọng cần nhớ là bản sao không phải là bản sao hoàn chỉnh của dữ liệu từ monolith—nó chỉ lưu trữ một tập hợp con nhỏ các thuộc tính của các thực thể Đơn hàng và Nhà hàng.

Như đã mô tả trước đó trong phần 13.3.1, có một vài cách khác nhau mà chúng ta có thể thay đổi khối FTGO để nó xuất bản các sự kiện miền Order và Restaurant. Một tùy chọn là sửa đổi tất cả các vị trí trong khối cập nhật Orders và Restaurants để xuất bản các sự kiện miền cấp cao. Tùy chọn thứ hai là theo dõi nhật ký giao dịch để sao chép các thay đổi dưới dạng sự kiện. Trong trường hợp cụ thể này, chúng ta cần đồng bộ hóa hai cơ sở dữ liệu. Chúng ta không yêu cầu khối FTGO xuất bản các sự kiện miền cấp cao, vì vậy cả hai cách tiếp cận đều ổn.

Delayed Order Service triển khai trình xử lý sự kiện đăng ký sự kiện từ khối đơn và cập nhật các thực thể Order và Restaurant của nó. Chi tiết của trình xử lý sự kiện phụ thuộc vào việc khối đơn có công bố các sự kiện cấp cao cụ thể hay các sự kiện thay đổi cấp thấp. Trong cả hai trường hợp, bạn có thể nghĩ về trình xử lý sự kiện như là dịch một sự kiện trong ngữ cảnh giới hạn của khối đơn sang bản cập nhật của một thực thể trong ngữ cảnh giới hạn của dịch vụ.

Một lợi ích quan trọng của việc sử dụng bản sao là nó cho phép Delayed Order Service truy vấn hiệu quả các đơn hàng và giờ mở cửa của nhà hàng. Tuy nhiên, một nhược điểm là nó phức tạp hơn. Một nhược điểm khác là nó yêu cầu mono-lith để xuất bản các sự kiện Order và Restaurant cần thiết. May mắn thay, vì Delayed Delivery Service chỉ cần những gì về cơ bản là một tập hợp con của các cột trong bảng ORDERS và RESTAURANT, nên chúng ta sẽ không gặp phải các vấn đề được mô tả trong phần 13.3.1.

Việc triển khai một tính năng mới như quản lý đơn hàng bị trì hoãn dưới dạng dịch vụ độc lập sẽ đẩy nhanh quá trình phát triển, thử nghiệm và triển khai của tính năng đó. Hơn nữa, nó cho phép bạn triển khai tính năng bằng cách sử dụng một ngăn xếp công nghệ hoàn toàn mới thay vì ngăn xếp cũ của khối đơn. Nó cũng ngăn khối đơn phát triển. Quản lý đơn hàng bị trì hoãn chỉ là một trong nhiều tính năng mới được lên kế hoạch cho ứng dụng FTGO. Nhóm FTGO có thể triển khai nhiều tính năng này dưới dạng các dịch vụ riêng biệt.

Thật không may, bạn không thể triển khai tất cả các thay đổi dưới dạng dịch vụ mới. Thông thường, bạn phải thực hiện các thay đổi lớn đối với khối đơn để triển khai các tính năng mới hoặc thay đổi các tính năng hiện có. Bất kỳ sự phát triển nào liên quan đến khối đơn có thể sẽ chậm và khó khăn. Nếu bạn muốn đẩy nhanh việc cung cấp các tính năng này, bạn phải chia nhỏ khối đơn bằng cách di chuyển chức năng từ khối đơn vào các dịch vụ. Hãy cùng xem cách thực hiện điều đó.

#### Phá vỡ khối đá nguyên khối: trích xuất giao hàngsự quản lý

Để đẩy nhanh việc cung cấp các tính năng được triển khai bởi khối đơn, bạn cần chia khối đơn thành các dịch vụ. Ví dụ, hãy tưởng tượng rằng bạn muốn nâng cao quản lý phân phối FTGO bằng cách triển khai thuật toán định tuyến mới. Một trở ngại lớn đối với việc phát triển quản lý phân phối là nó bị vướng vào quản lý đơn hàng và là một phần của cơ sở mã khối đơn. Việc phát triển, thử nghiệm và triển khai quản lý phân phối có thể chậm. Để đẩy nhanh quá trình phát triển,bạn cần trích xuất quản lý giao hàng vào Dịch vụ giao hàng.

Tôi bắt đầu phần này bằng cách mô tả quản lý giao hàng và cách nó hiện được nhúng trong khối đơn. Tiếp theo, tôi thảo luận về thiết kế của Dịch vụ giao hàng độc lập mới và API của nó. Sau đó, tôi mô tả cách Dịch vụ giao hàng và khối đơn FTGO hợp tác với nhau. Cuối cùng, tôi nói về một số thay đổi mà chúng ta cần thực hiện đối vớikhối đá nguyên khốiđể hỗ trợ Dịch vụ giao hàng.

Chúng ta hãy bắt đầu bằng việc xem xét thiết kế hiện tại.

###### Tổng quan về chức năng quản lý giao hàng hiện có

Quản lý giao hàng chịu trách nhiệm lập lịch cho các nhân viên giao hàng nhận đơn hàng tại nhà hàng và giao hàng cho khách hàng. Mỗi nhân viên giao hàng đều có một kế hoạch là lịch trình các hành động nhận hàng và giao hàng. Hành động nhận hàng yêu cầu Nhân viên giao hàng nhận đơn hàng từ nhà hàng vào thời điểm cụ thể. Hành động giao hàng yêu cầu Nhân viên giao hàng giao hàng cho khách hàng. Các kế hoạch được sửa đổi bất cứ khi nào đơn hàng được đặt, hủy hoặc sửa đổi và khi địa điểm và tình trạng sẵn có của nhân viên giao hàng thay đổi.

Quản lý giao hàng là một trong những phần lâu đời nhất của ứng dụng FTGO. Như hình 13.16 cho thấy, nó được nhúng trong quản lý đơn hàng. Phần lớn mã để quản lý giao hàng nằm trong OrderService. Hơn nữa, không có biểu diễn rõ ràng nào vềGiao hàng. Nó được nhúng trong thực thể Đơn hàng, có nhiều trường liên quan đến giao hàng, chẳng hạn như schedulePickupTime và scheduleDeliveryTime.

Nhiều lệnh được triển khai bởi khối đơn lẻ sẽ gọi đến chức năng quản lý phân phối, bao gồm:

* + - * chấp nhận đơn hàng()—Được gọi khi nhà hàng chấp nhận đơn hàng và cam kết chuẩn bị vào thời điểm nhất định. Hoạt động này gọi quản lý giao hàng để lên lịch giao hàng.
      * hủy đơn hàng()—Được gọi khi người tiêu dùng hủy đơn hàng. Nếu cần thiết, nó sẽ hủy giao hàng.
      * noteCourierLocationUpdated()—Được kích hoạt bởi ứng dụng di động của người chuyển phát để cập nhật vị trí của người chuyển phát. Nó kích hoạt việc lên lịch lại việc giao hàng.
      * lưu ýCourierAvailabilityChanged()—Được kích hoạt bởi ứng dụng di động của người chuyển phát để cập nhật tình trạng sẵn sàng của người chuyển phát. Nó kích hoạt việc lên lịch lại việc giao hàng.

Ngoài ra, nhiều truy vấn khác nhau cũng truy xuất dữ liệu được quản lý phân phối lưu giữ, bao gồm:

* + - * lấyKế hoạchCourier()—Được gọi bởi ứng dụng di động của người chuyển phát và trả về kế hoạch của người chuyển phát
      * lấyTrạng thái đơn hàng()—Trả về trạng thái của đơn hàng, bao gồm thông tin liên quan đến việc giao hàng như đơn vị chuyển phát được chỉ định và ETA
      * lấyLịch sử đơn hàng()—Trả về thông tin tương tự nhưlấyTrạng thái đơn hàng()ngoại trừvề nhiều đơn hàng

Thông thường, những gì được trích xuất vào một dịch vụ là, như đã đề cập trong phần 13.2.3, toàn bộ một lát cắt theo chiều dọc, với các bộ điều khiển ở trên cùng và các bảng cơ sở dữ liệu ở dưới cùng. Chúng ta có thể

Đặt hànghoạt động: Chuyển phát nhanh hoạt động:



acceptOrder() hủy đơn hàng() lấy OrderStatus() lấy OrderHistory()

cập nhậtCourierLocation() cập nhậtCourierAvailability() lấyCourierPlan()

«đối tượng giá trị»Kế hoạch

«thực thể»Người chuyển phát nhanh

«thực thể»Đặt hàng

Khối FTGO

«Dịch vụ» CourierService

Giao diện lập trình ứng dụng (API)

|  |  |
| --- | --- |
| «Dịch vụ» OrderService | |
| ...  «quản lý giao hàng»lịch trìnhDelivery() lịch trình lạiDelivery() hủyDelivery() sửa đổiLịch trình()  ... | |
|  |  |

|  |  |
| --- | --- |
| «đối tượng giá trị»Hoạt động | |
|  |  |
| «đối tượng giá trị»thả xuống | |

**Hình 13.16 Quản lý giao hàng gắn liền với quản lý đơn hàng trong khối FTGO.**

«đối tượng giá trị»Nhặt lên

Nhà hàng «thực thể»

coi các lệnh và truy vấn liên quan đến Courier là một phần của quản lý giao hàng. Xét cho cùng, quản lý giao hàng tạo ra các kế hoạch chuyển phát nhanh và là người tiêu dùng chính của thông tin về vị trí và tính khả dụng của Courier. Nhưng để giảm thiểu nỗ lực phát triển, chúng tôi sẽ để các hoạt động đó trong khối đơn và chỉ trích xuấtcốt lõi của thuật toán. Do đó, lần lặp đầu tiên của Delivery Service sẽ không hiển thị API có thể truy cập công khai. Thay vào đó, nó sẽ chỉ được khối đơn khối gọi. Tiếp theo, chúng ta hãy khám phá thiết kế của Delivery Service.

###### Tổng quan về dịch vụ giao hàng

Dịch vụ giao hàng mới được đề xuất có trách nhiệm lập lịch, lập lại lịch,và hủy giao hàng. Hình 13.17 cho thấy góc nhìn cấp cao về kiến ​​trúc của ứng dụng FTGO sau khi trích xuất Dịch vụ giao hàng. Kiến trúc bao gồm khối FTGO và Dịch vụ giao hàng. Chúng cộng tác bằng cách sử dụng tích hợpkeo, bao gồm các API trong cả dịch vụ và khối đơn. Dịch vụ giao hàng có mô hình miền và cơ sở dữ liệu riêng.

**Dịch vụ giao hàng cung cấp API nào cho khối đơn?**

Khối FTGO Dịch vụ giao hàng

Mô hình miền đơn khối

Vận chuyểnMô hình miền dịch vụ

Keo tích hợp

Bộ chuyển đổi

Bộ chuyển đổi

Cơ sở dữ liệu Monolith

Cơ sở dữ liệu dịch vụ giao hàng

**Hành vi và dữ liệu nào được chuyển đến Dịch vụ giao hàng?**

**Monolith cung cấp API nào cho Dịch vụ phân phối?**

**Hình 13.17Góc nhìn cấp cao của ứng dụng FTGO sau khi trích xuấtDịch vụ giao hàng. Khối FTGO vàDịch vụ giao hàngcộng tác bằng cách sử dụng keo tích hợp, bao gồm các API trong mỗi loại. Hai quyết định chính cần đưa ra là chức năng và dữ liệu nào được chuyển đếnDịch vụ giao hàngvà làm thế nào để khối đá nguyên khối vàDịch vụ giao hàngcộng tác thông qua API?**

Để hoàn thiện kiến ​​trúc này và xác định mô hình miền của dịch vụ, chúng ta cần trả lời những câu hỏi sau:

* Hành vi và dữ liệu nào được chuyển đếnDịch vụ giao hàng?
* API làm gìDịch vụ giao hàngphơi bày trước khối đá nguyên khối?
* API nào mà khối đơn sắc tiếp xúc vớiDịch vụ giao hàng?

Những vấn đề này có liên quan với nhau vì sự phân bổ trách nhiệm giữamonolith và dịch vụ ảnh hưởng đến API. Ví dụ, Delivery Service sẽ cần phải gọi API do monolith cung cấp để truy cập dữ liệu trong cơ sở dữ liệu của monolith và ngược lại. Sau đó, tôi sẽ mô tả thiết kế của keo tích hợp cho phép

Dịch vụ giao hàngvà khối FTGO để cộng tác. Nhưng trước tiên, hãy xem xét thiết kế củaDịch vụ giao hàng'Smô hình miền.

###### Thiết kế mô hình miền dịch vụ giao hàng

Để có thể trích xuất quản lý phân phối, trước tiên chúng ta cần xác định các lớp thực hiện nó. Sau khi thực hiện xong, chúng ta có thể quyết định lớp nào sẽ chuyển đến DeliveryDịch vụ để hình thành logic miền của nó. Trong một số trường hợp, chúng ta sẽ cần phải chia tách các lớp. Chúng ta cũng cần phải quyết định dữ liệu nào sẽ được sao chép giữa dịch vụ và khối đơn.

Chúng ta hãy bắt đầu bằng cách xác định các lớp thực hiện quản lý phân phối.

**TÔIXÁC ĐỊNH CÁC THỰC THỂ VÀ LĨNH VỰC CỦA HỌ LÀ MỘT PHẦN CỦA QUẢN LÝ GIAO HÀNG**

Bước đầu tiên trong quá trình thiết kế Dịch vụ giao hàng là xem xét cẩn thận mã quản lý giao hàng và xác định các thực thể tham gia và các trường của chúng. Hình 13.18 cho thấy các thực thể và trường là một phần của quản lý giao hàng. Một số trường là đầu vào cho thuật toán lập lịch giao hàng, và một số khác là đầu ra. Hình cho thấy trường nào trong số các trường đó cũng được sử dụng bởi chức năng khác do khối đơn khối triển khai.

|  |  |  |
| --- | --- | --- |
| Đặt hàng | |  |
| «Đọc/Ghi Monolith»  «Dịch vụ Đọc»tiểu bang giao hàngĐịa chỉ  đã hứaThời gian giao hàngchuẩn bị theo thời gian  «Dịch vụ Đọc/Ghi»  «Monolith Read» đã lên lịchThời gian nhận hàng đã lên lịchThời gian giao hàng | |
|  |
|  |  | |

|  |  |
| --- | --- |
| Người chuyển phát nhanh | |
| «Đọc/Ghi Monolith»  «Dịch vụ Đọc»Vị trí khả dụng | |
|  |  |

«Dịch vụ Đọc/Ghi»

«Đọc Monolith»

Nhiệm vụ

Kế hoạch

«Đọc» địa chỉ

Nhà hàng

**Hình 13.18 Các thực thể và trường được truy cập bởi quản lý phân phối và các chức năng khác được triển khai bởi khối đơn. Một trường có thể được đọc hoặc ghi hoặc cả hai. Nó có thể được truy cập bởi quản lý phân phối, khối đơn hoặc cả hai.**

Thuật toán lập lịch giao hàng đọc các thuộc tính khác nhau bao gồmĐặt hàng'Snhà hàng,đã hứaThời gian giao hàng, VàĐịa chỉ giao hàngvàNgười chuyển phát nhanh'Svị trí,sự sẵn cóvà các kế hoạch hiện tại. Nó cập nhậtNgười chuyển phát nhanhkế hoạch của 's,Đặt hàng'Sđã lên lịch-Thời gian đón, Vàthời gian giao hàng theo lịch trình. Như bạn có thể thấy, các trường được quản lý phân phối sử dụng cũng được khối đơn khối sử dụng.

**DQUYẾT ĐỊNH DI CHUYỂN ĐẾN DỮ LIỆU NÀODGIAO HÀNGSDỊCH VỤ**

Bây giờ chúng ta đã xác định được những thực thể và trường nào tham gia vào quản lý phân phối, bước tiếp theo là quyết định xem chúng ta nên chuyển những thực thể và trường nào sang dịch vụ. Trong một kịch bản lý tưởng, dữ liệu mà dịch vụ truy cập được sử dụng độc quyền bởi dịch vụ, vì vậy chúng ta có thể chỉ cần chuyển dữ liệu đó sang dịch vụ và xong. Thật đáng buồn, hiếm khi đơn giản như vậy và tình huống này cũng không ngoại lệ. Tất cả các thực thể và trường được sử dụng bởi quản lý phân phối cũng được sử dụng bởi các chức năng khác được triển khai bởi khối đơn.

Do đó, khi xác định dữ liệu nào sẽ chuyển đến dịch vụ, chúng ta cần lưu ý hai vấn đề. Vấn đề đầu tiên là: dịch vụ truy cập dữ liệu còn lại trong khối đơn như thế nào? Vấn đề thứ hai là: khối đơn truy cập dữ liệu đã chuyển đến dịch vụ như thế nào? Ngoài ra, như đã mô tả trước đó trong phần 13.3, chúng ta cần cân nhắc cẩn thận cách duy trì tính nhất quán của dữ liệu giữa dịch vụ và khối đơn.

Trách nhiệm thiết yếu củaDịch vụ giao hàngđang quản lý các kế hoạch chuyển phát nhanh và cập nhậtĐặt hàng'Sthời gian đón khách theo lịch trìnhVàthời gian giao hàng theo lịch trìnhcác trường. Do đó, có lý khi nó sở hữu các trường đó. Chúng ta cũng có thể di chuyểnCourier.location (Địa điểm chuyển phát nhanh)VàChuyển phát nhanh.tính khả dụngcác lĩnh vực đểDịch vụ giao hàng. Nhưng vì chúng tôi đang cố gắng thực hiện thay đổi nhỏ nhất có thể nên chúng tôi sẽ giữ nguyên các trường đó trong khối nguyên khối ngay bây giờ.

**TTHIẾT KẾ CỦADGIAO HÀNGSLOGIC MIỀN DỊCH VỤ**

Hình 13.19 cho thấy thiết kế củaDịch vụ giao hàngMô hình miền của. Cốt lõi của dịch vụ bao gồm các lớp miền nhưVận chuyểnVàNgười chuyển phát nhanh. CácGiao hàng- Dịch vụImpllớp học là điểm khởi đầu cho hoạt động quản lý giao hànglogic. Nóthực hiệnDịch vụ giao hàngVàDịch vụ chuyển phát nhanhgiao diện, được gọi bởiTrình xử lý sự kiện dịch vụ giao hàngVàDeliveryServiceNotificationsHandlers, được mô tả sau trong phần này.

Logic kinh doanh quản lý phân phối chủ yếu là mã được sao chép từ khối thống nhất.Ví dụ, chúng ta sẽ sao chép thực thể Order từ khối đơn sang Delivery Service, đổi tên thành Delivery và xóa tất cả các trường ngoại trừ những trường được sử dụng bởi quản lý giao hàng. Chúng ta cũng sẽ sao chép thực thể Courier và xóa hầu hết các trường của thực thể này. Để phát triển logic miền cho Delivery Service, chúng ta sẽ cần gỡ rối mã khỏi khối đơn. Chúng ta sẽ cần phá vỡ nhiều phụ thuộc, điều này có thể tốn thời gian. Một lần nữa, việc cấu trúc lại mã dễ hơn nhiều khi sử dụng ngôn ngữ được gõ tĩnh, vì trình biên dịch sẽ là bạn của bạn.

Dịch vụ giao hàngkhông phải là một dịch vụ độc lập. Hãy xem xét thiết kế của keo tích hợp cho phépVận chuyểnDịch vụvà khối FTGO để hợp tác.



Dịch vụ giao hàng

noteCourierLocationUpdated(...) noteCourierAvailabilityUpdated(...)

«giao diện» CourierService

Giao hàngDịch vụImpl

lịch trình vô hiệu(...)hủy bỏ lệnh reschedule(...) hủy bỏ lệnh cancel(...)

«giao diện» DeliveryService

Sự kiện DeliveryServiceHandlers

«đối tượng giá trị»Kế hoạch

«thực thể»Vận chuyển

«thực thể»Người chuyển phát nhanh

DeliveryService NotificationHandlers

**Hình 13.19 Thiết kế củaDịch vụ giao hàngmô hình miền của**

###### Thiết kế keo tích hợp dịch vụ giao hàng

Khối FTGO cần phải được triệu hồiDịch vụ giao hàngđể quản lý việc giao hàng. Khối đơn cũng cần trao đổi dữ liệu vớiDịch vụ giao hàng. Sự hợp tác này được kích hoạt bởi keo tích hợp. Hình 13.20 cho thấy thiết kế củaDịch vụ giao hàngphókeo dán.Dịch vụ giao hàngcó API quản lý giao hàng. Nó cũng xuất bảnVận chuyểnVàNgười chuyển phát nhanhsự kiện miền. Khối FTGO xuất bảnNgười chuyển phát nhanhsự kiện miền.

Hãy cùng xem xét thiết kế của từng phần trong keo tích hợp, bắt đầu với API của Dịch vụ giao hàng để quản lý việc giao hàng.

**TTHIẾT KẾ CỦADGIAO HÀNGSDỊCH VỤGiao diện lập trình ứng dụng (API)**

Delivery Service phải cung cấp API cho phép monolith lên lịch, sửa đổi và hủy giao hàng. Như bạn đã thấy trong suốt cuốn sách này, cách tiếp cận được ưa thích là sử dụng tin nhắn không đồng bộ, vì nó thúc đẩy sự kết hợp lỏng lẻo và tăng tính khả dụng. Một cách tiếp cận là để Delivery Service đăng ký các sự kiện miền Order do monolith xuất bản. Tùy thuộc vào loại sự kiện, nó tạo ra,

Dịch vụ giao hàngthông báo



Sự kiện chuyển phát nhanh

FTGO

khối đá nguyên khối

Sự kiện chuyển phát nhanh

Sự kiện giao hàng

Vận chuyển

Dịch vụ

Sự kiện chuyển phát nhanh

«giao diện» CourierService

«giao diện» DeliveryService

Nhắn tinbộ chuyển đổi

Nhắn tinbộ chuyển đổi

Người đăng ký sự kiện giao hàng

«giao diện» DeliveryService

Người đăng ký sự kiện chuyển phát nhanh

Vận chuyểnDịch vụ proxy

Trình xử lý thông báo dịch vụ giao hàng

Sự kiện giao hàng

**Hình 13.20 Thiết kế củaDịch vụ giao hàngkeo dán.Dịch vụ giao hàngcó API quản lý phân phối. Dịch vụ và khối FTGO đồng bộ hóa dữ liệu bằng cách trao đổi các sự kiện miền.**

sửa đổi và hủy bỏ Giao hàng. Một lợi ích của cách tiếp cận này là khối nguyên khối khôngcần phải gọi rõ ràng Dịch vụ giao hàng. Nhược điểm của việc dựa vào các sự kiện miền là nó yêu cầu Dịch vụ giao hàng phải biết cách mỗi sự kiện Đơn hàng tác động đến Giao hàng tương ứng.

Một cách tiếp cận tốt hơn làDịch vụ giao hàngđể triển khai API dựa trên thông báo cho phép khối đơn nguyên có thể thông báo rõ ràngDịch vụ giao hàngđể tạo, sửa đổi và hủy các đơn giao hàng.Dịch vụ giao hàngAPI của 'bao gồm một kênh thông báo tin nhắn và ba loại tin nhắn:Lịch trìnhGiao hàng,Sửa đổiGiao hàng, hoặcHủyGiao hàng. MỘTtin nhắn thông báo chứaĐặt hàngthông tin cần thiết bởiDịch vụ giao hàng. Ví dụ, mộtLịch trìnhGiao hàngthông báochứa thời gian và địa điểm đón và thời gian và địa điểm giao hàng. Một lợi ích quan trọng của cách tiếp cận này làVận chuyểnDịch vụkhông có kiến ​​thức chi tiết vềĐặt hàngvòng đời. Nó hoàn toàn tập trung vào việc quản lý việc giao hàng và không có kiến ​​thức về đơn hàng.

API này không phải là cách duy nhất mà Delivery Service và FTGO monolith hợp tác. Họ cũng cần trao đổi dữ liệu.

**HỒ, thế đấyDGIAO HÀNGSDỊCH VỤ TRUY CẬPFTGOĐÁ NGUYÊN TẮC'DỮ LIỆU S**

Dịch vụ giao hàngcần phải truy cập vàoNgười chuyển phát nhanhdữ liệu vị trí và tính khả dụng, đó làdo khối đơn vị sở hữu. Vì đó có khả năng là một lượng dữ liệu lớn, nên không thực tế khi dịch vụ truy vấn khối đơn vị nhiều lần. Thay vào đó, một cách tiếp cận tốt hơn là khối đơn vị sao chép dữ liệu đểDịch vụ giao hàngbằng cách xuất bảnNgười chuyển phát nhanhsự kiện miền,CourierLocationĐã cập nhậtVàChuyển phát nhanhKhả dụngĐã cập nhật.Dịch vụ giao hàngcó mộtCourierSự kiệnNgười đăng kýđăng ký các sự kiện miền và cập nhật phiên bản của nóNgười chuyển phát nhanh. Nó cũng có thể kích hoạt việc lên lịch lại việc giao hàng.

**HỒ, thế đấyFTGOMONOLITH TRUY CẬP VÀODGIAO HÀNGSDỮ LIỆU DỊCH VỤ**

Khối FTGO cần phải đọc dữ liệu đã được chuyển đến Dịch vụ giao hàng,chẳng hạn như kế hoạch của Courier. Về mặt lý thuyết, khối đơn có thể truy vấn dịch vụ, nhưng điều đó đòi hỏi phải thay đổi đáng kể khối đơn. Hiện tại, dễ hơn là giữ nguyên mô hình miền và lược đồ cơ sở dữ liệu của khối đơn và sao chép dữ liệu từ dịch vụ trở lại khối đơn.

Cách dễ nhất để thực hiện điều đó làDịch vụ giao hàngđể xuất bảnNgười chuyển phát nhanhVàVận chuyểnsự kiện miền. Dịch vụ xuất bản mộtCourierPlanĐã cập nhậtsự kiện khi nó cập nhật mộtNgười chuyển phát nhanhkế hoạch của 's, và mộtLịch trình giao hàngCập nhậtsự kiện khi nó cập nhật mộtVận chuyển. Khối đơn khối sử dụng các sự kiện miền này và cập nhật cơ sở dữ liệu của nó.

Bây giờ chúng ta đã xem xét cách khối FTGO và Dịch vụ giao hàng tương tác với nhau, hãy cùng xem cách thay đổi khối này.

###### Thay đổi khối FTGO để tương tác với Dịch vụ giao hàng

Theo nhiều cách, việc triển khai Delivery Service là phần dễ hơn của quá trình trích xuất. Việc sửa đổi khối FTGO khó hơn nhiều. May mắn thay, việc sao chép dữ liệu từ dịch vụ trở lại khối sẽ làm giảm kích thước của thay đổi. Nhưng chúng ta vẫn cần thay đổi khối để quản lý việc phân phối bằng cách gọi Delivery Service. Hãy cùng xem cách thực hiện điều đó.

**DĐỊNH NGHĨA ADGIAO HÀNGSGIAO DIỆN DỊCH VỤ**

Bước đầu tiên là đóng gói mã quản lý giao hàng bằng giao diện Java tương ứng với API dựa trên tin nhắn được xác định trước đó. Giao diện này, được hiển thị trong hình 13.21, xác định các phương pháp để lên lịch, lên lịch lại và hủy giao hàng.



Giao hàngDịch vụImpl

void schedule(...) void reschedule(...) void cancel(...)

«giao diện» DeliveryService

Vận chuyển

quản lý khách hàng

Vận chuyển

sự quản lý

**Hình 13.21 Bước đầu tiên là xác địnhDịch vụ giao hàng, cái màlà một API thô, có thể điều khiển từ xa để gọilogic quản lý giao hàng.**

Cuối cùng, chúng ta sẽ triển khai giao diện này với một proxy gửi tin nhắn đến dịch vụ giao hàng. Nhưng ban đầu, chúng ta sẽ triển khai API này với một lớp gọi mã quản lý giao hàng.

Giao diện DeliveryService là giao diện có độ chi tiết thô phù hợp để triển khai bằng cơ chế IPC. Giao diện này định nghĩa các phương thức schedule(), reschedule() và cancel() tương ứng với các loại tin nhắn thông báo đã định nghĩa trước đó.

**RPHÂN TÍCH MONOLITH ĐỂ GỌIDGIAO HÀNGSGIAO DIỆN DỊCH VỤ**

Tiếp theo, như hình 13.22 cho thấy, chúng ta cần xác định tất cả các vị trí trong khối FTGO gọi quản lý phân phối và thay đổi chúng để sử dụng giao diện DeliveryService. Điều này có thể mất một thời gian và là một trong những khía cạnh đầy thách thức nhất khi trích xuất dịch vụ từ khối.



Khách hàng quản lý giao hàng

void schedule(...) void reschedule(...) void cancel(...)

«giao diện» DeliveryService

**Hình 13.22 Bước thứ hai là thay đổi khối FTGO để gọi quản lý phân phối thông quaDịch vụ giao hànggiao diện.**

Quản lý giao hàng

Giao hàngDịch vụImpl

Chắc chắn sẽ hữu ích nếu khối đơn được viết bằng ngôn ngữ kiểu tĩnh, chẳng hạn như Java, vì các công cụ thực hiện tốt hơn việc xác định các phụ thuộc. Nếu không, hy vọng bạn có một số bài kiểm tra tự động với phạm vi bao phủ đủ các phần mã cần thay đổi.

**TÔITHỰC HIỆNDGIAO HÀNGSGIAO DIỆN DỊCH VỤ**

Bước cuối cùng là thay thế lớp DeliveryServiceImpl bằng một proxy gửi tin nhắn thông báo đến Delivery Service độc ​​lập. Nhưng thay vì loại bỏ ngay lập tức triển khai hiện có, chúng ta sẽ sử dụng một thiết kế, được hiển thị trong hình 13.23, cho phép khối đơn chuyển đổi động giữa triển khai hiện có và Delivery Service. Chúng ta sẽ triển khai giao diện DeliveryService bằng một lớp sử dụng một nút chuyển đổi tính năng động để xác định xem có nên gọi triển khai hiện có hay Delivery Service hay không.



Khách hàng quản lý giao hàng

FeatureToggleBased DeliveryServiceImpl

lịch trình vô hiệu(...)hủy bỏ lệnh reschedule(...) hủy bỏ lệnh cancel(...)

«giao diện» DeliveryService

Triệu hồi

Triệu hồi

Gửi

tin nhắn

Giao hàngDịch vụProxy

Giao hàngDịch vụImpl

Thông báo giao hàng

Quản lý giao hàng

**Hình 13.23 Bước cuối cùng là thực hiệnDịch vụ giao hàngvới một lớp proxy gửi tin nhắnDịch vụ giao hàng. Một tính năng chuyển đổi kiểm soát xem khối FTGO có sử dụng triển khai cũ hay mới hay khôngDịch vụ giao hàng.**

Sử dụng một tính năng chuyển đổi làm giảm đáng kể rủi ro khi triển khai Delivery Service. Chúng ta có thể triển khai Delivery Service và thử nghiệm nó. Sau đó, khi chúng ta chắc chắn rằng nó hoạt động, chúng ta có thể lật nút chuyển đổi để định tuyến lưu lượng truy cập đến nó. Nếu sau đó chúng ta phát hiện ra rằng Delivery Service không hoạt động như mong đợi, chúng ta có thể chuyển lại về triển khai cũ.

**Về tính năng chuyển đổi**

*Tính năng chuyển đổi*, hoặc*cờ đặc trưng*, cho phép bạn triển khai các thay đổi mã mà không nhất thiết phải phát hành chúng cho người dùng. Chúng cũng cho phép bạn thay đổi động hành vi của ứng dụng bằng cách triển khai mã mới. Bài viết này của Martin Fowler cung cấp tổng quan tuyệt vời về chủ đề này:[https://martinfowler.com/articles/feature-toggle](https://martinfowler.com/articles/feature-toggles.html)

[.html](https://martinfowler.com/articles/feature-toggles.html).

Khi chúng tôi chắc chắn rằng Dịch vụ giao hàng đang hoạt động như mong đợi, chúng tôi có thể xóa mã quản lý giao hàng khỏi khối đơn.

Dịch vụ giao hàng và dịch vụ đơn hàng chậm trễ là những ví dụ về các dịch vụ mà nhóm FTGO sẽ phát triển trong hành trình của họ đến kiến ​​trúc dịch vụ vi mô. Họ sẽ đi đâu tiếp theo sau khi triển khai các dịch vụ này phụ thuộc vào các ưu tiên củadoanh nghiệp. Một cách có thể là trích xuất Dịch vụ Lịch sử Đơn hàng, được mô tả trong chương 7. Việc trích xuất dịch vụ này sẽ loại bỏ một phần nhu cầu Dịch vụ Giao hàng phải sao chép dữ liệu trở lại khối đơn.

Sau khi triển khai Dịch vụ Lịch sử Đơn hàng, nhóm FTGO có thể trích xuất các dịch vụ theo thứ tự được mô tả trong phần 13.3.2: Dịch vụ Đơn hàng, Dịch vụ Khách hàng, Dịch vụ Nhà bếp, v.v. Khi nhóm FTGO trích xuất từng dịch vụ, khả năng bảo trì và khả năng kiểm tra ứng dụng của họ dần được cải thiện và tốc độ phát triển của họ tăng lên.

#### Bản tóm tắt

* Trước khi di chuyển sang kiến ​​trúc microservice, điều quan trọng là phải chắc chắn rằng các vấn đề về phân phối phần mềm của bạn là kết quả của việc phát triển vượt ra ngoài kiến ​​trúc đơn khối của bạn. Bạn có thể đẩy nhanh quá trình phân phối bằng cách cải thiện quy trình phát triển phần mềm của mình.
* Điều quan trọng là phải di chuyển sang các dịch vụ vi mô bằng cách phát triển dần dần một ứng dụng strangler. Ứng dụng strangler là một ứng dụng mới bao gồm các dịch vụ vi mô mà bạn xây dựng xung quanh ứng dụng đơn khối hiện có. Bạn nên chứng minh giá trị sớm và thường xuyên để đảm bảo rằng doanh nghiệp hỗ trợ nỗ lực di chuyển.
* Một cách tuyệt vời để đưa microservices vào kiến ​​trúc của bạn là triển khai các tính năng mới dưới dạng dịch vụ. Làm như vậy cho phép bạn phát triển nhanh chóng và dễ dàng một tính năng bằng công nghệ hiện đại và quy trình phát triển. Đây là một cách tốt để nhanh chóng chứng minh giá trị của việc di chuyển sang microservices.
* Một cách để chia nhỏ khối đơn là tách tầng trình bày khỏi phần phụ trợ, tạo ra hai khối đơn nhỏ hơn. Mặc dù không phải là cải tiến lớn, nhưng điều đó có nghĩa là bạn có thể triển khai từng khối đơn độc một cách độc lập. Ví dụ, điều này cho phép nhóm UI lặp lại thiết kế UI dễ dàng hơn mà không ảnh hưởng đến phần phụ trợ.
* Cách chính để phá vỡ khối đơn là bằng cách di chuyển dần chức năng từ khối đơn vào các dịch vụ. Điều quan trọng là tập trung vào việc trích xuất các dịch vụ mang lại nhiều lợi ích nhất. Ví dụ, bạn sẽ đẩy nhanh quá trình phát triển nếu trích xuất một dịch vụ triển khai chức năng đang được phát triển tích cực.
* Các dịch vụ mới phát triển gần như luôn phải tương tác với khối đơn. Một dịch vụ thường cần truy cập dữ liệu của khối đơn và gọi chức năng của nó. Đôi khi khối đơn cần truy cập dữ liệu của một dịch vụ và gọi chức năng của nó. Để triển khai sự hợp tác này, hãy phát triển keo tích hợp, bao gồm các bộ điều hợp vào và ra trong khối đơn.
* Để ngăn mô hình miền của khối đơn khối làm ô nhiễm mô hình miền của dịch vụ, keo tích hợp nên sử dụng lớp chống hỏng, đây là lớp phần mềm chuyển đổi giữa các mô hình miền.
* Một cách để giảm thiểu tác động lên khối đơn vị khi trích xuất dịch vụ là sao chép dữ liệu đã được chuyển đến dịch vụ trở lại cơ sở dữ liệu của khối đơn vị. Vì lược đồ của khối đơn vị không thay đổi nên điều này loại bỏ nhu cầu thực hiện các thay đổi có khả năng lan rộng đối với cơ sở mã khối đơn vị.

***Bản tóm tắt* 471**

* + Việc phát triển một dịch vụ thường yêu cầu bạn phải triển khai các saga liên quan đến khối đơn. Nhưng có thể rất khó để triển khai một giao dịch có thể bù trừ đòi hỏi phải thực hiện các thay đổi rộng rãi đối với khối đơn. Do đó, đôi khi bạn cần phải sắp xếp cẩn thận việc trích xuất các dịch vụ để tránh triển khai các giao dịch có thể bù trừ trong khối đơn.
  + Khi tái cấu trúc thành kiến ​​trúc vi dịch vụ, bạn cần đồng thời hỗ trợ cơ chế bảo mật hiện có của ứng dụng đơn khối, thường dựa trên phiên trong bộ nhớ, và cơ chế bảo mật dựa trên mã thông báo được các dịch vụ sử dụng. May mắn thay, một giải pháp đơn giản là sửa đổi trình xử lý đăng nhập của đơn khối để tạo cookie chứa mã thông báo bảo mật, sau đó được chuyển tiếp đến các dịch vụ bằng cổng API.

# chỉ số

Số học

2PC (cam kết hai giai đoạn)[112](#_bookmark488)

Mẫu đăng ký của bên thứ 3[84](#_bookmark366)–[85](#_bookmark368),[108](#_bookmark469)

Mô hình xem 4+1 của kiến ​​trúc phần mềm[35](#_bookmark191)–[37](#_bookmark196) Mã trạng thái 500, HTTP[367](#_bookmark1250)

MỘT

AbstractAutowiringHttpRequestHandler lớp [423](#_bookmark1494)

Lớp AbstractHttpHandler [423](#_bookmark1496)

phương thức accept()[165](#_bookmark639),[172](#_bookmark657)

kiểm tra chấp nhận[335](#_bookmark1149)–[338](#_bookmark1156)

xác định[336](#_bookmark1152)

thực hiện các thông số kỹ thuật bằng cách sử dụng Cucumber[338](#_bookmark1155) viết bằng Gherkin[337](#_bookmark1154)–[338](#_bookmark1156)

phương thức acceptOrder() [460](#_bookmark1595)

Mã thông báo truy cập[28](#_bookmark155),[354](#_bookmark1207),[357](#_bookmark1218)

ACD (Độ nguyên tử, Độ đồng nhất, Độ bền)[111](#_bookmark477) ACID (Độ nguyên tử, Độ đồng nhất, Độ cô lập, Độ bền)

khả năng) giao dịch[98](#_bookmark438),[110](#_bookmark476) ACL (danh sách kiểm soát truy cập)[350](#_bookmark1186) Môi giới tin nhắn ActiveMQ[92](#_bookmark414) phương thức add()[310](#_bookmark1102)

phương thức addOrder()[249](#_bookmark909)–[250](#_bookmark910)

Lớp AggregateRepository[206](#_bookmark769)–[208](#_bookmark777)

tổng hợp[147](#_bookmark573),[374](#_bookmark1286),[439](#_bookmark1538)

ranh giới nhất quán[155](#_bookmark608)

tạo, tìm kiếm và cập nhật[207](#_bookmark774)–[208](#_bookmark777) định nghĩa lệnh tổng hợp [207](#_bookmark772) định nghĩa với ReflectiveMutableCommand-

Lớp ProcessingAggregate[206](#_bookmark770)–[207](#_bookmark771) thiết kế logic kinh doanh với[159](#_bookmark617)–[160](#_bookmark618) sự kiện tìm nguồn

lịch sử tổng hợp[186](#_bookmark687),[199](#_bookmark734)–[200](#_bookmark735) phương pháp tổng hợp và sự kiện[189](#_bookmark700)–[191](#_bookmark703)

dựa trên sự kiệnTổng hợp đơn hàng[191](#_bookmark702)–[193](#_bookmark704)

duy trì tổng hợp bằng cách sử dụng các sự kiện[186](#_bookmark693)–[188](#_bookmark698) sự kiện nguồn và lịch sử tổng hợp[199](#_bookmark734)–[200](#_bookmark735) ranh giới rõ ràng[154](#_bookmark607)–[155](#_bookmark610)

độ chi tiết [158](#_bookmark615)

xác định[155](#_bookmark609)

Tổng hợp đơn hàng[175](#_bookmark664)–[180](#_bookmark675)

phương pháp[177](#_bookmark670)–[180](#_bookmark675)

máy trạng thái[176](#_bookmark667)–[177](#_bookmark671)

cấu trúc của[175](#_bookmark665)–[176](#_bookmark666)

quy tắc cho[155](#_bookmark612)–[157](#_bookmark613)

Tổng hợp vé[169](#_bookmark651)–[173](#_bookmark660)

hành vi của[170](#_bookmark653)–[171](#_bookmark654)

Dịch vụ tên miền KitchenService[171](#_bookmark656)–[172](#_bookmark659) Lớp KitchenServiceCommandHandler

[172](#_bookmark658)–[173](#_bookmark660)

cấu trúc của lớp Ticket[170](#_bookmark652) sự bền bỉ truyền thống và tổng hợp

lịch sử[186](#_bookmark687)

bí danh[285](#_bookmark1027)

Mẫu thay thế [22](#_bookmark106)

AMI (Hình ảnh máy Amazon)[390](#_bookmark1351) sự bất thường[126](#_bookmark529)

Mẫu lớp chống tham nhũng[447](#_bookmark1552)

AOP (lập trình hướng khía cạnh)[373](#_bookmark1277),[378](#_bookmark1307)

Máng Apache[370](#_bookmark1270)

Apache Kafka[92](#_bookmark416)

Apache Openwhisk[416](#_bookmark1459)

Apache Shiro[351](#_bookmark1193)

Mẫu thành phần API[221](#_bookmark811)–[228](#_bookmark836) lợi ích và hạn chế của[227](#_bookmark831)–[228](#_bookmark836)

tăng chi phí chung[227](#_bookmark832)

thiếu sự nhất quán của dữ liệu giao dịch[228](#_bookmark835)

nguy cơ giảm khả năng cung cấp[227](#_bookmark833)–[228](#_bookmark834)

**473**

Mẫu thành phần API*(tiếp theo)*

vấn đề thiết kế[225](#_bookmark824)–[227](#_bookmark828)

mô hình lập trình phản ứng[227](#_bookmark827) vai trò của nhà soạn thảo API[225](#_bookmark825)–[227](#_bookmark826)

hoạt động truy vấn findOrder()[221](#_bookmark814)–[222](#_bookmark817),[224](#_bookmark822)

tổng quan về[222](#_bookmark816)–[224](#_bookmark820)

Cổng API[259](#_bookmark932)–[291](#_bookmark1041)

xác thực[354](#_bookmark1206)–[355](#_bookmark1208)

lợi ích của[267](#_bookmark966)

vấn đề thiết kế[268](#_bookmark973)–[271](#_bookmark984)

là công dân tốt trong kiến ​​trúc[270](#_bookmark983)–[271](#_bookmark984) xử lý lỗi một phần[270](#_bookmark982)

hiệu suất và khả năng mở rộng[268](#_bookmark974)–[269](#_bookmark978) trừu tượng lập trình phản ứng[269](#_bookmark979)–[270](#_bookmark981)

nhược điểm của[267](#_bookmark967)

thực hiệnsử dụng GraphQL[279](#_bookmark1017)–[291](#_bookmark1041) kết nối lược đồ với dữ liệu[285](#_bookmark1029)–[287](#_bookmark1031) định nghĩa lược đồ[282](#_bookmark1023)–[284](#_bookmark1025)

thực hiện truy vấn[284](#_bookmark1026)–[285](#_bookmark1028)

tích hợp máy chủ Apollo GraphQL với Express[289](#_bookmark1036)–[290](#_bookmark1039)

tối ưu hóa việc tải bằng cách sử dụng hàng loạt và bộ nhớ đệm[288](#_bookmark1032)

khách hàng viết[290](#_bookmark1038)–[291](#_bookmark1041) thực hiệnsử dụng Netflix Zuul[273](#_bookmark1001)

thực hiện bằng cách sử dụng các sản phẩm/dịch vụ có sẵn[271](#_bookmark990)–[272](#_bookmark999)

Sản phẩm cổng API[272](#_bookmark996)

Dịch vụ cổng API AWS[271](#_bookmark991)–[272](#_bookmark993)

Dịch vụ cân bằng tải ứng dụng AWS[272](#_bookmark995) triển khai sử dụng Spring Cloud

Cổng vào[273](#_bookmark1002)–[275](#_bookmark1006)

Ứng dụng ApiGatewaylớp học[279](#_bookmark1014)

Lớp OrderConfiguration [275](#_bookmark1005)–[276](#_bookmark1008)

Lớp OrderHandlers [276](#_bookmark1009)–[278](#_bookmark1011)

Lớp OrderService[278](#_bookmark1012)–[279](#_bookmark1013)

ánh xạ cookie USERINFO tới tiêu đề Authorization[455](#_bookmark1576)

Ví dụ về Netflix[267](#_bookmark969)–[268](#_bookmark971)

tổng quan về[259](#_bookmark935)–[266](#_bookmark963)

Thành phần API [261](#_bookmark937)

ngành kiến ​​​​trúc[263](#_bookmark951)–[264](#_bookmark955)

Backend cho mẫu frontend[264](#_bookmark960)–[266](#_bookmark963) API dành riêng cho khách hàng[262](#_bookmark939)

hàm cạnh[262](#_bookmark943)–[263](#_bookmark949)

mô hình sở hữu[264](#_bookmark956)

giao thức dịch[262](#_bookmark938)

yêu cầu định tuyến[260](#_bookmark936)

Ứng dụng ApiGatewaylớp học[279](#_bookmark1014)

Gói ApiGatewayMain[274](#_bookmark1003)

Sự kiện APIGatewayProxyRequest[417](#_bookmark1464),[421](#_bookmark1487)–[422](#_bookmark1489)

Sự kiện phản hồi Proxy của APIGateway[417](#_bookmark1465),[422](#_bookmark1490) API

định nghĩa trong microservicengành kiến ​​​​trúc[68](#_bookmark299)–[69](#_bookmark303) giao tiếp giữa các tiến trình[69](#_bookmark305)–[71](#_bookmark312)

tạo thông số kỹ thuật cho API dịch vụ dựa trên tin nhắn[89](#_bookmark402)–[90](#_bookmark407)

lớn, phá vỡthay đổi[70](#_bookmark311)–[71](#_bookmark312)

những thay đổi nhỏ, tương thích ngược[70](#_bookmark310) phiên bản ngữ nghĩa[70](#_bookmark306)

chỉ định REST API[74](#_bookmark330)

tái cấu trúc thành các dịch vụ vi mô[444](#_bookmark1545)–[445](#_bookmark1546),[465](#_bookmark1609)–[466](#_bookmark1610) kiểm tra các dịch vụ vi mô

kiểm tra hợp đồng người tiêu dùng cho API nhắn tin[305](#_bookmark1077)

kiểm tra tích hợp phía người tiêu dùng cho OrderServiceProxy của cổng API[325](#_bookmark1131)–[326](#_bookmark1133)

ví dụ hợp đồng cho REST API[324](#_bookmark1129)

[*Nhìn thấy*](#_bookmark1576)[*Mà còn*](#_bookmark932)Cổng API Các mẫu kiến ​​trúc ứng dụng

Kiến trúc vi dịch vụ[8](#_bookmark44)–[18](#_bookmark86),[40](#_bookmark220)

Kiến trúc nguyên khối[2](#_bookmark7)–[7](#_bookmark38),[22](#_bookmark108)–[34](#_bookmark181),[40](#_bookmark220)

cơ sở hạ tầng ứng dụng[24](#_bookmark117)

số liệu ứng dụng[28](#_bookmark141),[366](#_bookmark1245),[373](#_bookmark1284)–[376](#_bookmark1294) thu thập số liệu về mức độ dịch vụ[374](#_bookmark1287)–[375](#_bookmark1289)

cung cấp số liệu cho dịch vụ số liệu [375](#_bookmark1288)–[376](#_bookmark1294) hiện đại hóa ứng dụng[23](#_bookmark115)–[24](#_bookmark118),[430](#_bookmark1513)–[432](#_bookmark1517)

bảo mật ứng dụng[349](#_bookmark1181)

phương thức apply()[188](#_bookmark699),[193](#_bookmark705)

phong cách kiến ​​trúc[37](#_bookmark203)–[40](#_bookmark217)

lục giác[38](#_bookmark212)–[40](#_bookmark217)

nhiều lớp[37](#_bookmark204)–[38](#_bookmark211)

Kiến trúc dịch vụ vi mô[40](#_bookmark219)–[43](#_bookmark231) khớp nối lỏng lẻo,được định nghĩa[42](#_bookmark226)–[43](#_bookmark227)

sự không quan trọng tương đối của quy mô dịch vụ[43](#_bookmark229) vai trò của thư viện chia sẻ[43](#_bookmark228)

dịch vụ, được xác định[41](#_bookmark222)–[42](#_bookmark225)

lập trình hướng khía cạnh (AOP)[373](#_bookmark1278),[378](#_bookmark1307) mô hình I/O không đồng bộ (không chặn)[268](#_bookmark976) tương tác không đồng bộ[67](#_bookmark296)

Mẫu tin nhắn không đồng bộ[85](#_bookmark373)–[103](#_bookmark457) người nhận cạnh tranh và thứ tự tin nhắn

[94](#_bookmark425)–[95](#_bookmark428)

tạo đặc tả API[89](#_bookmark402)–[90](#_bookmark407) ghi lại các hoạt động không đồng bộ[90](#_bookmark403) ghi lại các sự kiện đã xuất bản[90](#_bookmark406)

tin nhắn trùng lặp[95](#_bookmark430)–[97](#_bookmark433)

theo dõi tin nhắn và loại bỏ các tin nhắn trùng lặp[96](#_bookmark432)–[97](#_bookmark433)

viết trình xử lý tin nhắn idempotent[96](#_bookmark431) cải thiện tính khả dụng[103](#_bookmark459)–[108](#_bookmark470)

loại bỏ tương tác đồng bộ[104](#_bookmark464)–[108](#_bookmark470)

giao tiếp đồng bộ và khả dụng[103](#_bookmark461)–[104](#_bookmark462)

phong cách tương tác [87](#_bookmark389)–[89](#_bookmark400)

thông báo một chiều[89](#_bookmark395)

xuất bản/đăng ký[89](#_bookmark396)

yêu cầu/phản hồi và yêu cầu không đồng bộ/phản ứng[87](#_bookmark390)–[88](#_bookmark394)

Không đồng bộmẫu tin nhắn*(tiếp theo)*

thư viện và khuôn khổ cho[100](#_bookmark452)–[103](#_bookmark457) nhắn tin cơ bản[101](#_bookmark454)

nhắn tin dựa trên lệnh/trả lời[102](#_bookmark456)–[103](#_bookmark457) sự kiện miền xuất bản[102](#_bookmark455)

môi giới tin nhắn[90](#_bookmark409)–[94](#_bookmark423)

lợi ích và hạn chế của[93](#_bookmark418)–[94](#_bookmark423) nhắn tin không qua môi giới[91](#_bookmark410)–[92](#_bookmark412) thực hiệnkênh tin nhắn sử dụng[93](#_bookmark417) tổng quan về[92](#_bookmark413)

Tổng quancủa[86](#_bookmark376)–[87](#_bookmark387)

nhắn tin giao dịch[97](#_bookmark435)–[100](#_bookmark450)

xuất bản sự kiện bằng cách sử dụng Pollingmẫu nhà xuất bản[98](#_bookmark442)–[99](#_bookmark443)

xuất bản sự kiện bằng cách sử dụng mẫu theo dõi Nhật ký giao dịch[99](#_bookmark444)–[100](#_bookmark450)

sử dụng cơ sở dữ liệubảng như hàng đợi tin nhắn[97](#_bookmark437)–[98](#_bookmark440)

tương tác yêu cầu/phản hồi không đồng bộ thực hiện[87](#_bookmark390)–[88](#_bookmark394)

kiểm tra tích hợp cho

kiểm tra hợp đồng phía người tiêu dùng[332](#_bookmark1146)–[335](#_bookmark1147) kiểm tra hợp đồng[330](#_bookmark1143)–[335](#_bookmark1147)

hợp đồng mẫu[331](#_bookmark1144)–[332](#_bookmark1145)

Nguyên tử, Tính nhất quán, Độ bền (ACD)[111](#_bookmark477) Tính nguyên tử, tính nhất quán, tính cô lập, tính bền vững

(ACID) giao dịch[98](#_bookmark438),[110](#_bookmark474)

giá trị thuộc tính[245](#_bookmark889)

ghi nhật ký kiểm toán[28](#_bookmark142),[186](#_bookmark688),[366](#_bookmark1246),[377](#_bookmark1302)–[378](#_bookmark1309) thêm mã vào logic kinh doanh[378](#_bookmark1306) hướng theo khía cạnhlập trình[378](#_bookmark1307)

sự kiện tìm nguồn[378](#_bookmark1308)

kiểm toán[350](#_bookmark1187)

xác thực và ủy quyền tái cấu trúc thành các dịch vụ vi mô[453](#_bookmark1573)–[455](#_bookmark1577)

Cổng API ánh xạ cookie USERINFO tới tiêu đề Authorization[455](#_bookmark1576)

LoginHandler đặt cookie USERINFO[454](#_bookmark1574)–[455](#_bookmark1575)

bảo mật trong kiến ​​trúc vi dịch vụ xử lý xác thực[354](#_bookmark1206)–[355](#_bookmark1208)

xử lý ủy quyền[356](#_bookmark1209) Khái niệm máy chủ ủy quyền[357](#_bookmark1217) kiểm tra tự động[28](#_bookmark143),[293](#_bookmark1045),[295](#_bookmark1054)–[296](#_bookmark1055) phun xăng tự động[411](#_bookmark1440) Avro[72](#_bookmark320)

Dịch vụ cổng API AWS[271](#_bookmark991)–[272](#_bookmark993)

Dịch vụ cân bằng tải ứng dụng AWS[272](#_bookmark995) AWS DynamoDB[242](#_bookmark882)–[252](#_bookmark915)

mô hình hóa dữ liệu và thiết kế truy vấn[244](#_bookmark887)–[249](#_bookmark906) phát hiện sự kiện trùng lặp[248](#_bookmark903)–[249](#_bookmark906) truy vấn findOrderHistory[245](#_bookmark890)–[247](#_bookmark898)

Bảng lịch sử đơn hàng FTGO[245](#_bookmark888) phân trang kết quả truy vấn[247](#_bookmark899) đang cập nhật đơn hàng[247](#_bookmark900)–[248](#_bookmark902)

OrderHistoryDaoDynamoLớpDb [249](#_bookmark908)–[252](#_bookmark915)

phương thức addOrder()[249](#_bookmark909)–[250](#_bookmark910)

phương thức findOrderHistory()[251](#_bookmark913)–[252](#_bookmark915)

phương thức idempotentUpdate()[250](#_bookmark912)–[251](#_bookmark914)

phương thức notePickedUp()[250](#_bookmark911)

Mô-đun OrderHistoryEventHandlers[243](#_bookmark884)–[244](#_bookmark885) AWS Gateway, triển khai các dịch vụ RESTful

sử dụng[419](#_bookmark1483)–[426](#_bookmark1502)

triển khai các hàm lambda bằng cách sử dụng khung Serverless[425](#_bookmark1501)–[426](#_bookmark1502)

thiết kế dịch vụ nhà hàng[419](#_bookmark1485)–[423](#_bookmark1497) dịch vụ đóng gói dưới dạng tệp ZIP[424](#_bookmark1499)

AWS Lambda

lợi ích của hàm lambda[418](#_bookmark1476) phát triển các hàm lambda[417](#_bookmark1463) nhược điểm của hàm lambda[419](#_bookmark1478) gọi hàm lambda[417](#_bookmark1469)–[418](#_bookmark1474)

định nghĩa các hàm lambda theo lịch trình[418](#_bookmark1472) xử lý sự kiện[418](#_bookmark1471)

xử lý các yêu cầu HTTP[417](#_bookmark1470)

gọi các hàm lambda bằng cách sử dụng các yêu cầu dịch vụ web[418](#_bookmark1473)

tổng quan về[416](#_bookmark1455)

Dịch vụ RESTful[419](#_bookmark1483)–[426](#_bookmark1502)

triển khai các hàm lambda bằng cách sử dụng khung Serverless[425](#_bookmark1501)–[426](#_bookmark1502)

thiết kế dịch vụ nhà hàng[419](#_bookmark1485)–[423](#_bookmark1497) dịch vụ đóng gói dưới dạng tệp ZIP[424](#_bookmark1499)

thuộc tính aws.region[363](#_bookmark1232)

Sợi trục[202](#_bookmark755)

Chức năng Azure, Microsoft[416](#_bookmark1457)

B

Phần cuối chomẫu frontend (BFF)[264](#_bookmark960)–[266](#_bookmark963) phân loại[288](#_bookmark1033)

@Before phương thức setUp()[309](#_bookmark1097) phương thức beforeHandling()[423](#_bookmark1495) Mẫu Big Ball of Mud[2](#_bookmark10)

viết lại vụ nổ lớn[430](#_bookmark1514) định dạng tin nhắn nhị phân[72](#_bookmark319) bối cảnh bị giới hạn[55](#_bookmark256)

nhắn tin dựa trên môi giới[90](#_bookmark409)–[94](#_bookmark423) lợi ích và hạn chế của[93](#_bookmark418)–[94](#_bookmark423)

thực hiệnkênh tin nhắn sử dụng[93](#_bookmark417) tổng quan về[92](#_bookmark413)

nhắn tin không qua môi giới [91](#_bookmark410)–[92](#_bookmark412) Mô-đun API trình duyệt[264](#_bookmark953) khả năng kinh doanh[40](#_bookmark221)

logic kinh doanh[146](#_bookmark572)–[219](#_bookmark806)

thêm mã ghi nhật ký kiểm tra vào[378](#_bookmark1306) sự kiện miền[160](#_bookmark621)–[168](#_bookmark647)

tiêu thụ[167](#_bookmark646)–[168](#_bookmark647)

được định nghĩa[161](#_bookmark628)

sự kiện làm giàu[161](#_bookmark630)–[162](#_bookmark631)

logic kinh doanh*(tiếp theo)*

tạo ra[164](#_bookmark638)–[165](#_bookmark640)

xác định[162](#_bookmark633)–[163](#_bookmark635)

xuất bản[166](#_bookmark641)–[167](#_bookmark644) lý do để xuất bản[160](#_bookmark624)–[161](#_bookmark626)

mô hình miềnthiết kế[152](#_bookmark600)–[160](#_bookmark618) tổng hợp[154](#_bookmark607)–[160](#_bookmark618)

vấn đề với ranh giới mờ[153](#_bookmark603)–[154](#_bookmark605) sự kiện tìm nguồn[184](#_bookmark680)–[202](#_bookmark750)

lợi ích của[199](#_bookmark732)–[200](#_bookmark738)

nhược điểm của[200](#_bookmark740)–[202](#_bookmark750)

sự kiện xuất bản[194](#_bookmark712)–[195](#_bookmark715) sự kiện miền đang phát triển[198](#_bookmark726)–[199](#_bookmark730)

xử lý các bản cập nhật đồng thời bằng cách sử dụng khóa tối ưu[193](#_bookmark708)–[194](#_bookmark710)

xử lý tin nhắn bất biến[197](#_bookmark722) tổng quan về[186](#_bookmark692)–[193](#_bookmark704)

ảnh chụp nhanh, cải thiện hiệu suấtvới[195](#_bookmark718)–[196](#_bookmark720)

sự kiên trì truyền thống[185](#_bookmark683)–[186](#_bookmark690) cửa hàng sự kiệnthực hiện[202](#_bookmark752)–[209](#_bookmark781)

Khung máy khách Eventuate cho Java[205](#_bookmark768)–[209](#_bookmark781) Cửa hàng sự kiện địa phương Eventuate[203](#_bookmark758)–[205](#_bookmark766)

Logic kinh doanh dịch vụ nhà bếp[168](#_bookmark649)–[173](#_bookmark660) Logic kinh doanh dịch vụ đặt hàng[173](#_bookmark662)–[182](#_bookmark676)

Tổng hợp đơn hàng[175](#_bookmark664)–[180](#_bookmark675)

Lớp OrderService[180](#_bookmark674)–[182](#_bookmark676)

mô hình tổ chức[147](#_bookmark575)–[152](#_bookmark598) Mẫu mô hình miền[150](#_bookmark587)–[151](#_bookmark589) thiết kế theo miền[151](#_bookmark592)–[152](#_bookmark598) Mẫu kịch bản giao dịch[149](#_bookmark579)–[150](#_bookmark584)

saga và sự kiện tìm nguồn cung ứng cùng nhau[209](#_bookmark783)–[218](#_bookmark805) tạo ra saga dựa trên dàn nhạc[211](#_bookmark787)–[212](#_bookmark790) thực hiện dựa trên vũ đạotruyện dài

sử dụng sự kiện nguồn[210](#_bookmark785) triển khai saga dựa trên sự kiện

người tham gia[213](#_bookmark792)–[216](#_bookmark796)

thực hiệnsaga orchestrators sử dụng event sourcing[216](#_bookmark798)–[218](#_bookmark805)

Các mẫu thiết kế logic kinh doanh Tổng hợp[147](#_bookmark576),[152](#_bookmark601)–[160](#_bookmark619)

Sự kiện miền[160](#_bookmark622)

Mô hình miền[150](#_bookmark588)–[151](#_bookmark590)

Nguồn sự kiện[184](#_bookmark681)

Kịch bản giao dịch[149](#_bookmark580)–[150](#_bookmark585)

lớp logic kinh doanh[38](#_bookmark206),[436](#_bookmark1532)

bằng biện pháp đối phó giá trị[131](#_bookmark549)–[132](#_bookmark550)

C

lưu trữ đệm[262](#_bookmark945),[288](#_bookmark1034)

thao tác cancel()[177](#_bookmark669)

phương thức cancelOrder() [460](#_bookmark1596)

Định lý CAP[113](#_bookmark492)

CCP (Nguyên tắc đóng chung)[56](#_bookmark260)

phiên họp tập trung[354](#_bookmark1205) thay đổi tỷ lệ thất bại[31](#_bookmark170) biên đạo múa[111](#_bookmark479)

saga dựa trên vũ đạo[118](#_bookmark507)–[121](#_bookmark516) lợi ích và hạn chế của[121](#_bookmark513)

thực hiện saga Create Order[118](#_bookmark508)–[119](#_bookmark509) triển khai sử dụng sự kiện nguồn[210](#_bookmark785) giao tiếp dựa trên sự kiện đáng tin cậy[120](#_bookmark510)–[121](#_bookmark512)

CI (Tích hợp liên tục)[6](#_bookmark31),[306](#_bookmark1082),[357](#_bookmark1215) Mẫu cầu dao điện[77](#_bookmark342)–[80](#_bookmark352)

phát triển các proxy RPI mạnh mẽ[79](#_bookmark345) phục hồi từ các dịch vụ không khả dụng[79](#_bookmark351)–[80](#_bookmark352)

Khái niệm khách hàng[358](#_bookmark1221)

Mẫu khám phá phía máy khách[82](#_bookmark358)–[83](#_bookmark361) lệnh tin nhắn[86](#_bookmark378)

[Phân chia trách nhiệm truy vấn lệnh.](#_bookmark129)*Nhìn thấy*

Mẫu CQRS

nhắn tin dựa trên lệnh/trả lời [102](#_bookmark456)–[103](#_bookmark457)

lệnh[41](#_bookmark223)

giai đoạn kiểm tra cam kết [306](#_bookmark1085) hồ sơ đã cam kết[130](#_bookmark541)

Đóng cửa chungNguyên tắc (CCP)[56](#_bookmark261)–[57](#_bookmark267) giao tiếp

linh hoạt[93](#_bookmark420)

quá trình liên kết an toàn[350](#_bookmark1188)

mẫu giao tiếp[23](#_bookmark115)–[25](#_bookmark125) biện pháp đối phó cập nhật giao hoán[130](#_bookmark543) giao dịch có thể đền bù[116](#_bookmark500),[128](#_bookmark537),[450](#_bookmark1563)

giao dịch bù trừ[450](#_bookmark1560)

kiểm tra thời gian biên dịch[297](#_bookmark1059)

kiểm tra thành phần[306](#_bookmark1087),[339](#_bookmark1158)–[340](#_bookmark1162) cho Dịch vụ Đặt hàng FTGO

Lớp OrderServiceComponentTestStepDefinitions[341](#_bookmark1165)–[344](#_bookmark1166)

đang chạy[344](#_bookmark1167)–[345](#_bookmark1168)

viết[340](#_bookmark1164)–[345](#_bookmark1168)

kiểm tra thành phần trong quá trình[339](#_bookmark1159)

kiểm tra thành phần ngoài quy trình[339](#_bookmark1160)–[340](#_bookmark1162) biểu thức điều kiện[248](#_bookmark904)

Ống dẫn[381](#_bookmark1327)

Bản đồ cấu hình[402](#_bookmark1404)

dịch vụ có thể cấu hình[360](#_bookmark1224)–[364](#_bookmark1237)

cấu hình bên ngoài dựa trên kéo [363](#_bookmark1235)–[364](#_bookmark1237) cấu hình bên ngoài dựa trên đẩy[362](#_bookmark1229)–[363](#_bookmark1233)

Lớp @ConfigurationProperties[276](#_bookmark1007) thử nghiệm hợp đồng tiêu dùng[301](#_bookmark1066)–[303](#_bookmark1072)

cho tương tác yêu cầu/phản hồi không đồng bộ[332](#_bookmark1146)–[335](#_bookmark1147)

cho API nhắn tin[305](#_bookmark1077)

cho các tương tác theo kiểu xuất bản/đăng ký[328](#_bookmark1138)–[330](#_bookmark1141)

cho các tương tác theo kiểu yêu cầu/phản hồi dựa trên REST[324](#_bookmark1130)–[326](#_bookmark1133)

nhóm người tiêu dùng[94](#_bookmark427)

do người tiêu dùng thúc đẩykiểm tra hợp đồng[28](#_bookmark144),[302](#_bookmark1069)

tham số consumerId[229](#_bookmark843)

người tiêu dùng-nhà cung cấpmối quan hệ[301](#_bookmark1067)

hợp đồng phía người tiêu dùngBài kiểm tra[28](#_bookmark147),[302](#_bookmark1069) thùng chứa

hình ảnh container[395](#_bookmark1373)

Triển khai dịch vụ dưới dạng một container [22](#_bookmark109),[393](#_bookmark1370) Người lái tàu[395](#_bookmark1374)–[398](#_bookmark1386)

triển khai liên tục[5](#_bookmark28)

triển khaiđường ống[305](#_bookmark1080)–[307](#_bookmark1089)

Tích hợp liên tục (CI)[6](#_bookmark32),[306](#_bookmark1083),[357](#_bookmark1216) bộ điều khiển, kiểm tra đơn vị cho[313](#_bookmark1114)–[315](#_bookmark1117) Conway, Melvin[30](#_bookmark163)

Luật của Conway[30](#_bookmark162)

tương quanNHẬN DẠNG[88](#_bookmark391)–[89](#_bookmark398),[120](#_bookmark511)

biện pháp đối phó[111](#_bookmark478),[126](#_bookmark530),[450](#_bookmark1561)

CQRS (Phân tách trách nhiệm truy vấn lệnh)[26](#_bookmark129),[63](#_bookmark283),[160](#_bookmark625),[228](#_bookmark839)–[236](#_bookmark868)

lợi ích của[235](#_bookmark857)–[236](#_bookmark863)

thực hiện hiệu quả[235](#_bookmark859)

cải thiện sự tách biệtcủa mối quan tâm[235](#_bookmark862)–[236](#_bookmark863) truy vấn trong sự kiện dựa trên nguồn

ứng dụng[235](#_bookmark860)

nhược điểm của[236](#_bookmark865)

kiến trúc phức tạp hơn[236](#_bookmark866) độ trễ sao chép[236](#_bookmark867)

động lực để sử dụng[229](#_bookmark841)–[232](#_bookmark849) truy vấn findAvailableRestaurants()

hoạt động[231](#_bookmark847)

hoạt động truy vấn findOrderHistory()[229](#_bookmark842)–[231](#_bookmark846) cần phải tách biệt những mối quan tâm[231](#_bookmark848)–[232](#_bookmark849)

tổng quan về[232](#_bookmark851)–[235](#_bookmark858)

chỉ truy vấndịch vụ[233](#_bookmark855)–[235](#_bookmark858)

lệnh tách biệttừ các truy vấn[232](#_bookmark852)–[233](#_bookmark854) lượt xem

thêm và cập nhật[241](#_bookmark879)–[242](#_bookmark880) thiết kế[236](#_bookmark870)–[242](#_bookmark880)

triển khai với AWS DynamoDB[242](#_bookmark882)–[252](#_bookmark915)

Tạo đơn hàng saga[114](#_bookmark497)–[115](#_bookmark499),[135](#_bookmark556)–[142](#_bookmark563)

Tạo đơn hàngSagangười điều phối[136](#_bookmark557)–[138](#_bookmark558)

Lớp CreateOrderSagaState [138](#_bookmark559) Khung của Eventuate Tram Saga[140](#_bookmark561)–[142](#_bookmark563)

thực hiện bằng cách sử dụng vũ đạo[118](#_bookmark508)–[119](#_bookmark509) thực hiện bằng cách sử dụng dàn nhạc[122](#_bookmark519)–[123](#_bookmark520) Lớp KitchenServiceProxy[139](#_bookmark560)

các hoạt động tạo, cập nhật và xóa (CRUD)[232](#_bookmark853)

phương thức create()[171](#_bookmark655),[204](#_bookmark760)

hoạt động createOrder()[114](#_bookmark498)

Tạo đơn hàngSagangười điều phối[136](#_bookmark557)–[138](#_bookmark558)

Lớp CreateOrderSagaState[138](#_bookmark559)

Lớp CreateOrderSagaTest[312](#_bookmark1106) Các mẫu quan tâm cắt ngang

bên ngoàicấu hình[28](#_bookmark150),[361](#_bookmark1225)

Khung gầm vi dịch vụ[28](#_bookmark152),[378](#_bookmark1312)–[382](#_bookmark1328)

Các hoạt động CRUD (tạo, cập nhật và xóa)[232](#_bookmark853)

Khung dưa chuột [338](#_bookmark1155)

Giao diện CustomerContactInfoRepository[445](#_bookmark1548),

[458](#_bookmark1586)

D

DAO (đối tượng truy cập dữ liệu)[39](#_bookmark216),[149](#_bookmark581),[239](#_bookmark875) lớp logic truy cập dữ liệu[436](#_bookmark1533)

tính nhất quán của dữ liệu[449](#_bookmark1558)–[453](#_bookmark1571)

Mẫu thành phần API và[228](#_bookmark835) duy trì trên các dịch vụ[58](#_bookmark272) tái cấu trúc thành các dịch vụ vi mô

saga và đền bùgiao dịch[451](#_bookmark1568)–[452](#_bookmark1569)

trích xuất trình tự các dịch vụ[452](#_bookmark1570)–[453](#_bookmark1571) hỗ trợ các giao dịch có thể bồi thường

[450](#_bookmark1566)–[451](#_bookmark1567)

Mẫu Saga[25](#_bookmark127)–[26](#_bookmark134),[114](#_bookmark495)–[117](#_bookmark502) mẫu thống nhất dữ liệu[25](#_bookmark126)

Mẫu Saga[25](#_bookmark127)–[26](#_bookmark130),[114](#_bookmark496)–[117](#_bookmark503)

Mô-đun DataLoader[288](#_bookmark1035)

DDD (thiết kế theo miền)[24](#_bookmark121),[34](#_bookmark179) Mẫu tổng hợp DDD[152](#_bookmark600)–[160](#_bookmark618) Debezi[100](#_bookmark446)

Phân tích theo mô hình năng lực kinh doanh[51](#_bookmark242)–[54](#_bookmark249) sự phân hủy[52](#_bookmark247)–[54](#_bookmark249)

xác định năng lực kinh doanh[51](#_bookmark246)–[52](#_bookmark248) mục đích của năng lực kinh doanh[51](#_bookmark244)

sự phân hủy[33](#_bookmark178)–[64](#_bookmark286) Phân tích theo tên miền phụ[54](#_bookmark253)

xác định kiến ​​trúc dịch vụ vi mô của ứng dụng[44](#_bookmark233)–[64](#_bookmark284)

định nghĩa API dịch vụ[61](#_bookmark277)–[64](#_bookmark284) hướng dẫn phân hủy[56](#_bookmark259)–[57](#_bookmark267) xác định hoạt động của hệ thống[45](#_bookmark236)–[50](#_bookmark240) những trở ngại cho sự phân hủy[57](#_bookmark269)–[61](#_bookmark278)

định nghĩa dịch vụ với Phân tích theo mô hình năng lực kinh doanh[51](#_bookmark242)–[54](#_bookmark249)

định nghĩa dịch vụ với Phân tích theo mẫu miền phụ[54](#_bookmark252)–[55](#_bookmark257)

hướng dẫn cho[56](#_bookmark259)–[57](#_bookmark267)

Nguyên tắc đóng chung[56](#_bookmark265)–[57](#_bookmark267) Nguyên tắc trách nhiệm duy nhất[56](#_bookmark264)

những trở ngại đối với[57](#_bookmark269)–[61](#_bookmark278)

lớp học của Chúa[58](#_bookmark275)–[61](#_bookmark278)

duy trì tính nhất quán của dữ liệu trên các dịch vụ[58](#_bookmark272)

độ trễ mạng[57](#_bookmark270)

có được cái nhìn nhất quán về dữ liệu[58](#_bookmark274) giao tiếp liên tiến trình đồng bộ[57](#_bookmark271)

mẫu hình

Phân tích theo năng lực kinh doanh[24](#_bookmark120),[51](#_bookmark243)–[54](#_bookmark250)

Phân tích theo tên miền phụ[24](#_bookmark120),[54](#_bookmark253)

Dịch vụ giao hàng chậm trễ

thay đổi khối FTGO để tương tác với[467](#_bookmark1615)–[470](#_bookmark1623)

giao diện định nghĩa[467](#_bookmark1616)–[468](#_bookmark1617)

giao diện thực hiện[468](#_bookmark1619)–[470](#_bookmark1623) tái cấu trúc khối đơn để gọi giao diện

[468](#_bookmark1618)

thiết kế cho [456](#_bookmark1581)–[457](#_bookmark1583)

mô hình miền[463](#_bookmark1602)–[464](#_bookmark1606)

quyết định dữ liệu nào sẽ di chuyển[464](#_bookmark1604) thiết kế logic miền[464](#_bookmark1605)

xác định những thực thể và trường nàolà một phần của quản lý giao hàng[463](#_bookmark1603)

chức năng giao hàng hiện có[460](#_bookmark1592)–[461](#_bookmark1597) keo dán tích hợp cho[457](#_bookmark1585)–[459](#_bookmark1588),[465](#_bookmark1608)–[467](#_bookmark1613)

Giao diện CustomerContactInfoRepository[458](#_bookmark1586)

thiết kế API[465](#_bookmark1609)–[466](#_bookmark1610)

cách Dịch vụ giao hàng truy cậpDữ liệu FTGO[466](#_bookmark1611)

FTGO truy cập dữ liệu như thế nào[467](#_bookmark1612)

xuất bản và tiêu thụ các sự kiện miền Order và Restau-rant[458](#_bookmark1587)–[459](#_bookmark1588)

tổng quan về[462](#_bookmark1599)–[463](#_bookmark1600)

cờ đã xóa[201](#_bookmark746)

thực hiện hành động[460](#_bookmark1593)

Giao hàngDịch vụImpllớp học[468](#_bookmark1620)

sự phụ thuộc[125](#_bookmark525)

giai đoạn triển khai[306](#_bookmark1088)

triển khai[383](#_bookmark1332)–[427](#_bookmark1503)

Mẫu định dạng đóng gói theo ngôn ngữ cụ thể[386](#_bookmark1334)–[390](#_bookmark1352)

lợi ích của[388](#_bookmark1338)–[389](#_bookmark1341)

nhược điểm của[389](#_bookmark1343)–[390](#_bookmark1352)

Dịch vụ RESTful sử dụng AWSLambda và AWS Gateway[419](#_bookmark1483)–[426](#_bookmark1502)

triển khai các hàm lambda bằng cách sử dụng khung Serverless[425](#_bookmark1501)–[426](#_bookmark1502)

thiết kế dịch vụ nhà hàng[419](#_bookmark1485)–[423](#_bookmark1497) dịch vụ đóng gói dưới dạng tệp ZIP[424](#_bookmark1499)

Mẫu triển khai không có máy chủ[415](#_bookmark1452)–[419](#_bookmark1480) lợi ích của hàm lambda[418](#_bookmark1476) phát triển các hàm lambda[417](#_bookmark1463) nhược điểm của lambdachức năng[419](#_bookmark1478) gọi hàm lambda[417](#_bookmark1469)–[418](#_bookmark1474) tổng quan về[416](#_bookmark1455)

Dịch vụ như mô hình container[393](#_bookmark1369)–[399](#_bookmark1392) lợi ích của[398](#_bookmark1389)

Người lái tàu[395](#_bookmark1372)–[398](#_bookmark1387)

nhược điểm của[399](#_bookmark1391)

Dịch vụ như mô hình máy ảo[390](#_bookmark1350)–[393](#_bookmark1367) lợi ích của[392](#_bookmark1357)

nhược điểm của[392](#_bookmark1363)–[393](#_bookmark1367) Mẫu lưới dịch vụ[380](#_bookmark1324) Mẫu xe ba bánh[410](#_bookmark1434)

với Kubernetes[399](#_bookmark1394)–[415](#_bookmark1450) triển khai cổng API[405](#_bookmark1415)–[406](#_bookmark1418)

triển khai dịch vụ nhà hàng[402](#_bookmark1407)–[405](#_bookmark1413) tổng quan về[399](#_bookmark1396)–[402](#_bookmark1405)

lưới dịch vụ[407](#_bookmark1425)–[415](#_bookmark1450)

triển khai không có thời gian chết[406](#_bookmark1420)–[407](#_bookmark1423)

triển khaiTính thường xuyên[31](#_bookmark168) Các mẫu triển khai

Triển khai dịch vụ dưới dạng một container[22](#_bookmark111),[393](#_bookmark1370) Triển khai dịch vụ dưới dạng VM[390](#_bookmark1353),[392](#_bookmark1358)

Định dạng đóng gói theo ngôn ngữ cụ thể[386](#_bookmark1335),[390](#_bookmark1348)

Triển khai không cần máy chủ[415](#_bookmark1453)–[419](#_bookmark1481)

Lưới dịch vụ[380](#_bookmark1324)

Xe đẩy có thùng[410](#_bookmark1436)

triển khaiđường ống[305](#_bookmark1081)–[307](#_bookmark1090)

Triển khai xem[36](#_bookmark195)

Quy tắc đích [413](#_bookmark1444)

đọc bẩn[127](#_bookmark534)

Phân phốimẫu theo dõi[28](#_bookmark139),[366](#_bookmark1243),[370](#_bookmark1274)–[373](#_bookmark1280) máy chủ theo dõi phân tán[373](#_bookmark1279) thư viện thiết bị[373](#_bookmark1276)

Giao dịch phân tánXử lý (DTP)[112](#_bookmark489) Người lái tàu[395](#_bookmark1372)–[398](#_bookmark1387)

xây dựng hình ảnh[395](#_bookmark1375)–[396](#_bookmark1378)

đẩy hình ảnh vào sổ đăng ký[396](#_bookmark1379)–[397](#_bookmark1383) container đang chạy[397](#_bookmark1384)–[398](#_bookmark1387)

lệnh xây dựng docker[396](#_bookmark1377) Container Docker[267](#_bookmark970) lệnh đẩy docker[397](#_bookmark1381) lệnh chạy docker[397](#_bookmark1385) lệnh thẻ docker[396](#_bookmark1380) tin nhắn tài liệu[86](#_bookmark377) sự kiện miền xuất bản[102](#_bookmark455)

sự kiện miền[160](#_bookmark621)–[168](#_bookmark647),[198](#_bookmark726)–[199](#_bookmark730)

tiêu thụ[167](#_bookmark646)–[168](#_bookmark647),[458](#_bookmark1587)–[459](#_bookmark1588)

được định nghĩa[161](#_bookmark628)

xác định[207](#_bookmark773)

sự kiện làm giàu [161](#_bookmark630)–[162](#_bookmark631) sự tiến hóa lược đồ sự kiện[198](#_bookmark727)–[199](#_bookmark728) tạo ra[164](#_bookmark638)–[165](#_bookmark640)

xác định[162](#_bookmark633)–[163](#_bookmark635)

quản lý các thay đổi lược đồ thông qua upcasting[199](#_bookmark729)

xuất bản[102](#_bookmark455),[166](#_bookmark641)–[167](#_bookmark644),[448](#_bookmark1555)–[449](#_bookmark1556),[458](#_bookmark1587)–[459](#_bookmark1588)

lý do để xuất bản[160](#_bookmark624)–[161](#_bookmark626) đăng ký[208](#_bookmark778)–[209](#_bookmark781),[448](#_bookmark1555)–[449](#_bookmark1556)

mô hình miền[54](#_bookmark254),[150](#_bookmark587)–[160](#_bookmark618) tổng hợp

ranh giới nhất quán[155](#_bookmark608)

thiết kế logic kinh doanh với[159](#_bookmark617)–[160](#_bookmark618) ranh giới rõ ràng[154](#_bookmark607)–[155](#_bookmark610)

độ chi tiết[158](#_bookmark615)

xác định các tập hợp[155](#_bookmark609)

quy tắc cho[155](#_bookmark612)–[157](#_bookmark613)

tạo tên miền cấp caongười mẫu[46](#_bookmark237)–[48](#_bookmark238)

mô hình miền*(tiếp theo)*

Dịch vụ giao hàng[463](#_bookmark1602)–[464](#_bookmark1606)

quyết định dữ liệu nào sẽ di chuyển[464](#_bookmark1604) thiết kế logic miền[464](#_bookmark1605)

xác địnhnhững thực thể và trường nào là một phần của quản lý phân phối[463](#_bookmark1603)

vấn đề với ranh giới mờ[153](#_bookmark603)–[154](#_bookmark605) chia tách[439](#_bookmark1537)–[440](#_bookmark1539)

dịch vụ tên miền KitchenService[171](#_bookmark656)–[172](#_bookmark659) các bài kiểm tra đơn vị cho[312](#_bookmark1110)–[313](#_bookmark1112)

hướng miềnthiết kế (DDD)[24](#_bookmark122),[34](#_bookmark180) DSL (ngôn ngữ dành riêng cho miền)[303](#_bookmark1074)

DTP (Giao dịch phân tán)Xử lý)[112](#_bookmark490)

ống câm[14](#_bookmark69)

tin nhắn trùng lặp[95](#_bookmark430)–[97](#_bookmark433)

theo dõi tin nhắn và loại bỏ các tin nhắn trùng lặp[96](#_bookmark432)–[97](#_bookmark433)

viết trình xử lý tin nhắn idempotent[96](#_bookmark431) Luồng DynamoDB[100](#_bookmark448)

E

hàm cạnh[271](#_bookmark987)

Cây đậu đàn hồi[391](#_bookmark1355)

Tìm kiếm đàn hồi[370](#_bookmark1266)

Chú thích @EnableGateway[279](#_bookmark1015)

kiểm tra đầu cuối[345](#_bookmark1170)–[346](#_bookmark1177)

thiết kế[345](#_bookmark1172)

đang chạy[346](#_bookmark1176)

viết[346](#_bookmark1174)

Xe buýt dịch vụ doanh nghiệp (ESB)[264](#_bookmark957) các thực thể, các bài kiểm tra đơn vị cho[309](#_bookmark1096)–[310](#_bookmark1099) Đối tượng thực thể, DDD[151](#_bookmark593)

các enum[283](#_bookmark1024)

ESB (Xe buýt dịch vụ doanh nghiệp)[264](#_bookmark958) sự kiện. Xem Sự kiện miền

trình xử lý sự kiện

sự kiện được tạo ra bởi dịch vụ AWS[418](#_bookmark1471) có tính chất đồng nhất[240](#_bookmark876)–[241](#_bookmark877)

các bài kiểm tra đơn vị cho[315](#_bookmark1119)–[317](#_bookmark1120) tin nhắn sự kiện[86](#_bookmark379)

sự kiện xuất bản[194](#_bookmark712)–[195](#_bookmark715)

Mẫu tin nhắn không đồng bộ[89](#_bookmark396)–[90](#_bookmark406),[98](#_bookmark442)–[100](#_bookmark450),[102](#_bookmark455)

sự kiện miền[160](#_bookmark621)–[168](#_bookmark647)

tiêu thụ[167](#_bookmark646)–[168](#_bookmark647)

được định nghĩa[161](#_bookmark628)

sự kiện làm giàu[161](#_bookmark630)–[162](#_bookmark631) tạo ra và xuất bản[164](#_bookmark637)–[167](#_bookmark644) xác định[162](#_bookmark633)–[163](#_bookmark635)

lý do cho [160](#_bookmark624)–[161](#_bookmark626)

sự kiện tìm nguồn[194](#_bookmark712)–[195](#_bookmark715),[199](#_bookmark733) sự kiên trì truyền thống và[186](#_bookmark689)

sử dụng thăm dò ý kiến [194](#_bookmark713)–[195](#_bookmark716)

sử dụng giao dịchkhai thác gỗ[195](#_bookmark714) sự kiện tìm nguồn[184](#_bookmark680)–[202](#_bookmark750)

ghi nhật ký kiểm toán[378](#_bookmark1308)

lợi ích của[199](#_bookmark732)–[200](#_bookmark738)

tránh vấn đề không khớp trở kháng O/R[200](#_bookmark736)

bảo tồn lịch sử tổng hợp[199](#_bookmark734)–[200](#_bookmark735) xuất bản sự kiện miền đáng tin cậy[199](#_bookmark733) cỗ máy thời gian dành cho nhà phát triển[200](#_bookmark737)

cập nhật đồng thời và khóa lạc quan[193](#_bookmark708)–[194](#_bookmark710)

nhược điểm của[200](#_bookmark740)–[202](#_bookmark750)

sự phức tạp[200](#_bookmark742)

xóa dữ liệu[201](#_bookmark745)

sự kiện đang diễn biến[201](#_bookmark743)

đường cong học tập[200](#_bookmark741) truy vấn sự kiện lưu trữ[202](#_bookmark749)

sự kiện xuất bản[194](#_bookmark712)–[195](#_bookmark715)

sử dụng thăm dò ý kiến[194](#_bookmark713)–[195](#_bookmark716)

sử dụng giao dịchkhai thác gỗ[195](#_bookmark714) sự kiện miền đang phát triển[198](#_bookmark726)–[199](#_bookmark730)

lược đồ sự kiệnsự tiến hóa[198](#_bookmark727)–[199](#_bookmark728) quản lý các thay đổi lược đồ thông qua

đang lên tiếng[199](#_bookmark729)

xử lý tin nhắn bất biến[197](#_bookmark722) với kho sự kiện dựa trên NoSQL[197](#_bookmark724) với kho sự kiện dựa trên RDBMS[197](#_bookmark723)

tổng quan về[186](#_bookmark692)–[193](#_bookmark704)

phương pháp tổng hợp cần thiết để tạo ra các sự kiện[189](#_bookmark700)–[191](#_bookmark703)

Tổng hợp đơn hàng dựa trên sự kiện[191](#_bookmark702)–[193](#_bookmark704) sự kiện đại diện cho những thay đổi trạng thái[188](#_bookmark697) duy trì tổng hợp bằng cách sử dụng các sự kiện[186](#_bookmark693)–[188](#_bookmark698)

saga và[209](#_bookmark783)–[218](#_bookmark805)

tạo ra saga dựa trên dàn nhạc[211](#_bookmark787)–[212](#_bookmark790) thực hiện dựa trên vũ đạotruyện dài

sử dụng sự kiện nguồn[210](#_bookmark785) triển khai saga dựa trên sự kiện

người tham gia[213](#_bookmark792)–[216](#_bookmark796)

thực hiệnsaga orchestrators sử dụng event sourcing[216](#_bookmark798)–[218](#_bookmark805)

ảnh chụp nhanh và cải thiện hiệu suất[195](#_bookmark718)–[196](#_bookmark720)

rắc rối với truyền thốngsự kiên trì[185](#_bookmark683)–[186](#_bookmark690) ghi nhật ký kiểm toán[186](#_bookmark688)

sự kiện xuất bản đã chốtđến logic kinh doanh[186](#_bookmark689) thiếu lịch sử tổng hợp[186](#_bookmark687)

Sự không phù hợp trở kháng quan hệ đối tượng[185](#_bookmark685)–[186](#_bookmark686)

Cửa hàng sự kiện[202](#_bookmark753)

triển khai cửa hàng sự kiện[202](#_bookmark752)–[209](#_bookmark781) Khung máy khách Eventuate cho Java[205](#_bookmark768)–[209](#_bookmark781)

Lớp AggregateRepository[207](#_bookmark774)–[208](#_bookmark777) định nghĩa lệnh tổng hợp[207](#_bookmark772)

triển khai cửa hàng sự kiện*(tiếp theo)*

định nghĩa tổng hợp với ReflectiveMutable-

Lớp CommandProcessingAggregate[206](#_bookmark770)–[207](#_bookmark771)

định nghĩa sự kiện miền[207](#_bookmark773) đăng ký sự kiện miền[208](#_bookmark778)–[209](#_bookmark781)

Cửa hàng sự kiện địa phương Eventuate[203](#_bookmark758)–[205](#_bookmark766) tiêu thụ sự kiện bằng cách đăng ký sự kiện

người môi giới[205](#_bookmark764)

sự kiện chuyển tiếp truyền bá sự kiện từ cơ sở dữ liệuđể nhắn tin cho người môi giới[205](#_bookmark765)

sơ đồ[203](#_bookmark759)–[205](#_bookmark763)

sự kiện bão táp[162](#_bookmark634)

I/O hướng sự kiện[269](#_bookmark980)

@EventHandlerPhương phápchú thích[208](#_bookmark779) sự kiện. Xem Sự kiện miềnChú thích @EventSubscriber[208](#_bookmark780)

Khung sự kiện[101](#_bookmark453),[202](#_bookmark756),[205](#_bookmark768)–[209](#_bookmark781)

và cập nhật tổng hợp vớiTổng hợp-

Lớp lưu trữ[207](#_bookmark774)–[208](#_bookmark777) định nghĩa lệnh tổng hợp[207](#_bookmark772)

định nghĩa tổng hợp với lớp ReflectiveMutable- CommandProcessingAggregate[206](#_bookmark770)–[207](#_bookmark771)

định nghĩa sự kiện miền[207](#_bookmark773) đăng ký sự kiện miền[208](#_bookmark778)–[209](#_bookmark781)

Cửa hàng sự kiện địa phương Eventuate[203](#_bookmark758)–[205](#_bookmark766) tiêu thụ sự kiện bằng cách đăng ký sự kiện

người môi giới[205](#_bookmark764)

sự kiện chuyển tiếp lan truyềnsự kiện từ cơ sở dữ liệu đến môi giới tin nhắn[205](#_bookmark765)

sơ đồ[203](#_bookmark759)–[205](#_bookmark763)

Sự kiện xảy raXe điện[100](#_bookmark449),[166](#_bookmark642)

Khung của Eventuate Tram Saga [140](#_bookmark561)–[142](#_bookmark563) Ngoại lệmẫu theo dõi[28](#_bookmark140),[366](#_bookmark1244),[376](#_bookmark1297)–[377](#_bookmark1303)

Khung thể hiện[289](#_bookmark1036)–[290](#_bookmark1039) mẫu API bên ngoài[253](#_bookmark918)–[291](#_bookmark1042)

Cổng API[76](#_bookmark337),[227](#_bookmark829),[254](#_bookmark921),[259](#_bookmark933)–[272](#_bookmark994)

Triển khai cổng API[271](#_bookmark986)–[291](#_bookmark1041) sử dụng GraphQL[279](#_bookmark1017)–[291](#_bookmark1041)

sử dụng Netflix Zuul[273](#_bookmark1001)

sử dụng các sản phẩm/dịch vụ có sẵn[271](#_bookmark990)–[272](#_bookmark999) sử dụng Spring Cloud Gateway[273](#_bookmark1002)–[275](#_bookmark1006)

Mẫu cổng API[76](#_bookmark337),[227](#_bookmark829),[254](#_bookmark921),[259](#_bookmark932)–[271](#_bookmark984)

lợi ích của[267](#_bookmark966)

vấn đề thiết kế[268](#_bookmark973)–[271](#_bookmark984)

nhược điểm của[267](#_bookmark967)

Ví dụ về Netflix[267](#_bookmark969)–[268](#_bookmark971)

tổng quan về[259](#_bookmark935)–[266](#_bookmark963)

Backend cho frontend[254](#_bookmark921),[262](#_bookmark941),[264](#_bookmark959)–[266](#_bookmark964)

vấn đề thiết kế[254](#_bookmark920)–[259](#_bookmark930)

ứng dụng JavaScript dựa trên trình duyệt[258](#_bookmark928) Khách hàng di động FTGO[255](#_bookmark923)–[258](#_bookmark925)

ứng dụng của bên thứ ba[258](#_bookmark929)–[259](#_bookmark930)

ứng dụng web[258](#_bookmark927)

được ngoại hóacấu hình[361](#_bookmark1225)

dựa trên kéo[363](#_bookmark1236)–[364](#_bookmark1238)

dựa trên đẩy[262](#_bookmark944)–[263](#_bookmark950)

Mẫu cấu hình bên ngoài[28](#_bookmark149),[361](#_bookmark1225)

F

Đối tượng nhà máy, DDD[151](#_bookmark595) cách ly lỗi[6](#_bookmark36)

cờ đặc trưng[469](#_bookmark1622)

tính năng chuyển đổi[469](#_bookmark1621)

biểu thức lọc [247](#_bookmark895)

tham số lọc[229](#_bookmark845)

hoạt động find() [204](#_bookmark761) hoạt động truy vấn findAvailableRestaurants()[231](#_bookmark847) phương thức findCustomerContactInfo()[447](#_bookmark1553)

hoạt động findOrder()[221](#_bookmark812)–[222](#_bookmark817),[224](#_bookmark822) hoạt động truy vấn findOrderHistory()[229](#_bookmark842)–[231](#_bookmark846),

[251](#_bookmark913)–[252](#_bookmark915)

chỉ số định nghĩavì[245](#_bookmark890)–[247](#_bookmark896) thực hiện[247](#_bookmark894)

Lớp FindRestaurantRequestHandler [421](#_bookmark1486)–[422](#_bookmark1493)

Khung phân hạch[416](#_bookmark1460)

Lưu loát[370](#_bookmark1269)

máng nước[370](#_bookmark1270)

hoạt động gấp[187](#_bookmark694) Ứng dụng FTGO

Các vấn đề thiết kế API cho máy khách di động[255](#_bookmark923)–[258](#_bookmark925) thay đổi khối đơn để tương tácvới Giao hàng

Dịch vụ[467](#_bookmark1615)–[470](#_bookmark1623)

kiểm tra thành phần choDịch vụ đặt hàng[340](#_bookmark1164)–[345](#_bookmark1168) triển khai với Kubernetes[399](#_bookmark1394)–[415](#_bookmark1450)

Cổng API[405](#_bookmark1415)–[406](#_bookmark1418)

Dịch vụ nhà hàng[402](#_bookmark1407)–[405](#_bookmark1413)

lưới dịch vụ[407](#_bookmark1425)–[415](#_bookmark1450)

triển khai không có thời gian chết[406](#_bookmark1420)–[407](#_bookmark1423) kiến trúc dịch vụ vi mô của[12](#_bookmark64)–[13](#_bookmark65) kiến trúc nguyên khối của[3](#_bookmark13)–[4](#_bookmark17)

ftgo-db-bí mật[404](#_bookmark1411)

Lớp FtgoGraphQLClient[290](#_bookmark1040)

phân tích chức năng[10](#_bookmark56)

ranh giới mơ hồ[153](#_bookmark603)–[154](#_bookmark605)

G

GDPR (ChungQuy định bảo vệ dữ liệu)[201](#_bookmark747) mẫu tổng quát[22](#_bookmark107)

Điểm cuối GET REST[271](#_bookmark988) phương thức getDelayedOrders()[456](#_bookmark1582)

truy vấn getOrderDetails()[368](#_bookmark1258) Dưa chuột

thực hiện các thông số kỹ thuật bằng cách sử dụng Cucumber[338](#_bookmark1155) viết bài kiểm tra chấp nhận[337](#_bookmark1154)–[338](#_bookmark1156)

Bộ dụng cụ đi[380](#_bookmark1317)

lớp học của Chúa[58](#_bookmark275)–[61](#_bookmark278)

GoLang (Ngôn ngữ Go)[4](#_bookmark18),[380](#_bookmark1318) Chức năng của Google Cloud[416](#_bookmark1458) sơ đồ dựa trên đồ thị[280](#_bookmark1018)

Đồ thị[279](#_bookmark1017),[281](#_bookmark1021)–[291](#_bookmark1041)

kết nối lược đồ với dữ liệu[285](#_bookmark1029)–[287](#_bookmark1031) định nghĩa lược đồ[282](#_bookmark1023)–[284](#_bookmark1025)

thực hiện truy vấn[284](#_bookmark1026)–[285](#_bookmark1028)

tích hợp máy chủ Apollo GraphQL với Express[289](#_bookmark1036)–[290](#_bookmark1039)

tối ưu hóa tải bằng cách sử dụng xử lý hàng loạt và lưu trữ đệm[288](#_bookmark1032) khách hàng viết[290](#_bookmark1038)–[291](#_bookmark1041)

gRPC[76](#_bookmark336)–[77](#_bookmark340)

H

phương thức handleHttpRequest() [421](#_bookmark1488)

phương thức handleRequest()[417](#_bookmark1466)

kiểm tra sức khỏe[82](#_bookmark359),[365](#_bookmark1241)

Mẫu API kiểm tra sức khỏe[27](#_bookmark137),[366](#_bookmark1248)–[368](#_bookmark1254)

thực hiện điểm cuối[367](#_bookmark1251)–[368](#_bookmark1252)

gọi điểm cuối[368](#_bookmark1253)

kiến trúc lục giác[3](#_bookmark14),[38](#_bookmark212)–[40](#_bookmark217) mẫu thiết kế cấp cao[20](#_bookmark99) Lửng mật[377](#_bookmark1299)

Phản hồi HttpServlet[422](#_bookmark1491)

Khiêm tốn, Jez[30](#_bookmark167)

TÔI

có tính chất đồng nhấtxử lý tin nhắn[96](#_bookmark431),[197](#_bookmark722)

Lượt xem CQRS[240](#_bookmark876)–[241](#_bookmark877)

người tham gia saga dựa trên sự kiện[213](#_bookmark793) với kho sự kiện dựa trên NoSQL[197](#_bookmark724)

với kho sự kiện dựa trên RDBMS[197](#_bookmark723) phương thức idempotentUpdate()[250](#_bookmark912)–[251](#_bookmark914) IDL (ngôn ngữ định nghĩa giao diện)[69](#_bookmark301)

-khả năng[8](#_bookmark45),[34](#_bookmark184),[37](#_bookmark198)

Quan điểm thực hiện[35](#_bookmark193)

bộ chuyển đổi đầu vào[3](#_bookmark15),[38](#_bookmark213)

mô hình cơ sở hạ tầng[23](#_bookmark115)–[24](#_bookmark116) hệ thống init, Linux[390](#_bookmark1354)

ngữ cảnh bảo mật trong bộ nhớ[353](#_bookmark1204) thư viện thiết bị[373](#_bookmark1276)

keo dán tích hợp[444](#_bookmark1544)–[449](#_bookmark1556) thiết kế API cho[444](#_bookmark1545)–[445](#_bookmark1546)

cho Dịch vụ giao hàng chậm trễ[457](#_bookmark1585)–[459](#_bookmark1588),[465](#_bookmark1608)–[467](#_bookmark1613) Kho thông tin liên hệ khách hàng

giao diện [458](#_bookmark1586) thiết kế API[465](#_bookmark1609)–[466](#_bookmark1610)

cách Dịch vụ giao hàng truy cập dữ liệu FTGO[466](#_bookmark1611) FTGO truy cập dữ liệu như thế nào[467](#_bookmark1612)

xuất bản và tiêu thụ các sự kiện miền Order và Restaurant[458](#_bookmark1587)–[459](#_bookmark1588)

cách monolith xuất bản và đăng ký các sự kiện miền[448](#_bookmark1555)–[449](#_bookmark1556)

thực hiện chống tham nhũnglớp[446](#_bookmark1551)–[448](#_bookmark1554) chọn phong cách tương tác và IPC

cơ chế[445](#_bookmark1547)–[446](#_bookmark1550)

kiểm tra tích hợp[319](#_bookmark1123)–[335](#_bookmark1147) yêu cầu/phản hồi không đồng bộ

tương tác[330](#_bookmark1143)–[335](#_bookmark1147)

hợp đồng mẫu[331](#_bookmark1144)–[332](#_bookmark1145)

kiểm tra tương tác yêu cầu/phản hồi không đồng bộ[332](#_bookmark1146)–[335](#_bookmark1147)

kiểm tra tích hợp tính bền bỉ[321](#_bookmark1125)–[322](#_bookmark1126) tương tác theo kiểu xuất bản/đăng ký[326](#_bookmark1135)–[330](#_bookmark1141)

hợp đồng xuất bản sự kiện OrderCreated[327](#_bookmark1136)–[328](#_bookmark1137)

kiểm tra cho Dịch vụ Lịch sử Đơn hàng [329](#_bookmark1140)–[330](#_bookmark1141) kiểm tra cho Dịch vụ Đặt hàng[328](#_bookmark1138)–[329](#_bookmark1139)

Tương tác theo kiểu yêu cầu/phản hồi dựa trên REST[322](#_bookmark1128)–[326](#_bookmark1133)

hợp đồng mẫu[324](#_bookmark1129)

kiểm tra cho API gateway OrderServiceProxy[325](#_bookmark1131)–[326](#_bookmark1133)

kiểm tra cho Dịch vụ Đặt hàng[324](#_bookmark1130)–[325](#_bookmark1132) phong cách tương tác[67](#_bookmark292)–[68](#_bookmark300),[87](#_bookmark389)–[89](#_bookmark400)

không đồng bộ[104](#_bookmark465)–[105](#_bookmark466)

thông báo một chiều[89](#_bookmark395)

xuất bản/phản hồi không đồng bộ[89](#_bookmark397)

xuất bản/đăng ký[89](#_bookmark396)

yêu cầu/phản hồi và yêu cầu không đồng bộ/phản ứng[87](#_bookmark390)–[88](#_bookmark394)

lựa chọn[445](#_bookmark1547)–[446](#_bookmark1550)

ngôn ngữ định nghĩa giao diện (IDL)[69](#_bookmark302) bất biến[153](#_bookmark604)

IPC (giao tiếp giữa các tiến trình)[24](#_bookmark124),[65](#_bookmark288),

[93](#_bookmark422)–[109](#_bookmark472)

Tổng quancủa[66](#_bookmark290)–[72](#_bookmark323)

định nghĩa API[68](#_bookmark299)–[69](#_bookmark303)

API đang phát triển[69](#_bookmark305)–[71](#_bookmark312)

phong cách tương tác [67](#_bookmark292)–[68](#_bookmark300)

định dạng tin nhắn[71](#_bookmark314)–[72](#_bookmark323)

sử dụng mẫu tin nhắn không đồng bộ[85](#_bookmark373)–[103](#_bookmark457) người nhận và tin nhắn cạnh tranh

đặt hàng[94](#_bookmark425)–[95](#_bookmark428)

tạo đặc tả API[89](#_bookmark402)–[90](#_bookmark407) tin nhắn trùng lặp[95](#_bookmark430)–[97](#_bookmark433)

cải thiện tính khả dụng[103](#_bookmark459)–[108](#_bookmark470)

phong cách tương tác [87](#_bookmark389)–[89](#_bookmark400)

thư viện và khuôn khổ cho[100](#_bookmark452)–[103](#_bookmark457) môi giới tin nhắn[90](#_bookmark409)–[94](#_bookmark423)

Tổng quancủa[86](#_bookmark376)–[87](#_bookmark387)

nhắn tin giao dịch[97](#_bookmark435)–[100](#_bookmark450)

sử dụng thủ tục Remote đồng bộmẫu gọi[72](#_bookmark325)–[85](#_bookmark370)

Mẫu cầu dao điện[77](#_bookmark342)–[80](#_bookmark352) gRPC[76](#_bookmark336)–[77](#_bookmark340)

NGHỈ NGƠI[73](#_bookmark328)–[76](#_bookmark338)

dịch vụkhám phá[80](#_bookmark355)–[85](#_bookmark370)

Istio[381](#_bookmark1325)

triển khai dịch vụ[410](#_bookmark1437)–[412](#_bookmark1442)

Đại diện sứ giả[410](#_bookmark1433)

lưới dịch vụ[408](#_bookmark1427)–[410](#_bookmark1438)

J

lệnh java -jar[395](#_bookmark1376) Jenkins[306](#_bookmark1084)

Cookie JSESSIONID[351](#_bookmark1196)

tin nhắn JSON[71](#_bookmark316)

JUL (java.util.logging)[369](#_bookmark1261)

JWT (Mã thông báo web JSON)[28](#_bookmark156),[356](#_bookmark1210)–[357](#_bookmark1213)

K

Kafka[92](#_bookmark416)

biểu thức điều kiện chính[247](#_bookmark897) Kibana[370](#_bookmark1267)

Dịch vụ nhà bếp

logic kinh doanh[168](#_bookmark649)–[173](#_bookmark660)

Tổng hợp vé[169](#_bookmark651)–[173](#_bookmark660)

Lớp KitchenServiceCommandHandler [172](#_bookmark658)–[173](#_bookmark660)

Lớp KitchenServiceProxy[139](#_bookmark560)

Gói Kong[272](#_bookmark997)

lệnh kubectl apply[404](#_bookmark1410) lệnh kubectl apply -f[406](#_bookmark1421) Kubernetes[399](#_bookmark1394)–[415](#_bookmark1450)

triển khai cổng API[405](#_bookmark1415)–[406](#_bookmark1418) triển khai dịch vụ nhà hàng[402](#_bookmark1407)–[405](#_bookmark1413) tổng quan về[399](#_bookmark1396)–[402](#_bookmark1405)

ngành kiến ​​​​trúc[400](#_bookmark1398)–[402](#_bookmark1401)

các khái niệm chính[402](#_bookmark1402)

lưới dịch vụ[407](#_bookmark1425)–[415](#_bookmark1450)

triển khai dịch vụ[410](#_bookmark1437)–[412](#_bookmark1442)

triển khai v2 của Consumer Service[414](#_bookmark1446) Istio[408](#_bookmark1427)–[412](#_bookmark1442)

định tuyến lưu lượng sản xuất đến v2[415](#_bookmark1449) quy tắc định tuyến để định tuyến đến phiên bản v1

[412](#_bookmark1441)–[413](#_bookmark1445)

định tuyến lưu lượng thử nghiệm đến v2[414](#_bookmark1447)

triển khai không có thời gian chết[406](#_bookmark1420)–[407](#_bookmark1423)

L

Thời gian[202](#_bookmark754)

hàm lambda[271](#_bookmark992),[416](#_bookmark1461)

lợi ích của[418](#_bookmark1476)

triển khai sử dụng Serverless framework[425](#_bookmark1501)–[426](#_bookmark1502) đang phát triển[417](#_bookmark1463)

nhược điểm của[419](#_bookmark1478)

triệu hồi[417](#_bookmark1469)–[418](#_bookmark1474)

định nghĩa các hàm lambda theo lịch trình[418](#_bookmark1472) xử lý các sự kiện được tạo ra bởi AWS

dịch vụ[418](#_bookmark1471)

xử lý các yêu cầu HTTP[417](#_bookmark1470) sử dụng yêu cầu dịch vụ web[418](#_bookmark1473)

Mẫu định dạng đóng gói theo ngôn ngữ cụ thể[386](#_bookmark1334)–[390](#_bookmark1352)

lợi ích của[388](#_bookmark1338)–[389](#_bookmark1341)

sử dụng tài nguyên hiệu quả[389](#_bookmark1340) triển khai nhanh chóng[389](#_bookmark1339)

nhược điểm của[389](#_bookmark1343)–[390](#_bookmark1352)

tự động xác định nơi đặt các phiên bản dịch vụ[390](#_bookmark1347)

thiếu sự đóng gói của ngăn xếp công nghệ[389](#_bookmark1344) thiếu sự cô lập[390](#_bookmark1346)

không có khả năng hạn chế tài nguyên tiêu thụ[389](#_bookmark1345)

độ trễ[419](#_bookmark1479)

phong cách kiến ​​trúc nhiều lớp[37](#_bookmark204)–[38](#_bookmark211) hệ thống tập tin phân lớp[397](#_bookmark1382)

thời gian dẫn[31](#_bookmark169),[293](#_bookmark1046)

ứng dụng dòng mã (LOC)[5](#_bookmark24) Dữ liệu LinkedIn[100](#_bookmark447)

Liên kết[381](#_bookmark1326)

sống độngProbe[404](#_bookmark1409)

Dịch vụ LoadBalancer[405](#_bookmark1416)

Ứng dụng LOC (dòng mã)[5](#_bookmark24)

Mẫu tổng hợp nhật ký[27](#_bookmark138),[365](#_bookmark1242),[368](#_bookmark1257)–[370](#_bookmark1271) cơ sở hạ tầng tổng hợp nhật ký[370](#_bookmark1265)

thế hệ nhật ký[369](#_bookmark1260)–[370](#_bookmark1264)

nhật ký4j[369](#_bookmark1262)

Đăng nhập lại[369](#_bookmark1263)

Hợp lýxem[35](#_bookmark192)

Trình xử lý đăng nhập[352](#_bookmark1200),[454](#_bookmark1574)–[455](#_bookmark1575)

Nhật ký[370](#_bookmark1268)

khớp nối lỏng lẻo[93](#_bookmark419),[121](#_bookmark514)

mất bản cập nhật[127](#_bookmark533)

Tôi

Phần CHÍNH, Semvers[70](#_bookmark307) hàm makeContextWithDependencies()[290](#_bookmark1037) phun xăng sidecar thủ công[411](#_bookmark1439)

Martin, Robert C.[57](#_bookmark266) máy chủ[400](#_bookmark1399) thời gian trung bình để phục hồi[31](#_bookmark171) Mẫu lưu niệm[196](#_bookmark719)

môi giới tin nhắn [85](#_bookmark374), [90](#_bookmark409)–[94](#_bookmark423) lợi ích và hạn chế của[93](#_bookmark418)–[94](#_bookmark423)

thực hiệnkênh tin nhắn sử dụng[93](#_bookmark417) tổng quan về[92](#_bookmark413)

đệm tin nhắn[93](#_bookmark421)

kênh tin nhắn[86](#_bookmark380)–[87](#_bookmark387),[93](#_bookmark417) lớp bộ điều hợp trình xử lý tin nhắn[86](#_bookmark381)

trình xử lý tin nhắn, các bài kiểm tra đơn vị cho[315](#_bookmark1119)–[317](#_bookmark1120) định danh tin nhắn[88](#_bookmark392)

thứ tự tin nhắn [94](#_bookmark425)–[95](#_bookmark428) lớp bộ điều hợp người gửi tin nhắn[86](#_bookmark382)

[nhắn tin.](#_bookmark371)*Nhìn thấy*Mẫu tin nhắn không đồng bộ[Mẫu phong cách nhắn tin.](#_bookmark371)*Nhìn thấy*Tin nhắn không đồng bộ

bộ sưu tập số liệu mẫu ing [262](#_bookmark947)

Khung vi mô[380](#_bookmark1319)

thư viện micrometer-registry-prometheus [375](#_bookmark1292)

Kiến trúc dịch vụ vi mô[8](#_bookmark43)–[14](#_bookmark71),[34](#_bookmark183),[43](#_bookmark230) như là hình thức mô-đun [11](#_bookmark58)–[12](#_bookmark60) lợi ích của[14](#_bookmark74)–[17](#_bookmark80)

liên tụccung cấp và triển khai các ứng dụng lớn, phức tạp[15](#_bookmark75)

cải thiện cô lập lỗi[16](#_bookmark78) độc lậpdịch vụ có thể mở rộng[16](#_bookmark77) thử nghiệm công nghệ mới và

sự nhận nuôi[16](#_bookmark79)–[17](#_bookmark80)

dịch vụ nhỏ, dễ bảo trì[15](#_bookmark76) xác định[44](#_bookmark233)–[64](#_bookmark284)

hướng dẫn phân hủy[56](#_bookmark259)–[57](#_bookmark267) định nghĩa API dịch vụ[61](#_bookmark277)–[64](#_bookmark284) xác định hoạt động của hệ thống[45](#_bookmark236)–[50](#_bookmark240)

những trở ngại trong việc phân tích một ứng dụng thành các dịch vụ[57](#_bookmark269)–[61](#_bookmark278)

định nghĩa dịch vụ với Phân tích theo mô hình năng lực kinh doanh[51](#_bookmark242)–[54](#_bookmark249)

định nghĩa dịch vụ với Phân tích theo mẫu miền phụ[54](#_bookmark252)–[55](#_bookmark257)

nhược điểm của[17](#_bookmark82)–[19](#_bookmark90)

thời gian áp dụng[18](#_bookmark89)–[19](#_bookmark90)

thách thức trong việc tìm kiếm dịch vụ phù hợp[17](#_bookmark83) hệ thống phân tán phức tạp[17](#_bookmark84)–[18](#_bookmark87) phối hợp triển khai[18](#_bookmark88)

mỗi dịch vụ có cơ sở dữ liệu riêng[12](#_bookmark62) Ứng dụng FTGO[12](#_bookmark64)–[13](#_bookmark65)

khớp nối lỏng lẻo,được định nghĩa[42](#_bookmark226)–[43](#_bookmark227) không phải là viên đạn bạc[19](#_bookmark93)–[20](#_bookmark94)

mối quan hệ giữa quá trình, tổ chức,Và[29](#_bookmark159)–[32](#_bookmark175)

mặt con người trong việc áp dụng các dịch vụ vi mô[31](#_bookmark174)–[32](#_bookmark175)

tổ chức phát triển và cung cấp phần mềm[29](#_bookmark161)–[30](#_bookmark164)

quá trình phát triển và cung cấp phần mềm[30](#_bookmark166)–[31](#_bookmark172)

sự không quan trọng tương đối của quy mô dịch vụ[43](#_bookmark229) vai trò của thư viện chia sẻ[43](#_bookmark228)

khối lập phương[8](#_bookmark50)–[11](#_bookmark59)

Tỷ lệ trục X[9](#_bookmark51)

Tỷ lệ trục Y[10](#_bookmark55)–[11](#_bookmark59)

Tỷ lệ trục Z[9](#_bookmark52)–[10](#_bookmark54)

kiến trúc hướng dịch vụ so với[13](#_bookmark67)–[14](#_bookmark71) dịch vụ, được xác định[41](#_bookmark222)–[42](#_bookmark225)

kiến trúc phần mềm[34](#_bookmark188)–[37](#_bookmark199)

Mô hình xem 4+1 của[35](#_bookmark191)–[37](#_bookmark196)

định nghĩa của[35](#_bookmark190)

sự liên quan của [37](#_bookmark197)

quản lý giao dịch[111](#_bookmark482)–[117](#_bookmark502) duy trì tính nhất quán của dữ liệu[114](#_bookmark495)–[117](#_bookmark502) nhu cầu giao dịch phân tán[112](#_bookmark485)

sự cố với các giao dịch phân tán[112](#_bookmark487)–[114](#_bookmark493) Mẫu khung gầm dịch vụ vi mô[28](#_bookmark153),[378](#_bookmark1313)–[382](#_bookmark1329)

lưới dịch vụ[380](#_bookmark1323)–[382](#_bookmark1329)

sử dụng [379](#_bookmark1316)–[380](#_bookmark1320) Phần NHỎ, Semvers[70](#_bookmark308) Máy trộn[409](#_bookmark1431)

Mô-đun API di động [264](#_bookmark952) Mockito[305](#_bookmark1075)

chế nhạo[296](#_bookmark1056)

tính mô-đun, kiến ​​trúc vi dịch vụ như một hình thức[11](#_bookmark58)–[12](#_bookmark60)

Trừu tượng đơn sắc[277](#_bookmark1010)

kiến trúc nguyên khối[1](#_bookmark5)–[32](#_bookmark176),[40](#_bookmark220)

lợi ích của[4](#_bookmark20)

nguyên nhân của địa ngục nguyên khối[4](#_bookmark22)–[7](#_bookmark39) sự đe dọa do sự phức tạp[4](#_bookmark23)–[5](#_bookmark25) con đường dài và gian khổ từ cam kết đến

triển khai [5](#_bookmark27)–[6](#_bookmark33)

thách thức về độ tin cậy[6](#_bookmark35)

thách thức mở rộng quy mô [6](#_bookmark34)

phát triển chậm [5](#_bookmark26)

công nghệ ngăn xếp lỗi thời[6](#_bookmark37)–[7](#_bookmark39) Kiến trúc đơn khối FTGO[3](#_bookmark13)–[4](#_bookmark17)

phương thức nhân()[310](#_bookmark1103)

MyBATIS[185](#_bookmark684)

N

Netflix Falcor[281](#_bookmark1022)

Netflix Hystrix[79](#_bookmark349)

Netflix Zuul[273](#_bookmark1001)

Netflix, như là cổng API[267](#_bookmark969)–[268](#_bookmark971) độ trễ mạng[57](#_bookmark270)

thời gian chờ mạng[79](#_bookmark346)

Dịch vụ NodePort[406](#_bookmark1417)

các nút[280](#_bookmark1019),[400](#_bookmark1400)

I/O không chặn[268](#_bookmark977)

yêu cầu không chức năng[8](#_bookmark46)

thuộc tính không phải khóa[246](#_bookmark893) Kho lưu trữ sự kiện dựa trên NoSQL

tạo ra sagangười điều phối khi sử dụng[211](#_bookmark789)–[212](#_bookmark790) xử lý tin nhắn bất biến khi sử dụng[197](#_bookmark724) SQL so với[237](#_bookmark872)–[238](#_bookmark873)

phương thức notePickedUp()[250](#_bookmark911)

Ồ

Không khớp trở kháng O/R (Quan hệ đối tượng)[185](#_bookmark685)–[186](#_bookmark686),[200](#_bookmark736)

Giao thức OAuth 2.0[357](#_bookmark1214)–[360](#_bookmark1222) mẫu thiết kế hướng đối tượng[20](#_bookmark100)

lập trình hướng đối tượng (OOP)[149](#_bookmark582)

Sự không phù hợp trở kháng đối tượng-quan hệ (O/R)[185](#_bookmark685)–[186](#_bookmark686),[200](#_bookmark736)

khả năng quan sát[349](#_bookmark1182)

mẫu quan sát được[27](#_bookmark136)–[28](#_bookmark145)

Số liệu ứng dụng[373](#_bookmark1285)–[376](#_bookmark1295)

Nhật ký kiểm toán[377](#_bookmark1304)–[378](#_bookmark1310)

Phân phối theo dõi[370](#_bookmark1275)–[373](#_bookmark1281)

Theo dõi ngoại lệ[376](#_bookmark1298)–[377](#_bookmark1300) API kiểm tra sức khỏe[366](#_bookmark1249)–[368](#_bookmark1255) Tổng hợp nhật ký[366](#_bookmark1249),[368](#_bookmark1259)–[370](#_bookmark1272)

dịch vụ có thể quan sát được[364](#_bookmark1240)–[378](#_bookmark1309) Mẫu số liệu ứng dụng[373](#_bookmark1284)–[376](#_bookmark1294)

thu thập số liệu về mức độ dịch vụ[374](#_bookmark1287)–[375](#_bookmark1289) cung cấp số liệu cho dịch vụ số liệu[375](#_bookmark1288)–[376](#_bookmark1294)

Mẫu ghi nhật ký kiểm tra[377](#_bookmark1302)–[378](#_bookmark1309) thêm mã vào logic kinh doanh[378](#_bookmark1306) hướng theo khía cạnhlập trình[378](#_bookmark1307)

sự kiện tìm nguồn[378](#_bookmark1308)

Mẫu theo dõi phân tán[370](#_bookmark1274)–[373](#_bookmark1280) máy chủ theo dõi phân tán[373](#_bookmark1279) thư viện thiết bị[373](#_bookmark1276)

Ngoại lệmẫu theo dõi[376](#_bookmark1297)–[377](#_bookmark1303) Mẫu API kiểm tra sức khỏe[366](#_bookmark1248)–[368](#_bookmark1254)

thực hiện điểm cuối[367](#_bookmark1251)–[368](#_bookmark1252)

gọi điểm cuối[368](#_bookmark1253)

Tổng hợp nhật kýmẫu[368](#_bookmark1257)–[370](#_bookmark1271) thế hệ nhật ký[369](#_bookmark1260)–[370](#_bookmark1264)

tổng hợp ghi nhật kýcơ sở hạ tầng[370](#_bookmark1265) ủy quyền dựa trên ole[353](#_bookmark1201)

một kích thước phù hợp với tất cả (OSFA)[262](#_bookmark942)

tương tác một-nhiều [67](#_bookmark294)

tương tác một-một[67](#_bookmark293)

thông báo một chiều[68](#_bookmark297),[89](#_bookmark395)

API kiểu thông báo một chiều[90](#_bookmark405)

OOP (lập trình hướng đối tượng)[149](#_bookmark583) token mờ đục[356](#_bookmark1211)

Mở cửa[416](#_bookmark1459)

khóa lạc quan[193](#_bookmark708)–[194](#_bookmark710) Lạc quan Ngoại tuyếnMẫu khóa[131](#_bookmark547) sự phối hợp[111](#_bookmark480),[399](#_bookmark1397)

saga dựa trên dàn nhạc[121](#_bookmark518)–[125](#_bookmark526) lợi ích và hạn chế của[125](#_bookmark524) tạo ra[211](#_bookmark787)–[212](#_bookmark790)

thực hiện saga Create Order[122](#_bookmark519)–[123](#_bookmark520) triển khai sử dụng sự kiện nguồn[216](#_bookmark798)–[218](#_bookmark805) mô hình dàn dựng saga như nhà nước

máy móc[123](#_bookmark521)–[124](#_bookmark522) nhắn tin giao dịchVà[125](#_bookmark523)

Tổng hợp đơn hàng[175](#_bookmark664)–[180](#_bookmark675)

dựa trên sự kiện[191](#_bookmark702)–[193](#_bookmark704)

phương pháp[177](#_bookmark670)–[180](#_bookmark675)

máy trạng thái[176](#_bookmark667)–[177](#_bookmark671)

cấu trúc của[175](#_bookmark665)–[176](#_bookmark666)

Đặt hàng sự kiện miền, xuất bản và tiêu thụ[458](#_bookmark1587)–[459](#_bookmark1588)

Dịch vụ Lịch sử đơn hàng [329](#_bookmark1140)–[330](#_bookmark1141) Dịch vụ đặt hàng

logic kinh doanh[173](#_bookmark662)–[182](#_bookmark676)

Tổng hợp đơn hàng[175](#_bookmark664)–[180](#_bookmark675)

Lớp OrderService[180](#_bookmark674)–[182](#_bookmark676)

do người tiêu dùng thúc đẩykiểm tra tích hợp hợp đồng cho[324](#_bookmark1130)–[325](#_bookmark1132)

kiểm tra hợp đồng do người tiêu dùng thúc đẩy[328](#_bookmark1138)–[329](#_bookmark1139) Lớp OrderCommandHandlers[142](#_bookmark565)–[143](#_bookmark566)

Lớp OrderService[133](#_bookmark553)–[134](#_bookmark554)

Lớp OrderServiceConfiguration[143](#_bookmark568)–[145](#_bookmark569)

Lớp OrderCommandHandlers[142](#_bookmark565)–[143](#_bookmark566)

Lớp OrderConfiguration [275](#_bookmark1005)–[276](#_bookmark1008)

Sự kiện OrderCreated[327](#_bookmark1136)–[328](#_bookmark1137)

Trình xử lý yêu cầu chi tiết đơn hàng [352](#_bookmark1198)

Lớp OrderHandlers [276](#_bookmark1009)–[278](#_bookmark1011)

OrderHistoryDaoDynamoLớpDb[249](#_bookmark908)–[252](#_bookmark915)

phương thức addOrder()[249](#_bookmark909)–[250](#_bookmark910)

phương thức findOrderHistory()[251](#_bookmark913)–[252](#_bookmark915)

phương thức idempotentUpdate()[250](#_bookmark912)–[251](#_bookmark914)

phương thức notePickedUp()[250](#_bookmark911)

Mô-đun OrderHistoryEventHandlers[243](#_bookmark884)–[244](#_bookmark885)

Lớp OrderService[133](#_bookmark553)–[134](#_bookmark554),[180](#_bookmark674)–[182](#_bookmark676),[278](#_bookmark1012)–[279](#_bookmark1013) OrderServiceComponentTestStepDefinitions

lớp học[341](#_bookmark1165)–[344](#_bookmark1166)

Lớp OrderServiceConfiguration[143](#_bookmark568)–[145](#_bookmark569)

Dịch vụ đặt hàngProxy[325](#_bookmark1131)–[326](#_bookmark1133)

OSFA (một kích thước phù hợp với tất cả)[262](#_bookmark940)

bộ chuyển đổi đầu ra[3](#_bookmark16),[38](#_bookmark214),[147](#_bookmark577)

yêu cầu nổi bật[79](#_bookmark348)

P

tham số phân trang[229](#_bookmark844)

khóa phân vùng[246](#_bookmark891)

Khung hộ chiếu[351](#_bookmark1195) Phần PATCH, Semvers[70](#_bookmark309)

mẫu và ngôn ngữ mẫu[20](#_bookmark96)–[23](#_bookmark112) theo tên

Đăng ký bên thứ 3[85](#_bookmark371) Mã thông báo truy cập[354](#_bookmark1207)

Tổng hợp[150](#_bookmark588)

Lớp chống tham nhũng[447](#_bookmark1552)

Thành phần API[223](#_bookmark819)

Cổng API[259](#_bookmark933)

Số liệu ứng dụng[373](#_bookmark1285)

Nhật ký kiểm toán[377](#_bookmark1305) Phần cuốicho giao diện[265](#_bookmark961) Bộ ngắt mạch[78](#_bookmark344)

Khám phá phía máy khách[83](#_bookmark360) Trách nhiệm truy vấn lệnh

sự phân biệt[228](#_bookmark837)

Kiểm tra hợp đồng do người tiêu dùng thúc đẩy[302](#_bookmark1070) Kiểm tra hợp đồng phía người tiêu dùng[303](#_bookmark1071) Phân tích theo năng lực kinh doanh[51](#_bookmark245)

mẫu và ngôn ngữ mẫu*(tiếp theo)*Phân tích theo tên miền phụ[54](#_bookmark253) Triển khai dịch vụ dưới dạng một container[393](#_bookmark1370) Triển khai dịch vụ dưới dạng VM[390](#_bookmark1353) Phân phối theo dõi[370](#_bookmark1275)

Sự kiện miền[160](#_bookmark622)

Mô hình miền[150](#_bookmark588)

Nguồn sự kiện [184](#_bookmark678)

Theo dõi ngoại lệ[376](#_bookmark1295)

Cấu hình bên ngoài[361](#_bookmark1225) API kiểm tra sức khỏe[366](#_bookmark1249)

Định dạng đóng gói theo ngôn ngữ cụ thể[387](#_bookmark1336) Tổng hợp nhật ký[368](#_bookmark1255)

Nhắn tin[85](#_bookmark369)

Kiến trúc vi dịch vụ[40](#_bookmark220)

Khung gầm vi dịch vụ[379](#_bookmark1314)

Kiến trúc nguyên khối[40](#_bookmark220)

Nhà xuất bản thăm dò ý kiến[98](#_bookmark439)

Gọi thủ tục từ xa[72](#_bookmark322) Truyện dài[114](#_bookmark496)

Tự đăng ký[82](#_bookmark358)

Triển khai không cần máy chủ [416](#_bookmark1456)

Khám phá phía máy chủ[85](#_bookmark371) Kiểm tra thành phần dịch vụ[335](#_bookmark1150) Lưới dịch vụ[380](#_bookmark1321)

Xe đẩy có thùng[410](#_bookmark1434)

Ứng dụng bóp cổ[432](#_bookmark1518) Giao dịchkhai thác gỗ[99](#_bookmark445) Kịch bản giao dịch[149](#_bookmark580)

Hộp thư đi giao dịch[98](#_bookmark439) nhóm các mẫu[23](#_bookmark114)–[29](#_bookmark157)

mẫu giao tiếp[24](#_bookmark123)–[25](#_bookmark125) mẫu thống nhất dữ liệu[25](#_bookmark126)

để thử nghiệm tự động các dịch vụ[28](#_bookmark143) để phân hủy các ứng dụng thành

dịch vụ[24](#_bookmark119)

để xử lýmối quan tâm xuyên suốt[28](#_bookmark151) để truy vấn dữ liệu[25](#_bookmark128)–[26](#_bookmark131)

mẫu quan sát được[27](#_bookmark136)–[28](#_bookmark145)

mẫu bảo mật[28](#_bookmark154)–[29](#_bookmark157)

triển khai dịch vụmẫu hình[26](#_bookmark133) các phần của mẫu

lực lượng[21](#_bookmark101)

mẫu liên quan[21](#_bookmark103)–[23](#_bookmark112)

ngữ cảnh kết quả[21](#_bookmark102)

trạng thái chờ xử lý[176](#_bookmark668) sự kiên trì

duy trì tổng hợp bằng cách sử dụng các sự kiện[186](#_bookmark693)–[188](#_bookmark698) cách tiếp cận truyền thống[185](#_bookmark683)–[186](#_bookmark690)

ghi nhật ký kiểm toán[186](#_bookmark688)

sự kiện xuất bản được gắn chặt vào logic kinh doanh[186](#_bookmark689)

thiếu lịch sử tổng hợp[186](#_bookmark687)

trở kháng quan hệ đối tượngkhông phù hợp[185](#_bookmark685)–[186](#_bookmark686)

kiểm tra tích hợp tính bền bỉ[321](#_bookmark1125)–[322](#_bookmark1126) Lớp bền bỉ[38](#_bookmark207)

biện pháp đối phó quan điểm bi quan[130](#_bookmark544)–[131](#_bookmark545) hành động nhặt[460](#_bookmark1594)

Phi công[409](#_bookmark1432)

giao dịch trục[128](#_bookmark538),[450](#_bookmark1564)

vỏ quả[402](#_bookmark1403)

kênh điểm-điểm[87](#_bookmark385)

chính sáchthực thi[409](#_bookmark1430)

thăm dò[194](#_bookmark713)–[195](#_bookmark716)

Mẫu nhà xuất bản thăm dò[98](#_bookmark442)–[99](#_bookmark443) cổng[38](#_bookmark215)

giai đoạn kiểm tra trước khi cam kết[306](#_bookmark1086) mẫu tiền nhiệm[21](#_bookmark104)

Lớp trình bày[38](#_bookmark205)

logic trình bày [436](#_bookmark1531) truy vấn dựa trên khóa chính[235](#_bookmark861) Quá trình xem[36](#_bookmark194)

phương thức process()[190](#_bookmark701),[193](#_bookmark706) phát triển dịch vụ sẵn sàng sản xuất

[348](#_bookmark1180)–[382](#_bookmark1330)

dịch vụ có thể cấu hình[360](#_bookmark1224)–[364](#_bookmark1237)

cấu hình bên ngoài dựa trên kéo[363](#_bookmark1235)–[364](#_bookmark1237)

đẩy dựa trên bên ngoàicấu hình[362](#_bookmark1229)–[363](#_bookmark1233)

Mẫu khung gầm dịch vụ vi mô[378](#_bookmark1313)–[382](#_bookmark1329) lưới dịch vụ[380](#_bookmark1323)–[382](#_bookmark1329)

sử dụng[379](#_bookmark1316)–[380](#_bookmark1320)

dịch vụ có thể quan sát được[364](#_bookmark1240)–[378](#_bookmark1309) Mẫu số liệu ứng dụng[373](#_bookmark1284)–[376](#_bookmark1294) Mẫu ghi nhật ký kiểm tra[377](#_bookmark1302)–[378](#_bookmark1309) Mẫu theo dõi phân tán[370](#_bookmark1274)–[373](#_bookmark1280) Mẫu theo dõi ngoại lệ[376](#_bookmark1297)–[377](#_bookmark1303) Mẫu API kiểm tra sức khỏe[366](#_bookmark1248)–[368](#_bookmark1254) Mẫu tổng hợp nhật ký[368](#_bookmark1257)–[370](#_bookmark1271)

dịch vụ an toàn[349](#_bookmark1185)–[360](#_bookmark1222)

xử lý xác thực trong APIcổng vào[354](#_bookmark1206)–[355](#_bookmark1208)

xử lý ủy quyền[356](#_bookmark1209)

trong ứng dụng đơn khối truyền thống[350](#_bookmark1192)–[353](#_bookmark1202)

sử dụng JWT để truyền danh tính và vai trò của người dùng[356](#_bookmark1210)–[357](#_bookmark1213)

sử dụng OAuth 2.0[357](#_bookmark1214)–[360](#_bookmark1222)

Prometheus[375](#_bookmark1293)

của cải,sơ đồ dựa trên đồ thị[280](#_bookmark1020) Bộ đệm giao thức[72](#_bookmark321)

nhà cung cấp dịch vụ[223](#_bookmark818)

lớp proxy[274](#_bookmark1004)

giao diện proxy[72](#_bookmark326)

bí danh hóa[201](#_bookmark748) Mô-đun API công khai[264](#_bookmark954) phương thức publish()[166](#_bookmark643)

xuất bản/phản hồi không đồng bộ[89](#_bookmark397)

xuất bản/đăng ký theo phong cáchtương tác thực hiện[89](#_bookmark396)

kiểm tra tích hợpvì[326](#_bookmark1135)–[330](#_bookmark1141)

hợp đồng xuất bản sự kiện OrderCreated[327](#_bookmark1136)–[328](#_bookmark1137)

kiểm tra cho Dịch vụ Lịch sử Đơn hàng [329](#_bookmark1140)–[330](#_bookmark1141) kiểm tra cho Dịch vụ Đặt hàng[328](#_bookmark1138)–[329](#_bookmark1139)

kênh đăng ký-đăng ký[87](#_bookmark386)

mô hình kéo của cấu hình bên ngoài[361](#_bookmark1227),[375](#_bookmark1290) mô hình đẩy của cấu hình bên ngoài[361](#_bookmark1226),

[375](#_bookmark1291)

Hỏi

thuộc tính chất lượng[8](#_bookmark47),[34](#_bookmark185),[37](#_bookmark200)

chất lượng dịch vụ[8](#_bookmark48),[37](#_bookmark201)

truy vấn[41](#_bookmark224)

đối số truy vấn[286](#_bookmark1030)

hoạt động query()[246](#_bookmark892),[249](#_bookmark905)

mẫu truy vấn [220](#_bookmark808)–[252](#_bookmark916)

Mẫu thành phần API[26](#_bookmark132),[64](#_bookmark285),[79](#_bookmark350),[221](#_bookmark811)–[228](#_bookmark836) lợi ích và hạn chế của[227](#_bookmark831)–[228](#_bookmark836)

vấn đề thiết kế[225](#_bookmark824)–[227](#_bookmark828)

hoạt động truy vấn findOrder()[221](#_bookmark814)–[222](#_bookmark817),[224](#_bookmark822)

tổng quan về[222](#_bookmark816)–[224](#_bookmark820)

Mẫu CQRS [26](#_bookmark130),[63](#_bookmark282),[160](#_bookmark622),[184](#_bookmark678),[221](#_bookmark809),[228](#_bookmark839)–[236](#_bookmark868)

lợi ích của[235](#_bookmark857)–[236](#_bookmark863)

nhược điểm của[236](#_bookmark865)

động lực để sử dụng[229](#_bookmark841)–[232](#_bookmark849) tổng quan về[232](#_bookmark851)–[235](#_bookmark858)

R

ThỏMQ[92](#_bookmark415)

giới hạn tỷ lệ[262](#_bookmark946)

Kho lưu trữ sự kiện dựa trên RDBMS

tạo saga orchestrator khi sử dụng[211](#_bookmark788) tin nhắn có giá trị bất biếnxử lý với[197](#_bookmark723)

mô hình lập trình phản ứng[227](#_bookmark827) sẵn sàngProbe[404](#_bookmark1408),[407](#_bookmark1422) giao diện cổng nhận [86](#_bookmark383) giảm hoạt động[187](#_bookmark695)

tái cấu trúc[428](#_bookmark1505)–[471](#_bookmark1624)

ứng dụnghiện đại hóa[430](#_bookmark1513)–[432](#_bookmark1517)

thể hiện giá trị[432](#_bookmark1519)

thiết kế dịch vụ như thế nàovà khối hợp tác[443](#_bookmark1542)–[455](#_bookmark1577)

xác thực và ủy quyền[453](#_bookmark1573)–[455](#_bookmark1577) tính nhất quán của dữ liệu[449](#_bookmark1558)–[453](#_bookmark1571)

keo dán tích hợp[444](#_bookmark1544)–[449](#_bookmark1556)

trích xuất giao hàngsự quản lý[459](#_bookmark1590)–[470](#_bookmark1623) thay đổi khối FTGO để tương tác với

Dịch vụ giao hàng[467](#_bookmark1615)–[470](#_bookmark1623) thiết kế miền dịch vụ giao hàng

người mẫu[463](#_bookmark1602)–[464](#_bookmark1606)

thiết kế keo tích hợp dịch vụ giao hàng[465](#_bookmark1608)–[467](#_bookmark1613)

chức năng giao hàng hiện có[460](#_bookmark1592)–[461](#_bookmark1597) Tổng quan về Giao hàngDịch vụ[462](#_bookmark1599)–[463](#_bookmark1600)

triển khai các tính năng mới như dịch vụ[455](#_bookmark1579)–[459](#_bookmark1588) thiết kế cho Dịch vụ giao hàng chậm trễ[456](#_bookmark1581)–[457](#_bookmark1583) keo tích hợp cho Giao hàng chậm trễ

Dịch vụ[457](#_bookmark1585)–[459](#_bookmark1588)

giảm thiểu những thay đổi[432](#_bookmark1520)–[433](#_bookmark1521)

tổng quan về[429](#_bookmark1507)–[433](#_bookmark1523)

lý do cho [429](#_bookmark1509)–[430](#_bookmark1511)

chiến lược cho[433](#_bookmark1525)–[442](#_bookmark1540)

trích xuất khả năng kinh doanh thành các dịch vụ[437](#_bookmark1536)–[442](#_bookmark1540)

triển khai các tính năng mới như dịch vụ[434](#_bookmark1527)–[435](#_bookmark1528)

tách tầng trình bày khỏi tầng phụ trợ[436](#_bookmark1530)–[437](#_bookmark1534)

cơ sở hạ tầng triển khai kỹ thuật[433](#_bookmark1522) Tái cấu trúc thành các mẫu dịch vụ vi mô

Lớp chống tham nhũng[446](#_bookmark1549)–[447](#_bookmark1552)

Ứng dụng bóp cổ[431](#_bookmark1515)–[432](#_bookmark1518) ReflectiveMutableCommandProcessingAggregate

lớp học[206](#_bookmark770)–[207](#_bookmark771)

Khái niệm mã thông báo làm mới[358](#_bookmark1219) Dịch vụ phát hành[408](#_bookmark1426)

Mô hình truyền thông đáng tin cậy Bộ ngắt mạch[77](#_bookmark343)–[80](#_bookmark353),[108](#_bookmark468)

Mẫu gọi thủ tục từ xa (RPI)[72](#_bookmark325)–[85](#_bookmark370)

Mẫu cầu dao điện[77](#_bookmark342)–[80](#_bookmark352) phát triển các proxy RPI mạnh mẽ[79](#_bookmark345)

phục hồi từ các dịch vụ không khả dụng[79](#_bookmark351)–[80](#_bookmark352) gRPC[76](#_bookmark336)–[77](#_bookmark340)

NGHỈ NGƠI[73](#_bookmark328)–[76](#_bookmark338)

lợi ích và hạn chế của[75](#_bookmark334)–[76](#_bookmark338) lấy nhiều tài nguyên trong một

lời yêu cầu[74](#_bookmark331)–[75](#_bookmark332)

hoạt động ánh xạ tới các động từ HTTP[75](#_bookmark333) Mô hình trưởng thành REST[74](#_bookmark329)

chỉ định REST API[74](#_bookmark330) khám phá dịch vụ[80](#_bookmark355)–[85](#_bookmark370)

tổng quan về[81](#_bookmark356)

sử dụng các mẫu khám phá dịch vụ cấp ứng dụng[81](#_bookmark357)–[83](#_bookmark362)

sử dụng các mẫu khám phá dịch vụ do nền tảng cung cấp[83](#_bookmark363)–[85](#_bookmark370)

tiêu đề kênh trả lời[88](#_bookmark393)–[89](#_bookmark399) Đối tượng lưu trữ, DDD[152](#_bookmark596) yêu cầu thuộc tính[10](#_bookmark53)

yêu cầu ghi nhật ký[262](#_bookmark948)

API theo kiểu phản hồi yêu cầu/bất đồng bộ[90](#_bookmark404) tương tác yêu cầu/phản hồi[87](#_bookmark390)–[89](#_bookmark397)

không đồng bộ[87](#_bookmark390)–[88](#_bookmark394)

kiểm tra tích hợp cho REST dựa trên[322](#_bookmark1128)–[326](#_bookmark1133)

Giao diện RequestHandler[417](#_bookmark1467) đọc lại giá trị biện pháp đối phó[131](#_bookmark546) Khái niệm máy chủ tài nguyên[358](#_bookmark1220) NGHỈ NGƠI[73](#_bookmark328)–[76](#_bookmark338)

lợi ích và hạn chế của[75](#_bookmark334)–[76](#_bookmark338) lấy nhiều tài nguyên trong một

lời yêu cầu[74](#_bookmark331)–[75](#_bookmark332)

hoạt động ánh xạ tới các động từ HTTP[75](#_bookmark333) Mô hình trưởng thành REST[74](#_bookmark329)

chỉ định REST API[74](#_bookmark330) Hãy yên tâmGiả lập MVC[314](#_bookmark1116)

Sự kiện miền nhà hàng[458](#_bookmark1587)–[459](#_bookmark1588) Dịch vụ nhà hàng

tạo ra các dịch vụ[404](#_bookmark1412)–[405](#_bookmark1413)

triển khai[402](#_bookmark1407)–[405](#_bookmark1413)

thiết kế của[419](#_bookmark1485)–[423](#_bookmark1497) Tóm tắtAutowiringHttpRequestHandler

lớp học[423](#_bookmark1494)

Lớp AbstractHttpHandler[423](#_bookmark1496) Lớp FindRestaurantRequestHandler

[421](#_bookmark1486)–[422](#_bookmark1493)

Tương tác theo phong cách yêu cầu/phản hồi dựa trên REST, các bài kiểm tra tích hợp cho[322](#_bookmark1128)–[326](#_bookmark1133)

hợp đồng mẫu[324](#_bookmark1129)

kiểm tra cho API gateway OrderServiceProxy[325](#_bookmark1131)–[326](#_bookmark1133)

kiểm tra cho Dịch vụ Đặt hàng[324](#_bookmark1130)–[325](#_bookmark1132) Dịch vụ RESTful[419](#_bookmark1483)–[426](#_bookmark1502)

triển khai các hàm lambda bằng cách sử dụng khung Serverless[425](#_bookmark1501)–[426](#_bookmark1502)

thiết kế dịch vụ nhà hàng[419](#_bookmark1485)–[423](#_bookmark1497) dịch vụ đóng gói dưới dạng tệp ZIP[424](#_bookmark1499)

giao dịch có thể truy xuất lại[117](#_bookmark501),[129](#_bookmark539),[450](#_bookmark1565)

phương thức revision()[179](#_bookmark672)

S

Saas (Phần mềm dưới dạng dịch vụ) [5](#_bookmark29) gói dàn dựng saga[140](#_bookmark562) Mẫu Saga[26](#_bookmark134)

SagaOrchestratorĐã tạosự kiện[216](#_bookmark800)

SagaOrchestratorSự kiện đã cập nhật[216](#_bookmark801) Sự kiện giả SagaReplyRequested[213](#_bookmark794) truyện dài[17](#_bookmark85),[58](#_bookmark273),[106](#_bookmark467),[110](#_bookmark475)–[145](#_bookmark570),[209](#_bookmark783)–[218](#_bookmark805),[450](#_bookmark1559)

phối hợp[117](#_bookmark505)–[125](#_bookmark526)

saga dựa trên vũ đạo[118](#_bookmark507)–[121](#_bookmark516)

saga dựa trên dàn nhạc[121](#_bookmark518)–[125](#_bookmark526) Tạo đơn hàng saga[135](#_bookmark556)–[142](#_bookmark563)

Tạo đơn hàngSagangười điều phối[136](#_bookmark557)–[138](#_bookmark558)

Lớp CreateOrderSagaState [138](#_bookmark559) Khung của Eventuate Tram Saga[140](#_bookmark561)–[142](#_bookmark563) Lớp KitchenServiceProxy[139](#_bookmark560)

tạo ra saga dựa trên dàn nhạc[211](#_bookmark787)–[212](#_bookmark790) với kho lưu trữ sự kiện dựa trên NoSQL[211](#_bookmark789)–[212](#_bookmark790) với kho sự kiện dựa trên RDBMS[211](#_bookmark788)

thực hiện các saga dựa trên vũ đạo bằng cách sử dụng nguồn sự kiện[210](#_bookmark785)

triển khai người tham gia saga dựa trên sự kiện[213](#_bookmark792)–[216](#_bookmark796)

thực hiệnsaga orchestrators sử dụng event sourcing[216](#_bookmark798)–[218](#_bookmark805)

duy trì sử dụng sự kiệnnguồn cung ứng[216](#_bookmark799) xử lý trả lời chính xác một lần[218](#_bookmark804) gửi tin nhắn lệnh một cách đáng tin cậy

[216](#_bookmark802)–[218](#_bookmark803)

thiếu sự cô lập[126](#_bookmark528)–[132](#_bookmark550) những bất thường gây ra bởi[127](#_bookmark532)

biện pháp đối phó để xử lý[128](#_bookmark536)–[132](#_bookmark550) Dịch vụ đặt hàng

Lớp OrderCommandHandlers[142](#_bookmark565)–[143](#_bookmark566)

Lớp OrderService[133](#_bookmark553)–[134](#_bookmark554)

Lớp OrderServiceConfiguration[143](#_bookmark568)–[145](#_bookmark569)

quản lý giao dịch[111](#_bookmark482)–[117](#_bookmark502) duy trì tính nhất quán của dữ liệu[114](#_bookmark495)–[117](#_bookmark502) nhu cầu giao dịch phân tán[112](#_bookmark485)

sự cố với các giao dịch phân tán[112](#_bookmark487)–[114](#_bookmark493) các bài kiểm tra đơn vị cho[310](#_bookmark1105)–[312](#_bookmark1107)

Hội nghị SATURN [34](#_bookmark189)

phương thức save()[207](#_bookmark775)

khả năng mở rộng[430](#_bookmark1510)

khối lập phương[8](#_bookmark50)–[11](#_bookmark59)

Tỷ lệ trục X[9](#_bookmark51)

Tỷ lệ trục Y[10](#_bookmark55)–[11](#_bookmark59)

Tỷ lệ trục Z[9](#_bookmark52)–[10](#_bookmark54)

dịch vụ an toàn[349](#_bookmark1185)–[360](#_bookmark1222)

xác thực trong cổng API[354](#_bookmark1206)–[355](#_bookmark1208) sự cho phép[356](#_bookmark1209)

trong ứng dụng đơn khối truyền thống[350](#_bookmark1192)–[353](#_bookmark1202) sử dụng JWT để truyền danh tính và vai trò của người dùng

[356](#_bookmark1210)–[357](#_bookmark1213)

sử dụng OAuth 2.0[357](#_bookmark1214)–[360](#_bookmark1222)

mẫu bảo mật[28](#_bookmark154)–[29](#_bookmark157)

Mã thông báo truy cập[28](#_bookmark146),[38](#_bookmark208),[354](#_bookmark1207)

Các câu lệnh SELECT[188](#_bookmark696) Mẫu tự đăng ký[82](#_bookmark358) khóa ngữ nghĩa[450](#_bookmark1562)

biện pháp đối phó khóa ngữ nghĩa[129](#_bookmark540)–[130](#_bookmark542) giao diện cổng gửi[86](#_bookmark384)

Triển khai không cần máy chủ với lambda[415](#_bookmark1452)–[419](#_bookmark1480) lợi ích của hàm lambda[418](#_bookmark1476) phát triển các hàm lambda[417](#_bookmark1463) nhược điểm của hàm lambda[419](#_bookmark1478) gọi hàm lambda[417](#_bookmark1469)–[418](#_bookmark1474)

định nghĩa các hàm lambda theo lịch trình[418](#_bookmark1472)

xử lý các sự kiện được tạo ra bởi các dịch vụ AWS[418](#_bookmark1471)

xử lý các yêu cầu HTTP[417](#_bookmark1470) sử dụng yêu cầu dịch vụ web[418](#_bookmark1473)

tổng quan về[416](#_bookmark1455)

Khung không có máy chủ[425](#_bookmark1501)–[426](#_bookmark1502) mẫu khám phá phía máy chủ[84](#_bookmark367)–[85](#_bookmark368) định nghĩa API dịch vụ[61](#_bookmark277)–[64](#_bookmark284)

chỉ định các hoạt động hệ thống cho các dịch vụ[61](#_bookmark279)–[62](#_bookmark280)

xác định các API cần thiết để hỗ trợ sự cộng tác giữa các dịch vụ[62](#_bookmark281)–[64](#_bookmark284)

Dịch vụ như một mẫu container[393](#_bookmark1369)–[399](#_bookmark1392) lợi ích của[398](#_bookmark1389)

Người lái tàu[395](#_bookmark1372)–[398](#_bookmark1387)

xây dựng hình ảnh Docker[395](#_bookmark1375)–[396](#_bookmark1378) đẩy hình ảnh Docker vào sổ đăng ký

[396](#_bookmark1379)–[397](#_bookmark1383)

chạy container Docker[397](#_bookmark1384)–[398](#_bookmark1387) nhược điểm của[399](#_bookmark1391)

Dịch vụ như một mô hình máy ảo[390](#_bookmark1350)–[393](#_bookmark1367)

lợi ích của[392](#_bookmark1357)

cơ sở hạ tầng đám mây trưởng thành[392](#_bookmark1361) các trường hợp dịch vụ bị cô lập[392](#_bookmark1360) Hình ảnh VM đóng gói công nghệ

chồng lên nhau[392](#_bookmark1359)

nhược điểm của[392](#_bookmark1363)–[393](#_bookmark1367)

sử dụng tài nguyên kém hiệu quả[393](#_bookmark1364) triển khai tương đối chậm[393](#_bookmark1365) chi phí quản lý hệ thống[393](#_bookmark1366)

kiểm tra thành phần dịch vụ[28](#_bookmark148),[335](#_bookmark1150)

khả năng cấu hình dịch vụ[349](#_bookmark1183)

định nghĩa dịch vụ[76](#_bookmark339)

Phân tích theo mô hình năng lực kinh doanh[51](#_bookmark242)–[54](#_bookmark249)

sự phân hủy [52](#_bookmark247)–[54](#_bookmark249)

xác định năng lực kinh doanh[51](#_bookmark246)–[52](#_bookmark248) mục đích của năng lực kinh doanh[51](#_bookmark244)

Phân tích theo miền phụmẫu[54](#_bookmark252)–[55](#_bookmark257)

triển khai dịch vụmẫu hình[26](#_bookmark133) khám phá dịch vụ[80](#_bookmark355)–[85](#_bookmark370)

Đăng ký bên thứ 3[84](#_bookmark365)–[85](#_bookmark371),[108](#_bookmark468)

Khám phá phía máy khách[82](#_bookmark358)–[83](#_bookmark360)

tổng quan về[81](#_bookmark356)

Tự đăng ký[82](#_bookmark358)

Khám phá phía máy chủ[84](#_bookmark365)–[85](#_bookmark371)

lưới dịch vụ[380](#_bookmark1323)–[382](#_bookmark1329),[407](#_bookmark1425)–[415](#_bookmark1450) triển khai v2 của Consumer Service[414](#_bookmark1446) Istio[408](#_bookmark1427)–[412](#_bookmark1442)

định tuyến lưu lượng sản xuất đến v2[415](#_bookmark1449) quy tắc định tuyến để định tuyến đến phiên bản v1

[412](#_bookmark1441)–[413](#_bookmark1445)

định tuyến kiểm tra lưu lượng truy cậpđến v2[414](#_bookmark1447) Đối tượng dịch vụ, DDD[152](#_bookmark597) phương thức service()[422](#_bookmark1492)

kiến trúc hướng dịch vụ(SOA)[13](#_bookmark67)–[14](#_bookmark71)

SES (Dịch vụ Email đơn giản)[2](#_bookmark8) SessionBasedSecurityInterceptor[352](#_bookmark1199)

phiên họp[351](#_bookmark1197)

phương thức setUp()[313](#_bookmark1111)

kênh bị phân mảnh[94](#_bookmark426)

Shiro[351](#_bookmark1193)

Mẫu xe ba bánh[410](#_bookmark1435)

Dịch vụ Email đơn giản (SES)[2](#_bookmark9) Lớp duy trì đơn[38](#_bookmark209) Lớp trình bày đơn[38](#_bookmark210)

Nguyên tắc trách nhiệm duy nhất (SRP) [56](#_bookmark262) ống thông minh[14](#_bookmark70)

ảnh chụp nhanh[195](#_bookmark718)–[196](#_bookmark720),[201](#_bookmark744)

SOA (kiến trúc hướng dịch vụ)[13](#_bookmark68) bài kiểm tra đơn vị xã hội[308](#_bookmark1094)

kiến trúc phần mềm[34](#_bookmark186)–[37](#_bookmark199)

Mô hình xem 4+1 của[35](#_bookmark191)–[37](#_bookmark196)

định nghĩa của[35](#_bookmark190)

sự liên quan của [37](#_bookmark197)

mẫu phần mềm[20](#_bookmark97)

Phần mềm dưới dạng dịch vụ (SaaS)[5](#_bookmark30) kiểm tra đơn vị đơn độc[308](#_bookmark1093) SoundCloud[265](#_bookmark962)

mẫu chuyên môn hóa[22](#_bookmark110)

Hợp đồng Spring Cloud[303](#_bookmark1073)–[305](#_bookmark1076) Cổng đám mây mùa xuân[273](#_bookmark1002)–[275](#_bookmark1006)

Ứng dụng ApiGatewaylớp học[279](#_bookmark1014)

Lớp OrderConfiguration [275](#_bookmark1005)–[276](#_bookmark1008)

Lớp OrderHandlers [276](#_bookmark1009)–[278](#_bookmark1011)

Lớp OrderService[278](#_bookmark1012)–[279](#_bookmark1013) Spring Mock MVC[314](#_bookmark1115) Bảo mật mùa xuân[351](#_bookmark1194)

Biến SPRING\_APPLICATION\_JSON[363](#_bookmark1231)

SQL[237](#_bookmark872)–[238](#_bookmark873)

SRP (Nguyên tắc trách nhiệm duy nhất) [56](#_bookmark263) máy trạng thái

người dàn dựng mô hình sagaBẰNG[123](#_bookmark521)–[124](#_bookmark522)

Tổng hợp đơn hàng[176](#_bookmark667)–[177](#_bookmark671)

Mẫu ứng dụng Strangler[431](#_bookmark1516)–[432](#_bookmark1518) Mẫu chiến lược[20](#_bookmark98)

gốc[296](#_bookmark1056),[339](#_bookmark1161)–[340](#_bookmark1162)

mẫu kế thừa[21](#_bookmark105)

SUT (hệ thống đang thử nghiệm)[294](#_bookmark1051) mô hình I/O đồng bộ[268](#_bookmark975) tương tác đồng bộ[67](#_bookmark295)

hoạt động hệ thống[45](#_bookmark234) giao cho các dịch vụ[61](#_bookmark279)–[62](#_bookmark280)

tạo mô hình miền cấp cao[46](#_bookmark237)–[48](#_bookmark238) xác định[48](#_bookmark239)–[50](#_bookmark240)

xác định[45](#_bookmark236)–[50](#_bookmark240)

hệ thống đang được thử nghiệm (SUT)[294](#_bookmark1052) Phương thức System.getenv()[362](#_bookmark1230)

T

đo từ xa [409](#_bookmark1429)

trường hợp thử nghiệm[294](#_bookmark1050)

kiểm tra gấp đôi[296](#_bookmark1057)

kim tự tháp thử nghiệm[298](#_bookmark1062)–[299](#_bookmark1063)

góc phần tư thử nghiệm[297](#_bookmark1060)–[298](#_bookmark1061)

@Test nên tính tổng()phương pháp[309](#_bookmark1098) Phương thức @Test shouldCreateOrder()[312](#_bookmark1108) bộ kiểm tra[294](#_bookmark1053)

thử nghiệm[292](#_bookmark1044)–[347](#_bookmark1178)

kiểm tra chấp nhận[335](#_bookmark1149)–[338](#_bookmark1156)

xác định[336](#_bookmark1152)

viết bằng Gherkin[337](#_bookmark1154)–[338](#_bookmark1156) thách thức của[299](#_bookmark1065)–[305](#_bookmark1078)

thử nghiệm hợp đồng tiêu dùng[301](#_bookmark1066)–[303](#_bookmark1072) người tiêu dùngthử nghiệm hợp đồng cho tin nhắn

API[305](#_bookmark1077)

Hợp đồng Spring Cloud[303](#_bookmark1073)–[305](#_bookmark1076) kiểm tra thành phần[339](#_bookmark1158)–[340](#_bookmark1162)

cho Dịch vụ Đặt hàng FTGO[340](#_bookmark1164)–[345](#_bookmark1168) kiểm tra thành phần trong quá trình[339](#_bookmark1159) kiểm tra thành phần ngoài quy trình

[339](#_bookmark1160)–[340](#_bookmark1162)

Do người tiêu dùng thúc đẩykiểm tra hợp đồng[28](#_bookmark146),[301](#_bookmark1068)–[302](#_bookmark1070)

Hợp đồng phía người tiêu dùngBài kiểm tra[28](#_bookmark149),[303](#_bookmark1071)

triển khaiđường ống[305](#_bookmark1081)–[307](#_bookmark1090)

kiểm tra đầu cuối[345](#_bookmark1170)–[346](#_bookmark1177)

thiết kế[345](#_bookmark1172)

đang chạy[346](#_bookmark1176)

viết[346](#_bookmark1174)

kiểm tra tích hợp[319](#_bookmark1123)–[335](#_bookmark1147)

kiểm tra hợp đồng cho tương tác yêu cầu/phản hồi không đồng bộ[330](#_bookmark1143)–[335](#_bookmark1147)

kiểm tra tích hợp tính bền bỉ[321](#_bookmark1125)–[322](#_bookmark1126) xuất bản/đăng ký theo phong cáchtương tác

[326](#_bookmark1135)–[330](#_bookmark1141)

Tương tác theo kiểu yêu cầu/phản hồi dựa trên REST[322](#_bookmark1128)–[326](#_bookmark1133)

tổng quan về[294](#_bookmark1049)–[299](#_bookmark1063)

kiểm tra tự động[295](#_bookmark1054)–[296](#_bookmark1055) các loại thử nghiệm khác nhau[297](#_bookmark1058) bản nháp và bản nháp[296](#_bookmark1056)

kim tự tháp thử nghiệm[298](#_bookmark1062)–[299](#_bookmark1063)

góc phần tư thử nghiệm[297](#_bookmark1060)–[298](#_bookmark1061)

Kiểm tra thành phần dịch vụ[28](#_bookmark146),[335](#_bookmark1150)

kiểm tra đơn vị[307](#_bookmark1092)–[317](#_bookmark1120)

cho bộ điều khiển[313](#_bookmark1114)–[315](#_bookmark1117)

cho các dịch vụ tên miền[312](#_bookmark1110)–[313](#_bookmark1112) cho các thực thể[309](#_bookmark1096)–[310](#_bookmark1099)

cho trình xử lý sự kiện và tin nhắn[315](#_bookmark1119)–[317](#_bookmark1120) cho saga[310](#_bookmark1105)–[312](#_bookmark1107)

cho các đối tượng có giá trị[310](#_bookmark1101) tiêu đề testuser[414](#_bookmark1448)

định dạng tin nhắn dạng văn bản[71](#_bookmark315)–[72](#_bookmark318) Tổng hợp vé[169](#_bookmark651)–[173](#_bookmark660)

hành vi của[170](#_bookmark653)–[171](#_bookmark654)

Dịch vụ tên miền KitchenService[171](#_bookmark656)–[172](#_bookmark659) Lớp KitchenServiceCommandHandler

[172](#_bookmark658)–[173](#_bookmark660)

cấu trúc của lớp Ticket[170](#_bookmark652) sự kết nối chặt chẽ[121](#_bookmark515)

thời gian chờ[79](#_bookmark347)

TLS (Bảo mật lớp truyền tải)[350](#_bookmark1189) các mã thông báo[356](#_bookmark1211)

Traefik[272](#_bookmark998)

quản lý giao thông[408](#_bookmark1428)

theo dõi nhật ký giao dịch[99](#_bookmark444)–[100](#_bookmark450),[195](#_bookmark714)

quản lý giao dịch[111](#_bookmark482)–[117](#_bookmark502) duy trì tính nhất quán của dữ liệu[114](#_bookmark495)–[117](#_bookmark502) nhu cầu giao dịch phân tán[112](#_bookmark485) sự cố với các giao dịch phân tán

[112](#_bookmark487)–[114](#_bookmark493)

[*Nhìn thấy*](#_bookmark1559)[*Mà còn*](#_bookmark85)truyện dài

Mẫu kịch bản giao dịch[149](#_bookmark579)–[150](#_bookmark584) @Chú thích giao dịch[111](#_bookmark483)

nhắn tin giao dịch [97](#_bookmark435)–[100](#_bookmark450) Mẫu nhà xuất bản thăm dò[98](#_bookmark442)–[99](#_bookmark443)

Mẫu đuôi nhật ký giao dịch[99](#_bookmark444)–[100](#_bookmark450) Mẫu hộp thư đi giao dịch[97](#_bookmark436)–[98](#_bookmark441),[109](#_bookmark471) sử dụng bảng cơ sở dữ liệu như hàng đợi tin nhắn

[97](#_bookmark437)–[98](#_bookmark440)

token trong suốt[356](#_bookmark1212)

Bảo mật lớp truyền tải (TLS)[350](#_bookmark1190) cam kết hai giai đoạn (2PC)[112](#_bookmark491)

Bạn

Ngôn ngữ phổ biến[54](#_bookmark255)

kiểm tra đơn vị[307](#_bookmark1092)–[317](#_bookmark1120)

cho bộ điều khiển[313](#_bookmark1114)–[315](#_bookmark1117)

cho các dịch vụ tên miền[312](#_bookmark1110)–[313](#_bookmark1112) cho các thực thể[309](#_bookmark1096)–[310](#_bookmark1099)

cho trình xử lý sự kiện và tin nhắn[315](#_bookmark1119)–[317](#_bookmark1120) cho saga[310](#_bookmark1105)–[312](#_bookmark1107)

cho các đối tượng có giá trị[310](#_bookmark1101) đang lên tiếng[199](#_bookmark729)

Câu lệnh CẬP NHẬT[193](#_bookmark709)

phương thức update()[204](#_bookmark762),[207](#_bookmark776),[215](#_bookmark795)

Hoạt động UpdateItem()[248](#_bookmark901) Cookie USERINFO

LoginHandler và[454](#_bookmark1574)–[455](#_bookmark1575)

ánh xạ tới tiêu đề Authorization[455](#_bookmark1576)

V

Đối tượng giá trị, DDD[151](#_bookmark594)

các đối tượng giá trị,các bài kiểm tra đơn vị cho[310](#_bookmark1101) biện pháp đối phó tập tin phiên bản[131](#_bookmark548)

Địa chỉ VIP (IP ảo)[83](#_bookmark364) Dịch vụ ảo[413](#_bookmark1443)

VM (máy ảo)[26](#_bookmark135)

T

Tệp WAR (Lưu trữ ứng dụng web)[2](#_bookmark11) WebSocket[257](#_bookmark924)

**X**

tin nhắn XML[71](#_bookmark317)

Z

KhôngMQ[91](#_bookmark411)

Zipkin[373](#_bookmark1282)

**Quá trình:**

DevOps/giao hàng liên tục/triển khai

Cho phép

Cho phép

Nhanh chóng, thường xuyên,

và cung cấp phần mềm đáng tin cậy

**Tổ chức:**

Các nhóm nhỏ, tự chủ, đa chức năng

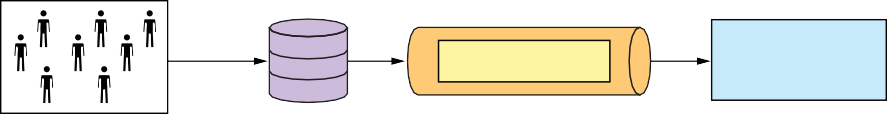
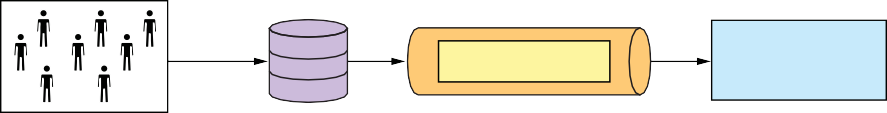
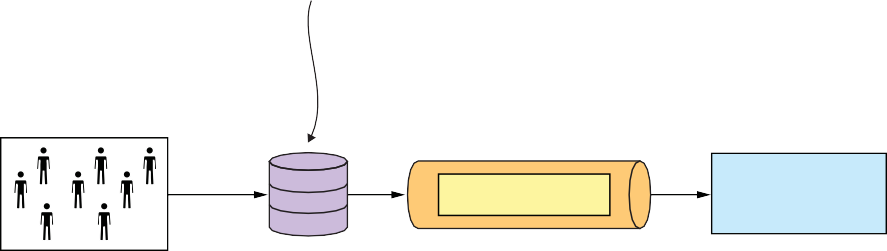
Cho phép

**Ngành kiến ​​​​trúc:**

Kiến trúc vi dịch vụ

**Việc cung cấp các ứng dụng lớn, phức tạp một cách nhanh chóng, thường xuyên và đáng tin cậyyêu cầu sự kết hợp của DevOps, bao gồm phân phối/triển khai liên tục, nhỏ,các nhóm tự chủ và kiến ​​trúc dịch vụ vi mô.**

**Các nhóm nhỏ, tự chủ, liên kết lỏng lẻo**



**Mỗi dịch vụ đều có kho mã nguồn riêng.**

**Mỗi dịch vụ đều có quy trình triển khai tự động riêng.**

**Dịch vụ nhỏ, đơn giản, đáng tin cậy, dễ bảo trì**

Phát triển FTGO

Đội ngũ quản lý đơn hàng

Đường ống triển khai

JenkinsDịch vụ ClOrder

Đội ngũ quản lý nhà hàng

Dịch vụ đặt hàngkho lưu trữ mã nguồn

Đường ống triển khai

JenkinsClDịch vụ nhà hàng

Đội ngũ quản lý giao hàng

Dịch vụ nhà hàngkho lưu trữ mã nguồn

Đường ống triển khai

JenkinsDịch vụ giao hàng

Dịch vụ giao hàngkho lưu trữ mã nguồn

Sản xuất

**Kiến trúc dịch vụ vi mô cấu trúc một ứng dụng như một tập hợp các dịch vụ được ghép nối lỏng lẻo được tổ chức xung quanh các khả năng kinh doanh. Mỗi nhóm phát triển, thử nghiệm và triển khai các dịch vụ của họđộc lập.**

PHÁT TRIỂN PHẦN MỀM

Các mẫu dịch vụ vi mô

Chris Richardson

S

phát triển thành công các ứng dụng dựa trên dịch vụ vi mô đòi hỏi phải nắm vững một tập hợp các hiểu biết và thực hành kiến ​​trúc mới. Trong cuốn sách độc đáo này, kiến ​​trúc dịch vụ vi mô

Người tiên phong và Nhà vô địch Java Chris Richardson thu thập, lập danh mục và giải thích 44 mẫu giải quyết các vấn đề như phân tích dịch vụ, quản lý giao dịch, truy vấn,và giao tiếp giữa các dịch vụ.

Các mẫu dịch vụ vi môdạy bạn cách phát triển và triển khai các ứng dụng dựa trên dịch vụ vi mô chất lượng sản xuất. Bộ mẫu thiết kế vô giá này được xây dựng dựa trên kinh nghiệm phân phối hệ thống trong nhiều thập kỷ, bổ sung các mẫu mới để viếtdịch vụ và kết hợp chúng thành các hệ thống có thể mở rộng và hoạt động đáng tin cậy trong điều kiện thực tế. Không chỉ là một danh mục mẫu, hướng dẫn thực tế này cung cấp lời khuyên dựa trên kinh nghiệm để giúp bạn thiết kế, triển khai, kiểm tra và triển khai ứng dụng dựa trên dịch vụ vi mô của mình.

Bên trong có gì

* Làm thế nào (và tại sao!) để sử dụng kiến ​​trúc vi dịch vụ
* Chiến lược phân tích dịch vụ
* Quản lý giao dịch và các mẫu truy vấn
* Hiệu quảchiến lược thử nghiệm
* Các mẫu triển khai bao gồm container và không có máy chủ

Được viết cho các nhà phát triển doanh nghiệp quen thuộc với tiêu chuẩn nhậpKiến trúc ứng dụng prise. Ví dụ có trong Java.

Chris Richardsonlà Nhà vô địch Java, một tảng đá JavaOne ngôi sao, tác giả của POJOs in Action của Manning và là người sáng tạo rabản gốc CloudFoundry.com.

Để tải xuống sách điện tử miễn phí của họ ở định dạng PDF, ePub và Kindle, chủ sở hữu của cuốn sách này nên truy cập manning.com/books/microservices-patterns

Xem trang đầu tiên

Tổng quan toàn diện về những thách thức mà các nhóm phải đối mặt

“

khi chuyển sang dịch vụ vi mô, với các giải pháp đã được thử nghiệm trong ngành

“

### đối với những vấn đề này.

—Tim Moore, Lightbend

### Xử lý thực dụng củamột cái mới quan trọng

“

“

cảnh quan kiến ​​trúc.

—Simeon Leyzerzon Phần mềm Excelsior

### Một tập hợp thông tin vững chắc sẽ giúp bạn nhanh chóng di chuyển đến thế giới hiện đại này

“

“

kiến trúc dựa trên đám mây.

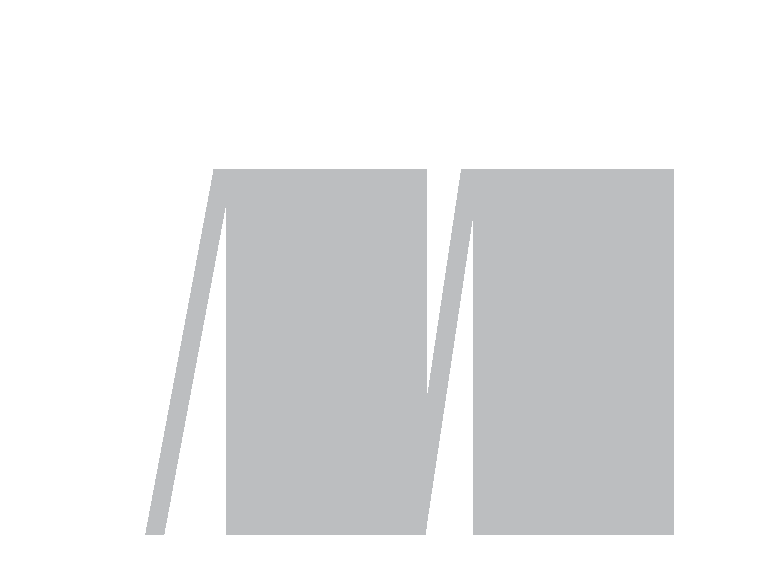
—John Guthrie, Dell/EMC

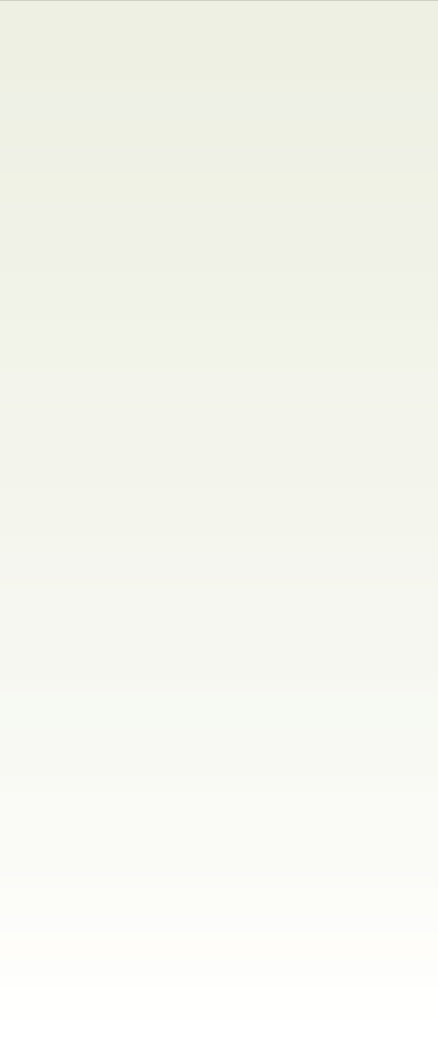
### Cách hiểu phương pháp tiếp cận dịch vụ vi mô và cách sử dụng nó trong thực tế.

“

“

—Potito Coluccelli Bizmatica Econocom

**NGƯỜI ĐÀN ÔNG**



49,99 đô la/lon 65,99 đô la[BAO GỒM SÁCH ĐIỆN TỬ]