Testing Microservices with Mountebank

Brandon Byars

SAMPLE CHAPTER

MANNING

Testing Microservices with Mountebank

by Brandon Byars

Sample Chapter 2

brief contents

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.

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

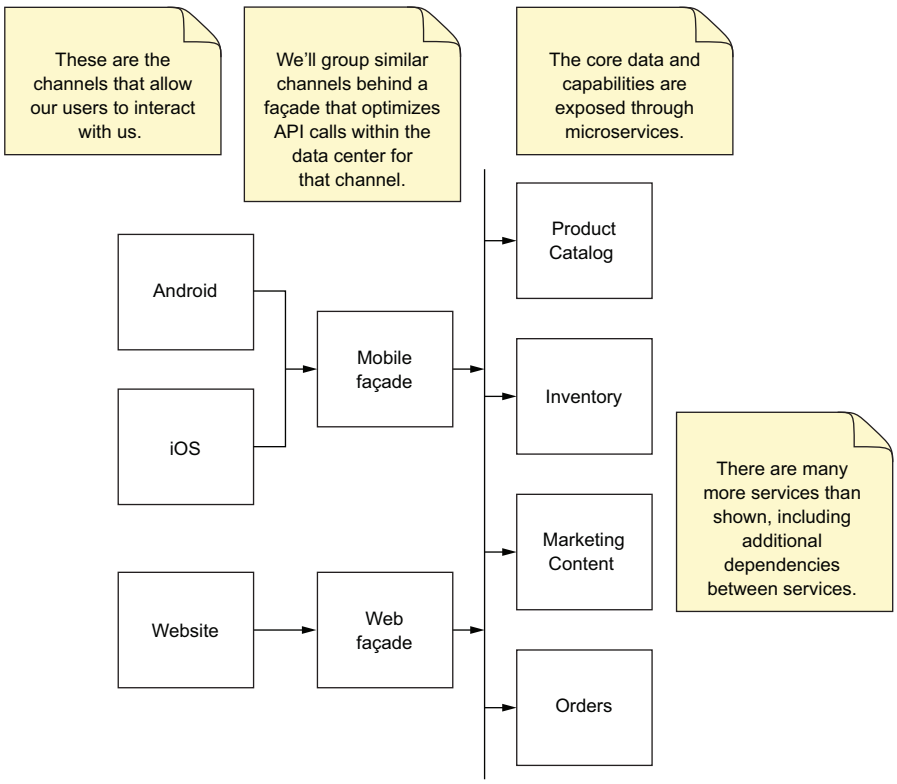

Figure 2.1 Your reference architecture for exploring mountebank

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

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).

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.

2.2 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.

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.

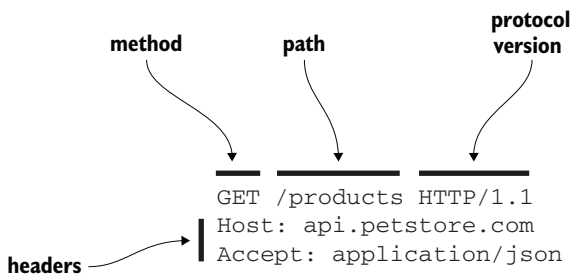

Figure 2.2 Breaking down the HTTP request for products

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

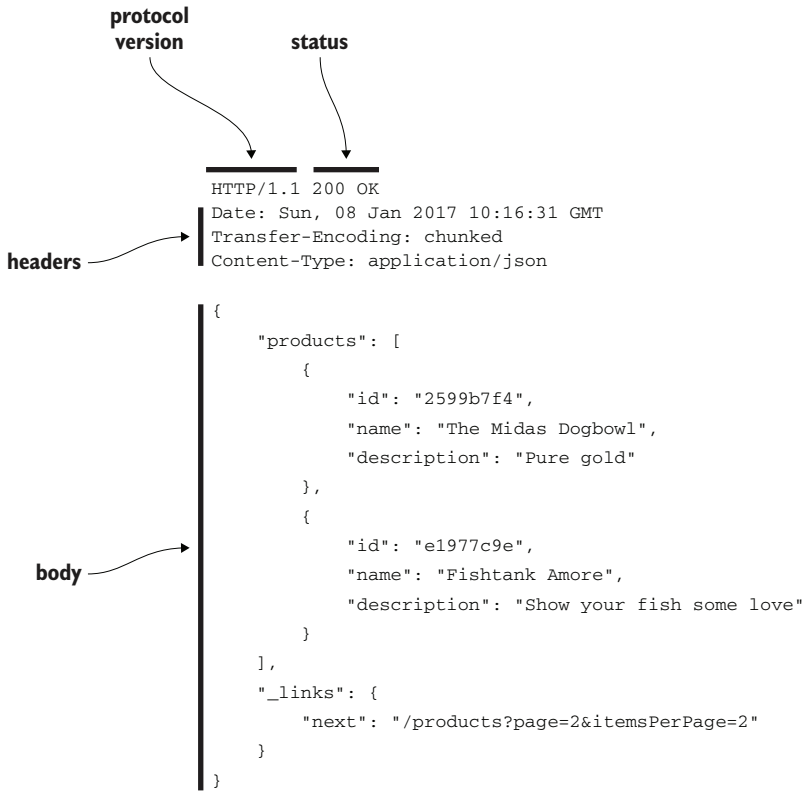

Figure 2.3 The response from the product catalog

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.

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.

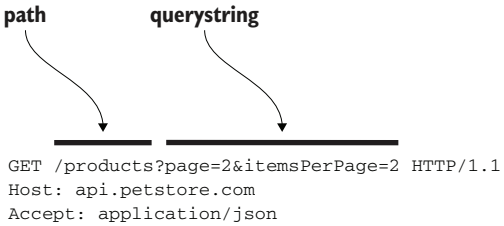

Figure 2.4 Adding a query parameter to an HTTP request

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.

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.

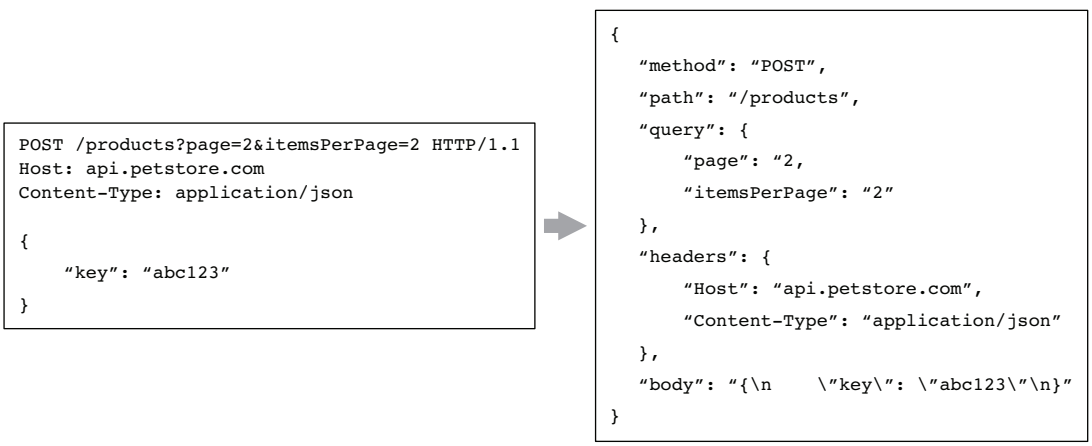

Figure 2.5 How mountebank views an HTTP request

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.

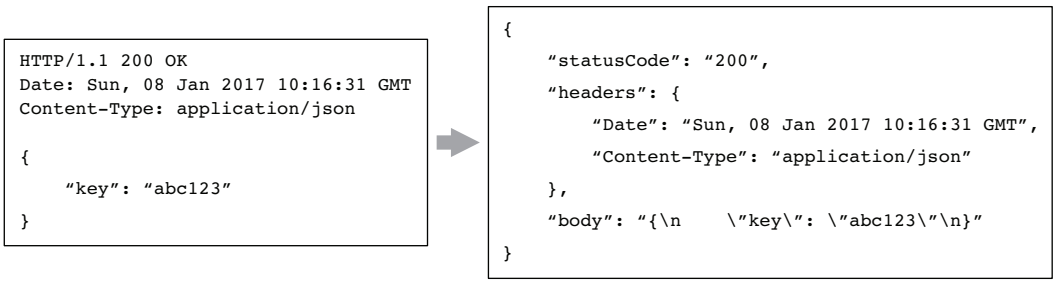

Figure 2.6 How mountebank represents an HTTP response

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).

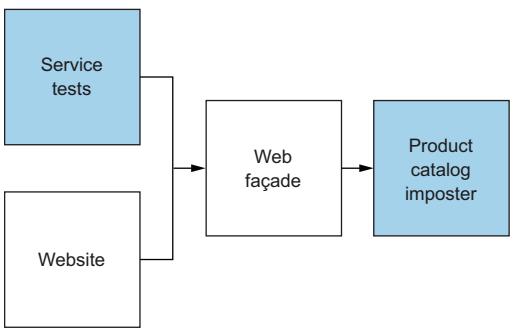

Figure 2.7 Virtualizing the product catalog service to test the web facade

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]

Listing 2.1 Creating an imposter on the command line

```
curl -X POST http://localhost:2525/imposters --data '{
  "port": 3000,
  "protocol": "http",
  "stubs": [{
    "responses": [{
      "is": {
        "statusCode": 200,
        "headers": {"Content-Type": "application/json"},
        "body": {
          "products": [
            {
              "id": "2599b7f4",
              "name": "The Midas Dogbowl",
              "description": "Pure gold"
            },
            {
              "id": "e1977c9e",
              "name": "Fishtank Amore",
              "description": "Show your fish some love"
            }
          ],
          "_links": {
            "next": "/products?page=2&itemsPerPage=2"
          }
        }
      }
    }]
  }]
}'
```

Creates new imposters

Minimally defines each imposter by a port and a protocol

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`.

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

[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.

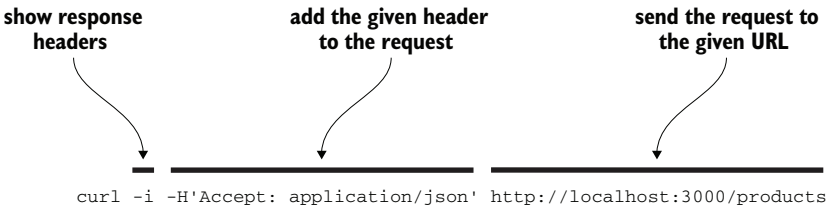

Figure 2.8 Using `curl` to send a request to your virtual product catalog service

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": [
    {
      "predicates": [{
        "equals": {
          "query": { "page": "2" }
        }
      }],
      "responses": [{
        "is": {
          "statusCode": 200,
          "headers": {"Content-Type": "application/json"},
          "body": { "products": [] }
        }
      }]
    },
    {
      "responses": [{
        "is": {
          "statusCode": 200,
          "headers": { "Content-Type": "application/json" },
          "body": {
            "products": [
              {
                "id": "2599b7f4",
                "name": "The Midas Dogbowl",
```

Using two stubs allows different responses for different requests.

Requires that the request querystring include page=2

Sends this response if the request matches the predicate

Otherwise, sends this response

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

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.

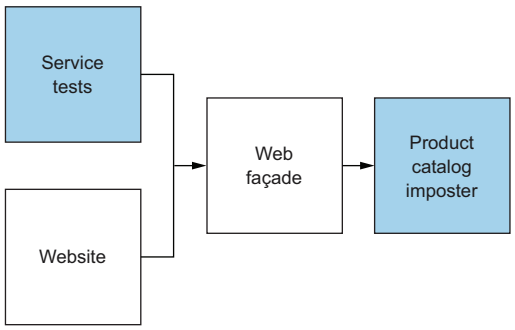

Figure 2.9 Combining product data with marketing copy

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

Comes from the product catalog service, but also will be used to look up content

Comes from the product catalog service

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');
var request = require('request-promise-any');

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

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

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

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

node.js libraries that make calling HTTP services easier

The node.js way of exposing a function to different files

Calls the REST API to create an imposter

The entry point to the fluent interface—each call creates a new stub

Adds a new request predicate to the stub

Adds a new response to the stub

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

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'),
  productPort = 3000;

function createProductImposter() {
  return imposter({
      port: productPort,
      protocol: "http",
      name: "Product Service"
    })
    .withStub()
    .matchingRequest({equals: {path: "/products"}})
    .respondingWith({
      statusCode: 200,
      headers: {"Content-Type": "application/json"},
      body: {
        products: [
          {
            id: "2599b7f4",
            name: "The Midas Dogbowl",
            description: "Pure gold"
          },
          {
```

Imports your fluent interface

Passes the root-level information into the entry function

Adds the request predicate

Adds the response

```
            id: "e1977c9e",
            name: "Fishtank Amore",
            description: "Show your fish some love"
          }
        ]
      }
    })
    .create();
}
```

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: {
        path: "/content",
        query: { ids: "2599b7f4,e1977c9e" }
      }
    })
    .respondingWith({
      statusCode: 200,
      headers: {"Content-Type": "application/json"},
```

Only respond if the path and query match as shown.

```
      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"
          }
        ]
      }
    })
    .create();
}
```

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).

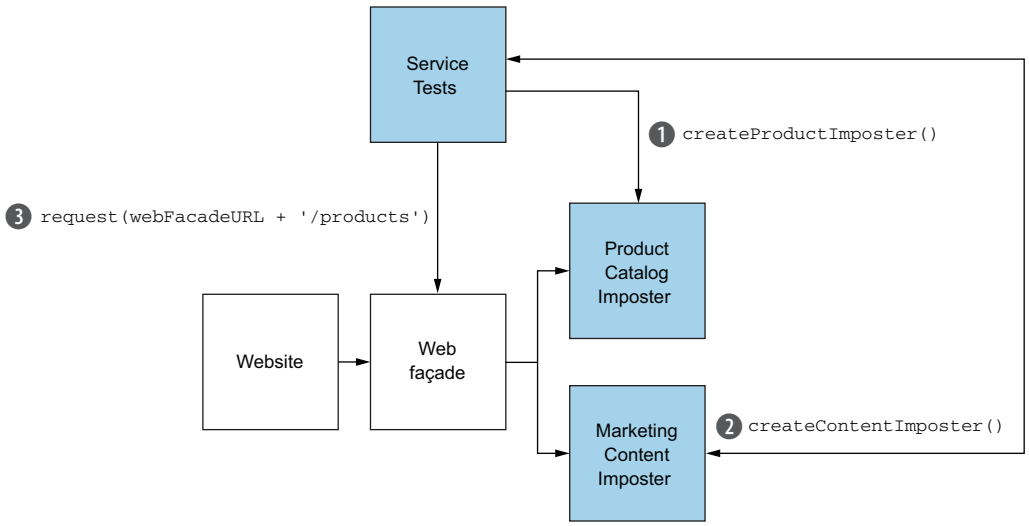

Figure 2.10 The steps of the service test to verify web façade's data aggregating

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

      assert.deepEqual(products, [
        {
          "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();
    }).then(function () {
      done();
    });
  });
});
```

Mocha groups multiple tests in a describe function.

Each it function represents a single test.

Arrange

Act

Assert

Cleanup

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
  });
}
```

Passes in the config object, as in listing 2.5

```
function destroyAll () {
  return request({
    method: "DELETE",
    uri: "http://localhost:2525/imposters"
  });
}
```

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.

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]