Author Picks

FREE

Exploring Microservice Development

Chapters selected by John Carnell

manning

Exploring Microservice Development

Selected by John Carnell

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617295591
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 23 22 21 20 19 18

contents

introduction

The only constant in software development is the ongoing change that buffets and ultimately transforms the field. In a breathtakingly short amount of time, established architectural patterns for development are tossed aside and new development models seem to spring up overnight. If you are a software professional, working in our industry can sometimes give you whiplash.

One of the latest architectural models, microservices, promises the opportunity for companies to deliver new features (and bug fixes) in an extremely short time period. Why? A microservices architecture allows you to break a complex monolithic application into small, distributed, and independent components that can be built and delivered independently of each other. For anyone who has worked in a traditional software company with a six-month lead time between design, development, QA, and deployment, the ability to meet customer needs this quickly can be a giddy, often career-changing experience.

However, a microservices architecture is not a silver bullet. Building a successful microservices architecture requires a cross-discipline approach that insists architects, developers, testers, and operations think differently about how they build and deploy their applications. Building a single microservice is easy. Building, testing, and deploying hundreds of services to work together to deliver a scalable and reliable user experience requires a great deal of operational and engineering maturity.

That is why when Manning asked me to put together a mini ebook on microservices, I did not want to just focus on what specific aspect of microservices. Instead, I have selected a good cross-section of material from Manning authors that covered not only microservice design, but also more concrete subjects, like debugging problems across your microservices and how to effectively test your microservices.

John Carnell
Spring Microservices in Action

Decomposition Strategies

The first step to building a microservice is establishing the granularity of your microservice's responsibilities. If you make your services too fine-grained with minimal response, you will end up with a large number of hard to-manage data services that do very little. If you make your services too coarse-grained, you will find that your microservices have turned into mini-monoliths that are difficult modify and manage.

This chapter provides techniques to identify the natural boundaries that exist between different components within an application. Using the decomposition techniques outlined in this chapter, you will be able to tease apart the data model for your application into smaller chunks and also how to decompose your traditional monolithic applications.

Chapter 2 from *Microservice Patterns* by Chris Richardson.

Decomposition strategies

This chapter covers

- What's software architecture and why it's important
- How to decompose an application into services
- How to use the bounded context concept from Domain-Driven Design to untangle data and make decomposition easier

Let's imagine that you've been given the job of defining the architecture for what's anticipated to be a large, complex application. Your boss, the CIO, heard that the microservices are a necessary part of accelerating software development. He strongly suggests that you consider the microservice architecture.

When defining a microservice architecture it's essential that you use the microservices pattern language described in chapter one. The pattern language consists of patterns that solve numerous problems including deployment, inter-process communication, and strategies for maintaining data consistency. The essence of the microservice architecture's functional decomposition. The first and most

important aspect of the architecture's, therefore, the definition of the services. As you stand in front of that blank whiteboard in the project's first architecture meeting, you'll most likely wonder how to do that!

Don't panic, in this chapter, you'll learn how to define a microservice architecture for an application. I discuss how architecture determines your application's *-ilities* including maintainability, testability, and deployability, which directly impact development velocity. You'll learn that the microservice architecture's an architecture style that gives an application high maintainability, testability, and deployability. I describe strategies for decomposing an application into services. You'll learn that services are organized around business concerns rather than technical concerns. I also show how to use idea from domain driven design to eliminate god classes, which are classes that are used throughout an application and cause tangled dependencies that prevent decomposition. But first, lets take a look at the purpose of architecture.

2.1 The purpose of architecture

The microservice architecture's obviously a kind of architecture. But what is it exactly and why does it matter? To answer that these question lets first look at what's meant by architecture. Len Bass and colleagues define architecture as follows:

> *The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*
>
> —*Documenting Software Architectures* Bass et al

This is obviously an abstract definition. More concretely, an application's architecture can be viewed from multiple perspectives, in the same way that a building's architecture can be viewed from the structural, plumbing, electrical, etc, perspectives. Phillip Krutchen (in his classic paper Architectural Blueprints—The "4+1" View Model of Software Architecture) designed the 4+1 view model of software architecture. The 4+1 model, which is shown in Figure 2.1 , defines four different views of a software architecture, each of which describes a particular aspect of the architecture. Each view consists of a particular set of (software) elements and relationships between them.

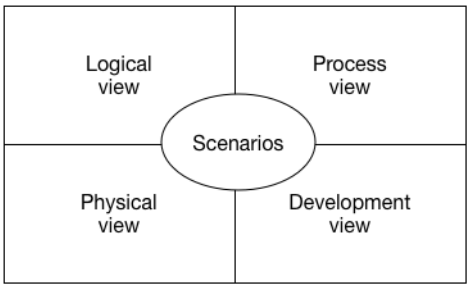


Figure 2.1 4+1 view model

The four views are:

- Logical view—classes, state diagrams, etc.
- Process view—the processes and how they communicate
- Physical view—the physical (virtual machines) and network connections
- Development view—components

In addition to these four views, there are the scenarios—the +1 in the 4+1 model—that animate views. Each scenario describes how the various architectural components within a particular view collaborate in order to handle a request. A scenario in the logical view, for examples, shows how the classes collaborate. Similarly, a scenario in the process view, shows how the processes collaborate.

The the 4+1 view model's an excellent way to describe an applications's architecture. Each view describes an important aspect of the architecture. The scenarios illustrate how the elements of a view collaborate. Lets now look at why architecture's important.

2.1.1 Why architecture matters

An application has two categories of requirements. The first category are the functional requirements, which define what the application must do. They're usually in the form of use cases or user stories. Architecture has little to do with the functional requirements. You can implement functional requirements with almost any architecture, even a big ball of mud.

Architecture's important because it enables an application to satisfy the second category of requirements, its non-functional requirements. These are also known as quality attributes and are the so-called *-ilities*. The non-functional requirements define the runtime qualities such as scalability, and reliability. They also define development time qualities including maintainability, testability, and deployability. The architecture that you choose for your application determines how well it meets these quality requirements.

2.1.2 The microservice architecture's an architectural style

The microservice architecture's what's known as an architectural style. David Garlan and Mary Shaw define an architectural style as follows:

> *An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.*
>
> —*David Garlan and Mary Shaw*, An Introduction to Software Architecture

A particular architectural style provides a limited palette of elements (a.k.a components) and relationships (a.k.a. connectors) from which you can define your application's architecture. The microservice architecture structures an application as a set of loosely coupled services. The components are services and the connectors are the

communication protocols that enable those services to collaborate. Figure 2.2 shows a possible microservice architecture for the FTGO application. Each service in this architecture corresponds to a different business function such as order management, and restaurant management.

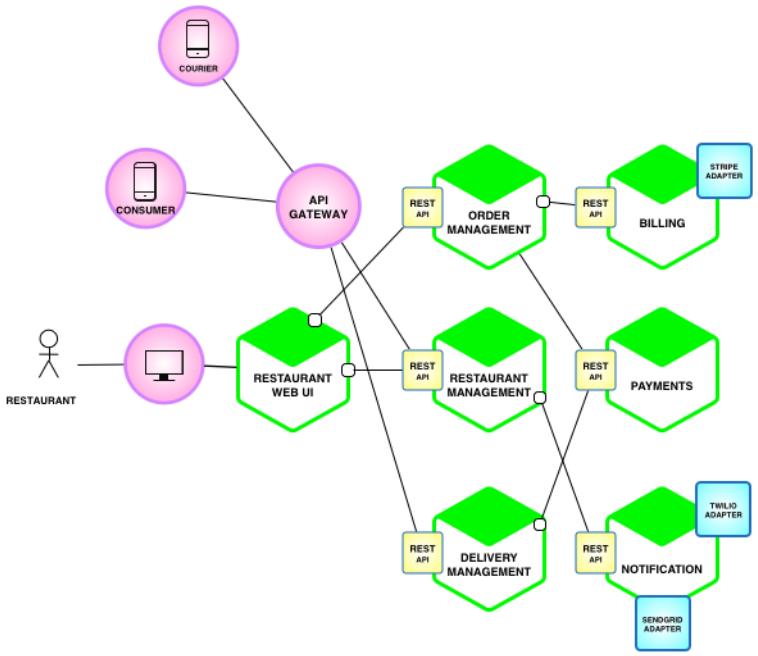


Figure 2.2 A possible microservice architecture for the FTGO application. It consists of numerous services.

Later in this chapter, I describe what's meant by business function. The connectors between services are implemented using REST APIs. In chapter 3, I describe inter-process communication in more detail.

A key constraint imposed by the microservice architecture's that the services are loosely coupled. Consequently, there are restrictions on how the services collaborate. In order to understand those restrictions, lets attempt to define the term *service*, describe what it means to be loosely coupled, and explain why this matters.

WHAT'S A SERVICE?

Intuitively, we think of a service as a standalone, independently deployable software component, which implements some useful functionality. But in reality, the term service's one of those words that we use daily without having a precise definition of its meaning. For example, OASIS has a remarkably vague definition for a service:

> *a mechanism to access an underlying capability*
>
> —OASIS
> https://en.wikipedia.org/wiki/Service_(systems_architecture)

A service has an API, which provides its clients with access to its functionality. Figure 2.3 shows an abstract view of a service.

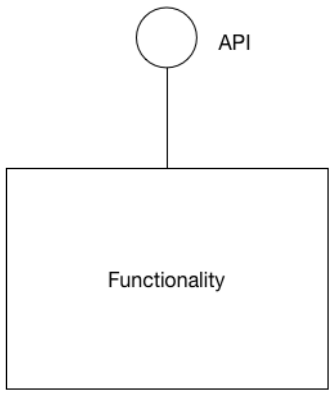


Figure 2.3 Abstract view of a service

A service's clients invoke the API. The service's implementation's hidden, unlike in a monolithic architecture where a developer can easy write code that bypasses a package's API and accesses its implementation. As a result, the microservice architecture enforces the application's modularity.

Typically, the API uses an inter-process communication mechanism, such as a messaging or RPC, but that isn't always the case. Later in the book, when I discuss deployment you'll see that a service can take many forms. It might be a standalone process, a web application or OSGI bundle running in a container, or a serverless cloud function. An essential requirement's that a service's independently deployable.

WHAT IS LOOSE COUPLING?

An important characteristic of the microservice architecture's that the services are loosely coupled. All interaction with a service's via its API, which encapsulates its implementation details. This enables the implementation of the service to change without impacting its clients. Loosely coupled services are key to improving an application's development time attributes including its maintainability and testability. Small, loosely coupled services are much easier to understand, change and test.

The requirement for services to be loosely coupled and to collaborate only via APIs prohibits services from communicating via a database. A service's persistent data's equivalent to the fields of a class and must kept private. Keeping data private helps ensure that the services are loosely coupled. It enables a developer to change their service's schema without having to coordinate with developers working on other services. It also improves runtime isolation. For example, it ensures that one service can't hold database locks that block another service. Later on, you'll learn that a downside of not sharing databases is the complexity of maintaining data consistency and querying across services.

The microservice architecture structures an application as a set of small, loosely coupled services. As a result, it improves the development time attributes—maintainability, testability, deployability, etc—and enables an organization to develop better software faster. It also improves an application's scalability, although this isn't the main goal. To develop a microservice architecture for your application you need to identify the services and determine how they communicate. Lets now look at how to define an application's microservice architecture.

2.1.3 Defining an application's microservice architecture

How do you create an architecture? As with any software development effort the starting points include the written requirements, hopefully domain experts, and perhaps an existing application. Like much of software development, defining an architecture's more art than science. No mechanical process produces an architecture. In this chapter, we describe a simple process for defining an application's architecture. It captures what needs to be done but in reality the process is likely to be iterative and involve a lot of creativity.

An application's raison d'etre's to handle requests, and we'll first identify the key requests. Instead of describing the requests in terms of specific IPC technologies such as REST or messaging, I use the more abstract notion of system operation. Once we've identified the system operations, we'll identify the services that comprise the application's microservice architecture. The system operations become the architectural scenarios that illustrate how the services collaborate. Lets now look at how to identify the system operations.

2.2 Identifying the system operations

The first step of defining an application's architecture's to define the system operations. The starting point's the application's requirements, which include user stories and their associated user scenarios (note – this is different from the architectural scenarios). The system operations are identified and defined using a two step process, which is shown in Figure 2.4. This process is inspired by object-oriented design process described in Craig Larman's book Applying UML and Patterns. The first step creates the high-level domain model consisting of the key classes, which provides a vocabulary to describe the system operations. The second step identifies the system operations and describe each one's behavior in terms of the domain model.

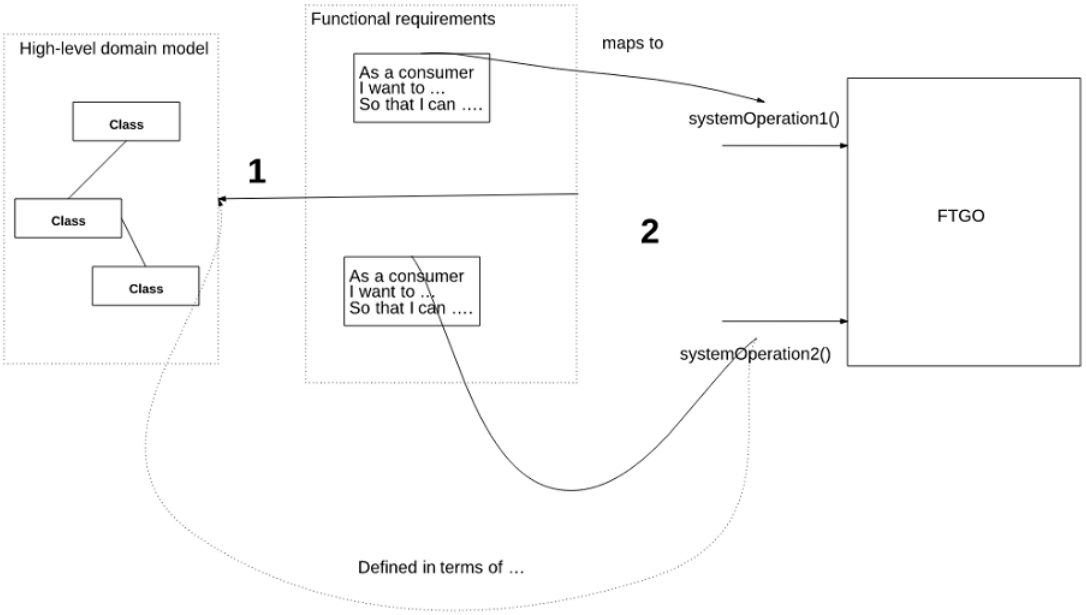


Figure 2.4 System operations are derived from the application's requirements using a two-step process

The domain model's derived primarily from the nouns of the user stories and the system operations are derived mostly from the verbs. The behavior of each system operation's described in terms of its effect on one or more domain objects and the relationships between them. A system operation can create, update or delete domain objects, as well as create or destroy relationships between them. Lets now look at how to define a high-level domain model. After that we'll define the system operations in terms of the domain model.

2.2.1 Creating a high-level domain model

The first step in the process of defining the system operations is to sketch a high-level domain model for the application. Note that this domain model's much simpler than what will ultimately be implemented. The application won't even have a single domain model because, as you'll learn below, each service has its own domain model. Despite being a drastic simplification, a high-level domain model's useful at this stage because it defines the vocabulary for describing the behavior of the system operations.

A domain model's created using standard techniques, such as analyzing the nouns in the stories and scenarios, and by talking to the domain experts. Consider, for example, the `Place Order` story. We can expand that story into numerous user scenarios including this one:

```
Given a consumer
  And a restaurant
When the consumer places an order for that restaurant
   with a delivery address/time that can be served by that restaurant
Then consumer's credit card is authorized
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the consumer
  And the order is associated with the restaurant
```

The nouns in this user scenario hints at the existence of various classes including Consumer, Order, Restaurant and CreditCard. Similarly, the `Accept Order` story can be expanded into scenarios such this one:

```
Given an order or a restaurant that is waiting to be accepted
  and a courier that is available to deliver the order
When a restaurant accepts an order with a promise to prepare by a particular
     time
Then the state of the order is changed to ACCEPTED
  And the order's promiseByTime is updated to the promised time
  And the courier is assigned to deliver the order
```

This scenario suggests the existence of a Courier class. The end result's, after a few iterations of analysis, a domain model that consists, unsurprisingly, of those classes and others such as MenuItem and Address. Figure 2.5 is a class diagram that shows the key classes.

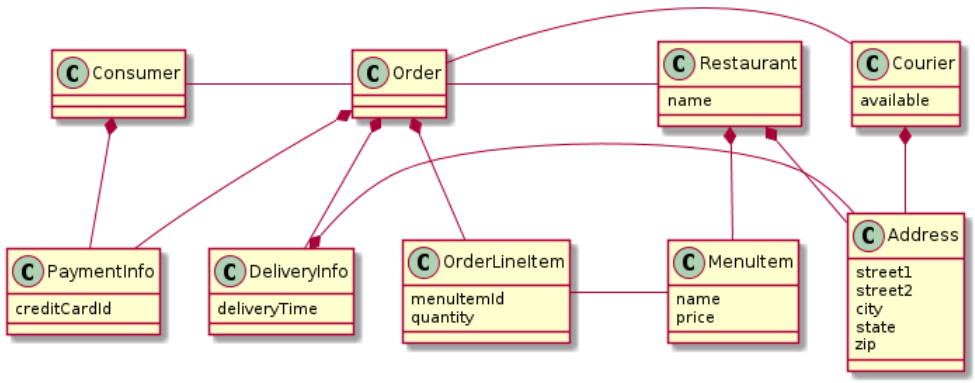


Figure 2.5 The key classes in FTGO domain model

The responsibilities of each class are as follows:

- Consumer—a consumer who places orders.
- Order—an order which is placed by a consumer. It describes the order and tracks its status.
- Restaurant—a restaurant that prepares orders for delivery to consumers
- Courier—a courier who deliver orders to consumers. It tracks the availability of the courier and their current location.

A class diagram such as the one above describes one aspect of an application's architecture. But it isn't much more than a pretty picture without the scenarios to animate it. The next step's to define the system operations, which correspond to architectural scenarios.

2.2.2 Defining system operations

Once you've defined a high-level domain model, the next step's to identify the requests that the application must handle. The details of the UI are out of scope, but you can imagine that in each user scenario the UI makes requests to the backend business logic to retrieve and update data. FTGO's primarily a web application, which means that most requests are HTTP-based, but it's possible that some clients might use messaging. Instead of committing to specific protocol, it makes sense, therefore, to use the more abstract notion of a system operation to represent requests.

Ttwo types of system operations exist:

- commands—system operations that create, update and delete data.
- queries—system operations that read, i.e. query, data.

Ultimately, these system operations correspond to REST, RPC or messaging endpoints. But for now, it's useful to think of them abstractly. Lets first identify some commands.

IDENTIFYING SYSTEM COMMANDS

A good starting point for identifying system commands is to analyze the verbs in the user stories and scenarios. Consider, for example, the `Create Order` story. It clearly suggests that the system must provide a `Create Order` operation. Many other stories individually map directly to system commands. Table 2.1 lists some of the key system commands.

Table 2.1 Key system commands for the FTGO application

Actor	Story	Command	Description
Consumer	Create Order	createOrder()	creates an order
Restaurant	Accept Order	acceptOrder()	indicates that the restaurant accepted the order and is committed to preparing it by the indicated time
Restaurant	Order Ready for Pickup	noteOrderReadyForPickup()	indicates that the order's ready for pickup
Courier	Update Location	noteUpdatedLocation()	updates the current location of the courier
Courier	Order picked up	noteOrderPickedUp()	indicates that the courier picked up the order
Courier	Order delivered	noteOrderDelivered()	indicates that the courier delivered the order

A command has a specification which defines its parameters, returns a value and its behavior in terms of the domain model classes. The behavior specification consists of pre-conditions, which must be true when the operation's invoked, and post-conditions, which are true are after the operation's invoked. Here, for example, is the specification of the `createOrder()` system operation:

Operation	createOrder(consumer id, payment method, delivery address, delivery time, restaurant id, order line items)
Returns	orderId, …
Preconditions	▪ the consumer exists and can place orders ▪ the line items correspond to the restaurant's menu items ▪ the delivery address and time can be serviced by the restaurant
Post-conditions	▪ the consumer's credit card was authorized for the order total ▪ an order was created in the PENDING_ACCEPTANCE state

The pre-conditions mirror the 'givens' in the `Place Order` user scenario described earlier. The postconditions mirror the 'thens' from the scenario. When a system operation's invoked it verifies preconditions and performs the actions required to make the postconditions true.

Here's the specification of the `acceptOrder()` system operation:

Operation	acceptOrder(restaurantId, orderId, readyByTime)
Returns	-
Preconditions	▪ the order.status is PENDING_ACCEPTANCE ▪ a courier's available to deliver the order
Post-conditions	▪ the order.status was changed to ACCEPTED ▪ the order.readyByTime was changed to the `readyByTime` ▪ the courier was assigned to deliver the order

It's pre- and post-conditions mirror the user scenario from earlier.

Most of the architecturally relevant system operations are commands. Sometimes queries, which retrieve data, are also important.

IDENTIFYING SYSTEM QUERIES

As well as implementing commands, an application must also implement queries. The queries provide the UI with the information a user needs to make decisions. At this stage, we don't have a particular UI design for FTGO application in mind, but consider, for example, the flow when a consumer places an order:

1. User enters delivery address and time
2. System displays available restaurants
3. User selects restaurant

4 System displays menu
5 User selects item and checks out
6 System creates order

This user scenario suggests the following queries:

- findAvailableRestaurants(deliveryAddress, deliveryTime)—retrieves the restaurants that can deliver to the specified delivery address at the specified time
- findRestaurantMenu(id)—retrieves information about a restaurant including the menus items

Of the two queries, `findAvailableRestaurants()` is probably the most architecturally significant. It's a complex query involving geosearch. The geosearch component of the query consists of finding all points—restaurants—that are near a location—the delivery address. It also filters out those restaurants aren't open when the order needs to be prepared and picked up. Moreover, performance's critical because this query's executed whenever a consumer wants to place an order.

The high-level domain model and the system operations capture what the application does. They help drive the definition of the application's architecture. The behavior of each system operation's described in terms of the domain model. Each important system operation represents an architecturally significant scenario which is part of the description of the architecture. Lets now look at how to define an application's microservice architecture.

2.3 Strategies for decomposing an application into services

Once the system operations are defined, the next step's to identify the application's services. As mentioned earlier, there isn't a mechanical process to follow. Various decomposition strategies can be used. Each one attacks the problem from a different perspective and uses its own terminology. But with all strategies, the end result's the same: an architecture consisting of services which are primarily organized around business rather than technical concepts. Let's look at the first strategy, which defines services corresponding to business capabilities.

2.3.1 Decompose by business capability

One strategy for creating a microservice architecture's to decompose by business capability. A business capability's a concept from business architecture modeling. A business capability's something that a business does in order to generate value. The set of capabilities for a given business depends on the kind of business. For example, the capabilities of an insurance company typically include Underwriting, Claims management, Billing, Compliance, and so on. The capabilities of an online store include Order management, Inventory management, Shipping and so on.

Business capabilities define what an organization does

An organization's business capabilities capture *what* an organization's business is. They are generally stable. In contrast, *how* an organization conducts its business changes over time, sometimes dramatically. This is particularly true today, with the rapidly growing use of technology to automate many business processes. For example, it wasn't that long ago when you deposited checks at your bank by handing them to a teller. It then became possible to deposit checks using an ATM. And, now today, you can conveniently deposit most checks using your smartphone. As you can see, the `Deposit check` business capability has remained stable but the manner in which it's done has drastically changed.

Identifying business capabilities

An organization's business capabilities are identified by analyzing the organization's purpose, structure, and business processes. Each business capability can be thought of as a service, except it's business-oriented rather than technical. Its specification consists of various components including inputs and outputs, and service-level agreements. For example, the input to an *Insurance Underwriting* capability's the consumer's application, and the outputs include approval and price.

A business capability's often focussed on a particular business object. For example, the Claim business object's the focus of the Claim management capability. A capability can often be decomposed into sub-capabilities. For example, the Claim management capability has several sub-capabilities including Claim information management, Claim review, and Claim payment management.

It isn't difficult to imagine that the business capabilities for FTGO include:

- Supplier management
  - Courier management—managing courier information
  - Restaurant information management—managing restaurant menus and other information including location and opening hours
- Consumer management—managing information about consumers
- Order taking and fulfillment
  - Order management—enabling consumers to create and manage orders
  - Restaurant order management—managing the preparation of orders at a restaurant
  - Logistics
  - Courier availability management—managing the real-time availability of couriers to delivery orders
  - Delivery management—delivering orders to consumers
- Accounting
  - Consumer accounting—managing billing of consumers
  - Restaurant accounting—managing payments to restaurants
  - Courier accounting—managing payments to couriers
- …

The top-level capabilities include Supplier management, Consumer management, Order taking and fulfillment, and accounting. There'll likely be many other top-level capabilities including marketing related capabilities. Most top-level capabilities are decomposed into sub-capabilities. For example, Order taking and fulfillment's decomposed of into three sub-capabilities.

On interesting aspect of this capability hierarchy's that there are two restaurant related capabilities: Restaurant information management, and Restaurant order management. This is because they represent two different aspects of restaurant operations. Lets now look at how to use business capabilities to define services.

FROM SERVICE TO BUSINESS CAPABILITIES

Once you've identified the business capabilities, you then define a service for each capability or group of related capabilities. Figure 2.6 shows the mapping from capabilities to services for the FTGO application. The top-level accounting capabilities are mapped to the `Accounting Service`, and each of the sub-capabilities of Supplier management are mapped to a service.

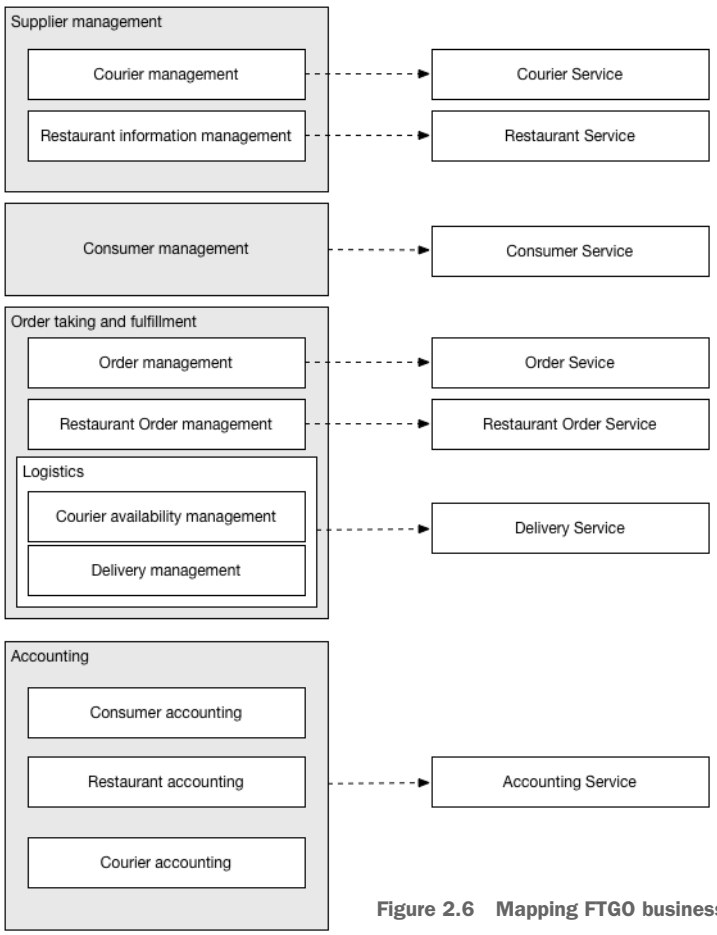


Figure 2.6 Mapping FTGO business capabilities to services

A key benefit of organizing services around capabilities is that, because they are stable, the resulting architecture's also relatively stable. The individual components of the architecture may evolve as the *how* aspect of the business changes but the architecture remains unchanged.

Having said that, it's important to remember that the services shown in figure 2.6 are merely the first attempt at defining the architecture. They may evolve over time as we learn more about the application domain. In particular, an important step in the architecture definition process is investigating how the services collaborate in each of the key architectural service. You might, for example, discover that a particular decomposition's inefficient due to excessive inter-process communication, and that you must combine services. Let's take a look at the scenarios.

2.3.2 Using scenarios to determine how the service collaborate

An application's architecture consists of both software elements—the services—and the relationships between them—communication mechanisms. Consequently, after having identified the services, we must decide how the services communicate. To do that we must think about how services collaborate during each scenario. The system operations define the scenarios and drive the definition of the architecture. Because a system operation's a request from the external world, the first decision to make's which service initially handles the request. After that we must decide what other services are involved in handling the request and how they communicate.

ASSIGNING SYSTEM OPERATIONS TO SERVICES

The first step's to decide which service's the initial entry point for a request. Many system operations neatly map to a service, but sometimes the mapping's less obvious. Consider, for example, the `noteUpdatedLocation()` operation, which updates the courier location. On the one hand, because it's related to couriers, this operation should be assigned to the `Courier` service. But on the other hand, it's the `Delivery Service` that needs the courier location. In this case, assigning an operation to a service that needs the information provided by the operation's a better choice. In other situations, it might make sense to assign an operation to the service which has the information necessary to handle it.

Table 2.2 shows which services in the FTGO application are responsible for which operations.

Table 2.2 Mapping system operations to services in the FTGO application

Service	Operation
Order Service	▪ createOrder() ▪ findAvailableRestaurants()
Restaurant Order Service	▪ acceptOrder() ▪ noteOrderReadyForPickup()
Delivery Service	▪ noteUpdatedLocation() ▪ noteOrderPickedUp() ▪ noteOrderDelivered()

After having assigned operations to services, the next step's to decide how the services collaborate in order to handle each system operation.

DETERMINING HOW SERVICES COLLABORATE

Some system operations are handled entirely by a single service. Other system operations span multiple services. The knowledge needed to handle one of these requests might, for instance, is scattered around multiple services. For example, in the FTGO application the `Consumer Service` handles the `createConsumer()` operation entirely by itself. When handling the `createOrder()` operation, the `Order Service` must invoke other services including:

- `Consumer Service`—verify that the consumer can place an order and obtain their payment information
- `Restaurant Order Service`—verify the order line items and that the delivery address/time is within the restaurant's service area
- `Accounting Service`—authorize the consumer's credit card

Similarly, when the restaurant accepts an order, the `Restaurant Order Service` must invoke the `Delivery Service` to schedule a courier to deliver the order. Figure 2.7 shows the services and the dependencies between them. Each dependency represents some kind of inter-service communication.

Even though figure 2.7 suggests that the services communicate using REST, it's important to remember that in practice they might use other communication mechanisms such as asynchronous messaging. In fact, in chapter 3, which covers inter-process communication, and chapter 4, which discusses transaction management, I describe how asynchronous messaging plays a major role in a microservice architecture.

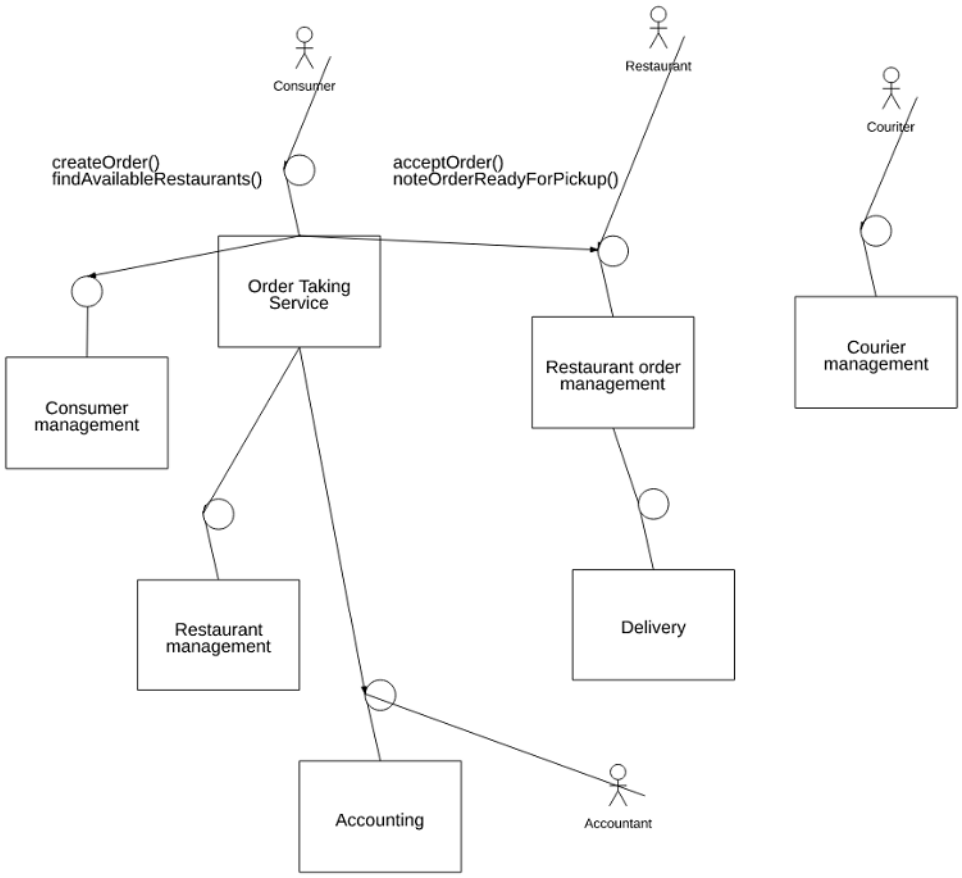


Figure 2.7 The FTGO services and their dependencies

2.3.3 *If only it were this easy…*

On the surface, the strategy of creating a microservice architecture by defining services corresponding to business capabilities looks quite reasonable. Unfortunately, there are some significant problems that need to be addressed.

SYNCHRONOUS INTER-PROCESS COMMUNICATION REDUCES AVAILABILITY

The first problem's how to use inter-service communication in a way that doesn't reduce availability. For example, the most straightforward way to implement the `createOrder()` operation's for the `Order Service` to synchronously invoke the other services using REST. The drawback of doing this using a protocol such as REST is that it reduces the availability of the `Order Service`. It won't create an order if any of those services are unavailable. Sometimes this is a worthwhile trade-off, but in chapter three you'll learn that using asynchronous messaging, which eliminates tight coupling and improves availability's often a better choice.

THE NEED FOR DISTRIBUTED TRANSACTIONS

The second problem's that a system operation that spans multiple services must somehow maintain data consistency across those services. For example, when a restaurant accepts an order the `Restaurant Order Service` invokes the `Delivery Service` to schedule delivery of the order. The `acceptOrder()` operation must reliably update data in the `Restaurant Order Service` and `Delivery Service`. The traditional solution's to use a two-phase commit-based, distributed transaction management mechanism. Unfortunately, as you'll learn in chapter four, this isn't a good choice for modern applications, and you must use a different approach to transaction management.

GOD CLASSES PREVENT DECOMPOSITION

The third and final problem's the existence of God classes, which are classes used throughout the application that make it difficult to decompose the business logic. An example of a god class in the FTGO application's the `Order` class. It has state and behavior for many different aspects of the FTGO application's business logic including order taking, restaurant order management, and delivery. Consequently, this class makes it extremely difficult to decompose any of the business logic that involves orders into the services described earlier.

Fortunately, there's a way to eliminate god classes. The technique comes from Domain-Driven Design (DDD), which provides an alternative way to decompose to an application. As with business capability based decomposition, this strategy takes a domain-oriented approach. The resulting architecture's the same, despite DDD using different terminology and having different motivations. One particularly valuable contribution of DDD's that it provides a way to eliminate the god classes.

2.3.4 Decompose by sub-domain/bounded context

DDD's an approach for building complex software applications which is centered on the development of an object-oriented, domain model. A domain model captures knowledge about a domain in a form that can be used to solve problems within that domain. It defines the vocabulary used by the team—what DDD calls the Ubiquitous language. The domain model's closely mirrored in the design and implementation of the application. DDD has two concepts that are incredibly useful when applying the microservice architecture: subdomains and bounded contexts.

FROM SUBDOMAINS TO SERVICES

DDD's quite different than the traditional approach to enterprise modeling which creates a single model for the entire enterprise. In such a model, there'd be, for example, a single definition of each business entity such as customer, order etc. The problem with kind of modeling's that getting different parts of an organization to agree on a single model's a monumental task. Also, it means that, from the perspective of a given part of the organization, the model's overly complex for their needs. Moreover, the domain model can be confusing because different parts of the organization might use either the same term for different concepts or different terms for the same con-

cept. DDD avoids these problems by defining multiple domain models, each one with an explicit scope.

DDD defines a separate domain model for each subdomain. A subdomain's a part of the domain, which is DDD's term for the application's problem space. Subdomains are identified using the same approach as identifying business capabilities: analyze the business and identify the different areas of expertise. The end result's likely to be subdomains which are similar to the business capabilities. The examples of subdomains in FTGO include order taking, order management, restaurant order management, delivery, and financials. As you can see, these subdomains are similar to the business capabilities described earlier.

DDD calls the scope of a domain model a bounded context. A bounded context includes the code artifacts etc that implement the model. When using the microservice architecture, each bounded context's a service or possibly a set of services. We can create a microservice architecture by applying DDD and defining a service for each subdomain. Figure 2.8 shows how the subdomains map to services, each with their own domain model.

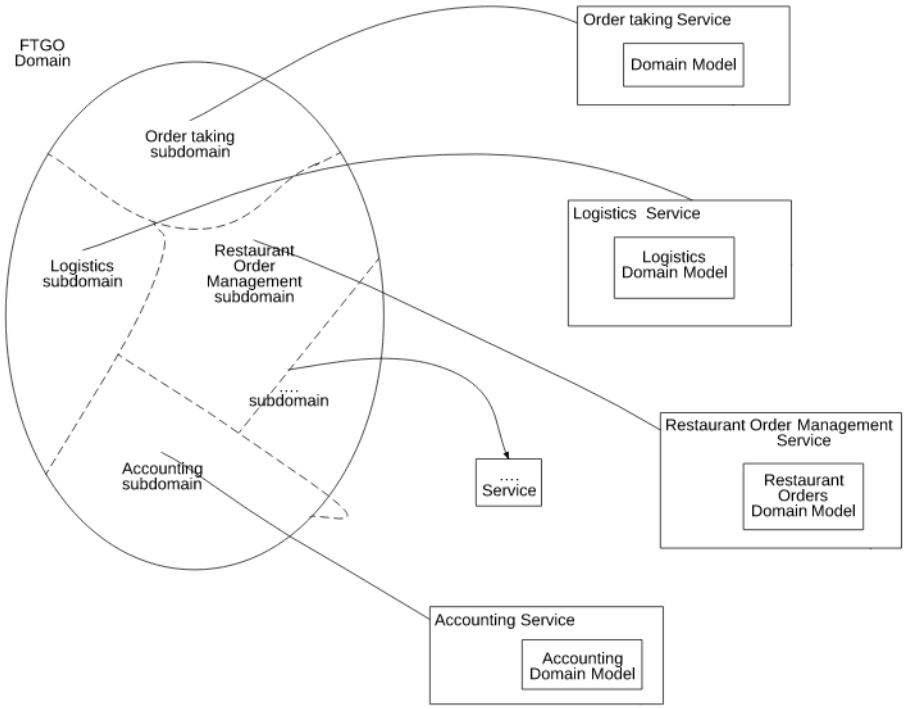


Figure 2.8 From subdomains to services

DDD and the microservice architecture are in near perfect alignment. The DDD concept of subdomains and bounded contexts map nicely to services within a microservice architecture. Also, the microservice architectures' concept of autonomous teams owning services is completely aligned with the DDD's concept of each domain model being owned and developed by a single team. What's even better's that the concept of a subdomain with its own domain model's a great way to eliminate God classes, which makes decomposition easier.

ELIMINATING GOD CLASSES

God classes[1] are the bloated classes used throughout an application. A god class typically implements business logic for many different aspects of the application. It typically has a large number of fields mapped to a database table with many columns. Most applications have at least one these classes, each one representing a concept which is central to the domain: accounts in banking, orders in e-commerce, policies in insurance etc. Because a god class bundles together state and behavior for many different aspects of an application it's an insurmountable obstacle to splitting any business logic that uses it into services.

The `Order` class is a great example of a god class in the FTGO application. That isn't surprising; the purpose of FTGO's to deliver food orders to customers. Most parts of the system involve orders. If the FTGO application had a single domain model then the `Order` class would be a huge class. It'd have state and behavior corresponding to many different parts of the application. Figure 2.9shows the structure of the class created using traditional modeling techniques.

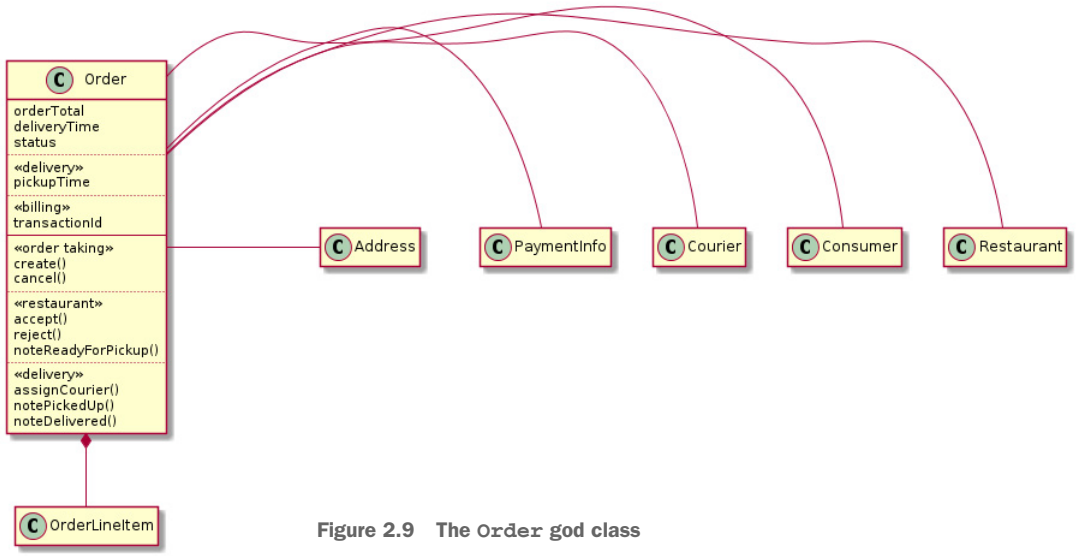


Figure 2.9 The `Order` god class

[1] wiki.c2.com/?GodClass

As you can see, the Order class has fields and methods corresponding order processing, restaurant order management, delivery and payments. This class also has a complex state model because that one model must describe state transitions from disparate parts of the application. This class, in its current form, makes it extremely difficult to split code into services.

One solution's to package the `Order` class into a library and create a central `Order` database. All services that process orders use this library and access the access database. The trouble with this approach's that it violates one of the key principles of the microservice architecture and results in undesirable, tight coupling. For example, any change to the `Order` schema requires the teams to update their code in lock step.

Another solution's to encapsulate the `Order` database in an `Order Service`, which is invoked by the other services to retrieve and update orders. The problem with this design's that the `Order Service` would be a data service with an anaemic domain model containing little or no business logic. Neither of these options are appealing, but fortunately DDD provides a solution.

DDD eliminates these god classes by treating each part of an application as separate subdomain with its own domain model. This means that service in the FTGO application associated with orders has a domain model with its own version of the Order class. A great example of the benefit of multiple domain models is the `Delivery Service`. Its view of an Order, which is shown in figure 2.10, is extremely simple: pickup address, pickup time, delivery address, and delivery time. Moreover, rather than call it an `Order` the `Delivery Service` uses the more appropriate name of `Delivery`.

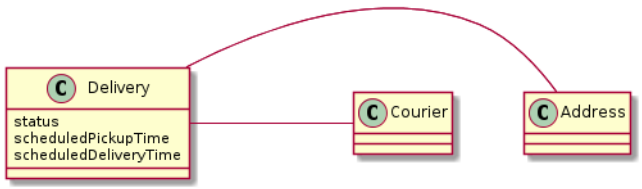


Figure 2.10 The delivery subdomain model

The `Delivery Service` isn't interested in any of the other attributes of an order..

The `Restaurant Order` service also has a much simpler view of an order. Its version of an Order, which is shown figure 2.11, consists of a status, a `prepareByTime` and a list of line item, which tells the restaurant what to prepare.

Figure 2.11 The restaurant order subdomain model

It's unconcerned with the consumer, payment, delivery etc.

The `Order` service has the most complex view of an order, which is shown in figure 2.12. Even though it has many fields and methods it's still much simpler than the original version.

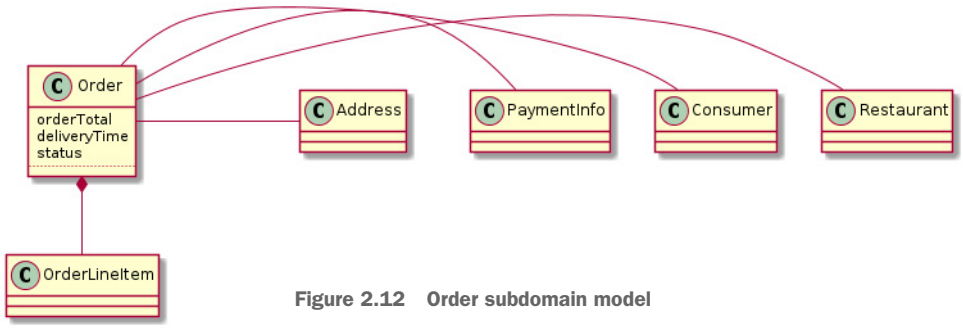


Figure 2.12 Order subdomain model

The Order class in each domain model represent different aspects of the same Order business entity. The FTGO application must maintain consistency between these different objects in different services. For example, once the `Order Service` authorizes consumers to create a card, it must trigger the creation in the `Restaurant Order Service`. Similarly, if the restaurant rejects the order via the `Restaurant Order Service`, it must be cancelled in the `Order Management` service and the customer's credited in the billing service. In chapter four, you'll learn how to maintain consistency between services using event-driven mechanism called sagas.

As you can see, DDD has some concepts that are extremely useful when defining a microservice architecture. It works well with business capability-based decomposition. Let's look at some other guidelines that we should keep in mind developing a microservice architecture.

2.3.5 Decomposition guidelines

A couple of principles from object-oriented design can be adapted and used when applying the microservice architecture. They were created by Robert C. Martin and described in his classic book Designing Object Oriented C++ Applications Using The Booch Method. The first principle's the Single Responsibility Principle (SRP) for defining the responsibilities of a class. The second principle's the Common Closure Principle (CCP) for organizing classes into packages. Let's take a look at these principles and see how they can be applied to the microservice architecture.

SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle states that

> *A class should have only one reason to change.*
>
> —*Robert C. Martin*

Each responsibility of a class is a potential reason for that class to change. If a class has multiple responsibilities that change independently then the class won't be stable. By

following the SRP, you define classes that each have a single responsibility and hence a single reason for change.

We can apply SRP when defining a microservice architecture and create small, cohesive services that each have single responsibility. This reduces the size of the services and increases their stability. The new FTGO architecture's an example of SRP in action. Each aspect of getting food to a consumer—order taking, order preparation, and delivery—is the responsibility of a separate service.

Common Closure Principle

The other useful principle's the Common Closure Principle. It states that

> *The classes in a package should be closed together against the same kinds of changes. a change that affects a package affects all the classes in that package.*
>
> —*Robert C. Martin*

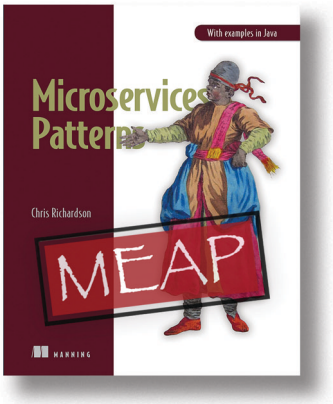
The idea's that when two classes change in lock step because of the same underlying reason, they belong in the same package. Perhaps, for example, those classes implement a different aspect of a particular business rule. The goal's that when that business rule changes developers, only need to change code in a small number—ideally only one—of packages. Adhering to the CCP significantly improves the maintainability of an application.

We can apply CCP when creating a microservice architecture and package components that change for the same reason into the same service. Doing this minimizes the number of services that need to be changed and deployed when some requirement changes. Ideally, a change only affects a single team and a single service. CCP's the antidote to the distributed monolith anti-pattern.

Decomposition by business capability and by subdomain along with SRP and CCP are good techniques for decomposing an application into services. In order to apply them and successfully develop a microservice architecture, you must solve some transaction management and inter-process communication issues.

2.4 Summary

- Architecture determines your aopplication's *-ilities* including maintainability, testability, and deployability, which directly impact development velocity
- The microservice architecture's an architecture style that gives an application high maintainability, testability, and deployability
- Services in a microservice architecture are organized around business concerns—business capabilities or subdomains—rather than technical concerns
- You can eliminate God classes, which cause tangled dependencies that prevent decomposition, by applying DDD and defining a separate domain model for each service



The monolithic architecture works well for small, simple applications. However, successful applications have a habit of growing. Eventually the development team ends up in what is known as monolithic hell. All aspects of software development and deployment become painfully slow. The solution is to adopt the microservice architecture, which structures an application as a services, organized around business capabilities. This architecture accelerates software development and enables continuous delivery and deployment of complex software applications.

Microservice Patterns teaches enterprise developers and architects how to build applications with the microservice architecture. Rather than simply advocating for the use the microservice architecture, this clearly-written guide takes a balanced, pragmatic approach. You'll discover that the microservice architecture is not a silver bullet and has both benefits and drawbacks. Along the way, you'll learn a pattern language that will enable you to solve the issues that arise when using the microservice architecture. This book also teaches you how to refactor a monolithic application to a microservice architecture.

What's inside:

- Understanding the microservice architecture
- When and when not to use the microservice architecture
- How to develop a microservice architecture for an application
- Transaction management and querying in a microservice architecture
- Effective testing strategies for microservices
- How to refactor a monolithic application into services

Readers should be familiar with the basics of enterprise application architecture, design, and implementation.

Distributed Tracing with Spring Cloud Sleuth and Zipkin

Microservices bring high business velocity by allowing development teams to quickly deliver critical bug fixes to new features. However, the trade-off for this speed is increased operational complexity. Understanding what is going on in your services and being able to link the log output of a user's transactions across multiple services in your application is critical to understanding how your services behave and debugging the errors you encounter within your application.

This chapter will demonstrate how to use "correlation ids" as a tracer bullet that will allow you to track a user's transactions across multiple microservice's logs. Then, a demonstration will teach you how to use a log aggregation tool to pull the logs from different microservices into a single, searchable database. Finally, we will use a microservice visualization tool called Zipkin to provide a graphical breakdown of a user's transaction as it flows across all the services involved in the transaction.

Chapter 9 from *Spring Microservices in Action*
by John Carnell.

Distributed tracing with Spring Cloud Sleuth and Zipkin

This chapter covers

- Using Spring Cloud Sleuth to inject tracing information into service calls
- Using log aggregation to see logs for distributed transaction
- Querying via a log aggregation tool
- Using OpenZipkin to visually understand a user's transaction as it flows across multiple microservice calls
- Customizing tracing information with Spring Cloud Sleuth and Zipkin

The microservices architecture is a powerful design paradigm for breaking down complex monolithic software systems into smaller, more manageable pieces. These manageable pieces can be built and deployed independently of each other; however, this flexibility comes at a price: complexity. Because microservices are distributed by nature, trying to debug where a problem is occurring can be maddening. The distributed nature of the services means that you have to trace one or more transactions across multiple services, physical machines, and different data stores, and try to piece together what exactly is going on.

This chapter lays out several techniques and technologies for making distributed debugging possible. In this chapter, we look at the following:

- Using correlation IDs to link together transactions across multiple services
- Aggregating log data from multiple services into a single searchable source
- Visualizing the flow of a user transaction across multiple services and understanding the performance characteristics of each part of the transaction

To accomplish the three things you're going to use three different technologies:

- *Spring Cloud Sleuth* (https://cloud.spring.io/spring-cloud-sleuth/)—Spring Cloud Sleuth is a Spring Cloud project that instruments your HTTP calls with correlation IDs and provides hooks that feed the trace data it's producing into OpenZipkin. It does this by adding the filters and interacting with other Spring components to let the correlation IDs being generated pass through to all the system calls.
- *Papertrail* (https://papertrailapp.com)—Papertrail is a cloud-based service (freemium-based) that allows you to aggregate logging data from multiple sources into single searchable database. You have options for log aggregation, including on-premise, cloud-based, open source, and commercial solutions. We'll explore several of these alternatives later in the chapter
- *Zipkin* (http://zipkin.io)—Zipkin is an open source data-visualization tool that can show the flow of a transaction across multiple services. Zipkin allows you to break a transaction down into its component pieces and visually identify where there might be performance hotspots.

To begin this chapter, we start with the simplest of tracing tools, the correlation ID.

NOTE Parts of this chapter rely on material covered in chapter 6 (specifically the material on Zuul pre-, response, and post filters). If you haven't read chapter 6 yet, I recommend that you do so before you read this chapter.

9.1 Spring Cloud Sleuth and the correlation ID

We first introduced the concept of correlation IDs in chapter 5 and 6. A correlation ID is a randomly generated, unique number or string that's assigned to a transaction when a transaction is initiated. As the transaction flows across multiple services, the correlation ID is propagated from one service call to another. In the context of chapter 6, you used a Zuul filter to inspect all incoming HTTP requests and inject a correlation ID if one wasn't present.

Once the correlation ID was present, you used a custom Spring HTTP filter on every one of your services to map the incoming variable to a custom `UserContext` object. With the `UserContext` object in place, you could now manually add the correlation ID to any of your log statements by making sure you appended the correlation ID to the log statement, or, with a little work, add the correlation ID directly to Spring's Mapped Diagnostic Context (MDC). You also wrote a Spring Interceptor that

would ensure that all HTTP calls from a service would propagate the correlation ID by adding the correlation ID to the HTTP headers on any outbound calls.

Oh, and you had to perform Spring and Hystrix magic to make sure the thread context of the parent thread holding the correlation ID was properly propagated to Hystrix. Wow—in the end this was a lot of infrastructure that was put in place for something that you hope will only be looked at when a problem occurs (using a correlation ID to trace what's going on with a transaction).

Fortunately, Spring Cloud Sleuth manages all this code infrastructure and complexity for you. By adding Spring Cloud Sleuth to your Spring Microservices, you can

- Transparently create and inject a correlation ID into your service calls if one doesn't exist.
- Manage the propagation of the correlation ID to outbound service calls so that the correlation ID for a transaction is automatically added to outbound calls.
- Add the correlation information to Spring's MDC logging so that the generated correlation ID is automatically logged by Spring Boots default SL4J and Logback implementation.
- Optionally, publish the tracing information in the service call to the Zipkin-distributed tracing platform.

NOTE With Spring Cloud Sleuth if you use Spring Boot's logging implementation, you'll automatically get correlation IDs added to the log statements you put in your microservices.

Let's go ahead and add Spring Cloud Sleuth to your licensing and organization services.

9.1.1 Adding Spring Cloud sleuth to licensing and organization

To start using Spring Cloud Sleuth in your two services (licensing and organization), you need to add a single Maven dependency to the pom.xml files in both services:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

This dependency will pull in all the core libraries needed for Spring Cloud Sleuth. That's it. Once this dependency is pulled in, your service will now

1. Inspect every incoming HTTP service and determine whether or a not Spring Cloud Sleuth tracing information exists in the incoming call. If the Spring Cloud Sleuth tracing data does exist, the tracing information passed into your microservice will be captured and made available to your service for logging and processing.
2. Add Spring Cloud Sleuth tracing information to the Spring MDC so that every log statement created by your microservice will be added to the logs.
3. Inject Spring Cloud tracing information into to every outbound HTTP call and Spring messaging channel message your service makes.

9.1.2 Anatomy of a Spring Cloud Sleuth trace

If everything is set up correctly, any log statements written within your service application code will now include Spring Cloud Sleuth trace information. For example, figure 9.1 shows what the service's output would look like if you were to do an HTTP `GET http://localhost:5555/api/organization/v1/organizations/e254f8c -c442-4ebe-a82a-e2fc1d1ff78a` on the organization service.

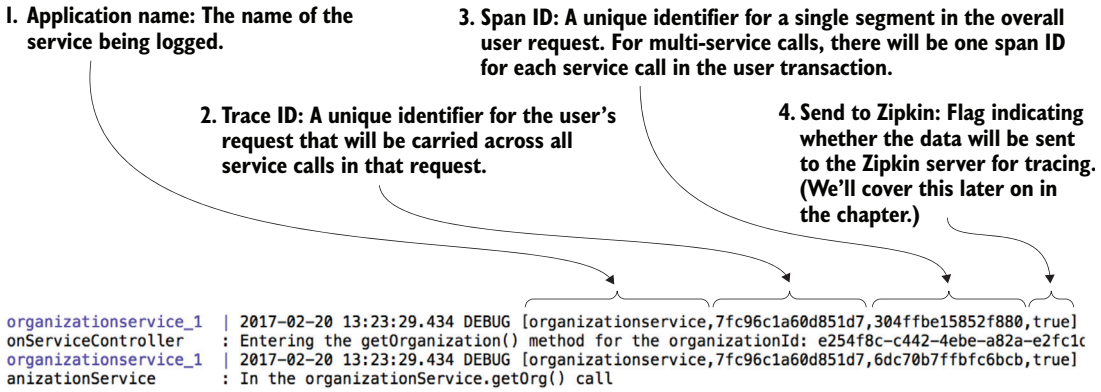


Figure 9.1 Spring Cloud Sleuth adds four pieces of tracing information to each log entry written by your service. This data helps tie together service calls for a user's request.

Spring Cloud Sleuth will add four pieces of information to each log entry. These four pieces (numbered to correspond with the numbers in figure 9.1) are

1. *Application name of the service*—This is going to be the name of the application the log entry is being made in. By default, Spring Cloud Sleuth uses the name of the application (`spring.application.name`) as the name that gets written in the trace.
2. *Trace ID*—Trace ID is the equivalent term for correlation ID. It's a unique number that represents an entire transaction.
3. *Span ID*—A span ID is a unique ID that represents part of the overall transaction. Each service participating within the transaction will have its own span ID. Span IDs are particularly relevant when you integrate with Zipkin to visualize your transactions.
4. *Whether trace data was sent to Zipkin*—In high-volume services, the amount of trace data generated can be overwhelming and not add a significant amount of value. Spring Cloud Sleuth lets you determine when and how to send a transaction to Zipkin. The true/false indicator at the end of the Spring Cloud Sleuth tracing block tells you whether the tracing information was sent to Zipkin.

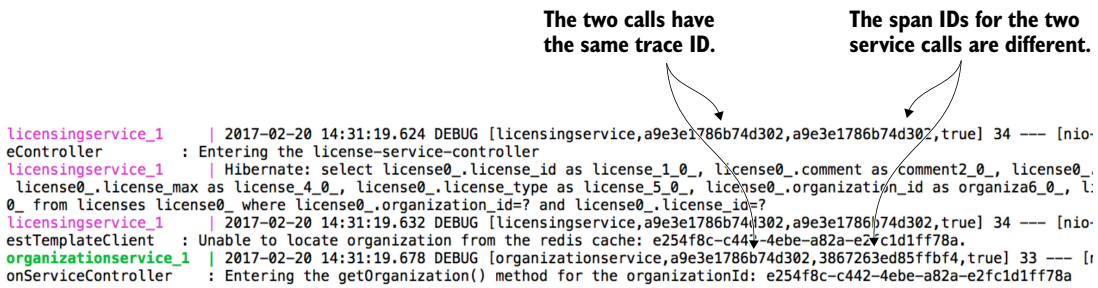


Figure 9.2 With multiple services involved in a transaction, you can see that they share the same trace ID.

Up to now, we've only looked at the logging data produced by a single service call. Let's look at what happens when you make a call to the licensing service at `GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a`. Remember, the licensing service also has to call out to the organization service. Figure 9.2 shows the logging output from the two service calls.

By looking at figure 9.2, you can see that both the licensing and organization services have the same trace ID `a9e3e1786b74d302`. However, the licensing service has a span ID of `a9e3e1786b74d302` (the same value as the transaction ID). The organization service has a span ID of `3867263ed85ffbf4`.

By adding nothing more than a few POM dependencies, you've replaced all the correlation ID infrastructure that you built out in chapters 5 and 6. Personally, nothing makes me happier in this world then replacing complex, infrastructure-style code with someone else's code.

9.2 Log aggregation and Spring Cloud Sleuth

In a large-scale microservice environment (especially in the cloud), logging data is a critical tool for debugging problems. However, because the functionality for a microservice-based application is decomposed into small, granular services and you can have multiple service instances for a single service type, trying to tie to log data from multiple services to resolve a user's problem can be extremely difficult. Developers trying to debug a problem across multiple servers often have to try the following:

- Log into multiple servers to inspect the logs present on each server. This is an extremely laborious task, especially if the services in question have different transaction volumes that cause logs to rollover at different rates.
- Write home-grown query scripts that will attempt to parse the logs and identify the relevant log entries. Because every query might be different, you often end up with a large proliferation of custom scripts for querying data from your logs.

- Prolong the recovery of a down service process because the developer needs to back up the logs residing on a server. If a server hosting a service crashes completely, the logs are usually lost.

Each of the problems listed are real problems that I've run into. Debugging a problem across distributed servers is ugly work and often significantly increases the amount of time it takes to identify and resolve an issue.

A much better approach is to stream, real-time, all the logs from all of your service instances to a centralized aggregation point where the log data can be indexed and made searchable. Figure 9.3 shows at a conceptual level how this "unified" logging architecture would work.

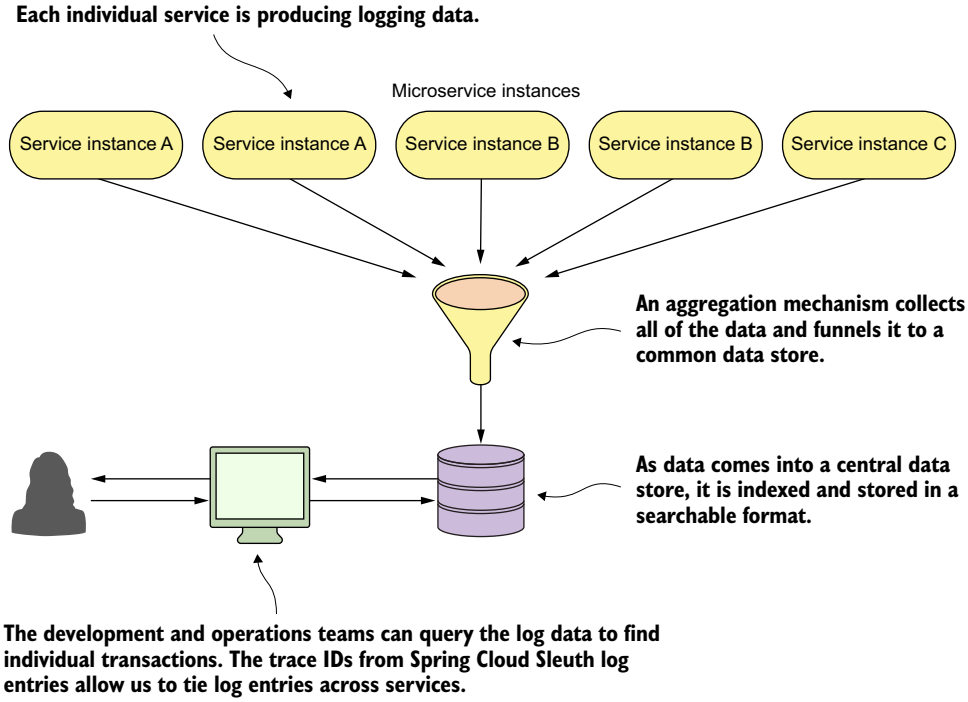


Figure 9.3 The combination of aggregated logs and a unique transaction ID across service log entries makes debugging distributed transactions more manageable.

Fortunately, there are multiple open source and commercial products that can help you implement the previously described logging architecture. Also, multiple implementation models exist that will allow you to choose between an on-premise, locally managed solution or a cloud-based solution. Table 9.1 summarizes several of the choices available for logging infrastructure.

Table 9.1 Options for Log Aggregation Solutions for Use with Spring Boot

Product Name	Implementation Models	Notes
Elasticsearch, Logstash, Kibana (ELK)	Open source Commercial Typically implemented on premise	http://elastic.co General purpose search engine Can do log-aggregation through the (ELK-stack) Requires the most hands-on support
Graylog	Open source Commercial On-premise	http://graylog.org Open-source platform that's designed to be installed on premise
Splunk	Commercial only On-premise and cloud-based	http://splunk.com Oldest and most comprehensive of the log management and aggregation tools Originally an on-premise solution, but have since offered a cloud offering
Sumo Logic	Freemium Commercial Cloud-based	http://sumologic.com Freemium/tiered pricing model Runs only as a cloud service Requires a corporate work account to signup (no Gmail or Yahoo accounts)
Papertrail	Freemium Commercial Cloud-based	http://papertrailapp.com Freemium/tiered pricing model Runs only as a cloud service

With all these choices, it might be difficult to choose which one is the best. Every organization is going to be different and have different needs.

For the purposes of this chapter, we're going to look at Papertrail as an example of how to integrate Spring Cloud Sleuth-backed logs into a unified logging platform. I chose Papertrail because

1. It has a freemium model that lets you sign up for a free-tiered account.
2. It's incredibly easy to set up, especially with container runtimes like Docker.
3. It's cloud-based. While I believe a good logging infrastructure is critical for a microservices application, I don't believe most organizations have the time or technical talent to properly set up and manage a logging platform.

9.2.1 A Spring Cloud Sleuth/Papertrail implementation in action

In figure 9.3 we saw a general unified logging architecture. Let's now see how the same architecture can be implemented with Spring Cloud Sleuth and Papertrail.

To set up Papertrail to work with your environment, we have to take the following actions:

1. Create a Papertrail account and configure a Papertrail syslog connector.
2. Define a Logspout Docker container (https://github.com/gliderlabs/logspout) to capture standard out from all the Docker containers.

3 Test the implementation by issuing queries based on the correlation ID from Spring Cloud Sleuth.

Figure 9.4 shows the end state for your implementation and how Spring Cloud Sleuth and Papertrail fit together for your solution.

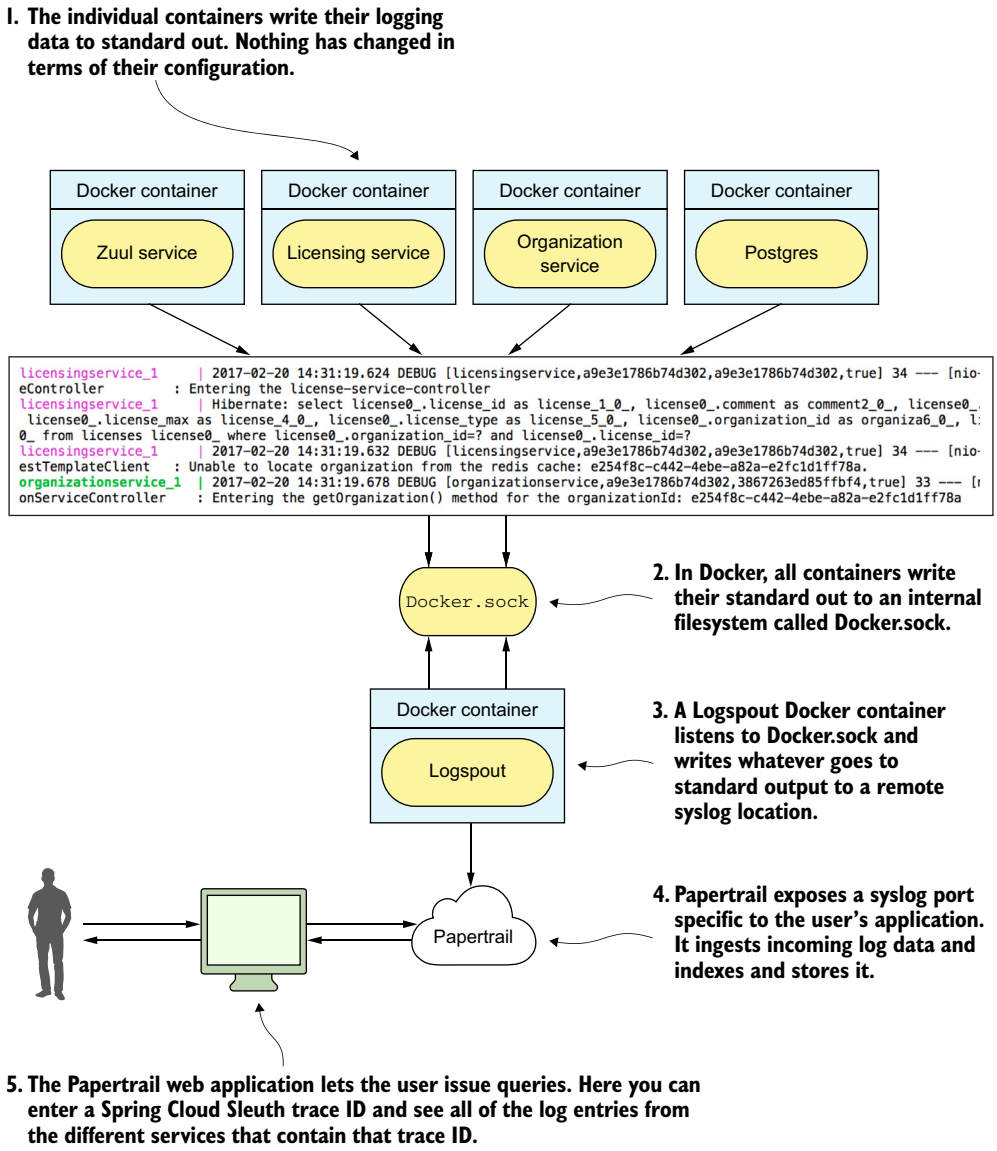


Figure 9.4 Using native Docker capabilities, logspot, and Papertrail allows you to quickly implement a unified logging architecture.

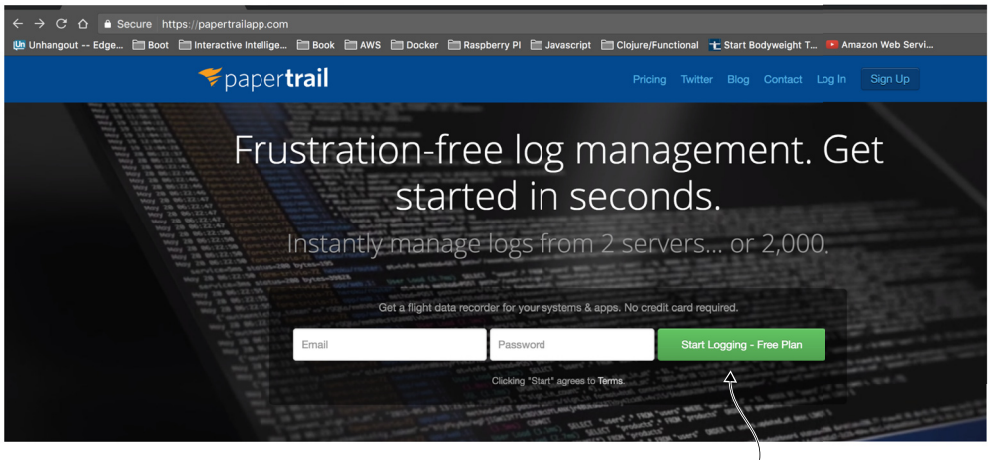


Figure 9.5 To begin, create an account on Papertrail.

9.2.2 Create a Papertrail account and configure a syslog connector

You'll begin by setting up a Papertrail. To get started, go to https://papertrailapp.com and click on the green "Start Logging – Free Plan" button. Figure 9.5 shows this.

Papertrail doesn't require a significant amount of information to get started; only a valid email address. Once you've filled out the account information, you'll be presented with a screen to set up your first system to log data from. Figure 9.6 shows this screen.

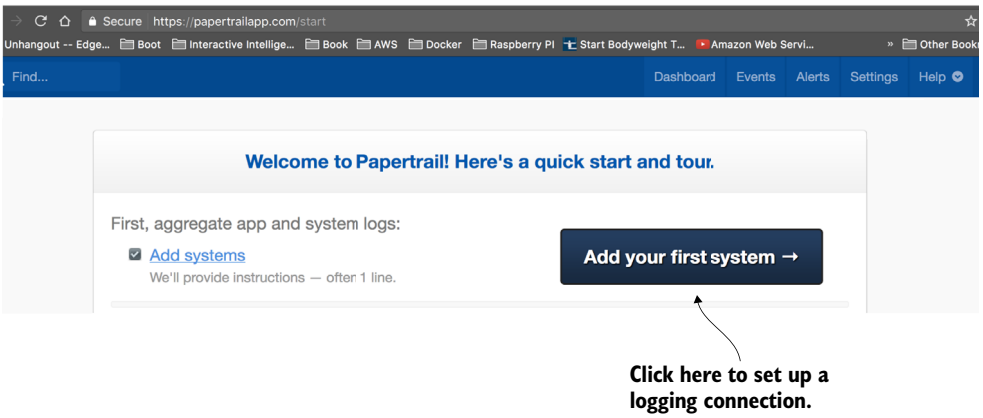


Figure 9.6 Next choose how you're going to send log data to Papertrail.

By default, Papertrail allows you to send log data to it via a Syslog call (https://en.wikipedia.org/wiki/Syslog). Syslog is a log messaging format that originated in UNIX. Syslog allows for the sending of log messages over both TCP and UDP. Papertrail will automatically define a Syslog port that you can use to write log messages to. For the purposes of this discussion, you'll use this default port. Figure 9.7 shows you the Syslog connect string that's automatically generated when you click on the "Add your first system" button shown in figure 9.6.

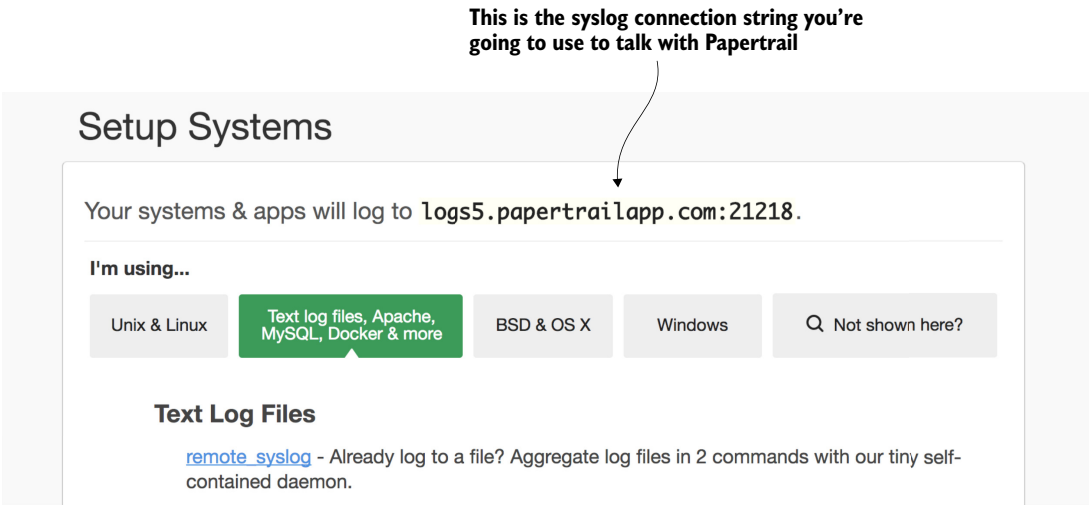


Figure 9.7 Papertrail uses Syslog as one of the mechanisms for sending data to it.

At this point you're all set up with Papertrail. You now have to configure your Docker environment to capture output from each of the containers running your services to the remote syslog endpoint defined in figure 9.7.

> **NOTE** The connection string from figure 9.7 is unique to my account. You'll need to make sure you use the connection string generated for you by Papertrail or define one via the Papertrail Settings > Log destinations menu option.

9.2.3 Redirecting Docker output to Papertrail

Normally, if you're running each of your services in their own virtual machine, you'll have to configure each individual service's logging configuration to send its logging information to a to a remote syslog endpoint (like the one exposed through Papertrail).

Fortunately, Docker makes it incredibly easy to capture all the output from any Docker container running on a physical or virtual machine. The Docker daemon communicates with all of the Docker containers it's managing through a Unix socket called `docker.sock`. Any container running on the server where Docker is running

can connect to the `docker.sock` and receive all the messages generated by all of the other containers running on that server. In the simplest terms, `docker.sock` is like a pipe that your containers can plug into and capture the overall activities going on within the Docker runtime environment on the virtual server the Docker daemon is running on.

You're going to use a "Dockerized" piece of software called Logspout (https://github.com/gliderlabs/logspout) that will listen to the `docker.sock` socket and then capture any standard out messages generated in Docker runtime and redirect the output to a remote syslog (Papertrail). To set up your Logspout container, you have to add a single entry to the docker-compose.yml file you use to fire up all of the Docker containers used for the code examples in this chapter. The docker/common/docker-compose.yml file you need to modify should have the following entry added to it:

```
logspout:
    image: gliderlabs/logspout
    command: syslog://logs5.papertrailapp.com:21218
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

NOTE In the previous code snippet, you'll need to replace the value in the "command" attribute with the value supplied to you from Papertrail. If you use the previous Logspout snippet, your Logspout container will happily write your log entries to my Papertrail account.

Now when you fire up your Docker environment in this chapter, all data sent to a container's standard output will be sent to Papertrail. You can see this for yourself by logging into your Papertrail account after you've started chapter 9's Docker examples and clicking on the Events button in the top right part of your screen.

Figure 9.8 shows an example of what the data sent to Papertrail looks like.

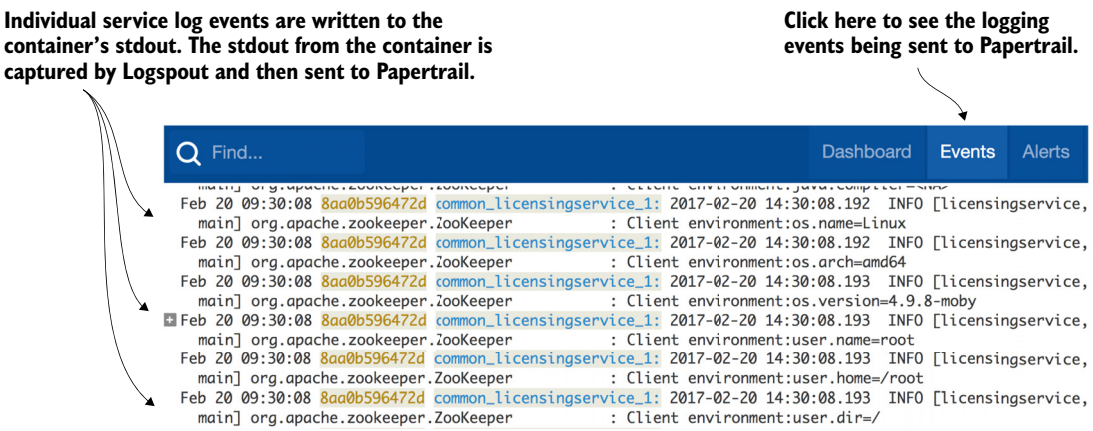


Figure 9.8 With the Logspout Docker container defined, data written to each container's standard out will be sent to Papertrail.

Why not use the Docker logging driver?

Docker 1.6 and above do allow you to define alternative logging drivers to write the stdout/stderr messages written from each container. One of the logging drivers is a syslog driver that can be used to write the messages to a remote syslog listener.

Why did I choose Logspout instead of using the standard Docker log driver? The main reason is flexibility. Logspout offers features for customizing what logging data gets sent to your log aggregation platform. The features Logspout offers include

- *The ability to send log data to multiple endpoints at once.* Many companies will want to send their log data to a log aggregation platform, and will also want security monitoring tools that will monitor the produced logs for sensitive data.
- *A centralized location for filtering which containers are going to send their log data.* With the Docker driver, you need to manually set the log driver for each container in your docker-compose.yml file. Logspout lets you define filters to specific containers and even specific string patterns in a centralized configuration.
- *Custom HTTP routes that let applications write log information via specific HTTP endpoints.* This feature allows you to do things like write specific log messages to a specific downstream log aggregation platform. For example, you might have general log messages from stdout/stderr go to Papertrail, where you might want to send specific application audit information to an in-house Elasticsearch server.
- *Integration with protocols beyond syslog.* Logspout allows you to send messages via UDP and TCP protocols. Logspout also has third-party modules that can integrate the stdout/stderr from Docker into Elasticsearch.

9.2.4 Searching for Spring Cloud Sleuth trace IDs in Papertrail

Now that your logs are flowing to Papertrail, you can really start appreciating Spring Cloud Sleuth adding trace IDs to all your log entries. To query for all the log entries related to a single transaction, all you need to do is take a trace ID and query for it in the query box of Papertrail's event screen. Figure 9.9 shows how to execute a query by the Spring Cloud sleuth trace ID we used earlier in section 9.1.2: `a9e3e1786b74d302..`

Consolidate logging and praise for the mundane

Don't underestimate how important it is to have a consolidated logging architecture and a service correlation strategy thought out. It seems like a mundane task, but while I was writing this chapter, I used log aggregation tools similar to Papertrail to track down a race condition between three different services for a project I was working on. It turned out that the race condition has been there for over a year, but the service with the race condition had been functioning fine until we added a bit more load and one other actor in the mix to cause the problem.

We found the issue only after spending 1.5 weeks doing log queries and walking through the trace output of dozens of unique scenarios. We wouldn't have found the problem without the aggregated logging platform that had been put in place. This experience reaffirmed several things:

1. *Make sure you define and implement your logging strategies early on in your service development*—Implementing logging infrastructure is tedious, sometimes difficult, and time-consuming once a project is well underway.
2. *Logging is a critical piece of microservice infrastructure*—Think long and hard before you implement your own logging solution or even try to implement an on-premise logging solution. Cloud-based logging platforms are worth the money that's spent on them.
3. *Learn your logging tools*—Almost every logging platform will have a query language for querying the consolidated logs. Logs are an incredible source of information and metrics. They're essentially another type of database, and the time you spend learning to query will pay huge dividends.

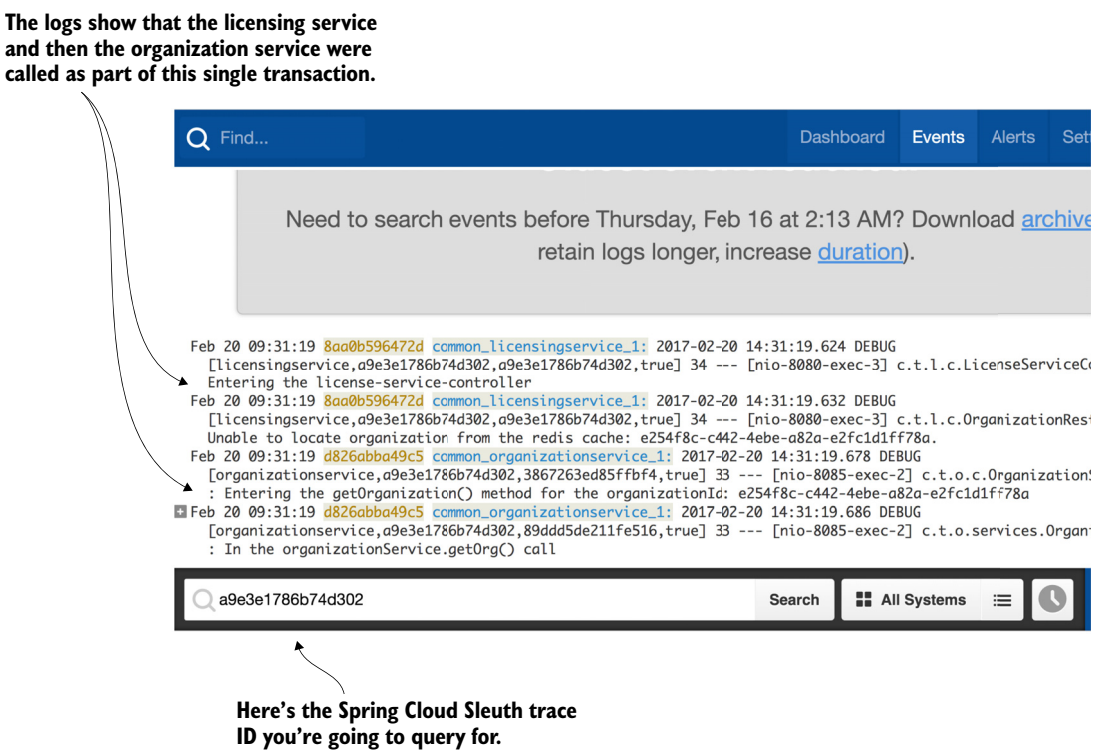


Figure 9.9 The trace ID allows you to filter all log entries related to that single transaction.

9.2.5 Adding the correlation ID to the HTTP response with Zuul

If you inspect the HTTP response back from any service call made with Spring Cloud Sleuth, you'll see that the trace ID used in the call is never returned in the HTTP response headers. If you inspect the documentation for Spring Cloud Sleuth, you'll see that the Spring Cloud Sleuth team believes that returning any of the tracing data can be a potential security issue (though they don't explicitly list their reasons why they believe this.)

However, I've found that the returning of a correlation or tracing ID in the HTTP response is invaluable when debugging a problem. Spring Cloud Sleuth does allow you to "decorate" the HTTP response information with its tracing and span IDs. However, the process to do this involves writing three classes and injecting two custom Spring beans. If you'd like to take this approach, you can see it in the Spring Cloud Sleuth documentation (http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.0.12.RELEASE/). A much simpler solution is to write a Zuul "POST" filter that will inject the trace ID in the HTTP response.

In chapter 6 when we introduced the Zuul API gateway, we saw how to build a Zuul "POST" response filter to add the correlation ID you generated for use in your services to the HTTP response returned by the caller. You're now going to modify that filter to add the Spring Cloud Sleuth header.

To set up your Zuul response filter, you need to add a single JAR dependencies to your Zuul server's pom.xml file: `spring-cloud-starter-sleuth`. The `spring-cloud-starter-sleuth` dependency will be used to tell Spring Cloud Sleuth that you want Zuul to participate in a Spring Cloud trace. Later in the chapter, when we introduce Zipkin, you'll see that the Zuul service will be the first call in any service invocation.

For chapter 9, this file can be found in zuulsvr/pom.xml. The following listing shows these dependencies.

Listing 9.1 Adding Spring Cloud Sleuth to Zuul

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Adding spring-cloud-starter-sleuth to Zuul will cause a trace ID to be generated for every service being called in Zuul

Once this new dependency is in place, the actual Zuul "post" filter is trivial to implement. The following listing shows the source code used to build the Zuul filter. The file is located in `zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/filters/ResponseFilter.java`.

Listing 9.2 Adding the Spring Cloud Sleuth trace ID via a Zuul POST filter

```
package com.thoughtmechanix.zuulsvr.filters;

//Rest of annotations removed for conciseness
import org.springframework.cloud.sleuth.Tracer;

@Component
public class ResponseFilter extends ZuulFilter{
private static final int     FILTER_ORDER=1;
private static final boolean  SHOULD_FILTER=true;
private static final Logger logger =
    ➥ LoggerFactory.getLogger(ResponseFilter.class);

@Autowired
Tracer tracer;

@Override
public String filterType() {return "post";}

@Override
public int filterOrder() {return FILTER_ORDER;}

@Override
public boolean shouldFilter() {return SHOULD_FILTER;}

@Override
public Object run() {
  RequestContext ctx = RequestContext.getCurrentContext();
  ctx.getResponse()
    ➥ .addHeader("tmx-correlation-id",
       ➥ tracer.getCurrentSpan().traceIdString());

  return null;
}}
```

The Tracer class is the entry point to access trace and span ID information.

You're going to add a new HTTP Response header called tmx-correlation-ID to hold the Spring Cloud Sleuth trace ID.

Because Zuul is now Spring Cloud Sleuth-enabled, you can access tracing information from within your `ResponseFilter` by autowiring in the `Tracer` class into the `ResponseFilter`. The `Tracer` class allows you to access information about the current Spring Cloud Sleuth trace being executed. The `tracer.getCurrentSpan()` `.traceIdString()` method allows you to retrieve as a String the current trace ID for the transaction underway.

It's trivial to add the trace ID to the outgoing HTTP response passing back through Zuul. This is done by calling

```
RequestContext ctx = RequestContext.getCurrentContext();
ctx.getResponse().addHeader("tmx-correlation-id",
      ➥ tracer.getCurrentSpan().traceIdString());
```

With this code now in place, if you invoke an EagleEye microservice through your Zuul gateway, you should get a HTTP response back called `tmx-correlation-id`.

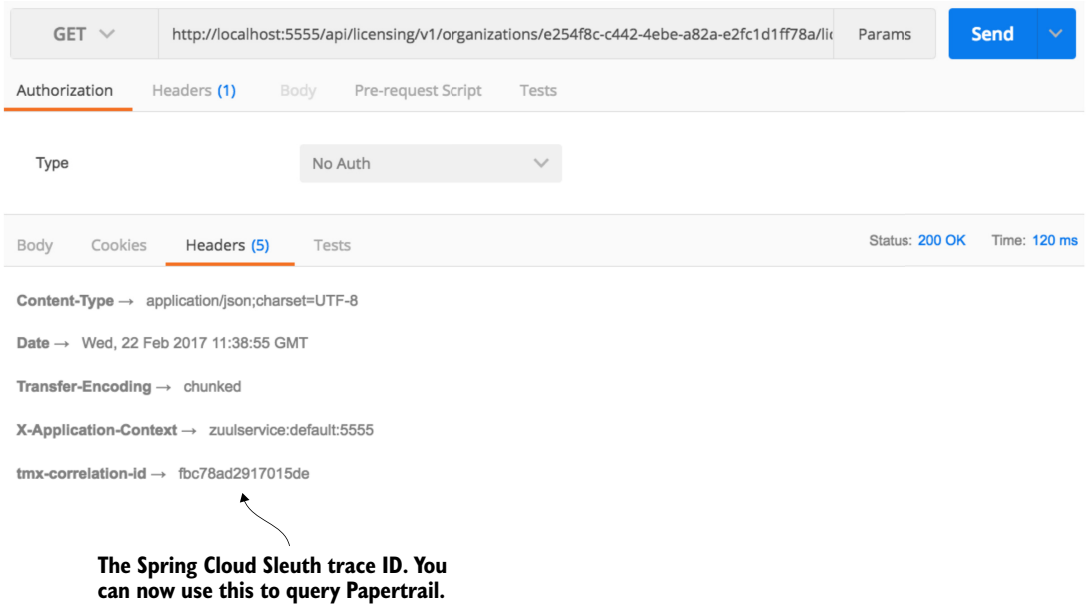


Figure 9.10 With the Spring Cloud Sleuth trace ID returned, you can easily query Papertrail for the logs.

Figure 9.10 shows the results of a call to `GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a`.

9.3 Distributed tracing with Open Zipkin

Having a unified logging platform with correlation IDs is a powerful debugging tool. However, for the rest of the chapter we're going to move away from tracing log entries and instead look at how to visualize the flow of transactions as they move across different microservices. A clean, concise picture can be work more than a million log entries.

Distributed tracing involves providing a visual picture of how a transaction flows across your different microservices. Distributed tracing tools will also give a rough approximation of individual microservice response times. However, distributed tracing tools shouldn't be confused with full-blown Application Performance Management (APM) packages. These packages can provide out-of-the-box, low-level performance data on the actual code within your service and can also provider performance data beyond response time, such as memory, CPU utilization, and I/O utilization.

This is where Spring Cloud Sleuth and the OpenZipkin (also referred to as Zipkin) project shine. Zipkin (http://zipkin.io/) is a distributed tracing platform that allows you to trace transactions across multiple service invocations. Zipkin allows you to graphically see the amount of time a transaction takes and breaks down the time spent in each microservice involved in the call. Zipkin is an invaluable tool for identifying performance issues in a microservices architecture.

Setting up Spring Cloud Sleuth and Zipkin involves four activities:

- Adding Spring Cloud Sleuth and Zipkin JAR files to the services that capture trace data
- Configuring a Spring property in each service to point to the Zipkin server that will collect the trace data
- Installing and configuring a Zipkin server to collect the data
- Defining the sampling strategy each client will use to send tracing information to Zipkin

9.3.1 Setting up the Spring Cloud Sleuth and Zipkin dependencies

Up to now you've included two sets of Maven dependencies to your Zuul, licensing, and organization services. These JAR files were the `spring-cloud-starter-sleuth` and the `spring-cloud-sleuth-core` dependencies. The `spring-cloud-starter-sleuth` dependencies are used to include the basic Spring Cloud Sleuth libraries needed to enable Spring Cloud Sleuth within a service. The `spring-cloud-sleuth-core` dependencies are used whenever you have to programmatically interact with Spring Cloud Sleuth (which you'll do again later in the chapter).

To integrate with Zipkin, you need to add a second Maven dependency called `spring-cloud-sleuth-zipkin`. The following listing shows the Maven entries that should be present in the Zuul, licensing, and organization services once the `spring-cloud-sleuth-zipkin` dependency is added.

Listing 9.3 Client-side Spring Cloud Sleuth and Zipkin dependences

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

9.3.2 Configuring the services to point to Zipkin

With the JAR files in place, you need to configure each service that wants to communicate with Zipkin. You do this by setting a Spring property that defines the URL used to communicate with Zipkin. The property that needs to be set is the `spring.zipkin.baseUrl` property. This property is set in each service's application.yml properties file.

> **NOTE** The `spring.zipkin.baseUrl` can also be externalized as a property in Spring Cloud Config.

In the application.yml file for each service, the value is set to `http://localhost:9411`. However, at runtime I override this value using the ZIPKIN_URI

(http://zipkin:9411) variable passed on each services Docker configuration (docker/common/docker-compose.yml) file.

Zipkin, RabbitMQ, and Kafka

Zipkin does have the ability to send its tracing data to a Zipkin server via RabbitMQ or Kafka. From a functionality perspective, there's no difference in Zipkin behavior if you use HTTP, RabbitMQ, or Kafka. With the HTTP tracing, Zipkin uses an asynchronous thread to send performance data. The main advantage to using RabbitMQ or Kafka to collect your tracing data is that if your Zipkin server is down, any tracing messages sent to Zipkin will be "enqueued" until Zipkin can pick up the data.

The configuration of Spring Cloud Sleuth to send data to Zipkin via RabbitMQ and Kafka is covered in the Spring Cloud Sleuth documentation, so we won't cover it here in any further detail.

9.3.3 Installing and configuring a Zipkin server

To use Zipkin, you first need to set up a Spring Boot project the way you've done multiple times throughout the book. (In the code for the chapter, this is call zipkinsvr.) You then need to add two JAR dependencies to the zipkinsvr/pom.xml file. These two jar dependences are shown in the following listing.

Listing 9.4 JAR dependencies needed for Zipkin service

```
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

This dependency contains the core classes for setting up the Zipkin server.

This dependency contains the core for classes for running the UI part of the Zipkin server.

@EnableZipkinServer vs. @EnableZipkinStreamServer: which annotation?

One thing to notice about the JAR dependencies above is that they're not Spring-Cloud-based dependencies. While Zipkin is a Spring-Boot-based project, the @EnableZipkinServer is not a Spring Cloud annotation. It's an annotation that's part of the Zipkin project. This often confuses people who are new to the Spring Cloud Sleuth and Zipkin, because the Spring Cloud team did write the @EnableZipkinStreamServer annotation as part of Spring Cloud Sleuth. The @EnableZipkinStreamServer annotation simplifies the use of Zipkin with RabbitMQ and Kafka.

I chose to use the @EnableZipkinServer because of its simplicity in setup for this chapter. With the @EnableZipkinStream server you need to set up and configure the services being traced and the Zipkin server to publish/listen to RabbitMQ

or Kafka for tracing data. The advantage of the `@EnableZipkinStreamServer` annotation is that you can continue to collect trace data even if the Zipkin server is unavailable. This is because the trace messages will accumulate the trace data on a message queue until the Zipkin server is available for processing the records. If you use the `@EnableZipkinServer` annotation and the Zipkin server is unavailable, the trace data that would have been sent by the service(s) to Zipkin will be lost.

After the Jar dependencies are defined, you now need to add the `@EnableZipkinServer` annotation to your Zipkin services bootstrap class. This class is located in `zipkinsvr/src/main/java/com/thoughtmechanix/zipkinsvr/ZipkinServerApplication.java`. The following listing shows the code for the bootstrap class.

Listing 9.5 Building your Zipkin servers bootstrap class

```
package com.thoughtmechanix.zipkinsvr;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import zipkin.server.EnableZipkinServer;

@SpringBootApplication
@EnableZipkinServer                          <-- The @EnableZipkinServer allows you to
public class ZipkinServerApplication {           quickly start Zipkin as a Spring Boot project.
public static void main(String[] args) {
  SpringApplication.run(ZipkinServerApplication.class, args);
  }
}
```

The key thing to note in this listing is the use of the `@EnableZipkinServer` annotation. This annotation enables you to start this Spring Boot service as a Zipkin server. At this point, you can build, compile, and start the Zipkin server as one of the Docker containers for the chapter.

Little configuration is needed to run a Zipkin server. One of the only things you're going to have to configure when you run Zipkin is the back end data store that Zipkin will use to store the tracing data from your services. Zipkin supports four different back end data stores. These data stores are

1. In-memory data
2. MySQL: http://mysql.com
3. Cassandra: http://cassandra.apache.org
4. Elasticsearch: http://elastic.co

By default, Zipkin uses an in-memory data store for storing tracing data. The Zipkin team recommends against using the in-memory database in a production system. The in-memory database can hold a limited amount of data and the data is lost when the Zipkin server is shut down or lost.

NOTE For the purposes of this book, you'll use Zipkin with an in-memory data store. Configuring the individual data stores used in Zipkin is outside of the scope of this book, but if you're interested in the topic, you can find more information at the Zipkin GitHub repository (https://github.com/openzipkin/zipkin/tree/master/zipkin-server).

9.3.4 Setting tracing levels

At this point you have the clients configured to talk to a Zipkin server and you have the server configured and ready to be run. You need to do one more thing before you start using Zipkin. You need to define how often each service should write data to Zipkin.

By default, Zipkin will only write 10% of all transactions to the Zipkin server. The transaction sampling can be controlled by setting a Spring property on each of the services sending data to Zipkin. This property is called `spring.sleuth.sampler.percentage`. The property takes a value between 0 and 1:

- A value of 0 means Spring Cloud Sleuth won't send Zipkin any transactions.
- A value of .5 means Spring Cloud Sleuth will send 50% of all transactions.

For our purposes, you're going to send trace information for all services. To do this, you can set the value of `spring.sleuth.sampler.percentage` or you can replace the default `Sampler` class used in Spring Cloud Sleuth with the `AlwaysSampler`. The `AlwaysSampler` can be injected as a Spring Bean into an application. For example, the licensing service has the `AlwaysSampler` defined as a Spring Bean in its `licensing-service/src/main/java/com/thoughtmechanix/licenses/Application.java` class as

```
@Bean
public Sampler defaultSampler() { return new AlwaysSampler();}
```

The Zuul, licensing, and organization services all have the `AlwaysSampler` defined in them so that in this chapter all transactions will be traced with Zipkin.

9.3.5 Using Zipkin to trace transactions

Let's start this section with a scenario. Imagine you're one of the developers on the EagleEye application and you're on-call this week. You get a support ticket from a customer who's complaining that one of the screens in the EagleEye application is running slow. You have a suspicion that the licensing service being used by the screen is running slow. But why and where? The licensing service relies on the organization service and both services make calls to different databases. Which service is the poor performer? Also, you know that these services are constantly being modified, so someone might have added a new service call into the mix. Understanding all the services that participate in the user's transaction and their individual performance times is critical to supporting a distributed architecture such as a microservice architecture.

You'll begin by using Zipkin to watch two transactions from your organization service as they're traced by the Zipkin service. The organization service is a simple service

that only makes a call to a single database. What you're going to do is use POSTMAN to send two calls to the organization service (`GET http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a`). The organization service calls will flow through a Zuul API gateway before the calls get directed downstream to an organization service instance.

After you've made two calls to the organization service, go to http://localhost:9411 and see what Zipkin has captured for trace results. Select the "organization service" from the dropdown box on the far upper left of the screen and then press the Find traces button. Figure 9.11 shows the Zipkin query screen after you've taken these actions.

Now if you look at the screenshot in figure 9.11, you'll see that Zipkin captured two transactions. Each of the transactions is broken down into one or more spans. In Zipkin, a span represents a specific service or call in which timing information is being captured. Each of the transactions in figure 9.11 has three spans captured in it: two spans in the Zuul gateway, and then a span for the organization service. Remember, the Zuul gateway doesn't blindly forward an HTTP call. It receives the incoming HTTP call, terminates the incoming call, and then builds a new call out to the targeted service (in this case, the organization service). This termination of the original call is how Zuul can add pre-, response, and post filters to each call entering the gateway. It's also why we see two spans in the Zuul service.

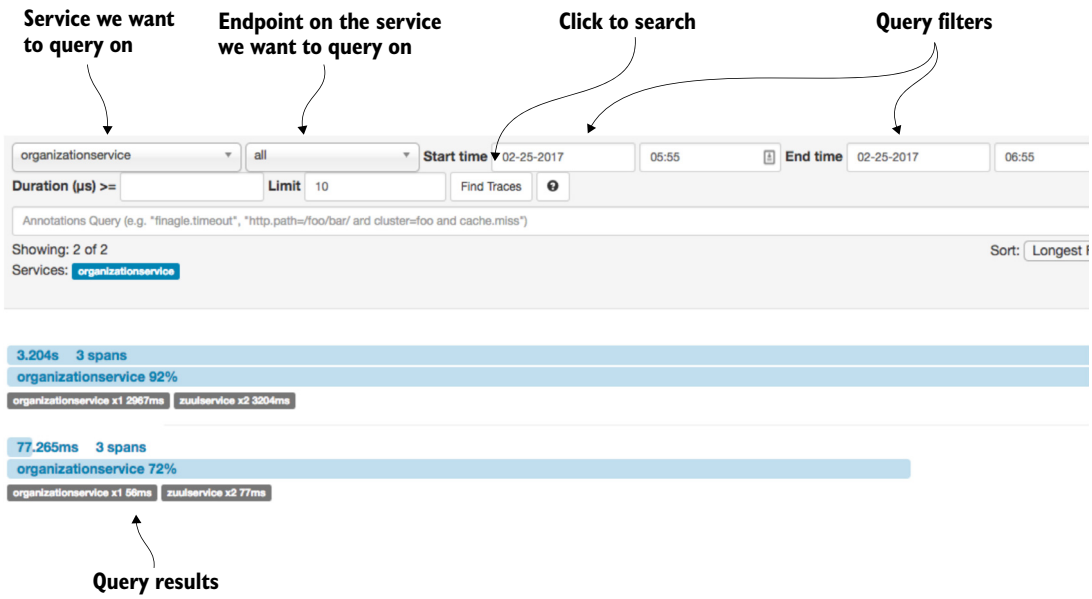


Figure 9.11 The Zipkin query screen lets you select the service you want to trace on, along with some basic query filters.

The two calls to the organization service through Zuul took 3.204 seconds and 77.2365 milliseconds respectively. Because you queried on the organization service calls (and not the Zuul gateway calls), you can see that the organization service took 92% and 72% of the total amount of time of the transaction time.

Let's dig into the details of the longest running call (3.204 seconds). You can see more detail by clicking on the transaction and drilling into the details. Figure 9.12 shows the details after you've clicked to drill down into further details.

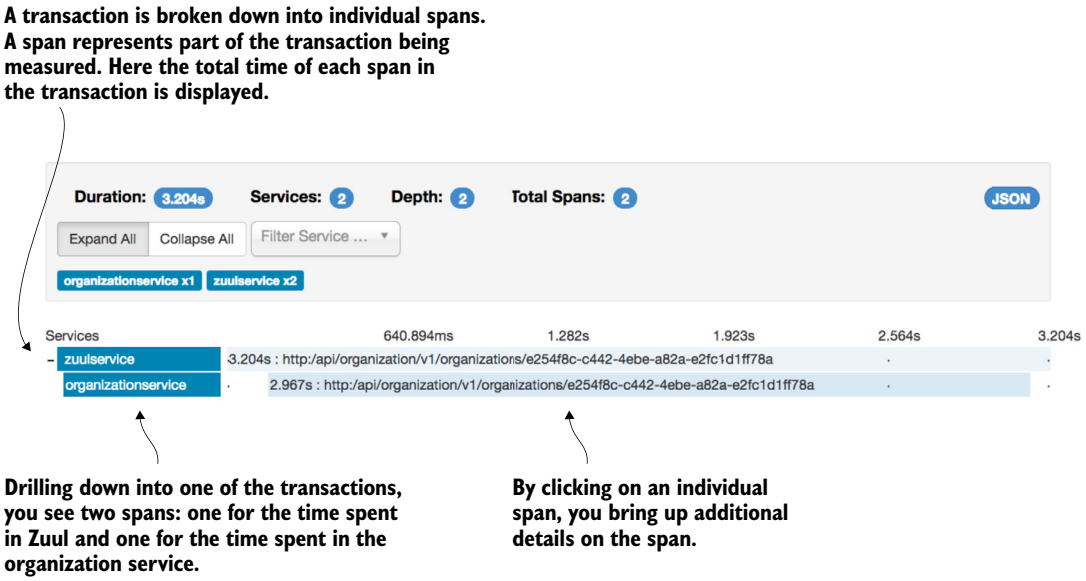


Figure 9.12 Zipkin allows you to drill down and see the amount of time each span in a transaction takes.

In figure 9.12 you can see that the entire transaction from a Zuul perspective took approximately 3.204 seconds. However, the organization service call made by Zuul took 2.967 seconds of the 3.204 seconds involved in the overall call. Each span presented can be drilled down into for even more detail. Click on the `organizationservice` span and see what additional details can be seen from the call. Figure 9.13 shows the detail of this call.

One of the most valuable pieces of information in figure 9.13 is the breakdown of when the client (Zuul) called the organization service, when the organization service received the call, and when the organization service responded back. This type of timing information is invaluable in detecting and identifying network latency issues.

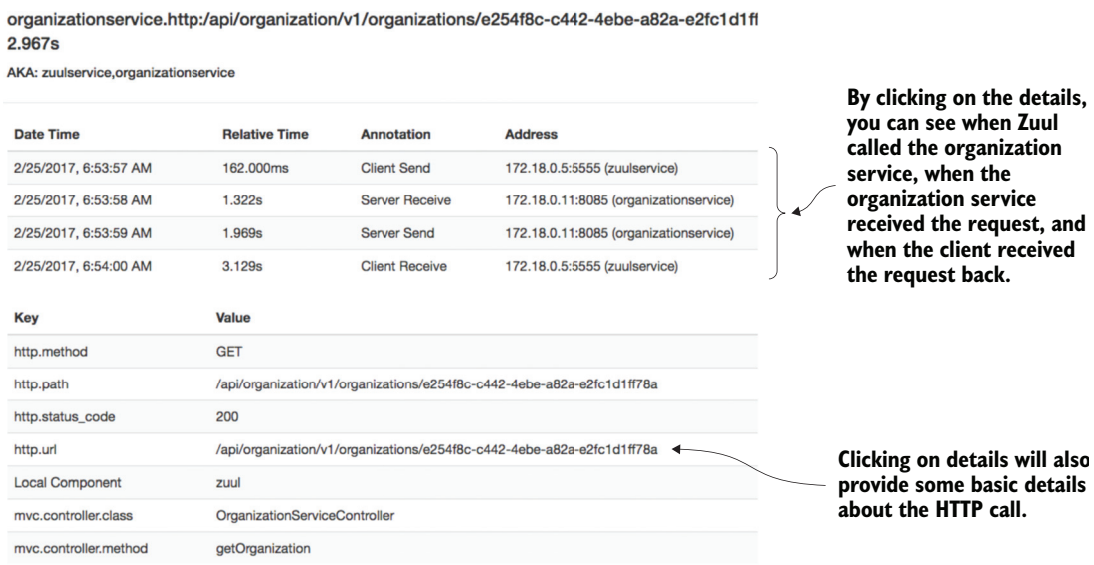


Figure 9.13 Clicking on an individual span gives further details on call timing and the details of the HTTP call.

9.3.6 Visualizing a more complex transaction

What if you want to understand exactly what service dependencies exist between service calls? You can call the licensing service through Zuul and then query Zipkin for licensing service traces. You can do this with a GET call to the licensing services `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a` endpoint.

Figure 9.14 shows the detailed trace of the call to the licensing service.

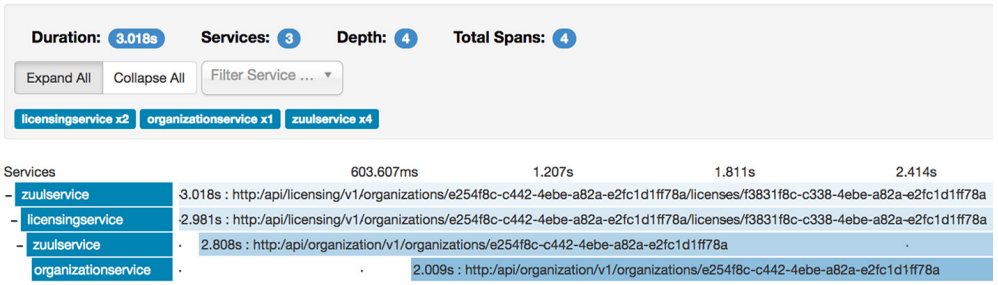


Figure 9.14 Viewing the details of a trace of how the licensing service call flows from Zuul to the licensing service and then through to the organization service

In figure 9.14, you can see that the call to the licensing service involves 4 discrete HTTP calls. You see the call to the Zuul gateway and then from the Zuul gateway to the licensing service. The licensing service then calls back through Zuul to call the organization service.

9.3.7 Capturing messaging traces

Spring Cloud Sleuth and Zipkin don't trace HTTP calls. Spring Cloud Sleuth also sends Zipkin trace data on any inbound or outbound message channel registered in the service.

Messaging can introduce its own performance and latency issues inside of an application. A service might not be processing a message from a queue quickly enough. Or there could be a network latency problem. I've encountered all these scenarios while building microservice-based applications.

By using Spring Cloud Sleuth and Zipkin, you can identify when a message is published from a queue and when it's received. You can also see what behavior takes place when the message is received on a queue and processed.

As you'll remember from chapter 8, whenever an organization record is added, updated, or deleted, a Kafka message is produced and published via Spring Cloud Stream. The licensing service receives the message and updates a Redis key-value store it's using to cache data.

Now you'll go ahead and delete an organization record and watch the transaction be traced by Spring Cloud Sleuth and Zipkin. You can issue a `DELETE http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a` via POSTMAN to the organization service.

Remember, earlier in the chapter we saw how to add the trace ID as an HTTP response header. You added a new HTTP response header called `tmx-correlation-id`. In my call, I had the `tmx-correlation-id` returned on my call with a value of `5e14cae0d90dc8d4`. You can search Zipkin for this specific trace ID by entering the trace ID returned by your call via the search box in the upper-right hand corner of the Zipkin query screen. Figure 9.15 shows where you can enter the trace ID.

Figure 9.15 With the trace ID returned in the HTTP Response `tmx-correlation-id` field you can easily find the transaction you're looking for.

With the trace ID in hand you can query Zipkin for the specific transaction and can see the publication of a delete message to your output message change. This message channel, `output`, is used to publish to a Kafka topic call `orgChangeTopic`. Figure 9.16 shows the `output` message channel and how it appears in the Zipkin trace.

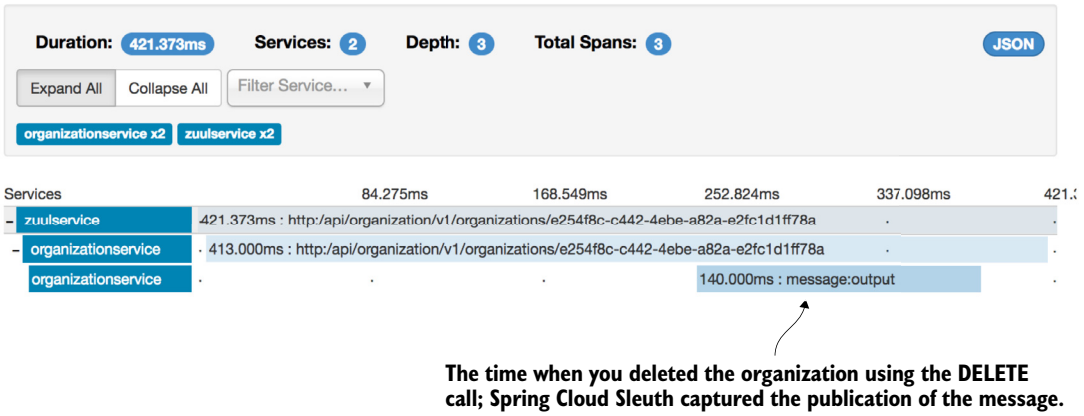


Figure 9.16 Spring Cloud Sleuth will automatically trace the publication and receipt of messages on Spring message channels.

You can see the licensing service receive the message by querying Zipkin and looking for the received message. Unfortunately, Spring Cloud Sleuth doesn't propagate the trace ID of a published message to the consumer(s) of that message. Instead, it generates a new trace ID. However, you can query Zipkin server for any license service transactions and order them by newest message. Figure 9.17 shows the results of this query.

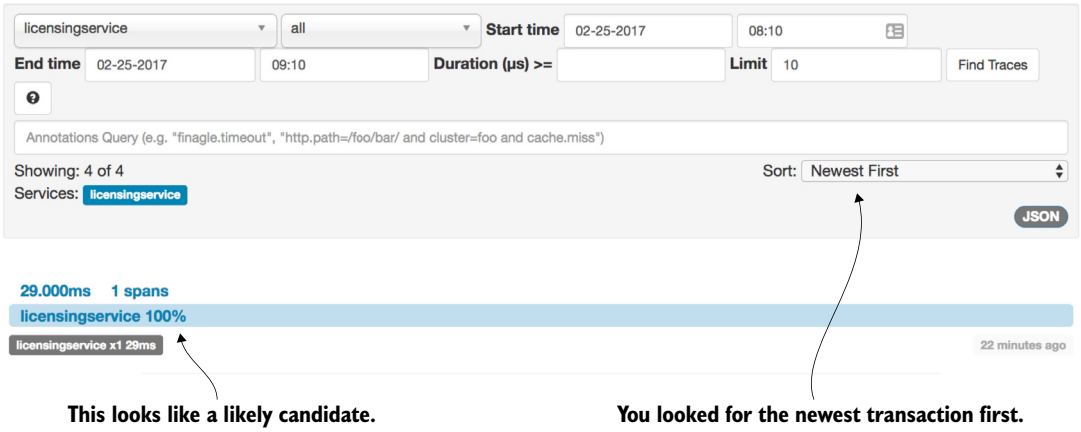


Figure 9.17 You're looking for the licensing service invocation where a Kafka message is received.

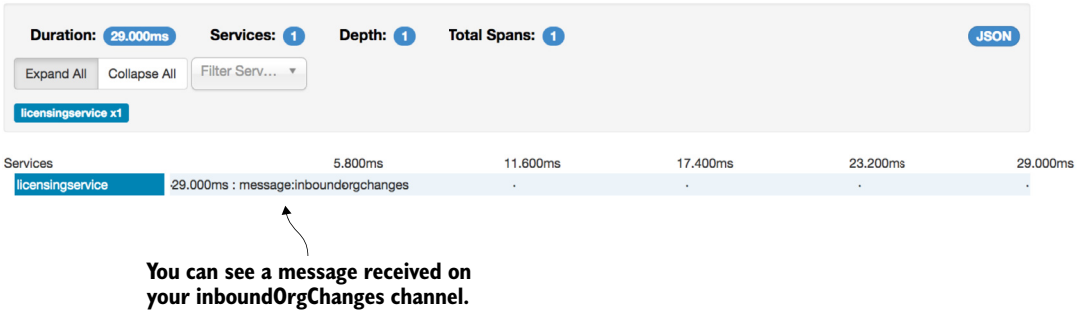


Figure 9.18 Using Zipkin you can see the Kafka message being published by the organization service.

Now that you've found your target licensing service transaction, you can drill down into the transaction. Figure 9.18 shows the results of this drilldown.

Until now you've used Zipkin to trace your HTTP and messaging calls from within your services. However, what if you want to perform traces out to third-party services that aren't instrumented by Zipkin? For example, what if you want to get tracing and timing information for a specific Redis or Postgres SQL call? Fortunately, Spring Cloud Sleuth and Zipkin allow you to add custom spans to your transaction so that you can trace the execution time associated with these third-party calls.

9.3.8 Adding custom spans

Adding a custom span is incredibly easy to do in Zipkin. You can start by adding a custom span to your licensing service so that you can trace how long it takes to pull data out of Redis. Then you're going to add a custom span to the organization service to see how long it takes to retrieve data from your organization database.

To add a custom span to the licensing service's call to Redis, you're going to instrument the `licensing-service/src/main/java/com/thoughtmechanix/licenses/clients/OrganizationRestTemplateClient.java` class. In this class you're going to instrument the `checkRedisCache()` method. The following listing shows this code.

Listing 9.6 Instrumenting the call to read licensing data from Redis

```
import org.springframework.cloud.sleuth.Tracer;

//Rest of imports removed for conciseness
@Component
public class OrganizationRestTemplateClient {
    @Autowired
    RestTemplate restTemplate;

    @Autowired
    Tracer tracer;

    @Autowired
    OrganizationRedisRepository orgRedisRepo;
```

The Tracer class is used to programmatically access the Spring Cloud Sleuth trace information.

```
    private static final Logger logger =
      ➥ LoggerFactory
        .getLogger(OrganizationRestTemplateClient.class);

 private Organization checkRedisCache(String organizationId) {
   Span newSpan = tracer.createSpan("readLicensingDataFromRedis");
   try {
     return orgRedisRepo.findOrganization(organizationId);
   }
   catch (Exception ex){
     logger.error("Error encountered while
         ➥ trying to retrieve organization
         ➥ {} check Redis Cache.  Exception {}",
         ➥ organizationId, ex);
     return null;
   }
   finally {
     newSpan.tag("peer.service", "redis");
     newSpan.logEvent(
        org.springframework.cloud.sleuth.Span.CLIENT_RECV);
     tracer.close(newSpan);
   }
 }

    //Rest of class removed for conciseness
}
```

For your custom span, create a new span called "readLicensingDataFromRedis".

You can add tag information to the span. In this class you provide the name of the service that's going to be captured by Zipkin

Close the span out with a finally block.

Log an event to tell Spring Cloud Sleuth that it should capture the time when the call is complete.

Close out the trace. If you don't call the close() method, you'll get error messages in the logs indicating that a span has been left open

The code in listing 9.6 creates a custom span called `readLicensingDataFromRedis`. Now you'll also add a custom span, called `getOrgDbCall`, to the organization service to monitor how long it takes to retrieve organization data from the Postgres database. The trace for organization service database calls can be seen in the `organization-service/src/main/java/com/thoughtmechanix/organization/services/OrganizationService.java` class. The method containing the custom trace is the `getOrg()` method call.

The following listing shows the source code from the organization service's `getOrg()` method.

Listing 9.7 The instrumented `getOrg()` method

```
package com.thoughtmechanix.organization.services;

//Removed the imports for conciseness
@Service
public class OrganizationService {
  @Autowired
  private OrganizationRepository orgRepository;

  @Autowired
  private Tracer tracer;
```

```
 @Autowired
 SimpleSourceBean simpleSourceBean;

 private static final Logger logger =
    ➥ LoggerFactory.getLogger(OrganizationService.class);

 public Organization getOrg (String organizationId) {
Span newSpan = tracer.createSpan("getOrgDBCall");

logger.debug("In the organizationService.getOrg() call");
try {
  return orgRepository.findById(organizationId);
}finally{
  newSpan.tag("peer.service", "postgres");
  newSpan
     .logEvent(
        org.springframework.cloud.sleuth.Span.CLIENT_RECV);
  tracer.close(newSpan);
 }
}

    //Removed the code for conciseness
}
```

With these two custom spans in place, restart the services and then hit the GET `http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442 -4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a` endpoint. If we you look at the transaction in Zipkin, you should see the addition of the two additional spans. Figure 9.19 shows the additional custom spans added when you call the licensing service endpoint to retrieve licensing information.

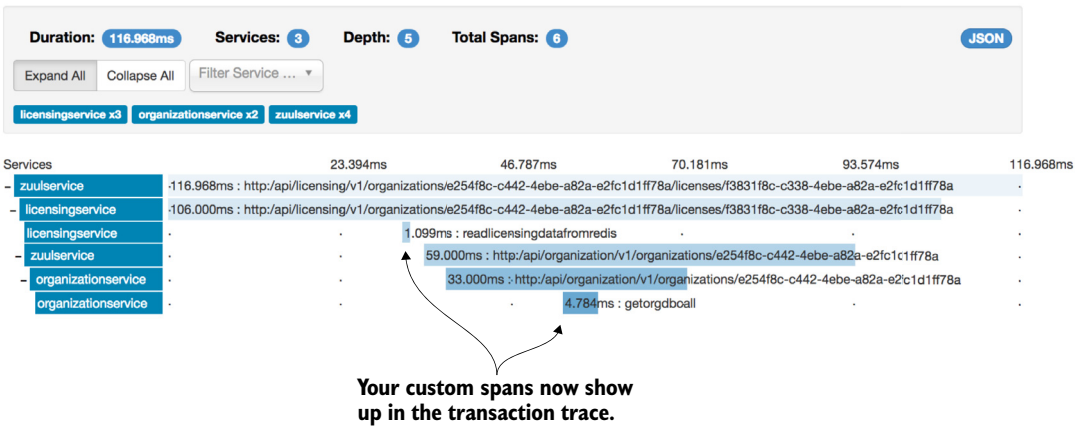
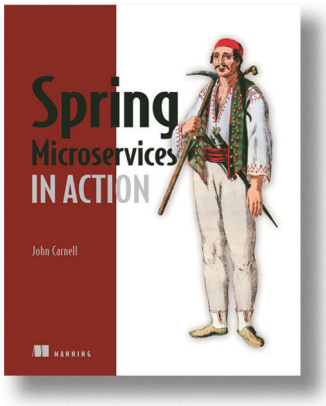


Figure 9.19 With the custom spans defined, they'll now show up in the transaction trace.

From figure 9.19 you can now see additional tracing and timing information related to your Redis and database lookups. You can break out that the read call to Redis took 1.099 milliseconds. Since the call didn't find an item in the Redis cache, the SQL call to the Postgres database took 4.784 milliseconds.

9.4 Summary

- Spring Cloud Sleuth allows you to seamlessly add tracing information (correlation ID) to your microservice calls.
- Correlation IDs can be used to link log entries across multiple services. They allow you to see the behavior of a transaction across all the services involved in a single transaction.
- While correlation IDs are powerful, you need to partner this concept with a log aggregation platform that will allow you to ingest logs from multiple sources and then search and query their contents.
- While multiple on-premise log aggregation platforms exist, cloud-based services allow you to manage your logs without having to have extensive infrastructure in place. They also allow you to easily scale as your application logging volume grows.
- You can integrate Docker containers with a log aggregation platform to capture all the logging data being written to the containers stdout/stderr. In this chapter, you integrated your Docker containers with Logspout and an online cloud logging provider, Papertrail, to capture and query your logs.
- While a unified logging platform is important, the ability to visually trace a transaction through its microservices is also a valuable tool.
- Zipkin allows you to see the dependencies that exist between services when a call to a service is made.
- Spring Cloud Sleuth integrates with Zipkin. Zipkin allows you to graphically see the flow of your transactions and understand the performance characteristics of each microservice involved in a user's transaction.
- Spring Cloud Sleuth will automatically capture trace data for an HTTP call and inbound/outbound message channel used within a Spring Cloud Sleuth enabled service.
- Spring Cloud Sleuth maps each of the service call to the concept of a span. Zipkin allows you to see the performance of a span.
- Spring Cloud Sleuth and Zipkin also allow you to define your own custom spans so that you can understand the performance of non-Spring-based resources (a database server such as Postgres or Redis).



Microservices break up your code into small, distributed, and independent services that require careful forethought and design. Fortunately, Spring Boot and Spring Cloud simplify your microservice applications, just as the Spring Framework simplifies enterprise Java development. Spring Boot removes the boilerplate code involved with writing a REST-based service. Spring Cloud provides a suite of tools for the discovery, routing, and deployment of microservices to the enterprise and the cloud.

Spring Microservices in Action teaches you how to build microservice-based applications using Java and the Spring platform. You'll learn to do microservice design as you build and deploy your first Spring Cloud application. Throughout the book, carefully selected real-life examples expose microservice-based patterns for configuring, routing, scaling, and deploying your services. You'll see how Spring's intuitive tooling can help augment and refactor existing applications with microservices.

What's inside:

- Core microservice design principles
- Managing configuration with Spring Cloud Config
- Client-side resiliency with Spring, Hystrix, and Ribbon
- Intelligent routing using Netflix Zuul
- Deploying Spring Cloud applications

This book is written for developers with Java and Spring experience.

Testing Using Canned Responses

Testing is a critical part of a microservices-based application. Writing a microservices-based application involves testing the expected behavior of the user's transaction in an application as it is executed across multiple microservices. The ability to easily "mock" and simulate dependencies between multiple microservices means that you can focus on testing a specific microservice and its interactions with other microservices without having to have the entire application (and its corresponding services running).

This chapter will demonstrate how to use the microservice mocking tool, Mountebank, to cleanly mock-out the HTTP-based interface of a downstream microservice call without having to modify the microservice you are testing in any way. Mountebank is a testing server that allows you imitate a microservice dependency without using code-base based mock stubs. This approach is powerful because it allows you to test your service in its natural state without having to actually rundown stream services.

Chapter 3 from *Testing Microservices with Mountebank* by Brandon Byars.

Testing Using Canned Responses

This chapter covers

- The `is` response type, which is the fundamental building block for a stub
- Using `is` responses in secure scenarios, with HTTPS servers and mutual authentication
- Persisting your imposter configuration using file templates

During a famous US White House scandal of the 1990's, Bill Clinton defended his prior statements by saying "It depends on what the meaning of *is* is." Although the grand jury and politicians failed to ultimately come to an agreement on the question, fortunately mountebank has no uncertainty on the matter.

It turns out that *is* is quite possibly the most important, and the most foundational, concept in all of mountebank. An imposter, capturing the core idea of binding a protocol to a port, might beg to differ, but by itself an imposter adds little to a

testing strategy. A response that looks like the real response—a response that, as far as the system under test's concerned, *is* the real response—changes everything. *Is* is the the key to being fake. Without *is*, a service binding a protocol to a port's a lame beast at best. Adding the ability to respond, and to respond as if the service *is* the real service, turns that service into a genuinely useful imposter.

In mountebank, the `is` response type's how you create canned responses;, responses that simulate a real response in some static way that you configure. It's one of three response types (`proxy` and `inject` being the other two), but the most important one. In this chapter, we'll explore `is` responses both by using the REST API and by persisting them in configuration files.

We'll also start to layer in key security concerns. Although all of our examples to date assume HTTP, the reality's that any serious web-based service built today uses HTTPS, layering on transport layer security (TLS) to the HTTP protocol. Because security—particularly authentication—is generally one of the first aspects of any microservice implementation we run into when writing tests, we'll look at using an HTTPS server that uses certificates for validating the client.

Finally, we'll explore how to persist imposter configuration. As you've no doubt realized by now, stubbing out services over the wire can be significantly more verbose than stubbing out objects in-process, and finding a way to lay out that configuration in a maintainable way's essential to use service virtualization to shift tests earlier in the development lifecycle.

3.1 The Basics of Canned Responses

It's a bit rude for any book on software development to skip out on the customary "Hello, world" example.[1] Let's see what a "Hello, world" response looks like in HTTP:

Listing 3.1 Hello World in an HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, world!
```

As we saw in the last chapter, returning this response in mountebank's as simple as translating the response we want into the appropriate JSON structure.

Listing 3.2 The HTTP response structure in JSON

```
{
  "statusCode": 200,
  "headers": { "Content-Type": "text/plain" },
  "body": "Hello, world!"
}
```

[1] Brian Kernighan and Dennis Ritchie showed how to print "Hello, world!" to the terminal in their venerable book "The C Programming Language." It's become a common introductory example.

To create an HTTP imposter, listening on port 3000, that returns this response, save the following in a `helloWorld.json` file:

Listing 3.3 The imposter configuration to respond with Hello, world!

```
{
  "protocol": "http",
  "port": 3000,
  "stubs": [{
    "responses": [{
      "is": {
        "statusCode": 200,
        "headers": { "Content-Type": "text/plain" },
        "body": "Hello, world!"
      }
    }]
  }]
}
```

The protocol defines the response structure

Tells mountebank to use an `is` response

Defines the canned response to be translated into HTT

We represent the JSON response we want from Listing 3.2 inside the `is` response, and expect mountebank to translate that to the HTTP shown in Listing 3.1 because we've set the `protocol` to http. With `mb` running, we can send an HTTP POST to localhost:2525/imposters to create this imposter. We'll use the `curl` command, introduced in chapter two, to send the HTTP request.[2]

```
curl -d@helloWorld.json http://localhost:2525/imposters
```

The `-d@` command line switch reads the file that follows and sends the contents of that file as an HTTP POST body. We can verify that the imposter has been created correctly by sending any HTTP request we want to port 3000.[3]

```
curl -i http://localhost:3000/any/path?query=does-not-matter
```

The response's almost, *but not quite*, the simple Hello World response shown in Listing 3.1:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: close
Date: Wed, 08 Feb 2017 01:42:38 GMT
Transfer-Encoding: chunked

Hello, world!
```

Three additional HTTP headers somehow crept in. Understanding where these headers came from requires us to revisit a concept we described in chapter one as the *default response*.

[2] Feel free to follow along using Postman or some graphical REST client. The examples are also available at github.com/bbyars/mountebank-in-action.

[3] In the examples that follow, I'll continue to use the `-i` command line parameter for `curl`. This tells `curl` to print the response headers to the terminal.

3.1.1 The Default Response

You may recall the following diagram, which describes how mountebank selects which response to return based on the response:

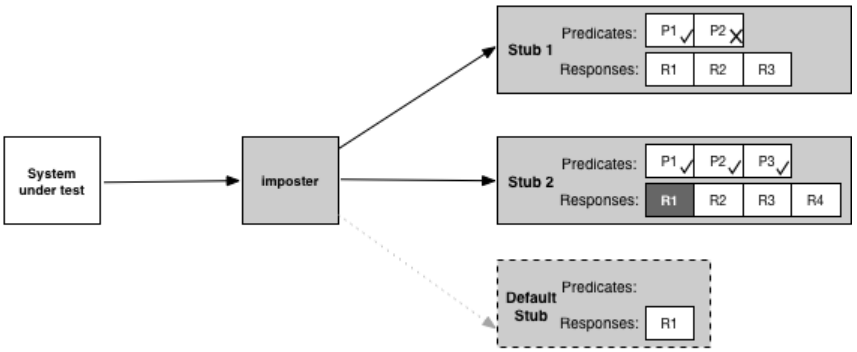


Figure 3.1 How mountebank selects a response

This diagram implies that, if the request doesn't match any predicate, there's a hidden "default stub" that will be used. That default stub contains no predicates, and it always matches the request, which contains exactly one response—the default response. We can see this default response if we create an imposter without any stubs.

```
curl http://localhost:2525/imposters --data '
{
  "protocol": "http",
  "port": 3000
}'
```

We haven't defined *any* responses with that lame beast of an imposter, only that we want an HTTP server listening on port 3000.

> **TIP** Because we're using port 3000 across multiple examples, you may find that you need to shut down and restart mountebank between examples to avoid a port conflict. Alternatively, you can use the API to clean up the previous imposter(s) by sending an HTTP `DELETE` command to localhost:2525/imposters (to remove all existing imposters) or to localhost:2525/imposters/3000 (to remove the imposter on port 3000). If you're using `curl`, the command's `curl -X DELETE` localhost:2525/imposters.

If we send any HTTP request to that port, we get the default response:

Listing 3.4 The default response in mountebank

```
HTTP/1.1 200 OK
Connection: close
Date: Wed, 08 Feb 2017 02:04:17 GMT
Transfer-Encoding: chunked
```

We looked at the first line of the response in chapter two; the 200 status code indicates that the request was processed successfully. The `Date` header's a standard response header that any responsible HTTP server sends, providing the server's understanding of the current date and time. The other two headers require a bit more explanation.

HTTP connections: To reuse or not to reuse?

HTTP's an application protocol built on top of the hard work of a few lower level network protocols, the most important of which (for our purposes) is TCP. TCP's responsible for establishing the connection between the client and the server through a series of messages that are often referred to as the "TCP handshake."

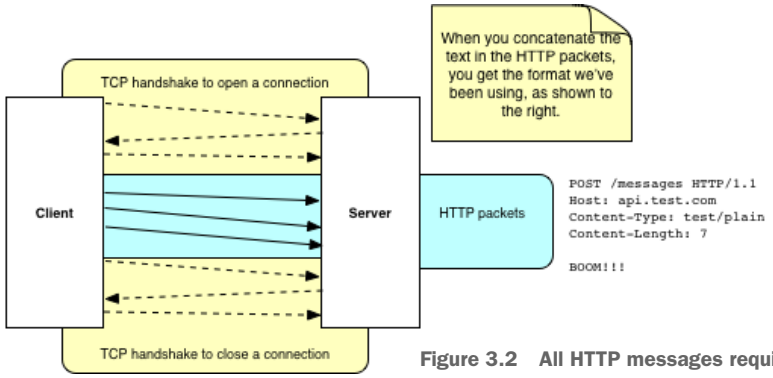


Figure 3.2 All HTTP messages require a TCP connection

Although those dashed lines representing the TCP messages to establish the connection are necessary, they aren't necessary for *every* request. Once the connection's established, the client and server could reuse it for multiple HTTP messages, which is important particularly for websites that need to serve HTML, JavaScript, CSS, and a set of images, each of which is a round-trip between the client and server.

HTTP supports "keepalive connections" as a performance optimization. A server tells the client to keep the connection open by setting the Connection header to "Keep-Alive". Mountebank defaults it to "Close," which tells the client to negotiate the TCP handshake for every request. If you're writing service tests, performance likely doesn't matter, and you may prefer the determinism that comes with a fresh connection for each request. If you're writing performance tests, or ensuring that your application behaves well with keepalive connections, then you should definitely change the default.

Knowing where an HTTP body ends

Notice in Figure 3.2 that a single HTTP request may consist of multiple packets (the operating system breaks up data into a series of "packets" to optimize sending them over the network). The same's true of a server response: what looks to be a single response may get transmitted in multiple packets. A consequence of this is that clients and servers need some way of knowing when an HTTP message's complete. With

headers it's easy: the headers end when you get a blank header line, but there's no way to predict where blank lines will occur in HTTP bodies, and a different strategy's needed. HTTP provides two strategies.

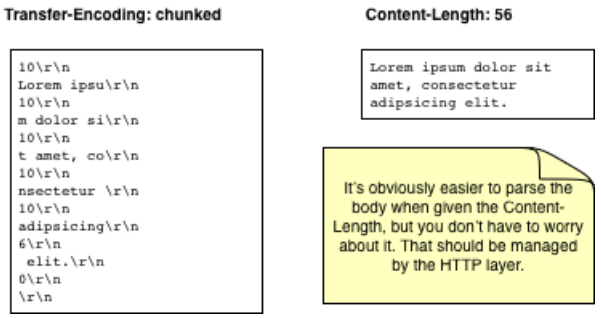


Figure 3.3 Using chunked encodings or content length to calculate where the body ends

The default imposter behavior sets the "Transfer-Encoding: chunked" header, which breaks the body into a series of "chunks," and prefixes each chunk with the number of bytes in the chunk. Special formatting delineates each chunk, making parsing relatively easy. The advantage of sending the body a chunk at a time's that the server can start streaming data to the client before the server has all the data itself. The alternative strategy's to calculate the length of the entire HTTP body before sending it and provide that information in the header. To select that strategy, the server sets a "Content-Length" header to the number of bytes in the body. Mountebank imposters must choose one strategy, but there's no reason to use chunked encoding other than the default in the web framework mountebank uses. The two strategies are mutually exlusive, and if you need to set the Content-Length header, the Transfer-Encoding header won't be set.

3.1.2 Understanding how the default response works

Now that you've seen what the default response looks like, it's probably a good time to admit that there's no such thing as a default stub in mountebank. This is a bald-face lie. I'm sorry—I *did* feel a little guilty writing it—but it's a useful simplification for situations where no stub matches the request. And, in case you haven't noticed yet, lying's exactly what mountebank does.

The reality's that mountebank merges the default response into *any* response you provide. Not providing a response's the same as providing an empty response, which is why we see the purest form of the default response in Listing 3.4, but we could also provide a partial response. For example, the following response structure doesn't provide all of the response fields:

```
{
  "is": {
    "body": "Hello, world!"
  }
}
```

Not to worry. Mountebank still returns a full response, helpfully filling in the blanks for you:

```
HTTP/1.1 200 OK
Connection: close
Date: Sun, 12 Feb 2017 17:38:39 GMT
Transfer-Encoding: chunked

Hello, world!
```

3.1.3 Changing the default response

The ability for mountebank to merge in defaults for the response's a pleasant convenience. As shown above, it means you only need to specify the fields which are different than the defaults, simplifying the response configuration. That's only useful if the defaults represent what you typically want. Fortunately, mountebank allows you to change the default response to better suit your needs.

Imagine a test suite that only wants to test error paths. We can default the `statusCode` to a 400 ("Bad Request") to avoid having to specify it in each response. Although we can't get rid of the Date header, we'll go ahead and change the other default headers to use keepalive connections and set the Content-Length header.

Listing 3.5 Changing the default response

```
{
  "protocol": "http",
  "port": 3000,
  "defaultResponse": {
    "statusCode": 400,
    "headers": {
      "Connection": "Keep-Alive",
      "Content-Length": 0
    }
  },
  "stubs": [{
    "responses": [{
      "is": { "body": "BOOM!!!" }
    }]
  }]
}
```

Changes the built-in default response for this imposter only

Defaults to a Bad Request

Adds or changes default headers

Use keepalive connections

Set the Content-Length header.

The response details are merged into the default response

If we send a test request to the imposter, it merges in the new default fields to the response:

Listing 3.6 A response using the new defaults

```
HTTP/1.1 400 Bad Request
Connection: Keep-Alive
Content-Length: 7
Date: Fri, 17 Feb 2017 16:29:00 GMT

BOOM!!!
```

400 comes from our default status code

We're now using keepalive connections

The Content-Length value was corrected from 0 to 7, and the Transfer-Encoding header's gone

The Date header remains from the original default response

The body from our `is` response's merged in

Notice in particular that the Content-Length header was set to the correct value. Mountebank imposters won't send out invalid HTTP responses.

3.1.4 Cycling through responses

Let's imagine one more test scenario: this time we'll test what happens when we submit an order through an HTTP `POST` to an Order service. Part of the order submission process involves checking to make sure there's sufficient inventory. The tricky part, from a testing perspective, is that inventory's sold and restocked—it doesn't stay static for the same product. This means that the exact same request to the Inventory service can respond with a different result each time.

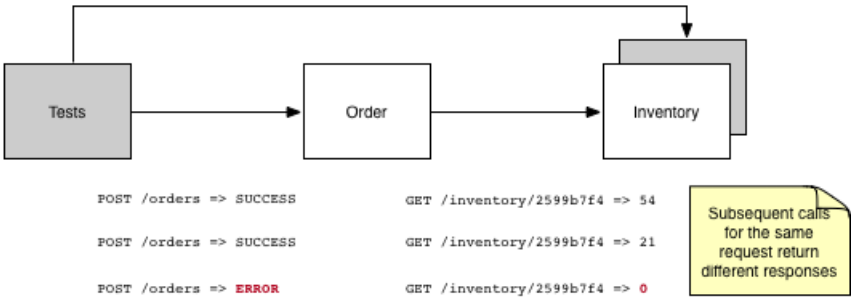


Figure 3.4 Inventory checks return volatile results for the same request

In chapter two, we saw a similar example paging the Product Catalog service, which returned different responses for the same path. In that example we were able to use different predicates to determine which response to send based on the `page` query parameter. In the inventory example, there's nothing about the request that allows us to select one response over the other.

What we need's a way to cycle through a set of responses to simulate the volatility of the on-hand inventory for a fast-selling product. The solution's to use the fact that each stub contains a *list* of responses. Mountebank returns those responses in the order provided.[4]

Listing 3.7 Returning a list of responses for the same stub

```
{
  "port": 3000,
  "protocol": "http",
  "stubs": [
    {
      "responses": [
```

[4] Note that in the following example and several others through the book, I'll use an overly simplified response to save space and remove some of the noise. No self-respecting Inventory service would ever return only a single number, but it makes the intent of the example stand out more easily, allowing you to focus on the fact that *some* data's different for each response.

```
        { "is": { "body": "54" } },
        { "is": { "body": "21" } },
        { "is": { "body": "0" } }
      ]
    }
  ]
}
```

These responses are returned in order

The first call returns 54, the second call returns 21, and the third call returns 0. If our tests need to trigger a fourth call, it will once again return 54, then 21, and 0 again. Mountebank treats the list of responses as an infinite list, with the first and last entries connected like a circle, a data structure called a circular buffer.

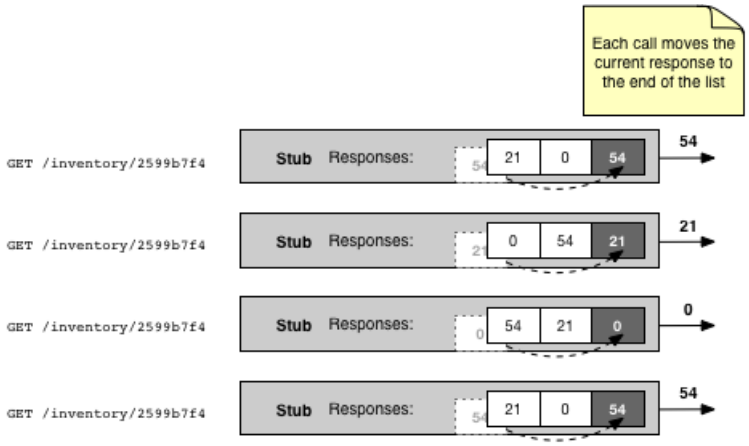


Figure 3.5 Inventory checks return volatile results for the same request

As shown in Figure 3.5, the illusion of an infinite list's maintained by shifting each response to the end of the list when it's returned. You can cycle through them as many times as you need.

In chapter seven, we'll look at all kinds of interesting post-processing actions you can take on a response, but for now, there isn't anything more to know about canned responses. Let's switch gears and see how to layer in security.

3.2 HTTPS imposters

To keep things simple we've focused on HTTP services. The reality's that real services require security, and that means using HTTPS. The "S" stands for SSL/TLS, which adds encryption, and, optionally, identity verification to each request.

The SSL layer of HTTPS is handled by the infrastructure. As far as the application's concerned, each request and response's standard HTTP. The details of how the SSL layer work are a bit complex, but the key concepts that make it work are the server's *certificate* and the server's *keys*. The HTTPS server presents the client an SSL certificate during the handshake process, which describes the server's identity, includ-

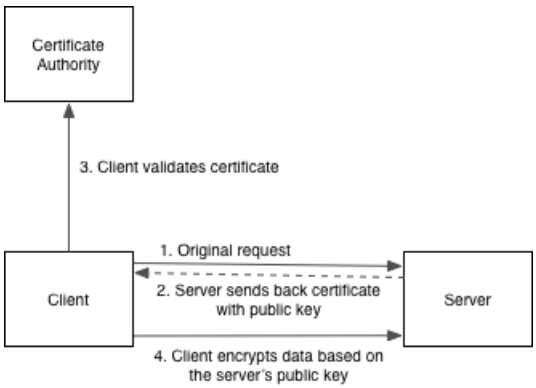


Figure 3.6 The basics of SSL

ing information like the owner, the domain it's attached to, and the validity dates. It's entirely possible that a malicious server may try to pass itself off as, say, Google, in the hopes you'll pass it confidential information that you'd only intend to pass to Google. This is why Certificate Authorities (CAs) exist. Trust has to start somewhere, and CAs are the foundation of trust in the SSL world. By sending a certificate, which contains a digital signature, to a CA trusted by your organization, you can confirm that the certificate's, in fact, from Google.

The certificate also includes the server's *public key*. The easiest approach to encryption's to use a single encryption key for both encryption and decryption. The type of encryption SSL relies on uses a neat trick that requires different keys for those two operations: the public key's used for encryption, and a separate *private key* is used for decryption. This allows the server to share its public key and the client to use that key for encryption, knowing that only the server can decrypt the resulting payload because only the server has the private key.

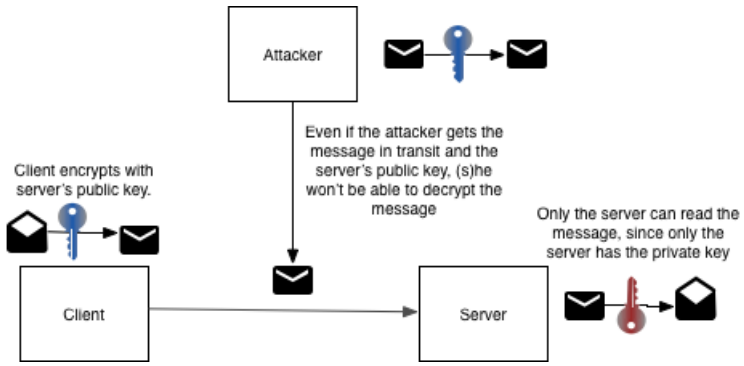


Figure 3.7 Using two keys prevents attackers from reading messages in transit even when the encryption key's shared

The good news is that creating an HTTPS imposter can look exactly like creating an HTTP imposter. The only required difference's that you set the protocol to "https".

```
{
  "protocol": "https",
  "port": 3000
}
```

This is great for quickly setting up an HTTPS server, but it uses a default certificate (and key pair) that ships with mountebank. That certificate's both insecure and untrusted. Although that may be OK for some types of testing, any respectable service call should validate that the certificate's trusted which, as shown in Figure 3.4 above, involves a call to a trusted Certificate Authority (CA).

That leaves two options. The first's that we could configure the service we're testing not to validate that the certificate's trusted. Don't do this. You don't want to risk leaving code like that in during production, and you don't want to test one behavior for your service (that doesn't do a certificate validation) and deploy a completely different behavior to production (that does validation). The whole point of testing, after all, is to gain confidence in what you're sending to production, which requires that you *test* what's going to production.

The second option's to use a certificate which is trusted, at least in the test environment. Organizations can run their own CA, making trust part of the environment rather than part of the application. You can set up your test instances of your virtual services with appropriately trusted certificates.

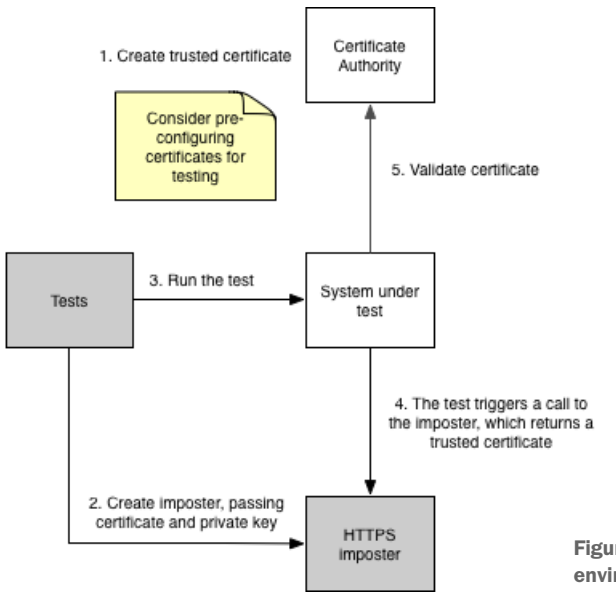


Figure 3.8 Setting up a test environment with HTTPS

With this approach, the test creates the imposter with both the certificate and the private key. You can pass them in what's known as PEM format; we'll look at how to create them shortly:

Listing 3.8 Creating an HTTPS imposter

```
{
  "protocol": "https",
  "port": 3000,
  "key": "-----BEGIN RSA PRIVATE KEY-----\n...",
  "cert": "-----BEGIN CERTIFICATE-----\n..."
}
```

The real text's much longer than shown

This setup's still insecure in that the test needs to know the private key to create the imposter, and the imposter knows how to decrypt communication from the system under test. The certificate's tied to the domain name in the URL; as long as you segment that domain name to your test environment, you're not risking leaking any production secrets. With appropriate environment separation, this approach allows you to test the system under test without changing its behavior to allow untrusted certificates.

3.2.1 Setting up a trusted HTTPS imposter

Historically, getting a certificate trusted by a public CA has been a painful and confusing process, and it cost enough money to discourage their use in exactly the kind of lightweight testing scenarios that mountebank supports. Using SSL's such a key cornerstone of internet security that major players are pushing to change that process, to the point where it's easy for even hobbyists without a corporate purse to create genuine certificates for the domains they register.

Let's Encrypt (letsencrypt.org/) is a free option that supports creating certificates for domains with minimal fuss, backed by a public CA. Every CA requires validation from the domain owner to ensure that no one's able to grab a certificate for a domain they don't own. Let's Encrypt allows you to completely automate the process by round-tripping a request based on a DNS lookup of the domain listed in the certificate.

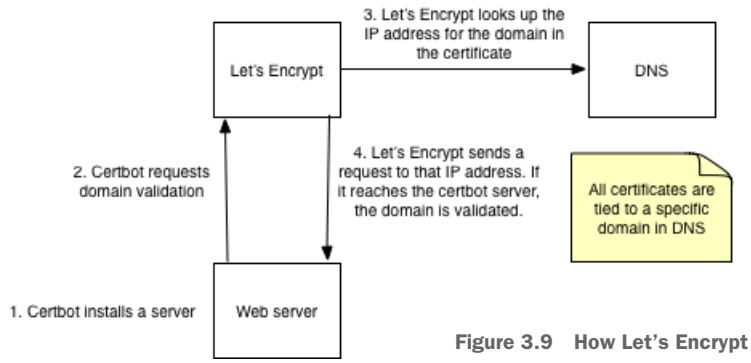


Figure 3.9 How Let's Encrypt validates the domain

Let's Encrypt uses a command line tool called "certbot" (certbot.eff.org/#ubuntutyakkety-nginx) to automate the creation of certificates. Certbot expects you to install a client on the machine receiving the SSL request. The client stands up a web server and sends a request to a Let's Encrypt server. Let's Encrypt in turn looks up the domain for which you're requesting a certificate in DNS and sends a request to that IP address. If that request reaches the certbot server that created the first request, Let's Encrypt has validated that you own the domain.

The actual certbot command depends on the web server you're using, and because it's constantly evolving, you should check their documentation for the details. In the general case, you might run:

```
certbot certonly --webroot -w /var/test/petstore -d test.petstore.com
```

That creates a certificate for the `test.petstore.com` domain, which is served out of a web server running in `/var/test/petstore`. Simplifications exist if you're using a common web server like Apache or Nginix; see certbot.eff.org/docs/using.html#getting-certificates-and-choosing-plugins for details.

By default, that directory certbot stores the SSL information in is `/etc/letsencrypt/live/$domain`, where `$domain` is the domain name of your service. If you look in that directory, you'll find a few files, but two relevant for our purposes: `privkey.pem` contains the private key, and `cert.pem` contains the certificate. The contents of those two files are what you put in the `key` and `cert` fields when creating the HTTPS imposter.

A PEM file has newlines. An example certificate might look like the following:

```
-----BEGIN CERTIFICATE-----
MIIDejCCAmICCQDlIe97PDjXJDANBgkqhkiG9w0BAQUFADB/MQswCQYDVQQGEwJV
UzEOMAwGA1UECBMFVGV4YXMxFTATBgNVBAoTDFRob3VnaHRXb3JrczEMMAoGA1UE
CxMDT1NTMRMwEQYDVQQDEwptYnRlc3Qub3JnMSYwJAYJKoZIhvcNAQkBFhdicmFu
ZG9uLmJ5YXJzQGdtYWlsLmNvbTAeFw0xNTA1MDMyMDE3NTRaFw0xNTA2MDIyMDE3
NTRaMH8xCzAJBgNVBAYTAlVTMQ4wDAYDVQQIEwVUZXhhczEVMBMGA1UEChMMVGhv
dWdodFdvcmtzMQwwCgYDVQQLEwNPU1MxEzARBgNVBAMTCm1idGVzdC5vcmcxJjAk
BgkqhkiG9w0BCQEWF2JyYW5kb24uYnlhcnNAZ21haWwuY29tMIIBIjANBgkqhkiG
9w0BAQEFAAOCAQ8AMIIBCgKCAQEA5V88ZyZ5hkPF7MzaDMvhGtGSBKIhQia2a0vW
6VfEtf/Dk80qKaalrwiBZlXheT/zwCoO7WBeqh5agOs0CSwzzEEie5/J6yVfgEJb
VROpnMbrLSgnUJXRfGNf0LCnTymGMhufz2utzcHRtgLm3nf5zQbBJ8XkOaPXokuE
UWwmTHrqeTN6munoxtt99olzusraxpgiGCil2ppFctsQHle49Vjs88KuyVjC5AOb
+P7Gqwru+R/1vBLyD8NVNl1WhLqaaeaopb9CcPgFZClchuMaAD4cecndrt5w4iuL
q91g71AjdXSG6V3R0DC2Yp/ud0Z8wXsMMC6X6VUxFrbeajo8CQIDAQABMA0GCSqG
SIb3DQEBBQUAA4IBAQCobQRpj0LjEcIViG8sXauwhRhgmmEyCDh57psWaZ2vdLmM
ED3D6y3HUzz08yZkRRr32VEtYhLldc7CHItscD+pZGJWlpgGKXEHdz/EqwR8yVhi
akBMhHxSX9s8N8ejLyIOJ9ToJQOPgelI019pvU4cmiDLihK5tezCrZfWNHXKw1hw
Sh/nGJ1UddEHCtC78dz6uIVIJQC0PkrLeGLKyAFrFJp4Bim8W8fbYSAffsWNATC+
dVKUlunVLd4RX/73nY5EM3ErcDDOCdUEQ2fUT59FhQF89DihFG4xW4OLq42/pgmW
KQBvwwfJxIFqg4fdnJUkHoLX3+glQWWrz80cauVH
-----END CERTIFICATE-----
```

You'll want to keep the newlines, escaped in typical JSON fashion with '\n', in the strings you send mountebank. In this example, shortening the field for clarity, the resulting imposter configuration might look like this:

```
{
  "protocol": "https",
  "port": 3000,
  "key": "-----BEGIN RSA PRIVATE KEY-----\nMIIEpAIBAAKC...",
  "cert": "-----BEGIN CERTIFICATE-----\nMIIDejCCAmICCQD..."
}
```

Note that, although it's awkward to show in book format, the string includes all the way up to the end of the file (e.g., "…\nWWrz80cauVH\n—END CERTIFICATE—" for the certificate).

And… that's it. Everything else about our imposter remains the same. Once we've set the certificate and private key, the SSL layer's able to convert encrypted messages into HTTP requests and responses, which means the `is` responses we've already created continue to work. It may seem like a fair chunk of work to setup the certificates, but it's the nature of SSL. Fortunately, tools like Let's Encrypt and shortcuts like using wildcard certificates simplify the process considerably.

Using wildcard certificates to simplify testing

A typical certificate's associated with a single domain name such as `mypetstore.com`. By adding a wildcard in front of the domain, the certificate becomes valid for all subdomains. We could, for example, create a `*.test.mypetstore.com` certificate, and that certificate's valid for `products.test.mypetstore.com` as well as `inventory.test.mypetstore.com`. It wouldn't be valid for production domains, which don't include `test` as part of their domain name.

A wildcard certificate's ideal for testing scenarios. You may find it easy enough to manually add a wildcard certificate to the CA, tied exclusively to a testing subdomain, and reuse the certificate and private key for all imposters.

3.2.2 Using mutual authentication

It turns out that certificates aren't only valid for HTTPS servers; they're also a common way to validate the identity of clients. You don't see this when browsing the internet because public websites can't afford to make assumptions about the browsers that access them, but in a microservices architecture, it's important to validate that only authenticated clients can make a request to a server.

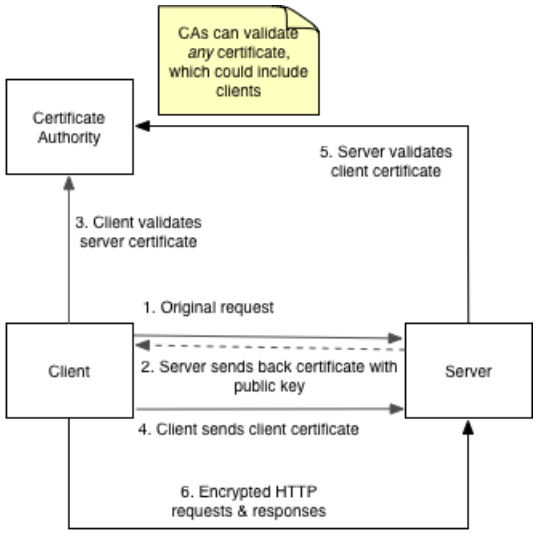


Figure 3.10 Setting up a test environment with HTTPS

If the service you're testing expects to validate its identity with your imposter using a client certificate, then your imposter needs to be able to be configured in a way that expects that certificate. This is as simple as adding a `mutualAuth` field to the configuration:

Listing 3.9 Adding mutual authentication to an imposter

```
{
  "protocol": "https",
  "port": 3000,
  "key": "-----BEGIN RSA PRIVATE KEY-----\nMIIEpAIBAAKC...",
  "cert": "-----BEGIN CERTIFICATE-----\nMIIDejCCAmICCQD...",
  "mutualAuth": true
}
```

When true, the server expects a client certificate

Now the server challenges the client with a certificate request. Using certificates, both for HTTPS and for mutual authentication, allows you to virtualize servers in secure environments. The fact that we've to escape the PEM files in JSON gets quite clunky. Let's look at how to make maintaining that data a bit easier using configuration files.

3.3 Saving the responses in a configuration file

By now you're probably realizing that, as the complexity of the responses and security configuration increases, the JSON that you send mountebank can be quite complex. This is true even for a single field, like the multi-line PEM files that need to be encoded as a JSON string. Fortunately, mountebank has robust support for persisting the configuration in a friendly format.

Now that we've added an Inventory service and seen how to convert it to HTTPS, let's see how we'd format the imposter configuration in files to make it easier to manage. The first bit of ugliness we'll want to solve's storing the certificate and private key in separate

PEM files to avoid a long JSON string. If we store those as `cert.pem` and `key.pem` in the `ssl` directory, then we can create a file for the Inventory imposter as `inventory.ejs`:

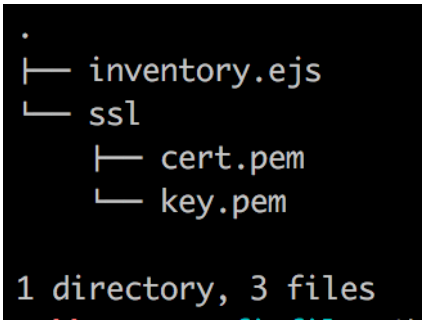


Figure 3.11 The tree structure for the secure Inventory imposter configuration

Save the following in `inventory.ejs`:

Listing 3.10 Storing the Inventory service in a configuration file

```
{
  "port": 3000,
  "protocol": "https",
  "cert": "<%- stringify(filename, 'ssl/cert.pem') %>",
  "key": "<%- stringify(filename, 'ssl/key.pem') %>",
  "stubs": [
    {
      "responses": [
        { "is": { "body": "54" } },
        { "is": { "body": "21" } },
        { "is": { "body": "0" } }
      ]
    }
  ]
}
```

These automatically convert the multiline file content into a JSON string

If you start mountebank with the appropriate command line flag, the Inventory service will be available at startup.

```
mb --configfile inventory.ejs
```

Mountebank uses a templating language called EJS (www.embeddedjs.com/) to interpret the config file, which uses a fairly standard set of templating primitives. The content between `<%-` and `%>` is dynamically evaluated and interpolated. Mountebank adds the `stringify` function, which does an equivalent of a JavaScript `JSON.stringify` call on the contents of the given file. In this case that escapes the newlines. The benefit to us is that the configuration's much easier to read. (The `filename` variable's passed in by mountebank. It's a bit of a hack needed to make relative paths work.)

With those two templating primitives—the angle brackets to interpolate in dynamic data and the `stringify` function to turn that data into presentable JSON —

we can build robust templates. Storing the SSL information separately's useful, but the Inventory imposter was intentionally over-simplified to focus on the behavior of the `responses` array. Let's add in the Product Catalog and Marketing Content services we saw in chapter two.

3.3.1 Saving multiple imposters in the config file

As we saw, templating allows us to break up our configuration into multiple files. We'll take advantage of that to revisit the Product and Content imposter configurations we saw in chapter two, putting each imposter in one or more files. The first thing we need to do's define the root configuration, which now needs to take a list of imposters. The tree structure looks like this:

Figure 3.12 The tree structure for multiple service

Save this as `imposters.ejs`:

Listing 3.11 The root configuration file, referencing other imposters

```
{
  "imposters": [
    <% include inventory.ejs %>,
    <% include product.ejs %>,
    <% include content.ejs %>
  ]
}
```

`include` interpolates content from other files as-is.

The `include` function comes from EJS. Like the `stringify` function, it loads in content from another file. Unlike `stringify`, the `include` function doesn't change the data; it's interpolated in as-is. We can use the `include` EJS function and the `stringify` mountebank function to lay out our content any way we like. For complex configuration, we can store the response bodies—JSON, XML, or any other complex representation—in different files with newines and load them in as needed. To keep it simple, we'll save each of our imposters in a different file, loading in the same wildcard certificate and private key. Save the Product Catalog imposter configuration that we saw in Listing 2.1 in `product.ejs`:

```
{
  "protocol": "https",
  "port": 3001,
  "cert": "<%- stringify(filename, 'ssl/cert.pem'); %>",
  "key": "<%- stringify(filename, 'ssl/key.pem'); %>",
  "stubs": [{
    "responses": [{
      "is": {
        "statusCode": 200,
        "headers": { "Content-Type": "application/json" },
        "body": {
          "products": [
            {
              "id": "2599b7f4",
              "name": "The Midas Dogbowl",
              "description": "Pure gold"
            },
            {
              "id": "e1977c9e",
              "name": "Fishtank Amore",
              "description": "Show your fish some love"
            }
          ]
        }
      }
    }],
    "predicates": [{
      "equals": { "path": "/products" }
    }]
  }]
}
```

Note that we converted it to HTTPS

We used a different port to avoid a port conflict

The stub configuration's the same as in chapter two

Finally, save the Marketing Content imposter configuration we saw in Listing 2.6 in a file called `content.ejs`:

```
{
  "protocol": "https",
  "port": 3002,
  "cert": "<%- stringify(filename, 'ssl/cert.pem'); %>",
  "key": "<%- stringify(filename, 'ssl/key.pem'); %>",
  "stubs": [{
    "responses": [{
      "is": {
        "statusCode": 200,
        "headers": { "Content-Type": "application/json" },
        "body": {
          "content": [
            {
              "id": "2599b7f4",
              "copy": "Treat your dog like the king he is",
              "image": "/content/c5b221e2"
            },
            {
              "id": "e1977c9e",
              "copy": "Love your fish; they'll love you back",
```

Also on a different port

```
              "image": "/content/a0fad9fb"
            }
          ]
        }
      }
    }],
    "predicates": [{
      "equals": {
        "path": "/content",
        "query": { "ids": "2599b7f4,e1977c9e" }
      }
    }]
  }]
}
```

Now you can start mountebank by pointing to the root configuration file:

```
mb --configfile imposters.ejs
```

Notice what happens in the logs:

```
info: [mb:2525] mountebank v1.11.0 now taking orders -
<linearrow /> point your browser to http://localhost:2525 for help
info: [mb:2525] PUT /imposters
info: [https:3000] Open for business...
info: [https:3001] Open for business...
info: [http:3002] Open for business...
```

All three imposters are up and running. Of interest's the log entry pointing out that an HTTP `PUT` command was sent the mountebank URL of localhost:2525/imposters. After running the contents of the configuration file through EJS, the `mb` command sends the results as the request body of the `PUT` command, which creates (or replaces) all the imposters in one shot. Nearly every feature in mountebank's built API first, and anything you can do on the command line you could implement yourself using the API. If you've more advanced persistence requirements, you could construct the JSON yourself and send it to mountebank using `curl`:

```
curl -X PUT http://localhost:2525/imposters --data '{
  "imposters": [
    {
      "protocol": "https",
      "port": 3000
    },
    {
      "protocol": "https",
      "port": 3001
    }
  ]
}
```

For clarity, I've left out all the important bits of the imposter configuration. You may find the `PUT` command a convenience in automated test suites where a setup step overwrites the entire set of imposters with one API call, rather than relying on each individual test to send the `DELETE` calls to clean up their imposters.

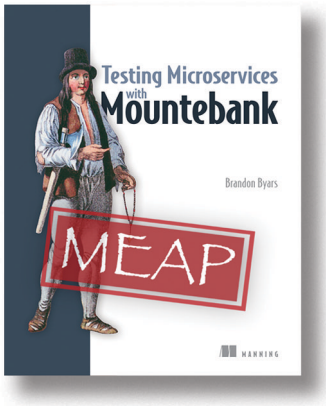
If you load the imposters through a configuration file, then the imposter setup's part of starting mountebank, which you're expected to do before running your tests. That allows you to remove some of the setup steps in the test itself—specifically, those related to configuring and deleting the imposters.

3.4 Summary

In this chapter you learned:

- The `is` response type allows you to create a canned response. The fields you specify in the response object merge in with the default response. You can change the default response if you need to.
- One stub can return multiple responses. The list of responses acts like a circular buffer, and once the last response's returned, mountebank cycles back to the first response.
- HTTPS imposters are possible, but you must create the key pair and certificate. Let's Encrypt's a free service that lets you automate the process.
- Setting the `mutualAuth` flag on an imposter means that it will accept client certificates used for authentication.
- Mountebank uses EJS templating for persisting the configuration of your imposters. You load them at startup by passing the root template as the parameter to the `--configfile` command line option.

This chapter covered many of the basics of mountebank. In the next chapter, we'll complete the foundation of mountebank by deep diving into predicates.



Microservices are independent, single-responsibility units of code that form a system with other microservices. It's difficult to test an individual microservice since each one depends on the other services. Mountebank solves this conundrum through service virtualization—imitating other components in the system so that you can test a microservice in isolation. Mountebank is the most capable service virtualization tool around, providing a programmable stand-in for a real dependency. As the only open-source virtualization tool with support for multiple protocols and scaling for load testing mountebank isn't snake oil; it's the cure-all for all your service virtualization needs.

Testing Microservices with Mountebank is your guide to the ins and outs of testing microservices with service virtualization. Starting from your first test with mountebank you'll dive into testing with canned responses, using predicates, and recording and replaying behavior to your tests. Following real world use-cases you'll master the art of programming mountebank with your own dynamic responses and explore working with binary protocols. This book also explains using mountebank for load testing, in a continuous delivery pipeline, and more.

What's inside:

- Approaches to service virtualization
- Testing using canned responses
- Programming mountebank
- Understanding behaviors
- Creating record / replay behavior
- Adding contract tests

Readers need programming skills and should be generally familiar with SOA or microservice systems.

index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

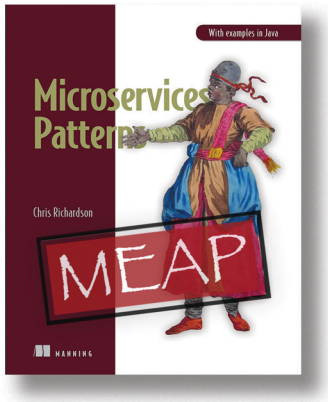
M

T

U

V

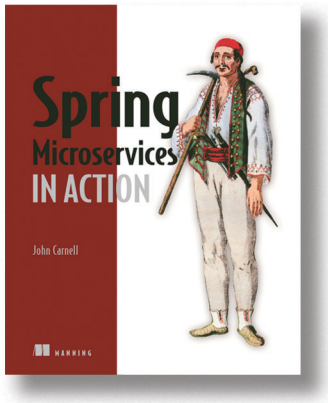
Z



Microservice Patterns

by Chris Richardson

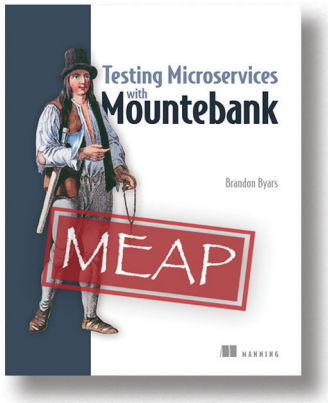
ISBN: 9781617294549
375 pages
$49.99
November 2017



Spring Microservices in Action

by John Carnell

ISBN: 9781617293986
384 pages
$49.99
June 2017



Testing Microservices with Mountebank

by Brandon Byars

ISBN: 9781617294778
275 pages
$44.99
March 2017