
Node Replay

When API testing slows you down: record and replay HTTP responses like a boss

Things that will ruin your day when tests make HTTP requests to other services:

- That other service has the uptime of Twitter's API
- Network late ncy
- Being-rate limited and having to wait an hour for the next test run
- Same request returns different result each time
- Everyone else on the network is deep in BitTorrent territory

Things **node-replay** can do to make these problems go away:

- Record API response once, replay as often as necessary
- Stub HTTP requests (TBD)
- Replay different responses to same request (great for testing error handling)
- Not suck

How to use node-replay

Like this:

```
1 npm install replay
```

Now write some simple test case:

```
1 const assert = require('assert');
2 const HTTP   = require('http');
3 const Replay = require('replay');
4
5 HTTP.get({ hostname: 'www.iheartquotes.com', path: '/api/v1/random' },
6   function(response) {
7     var body = '';
7     response.on('data', function(chunk) {
8       response.body = response.body + chunk;
9     });
10    response.on('end', function() {
11
12      // Now check the request we made to the I <3 Quotes API
13      assert.equal(response.statusCode, 200);
14      assert.equal(response.body, 'Oxymoron 2. Exact estimate\n\n[
15        codehappy] http://iheartquotes.com/fortune/show/38021\n');
16      console.log('Woot!');
```

```
17   });  
18   });
```

This, of course, will fail the first time you run it. You'll see:

```
1  Error: Connection to http://www.iheartquotes.com:80/api/v1/random  
   refused: not recording and no network access  
2    at Array.0 (/Users/assaf/projects/node-replay/lib/replay/proxy.  
   coffee:87:21)  
3    at EventEmitter._tickCallback (node.js:192:40)
```

Unless you tell it otherwise, **node-replay** runs in **replay** mode. In this mode it will replay any previously captured HTTP response, but it will not allow any outgoing network connection.

That's the default mode for running tests. "Why?" you ask. Good question. Running in **replay** mode forces any test you run to use recorded responses, and so it will run (and fail or pass) the same way for anyone else, any other day of the week, on whatever hardware they use. Even if they're on the AT&T network.

Running in **replay** mode helps you write repeatable tests. Repeatable tests are a Good Thing.

So the first thing you want to do to get that test to pass, is to run **node-replay** in **record** mode. In this mode it will replay any recorded response, but if no response was recorded, it will make a request to the server and capture the response.

Let's do that:

```
1  REPLAY=record node test.js
```

That wasn't too hard, but the test is still failing. "How?" You must be wondering and scratching your head in total disbelief. It's actually quite simple.

Every request you make to 'I 3 Quotes' returns a different quote, and that test is looking for a very specific quote. So the test will fail, and each time fail with a different error.

So one way we can fix this test is to change the assertion. Look at the error message, get the actual quote and make the assertion look for that value.

Now run the test:

```
1  $ node test.js  
2  => Woot!
```

Did the test pass? Of course it did. Run it again. Still passing? Why, yes.

So let's have a look at that captured response. All the responses recorded for 'I <3 Quotes>' will be listed here:

```
1 ls fixtures/www.iheartquotes.com/
```

There should be only one file there, since we only recorded one response. The file name is a timestamp, but feel free to rename it to something more descriptive.

The name of a response file doesn't matter, it can be whatever you want. The name of the directory does, though, it matches the service hostname (and port when not 80).

So that was one way to fix the failing test. Another one is to change the recorded response to match the assertion. Being able to edit (and create new) responses is quite important. Sometimes it's the easiest way to create mock responses for testing, e.g. if you're trying to test failure conditions that are hard to come by.

So let's edit the response file and change the body part, so the entire response reads like this:

```
1 /api/v1/random
2
3 HTTP/1.1 200 OK
4 server: nginx/0.7.67
5 date: Fri, 02 Dec 2011 02:58:03 GMT
6 content-type: text/plain
7 connection: keep-alive
8 etag: "a7131ebc1e81e43ea9ecf36fa2fdf610"
9 x-ua-compatible: IE=Edge,chrome=1
10 x-runtime: 0.158080
11 cache-control: max-age=0, private, must-revalidate
12 content-length: 234
13 x-varnish: 2274830138
14 age: 0
15 via: 1.1 varnish
16
17 Oxymoron 2. Exact estimate
18
19 [codehappy] http://iheartquotes.com/fortune/show/38021
```

All responses are stored as text files using the simplest format ever, so you can edit them in Vim, or any of the many non-Vim text editors in existence:

- First comes the request path (including query string)
- Followed by any headers sent as part of the request (like `Accept` and `Authorization`)
- Then an empty line
- Next the response status code and (optional) HTTP version number
- Followed by any headers sent as part of the response
- Then another empty line
- And the rest taken by the response body

If you need to use regular expressions to match the request URL, add `REGEXP` between the method and path, for example:

```
1 GET REGEXP /\Aregexp\d/i
2
3 HTTP/1.1 200 OK
4 Content-Type: text/html
```

Settings

We've got them. Just enough to make you happy and not enough to take all day to explain.

The first and most obvious is the mode you run **node-replay** in:

bloody – All requests go out, none get replayed. Use this if you want to remember what life was before you started using **node-replay**. Also, to test your code against changes to 3rd party API, because these do happen. Too often.

cheat – Replays recorded responses, and allow HTTP outbound requests. This is mighty convenient when you're writing new tests or changing code to make new, un-recorded HTTP requests, but you haven't quite settled on which requests to make, so you don't want any responses recorded quite yet.

record – Replays recorded responses, or captures responses for future replay. Use this whenever you're writing new tests or code that makes new HTTP requests.

replay – Replays recorded responses, does not allow outbound requests. This is the default mode. That's another way of saying, "you'll be running in this mode most of the time".

You can set the mode by setting the environment variable `REPLAY` to one of these values:

```
1 REPLAY=record node test.js
```

Of from your code by setting `replay.mode`:

```
1 const Replay = require('replay');
2 Replay.mode = 'record';
```

Of course, **node-replay** needs to store all those captured responses somewhere, and by default it will put them in the directory `fixtures`. Bet you have an idea for a better directory name. Easy to change.

Like this:

```
1 Replay.fixtures = __dirname + '/fixtures/replay';
```

You can tell **node-replay** what hosts to treat as “localhost”. Requests to these hosts will be routed to 127.0.0.1, without capturing or replay. This is particularly useful if you’re making request to a test server and want to use the same URL as production.

For example:

```
1 Replay.localhost('www.example.com');
```

If you don’t want requests going to specific server, you can add them to the drop list. For example, Google Analytics, where you don’t care that the request go through, and you don’t want to record it.

```
1 Replay.drop('www.google-analytics.com', 'rollbar.com');
```

Likewise, you can tell **node-replay** to pass through requests to specific hosts:

```
1 Replay.passThrough('s3.amazonaws.com');
```

If you’re running into trouble, try turning debugging mode on. It helps. Sometimes.

```
1 $ DEBUG=replay node test.js
2 => Requesting http://www.iheartquotes.com:80/api/v1/random
3 => Woot!
```

By default, **node-replay** will record the following headers with each request, and use only these headers when matching pre-recorded requests:

- Headers starting with **Accept** (eg **Accept-Encoding**)
- **Authorization**
- **Body** (used to match the body of a POST request)
- **Content-Type**
- **Host**
- Headers starting with **If-** (eg **If-Modified-Since**)
- Headers starting with **X-** (eg **X-Requested-With**)

You can modify the list of matched headers, adding or removing headers, by changing the value of **Replay.headers**. The value is an array of regular expressions.

For example, to capture **content-length** (useful with file uploads):

```
1 Replay.headers.push(/^content-length/);
```

Since headers are case insensitive, we always match on the lower case name.

If you want more control over the responses that you record (for example weeding out error responses for really flaky backends or when you are polling waiting for something to happen), you can use recor-

dReponseControl to exercise fine grain control over the recording process. For example, to only save the responses from myhostname.com port 8080 if the response was successful:

```
1 Replay.recordResponseControl = {
2   "myhostname.com:8080" : function(request, response) {
3     return response.statusCode < 400;
4   }
5 };
```

Geeking

To make all that magic possible, **node-replay** replaces `require('http').request` with its own method. That method returns a `ProxyRequest` object that captures the request URL, headers and body.

When it's time to fire the request, it gets sent through a chain of proxies. The first proxy to have a response, returns it (via callback, this is Node.js after all). That terminates the chain. A proxy that doesn't have a response still has to call the callback, but with no arguments. The request will then pass to the next proxy down the chain.

The proxy chain looks something like this:

- Logger dumps the request URL when running with `DEBUG=replay`
- The pass-through proxy will pass the request directly to the server in `bloody` mode, or when talking to `localhost`
- The recorder proxy will either replay a captured request (if it has one), talk to the server and capture the response (in `record` mode), or pass to the next proxy
- The pass-through proxy (2nd one) will pass the request to the server in `cheat` mode, return nothing in all other modes

Loading pre-recorded responses to memory, from where they can be replayed, and storing new ones on disk, is handled by ... cue big band ... the `Catalog`.

Final words

node-replay is released under the MIT license. Pull requests are welcome.