Testing Microservices with Mountebank

Brandon Byars

MANNING

Testing Microservices with Mountebank

Brandon Byars

Copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book 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.

PO Box 761
Shelter Island, NY 11964

Development editor: Elesha Hyde
Technical development editor: John Guthrie
Review editor: Ivan Martinović
Project manager: Vincent Nordhaus
Copy editor: Carl Quesnel
Proofreader: Keri Hales
Technical proofreader: Alessandro Campeis
Typesetter and cover designer: Marija Tudor

ISBN 9781617294778

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – SP – 23 22 21 20 19 18

About this book

I wrote *Testing Microservices with Mountebank* to show how service virtualization helps you test microservices, and how mountebank is a powerful service virtualization tool. That necessarily requires building a deep familiarity with mountebank—the middle section of this book is dedicated to the subject—but many of the lessons apply with any service virtualization tool.

WHO SHOULD READ THIS BOOK

Mountebank is a developer-friendly tool, which makes developers a primary audience of *Testing Microservices with Mountebank*. Some familiarity with test automation is expected, but I avoid using any advanced language features throughout this book to keep the focus on the tool and the approach. Automation-friendly QA testers also will find value in this book, as will those who specialize in performance testing. Finally, service virtualization is increasingly an architectural concern, and within these pages I hope to arm solution architects with the arguments they need to make the right decisions.

HOW THIS BOOK IS ORGANIZED

You'll find three parts and 10 chapters in this book.

- Part 1 introduces the overall testing philosophy of distributed systems.
    - Chapter 1 provides a brief refresher of microservices, as well as a critique of traditional end-to-end testing. It helps explain how service virtualization fits in a microservices world and provides a mental model for mountebank.
    - Chapter 2 sets up an example architecture that we will revisit a few times throughout the book and shows how you can use mountebank to automate deterministic tests despite a distributed architecture.
- Part 2 deep dives into mountebank, giving you a comprehensive overview of its capabilities.

- Chapter 3 provides foundational material for understanding basic mountebank responses in the context of HTTP and HTTPS. It also describes basic ways of managing test data through configuration files.
- Chapter 4 explores predicates—mountebank's way of responding differently to different types of requests. It also introduces mountebank's capability around matching XML and JSON.
- Chapter 5 looks at mountebank's record and replay capability. Mountebank uses proxies to real systems to let you capture realistic test data.
- Chapter 6 shows you how to program mountebank itself, by using a feature called injection to write your own predicates and responses in JavaScript. We look at how injection helps solve some thorny problems around CORS and OAuth handshakes, including virtualizing GitHub's public API.
- Chapter 7 rounds out the core capabilities of the mountebank engine by looking at behaviors—postprocessing steps applied to responses. Behaviors let you add latency, look up data from external sources, and perform a host of other transformation steps.
- Chapter 8 shows how all of the concepts in chapters 3– 7 extend beyond HTTPS. The engine of mountebank is protocol-agnostic, and we show TCP-based examples, including an extended .NET Remoting test scenario.

- Part 3 takes a step back to put service virtualization in a broader context.
  - Chapter 9 explores an example test pipeline for microservices, from unit tests to manual exploratory tests, and shows where service virtualization does and doesn't fit.
  - Chapter 10 shows how service virtualization helps with performance testing. It includes a fully worked out example virtualizing a publicly available API.

ABOUT THE CODE

This book uses a number of code examples to help illustrate the concepts. Some of them are whimsical (see chapter 4), some are based on virtualizing real public APIs (see chapters 6 and 10), and some are just downright gnarly (see chapter 8). I tried my best to keep the examples interesting across the wide range of problems that service virtualization can help with. This was no easy task. Some problems are easily understood. Some, like virtualizing a .NET Remoting service returning binary data, are not. My hope is that I've maintained enough of a sense of humor to keep you interested with the easy problems, and with the complex behavior, I've given you enough of a

sense of what's possible that you'll be capable of innovating on your own.

Source code for the book is publicly available at https://github.com/bbyars/mountebank-in-action.

BOOK FORUM

Purchase of *Testing Microservices with Mountebank* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and other users. To access the forum go to www.manning.com/books/testing-microservices-with-mountebank. You can also learn more about Manning's forums and the rules of conduct at https://forums.manning.com/forums/about.

Manning's commitment to its readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contributions to the forum remain voluntary (and unpaid). We suggest you ask the author challenging questions, lest his interest stray.

ONLINE RESOURCES

Need additional help?

The mountebank website at https://www.mbtest.org provides the most up-to-date documentation for the tool.

The Google group site is https://groups.google.com/forum/#!forum/mounte bank-discuss. Feel free to ask any questions about the tool there.

Part 1. First Steps

Welcome, friend.

Those words adorn the welcome mat near my front door, and they are the first words you will see when you visit the mountebank website (https://www.mbtest.org). I'd very much like them to be the first words you read in this book, as I welcome you to the wonderful world of service virtualization in general and mountebank in particular.

This first part aims to provide context behind that introduction, setting the stage for introducing mountebank as part of your testing and continuous delivery stack. Because one of the main drivers behind service virtualization is the increasingly distributed nature of computing, chapter 1 starts with a brief review of microservices, with a focus on how they change the way we test software. It puts service virtualization in context and provides a gentle introduction to the main components of mountebank.

Chapter 2 demonstrates mountebank in action. You'll get some dirt under your nails as you write your first test using service virtualization, providing a simple launching point to explore the full capabilities of mountebank in part 2.

Chapter 1. Testing microservices

This chapter covers

- A brief background on microservices
- The challenges of testing microservices
- How service virtualization makes testing easier
- An introduction to mountebank

Sometimes, it pays to be fake.

I started developing software in the days when the web was starting to compete with desktop applications in corporate organizations. Browser-based applications brought tremendous deployment advantages, but we tested them in almost the same way. We wrote a monolithic application, connected it to a database, and tested exactly like our users would: by using the application. We tested a *real* application.

Test-driven development taught us that good object-oriented design would allow us to test at much more granular levels. We could test classes and methods in isolation and get feedback in rapid iterations. Dependency injection—passing in the dependencies of a class rather than instantiating them on demand—made our code both more flexible and more testable. As long as we passed in test dependencies that had the same interface as the real ones, we could completely isolate the bits of code we wanted to test. We gained more confidence in the code we wrote by being able to inject fake dependencies into it.

Before long, clever developers produced open-source libraries that made creating these fake dependencies easier, freeing us to argue about more important things, like what to call them. We formed cliques based on our testing styles: the mockists reveled in the purity of using mocks; the classicists proudly stood by their stubborn reliance on stubs.

But neither side argued about the fundamental value of testing against fake dependencies.

You probably have better things to spend your time on than reading about the differences between classicists and mockists, but if you can't help yourself, you can read more at http://martinfowler.com/articles/mocksArentStubs.html.

It turns out when it comes to design, what's true in the small is also true in the large. After we made a few halting attempts at distributed programming, the ever-versatile web gave us a convenient application protocol—HTTP—for clients and servers to talk to each other. From proprietary RPC to SOAP to REST and back again to proprietary RPC, our architectures outgrew the single codebase, and we once again needed to find ways to test entire services without getting tangled in their web of runtime dependencies. The fact that most applications were built to retrieve the URLs for dependent services from some sort of configuration that varied per environment meant dependency injection was built in. All we needed to do was configure our application with URLs of *fake* services and find easier ways to create those fake services.

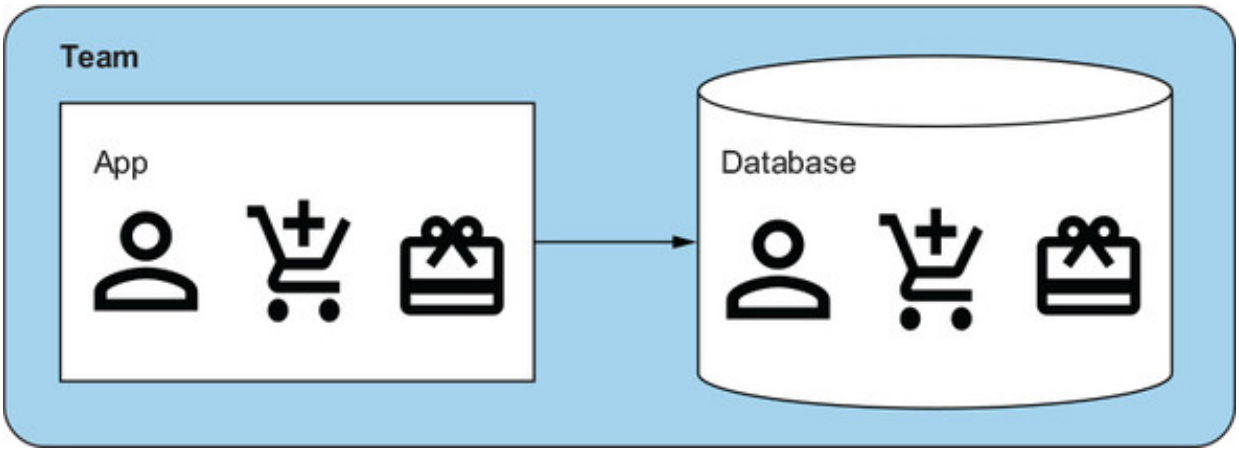
Mountebank creates fake services, and it's tailor-made for testing microservices.

1.1. A MICROSERVICES REFRESHER

Most applications are written as *monoliths*, a coarse-grained chunk of code that you release together with a shared database. Think of an e-commerce site like Amazon.com. A common use case is to allow a customer to see a history of their orders, including the products they have purchased. This is conceptually easy to do as long as you keep everything in the same database.

In fact, Amazon did this in their early years, and for good reason. The company had a monolithic codebase they called "Obidos" that looked quite similar to figure 1.1. Configuring the database that way makes it easy to join different domain entities, such as customers and orders to show a customer's order history or orders and products to show product details on an order. Having everything in one database also means you can rely on transactions to maintain consistency, which makes it easy to update a product's inventory when you ship an order, for example.

Figure 1.1. A monolithic application handles view, business, and persistence logic for multiple domains.

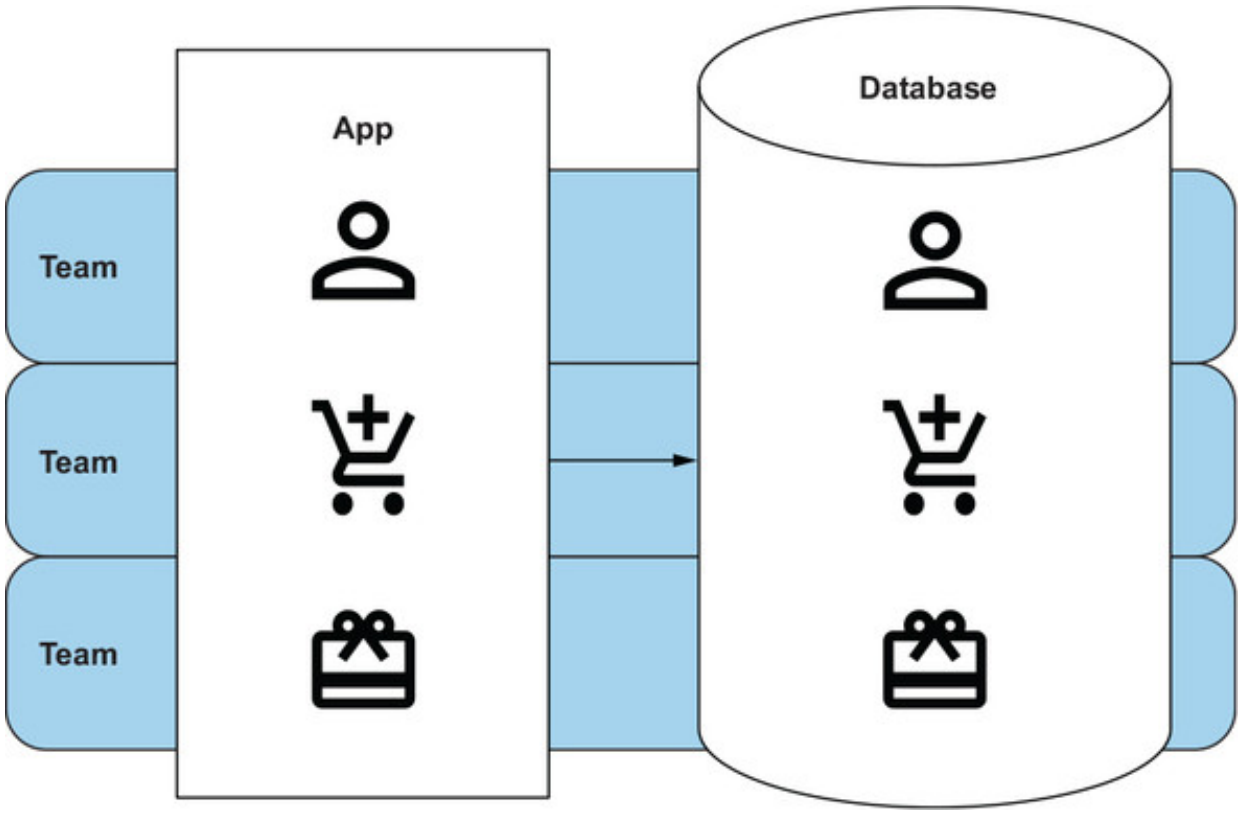


This setup also makes testing—the focus of this book—easier. Most of the tests can be in process, and, assuming you are using dependency injection, you can test pieces in isolation using mocking libraries. Black-box testing the application only requires you to coordinate the application deployment with the database schema version. Test data management comes down to loading the database with a set of sample test data. You can easily solve all of these problems.

1.1.1. The path toward microservices

It's useful to follow the history of Amazon.com to understand what compels organizations to move away from monolithic applications. As the site became more popular, it also became bigger, and Amazon had to hire more engineers to develop it. The problems started when the development organization was large enough that multiple teams had to develop different parts of Obidos (figure 1.2).

Figure 1.2. Scaling a monolith means multiple teams have to work in the same codebase.

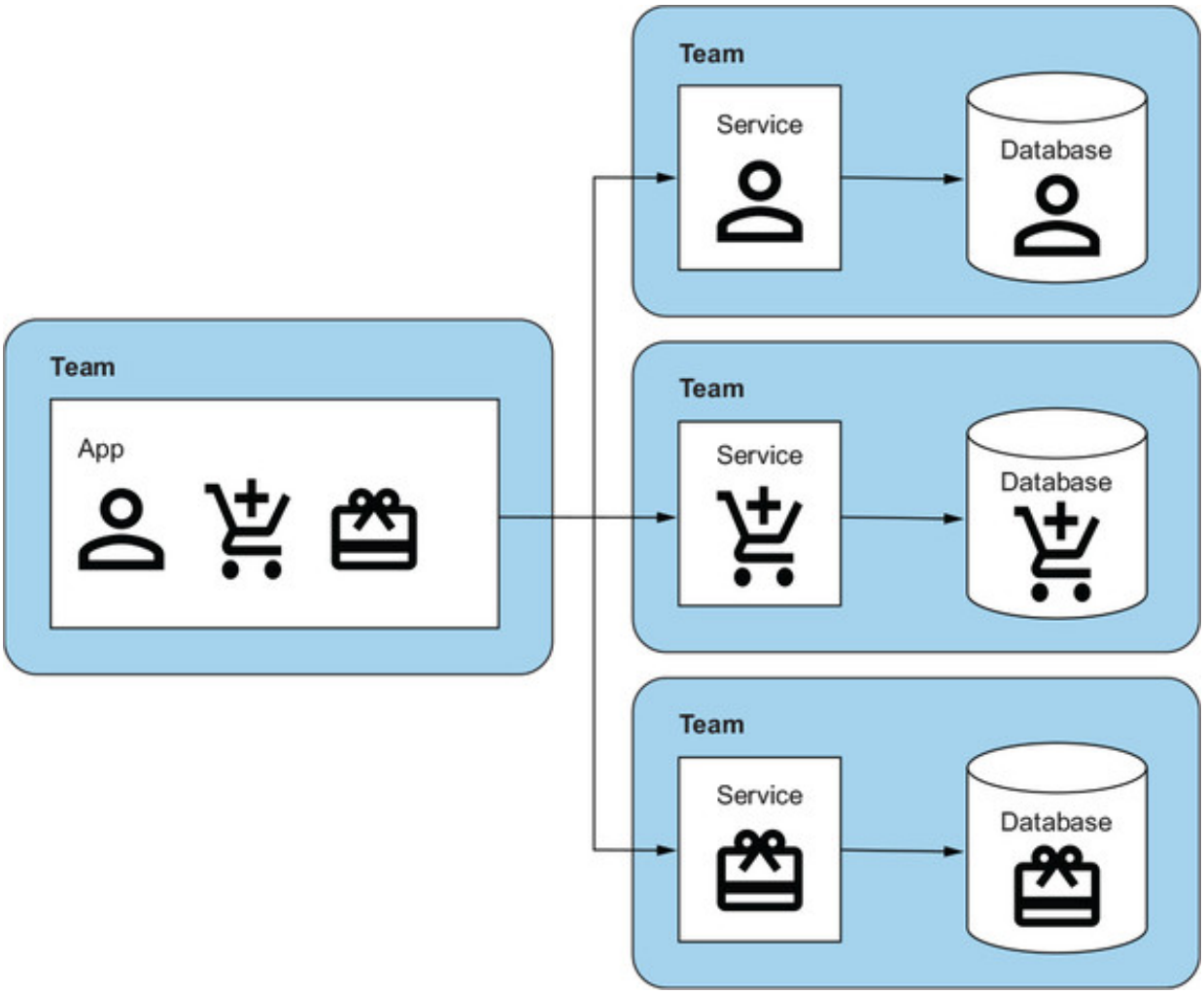


The breaking point came in 2001, as the company struggled to evolve pieces of the application because of the coupling between teams. By CEO mandate, the engineering organization split Obidos into a series of services and organized its teams around them. [2] After the transformation, each team was able to change the code relevant to the domain of their service with much higher confidence that they weren't breaking other teams' code—no other team shared their codebase. Amazon now has tremendous ability to develop different parts of the website experience independently, but the transformation has required a change of paradigm. Whereas Obidos used to be solely responsible for rendering the site, nowadays a single web page at Amazon.com can generate over a hundred service calls (figure 1.3).

2

See https://queue.acm.org/detail.cfm?id=1142065 for details.

Figure 1.3. Services use different databases for different domains.



The upshot is that each service can focus on doing one thing well and is much easier to understand in isolation. The downside is that such an architecture pushes the complexity that used to exist inside the application into the operational and runtime environment. Showing both customer details and order details on a single screen changes from being a simple database join to orchestrating multiple service calls and combining the data in the application code. Although each service is simple in isolation, the system as a whole is harder to understand.

Netflix was one of the first companies of its size to migrate its core business to Amazon's cloud services, which significantly influenced the way the company thought about services. Once again, the need to scale its development efforts is what drove this change. Adrian Cockcroft, formerly the Netflix lead cloud architect, noted two opposing tensions.[3] First, demands on IT had increased by a factor of 1,000 in recent years as technology moved from managing payroll and a few enterprise services to becoming the core differentiator for digitally native companies. Second, as the number of engineers increased, the communication and coordination overhead of activities like troubleshooting a broken build became a significant slowdown to delivering software.

See https://www.infoq.com/presentations/migration-cloud-native.

Netflix experienced this slowdown once the organization grew to about 100 engineers, with multiple teams competing in the same codebase. Like Amazon, the company solved the problem by breaking the monolithic application into services, and it made a conscious effort to make each service do only one thing. This architectural approach—what we now call microservices—supported development teams working independently of each other, allowing the company to scale its development organization.

Although scaling development efforts is the primary force leading us toward microservices, it has a welcome side effect: it's much easier to release a small service than it is to release a large application. If you can release services independently, you can release features to the market much more quickly. By removing the need for release coordination between teams, microservices provide an architectural solution for what would normally have to be solved manually. Customers are the beneficiaries. Both Amazon and Netflix are known for the ability to rapidly innovate in the marketplace but to do so required rethinking how they organized to deliver software and how they tested software.

1.1.2. Microservices and organizational structure

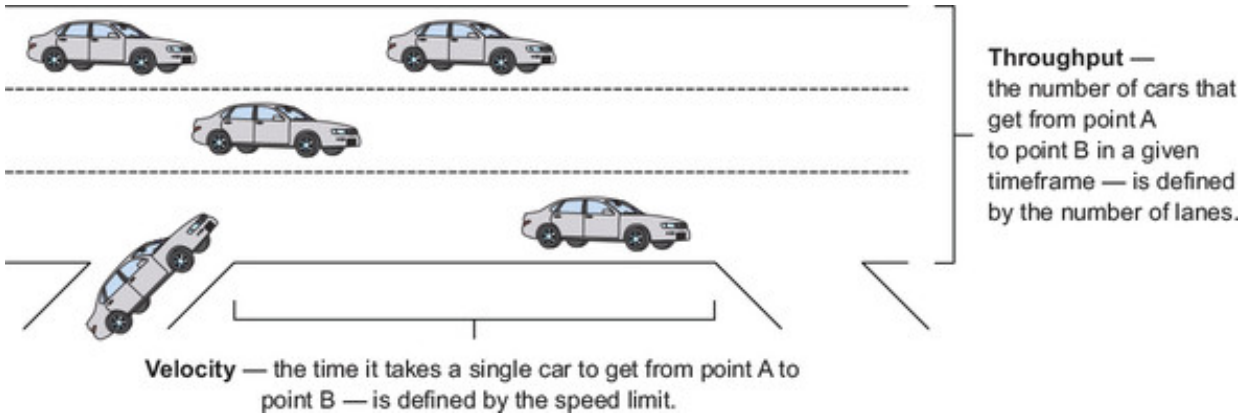
Before we get into how mountebank makes testing microservices easier, you need to understand how microservices require a different testing mentality. It all starts with team organization and ends with a full-frontal assault on traditional QA approaches that gate releases through coordinated end-to-end testing.

Microservices require you to rethink traditional organizational structures. Silos exist in any large organization, but some silos are anathema to the goal of microservices, which is to allow independent releases of small services with minimal coordination. Traditional organizations use certain silos as "gates" that validate a deployment before it's released. Each gate acts as a point of coordination. Coordination is one way to gain confidence in a release, but it's slow and couples teams together. Microservices work best when you silo the organization by business capability and allow the development team to own the release of its code autonomously.

A useful metaphor to explain the concept is to imagine your IT organization as a highway. When you need to increase throughput—the number of features released over a given timeframe—you do so by adding lanes to the highway. Having more lanes means you can support more cars at the same time (figure 1.4). This is analogous to hiring more developers to do more work in parallel. You also may want to be able to release a single feature more quickly. This is equivalent to raising the speed limit on the

highway, enabling a single car to get from point A to point B in less time. So far, so good.

Figure 1.4. During normal traffic, the number of lanes and speed limit define throughput and velocity.

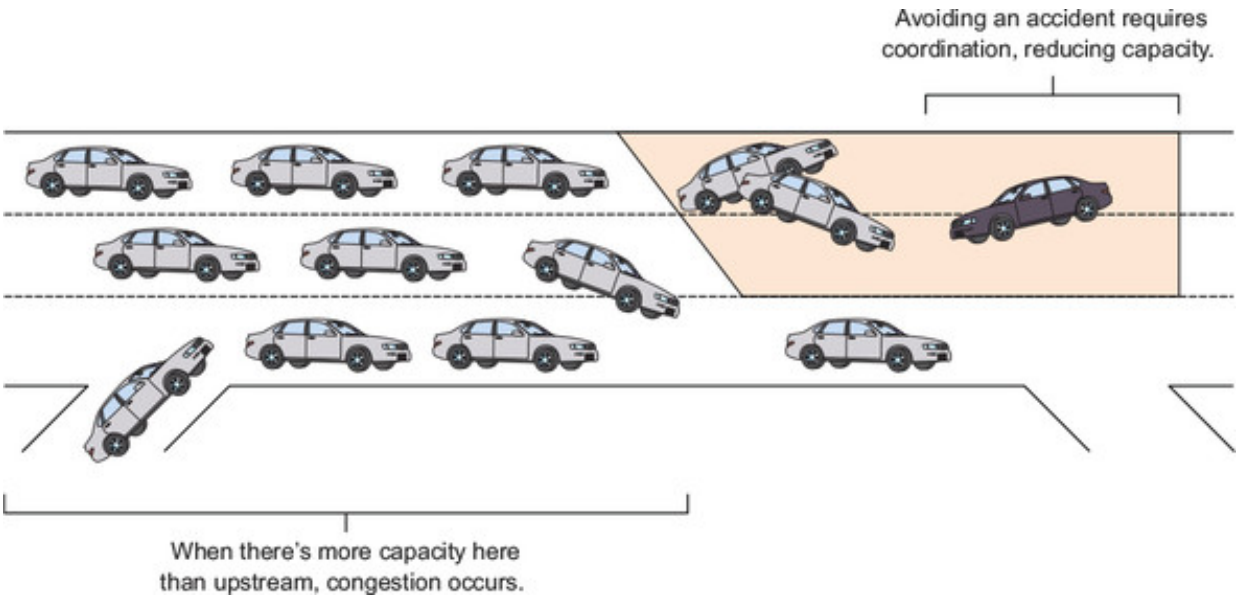


One thing will kill both throughput *and* velocity, no matter how many lanes you have or what the speed limit is: congestion. Congestion also has an indirect cost that you have almost certainly experienced if you live in a big city. Navigating through stop-and-go traffic is a soul-crushing experience. It's demotivating. It's hard to get excited to get in your car and drive. Many large IT organizations that, with the best of intentions, create unintended congestion, suffer from a real motivational cost.

They create congestion in two ways. First, they overuse the highway, having too many cars given the space available. Second, they add coordination that creates congestion. This second way is harder to eradicate.

One way to require coordination on the highway is to add a toll gate (especially those old-timey ones, before an automated camera replaced the need to pay physical money to a real person). Another way is to have fewer upstream lanes than you have downstream, where "upstream" refers to the section of the highway closer to the exit in highways, and closer to a production release in IT. Reducing the number of upstream lanes throttles traffic by requiring merging multiple lanes into one (figure 1.5). Sometimes this reduction happens by design or because of road construction. Other times it happens because of an accident, which results from an unfortunate degree of coupling between two cars.

Figure 1.5. Having fewer upstream lanes increases congestion.



Like all models, the highway metaphor is imperfect, but it highlights some useful points. As you saw earlier in the Amazon and Netflix examples, microservices often originate from an organization's desire to increase feature throughput. A helpful side effect is that a smaller codebase has a higher speed limit, increasing velocity. But both of these advantages are negated if you don't change the organization to remove congestion.

In organizational terms, fixing overutilization is easy in principle, though it's often politically challenging. You can either hire more people (add more lanes to the highway) or reduce the amount of work in progress (throttle entry to the highway).

The other reasons for congestion are harder to fix. It's common to see organizations with less upstream capacity than downstream capacity. One toll gate example would be increasing coordination by requiring releases to go through a central release management team that throttles development throughput. Accidents are even more common. Every time someone discovers a bug or the build breaks in a codebase that multiple teams share, you have an accident requiring coordination, and both throughput and velocity suffer. Adrian Cockcroft cited this exact reason for driving Netflix toward microservices.

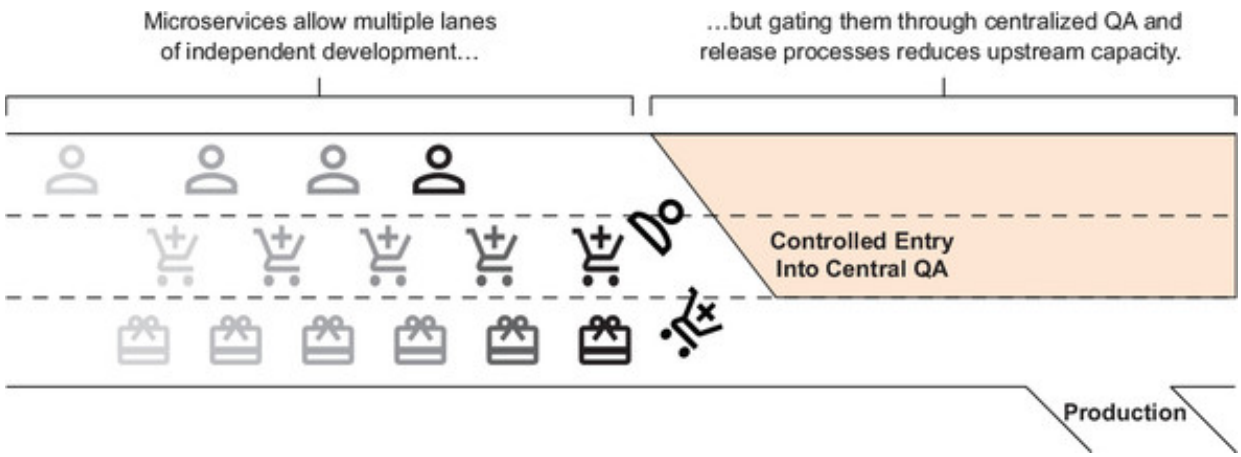
Microservices provide a technical solution for reducing congestion caused by accidents. By not sharing the same codebase, broken builds don't affect multiple teams, effectively creating different lanes for different services. But toll gates are still a problem, and to fully unlock the throughput and velocity advantages we hope to gain through microservices, we have to address organizational complexity. That comes in many forms (for example, from operations to database schema management), but there's one form of upstream congestion that's particularly relevant to this book: our QA organization. Microservices fundamentally challenge the way you test.

1.2. THE PROBLEM WITH END-TO-END TESTING

Traditionally, a central QA team could partner with a central release management team to coordinate a schedule of changes that needed to be deployed into production. They could arrange it such that only one change went through at a time, and that change could be tested against the production versions of its runtime dependencies before being released. Such an approach is perfectly reasonable up to a point. Beyond that—and this is often where organizations turn to microservices—it's inappropriate.

You still need confidence that the entire system will work when you release part of it. Gaining that confidence through traditional end-to-end testing and coordinated releases couples all of the services together, moving the coupling bottleneck from the development organization to the QA organization (figure 1.6).

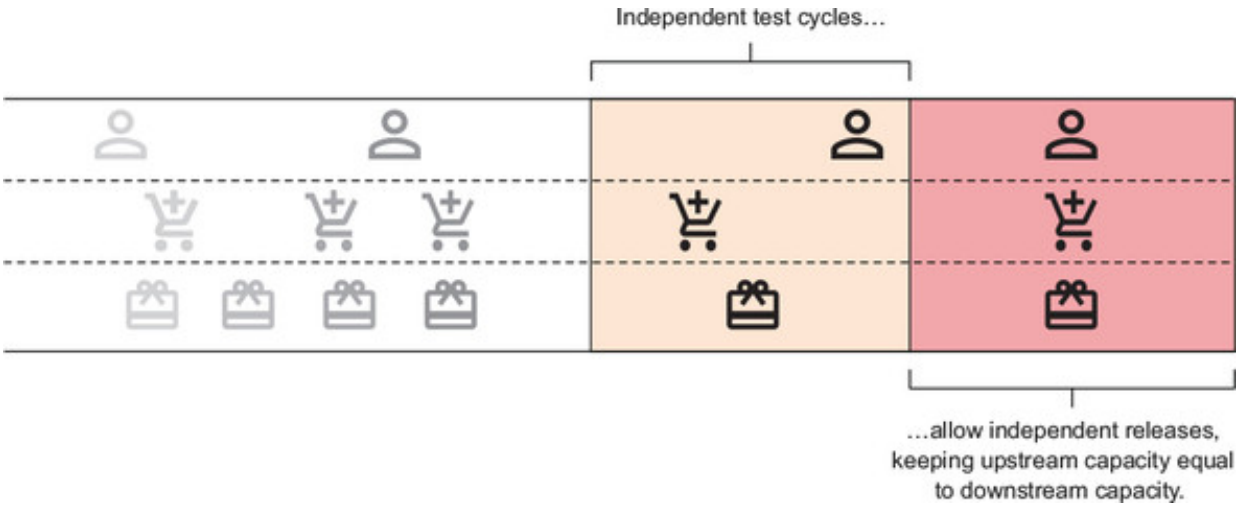
Figure 1.6. Centralized QA processes recouple releases together, causing a bottleneck.



Coordinated integration testing between services recouples codebases that have been decoupled through service decomposition, destroying the scaling and rapid delivery advantages of microservices. As soon as you couple the releases of services together, you have reintroduced the communication and coordination overhead you were trying to avoid. It doesn't matter how many services you have; when you have to release them at the same time, you have a monolith.

The only way to truly scale the technology organization is to decouple releases, so that it can deploy a service independently of the service's dependencies (figure 1.7). This requires a fundamental rethinking of the test strategy for microservices. End-to-end testing doesn't completely disappear, but relying on it as a gate to releasing software becomes another sacrifice on the path toward microservices. The question remains: how do you gain the confidence you need in your changes before releasing them?

Figure 1.7. Independent testing works to avoid release congestion.



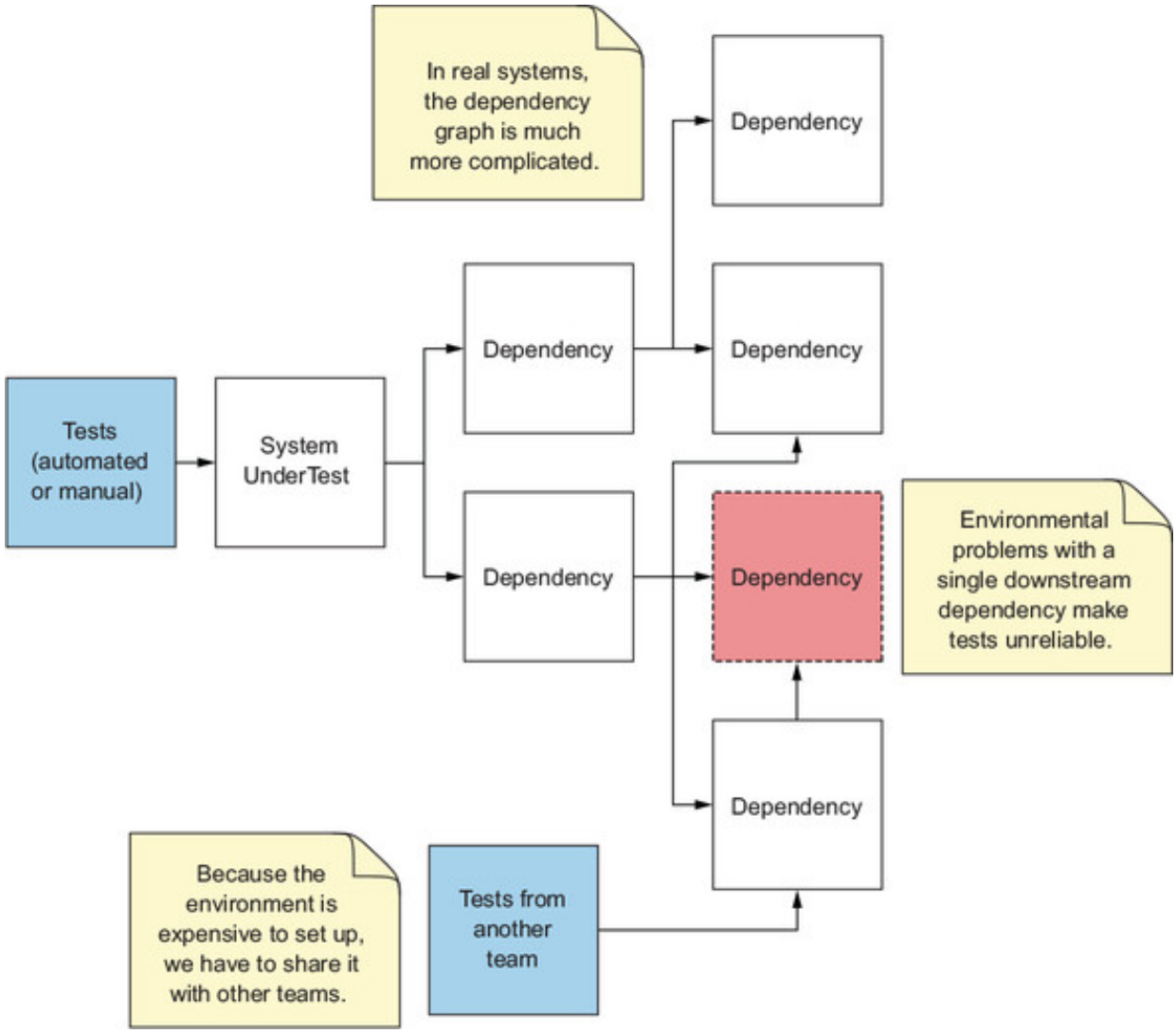
1.3. UNDERSTANDING SERVICE VIRTUALIZATION

The problem with dependencies is that you can't depend on them.

Michael Nygard, "Architecture Without an End State"

Additional problems exist besides coordination, as shown in figure 1.8. Running in a shared environment means tests may pass or fail for reasons that have nothing to do with either the service you are testing or the tests themselves. They could fail because of resource contention with other teams who are touching the same data, overwhelming the server resources of a shared service, or environmental instability. They could fail, or be nearly impossible to write to begin with, because of an inability to get consistent test data set up in all of the services.

Figure 1.8. End-to-end testing introduces several problems of coordination.



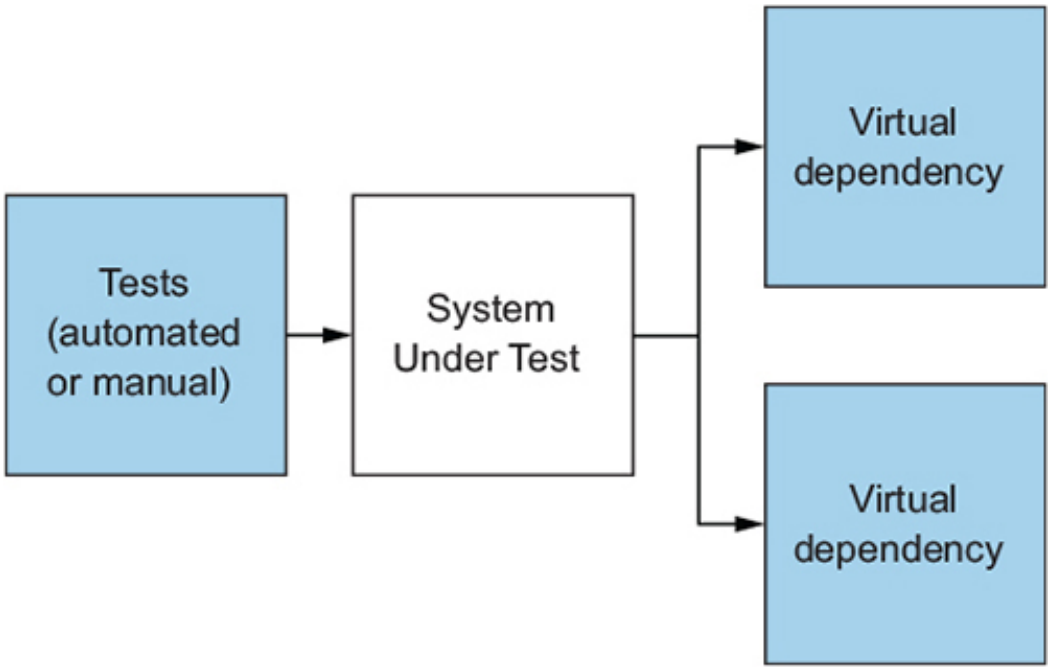
It turns out that other industries have already solved this problem. A car, for example, is made up of a multitude of components, each of which can be released to the market independently of the car as a whole. By and large, nobody buys an alternator or a flywheel for anything other than to fix a car. All the same, it's common for companies to manufacture and sell those parts separately *even though they have never tested them in your specific car*.

A car battery comes standard with negative and positive terminals. You can test the battery—outside the car—by using a voltmeter attached to those two terminals and verifying that the voltage is between 12.4 and 12.7 volts. When you start the car, the alternator is responsible for charging the battery, but you can verify the behavior of the battery independently of the alternator by providing a current as input to the battery and measuring the voltage. Such a test tells you that, *if* the alternator is behaving correctly, *then* the battery also behaves correctly. You can gain most of the confidence you need to verify the battery is working by using a *fake* alternator.

Service virtualization involves nothing more than using test doubles that operate over the wire and is analogous to how you test car batteries without a real alternator. You silo the runtime environment into the bits relevant to test a single service or application

and fake the rest, assuming standard interfaces. In traditional mocking and stubbing libraries, you would stub out a dependency and inject that into your object's constructor, allowing your tests to probe the object under test in isolation. With service virtualization, you virtualize a service and configure the service under test to use the virtualized service's URL as a runtime dependency (figure 1.9). You can set up the virtual service with a specific set of canned responses, allowing you to probe the service under test in isolation.

Figure 1.9. Testing using service virtualization



Service virtualization lets you do black box testing of the service while tightly controlling the runtime environment in which it operates. Although it falls short of the end-to-end confidence that integration tests give you, it does make testing much easier. If you need to test what your shopping cart will do if you try to submit the order when you are out of inventory, you don't have to figure out how to change the inventory system to run your test. You can virtualize the inventory service and configure it to respond with an out-of-inventory message. You can take full advantage of the reduced coupling that the narrow service interface provides to dramatically reduce the amount of test setup required.

Deterministic tests are tests that always pass or fail when given the same code to test. Nondeterminism is a tester's worst enemy. Every time you try to "fix" a broken test by re-running it because it worked last time, the devil on your shoulder does a little dance while the angel on your other shoulder lets out a big sigh. Automated tests create a social contract for a team: when a test breaks, you fix the code. When you allow flaky tests, you chip away at that social contract. All kinds of bad behavior might occur when teams lose confidence that their tests are giving them meaningful feedback, including

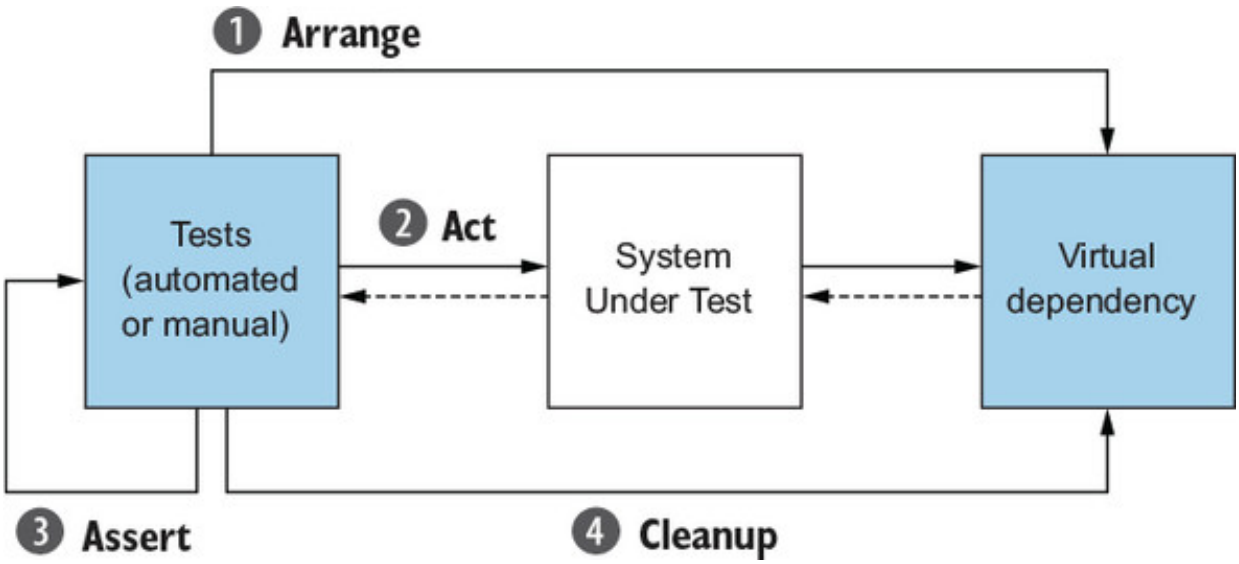
completely ignoring the output of a build.

For your tests to run deterministically, you need to control what the virtual service returns. You can seed the response in several ways, depending on the type and complexity of the tests. What works for writing automated behavioral tests against your service and for testing edge cases likely won't work when running performance tests where you need to execute thousands of requests against the virtual service.

1.3.1. Test-by-test setup using an API

The simplest approach is to mirror what mocking libraries do: directly collude with the mocked objects. The unit testing community often speaks about the *3A pattern*, which is to say that each test has three components: arrange, act, and assert. First you set up the data needed for the test to run (arrange), then you execute the system under test (act), and finally you assert that you got the expected response (figure 1.10). Service virtualization can support this approach through an API that lets you configure the virtual service dynamically.

Figure 1.10. Service virtualization supports a standard unit testing pattern.

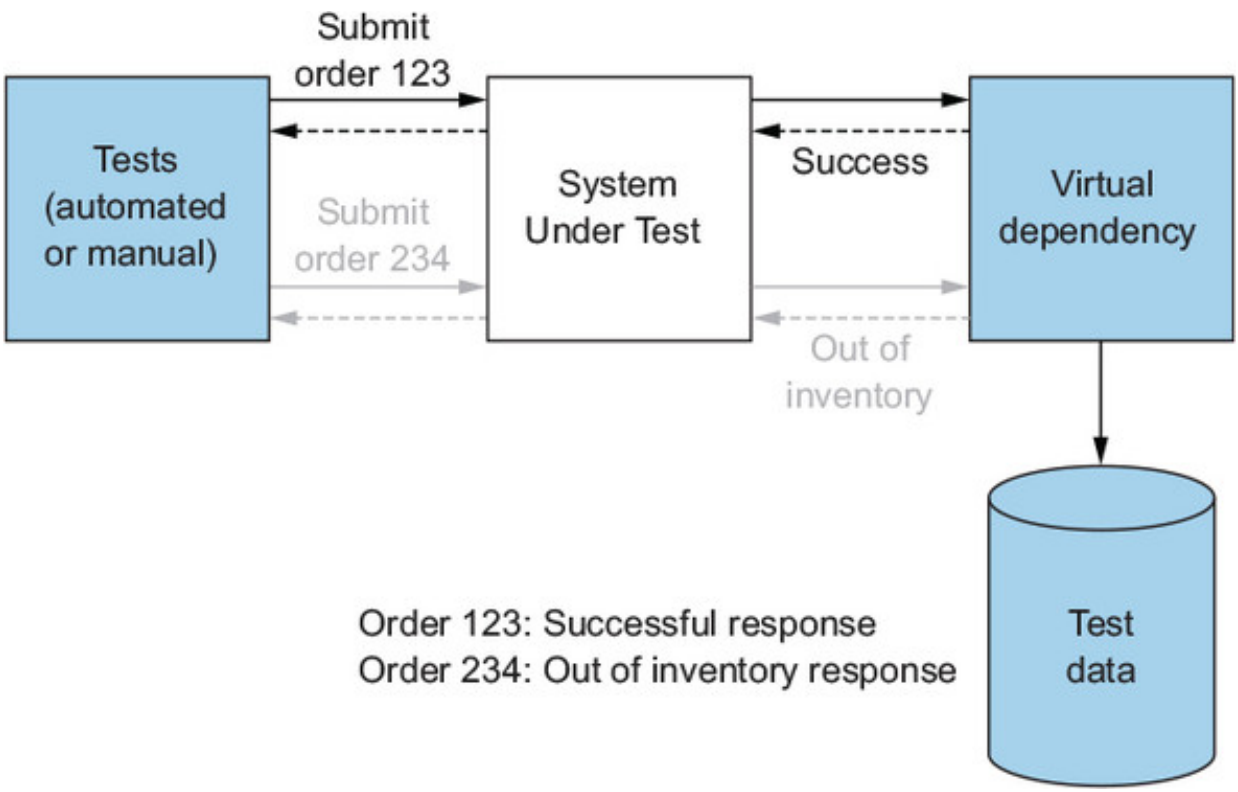


This approach supports creating a laboratory environment for each test, in order to strictly control the inputs and dependencies of the service under test. However, it does make a couple of fundamental assumptions. First, it expects each test to start with a clean slate, which means that each test must remove the test data it added to prevent that data from interfering with subsequent tests. Second, the approach doesn't work if multiple tests are run in parallel. Both of these assumptions fit nicely into automated behavioral tests, as test runners typically ensure tests are run serially. As long as each developer runs their own virtual service, you can avoid the resource contention that comes with concurrent test runs.

1.3.2. Using a persistent data store

Creating test data test-by-test doesn't work well for manual testers, it doesn't work if multiple testers are hitting the same virtual service, and it doesn't work in situations (like performance testing) where you need a large batch of data. To address these concerns, you can configure the virtual service to read the test data from a persistent store. With the test-by-test approach to test data creation, all you have to do is tell the virtual service what the next response should look like. With a data store, you will need some way of deciding which response to send based on something from the request. For example, you might send back different responses based on identifiers in the request URL (figure 1.11).

Figure 1.11. Using persistent test data from a data store



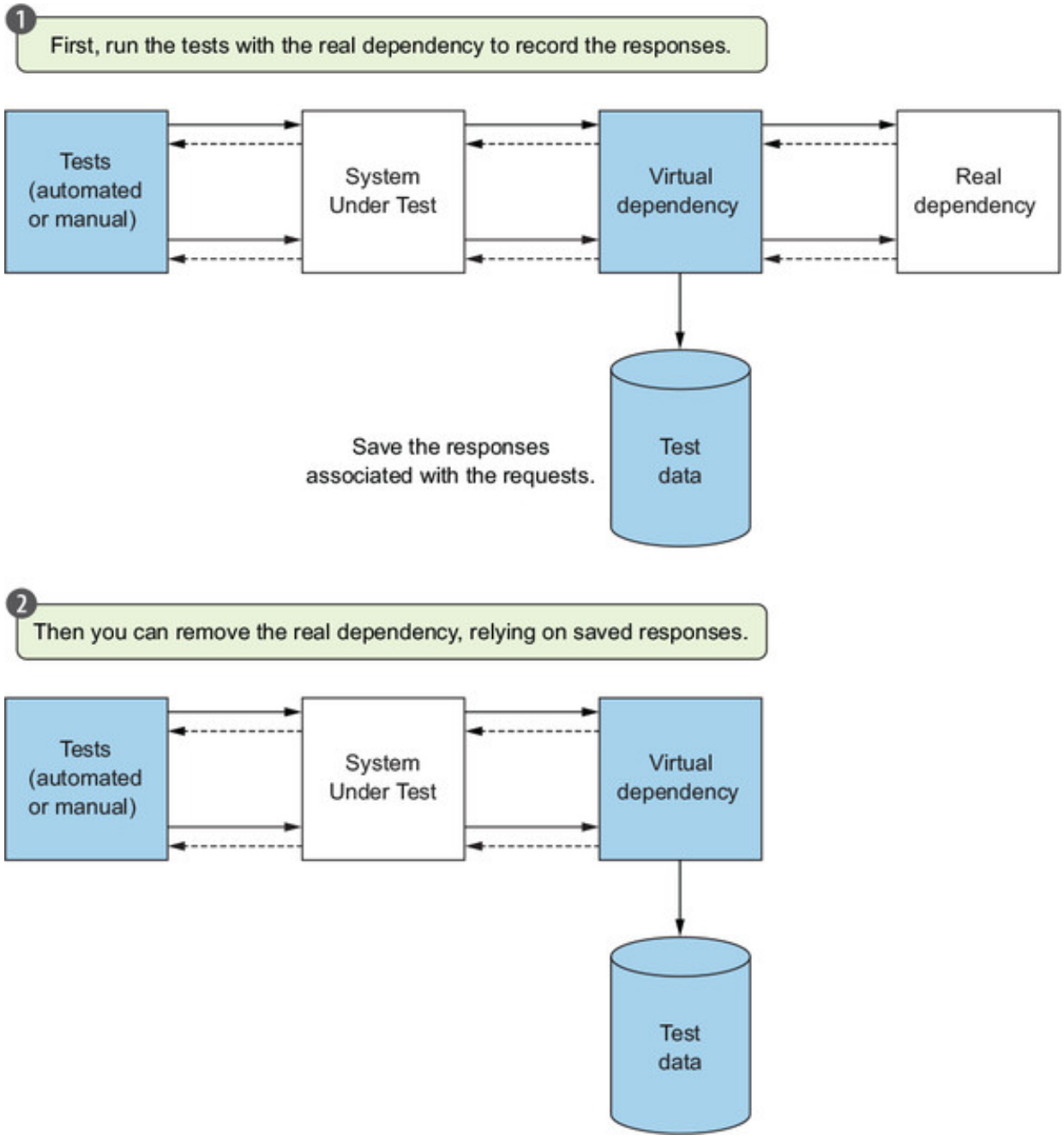
The downside of this approach is that the arrange step is removed from the test, meaning that to understand what each test is trying to do, you need some information that it doesn't directly specify. If you are testing what happens when you submit orders under various scenarios, for example, you'd have to know that order 123 should have appropriate inventory, whereas order 234 should experience an out-of-inventory situation. The configuration that sets that up is in a data store instead of in the arrange section of your test.

1.3.3. Record and replay

Once you have determined where to store the test data, the next question is how to create it. This is rarely an issue for automated behavioral tests because you would

create the data specific to the testing scenario. But if you are using a persistent data store, creating the test data is often a significant challenge, especially when you want large quantities of realistic data. The solution is often to record interactions with the real dependency in a way that allows you to play them back through a virtual service (figure 1.12).

Figure 1.12. Capturing real traffic for later replay



The trick with recording responses is that you still have to specify some condition on the request that has to match before playing back the recorded response. You need to know that the order identifier in the URL is what's used to separate the successful order submit response from the out-of-inventory response.

Service virtualization isn't a silver bullet, and by itself it's not enough to cover the confidence gap created by giving up end-to-end testing. But it's a critical component of

a modern test strategy in a distributed world, and we'll explore ways of closing that confidence gap in chapters 9 and 10, when we combine service virtualization with other techniques to create a continuous delivery pipeline.

Service virtualization isn't just for microservices!

Although the focus here is on microservices and the changes in test strategy they require, service virtualization is a useful tool in many other contexts. A common use case is mobile development, where the mobile team needs to be able to develop independently of the team building the API. The need to compete in the mobile ecosystem has driven many organizations to change their integration approach to one based on HTTP-based APIs, and mobile developers can take advantage of that fact to virtualize the APIs as they develop the front-end code.

1.4. INTRODUCING MOUNTEBANK

Mountebank means a charlatan. It comes from Italian words meaning, literally, to mount a bench, which captures the behavior of snake oil salesmen who duped uninformed consumers into forking over money for quack medicine. It's a useful word for describing what mountebank[4] the tool does, which is to conspire with your tests to dupe the system under test into believing that mountebank is a real runtime dependency.

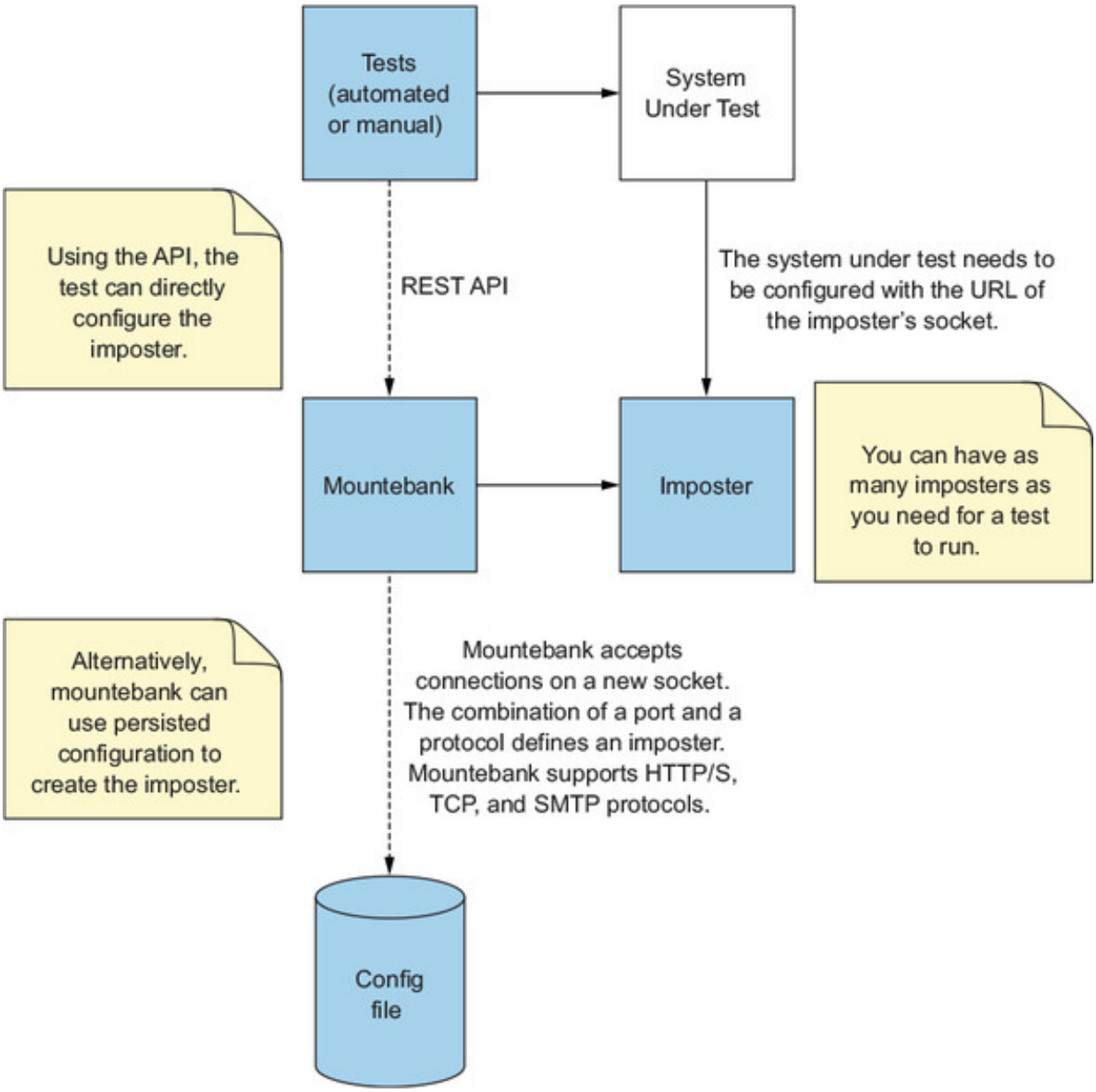
4

Since its initial release, I have always preferred to lowercase the "m" in "mountebank" when writing about the tool. Largely this has to do with the way the documentation is written—in a personified voice from a snake oil salesman who claims false humility by, among other things, not capitalizing his name. Whatever the historical origins, it's now a mildly unexpected stylistic twist. Sorry about that.

Mountebank is a service virtualization tool. It supports all of the service virtualization scenarios we have looked at: behavioral testing using the API test-by-test, using a persistent data store, and acting as a proxy for record and replay situations. Mountebank also supports the ability to pick a response based on certain criteria of the request and to select which request fields you want to differentiate the responses during the playback stage of record-playback. Most of the remainder of this book explores those scenarios and more to help you build a robust test strategy for microservices, because a robust test strategy is the key to unlocking release independence.

Mountebank is a standalone application that provides a REST API to create and configure virtual services, which are called *imposters* in the API (figure 1.13). Rather than configuring the service that you are testing to point to URLs of real services, you configure it to point to the imposters you create through mountebank.

Figure 1.13. Configuring virtual services with a simple mountebank imposter



Each imposter represents a socket that acts as the virtual service and accepts connections from the real service you are testing. Spinning up and shutting down imposters is a lightweight operation, so it's common to use the arrange step of automated tests to create the imposter, then shut it down in the cleanup stage of each test. Although we will use HTTP/S for most examples in this book, mountebank supports other protocols, including binary messages over TCP, and more protocols are expected soon.

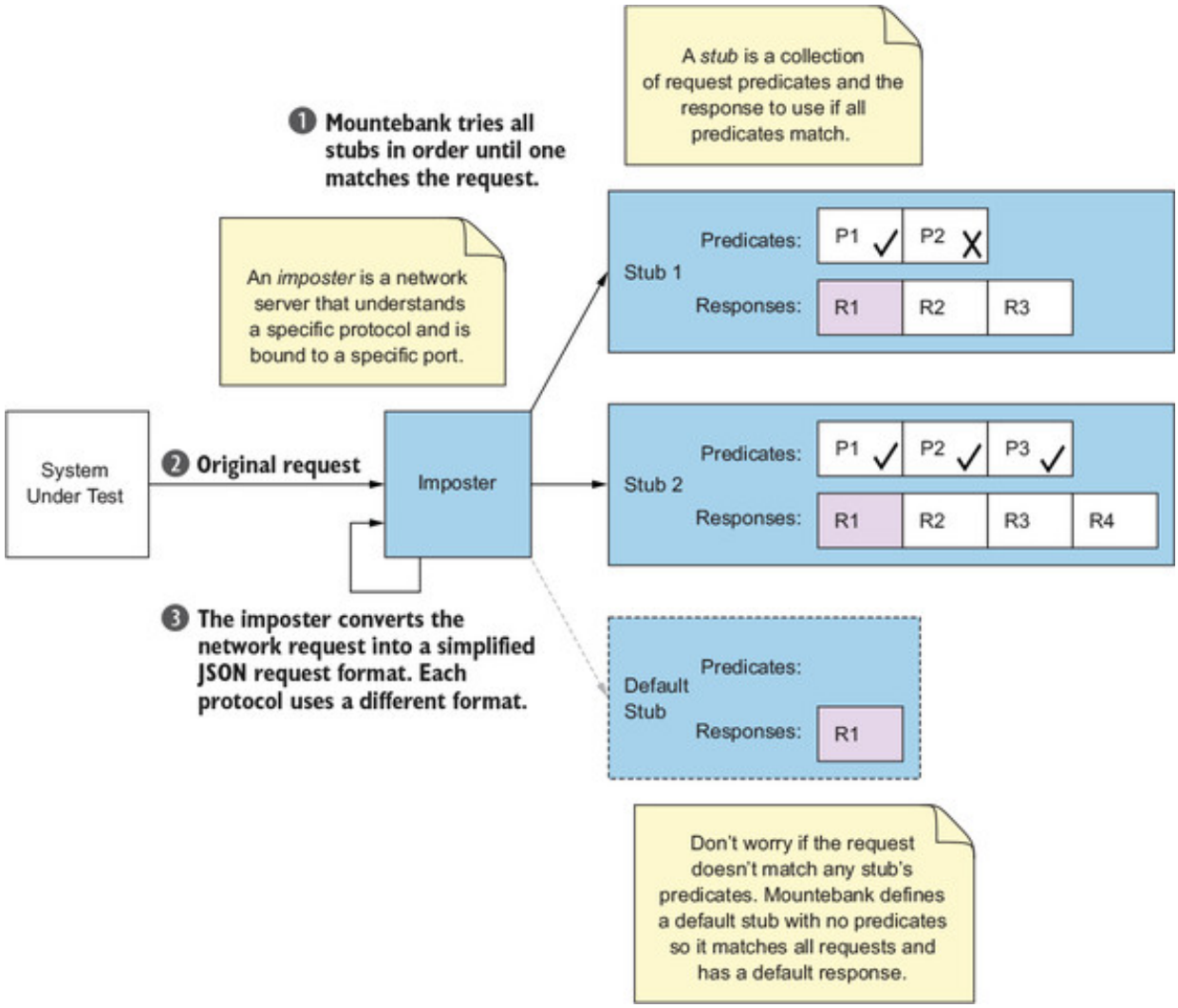
As Larry Wall, the creator of the Perl scripting language, once said, the goal of a good

tool is to make the easy things easy and the hard things possible.[5] Mountebank tries to accomplish this with a rich set of request-matching and response-generation capabilities, balanced by as many defaults as reasonably possible. Figure 1.14 shows how mountebank matches a request to a response.

5

Most developers who have had the misfortune of using many "enterprise" tools will realize that this statement is far from the truism it may sound like.

Figure 1.14. Matching a request to a response with mountebank



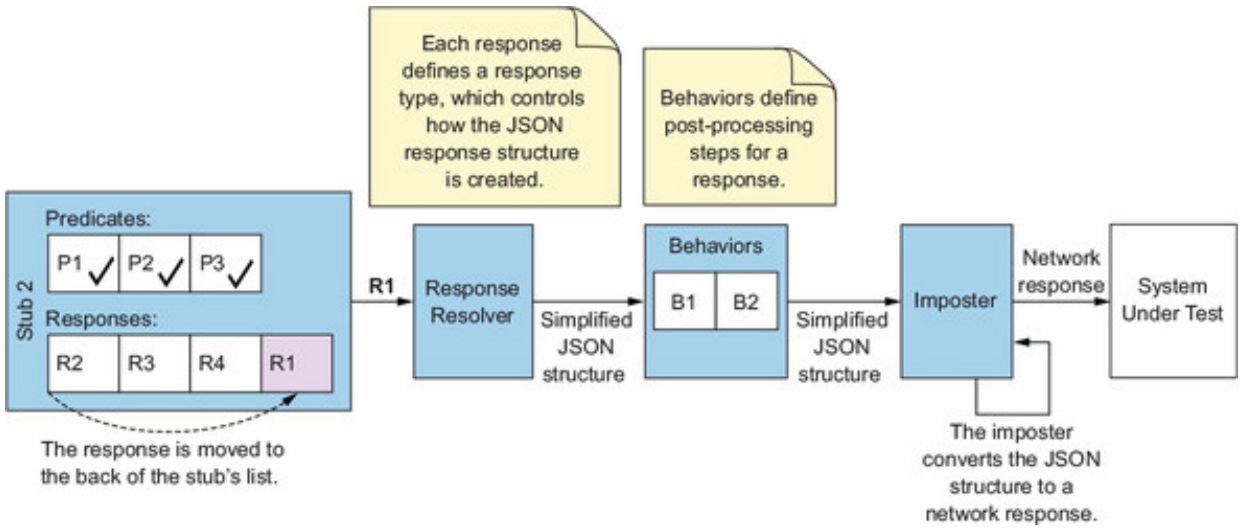
Network protocols are complicated beasts. The first job of the imposter is to simplify a protocol-specific request into a JSON structure so that you can match the request against a set of predicates. Each protocol gets its own request JSON structure; we will look at HTTP's structure in the next chapter.

You configure each imposter with a list of stubs. A *stub* is nothing more than a collection of one or more responses and, optionally, a list of predicates. *Predicates* are defined in terms of request fields, and each one says something like "Request field X must equal 123." No self-respecting mocking tool would leave users with a simple

equals as the only comparison operator, and mountebank ups the game with special predicate extensions to make working with XML and JSON easier. Chapter 4 explores predicates in detail.

Mountebank passes the request to each stub in list order and picks the first one that matches all the predicates. If the request doesn't match any of the stubs defined, mountebank returns a default response. Otherwise, it returns the first response for the stub, which brings us to how responses are generated (figure 1.15).

Figure 1.15. Response generation in mountebank using predicates and responses in stubs



The first thing that happens is that the selected response shifts to the back of the list. This allows you to cycle through the responses in order each time a request matches the stub's predicates. Because mountebank moves the request to the back instead of removing it, you never run out of responses—you can start reusing them. This data structure is what the academics call a *circular buffer*, because a circle prefers to start over rather than end.

The response resolver box in figure 1.15 is a bit of a simplification. Each response is responsible for generating a JSON structure representing the protocol-specific response fields (like the HTTP status code), and you can generate those fields in different ways. Mountebank has three different response types that take entirely different approaches to generating the JSON:

- An *is* response type returns the provided JSON as-is, creating a canned response. We explore canned responses in chapter 3.
- A *proxy* response type forwards the request on to a real dependency and converts its response into a JSON response structure. You use proxies for record-playback functionality, and we describe them in chapter 5.

- An *inject* response type allows you to programmatically define the response JSON using JavaScript. Injection is how you can extend mountebank when its built-in capabilities don't quite do what you need, and we cover that in chapter 6.

Once the response is resolved, mountebank passes the JSON structure to behaviors for post-processing. *Behaviors*, which we discuss in chapter 7, include, among others:

- Copying values from the request into the response
- Adding latency to the response
- Repeating a response, rather than moving it to the back of the list

Up to this point, mountebank has dealt only with JSON, and every operation (with the exception of forwarding a proxy request) has been protocol-agnostic. Once the response JSON is finalized, the imposter converts the JSON to a protocol-aware network response and sends it over the wire. Although we will spend much of our time in this book looking at HTTP requests and responses, all of the core capabilities of mountebank work with any supported network protocol (even binary ones), and in chapter 8, we will show some non-HTTP examples.

To keep simple things simple, nearly everything in mountebank is optional. That allows you to get started gently, which we will do in the next chapter.

1.5. THE SERVICE VIRTUALIZATION TOOL ECOSYSTEM

This book is about two things: mountebank and how service virtualization fits into your microservices test strategy. Although both topics are valuable, the second one is much broader than mountebank.

The service virtualization ecosystem offers several quality tools, both open source and commercial. Commercial tooling is still quite popular in large enterprises. HP, CA, and Parasoft all offer commercial service virtualization tools, and SmartBear took the (originally noncommercial) SoapUI and converted it into part of their commercial service virtualization toolkit. Many of the commercial tools are high quality and offer a richer set of capabilities than the open source tooling, such as broader protocol support, but in my experience, they both devalue the developer experience and hinder true continuous delivery. (Chapter 9 offers a fuller critique.) Of the open source tools, I believe that mountebank comes the closest to the full feature set of commercial tools.

The open source tooling offers a rich set of options primarily aimed at virtualizing HTTP. WireMock is probably the most popular alternative to mountebank. Whereas mountebank aims to be cross-platform by having its public API be REST over HTTP,

WireMock (and many others) optimizes for a specific platform. Although this involves tradeoffs, WireMock is easier to get started with in a purely Java-based project, as you don't have to worry about calling an HTTP API or any complicated wiring into the build process.

Mountebank has an ecosystem of language bindings and build plugins, but you will have to search for them, and they may not expose the full capabilities of the tool. (In the next chapter, you will see an example using JavaScript to wrap the REST API, and chapter 8 has an example using a pre-built C# language binding.) That said, mountebank has broader portability than WireMock.

Another popular example is Hoverfly, a newer Go-based service virtualization tool that baked in middleware as part of the toolchain, allowing a high degree of customization. Mountebank offers middleware in the form of the `shellTransform` behavior, which we look at in chapter 7. Moco and stubby4j are other popular options that are Java-based, although stubby4j has been ported to multiple languages.

As you will see in part 3 of this book, service virtualization helps in a number of scenarios, and one tool isn't always right for every scenario. Many of the commercial tools aim for centralized testing, including performance tests. Many of the open source tools aim for a friendly developer experience when doing functional service tests as part of the development process. I believe mountebank is unique in the open source world in that it aspires to support the full spectrum of service virtualization use cases, including performance testing (which we look at in chapter 10). That said, you won't hurt my feelings if you use another tool for certain types of testing, and I hope that this book helps you identify what you need in the different types of tests to thrive in a microservices world.

SUMMARY

- Microservices represent an architectural approach that can increase both delivery throughput and velocity.
- To realize the full potential of microservices, you must release them independently.
- To gain release independence, you must also test independently.
- Service virtualization allows independent black-box testing of services.
- Mountebank is an open source service virtualization tool for testing microservices.

Chapter 2. Taking mountebank for a test drive

This chapter covers

- Understanding how mountebank virtualizes HTTP
- Installing and running mountebank
- Exploring mountebank on the command line
- Using mountebank in an automated test

In trying to do for pet supplies what Amazon did for books, Pets.com became one of the most spectacular failures of the dot-com bust that occurred around the turn of the millennium. On the surface, the company had everything it needed to be successful, including a brilliant marketing campaign that featured a famous sock puppet. Yet it flamed out from IPO to liquidation in under a year, becoming synonymous with the bursting of the dot-com bubble in the process.

Business-minded folk claim that Pets.com failed because no market existed for ordering pet supplies over the internet. Or it failed because of the lack of a viable business plan…or maybe because the company sold products for less than it cost to buy and distribute them. But as technologists, we know better.

Pets.com made only two mistakes that mattered. They didn't use microservices, and, more importantly, they didn't use mountebank.[1] In an era in which social media and meme generators have conspired to bring cat picture innovation to new heights, it's clear that we need internet-provided pet supplies now more than ever. The time to correct the technical mistakes of Pets.com is long overdue. In this chapter, we will get started on a microservices architecture for a modern pet supply company and show how you can use mountebank to maintain release independence between services.

1

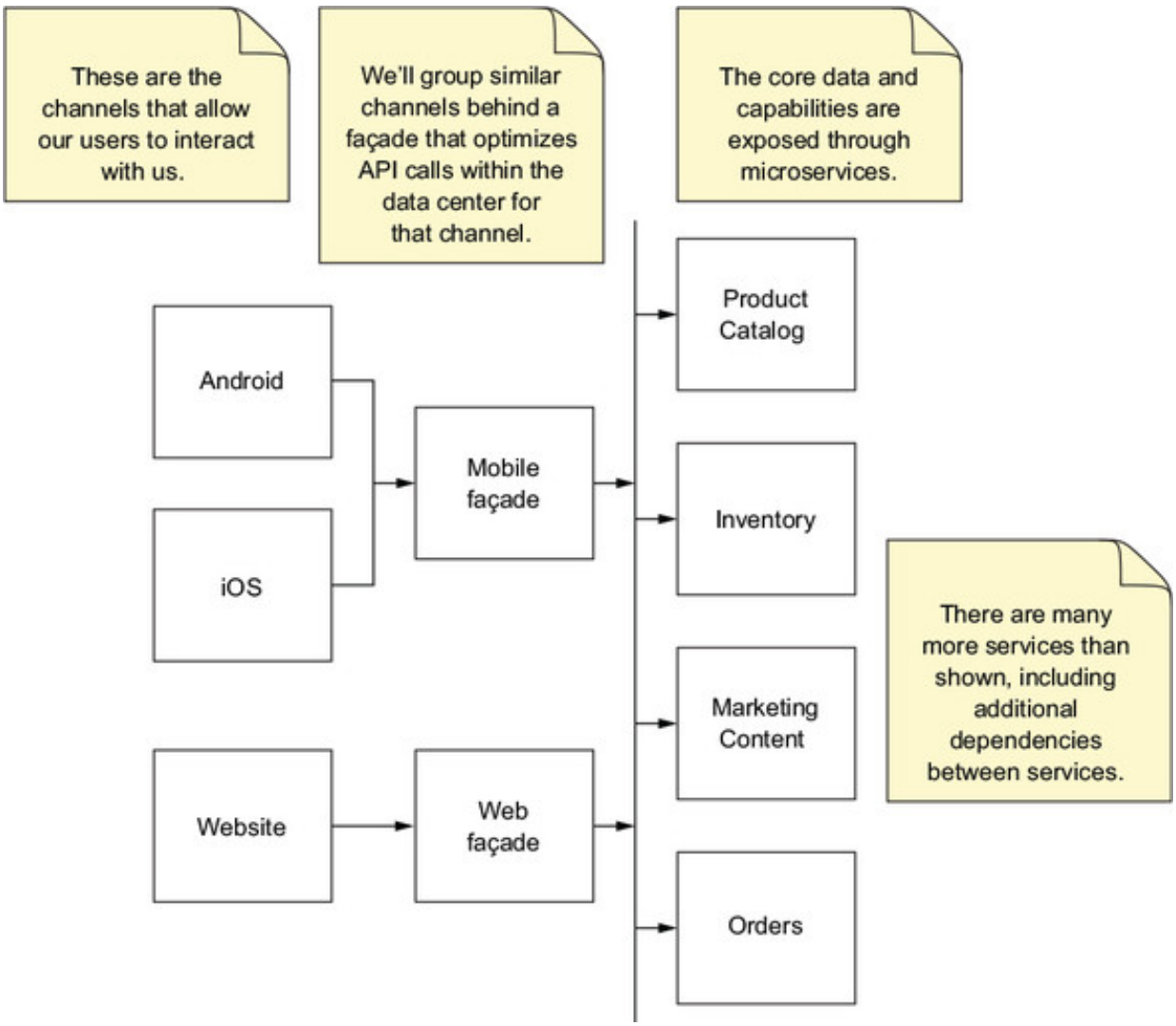
I am, of course, joking. Neither microservices nor mountebank existed back then.

2.1. SETTING UP THE EXAMPLE

Though building an online pet supply site is a bit tongue-in-cheek, it will serve as a useful reference to get comfortable with mountebank. As an e-commerce platform, it looks similar to the Amazon.com example you saw in chapter 1.

The architecture shown in figure 2.1 is simplified, but it's complex enough to work with. Each of the services on the right have its own set of runtime dependencies, but we will look at the architecture from the perspective of the website team. One of the hallmarks of a good architecture is that, although you will need to understand something about your dependencies, you shouldn't need to know anything about the other teams' dependencies. I have also introduced a *façade* layer that represents presentational APIs relevant to a specific channel. This is a common pattern to aggregate and transform downstream service calls into a format optimized for the channels (mobile, web, and so on).

Figure 2.1. Your reference architecture for exploring mountebank



An advantage of using HTTP for integration is that, unlike libraries and frameworks, you can use an API without knowing what language the API was written in.[2] It would be perfectly acceptable, for example, for you to write the product catalog service in Java

and the inventory service in Scala. Indeed, having the ability to make new technology adoption easier is another side benefit of microservices.

2

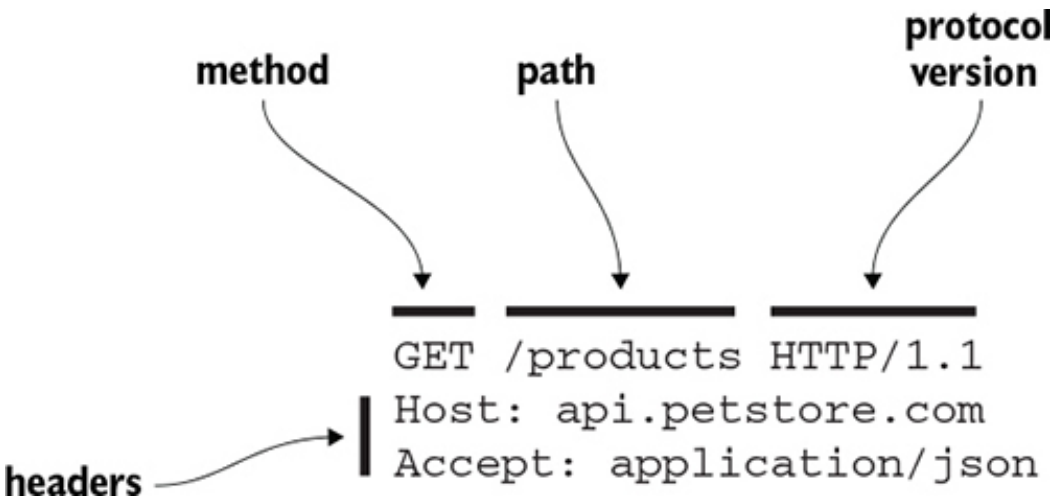
It's this fact that makes mountebank usable in any language.

2.2. HTTP AND MOUNTEBANK: A PRIMER

HTTP is a text-based request-response network protocol. An HTTP server knows how to parse that text into its constituent parts, but it's simple enough that you can parse it without a computer. Mountebank assumes that you are comfortable with those constituent parts. After all, you can't expect to provide a convincing fake of an HTTP service if you don't first understand what a real one looks like.

Let's deep-dive into HTTP using one of the first features you need to support: listing the available products. Fortunately, the product catalog service has an endpoint for retrieving the products in JSON format. All you have to do is make the right API call, which looks like figure 2.2 in HTTP-speak.

Figure 2.2. Breaking down the HTTP request for products

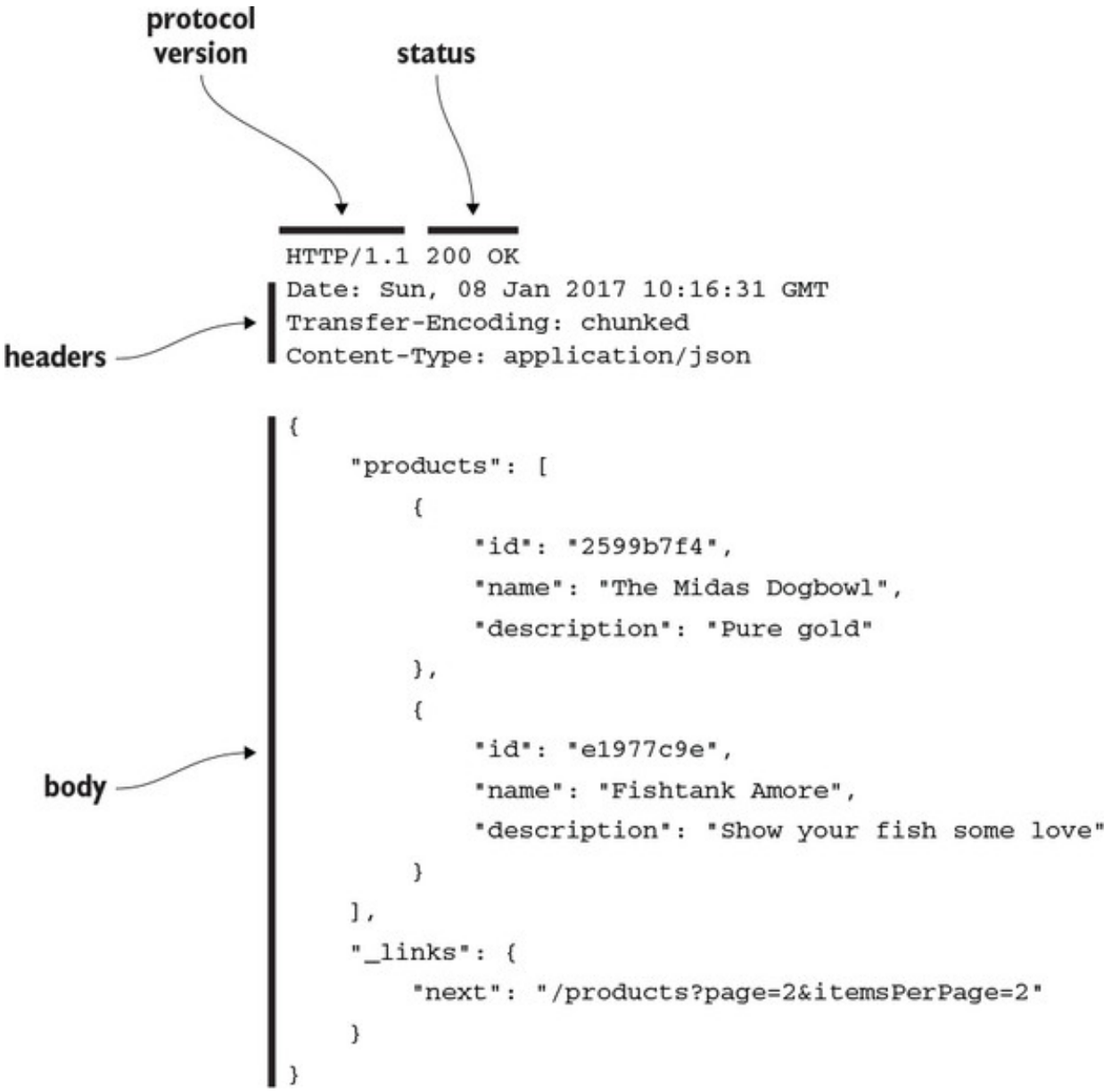


The first line of any HTTP request contains three components: the method, the path, and the protocol version. In this case, the method is `GET`, which denotes that you are retrieving information rather than trying to change state on some server resource. Your path is `/products`, and you are using version 1.1 of the HTTP protocol. The second line starts the headers, a set of newline-separated key-value pairs. In this example, the `Host` header combines with the path and protocol to give the full URL like you would see in a browser: http://api.petstore.com/products. The `Accept` header tells the server that you are expecting JSON back.

When the product catalog service receives that request, it returns a response that looks

like figure 2.3. A real service presumably would have many more data fields and many more items per page, but I have simplified the response to keep it digestible.

Figure 2.3. The response from the product catalog

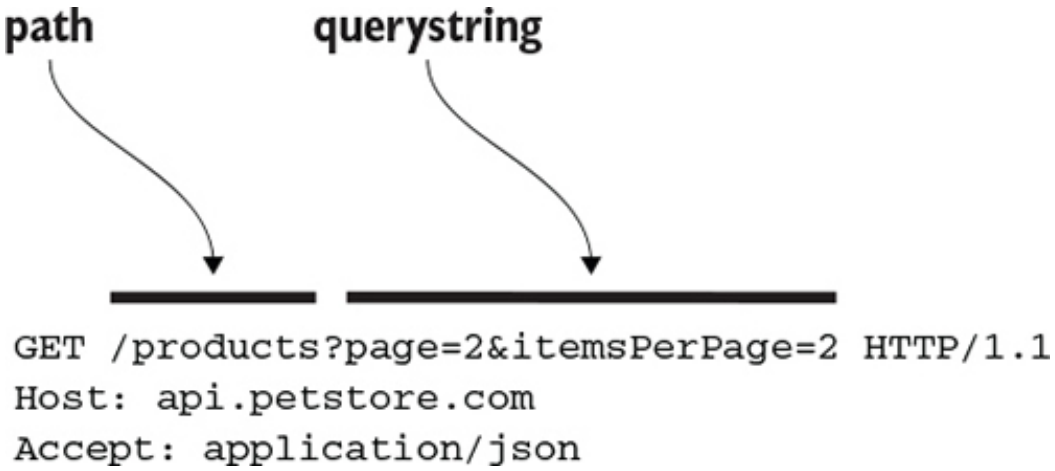


A high degree of symmetry exists between HTTP requests and responses. As with the first line of the request, the first line of the response contains metadata, although for responses the most important metadata field is the status code. A 200 status is HTTP-speak for success, but in case you forgot, it tells you with the word `OK` following the code. Other codes have other words that go with them, like `BAD REQUEST` for a 400, but the text doesn't serve any purpose other than a helpful hint. The libraries that you use for integrating with HTTP services only care about the code, not the text.

The headers once again follow the metadata, but here you see the HTTP body. The body is always separated from the headers by an empty line, and even though your HTTP request did not have a body, you will see plenty of examples in this book that do.

This particular body includes a link to the next page of results, which is a common pattern for implementing paging in services. If you were to craft the HTTP request that follows the link, it would look similar to the first request, as shown in figure 2.4.

Figure 2.4. Adding a query parameter to an HTTP request



The difference appears to be in the path, but every HTTP library that I'm aware of would give you the same path for both the first and second request. Everything after the question mark denotes what is called the *querystring*. (Mountebank calls it the *query*.) Like the headers, the query is a set of key-value pairs, but they are separated by the `&` character and included in the URL, separated from the path with a `?` character.

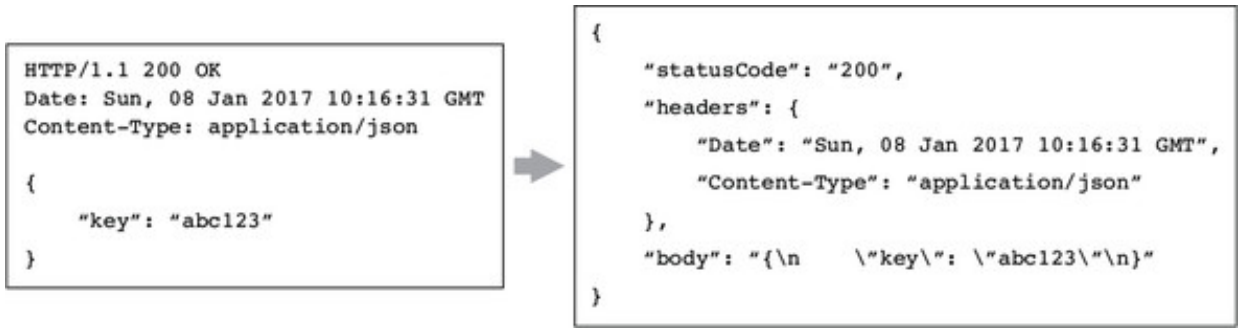
HTTP can attribute much of its success to its simplicity. The textual format makes it almost as easy for pizza-fueled computer programmers to read as it is for electricity-fueled computers to parse. That's good for you because writing virtual services requires you to understand the protocol-specific request and response formats, which are treated as simple JSON objects that mimic closely the standard data structures used by HTTP libraries in any language. To generalize, figure 2.5 shows how mountebank translates an HTTP request.

Figure 2.5. How mountebank views an HTTP request

Notice in figure 2.5 that even though the body is represented in JSON, HTTP itself doesn't understand JSON, which is why the JSON is represented as a simple string value. In later chapters, we will look at how mountebank makes working with JSON easier.

Figure 2.6 shows how mountebank represents an HTTP response.

Figure 2.6. How mountebank represents an HTTP response



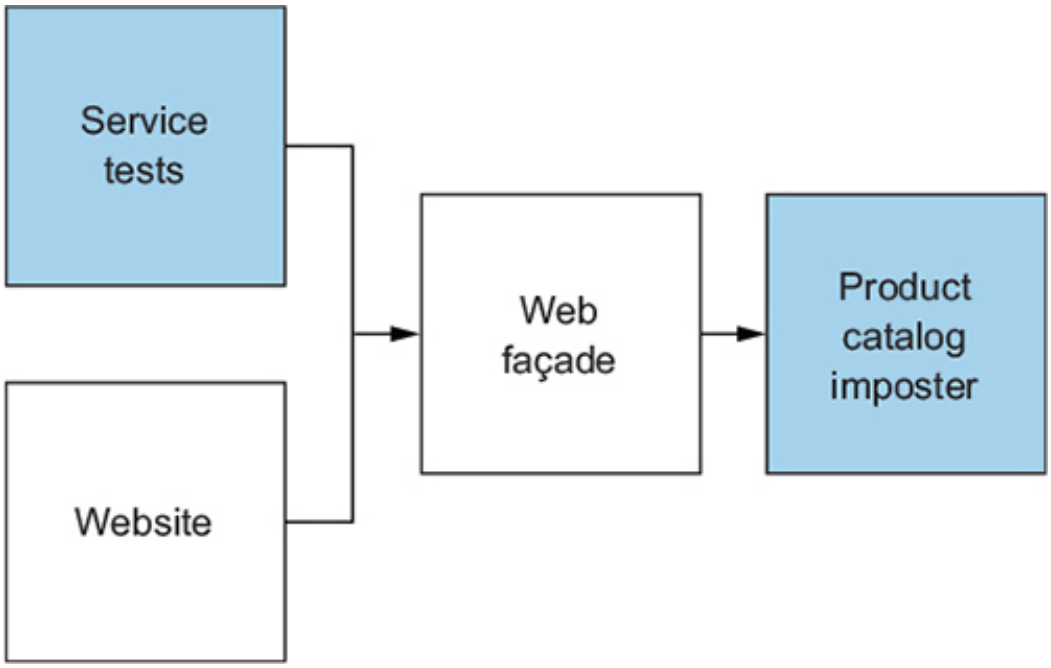
This type of translation happens for all the protocols mountebank supports—simplifying the application protocol details into a JSON representation. Each protocol gets its own JSON representation for both requests and responses. The core functionality of mountebank performs operations on those JSON objects, blissfully unaware of the semantics of the protocol. Aside from the servers and proxies to listen to and forward network requests, the core functionality in mountebank is protocol-agnostic.

Now that you've seen how to translate HTTP semantics to mountebank, it's time to create your first virtual service.

2.3. VIRTUALIZING THE PRODUCT CATALOG SERVICE

Once you understand how to integrate your codebase with a service, the next step is to figure out how to virtualize it for testing purposes. Continuing with our example, let's virtualize the product catalog service so you can test the web façade in isolation (figure 2.7).

Figure 2.7. Virtualizing the product catalog service to test the web facade



Remember, an *imposter* is the mountebank term for a virtual service. Mountebank ships with a REST API that lets you create imposters and write tests against them in any language.

Mountebanks, imposters, and funny sounding docs

Much of the mountebank documentation is written in the voice of a true mountebank, prone to hyperbole and false modesty, where even the word "mountebank" shifts from describing the tool itself to the author of the documentation (yours truly). When I originally wrote the tool, I made *imposters* the core domain concept, in part because it fit the theme of using synonyms for charlatans to describe fake services, and in part because it self-deprecatingly made fun of my own Impostor Syndrome, a chronic ailment of consultants like myself. And yes, as Paul Hammant (one of the original creators of the popular Selenium testing tool and one of the first users of mountebank) pointed out to me, impostor (with an "or" instead of "er" at the end) is the "proper" spelling. Now that mountebank is a popular tool used all over the world, complete with a best-selling book (the one you are holding), Paul also helpfully suggested that I change the docs to remove the hipster humor. Unfortunately, he has yet to indicate where I'm supposed to find the time for such pursuits.

Before you start, you will need to install mountebank. The website, http://www.mbtest.org/docs/install, lists several installation options, but you'll use `npm`, a package manager that ships with node.js, by typing the following in a terminal window:

```
npm install -g mountebank
```

The `-g` flag tells `npm` to install mountebank globally, so you can run it from any directory. Let's start it up:

```
mb
```

You should see the mountebank log on the terminal:

```
info: [mb:2525] mountebank v1.13.0 now taking orders -
➥ point your browser to http://localhost:2525 for help
```

The log will prove invaluable in working with mountebank in the future, so it's a good idea to familiarize yourself with it. The first word (`info`, in this case) tells you the log level, which will be either `debug`, `info`, `warn`, or `error`. The part in brackets (`mb:2525`) tells you the protocol and port and is followed by the log message. The administrative port logs as the `mb` protocol and starts on port 2525 by default. (The `mb` protocol is HTTP, but mountebank logs it differently to make it easy to spot.) The imposters you create will use different ports but log to the same output stream in the terminal. The startup log message directs you to open http://localhost:2525 in your web browser, which will provide you the complete set of documentation for the version of mountebank you are running.

To demonstrate creating imposters, you will use a utility called `curl`, which lets you make HTTP calls on the command line. `curl` comes by default on most Unix-like shells, including Linux and macOS. You can install it on Windows using Cygwin, or use PowerShell, which ships with modern versions of Windows. (We will show a PowerShell example next.) Open another terminal window and run the code shown in the following listing.[3]

3 To avoid carpal tunnel syndrome, you can download the source at https://github.com/bbyars/mountebank-in-action.

Listing 2.1. Creating an imposter on the command line

```
curl -X POST http://localhost:2525/imposters --data '{      1
  "port": 3000,                                             2
  "protocol": "http",                                       2
  "stubs": [{
    "responses": [{
```

```
        "is": {                                                    3
          "statusCode": 200,                                       3
          "headers": {"Content-Type": "application/json"},         3
          "body": {                                                3
            "products": [                                          3
              {                                                    3
                "id": "2599b7f4",                                  3
                "name": "The Midas Dogbowl",                       3
                "description": "Pure gold"                         3
              },                                                   3
              {                                                    3
                "id": "e1977c9e",                                  3
                "name": "Fishtank Amore",                          3
                "description": "Show your fish some love"          3
              }                                                    3
            ],                                                     3
            "_links": {                                            3
              "next": "/products?page=2&itemsPerPage=2"            3
            }                                                      3
          }                                                        3
        }                                                          3
      }]
    }]
}'
```

- ***1* Creates new imposters**
- ***2* Minimally defines each imposter by a port and a protocol**
- ***3* Defines a canned HTTP response**

An important point to note is that you are passing a JSON object as the `body` field. As far as HTTP is concerned, a response body is a stream of bytes. Usually HTTP interprets that stream as a string, which is why mountebank typically expects a string as well.[4] That said, most services these days use JSON as their *lingua franca*. Mountebank, being itself a modern JSON-speaking service, can properly accept a JSON `body`.

4

Mountebank supports binary response bodies, encoding them with Base64. We look at binary support in chapter 8.

The equivalent command on PowerShell in Windows expects you to save the request body in a file and pass it in to the `Invoke-RestMethod` command. Save the JSON after the `--data` parameter from the `curl` command code above into a file called imposter.json, then run the following command from the same directory:

```
Invoke-RestMethod -Method POST -Uri http://localhost:2525/imposters
➥ -InFile imposter.json
```

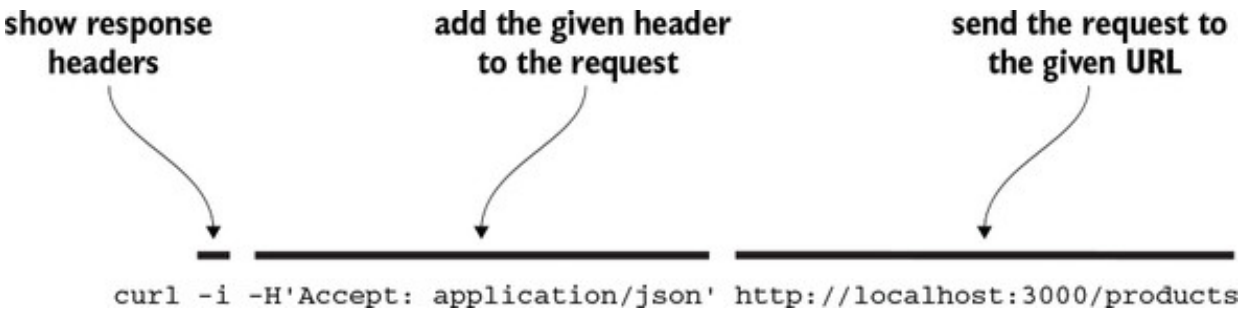
Notice what happens in the logs:

```
info: [http:3000] Open for business...
```

The part in brackets now shows the new imposter. As you add more imposters, this will become increasingly important. You can disambiguate all log entries by looking at the imposter information that prefixes the log message.

You can test your imposter on the command line as well, using the `curl` command we looked at previously, as shown in figure 2.8.

Figure 2.8. Using `curl` to send a request to your virtual product catalog service



The `curl` command prints out the HTTP response as shown in the following listing.

Listing 2.2. The HTTP response from the `curl` command

```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Date: Thu, 19 Jan 2017 14:51:23 GMT
Transfer-Encoding: chunked

{
  "products": [
    {
      "id": "2599b7f4",
      "name": "The Midas Dogbowl",
      "description": "Pure gold"
    },
    {
      "id": "e1977c9e",
      "name": "Fishtank Amore",
      "description": "Show your fish some love"
    }
```

```
    ],
    "_links": {
      "next": "/products?page=2&itemsPerPage=2"
    }
  }
```

That HTTP response includes a couple of extra headers, and the date has changed, but other than that, it's exactly the same as the real one returned by the service shown in figure 2.3. You aren't accounting for all situations though. The imposter will return exactly the same response no matter what the HTTP request looks like. You could fix that by adding predicates to your imposter configuration.

As a reminder, a *predicate* is a set of criteria that the incoming request must match before mountebank will send the associated response. Let's create an imposter that only has two products to serve up. We will use a predicate on the query parameter to show an empty result set on the request to the second page. For now, restart `mb` to free up port 3000 by pressing Ctrl-C and typing `mb` again. (You will see more elegant ways of cleaning up after yourself shortly.) Then use the command shown in the following listing in a separate terminal.

Listing 2.3. An imposter with predicates

```
curl -X POST http://localhost:2525/imposters --data '{
  "port": 3000,
  "protocol": "http",
  "stubs": [                                                    1
    {
      "predicates": [{
        "equals": {                                             2
          "query": { "page": "2" }                              2
        }                                                       2
      }],
      "responses": [{
        "is": {                                                 3
          "statusCode": 200,                                    3
          "headers": {"Content-Type": "application/json"},      3
          "body": { "products": [] }                            3
        }                                                       3
      }]
    },
    {
      "responses": [{
        "is": {                                                 4
          "statusCode": 200,                                    4
          "headers": { "Content-Type": "application/json" },    4
          "body": {                                             4
            "products": [                                       4
              {                                                 4
```

```
                  "id": "2599b7f4",                                    4
                  "name": "The Midas Dogbowl",                         4
                  "description": "Pure gold"                           4
                },                                                     4
                {                                                      4
                  "id": "e1977c9e",                                    4
                  "name": "Fishtank Amore",                            4
                  "description": "Show your fish some love"            4
                }                                                      4
              ],                                                       4
              "_links": {                                              4
                "next": "/products?page=2&itemsPerPage=2"              4
              }                                                        4
            }                                                          4
          }                                                            4
        }]
      }
    ]
  }'
```

- ***1* Using two stubs allows different responses for different requests.**
- ***2* Requires that the request querystring include page=2**
- ***3* Sends this response if the request matches the predicate**
- ***4* Otherwise, sends this response**

Now, if you send a request to the imposter without a querystring, you'll get the same response as before. But adding `page=2` to the querystring gives you an empty product list:

```
curl -i http://localhost:3000/products?page=2

HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Date: Sun, 21 May 2017 17:19:17 GMT
Transfer-Encoding: chunked

{
    "products": []
}
```

Exploring the mountebank API on the command line is a great way to get familiar with it and to try sample imposter configurations. If you change the configuration of your web façade to point to http://localhost:3000 instead of https://api.petstore.com, you will get the products we have defined and can manually test the website. You have already

taken a huge step toward decoupling yourself from the real services.

Postman as an alternative to the command line

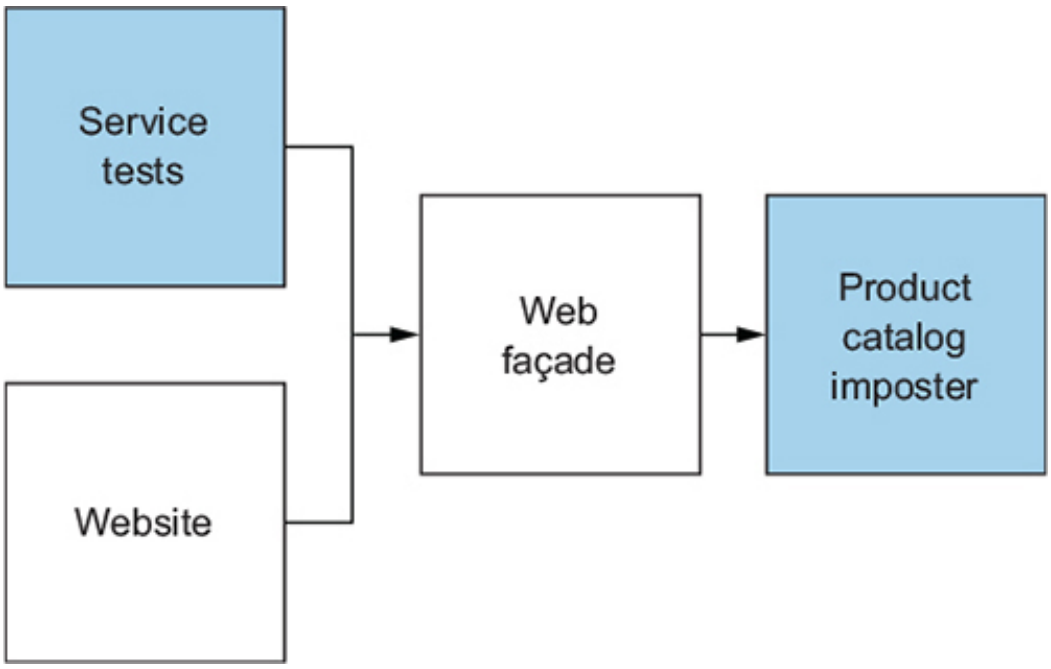
Although using command-line tools like `curl` is great for lightweight experimentation and perfect for the book format, it's often useful to have a more graphical approach to organize different HTTP requests. Postman (https://www.getpostman.com/) has proven to be an extremely useful tool for playing with HTTP APIs. It started out as a Chrome plugin but now has downloads for Mac, Windows, and Linux. It lets you fill in the various HTTP request fields and save requests for future use.

That said, the real benefit of service virtualization is in enabling automated testing. Let's see how you can wire up mountebank to your test suite.

2.4. YOUR FIRST TEST

To properly display the products on the website, the web façade needs to combine the data that comes from the product catalog service with marketing copy that comes from a marketing content service (figure 2.9). You will add tests that verify that the data that gets to the website is valid.

Figure 2.9. Combining product data with marketing copy



The data that the web façade provides to the website should show both the product catalog data and the marketing content. The response from the web façade should look like the following listing.

Listing 2.4. Combining product data with marketing content

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 19 Jan 2017 15:43:21 GMT
Transfer-Encoding: chunked

{
  "products": [
    {
      "id": "2599b7f4",                                  1
      "name": "The Midas Dogbowl",                       2
      "description": "Pure gold",                        2
      "copy": "Treat your dog like the king he is",      3
      "image": "/content/c5b221e2"                       3
    },
    {
      "id": "e1977c9e",
      "name": "Fishtank Amore",
      "description": "Show your fish some love",
      "copy": "Love your fish; they'll love you back",
      "image": "/content/a0fad9fb"
    }
  ],
  "_links": {
    "next": "/products?page=2&itemsPerPage=2"
  }
}
```

- ***1* Comes from the product catalog service, but also will be used to look up content**
- ***2* Comes from the product catalog service**
- ***3* Comes from the marketing content service**

Let's write a service test that validates that *if* the product catalog and content services return the given data, *then* the web façade will combine the data as shown above. Although mountebank's HTTP API allows you to use it in any language, you will use JavaScript for the example. The first thing you will need to do is make it easy to create imposters from your tests. A common approach to make building a complex configuration easier is to use what is known as a fluent interface, which allows you to chain function calls together to build a complex configuration incrementally.

The code in listing 2.5 uses a fluent interface to build up the imposter configuration in code. Each `withStub` call creates a new stub on the imposter, and each `matchingRequest` and `respondingWith` call adds a predicate and response, respectively, to the stub. When you are done, you call `create` to use mountebank's

REST API to create the imposter.

Listing 2.5. Using a fluent interface to build imposters in code

```
require('any-promise/register/q');                    1
var request = require('request-promise-any');         1

module.exports = function (options) {                 2
  var config = options || {};
  config.stubs = [];

  function create () {                                3
    return request({
      method: "POST",
      uri: "http://localhost:2525/imposters",
      json: true,
      body: config
    });
  }

  function withStub () {                              4
    var stub = { responses: [], predicates: [] },
      builders = {
        matchingRequest: function (predicate) {       5
          stub.predicates.push(predicate);
          return builders;
        },
        respondingWith: function (response) {         6
          stub.responses.push({ is: response });
          return builders;
        },
        create: create,
        withStub: withStub
      };

    config.stubs.push(stub);
    return builders;
  }

  return {
    withStub: withStub,
    create: create
  };
};
```

- ***1*** **node.js libraries that make calling HTTP services easier**
- ***2*** **The node.js way of exposing a function to different files**
- ***3*** **Calls the REST API to create an imposter**
- ***4*** **The entry point to the fluent interface—each call creates a new stub**

- ***5*** **Adds a new request predicate to the stub**
- ***6*** **Adds a new response to the stub**

JavaScript: ES5 vs. ES2015

Modern JavaScript syntax is defined in the version of the EcmaScript (ES) specification. At the time of this writing, ES2015, which adds a bunch of syntactic bells and whistles, is seeing wide adoption, but ES5 still has the broadest support. Although those syntactic bells and whistles are nice once you get used to them, they make the code a little more opaque for non-JavaScript developers. Because this isn't a book on JavaScript, I use ES5 here to keep the focus on mountebank.

You will see how the fluent interface makes the consuming code more elegant shortly. The key to making it work is exposing the `create` and `withStub` functions in the builder, which allows you to chain functions together to build the entire configuration and send it to mountebank.

Assuming you saved the code above in a file called imposter.js, you can use it to create the product catalog service response on port 3000. The code in listing 2.6 replicates what you did earlier on the command line and shows how the function chaining that the fluent interface gives you makes the code easier to follow. Save the following code in test.js.[5]

Listing 2.6. Creating the product imposter in code

```
var imposter = require('./imposter'),                    1
  productPort = 3000;

function createProductImposter() {
  return imposter({                                      2
      port: productPort,
      protocol: "http",
      name: "Product Service"
    })
    .withStub()
    .matchingRequest({equals: {path: "/products"}})      3
    .respondingWith({                                    4
      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"
          }
        ]
      }
    })
    .create();                                          5
}
```

- ***1* Imports your fluent interface**
- ***2* Passes the root-level information into the entry function**
- ***3* Adds the request predicate**
- ***4* Adds the response**
- ***5* Sends a POST to the mountebank endpoint to create the imposter**

It's worth noting a couple of points about the way you are creating the product catalog imposter. First, you have added a `name` to the imposter. The `name` field doesn't change any behavior in mountebank other than the way the logs format messages. The `name` will be included in the text in brackets to make it easier to understand log messages by imposter. If you look at the mountebank logs after you create this imposter, you will see the `name` echoed:

```
info: [http:3000 Product Service] Open for business...
```

That's a lot easier than having to remember the port each imposter is running on.

The second thing to note is that you are adding a predicate to match the `path`. This isn't strictly necessary, as your test will correctly pass without it if the web façade code is doing its job. However, adding the predicate makes the test better. It not only verifies the behavior of the façade given the response, it also verifies that the façade makes the right request to the product service.

We haven't looked at the marketing content service yet. It accepts a list of IDs on a querystring and returns a set of content entries for each ID provided. The code in the following listing creates an imposter using the same IDs that the product catalog service

provides. (Add this to the test.js file you created previously.)

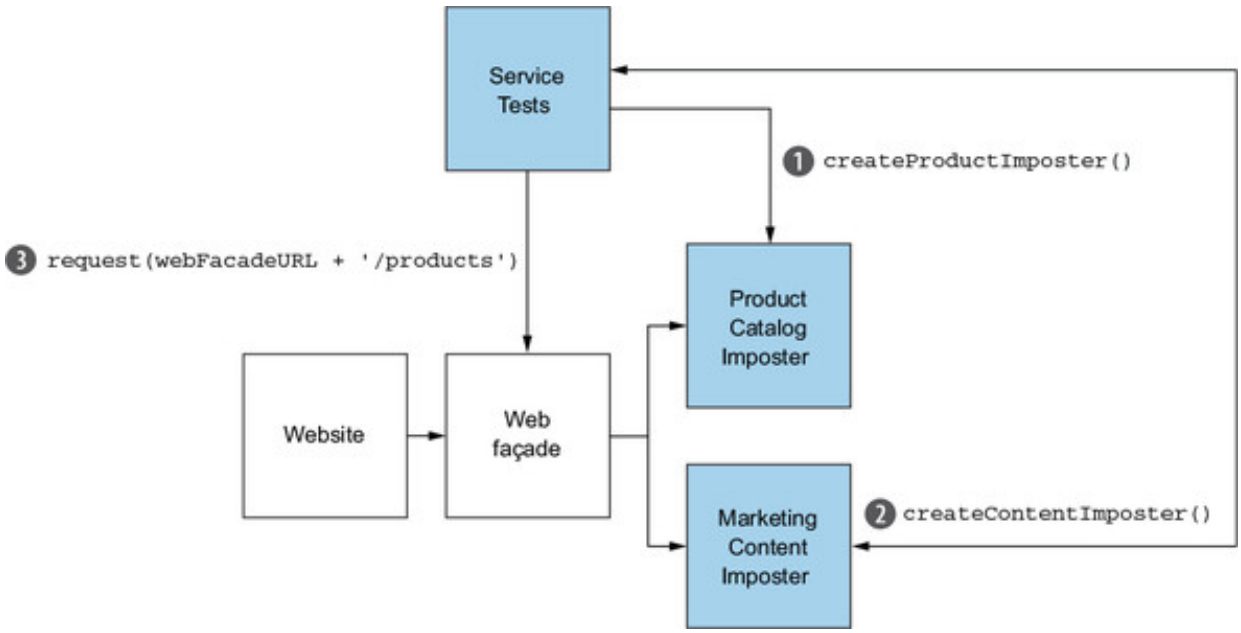
Listing 2.7. Creating the content imposter

```
var contentPort = 4000;

function createContentImposter() {
  return imposter({
      port: contentPort,
      protocol: "http",
      name: "Content Service"
    })
    .withStub()
    .matchingRequest({
      equals: {                                                    1
        path: "/content",                                          1
        query: { ids: "2599b7f4,e1977c9e" }                        1
      }                                                            1
    })
    .respondingWith({
      statusCode: 200,
      headers: {"Content-Type": "application/json"},
      body: {
        content: [                                                 2
          {                                                        2
            id: "2599b7f4",                                        2
            copy: "Treat your dog like the king he is",            2
            image: "/content/c5b221e2"                             2
          },                                                       2
          {                                                        2
            id: "e1977c9e",                                        2
            copy: "Love your fish; they'll love you back",         2
            image: "/content/a0fad9fb"                             2
          }                                                        2
        ]                                                          2
      }
    })
    .create();
}
```

- ***1* Only respond if the path and query match as shown.**
- ***2* The entries that the content service would return**

Armed with the `createProductImposter` and `createContentImposter` functions, you now can write a service test that calls the web façade over the wire and verifies that it aggregates the data from the product catalog and marketing content services appropriately (figure 2.10).

Figure 2.10. The steps of the service test to verify web façade's data aggregating



For this step, you will use a JavaScript test runner called Mocha, which wraps each test in an `it` function and collections of tests in a `describe` function (similar to a test class in other languages). Finish off the test.js file you have been creating by adding the code in the following listing.

Listing 2.8. Verifying the web façade

```
require('any-promise/register/q');
var request = require('request-promise-any'),
  assert = require('assert'),
  webFacadeURL = 'http://localhost:2000';

describe('/products', function () {                                    1
  it('combines product and content data', function (done) {            2
    createProductImposter().then(function () {                         3
      return createContentImposter();                                  3
    }).then(function () {
      return request(webFacadeURL + '/products');                      4
    }).then(function (body) {
      var products = JSON.parse(body).products;

      assert.deepEqual(products, [                                     5
        {
          "id": "2599b7f4",
          "name": "The Midas Dogbowl",
          "description": "Pure gold",
          "copy": "Treat your dog like the king he is",
          "image": "/content/c5b221e2"
        },
        {
          "id": "e1977c9e",
          "name": "Fishtank Amore",
          "description": "Show your fish some love",
          "copy": "Love your fish; they'll love you back",
```

```
            "image": "/content/a0fad9fb"
          }
        ]);
        return imposter().destroyAll();                    6
      }).then(function () {
        done();                                            7
      });
    });
  });
```

- *1* **Mocha groups multiple tests in a describe function.**
- *2* **Each it function represents a single test.**
- *3* **Arrange**
- *4* **Act**
- *5* **Assert**
- *6* **Cleanup**
- *7* **Tells mocha that the asynchronous test is finished**

Notice that you added one step to the test to clean up the imposters. Mountebank supports a couple ways of removing imposters. You can remove a single imposter by sending a `DELETE` HTTP request to the `/imposters/:port` URL (where :port represents the port of the imposter), or remove all imposters in a single call by issuing a `DELETE` request to `/imposters`. Add them to your imposter fluent interface in imposter.js, as shown in the following listing.

Listing 2.9. Adding the ability to remove imposters

```
function destroy () {
  return request({
    method: "DELETE",
    uri: "http://localhost:2525/imposters/" + config.port          1
  });
}
function destroyAll () {
  return request({
    method: "DELETE",
    uri: "http://localhost:2525/imposters"
  });
}
```

- *1* **Passes in the config object, as in listing 2.5**

Whew! You now have a complete service test that verifies some fairly complex

aggregation logic of a service in a black-box fashion by virtualizing its runtime dependencies. (You had to create some scaffolding, but you will be able to reuse the imposters.js module in all of your tests moving forward.) The prerequisites for running this test are that both the web façade and mountebank are running, and you have configured the web façade to use the appropriate URLs for the imposters (http://localhost:3000 for the product catalog service, and http://localhost:4000 for the marketing content service).[6]

JavaScript promises

Your test code relies on a concept called *promises* to make it easier to follow. JavaScript hasn't traditionally had any I/O, and when node.js added I/O capability, it did so in what is known as a nonblocking manner. This means that system calls that need to read or write data to something other than memory are done asynchronously. The application requests the operating system to read from disk, or from the network, and then moves on to other activities while waiting for the operating system to return. For a web service like the kind you are building, "other activities" would include processing new HTTP requests.

The traditional way of telling node.js what to do when the operating system has finished the operation is to register a callback function. In fact, the `request` library that you are using to make HTTP calls works this way by default, as shown in this callback-based HTTP request:

```
var request = require('request');
request('http://localhost:4000/products', function (error, response, body)
  // Process the response here
})
```

The problem with this approach is that it gets unwieldy to nest multiple callbacks, and downright tricky to figure out how to loop over a sequence of multiple asynchronous calls. With promises, asynchronous operations return an object that has a `then` function, which serves the same purpose as the callback. But promises add all kinds of simplifications to make combining complex asynchronous operations easier. You will use them in your tests to make the code easier to read.

Part 3 of this book will show more fully worked-out automated tests and how to include

them in a continuous delivery pipeline. First, though, you need to get familiar with the capabilities of mountebank. Part 2 breaks down the core mountebank capabilities step by step, starting in the next chapter by exploring canned responses in depth and adding HTTPS to the mix.

SUMMARY

- Mountebank translates the fields of the HTTP application protocol into JSON for requests and responses.
- Mountebank virtualizes services by creating *imposters*, which bind a protocol to a socket. You can create imposters using mountebank's RESTful API.
- You can use mountebank's API in automated tests to create imposters returning a specific set of canned data to allow you to test your application in isolation.

Part 2. Using mountebank

The test in chapter 2 was a behavioral test, but service virtualization can satisfy a wide spectrum of testing needs. Understanding how mountebank fits into that spectrum requires exploring the full capabilities of the tool.

The test we just looked at used basic building blocks of service virtualization—and indeed of any stubbing tool—the ability to evaluate the request to determine how to respond. We'll look at these capabilities over the course of the next two chapters, including additional context around HTTPS, managing configuration files, and taking advantage of mountebank's built-in XML and JSON parsing.

Chapters 5 and 6 demonstrate more advanced response generation, allowing a more interesting set of test scenarios. By adding record and replay capability, you can generate test data dynamically to perform large-scale tests and to build the foundation for performance testing (which we'll examine in part 3). The ability to programmatically change responses gives you key flexibility to support hard-to-test scenarios like OAuth.

Behaviors, or postprocessing steps, provide advanced functionality. From managing test data in a CSV file to adding latency to your responses, behaviors give you a robust set of tools both to simplify testing and to support a wider range of testing scenarios. We explore behaviors in chapter 7.

We round out this section by looking at mountebank's support for protocols, which is the glue that makes everything else possible. Although we spend much of the book exploring HTTP use cases, mountebank supports multiple protocols, and in chapter 8 we explore how it works with additional TCP-based protocols.

Chapter 3. 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 U.S. White House scandal of the 1990s, then-president Bill Clinton defended his prior statements by saying "It depends on what the meaning of *is* is." The grand jury and politicians ultimately failed to come to an agreement on the question, but, 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. Although an imposter, capturing the core idea of binding a protocol to a port, might beg to differ, by itself it adds little to a testing strategy. A response that looks like the real response—a response that, as far as the system under test is concerned, *is* the real response—changes everything. *Is* is the key to being fake. Without *is*, a service binding a protocol to a port is 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 is how you create *canned responses*, or responses that simulate a real response in some static way that you configure. Although it is one of three response types (`proxy` and `inject` being the other two), it's the most important one. In this chapter, we will explore `is` responses both by using the REST API and by persisting them in configuration files.

We will also start to layer in key security concerns. Although all of our examples so far have assumed HTTP, the reality is that any serious web-based service built today will use HTTPS, layering transport layer security (TLS) onto the HTTP protocol. Because security—especially authentication—is generally one of the first aspects of any

microservice implementation you run into when writing tests, we will look at using an HTTPS server that uses certificates to validate the client.

Finally, we will explore how to persist imposter configurations. As you have no doubt realized by now, stubbing out services over the wire can be significantly more verbose than stubbing out objects in-process. Figuring out how to lay out that configuration in a maintainable way is essential to using service virtualization to shift tests earlier in the development life cycle.

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] A "Hello, world!" response looks like the following listing in HTTP.

1

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

Listing 3.1. Hello world! in an HTTP response

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

Hello, world!
```

As you saw in the last chapter, returning this response in mountebank is as simple as translating the response you want into the appropriate JSON structure, as follows.

Listing 3.2. The HTTP response structure in JSON

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

To create an HTTP imposter, listening on port 3000, that will return this response, save the following code in a helloWorld.json file.

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

```
{
  "protocol": "http",
```

```
    "port": 3000,
    "stubs": [{
      "responses": [{
        "is": {                                          2
          "statusCode": 200,                             3
          "headers": { "Content-Type": "text/plain" },   3
          "body": "Hello, world!"                        3
        }
      }]
    }]
  }
```

- ***1* Protocol defines response structure**
- ***2* Tells mountebank to use an is response**
- ***3* Defines the canned response to be translated into HTTP**

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

2

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

```
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. You can verify that mountebank has created the imposter correctly by sending any HTTP request you want to port 3000:[3]

3

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

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

The response is almost, *but not quite*, the same as the "Hello, world!" response shown in listing 3.1:

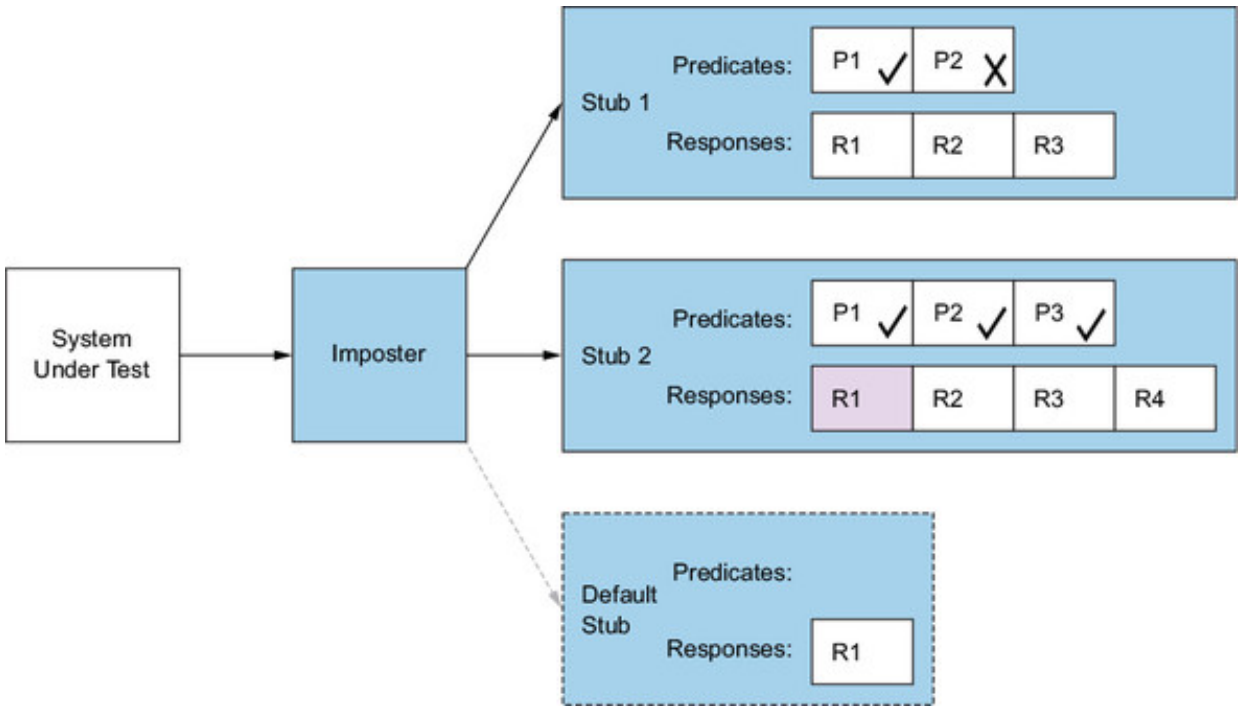
```
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: close
Date: Wed, 08 Feb 2017 01:42:38 GMT
Transfer-Encoding: chunked

Hello, world!
```

Three additional HTTP headers somehow crept in. Understanding where these headers came from requires us to revisit a concept described in chapter 1 as the *default response*.

3.1.1. The default response

You may recall the diagram shown in figure 3.1, 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, a hidden *default stub* will be used. That default stub contains no predicates, so it always matches the request, and it contains exactly one response—the default response. You can see this default response if you create an imposter without any stubs:

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

Note

Because you're using port 3000 across multiple examples, you may find that you have 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 http://local host:2525/imposters (to remove all existing imposters) or to http://localhost:2525/imposters/3000 (to remove only the imposter on port 3000). If you're using `curl`, the command would be `curl -X DELETE http://local host:2525/imposters`.

You have not defined *any* responses with that lame beast of an imposter; you have only said you want an HTTP server listening on port 3000. If you send any HTTP request to that port, you get the default response shown in the following listing.

Listing 3.4. The default response in mountebank

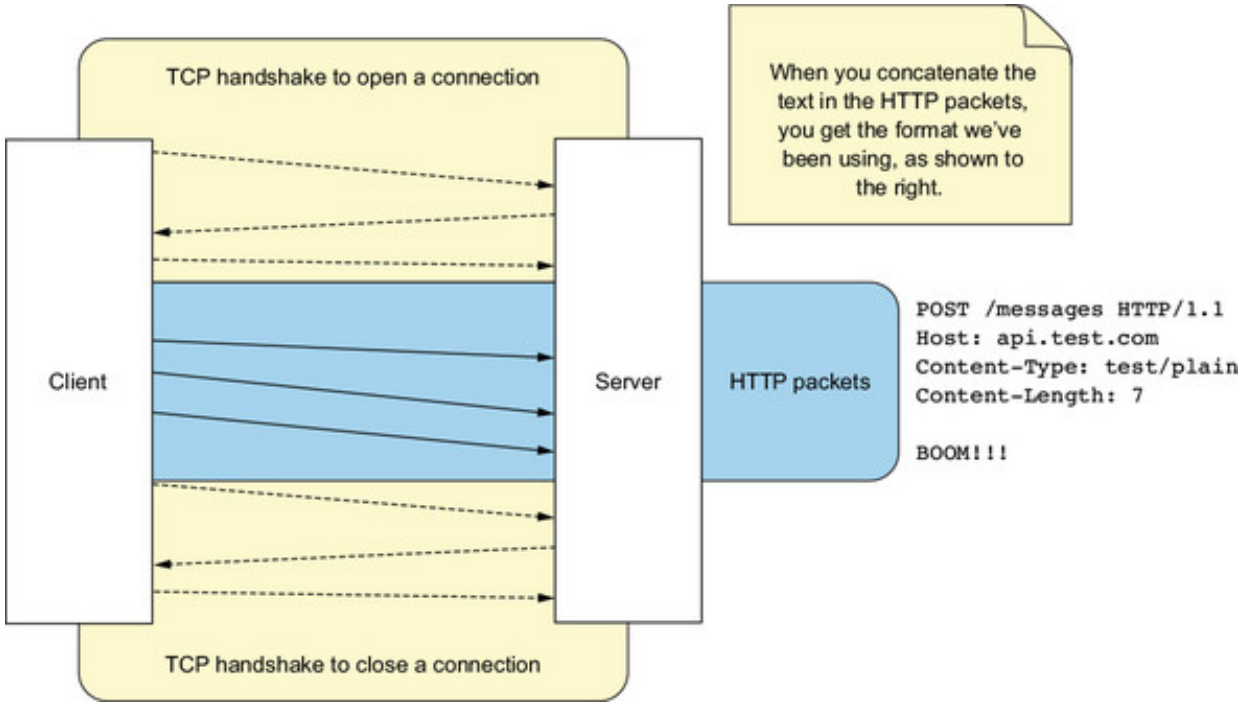
```
HTTP/1.1 200 OK
Connection: close
Date: Wed, 08 Feb 2017 02:04:17 GMT
Transfer-Encoding: chunked
```

We looked at the first line of the response in chapter 2, and the `200` status code indicates that mountebank processed the request successfully. The `Date` header is 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 is 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 is responsible for establishing the connection between the client and the server through a series of messages often referred to as the TCP handshake (figure 3.2).

Figure 3.2. TCP making the connection for HTTP messages



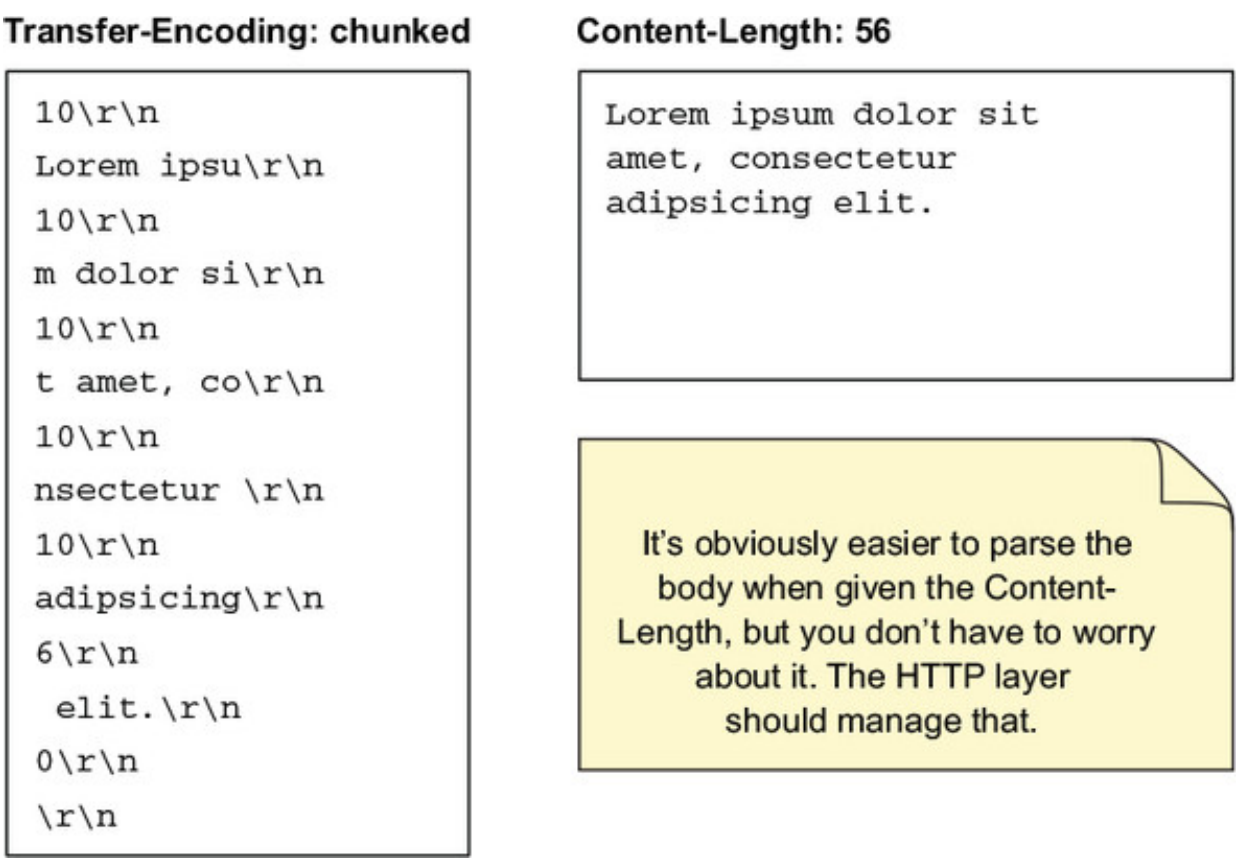
Although the TCP messages to establish the connection (represented by the dashed lines) are necessary, they aren't necessary for *every* request. Once the connection is established, the client and server can reuse it for multiple HTTP messages. That ability is important, particularly for websites that need to serve HTML, JavaScript, CSS, and a set of images, each of which requires a round trip between the client and the server.

HTTP supports *keep-alive 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 are 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 are writing performance tests, where the services you are virtualizing should be tuned with keep-alive connections, or if your purpose is to ensure your application behaves well with keep-alive connections, you should 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 is 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 is 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, so you need a different strategy. HTTP provides two strategies, as shown in figure 3.3.

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 one with the number of bytes it contains. Special formatting delineates each chunk, making parsing relatively easy. The advantage of sending the body a chunk at a time is that the server can start streaming data to the client before the server has all of the data. The alternative strategy is 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.

When I created mountebank, I had to choose one default strategy. In truth, the web framework mountebank is written in chose it for me, which is the only reason mountebank imposters default to chunked encoding. The two strategies are mutually exclusive, so 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 have seen what the default response looks like, it is probably a good time to admit that there is no such thing as a default stub in mountebank. That's a bald-faced 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

is exactly what mountebank does.

The reality is that mountebank merges the default response into *any* response you provide. Not providing a response is the same as providing an empty response, which is why you see the purest form of the default response in listing 3.4. But you also could 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 will still return 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

Mountebank's ability to merge in defaults for the response is a pleasant convenience. As I suggested, it means you only need to specify the fields that are different from the defaults, which simplifies the response configuration. But 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. You can default the `statusCode` to a `400 Bad Request` to avoid having to specify it in each response. Although you can't get rid of the `Date` header (it's required for valid responses), you'll go ahead and change the other default headers to use keep-alive connections and set the `Content-Length` header, as in the following listing.

Listing 3.5. Changing the default response

```
{
  "protocol": "http",
  "port": 3000,
```

```
    "defaultResponse": {                    1
      "statusCode": 400,                    2
      "headers": {                          3
        "Connection": "Keep-Alive",         4
        "Content-Length": 0                 5
      }
    },
    "stubs": [{
      "responses": [{
        "is": { "body": "BOOM!!!" }         6
      }]
    }]
  }
```

- *1* **Changes the built-in default response for this imposter only**
- *2* **Defaults to a Bad Request**
- *3* **Adds or changes default headers**
- *4* **Uses keep-alive connections**
- *5* **Sets the Content-Length header**
- *6* **The response details will be merged into the default response.**

If you now send a test request to the imposter, it merges the new default fields into the response, as follows.

Listing 3.6. A response using the new defaults

```
HTTP/1.1 400 Bad Request                1
Connection: Keep-Alive                  2
Content-Length: 7                       3
Date: Fri, 17 Feb 2017 16:29:00 GMT     4

BOOM!!!                                 5
```

- *1* **400 comes from your default status code.**
- *2* **You're now using keep-alive connections.**
- *3* **The Content-Length value was corrected from 0 to 7, and the Transfer-Encoding header is gone.**
- *4* **The Date header remains from the original default response.**
- *5* **The body from your is response is merged in.**

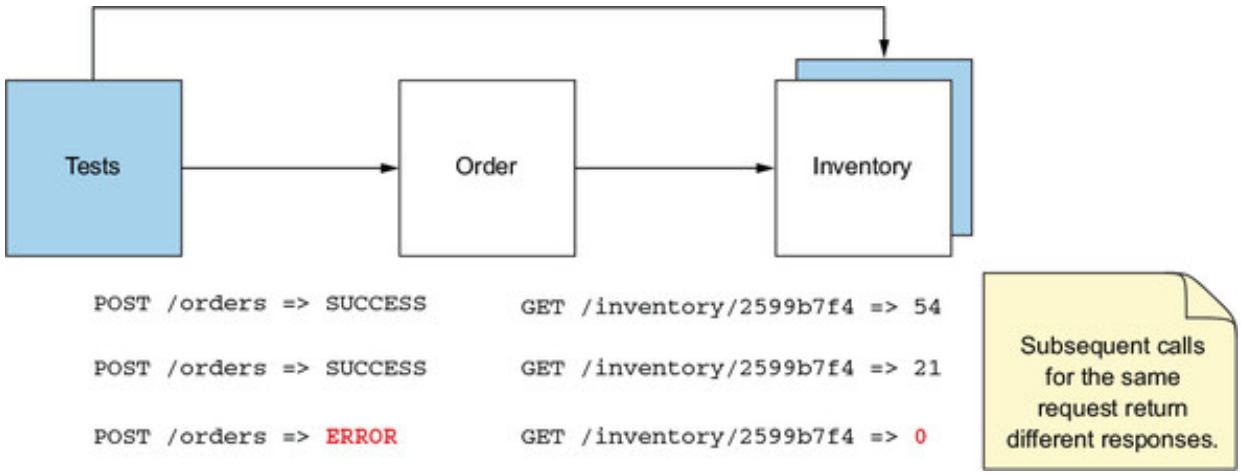
Notice in particular that mountebank set the `Content-Length` header to the correct

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, you will test what happens when you submit an order through an HTTP POST to an order service. Part of the order submission process involves checking to make sure inventory is sufficient. The tricky part, from a testing perspective, is that inventory is 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).

Figure 3.4. Inventory checks return volatile results for the same request.



In chapter 2, you saw a similar example with requests to the product catalog service, which returned different responses for the same path. In that example, you were able to use different predicates to determine which response to send based on the `page` query parameter, but in the inventory example, nothing about the request allows you to select one response over the other.

What you need is 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 is to use the fact that each stub contains a *list* of responses, as shown in the following listing. Mountebank returns those responses in the order provided.[4]

4

Note that in the following example and several others throughout the book, I will 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 clearly, allowing you to focus on the fact that *some* data is different for each response.

Listing 3.7. Returning a list of responses for the same stub

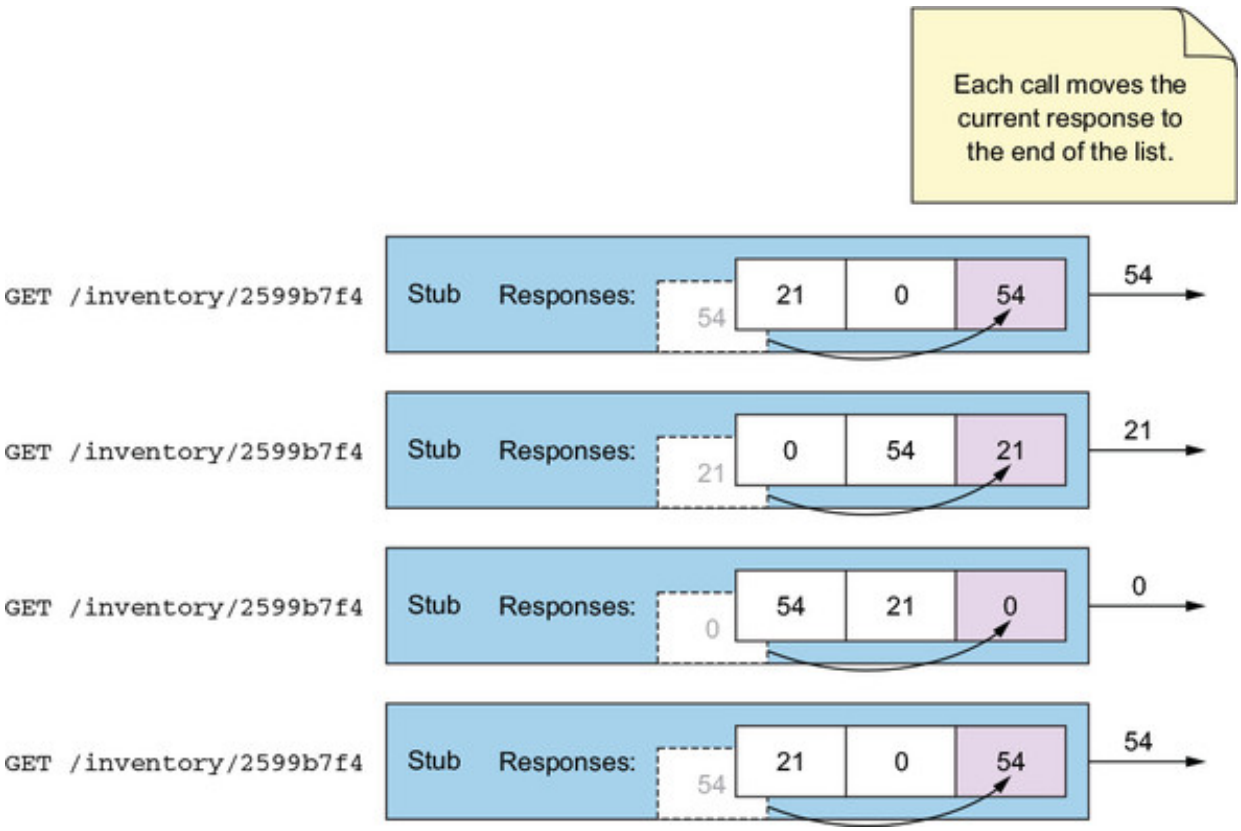
```
{
  "port": 3000,
  "protocol": "http",
  "stubs": [
    {
      "responses": [
        { "is": { "body": "54" } },      1
        { "is": { "body": "21" } },      1
        { "is": { "body": "0" } }        1
      ]
    }
  ]
}
```

- *1* **Responses that are returned in order**

The first call returns 54, the second call returns 21, and the third call returns 0. If your tests need to trigger a fourth call, it will once again return 54, then 21, and 0. Mountebank treats the list of responses as an infinite list, with the first and last entries connected like a circle, the *circular buffer* data structure discussed in chapter 1.

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

Figure 3.5. Each stub cycles through the responses forever.

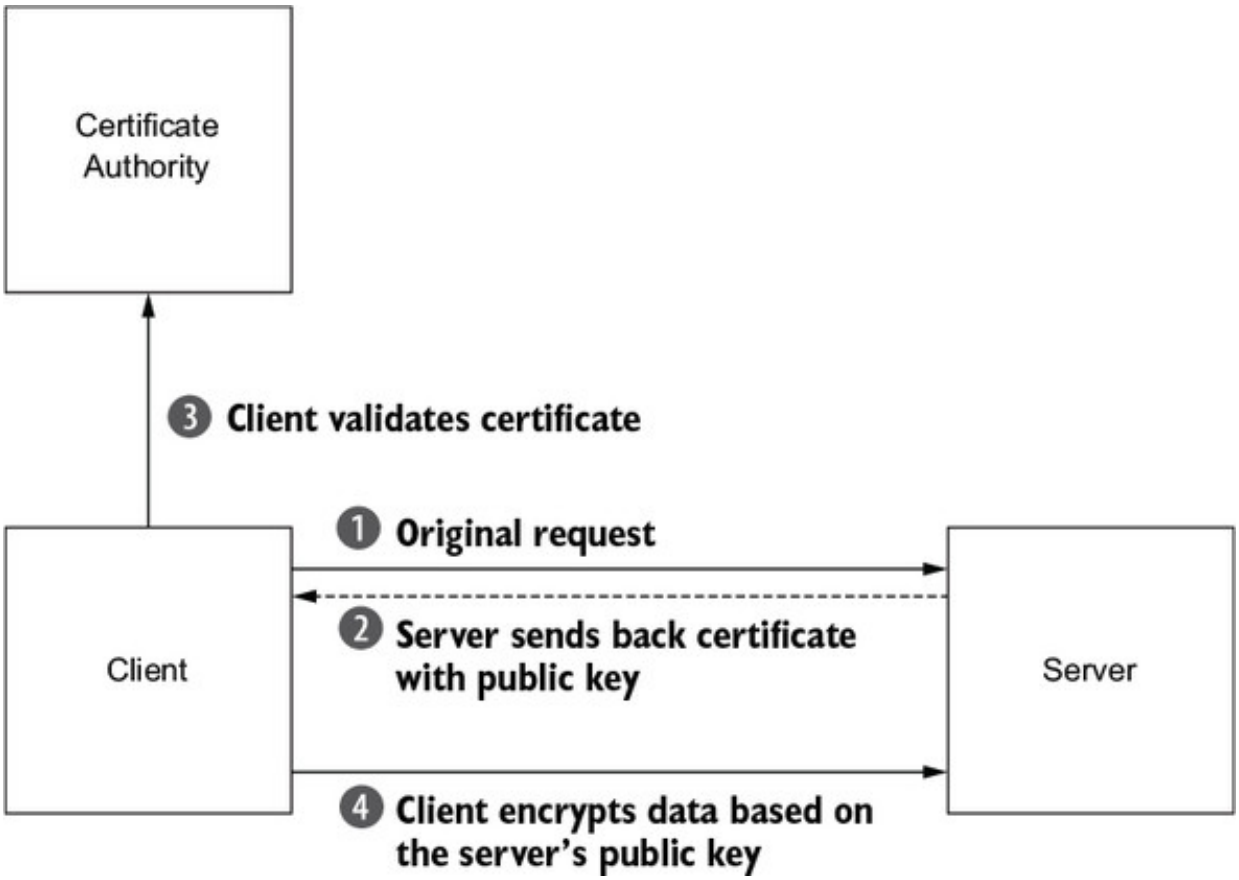


In chapter 7, we'll look at all kinds of interesting post-processing actions you can take on a response, but for now, you don't need to know anything more about canned responses. Let's switch gears and see how to layer in security.

3.2. HTTPS IMPOSTERS

To keep things simple, we've focused on HTTP services so far. The reality is 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. Figure 3.6 shows the basic structure for SSL.

Figure 3.6. The basic structure of SSL/TLS



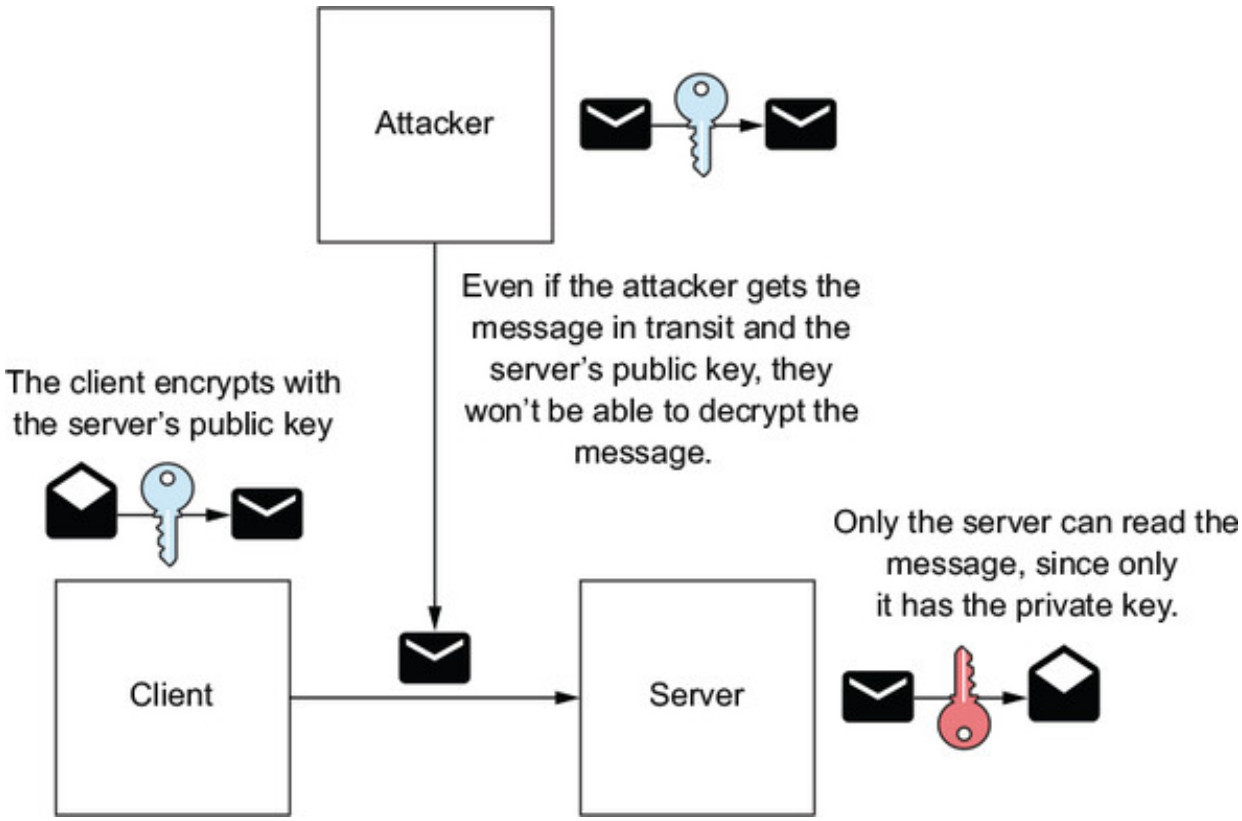
The infrastructure handles the SSL layer of HTTPS so that, as far as the application is concerned, each request and response is standard HTTP. The details of how the SSL layer works are a bit complex, but the key concepts that make it work are the server's *certificate* and the server's *keys*. The HTTPS server presents the client an SSL certificate during the handshake process, which describes the server's identity, including information such as 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 of you passing it confidential information that you would only intend to pass to Google. That's 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 that your organization trusts, you can confirm that the certificate is in fact from Google.

The certificate also includes the server's *public key*. The easiest approach to encryption is to use a single key for both encryption and decryption. Because of its efficiency, most of the communication relies on single-key encryption, but first the client and server have to agree on the key used without anyone else knowing it. The type of encryption SSL relies on during this handshake uses a neat trick that requires different keys for those two operations: the public key is used for encryption, and a separate *private key* is used for decryption (figure 3.7). This allows the server to share its public key and the client to use that key for encryption, knowing that only the server will be able to decrypt the resulting payload because only it has the private key.

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



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

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

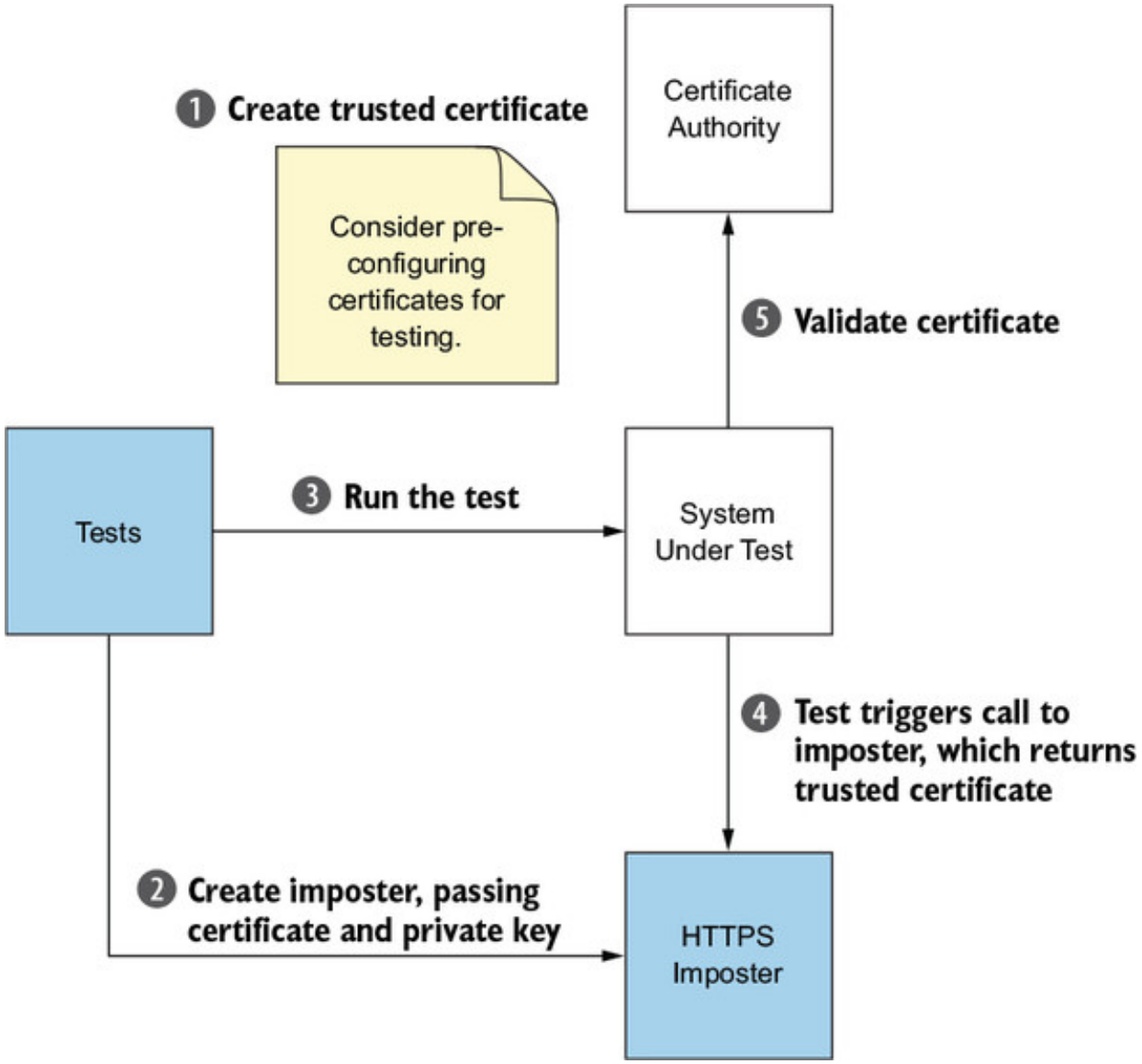
This is great for quickly setting up an HTTPS server, but it uses a default certificate (and key pair) that ships with mountebank. That certificate is both insecure and untrusted. Although that may be OK for some types of testing, any respectable service call should validate that the certificate is trusted, which, as shown in figure 3.6, involves a call to a trusted CA. By default, the HTTPS library that your system under test is using should reject mountebank's built-in certificate, meaning that it won't be able to connect to your virtual HTTPS service.

That leaves three options. The first is that you could configure the service you're testing not to validate that the certificate is 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 actually *test* what's going to production.

The second option is to avoid testing with HTTPS. Instead, create an HTTP imposter and configure your system under test to point to it. The networking libraries that your system under test uses should support that change without any code changes, and you usually can trust them to work with HTTPS. This is a reasonable option to use when you are testing on your local machine.

The third option, shown in figure 3.8, is to use a certificate that 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. That allows you to set up the 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, as shown in the following listing. You can pass them in what is known as PEM format; we will look at how to create them shortly.

Listing 3.8. Creating an HTTPS imposter

```
{
  "protocol": "https",
  "port": 3000,
  "key": "-----BEGIN RSA PRIVATE KEY-----\n...",   1
  "cert": "-----BEGIN CERTIFICATE-----\n..."       1
}
```

- ***1*** **Much abbreviated from the actual text**

This setup is still insecure. The test needs to know the private key to create the imposter, so the imposter will know how to decrypt communication from the system under test. The certificate is tied to the domain name in the URL, so as long as you segment that domain name to your test environment, you aren't risking leaking any

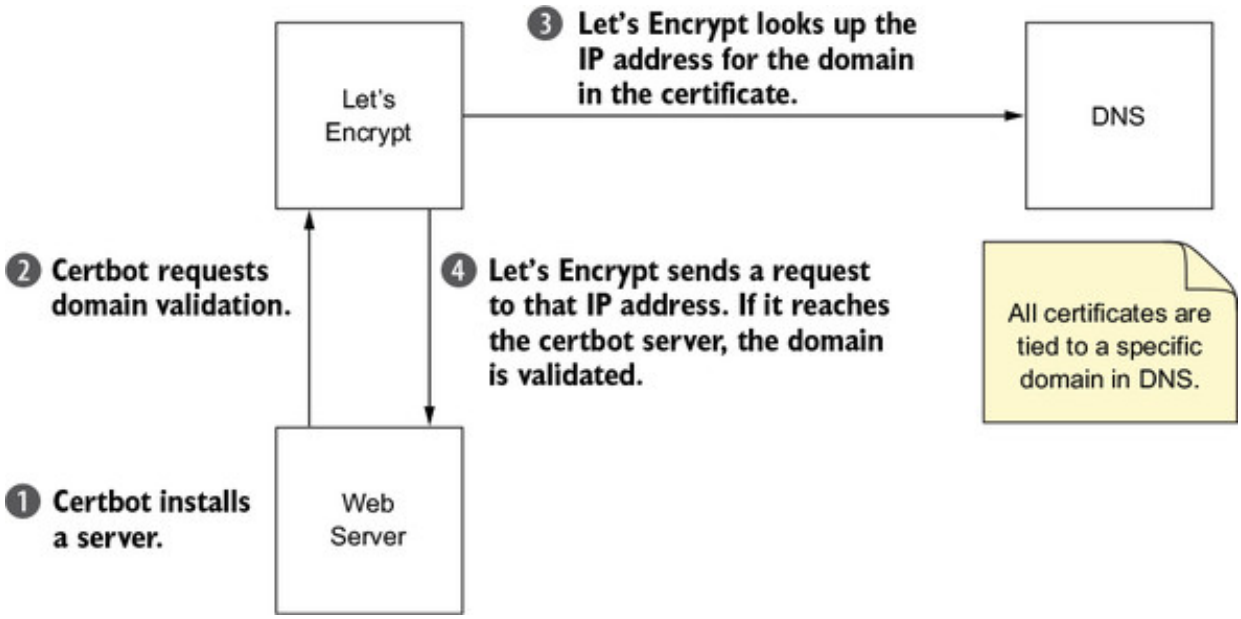
production secrets. With appropriate environment separation, this approach allows you to test the system under test without changing its behavior to allow untrusted certificates.

3.2.1. Setting up a trusted HTTPS imposter

Historically, getting certificates 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 is such a 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 (https://letsencrypt.org/) is a free option that supports creating certificates for domains with minimal fuss, backed by a public CA. Every CA will require validation from the domain owner to ensure that no one is 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).

Figure 3.9. How Let's Encrypt validates the domain



Let's Encrypt uses a command-line tool called Certbot (https://certbot.eff.org) 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 are 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 `certbot` command depends on the web server you are using, and because it is constantly evolving, you should check the documentation for details. In the general case, you might run:

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

That would create a certificate for the test.petstore.com domain that is served out of a web server running in /var/test/petstore. Simplifications are available if you're using a common web server like Apache or Nginx. See https://certbot.eff.org/docs/using.html#getting-certificates-and-choosing-plugins for details.

By default, the directory that certbot stores the SSL information in is /etc/lets encrypt/live/$domain, where $domain is the domain name of your service. If you look in that directory, you will find a few files, but two are 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 would 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 would include all the way up to the end of the file (from the example, "...\nWWrz8ocauVH\n-----END CERTIFICATE-----") for the certificate.

And...that's it. Everything else about your imposter remains the same. Once you have set the certificate and private key, the SSL layer is able to convert encrypted messages into HTTP requests and responses, which means the `is` responses you have already created continue to work. It may seem like a lot of work to set up the certificates, but that'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 is 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. You could, for example, create a `*.test.mypetstore.com` certificate, and that certificate would be valid for `products.test.mypetstore.com` as well as `inventory.test.mypetstore.com`. It wouldn't be valid for production domains that don't include `test` as part of their domain name.

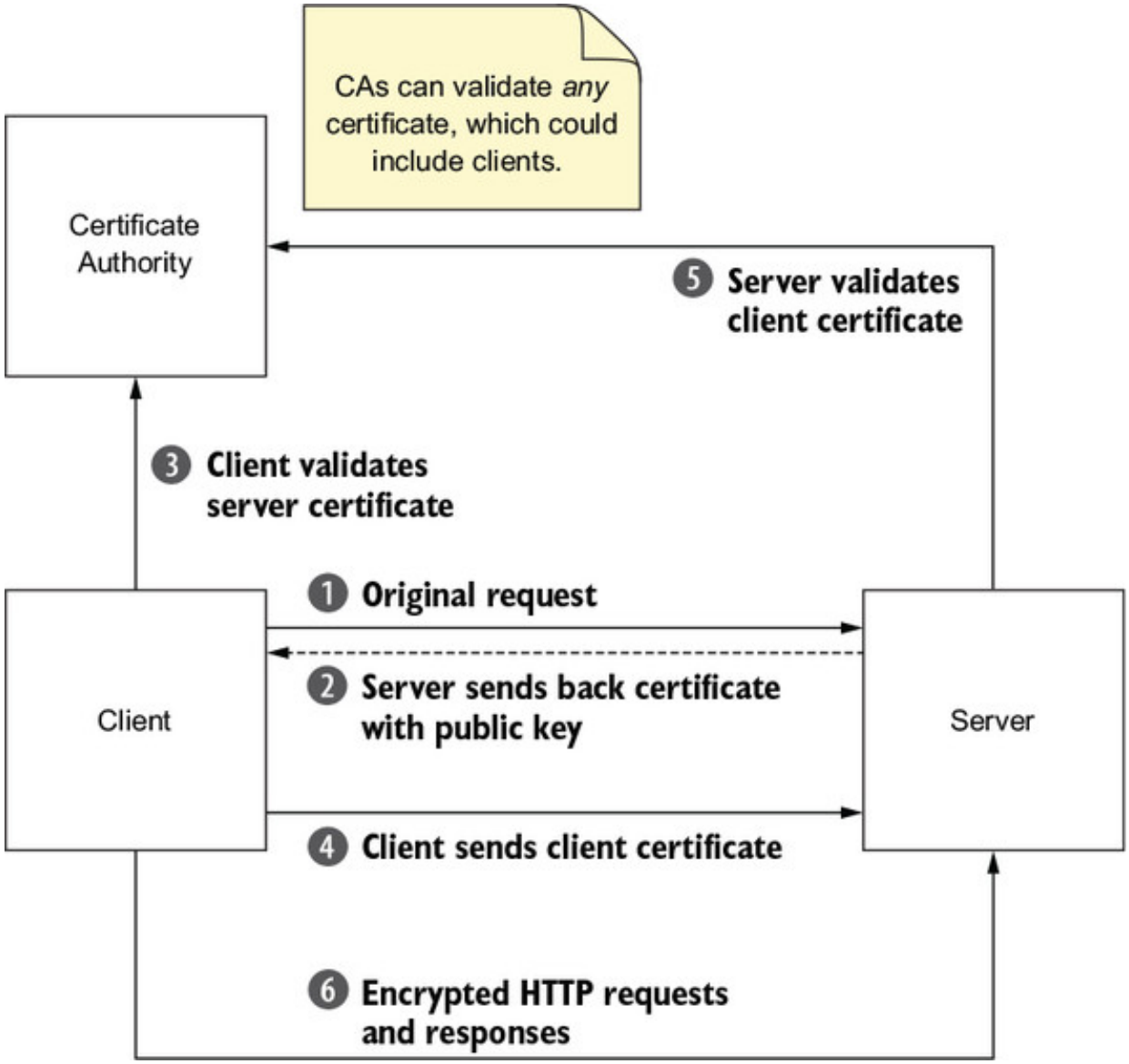
A wildcard certificate is ideal for testing scenarios. You may find it easy to manually add a wildcard certificate to the CA, tied exclusively to a testing subdomain, and reuse the certificate and private key for all imposters.

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 (figure 3.10). You don't see this when browsing the internet, because public websites have to assume the validity of 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 to validate clients as well as servers



If the service you are testing expects to validate its identity with your imposter using a client certificate, you need to be able to configure your imposter in a way that expects that certificate. This is as simple as adding a `mutualAuth` field to the configuration set to `true`, as shown in the following listing.

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                                          1
}
```

- ***1*** **The server will expect a client certificate.**

Now the server will challenge the client with a certificate request. Using certificates, both for HTTPS and for mutual authentication, allows you to virtualize servers in secure environments. But the fact that you have 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 multiline 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 you've added an inventory service and seen how to convert it to HTTPS, let's see how you would format the imposter configuration in files to make it easier to manage. The first bit of ugliness you'll want to solve is storing the certificate and private key in separate PEM files so you can avoid a long JSON string. If you store those as `cert.pem` and `key.pem` in the `ssl` directory, then you can create a file for the inventory imposter as `inventory.ejs` (figure 3.11).

Figure 3.11. The tree structure for the secure inventory imposter configuration

```
.
├── inventory.ejs
└── ssl
    ├── cert.pem
    └── key.pem

1 directory, 3 files
```

Mountebank uses a templating language called EJS (http://www.embeddedjs.com/) to interpret the config file, which uses a fairly standard set of templating primitives. The content between `<%-` and `%>`, as shown in the following listing, is dynamically evaluated and interpolated into the surrounding quotes. 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') %>",   1
    "key": "<%- stringify(filename, 'ssl/key.pem') %>",    1
    "stubs": [
      {
        "responses": [
          { "is": { "body": "54" } },
          { "is": { "body": "21" } },
          { "is": { "body": "0" } }
        ]
      }
    ]
  }
```

- ***1* Converts the multiline file content into a JSON string**

The inventory service will be available at startup, if you start mountebank with the appropriate command-line flag:

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

Mountebank adds the `stringify` function to the templating language, which does the equivalent of a JavaScript `JSON.stringify` call on the contents of the given file. In this case, the `stringify` call escapes the newlines. The benefit to you is that the configuration is much easier to read. (The `filename` variable is 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 that will be replaced by dynamic data and the `stringify` function to turn that data into presentable JSON—you can build robust templates. Storing the SSL information separately is useful, but I intentionally oversimplified the inventory imposter to focus on the behavior of the `responses` array. Let's add in the product catalog and marketing content services you saw in chapter 2.

3.3.1. Saving multiple imposters in the config file

As you saw, templating allows you to break up your configuration into multiple files. You'll take advantage of that to revisit the product catalog and marketing content imposter configurations you saw in chapter 2, putting each imposter in one or more files. The first thing you need to do is define the root configuration, which now needs to take a list of imposters. The tree structure will look like figure 3.12.

Figure 3.12. The tree structure for multiple services

```
.
├── content.ejs
├── imposters.ejs
├── inventory.ejs
├── product.ejs
└── ssl
    ├── cert.pem
    └── key.pem

1 directory, 6 files
```

Save the following listing as `imposters.ejs`.

Listing 3.11. The root configuration file, referencing other imposters

```
{
  "imposters": [
    <% include inventory.ejs %>,      1
    <% include product.ejs %>,        1
    <% include content.ejs %>         1
  ]
}
```

- ***1*** **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 brings the data as is from the referenced file. You can use the `include` EJS function and the `stringify` mountebank function to lay out your content any way you like. For complex configurations, you can store the response bodies—JSON, XML, or any other complex representation—in different files with newlines and load them in as needed. To keep it simple, you'll save each imposter in its own file, loading in the same wildcard certificate and private key. Save the product catalog imposter configuration that you saw in listing 2.1 in `product.ejs`, with some modifications as shown in the following listing.

Listing 3.12. The updated version of the product catalog imposter configuration

```
{
  "protocol": "https",                                    1
  "port": 3001,                                           2
  "cert": "<%- stringify(filename, 'ssl/cert.pem'); %>",
  "key": "<%- stringify(filename, 'ssl/key.pem'); %>",
  "stubs": [{                                             3
```

```
    "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" }
    }]
  }]
}
```

- ***1* Converts to HTTPS**
- ***2* Uses a different port to avoid a port conflict**
- ***3* Same stub configuration as in chapter 2**

Finally, save the marketing content imposter configuration you saw in listing 2.7 in a file called `content.ejs`, with the modifications shown in the following listing.

Listing 3.13. The updated version of the marketing content imposter configuration

```
{
  "protocol": "https",
  "port": 3002,                                                    1
  "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",
                "image": "/content/a0fad9fb"
              }
            ]
          }
        }
      }],
      "predicates": [{
        "equals": {
          "path": "/content",
          "query": { "ids": "2599b7f4,e1977c9e" }
        }
      }]
    }]
}
```

- ***1*** **Uses a different port**

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 v1.13.0 now taking orders -
➥ 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: [https:3002] Open for business...
```

All three imposters are up and running. Of interest is the log entry pointing out that an HTTP `PUT` command was sent the mountebank URL of http://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 is exposed via an API first, so anything you can do on the command line, you can implement using the API. If you had more advanced persistence requirements, you could construct the JSON and send it to mountebank using `curl`, as in the following listing.

Listing 3.14. Using curl to send the JSON to mountebank

```
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 all of the individual tests to send the `DELETE` calls to clean up their imposters.

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

SUMMARY

- 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, so once the last response is returned, mountebank cycles back to the first response.
- HTTPS imposters are possible, but you have to create the key pair and certificate. Let's Encrypt is 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.

Chapter 4. Using predicates to send different responses

This chapter covers

- Using predicates to send different responses for different requests
- Simplifying predicates on JSON request bodies
- Using XPath to simplify predicates on XML request bodies

During his younger years, Frank William Abagnale Jr. forged a pilot's license and traveled the world by deadheading.[1] Despite not being able to fly, his successful impersonation of a pilot meant that his food and lodging were fully paid for by the airline. When that well dried up, he impersonated a physician in New Orleans for nearly a year without any medical background, supervising resident interns. When he didn't know how to respond after a nurse said a baby had "gone blue," he realized the life and death implications of that false identity and decided to make yet another change. After forging a law transcript from Harvard University, he kept taking the Louisiana bar exam until he passed it, then he posed as an attorney with the attorney general's office. His story was memorialized in the 2002 movie, *Catch Me If You Can*.

1

The term refers to a pilot riding as a passenger on a flight to get to work. For example, a pilot who took up an assignment to fly from New York to London would need to first ride as a passenger to New York if he lived in Denver.

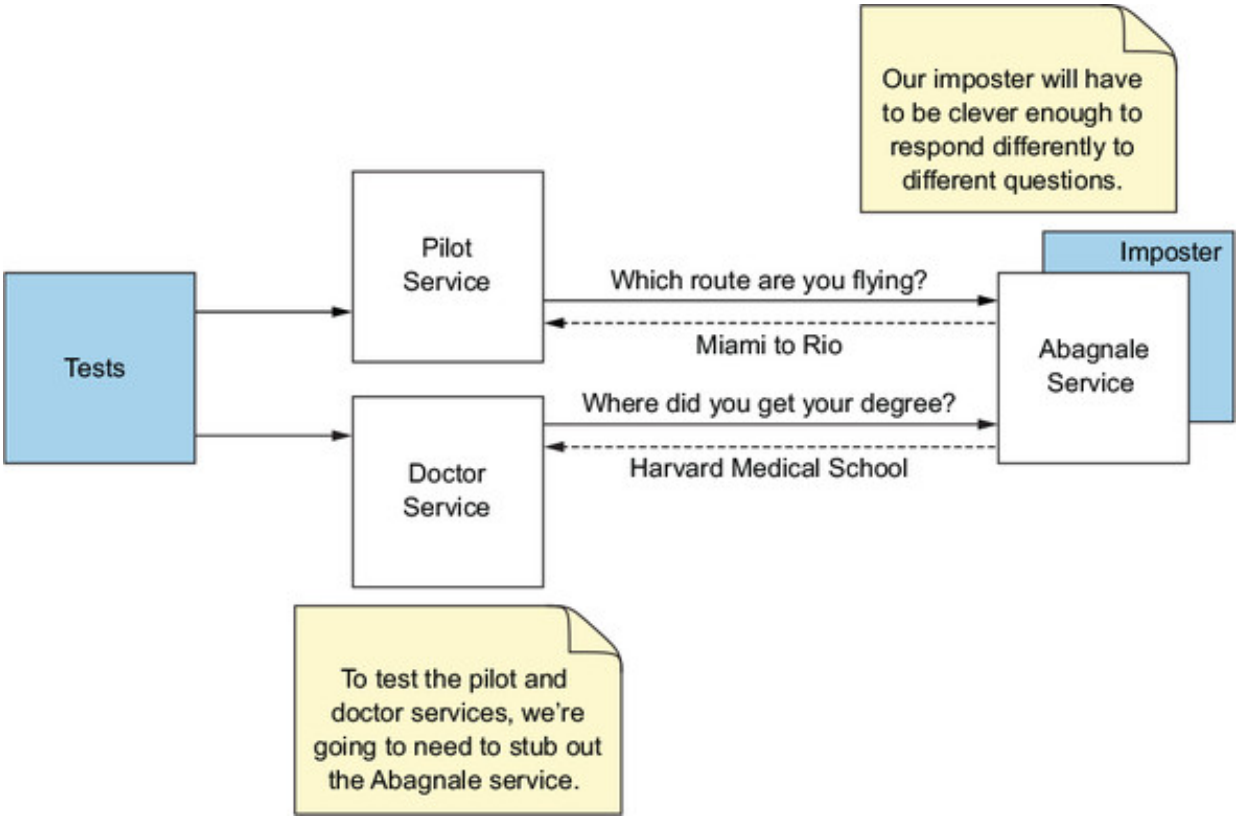
Frank Abagnale was one of the most successful imposters of all time.

Although mountebank cannot guarantee you Abagnale's indefatigable confidence, it does give you the ability to mimic one of the other key factors of his success: tailoring your response to your audience. Had Abagnale acted like a pilot when facing a room full of medical interns, he would have had a much shorter medical career. Mountebank uses *predicates* to determine which response to use based on the incoming request, giving your imposter the ability pretend to be a virtual pilot for one request and a virtual doctor for the next one.

4.1. THE BASICS OF PREDICATES

Testing a service that depends on a fictional Abagnale service (figure 4.1) is hard work. It involves doing something that has almost certainly never been done before in the history of mocking frameworks: you have to create a *virtual* imposter that pretends to be a *real* imposter.

Figure 4.1. The Abagnale service adapts its response to the questions you ask it.



Fortunately, mountebank makes this easy. If you assume your system under test embeds the question it asks the Abagnale service inside the HTTP body, then your imposter configuration can look something like the following listing.[2]

2 To keep the examples as simple as possible, we'll use HTTP.

Listing 4.1. Creating an Abagnale imposter

```
{
  "protocol": "http",
  "port": 3000,
  "stubs": [
    {
      "predicates": [{
        "contains": { "body":
        ➥ "Which route are you flying?" }          1
      }],
```

```
        "responses": [{
          "is": { "body": "Miami to Rio" }              2
        }]
      },
      {
        "predicates": [{
          "startsWith": { "body":
          ➥ "Where did you get your degree?" }         3
        }],
        "responses": [{
          "is": { "body": "Harvard Medical School" }    4
        }]
      },
      {
        "responses": [{
          "is": { "body":
          ➥ "I'll have to get back to you" }           5
        }]
      }
    ]
  }
```

- ***1*** **When asked a pilot question…**
- ***2*** **…respond like a pilot.**
- ***3*** **When asked a doctor question…**
- ***4*** **…respond like a doctor.**
- ***5*** **If you don't know what type of question it is, stall!**

Each predicate matches on a request field. The examples in listing 4.1 all match the `body`, but any of the other HTTP request fields you saw in the previous chapter are fair game: `method`, `path`, `query`, and `headers`.

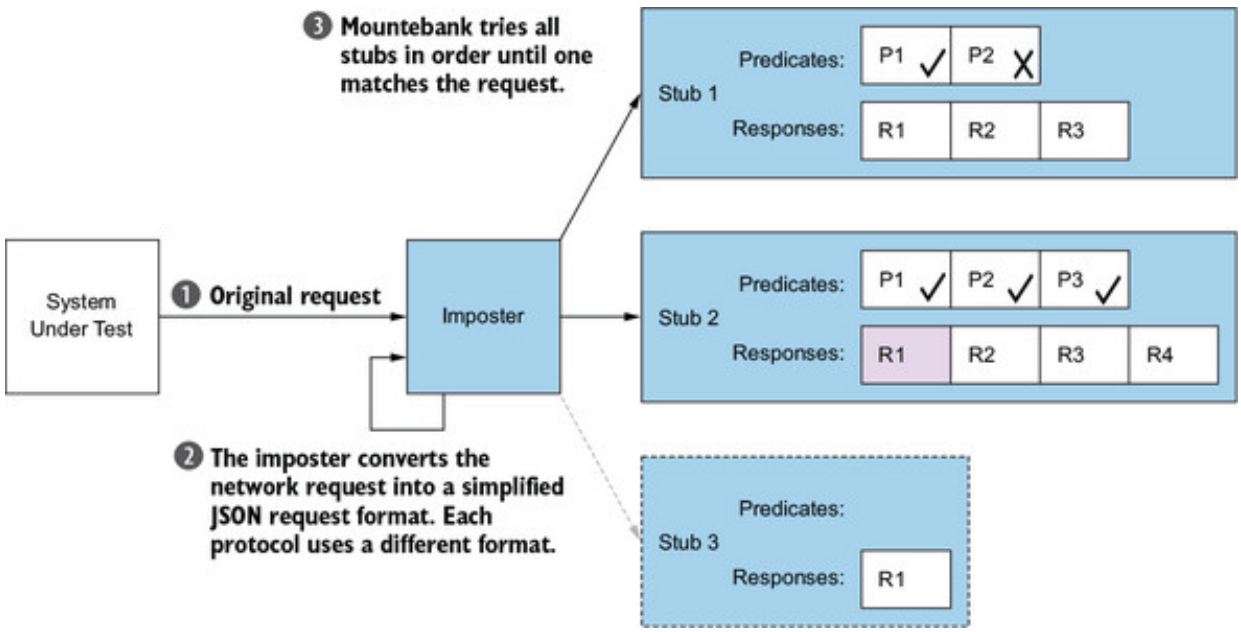
In chapter 2, we showed an example using `equals` predicates. Our simple Abagnale imposter shows off a couple more possibilities, using `contains` and `startsWith`. We'll look at the range of predicates shortly, but most of them are pretty self-explanatory. If the body contains the text "Which route are you flying?" then the imposter responds with "Miami to Rio," and if the body starts with the text "Where did you get your degree?" the imposter responds with "Harvard Medical School." This allows you to test the doctor and pilot services without depending on the full talents of Mr. Abagnale.

Note in particular the last stub, containing the "I'll have to get back to you" bit of misdirection. It contains no predicates, which means that all requests will match it,

including those that match the predicates in other stubs. Because a request can match multiple stubs, mountebank always picks the first match, based on array order. This allows you to represent a fallback default response by putting it at the end of the `stubs` array without any predicates.

We haven't paid too much attention to stubs as a standalone concept because they only make sense in the presence of predicates. As you saw in the last chapter, it's possible to send different responses to the exact same request, which is why the `responses` field is a JSON array. This simple fact, combined with the need to tailor the response to the request, is the *raison d'etre* of stubs. Each imposter contains a list of stubs. Each stub contains a circular buffer for the responses. Mountebank selects which stub to use based on the stub's predicates (figure 4.2).

Figure 4.2. Mountebank matches the request against each stub's predicates.

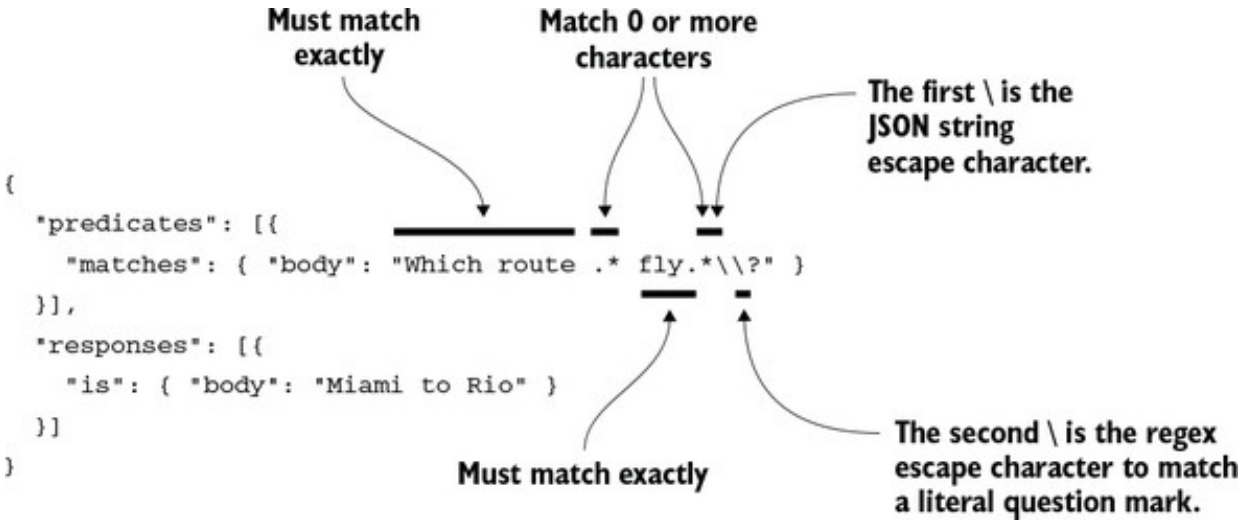


Because mountebank uses a "first-match" policy on the stubs, having multiple stubs that could respond to the same request isn't a problem.

4.1.1. Types of predicates

Let's take a closer look at the simplest predicate operators (figure 4.3). These all behave much as you would expect. But a few more interesting types of predicates are available, starting with the incredibly useful `matches` predicate.

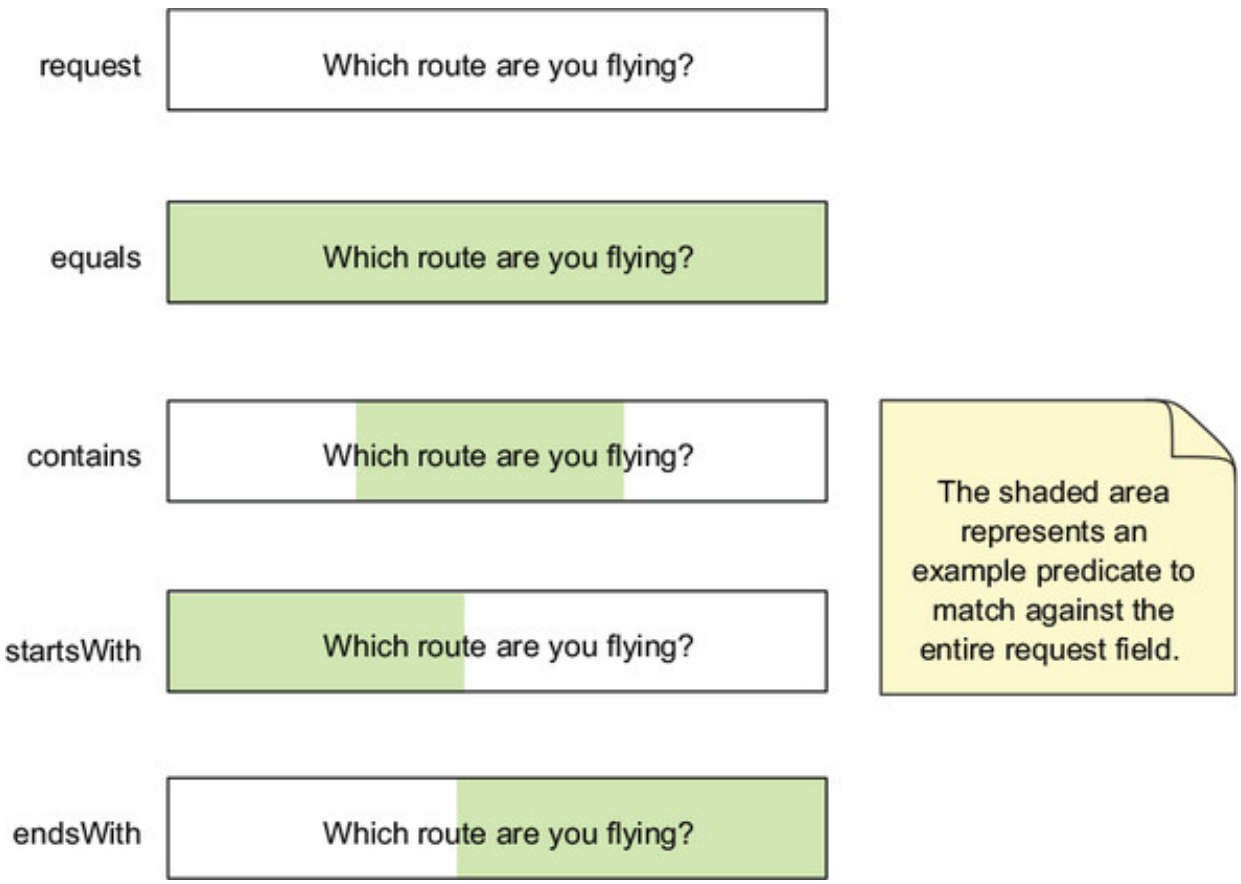
Figure 4.3. Mountebank matches the request against each stub's predicates.



The matches predicate

Your Abagnale service needs to respond intelligently to questions in both the present tense and the past tense. “Which route are you flying?” and “Which route did you fly?” should both trigger a response of “Miami to Rio.” You could write multiple predicates, but the `matches` predicate lets you simplify your configuration with the use of a *regular expression* (or *regex*, as used in figure 4.4).

Figure 4.4. How simple predicates match against the full request field

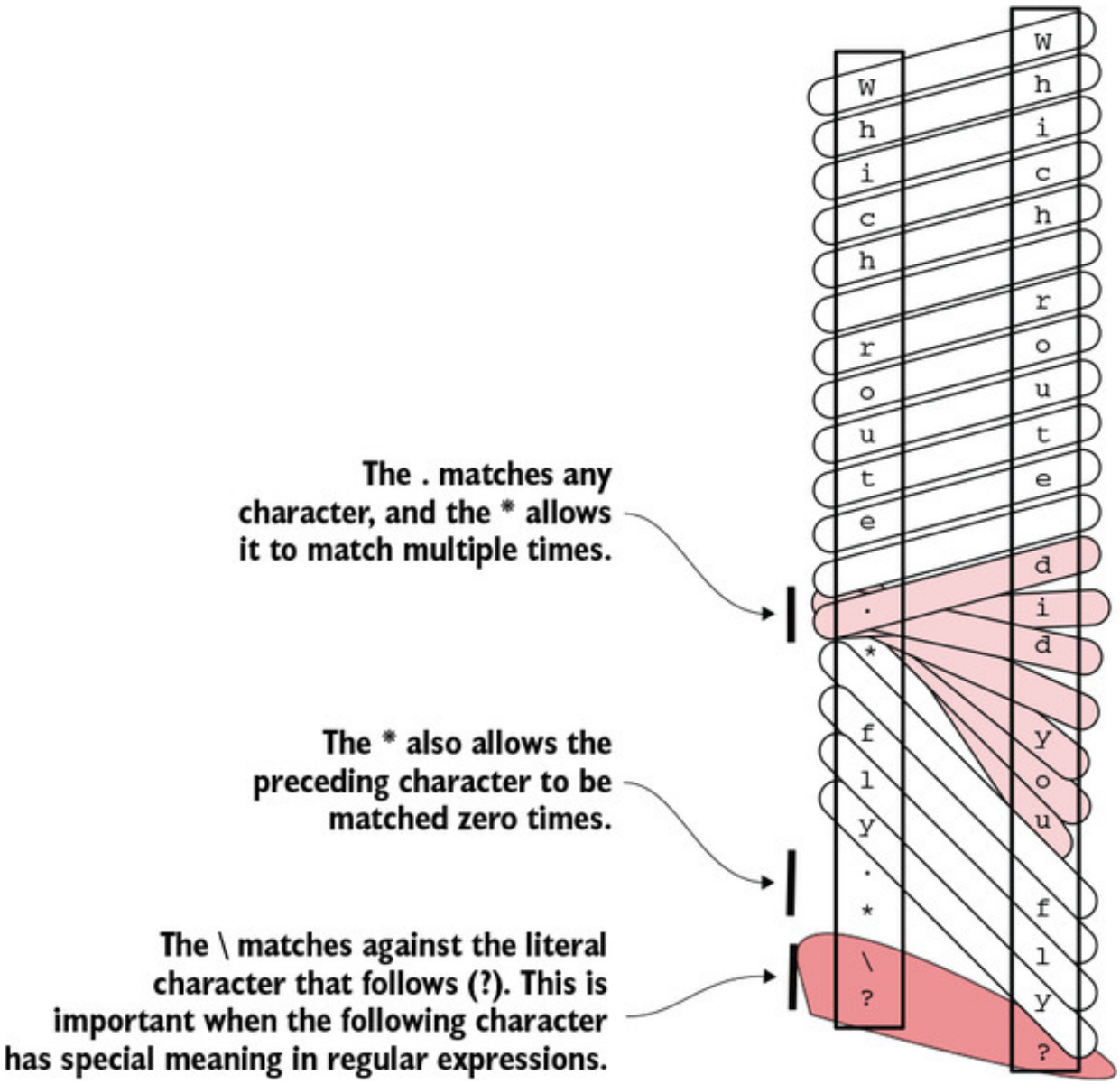


Regular expressions include a wonderfully rich set of metacharacters to simplify pattern matching. This example uses three:

- `.` — matches any character except a newline
- `*` — matches the previous character zero or more times
- `\` — escapes the following character, matching it literally

It may look like we had to double-escape the question mark, but that's only because `\` is a JSON string escape character as well as a regular expression escape character. The first `\` JSON-escapes the second `\`, which regex escapes the question mark, because it turns out that a question mark is also a special metacharacter. Like the asterisk, it sets an expectation of the previous character or expression, but unlike the `*` metacharacter, `?` matches it only zero times or one time. If you have never seen regular expressions before, that's a bit confusing, so let's break down your pattern against the request field value of "Which route did you fly?" (See figure 4.5.)

Figure 4.5. How a regular expression matches against a string value



Most characters match one-for-one against the request field. As soon as you reach the

first metacharacters (`.*`), the pattern matches *as much as it can* until the next literal character (a blank space). The wildcard pattern `.*` is a simple way of saying "I don't care what they entered." As long as the rest of the pattern matches, then the entire regular expression matches.

The second `.*` matches because the `*` allows a zero-character match, which is another way of saying it can be satisfied even if it doesn't match anything. Conveniently, it also would have matched the "ing" if you had entered "flying" instead of "fly," which is why regular expressions are so flexible. Finally, the pattern `\?` matches the ending question mark.

The `matches` predicate is one of the most versatile predicates in mountebank. With a couple of additional metacharacters, it can completely replace the other predicates we've looked at so far. We will demonstrate that with another example.

Replacing other predicates with regular expressions

A key to Abagnale's success is that he knew when it was time to run. After narrowly escaping arrest in New Orleans, he switched from being a pilot to a doctor with a new identity. When he realized the gravity of his medical ignorance, he switched again to being a lawyer. Recognizing when the jig is up requires seeing suspicious questions for what they are.

The Abagnale service responds with desperation when asked to see his driver's license, saying "Catch me if you can!" It gives the same response if the questioner asks for his "state's driver's license," or his "current driver's license," so you'll have to be a little loose with your matching.

You will need to match if the question *contains* the phrase "driver's license." You want to confirm that those characters represent a question by ensuring not only that they contain a question mark, but that they *end with* a question mark, and, to make absolutely sure that you are catching the right question, you will also ensure that the phrase *starts with* "Can I see your." Note that you can achieve all of this without regular expressions, but it would require combining multiple predicates, as follows:

```
{
  "predicates": [
    { "startsWith": { "body": "Can I see your" } },
    { "contains": { "body": "driver's license" } },
    { "endsWith": { "body": "?" } }
  ]
}
```

You can match exactly the same bodies with one `matches` predicate, as shown in the following listing.

Listing 4.2. Using the `matches` predicate to do the job of a set of `startsWith`, `contains`, and `endsWith` predicates

```
{
  "predicates": [{
    "matches": { "body": "^Can I see your.*driver's license.*\\?" }
  }],
  "responses": [{
    "is": { "body": "Catch me if you can!" }
  }]
}
```

You've already seen how the `.*` metacharacters match any characters, or no characters at all. Wrapping text with those metacharacters on either side is equivalent to using a `contains` predicate (figure 4.6).

Figure 4.6. Emulating the `contains` predicate with a regular expression

The `^` metacharacter matches only if the following character occurs at the beginning of the string, which allows you to recreate the `startsWith` predicate (figure 4.7).

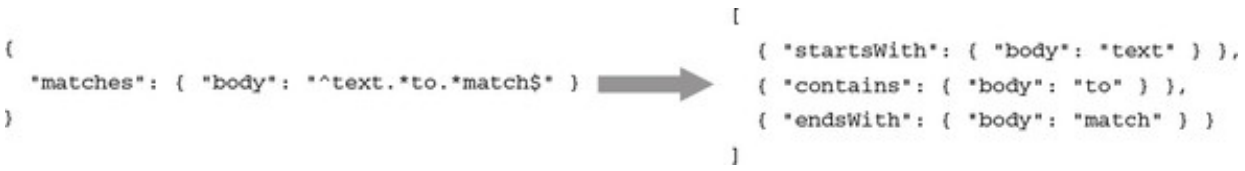
Figure 4.7. Emulating the `startsWith` predicate with a regular expression

Finally, the `$` metacharacter matches only if the preceding character occurs at the end of the string, which mimics the `endsWith` predicate (figure 4.8).

Figure 4.8. Emulating the `endsWith` predicate with a regular expression

The beauty of regular expressions is that you can combine all of those criteria into a single pattern (figure 4.9).

Figure 4.9. Emulating the `startsWith`, `contains`, and `endsWith` predicates with one regular expression



If you remove the `.*` metacharacter but leave in the `^` and `$` metacharacters that anchor a match at the beginning and end of a string of text, you create what equates to the `equals` predicate, although in this case the `equals` predicate is more readable (figure 4.10).

Figure 4.10. Emulating the `equals` predicate with a regular expression

Regular expression patterns can greatly simplify your use of predicates. Let's look at a more common use case of the `matches` predicate next.

Matching any identifier on the path

Although our predicates so far have focused on the http `body` field, predicates can work on any request field. A common pattern is to match the `path` field. Frank Abagnale took on a number of names, and in typical RESTful fashion, your Abagnale service allows you to query them by sending a `GET` request to `/identities`, or see the details about a single persona by looking at `/identities/{id}`, where `{id}` is the identifier for that particular identity. Let's start by matching the `/identities/{id}` path.

If you had a test scenario that involved hitting this endpoint, you could use the `matches` predicate to match any numeric identifier passed in, as shown in the following listing.

Listing 4.3. Using the `matches` predicate to match any identity resource

```
{
  "predicates": [{
    "matches": { "path": "/identities/\\d+" }
  }],
  "responses": [{
    "is": {
      "body": {
        "name": "Frank Williams",
        "career": "Doctor",
        "location": "Georgia"
      }
    }
  }]
```

```
    }
```

The metacharacters used here, `\d+`, represent one or more digits, so the pattern will match `/identities/123` and `/identities/2` but not `identities/frank-williams`. Several other useful metacharacters are available, including (but not limited to!) the ones listed in table 4.1.

Table 4.1. Regular expression metacharacters

Metacharacter	Description	Example
\	Unless it's part of a metacharacter like those described below, it escapes the next character, forcing a literal match.	4 * 2\? matches "What is 4 * 2?"
^	Matches the beginning of the string	^Hello matches "Hello, World!" but not "Goodbye. Hello."
$	Matches the end of the string	World!$ matches "Hello, World!" but not "World! Hello."
.	Matches any non-newline character	..... matches "Hello" but not "Hi"
*	Matches the previous character 0 or more times	a*b matches "b" and "ab" and "aaaaaab"
?	Matches the previous character 0 or 1 times	a?b matches "b" and "ab" but not "aab"
+	Matches the previous character 1 or more times	a+b matches "ab" and "aaaab" but not "b"
\d	Matches a digit	\d\d\d matches "123" but not "12a"
\D	Inverts \d, matching nondigit characters	\D\D\D matches "ab!" but not "123"
\w	Matches an alphanumeric "word" character	\w\w\w matches "123" and "abc" but not "ab!"
\W	Inverts \w, matching nonalphanumeric symbols	\W\W\W matches "!?." but not "ab."
\s	Matches a whitespace character (mainly spaces, tabs, and newlines)	Hello\sworld matches "Hello world" and "Hello world"
\S	Inverts \s, matching any non-space character	Hello\Sworld matches "Hello-world" and "Hello----world"

Regular expressions allow you to define robust patterns to match characters and are a

rich subject in their own right. Several excellent books are available on the subject, and a number of internet sites provide tutorials. If you are looking for a quick start, I recommend the tutorials on http://www.regular-expressions.info/. We look at more examples in chapter 7.

4.1.2. Matching object request fields

Google supports full-text searching using the `q` querystring parameter, so, for example, https://www.google.com/?q=mountebank will show web pages that are somehow relevant to the search text "mountebank." Other web services, like the Twitter API, have adopted the `q` parameter as a search option even when searching more JSON-structured data like your Abagnale service. Having a single search parameter allows a Google-like user experience with a single text box, where the user doesn't have to specify the fields they are matching. They don't even have to match a field completely. Implementing a full-text search can be a little tricky, but you don't need to worry about that; you need to pretend to be a service that implements full-text searching.

The `/identities` path for your Abagnale service supports searching using the `q` querystring parameter. For example, `/identities?q=Frank` will search for all of Abagnale's identities that are somehow relevant to "Frank," which you can use as a shortcut to find those identities where he used his real first name. The predicate for querystring parameters looks a little different, but only because the querystring is an object field instead of a string field, as shown in the following listing.

Listing 4.4. Adding a predicate for a query parameter

```
{
  "predicates": [{
    "equals": {
      "query": { "q": "Frank" }              1
    }
  }],
  "responses": [{
    "is": {
      "body": {
        "identities": [                      2
          {
            "name": "Frank Williams",
            "career": "Doctor",
            "location": "Georgia"
          },
          {
            "name": "Frank Adams",
            "career": "Brigham Young Teacher",
            "location": "Utah"
          }
```

```
          ]
        }
      }
    }]
  }
```

- ***1*** **Because query is an object field, the predicate value is also an object.**
- ***2*** **Returns an array**

For HTTP requests, both `query` and `headers` are object fields. To get to the right query parameter (or header), you have to add an extra level to your predicate.

4.1.3. The deepequals predicate

In some situations, you want to match only if *no* query parameters were passed; for example, sending a GET to `/identities` without any searching or paging parameters should return all identities. None of the predicates shown so far supports this scenario, as they all work on a single request field. For more complex key-value pair structures, like HTTP queries and headers, the other predicates expect you to navigate down to the primitive field inside, like the `q` parameter within the querystring we just looked at.

The `deepEquals` predicate matches an entire object structure, allowing you to specify an empty querystring:

```
{
  "deepEquals": { "query": {} }
}
```

Shortly you will see how it's possible to combine multiple predicates, which allows you to require two query parameters. But the `deepEquals` predicate is the only way to guarantee that those two query parameters *and nothing else* are passed:

```
{
  "deepEquals": {
    "query": {
      "q": "Frank",
      "page": 1
    }
  }
}
```

With this predicate, a querystring of `?q=Frank&page=1` would match, but a querystring of `?q=Frank&page=1&sort=desc` wouldn't.

4.1.4. Matching multivalued fields

Another interesting characteristic of HTTP query parameters and headers is that you can pass the same key multiple times. Your Abagnale service supports multiple `q` parameters and returns only those matches that satisfy all of the provided queries. For example, `GET /identities?q=Frank&q=Georgia` would return only Frank Williams, because Frank Adams worked in Utah.

```
{
  "identities": [{
    "name": "Frank Williams",
    "career": "Doctor",
    "location": "Georgia"
  }]
}
```

All of the predicates we looked at so far support multivalued fields, but, once again, `deepEquals` is significantly different from the others. If you use an `equals` predicate, the predicate will pass if *any* of the values equals the predicate value:

```
{
  "equals": {
    "query": {
      "q": "Frank"
    }
  }
}
```

The `deepEquals` predicate requires *all* of the values to match. Mountebank represents such multivalue fields as arrays in the request.[3] This particular request would look something like this in mountebank:

3

You also can use this trick when creating responses with multivalue fields. This is most commonly seen with the `Set-Cookie` response header.

```
{
  "method": "GET",
  "path": "/identities",
  "query": {
    "q": ["Frank", "Georgia"]
  }
}
```

The trick is to pass the array as the predicate value:

```
{
  "deepEquals": {
    "query": {
      "q": ["Georgia", "Frank"]
    }
  }
}
```

Note that the order of the values doesn't matter. You can use the array syntax for any predicate, not just `deepEquals`, but `deepEquals` is the only one that requires an exact match. The example in the following listing demonstrates the difference.

Listing 4.5. Using predicate arrays

```
{
  "stubs": [
    {
      "predicates": [{
        "deepEquals": {                                  1
          "query": { "q": ["Frank", "Georgia"] }         1
         }                                               1
      }],
      "responses": [{
        "is": { "body": "deepEquals matched" }
      }]
    },
    {
      "predicates": [{
        "equals": {                                      2
          "query": { "q": ["Frank", "Georgia"] }         2
        }                                                2
      }],
      "responses": [{
        "is": { "body": "equals matched" }
      }]
    }
  ]
}
```

- ***1* Requires exact match**
- ***2* Requires these elements to be present**

If you send a request to /identities?q=Georgia&q=Frank, the response body will show that the `deepEquals` predicate matched because all of the array elements matched *and* no additional array elements were present in the request. But if you send a request

to /identities?q=Georgia&q=Frank&q=Doctor, the `deepEquals` predicate will no longer match because the predicate definition isn't expecting "Doctor" as an array element. The `equals` predicate *will* match, because it allows additional elements in the request array that aren't specified in the predicate definition.

4.1.5. The exists predicate

There's one more primitive predicate to look at. The `exists` predicate tests for either the existence or nonexistence of a request field. If you have a test that depends on the `q` parameter to be passed and the `page` parameter *not* being passed, then `exists` is what you want:

```
{
  "exists": {
    "query": {
      "q": true,
      "page": false
    }
  }
}
```

The `exists` predicate also comes in quite handy when you want to check for the presence of a header. For example, you may decide that for testing purposes, you want to verify that the service handles an HTTP challenge correctly (represented by a 401 status code) when the `Authorization` request header is missing, without worrying about whether the credentials stored in the `Authorization` header are correct, as shown here:

```
{
  "predicates": [{
    "exists": {
      "headers": { "Authorization": false }
    }
  }],
  "responses": [{
    "is": { "statusCode": 401 }
  }]
}
```

The `headers` field in this snippet specifies the condition that there's no Authorization header, and the `is` response returns a 401 status code.

The `exists` predicate works on string fields like the `body`, which is considered not to

exist if it's an empty string. It's usually more useful in conjunction with the JSON or XML support described later in this chapter.

4.1.6. Conjunction junction

The `predicates` field is an array. Every predicate in the array must match for mountebank to use that stub. You can reduce the array to a single element by using the `and` predicate, so the following two sets of predicates will match the exact same requests (for example, one with a body of "Frank Abagnale"):

```
{
  "predicates": [
    { "startsWith": { "body": "Frank" } },
    { "endsWith": { "body": "Abagnale" } }
  ]
}
```

and

```
{
  "predicates": [{
    "and": [
      { "startsWith": { "body": "Frank" } },
      { "endsWith": { "body": "Abagnale" } }
    ]
  }]
}
```

By itself, the `and` predicate isn't very useful. But combined with its conjunction cousin, the `or` predicate, and its distant disjunction relative, the `not` predicate, you can create predicates of dizzying complexity in a festival of Booleanism. For example, the predicate in listing 4.6 matches requests that return Frank Williams, regardless of whether the system under test directly calls that persona URL (assumed to be `/identities /123`) or they search for him at `/identities?q=Frank+Williams`, but only if no `page` query parameter is added.

Listing 4.6. Combining multiple predicates using `and`, `or`, and `not`

```
{
  "or": [
    { "equals": { "path": "/identities/123" } },          1
    {
      "and": [
        { "equals": { "path": "/identities" } },          2
```

```
          {
            "and": [
              {
                "contains": { "query": { "q": "Frank" } }      3
              },
              {
                "contains": { "query": { "q": "Williams" } }   4
              },
              {
                "not": {                                       5
                  "exists": { "query": { "page": true } }      5
                }                                              5
              }
            ]
          }
        ]
      }
    ]
  }
```

- ***1*** **Matches if the request goes directly to the URL…**
- ***2*** **…or if it searches**
- ***3*** **…with a query containing Frank**
- ***4*** **…and a query containing Williams**
- ***5*** **…with no paging (you can get rid of the not predicate by changing the page value to false).**

Sometimes it's necessary to create complex conditions, and the rich set of predicates mountebank supports enables you to specify such conditions. But to make your configuration readable and maintainable, it's good practice to make the predicates as simple as possible for your use case. The conjunctions are there when you need them, but you'll probably be happier if you can avoid using them too much.

4.1.7. A complete list of predicate types

There's one other predicate you haven't seen yet—the `inject` predicate—but you'll have to wait until chapter 6 to take a look at it. Before we proceed, let's review the predicates you have at your disposal. For your reference, table 4.2 provides the complete list of predicate operators that mountebank supports.

Table 4.2. All predicates that mountebank supports

Operator	Description

equals	Requires the request field to equal the predicate value
deepEquals	Performs nested set equality on object request fields
contains	Requires the request field to contain the predicate value
startsWith	Requires the request field to start with the predicate value
endsWith	Requires the request field to end with the predicate value
Matches	Requires the request field to match the regular expression provided as the predicate value
exists	Requires the request field to exist as a nonempty value (if true) or not (if false)
not	Inverts the subpredicate
or	Requires any of the subpredicates to be satisfied
and	Requires all of the subpredicates to be satisfied
inject	Requires a user-provided function to return true (see chapter 6)

4.2. PARAMETERIZING PREDICATES

Each predicate consists of an operator and zero or more parameters that alter the behavior of the predicate in certain ways. Two parameters, `xpath` and `jsonpath`, change the scope of the predicate to a value embedded in the HTTP body; we look at those shortly. The other parameter affects the way the predicate evaluates the request field.

4.2.1. Making case-sensitive predicates

All predicates are case-insensitive by default. The following predicate, for example, will be satisfied regardless of whether the `q` parameter is “Frank” or “frank” or “FRANK”:

```
{
  "equals": {
    "query": { "q": "frank" }
  }
}
```

This is even true for the `matches` predicate. Regular expressions are case-sensitive by default, but mountebank changes their default to match the behavior of the other predicates. If you need case sensitivity, you can set the `caseSensitive` parameter to `true`, as follows.

Listing 4.7. Using a case-sensitive predicate

```
{
  "equals": {
    "query": { "q": "Frank" }
  },
  "caseSensitive": true
}
```

"FRANK" and "frank" will no longer satisfy the predicate.

Mountebank's default behavior also treats the keys in a case-insensitive manner, so that without the `caseSensitive` parameter, the predicate above also would match a querystring of `?Q=FRANK`. This is often appropriate, especially for HTTP headers where the case of the headers shouldn't matter. Adding the `caseSensitive` parameter forces case sensitivity on both the keys and the values.

4.3. USING PREDICATES ON JSON VALUES

JSON is the *lingua franca* of most RESTful APIs these days. As you have seen previously, it's possible to create mountebank responses that use JSON objects instead of strings for the `body` field. Mountebank also provides ample support for creating predicates against JSON bodies.

4.3.1. Using direct JSON predicates

Despite Frank Abagnale's cleverness, there's nothing magical about what he does. When he needs to add a new identity, for example, he `POST`s the JSON representation of the identity to the `/identities` URL, like the rest of us.

Because the HTTP body is a string as far as mountebank is concerned, you can use a `contains` predicate to capture a particular JSON field. But doing so is inconvenient, as the white space has to match between the key and the value. The `matches` predicate gives you a lot more flexibility at the cost of readability. Fortunately for you, mountebank's willingness to treat HTTP bodies as JSON as well as strings allows you to navigate the JSON object structure like you have navigated the `query` object structure previously, as shown in the following listing.

Listing 4.8. Using a direct JSON predicate on an HTTP body

```
{
  "predicates": [{
    "equals": {
      "body": {                          1
        "career": "Doctor"               1
```

```
        }                                    1
      }
    }],
    "responses": [{
      "is": {
        "statusCode": 201,
        "headers": { "Location": "/identities/123" },
        "body": "Welcome, Frank Williams"
      }
    }]
  }
```

- ***1*** **The "career" field at the root of the JSON body has to equal "Doctor."**

You can navigate as many levels deep in the JSON object as you need to. Mountebank treats arrays no differently from how it treats the multivalued fields described in section 4.1.4, which used the example of a repeating key on the querystring. For complex queries, you are probably better off using JSONPath.

4.3.2. Selecting a JSON value with JSONPath

You can initialize the set of identities that the Abagnale service provides by sending a `PUT` command to the `/identities` path, passing in an array of identities, as shown here:

```
{
  "identities": [
    {
      "name": "Frank Williams",
      "career": "Doctor",
      "location": "Georgia"
    },
    {
      "name": "Frank Adams",
      "career": "Teacher",
      "location": "Utah"
    }
  ]
}
```

This test scenario requires you to send a 400 if the `PUT` command includes a career of "Teacher" as the last member of the array and a 200 otherwise. That is obviously a bit of a stretch, but it enables you to show off the power of JSONPath. JSONPath is a query language that simplifies the task of selecting values from a JSON document and excels with large and complex documents. Stefan Goessner came up with the idea and documented its syntax at http://goessner.net/articles/JsonPath/.

Let's look at the entire imposter configuration in the following listing.

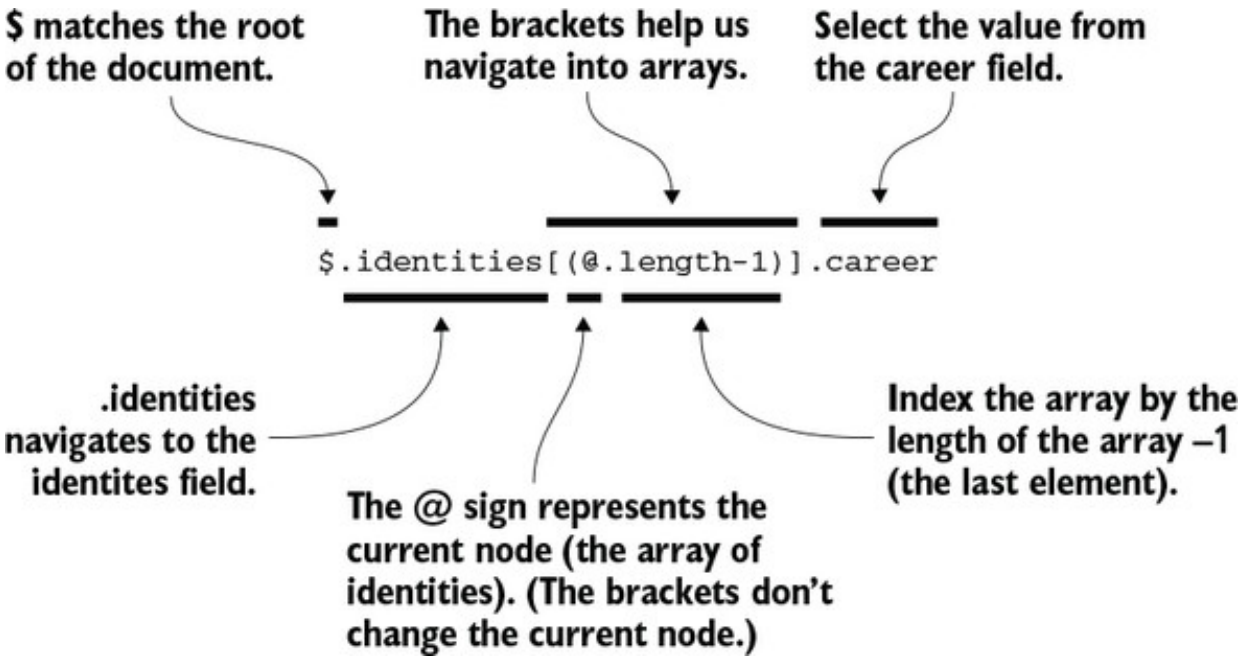
Listing 4.9. Using JSONPath to match only the last element of an array

```
{
  "protocol": "http",
  "port": 3000,
  "stubs": [{
    "predicates": [
      { "equals": { "method": "PUT" } },                    1
      { "equals": { "path": "/identities" } },              1
      {
        "jsonpath": {                                       2
          "selector": "$.identities[(@.length-1)].career"   2
        },                                                  2
        "equals": { "body": "Teacher" }                     3
      }
    ],
    "responses": [{ "is": { "statusCode": 400 } }]          4
  }]                                                        5
}
```

- ***1* Only matches a PUT request to /identities**
- ***2* Limits the scope of the predicate to the JSONPath query**
- ***3* The JSON value selected within the body must equal "Teacher."**
- ***4* Returns a 400 status code**
- ***5* Returns the built-in default 200 response if the predicate doesn't match**

You set the predicate operator to have the `body` equal `Teacher`, even though the `body` contains an entire JSON document. The `jsonpath` parameter modifies its attached predicate and limits its scope to the result of the query. Let's take a look at the query as annotated in figure 4.11.

Figure 4.11. Breaking down a JSONPath query



JSONPath provides tremendous flexibility for selecting the value you need in a large JSON document. Its older cousin, XPath, does the same thing for XML documents.

4.4. SELECTING XML VALUES

Although XML isn't as common in services created in recent years, it's still a prevalent service format and is universally used for SOAP services. If you allowed sending XML as well as JSON with the `PUT` call to `/identities` that we just looked at—perhaps because Abagnale needs to impersonate an enterprise architect—you could expect a body that looks like this:

```
<identities>
  <identity career="Doctor">
    <name>Frank Williams</name>
    <location>Georgia</location>
  </identity>
  <identity career="Teacher">
    <name>Frank Adams</name>
    <location>Utah</location>
  </identity>
</identities>
```

Brigham Young University disputes Abagnale's account of teaching there. Our test scenario could detect Abagnale trying to claim he was a teacher in Utah and send a 400 status code. This is a complex enough query that the existing predicate operators are not up to the task.[4] It also involves using XML attributes in the query.

4

Not even the `matches` predicate, because you cannot parse XML (or HTML) with regular expressions. Trying to parse XML with regular expressions causes the unholy child to weep the blood of virgins: http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags/1732454#1732454.

In the simple case shown in the following listing, the `xpath` parameter mirrors the `jsonpath` parameter, limiting the scope of what the predicate operator examines.

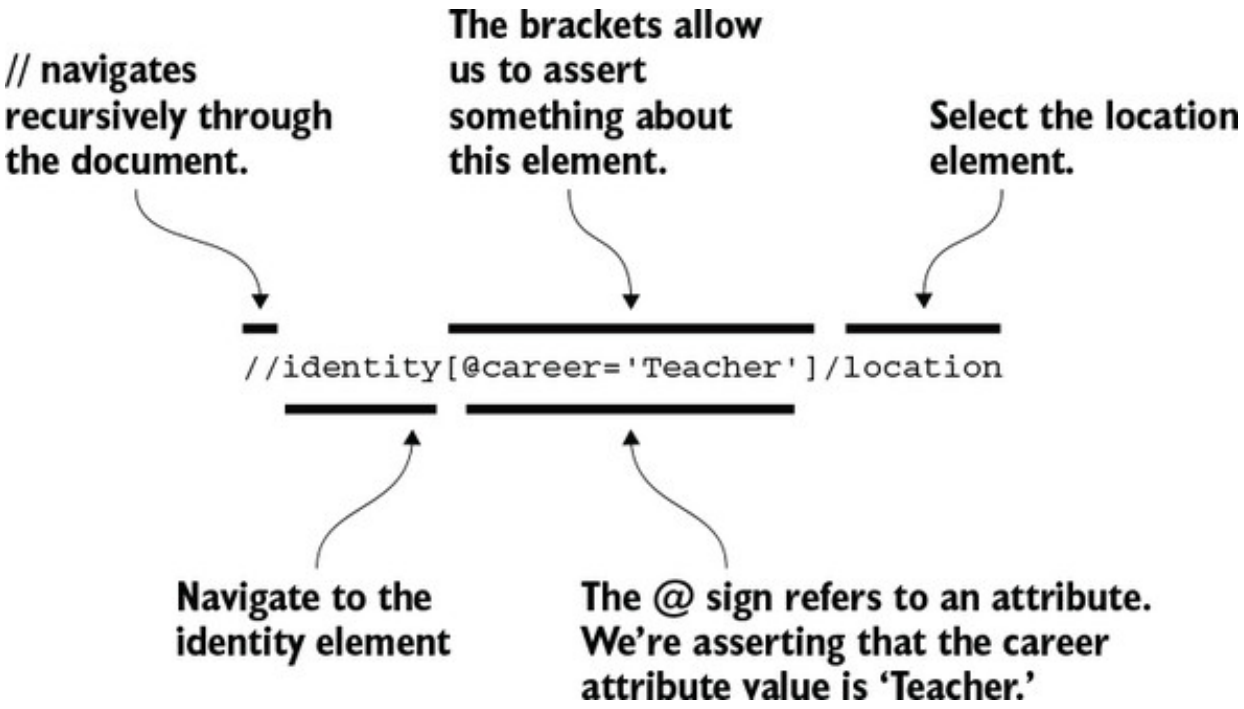
Listing 4.10. Using XPath to prevent Abagnale from claiming he taught in Utah

```
{
  "predicates": [
    { "equals": { "method": "PUT" } },                          1
    { "equals": { "path": "/identities" } },                    1
    {
      "xpath": {
        "selector":
        ➥ "//identity[@career='Teacher']/location"             2
      },
      "equals": { "body": "Utah" }                              3
    }
  ],
  "responses": [{ "is": { "statusCode": 400 } }]                4
}
```

- ***1* Verifies this is a PUT to /identities**
- ***2* Limits the predicate to the given value**
- ***3* The value must equal "Utah."**
- ***4* Returns a Bad Request**

XPath predates JSONPath, and unsurprisingly, their syntax is similar. Figure 4.12 breaks down the XPath expression.

Figure 4.12. Breaking down an XPath query



One of the most unfortunate design decisions in XML is the support for *namespaces*, which has caused needless harm to walls the world over as programmers everywhere banged their heads on them when confronted with namespaced documents. The idea is sensible enough: as you combine multiple XML documents, you need a way of resolving naming conflicts.[5]

5

At least we thought we did. Oddly, the same problem should exist with JSON documents, which lack namespaces, but...

Let's future-proof your XML document by adding namespaces:

```
<identities xmlns:id="https://www.abagnale-spec.com/identity"
            xmlns:n="https://www.abagnale-spec.com/name">
  <id:identity career="Doctor">
    <n:name>Frank Williams</n:name>
    <location>Georgia</location>
  </id:identity>
  <id:identity career="Teacher">
    <n:name>Frank Adams</n:name>
    <location>Utah</location>
  </id:identity>
</identities>
```

Despite your best attempts at future-proofing, you still require a breaking change in version 2 of the Abagnale service that moves the name value to an attribute instead of an XML tag:

```
<identities xmlns:id="https://www.abagnale-spec.com/identity"
            xmlns:n="https://www.abagnale-spec.com/name">
  <id:identity career="Doctor" n:name="Frank Williams">
    <location>Georgia</location>
  </id:identity>
  <id:identity career="Teacher" n:name="Frank Adams">
    <location>Utah</location>
  </id:identity>
</identities>
```

You need to write a test scenario that verifies that the Abagnale service responds with a 400 if you pass the `name` field in the wrong spot, as in the following listing. This is a great opportunity to use the `exists` predicate operator. You also have to add namespaces to your query, as `name` isn't the same as `n:name` in XML.

Listing 4.11. Using XPath to assert that the name attribute exists and the name tag doesn't

```
{
  "predicates": [
    { "equals": { "method": "PUT" } },                          1
    { "equals": { "path": "/identities" } },                    1
    {
      "or": [                                                   2
        {
          "xpath": {
            "ns": {
              "i":
              ➥ "https://www.abagnale-spec.com/identity",      3
              "n": "https://www.abagnale-spec.com/name"         3
            },
            "selector": "//i:identity/n:name"                   4
          },
          "exists": { "body": true }                            4
        },
        {
          "xpath": {
            "selector": "//i:identity[@n:name]",                5
            "ns": {
              "i": "https://www.abagnale-spec.com/identity",
              "n": "https://www.abagnale-spec.com/name"
            }
          },
          "exists": { "body": false }                           5
        }
      ]
    }
  ],
  "responses": [{ "is": { "statusCode": 400 } }]                6
}
```

- ***1*** **Verifies this is a PUT to /identities**
- ***2*** **There are two possible scenarios.**
- ***3*** **Adds the namespace map**
- ***4*** **Matches if the n:name tag exists…**
- ***5*** **…or if the n:name attribute doesn't exist**
- ***6*** **Sends a Bad Request if the predicates match.**

The `xpath` parameter allows you to pass in a namespace map in the `ns` field, which takes a prefix and a URL. The URL has to match the one defined in the XML document, but the prefix can be whatever you want. XPath namespace queries use the prefix in front of each element.

And with that beast of a stub, it's time to take a step back and review what you learned.

SUMMARY

- Predicates allow mountebank to respond differently to different requests. Mountebank ships with a full range of predicate operators, including the versatile `matches` operator, which matches request fields based on a regular expression.
- The `deepEquals` predicate operator is used to match an entire object structure, such as the query object. You also can match a single field within the object (for example, a single query parameter) with one of the standard predicate operators by navigating into the object structure.
- Predicates are case-insensitive by default. You can change that by setting the `caseSensitive` predicate parameter to `true`.
- `jsonpath` and `xpath` predicate parameters limit the scope on the request field to the part that matches the JSONPath or XPath selector.

Chapter 5. Adding record/replay behavior

This chapter covers

- Using proxy responses to capture real responses automatically
- Replaying the saved responses with the correct predicates
- Customizing proxies by changing the headers or using mutual authentication

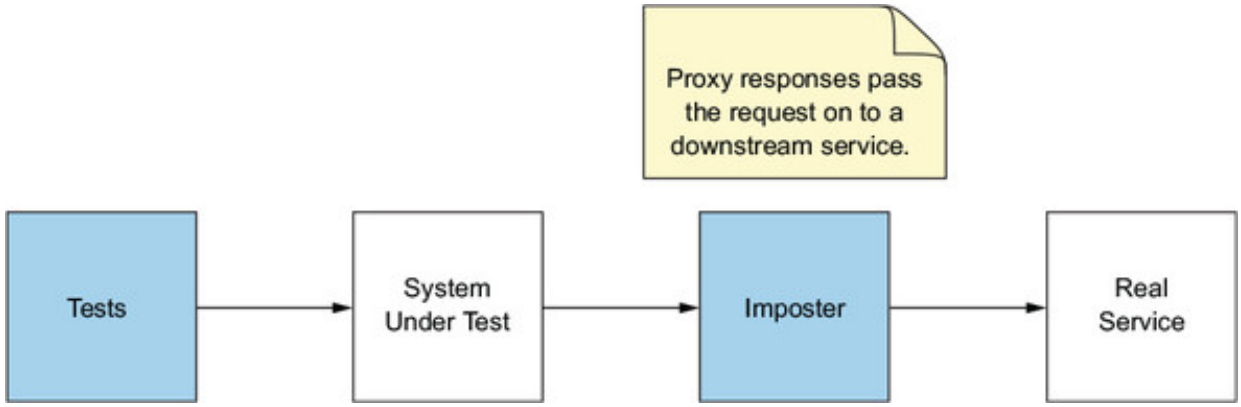
The best imposters don't simply pretend to be someone else; they actively copy the person they impersonate. This mimicry requires both observation and memory: observation to study the behaviors of the person being impersonated and memory to be able to replay those behaviors at a later time. Satirists on comedy shows like *Saturday Night Live*, where actors and actresses often pretend to be famous U.S. political figures, base their performances on those skills.

Mountebank lacks the comedic flair of *Saturday Night Live* impersonators, but it does support a high-fidelity form of mimicry. Rather than creating a canned response for each request, an imposter can go directly to the source. It's as if the imposter is wearing an earpiece, and every time your system under test asks it a question, the real service whispers the answer in your imposter's ear. Better yet, mountebank imposters have a great memory, so once the imposter has heard a response, it can replay the response in the future even without the earpiece. Thanks to the magic of `proxy` responses, a mountebank imposter can be almost indistinguishable from the real thing.

5.1. SETTING UP A PROXY

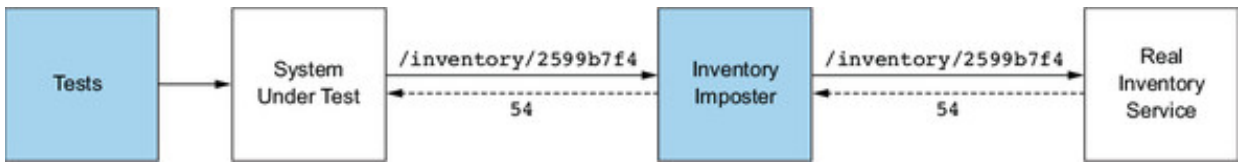
The `proxy` response type lets you put a mountebank imposter between the system under test and a real service that it depends on, saving a real response in the process (figure 5.1).

Figure 5.1. An imposter acting as a proxy



This arrangement allows capturing real data that you can replay in tests, rather than hand-creating it using canned responses. To illustrate, let's revisit the imaginary pet store architecture you first saw in chapter 3. The pet store, like all modern e-commerce shops, needs a service to keep track of inventory, and, to keep it simple, ours takes a product ID on the URL and sends back the on-hand stock for that product. In chapter 3, you virtualized it with hand-created `is` response types. Let's reimagine it using a proxy, which requires the real service to be available to capture the responses, as shown in figure 5.2.

Figure 5.2. Using a proxy to query the downstream inventory



The simplest imposter configuration looks like the following listing.[1]

1

You can follow the examples from the GitHub repository at https://github.com/bbyars/mountebank-in-action.

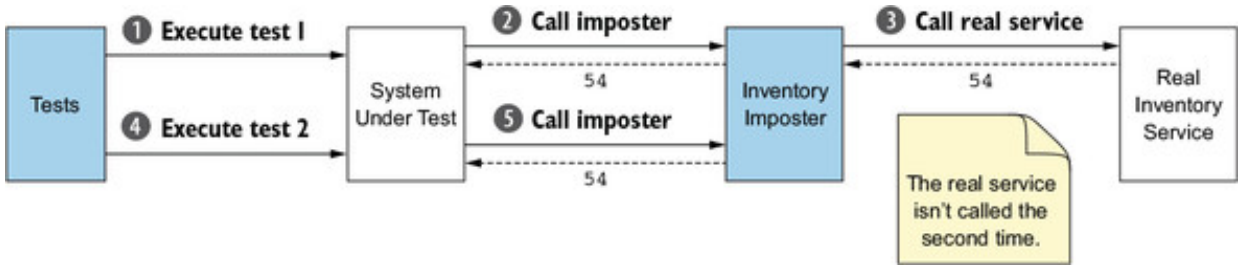
Listing 5.1. Imposter configuration for a basic proxy

```
{
  "port": 3000,
  "protocol": "http",
  "stubs": [{
    "responses": [{
      "proxy": { "to": "http://api.petstore.com" }      1
    }]
  }]
}
```

- ***1*** **New response type**

The difference is in the new response type. Whereas an `is` response tells the imposter to return the given response, a `proxy` response tells the imposter to fetch the response from the downstream service. The basic form of the `proxy` response shown in the listing passes the request unchanged to the downstream service and sends the response unchanged back to the system under test. By itself, that isn't a very useful thing, but the proxy remembers the response and will replay it the next time it sees the same request, rather than fetching a new response (figure 5.3).

Figure 5.3. By default, the proxy returns the first result as the response to all subsequent calls.



Mountebank exposes the current state of each imposter through the API. If you send a `GET` request to http://localhost:2525/imposters/3000, you will see the saved response.[2] It's worth looking at the changed imposter configuration in some detail after the first call to the proxy, as shown in the following listing.

2 The 3000 at the end of the URL is the imposter's port.

Listing 5.2. Saved proxy responses change imposter state

```
{
  "protocol": "http",
  "port": 3000,
  "numberOfRequests": 1,
  "requests": [],
  "stubs": [{
    "predicates": [],                                    1
    "responses": [{
      "is": {                                            2
        "statusCode": 200,
        "headers": {
          "Connection": "close",
          "Date": "Sat, 15 Apr 2017 17:04:02 GMT",
          "Transfer-Encoding": "chunked"
        },
        "body": "54",
        "_mode": "text",
        "_proxyResponseTime": 10                         3
      }
```

```
      }]
    },
    {                                                    4
      "responses": [{
        "proxy": {
          "to": "http://localhost:8080",
          "mode": "proxyOnce"                            5
        }
      }]
    }],
    "_links": {
      "self": {
        "href": "http://localhost:2525/imposters/3000"   6
      }
    }
  }
```

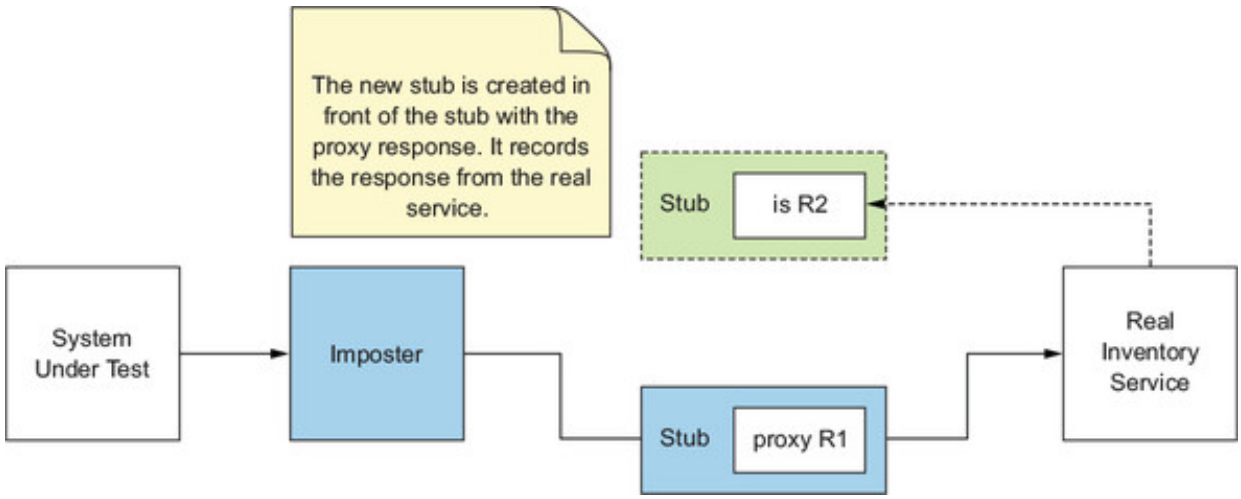
- ***1*** **No predicates? We'll come back to that....?**
- ***2*** **Saves the response as an is response**
- ***3*** **Saves the time to call the downstream service**
- ***4*** **Your original stub is still there.**
- ***5*** **Only calls downstream once**
- ***6*** **The URL you called to get the configuration**

There's a lot in there, and we aren't quite ready to get to all of it yet. Mountebank recorded the time it took to call the downstream service in the `_proxyResponseTime` field; you can use this to add simulated latency during performance testing. We explore how to do that in chapters 7 and 10. The most important observations for now are:

- Mountebank proxies the first call to the base URL given in the `to` field of the proxy configuration. It appends the request path and query parameters and passes through the request headers and body unchanged.
- Mountebank captures the response as a new stub with an `is` response. It saves it *in front of* the stub with the proxy response. (This is what the `proxyOnce` mode means; we will look at the alternative shortly.)
- The newly created stub has no predicates. Because mountebank always uses the first match when iterating over stubs, it will never again call the proxy response, because a stub with no predicates always matches.

Proxies change the state of the imposter. By default, they create a new stub (figure 5.4).

Figure 5.4. The proxy saves the downstream response in a new stub.



The default behavior of a proxy (defined by the `proxyOnce` mode) is to call the downstream service the first time it sees a request it doesn't recognize, and from that point forward to send the saved response back for future requests that look similar. Unfortunately, the example we've been considering isn't discriminatory in how it recognizes requests; *all* requests match the generated stub. Let's fix that.

5.2. GENERATING THE CORRECT PREDICATES

Proxies will create new responses formed from the downstream service response, but you need to give them hints on how to create the request predicates that determine when mountebank will replay those responses. We will start by looking at how you can replay different responses for different request paths.

5.2.1. Creating predicates with predicateGenerators

Your inventory service includes the product id on the path, so sending a `GET` to `/inventory/2599b7f4` returns the inventory for product 2599b7f4, and a `GET` to `/inventory/e1977c9e` returns inventory for product e1977c9e. Let's augment the proxy definition you set up in listing 5.1 to save the response for each product separately. Because it's the `path` that varies between those requests, you need to tell mountebank to create a new `is` response with a `path` predicate. You do so using a `proxy` parameter called `predicateGenerators`. As the name indicates, the `predicateGenerators` are responsible for creating the predicates on the saved responses. You add an object for each predicate you want to generate underneath a `matches` key, as in the following listing.

Listing 5.3. Imposter response that saves a different response for each path

```
{
  "proxy": {
    "to": "http://localhost:8080",     1
```

```
      "predicateGenerators": [{
        "matches": {
          "path": true                    2
        }
      }]
    }
}
```

- ***1* Proxy new paths to the given base URL...**
- ***2* ...but generate a predicate for each new path**

You can test it out with a couple calls using different paths:

```
curl http://localhost:3000/inventory/2599b7f4
curl http://localhost:3000/inventory/e1977c9e
```

Let's look at the state of the imposter again after the change:

```
curl http://localhost:2525/imposters/3000
```

The `stubs` field contains the newly created responses with their predicates, as in the following listing.

Listing 5.4. Saved proxy responses with predicates

```
{
  "stubs": [
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/2599b7f4" }    1
      }],
      "responses": [{
        "is": { "body": "54" }                             2
      }]
    },
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/e1977c9e" }    3
      }],
      "responses": [{
        "is": { "body": "100" }                            4
      }]
    },
    {
      "responses": [{
        "proxy": {
          "to": "http://localhost:8080",
```

```
            "predicateGenerators": [{
              "matches": {
                "path": true
              }
            }],
            "mode": "proxyOnce"
          }
        }]
      }
    ]
  }
```

- ***1*** **Saves the first call as the first stub**
- ***2*** **To save space, most of the response isn't shown.**
- ***3*** **Creates a new stub with a different predicate**
- ***4*** **The response will be different.**

The generated predicates use `deepEquals` for most cases. Recall that the `deepEquals` predicate requires that all fields be present for object fields like `query` and `headers`, so if you include either of those using the simple syntax shown in listing 5.4, the complete set of querystring parameters or request headers would have to be present in a subsequent request for mountebank to serve the saved response:

```
{
  "predicateGenerators": [{
    "matches": {
      "path": true,
      "query": true              1
    }
  }]
}
```

- ***1*** **All query parameters will need to match.**

As you saw in the last chapter, when defining predicates for object fields, you can be more specific if you need to be. If, for example, you want to save a different response for each different `path` and `page` query parameter, regardless of what else is on the querystring, you navigate into the `query` object:

```
{
  "predicateGenerators": [{
    "matches": {
      "path": true,
      "query": {
```

```
          "page": true          1
        }
      }
    }]
}
```

- *1* **Only the page parameter needs to match**

5.2.2. Adding predicate parameters

The `predicateGenerators` field closely mirrors the standard `predicates` field and accepts all the same parameters. Each object in the `predicateGenerators` array generates a corresponding object in the newly created stub's `predicates` array. If, for example, you wanted to generate a case-sensitive match of the `path` and a case-*insensitive* match of the `body`, you could add two `predicateGenerators`, as shown in the following listing.

Listing 5.5. Generating case-sensitive predicates

```
{
  "responses": [{
    "proxy": {
      "to": "http://localhost:8080",
      "predicateGenerators": [
        {                                    1
          "matches": { "path": true },       1
          "caseSensitive": true              1
        },                                   1
        {                                    2
          "matches": { "body": true }        2
        }                                    2
      ]
    }
  }]
}
```

- *1* **Generates a case-sensitive predicate**
- *2* **Generates a default case-insensitive predicate**

The newly created stub has both predicates:

```
{
  "predicates": [
    {
      "caseSensitive": true,
      "deepEquals": { "path": "..." }
```

```
      },
      {
        "deepEquals": { "body": "..." }
      }
    ],
    "responses": [{
      "is": { ... }
    }]
  }
```

As you saw in the last chapter, more parameters are available beyond `caseSensitive`. The `jsonpath` and `xpath` predicate parameters allow you to limit the scope of the predicate, and you can generate those too.

Generating JSONPath predicates

In chapter 4, we demonstrated JSONPath predicates in the context of virtualizing the inimitable Frank Abagnale service, which showed a list of fake identities the famous (real) imposter had assumed. A partial list of identities might look like this:

```
{
  "identities": [
    {
      "name": "Frank Williams",
      "career": "Doctor",
      "location": "Georgia"
    },
    {
      "name": "Frank Adams",
      "career": "Teacher",
      "location": "Utah"
    }
  ]
}
```

If you needed a predicate that matched the `career` field of the last element of the `identities` array, then you could use the same JSONPath selector you saw in the previous chapter. Because you now want mountebank to *generate* the predicate, you specify the selector in the `predicateGenerators` object and rely on the proxy to fill in the value, as in the following listing.

Listing 5.6. Specifying a `jsonpath predicateGenerators`

```
{
  "proxy": {
    "to": "http://localhost:8080",
    "predicateGenerators": [{
```

```
        "jsonpath": {                                          1
          "selector": "$.identities[(@.length-1)].career"      1
        },                                                     1
        "matches": { "body": true }                            2
      }]
    }
}
```

- ***1* Saves the value defined by the selector…**
- ***2* …from the body field.**

Remember, `predicateGenerators` work on the incoming *request*, so the JSONPath selector saves off the value in the request body. If you sent the Abagnale JSON in listing 5.6 to your proxy, the generated stub would look something like this:

```
{
  "predicates": [{
    "jsonpath": {
      "selector": "$.identities[(@.length-1)].career"
    },
    "deepEquals": { "body": "Teacher" }            1
 }],
 "responses": [{
    "is": { ... }
 }]
}
```

- ***1* The value captured from the selector in the incoming request body**

Future requests that match the given selector will use the saved response.

Generating XPath predicates

The same technique works for XPath. If you translate Abagnale's list of identities to XML, it might look like:

```
<identities>
  <identity career="Doctor">
    <name>Frank Williams</name>
    <location>Georgia</location>
  </identity>
  <identity career="Teacher">
    <name>Frank Adams</name>
    <location>Utah</location>
  </identity>
</identities>
```

The `predicateGenerators` mirrors the `xpath` predicate you saw in chapter 4, so if you had a need for a predicate to match on the location where Abagnale pretended to be a teacher, the following listing would do the trick.

Listing 5.7. Specifying an `xpath predicateGenerators`

```
{
  "proxy": {
    "to": "http://localhost:8080",
    "predicateGenerators": [{
      "xpath": {                                                    1
        "selector": "//identity[@career='Teacher']/location"       1
      },                                                            1
      "matches": { "body": true }                                   2
    }]
  }
}
```

- ***1* Saves the value defined by the selector...**
- ***2* ...from the body field.**

The predicates that the proxy creates show the correct location:

```
{
  "predicates": [{
    "xpath": {
      "selector": "//identity[@career='Teacher']/location"
    },
    "deepEquals": { "body": "Utah" }          1
 }],
 "responses": [{
    "is": { ... }
 }]
}
```

- ***1* The value captured from the selector in the incoming request body**

Capturing multiple JSONPath or XPath values

JSONPath and XPath selectors both can capture multiple values. To take a simple example, look at the following XML:

```
<doc>
  <number>1</number>
  <number>2</number>
  <number>3</number>
```

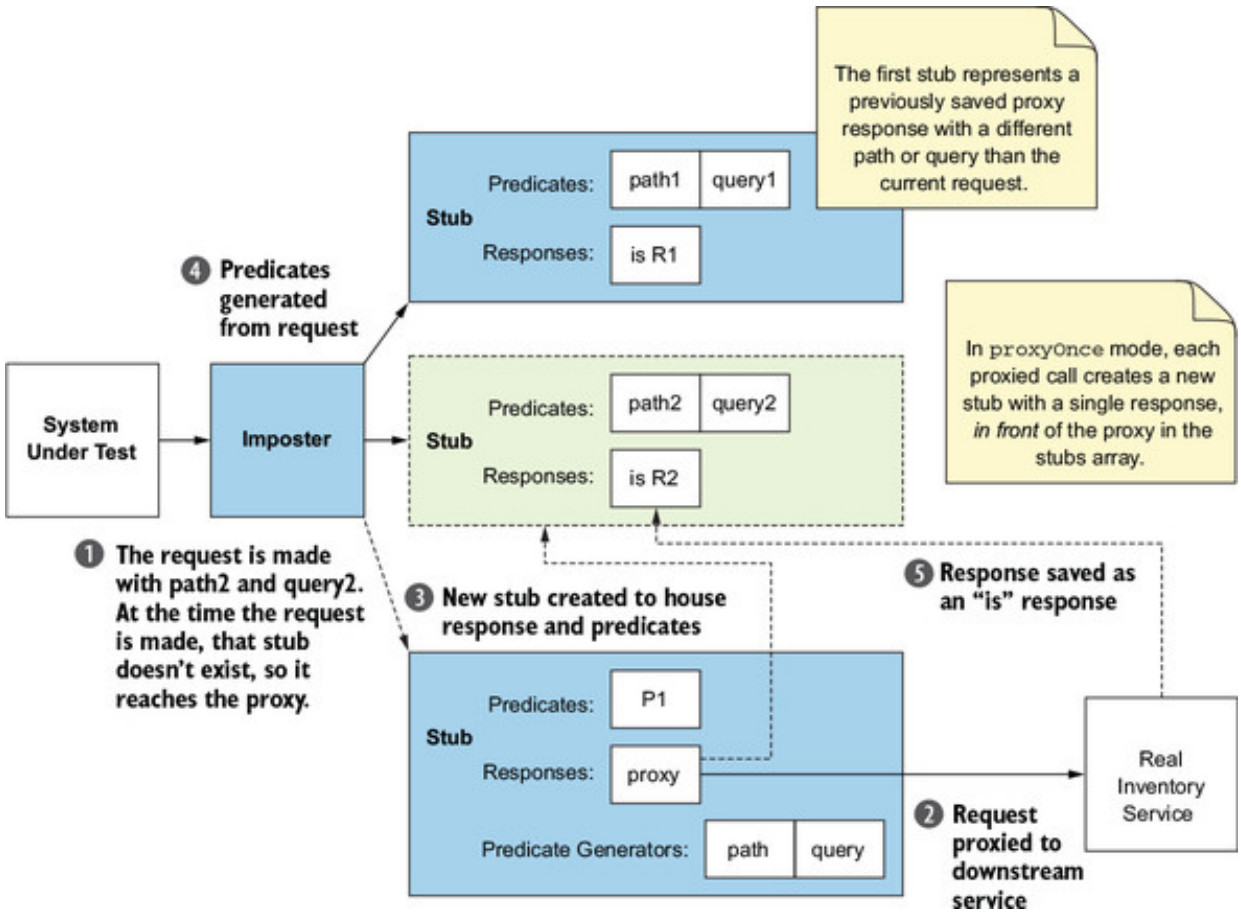
```
</doc>
```

If you use the XPath selector of `//number` on this XML document, you get three values: 1, 2, and 3. The `predicateGenerators` field is smart enough to capture multiple values and save them using a standard predicate array, which requires all results to be present in subsequent requests to match but allows them to be present in any order.

5.3. CAPTURING MULTIPLE RESPONSES FOR THE SAME REQUEST

The examples we looked at so far have been great for minimizing traffic to a downstream service while still collecting real responses. For each type of request, defined by the `predicateGenerators`, you only pass the request to the real service once. This is the promise of the default mode, appropriately called `proxyOnce`. The guarantee is satisfied by ensuring that mountebank creates new stubs before the stub with the proxy response (figure 5.5). Mountebank's first match policy will ensure that subsequent requests matching the generated predicates don't reach the proxy response.

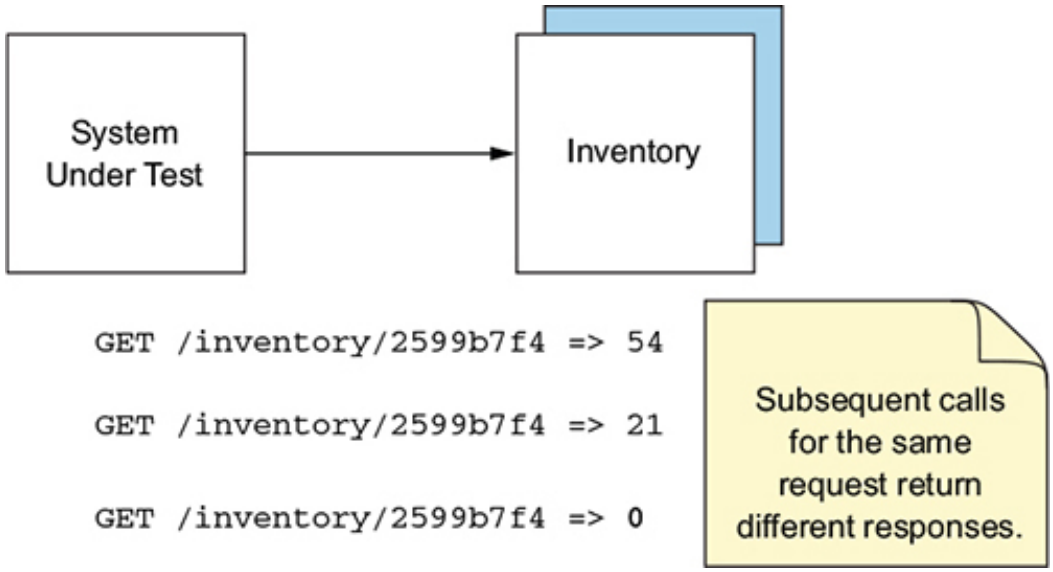
Figure 5.5. In `proxyOnce` mode, mountebank creates new stubs before the stub with the proxy.



A significant downside to `proxyOnce` is that each generated stub can have only one response. This is a problem for your inventory service, which returns different

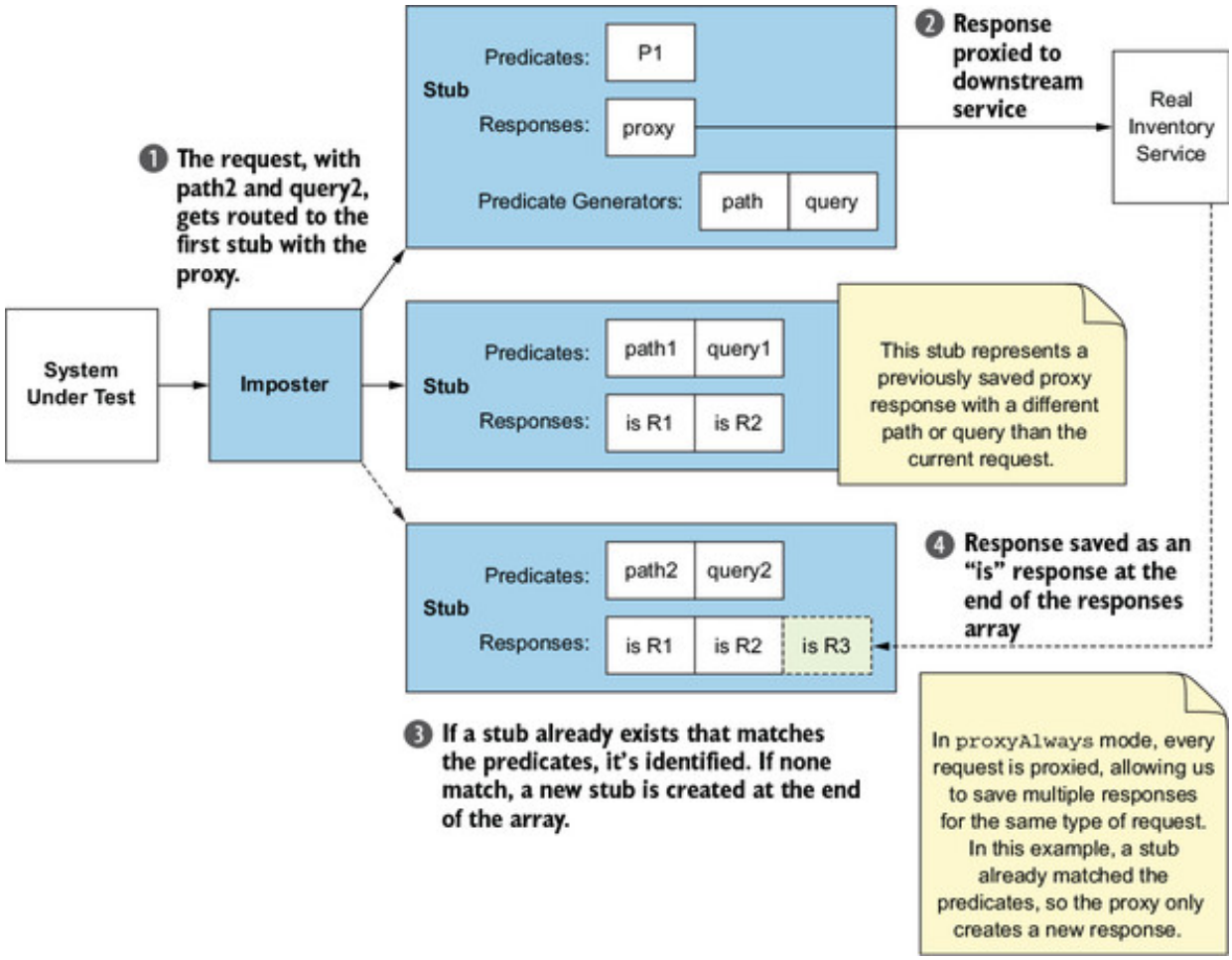
responses over time for the same request, reflecting volatility in stock for an item (figure 5.6).

Figure 5.6. Volatile responses for the same request



In `proxyOnce` mode, mountebank captures only the first response (54). If your test cases relied on the volatility of inventory over time, you'd need a proxy that would let you capture a richer data set to replay. The `proxyAlways` mode ensures that *all* requests reach the downstream service, allowing you to capture multiple responses for a single request type (figure 5.7).

Figure 5.7. In `proxyAlways` mode, new stubs are created after the stub with the proxy



Creating this type of proxy is as simple as passing in the mode, as in the following listing.

Listing 5.8. Creating a `proxyAlways` proxy response

```
{
  "proxy": {
    "to": "http://localhost:8080",
    "mode": "proxyAlways",                1
    "predicateGenerators": [{
      "matches": { "path": true }
    }]
  }
}
```

- ***1*** **Ensures that all responses are captured**

The key difference in the mechanics between `proxyOnce` and `proxyAlways`, as shown in figures 5.5 and 5.7, is that `proxyOnce` generates new stubs *before* the stub containing the proxy response, whereas `proxyAlways` generates stubs *after* the proxy stub. Both approaches rely on mountebank's first-match policy when matching a request to a stub. In the `proxyOnce` case, a subsequent request matching the generated predicates is guaranteed to match before the proxy stub, and in the `proxyAlways` case, the proxy

stub is guaranteed to match before the generated stubs.

But `proxyAlways` does more than create new stubs. It first looks to see if a stub whose predicates already exists match the generated predicates, and, if so, it *appends* the saved response to that stub. This behavior allows multiple responses to be saved for the same request. You can see this by calling the imposter in listing 5.8 a few times and querying its state (by sending a `GET` request to http://localhost:2525/imposters/3000, assuming it was started on port 3000). To save space and make the salient bits stand out, I've omitted the full response inside each generated `is` response in the following listing.

Listing 5.9. The imposter state after a few calls to a `proxyAlways` response

```
{
  "stubs": [
    {
      "responses": [{
        "proxy": {
          "to": "http://localhost:8080",
          "mode": "proxyAlways",                          1
          "predicateGenerators": [{
            "matches": { "path": true }
          }]
        }
      }]
    },
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/2599b7f4" }   2
      }],
      "responses": [
        { "is": { "body": "54" } },                       3
        { "is": { "body": "21" } },                       3
        { "is": { "body": "0" } }                         3
      ]
    },
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/e1977c9e" }   4
      }],
      "responses": [{
        "is": { "body": "100" }                           5
      }]
    }
  ]
}
```

- ***1*** **Ensures that all requests are proxied**

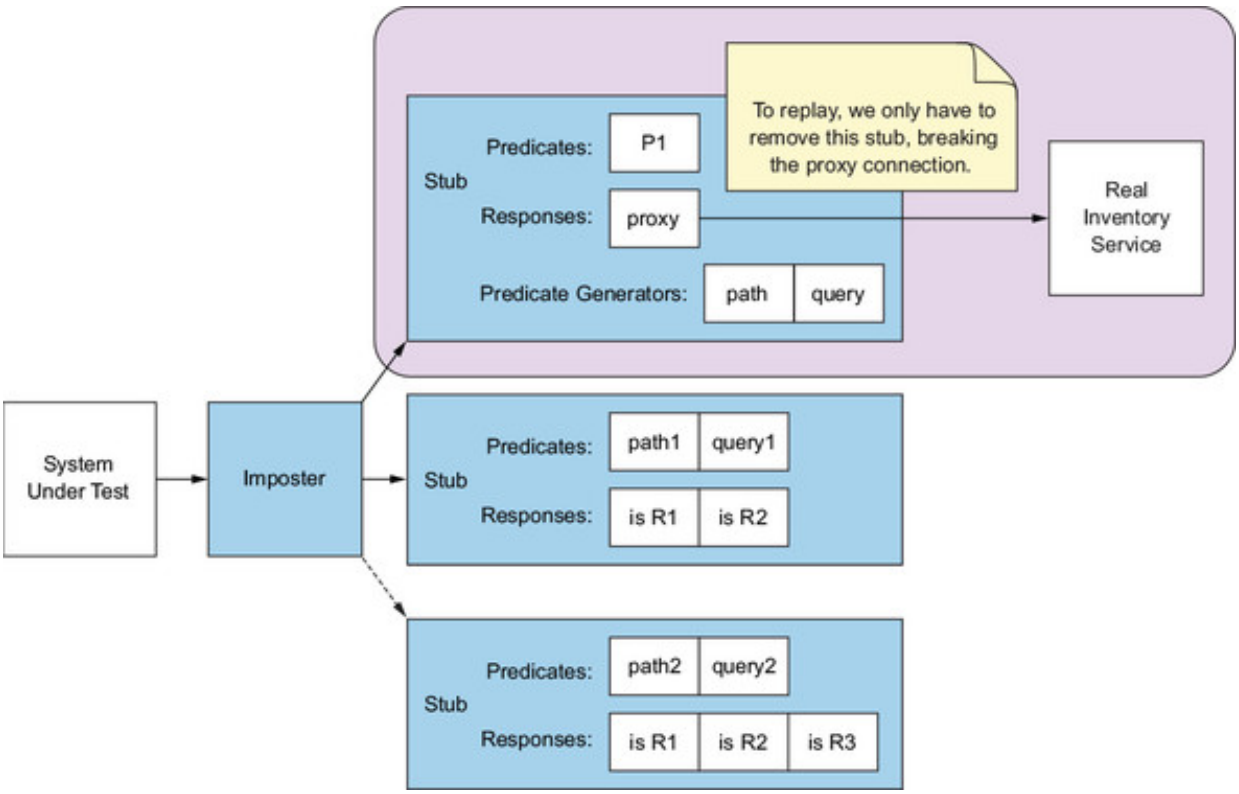
- **2 First request type**
- **3 All responses saved**
- **4 Second request type**
- **5 Only one response**

A `proxyAlways` proxy allows you to capture a full set of test data that is as rich as your downstream service (at least for the requests sent to it). Although this is great for supporting complex test cases, it comes with a significant problem. As you can see in listing 5.9, none of the saved responses will ever get called. With `proxyOnce`, you don't need to worry about switching from recording to replaying; it happens automatically. Not so with `proxyAlways`, so it's time to look at how you can tell mountebank to replay all the data it has captured.

5.4. WAYS TO REPLAY A PROXY

Conceptually, the switch from recording to replaying is as simple as removing the proxy response (figure 5.8).

Figure 5.8. Replaying involves removing the proxy



All it takes to switch into replay mode is a single command, as follows:

```
mb replay
```

If you watch the logs after that command, you'll see something like the following:

```
info: [mb:2525] GET /imposters?replayable=true&removeProxies=true
info: [mb:2525] PUT /imposters
info: [http:3000] Ciao for now
info: [http:3000] Open for business...
```

The switch involves resetting all imposters, removing their proxies. You can see in the third line that you're shutting down the existing imposter (`Ciao for now`) and restarting it on the fourth line (`Open for business...`). The line before shows the API call to send the altered configuration; this is the same line you'd see if you started mountebank with the `--configfile` command-line option.

The first line shows a different part of the mountebank REST API. Just as you can query the state of a single imposter by sending a `GET` request to http://localhost:2525/imposters/3000 (assuming the imposter is on port 3000), you can query *all* imposters at http://localhost:2525/imposters. The `replay` command adds two query parameters to that call:

- Because the configuration for all imposters is quite possibly a considerable amount of data, it's trimmed by default. The `replayable` query parameter ensures that all data essential (and no more) for replay is returned.
- The `removeProxies` parameter removes the proxy responses, leaving only the captured `is` responses.

The `mb replay` command replays the responses as is. If you need to tweak the captured responses for any reason, you can always use the API call to get the data yourself and process it as appropriate. Even better, you can let mountebank's command line do the job for you. The following command saves the current state of all imposters, with proxies removed:

```
mb save --savefile imposters.json --removeProxies
```

The `mb save` command saves all imposter configuration to the given file. The `--savefile` argument specifies where to save the configuration, and the `--removeProxies` flag strips the proxy responses from the configuration. Functionally, the `mb replay` command is nothing but an `mb save` followed by a restart. The following command reimplements the replay command:

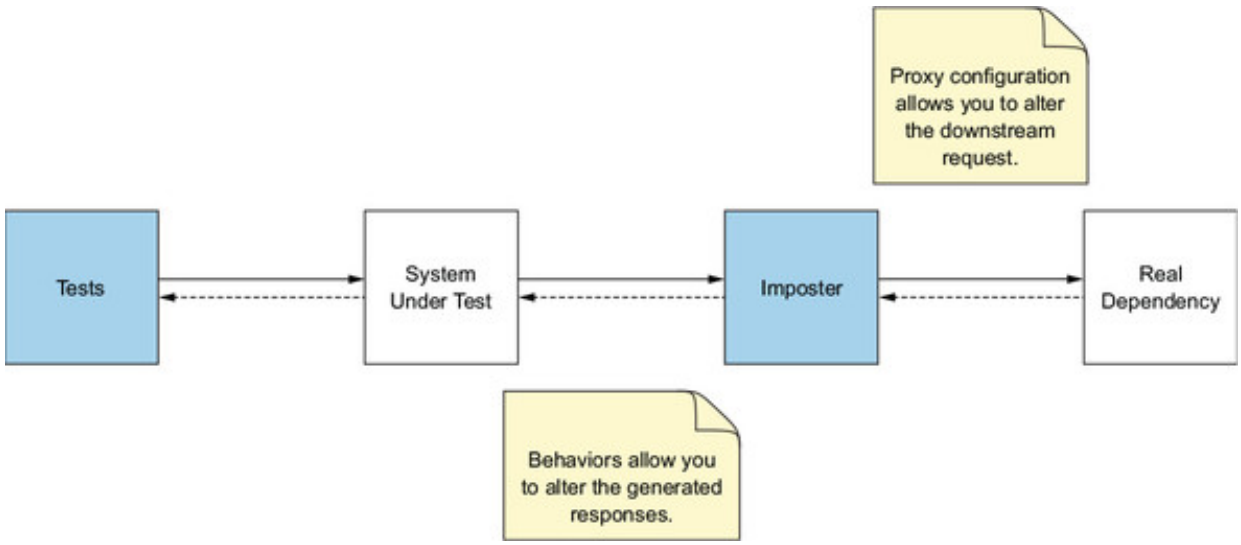
```
mb save --savefile imposters.json --removeProxies
mb restart --configfile imposters.json
```

The ability to save all responses to a downstream service in `proxyAlways` mode and replay them with a single command dramatically simplifies capturing data for rich test suites.

5.5. CONFIGURING THE PROXY

Proxies are configurable, both on the request they send to the downstream service and the generated responses they send back to the system under test (figure 5.9).

Figure 5.9. You can configure both the proxy request and the generated response.

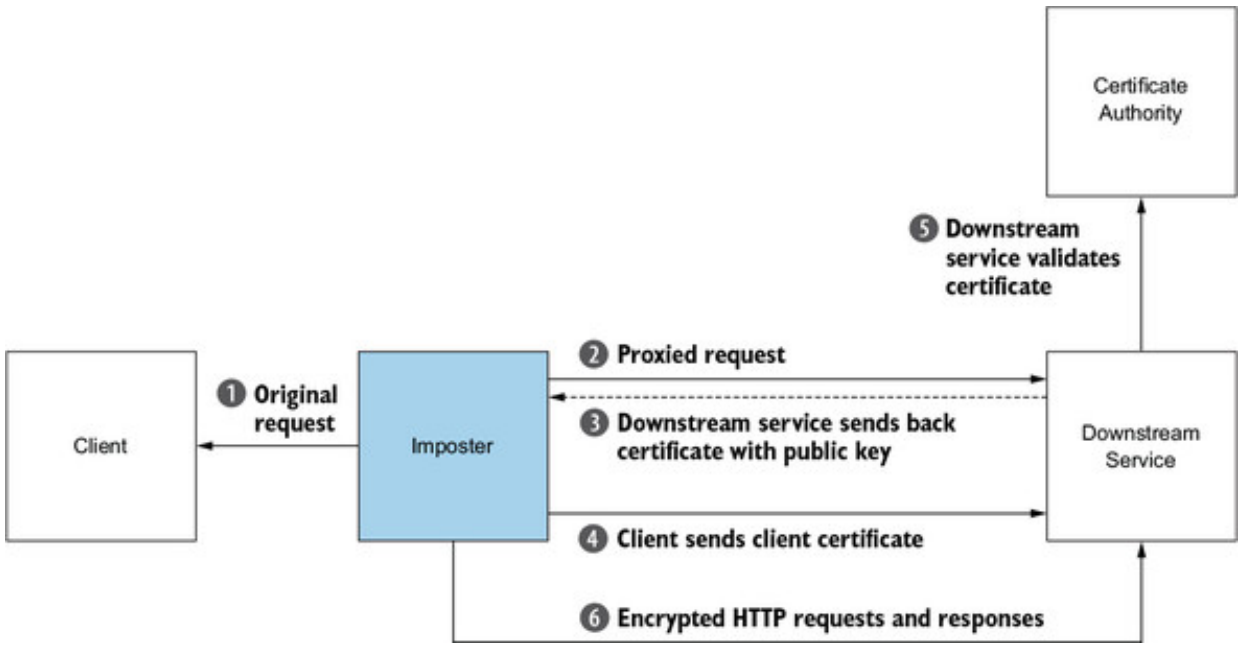


We look at how to alter the responses in chapter 7 when we examine behaviors. You can apply two basic types of configuration to the proxied request: using certificate-based mutual authentication and adding custom headers.

5.5.1. Using mutual authentication

Recall from chapter 3 that you can configure imposters to expect a client certificate by setting the `mutualAuth` field to true. In that case, configuring the certificate and private key is the responsibility of the system under test. If the downstream service you are proxying to requires mutual authentication, then you have to configure the certificate on the proxy itself (figure 5.10).

Figure 5.10. Configuring the proxy to pass a client certificate



In this case, setting up the proxy is similar to how you set HTTPS imposters. You set the certificate and private key in PEM format directly on the proxy, as shown in the following listing.

Listing 5.10. A proxy configured to pass a client certificate

```
{
  "proxy": {
    "to": "https://localhost:8080",
    "mode": "proxyAlways",
    "predicateGenerators": [{
      "matches": { "path": true }
    }],
    "key": "-----BEGIN RSA PRIVATE KEY-----\n...",   1
    "cert": "-----BEGIN CERTIFICATE-----\n..."       1
  }
}
```

- ***1*** **The actual text is much longer.**

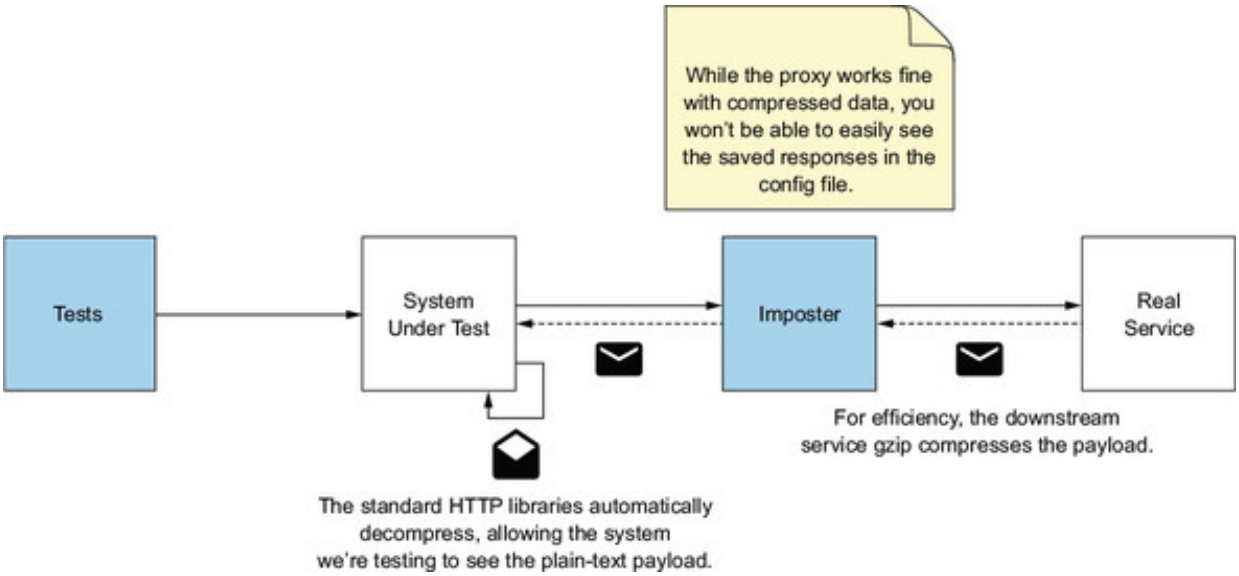
Refer to chapter 3 for the full PEM format and how to create certificates.

5.5.2. Adding custom headers

Occasionally, it's useful to add another header that gets passed to the downstream service. For example, many services return compressed responses for efficiency. Although the original data may be human-readable JSON, after compression it turns into unreadable binary. By default, any proxies you set up will respond with the compressed data unchanged. Because negotiating gzipped compression through headers is a standard operation in HTTP, the HTTP libraries that the system under test

uses would decompress the data, allowing the code that you're testing to see the plain text response (figure 5.11).

Figure 5.11. Proxying compressed responses



The standard proxy configurations that you've seen so far have no issues returning the compressed data to the system under test. The problem occurs when you want to actually *look* at the data, such as after saving it in a configuration file using the `mb save` command. You may want to examine the JSON bodies of the generated `is` responses and perhaps tweak them to better fit your test cases, but you won't be able to. All you will be able to see are encoded binary strings.[3]

3

We'll look at how mountebank handles binary data in chapter 8.

HTTP provides a way for clients to tell servers not to send back compressed data by setting the `Accept-Encoding` header to `"identity"`. The original request from the system under test likely doesn't include this header, because it can handle the compressed data just fine (and using compressed data in production is a good idea anyway for efficiency reasons). Fortunately, you can inject headers into the proxied request, as shown in the following listing.

Listing 5.11. Injecting a header into the request to prevent response compression

```
{
  "proxy": {
    "to": "http://localhost:8080",
    "mode": "proxyAlways",
    "predicateGenerators": [{
      "matches": { "path": true }
    }],
```

```
      "injectHeaders": {
        "Accept-Encoding": "identity"          1
      }
    }
  }
```

- ***1*** **Prevents response compression**

If you need to inject multiple headers, add multiple key/value pairs to the `injectHeaders` object. Each header will be added to the original request headers.

5.6. PROXY USE CASES

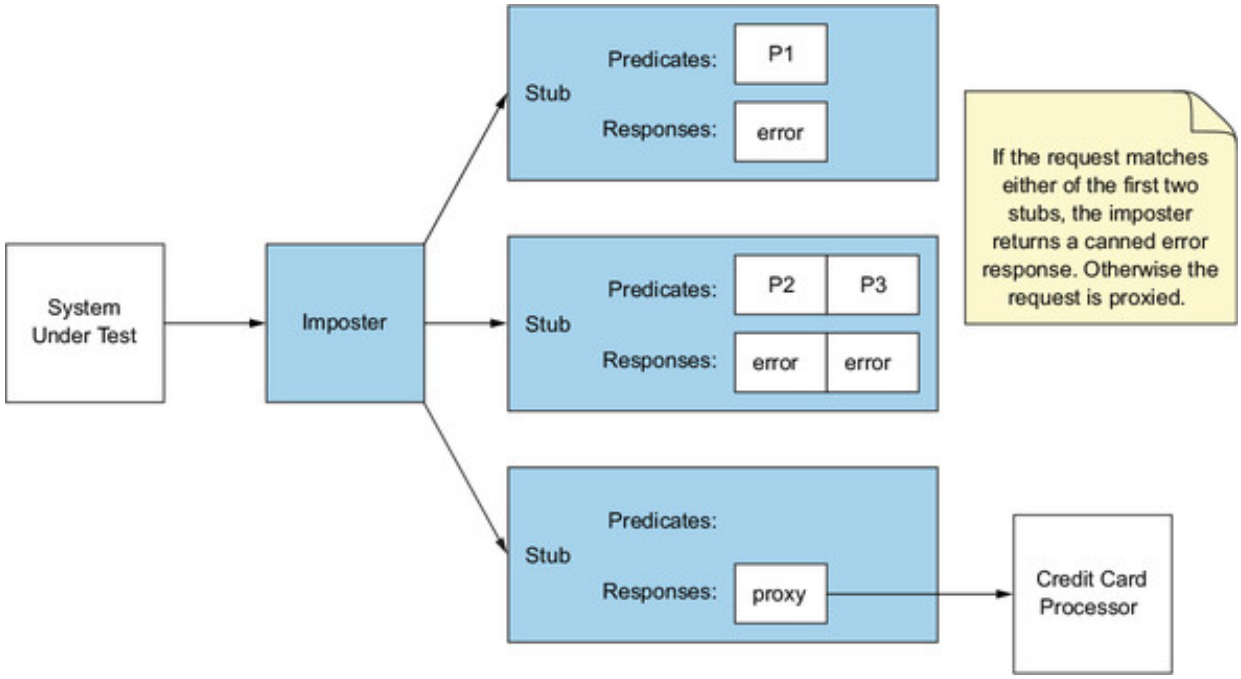
The examples so far in this chapter have focused on using proxies to record and replay. That is the most common use case for proxies, as it allows you to capture a rich set of test data for your test suite by recording real traffic. In addition, proxies have at least two other use cases: as a fallback response and as a way of presenting an HTTP face to an HTTPS service.

5.6.1. Using a proxy as a fallback

Though it isn't a common scenario, sometimes it's convenient to test against a real dependency when that dependency is stable, reliable, and highly available. One project I was on involved testing against a software-as-a-service (SaaS) credit card processor, and the SaaS provider supported a reliable preproduction environment for testing. In fact, it was perhaps *too* reliable. "Happy path" testing (testing the expected flow through the service) was easy, but the service was so reliable that testing error conditions was difficult.

You can get the best of both worlds by using a partial proxy. Most calls flow through to the credit card processing service, but a few special requests trigger canned error responses. Mountebank supports this scenario by relying on its first-match policy, putting the error conditions first and the proxy last (figure 5.12).

Figure 5.12. Mixing canned responses with a fallback proxy



Notice that proxy has no predicates, which means all requests that don't match the predicates on the previous stubs will flow through to the proxy. Ensuring that all requests flow through to the credit card processor involves putting the proxy in `proxyAlways` mode. In the following listing, the code to do this relies on putting the proxy stub last.

Listing 5.12. Using a partial proxy

```
{
  "port": 3000,
  "protocol": "http",
  "stubs": [
    {
      "predicates": [{                                  1
        "contains": { "body": "5555555555555555" }      1
      }],                                               1
      "responses": [{                                   2
        "is": { "body": "FRAUD ALERT... " }             2
      }]                                                2
    },
    {
      "predicates": [{                                  3
        "contains": { "body": "4444444444444444" }      3
      }],                                               3
      "responses": [{                                   4
        "is": { "body": "INSUFFICIENT FUNDS..." }       4
      }]                                                4
    },
    {
      "responses": [{
        "proxy": {                                      5
          "to": "http://localhost:8080",                5
          "mode": "proxyAlways"                         5
```

```
            }                                                    5
          }]
        }
      ]
    }
```

- *1* **If the body contains this credit card #...**
- *2* **...send a fraud-alert response.**
- *3* **If it contains this credit card #...**
- *4* **...send an over-balance response.**
- *5* **All other calls go to the real service.**

In this scenario, you would not use `mb replay`, because you aren't trying to virtualize the downstream service. Mountebank still creates new stubs and responses, so for any long-lived partial proxy, you'll run into memory leaks. A future version of mountebank will support configuring proxies not to remember each response.

5.6.2. Converting HTTPS to HTTP

Another less common scenario is to make an HTTPS service easier to test against. When I've seen this done, it has been as a workaround in an enterprise test environment that hasn't configured SSL correctly, leading to certificates that don't validate against the Certificate Authority. As I mentioned in chapter 3, I strongly advise against changing the system under test to accept invalid certificates, because you risk releasing that configuration into production. Although the best solution is to fix the test environment certificates, the division of labor in some enterprises makes that difficult to do. Assuming that you're confident in the ability of the system under test to negotiate HTTPS with valid certificates (behavior that's nearly always provided by the core language libraries), you can rely on mountebank to bridge the misconfigured HTTPS to HTTP, as shown in the following listing.

Listing 5.13. Using a proxy to bridge HTTPS to HTTP

```
{
  "port": 3000,
  "protocol": "http",                          1
  "stubs": [{
    "responses": [{
      "proxy": {
        "to": "https://localhost:8080",        2
        "mode": "proxyAlways"
      }
    }]
```

```
    }]
}
```

- ***1*** **The imposter itself is an HTTP server…**
- ***2*** **…but forwards requests to an HTTPS server.**

Because mountebank is designed to help test in environments that have yet to be fully configured, the imposter itself doesn't validate the certificate during the proxy call. This doesn't require any similar change in configuration in the system under test.

SUMMARY

- Proxy responses capture real downstream responses and save them for future replay. The default `proxyOnce` mode saves the response in front of the `proxy` stub, meaning you don't need to do anything to replay the response.
- The `proxyAlways` mode allows you to capture a full set of test data by capturing multiple responses for the same logical request. You have to explicitly switch from record mode to replay mode by using `mb replay`.
- The `predicateGenerators` field tells mountebank how to create the predicates based on the incoming request. All fields used to discriminate requests are listed under the `matches` object. You configure parameters no differently than you would for normal predicates.
- Proxies support mutual authentication. You are responsible for setting the `key` and `cert` fields.
- You can alter the request headers that your proxied request sends to the downstream service with the `injectHeaders` field. This is useful, for example, to disable compression so you can save a text response.

Chapter 6. Programming mountebank

This chapter covers

- Matching requests even when none of the standard predicates do the trick
- Adding dynamic data to mountebank responses
- Making network calls to create a stub response
- Securing mountebank against unwanted remote execution
- Debugging your scripts within mountebank

Most developers have the experience of a tool or framework that makes solving a problem harder than it should be or outright impossible. A primary goal in mountebank is to keep things that should be easy actually easy, and to make hard things possible. Comprehensive default settings, predicates, and simple `is` responses enable you to solve easy problems with simple solutions. Support for certificates, JSON, XML, and proxies help you to create solutions for a harder set of problems. But sometimes, that's not enough.

At times, the built-in features of mountebank may not directly support your use case. Matching predicates based on JSON and XML is nice, but what if your requests use CSV format? Or say you have a service that needs to replay information from an earlier request in a subsequent response, requiring stateful memory. Maybe your use case extends beyond what a proxy is capable of doing. Or perhaps you need to grab some data from outside mountebank itself and insert it into the stub response.

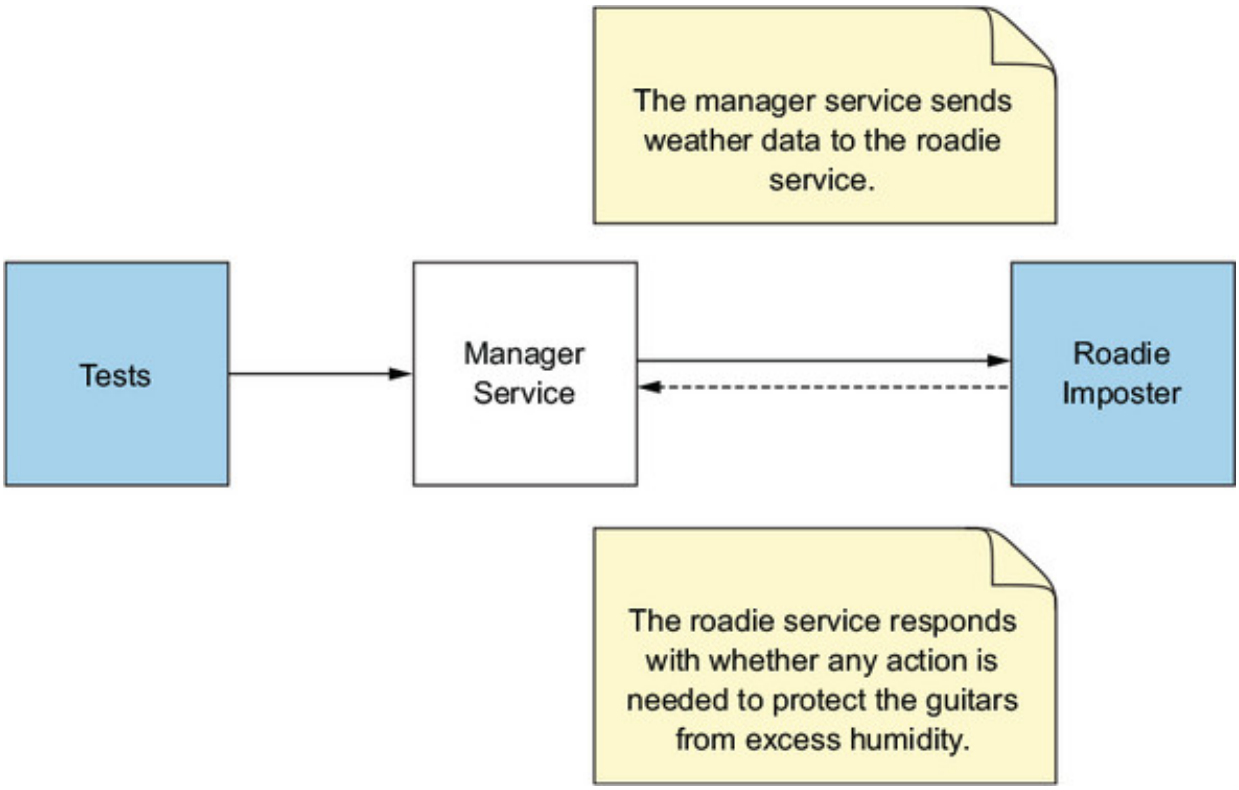
You need `inject`.

Mountebank ships with several scriptable interfaces, which we'll examine in chapters 7 and 8. The main scriptable interface comes in the form of a new predicate type and a new response type, both called `inject`, that allows you to match requests and create responses in ways mountebank wasn't designed to support.

6.1. CREATING YOUR OWN PREDICATE

Like most technologists, I like to while away the idle hours pretending to be a rock star. Scaling that desire into a set of microservices requires both a manager service (to manage my itinerary) and a roadie service (to manage my instruments). The two services collaborate to protect my guitars against excess humidity (figure 6.1).

Figure 6.1. Service collaboration used to protect my guitars



Let's assume that the manager service (the system under test) is responsible for sending updated weather reports in CSV format to the roadie service, which uses that information to protect my guitars against dangerous humidity levels. Nice acoustic guitars are made of wood that generally wants to stay in the 40–60% humidity range, and the roadie service is responsible for alerting the manager service when a plan is needed to protect the guitars against excessive humidity.

Although virtualizing a roadie service is a bit fantastical, you'll run into services that use something other than JSON or XML as their *lingua franca* from time to time. CSV is still a relatively popular format in certain types of service integrations, particularly those that involve passing some bulk-style information between teams or organizations, such as:

- Retrieving an augmented set of customer information from a marketing research partner who has added (for example) demographic information to your customer information

- Retrieving an updated list of tax rates by ZIP code in the United States from an outside partner
- Retrieving bulk reporting information from an internal team

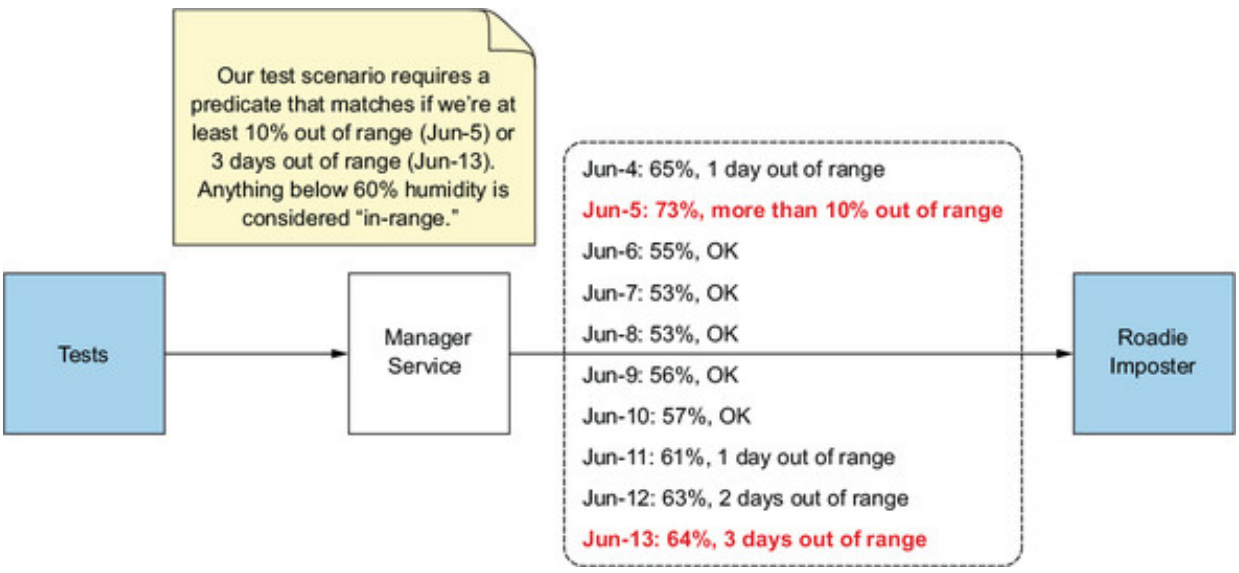
For this example, you'll expect the manager service to pass weather data showing the next 10 days of weather information in a format similar to what you might get on a site like weather.com, as shown in the following listing.

Listing 6.1. A weather CSV service payload

```
Day,Description,High,Low,Precip,Wind,Humidity
4-Jun,PM Thunderstorms,83,71,50%,E 5 mph,65%
5-Jun,PM Thunderstorms,82,69,60%,NNE 8mph,73%
6-Jun,Sunny,90,67,10%,NNE 11mph,55%
7-Jun,Mostly Sunny,86,65,10%,NE 7 mph,53%
8-Jun,Partly Cloudy,84,68,10%,ESE 4 mph,53%
9-Jun,Partly Cloudy,88,68,0%,SSE 11mph,56%
10-Jun,Partly Cloudy,89,70,10%,S 15 mph,57%
11-Jun,Sunny,90,73,10%,S 16 mph,61%
12-Jun,Partly Cloudy,91,74,10%,S 13 mph,63%
13-Jun,Partly Cloudy,90,74,10%,S 17 mph,64%
```

Mountebank doesn't natively support CSV, and it doesn't have a `greaterThan` predicate to look for humidity levels greater than 60%, so you are already outside of its built-in capabilities, but let's raise the stakes even higher. Your expectations of the roadie service are that it only presses the panic button if you are out of range for at least three consecutive days, *or* if you are more than 10% out of range for a single day (figure 6.2).

Figure 6.2. A test scenario requiring advanced predicate logic



That makes for a complex predicate, but no one said becoming a rock star would be

easy. You can create your own predicate using JavaScript and the `inject` predicate, but first you have to start mountebank with the `--allowInjection` command line flag:

```
mb --allowInjection
```

The logs show a warning at startup that hints at why injection is disabled by default:

```
info: [mb:2525] mountebank v1.13.0 now taking orders -
➥ point your browser to http://localhost:2525 for help
warn: [mb:2525] Running with --allowInjection set.
➥ See http://localhost:2525/docs/security for security info
```

You can go ahead and view the docs provided at the given URL now, or wait until you get to the topic of security later in this chapter. One thing you should *not* do is ignore the warning. It's great that you can do wonderfully complex feats of logic with JavaScript injection and mountebank. Unfortunately, so can malicious attackers on your network who can access your machine. They now have a remote execution engine listening on a socket. You can protect yourself, and we look at ways to do that at the end of this chapter. For now, the safest option is to add the `--localOnly` flag, which disallows remote connections:

```
mb restart --allowInjection --localOnly
```

All of the predicates we looked at in chapter 4 operate on a single request field. Not so with `inject`, which gives you complete control by passing the entire request object into a JavaScript function you write. That JavaScript function returns `true` if the predicate matches the request and `false` otherwise, as shown in the following listing.[1]

1 JavaScript as a language has a lot of warts, and experts will tell you that you can return `truthy` or `falsy`. I never cared to learn what that meant, so I stick with `true` or `false` and recommend you do too.

Listing 6.2. The structure of an `inject` predicate

```
function (request) {
  if (...) {            1
    return true;        2
  }
```

```
    else {
      return false;      3
    }
  }
```

- ***1*** **Condition can use entire request**
- ***2*** **Predicate matched**
- ***3*** **Predicate didn't match**

Plugging the function into the `predicates` array of a stub involves JSON-escaping the function, which replaces newlines with `'\n'`, and escaping double quotes:

```
{
  "predicates": [{
    "inject": "function (request) {\n  if (...) {\n
➥ return true;\n  }\n  else {\n    return false;\n  }\n}"
  }],
  "responses: [{
    "is": { ... }
  }]
}
```

I wouldn't recommend doing the JSON-escaping by hand. In the examples that follow, you will use EJS templating and the `stringify` function mountebank adds to EJS. (Refer to chapter 3 for details on how to lay out configuration files.)[2] If you're building up the imposter configuration in code, your JSON libraries should manage the escaping for you.

2

Browse the GitHub repo at https://github.com/bbyars/mountebank-in-action to see fully worked-out examples.

All you have to do now is write the JavaScript function. I have intentionally created a complex example to show that injection is up to nearly any task. Let's look at the JavaScript bit by bit, starting with a function to parse CSV. You need a function that will take raw text, like that shown in listing 6.1, and convert it into an array of JavaScript objects:

```
[
  {
    "Day": "4-Jun",
    "Description": "PM Thunderstorms",
    "High": 83,
```

```
    "Low": 71
    "Precip": "50%",
    "Wind": "E 5 mph",
    "Humidity": "65%"
  },
  ...
]
```

You will call the function shown in the following listing `csvToObjects`.

Listing 6.3. A JavaScript function to parse CSV data—`csvToObjects`

```
function csvToObjects (csvData) {                    1
  var lines = csvData.split('\n'),                   2
      headers = lines[0].split(','),
      result = [];


  // Remove the headers
  lines.shift();                                     3

  lines.forEach(function (line) {
    var fields = line.split(','),                    4
        row = {};

    for (var i = 0; i < headers.length; i++) {
      var header = headers[i],
          data = fields[i];
      row[header] = data;                            5
    }

    result.push(row);                                6
  });

  return result;
}
```

- ***1*** **csvData is the raw text.**
- ***2*** **Splits input by line endings**
- ***3*** **Removes first line (headers)**
- ***4*** **Splits line by commas**
- ***5*** **Adds data keyed by header**
- ***6*** **Adds to array**

As far as CSV parsing functions go, this is as simple as it gets. It works for your data, but not for more complex data involving escaped commas inside quotes and other edge

scenarios.

The next function you will need is one to look for three consecutive days of humidity over 60%, as shown in the following listing.

Listing 6.4. Looking for three consecutive days out of range

```
function hasThreeDaysOutOfRange (humidities) {    1
  var daysOutOfRange = 0,
      result = false;

  humidities.forEach(function (humidity) {
    if (humidity < 60) {
      daysOutOfRange = 0;                          2
      return;
    }

    daysOutOfRange += 1;
    if (daysOutOfRange >= 3) {                     3
      result = true;
    }
  });

  return result;
}
```

- ***1* Accepts an array of integers representing humidity level**
- ***2* Resets counter if humidity is in range**
- ***3* Sets result if humidity is out of range for three days**

Detecting three consecutive days of out-of-range humidity is complicated enough to extract into a separate function, but it isn't too difficult to code. The last check—looking for a single day more than 10% out of range—is simple enough that you can do it inline using the JavaScript `some` function of arrays, which returns `true` if the supplied function is true for any element of the array. The predicate function looks like the following listing.

Listing 6.5. A predicate to test for excess humidity levels

```
function (request) {                                  1
  function csvToObjects (csvData) { ... }             2

  function hasThreeDaysOutOfRange (humidities) { ... }  3

  var rows = csvToObjects(request.body),
      humidities = rows.map(function (row) {          4
```

```
          return parseInt(row.Humidity.replace('%', '')); 4
        }),                                               4
        hasDayTenPercentOutOfRange = humidities.some(     5
          function (humidity) { return humidity >= 70; }  5
        );                                                5

    return hasDayTenPercentOutOfRange ||                  6
           hasThreeDaysOutOfRange(humidities);            6
  }
```

- ***1*** **Passes the request object in**
- ***2*** **See listing 6.3.**
- ***3*** **See listing 6.4.**
- ***4*** **Converts the CSV rows to a list of humidity integers**
- ***5*** **Converts the CSV rows to a list of humidity integers**
- ***6*** **Matches if either condition is true**

When you include an `inject` predicate in a stub, mountebank passes the entire request object to the provided function. You have included your `csvToObjects` and `hasThreeDaysOutOfRange` functions as subfunctions inside the parent predicate function. You can use this approach to include code of considerable complexity.

Adding this predicate allows you to mimic the roadie service effectively, virtualizing its behavior with high fidelity. Although that highlights the power of JavaScript injection, it also raises an important concern about service virtualization.

Virtualizing the roadie service has been a wonderful example to demonstrate the power of `inject`. However, it does come with two pretty serious drawbacks: it hasn't helped one bit in terms of making me an actual rock star, and it's likely the kind of thing you'd want to avoid virtualizing in a real application stack. Remember, service virtualization is a testing strategy that gives you determinism when testing a service that has runtime dependencies. It's not a way of reimplementing runtime dependencies in a different platform. Although mountebank provides advanced functionality to make your stubs smarter when you need them to be, your best bet is to not need them to be so smart. The dumber your virtual services can be, the more maintainable your test architecture will be.

6.2. CREATING YOUR OWN DYNAMIC RESPONSE

You also can create your own response in mountebank. The `inject` response joins `is`

and `proxy` to round out the core response types, and it represents a dynamic response that JavaScript generates. In its simplest form, the response injection function mirrors that for predicates, accepting the entire request as a parameter. It's responsible for returning a response object that mountebank will merge with the default response. Think of it as creating an `is` response using a JavaScript function, as follows.

Listing 6.6. The basic structure of response injection

```
{
  "responses": [{
    "inject": "function (request) { return { statusCode: 400 }; }"
  }]
}
```

In this basic form, it's quite easy to replace your predicate injection you used to virtualize the roadie service with response injection. Because you're giving the response injection function the same request as the predicate injection function, you could remove the predicate injection and move the conditions to the function that generates the response, as shown in the following listing.

Listing 6.7. A response injection function to virtualize the roadie service humidity checks

```
function (request) {                                          1
  function csvToObjects (csvData) { ... }                     2

  function hasThreeDaysOutOfRange (humidities) { ... }        3

  var rows = csvToObjects(request.body),
    humidities = rows.map(function (row) {
      return parseInt(row.Humidity.replace('%', ''));
    }),
    hasDayTenPercentOutOfRange = humidities.some(
      function (humidity) { return humidity >= 70; }
    ),
    isTooHumid = hasDayTenPercentOutOfRange ||                4
                 hasThreeDaysOutOfRange(humidities);

  if (isTooHumid) {
    return {                                                  5
      statusCode: 400,                                        5
      body: 'Humidity levels dangerous, action required'      5
    };                                                        5
  }
  else {
    return {                                                  6
      body: 'Humidity levels OK for the next 10 days'         6
    };                                                        6
```

```
    }
}
```

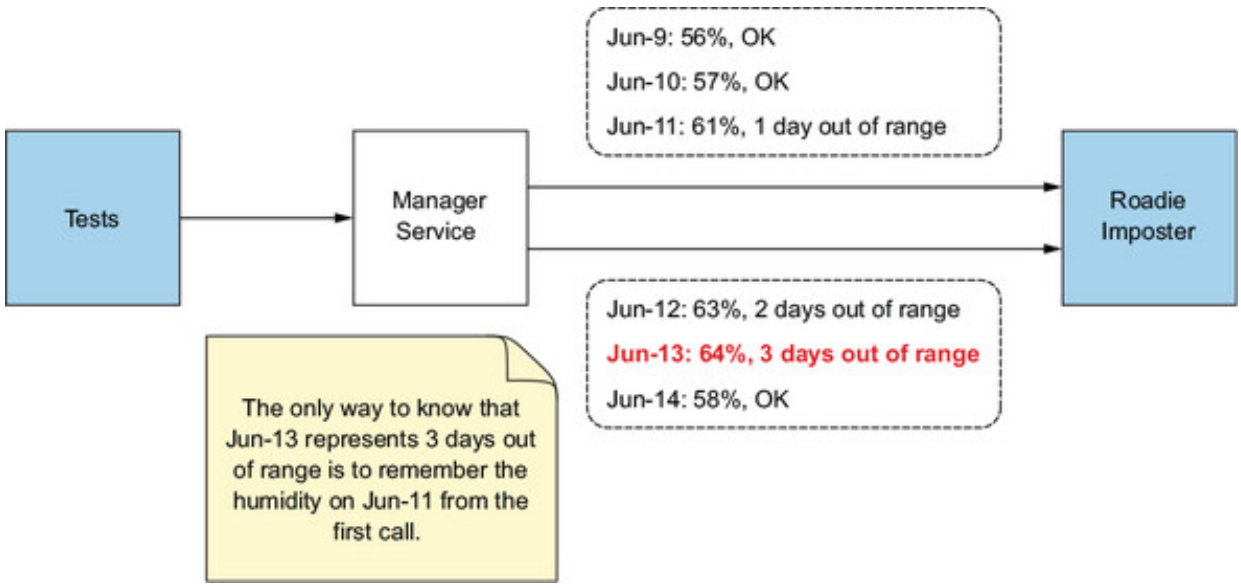
- ***1*** **Mountebank passes request in**
- ***2*** **See listing 6.3.**
- ***3*** **See listing 6.4.**
- ***4*** **Capture condition**
- ***5*** **Return failure response**
- ***6*** **Return happy path response**

Instead of returning `true` or `false` to determine whether a predicate matches, a response injection returns the response object, or at least the portion of it that isn't the default response. In this case, it returns a 400 status code and a body indicating action is required if the humidity check requires you to take action, or a default 200 code with a body letting you know that the humidity levels are OK.

6.2.1. Adding state

At first glance, there isn't much difference between using a predicate injection and using a response injection in this example. But for complex workflows, response injections have a key advantage: they can keep state. To see how that can be useful, imagine having to virtualize a scenario where your manager service sends *multiple* weather reports, and the roadie service needs to detect three consecutive days out of range even if they span two reports that the manager sent (figure 6.3).

Figure 6.3. Two reports need to be spanned to detect dangerous humidity.



Mountebank passes a `state` parameter into your response injection functions that you

can use to remember information across multiple requests. It's initially an empty object, but you can add whatever information you want to it each time the injection function executes. In this example, you will have to save the humidity results by day so you can detect dangerous humidity levels even if three consecutive days over 60% humidity span two requests from the manager service.

You start by adding the parameter to your function and initializing it with the variables you want to remember. In this case, `state` will remember the days, and if the roadie service sees the weather report for a day it hasn't seen yet, your function will add the humidity to a list:

```
function (request, state) {
  if (!state.humidities) {
    state.days = [];
    state.humidities = [];
  }

  ...
}
```

Now the rest of the function is nearly identical to listing 6.7. You only have to add to the `state.humidities` array at the appropriate time and do your checks on that array instead of a local variable, as in the following listing.

Listing 6.8. Remembering state between responses

```
function (request, state) {
  function csvToObjects (csvData) {...}

  function hasThreeDaysOutOfRange (humidities) {...}

  // Initialize state arrays
  if (!state.humidities) {
    state.days = [];
    state.humidities = [];
  }

  var rows = csvToObjects(request.body);
  rows.forEach(function (row) {
    if (state.days.indexOf(row.Day) < 0) {                    1
      state.days.push(row.Day);
      state.humidities.push(row.Humidity.replace('%', ''));   2
    }
  });

  var hasDayTenPercentOutOfRange =
    state.humidities.some(function (humidity) {               3
```

```
        return humidity >= 70;
    });

    if (hasDayTenPercentOutOfRange ||
      hasThreeDaysOutOfRange(state.humidities)) {
      return {
        statusCode: 400,
        body: 'Humidity levels dangerous, action required'
      };
    }
    else {
      return {
        body: 'Humidity levels OK'
      };
    }
  }
```

- ***1*** **Only adds to the list if it hasn't seen this day before**
- ***2*** **Adds new humidity**
- ***3*** **Switches these functions to use state variable**

Voilà! Now the virtual roadie service can keep track of humidity levels across multiple requests. There is only one feature of injection left to look at, but it is a big topic: asynchronous operations.

6.2.2. Adding async

Asynchronicity is baked into JavaScript, to the extent that it is generally required to access any file or network resource used to craft a dynamic response. Understanding why requires a quick tour of how programming languages manage I/O, as JavaScript is fairly unusual in this regard. Until Microsoft introduced the `XMLHttpRequest` object that powers AJAX requests, JavaScript lacked any form of I/O found in the base class libraries of other languages. It took Node.js to add a full complement of I/O functions to JavaScript, but it did so following the AJAX pattern familiar to a generation of web developers: using callbacks.

Take a look at the following code to sort the lines in a file. This is in Ruby, but the code would be similar in Python, Java, C#, and most traditional languages.

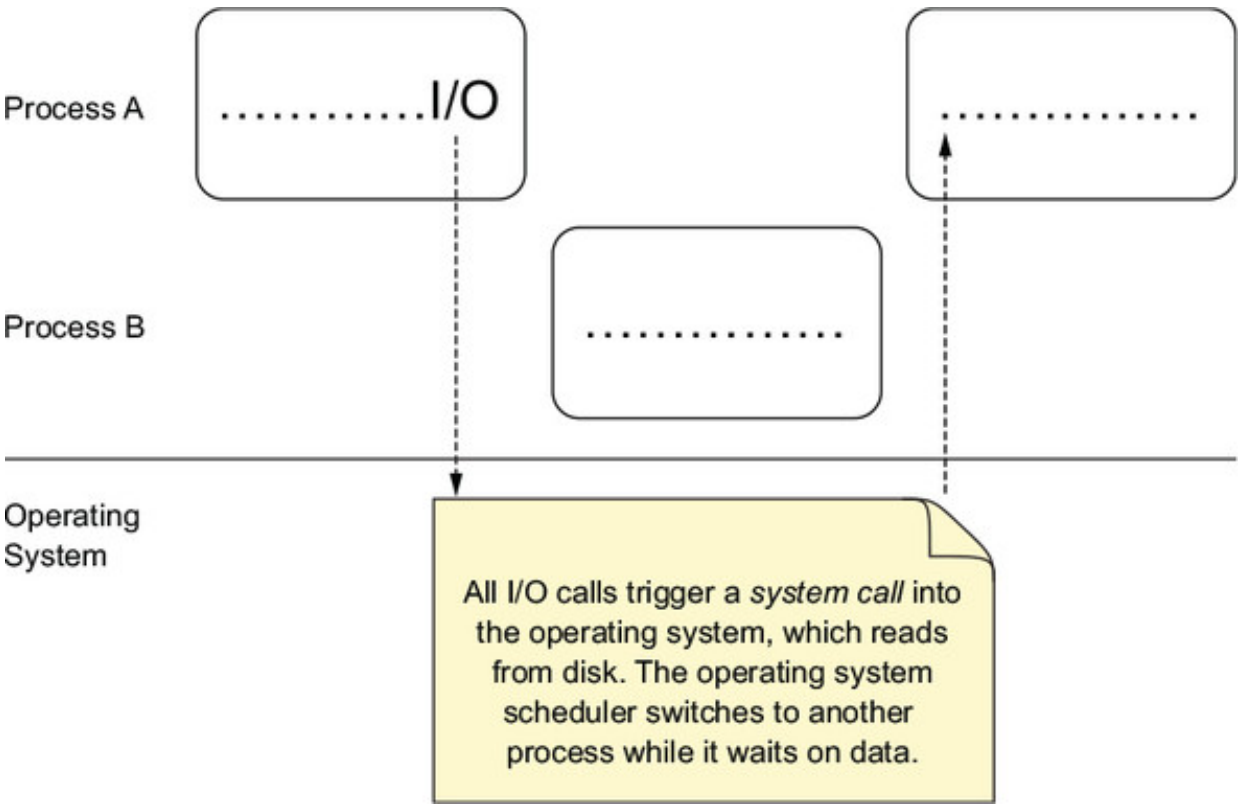
Listing 6.9. Using traditional I/O to sort lines in a file

```
lines = File.readlines('input.txt')
puts lines.sort
puts "The end..."
```

First you read all the lines in the input.txt file into an array, then you sort the array, printing the output to the console. Finally, you print "The end..." to the console. At first blush, nothing could be simpler, but the `File.readlines` function is hiding considerable complexity.

As shown in figure 6.4, under the hood, Ruby has to make a system call into the operating system (OS), because only the OS has the appropriate privileges to interact with the hardware, including the disk storing input.txt. To buy time while it waits on the results, the OS scheduler switches to another process to execute for a period of time. When the data from the disk is available, the OS feeds it back into the original process. Computers move fast enough that this is largely transparent to the user; for most I/O operations, the application will still feel quite responsive. It's also transparent to the developer, as the linear nature of the code matches a mental model, which is why *blocking I/O*—having the process block until the operation completes—is the most common form of I/O.

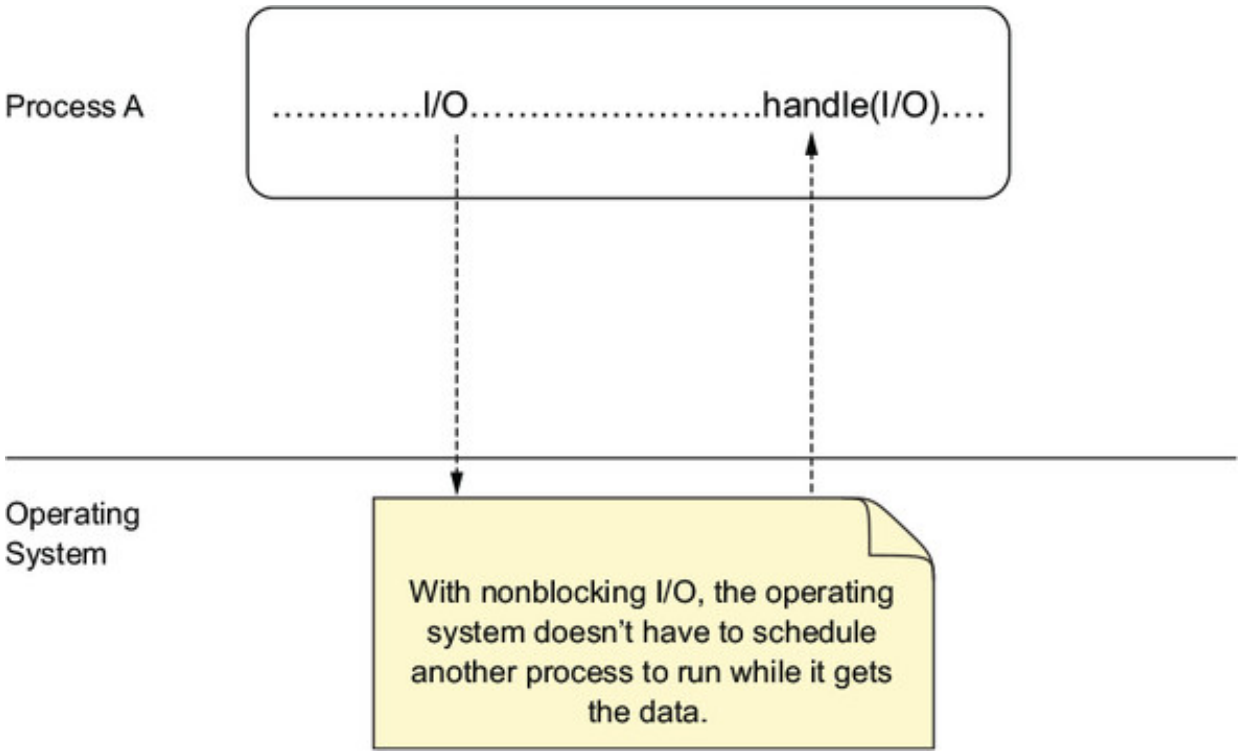
Figure 6.4. What happens with traditional blocking I/O



JavaScript was born of the web, a programming environment rich with events, such as responding when a user presses a button or types in a text field. AJAX, which made web pages more responsive by allowing the user to fetch data from the server without refreshing the entire page (a form of I/O involving the network), maintained that event model, treating getting a response from the server as an event. When Ryan Dahl wrote Node.js to add more I/O capability to JavaScript, he intentionally maintained that

event model because he wanted to explore *nonblocking I/O* in a mainstream language. The fact that developers were already used to AJAX events made JavaScript a natural fit (figure 6.5).

Figure 6.5. Nonblocking I/O doesn't block the process.



Each I/O operation registers a callback function that executes when the OS has data, and program execution continues immediately to the next line of code. Let's rewrite the Ruby file sort operation in JavaScript, using nonblocking I/O, as in the following listing.

Listing 6.10. File sort using nonblocking I/O

```
var fs = require('fs');                                    1

fs.readFile('input.txt', function (err, data) {           2
  var lines = data.toString().split('\n');
  console.log(lines.sort());
});

console.log('The end...');                                 3
```

- ***1* Built-in node library for filesystem operations**
- ***2* Ignores error handling**
- ***3* Execution continues before the file is actually read**

The callback function is passed as a parameter to `fs.readFile`, and execution

immediately continues to the next line of code. At some later point, when the OS has the contents of input.txt, the callback function will execute, providing the sorted lines. That flow means that, in this example, you will print "The end..." to the console *before* you print the sorted lines of input.txt.

As of this writing, predicate injection doesn't support asynchronous operations. But async support *is* important when it comes to response injection, as I/O operations are often valuable in scripting dynamic responses. Let's show an example by virtualizing a simple OAuth flow in mountebank.

OAuth

OAuth is a delegated authorization framework. It allows one party (the resource owner) to allow another party (the client) to act on a resource held by a third party (the resource server), where the identity of the resource owner is guaranteed by a fourth party (the authorization server). You can often collapse these four roles in various ways, creating a number of alternative flows.

A standard use case is where a person (the resource owner) allows a web app (the client) to act on a resource held by a third party like GitHub (resource server) after presenting credentials to that third party (authorization server). This flow allows the web app to perform secure operations in GitHub on behalf of the user, even though the user never provided their GitHub credentials to that website.

This OAuth flow is common, and its mechanics are difficult to stub out. In the section "Virtualizing an OAuth-backed GitHub client," I use that as an opportunity to show how to manage async in an `inject` response by building a small GitHub web app and virtualizing the GitHub API for testing purposes.

Virtualizing an OAuth-backed GitHub client

GitHub has a robust marketplace of client applications available at https://github.com/marketplace. Unfortunately, none of them solve an immediate problem of the readers of this book: adding a star to the mountebank repo.[3] But GitHub has a public RESTful API that allows you to build an app. You will treat that app as your system under test, requiring you to virtualize the GitHub API itself.

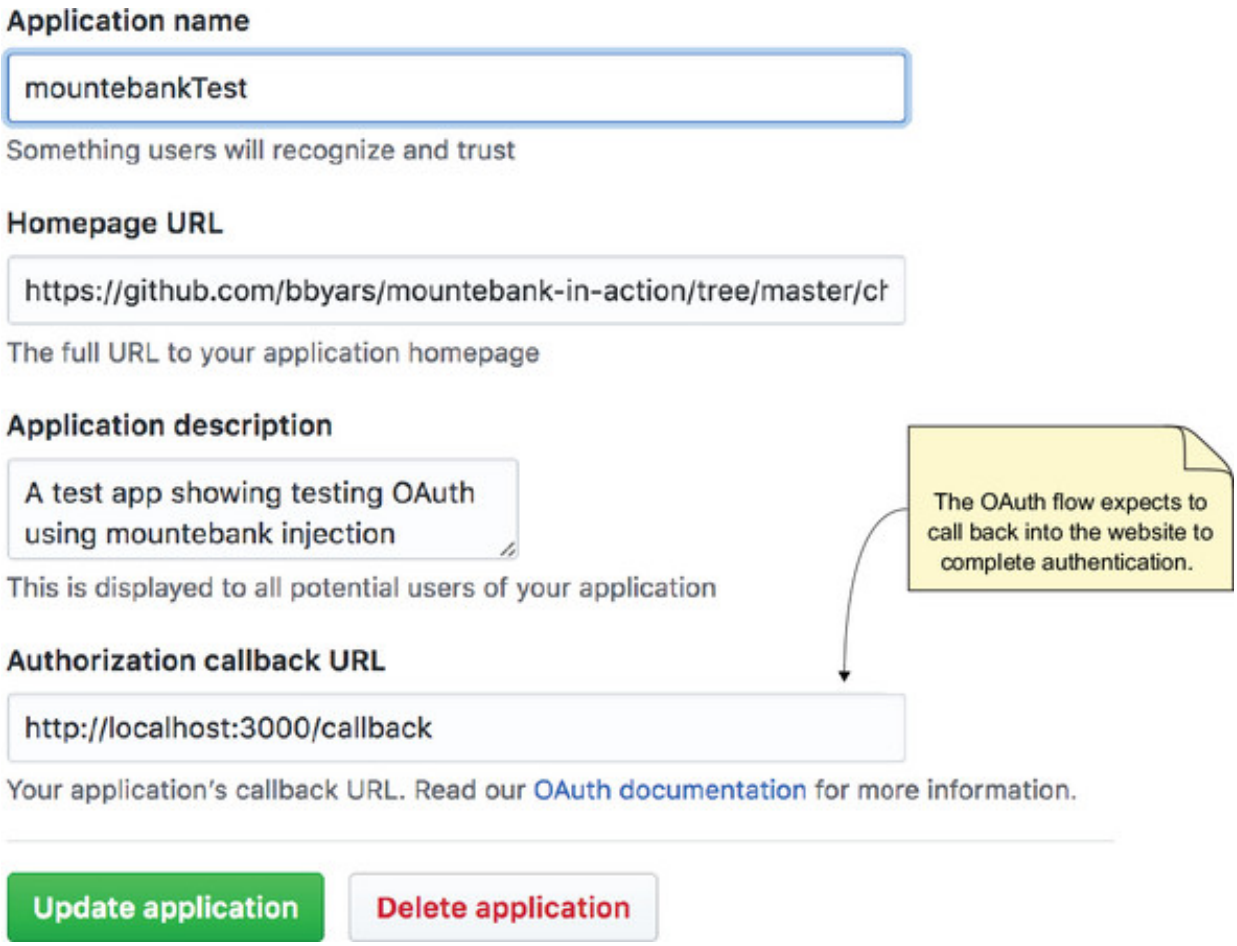
3

For readers willing to star the old-fashioned way, you can do so by visiting the repo directly at https://github.com/bbyars/mountebank.

GitHub uses OAuth, which requires a complex set of interactions before GitHub will accept an API call to star a repo.[4] The first thing you need to do is to register your new application, which you can do at https://github.com/settings/applications/new (figure 6.6).

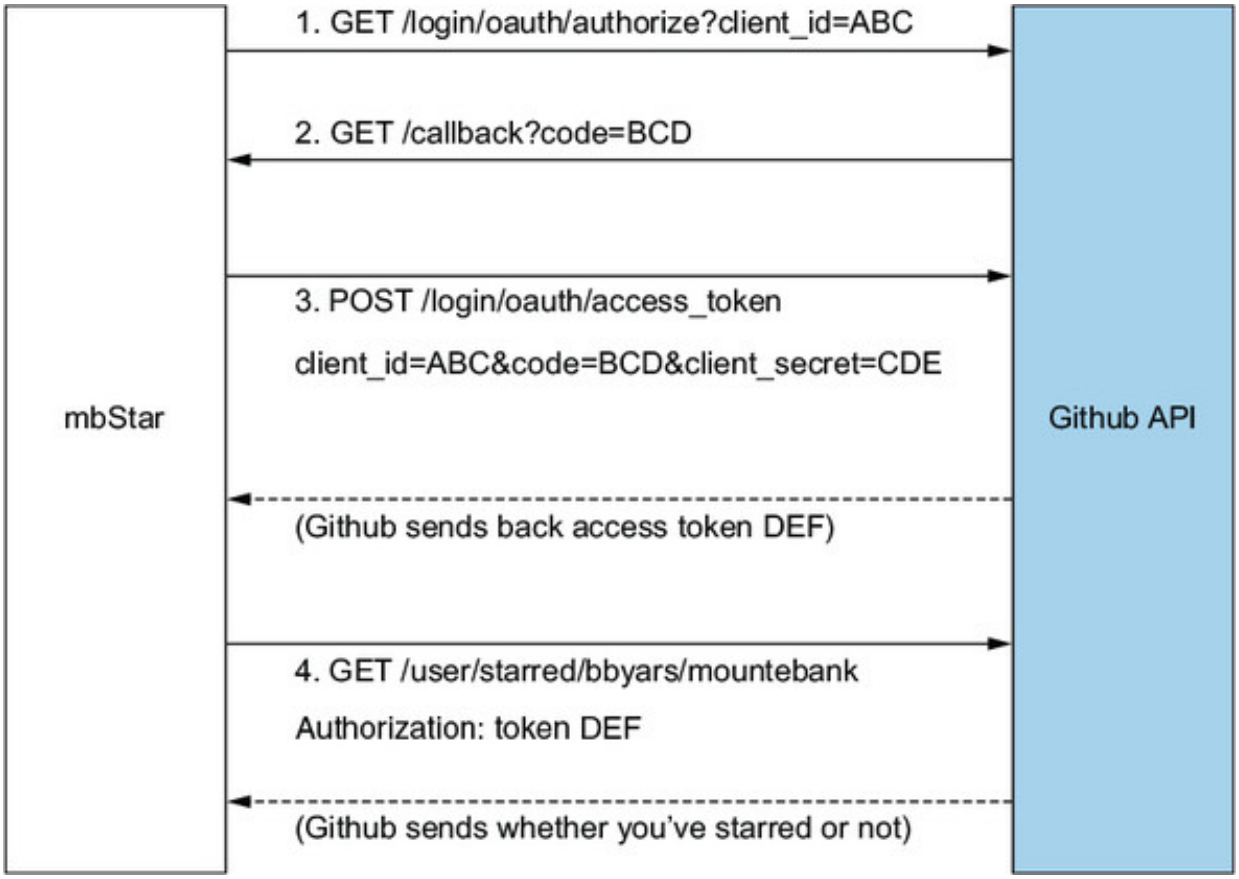
4 We will use a basic OAuth web flow described at https://developer.github.com/v3/guides/basics-of-authentication/.

Figure 6.6. Registering a GitHub application



The OAuth flow expects to call back to complete the authentication and uses the URL you provided during registration to call back into. The general flow is shown in figure 6.7.

Figure 6.7. Understanding the GitHub OAuth flow



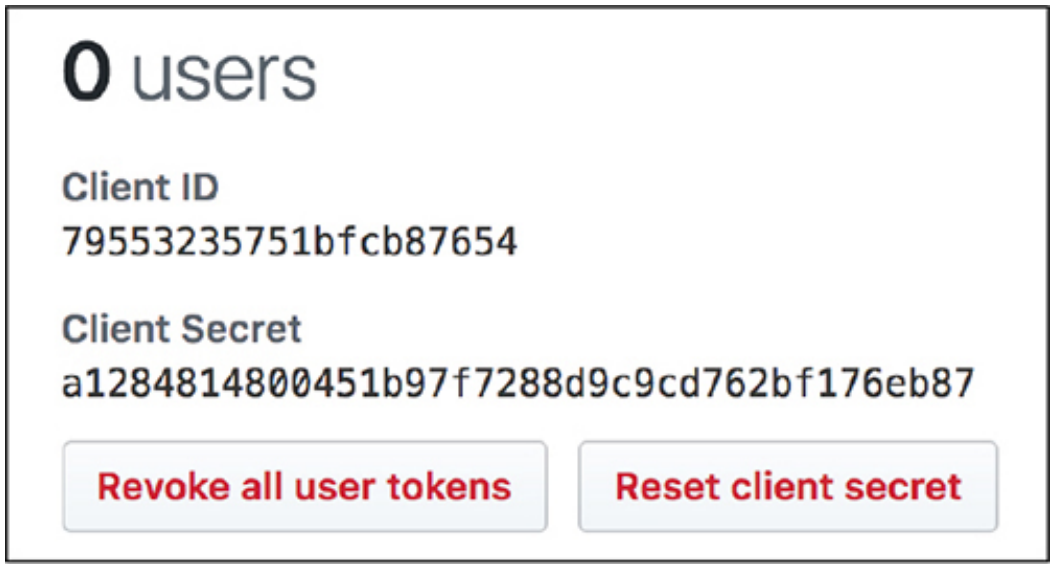
The application calls the /login/oauth/authorize endpoint, passing a `client_id`. GitHub calls the callback URL you provided during registration, passing a random `code`. The application is then expected to call a /login/oauth/access_token URL, sending the `client_id`, the `code` and a `client_secret`. If all of that is done correctly, GitHub sends back a token that the application can use to authorize subsequent calls. Although the fictional app is designed to star the mountebank repo if you haven't already done so, I will only show how to test whether you have already starred or not. (On the advice of my legal staff, I have decided to leave starring mountebank as an exercise for the reader.)

The `client_id` and `client_secret` are provided during registration on GitHub (figure 6.8). Like a private key, you should keep the `client_secret` super secret. You shouldn't store it in source control, and you most certainly shouldn't publish it in a book for millions to read.[5]

5

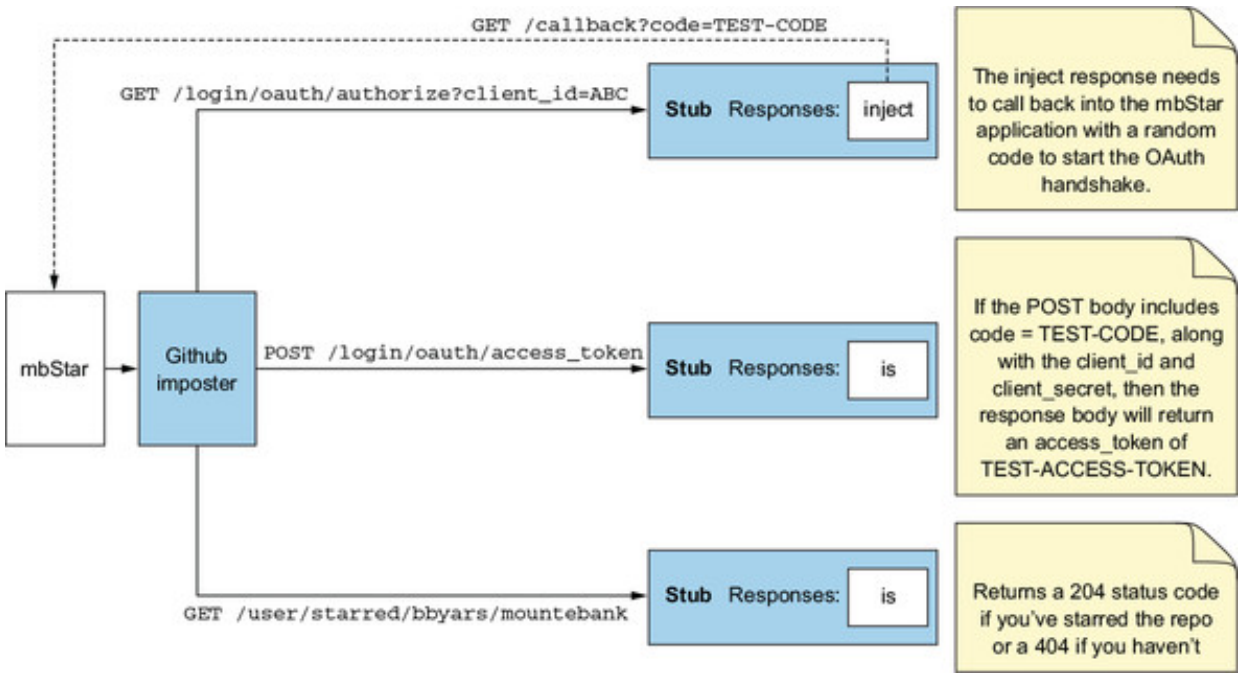
Don't worry—I removed this toy app before you had a chance to read these words. You'll have to register your own app to fully follow the source code in question, although you can use the same code located at https://github.com/bbyars/mountebank-in-action

Figure 6.8. Viewing the client secret to communicate to GitHub



Your test case should validate this entire flow, requiring you to virtualize three GitHub endpoints. Structurally, the GitHub imposter needs to have three stubs representing those endpoints (figure 6.9). You can virtualize the last two calls in order to get the token and to check if you have starred the mountebank repo, with simple `is` responses. You can't virtualize the first call, to actually authorize, with an `is` response. in order to authorize. The fact that it has to call back into the system under test requires you to move beyond simple stubbing approaches. You'll use an `inject` response to do the callback with an easily identifiable test code.

Figure 6.9. The three stubs needed to virtualize this GitHub workflow



Using easily identifiable names (like TEST-CODE and TEST-ACCESS-TOKEN) for the test data that the imposter creates is a useful tip to make it easier to spot in a complex workflow.

The first endpoint (/login/oauth/authorize) starts the OAuth handshake by sending the random code (TEST-CODE) to your web app. It's also the most complicated response, involving calling back to the system under test, which you cannot solve using an `is` or a `proxy` response. Conceptually, the stub looks like the following listing.

Listing 6.11. Stub using response injection to make OAuth callback

```
{
  "predicates": [{
    "equals": {
      "method": "GET",
      "path": "/login/oauth/authorize",
      "query": {
        "client_id": "<%= process.env.GH_CLIENT_ID %>"          1
      }
    }
  }],
  "responses": [{
    "comment": "This sends back a code of TEST-CODE",           2
    "inject": "<%- stringify(filename, 'auth.js') %>"           3
  }]
}
```

- ***1* Adds environment variable**
- ***2* Mountebank ignores this line.**
- ***3* Brings in injection function**

Both the tests and the example web app use environment variables for the `client_id` and the `client_secret`, and you will use EJS templating to interpolate them into your configuration file. Notice also the added `comment` field in the response. Mountebank ignores any fields it doesn't recognize, so you can always add more metadata. For complicated workflows like this one, such comments can help you follow along more easily.

The following listing shows the injection function in `auth.js`.

Listing 6.12. Injection function to make OAuth callback

```
function (request, state, logger, callback) {
  var http = require('http'),
    options = {
      method: 'GET',
      hostname: 'localhost',
      port: 3000,
      path: '/callback?code=TEST-CODE'                          1
```

```
    },
    httpRequest = http.request(options, function (response) {
      var body = '';                                          2
      response.setEncoding('utf8');                           2
      response.on('data', function (chunk) {                  2
        body += chunk;                                        2
      });                                                     2
      response.on('end', function () {
        callback({ body: body });                             3
      });
    });

  httpRequest.end();                                          4
}
```

- ***1*** **Callback with TEST-CODE**
- ***2*** **Node.js code to collect the response body**
- ***3*** **Asynchronously returns the response**
- ***4*** **Sends the request and returns from the function**

Much of the code is manipulating the Node.js `http` module to make the call to http://localhost:3000/callback?code=TEST-CODE. Because the HTTP call involves network I/O, the function returns immediately after the call to `httpRequest.end()`. When the network call returns, Node.js invokes the function passed as a parameter to the `http.request()` call. Node's `http` library streams the HTTP response back, so you may receive multiple `data` events and have to collect the response body as you go. When you have received the entire response, Node triggers the `end` event, at which point you can create the response you want. In your case, you'll return the same body the callback URL provided. Passing that to the `callback` function parameter ends your response injection, returning the parameter as the response to mountebank.

For example, if your call to http://localhost:3000/callback?code=TEST-CODE returns a body of "You have already starred the mountebank repo," then the end result of the injection function would be equivalent to the following `is` response:

```
{
  "is": {
    "body": "You have already starred the mountebank repo"
  }
}
```

Remember, a key goal of mountebank is to make simple things easy to do and to make

hard things possible. Virtualizing an OAuth flow is hard. This is about as complicated a workflow as you will see in most tests involving service virtualization. It's certainly a lot to walk through, but you were able to do the hard bits of it with about 20 lines of JavaScript, and when you run into similar problems, you will be thankful that solving them is at least possible with mountebank.

Validating the OAuth authorization

Let's look at the next stub, for the /login/oauth/access_token endpoint. This one should only match if the app reflected back the `TEST-CODE` in its request body and correctly sent the preconfigured `client_id` and `client_secret`. You can use predicates and a simple `is` response to send back a test access token, as shown in the following listing.

Listing 6.13. The stub to get an access token

```
{
  "predicates": [
    {
      "equals": {
        "method": "POST",
        "path": "/login/oauth/access_token"
      }
    },
    {
      "contains": {
        "body":
➥ "client_id=<%= process.env.GH_CLIENT_ID %>"                    1
      }
    },
    {
      "contains": {
        "body":
➥ "client_secret=<%= process.env.GH_CLIENT_SECRET %>"            2
      }
    },
    {
      "contains": {
        "body": "code=TEST-CODE"                                  3
      }
    }
  ],
  "responses": [{
    "is": {
      "body": {
        "access_token": "TEST-ACCESS-TOKEN",                      4
        "token_type": "bearer",
        "scope": "user:email"
      }
```

```
      }
    }]
  }
```

- ***1*** **Requires client_id from environment**
- ***2*** **Requires client_secret from environment**
- ***3*** **TEST-CODE comes from injection in previous stub**
- ***4*** **Returns a test access token**

Once your web app retrieves the `access_token`, it has successfully navigated the OAuth flow.

Checking if you've starred mountebank

At this point, the app should be armed with an access token and should make the GitHub call to check if you've starred the mountebank repo or not. Your predicate needs to validate the token, and, once again, you can use a simple `is` response, as shown in the following listing.

Listing 6.14. Stub to check if you've starred the mountebank repo

```
{
  "predicates": [{
    "equals": {
      "method": "GET",
      "path": "/user/starred/bbyars/mountebank",
      "headers": {
        "Authorization": "token TEST-ACCESS-TOKEN"      1
      }
    }
  }],
  "responses": [{
    "comment": "204=yes, 404=no",                       2
    "is": { "statusCode": 404 }                         3
  }]
}
```

- ***1*** **Validates token**
- ***2*** **Mountebank ignores this line.**
- ***3*** **Indicates that user hasn't starred repo**

With OAuth now virtualized, any other API endpoints should be quite easy to stub out. All you have to do is check for the `Authorization` header as shown in listing 6.14.

6.2.3. Deciding between response vs. predicate injection

Mountebank passes the `request` object to both predicate and response injection functions, so you could put conditional logic based on the request in either location. Compared to response injection, predicate injection is relatively easy to use. If you need to send a static response back based on a dynamic condition, programming your own predicate and using an `is` response stays true to the intention of predicates in mountebank. But response injection is considerably more capable, and you won't hurt my feelings if you move your conditional logic to a response function so you can take advantage of state or async support.

A predicate injection function takes only two parameters:

- **`request`**— The protocol-specific request object (HTTP in all of the examples)
- **`logger`**— Used to write debugging information to the mountebank logs

The response injection function includes those parameters and adds two more:

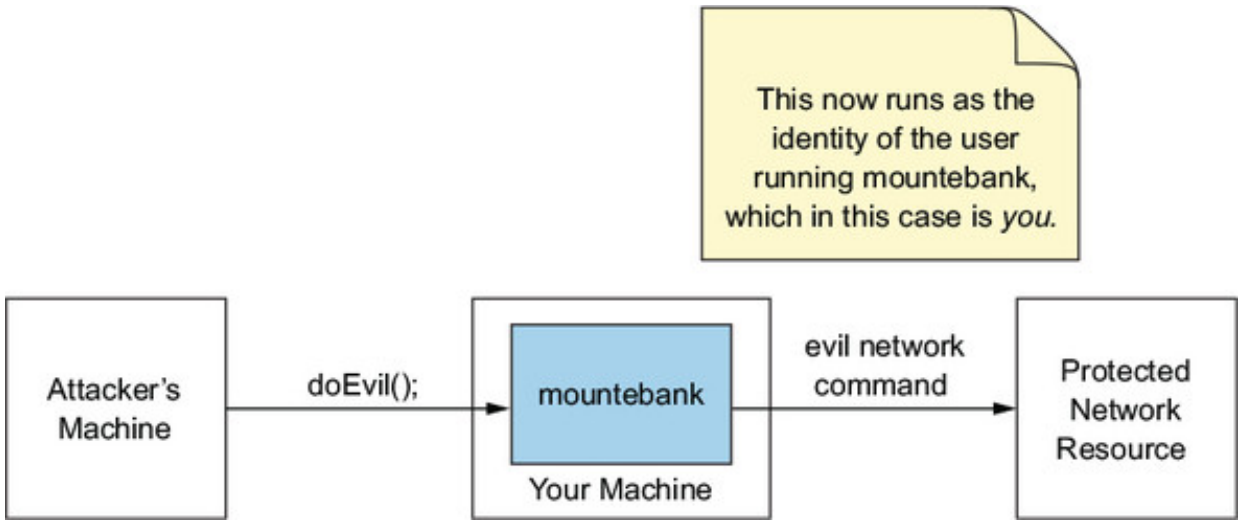
- **`state`**— An initially empty object that you can add state to; will be passed into subsequent calls for the same imposter
- **`callback`**— A callback function to support asynchronous operations

Response functions always have the option of synchronously returning a response object. You only need the `callback` if you use nonblocking I/O and need to return the response object asynchronously.

6.3. A WORD OF CAUTION: SECURITY MATTERS

JavaScript injection is disabled by default when you start mountebank, and for good reason. With injection enabled, mountebank becomes a potential remote execution engine accessible to anyone on the network. When mountebank is run naively, an attacker can take advantage of that fact to do evil things on the network while spoofing your identity (figure 6.10).

Figure 6.10. Using mountebank to spoof your identity during a network attack



Injection is an enormously useful feature, but you have to use it with an understanding of the security implications. This is why mountebank shows a warning message in the logs every time you start it with the `--allowInjection` flag. You can take some precautions to protect yourself.

The first precaution is to not run `mb` under your user account. Starting mountebank as an unprivileged user, ideally one without domain credentials on your network, goes a long way toward protecting yourself. You should always use the least privileged user you can get away with, adding in network access only when your tests require it.

The next layer of security is to restrict which machines can access the mountebank web server. We've used the `--localOnly` flag throughout this chapter, which restricts access to processes running on the same machine. This option is perfect when your tests run on the same machine as mountebank, and it should be the default choice most of the time. When you do require remote tests (during extensive load testing, for example), you can still restrict which machines can access mountebank's web server with the `--ipWhitelist` flag, which captures a pipe-delimited set of IP addresses. For example:

```
mb --allowInjection --ipWhitelist "10.22.57.137|10.22.57.138"
```

In this example, the only remote IP addresses allowed access to mountebank are 10.22.57.137 and 10.22.57.138.

6.4. DEBUGGING TIPS

Writing injection functions has all the same complexity as writing any code, except that it's much harder to debug them through an IDE because they run in a remote process. I often resort to what, back in my college days when we coded in C, we called `printf`

debugging. In JavaScript, it looks something like this:

```
function (request) {
  // Function definition...

  var rows = csvToObjects(request.body),
    humidities = rows.map(function (row) {
      return parseInt(row.Humidity.replace('%', ''));
    });

console.log(JSON.stringify(humidities));        1

  return {};                                    2
}
```

- ***1* Shows full object structure**
- ***2* I'll figure this out later....**

The `console.log` function in JavaScript prints the parameter to the standard output of the running process, which in this case is `mb`. The `JSON.stringify` function converts an object to a JSON string, allowing you to inspect the full object graph. This code is quite ugly—I outdented the `console.log` function to not lose sight of it, and I returned an empty object for the response, relying on the standard default response. If you've written code for longer than a few seconds, you'll likely recognize the pattern. Most code starts out ugly before you figure out how to communicate your intent to a ruthlessly precise computer.

To make the output a little easier to spot in the logs, mountebank passes another parameter to both predicate and response injection: the logger itself. In typical logging fashion, the logger has four functions: `debug`, `info`, `warn`, and `error`. The `debug` messages are usually not shown to the console (unless you start `mb` with the `--loglevel debug` flag). To make your debugging messages stand out, use the `warn` or `error` functions, which will print your debugging output to the console in a different color:

```
function (request, state, logger) {
  // Function definition...

  var rows = csvToObjects(request.body),
    humidities = rows.map(function (row) {
      return parseInt(row.Humidity.replace('%', ''));
    });

logger.warn(JSON.stringify(humidities));          1
```

```
    return {};
}
```

- ***1* Prints to the console as yellow text**

The code above shows logging for response injection. Predicate injections pass in the `logger` as the second parameter as well.

SUMMARY

- You can create your own predicates with a JavaScript function that accepts the `request` object and returns a Boolean representing whether the predicate matched or not.
- You also can create your own response with a JavaScript function that accepts the `request` object and returns an object representing the response.
- If you need to remember state between requests, mountebank passes an initially empty `state` object to the response function. Any fields you set on the object will persist between calls.
- Because JavaScript and node.js use nonblocking I/O, most response functions that need to access data outside the process will have to return asynchronously. Instead of returning an object, you can pass the response object to the `callback` parameter.
- Injection is powerful, but it also creates a remote execution engine running on your machine. Anytime you run mountebank with injection enabled, you should limit remote connections and use an unprivileged identity.

Chapter 7. Adding behaviors

This chapter covers

- Programmatically postprocessing a response
- Adding latency to a response
- Repeating a response multiple times
- Copying input from the request into the response
- Looking up data from a CSV file to plug into a response

The basic `is` response is easy to understand but limited in functionality. Proxy responses provide high-fidelity mimicry, but each saved response represents a snapshot in time. Response injection provides significant flexibility but comes with high complexity. Sometimes, you want the simplicity of `is` and `proxy` responses with a touch of dynamism, all without the complexity of `inject`.

Software engineers who hail from the object-oriented school of thought use the term *decorate* to mean intercepting a plain message and augmenting it in some way before forwarding it on to the recipient. It's like what the postal service does when it applies a postmark to your letter after sorting it. The original letter you sent is still intact, but the postal workers have *decorated* it to postprocess it with a bit of dynamic information. In mountebank, *behaviors* represent a way of decorating responses before the imposter sends them over the wire.

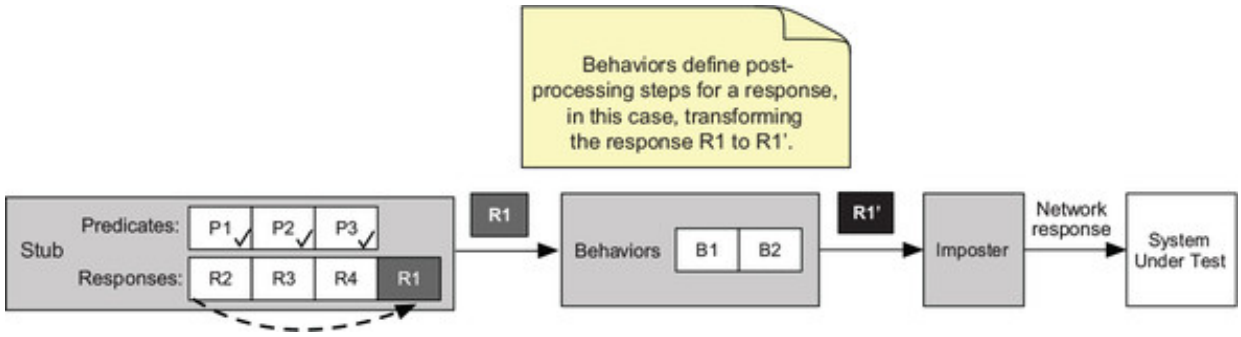
Because of their flexibility and utility, behaviors also represent one of the most rapidly evolving parts of mountebank itself. We'll look at all of the behaviors available as of this writing (representing v1.13), but expect more to come in the future.

7.1. UNDERSTANDING BEHAVIORS

If you ignore the complexity of interacting with the network and different protocols, mountebank has only three core concepts:

- *Predicates* help route requests on the way in.
- *Responses* generate the responses on the way out.
- *Behaviors* postprocess the responses before shipping them over the wire (figure 7.1).

Figure 7.1. Behaviors can transform a response from a stub before it goes out via the imposter.



Although nothing prevents you from using behaviors with `inject` responses, most behaviors exist to allow you to reduce the amount of complexity inherent in using JavaScript to craft the response. Behaviors are a way of avoiding the complexity of `inject` responses in favor of simpler `is` responses, while still being able to provide appropriate dynamism to your response.

Behaviors sit alongside the type of response in the stub definition, as shown in listing 7.1. You can combine multiple behaviors together, but only one of each type. No behavior should depend on the order of execution of other behaviors. That is an implementation detail subject to change without notice.

Listing 7.1. Adding behaviors to a stub definition

```
{
  "responses": [{
    "is": { "statusCode": 500 },
    "_behaviors": {                  1
      "decorate": ...,               2
      "wait": ...                    3
    }                                1
  }]
}
```

- ***1* All postprocessing steps on the is response**
- ***2* See section 7.2.**
- ***3* See section 7.3.**

In this example, the `is` response will first merge the 500 status code into the default

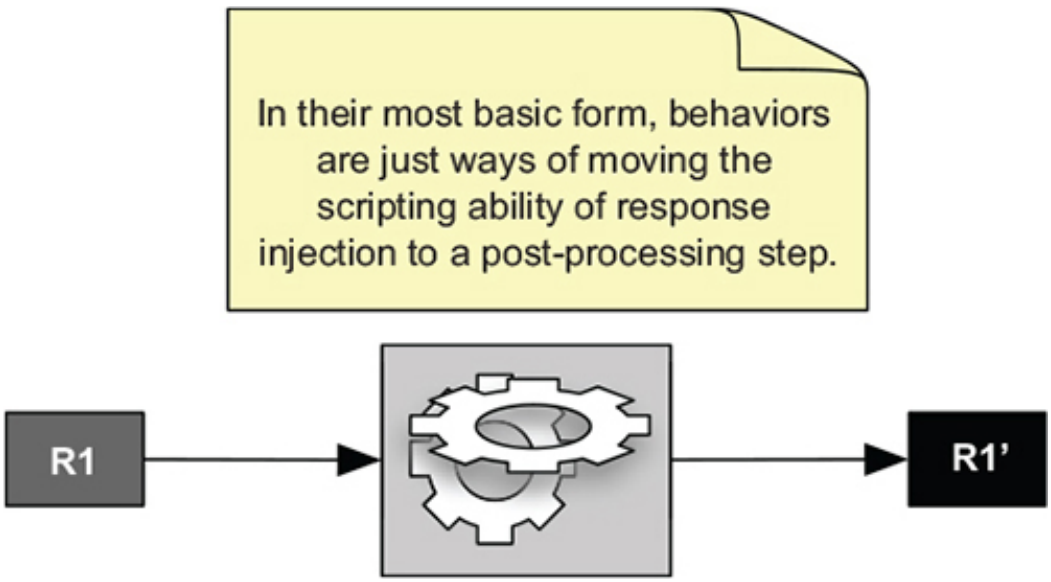
response, then it will pass the generated response object to both the `decorate` and `wait` behaviors. Each behavior will postprocess the response in a specific way, which we'll look at shortly.

Some behaviors still rely on programmatic control of the postprocessing, which requires the `--allowInjection` flag to be set when starting `mb`. This carries with it all the same security considerations we examined in the last chapter. We look at those behaviors next.

7.2. DECORATING A RESPONSE

The bluntest instruments in the behavior toolbox are the `decorate` and `shellTransform` behaviors, which accept the response object as input and transform it in some way, sending a new response object as output (figure 7.2).

Figure 7.2. Decoration allows you to postprocess the response.



They're quite similar to response injection, except they provide more focused injection (in the case of `decorate`), or more flexibility (`shellTransform`).

7.2.1. Using the decorate function

Without behaviors, you'd be forced to use an `inject` response if only one field of the response were dynamic. For example, assume you want to send the following response body back:

```
{
  "timestamp": "2017-07-22T14:49:21.485Z",
  "givenName": "Stubby",
  "surname": "McStubble",
  "birthDate": "1980-01-01"
```

```
}
```

You could capture this body in an `is` response, setting the `body` field to the JSON, but that would assume that an outdated `timestamp` is irrelevant to the test case at hand. That isn't always a valid assumption. Unfortunately, translating that to an `inject` response hides the intent, as shown in the following listing.

Listing 7.2. Using an `inject` response to send a dynamic timestamp

```
{
  "responses": [{
    "inject": "function () { return { body: { timestamp: new Date(),
➥ givenName: 'Stubby', surname: 'McStubble', birthDate:
➥ '1980-01-01' } } }"
  }]
}
```

To make sense of what the response is doing, you have to extract the JavaScript function and stare at it. Compare that to combining an `is` response with a `decorate` response, which sends the same JSON over the wire without awkward translation, as you can see in the following listing.[1]

1

As always, you can follow along with the book's source code at https://github.com/bbyars/mountebank-in-action.

Listing 7.3. Combining an `is` response with a `decorate` behavior

```
{
  "responses": [{
    "is": {                                                    1
      "body": {                                                1
        "givenName": "Stubby",                                 1
        "surname": "McStubble",                                1
        "birthDate": "1980-01-01"                              1
      }                                                        1
    },
    "_behaviors": {
      "decorate": "function (request, response, logger) {      2
➥ response.body.timestamp = new Date(); }"                    2
    }
  }]
}
```

- ***1*** **Return this…**

- **2 ...but add a current timestamp.**

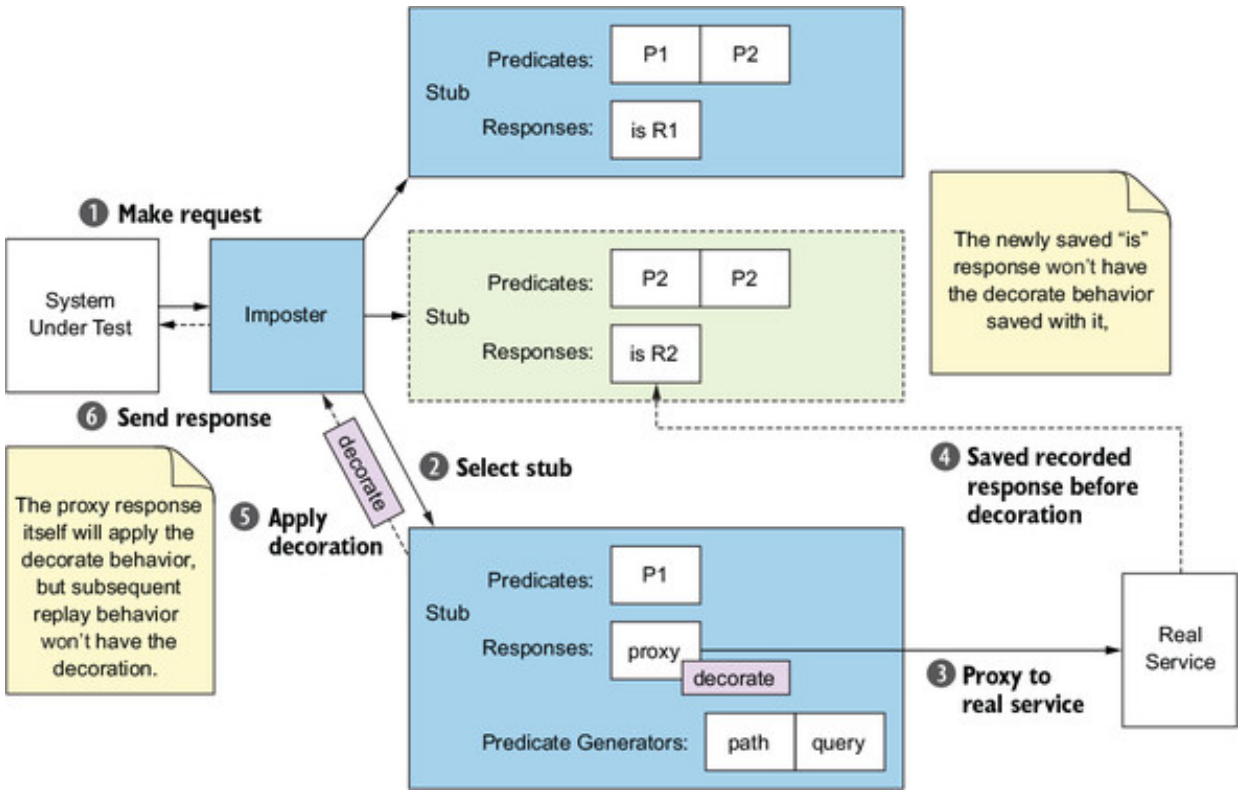
As noted, this is like the post office adding a postmark to your envelope. You provide the core content with an `is` response, and the `decorate` behavior adds the current timestamp to the message. The end response is the same as with the `inject` approach, but separating the static part of the response from the dynamic part often makes the code easier to maintain.

The `decorate` function isn't as capable as a full `inject` response. Behaviors don't have access to any user-controlled state like response injection. They don't allow asynchronous responses, which eliminates a large class of JavaScript I/O operations, as discussed in chapter 6. That said, the `decorate` behavior does allow the majority of the response message to be visible in a static `is` response, which simplifies maintenance of your test data.

7.2.2. Adding decoration to saved proxy responses

Behaviors are agnostic to the type of response they are applied to, which means you can decorate a `proxy` response as well. But by default, the decoration applies only to the `proxy` response itself, not to the `is` response it saves. (See figure 7.3.)

Figure 7.3. Behaviors applied to proxies don't transfer to the saved responses.



You *can* add a couple of behaviors to the saved `is` responses, including `decorate`. You have to configure the proxy with the `decorate` behavior. We'll work with a more

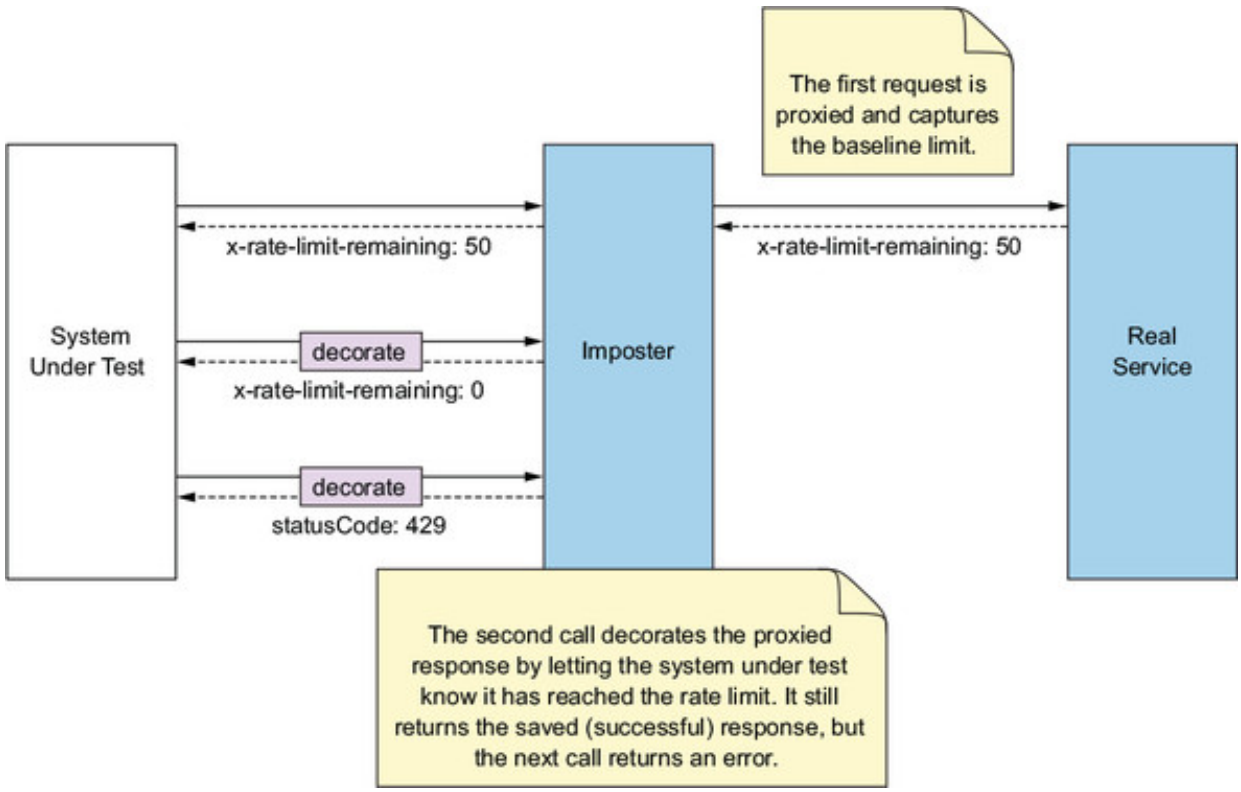
complicated example than updating a timestamp to show how proxying and decoration work hand in glove.

Most industrial APIs include some sort of rate limiting to prevent denial of service. The Twitter API represents a standard approach, where Twitter sends back an `x-rate-limit-remaining` header in the response to let the user know how many requests the API user has left for a certain time frame. Once those requests are spent, Twitter will send a 429 HTTP status code (Too Many Requests) until the time period is up.[2] At times you may want to test the consumer's response when it triggers rate limit errors midstream through a workflow. One option is to proxy all requests to the downstream rate-limited service (using the `proxyAlways` mode described in chapter 5). But it may be difficult to capture a rate limit scenario through proxying real traffic. Another option is to capture the first response and use decoration to trigger a rate limit error after a few requests (figure 7.4).

2

You can read the full details at https://dev.twitter.com/rest/public/rate-limiting.

Figure 7.4. Manufacturing a rate limit exception on a recorded response



Setting up this scenario requires you to proxy to the downstream server to save the response but add a decorate behavior on the *saved* response, as in the following listing. The original proxy response will be undecorated, returning the response captured from the downstream service.

Listing 7.4. Adding a decorate behavior to recorded responses

```
{
  "responses": [{
    "proxy": {
      "to": "http://downstream-service.com",     1
      "mode": "proxyOnce",                       2
      "addDecorateBehavior": "..."               3
    }
  }]
}
```

- ***1*** **The base URL of the downstream service**
- ***2*** **Only captures the first response**
- ***3*** **Adds a decorator to the recorded response (see listing 7.5 for further content)**

The decorator function will have access to the saved response and can change the `x-rate-limit-remaining` header or return an error as desired by the test case. In the code below, you will decrement the header 25 requests at a time to accelerate reaching a rate limit error, but you can tweak that value according to your test scenario. Because `decorate` functions don't have access to the same ability to save state that `inject` responses do, you have to use a file to store the last value sent for the `x-rate-limit-remaining` header, as shown in the following listing.

Listing 7.5. Decorator function to accelerate a rate limit exception

```
function (request, response) {
  var fs = require('fs'),                              1
    currentValue = parseInt(                           2
      response.headers['x-rate-limit-remaining']),     2
    decrement = 25;

  if (fs.existsSync('rate-limit.txt')) {               3
      currentValue = parseInt(                         3
        fs.readFileSync('rate-limit.txt'));            3
  }                                                    3

  if (currentValue <= 0) {
    response.statusCode = 429;                         4
    response.body = {                                  4
      errors: [{                                       4
        code: 88,                                      4
        message: 'Rate limit exceeded'                 4
      }]                                               4
    };                                                 4
```

```
      response.headers['x-rate-limit-remaining'] = 0; 4
    }
    else {
      fs.writeFileSync('rate-limit.txt',              5
        currentValue - decrement);                    5
      response.headers['x-rate-limit-remaining'] =    6
        currentValue - decrement;                     6
    }
  }
```

- ***1*** **Node's module for filesystem access**
- ***2*** **Gets the value in the recorded response**
- ***3*** **Gets the saved value**
- ***4*** **Sends the error response**
- ***5*** **Saves the next value**
- ***6*** **Updates the header**

What happened to nonblocking I/O?

In the last chapter, I described how JavaScript and Node.js use nonblocking I/O, which required adding asynchronous support to response injection. That is still true, but Node.js has added a small number of blocking, synchronous calls for common filesystem operations. Notice how the function names used in listing 7.5 end in `Sync` (`fs.existsSync`, `fs.readFileSync`, and `fs.writeFileSync`). These are special convenience functions that break out of the standard nonblocking I/O model. To the best of my knowledge, no such convenience functions exist for I/O that has to traverse the network.

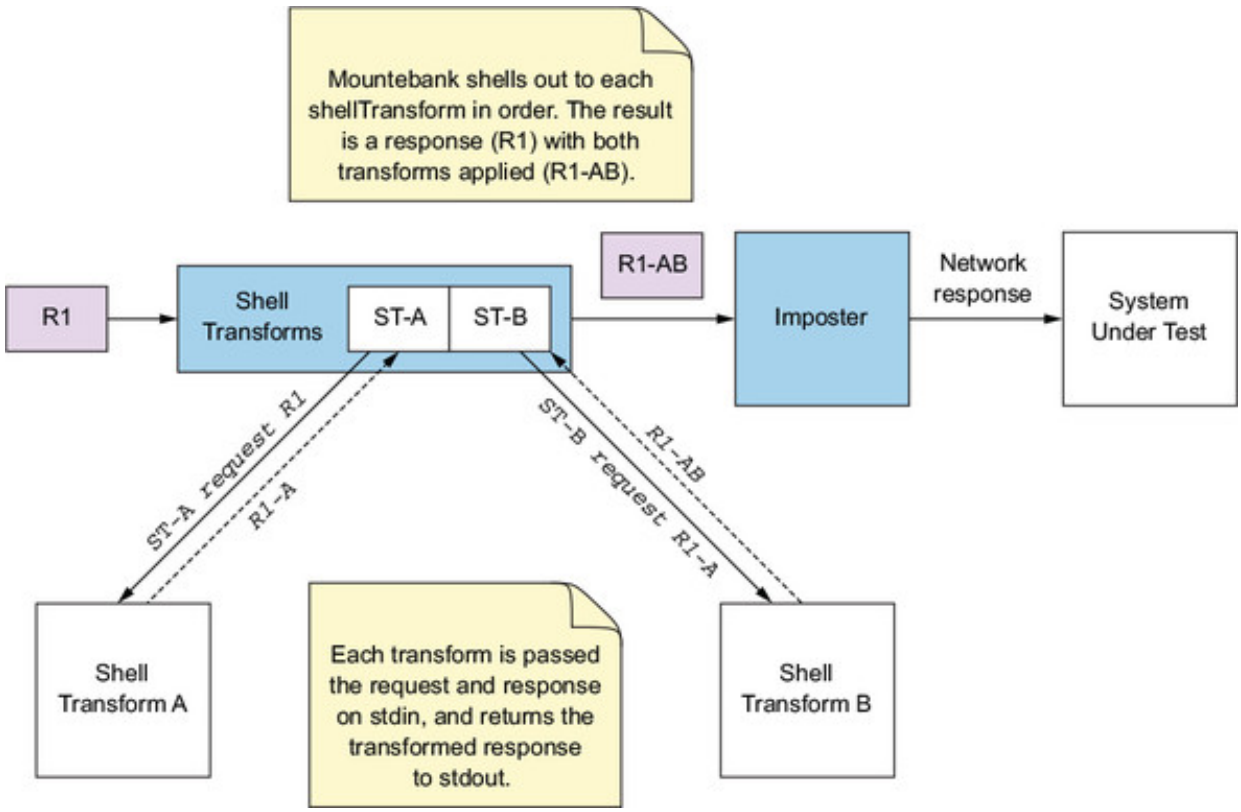
You were forced to use the filesystem to save state because decorators cannot save state directly, and you were forced to use the `Sync` functions because decorators do not support asynchronous operations. Both of these are supported in response injection, as described in chapter 6. Future versions of mountebank may support them in decorators as well. The next behavior, `shellTransform`, suffers from neither of these limitations.

7.2.3. Adding middleware through shellTransform

The next behavior is both the most general purpose and the most powerful, and both of those aspects come with added complexity. Like `decorate`, `shellTransform` allows

you programmatic postprocessing of the response. But it doesn't require the use of JavaScript, and it allows you to chain together a series of postprocessing transformations (figure 7.5).

Figure 7.5. The `shellTransform` behavior allows you to combine multiple transformations piped through the shell.



To see how it works, let's take the two transformations you've already seen (adding a timestamp and triggering a rate limit exception) and convert them to `shellTransform` behaviors. You implement each transformation as a command line application that accepts the JSON-encoded request and JSON-encoded response as parameters passed in on standard input and returns the transformed JSON-encoded response on standard output. You'll start by wiring up the imposter configuration, as shown in the following listing.

Listing 7.6. Imposter configuration for `shellTransform`

```
{
  "responses": [{
    "is": {
      "headers": {                              1
        "x-rate-limit-remaining": 3             1
      },                                        1
      "body": {                                 2
        "givenName": "Stubby",                  2
        "surname": "McStubble",                 2
        "birthDate": "1980-01-01"               2
      }                                         2
```

```
    },
    "_behaviors": {
      "shellTransform": [
        "ruby scripts/applyRateLimit.rb",        3
        "ruby scripts/addTimestamp.rb"           4
      ]
    }
  }]
}
```

- ***1* Transformed with applyRateLimit.rb**
- ***2* Transformed with addTimestamp.rb**
- ***3* This will execute first.**
- ***4* Then this will execute on the transformed response.**

In this example, you have chosen to pipe the transformations through Ruby scripts, but it could've been any language. The code in applyRateLimit.rb is a simple Ruby conversion of the code in listing 7.5. Mountebank passes two parameters on standard input—the request and the response. Here you only need the response, as shown in the following listing.

Listing 7.7. Ruby script to transform the response to trigger a rate limit error

```
require 'json'                                              1

response = JSON.parse(ARGV[1])                              2
headers = response['headers']
current_value = headers['x-rate-limit-remaining'].to_i

if File.exists?('rate-limit.txt')
  current_value = File.read('rate-limit.txt').to_i
end

if current_value <= 0
  response['statusCode'] = 429
  response['body'] = {
    'errors' => [{
      'code' => 88, 'message' => 'Rate limit exceeded'
    }]
  }
  response['headers']['x-rate-limit-remaining'] = 0
else
  File.write('rate-limit.txt', current_value - 25)
  headers['x-rate-limit-remaining'] = current_value - 25
end

puts response.to_json                                       3
```

- ***1*** **Ruby module imported for JSON handling**
- ***2*** **The second command-line parameter is the current JSON response.**
- ***3*** **Prints the transformed response to stdout**

Some syntax changes are obvious from the JavaScript code to the Ruby code (primarily a different hash syntax) and they use some different functions (`to_i` instead of `parseInt`), but most of the code looks like a Ruby decorator. The key differences are the input (parsing the response from the command line) and the output (printing the transformed response to stdout). In the following listing, you do the same thing with addTimestamp.rb.

Listing 7.8. Ruby script to add a timestamp to the response JSON

```
require 'json'

response = JSON.parse(ARGV[1])
response['body']['timestamp'] = Time.now.getutc
puts response.to_json
```

By chaining together transformations, `shellTransform` acts as a way of adding a transformation pipeline to your response handling, allowing as much complexity as you require. My standard advice still applies: it's good to have such power for when you absolutely need it, but try not to need it.

7.3. ADDING LATENCY TO A RESPONSE

The next behavior we'll look at is a whole lot simpler and doesn't require you to set the `--allowInjection` command line flag. Sometimes you need to simulate latency in responses, and the `wait` behavior tells mountebank to take a nap before returning the response. You pass it the number of milliseconds to sleep as follows.

Listing 7.9. Using a `wait` behavior to add latency

```
{
  "is": {
    "body": {
      "name": "Sleepy"
    }
  },
  "_behaviors": {
    "wait": 3000          1
  }
}
```

- *1* **Adds 3 seconds of latency**

Like the `decorate` behavior, you can add the `wait` behavior to saved responses that proxies generate. When you set the `addWaitBehavior` to `true` on a proxy, mountebank will automatically fill in the generated `wait` behavior based on how long the real downstream call took. I show how to use that to create robust performance tests in chapter 10.

7.4. REPEATING A RESPONSE MULTIPLE TIMES

Sometimes you need to send the same response multiple times before moving on to the next response. You can copy the same response multiple times in the `responses` array, but that practice is generally frowned on by the software community, as it hurts maintainability. It's an important enough concept that it even has its own acronym: DRY (Don't Repeat Yourself).

The `repeat` behavior lets the computer do the repeating for you (figure 7.6). It accepts the number of times you want to repeat the response, and mountebank helps you avoid those snooty software engineers looking down their noses at you for not being DRY enough.

Figure 7.6. Repeating a response multiple times

A common use case involves triggering an error response after a set number of happy path responses. An example I already used a couple of times in this book involves querying an inventory service. In chapters 3 and 4, I showed how you can use a list of responses to show the inventory for a product running out over time:[3]

3

As we did earlier, we are returning an overly simplistic body to focus on only the point of the example.

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

Most test cases don't require this level of specificity. In slightly oversimplified terms, a consumer of an inventory service only cares about two scenarios:

- The inventory is greater than zero (or greater than or equal to the quantity being ordered).
- The inventory is zero (or less than the quantity being ordered).

Simplifying even further, the only two scenarios that matter for this test case are

- A happy path
- An out-of-inventory error

The only slightly complicated factor is that you might want to return a few happy path responses before returning an out-of-inventory error. You can do that with only two responses and a `repeat` behavior, as shown in the following listing.

Listing 7.10. Using a `repeat` behavior to return an error after a small number of successes

```
{
  "responses": [
    {
      "is": { "body": "9999" },          1
      "_behaviors": { "repeat": 3 }      2
    },
    {
      "is": { "body": "0" }              3
    }
  ]
}
```

- ***1*** **Return the happy path…**
- ***2*** **…three times.**
- ***3*** **Then return the error path.**

Test case construction

We have looked at some advanced mountebank capabilities over the course of this book. Sometimes those capabilities are indispensable for solving complicated test problems. But you can simplify most test cases to a small essential core of what you're trying to test, and removing the noise in the test data setup helps keep the focus on that core. The `repeat` example we just looked at shows how the thought process of

simplifying your test case has knock-on benefits to your test data management.

7.5. REPLACING CONTENT IN THE RESPONSE

You can always add dynamic data to a response through an `inject` response, or through the `decorate` and `shellTransform` behaviors. But two additional behaviors support inserting certain types of dynamic data into the response without the overhead of programmatic control.

7.5.1. Copying request data to the response

The `copy` behavior allows you to capture some part of the request and insert it into the response. Imagine that the system under test depended on a service reflecting the account ID from the request URL back in the response body, so that (for example) when you send a `GET` request to /accounts/8731, you get a response that reflects that ID and otherwise resembles my account profile in various online forums I participate in:

```
{
  "id": "8731",              1
  "name": "Brandon Byars",
  "description": "Devilishly handsome",
  "height": "Lots of it",
  "relationshipStatus": "Available upon request"
}
```

- ***1*** **This has to match the ID in the path.**

This response has two core aspects:

- The `id`, which has to match the one provided on the request URL
- The test data you need for your scenario

A standard `is` response supports managing scenario-specific test data, and the `copy` behavior allows you to insert the `id` from the request. Copying the `id` from the request to the response requires you to reserve a slot in the response that you can replace and to be able to select only the data you want from the request. The first part is easier—you add any token you choose to the response, as shown in the following listing.

Listing 7.11. Specifying a token in the response to replace with a value from the request

```
{
  "is": {
    "body": {
      "id": "$ID",                    1
      "name": "Brandon Byars",
      "description": "Devilishly handsome",
      "height": "Lots of it",
      "relationshipStatus": "Available upon request"
    }
  }
}
```

- ***1* A placeholder that the copy behavior will replace with the value**

The first section of the `copy` behavior requires that you specify the request field you're copying from and the response token you need to replace:

```
{
  "from": "path",
  "into": "$ID",
  ...
}
```

The only part you need to fill in is the part that selects the `id`. The `copy` behavior (and the `lookup` behavior, which we look at next) uses some of the same predicate matching capabilities we looked at in chapter 4, specifically regular expressions, XPath, and JSONPath. Recall that each predicate applies a matching operation against a request field. Whereas a predicate tells you whether or not the match was successful, the `copy` and `lookup` behaviors are able to grab the specific text in the request field that matched.

For this example, a regular expression will do the trick. You need to capture a string of digits at the end of the request `path`. You can use some of the regex primitives we looked at in chapter 4 to make that selection:

- `\d` — A digit, 0–9 (you have to double-escape the backslash in JSON)
- `+` — One or more times
- `$` — The end of the string

Putting it all together, the stub would look like the following listing.

Listing 7.12. Using a `copy` behavior to insert the ID from the URL into the response body

```
{
  "responses": [{
    "is": {
      "body": {
        "id": "$ID",                                          1
        "name": "Brandon Byars",
        "description": "Devilishly handsome",
        "height": "Lots of it",
        "relationshipStatus": "Available upon request"
      }
    },
    "_behaviors": {
      "copy": [{                                              2
        "from": "path",                                       3
        "into": "$ID",                                        4
        "using": {                                            5
          "method": "regex",                                  5
          "selector": "\\d+$"                                 5
        }                                                     5
      }]
    }
  }]
}
```

- ***1*** **The token to replace**
- ***2*** **An array—multiple replacements are allowed**
- ***3*** **The request field to copy from**
- ***4*** **The token to replace**
- ***5*** **The selection criteria to act on the request path**

There's a lot more to this behavior, but before we get too far, I should point out a couple of aspects that you may have already noticed. First, the `copy` behavior accepts an array, which means you can make multiple replacements in the response. Each replacement should use a different token, and each one can select from a different part of the request.

The other thing to notice is that you never specify *where* the token is in the response. That's by design. You could have put the token in the `headers` or even the `statusCode`, and mountebank would replace it. If the token is listed multiple times, mountebank will replace each instance, regardless of where it's located in the response.

Using a grouped match

The previous example made an assumption that you could define a regular expression

that entirely matched the value you needed to grab *and nothing else*. That's a pretty weak assumption.

Many services use some form of a globally unique identifier (GUID) as an `id`, and the `path` often extends beyond the part containing the `id`. For example, the `path` might be /accounts/5ea4d2b5/profile, where "5ea4d2b5" is the `id` you need to copy. You can no longer rely on `\\d+` as a selector because the `id` contains more than digits. You certainly can rely on other mechanisms to match—for instance, by recognizing that the `id` follows the word "accounts" in the path:

```
accounts/\\w+
```

That selector uses the "\w" regular expression metacharacter to capture a *word* character (letters and digits) and adds the "+" to ensure that you capture one or more of them. Then it prefixes the "accounts/" to ensure you're grabbing the right portion of the path. With this expression, you do successfully grab the `id`. Unfortunately, you grab the "accounts/" literal string as well, and the replaced `body` would look like:

```
{
  "id": "accounts/5ea4d2b5",
  ...
}
```

Regular expressions support grouped matches to allow you to grab only the data you need from a match. Every regular expression has a default first group that's the entire match. Every time you surround a portion of the selector with parentheses, you describe another group. You will adjust your selector to add a group around the `id` portion of the `path`, while leaving the literal string "accounts/" to make sure you are grabbing the right portion of the `path`:

```
accounts/(\\w+)
```

When you match this regular expression against the string "/accounts/5ea4d2b5/profile", you get an array of match groups that looks like

```
[
  "accounts/5ea4d2b5",
  "5ea4d2b5"
]
```

The first group is the entire match, and the second is the first parenthetical group. If you place an unadorned token in the response, as you did in the previous section, mountebank will replace it with the first index of the array, which corresponds to the entire match. But you can add an index to the token corresponding to the index of the match group array, as shown in the following listing, which allows you to pinpoint the part of the `path` you want to copy with laser precision.

Listing 7.13. Using a grouped match to copy a portion of the request `path`

```
{
  "is": {
    "body": {
      "id": "$ID[1]",                    1
      ...
    }
  },
  "_behaviors": {
    "copy": [{
      "from": "path",
      "into": "$ID",                     2
      "using": {
        "method": "regex",
        "selector": "accounts/(\\w+)"    3
      }
    }]
  }
}
```

- ***1* Specifies the index in the response token**
- ***2* Specifies the base token in the behavior**
- ***3* Uses a grouped match for more precision**

You can always use indexed tokens in the response. Assuming you specify a token of `$ID` like you did in listing 7.13, then putting `$ID` in the response is equivalent to putting `$ID[0]`. I doubt that matters much for regular expressions, as I suspect most real-world use cases will have to use groups to grab the exact value they want. That isn't necessarily true for the other selection approaches that the `copy` behavior supports: `xpath` and `jsonpath`.

Using an XPath selector

Although a regular expression elegantly supports grabbing a value out of *any* request field, `xpath` and `jsonpath` selectors can be useful to match values inside an incoming request body. They work similarly to how `xpath` and `jsonpath` predicates work, as

described in chapter 4. The key difference is that predicate XPath and JSONPath selectors are used with a matching operator (like equals) to test whether the request matches, whereas using those selectors with behaviors helps grab the matching text in the request to change the response.

Take the following request body from the system under test, representing a list of accounts:

```
<accounts xmlns="https://www.example.com/accounts">
  <account id="d0a7b1b8" />
  <account id="5ea4d2b5" />
  <account id="774d4feb" />
</accounts>
```

Your virtual response needs to reflect back details of the second account in the request. Those details are specific to your test scenario, but the ID has to match what was sent in the request body. You can use an XPath selector to grab the `id` attribute of the second `account` attribute, as shown in the following listing.

Listing 7.14. Using an XPath selector to copy a value from the request to the response

```
{
  "responses": [{
    "is": {
      "body": "<account><id>$ID</id>...</account>"      1
    },
    "_behaviors": {
      "copy": [{
        "from": "body",
        "into": "$ID",                                  2
        "using": {
          "method": "xpath",                            3
          "selector": "//a:account[2]/@id",             3
          "ns": {                                       3
            "a": "https://www.example.com/accounts"     3
          }                                             3
        }
      }]
    }
  }]
}
```

- ***1*** **Tokenizes the response body**
- ***2*** **Defines a token in the copy behavior**

- ***3* Selects the value from the XML request body**

The `xpath` selector and namespace support work identically to `xpath` predicates (chapter 4) and predicate generators (chapter 5). As you saw with predicates, mountebank also supports JSONPath. We will look at an example with the `lookup` behavior shortly.

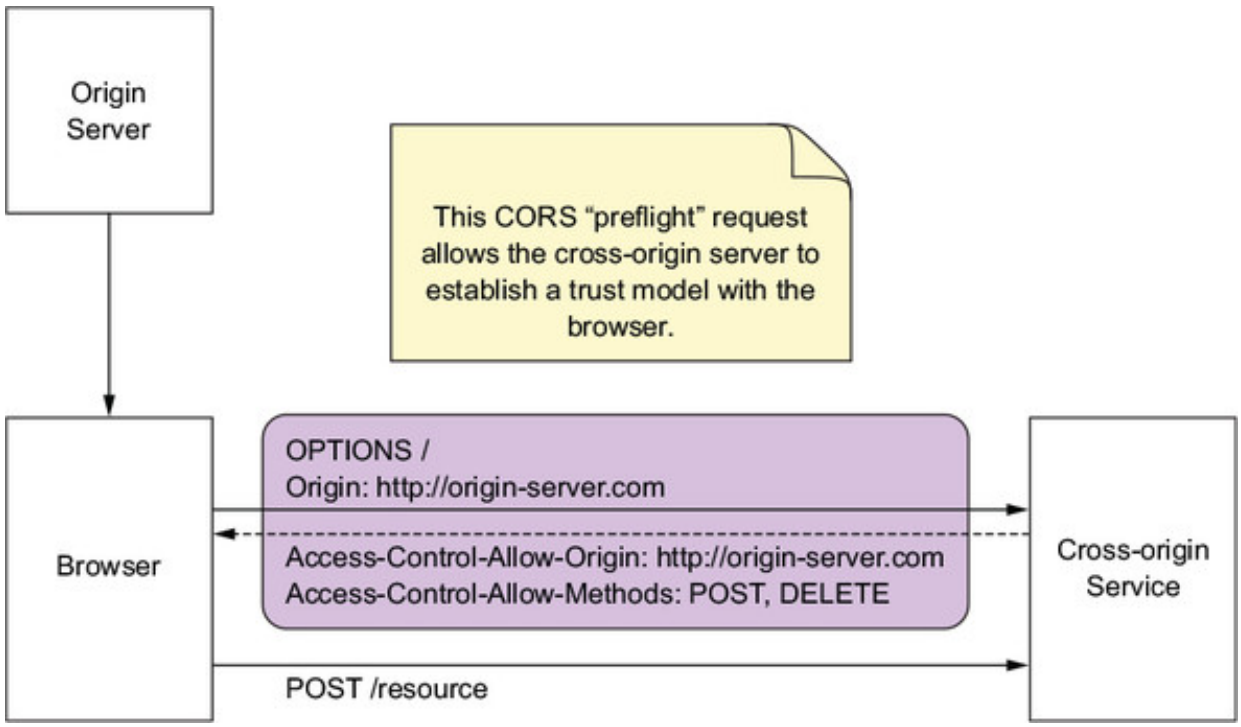
Virtualizing a CORS preflight response

Cross-origin resource sharing, or CORS, is a standard that allows browsers to make cross-domain requests. In the olden days, a browser would only execute JavaScript calls to the same domain as the one that served up the hosting web page. This *same-origin policy* is the bedrock of browser security, as it helps prevent a host of malicious JavaScript injection attacks. But as websites became more dynamic and needed to pull behavior from disparate resources spread across multiple domains, it also proved to be too restrictive. Creative developers found creative hacks to bypass the same-origin policy, like JSONP, which manipulates the `script` element in an HTML document to pass JavaScript from a different domain to a callback function already defined. JSONP is confusing and hard to understand because it works around the browser's built-in security mechanism.

The CORS standard evolves the browser's security model to allow the browser and server to both weigh in on whether a cross-domain request is valid or not. The standard requires a *preflight* request for certain types of cross-domain requests to determine whether the request is valid. A preflight request is an `HTTP OPTIONS` call with a few special headers that commonly trip up testers when creating virtual services[4] (figure 7.7).

4 Future versions of mountebank likely will make virtualizing CORS preflight requests easier.

Figure 7.7. A CORS preflight request to establish trust



The browser is set to automatically send these preflight requests for some types of cross-origin requests. If you want to virtualize the cross-origin service so you can test the browser application, your virtual service needs to know how to respond to a preflight request in a way that enables the browser to make the *actual* cross-origin request (for example, the call to `POST /resource` in figure 7.7). Copying the `Origin` header value from the request header into the response header, as shown in the following listing, is a great example of using the `copy` behavior and demonstrates tokenizing something other than the HTTP body.

Listing 7.15. Virtualizing a CORS preflight request

```
{
  "predicates": [{                                                    1
    "equals": { "method": "OPTIONS" }                                 1
  }],                                                                 1
  "responses": [{
    "is": {
      "headers": {
        "Access-Control-Allow-Origin": "${ORIGIN}",                   2
        "Access-Control-Allow-Methods": "PUT, DELETE"
      }
    },
    "_behaviors": {
      "copy": [{
        "from": { "headers": "Origin" },                              3
        "into": "${ORIGIN}",                                          4
        "using": { "method": "regex", "selector": ".+" }              5
      }]
    }
  }]
```

```
}
```

- ***1*** **Looks for a preflight request signature**
- ***2*** **Tokenizes the response header**
- ***3*** **Looks in the request Origin header**
- ***4*** **Replaces the response header token…**
- ***5*** **…with the entire request header value.**

The regular expression ".+" means "one or more characters" and effectively captures the entire request header. Because you don't need to use a grouped match, you can use the token in the response without an array index. You can do a lot more with CORS configuration, but the `copy` approach satisfies the need for creating a flexible virtual service that reflects the client requests in a way that enables the client to make subsequent requests.

7.5.2. Looking up data from an external data source

Service virtualization is great for testing error flows, which are often difficult to reproduce on demand in real systems but happen enough in live systems that you still need to test for them. The challenge is creating a set of test data that captures all the error flows in a visible and maintainable way. Take account creation, for example. Using predicates, you could set up a virtual *account* service that responds with different error conditions based on the name of the account you are trying to create. Let's say the first error flow you need to test is what happens when the account already exists. The following configuration would ensure your virtual service returns an error for a duplicate user when the name is "Kip Brady," assuming that's passed in as a JSON `name` field in the request:

```
{
  "stubs": [{
    "predicates": [{
      "equals": { "body": "Kip Brady" },
      "jsonpath": { "selector": "$..name" }
    }],
    "responses": [{
      "is": {
        "statusCode": 400,
        "body": {
          "errors": [{
            "code": "duplicateEntry",
            "message": "User already exists"
          }]
```

```
          }
        }
      }]
    }]
  }
```

If you want “Mary Reynolds” to represent a user too young to register, you can use the same JSONPath selector to look for a different value:

```
{
  "stubs": [
    {
      "predicates": [{
        "equals": { "body": "Kip Brady" },
        "jsonpath": { "selector": "$..name" }
      }],
      "responses": [{
        "is": {
          "statusCode": 400,
          "body": {
            "errors": [{
              "code": "duplicateEntry",
              "message": "User already exists"
            }]
          }
        }
      }]
    },
    {
      "predicates": [{
        "equals": { "body": "Mary Reynolds" },
        "jsonpath": { "selector": "$..name" }
      }],
      "responses": [{
        "is": {
          "statusCode": 400,
          "body": {
            "errors": [{
              "code": "tooYoung",
              "message": "You must be 18 years old to register"
            }]
          }
        }
      }]
    }
  ]
}
```

“Tom Larsen” can represent a 500 server error, and “Harry Smith” can represent an overloaded server. Both require new stubs.

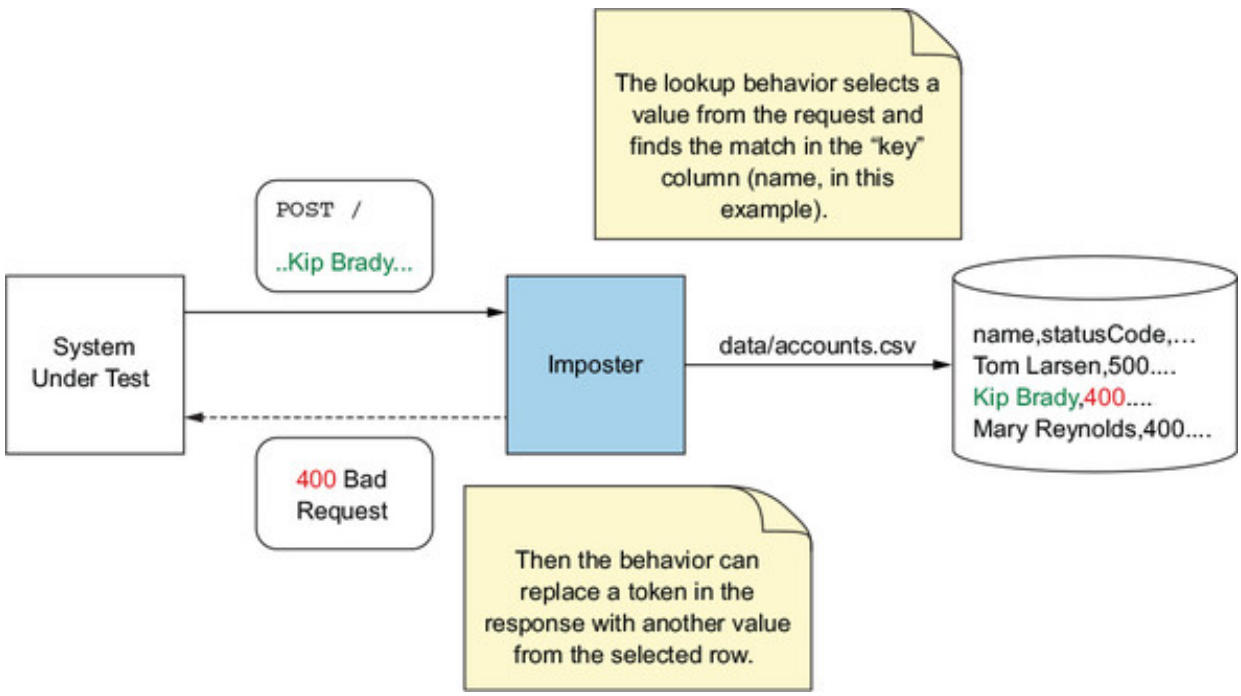
This is obviously an unsustainable approach to managing test data. The JSONPath selector is the same among all the stubs, as is the structure of the JSON body. You'd like to be able to centralize the test data in a CSV file, as shown in the following listing.

Listing 7.16. Centralizing error conditions in a CSV file

```
name,statusCode,errorCode,errorMessage
Tom Larsen,500,serverError,An unexpected error occurred
Kip Brady,400,duplicateEntry,User already exists
Mary Reynolds,400,tooYoung,You must be 18 years old to register
Harry Smith,503,serverBusy,Server currently unavailable
```

The `lookup` behavior, a close cousin of the `copy` behavior, allows you to do this. Like the `copy` behavior, it replaces tokens in the response with dynamic data. The key difference is where that dynamic data comes from. For the `copy` behavior, it's the request. For the `lookup` behavior, it's an external data source. As of this writing, the only data source mountebank supports is a CSV file (figure 7.8). That likely will change by the time you are reading this.

Figure 7.8. Looking up a value from a CSV file



Before we get to how the replacement happens, let's look at the lookup operation itself. As you can see in figure 7.8, a successful lookup requires three values:

- A key selected from the request (Kip Brady)
- The connection to the external data source (data/accounts.csv)
- The key column in the external data source (name)

Those three values are sufficient to capture a row of values you can use in the replacement. You represent them in a way that closely resembles the `copy` behavior, as shown in the following listing.

Listing 7.17. Using a lookup behavior to retrieve external test data

```
{
  "responses": [{
    "is": { ... },                                    1
    "_behaviors": {
      "lookup": [{
        "key": {                                      2
          "from": "body",                             2
          "using": {                                  2
            "method": "jsonpath",                     2
            "selector": "$..name"                     2
          }                                           2
        },                                            2
        "fromDataSource": {
          "csv": {                                    3
            "path": "examples/accounts.csv",          4
            "keyColumn": "name"                       5
          }
        },
        "into": "${row}"                              6
      }]
    }
  }]
}
```

- ***1*** **We'll come back to this.**
- ***2*** **Selects the value from the request**
- ***3*** **Type of data source**
- ***4*** **Path to CSV file**
- ***5*** **Column name to match request value**
- ***6*** **Response token name**

The `key` part of the `lookup` behavior is similar to the `copy` behavior we looked at earlier. It allows you to use a `regex`, `xpath`, or `jsonpath` selector to look in a request field and grab a value. You can add an `index` field to use a grouped regular expression match.

The `into` field is also the same as what you saw with `copy`. Here you used `${row}` as the token name. It can be anything you like. As far as mountebank is concerned, it it's a

string. The addition is what you see in the `fromDataSource` field. For CSV data sources, you specify the path to the file (relative to the running `mb` process) and the name of the key column.

If you pass `"Kip Brady"` in as the `name`, your token (`${row}`) matches an entire row of values from the CSV file. In JSON format, it would look like this:

```
{
  "name": "Kip Brady",
  "statusCode": "400",
  "errorCode": "duplicateEntry",
  "errorMessage": "User already exists"
}
```

This highlights a secondary difference between `copy` and `lookup`: with a `lookup` behavior, your token represents an entire row of values, meaning each individual replacement has to be indexed with the column name. Let's look at the `is` response for your example, which tokenizes the responses you previously had to copy into multiple stubs, in the following listing.

Listing 7.18. Using tokens to create a single response for all error conditions

```
{
  "is": {
    "statusCode": "${row}['statusCode']",     1
    "body": {
      "errors": [{
        "code": "${row}['errorCode']",        1
        "message": "${row}['errorMessage']"   1
      }]
    }
  },
  "_behaviors": {
    "lookup": [{ ... }]                       2
  }
}
```

- ***1* Looks up the appropriate field in the row you looked up**
- ***2* See listing 7.17.**

The `lookup` behavior treats the token like a JSON object that you can key by the column name. This allows you to retrieve an entire row of data in the `lookup` behavior and use fields within the row to populate the response.

7.6. A COMPLETE LIST OF BEHAVIORS

For reference, table 7.1 provides a complete list of behaviors that mountebank supports, including whether they support affecting saved proxy responses and whether they require the `--allowInjection` command-line flag.

Table 7.1. All behaviors that mountebank supports

Behavior	Works on saved proxy responses?	Requires injection support?	Description
decorate	yes	yes	Uses a JavaScript function to postprocess the response
shellTransform	no	yes	Sends the response through a command-line pipeline for postprocessing
wait	yes	no	Adds latency to a response
repeat	no	no	Repeats a response multiple times
copy	no	no	Copies a value from the request into the response
lookup	no	no	Replaces data in the response with data from an external data source based on a key from the request

Behaviors are powerful additions to mountebank, and we covered a lot of ground in this chapter. We will round out mountebank's core capabilities in the next chapter when we look at protocols.

SUMMARY

- The `decorate` and `shellTransform` behaviors are similar to response injection in that they allow programmatic transformation of the response. But they apply postprocessing transformation, and the `shellTransform` allows multiple

transformations.

- The `wait` behavior allows you to add latency to a response by passing in the number of milliseconds to delay it.
- The `repeat` behavior supports sending the same response multiple times.
- The `copy` behavior accepts an array of configurations, each of which selects a value from the request and replaces a response token with that value. You can use regular expressions, JSONPath, and XPath to select the request value.
- The `lookup` behavior also accepts an array of configurations, each of which looks up a row of data from an external data source based on the value selected from the request using regular expressions, JSONPath, and XPath. The token in the response is indexed by the field name.

Chapter 8. Protocols

This chapter covers

- How protocols work in mountebank
- An in-depth look at the TCP protocol
- How to stub a bespoke text-based TCP protocol
- How to stub a binary .NET Remoting service

Let's get real: faking it only gets you so far. At some point, even the best virtual services have to lay down some real bits on the wire.

The functionality we have explored so far—responses, predicates, and behaviors—is largely the realm of stubbing and mocking tools. Those stubbing and mocking tools were created for creating test doubles in process, allowing you to perform focused tests by methodically manipulating dependencies. Responses correspond to what stub functions return, and predicates exist to provide different results based on the way the function is called (for example, returning a different result based on a request parameter). I'm not aware of any in-process mocking tools that have the concept of behaviors per se, but there's nothing preventing them. Behaviors are transformations on the result.

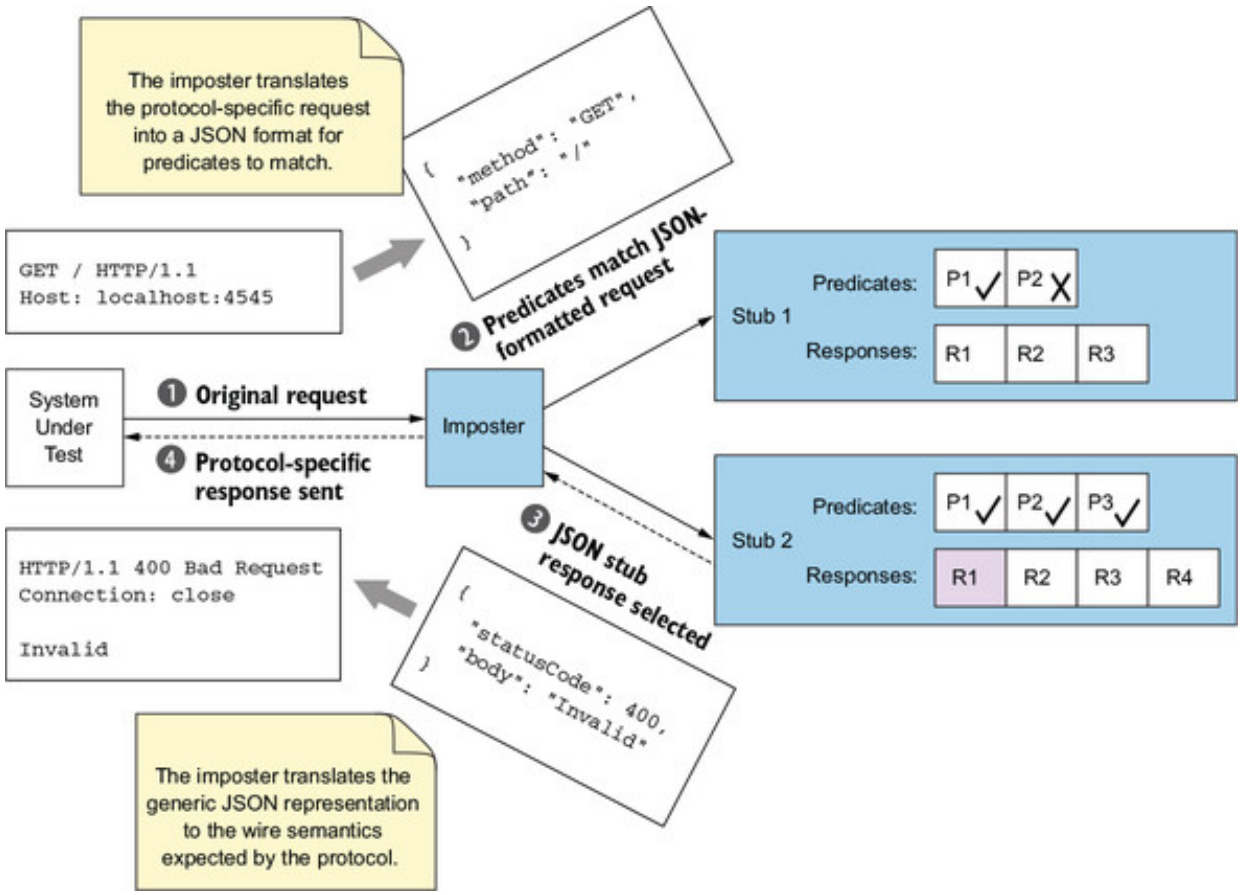
But there is one thing that virtual services do that traditional stubs don't: they respond over the network. All the responses so far have been HTTP, but mountebank supports other ways of responding. Enterprise integration is often messy, and sometimes you'll need to virtualize non-HTTP services. Whether you have custom remote procedure calls or a mail server as part of your stack, service virtualization can help. And now, at long last, it's time to explore mountebank's support for network protocols.

8.1. HOW PROTOCOLS WORK IN MOUNTEBANK

Protocols are where the rubber meets the road in mountebank. The core role of the protocol is to translate the incoming network request into a JSON representation for predicates to operate on, and to translate the JSON mountebank response structure

into the wire format expected by the system under test (figure 8.1).

Figure 8.1. The flow of a mountebank-generated HTTP response



All of the mountebank imposters that we've seen so far are full-featured HTTP servers, and the bits they put on the wire conform to the HTTP protocol. That's the secret sauce that allows you to repoint the system under test to mountebank with no changes: it sends out an HTTP request and gets an HTTP response back. It has no need to know that an imposter formed the response by employing a secret cabal of stubs to match the request with predicates, form a JSON response, and postprocess it with behaviors. All the system under test cares about is that mountebank accepts an HTTP request and returns a bunch of bits over the wire that look like an HTTP response.

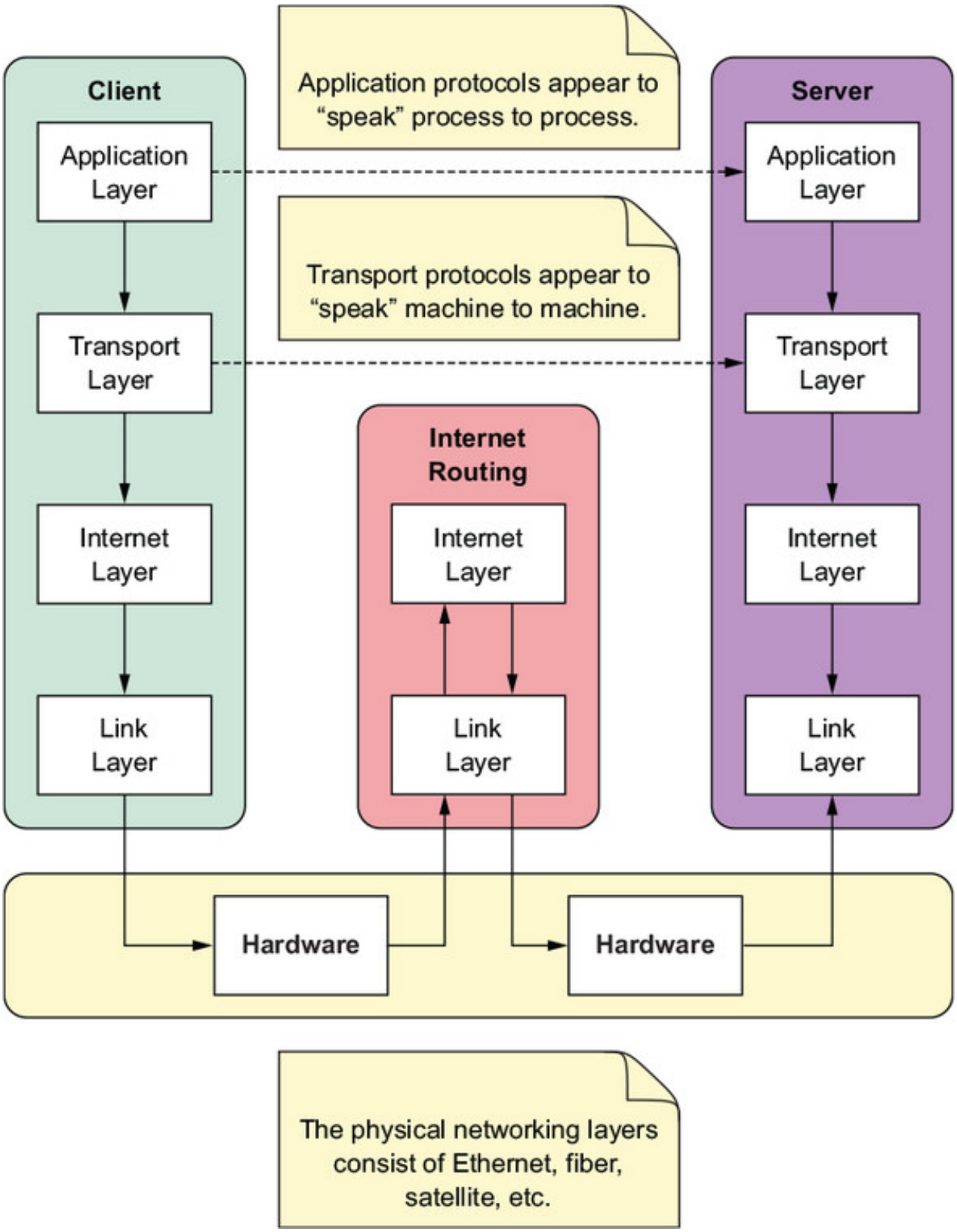
Mountebank, unique in the open source service virtualization world, is multiprotocol. The clean separation of concepts allows the stubbing functionality to work regardless of what protocol your system under test expects. We'll explore that in the context of older remote procedure call (RPC) protocols, but before we get there, let's start with a primer on a foundational building block of network communication.

8.2. A TCP PRIMER

Mountebank supports the TCP protocol, but TCP isn't on equal footing with HTTP/S. It's more accurate to say that mountebank supports a range of custom application

protocols built on top of TCP. Most conceptual networking models show protocols in layers, and it takes a whole suite of protocols to make something as complex as the internet work. (See figure 8.2.)

Figure 8.2. A client application talking to a server application over the internet



The genius of the TCP/IP stack is that clients and server processes can act as if they're talking directly to each other, even when they're on remote machines. When you visit the mountebank website (http://www.mbtest.org), your browser can act as if it's speaking directly with the web server on the other end. The same is true when your system under

test makes an HTTP call. Regardless of whether the service it's accessing is real or virtualized, the client code can operate as if there's a direct connection to the server.

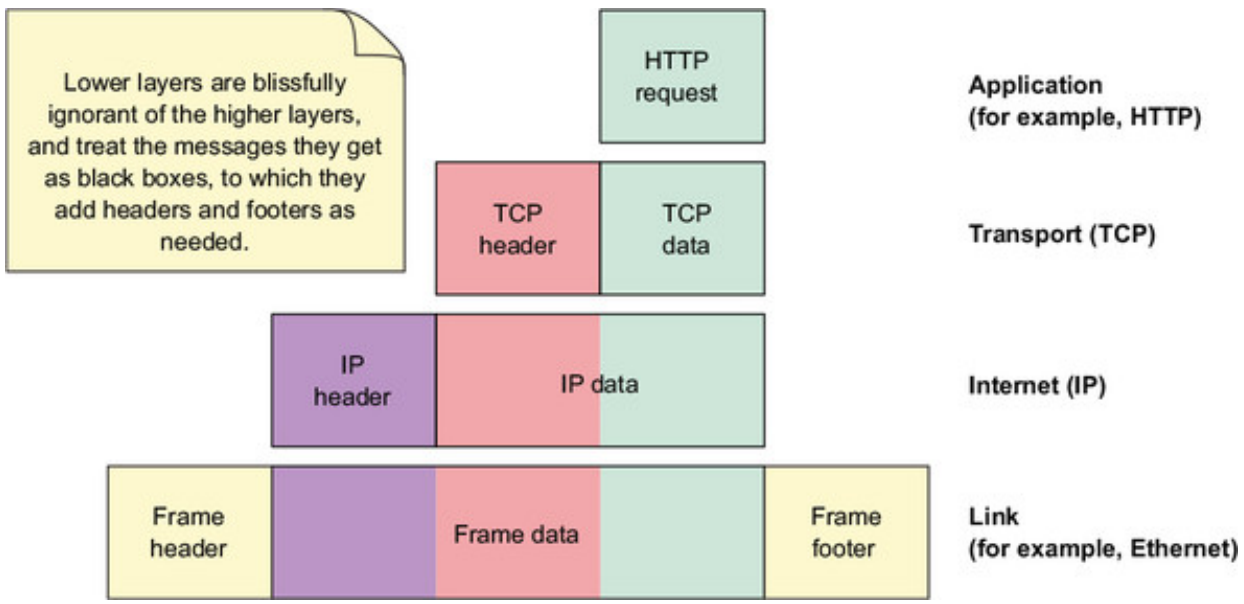
The reality is more complicated. HTTP is an *application protocol* and depends on TCP as the *transport protocol* to deliver to the remote host. TCP in turn relies on downstream protocols to route between networks (the IP protocol) and to interface with the routers on the same network (the network is often referred to as the "link," which is why the lowest layer is called the *link layer* in figure 8.2).

What your web browser is doing is forming an HTTP request and passing it to the local operating system, which hands off the request to the TCP protocol implementation. TCP lovingly wraps the HTTP message in an envelope that adds a bunch of delivery guarantees and performance optimizations. Then it hands control over to the IP protocol, which once again wraps the whole message and adds addressing information that the core infrastructure of the internet knows how to use to route to the correct remote machine. Finally, the doubly wrapped HTTP message is handed off to the device driver for the network interface on your computer, which yet again wraps the message with some Ethernet or Wi-Fi information needed to transmit the whole package to your router, which happily forwards it to the next network. (See figure 8.3.) The process works in reverse once it reaches the right server machine.[1]

1

Figure 8.3 is inspired by a similar image on the Wikipedia page describing the internet protocol suite, which has a more comprehensive explanation of how layering works: https://en.wikipedia.org/wiki/Internet_protocol_suite.

Figure 8.3. Transforming an HTTP request to route across the network



TCP enables host-to-host communication, but it's the application protocols on top that allow a client process to talk to a server process. HTTP is the most famous application

protocol but far from the only one. Mountebank aspires to treat well-known application protocols like HTTP as first class citizens, but a host of niche or custom application protocols exist in the archaeological substratum of enterprise integration, and mountebank's support of the TCP protocol also provides a way to virtualize them.

8.3. STUBBING TEXT-BASED TCP-BASED RPC

For those of you who grew up in a world where distributed programming was commonplace, it can be a bit bewildering to look at some of the ways applications used to integrate. Imagine you are a C++ programmer in a bygone era being paid to integrate two applications over the network. The challenges of distributed programming aren't commonly understood yet. Early attempts at formalizing RPC using standards like CORBA may have happened, but they seem overly complicated. It seems much simpler to pass a function name and a few parameters to a remote socket and expect it to return a status code indicating whether the function succeeded and a return value. If you take out the networking, it looks similar to how in-process function calls work.

Now, 20 or 30 years later, younger generations are still adding capabilities to your custom RPC code because it's so central to keeping the lights on that it's cheaper to keep it than to rip it out. They may not like it, but none of them have ever written code that's stayed in production for decades. Like it or not, this is a common scenario in many long-standing enterprises.

Let's imagine that the remote server manages core inventory. The payload of a TCP packet making an RPC request to change the inventory may look like this, for example:

```
updateInventory      1
5131                 2
-5                   3
```

- ***1* Function name**
- ***2* First parameter (for example, productId)**
- ***3* Second parameter (for example, inventoryDelta)**

And the response may look like this:

```
0        1
1343     2
```

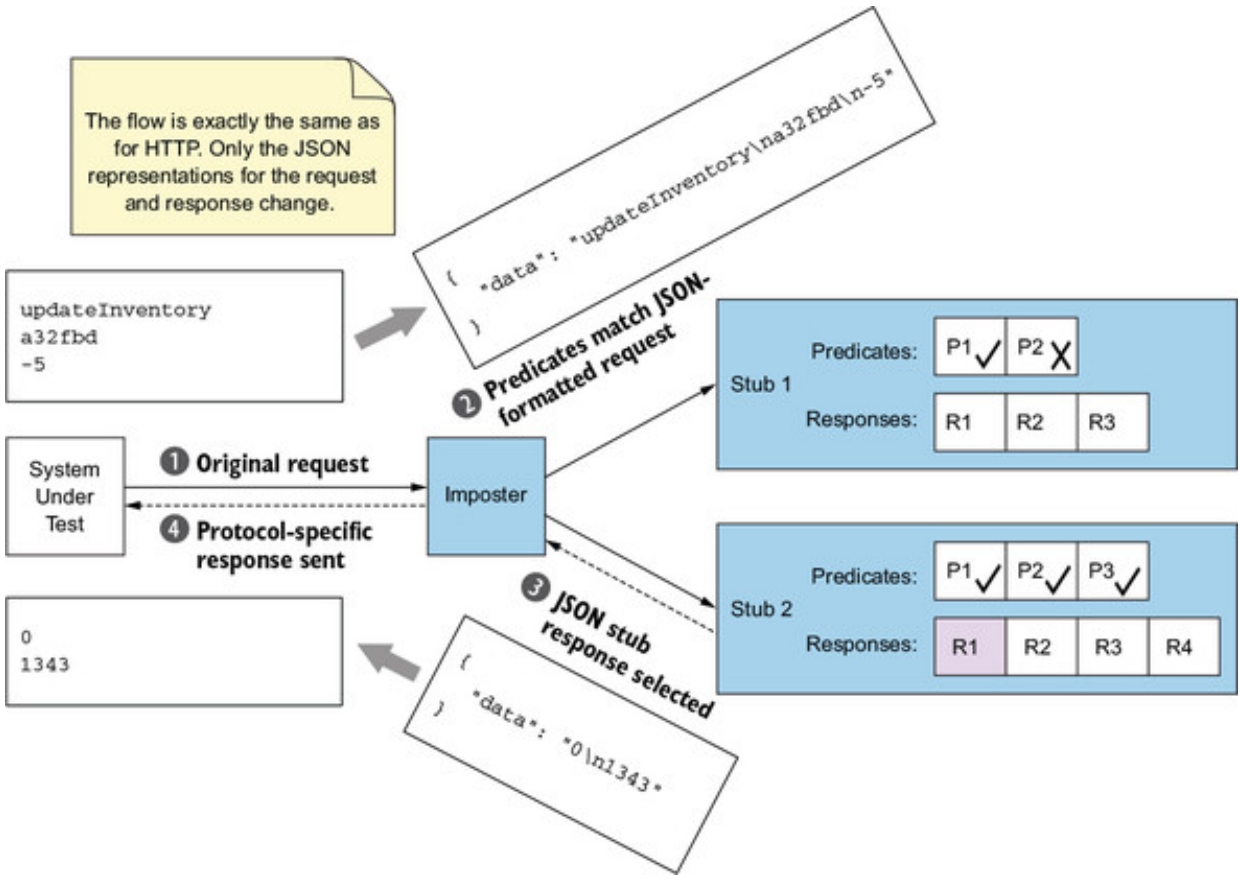
- ***1* Status code**

- *2* **Total inventory**

In this example, new lines separate parameters, and the schema is implicit rather than defined by something like JSON or XML. Mountebank won't be able to understand the semantics of the RPC, so it encapsulates the payload in a single `data` field.

The flow of data, as shown in figure 8.4, looks similar to the flow of data in a virtual HTTP service (figure 8.1). The only difference is the format for the request and response.

Figure 8.4. Virtualizing a custom TCP protocol



8.3.1. Creating a basic TCP imposter

Creating the TCP imposter is as simple as changing the `protocol` field to `tcp`:

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

That configuration is sufficient for mountebank to spin up a TCP server listening on port 3000. It'll accept the request, but the response will be blank. If you want to virtualize the call to `updateInventory`, you do so with the same predicates and

response capability that you've seen for HTTP, as shown in the following listing. The only difference is the JSON structure for requests and responses. For `tcp`, both the request and response contain a single `data` field.

Listing 8.1. Virtualizing a TCP `updateInventory` call

```
{
  "protocol": "tcp",
  "port": 3000,
  "stubs": [{
    "predicates": [{
      "startsWith": { "data": "updateInventory" }    1
    }],
    "responses": [{
      "is": { "data": "0\n1343" }                    2
    }]
  }]
}
```

- ***1* Looks for the appropriate function**
- ***2* Returns the protocol-specific response format**

You can test out your imposter using an application like telnet. Telnet opens an interactive TCP connection to a server, which makes it tricky to script. Netcat (http://nc110.sourceforge.net/) is like a noninteractive telnet, which makes it ideally suited for testing TCP-based services. You can trigger your inventory RPC call response and test out your TCP imposter with netcat using the following command:

```
echo "updateInventory\na32fbd\n-5" | nc localhost 3000
```

You wrap the request message in a string and pipe it to netcat (`nc`), sending it to the correct socket. It'll send back the virtual response you configured to the terminal:

```
0
1343
```

8.3.2. Creating a TCP proxy

TCP imposters work with the other response types as well. You can record and replay using a `proxy` like you can with HTTP. For example, if the real service is listening on port 3333 of remoteservice.com, you could set up a record/replay imposter by pointing the `to` field on the `proxy` configuration to the remote socket, as shown in the following

listing.

Listing 8.2. Using a TCP record/replay `proxy`

```
{
  "protocol": "tcp",
  "port": 3000,
  "stubs": [{
    "responses": [{
      "proxy": {
        "to": "tcp://remoteservice.com:3333"    1
      }
    }]
  }]
}
```

- ***1*** **Destination of remote service**

The `proxy` behavior is identical to what you saw in chapter 5. In the default mode (`proxyOnce`), mountebank will save the response and serve it up on the next request without making the downstream call again. You can test this out with netcat:

```
echo "updateInventory\na32fbd\n-5" | nc -q 1 localhost 3000
```

Notice the added "-q 1" parameter. By default, when netcat makes a TCP request, it closes the client end of the connection immediately, which is appropriate for one-way fire-and-forget-style communication. It isn't appropriate for two-way request-response-style communication common in RPC. Because it takes a small amount of time for mountebank to make the downstream call to get the response, by the time mountebank tries to respond, it may discover that nobody is listening. The "-q 1" parameter tells netcat to wait one second before closing the connection, so you'll see the response on the terminal.

Unfortunately, the "-q" parameter isn't present on all versions of netcat, including the default version on Mac and Windows computers. If you leave it off, you won't get a response on the terminal and you'll see the error in the mountebank logs when it cannot send the response. But subsequent calls will still work, as mountebank now has a saved version of the response and can respond immediately.

You can use `predicateGenerators` with the TCP protocol as well, but it isn't very discriminatory because there's only one field. For example, the following configuration makes a new downstream call anytime *anything* in the RPC call is different.

Listing 8.3. A TCP `proxy` with `predicateGenerators`

```
{
  "protocol": "tcp",
  "port": 3000,
  "stubs": [{
    "responses": [{
      "proxy": {
        "to": "tcp://remoteservice.com:3333",
        "predicateGenerators": [{
          "matches": { "data": true }      1
        }]
      }
    }]
  }]
}
```

- ***1*** **Generates a new stub for each new payload**

Although it would be nice to generate predicates only on the function name, you can't. Mountebank has no way of knowing what the function name is, so any parameter that changes will force a new downstream call.

If you're lucky, the custom RPC protocol uses a payload format that mountebank understands: JSON or XML. If so, then you can get a bit more specific. We look at an example next.

8.3.3. Matching and manipulating an XML payload

I doubt you'll see many of these custom RPC protocols that use JSON, for the simple reason that by the time JSON was created, HTTP was already a predominant application integration approach. If you do see it, all of the JSONPath capability you've seen so far will work.

XML has been around a bit longer, and in fact one of the first attempts at using HTTP for integration was called POX over HTTP, where POX stood for Plain Ol' XML. Let's translate your `updateInventory` RPC payload to XML:

```
<functionCall>
  <functionName>updateInventory</functionName>
  <parameters>
    <parameter name="productId" value="5131" />
    <parameter name="amount" value="-5" />
  </parameters>
</functionCall>
```

You can easily imagine other function calls, for example, this one might be a call to get the inventory for product 5131:

```
<functionCall>
  <functionName>getInventory</functionName>
  <parameters>
    <parameter name="productId" value="5131" />
  </parameters>
</functionCall>
```

Now it becomes easier to build a more robust set of `predicateGenerators` for your `proxy`. If you want to save different responses for different combinations of function names and product IDs, you can do so, as shown in the following listing.

Listing 8.4. Using XPath `predicateGenerators` with a TCP `proxy`

```
{
  "responses": [{
    "proxy": {
      "to": "tcp://localhost:3333",
      "predicateGenerators": [
        {
          "matches": { "data": true },                    1
          "xpath": {                                      1
            "selector": "//functionName"                  1
          }                                               1
        },
        {
          "matches": { "data": true },                    2
          "xpath": {                                      2
            "selector":                                   2
➥ "//parameter[@name='productId']/@value"                2
          }                                               2
        }
      ]
    }
  }]
}
```

- ***1*** **The XML functionName must match.**
- ***2*** **The productId must match.**

All of the behaviors work with the TCP protocol as well, including those like the `copy` behavior that can use XPath to select values from the request.

8.4. BINARY SUPPORT

Not all application protocols speak in plain text. Many of them instead pass a binary request/response stream, which makes them challenging to virtualize. Challenging, but not impossible—remember mountebank's mission statement: to keep the easy things easy while making the hard things possible.

Mountebank makes virtualizing binary protocols possible in two ways. First, it supports serializing request and response binary streams using Base64 encoding. Second, nearly all of the predicates work against a binary stream with exactly the same semantics they use against text.

8.4.1. Using binary mode with Base64 encoding

JSON—the *lingua franca* of mountebank—doesn't directly support binary data. The workaround is to encode a binary stream into a string field. Base64 reserves 64 characters—upper- and lowercase letters, digits, and two punctuation marks—and maps them to the binary equivalents. Having 64 options allows you to encode six bits at a time ($2^6 = 64$).

Any modern language library will support Base64 encoding. Here's an example in JavaScript (node.js):

```
var buffer = new Buffer("Hello, world!");
console.log(buffer.toString("base64"));            1
```

- ***1*** **Prints out SGVsbG8sIHdvcmxkIQ==**

Similarly, decoding from Base64 is done using the `Buffer` type as well, as in this JavaScript example:

```
var buffer = new Buffer("SGVsbG8sIHdvcmxkIQ==");   1
console.log(buffer.toString("utf8"));              2
```

- ***1*** **Base64 encoded value**
- ***2*** **Prints out Hello, world!**

Having a TCP imposter return binary data requires letting mountebank know that you want a binary imposter and Base64 encoding the response, as shown in the following listing.

Listing 8.5. Setting up a binary response from an imposter

```
{
  "protocol": "tcp",
  "port": 3000,
  "mode": "binary",                                    1
  "stubs": [{
    "responses": [{
      "is": { "data": "SGVsbG8sIHdvcmxkIQ==" }     2
    }]
  }]
}
```

- **1 Switches to binary mode**
- **2 Returns binary equivalent of "Hello, world!"**

Once you set the `mode` to `binary`, mountebank knows to do the following:

- Interpret all response data as Base64-encoded binary streams, which it will decode when it responds over the wire
- Save all proxy responses as Base-64-encoded versions of the wire response
- Interpret all predicates as Base-64-encoded binary streams, which it will decode to match against the raw wire request

That last bullet point requires more explanation.

8.4.2. Using predicates in binary mode

Binary data is a stream of bits. It might be, for example, 01001001 00010001. I have spaced the bits into two eight-bit combinations (octets) because, although a long string of zeros and ones can be poetry to a computer, it gets a little tricky for us human folk to read. Using two octets also allows you to encode them as two numbers from 0–255 (2^8). In this case, that would be 73 17, or, in hexadecimal, 0x49 0x11. Hexadecimal is nice because it lacks any ambiguity—each two-digit hexadecimal number has 256 possibilities (16^2), the same number of possibilities encoded in an eight-bit octet.

Let's say you wanted to create a predicate that contains that binary stream. To do so, you'd first need to encode it:

```
var buffer = new Buffer([0x49, 0x11]);
console.log(buffer.toString('base64'));     1
```

- **1 Prints out "SRE="**

Now the predicate definition is conceptually the same as it is for text, as the following listing demonstrates.

Listing 8.6. Using a binary `contains` predicate

```
{
  "protocol": "tcp",
  "port": 3000,
  "mode": "binary",                                    1
  "stubs": [
    {
      "predicates": [{
        "contains": { "data": "SRE=" }                 2
      }],
      "responses": [{
        "is": { "data": "TWF0Y2hlZAo=" }               3
      }]
    },
    {
      "responses": [{
        "is": { "data": "RGlkIG5vdCBtYXRjaAo=" }       4
      }]
    }
  ]
}
```

- ***1* Puts the imposter in binary mode**
- ***2* 0x49 0x11**
- ***3* "Matched"**
- ***4* "Didn't match"**

If we add an octet—say 0x10—to our binary stream, the `contains` predicate still matches. The binary stream 0x10 0x49 0x11 encodes to "EEkR," which clearly doesn't contain the text "SRE=". Had you not configured the imposter to be in binary mode, mountebank would've performed a simple string operation, and the predicate wouldn't match. By switching to binary mode, you're telling mountebank to instead decode the predicate (`SRE=`) to a binary array ([0x49, 0x11]) and to see if the incoming binary stream ([0x10, 0x49, 0x11]) contains those octets. It does, so the predicate matches. You can test on the command line by using the `base64` utility, which ships by default with most POSIX shells (for example, Mac and Linux):

```
echo EEkR | base64 --decode | nc localhost 3000
```

You get a response of "Matched," which corresponds to the first response.

Nearly all of the predicates work this way: by matching against a binary array. The one exception is `matches`. Regular expressions don't make sense in a binary world; the metacharacters don't translate. In practice, `contains` is probably the most useful binary predicate. It turns out that many binary RPC protocols encode a function name inside the request. The parameters may be serialized objects that are hard to match, but the function name is encoded text. We look at a real-world example next.

8.5. VIRTUALIZING A .NET REMOTING SERVICE

Back in the days of yore, a *town crier* often made public announcements. A crier would ring a handbell and shout "Hear ye! Hear ye!" to get everyone's attention before making the announcement. This neatly solved the problem of delivering a message to a public that was still largely illiterate.

Providing a town crier RPC service[2] may come off as a little antiquated, but it helps get your mindset into those halcyon days of yesteryear when we thought making remote function calls look like in-process function calls was a good thing.[3] .NET Remoting wasn't the first attempt at creating a largely transparently distributed RPC mechanism, but it did have a brief period of popularity and is representative of the broader class of RPC protocols you are likely to run across in the enterprise.

2 OK, the scenario may not be quite as real-world as advertised, but the protocol is....

3 Peter Deutsch wrote that nearly everyone who first builds a distributed application makes a key set of assumptions, all of which are false in the long run and inevitably cause big trouble. See https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing.

8.5.1. Creating a simple .NET Remoting client

.NET Remoting allows you to call a remote method like you would a local method using the .NET framework. For example, let's assume that to make a pronouncement, you have to fill in an `AnnouncementTemplate`:

```
[Serializable]                                          1
public class AnnouncementTemplate
{
    public AnnouncementTemplate(string greeting,
        string topic)
```

```
    {
        Greeting = greeting;                    2
        Topic = topic;                          3
    }

    public string Greeting { get; }             4
    public string Topic { get; }                4
}
```

- *1* **Ensures that it can be passed over the wire**
- *2* **So you can change “Hear ye!” to “Oyez!”**
- *3* **The topic to be announced**
- *4* **Public getters**

Don’t worry if you’re not a C# expert. This code is as simple as it gets in most enterprise applications, which often are written in Java or C#. It creates a basic class that accepts the greeting and topic for the pronouncement and exposes them as read-only properties. The only interesting nuance is the `[Serializable]` attribute at the top. That’s a bit of C# magic that allows the object to be passed between processes.

Once you create an `AnnouncementTemplate`, you’ll pass it to the `Announce` method of the `Crier` class, as defined in the following listing.[4]

4

The source code for this example is considerably more complicated than most other examples in this book. As always, you can download it at https://github.com/bbyars/mountebank-in-action. In addition, many of the command line examples in the book have shown a bit of a bias toward macOS and Linux. This example is geared toward Windows and will require additional effort to run on other operating systems.

Listing 8.7. The `Crier` class definition

```
public class Crier : MarshalByRefObject                1
{
    public AnnouncementLog Announce(
        AnnouncementTemplate template)
    {
        return new AnnouncementLog(                     2
            $"{template.Greeting}! {template.Topic}");
    }
}
```

- *1* **Allows serialization over the wire**
- *2* **Returns a log capturing the announcement**

The `Crier` class inherits from `MarshalByRefObject`. Again, that is largely irrelevant to the example, except for the fact that it's the bit of magic that allows an instance of the `Crier` class to be called from a remote process. The `Announce` method formats the greeting and topic into a single string (that strange `$"{template.Greeting}! {template.Topic}"` line is C#'s string interpolation) and returns it wrapped inside an `AnnouncementLog` object, which looks like this:

```
[Serializable]                                          1
public class AnnouncementLog
{
    public AnnouncementLog(string announcement)
    {
        When = DateTime.Now;                            2
        Announcement = announcement;
    }

    public DateTime When { get; }
    public string Announcement { get; }

    public override string ToString()
    {
        return $"({When}): {Announcement}";             3
    }
}
```

- ***1* Makes it remotely accessible**
- ***2* Captures the time of the announcement**
- ***3* Formats a log entry for the announcement**

For our purposes, the `AnnouncementTemplate`, `Crier`, and `AnnouncementLog` form the entirety of the domain model. You could've simplified it to the `Crier` class, but we used to think that passing entire object graphs over the wire was a good idea, and adding a couple of simple classes that the `Crier` uses—one as input, one as output—helps make the example ever so slightly more realistic.

You can call `Crier` locally, but that's kind of boring, and more in the realm of traditional mocking tools should you decide to test it. Instead, you will call a remote `Crier` instance. The source repo for this book shows how to code a simple server that listens on a TCP socket and acts as a remote `Crier`. We'll focus on the client, testing it by virtualizing the server. To do that, your virtual service needs to respond like a .NET TCP Remoting service would.

The following listing shows the client you'll test. It represents a gateway to a remote

`Crier` instance.

Listing 8.8. A gateway to a remote `Crier`

```
public class TownCrierGateway
{
    private readonly string url;

    public TownCrierGateway(int port)
    {
        url = $"tcp://localhost:{port}/TownCrierService";    1
    }

    public string AnnounceToServer(
        string greeting, string topic)
    {
        var template = new AnnouncementTemplate(
            greeting, topic);
        var crier = (Crier)Activator.GetObject(              2
            typeof(Crier), url);
        var response = crier.Announce(template);             3
        return $"Call Success!\n{response}";                 4
    }
}
```

- ***1* URL to the remote service**
- ***2* Gets a remote object reference**
- ***3* Makes the (remote) method call**
- ***4* Adds metadata to the response**

Notice that the call to `crier.Announce` looks like a local method call. It's not. This is the magic of .NET Remoting. The line above retrieves a remote reference to the object based on the URL in the constructor. That desire to make remote function calls look like local function calls is highly representative of this era of distributed computing.

8.5.2. Virtualizing the .NET Remoting server

All the `TownCrierGateway` class does is add a success message to the response of the remote call. That's enough for you to write a test without getting lost in too much unnecessary complexity. You could write the test two ways, assuming you aim to virtualize the remote service.

The first way is to create a mountebank stub that proxies the remote service and captures the response. You could replay the response in your test.

The second way is much cooler. You could *create* the response (as in an instance of the `AnnouncementLog` class) in the test itself, as you would with traditional mocking tools, and have mountebank return it when the client calls the `Announce` method. *Much* cooler.

Fortunately, Matthew Herman has written an easy to use mountebank library for C# called MbDotNet.[5] Let's use it to create a test fixture. I like to write tests using the Law of Wishful Thinking, by which I mean I write the code I want to see and figure out how to implement it later. This allows my test code to clearly specify the intent without getting lost in the details. In this case, I want to create the object graph that mountebank will return inside the test itself and pass it off to a function that creates the imposter on port 3000 using a `contains` predicate for the remote method name. That's a lot to hope for, but I've wrapped it up in a function called `CreateImposter`, as shown in the following listing.

5

An entire ecosystem of these client bindings exists for mountebank. I do my best to maintain a list at http://www.mbtest.org/docs/clientLibraries, but you can always search GitHub for others. Feel free to add a pull request to add your own library to the mountebank website.

Listing 8.9. Basic test fixture using MbDotNet

```
[TestFixture]
public class TownCrierGatewayTest
{
    private readonly MountebankClient mb =                1
        new MountebankClient();                           1

    [TearDown]                                            2
    public void TearDown()                                2
    {                                                     2
        mb.DeleteAllImposters();                          2
    }                                                     2
    [Test]
    public void ClientShouldAddSuccessMessage()
    {
        var stubResult = new AnnouncementLog("TEST");     3
        CreateImposter(3000, "Announce", stubResult);     3
        var gateway = new TownCrierGateway(3000);         3

        var result = gateway.AnnounceToServer(            4
            "ignore", "ignore");                          4

        Assert.That(result, Is.EqualTo(                   5
            $"Call Success!\n{stubResult}"));             5
    }
}
```

- ***1*** **MbDotNet's gateway to the mountebank REST API**
- ***2*** **Deletes all imposters after every test**
- ***3*** **Arrange**
- ***4*** **Act**
- ***5*** **Assert**

This fixture uses NUnit annotations[6] to define a test. NUnit ensures that the `TearDown` method will be called after every test, which allows you to elegantly clean up after yourself. When you create your test fixture, you create an instance of the mountebank client (which assumes `mb` is already running on port 2525) and remove all imposters after every test. This is the typical pattern when you use mountebank's API for functional testing.

6

A popular C# testing framework; see http://nunit.org/.

The test itself uses the standard Arrange-Act-Assert pattern of writing tests introduced back in chapter 1. Conceptually, the *Arrange* stage sets up the system under test, creating the `TownCrierGateway` and ensuring that when it connects to a virtual service (on port 3000), the virtual service responds with the wire format for the object graph represented by `stubResult`. The *Act* stage calls the system under test, and the *Assert* stage verifies the results. This is nearly identical to what you would do with traditional mocking tools.

Wishful thinking only gets you so far. MbDotNet simplifies the process of wiring up your imposter using C#. You'll delay only the serialization format for the response under a wishfully-thought-of method I have named `Serialize`:

```
 private void CreateImposter(int port,
    string methodName, AnnouncementLog result)
{
    var imposter = mb.CreateTcpImposter(
        port, "TownCrierService", TcpMode.Binary);
    imposter.AddStub()
        .On(ContainsMethodName(methodName))          1
        .ReturnsData(Serialize(result));             2
    mb.Submit(imposter);                             3
}

private ContainsPredicate<TcpPredicateFields> ContainsMethodName(
```

```
        string methodName)
    {
        var predicateFields = new TcpPredicateFields
        {
            Data = ToBase64(methodName)
        };
        return new ContainsPredicate<TcpPredicateFields>(
            predicateFields);
    }

    private string ToBase64(string plaintext)
    {
        return Convert.ToBase64String(
            Encoding.UTF8.GetBytes(plaintext));
    }
```

- ***1* Adds predicate**
- ***2* Adds response**
- ***3* Calls the REST API (We'll get to the Serialize method soon.)**

The `CreateImposter` and `ContainsMethodName` methods uses the MbDotNet API, which is a simple wrapper around the mountebank REST API. The REST call is made when you call `mb.Submit`. The `ToBase64` method uses the standard .NET library calls to encode a string in Base64 format.

All that's left is to fill in the `Serialize` method. This is the method that has to accept the object graph you want your virtual service to return and transform it into the stream of bytes that looks like a .NET Remoting response. That means understanding the wire format of .NET Remoting.

That's hard.

The good news is that, with many popular RPC protocols, someone else has usually done the hard work for you. For .NET Remoting, that someone else is Xu Huang, who has created .NET Remoting parsers for .NET, Java, and JavaScript.[7] You'll use the .NET implementation to create the `Serialize` function.

7

See https://github.com/wsky/RemotingProtocolParser.

The code appears in listing 8.10. Don't try too hard to understand it all. The point isn't to teach you the wire format for .NET Remoting. Instead, it's to show that, with a little bit of work, you can usually create a generalized mechanism for serializing a stub response

into the wire format for real-world RPC protocols. Once you have done the hard work, you can reuse it throughout your test suite to make writing tests as easy as creating the object graph you want the virtual service to respond with and letting your serialization function do the work of converting it to an RPC-specific format.

Listing 8.10. Serializing a stub response for .NET Remoting

```
public string Serialize(Object obj)
{
    var messageRequest = new MethodCall(new[] {                    1
        new Header(MessageHeader.Uri,
            "tcp://localhost:3000/TownCrier"),
        new Header(MessageHeader.MethodName,
            "Announce"),
        new Header(MessageHeader.MethodSignature,
            SignatureFor("Announce")),
        new Header(MessageHeader.TypeName,
            typeof(Crier).AssemblyQualifiedName),
        new Header(MessageHeader.Args,
            ArgsFor("Announce"))
    });
    var responseMessage = new MethodResponse(new[]                 c
    {
        new Header(MessageHeader.Return, obj)
    }, messageRequest);

    var responseStream = BinaryFormatterHelper.SerializeObject(
        responseMessage);
    using (var stream = new MemoryStream())
    {
        var handle = new TcpProtocolHandle(stream);
        handle.WritePreamble();                                    3
        handle.WriteMajorVersion();                                3
        handle.WriteMinorVersion();                                3
        handle.WriteOperation(TcpOperations.Reply);                3
        handle.WriteContentDelimiter(                              3
            TcpContentDelimiter.ContentLength);                    3
        handle.WriteContentLength(                                 3
            responseStream.Length);                                3
        handle.WriteTransportHeaders(null);                        3
        handle.WriteContent(responseStream);                       4
        return Convert.ToBase64String(                             5
            stream.ToArray());
    }
}

private Type[] SignatureFor(string methodName)                     6
{
    return typeof(Crier)
        .GetMethod(methodName)
        .GetParameters()
        .Select(p => p.ParameterType)
```

```
            .ToArray();
    }

    private Object[] ArgsFor(string methodName)              6
    {
        var length = SignatureFor(methodName).Length;
        return Enumerable.Repeat(new Object(), length).ToArray();
    }
```

- ***1*** **Request metadata**
- ***2*** **Wraps response**
- ***3*** **Writes response metadata**
- ***4*** **Writes response (with request metadata)**
- ***5*** **Converts to Base64**
- ***6*** **Supports RPC methods other than Announce**

The `SignatureFor` and `ArgsFor` methods are simple helper methods that use .NET reflection (which lets you inspect types at runtime) to make the `Serialize` method general purpose. The request metadata expects some information about the remote function signature, and those two methods allow you to dynamically define enough information to satisfy the format. The rest of the `Serialize` method uses Xu Huang's library to wrap your stub response object with the appropriate metadata, so when mountebank returns it over the wire, your .NET Remoting client will see it as a legitimate RPC response.

Remember the key goal of mountebank: to make easy things easy and hard things possible. The fact that, with a little bit of underlying serialization code, you can elegantly stub out binary .NET Remoting (and some of its cousins) over the wire is a killer feature.

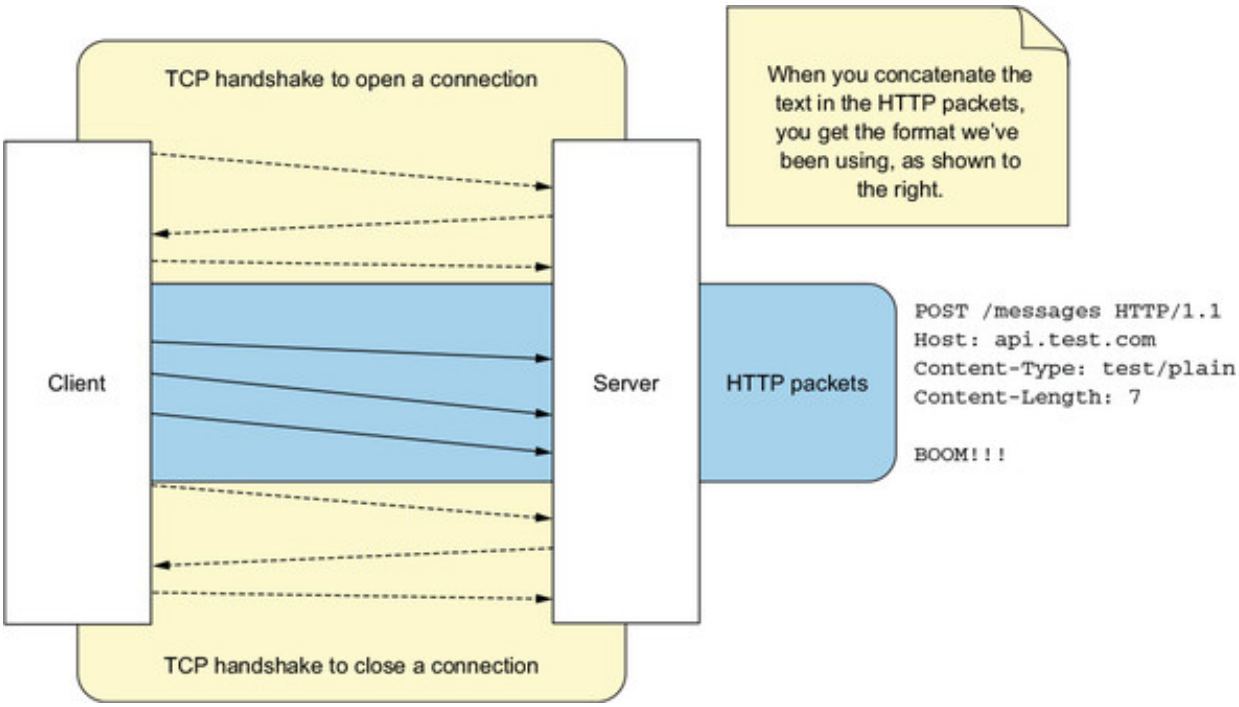
In case you have forgotten how cool that is, I suggest you look back at listing 8.9 and see how simple the test is.

8.5.3. How to tell mountebank where the message ends

There's one other bit of complexity you have to deal with to fully virtualize an application protocol using mountebank's TCP protocol. We hinted at it back in chapter 3, when we looked at how an HTTP server knows when an HTTP request is complete. You may recall a figure that looked like figure 8.5.

Figure 8.5. Using `Content-Length` to wrap multiple packets into one HTTP

request



As a transport protocol, TCP opens and closes a new connection using a *handshake*. That handshake is transparent to application protocols. TCP then takes the application request and chunks it into a series of packets, sending each packet over the wire. A packet will range between 1,500 and around 64,000 bytes, though smaller sizes are possible. You'll get the larger packet size when you test on your local machine (using what's called the *loopback* network interface), whereas lower level protocols like Ethernet use smaller packet sizes when passing data over the network.

Because a logical application request may span multiple packets, the application protocol needs to know when the logical request ends. HTTP often uses the `Content-Length` header to provide that information. Because this header occurs early in the HTTP request, the server can wait until it receives enough bytes to satisfy the given length, regardless of how many packets it takes to deliver the full request.

Every application protocol must have a strategy for determining when the logical request ends. Mountebank uses two strategies:

- The default strategy, which assumes a one-to-one relationship between a packet and a request
- Receiving enough information to know when the request ends

The examples have worked so far because you've only tested with short requests. You will change that with a simple `proxy`, saved as `remoteCrierProxy.json`, as shown in the following listing.

Listing 8.11. Creating a TCP `proxy` to a .NET Remoting server

```
{
  "protocol": "tcp",
  "port": 3000,
  "mode": "binary",
  "stubs": [{
    "responses": [{
      "proxy": { "to": "tcp://localhost:3333" }
    }]
  }]
}
```

The source code for this book includes the executable for the .NET Remoting server. You give it the port to listen to when you start it up:

```
Server.exe 3333
```

You start the mountebank server in the usual way:

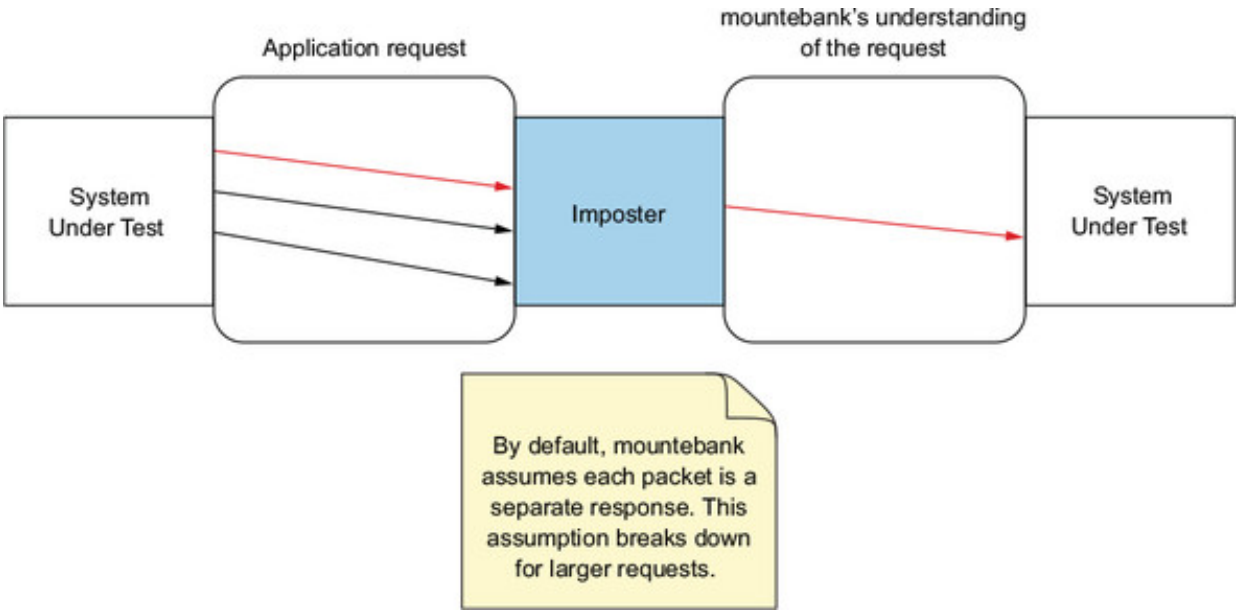
```
mb --configfile remoteCrierProxy.json
```

Finally, if you start the .NET Remoting client on port 3000, it's configured by default to send a request greeting and topic that'll exceed the size of a single packet:

```
Client.exe 3000
```

You can see in the mountebank logs that it tried to proxy, but the server didn't respond, and the client threw an error. By default, mountebank grabs the first packet and assumes it's the entire request. It passes it to the server—a real, bona fide .NET Remoting server—which looks inside the packet and sees that it should expect more packets to come for the request, so it continues to wait. Mountebank, thinking it has seen the entire request, tries to respond. The whole process blows up (figure 8.6).

Figure 8.6. Mismatched expectations around when the request ends



Once your request reaches a certain size, you have to opt for the second strategy: telling mountebank when the request ends. The imposter will keep an internal buffer. Every time it receives a new packet, it adds the packet data to the buffer and passes the entire buffer to a JavaScript function that you define, which returns `true` if the request is complete or `false` otherwise.

You pass in the function as the `endOfRequestResolver`. For this example, you'll add it using a template to include the function in a separate file called resolver.js, as shown in the following listing.

Listing 8.12. Adding an `endOfRequestResolver`

```
{
  "protocol": "tcp",
  "port": 3000,
  "mode": "binary",
  "endOfRequestResolver": {
     "inject": "<%- stringify(filename, 'resolver.js') %>"
  },
  "stubs": [{
     "responses": [{
       "proxy": { "to": "tcp://localhost:3333" }
    }]
  }]
}
```

.NET Remoting embeds a content length in the metadata for the request. You can use that in your function to determine if you've collected all the packets for the request or not. You'll once again rely on Xu Huang's parsing library, which includes a Node.js implementation, to do the heavy lifting. As before, the intention isn't to learn everything about .NET Remoting; it's to show how you would virtualize a real-world

application protocol. Don't worry too much about the details of the message format. The essential part is that you grab the content length from the message and test it against the length of the buffer mountebank passes you to see whether you've received the entire request, as shown in the following listing.

Listing 8.13. The function to determine if you have seen the entire request yet

```
function (requestData, logger) {                                    1
  var path = require('path'),
    parserPath = path.join(process.cwd(),                           2
➥ '/../RemotingProtocolParser/nodejs/lib/remotingProtocolParser'),
    r = require(parserPath).tcpReader(requestData);


  logger.debug('Preamble: %s', r.readPreamble());
  logger.debug('Major: %s', r.readMajorVersion());
  logger.debug('Minor: %s', r.readMinorVersion());
  logger.debug('Operation: %s', r.readOperation());
  logger.debug('Delimiter: %s',
    r.readContentDelimiter());
  logger.debug('ContentLength: %s',                                 3
    r.readContentLength());                                         3
  logger.debug('Headers: %s',
    JSON.stringify(r.readHeaders()));

  var expectedLength = r.offset + r.contentLength + 1;              4
  logger.info('Expected length: %s, actual length: %s',
    expectedLength, requestData.length);
  return requestData.length >= expectedLength;                      5
}
```

- ***1*** **requestData is a Node.js Buffer object.**
- ***2*** **Includes Xu Huang's library**
- ***3*** **Refers to the length of the content section, not the entire message**
- ***4*** **Calculates the expected length**
- ***5*** **Tests buffer length against expected**

The parsing library isn't published as an `npm` module. If it was, you could install it locally and include it without referencing a specific file path. In your case, you cloned Huang's repository according to the path expected in the second line of the function.[8]

8 I've done this in the source code repo for this book, including a complete copy of the library.

The parsing library doesn't support random access, so you can't ask it the content length and compare that against your request buffer. Instead, it maintains a stateful `offset` and expects that you read all the metadata fields in order. To help you debug, I wrapped those metadata fields in a `logger.debug` function. You'll be able to see them in the mountebank logs if you run with the `--loglevel debug` command-line flag.

Now that you've written your function, you can try the proxy again. This time, because you're using a JavaScript function, you have to pass the `--allowInjection` flag:

```
mb --configfile imposter.json --allowInjection
```

Restart the server on port 3333 and run the client again, pointing to the mountebank proxy:

```
Client.exe 3000
```

This time, everything works. You now have a fully functional virtual server for .NET Remoting. Congratulations! You've completed the hardest example in the entire book.

Hard, but possible.

And with that, you've now completed your tour of mountebank. But knowing how to use a tool isn't the same as knowing *when* you should use it. That's what the next chapter is about.

Another example: Java serialization over a Mule ESB

In late 2013, I was working at a major airline company. Several years earlier, the website had been rewritten to communicate with a service tier over a Mule enterprise service bus (ESB). The ESB connector communicated over TCP and returned a serialized Java object graph. Unfortunately, passing raw objects over the wire created a tight coupling between the web tier and the service tier. It also ran a multibillion-dollar website for most of a decade. Production-hardened enterprise software rarely looks like the beautiful architectures you read about.

I was on a team creating REST APIs for a new mobile app that needed to go out before the website could be replaced, so our APIs had to integrate with the service tier. Although we had a top-notch team comfortable with automated testing, the friction of testing without also breaking the web tier was so painful that we gave up. We wrote

automated tests when we could, but usually it was too hard, and bugs started creeping in.

Most of this book has described mountebank for HTTP, but HTTP wasn't the first protocol mountebank supported. I created mountebank to test a binary Mule ESB TCP connector, serving up Java objects in our tests in much the same way we looked at for .NET Remoting. At the time, a number of quality open source virtualization tools were available for HTTP, but none could stub out a binary TCP protocol. That's largely still true today.

SUMMARY

- In mountebank, protocols are responsible for transforming a network request into a JSON request for predicate matching, as well as taking a mountebank JSON response and transforming it into a network response.
- Mountebank supports adding an application protocol on top of its TCP protocol. All predicates, response types, and behaviors continue to work with the TCP protocol; only the JSON structure for requests and responses differs.
- Mountebank supports binary payloads by Base64 encoding the data. You have to switch the imposter `mode` to `binary` for mountebank to correctly handle the encoding.
- Once you figure out how to serialize an object graph into the wire format expected by an RPC protocol, you can write tests that look similar to ones that use traditional stubbing tools.
- By default, mountebank assumes each incoming packet represents an entire request when using the TCP protocol. To let mountebank know when the request ends, you can pass in an `endOfRequestResolver` JavaScript function.

Part 3. Closing the Loop

Now that you have the full breadth of mountebank functionality under your belt, part 3 puts its usage in context.

Service virtualization is a powerful tool, but, like any tool, it has its limits. In chapter 9, we explore it in the context of continuous delivery. We'll build a test pipeline from start to finish for some of the microservices we've looked at previously in this book and discuss where service virtualization fits and where it doesn't. We'll also look at how to gain additional confidence in your test suite with contract tests that give you lightweight validation that play together well with your services without going down the path toward full end-to-end testing.

We close out the book by looking at performance testing, always a difficult subject and one made even more so by microservices. The need to understand your service's performance characteristics in a networked environment is challenged by the cost and complexity of securing an end-to-end environment for testing. Service virtualization is a natural fit for performance testing and combines many of the features we've looked at previously, including proxies and behaviors.

Chapter 9. Mountebank and continuous delivery

This chapter covers

- A basic refresher on continuous delivery
- Testing strategy for continuous delivery and microservices
- Where service virtualization applies inside a broader testing strategy

A sysadmin, a DBA, and a developer walk into a bar. The sysadmin orders a light lager to maximize uptime, the DBA orders a 30-year-aged single malt to avoid undue adulteration, and the developer orders a Pan Galactic Gargle Blaster because it hasn't been invented yet. An hour later, the DBA has gone home already, the developer has moved on to a more modern bar, and the slightly wobbly and heavily overutilized sysadmin is holding down the fort, while also holding a lager in one hand, a single malt in another hand, and a Pan Galactic in another hand.[1]

1

In *The Hitchhikers Guide to the Galaxy*, Douglas Adams describes the Pan Galactic Gargle Blaster as the alcoholic equivalent of a mugging—expensive and bad for the head.

Traditional siloed organizational structure forces a complicated dance to get anything done. It's no surprise that, in large enterprises, IT and the business rarely have a healthy relationship. Historically, the common approach to improving the situation was to add more process discipline, which further complicated the dance, making it harder to release code into production (and, by consequence, reduced value to customers). Having increasingly well-defined handoffs between a developer, a DBA, and a sysadmin exemplifies process discipline. Every time you fill out a database schema change request form or an operational handoff document, you have seen process discipline in action.

Continuous delivery changes the equation by emphasizing *engineering discipline* over *process discipline*. It's about automating the steps required to build confidence so that the business can release new code on demand. Although engineering discipline encompasses a wide spectrum of practices, testing plays a central role. In this chapter,

we look at a sample testing strategy for a microservices world and show where service virtualization does and doesn't fit.

9.1. A CONTINUOUS DELIVERY REFRESHER

Jez Humble and Dave Farley wrote *Continuous Delivery* to capture the key practices they saw enabling the rapid delivery of software. In chapter 1, I showed you how the traditional process discipline of centralized release management and toll gates increases congestion and slows delivery. The emphasis is on safety, providing additional checks to increase confidence that the software being delivered will work.

In contrast, continuous delivery (CD) focuses on automation, emphasizing safety, speed, *and* sustainability of delivering software. It requires the code to be in a deployable state at all times, forcing you to abandon the ideas of *dev complete*, *feature complete*, and *hardening iterations*. Those concepts are holdovers from the world of yesteryear, in which we papered over a lack of engineering discipline by adding more layers of process.

A glossary of terms surrounding continuous delivery

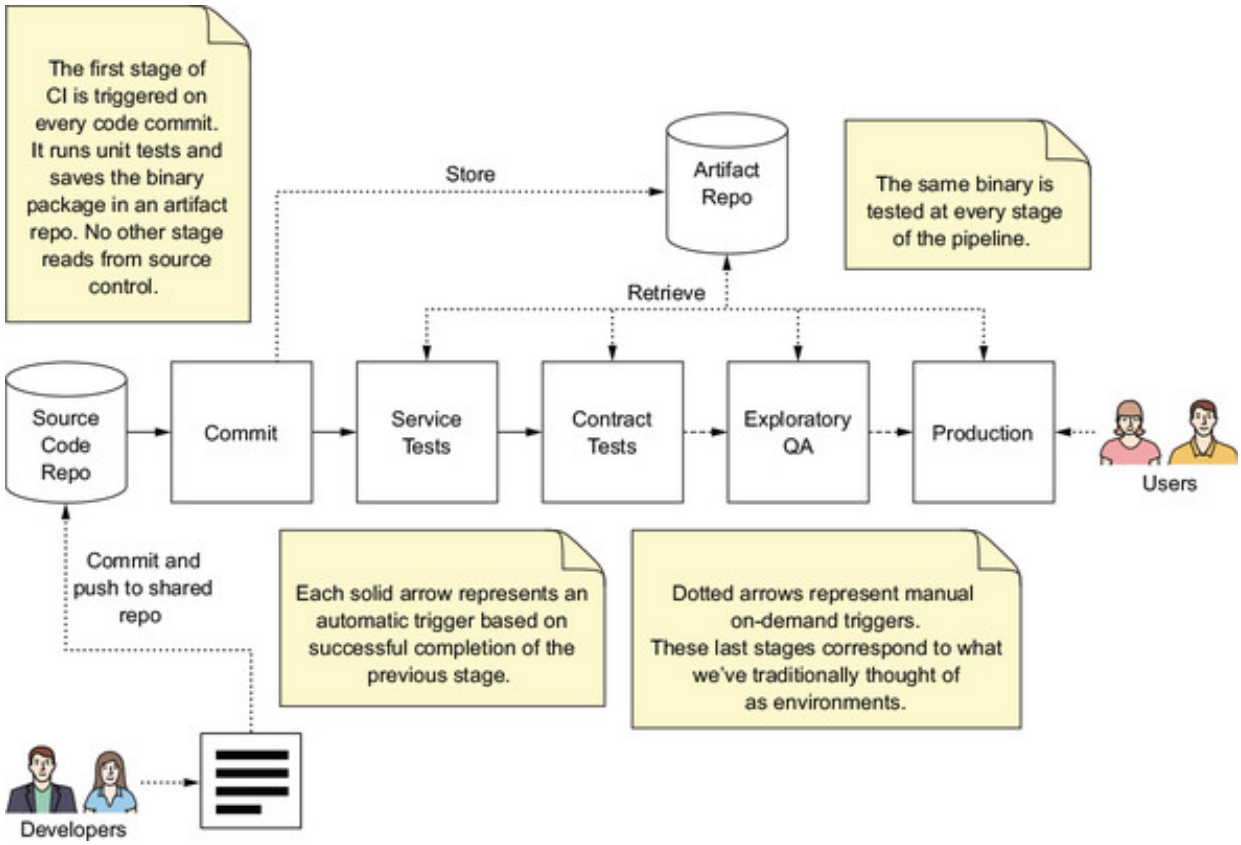
I introduce several important terms in this chapter:

- ***Continuous integration*—** Although continuous integration (CI) is often confused with running an automated build after every commit through a tool like Jenkins, it's actually the practice of ensuring that your code is merged with and works with everyone else's on a continual basis (at least once a day).
- ***Continuous delivery*—** The set of software development practices that ensures code is always releasable. The full spectrum of CD practices ranges from developer-facing techniques like feature toggles, which provide a way of hiding code that's still a work in progress, to production-facing approaches like monitoring and canary testing, which scales up a release to a customer base over time. In between comes testing, the focus of this book.
- ***Deployment pipeline*—** The path code takes from the time it's committed to the time it reaches production.
- ***Continuous deployment*—** An advanced type of continuous delivery that removes all manual interventions from the deployment pipeline.

In CD, every commit of the code either fails the build or can be released to production. There's no need to decide up front which commit represents the release version. Although still common, that approach encourages sloppy engineering practices. It enables you to commit code that cannot be released to production, with the expectation that you'll fix it later. That attitude requires IT to own the timing of software delivery, taking control out of the hands of the business and the product manager.

The core organizing concept that makes CD possible is the *deployment pipeline*. It represents the value stream of the code's journey from commit to production and is often directly represented in continuous integration (CI) tools (figure 9.1).

Figure 9.1. A deployment pipeline defines the path from commit to production.



Every code commit automatically triggers a build, which usually includes compilation, running unit tests, and static analysis. A successful build saves off a package in an artifact repository—usually a binary artifact, even if it's just a tarball of source code for interpreted languages like Ruby and JavaScript. Every set of verifications downstream runs against a deployed instance of that package, until it ultimately reaches production.

The path that code takes on its way to providing value to real users varies from organization to organization and even between teams within the same organization. Much of it is defined by how you decide to test your application.

9.1.1. Testing strategy for CD with microservices

Testing in a very large-scale distributed setting is a major challenge.

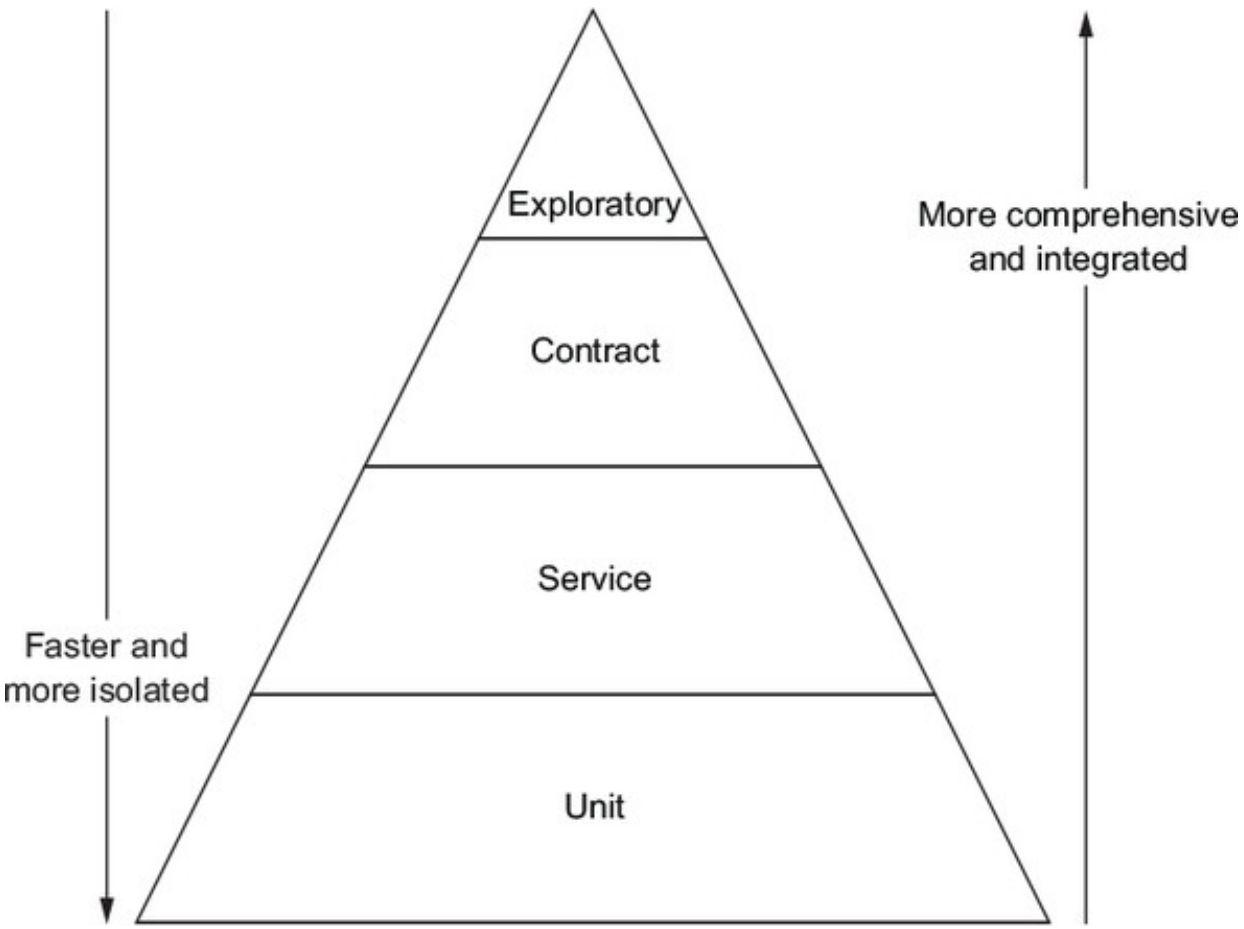
Werner Vogels, Amazon CTO

A common approach to visualizing testing strategy comes in the form of a pyramid. The visual works because it acknowledges that confidence comes from testing at multiple layers. It also shows that there's value in pushing as much of the testing as possible into the lower levels, where tests are both easier to maintain and faster to run. As you move to higher levels, the tests become harder to write, to maintain, and to troubleshoot when they break. They're also more comprehensive and often better at catching difficult bugs. Each team will need to customize a test pyramid to its needs, but you can think of a template for microservices that looks like figure 9.2.[2]

2

You also may be interested in Toby Clemson's description of the types of testing for microservices at http://martinfowler.com/articles/microservice-testing/.

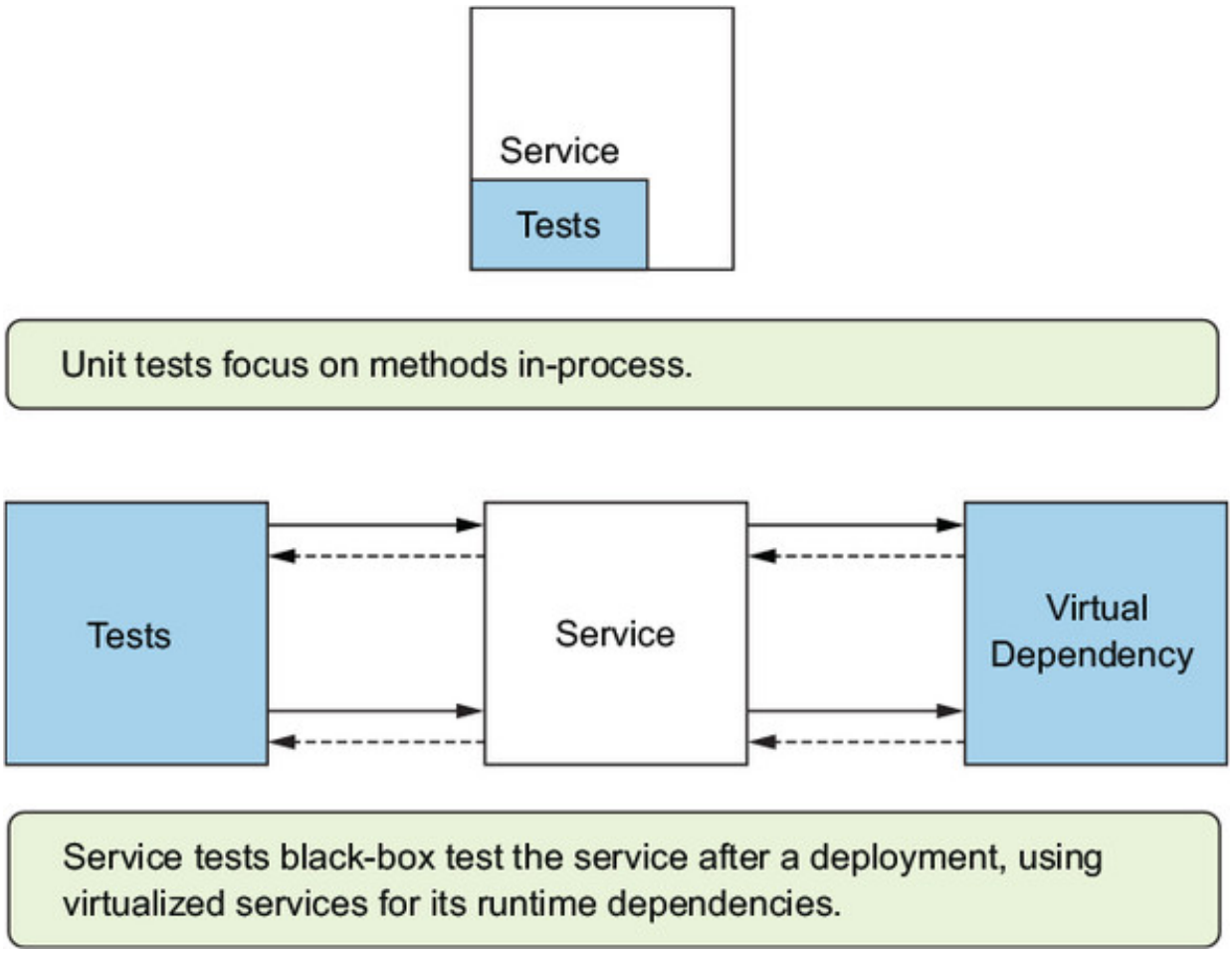
Figure 9.2. Simplified test pyramid for microservices



People have argued endlessly over what makes a unit test different from higher level tests, but for the purposes of this diagram, the key difference is that you should be able to run a unit test without deploying your service into a runtime. That makes unit tests

in-process and independent of anything from the environment (see figure 9.3).

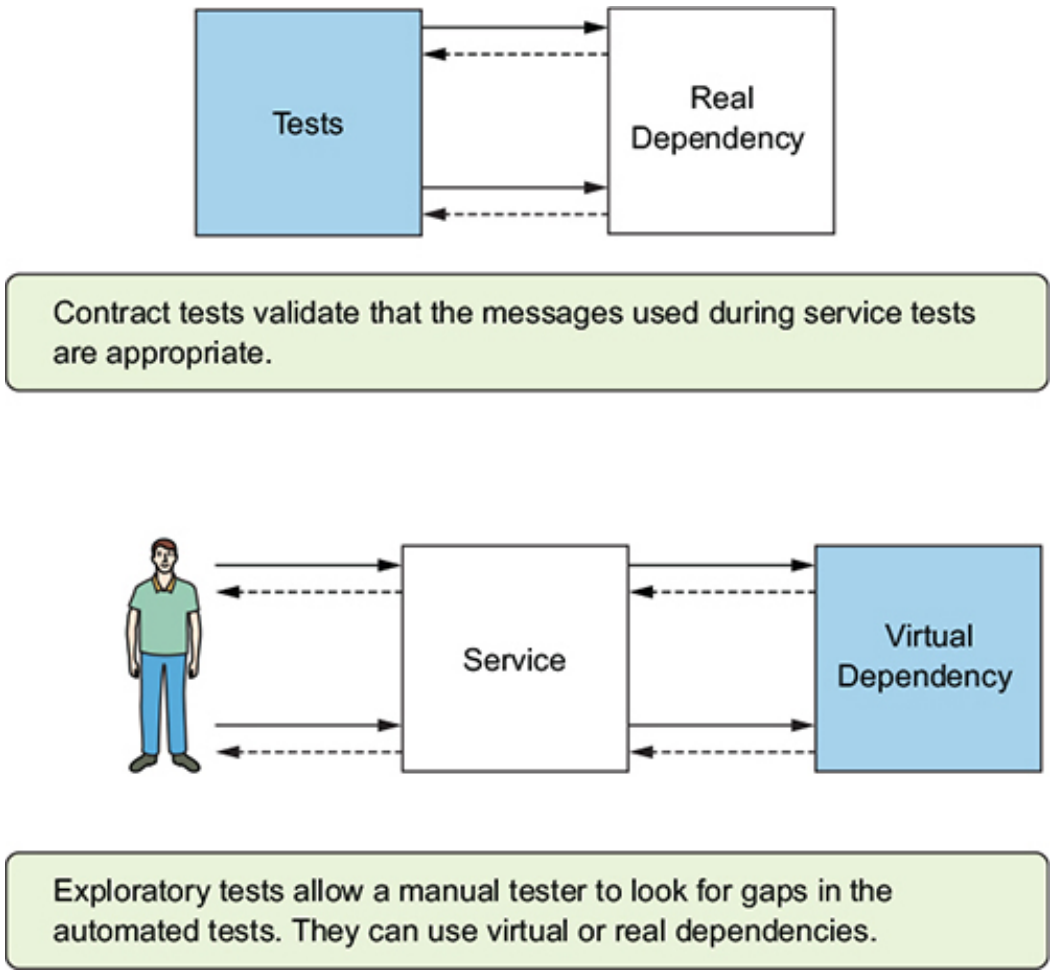
Figure 9.3. The basic structure for unit and service tests



Though there's some different terminology out there, I've used the term *service test* to describe a black-box test that validates your service's behavior over the wire. Such tests do require a deployment, but you use service virtualization to maintain isolation from your runtime dependencies. This layer allows you to do out-of-process, black-box testing while maintaining determinism. Service virtualization enables you to remove nondeterminism from your tests by allowing each test to control the environment it runs in, as shown in figure 9.3.

You should be able to test the bulk of the behavior of your service through a combination of unit tests and service tests. They let you know that your service behaves correctly, assuming certain responses from its dependencies, but they don't guarantee that those stubbed responses are appropriate. Contract tests give you validation that breaking-contract-level changes haven't occurred (see figure 9.4). Service tests say, in effect, that *if* the service gets these responses from its dependencies, *then* it behaves correctly. Contract tests validate that the service does, in fact, get those responses. Good contract tests avoid deep behavioral testing of the dependencies—you should test them independently—but give you confidence in your stubs.

Figure 9.4. The basic structure for contract and exploratory tests



I've included exploratory testing as part of the test pyramid because most organizations find some value in manual testing. Good exploratory testers follow their nose to find gaps in an automated test suite. Such tests can be integrated or rely on service virtualization to test certain edge cases. Figure 9.4 shows exploratory testing using service virtualization.

Other types of testing exist that don't fit as well in the test pyramid metaphor. Cross-functional requirements like security, performance, and failover for availability often require specialized testing and are less about the behavior of the system than about its resiliency. Performance testing is an area where service virtualization shines, as it allows you to replicate the performance of your dependencies without requiring a fully integrated, production-like environment to run in. In chapter 10, we explore how service virtualization enables performance and load testing.

Finally, you should never forget that error prevention is only one piece of testing strategy. The rapid release cycles of microservices encourage you to invest heavily in error detection and remediation as well as prevention, as they contribute to your overall confidence in releasing software. Although error detection and remediation are not the focus of this book, companies that have used microservices effectively generally stage

their releases, such that only a small percentage of users can see the new release at first. Robust monitoring detects whether the users experience any problem, and rolling back is as easy as switching those users to the code everyone else is using. If no problems are detected, the release system will switch more and more users to the new code over time until 100% of users are using the release, at which time you can remove the previous release.[3] Advanced monitoring allows you to detect errors before your users do. Although your testing strategy is a key component of continuous delivery, engineering discipline significantly increases the scope of automation.

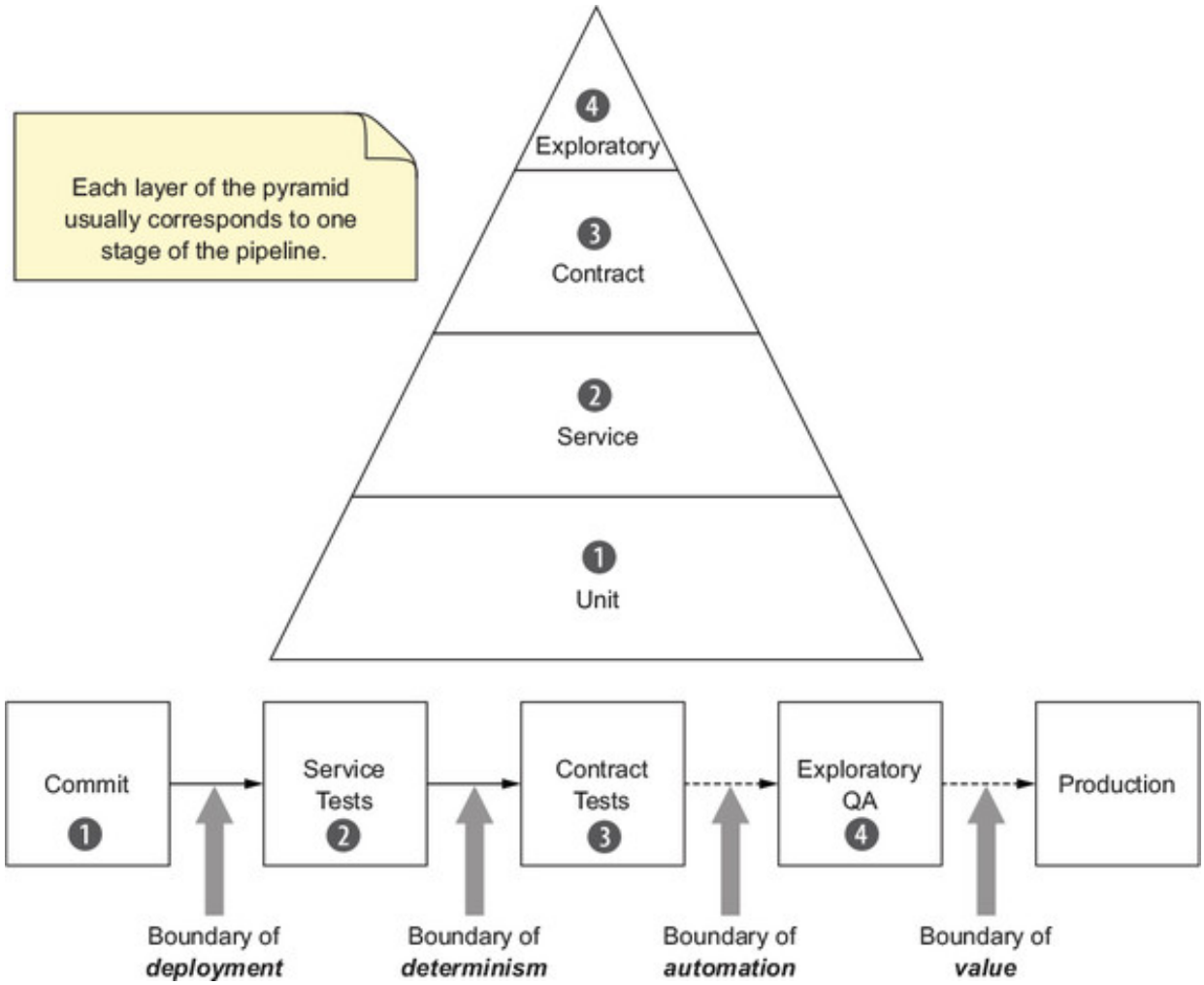
3

This is called canary testing. You can read more about it at https://martinfowler.com/bliki/CanaryRelease.html.

9.1.2. Mapping your testing strategy to a deployment pipeline

Whatever your particular test pyramid looks like, mapping it to your deployment pipeline is generally pretty straightforward (figure 9.5).

Figure 9.5. Mapping your test pyramid to a deployment pipeline



I like to think of boundary conditions moving from one stage to the next. In figure 9.5, I've shown the following boundaries:

- The boundary of *deployment* represents the first time you've deployed the application (or service). All tests to the left are run in-process; all tests to the right are run out-of-process and implicitly test the deployment process itself, as well as the application.
- The boundary of *determinism* represents the first time you've integrated your application into other applications. Tests to the right of this boundary may fail because of environmental conditions. Tests to the left of this boundary should fail only for reasons entirely under the application team's control.
- The boundary of *automation* represents where you switch to manual, exploratory testing. (Note that the deployment itself is still automated, but the trigger to deploy requires a human pressing a button.) Some companies, for some products, have managed to eliminate this boundary altogether, automatically releasing code to production without any manual verifications. This is an advanced form of continuous delivery called *continuous deployment* and is clearly not appropriate in all environments. The software that helps keep an airplane in the air requires a much higher degree of confidence than your favorite social media platform.
- The boundary of *value* is the point at which real users have access to the new software.

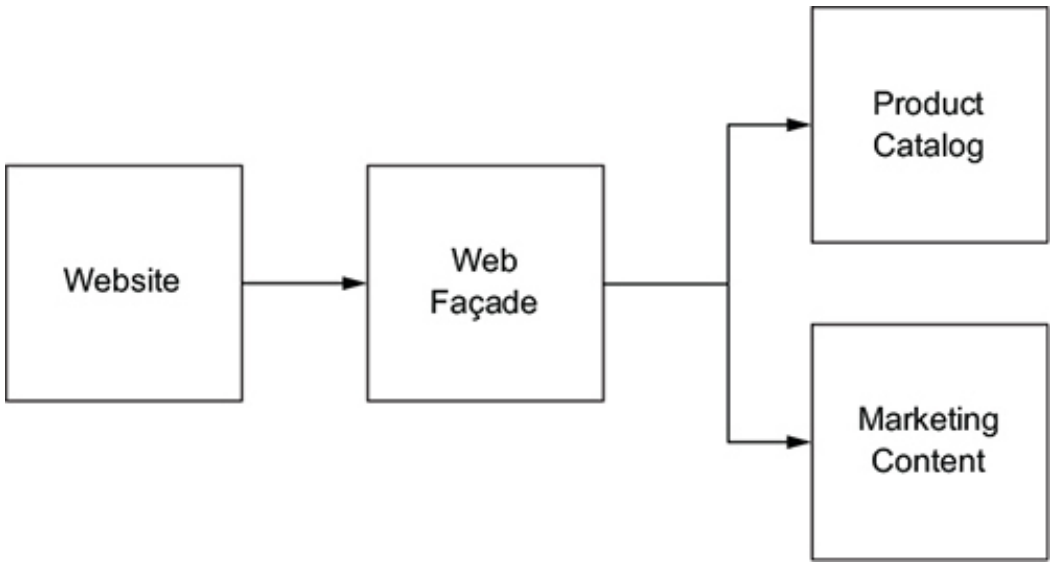
Because this book is about testing, we only tackle the *test pipeline*, those early stages of the pipeline that give you confidence that you can ship to production. The full deployment pipeline would include the production deployment as well.

9.2. CREATING A TEST PIPELINE

It has been a while since we looked at the example pet store website in chapter 2. We used it as a small-scale simulacrum of a microservices-backed e-commerce application and focused on the web façade service, which aggregated results from two other services (figure 9.6):

- The product catalog service, responsible for sending product information
- The content service, responsible for sending marketing copy about a product

Figure 9.6. An example set of microservices



For demonstration purposes, we'll take the perspective of the team writing the web façade code that aggregates product and marketing data to present to the website. The example is simple enough to digest in short order and complicated enough to require meaningful testing. The code is a simple Express app (a popular node.js web application framework). The following listing shows what you need to initialize the service.[4]

4

See the full source code at https://github.com/bbyars/mountebank-in-action.

Listing 9.1. Web façade initialization code

```
var express = require('express'),
  productServiceURL = process.env['PRODUCT_SERVICE_URL'],    1
  contentServiceURL = process.env['CONTENT_SERVICE_URL'],    1
  productsGateway = require('./models/productsGateway')      2
    .create(productServiceURL),                              2
  contentGateway = require('./models/contentGateway')        2
    .create(contentServiceURL),                              2
  productCatalog = require('./models/productCatalog')        3
    .create(productsGateway, contentGateway);                3

var app = express();                                         4
```

- ***1*** **Configures external services**
- ***2*** **Gateway code to external services**
- ***3*** **Module to do the aggregation**
- ***4*** **Creates the Express app**

Using environment variables for configuration is a common approach and will allow

you to use different URLs in different environments (and to use service virtualization for service tests). The two gateway objects are simple wrappers over the HTTP calls, allowing you to centralize error handling and logging around external service calls.

The code that responds to an HTTP request and returns the aggregated results of the product and content services is quite straightforward, delegating the complex logic to the `productCatalog` object, as shown in the following listing.

Listing 9.2. Web façade code to aggregate product and content data

```
app.get('/products', function (request, response) {          1
  productCatalog.retrieve().then(function (results) {        2
    response.json({ products: results });                    3
  }, function (err) {                                        4
    response.statusCode = 500;                               4
    response.send(err);                                      4
  });
});
```

- ***1*** **Responds to GET /products**
- ***2*** **Delegates aggregation**
- ***3*** **Returns results as JSON**
- ***4*** **Error handling**

You could've left the aggregation logic directly in the function that handles the HTTP request. You didn't, in part, because that would've made it harder to unit test.

9.2.1. Creating unit tests

Test-driven development (TDD) is often also called test-driven design because the act of writing unit tests helps enforce loose coupling and high cohesion in your production code.[5] Bundling all the aggregation logic into your HTTP handling code would have required a significant amount of setup to test it—exactly the type of friction that reduces your motivation to write tests to begin with. It also would've been a worse design, coupling HTTP handling logic with aggregation logic. The drive to make unit testing your code as easy as possible is one of the best influences for keeping your codebase modular. Unit tests are as much about helping you design your application as they are about finding bugs.

5 TDD is more than writing unit tests. It's a practice that involves writing a small test before the code exists and

building just enough code to make the test pass. Those small iterations with refactoring in between help grow the design of the codebase organically.

Unit tests should be in-process to the application under test, and each unit test should focus on a small piece of code. Consequently, you should never use service virtualization in your unit tests. This is the realm of traditional mocks and stubs.

Let's look at the `productCatalog` code. We'll start with the wrapper logic needed to create an instance, as shown in the following listing, and export it to another JavaScript file.

Listing 9.3. The shell of the `productCatalog` module

```
function create(productsGateway, contentGateway) {   1
  function retrieve () { ... }                       2

  return {                                           3
    retrieve: retrieve                               3
  };                                                 3
}

module.exports = {                                   4
  create: create                                     4
};                                                   4
```

- ***1* Creates an instance using dependency injection**
- ***2* See listing 9.4.**
- ***3* Returns the instance with one function**
- ***4* Exports the creation method to other files**

Much of this is JavaScript and node.js plumbing. The creation function[6] accepts the two gateway objects as parameters. This pattern—dependency injection—is another area where good unit testing practices intersect with good design. If you created the gateways at the exact spot you needed them, you wouldn't be able to swap out the gateway instances with another object for testing. That would mean you would be forced to test the full end-to-end flow each time because the gateways are responsible for making the HTTP calls to the external services. It also would've created a tight coupling, preventing higher order code from adding decorators around the gateways for added functionality.

6 You could've also used JavaScript constructors, but they add a bit more magic than a simple creation function.

The `retrieve` function uses those gateways to retrieve and aggregate the data from the product and content services, as shown in the following listing.

Listing 9.4. The code that retrieves and aggregates downstream services

```
function retrieve () {
  var products;

  return productsGateway.getProducts()                    1
  .then(function (response) {
    products = response.products;

    var productIds = products.map(function (product) {
      return product.id;                                  2
    });                                                   2
    return contentGateway.getContent(productIds);         3
  }).then(function (response) {
    var contentEntries = response.content;

    products.forEach(function (product) {
      var contentEntry = contentEntries.find(             4
        function (entry) {                                4
          return entry.id === product.id;                 4
        });                                               4
      product.copy = contentEntry.copy;                   5
      product.image = contentEntry.image;                 5
    });

    return products;
  });
}
```

- ***1*** **Gets products from the product catalog service**
- ***2*** **Maps to the IDs only**
- ***3*** **Gets content for those products**
- ***4*** **Matches the content entry to the product by ID**
- ***5*** **Adds marketing content data**

Clearly, the bulk of the web façade complexity lies in this function, making it a great place to focus your unit testing efforts. To keep the unit test in-process, you will have to stub out the two gateways.[7] You'll use a common JavaScript mocking library called Sinon to help.[8] Sinon allows you to tell the gateways what to return, which supports a very readable test setup (the Arrange step of the standard test pattern of Arrange-Act-Assert), as shown in the following listing.

7

I haven't shown the gateway code because it's not relevant to the example. See the GitHub repo for details.

8

See http://sinonjs.org/, although it's not hard to write your own stubs if you'd rather avoid using an external library.

Listing 9.5. The test setup, using dependency injection and stubs

```
it('should merge results', function (done) {
  var productsResult = {                                    1
        products: [                                         1
          { id: 1, name: 'PRODUCT-1' },                     1
          { id: 2, name: 'PRODUCT-2' }                      1
        ]                                                   1
    },                                                      1
    productsGateway = {                                     2
        getProducts: sinon                                  2
                       .stub()                              2
                       .returns(Q(productsResult))          2
    },                                                      2
    contentResults = {                                      3
      content: [                                            3
        { id: 1, copy: 'COPY-1', image: 'IMAGE-1' },        3
        { id: 2, copy: 'COPY-2', image: 'IMAGE-2' }         3
      ]                                                     3
    },                                                      3

    contentGateway = {                                      4
      getContent: sinon                                     4
                    .stub()                                 4
                    .withArgs([1, 2])                       4
                    .returns(Q(contentResults))             4
    },                                                      4
    catalog = productCatalog.create(                        5
        productsGateway, contentGateway);                   5

  // ACT                                                    6
  // ASSERT                                                 6
});
```

- ***1*** **Stages the product catalog service results**
- ***2*** **Sets up the product stub**
- ***3*** **Stages the content service results**
- ***4*** **Sets up the content stub**
- ***5*** **Passes the stubs into the catalog**

- ***6* See listing 9.6.**

Most of the code is setting up the JSON responses that the gateways are responsible for returning. As you have seen in previous examples, I recommend using test data that's easy to spot to make the assertions easy to read, which is why I have opted for strings like "COPY-1." You stub out the two gateway functions—`getProducts` on the `productsGateway` and `getContent` on the `contentGateway`—using Sinon's `stub()` function and chain on the result you want with the `returns` function. Notice that when you create the stub for the `contentGateway`, you add a `withArgs ([1, 2])` function call. This is like using predicates in mountebank. You're telling Sinon to return the given result if the arguments match what you specify.

The only other nuance to the test code is the mysterious use of the `Q` function, which is used in the stub responses. `Q` is a promise library that helps contain the complexity of using asynchronous code in JavaScript. The real gateways have to reach across the network to retrieve results, and because JavaScript uses nonblocking I/O for network calls, using promises helps make the asynchronous code easier to understand. If you look back to the `retrieve` function in listing 9.4, you'll see that you call a `then` function after each gateway call and pass in the code to execute when the I/O is finished. Wrapping the objects inside the `Q` function adds the `then` function to your stub results, so the production code works on the stubs as expected.

Let's close off the example by looking at the Act and Assert stages of the test in the following listing.

Listing 9.6. The unit test assertion

```
it('should merge results', function (done) {
  // ARRANGE                                          1

  catalog.retrieve().done(function (result) {         2
    assert.deepEqual(result, [                        3
      { id: 1, name: 'PRODUCT-1',                     3
        copy: 'COPY-1', image: 'IMAGE-1' },           3
      { id: 2, name: 'PRODUCT-2',                     3
        copy: 'COPY-2', image: 'IMAGE-2' },           3
    ]);                                               3

    done();                                           4
  });
});
```

- ***1* See listing 9.5.**

- ***2* Act**
- ***3* Assert**
- ***4* Tells the test runner that the test has finished**

Anytime you're using asynchronous code, you have to tell the test runner that the test is complete. The assertion verifies that you merged the results of the two gateways correctly. The done test parameter is a function that you call after your assertion to signify the end of the test execution.

You could, and should, write many more unit tests on the `retrieve` function. For example, you could write unit tests to specify what happens in each of these scenarios:

- There's no marketing copy for a product.
- A downstream service times out (resulting in a gateway error).
- You get missing JSON fields from the marketing content service.
- The marketing content service returns products in a different order than the product catalog service.

It's much easier to write the code to support these scenarios in a suite of unit tests than it is to validate them with higher level tests. Unit tests should be numerous and run quickly, which is why they form the base of the testing pyramid.

You can create a build script that runs the unit tests and wire that into the first stage of your continuous integration tool. Once you have done so, you have automated the first stage of your test pipeline.

9.2.2. Creating service tests

Service tests should exercise your application over the wire, which rules out in-process stubs. This is where service virtualization shines, as service virtualization is the out-of-process equivalent of stubbing.

Although you can always set up your imposters using a config file, I recommend using mountebank's API when possible for service tests. The API allows you to create the test data for each test separately rather than having to depend on an implicit linkage between a magic key in your test setup and the scenario you're testing. You used mountebank's API in chapter 2.

I've modified the example slightly to highlight the point about keeping the test data as simple as possible. Let's take a fresh look at the test first in the following listing; we

look at the helper functions again shortly.

Listing 9.7. A service test that validates web façade aggregation

```
it('aggregates data', function (done) {
  createProductImposter(['1', '2']).then(function () {    1
    return createContentImposter(['1', '2']);             1
  }).then(function () {
    return request(webFacadeURL + '/products');           2
  }).then(function (body) {
    var products = JSON.parse(body).products;

    assert.deepEqual(products, [                          3
      {
        "id": "ID-1",
        "name": "NAME-1",
        "description": "DESCRIPTION-1",
        "copy": "COPY-1",
        "image": "IMAGE-1"
      },
      {
        "id": "ID-2",
        "name": "NAME-2",
        "description": "DESCRIPTION-2",
        "copy": "COPY-2",
        "image": "IMAGE-2"
      }
    ]);
    return imposter().destroyAll();                       4
  }).done(function () {
    done();                                               5
  });
});
```

- ***1*** **Arrange**
- ***2*** **Act**
- ***3*** **Assert**
- ***4*** **Cleanup**
- ***5*** **Tells the test runner you're done**

The `createProductImposter` and `createContentImposter` functions are similar. They use mountebank's API to create the virtual services. Both functions accept an array of suffixes, which they use to append to the test data within each field name. You can see what that results in by looking at the assertion in listing 9.7. The code to do that does a simple string append to each field name:

```
function addSuffixToObjects (suffixes, fields) {
  return suffixes.map(function (suffix) {
    var result = {};
    fields.forEach(function (field) {
      result[field] = field.toUpperCase() + '-' + suffix;
    });
    return result;
  });
}
```

With that helper function, the imposter creation uses the same fluent API you built in chapter 2 that wraps mountebank's RESTful API, as shown in the following listing.

Listing 9.8. The imposter creation functions

```
var imposter = require('./imposter'),      1
  productPort = 3000;

function createProductImposter (suffixes) {
  var products = addSuffixToObjects(suffixes,
    ['id', 'name', 'description']);

  return imposter({
    port: productPort,
    protocol: "http",
    name: "Product Catalog Service"
  })
    .withStub()
    .matchingRequest({equals: {path: "/products"}})
    .respondingWith({
      statusCode: 200,
      headers: {"Content-Type": "application/json"},
      body: { products: products }
    })
    .create();
}

var contentPort = 4000;

function createContentImposter(suffixes) {
  var contentEntries = addSuffixToObjects(suffixes,
    ['id', 'copy', 'image']);

  return imposter({
    port: contentPort,
    protocol: "http",
    name: "Marketing Content Service"
  })
    .withStub()
    .matchingRequest({
      equals: {
```

```
          path: "/content",
          query: {ids: "ID-1,ID-2"}
        }
      })
      .respondingWith({
        statusCode: 200,
        headers: {"Content-Type": "application/json"},
        body: { content: contentEntries }
      })
      .create();
  }
```

- ***1*** **See listing 2.5.**

Arguably, the hardest part of managing a suite of service tests is maintaining the test data. Test data management is a complicated subject, and many vendors are willing to sell you solutions that promise to ease the pain. Although such tools may help in complex integrated test scenarios, I believe you should use them sparingly. Too often, they're used as a way of avoiding shifting the tests *to the left*, where left refers to the left side of the deployment pipeline (close to development).

Creating test cases with appropriate isolation via service virtualization is key. The example in listing 9.7 virtualizes a couple of simple service schemas. Real world schemas are often much more complex. In such cases, you will want to save off the responses in separate files and use string interpolation to add in any dynamic data needed. You can reference the specific scenario directly in your test using a key that identifies the scenario. For example, if you wanted to test what happens when the product catalog service returns a product but there's no marketing content for it, use a product ID of `NO-CONTENT`. Leaving breadcrumb trails in your test data will make maintaining it much easier.

The team writing tests needs to own the configuration for the virtualized service

Generally, two types of service virtualization tools are available.

The first type comes from the open source community. They almost always support HTTP/S. Although most of them support record-playback through proxying, they often expect the team writing the tests—the client team—to define the data that the virtual service returns.

The second type represents the commercial virtualization tools. They usually are more

feature-rich and support a more complete set of protocols. But because of the licensing model, they typically expect a central team to own the virtual service definitions. For automated service tests, that's exactly backwards from how it should be.

Automated testing requires fine-grained control over the testing scenarios. Those scenarios will require a different set of test data than another team's automated tests, even if both test suites have a shared dependency that requires virtualizing. Having to go through a central team to set up your test data adds unnecessary friction, which has the unfortunate side effect of discouraging your developers from writing the tests. The complexity of the configuration will also become unwieldy to understand and maintain if your test data is comingled with that of other teams, adding more friction.

At this stage of the deployment pipeline, your team needs to be in complete control of its test data. That means your team needs to write the configuration for the virtual services. Relying on a central team or the team producing the service that you depend on to write the configuration for you will always result in a deficient test suite.

Throughout much of this book, I have described mountebank's mission statement as keeping things that should be easy actually easy while making hard things possible. Rephrasing that as a competitive product strategy, my goal with mountebank is to provide the power of commercial service virtualization tools with the CD-friendliness of the open source tools.

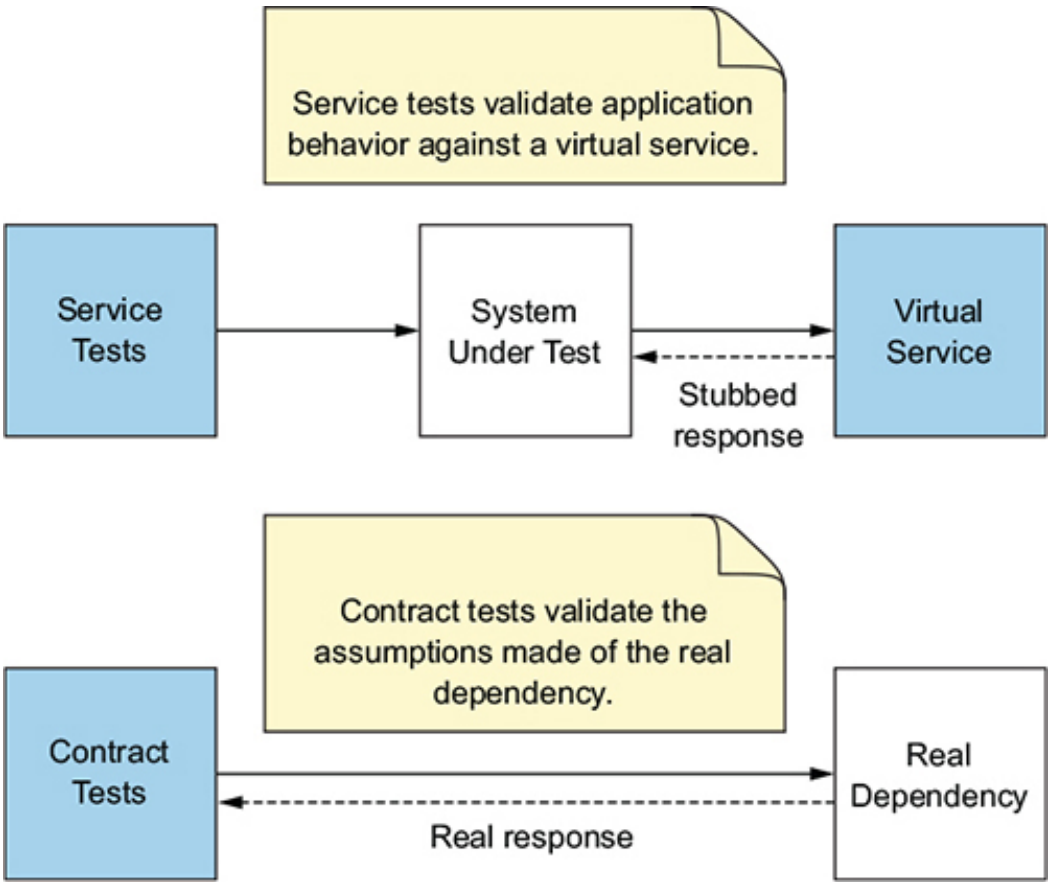
9.2.3. Balancing service virtualization with contract tests

Unit tests help you design your application and catch simple bugs created while refactoring. Service tests treat your application as a black box and help catch bugs from a consumer's point of view. Adding service virtualization keeps service tests deterministic, sealing your test scenarios in a pristine laboratory environment.

Production is more like a war zone than a lab. Although service tests give you confidence that your application works with certain assumptions about your runtime dependencies, they do nothing to validate those assumptions. Like it or not, your runtime dependencies change over time, as they have their own release cycle, so you need some dynamic way of detecting breaking changes in services that you depend on (for example, checking that the marketing content service returns the marketing copy in a top-level JSON field called `copy`). That's the responsibility of the next stage of the test pipeline: where you run your contract tests.

Contract tests move you into the realm of integration, and any time you integrate to the outside world (code written by another team or another company), you're no longer in a friendly, deterministic environment. Your tests may fail because your code has a bug, because *their* code has a bug, because of configuration bugs in the environment itself, or because the network hiccupped. Consequently, you test as much as you can in the friendly confines of unit and service tests. Contract tests shouldn't be deep behavioral tests; they should be lightweight validations of your assumptions about your runtime dependencies. In effect, they validate that your stub definitions are compatible with the real service (figure 9.7).

Figure 9.7. Contract tests validate the assumptions made in service tests.



You generally want to avoid behavioral testing of your dependencies—that's the job of the team building those dependencies. Except in dysfunctional situations, you are better off treating them no differently than you would a software as a service (SaaS) application that you'd pay to use, or an API from a third party like Google. Aside from the obvious cost of behavioral testing of your dependencies, it also increases the fragility of the environment configuration, as behaviorally testing your dependency requires *its* dependencies to also be functional. This leads you back down the path of end-to-end integration testing, creating exactly the kind of traffic jams on the path to production that you're trying to avoid by using service virtualization.

A contract test example

The example we've been looking at is a bit too simplistic to show the value of contract testing, in large part because we asserted only that the marketing content gets correctly merged into the product data. The tests haven't put too much emphasis on what that product data is, but the website would depend on a certain set of fields to display the data appropriately. That means that the web façade service must expect a certain set of fields coming out of the product catalog service. Contract testing helps verify that those expectations remain true as the product service changes.

Let's say you expect a name, description, and availability dates for a product. You expect the dates to be in ISO format (YYYY-MM-DD) to ensure that you're parsing it correctly. A contract test validates that those fields exist in the place and format where you expect them. Let's start by showing the helpers to validate the type and format of data you get back, which use a regular expression to validate the date format:

```
function assertString (obj) {
  assert.ok(obj !== null && typeof obj === 'string');
}

function assertISODate (obj) {
  assertString(obj);
  assert.ok(/201\d-[01]\d-[0123]\d/.test(obj), 'not ISO date');
}
```

The regular expression uses the same `\d` metacharacter you have seen previously, which represents a digit. The rest of the regular expression matches either literal numbers (for example, to ensure that the year starts with the decade this book was written in—201x), or a limited set of numbers represented in brackets (for example, the month must start with 0 or 1, and the day with 0 through 3). This isn't a perfect test—it allows 2019-19-39, for example—but it's probably good enough. If you need more confidence, you can add more advanced date parsing to the test code.

You can use the `assertString` and `assertISODate` helpers to write the test, which validates that the fields exist in the location and format you expect them, as shown in the following listing.

Listing 9.9. A contract test validating the placing and format of fields from a real dependency

```
it('should return correct product fields', function () {
  return request(productServiceURL + '/products')          1
  .then(function (body) {                                  1
    var products = JSON.parse(body).products;
    assert.ok(products.length > 0, 'catalog empty');       2
```

```
    products.forEach(function (product) {                3
      assertString(product.name);                        3
      assertString(product.description);                 3
      assertISODate(product.availabilityDates.start);    3
      assertISODate(product.availabilityDates.end);      3
    });
  });
});
```

- ***1*** **Calls the real product catalog service**
- ***2*** **Performs a sanity check**
- ***3*** **Validates field formats**

Before we get into what this function is testing, let's take a step back to think about what it's *not* testing. It's not testing the full breadth of the product catalog service. The product catalog service may return dozens of fields that aren't relevant to your service (the web façade), so you don't test them. Remember, you are testing your assumptions about the product catalog service, not the product catalog service itself.

What you *are* testing is that the format of the data you get back corresponds to what you expect. You're doing that for every product returned, but you also could've done it for only the first product in the array. It's a time versus comprehensiveness tradeoff, and you'll have to face that tradeoff on a case-by-case basis. It's generally a safe assumption that a well-behaved service will return the same schema for all elements in an array.

Managing test data

The hardest part—by far—of contract testing is managing test data. Our example more or less avoids the problem by testing a read-only endpoint, although we still added a sanity check in the test to ensure that at least one product was returned. Contract testing services that allow you to change state are possible but require some support from the service.

The cleanest method is to have your test create the data before reading it. For example, you may submit an order by sending a POST to `/orders` and retrieve it by sending a GET to `/orders/123`, where 123 is the order ID in the response to the first call. With this approach, every test execution creates new data, which ensures that the data is isolated from every other test execution. However, it does require the ability to create test data. That's a reasonable enough assumption for the orders service, but the product catalog service is unlikely to provide APIs to create new products, as that is generally a

back-office process.

An alternative is to coordinate with the provider team on a set of data you can rely on for testing purposes. The provider team is then responsible for maintaining a set of golden test data and ensuring that it's available with each release of the software into the test environment. Any such golden data should be nonmutable by your tests, so they can run repeatedly on the same data.

Where service virtualization fits

It's possible to use contract testing without any virtual services. This assumes that the dependencies are deployed in a shared environment with all of *their* dependencies also available, on down the stack to the systems of record. Some organizations are either small enough or have invested heavily enough in shared infrastructure to make this possible.

An alternative strategy is for the team that manages the dependency to deploy a test instance available for contract testing and stub out its dependency through service virtualization. This is likely to increase the availability and determinism of the test instance, as it's now less subject to the whims of environmental hiccups.

As a client of a service that another team provides, you have the right to set some expectations of that service. An expectation that a test instance is available is both reasonable and common. You're better off if you can treat that test instance as a black box and have the provider team decide whether they run their test instance integrated or with virtual dependencies.

9.2.4. Exploratory testing

Historically, tests were divided into scripted and unscripted, where the script referred to a documented set of steps and expectations a manual tester executed. The heyday of meticulously cataloging test cases in commercial tools so that unskilled QA testers (remote from the application team) could execute them without any system context is gone. You still do scripted tests, but you automate them nowadays. Test design occurs as you write the test, and test execution occurs every time you run the test.

Exploratory testing combines test design and test execution into one activity, bringing discipline to unscripted testing. The ability to follow their nose is one of the defining characteristics of great QA testers. Exploratory testing allows them to investigate the software with an attitude of curiosity, unearthing its sharp edges through creativity rather than through predefined scripts.

The pendulum has swung quite far from the days of yesteryear, when all scripted tests were executed manually, to the point where I occasionally perceive a stigma associated with manual testing altogether. This is unfortunate. Exploratory testing is a fine art, worthy of its own study.[9] Whereas exhaustive automated testing mechanizes the toil of scripted test execution, exploratory testing puts the ghost back in the machine. It relies on human ingenuity to find gaps in your automation. Despite the widely held perception that microservices are too technical to manually test, there's a great deal of similarity between how you manually test an API[10] and how you test a traditional GUI.

9

James Bach gives a good introductory overview at http://www.satisfice.com/articles/what_is_et.shtml.

10

Services and APIs are often used interchangeably. Here I use "service" (or "microservice") as the implementation and "API" as the interface. The users of a service only see the API. In fact, they have no way of knowing if the API is implemented with one service or multiple, because you could use a reverse proxy to route requests to different services under the hood.

I've seen people fall into two traps, both of which hurt their ability to gain confidence in your service. The first is treating the service as an implementation detail, a cog in a larger value chain, such that the only meaningful test is of the overall end user delivery (where "end users" might be customers or business users). The second trap is believing that the service is too technical to test on its own.

Manually testing APIs

Overcoming the first trap requires a mindset shift. The more your organization thinks of the API exposed by your microservice as a product, the more likely you are to gain the scaling benefits of microservices. Amazon provides an easy to spot example: Amazon Web Services (AWS). AWS started off as a simple object store (S3), with an API to store and retrieve files. In short order, Amazon released EC2, which allowed programmatic access to managing virtual machines. Both S3 and EC2 are products, as are the hundreds of other products in the AWS suite. They have teams that manage them, they have customers, they provide self-service capabilities, and they hide the underlying complexity of those capabilities.

AWS represents a collection of public APIs, but the same principle applies for APIs you build for your enterprise. The trick is realizing that your internal development teams *are* customers. They have needs and use your service to fulfill those needs, saving them time and reducing the complexity of their overall solution. Understanding their needs helps focus exploratory testing.

Once you recognize that your API is a black box to your customers, you are free to test whether the black box behaves correctly. A good exploratory testing session would start by trying to solve an end-to-end customer problem with your APIs and adjusting the path from one API call to the next based on what you learn as you go. Do the errors make sense? Does the response provide hints as to what happens next? Sometimes you might discover that, although your API is functionally stable, it has significant usability gaps.

Another way to look at the second trap is thinking that, because there's no UI, manual testing of APIs doesn't make sense. Once you treat your API as a product, this argument disappears. Anytime you have customers (developers), you have a user interface. For APIs, that UI happens to be JSON over HTTP (or equivalent).

In fact, you have been manually testing an API throughout this entire book. Every time you use `curl` (or Postman, a graphical equivalent) to send an HTTP request to mountebank, you're using mountebank's developer-facing UI to test it.

Where service virtualization fits

You certainly can do exploratory testing without service virtualization. Indeed, at least some of the time, you should. It helps gain full system context and understand the types of data that downstream systems emit.

But exploratory testing requires QA testers to get creative. A large part of the exploration is finding out what they *should* test, which requires playing with some unusual setups. Virtualizing the dependencies can help provide additional knobs to tune during the exploration.

A real-world scenario may help make that advice concrete. In chapter 8, I described how we used mountebank to help test APIs that powered the consumer-facing mobile application for a large airline. Our team was blessed with a couple of superbly capable QA testers who, through exploratory testing, unearthed several problems with our APIs before we released them to the public.

Although some of their testing was manual, they used mountebank to test flows under certain scenarios. Going to the downstream integration points for those scenarios was quite onerous, so when they wanted to follow a flow involving a canceled flight (or a rerouted flight, overbooked flight, delayed flight, and so on), they used a set of mountebank imposters to facilitate the testing experience. The first time they tested a flow within a given scenario, like a canceled flight, they did so fully integrated so they could see real data. Once they had the data, they used mountebank imposters on

subsequent test explorations.

Exploratory testing completes our whirlwind tour of the role of testing in a continuous delivery world, including where service virtualization fits and where it doesn't. We'll round out the next chapter by using mountebank to help us with performance testing.

SUMMARY

- A CD deployment pipeline includes automation that extends beyond the realm of testing, but testing is central to providing the confidence needed to release software frequently. The testing portion of the pipeline requires validations at multiple layers.
- Unit testing is as much a design activity as it's a bug-catching activity. Unit testing is in-process and, as such, uses traditional stubbing approaches instead of service virtualization.
- Service tests are postdeployment black-box tests of your application. Service virtualization ensures appropriate determinism.
- Contract tests help validate the assumptions that your application and your service tests make. They should focus on testing your assumptions rather than behaviorally testing the dependent service.
- Exploratory testing unleashes human creativity to find flaws in your software. Service virtualization may play a role in validating testers' hunches, or you may avoid it in favor of deeper integration testing.

Chapter 10. Performance testing with mountebank

This chapter covers

- How service virtualization enables performance testing
- How to capture appropriate test data with real latencies for load testing
- How to scale mountebank for load purposes

The final type of testing we'll look at in this book is performance testing, which covers a range of use cases. The simplest type of performance test is a *load test*, which uncovers the system behavior under a certain expected load. Other types of performance tests include *stress tests*, which show system behavior when the load exceeds available capacity, and *soak tests*, which show what happens to the system as it endures load over an extended amount of time.

All of the tests we've looked at up to this point have attempted to prove system correctness, but with performance tests, the goal is to understand system behavior more than to prove its correctness. The understanding gained from performance testing does help to improve correctness through unearthing bugs (such as memory leaks) in the application, and it helps to ensure that the operational environment of the application is capable of supporting expected loads. But no application can support infinite load, and stress testing in particular is designed to break the application by finding the upper limits of capacity. While that's happening, a certain degree of errors is expected in many kinds of performance testing. The goal is to verify system behavior in aggregate rather than verify each service call independently. Performance tests often help define service level objectives—for instance, that a service responds within 500 milliseconds 99% of the time under expected load.

Performance testing can be difficult. Fortunately, mountebank is there to help.

10.1. WHY SERVICE VIRTUALIZATION ENABLES PERFORMANCE TESTING

One of the first difficulties organizations run into when putting together a performance test plan is finding an environment in which to run it. Sometimes, that environment is

production.

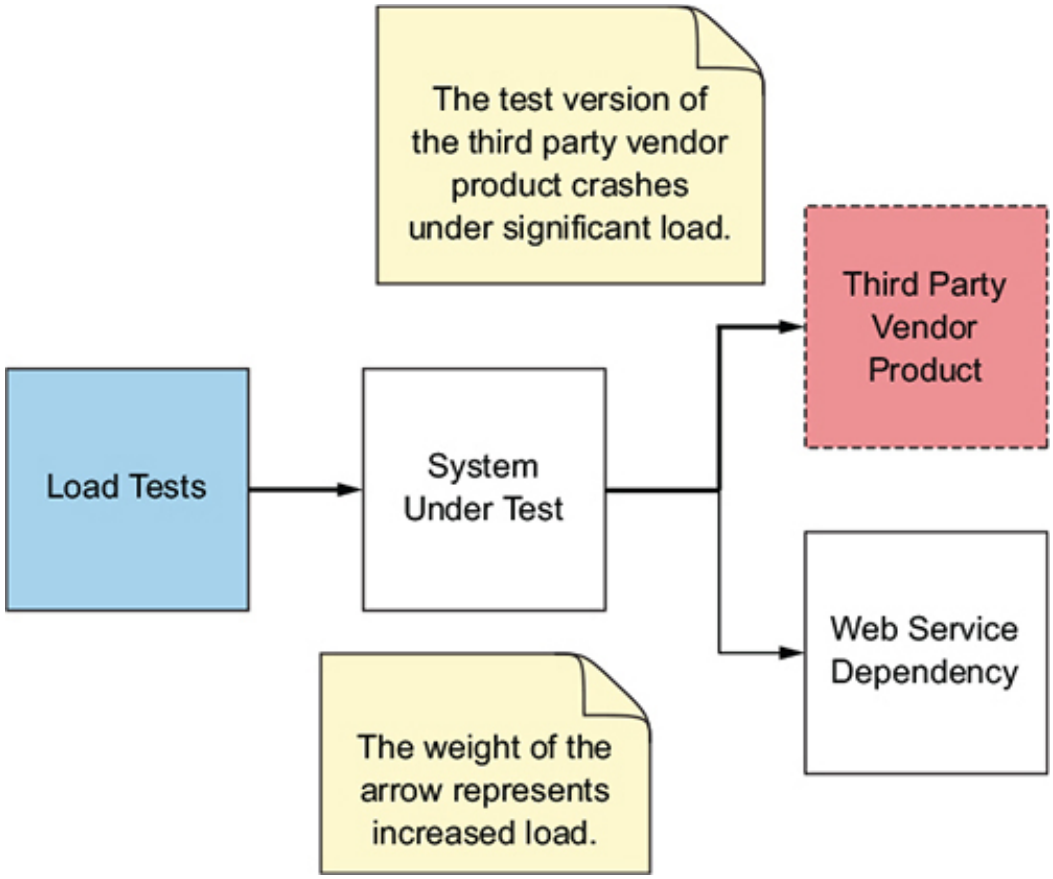
Believe it or not, production can be a natural place to performance test under certain conditions. The first time you deploy a new application to production, it's usually before users are able to use it. That gives you an opportunity to validate the capacity of the system, as long as you are careful with the data. In more advanced scenarios, with new features in existing applications, you may even want to validate performance by synthetically manufacturing load in production before users are aware of the new feature. Facebook calls this dark launching, and did it for two weeks prior to allowing customers to set their own username. The functionality existed in production but was hidden, and a subset of user queries was routed to the new feature to verify it held up under load.[1] Facebook's scale may be unique—imagine generating load from 1.5 billion people—but approaches like dark launching can be valuable anytime you want to have additional confidence that a feature scales before releasing it to the public.

1

See https://www.facebook.com/notes/facebook-engineering/hammering-usernames/96390263919/ for more information.

Most performance testing happens prior to production, with the unfortunate corollary that it rarely happens with integrations scaled to support production load. An all-too-common scenario is that, when performance testing outside of production, the application dependencies crash well before the application, making it impossible to verify service-level objectives (figure 10.1). When performance testing the application, you are implicitly making the assumption that *the application* is the weak link in the system. If it isn't, then you aren't really testing the application and will be unable to discover the load it can support with the hardware it's using.

Figure 10.1. When runtime dependencies are unstable, you can't verify the performance of your application.

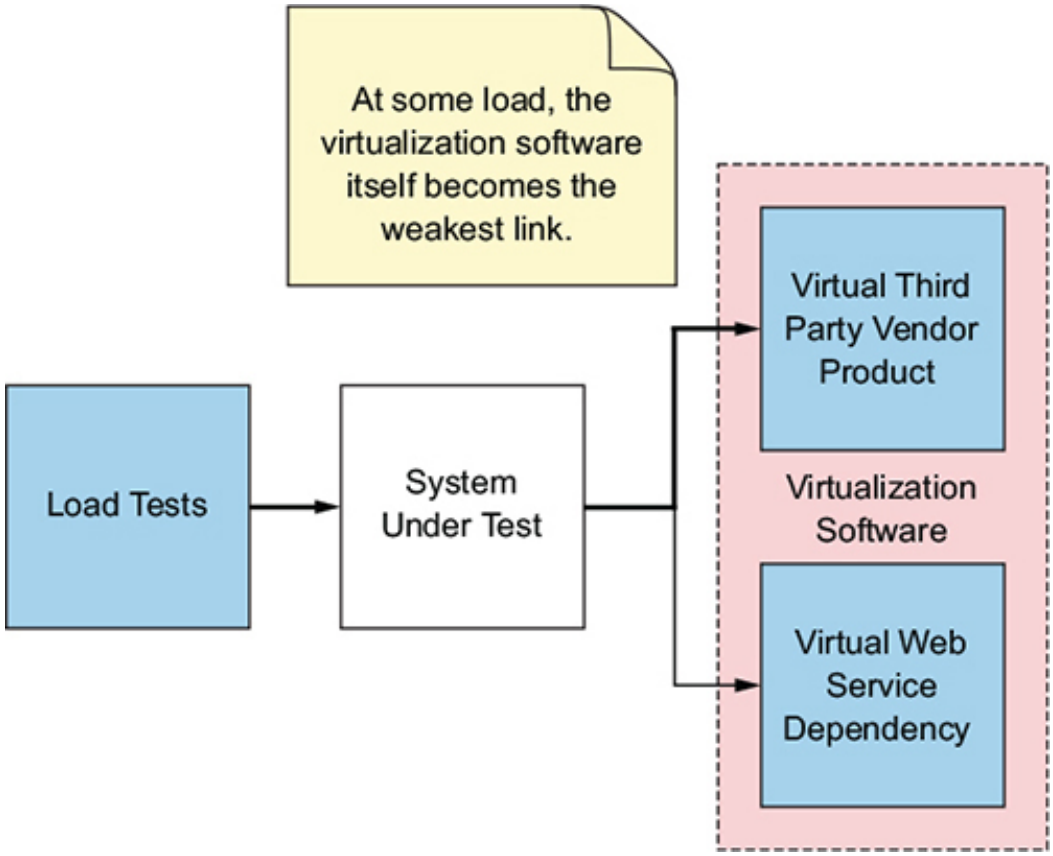


Although those runtime dependencies are stable enough to handle production load in production, creating that level of stability isn't always economically feasible in lower environments. Supporting additional load requires additional hardware, and doubling the cost of production hardware for testing purposes is generally a hard sell. Many other reasons exist, some reasonable and some unfortunate, preventing nonproduction runtime dependencies from supporting the load you need to make your application the weak link. For example, it's often difficult and expensive to scale COTS (custom off-the-shelf software), especially when that COTS package runs on a mainframe.

Figure 10.1 looks like many other diagrams you have seen already, and for good reason. Performance testing sits within a class of problems that requires a more deterministic approach to testing an application with nondeterministic runtime dependencies. I hope by now you can spot exactly the class of problems that service virtualization aims to help. It's a problem that says "*if* the rest of the runtime ecosystem is more stable than my application, *then* I can determine the performance characteristics of my application." Service virtualization helps ensure that the application is the weakest link in the runtime ecosystem.

At some scale, you run into another problem: the virtualization tool itself becomes a weaker link than the application (figure 10.2). It's in effect a hidden dependency that exposes itself when the application's capacity exceeds that of the virtualization tool.

Figure 10.2. At a certain scale, the virtualization tool itself becomes the problem.



This problem exists with mountebank no differently from any other tool. The solution involves horizontally scaling the virtualization tool—running multiple instances with shared test data and using a load balancer to spread the load over the multiple instances. This is where mountebank stands apart from the crowd. Scaling commercial tooling is expensive, requiring additional licensing. Although a single instance of mountebank won't perform as well as a single instance of most of the commercial tools in the space, mountebank scales for free.

Capital One went through performance testing pain as it moved its mobile servicing platform to the cloud. Jason Valentino wrote about the cloud migration and acknowledged that they never anticipated the difficulty of challenges like performance testing.[2]

2

See https://medium.com/capital-one-developers/moving-one-of-capital-ones-largest-customer-facing-apps-to-aws-668d797af6fc.

> *In fact, halfway through we discovered our corporate mocking software couldn't handle the sheer amount of performance testing we were running as part of this effort* ***(we completely crushed some pretty industrial enterprise software in the process)****. As a result, we made the call to move the entire program over to a Mountebank OSS-based solution with a custom provision to give us the ability to expand/shrink our mocking needs on demand.*

Jason Valentino (emphasis his)

In the remainder of this chapter, we will stress test a sample service, finding its capacity. In doing so, we will follow a four-step process:[3]

3

The steps are largely the same for other types of performance tests.

- Define your scenarios.
- Capture the test data for each scenario.
- Create the tests for a scenario.
- Scale mountebank as needed.

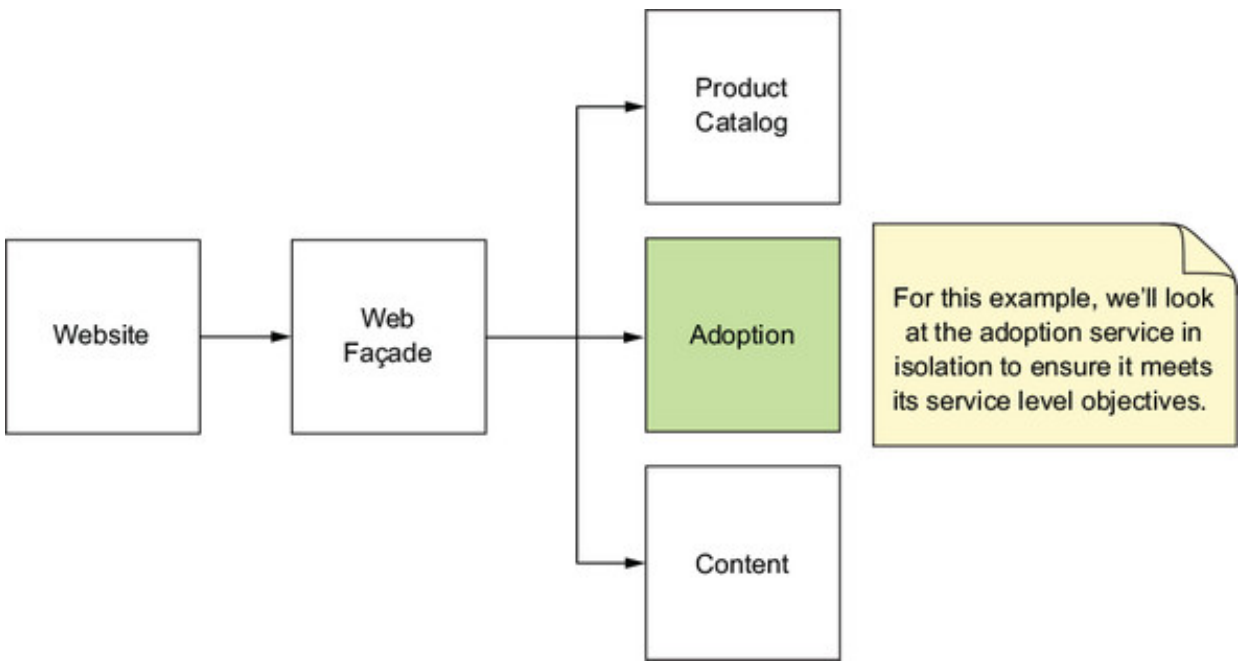
Let's look at each of these steps in turn.

10.2. DEFINING YOUR SCENARIOS

Performance tests are all about figuring out common paths users will likely take, then calling those paths a *lot*.

For example, let's return to our favorite online pet store, but add a new component to make a more realistic performance testing scenario. You'll add a new adoption service that provides the pet adoption information, helping connect potential owners with rescue pets. (See figure 10.3.)

Figure 10.3. Adding the adoption service to your pet store microservices



The service integrates with a public API from RescueGroups.org,[4] which makes it a perfect place to use service virtualization. Although you'd like to test the adoption service to make sure it can handle load, slamming a public API providing animal adoption information free of charge seems rude, especially when it's only for testing purposes. Every time you run your performance tests connected to a free public pet adoption service, your unintentional denial of service attack kills a kitten.

4 See https://userguide.rescuegroups.org/display/APIDG/HTTP+API for API details.

A scenario is a multistep flow that captures user intent. The trick is to put yourself in the mind of a user and imagine a common sequence of activities the user would want to complete. In this case, because the adoption service is an API, the direct users will be other developers, but it will be in support of users on a website or mobile device, and their intent will be reflected in the sequence of API calls. You'd expect the end customers to search for nearby pets, maybe change the search parameters a few times, then click on a few pets. Let's formalize that into a performance test scenario:

- User searches for pets within a 20-mile radius of zip code 75228.
- User searches for pets within a 50-mile radius of zip code 75228.
- User gets details on the first three pets returned.

That scenario requires two APIs and five API calls in the process of completing two searches and providing three sets of details. The sequence of API calls with the adoption service would look like this:[5]

5 The GitHub repo for this book has the source code: https://github.com/bbyars/mountebank-in-action.

- GET /nearbyAnimals?postalCode=75228&maxDistance=20
- GET /nearbyAnimals?postalCode=75228&maxDistance=50
- GET /animals/10677691
- GET /animals/10837552
- GET /animals/11618347

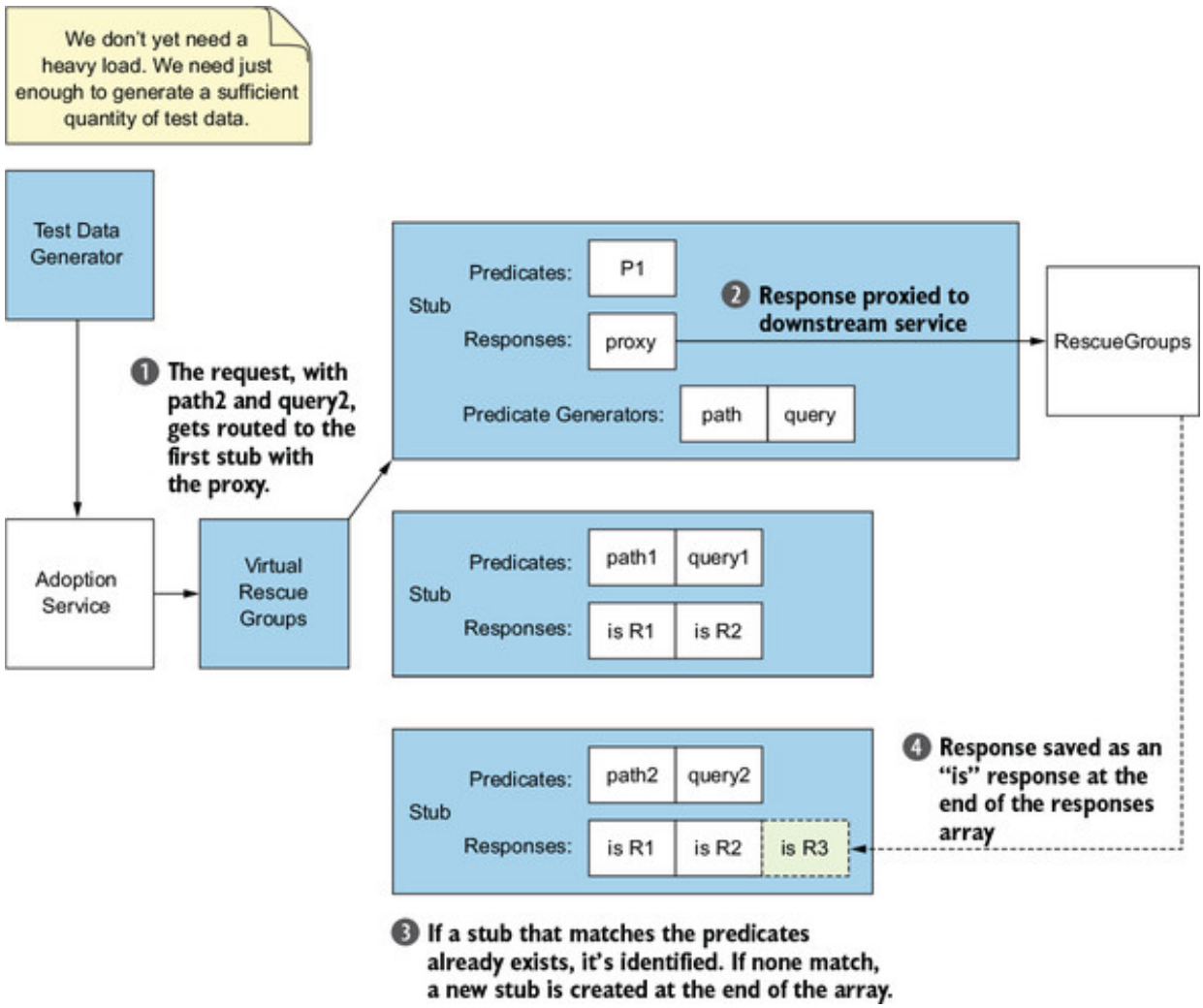
The animal IDs may vary from run to run, as the data that the searches return changes over time. A robust test scenario would support dynamically pulling the IDs from the search, but you'll keep it simple to focus on the essentials.

Now that you have a multistep scenario defined, it's time to capture the test data.

10.3. CAPTURING THE TEST DATA

Accurately simulating a runtime environment for load tests requires that virtual services both respond similarly to how real services respond and *perform* like real services perform *in production*. A proxy can capture both bits of information, and for performance testing you'll almost always want to use `proxyAlways` mode. The default `proxyOnce` mode is convenient in situations when you want the saved responses to respond after the first call to the downstream service, but it's natural to separate the test data capture from the test execution in performance testing. Also, the richer set of data you are able to capture with `proxyAlways` often comes in handy. Recall from chapter 5 that `proxyAlways` mode means that every call will be proxied to the downstream system, allowing you to record multiple responses for the same request (where the request is defined by the `predicateGenerators`) (figure 10.4).

Figure 10.4. A `proxyAlways` proxy allows capturing complex test data.



Notice that the leftmost box in figure 10.4 isn't the performance tests themselves. You aren't ready for those yet; they come after you shut down the connection to the real

RescueGroups API. At this stage, you want just enough load to capture meaningful test data. Anything more than that is an unnecessary load on downstream services.

10.3.1. Capturing the responses

This scenario is simple enough that you can capture data for the five API calls and replay it over and over again during the performance test run. Technically, this means you don't need `proxyAlways` mode for your proxy, but it's generally a good idea to use it anyway when you are doing anything mildly complicated with test data capture. The proxy stub looks like the following listing.

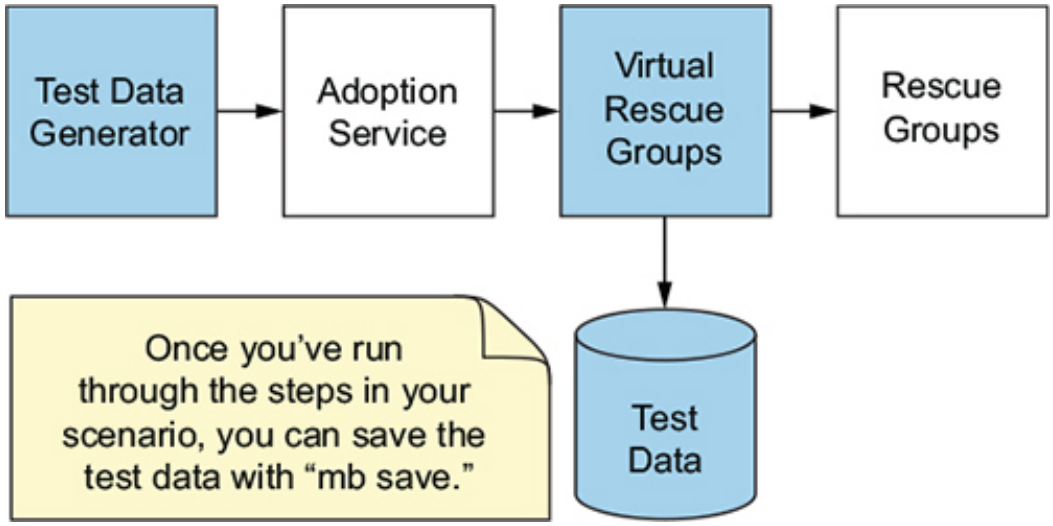
Listing 10.1. Basic proxy `response` to capture test data

```
{
  "responses": [{
    "proxy": {
      "to": "https://api.rescuegroups.org/",     1
      "predicateGenerators": [                   2
        { "matches": { "body": true } }          2
      ]                                          2
    }
  }]
}
```

- ***1* Proxy to real service**
- ***2* Saves new response for every unique request body**

The adoption service uses different URLs and query parameters for each of the five API calls, but behind the curtain, they route to the same URL in the RescueGroups.org API. RescueGroups.org uses a super-generic API in which every call is an HTTP `POST` to the same path (/http/v2.json). The JSON in the request body defines the intention of the call, any filters used, and so on. Recall from chapter 5 that you use a proxy's `predicateGenerators` to define the predicates for the saved response. Because each of your five API calls will send a unique request body to the RescueGroups.org API, differentiating requests by `body` makes sense. If you wanted to be more specific, you could use a JSONPath predicate generator to split on the exact fields within the `body` that are different, but that's overkill for this example. Once you have the proxy configured, you have to run your test scenario and save the test data (figure 10.5).

Figure 10.5. Using a proxy to capture test data



Anytime you use service virtualization, you have to be able to swap out the URL of the downstream dependency in the system under test. The adoption service in the GitHub repo for this chapter supports using an environment variable to change the URL of the downstream service. Assuming you run your proxy imposter on port 3000, you could configure the adoption service like this:

```
export RESCUE_URL=http://localhost:3000/
```

With the proxy running and the adoption service pointed to it instead of the real service, you can capture the data for your five API calls with whatever HTTP engine you want, including `curl`. Assuming the adoption service is running on port 5000, that might look like

```
curl http://localhost:5000/nearbyAnimals?postalCode=75228&maxDistance=20
curl http://localhost:5000/nearbyAnimals?postalCode=75228&maxDistance=50
curl http://localhost:5000/animals/10677691
curl http://localhost:5000/animals/10837552
curl http://localhost:5000/animals/11618347
```

Once you have done that, you can save the test data:

```
mb save --removeProxies --savefile mb.json
```

With that, you have your responses. However, you need one more thing.

10.3.2. Capturing the actual latencies

To get an accurate performance test, you'll want to simulate the actual latency from the downstream service. In chapter 7, we looked at the `wait` behavior, which allows you to

tack on latency to each response. You can capture it from the downstream system by setting the `addWaitBehavior` attribute of the proxy to `true`, as shown in the following listing.

Listing 10.2. Capturing latency from the downstream system

```
{
  "responses": [{
    "proxy": {
      "to": "https://api.rescuegroups.org/",
      "predicateGenerators": [
        { "matches": { "body": true } }
      ],
      "addWaitBehavior": true          1
    }
  }]
}
```

- ***1* Captures actual latency**

If you capture your test data again by making the five API calls to the adoption service and saving the data with `mb save`, the proxy will have automatically added the wait behavior to each of the saved responses. For example, in my test run, here's a trimmed down version of a saved response:

```
{
  "is": {
    "statusCode": 200,
    "headers": { ... },      1
    "body": "...",           1
    "_mode": "text"
  },
  "_behaviors": {
    "wait": 777              2
  }
}
```

- ***1* Omitted for clarity**
- ***2* Wait 777 ms before responding**

The two searches in the test run used to prepare this sample took 777 and 667 milliseconds, respectively, and the three requests for animal details took 292, 322, and 290 milliseconds. Those wait times were saved with each response to be replayed during the performance test run. The more data you capture during proxying, the more variability you'll have with your latencies.

10.3.3. Simulating wild latency swings

Our example assumes that the downstream system behaves correctly. That's a perfectly valid assumption to make when you want to see what happens when the system under test is the weakest link in the chain, but sometimes you also want to expose cascading errors that happen when downstream systems become overloaded, returning a higher percentage of errors and (even worse) responding increasingly slowly. If the environment supports recording proxy data under load, you may be able to capture the data from a downstream test system. If not, you'll have to simulate it.

The `wait` behavior supports an advanced configuration for this use case. Instead of passing the number of milliseconds to wait before returning a response, you can pass it a JavaScript function. Assuming you have started `mb` with the `--allowInjection` command-line flag, you can simulate a wild latency swing with the following function. It usually responds within a second, but roughly every 10 times it takes an order of magnitude longer.

Listing 10.3. Adding random latency swings with JavaScript injection

```
function () {
  var slowdown = Math.random() > 0.9,
    multiplier = slowdown ? 10000 : 1000;
  return Math.floor(Math.random() * multiplier);
}
```

Instead of passing an integer representing a number of milliseconds, you'd pass the entire JavaScript function to the `wait` behavior. Assuming you saved the previous function in a file called randomLatency.js, you could use EJS templating:

```
{
  "is": { ... },
  "_behaviors": {
    "wait": "<%- stringify(filename, 'randomLatency.js') %>"
  }
}
```

The downside is that this doesn't work naturally with proxying, which captures actual latency.

10.4. RUNNING THE PERFORMANCE TESTS

You can write all of the tests up to this point in the book with traditional unit testing tools from the JUnit family. Performance testing requires more specialized tools that

provide a few key features the JUnit-style tools don't:

- Scenario recording, usually by configuring the tool as a proxy between your HTTP executor (generally a browser) and the application you are testing
- Domain-specific languages (DSLs) for adding pauses, simulating the think time for users in between actions, and ramping up users
- The ability to use multiple threads to simulate multiple users sending concurrent requests
- Reporting capability to give you the performance characteristics of your application after a test run

Although many commercial options exist, some excellent open source performance testing tools are available that don't require you to open your checkbook. JMeter (http://jmeter.apache.org/) and a newer offshoot called Gatling (https://gatling.io/) are popular choices. You'll use a Gatling script simple enough to allow you to keep the focus on service virtualization without having to learn a whole new tool.

The Gatling download is a simple zip file, which you can unpack in any directory you desire. Set the `GATLING_HOME` environment variable to that directory to make the example easier to follow. For example, if you have unpacked it on your home directory in Linux or macOS, type this in your terminal (assuming you downloaded the same version used for this example):

```
export GATLING_HOME=~/gatling-charts-highcharts-bundle-2.3.0
```

The next step is to create a Gatling script that represents your scenario using its Scala-based DSL. I copied the sample scenario that ships with Gatling and adjusted it as shown in listing 10.4. As someone who has never programmed in Scala before, I found myself able to read and write most of the script quite fluently thanks to the expressive DSL. It executes your five API calls with pauses in between, representing the think time in seconds that the end users likely will take to process the results.

Listing 10.4. A Gatling script for your test scenario

```
class SearchForPetSimulation extends Simulation {
  val httpProtocol = http
    .baseURL("http://localhost:5000")                          1

  val searchScenario = scenario("SearchForPetSimulation")
    .exec(http("first search")                                 2
      .get("/nearbyAnimals?postalCode=75228&maxDistance=20"))  2
```

```
      .pause(10)                                                    2
      .exec(http("second search")                                   2
        .get("/nearbyAnimals?postalCode=75228&maxDistance=50"))     2
      .pause(15)                                                    2
      .exec(http("first animal")                                    3
        .get("/animals/10677691"))                                  3
      .pause(5)                                                     3
      .exec(http("second animal")                                   3
        .get("/animals/10837552"))                                  3
      .pause(5)                                                     3
      .exec(http("third animal")                                    3
        .get("/animals/11618347"))                                  3

    setUp(
      searchScenario.inject(rampUsers(100) over (10 seconds))       4
    ).protocols(httpProtocol)
  }
```

- ***1* Base URL of adoption service**
- ***2* Searches, with think time**
- ***3* Animal details with think time**
- ***4* Simulates 100 users**

The most interesting bit is near the bottom, which describes how many concurrent users you want to simulate and how long to ramp them up. It's fairly unrealistic to expect that, at max load, all users start at the same time, so most performance test scenarios account for a ramp-up period. Although 100 users isn't much, it helps you test out your scenario.

To run Gatling, navigate to the code for chapter 10 in the GitHub repo for this book and enter the following into your terminal.

Listing 10.5. Testing your performance script

```
$GATLING_HOME/bin/gatling.sh \
    -sf gatling/simulations                       1
    -s adoptionservice.SearchForPetSimulation     2
    -rf gatling/reports                           3
```

- ***1* Points to your simulations directory**
- ***2* Runs the correct scenario**
- ***3* Saves the output here**

On my machine, running this scenario for 100 users takes a little under a minute, which is hardly enough to stress either the software or hardware but enough to validate the script. Once you're satisfied it's working, bump the users up an order of magnitude or two and rerun to see what happens:

```
setUp(
  searchScenario.inject(rampUsers(1000) over (10 seconds))
).protocols(httpProtocol)
```

On the MacBook Pro I'm using to create this example, the adoption service can handle 1,000 users with no problem but croaks pretty hard at 10,000 users. When I tried running with that many users, Google Chrome crashed, my editor froze, and I may have cried a little because of a failure to save work in progress, but, fortunately, no kittens died.

That's useful information—not just the kittens, but the number of users: the adoption service, when run on my laptop, has the capacity to support somewhere between 1,000 and 10,000 users concurrently. In addition to highlighting a horrific lack of error handling in the adoption service (subsequently improved), that information would help you determine the appropriate hardware to run in production based on the expected number of concurrent users trying to save a rescue animal from the pound.

I experimented a bit until I found a reasonable number of users that stressed the adoption service on my laptop without completely breaking it, which turned out to be 3,125 concurrent users. Understanding your application's behavior under stress is a useful activity for determining what happens at expected peak load and helps to validate your service-level objectives.

The test reports are saved in the "gatling/reports" directory you passed into the `-rf` parameter when you started Gatling. The HTML page gives you all kinds of information that helps you understand the performance characteristics of your application. The table shown in figure 10.6 comes from a report on one of my runs and shows the % KO (errors, with KO being both a common boxing abbreviation for knockout and a clever anagram of OK) and statistical information around response times for each request.

Figure 10.6. Gatling saves error rates and response time data for each step of the scenario.

STATISTICS													Expand all groups | Collapse all groups
Requests	Executions					Response Time (ms)							
	Total	OK	KO	%KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	15625	15591	34	0%	318.878	8	447	906	1085	1386	1610	575	286
first search	3125	3125	0	0%	63.776	905	912	919	1022	1152	1299	927	46
second search	3125	3125	0	0%	63.776	625	636	644	673	714	729	641	15
first animal	3125	3113	12	0%	63.776	11	314	333	440	471	501	329	45
second animal	3125	3107	18	1%	63.776	12	347	380	1303	1408	1581	458	299
third animal	3125	3121	4	0%	63.776	8	356	488	1335	1450	1610	519	332

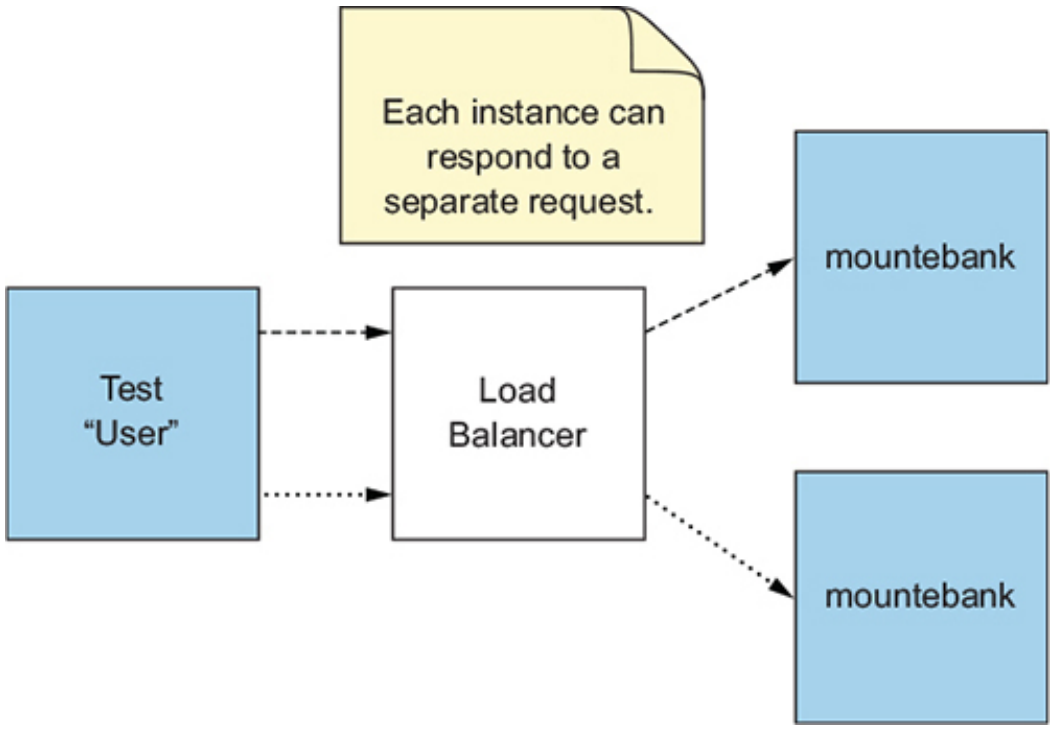
Service performance is often quoted as something to the effect of "99% of the time we promise to return in under 500 milliseconds at or below peak load." Clearly, you have some work in front of you before you can make that kind of guarantee.

10.5. SCALING MOUNTEBANK

Given that the adoption service is a simple example, I was unable to get mountebank to crash under load before the service did. For production-quality services built by enterprises and deployed in a high-availability environment, that won't always be the case. When mountebank itself becomes the weakest link in your chain, you have some options.

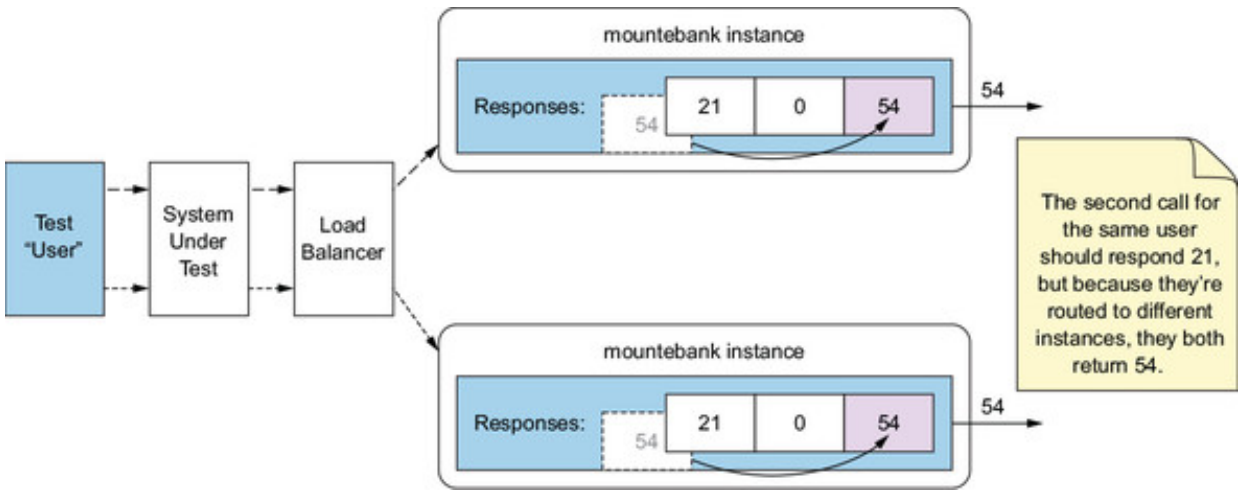
The first and most obvious is to run multiple mountebank instances behind a load balancer (figure 10.7). This allows different requests to route to different mountebank instances, each configured with the same test data.

Figure 10.7. Load balancing using multiple mountebank instances



One situation requires additional thought. If the test data supports sending different responses for the same logical request, then you'll no longer be able to rely on a deterministic ordering of those responses. The diagram in figure 10.8 shows two calls to your virtualized instance of an inventory service, which should return 54 on the first call and 21 on the second. Instead, it returns 54 twice in a row.

Figure 10.8. Responses for the same request under load balancing yield unexpected results.



There's no good way around this. Even if you use server affinity, a load balancer configuration that binds a client to the same mountebank instance for each request, you will likely still have the problem, because it's the system under test, not the test user, that's making the request to mountebank. In addition to load balancing, you should do a few things on each instance of mountebank to guarantee optimal performance.

First, avoid using the `--debug` and `--mock` command line options when running `mb`.

These options capture additional information about the requests that the system under test makes to mountebank, which is useful both for debugging the imposter configuration and for verifying that the system under test made the correct requests. Although capturing that information during a behavioral test can be useful, performance tests require long-lived imposters. Computer programmers have a common phrase they use to describe the process by which a long-lived system remembers information without any mechanism of forgetting it: a memory leak.

Second, you will want to decrease the log output of mountebank. Mountebank uses a standard set of logging configuration levels—`debug`, `info`, `warn`, and `error`—and defaults to `info`. That sends some log output to the terminal and logging file on every request, which is unnecessary and unhelpful when you intend to send thousands of requests its way. I would recommend running with a `warn` level when you are writing and debugging performance scripts and with `error` during the test run. You do that by passing `--loglevel warn` or `--loglevel error` as a flag to the `mb` command.

Finally, you'll generally want to configure the responses mountebank sends back to use keep-alive connections, which avoid the TCP handshake on every new connection. Keep-alive connections are a *huge* performance increase, and the proxy will generally capture them because most HTTP servers use keep-alive connections by default. Unexpectedly, RescueGroups.org doesn't, at least in my test run, so the example avoids the use of keep-alive connections. This is probably appropriate because you're trying to accurately simulate the downstream system behavior. But it wouldn't be hard to write a simple script to postprocess your test data file and change the `Connection` headers to `keep-alive` in all of your saved responses, should you choose to do so.

Also remember that, if you're adding in simulated `is` or `inject` responses that you didn't capture through proxying, you'll have to manually set the `Connection` header to `keep-alive`, as mountebank defaults to `close` for historical reasons. The easiest way is to change the default response, as you saw in chapter 3:

```
{
  "protocol": "http",
  "port": 3000,
  "defaultResponse": {
    "headers": {
      "connection": "keep-alive"
    }
  },
  "stubs": [...]
}
```

And at long last, you have a complete performance testing environment that doesn't depend on any downstream system. And, best of all, no kittens were killed in the process.

Performance testing wraps up our tour of service virtualization. Although mountebank is clearly not the only tool you'll need in your tool belt, it makes many important contributions to allowing you to validate your application through a continuous delivery pipeline, even in a microservices world with a great deal of runtime complexity. Mountebank is always changing, and there's an active mailing list on the website, which I encourage you to use anytime you get stuck. Don't be a stranger!

SUMMARY

- Service virtualization enables performance testing by protecting downstream systems from load. It enables you to test your application as if it's the weakest link in the chain.
- Once you've determined your scenarios, you'll generally want to use a proxy in `proxyAlways` mode to capture test data. Set the `addWaitBehavior` to `true` to capture actual latencies.
- Tools like Gatling and JMeter support converting your test scenarios into robust performance scripts. Ensure you are running with virtual services if your goal is to find the capacity of your service without impacting downstream services.
- If mountebank itself becomes the constraint, scale through load balancing. Improve performance of each instance by decreasing logging and using keep-alive connections.