Testing Microservices with Mountebank

Brandon Byars

SAMPLE CHAPTER

MANNING

Testing Microservices with Mountebank

by Brandon Byars

Sample Chapter 5

brief contents

PART 1 FIRST STEPS 1

1 ■ Testing microservices 3
2 ■ Taking mountebank for a test drive 22

PART 2 USING MOUNTEBANK 41

3 ■ Testing using canned responses 43
4 ■ Using predicates to send different responses 65
5 ■ Adding record/replay behavior 87
6 ■ Programming mountebank 108
7 ■ Adding behaviors 130
8 ■ Protocols 153

PART 3 CLOSING THE LOOP 177

9 ■ Mountebank and continuous delivery 179
10 ■ Performance testing with mountebank 201

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).

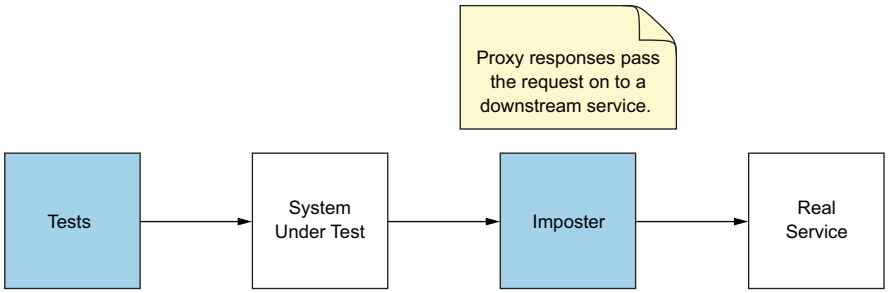

Figure 5.1 An imposter acting as a proxy

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.

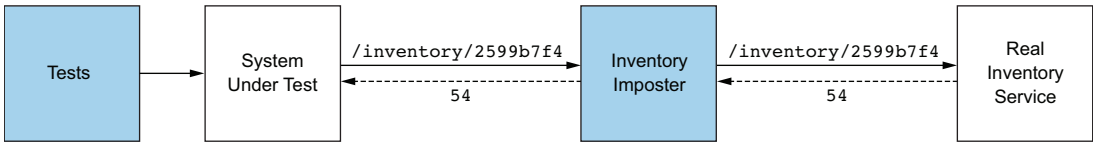

Figure 5.2 Using a proxy to query the downstream inventory

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" }
    }]
  }]
}
```

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).

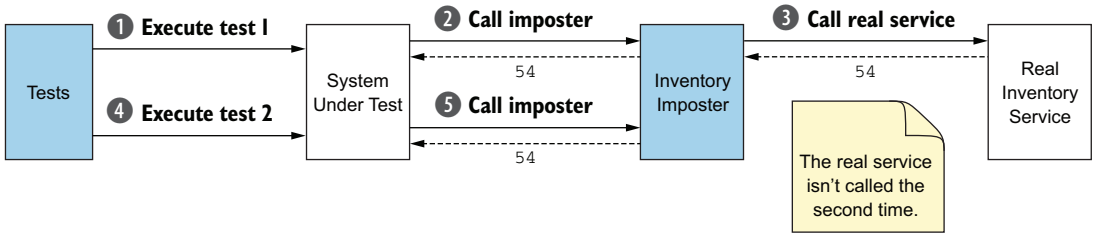

Figure 5.3 By default, the proxy returns the first result as the response to all subsequent calls.

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.

Listing 5.2 Saved proxy responses change imposter state

```
{
  "protocol": "http",
  "port": 3000,
  "numberOfRequests": 1,
  "requests": [],
  "stubs": [{
    "predicates": [],
```

No predicates? We'll come back to that....?

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

```
    "responses": [{
      "is": {
        "statusCode": 200,
        "headers": {
          "Connection": "close",
          "Date": "Sat, 15 Apr 2017 17:04:02 GMT",
          "Transfer-Encoding": "chunked"
        },
        "body": "54",
        "_mode": "text",
        "_proxyResponseTime": 10
      }
    }]
  },
  {
    "responses": [{
      "proxy": {
        "to": "http://localhost:8080",
        "mode": "proxyOnce"
      }
    }]
  }],
  "_links": {
    "self": {
      "href": "http://localhost:2525/imposters/3000"
    }
  }
}
```

Saves the response as an is response

Saves the time to call the downstream service

Your original stub is still there.

Only calls downstream once

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).

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.

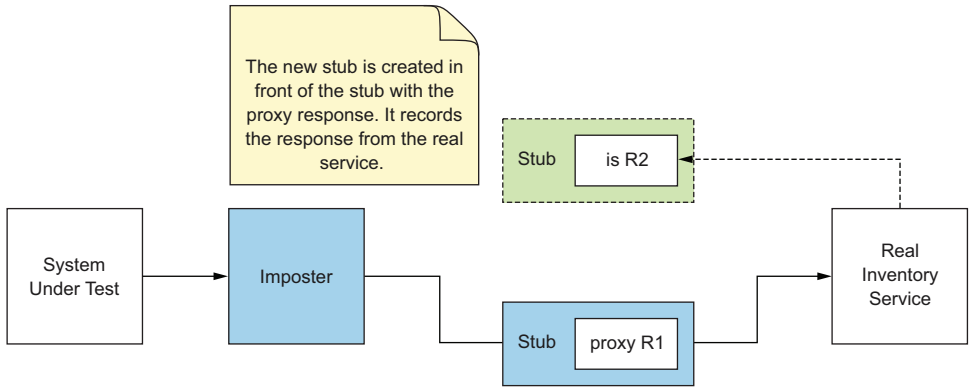

Figure 5.4 The proxy saves the downstream response in a new stub.

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

Proxy new paths to the given base URL…

…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" }
      }],
      "responses": [{
        "is": { "body": "54" }
      }]
    },
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/e1977c9e" }
      }],
      "responses": [{
        "is": { "body": "100" }
      }]
    },
    {
      "responses": [{
        "proxy": {
          "to": "http://localhost:8080",
          "predicateGenerators": [{
            "matches": {
              "path": true
            }
          }],
          "mode": "proxyOnce"
        }
      }]
    }
  ]
}
```

Saves the first call as the first stub

To save space, most of the response isn't shown.

Creates a new stub with a different predicate

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
    }
  }]
}
```

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
      }
    }
  }]
}
```

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": [
        {
          "matches": { "path": true },
          "caseSensitive": true
        },
        {
          "matches": { "body": true }
        }
      ]
    }
  }]
}
```

Generates a case-sensitive predicate

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

Saves the value defined by the selector…

…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" }
 }],
 "responses": [{
    "is": { ... }
 }]
}
```

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": {
        "selector": "//identity[@career='Teacher']/location"
      },
```

Saves the value defined by the selector…

```
      "matches": { "body": true }
    }]
  }
}
```

…from the body field.

The predicates that the proxy creates show the correct location:

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

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.

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).

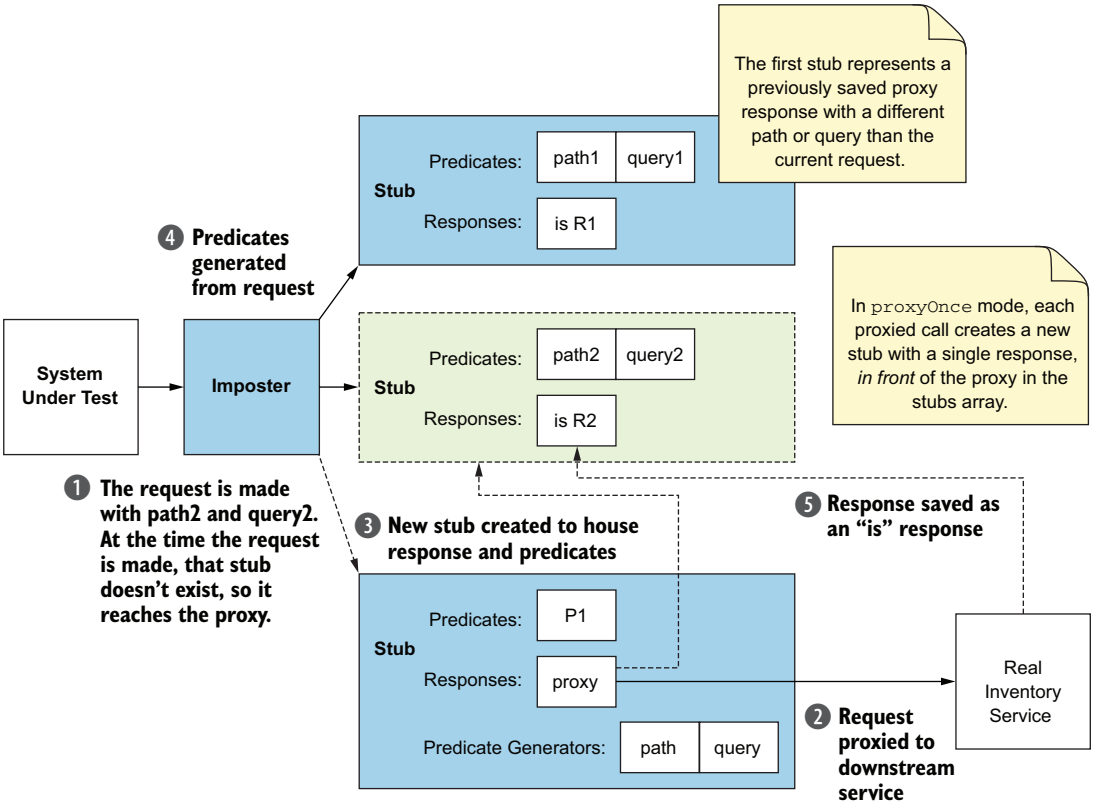

Figure 5.5 In `proxyOnce` mode, mountebank creates new stubs before the stub with the proxy.

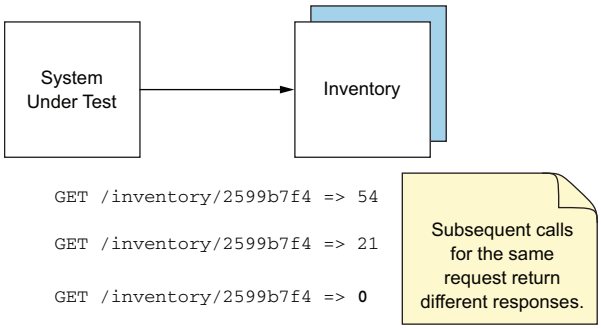

Figure 5.6 Volatile responses for the same request

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).

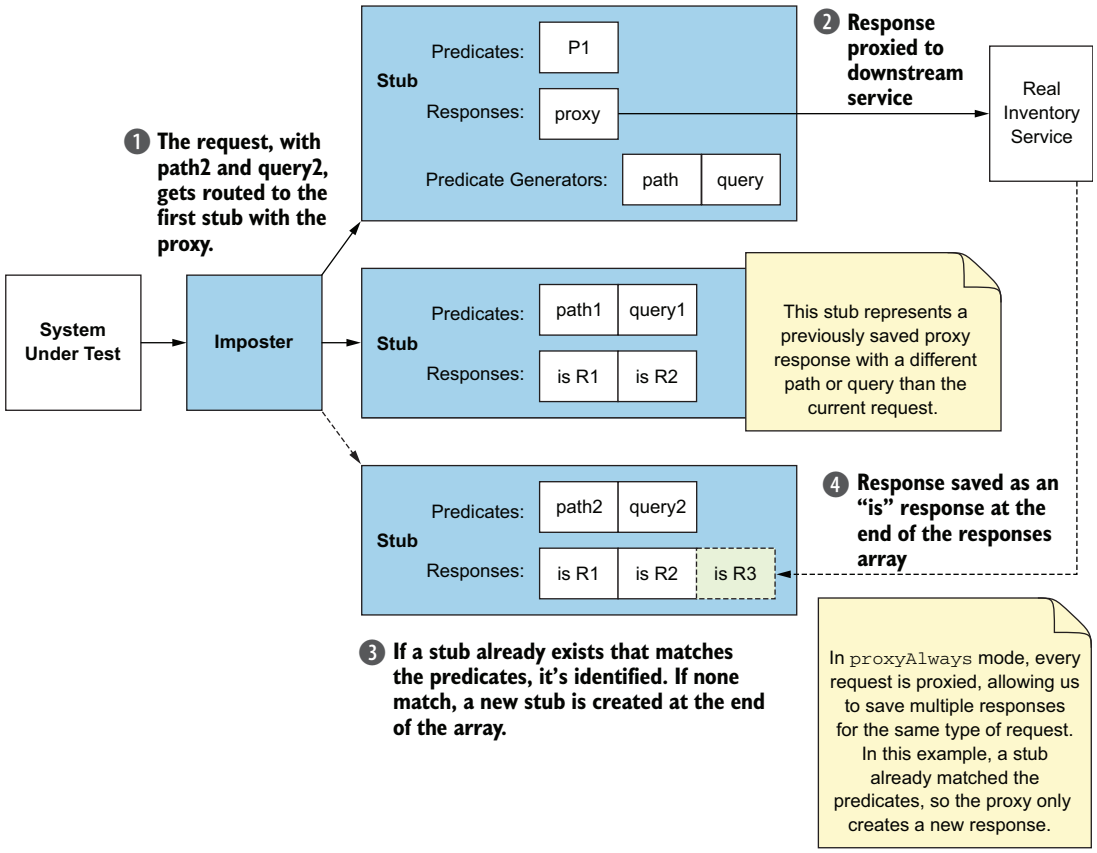

Figure 5.7 In `proxyAlways` mode, new stubs are created after the stub with the proxy

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",
    "predicateGenerators": [{
```

Ensures that all responses are captured

```
      "matches": { "path": true }
    }]
  }
}
```

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",
          "predicateGenerators": [{
            "matches": { "path": true }
          }]
        }
      }]
    },
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/2599b7f4" }
      }],
      "responses": [
        { "is": { "body": "54" } },
        { "is": { "body": "21" } },
        { "is": { "body": "0" } }
      ]
    },
    {
      "predicates": [{
        "deepEquals": { "path": "/inventory/e1977c9e" }
      }],
```

Ensures that all requests are proxied

First request type

All responses saved

Second request type

```
      "responses": [{
        "is": { "body": "100" }
      }]
    }
  ]
}
```

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).

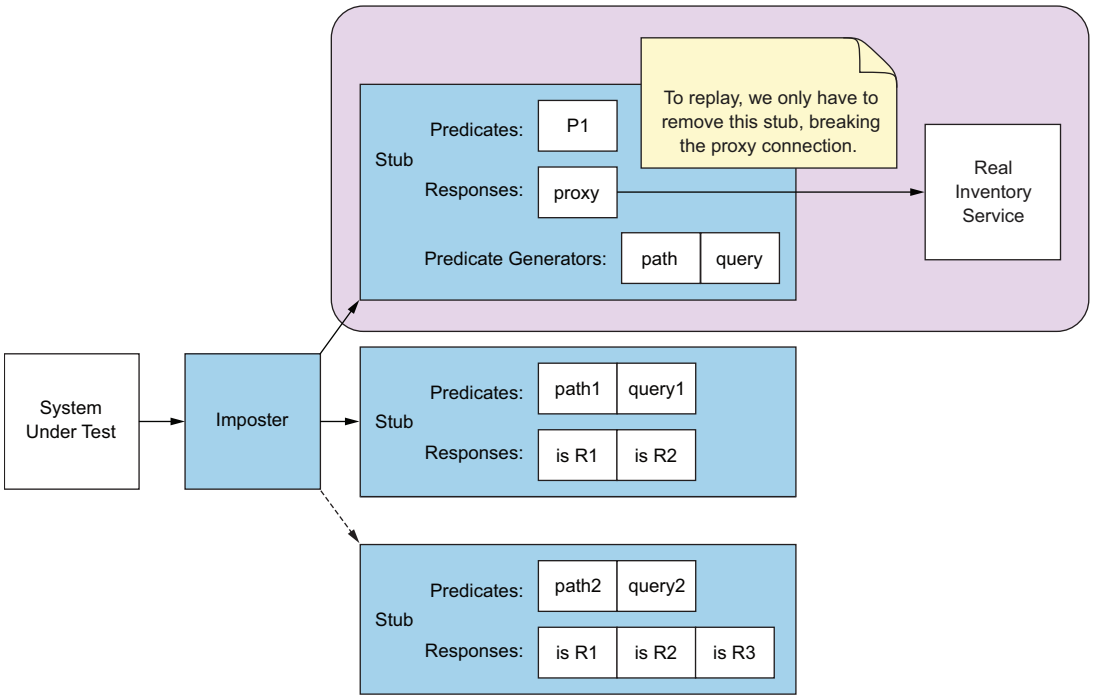

Figure 5.8 Replaying involves removing the proxy

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).

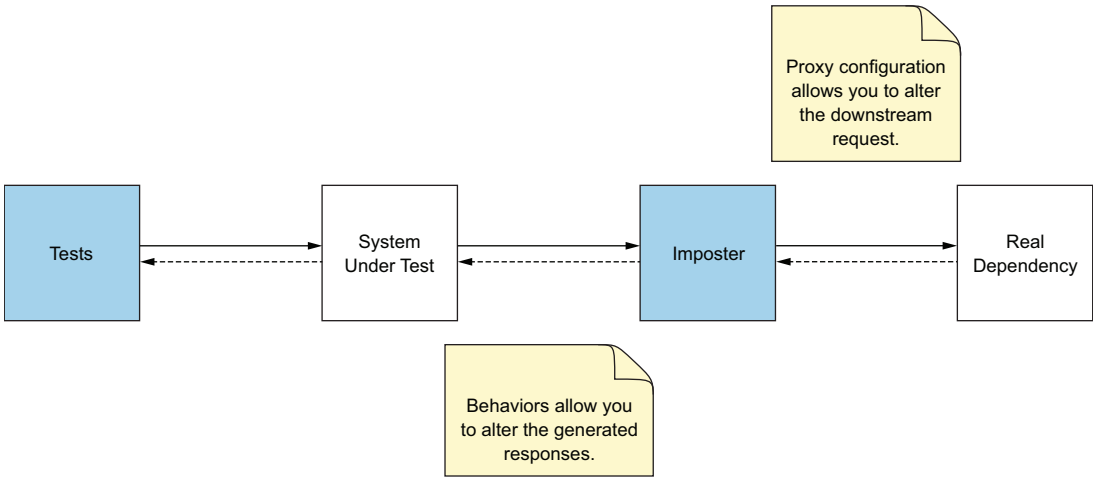

Figure 5.9 You can configure both the proxy request and the generated response.

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).

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...",
    "cert": "-----BEGIN CERTIFICATE-----\n..."
  }
}
```

The actual text is much longer.

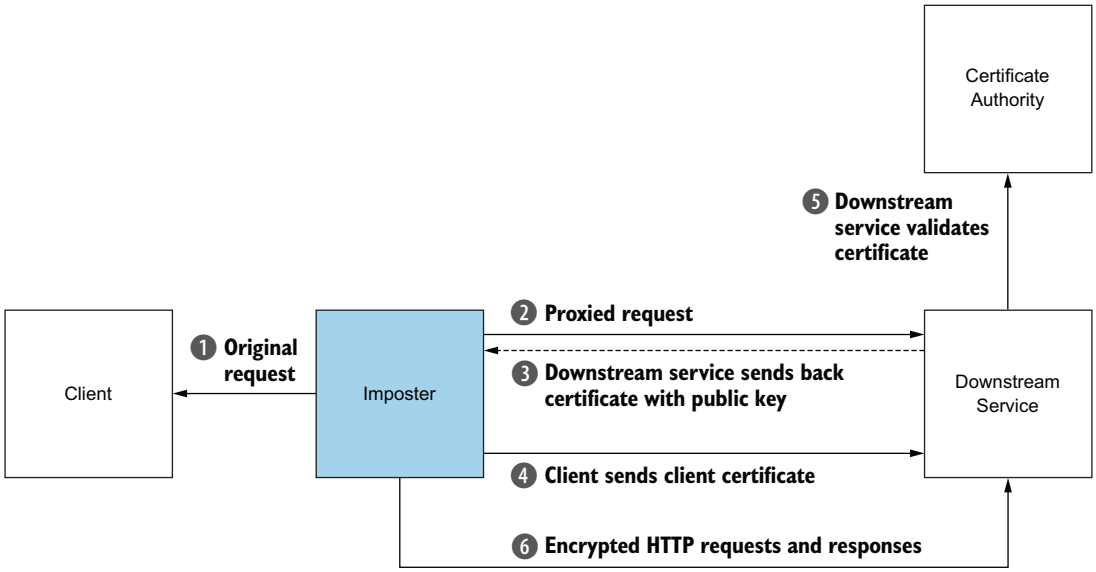

Figure 5.10 Configuring the proxy to pass a client certificate

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).

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

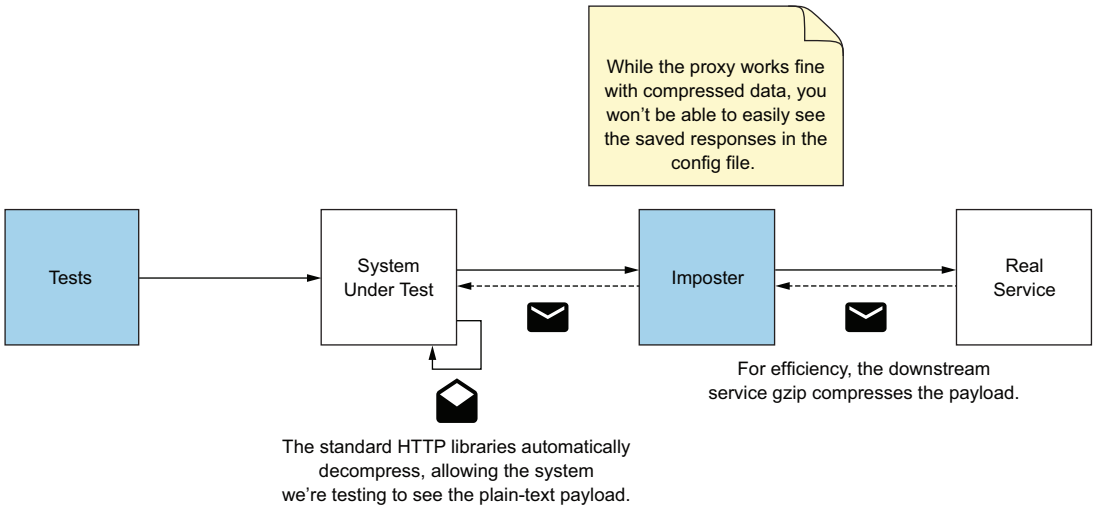

Figure 5.11 Proxying compressed responses

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]

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"
    }
  }
}
```

Prevents response compression

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

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).

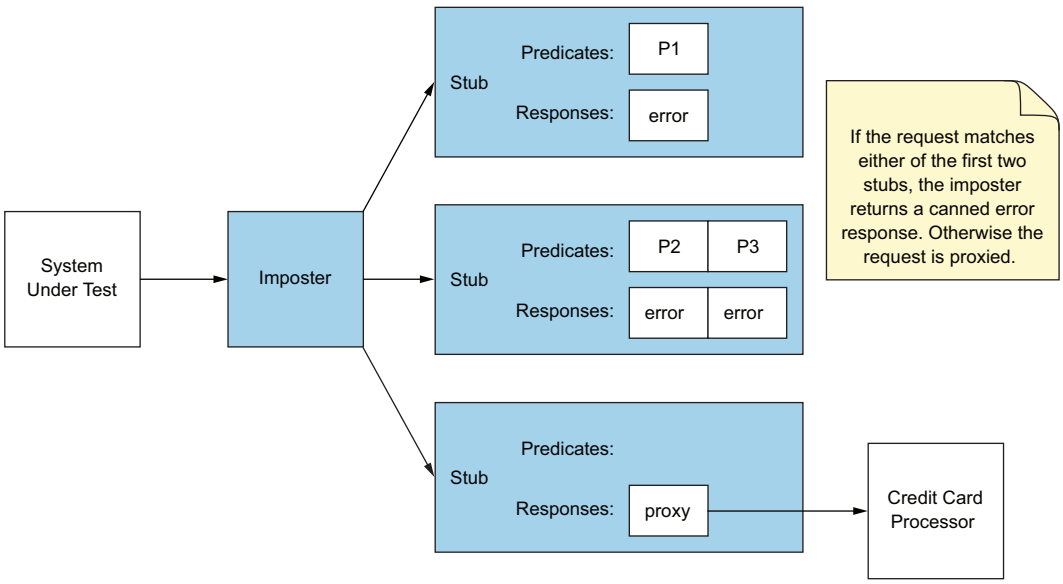

Figure 5.12 Mixing canned responses with a fallback proxy

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": [{
        "contains": { "body": "5555555555555555" }
      }],
      "responses": [{
        "is": { "body": "FRAUD ALERT... " }
      }]
    },
    {
      "predicates": [{
        "contains": { "body": "4444444444444444" }
      }],
      "responses": [{
        "is": { "body": "INSUFFICIENT FUNDS..." }
      }]
    },
    {
      "responses": [{
        "proxy": {
          "to": "http://localhost:8080",
          "mode": "proxyAlways"
        }
      }]
    }
  ]
}
```

If the body contains this credit card #…

…send a fraud-alert response.

If it contains this credit card #…

…send an over-balance response.

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",
  "stubs": [{
    "responses": [{
      "proxy": {
        "to": "https://localhost:8080",
        "mode": "proxyAlways"
      }
    }]
  }]
}
```

The imposter itself is an HTTP server...

...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.

TESTING/SOFTWARE DEVELOPMENT

Testing Microservices with Mountebank

Brandon Byars

Even if you lab test each service in isolation, it's challenging—and potentially dangerous—to test a live microservices system that's changing and growing. Fortunately, you can use Mountebank to "imitate" the components of a distributed microservices application to give you a good approximation of the runtime conditions as you test individual services.

Testing Microservices with Mountebank introduces the powerful practice of service virtualization. In it, author Brandon Byars, Mountebank's creator, offers unique insights into microservices application design and state-of-the-art testing practices. You'll expand your understanding of microservices as you work with Mountebank's imposters, responses, behaviors, and programmability. By mastering the powerful testing techniques in this unique book, your microservices skills will deepen and your applications will improve. For real.

What's Inside

- The core concepts of service virtualization
- Testing using canned responses
- Programming Mountebank
- Performance testing

Written for developers familiar with SOA or microservices systems.

Brandon Byars is the author and chief maintainer of Mountebank and a principal consultant at ThoughtWorks.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/testing-microservices-with-mountebank

"A complete and practical introduction to service virtualization using Mountebank, with lots of usable examples."
—Alain Couniot, STIB-MIVB

"All you need to know to get microservices testing up and running efficiently and effectively."
—Bonnie Bailey, Motorola Solutions

"Shows how to test your microservices and maintain them. You'll learn that tests don't need to be so hard!"
—Alessandro Campeis, Vimar

"A must-have for anyone who is serious about testing microservices. Covers a lot of patterns and best practices that are valuable for promoting service isolation."
—Nick McGinness, Direct Supply

ISBN-13: 978-1-61729-477-8
ISBN-10: 1-61729-477-2
54999
9 781617 294778

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]