deproxy Documentation

Release 0.22-SNAPSHOT

izrik

Mar 17, 2017

Contents

1 How It Works **3**
- 1.1 Deproxy . . . 3
- 1.2 Request/Response . . . 3
- 1.3 Handlings . . . 4
- 1.4 Message Chains . . . 5
- 1.5 Orphaned Handlings . . . 5
- 1.6 Connections . . . 7

2 Using Deproxy in Tests **9**

3 Making Requests **11**
- 3.1 Parameters . . . 11
- 3.2 Named Parameters . . . 12

4 Endpoints **13**
- 4.1 How To: Single Server . . . 13
- 4.2 How To: Auxiliary Service . . . 14
- 4.3 How To: Multiple Servers . . . 15
- 4.4 Routing . . . 16

5 Handlers **19**
- 5.1 Specifying Handlers . . . 19
- 5.2 Built-in Handlers . . . 21
- 5.3 Custom Handlers . . . 22
- 5.4 Handler Context . . . 23
- 5.5 Default Response Headers . . . 24

6 Client Connectors **25**
- 6.1 Built-in Connectors . . . 25
- 6.2 Specifying Connectors . . . 25
- 6.3 Custom Connectors . . . 26
- 6.4 Default Request Headers . . . 28

7 Server Connectors **29**
- 7.1 Built-in Connectors . . . 29
- 7.2 Specifying Connectors . . . 29
- 7.3 Custom Connectors . . . 30

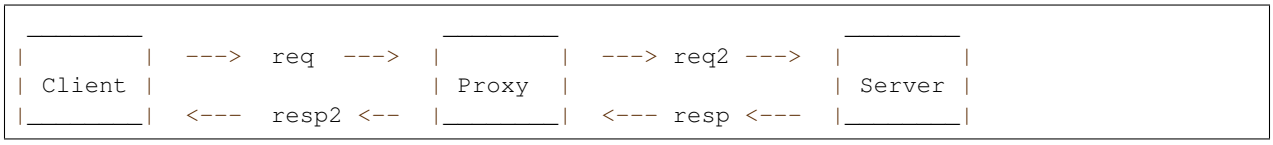
8 Standard Headers 33

9 Search 35

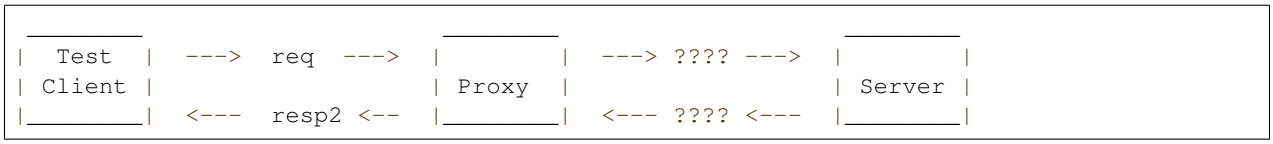
deproxy (or "Deproxy") is a tool for performing high-level, black-box, functional/regression testing of proxies, and other HTTP intermediaries. It is written in Groovy (ported from python). It is meant to be incorporated into unit tests for functional testing.

Testing normal client/server interaction is relatively straight-forward: Use a specialized test client to send requests to the server, and compare the response that the server returns to what it ought to return.:

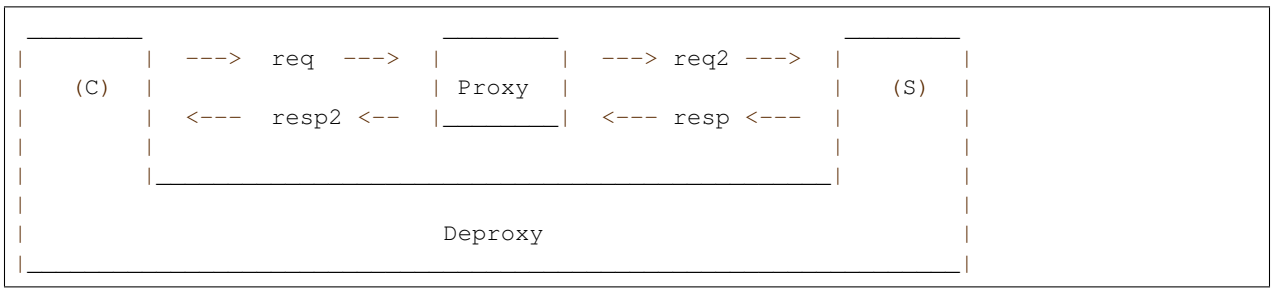
Proxies sit in-between an HTTP client (e.g. novaclient) and an HTTP server (e.g. the Nova API nodes). This makes testing a little more difficult.:



A proxy can modify either the incoming request to the server, or the outgoing response to the client, or both. In addition, it may handle the request itself (e.g. in the case of caching or authenication), and prevent it from reaching the server in the first place. The functionality and positioning of the proxy provides more of a challenge to functionality testing. The traditional model is not enough. Because a test client only sees one side of the transaction, it can't make definitive determinations about the server's side of it.



If we don't have a copy of the request that the server received, then we can't compare it to the request sent, which means we don't know for sure that the proxy is modifying it correctly. Likewise, if we don't have a copy of the response that the server originally sent, we can't conclusively prove that the proxy is modifying responses correctly. [Some specific cases don't have this problem, such as whether the proxy overwrites the "Server" header on a response; that can be confirmed because a response will only ever have one "Server" header, and that can easily be checked by a test client.] But in the general case, we can't say for sure about other functional requirements. Additionally, if the proxy is required to prevent a request from even reaching the server (as in the case of invalid authentication credentials in the request) a test client cannot determine whether or not any such request was in fact forwarded, because all it sees is the error response from the proxy. For that, we'd need to be able see both sides of the exchange, and record all requests that made it to the server. And that is what a deproxy does. It de-proxifies the proxy:



A deproxy acts as both the client and the server, and the proxy it is testing will forward requests from one side to the other. Any requests received by the server side are matched up with the requests that started them. A call to a deproxy's makeRequest method will return the request that the client side sent, the request that the server side received, the response that the server side sent, and the response that the client side received. In this way, we can conclusively prove whether or not the proxy modified requests and responses correctly. We can even conclusively show when no

request makes it to the server in the first place, because the receivedRequest and sentResponse fields will be null.

But this is just scratching the surface. The *com.rackspace.deproxy* package contains additional tools and utilities for custom server responses, mocking, testing multiple endpoints, and more.

Contents:

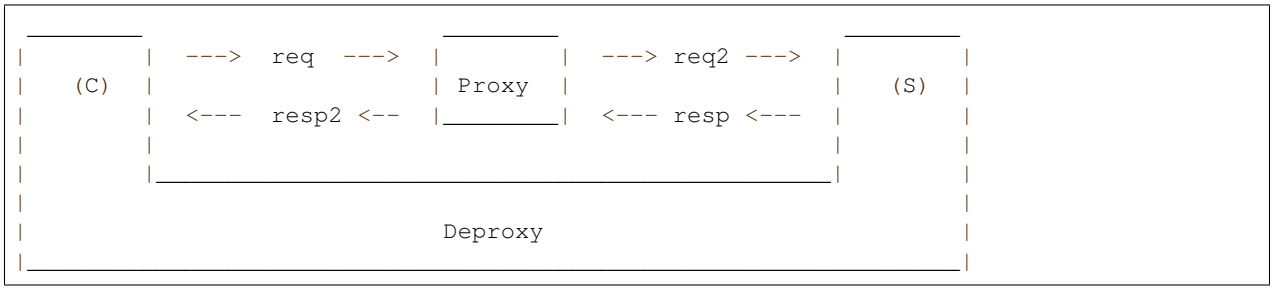
CHAPTER 1

How It Works

Deproxy is intended to test complex HTTP proxies, rather than just clients or servers. For this reason, it deals with more than just HTTP requests and responses. An entire complex system of HTTP components can be involved, and Deproxy keeps track of them all. What follows is a description of various scenarios and how to test them using Deproxy.

Deproxy

At the heart of the system is the Deproxy class. It acts as both the client and server on opposite sides of a proxy to be tested:



You can use this in your test code by creating an instance of the Deproxy class, potentially adding one or more endpoints (about which, more *here*), and then calling the makeRequest method to initiate an HTTP request from the client side to the proxy. makeRequest allows you to craft a custom request, including the HTTP method, headers, and request body to send. You can also indicate how the server-side should respond and what additional complex behavior the client-side should exhibit (e.g. re-using connections).

Request/Response

We'll start by describing the simplest possible arrangement: a client makes a request to a server.:

In this instance, all we have to keep track of is the request sent and the response received. To test a server, all we would need to do is use an HTTP client to send a request to the server, then compare what the server got back with what we expected. Simple enough, right? So simple, in fact, that we wouldn't even really need Deproxy to test it.

Nevertheless, Deproxy contains the facilities to do so. Simply, create a Deproxy and then call it's makeRequest method, specifying the URL of the server. It will create a Request object, and send it over the wire. Then, it will receive a response from the server, and convert it into a Response object. Each of these classes stores basic information about HTTP messages;

- Request stores a `method` and a `path`, both as strings.
- Response stores a `code` and a `message`, both as strings.
- Both classes store a collection of headers. This collection stores headers as name/value pairs, in the order that they travel across the wire. You can also do by-name lookup, which is case-insensitive.
- Both classes store an optional message body. The message body will be either a string or an array of `bytes`, depending on whether Deproxy could figure out what kind of data it is.

The Request sent and the Response recieved will be returned back to you from makeRequest, along with a bunch of other stuff. Then you can make assertions on the request and response as in any unit test. That's client/server testing in a nutshell.

Handlings

Next, let's consider a situation with more moving parts.:

Now things are getting interesting.

1. The client sends a request to the proxy
2. The proxy potentially modifies the request and sends it along to the server
3. The server returns a response to the proxy
4. The proxy potentially modifies the responses and sends it bck to the client

If our goal is to test the behavior of the proxy and the modifications it makes to requests, responses, or both, then we have to keep track of more information. Not only that, we need to distinguish between two Request/Response exchanges. We can create an Endpoint to represent the server, and make requests to the proxy using the makeRequest method. When the endpoint receives a request from the proxy, it will return a response. We say that it "handles" the request. Both the request the endpoint receives and the response it sends back are collected into something called a Handling. A handling represents the request/response pair at the server side of the equation. So, the call to makeRequest should return:

- the sent request
- the received request and sent response, as a Handling
- the received response

But to have a more complete model, we should consider additional cases.

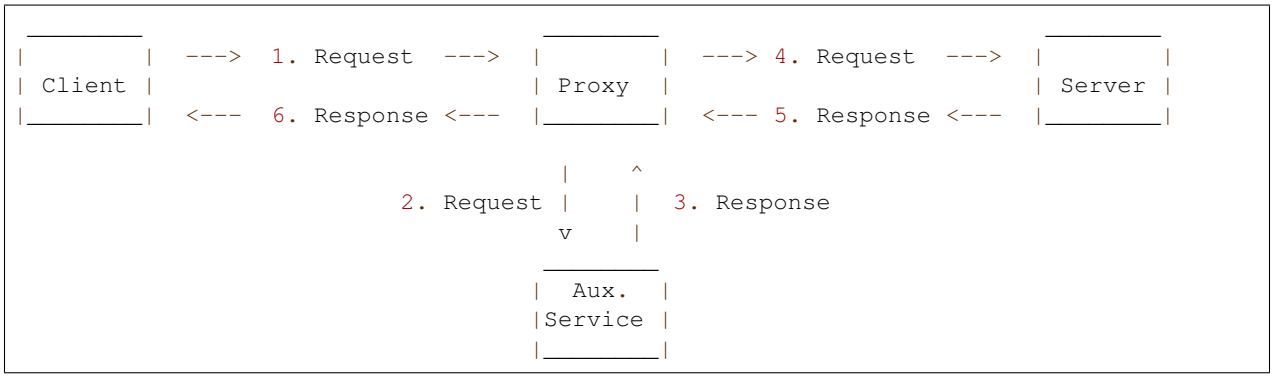
Here's another situation in which the Handling concept proves helpful:

Suppose a proxy is limiting requests to the server to X per minute. Or serving responses out of a cache. Or something like that. In these cases, there are circumstances in which we expect the proxy to *not* forward the request to the server, but instead to serve a response itself, whether an error or a cached response.

As it turns out, we can test whether or not the proxy is forwarding requests, in addition to checking that the response is correct. If the mock-server endpoint never receives a request, then it never generates a response and no Handling object is generated.

Message Chains

But what if we want to track more than a single Handling? Consider another situation. Suppose the proxy that we're testing is used to authenticate client requests *before* forwarding them on to the server. And suppose further that this authentication has to go through an auxiliary, shared service that manages authentication for a number of different servers and services. When a client sends a request, the proxy takes the credentials, and asks the authentication service whether the credentials are correct or not. If correct, the proxy will forward the request to the server; if not, the proxy will return an error code to the client.

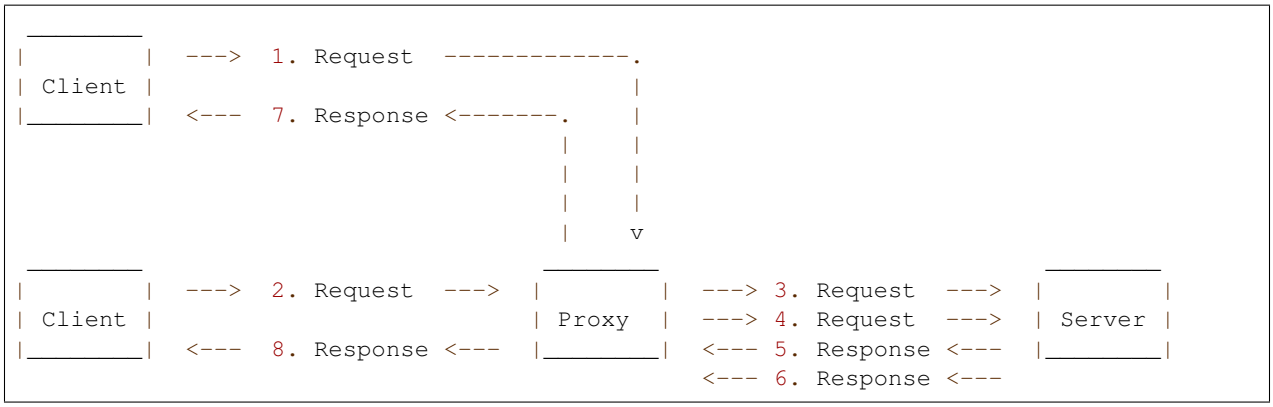


Now we have to keep track of more than one request coming from the proxy, and more than one handling. Moreover, the proxy might have to make multiple requests to the auxiliary service. Or there could be multiple auxiliary servers that the proxy must coordinate with, each doing something different. In order to test the proxy's behavior in all of these situations, we need to keep track of a lot more stuff. Ultimately, what we need is a comprehensive record of everything that happens as a result of the original request the client made. We call that a MessageChain. Everything from the client to the proxy to the auxiliary services to the end server and back again is stored in a single, easy-to-assert object.

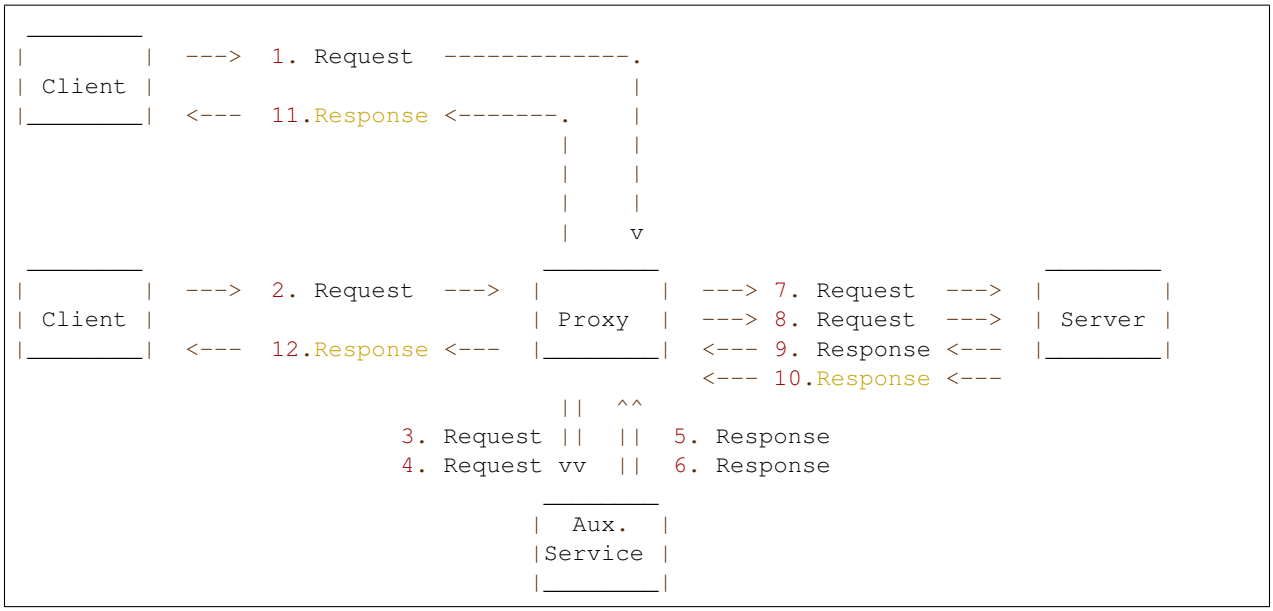
We can simulate the auxiliary service using a second Endpoint in addition to the first. That endpoint can be made to return canned responses to the proxy's authentication requests. All handlings from both endpoints will be stored in a single MessageChain object, which makeRequest returns back to its caller.

Orphaned Handlings

In order to Deproxy keeps track of separate MessageChains as a result of separate calls to makeRequest. This is even the case when making simultaneous calls on different threads.



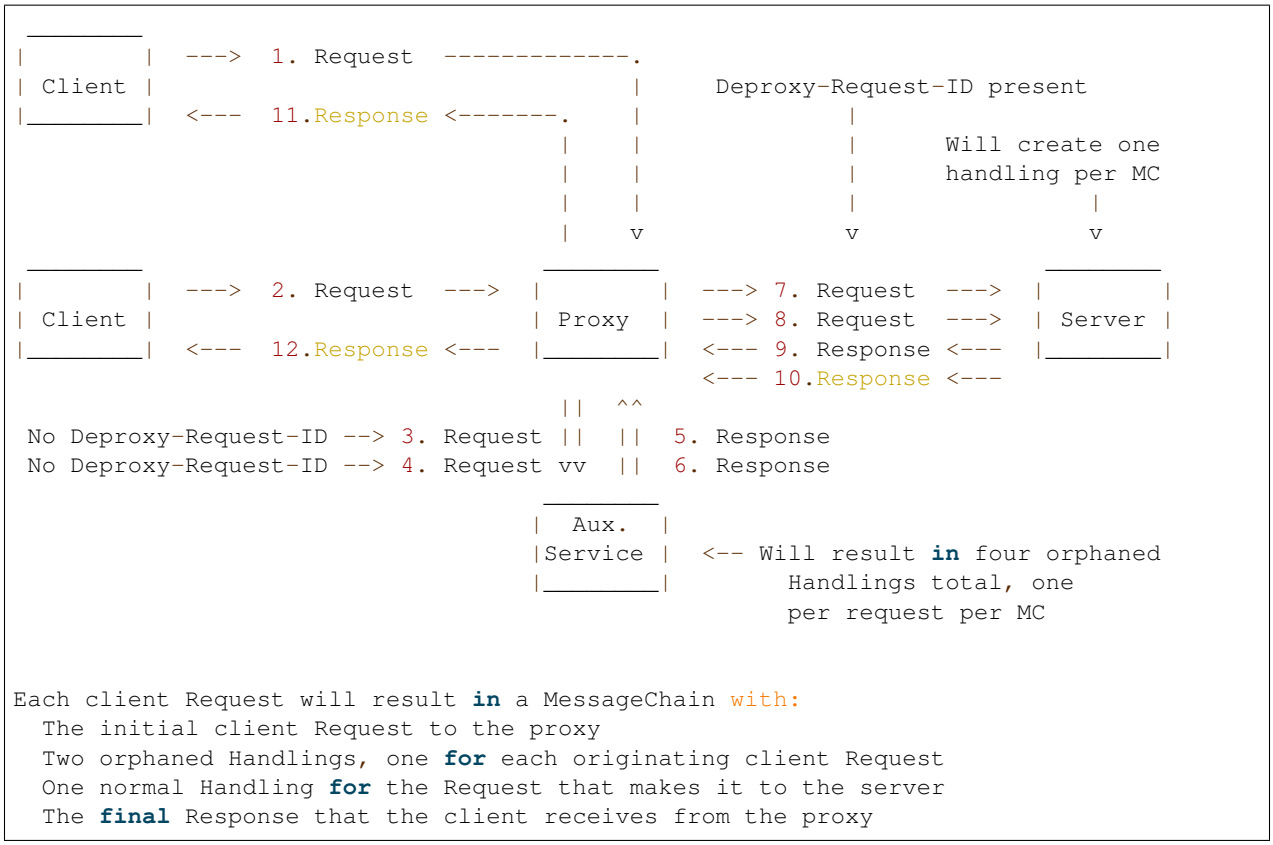
In such a situation, there needs to be a way to distinguish which requests are associated with which MessageChains when they reach the server. Depending on the timing, the second request made might reach the server first. In order to keep track, makeRequest adds a special tracking header (Deproxy-Request-ID) with a unique identifier to each outgoing request, and associates it with the MessageChain for that request. Typically, a proxy won't remove such a header from the request unless configured to do so, so this a reasonably safe way to keep track. When the request reaches the endpoint, the tracking header value is used to get the associated MessageChain for the originating call to makeRequest, and a Handling object is added to the list.



A problem arises, however, in cases where a request reaches an endpoint without the tracking header. This could happen a number of ways:

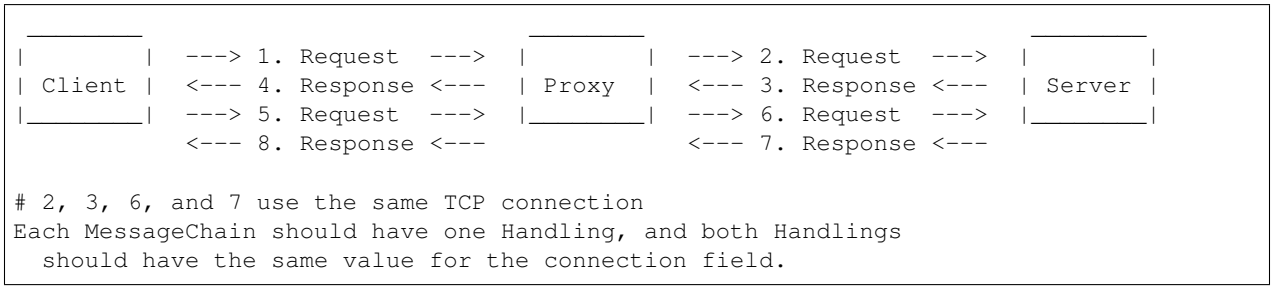
- The proxy might be configured to remove all but a certain predetermined white-list of headers
- The proxy might be initiating a new request to an auxiliary service, which wouldn't retain the tracking header
- A completely unrelated request might have reached the endpoint from another source

Whatever the cause, it represents a problem for us, because it's not possible to tie the Handling to a particular MessageChain without the tracking header. We call this an *orphaned* handling, and store it in the MessageChain's orphanedHandlings field. Instead, what the endpoint will do is add the Handling to *all* active MessageChains as an orphaned handling.



Connections

HTTP applications typically have support for persistent connections, which allow for multiple HTTP transactions using the same TCP connection. In Deproxy, when an endpoint receives a new connection, the connection is given a unique id. All Handling objects created by that endpoint from that TCP connection are tagged with the connection's id value. If we want to test whether or not the proxy is using connection pooling, for example, we could simply make two identical calls to makeRequest. Assuming that the requests are forwarded by the proxy to the server and is re-using connections, the MessageChains that we get back will each have a single Handling object and both Handling objects will have the same `connection` value. If the proxy is not re-using connections, then the two Handling objects will have different `connection` values.



CHAPTER 2

Using Deproxy in Tests

To use deproxy in your unit tests:

1. In the test class's setup method, create a Deproxy object and endpoint(s), and configure your proxy to forward requests to the endpoint's port.
2. In the actual test method, use the makeRequest method to send a request to the proxy, and get a message chain back.
3. Still in the test method, make assertions against the returned message chain.
4. In the cleanup method, shutdown the Deproxy object by calling shutdown().

Here's a code example of a unit test that tests the fictional theProxy library:

```
import org.theProxy.*
import com.rackspace.deproxy.*
import org.junit.*
import static org.junit.Assert.*

class TestTheProxy {

    Deproxy deproxy
    Endpoint endpoint
    TheProxy theProxy

    @Before
    void setup() {

        deproxy = new Deproxy()
        endpoint = deproxy.addEndpoint(9999)

        // Set up the proxy to listen on port 8080, forwarding requests to
        // localhost:9999
        theProxy = new TheProxy()
        theProxy.port = 8080
        theProxy.targetHostname = "localhost"
        theProxy.targetPort = 9999
```

```
        // Set up the proxy to add an X-Request header to requests
        theProxy.requestOperations.add(
            addHeaderOperation(name:  "X-Request",
                               value: "This is a request"))

        // Set up the proxy to add an X-Response header to responses
        theProxy.responseOperations.add(
            addHeaderOperation(name:  "X-Response",
                               value: "This is a response"))
    }

    @Test
    void testTheProxy() {

        def mc = deproxy.makeRequest(method: "GET",
                                     url: "http://localhost:8080/")

        // the endpoint returns a 200 by default
        assertEquals("200", mc.receivedResponse.code)

        // the request reached the endpoint once
        assertEquals(1, mc.handlings.size())

        // the X-Request header was not sent, but was added by the proxy and
        // received by the endpoint
        assertFalse(mc.sentRequest.headers.contains("X-Request"))
        assertTrue(mc.handlings[0].request.headers.contains("X-Request"))

        // the X-Response header was not sent by the endpoint, but was added
        // by the proxy and received by the client
        assertFalse(mc.handlings[0].response.headers.contains("X-Response"))
        assertTrue(mc.receivedResponse.headers.contains("X-Response"))
    }

    @After
    void cleanup() {

        if (theProxy) {
            theProxy.shutdown()
        }
        if (deproxy) {
            deproxy.shutdown()
        }
    }
}
```

CHAPTER 3

Making Requests

The makeRequest method is the primary means of sending requests to HTTP applications. It prepares a Request object to be sent, constructs a MessageChain object to track the request and specify custom handlers, and passes the Request object to the ClientConnector.

Parameters

```
public MessageChain makeRequest(
    String url,
    String host="",
    port=null,
    String method="GET",
    String path="",
    headers=null,
    requestBody="",
    defaultHandler=null,
    Map handlers=null,
    boolean addDefaultHeaders=true,
    boolean chunked=false,
    ClientConnector clientConnector=null) { ... }
```

- `url` - The URL of the request to be made. This will be broken up into scheme, host, port, and path (and query parameter) components. The host, and port will be passed to the client connector to be used to make the connection, and the path will form part of the Request object. Parts of this can be overriden by other parameters. This parameter gets passed to java.net.URI, so it must be a valid uri, with no bad characters. If you need to send invalid data in the request for testing purposes, use the `host` and `path` parameters.
- `host` - The host to which the request will be sent. If both `host` and `url` are given, `host` will override the host component of `url`.
- `port` - The port to which the request will be sent. If both `port` and `url` are given, `port` will override any host component of `url`.

- `method` - The HTTP method of the Request object. This is typically `GET`, `POST`, `PUT`, or some other method defined in RFC 2616 § 5.1.1 and RFC 2616 § 9 . However, deproxy will allow any string, to test custom extension methods and invalid methods. The default is `GET`.
- `path` - The path of the Request object. If both `path` and `url` are given, `path` will override the path component of `url`.
- `headers` - The headers of the Request object. This parameter can be a map, with key-value pairs corresponding to "Key: Value" headers in arbitrary order, or a HeaderCollection with preserved order.
- `requestBody` - The body of the Request object.
- `defaultHandler` - A handler to service the request. Any endpoint that receives this request (or, more accurately, a request with the same Deproxy-Request-ID header) will use `defaultHandler` instead of it's own default. This is a good way to customize per-request handling of a few requests while still relying on the endpoints default handler to cover most other requests. See *Handler Resolution Procedure*, step 2.
- `handlers` - A map of endpoints (or endpoint names) to handlers. If an endpoint or its name is a key in the map, and that endpoint receives this request (or request with the same Deproxy-Request-ID header), then that endpoint will use the value associated with the endpoint to handle the request, instead of relying on the endpoint's own default handler. See *Handler Resolution Procedure*, step 1.
- `addDefaultHeaders` - A boolean value that instructs the client connector to add default request headers to the request before sending. Custom connectors are not required to honor this parameter. See *Default Request Headers*. The default is `true`.
- `chunked` - A boolean value that instructs the client connecto to send the request body using the `chunked` transfer encoding. If `addDefaultHeaders` is true, the DefaultClientConnector will also add the appropriate `Transfer-Encoding` header. The default is `false`.
- `clientConnector` - A custom ClientConnector. If not given, whatever was specified as the `defaultClientConnector` parameter to the Deproxy constructor will be used.

Note that there are multiple ways to specify some information. For example, if no value is given for the `path` parameter, then it will be taken from the path component of `url`. But if both `path` and `url` are given, then `path` will override `url`. The same goes for `host` and `port`.

Named Parameters

makeRequest has a special override to handle named parameters. The following are equivalent:

```
deproxy.makeRequest("http://example.com/resource?name=value", null, null, "GET")

deproxy.makeRequest(url: "http://example.com/resource?name=value", method: "GET")

deproxy.makeRequest(method: "GET", url: "http://example.com/resource?name=value")
```

CHAPTER 4

Endpoints

Endpoints are objects that represent the server-side of http transactions. They are instances of the Endpoint class. Endpoint receive HTTP requests and return HTTP responses. The responses are generated by *handlers*, which can be static functions, object instance methods, or closures. Handlers can be set when the endpoint is created, or specified on a per-client-request basis. Endpoints are created using the addEndpoint method of the Deproxy class. (Instantiating an Endpoint object via the construct is discouraged.)

Endpoints are very flexible. You can create intricate testing situations with them in combination with custom handlers.

How To: Single Server

In the simplest case, a proxy sits in between the client and a server.

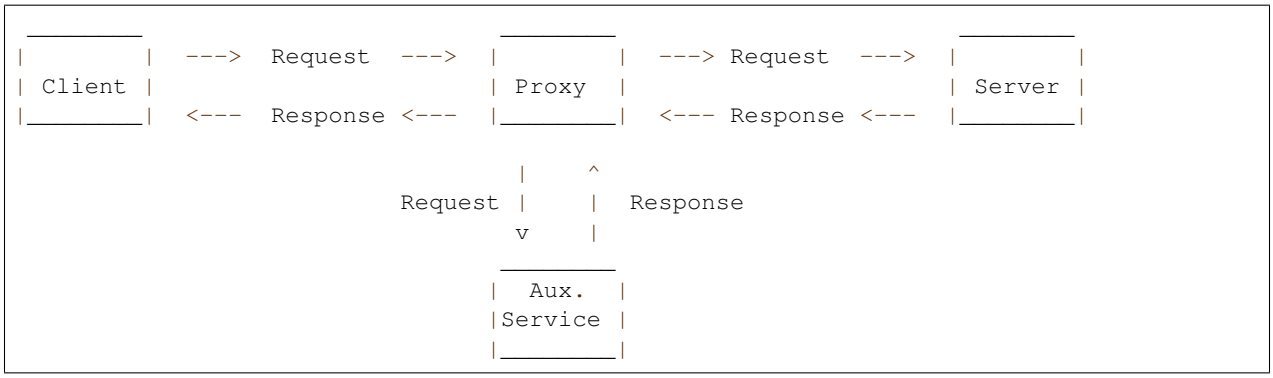
```
 _________                 _________                 _________
|         |  --->  Request  --->  |         |  ---> Request  --->  |         |
| Client  |                       | Proxy   |                      | Server  |
|_________|  <---  Response <---  |_________|  <--- Response <---  |_________|
```

Since we're testing the proxy, we want to be able to control the responses that the server sends in reaction to requests from the proxy. We can simulate the server using a single Endpoint. By default, the endpoint will simply return 200 OK responses, unless it is given a different handler upon creation.

```
Deproxy deproxy = new Deproxy()

Endpoint endpoint = deproxy.addEndpoint(9999)

def theProxy = new TheProxy(port: 8080,
                            targetHostname: "localhost",
                            targetPort: 9999)

def mc = deproxy.makeRequest(url: "http://localhost:8080/path/to/resource")

assert mc.receivedResponse.code == "200"
assert mc.handlings.size() == 1
```

How To: Auxiliary Service

A more complicated case is when the proxy has to call out to some auxiliary service for additional information.



An excellent example would be an authentication system. The client sends the request to the proxy and includes credentials. In order to determine if the credentials are valid, the proxy makes a separate HTTP request to the auth service, which then responds with yea or nay. Depending on whether the credentials are valid or not, the proxy will either forward the request on to the server, or return an error back to the client.

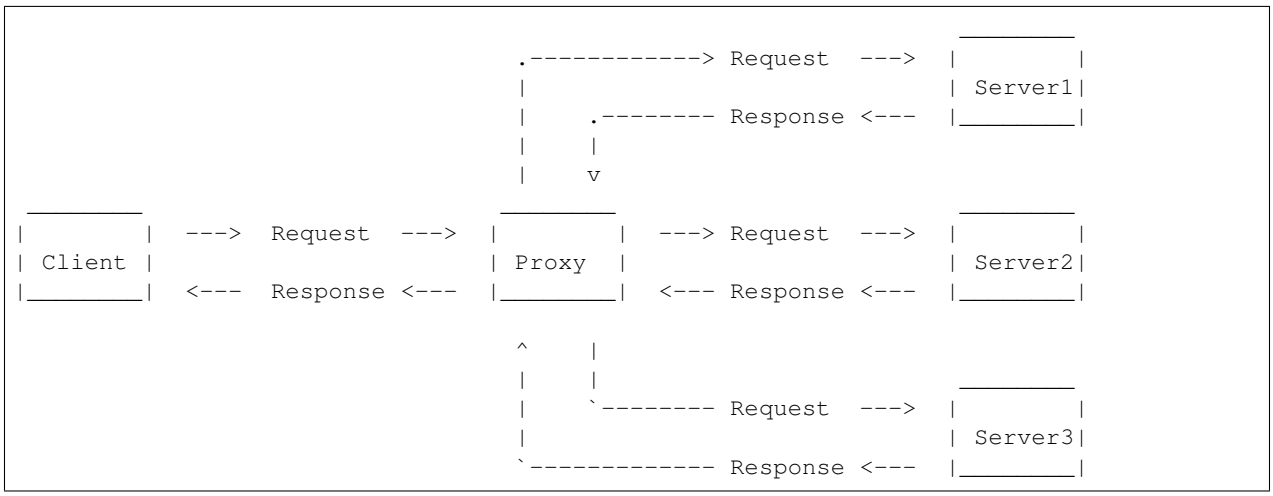
In this setup, we can simulate both the server and the auxiliary service with Endpoint objects. The endpoint representing the auth service would have to be given a custom handler, that could interpret and respond to the authentication requests that the proxy makes according to whatever contract is necessary.

```
Deproxy deproxy = new Deproxy()

def endpoint = deproxy.addEndpoint(9999)

def authResponder = new AuthResponder()
def authService = deproxy.addEndpoint(7777, defaultHandler: authResponder.handler)

def theProxy = new TheProxy(port: 8080,
                            targetHostname: "localhost",
                            targetPort: 9999,
                            authServiceHostname: "localhost",
                            authServicePort: 7777)

def mc = deproxy.makeRequest(url: "http://localhost:8080/path/to/resource",
                             headers: ['X-User': 'valid-user'])

assert mc.receivedResponse.code == "200"
assert mc.handlings.size() == 1
assert mc.handlings[0].endpoint == endpoint
assert mc.orphanedHandlings.size() == 1
assert mc.orphanedHandlings[0].endpoint == authService

def mc = deproxy.makeRequest(url: "http://localhost:8080/path/to/resource",
                             headers: ['X-User': 'invalid-user'])

assert mc.receivedResponse.code == "403"
assert mc.handlings.size() == 0
assert mc.orphanedHandlings.size() == 1
```

```
assert mc.orphanedHandlings[0].endpoint == authService
```

How To: Multiple Servers

Sometimes, a proxy might be set up in front of multiple servers.



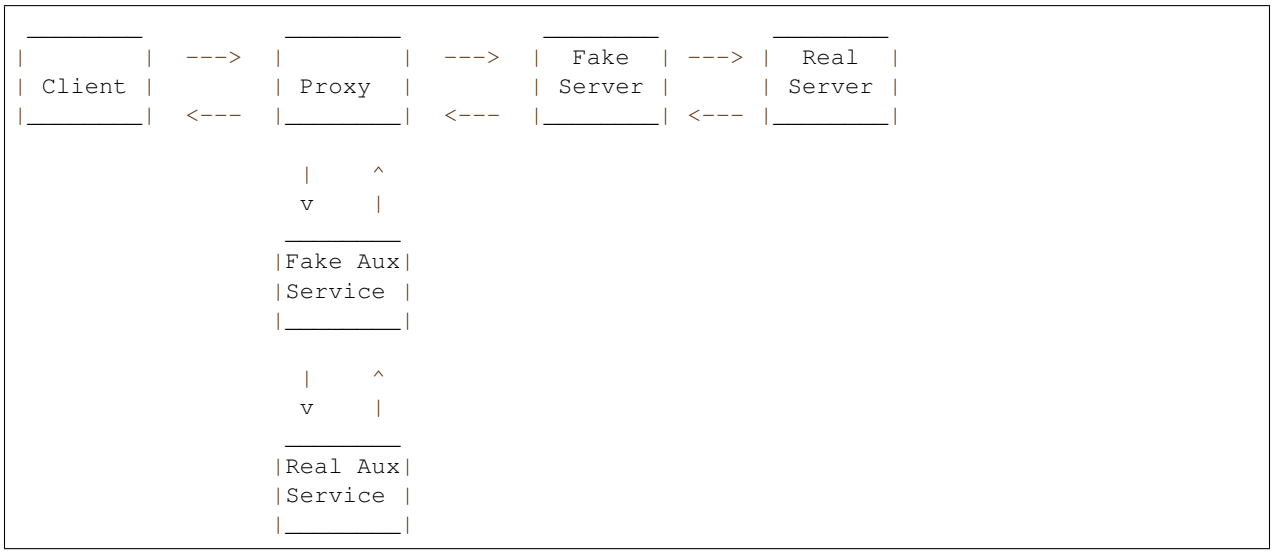
This might be the case, for example, if it is acting as a load balancer, or providing access to different versions of a ReST api based on uri. This is simple enough to simulate by creating multiple endpoint objects, and configuring the proxy to forward client requests to them.

```
Deproxy deproxy = new Deproxy()

def endpoint1 = deproxy.addEndpoint(9999)
def endpoint2 = deproxy.addEndpoint(9998)
def endpoint3 = deproxy.addEndpoint(9997)

def theProxy = new TheProxy(port: 8080,
                            targets: [
                                ['hostname': "localhost", port: 9999],
                                ['hostname': "localhost", port: 9998],
                                ['hostname': "localhost", port: 9997]],
                            loadBalanceBehavior: Behavior.RoundRobin)

def mc = deproxy.makeRequest(url: "http://localhost:8080/path/to/resource")

assert mc.receivedResponse.code == "200"
assert mc.handlings.size() == 1
assert mc.handlings[0].endpoint == endpoint1

def mc = deproxy.makeRequest(url: "http://localhost:8080/path/to/resource")

assert mc.receivedResponse.code == "200"
assert mc.handlings.size() == 1
assert mc.handlings[0].endpoint == endpoint2

def mc = deproxy.makeRequest(url: "http://localhost:8080/path/to/resource")

assert mc.receivedResponse.code == "200"
```

```
assert mc.handlings.size() == 1
assert mc.handlings[0].endpoint == endpoint3
```

Routing

Even more complex situations can be created using the Route built-in handler, to route requests to existing servers.



This will work for requests from the proxy to an auxiliary service, or from the client/proxy to the server. It can save us the trouble of implementing our own handlers to simulate the server or auxiliary service. But why this round-about way of doing it? Why not just configure the proxy to send requests to those locations directly? The real advantage to this method is that the requests go through an endpoint, so the Request and Response get captured and attached to a MessageChain. When makeRequest returns, we can make assertions against those requests and responses, which would be entirely invisible to us if the proxy had sent them directly.

```
Deproxy deproxy = new Deproxy()

def endpoint = deproxy.addEndpoint(9999,
        defaultHandler: Handlers.Route("real.server.example.com"))

def authService = deproxy.addEndpoint(7777,
        defaultHandler: Handlers.Route("real.auth.service.example.com"))

def theProxy = new TheProxy(port: 8080,
                            targetHostname: "localhost",
                            targetPort: 9999,
                            authServiceHostname: "localhost",
                            authServicePort: 7777)
```

Default Response Headers

By default, an endpoint will add a number of headers on all out-bound responses. This behavior can be turned off in custom handlers by setting the HandlerContext's sendDefaultResponseHeaders field to false (it is true by default). This can be useful for testing how a proxy responds to a misbehaving origin server. Each of the following headers is

added if it has not already been explicitly added by the handler, and subject to certain conditions (e.g., presence of a response body):

- **Server** The identifying information of the server software, "deproxy" followed by the version number.
- **Date** The date and time at which the response was returned by the handler, in RFC 1123 format.
- **Content-Type** If the response contains a body, then the endpoint will try to guess. If the body is of type `String`, then it will add a Content-Type header with a value of `text/plain`, If the body is of type `byte[]`, it will use a value of `application/octet-stream`. If the response does not contain a body, then this header will not be added.
- **Transfer-Encoding** If the response has a body, and the usedChunkedTransferEncoding field is true, this header will have a value of `chunked`. If it has a body but usedChunkedTransferEncoding is false, the header will have a value of `identity`. If there is no body, then this header will not be added.
- **Content-Length** If the response has a body, and the usedChunkedTransferEncoding field is false, then this header will have a value equal to the decimal count of octets in the body. If the body is a `String`, then the length is the number of bytes after encoding as ASCII. If the body is of type `byte[]`, then the length is just the number of bytes in the array. If the response has a body, but usedChunkedTransferEncoding is true, then this field is not added. If the response does not have a body, then this header will be added with a value of `0`.
- **Deproxy-Request-ID** If the response is associated with a message chain, then the ID of that message chain is assigned to this header and added to the response.

Note: If the response has a body, and sendDefaultResponseHeaders is set to false, and the handler doesn't explicitly set the Transfer-Encoding header or the Content-Length header, then the client/proxy may not be able to correctly read the response body.

CHAPTER 5

Handlers

Handlers are the things that turn requests into responses. A given call to makeRequest can take a `handler` argument that will be called for each request that reaches an endpoint. Deproxy includes a number of built-in handlers for some of the most common use cases. Also, you can define your own handlers.

```
def deproxy = new Deproxy()
def e = deproxy.addEndpoint(9999)
def mc = deproxy.makeRequest('http://localhost:9999/')
println mc.receivedResponse.headers
// [
//  Server: deproxy 0.16-SNAPSHOT,
//  Date: Wed, 04 Sep 2013 16:20:56 GMT,
//  Content-Length: 0,
//  Deproxy-Request-ID: 60e2a2bd-a179-4b50-a8c4-8d5b73d0218a
// ]

mc = deproxy.makeRequest(url: 'http://localhost:9999/',
        defaultHandler: Handlers.&echoHandler)
println mc.receivedResponse.headers
// [
//  Deproxy-Request-ID: 6021d10a-f252-4816-9eb6-104b0aaf91f1,
//  Host: localhost,
//  Accept: */*,
//  Accept-Encoding: identity,
//  User-Agent: deproxy 0.16-SNAPSHOT,
//  Server: deproxy 0.16-SNAPSHOT,
//  Date: Wed, 04 Sep 2013 16:20:56 GMT,
//  Content-Length: 0
// ]
```

Specifying Handlers

Handlers can be specified in multiple ways, depending on your needs.

- Passing a handler as the `defaultHandler` parameter when creating a `Deproxy` object will set the handler to be used for every request serviced by any endpoint on that object. This covers every request coming in, whether it is originally initiated by some call to `makeRequest` (simply called a 'handling') or by some other client (called an 'orphaned handling' because it isn't tied to any single message chain).

```
def echoServer = new Deproxy(Handlers.&echoHandler)
println echoServer.defaultHandler
// org.codehaus.groovy.runtime.MethodClosure@1278dc4c
```

- Passing a handler as the `defaultHandler` parameter to `addEndpoint` will set the handler to be used for every request that the created endpoint receives, whether normal or orphaned.

```
def deproxy = new Deproxy()
println deproxy.defaultHandler
// null

def echoEndpoint = deproxy.addEndpoint(9998, 'echo-endpoint', 'localhost',
        Handlers.&echoHandler)
println echoEndpoint.defaultHandler
// org.codehaus.groovy.runtime.MethodClosure@6ef2ea42
```

- Passing a handler as the `defaultHandler` parameter to `makeRequest` will set the handler used for every request associated with the message chain, no matter which endpoint receives it. This does not affect orphaned requests from non-deproxy clients, or requests that lose their `Deproxy-Request-ID` header for some reason.

```
def mc = deproxy.makeRequest(url: 'http://localhost:9998/',
        defaultHandler: Handlers.&simpleHandler)
```

- Passing a `dict` or other mapping object as the `handlers` parameter to `makeRequest` will specify specific handlers to be used for specific endpoints for all requests received associated with the message chain. This does not affect orphaned requests. The mapping object must have endpoint objects (or their names) as keys, and the handlers as values.

```
def deproxy = new Deproxy()
def endpoint1 = deproxy.addEndpoint(9997, 'endpoint-1')
def endpoint2 = deproxy.addEndpoint(9996, 'endpoint-2')
def endpoint3 = deproxy.addEndpoint(9995, 'endpoint-3')
def mc = deproxy.makeRequest(url: 'http://localhost:9997/',
        handlers: [
                endpoint1: customHandler1,
                endpoint2: customHandler2,
                'endpoint-3': customHandler3
        ])
```

Handler Resolution Procedure

Given the various ways to specify handlers, and the different needs for each, there must be one way to unambiguously determine which handler to use for any given request. When an endpoint receives and services a request, the process by which a handler is chosen for it is defined so:

1. If the incoming request is tied to a particular message chain by the presence of a `Deproxy-Request-ID` header, and the call to `makeRequest` includes a `handlers` parameters,

   (a) if that `handlers` mapping object has the current servicing endpoint as a key, use the associated value as the handler.

(b) if the mapping object doesn't have the current servicing endpoint as a key, but does have the endpoint's *name* as a key, then use the associated value of the name as the handler.

(c) otherwise, continue below

2. If the call to `makeRequest` didn't have a `handlers` argument or if the servicing endpoint was not found therein, but the call to `makeRequest` *did* include a `defaultHandler` argument, use that as the handler.
3. If the incoming request cannot be tied to a particular message chain, but the servicing endpoint's `defaultHandler` attribute is not *None*, then use the value of that attribute as the handler.
4. If the servicing endpoint's `defaultHandler` is None, but the parent `Deproxy` object's `defaultHandler` attribute is not *None*, then use that as the handler.
5. Otherwise, use `simpleHandler` as a last resort.

Built-in Handlers

The following handlers are built into deproxy. They can be used to address a number of common use cases. They also demonstrate effective ways to define additional handlers.

- **simpleHandler** The last-resort handler used if none is specified. It returns a response with a 200 status code, an empty response body, and only the basic Date, Server, and request id headers.

```
mc = deproxy.makeRequest(url: 'http://localhost:9994/',
        defaultHandler: Handlers.&simpleHandler)
println mc.receivedResponse.headers
// [
//  Server: deproxy 0.16-SNAPSHOT,
//  Date: Wed, 04 Sep 2013 16:45:44 GMT,
//  Content-Length: 0,
//  Deproxy-Request-ID: 398bbcf7-d342-4457-8e8e-0b7e8f8ca826
// ]
```

- **echoHandler** Returns a response with a 200 status code, and copies the request body and request headers.:

```
mc = deproxy.makeRequest(url: 'http://localhost:9994/',
        defaultHandler: Handlers.&echoHandler)
println mc.receivedResponse.headers
// [
//  Deproxy-Request-ID: 5f488584-fbe2-4322-bab2-8e9c157e84be,
//  Host: localhost,
//  Accept: */*,
//  Accept-Encoding: identity,
//  User-Agent: deproxy 0.16-SNAPSHOT,
//  Server: deproxy 0.16-SNAPSHOT,
//  Date: Wed, 04 Sep 2013 16:45:44 GMT,
//  Content-Length: 0
// ]
```

- **Delay(timeout, nextHandler)** This is actually a factory function that returns a handler. Give it a timeout in milliseconds and a second handler function, and it will return a handler that will wait the desired amount of time before calling the second handler.

```
mc = deproxy.makeRequest(url: 'http://localhost:9994/',
        defaultHandler: Handlers.Delay(3000))
println mc.receivedResponse.headers
// [
```

```
//  Server: deproxy 0.16-SNAPSHOT,
//  Date: Wed, 04 Sep 2013 16:45:47 GMT,
//  Content-Length: 0,
//  Deproxy-Request-ID: cb92db72-fb53-46c6-b143-d884af5f536d
// ]

mc = deproxy.makeRequest(url: 'http://localhost:9994/',
        defaultHandler: Handlers.Delay(3000, Handlers.&echoHandler))
println mc.receivedResponse.headers
// [
//  Deproxy-Request-ID: 31eb3d8a-9eba-4fdc-80a5-03101b10aec5,
//  Host: localhost,
//  Accept: */*,
//  Accept-Encoding: identity,
//  User-Agent: deproxy 0.16-SNAPSHOT,
//  Server: deproxy 0.16-SNAPSHOT,
//  Date: Wed, 04 Sep 2013 16:45:50 GMT,
//  Content-Length: 0
// ]
```

- **Route(scheme, host, deproxy)** This is actually a factory function that returns a handler. The handler forwards all requests to the specified host on the specified port. The only modification it makes to the outgoing request is to change the Host header to the host and port that it's routing to. You can also tell it to use HTTPS [*not yet implemented*], and specify a custom client connector. The response returned from the handler is the response returned from the specified host.

```
mc = deproxy.makeRequest(url: 'http://localhost:9994/ip',
        defaultHandler: Handlers.Route("httpbin.org", 80))
println mc.receivedResponse.headers
// [
//  Date: Thu, 12 Sep 2013 18:19:25 GMT,
//  Server: gunicorn/0.17.4,
//  X-Cache: MISS from [ ... ],
//  Connection: Keep-Alive,
//  Content-Type: application/json,
//  Content-Length: 45,
//  Access-Control-Allow-Origin: *,
//  Deproxy-Request-ID: 6c5b0741-87dc-456b-ae2f-87201efcf6e3
// ]
```

Custom Handlers

You can define your own handlers and pass them as the `handler` parameter to makeRequest. Any method or closure that accepts a request parameter and returns a `Response` object will do. Methods can be instance or static. Closures can be stored or inline.

```
def customHandler(request) {
    return new Response(606, 'Spoiler', null, 'Snape Kills Dumbledore')
}

// ...


def mc = deproxy.makeRequest(url: "http://localhost:9999",
        defaultHandler: this.&customHandler)
```

```
println mc.receivedResponse
// Response(
//  code=606,
//  message=Spoiler,
//  headers=[
//      Server: deproxy 0.16-SNAPSHOT,
//      Date: Wed, 04 Sep 2013 17:00:19 GMT,
//      Content-Length: 22,
//      Content-Type: text/plain,
//      Deproxy-Request-ID: fe2f9d2d-ec03-4b7e-b0b2-19f35c5b6df8],
//  body=Snape Kills Dumbledore
// )


mc = deproxy.makeRequest(url: "http://localhost:9999",
        defaultHandler: { request ->
            return new Response(
                    607,
                    "Something Else",
                    ['Custom-Header': 'Value'],
                    "Some other body")
        })
println mc.receivedResponse
// Response(
//  code=607,
//  message=Something Else,
//  headers=[
//      Custom-Header: Value,
//      Server: deproxy 0.16-SNAPSHOT,
//      Date: Wed, 04 Sep 2013 17:00:19 GMT,
//      Content-Length: 15,
//      Content-Type: text/plain,
//      Deproxy-Request-ID: 8d46b115-d7ec-4505-b5ba-dc61c60a0518],
//  body=Some other body
// )
```

Handler Context

If you define a handler with two parameters, then second will be given a HandlerContext object, which has fields used for giving directives back to the endpoint about how the Response should be sent. For example, you could set the sendDefaultResponseHeaders field to false, to tell the endpoint not to add default response headers to the response.

```
def customHandler = { request, context ->

    context.sendDefaultResponseHeaders = false

    return new Response(503, "Something went wrong", null,
            "Something went wrong in the server\n" +
                    "and it didn't return correct headers!'")
}
def mc = deproxy.makeRequest(url: 'http://localhost:9999/',
        defaultHandler: customHandler)
println mc.receivedResponse
// Response(
//  code=503,
```

```
//  message=Something went wrong,
//  headers=[
//      Deproxy-Request-ID: f3ee8e35-66c1-4b7f-a0be-1b64e94615e6],
//  body=
// )
```

Additionally, you can set the usedChunkedTransferEncoding field to true, to tell the endpoint to use chunked transfer coding to send the body to the recipient in chunks.

Default Response Headers

By default, an endpoint will add a number of headers on all out-bound responses. This behavior can be turned off in custom handlers by setting the HandlerContext's sendDefaultResponseHeaders field to false (it is true by default). This can be useful for testing how a proxy responds to a misbehaving origin server. Each of the following headers is added if it has not already been explicitly added by the handler, and subject to certain conditions (e.g., presence of a response body):

- **Server** The identifying information of the server software, "deproxy" followed by the version number.
- **Date** The date and time at which the response was returned by the handler, in RFC 1123 format.
- **Content-Type** If the response contains a body, then the endpoint will try to guess. If the body is of type `String`, then it will add a Content-Type header with a value of `text/plain`, If the body is of type `byte[]`, it will use a value of `application/octet-stream`. If the response does not contain a body, then this header will not be added.
- **Transfer-Encoding** If the response has a body, and the usedChunkedTransferEncoding field is true, this header will have a value of `chunked`. If it has a body but usedChunkedTransferEncoding is false, the header will have a value of `identity`. If there is no body, then this header will not be added.
- **Content-Length** If the response has a body, and the usedChunkedTransferEncoding field is false, then this header will have a value equal to the decimal count of octets in the body. If the body is a `String`, then the length is the number of bytes after encoding as ASCII. If the body is of type `byte[]`, then the length is just the number of bytes in the array. If the response has a body, but usedChunkedTransferEncoding is true, then this field is not added. If the response does not have a body, then this header will be added with a value of `0`.
- **Deproxy-Request-ID** If the response is associated with a message chain, then the ID of that message chain is assigned to this header and added to the response.

Note: If the response has a body, and sendDefaultResponseHeaders is set to false, and the handler doesn't explicitly set the Transfer-Encoding header or the Content-Length header, then the client/proxy may not be able to correctly read the response body.

CHAPTER 6

Client Connectors

Deproxy uses *client connectors* to provide fine-grained control over how a client will send a Request object to a destination and receive a Response object. makeRequest already gives you the ability to craft an exact HTTP request. Connectors go the next step and specify how sockets are created, and transfer bytes to/from the socket. By default, makeRequest will use a DefaultClientConnector, which will simply open a socket, write the request to the socket, and read a response from the socket. Also, it can optionally add some default headers to the request before sending.

Built-in Connectors

Deproxy provides the following built-in client connectors:

- BareClientConnector - This connector opens a socket, sends the request, and the reads the response. It doesn't modify the request or response in any way. It won't even add a Host header. It also doesn't employ any clever tricks, like following 300-level redirection responses. If there is any failure to connect or error while transmitting, BareClientConnector will throw an exception. However, simple error codes like 501 and 404 will *not* trigger an exception.
- DefaultClientConnector - This connector inherits from BareClientConnector. If the `sendDefaultRequestHeaders` field in RequestParams is set to `true`, then it will add some default headers to the request before calling BareClientConnector.sendRequest.

Both connector classes have an optional `socket` parameter that allows to use a previously-established connection when creating the connector. If you specify it, the connectors will use this socket; otherwise, they will create a new socket for every request. Specifying a specific socket to use can be useful for re-using connections. The built-in connectors don't provide any connection pooling or re-use by themselves.

Specifying Connectors

A connector can be specified for use for a particular request by passing it as the `clientConnector` parameter to makeRequest, like so:

```
def deproxy = new Deproxy()

def mc = deproxy.makeRequest(url: "http://example.org",
            clientConnector: new BareClientConnector())

assert mc.handlings.size() == 0
assert mc.receivedResponse.code == "400"    // Bad Request, due to missing Host header

def mc = deproxy.makeRequest(url: "http://example.org",
            headers: ['Host': 'example.org'],
            clientConnector: new BareClientConnector())

assert mc.handlings.size() == 0
assert mc.receivedResponse.code == "200"
```

Of course, you don't have to use a new connector each time. You can store a connector to a variable and use it for multiple requests.

Custom Connectors

You can create a custom client connector by implementing the ClientConnector interface.

Suppose you want to test a proxy for more than just its handling of certain request information. For example, how does it handle connection interruptions?

1. The client sends a request to the proxy
2. The proxy potentially modifies the request and sends it along to the server
3. Before the server can return a response, the client closes the connection to the proxy

What will the proxy do in this case? Throw an exception and log an error? Hang and catch fire? In order to test how the proxy will behave in this situation, we can create a custom client connector that closes the socket before receiving a response. We can couple that with a handler on the endpoint side that delays for a few seconds.

Here's some example code for the connector:

```
class DisconnectConnector implements ClientConnector {

    CountDownLatch latch = new CountDownLatch(1)

    @Override
    Response sendRequest(Request request, boolean https, host, port,
                         RequestParams params) {

        """Send the given request to the host,
            then wait for a few seconds and cut the connection."""

        def hostIP = InetAddress.getByName(host)


        // open the connection
```

```
        // (ignore https for now)
        Socket s = new Socket(host, port)

        def outStream = s.getOutputStream();
        def writer = new PrintWriter(outStream, true);

        // send the request
        def requestLine = String.format("%s %s HTTP/1.1",
                                        request.method, request.path ?: "/")
        writer.write(requestLine);
        writer.write("\r\n");

        writer.flush();

        HeaderWriter.writeHeaders(outStream, request.headers)

        writer.flush();
        outStream.flush();

        BodyWriter.writeBody(request.body, outStream,
                params.usedChunkedTransferEncoding)


        // wait for the handler to signal
        latch.await()

        // prematurely close the connection
        s.close()

        // wait long enough for the endpoint to
        // attach the server-side response
        sleep 3000

        return null
    }

    def handler(request) {

        sleep 2000

        // tell the connector to proceed
        latch.countDown()

        sleep 2000

        return new Response(200)
    }
}
```

And here's the test that uses it:

```
def deproxy = new Deproxy()
def endpoint = deproxy.addEndpoint(9999)

def theProxy = new TheProxy(port: 8080,
        targetHostname: "localhost",
        targetPort: 9999)
```

```
def connector = new DisconnectConnector()

def mc = deproxy.makeRequest(url: "http://localhost:8080/",
        headers: ['Host': 'localhost:8080'],
        clientConnector: connector,
        defaultHandler: connector.&handler)

assert mc.handlings.size() == 1
assert mc.handlings[0].response.code == "200"
assert mc.receivedResponse == null
```

So what should happen is that the server returns a response to the proxy, but that response never makes it back to the client. Therefore, there's a handling in the MessageChain, but `receivedResponse` is null.

Default Request Headers

By default, the DefaultClientConector will add a number of headers on all out-bound requests. This behavior can be turned off by setting the `addDefaultHeaders` parameter to makeRequest to false (it is true by default). This can be useful for testing how a proxy responds to a misbehaving client. Each of the following headers is added if it has not already been explicitly added by the caller, and subject to certain conditions (e.g., presence of a response body):

- **Host** This value is taken from the `hostname` parameter passed to the connector. If the port given is not the default for the specified uri scheme (80 for http, 443 for https), then the port number will be appended to the hostname. E.g., `localhost:9999`, `example.com`
- **User-Agent** The identifying information of the client software, "deproxy" followed by the version number.
- **Accept** If not already present, this header is added with a value of `*/*`.
- **Accept-Encoding** If not already present, this header is added with a value of `identity`.
- **Content-Type** If the request contains a body, then the connector will try to guess. If the body is of type `String`, then it will add a Content-Type header with a value of `text/plain`. If the body is of type `byte[]`, it will use a value of `application/octet-stream`. If the request does not contain a body, then this header will not be added.
- **Transfer-Encoding** If the request has a body, and usedChunkedTransferEncoding is true, this header will have a value of `chunked`. If it has a body but usedChunkedTransferEncoding is false, the header will have a value of `identity`. If there is no body, then this header will not be added.
- **Content-Length** If the request has a body, and the usedChunkedTransferEncoding is false, then this header will have a value equal to the decimal count of octets in the body. If the body is a `String`, then the length is the number of bytes after encoding as ASCII. If the body is of type `byte[]`, then the length is just the number of bytes in the array. If the request has a body, but usedChunkedTransferEncoding is true, then this field is not added. If the request does not have a body, then this header will be added with a value of `0`.

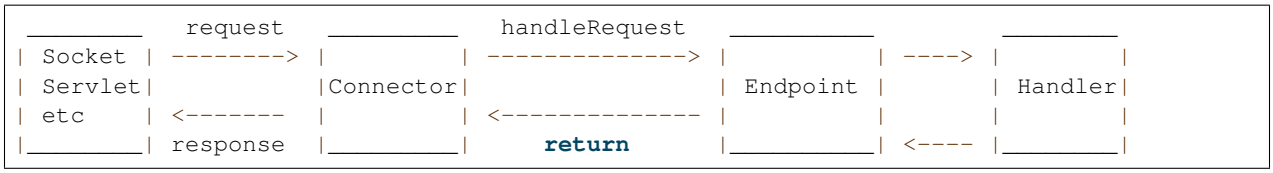
Note: If the request has a body, and sendDefaultRequestHeaders is set to false, and the handler doesn't explicitly set the Transfer-Encoding header or the Content-Length header, then the client/proxy may not be able to correctly read the request body.

Note: If the request does not have a Host header, rfc-compliant servers and proxies will reject it with a 400 response.

CHAPTER 7

Server Connectors

Deproxy uses *server connectors* to provide fine-grained control over how an endpoint receives a Request object and returns a Response object. Connectors can specify how sockets are created, and bytes are transferred to/from that socket. By default, an endpoint will will use a SocketServerConnector, which will serve requests over a socket.



Built-in Connectors

Deproxy provides the following built-in server connectors:

- SocketServerConnector - This connector acts as a stand-alone HTTP server. It creates a socket on the given port, starts a thread that listens on that socket for incoming connections, and spawns a new thread to handle each new incoming TCP connection. Requests are read off the wire and converted from octet sequences into Request objects. Likewise, Response objects are converted to octet sequences and sent back to the source of the request. Each spawned thread can handle multiple HTTP requests in succession. However, the requests aren't pipelined. That is, each request is handled and a response to it sent before the next request is read. This connector does not yet support HTTPS.
- ServletServerConnector - This connector extends javax.servlet.http.HttpServlet and can therefore be embedded into a servlet container, such as Tomcat. It relies on the container to handle the management of sockets, threading, and translation of request and response from/to octet sequences.

Specifying Connectors

The Endpoint constructor accepts a `connectorFactory` parameter. This can be any method or closure that accepts an Endpoint as a parameter and returns an object that implements the ServerConnector inter-

face. If no `connectorFactory` is specified, the Endpoint will default to SocketServerConnector. If a `connectoryFactory` is specified, then that factory will be used to get the connector during Endpoint construction. Additionally, if an argument is passed to `connectorFactory`, then the `port` parameter of both addEndpoint and the Endpoint constructor will be ignored; the `port` argument is only passed to SocketserverConnector constructor.

```
def deproxy = new Deproxy()

def servletEndpoint = deproxy.addEndpoint(
        connectorFactory: ServletServerConnector.&Factory);

...

Tomcat.addServlet(rootCtx, "deproxy-servlet",
                  servletEndpoint.serverConnector as Servlet);
```

Custom Connectors

You can create a custom server connector by implementing the ServerConnector interface.

The ServerConnector interface only has a single `shutdown` method. The Endpoint is passive and relies on the connector to initiate the handling process. It is the responsibility of a custom connector to:

1. retrieve a Request object from some source,
2. pass the request to the Endpoint via the handleRequest method,
3. receive the Response back from handleRequest, and
4. send the response back to the origin of the request.

Suppose you want to test a proxy for more than just its handling of certain request information. For example, how does it handle connection interruptions?

1. The client sends a request to the proxy
2. The proxy forwards the request to the server
3. While the server is returning the response, it hangs. Not all of the octets of the response are sent back to the proxy.

What will the proxy do in this case? Throw an exception and log an error? Hang and catch fire? In order to test how the proxy will behave in this situation, we can create a custom server connector that sleeps for an arbitrarily long time while sending data.

Here's some example code for the connector:

```
class SlowResponseServerConnector extends SocketServerConnector {

    public SlowResponseServerConnector(Endpoint endpoint, int port) {
        super(endpoint, port)
    }

    @Override
    void sendResponse(OutputStream outStream, Response response,
```

```
                        HandlerContext context=null) {

        def writer = new PrintWriter(outStream, true);

        if (response.message == null) {
            response.message = ""
        }

        writer.write("HTTP/1.1 ${response.code} ${response.message}")
        writer.write("\r\n")

        writer.flush()

        // sleep for a really long time. don't return headers
        Thread.sleep(Long.MAX_VALUE)
    }
}
```

And here's the test that uses it:

```
def deproxy = new Deproxy()
def endpoint = deproxy.addEndpoint(
        connectorFactory: { e ->
            new SlowResponseServerConnector(e, 9999)
        })

def theProxy = new TheProxy(port: 8080,
        targetHostname: "localhost",
        targetPort: 9999)

def mc = deproxy.makeRequest(url: "http://localhost:8080/")

assert mc != null
assert mc.handlings.size() == 1
assert mc.handlings[0].response.code == "200"
assert mc.receivedResponse.code == "502"
```

The server stops sending data halfway through sending the response. The handler in effect is simpleHandler, so the response generated should be a 200. However, because the full response never makes it back to the proxy, the proxy should eventually timeout and return a 502 Bad Gateway response to the client.

ServerConnector Lifecycle

When a Deproxy is shutdown, all of it's Endpoints are shutdown as well.

- **SocketServerConnector - The default connector.**
    1. When the connector is created, it opens a socket on the designated port and spawns a thread to listen for connections to that socket.
    2. Whenever a new connection is made, the listener thread will spawn a new handler thread.
    3. **The handler thread will proceed to service HTTP request, like so:**
        (a) First, the incoming request is read from the socket, and parsed into a Request object.
        (b) Next, the connector will pass the Request to the endpoint by calling the handleRequest method.
        (c) **The endpoint will:**

i. Examine the request headers for a `Deproxy-Request-ID` header, and then try to match it to an existing MessageChain (created before in a call to makeRequest).

ii. Determine which handler to use (see *Handler Resolution Procedure*), and pass the Request object to the handler to get a Response object.

iii. If there is a MessageChain associated with the request, a Handling will be created and attached to the message chain. Otherwise, it will be attached to the orphanedHandlings list of all active message chains.

iv. Return the response back to the connector.

(d) The connector then sends the response back to the sender.

(e) Finally, if the Request or handler indicated that the connection should be closed (by setting the `Connection` header to `close`), then the handler thread will exit the loop and close the connection. Otherwise, it will return to step `a.` above.

4. When shutdown is called on a parent Endpoint object, the connector will be shutdown. Its listener thread will stop listening, and no longer receive any new connections. Any long-running handler threads will continue to run until finished or the JVM terminates, whichever comes first.

- ServletServerConnector - This connector expects to be loaded into a servlet container. Therefore, it neither creates threads nor opens sockets, and its shutdown method does nothing.

CHAPTER 8

Standard Headers

Experimental/In-progress

Deproxy provides a number of classes that correspond to the standard headers defined in RFC 2616. These classes provide a useful object model to construct headers that conform to the rfc, preventing accidental mistakes like leaving an extra space or period in a Host header.

- HostHeader - Defines a Host header. Takes a string for the hostname and an optional integer port.

CHAPTER 9

Search

- search