FiSTB 2017 Tutorial

REST API Testing and Automating Tutorial

Alan Richardson

" a mix of presentation, demos, discussion and hands on exercises to create an immersive and fast paced interactive day of learning"

http://testingassembly.fistb.fi/tutorial

- www.eviltester.com
- twitter: `@` eviltester

Logistics

- Time: Tuesday 19.9.2017 (9:00 - 17:00, registration and morning coffee at 8:30)
- 13 sections, about 30 mins each
- Venue: Hotel Presidentti, Etel채inen Rautatiekatu 4, Helsinki (room: Kaarlo, 3rd floor).

Slides and Handouts and Links

compendiumdev.co.uk/page/atFistbTute2017

- Also on the bottom of every slide.

Description

Web based applications often offer more than a GUI to interact with, very often they have an API layer. This layer clearly needs testing and we can use it to support our other testing efforts - GUI, performance, etc. In this tutorial we will work with several applications to learn the basics of automating and testing REST APIs.

You will learn:

- Basics of REST Web Services
- What tools we can use to interact with REST APIs
- How we can use REST APIs in our testing
- How to test a REST API
- Multiple ways to automate REST Web Service APIs
- Abstraction layers for automating REST APIs

Hands on Experience

You will gain hands on experience with:

- Exploratory Testing REST APIs
- Using different tools to interact with a REST API
- Automating a REST API

This tutorial uses a mix of presentation, demos, discussion and hands on exercises to create an immersive and fast paced interactive day of learning.

Requirements

Requirements to attend workshop:

- You will need a laptop to take part
- Wifi connection
- You will need the ability to install software on to your laptop
- During the workshop we will use REST Client and an HTTP Proxy
    - REST Client Postman (https://www.getpostman.com/)
    - An HTTP Proxy: Fiddler or Charles or ZAP Proxy
    - if you want to be up and running quickly then install Postman, and one of the proxy tools prior to the workshop

Requirements

- Some sample applications are written in Java 1.8 these will only work if you have Java 1.8 (or higher) installed (run `java -version`)
- There is an optional hands on coding section where examples will be provided using Java - if you would like to extend the automated code samples then install Java JDK version 1.8 or above and a Java IDE (recommended IntelliJ) - you do not need to be able to code to take part in the workshop

SECTION - HTTP BASICS

Overview of Section - HTTP Basics

- What is a Web Application?
- What is an HTTP Request?
- Viewing/Making HTTP Requests in Browser
- Using Browser Dev Tools

Exercises: in browser - GET, viewing traffic, Ajax requests

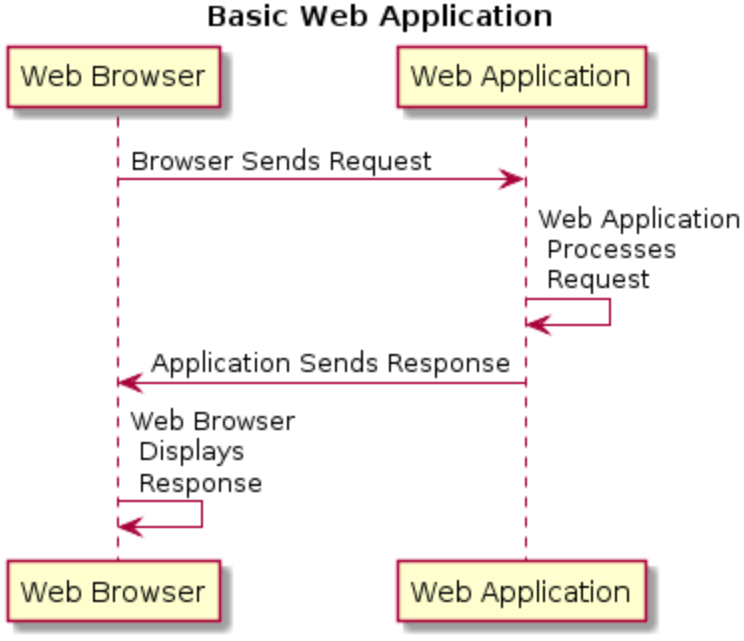
What is a Web Application?

- A web hosted HTTP accessed application with a GUI and possibly an API

Examples of Web Applications

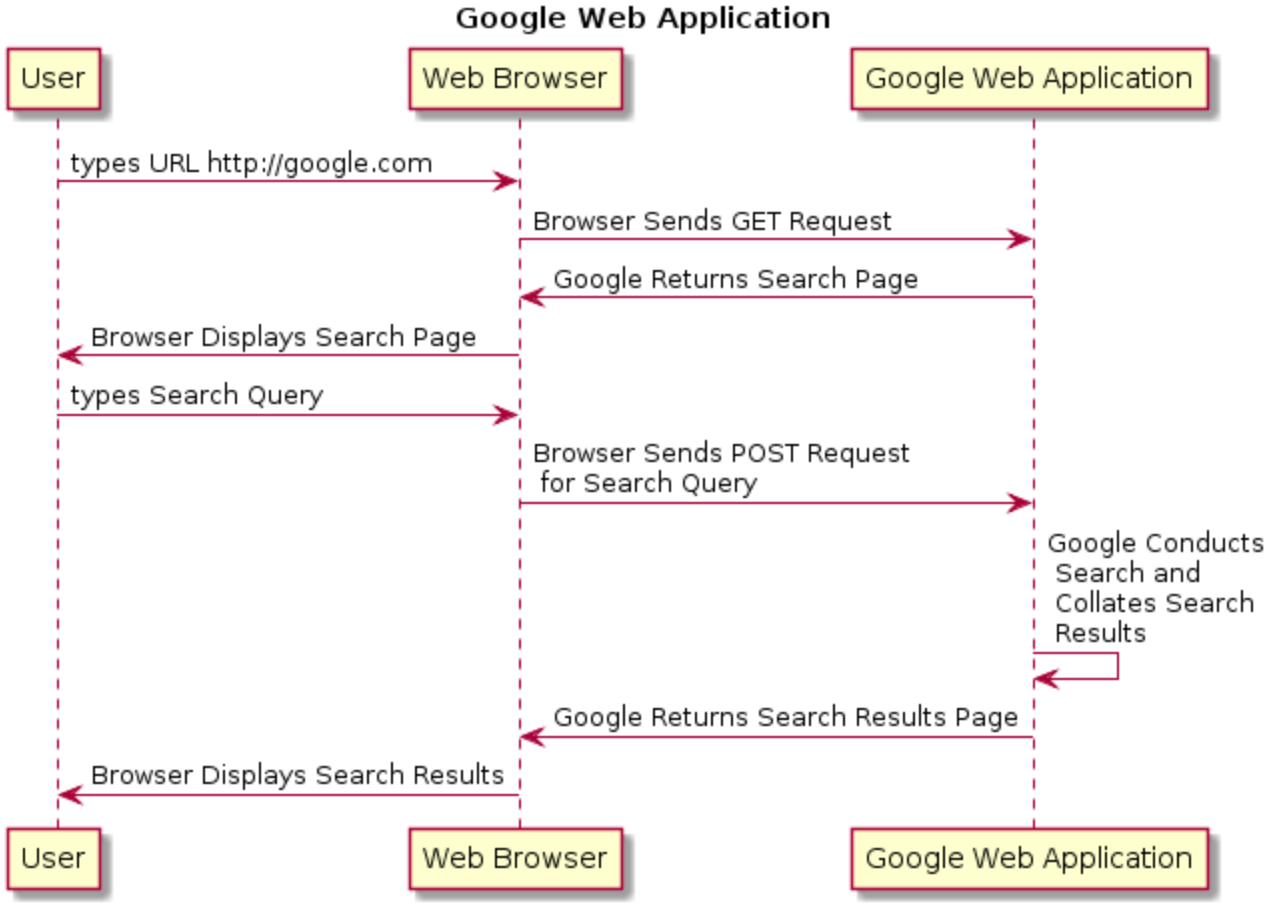
- Google
- Twitter
- etc.

Overview - Browser - Web Application



Example - A Web Application

- Diagram showing browser making GET, POST requests to Server



What is HTTP?

- Verbs - GET, POST, DELETE, PUT, HEAD, OPTIONS, PATCH
- URL (URI)
- Headers - cookies, accept formats, user agent, content of message, authentication, etc.
- Data contained in message body - Form, JSON, XML

Example HTTP Request

use browser to GET

http://compendiumdev.co.uk/apps/api/mock/reflect

Formatted for readability - headers are normally on one line.

```
GET http://compendiumdev.co.uk/apps/api/mock/reflect HTTP/1.1
Host: compendiumdev.co.uk
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
            AppleWebKit/537.36 (KHTML, like Gecko)
            Chrome/60.0.3112.90 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,
        application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
```

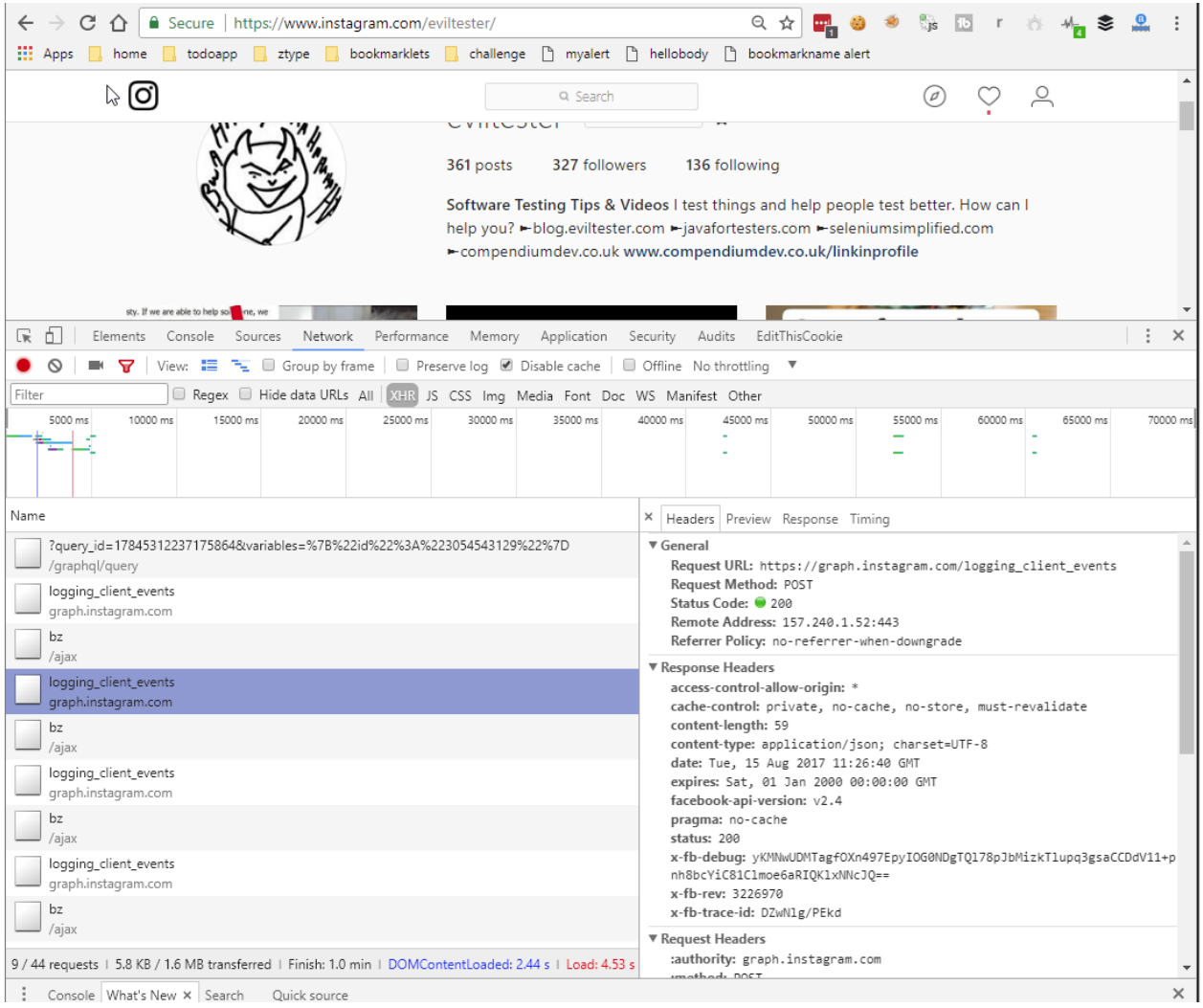
HTTP Requests - Human or System

- Human
  - user types URL into browser search bar (GET)
  - user submits form (POST)
- System

  -System automatically polls server for new content via JavaScript

  - AJAX (Asynchronous JavaScript and XML)
  - XHR (XML HTTP Request)
  - GET / POST
  - often returns JSON

View Browser Requests in Dev Tools Network Tab



Demo: View Browser Requests in Dev Tools

- instagram.com/eviltester
- Chrome
  - `Inspect Element` or `ctrl+shift+i` or `tools\more tools\Developer Tools`
  - Network Tab
- Edge
  - F12 Developer tools
  - Network Tab
- Firefox
  - Inspect Element
  - Network tab

Make sure 'All' or 'XHR' is selected.

Group Exercise: View Browser Requests in Dev Tools Network Tab Examples

- Twitter Feed?
  - e.g. https://twitter.com/eviltester/lists/work
- WebGraphViz?
  - e.g. http://www.webgraphviz.com/
- Instagram?
  - https://www.instagram.com/eviltester
- etc. probably app or web page that updates automatically

Exercise: View Browser Requests in Dev Tools Network Tab

- Open browser, Visit a site, Open Network Tab, Inspect XHR requests
- visit a twitter feed, inspect the update XHR requests
  - e.g. https://twitter.com/eviltester/lists/work
- visit an Instagram feed, inspect the update XHR requests
  - e.g. https://www.instagram.com/eviltester/
- Find other sites, inspect the update XHR requests
  - e.g. Gmail, Ebay, what else?

JavaScript AJAX/XHR Browser Risks

- Memory Leaks
- Blocking Code (JavaScript is single threaded) crashing browser
- Cross browser compatibility issues

Bonus: In Chrome try the Task Manager (shift+esc) and see if these JavaScript pages have any issues.

- Chrome Menu \ More Tools \ Task Manager
- ebay often has memory leaks, or just doesn't care about your browser

These risks are not present with an API.

What is a Web Application?

- A web hosted HTTP accessed application with a GUI and possibly an API
- Did any of the previous Web Application examples use an API?

SECTION - WEB API BASICS

Overview of Section - WEB API Basics

- What is a Web API?
- Ajax/XHR and API
- Why Web HTTP API?
- Exercises using http://swapi.co

What is a Web API?

- Web Service
  - w3.org/TR/ws-gloss
- API
  - wikipedia.org/wiki/Application_programming_interface

Web Application with an interface designed for use by other software.

Why an API?

- Other systems to access
- Customisation
- Mobile Apps often use API
    - bag a SNES classic

Exercise: A Web Application which uses API

- https://swapi.co
  - make a request from the GUI
  - Use network tab

JavaScript Using API Via AJAX/XHR

- AJAX/XHR requests have security protocols for same domain
- JSONP for cross domain access
- Very often API is used under covers, e.g. a serverside script/app on same domain uses an API on server side rather than client side

We Need Tools

Because Web Service designed for software, we need tools to access them.

Tools

- cURL
  - command line based
  - API examples often shown in cURL
  - recommended that you learn this eventually
  - download
- GUI Clients
  - Postman
  - Insomnia

Demo cURL

```
curl http://localhost:4567/heartbeat -i
curl -X GET http://localhost:4567/users
curl http://localhost:4567/lists -H "accept: application/xml"
```

Can be complicated but useful for emergencies, scripting, bug reporting.

Hint: can use Postman or Insomnia to generate cURL code but different continuation characters on different operating systems: `^` *Windows and* `\` *on Mac/Linux also* `"` *and* `'` *differences.*

Demo Postman

- Postman make GET request
- Postman console
- Postman set basic auth
- Postman add a header
- Postman Collections
- Postman Environment Variables

Demo Insomnia

- Insomnia make GET request
- Insomnia Timeline
- Insomnia set basic auth
- Insomnia add a header
- Insomnia Workspace
- Insomnia Environment Variables

Exercise: Install Tools for accessing HTTP API Web Services

Install either:

- Postman GetPostman.com
- Insomnia insomnia.rest

Exercise: Call a webservice using tools

- GET https://swapi.co/api/people/1

'MOCK' Web Services

- GET http://compendiumdev.co.uk/apps/api/mock/heartbeat
- GET http://jsonplaceholder.typicode.com/users/1/todos

see exercises section for more

What is a Web Service / Web Application?

- A web hosted HTTP accessed application without a GUI

What is an API?

- Application Programming Interface designed for use by software

Note: error messages need to be human readable

Why test interactively and not just automate?

- observe traffic
- create varied requests
- experiment fast
- setup data
- send 'invalid' requests
- exploratory testing of API
- test while API still 'flexible'
- Interactive CRUD testing - CREATE, READ, UPDATE, DELETE

SECTION - HTTP REQUESTS AND RESPONSES

Overview of Section - HTTP Requests and Responses

- HTTP Verbs - GET, POST, DELETE
- Headers
- Responses
  - Status Codes - e.g. 200, 404, 500
- This is the foundation for most web, HTTP, REST testing and automating.

HTTP Request sent from Postman

```
GET http://localhost:4567/heartbeat HTTP/1.1
cache-control: no-cache
Postman-Token: ddf30bfe-b7e2-4d3c-b478-1103a5a174e5
User-Agent: PostmanRuntime/6.2.5
Accept: */*
Host: localhost:4567
accept-encoding: gzip, deflate
Connection: keep-alive
```

- important stuff: Verb (GET), Http version (1.1), User-Agent, Accept, Host, endpoint

HTTP GET Request sent from cURL

Command:

```
curl http://localhost:4567/heartbeat ^
-H "accept: application/xml" ^
--proxy 127.0.0.1:8888
```

Request:

```
GET http://localhost:4567/heartbeat HTTP/1.1
User-Agent: curl/7.39.0
Host: localhost:4567
Connection: Keep-Alive
accept: application/xml
```

HTTP Response to Postman GET /heartbeat request

```
HTTP/1.1 200 OK
Date: Thu, 17 Aug 2017 10:34:32 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.4.v20170414)
```

- cURL response was same but content-type was `application/xml`
- important stuff: Status Code (200 OK), Http version (1.1), Date, Content-Type

Raw HTTP Requests and Responses

- we need to be able to read them
- we will rarely have to create them by hand
- lookup headers you don't know
  - https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- some fields are for the server some are for the application some are documentation

Basic HTTP Verbs

- GET - retrieve data
- POST amend/create from partial information
- PUT - create or replace from full information
- DELETE - delete items
- OPTIONS - verbs available on this url

References

- W3c Standard
- IETF standard
- httpstatuses.com
- http://www.restapitutorial.com/lessons/httpmethods.html

HTTP GET Verb

- GET - retrieve data
- GET verbs can be issued by a browser
  - click on link
  - visit a site
- GET `http://compendiumdev.co.uk/apps/api/mock/reflect`
- Important Headers
  - User-Agent - tells server app type
  - Accept - what format response you prefer

Demo

HTTP GET Verb Example

```
curl http://localhost:4567/heartbeat ^
-H "accept: application/xml" ^
--proxy 127.0.0.1:8888
```

```
GET http://localhost:4567/heartbeat HTTP/1.1
User-Agent: curl/7.39.0
Host: localhost:4567
Connection: Keep-Alive
accept: application/xml
```

User-Agent Header

- Often not sent when accessing an API
- Marks request as coming from a browser

```
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/60.0.3112.90 Safari/537.36
```

Accept Header

- Defines the payload types that the receiver will accept
- If this was an API call it would likely return XML

```
Accept: text/html,application/xhtml+xml,application/xml;
q=0.9,image/webp,image/apng,*/*;q=0.8
```

Common values:

- `text/html`
- `application/json`
- `application/xml`

HTTP Status Codes

- 1xx Informational
  - 100 Continue
- 2xx Success
  - e.g. 200 OK
- 3xx Redirection
  - e.g. 301 Moved Permanently
- 4xx Client Error
  - e.g. 404 Not Found
- 5xx Server Error
  - e.g. 500 Internal Server Error

Common HTTP Status Codes

Status Code	Status Code
200 OK	405 Method Not Allowed
201 Created	409 Conflict
301 Moved Permanently	500 Internal Server Error
307 Temporary Redirect	501 Not Implemented
400 Bad Request	502 Bad Gateway
401 Unauthorized	503 Service Unavailable
403 Forbidden	504 Gateway Timeout
404 Not Found	

HTTP Status code references

- https://httpstatuses.com/
- https://moz.com/blog/response-codes-explained-with-pictures
- https://http.cat/
- https://httpstatusdogs.com/

Common HTTP Status codes in response to a GET

- **200** - OK, found the url, returned contents
- **301, 307, 308** - content has moved, new url in `location` header
- **404** - url not found
- **401** - you need to give me authorisation details see `WWW-Authenticate` header
- **403** - url probably exists but you are not allowed to access it

Basic Auth Header

- This application uses Basic Auth Authentication
- `Authorization` Header

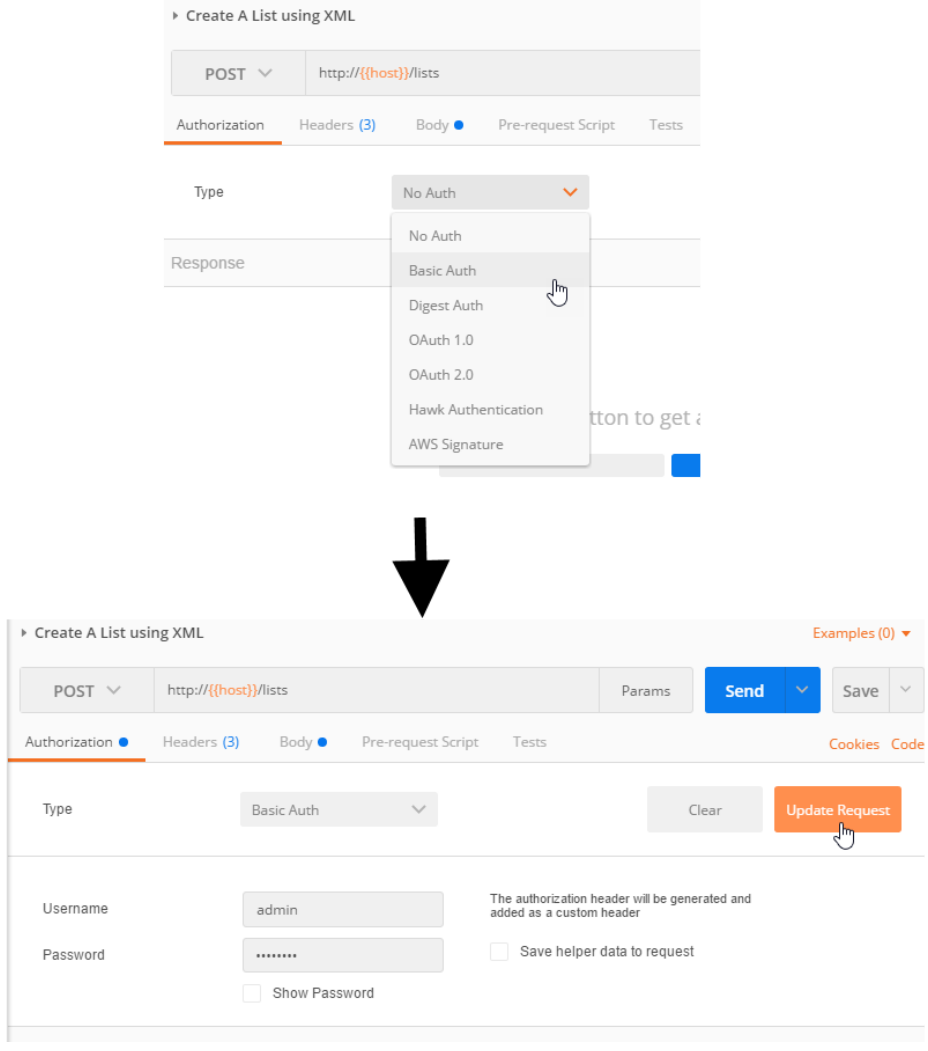
e.g. `Authorization: Basic dXNlcjpwYXNzd29yZA==`

`dXNlcjpwYXNzd29yZA==` is base64 encoded "user:password"

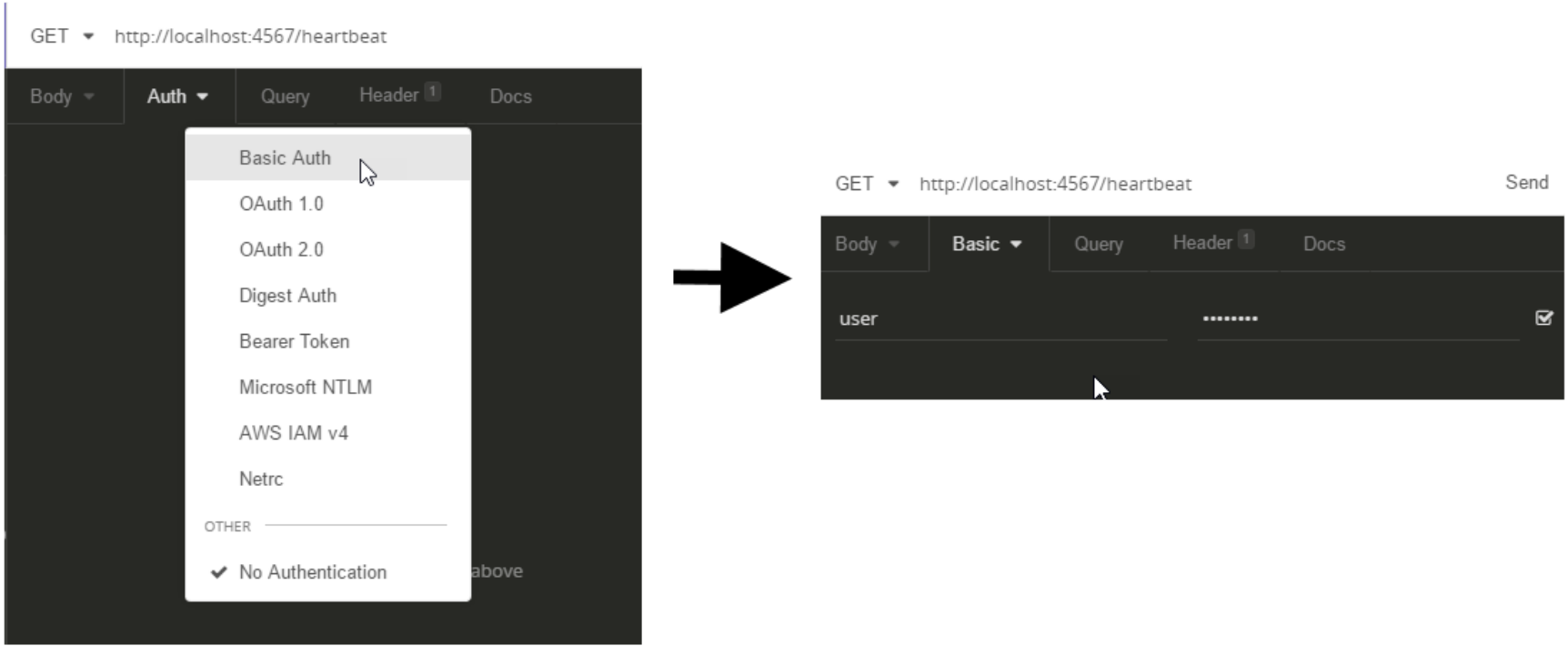
see base64decode.org

- cURL you need to add the header
- Postman & Insomnia use the Authorization and Auth tabs

Create Basic Auth Header in Postman



Create Basic Auth Header in Insomnia



HTTP POST Verb

- POST amend/create from partial information
- send a 'body' format of content in the 'content-type' header
- usually used to create or amend data
- browser will usually send a POST request when submitting a form

Demo

HTTP POST Verb Send Example

```
curl -X POST http://localhost:4567/lists ^
-H "accept: application/xml" ^
-H content-type:application/json ^
-H "Authorization: Basic dXNlcjpwYXNzd29yZA==" ^
-d "{title:'a list title'}" ^
--proxy 127.0.0.1:8888
```

HTTP POST Verb Request Example

```
POST http://localhost:4567/lists HTTP/1.1
User-Agent: curl/7.39.0
Host: localhost:4567
Connection: Keep-Alive
accept: application/json
content-type: application/json
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Length: 22

{title:'a list title'}
```

HTTP POST Verb Response Example

```
HTTP/1.1 201 Created
Date: Thu, 17 Aug 2017 12:11:12 GMT
Content-Type: application/json
Location: /lists/f8134dd6-a573-4cf5-a6c6-9d556118ed0b
Server: Jetty(9.4.4.v20170414)
Content-Length: 171

{"lists":[{
"guid":"f8134dd6-a573-4cf5-a6c6-9d556118ed0b",
"title":"a list title",
"description":"",
"createdDate":"2017-08-17-13-11-12",
"amendedDate":"2017-08-17-13-11-12"}]}
```

Common HTTP Status codes in response to a POST

- **200** - OK, did whatever I was supposed to
- **201** - OK created new items
- **202** - OK, I'll do that later
- **204** - OK, I have no more information to give you
- **400** - what? that request made no sense
- **404** - I can't post to that url it is not found
- **401** - need authorisation see `WWW-Authenticate` header
- **403** - url probably exists but you are not allowed to access it
- **409** - can't do that, already exists
- **500** - your request made me crash

HTTP Message Body Format - JSON

- JSON - JavaScript Object Notation
- an actual Object in JavaScript
- common data transfer and marshalling format for other languages
- https://en.wikipedia.org/wiki/JSON
- http://json.org
- http://countwordsfree.com/jsonviewer
- schema exists for JSON http://json-schema.org/

JSON Example Explained

```
{
 "lists":
  [
   {
    "guid":"f8134dd6-a573-4cf5-a6c6-9d556118ed0b",
    "title":"a list title",
    "description":"",
    "createdDate":"2017-08-17-13-11-12",
    "amendedDate":"2017-08-17-13-11-12"
   }
  ]
}
```

- An object, which has an array called "lists".
- the lists array contains an object with fields: `guid`, `title`, `description`, `createdDate`, `amendedDate` - all String fields.

XML Example Explained

```
<?xml version="1.0" encoding="UTF-8" ?>
<lists>
  <list>
    <guid>f8134dd6-a573-4cf5-a6c6-9d556118ed0b</guid>
    <title>a list title</title>
    <description></description>
    <createdDate>2017-08-17-13-11-12</createdDate>
    <amendedDate>2017-08-17-13-11-12</amendedDate>
  </list>
</lists>
```

- elements, nested elements
- tags, values

HTTP Message Body Format - XML

- XML - eXtended Markup Language
- HTML is often XML
- another common marshalling format
- can be validated against XML schema
- http://countwordsfree.com/xmlviewer

HTTP DELETE Verb

- DELETE - delete items

Demo

HTTP DELETE Send Example

```
curl -X DELETE http://localhost:4567/lists/{guid} ^
-H "Authorization: Basic YWRtaW46cGFzc3dvcmQ=" ^
--proxy 127.0.0.1:8888
```

HTTP DELETE Request Example

```
DELETE http://localhost:4567/lists/{guid} HTTP/1.1
User-Agent: curl/7.39.0
Host: localhost:4567
Accept: */*
Connection: Keep-Alive
Authorization: Basic YWRtaW46cGFzc3dvcmQ=
```

HTTP DELETE Response Example

```
HTTP/1.1 204 No Content
Date: Thu, 17 Aug 2017 12:20:35 GMT
Content-Type: application/json
Server: Jetty(9.4.4.v20170414)
```

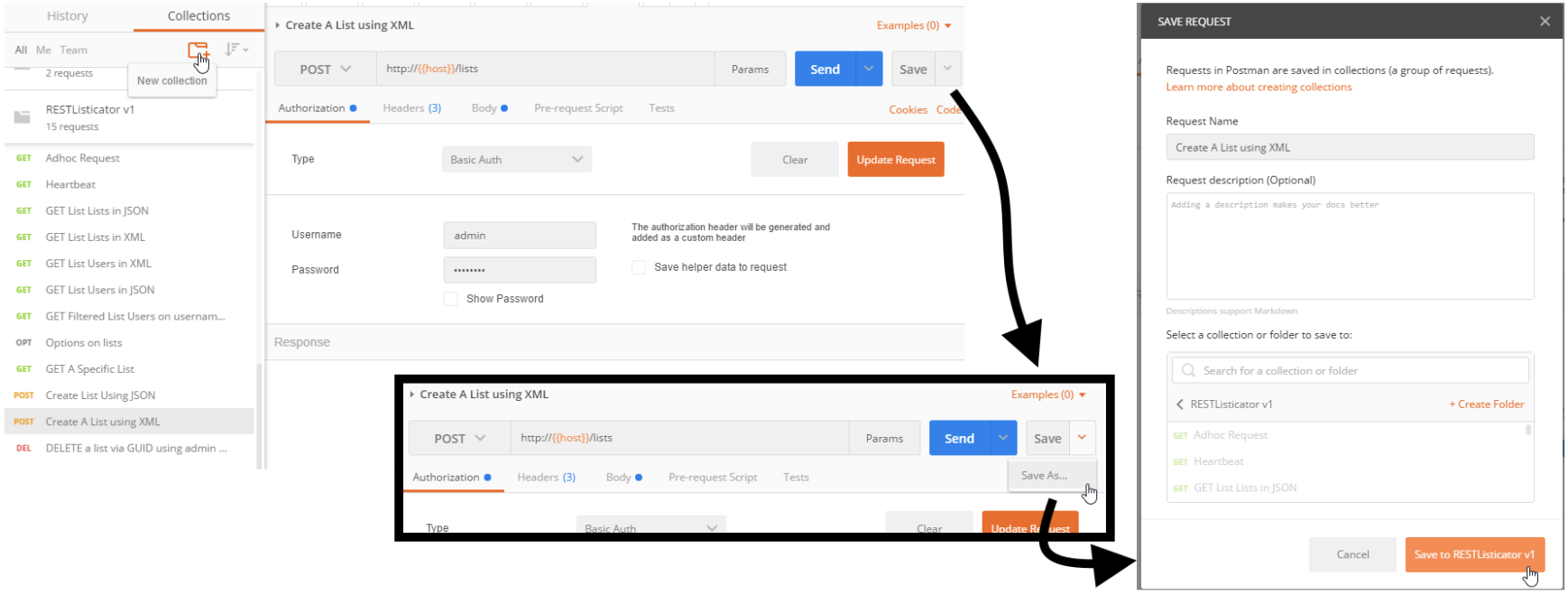
Common HTTP Status codes in response to a DELETE

- **200** - OK, did whatever I was supposed to
- **202** - OK, I'll do that later
- **204** - OK, I have no more information to give you
- **404** - I can't post to that url it is not found
- **401** - you need to give me authorisation details see `WWW-Authenticate` header
- **403** - url probably exists but you are not allowed to access it
- **500** - your request made me crash

Postman Collections

- "save as" requests to collections for re-use
- can share collections or export to file

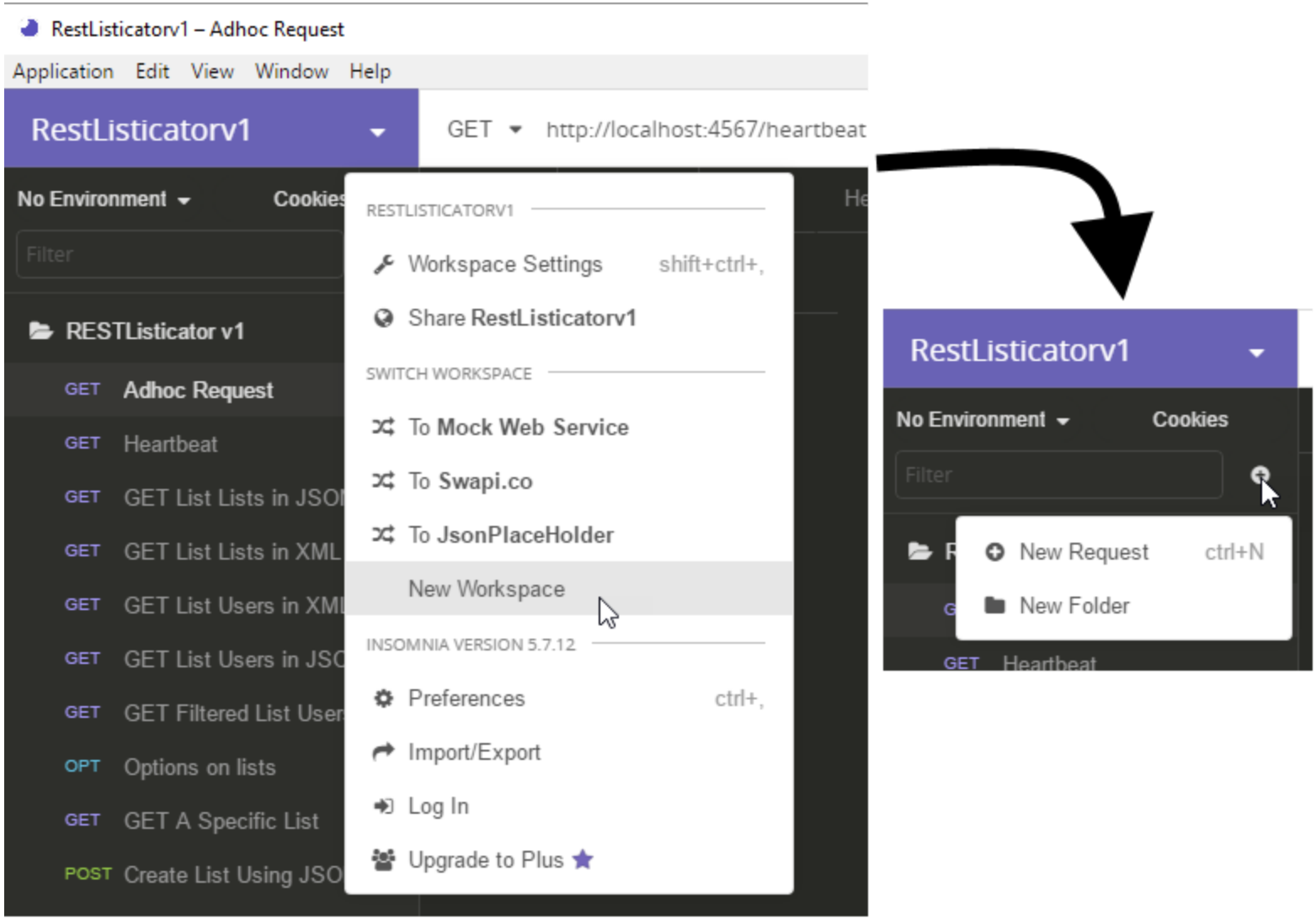
Postman Collections



Insomnia Workspaces

- create new Workspace
- create new Request in Workspace
- changes automatically saved to workspace
- can export workspace to files

Insomnia Workspaces



Environment Variables

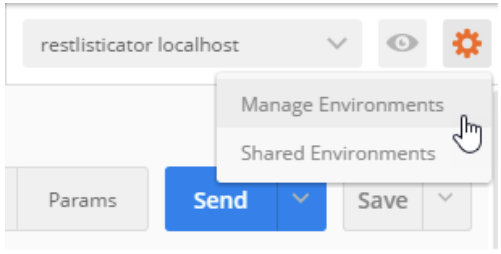
Postman:

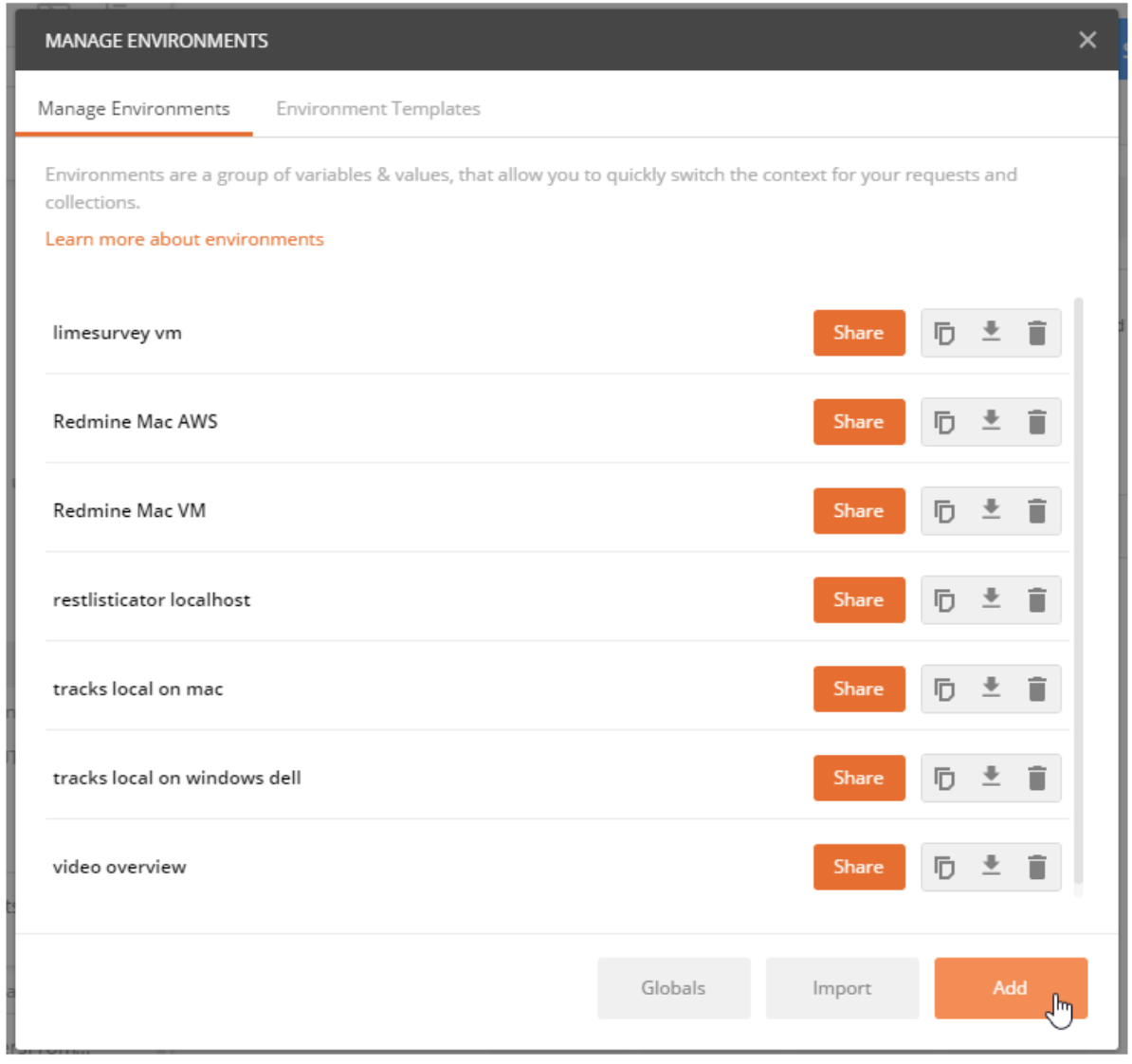
- use environment variables e.g. `{{host}}` instead of `localhost:4567`
- `GET httt://{{host}}/lists`

Insomnia:

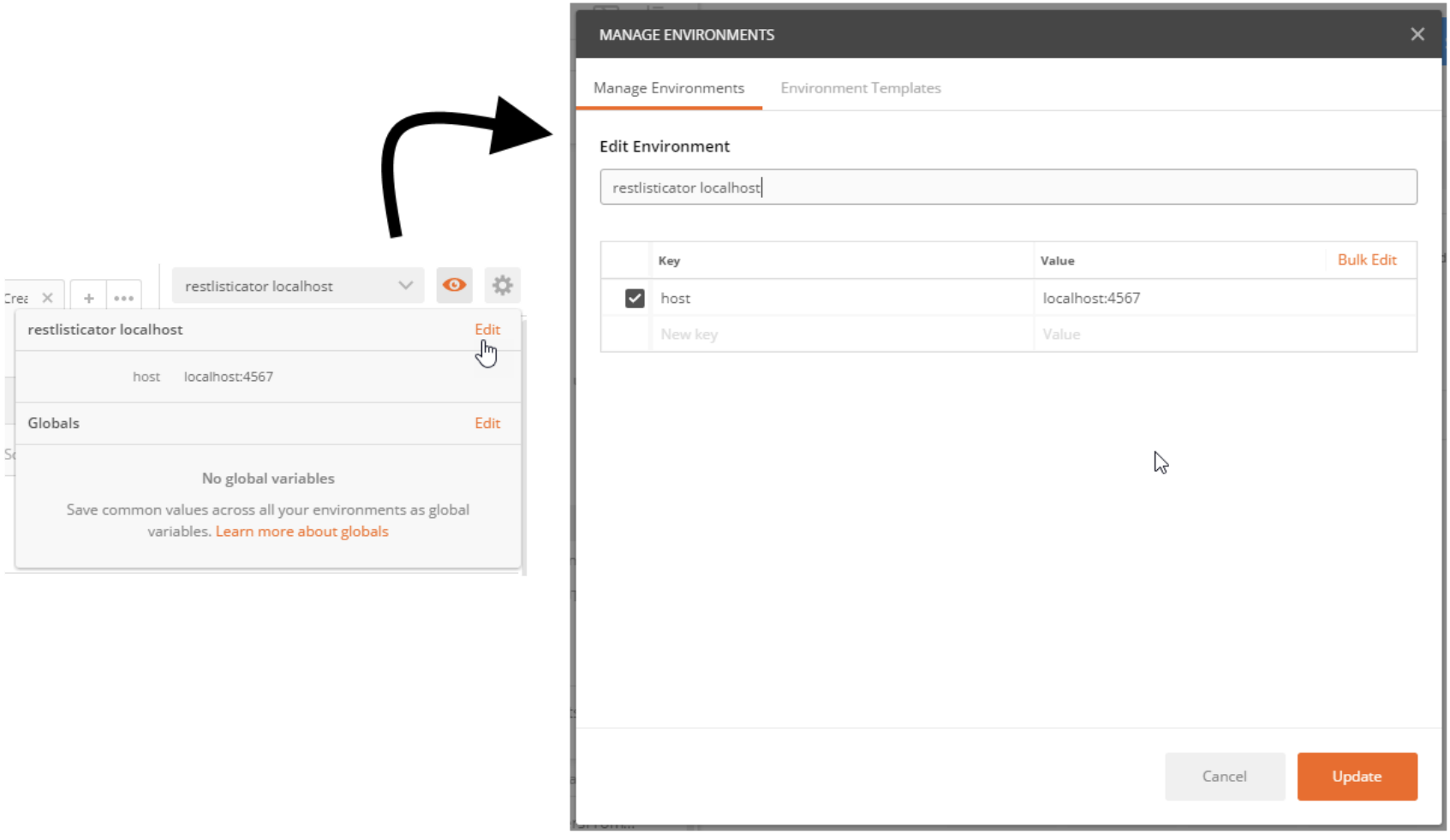
- use environment variables e.g. `{'host':'localhost:4567'}`
- just type `host` for auto complete in URL editing

Postman Create Environment

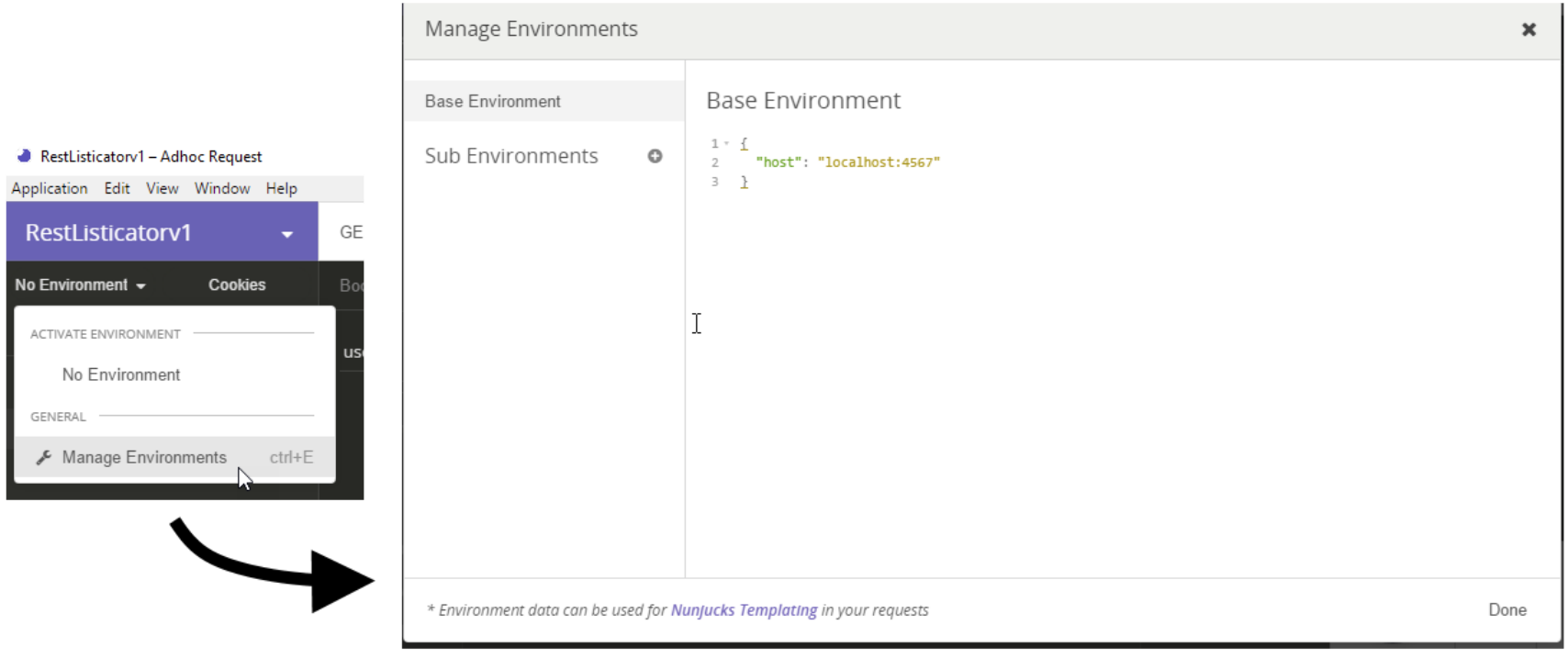




Postman Add Environment Variables



Insomnia Environment Management



How to test with this information

- Read the standards for the verbs and the status codes.
- Projects often argue about interpretations.
- Some of the standards are exact enough that it is possible to say "I observed X" it does not match the standard - include links and quotes to the standards.

Exercises

- install the REST listicator
  - http://compendiumdev.co.uk/downloads/apps/restlisticator
    - /v1/rest-list-system.jar download the `.jar` file
    - /v1/documentation.html read the documentation
- in the directory you downloaded it to type:
  - `java -jar rest-list-system.jar`
- try out the above GET, POST, DELETE, OPTIONS and POST using a GUI client e.g. Postman, or Insomnia
- see more exercises in exercise section
- explore the HTTP Client functionality and test the API based on its documentation
- see Exercises section for more exercises

Section - Exercise Time

- Time to experiment
- Explore the HTTP Verbs etc. mention
- Use the tools mentioned
- Test the webservices mentioned
- Explore the exercises in the exercises section

SECTION - MORE HTTP REQUESTS AND RESPONSES

Overview of Section - More HTTP Requests and Responses

- HTTP Verbs - PUT, OPTIONS
- URI vs URL
- HTTP Standards
- cURL

URI - Universal Resource Identifier

`scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]`

- `http://compendiumdev.co.uk/apps/api/mock/reflect`
  - scheme = `http`
  - host = `compendiumdev.co.uk`
  - path = `apps/api/mock/reflect`

wikipedia.org/wiki/Uniform_Resource_Identifier

A URL is a URI

URI vs URL vs URN

- URI - Universal Resource Identifier
  - 'generic' representation - might not include the 'scheme'
  - `http://compendiumdev.co.uk/apps/api/mock/reflect`
  - `compendiumdev.co.uk/apps/api/mock/reflect`
  - `/apps/api/mock/reflect`
- URL - Universal Resource Locator
  - `http://compendiumdev.co.uk/apps/api/mock/reflect`
  - defines how to locate the identified resource
- URN - Universal Resource Name
  - not often used - uses scheme `urn`

Scheme(s)

- http
- https
- ftp
- mailto
- file

Query Strings

```
GET /lists/{guid}?without=title,description
GET http://localhost:4567/lists/f13?without=title,description
```

Query String:

```
?without=title,description
```

- starts with `?`
- params separated with `&`

More About Query Strings

```
GET /lists/{guid}?without=title,description
```

- usually `name=value` pairs separate by '&'
  - convention since anything after the `?` is the Query string
  - app then parses as required
- can be used with any verb
- `GET` request - all params are send as query strings

https://en.wikipedia.org/wiki/Query_string

HTTP Standards?

- rfc7231 (HTTP/1.1): Semantics and Content
- rfc7230 (HTTP/1.1): Message Syntax and Routing

HTTP PUT Verb

- PUT - create or replace from full information

Full information means it should be idempotent - send it again and get exactly the same request

Demo

HTTP PUT Send Example

```
curl -X PUT http://localhost:4567/lists ^
-H "Authorization: Basic dXNlcjpwYXNzd29yZA==" ^
--proxy 127.0.0.1:8888 ^
-d @createlistwithput.txt
```

where `createlistwithput.txt` file contains

```
{"title":"title added with put",
"description":"list description",
"guid": "guidcreatedwithput2017081714440",
"createdDate": "2017-08-17-14-40-34",
"amendedDate": "2017-08-17-14-40-34"}
```

HTTP PUT Request Example

```
PUT http://localhost:4567/lists HTTP/1.1
User-Agent: curl/7.39.0
Host: localhost:4567
Accept: */*
Connection: Keep-Alive
Authorization: Basic dXNlcjpwYXNzd29yZA==
Content-Length: 180
Content-Type: application/json

{"title":"title added with put",
"description":"list description",
"guid": "guidcreatedwithput2017081714440",
"createdDate": "2017-08-17-14-40-34",
"amendedDate": "2017-08-17-14-40-34"}
```

HTTP PUT Response Example

```
HTTP/1.1 201 Created
Date: Thu, 17 Aug 2017 13:41:46 GMT
Content-Type: application/json
Server: Jetty(9.4.4.v20170414)
Content-Length: 0
```

HTTP OPTIONS Verb

- OPTIONS - verbs available on this url
- returns an `Allow` header describing the allowed HTTP Verbs

HTTP OPTIONS Send Example

```
curl -X OPTIONS http://localhost:4567/lists ^
--proxy 127.0.0.1:8888
```

Demo

HTTP OPTIONS Request Example

```
OPTIONS http://localhost:4567/lists HTTP/1.1
User-Agent: curl/7.39.0
Host: localhost:4567
Accept: */*
Connection: Keep-Alive
```

HTTP OPTIONS Response Example

```
HTTP/1.1 200 OK
Date: Thu, 17 Aug 2017 12:24:39 GMT
Allow: GET, POST, PUT
Content-Type: text/html;charset=utf-8
Server: Jetty(9.4.4.v20170414)
Content-Length: 0
```

Common HTTP Status codes in response to a OPTIONS

- **200** - OK, did whatever I was supposed to
- **404** - I can't post to that url it is not found

HTTP OPTIONS Verb - Example swapi.co

e.g. swapi.co

OPTIONS - https://swapi.co/api/people/1/


```
{
    "name": "People Instance",
    "description": "",
    "renders": [
        "application/json",
        "text/html",
        "application/json"
    ],
    "parses": [
        "application/json",
        "application/x-www-form-urlencoded",
        "multipart/form-data"
    ]
}
```

Exercises

- try out the above OPTIONS and PUT using a GUI client e.g. Postman, or Insomnia
- try with different body content e.g. xml vs json
- try requesting `application/xml` instead of `application/json`
- explore the HTTP Client functionality and test the API based on its documentation
- see Exercises section for more exercises

SECTION - REST API BASICS

Overview of Section - REST API BASICS

- What is a REST API?
- CRUD and REST
- HTTP Verbs - HEAD, PATCH
- Authentication and Authorisation
- Postman collection runner

What is a REST API?

- HTTP API - generic, anything goes
- REST API
  - the HTTP Verbs mean something specific e.g. should not Delete with a POST request
  - URI are 'nouns' and describe entities

REST Standards?

Representational State Transfer

- Loose standards
- Lots of disagreement on teams and online
- DISSERTATION: "Architectural Styles and the Design of Network-based Software Architectures" by Roy Fielding
    - ics.uci.edu/~fielding/pubs/dissertation/top.htm

Guidance

- Idempotent - same request, same result (on server, not necessarily in response)
- Stateless - server does not need to maintain state of client requests between requests e.g.
  - request 1: select these files,
  - request 2: delete files selected in previous request
- Cacheable - on the server side e.g. GET can be cacheable until entities in GET are updated
- Does it comply with HTTP Standard Guidance?

CRUD

- Verbs are not as simple as Create, Read, Update, Delete

CRUD Action	Verb
Create	POST, PUT
Read	GET
Update	POST, PUT, PATCH
Delete	DELETE

Endpoints vs URL

Very often when discussing REST APIs we talk about 'endpoints'.

Basically the 'path' part of the URL.

The following are the same Endpoint

- `/lists`
- `/lists?title="title"`

Payloads vs Body

A Payload is the content of the body of the HTTP request.

- XML and JSON
- Tends not to be Form encoded
- Request defined by `content-type` header
- Response requested in `accept` header
- usually unmarshalled into an object in the application

Requesting Formats

Header	Means
Accept: application/json	Please return JSON
Accept: application/xml	Please return XML
Content-Type: application/json	This payload is JSON
Content-Type: application/xml	This payload is XML

- XML might also be : `text/xml`
- The server might not support a particular format it might default to JSON or XML and ignore the header

Authentication

If you make a request to a server and receive a 401 then you are not authenticated.

`WWW-Authenticate` header should challenge you with the authentication required.

- Generally avoid header sending by known authenticating information in request.
- Common bug is `WWW-Authenticate` not sent back in response.

Common Authentication Approaches

- Basic Auth Header
  - `Authorization: Basic Ym9iOmRvYmJz`
  - base 64 encoded `username:password`
- Cookies
  - when 'login' server sends back a 'session cookie'
  - send 'session cookie' in future requests
- Custom Headers
  - API `secret codes`
  - e.g. `X-API-AUTH: thisismysecretapicode`

Common Authentication Approaches

- URL authentication
  - `https://username:password@www.example.com/`
  - deprecated
  - used to be very common when automating web GUIs

Recommended reading developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

Authentication vs Authorization

Authentication

- Are you authenticated?
- Does the system know who you are?
- Are your auth details correct?

Authorization

- you are authenticated
- do you have permission to access this endpoint?

Real World vs Standards

Teams debate this all the time.

- Login? stackoverflow.com/questions/13916620
- Put vs Post stackoverflow.com/questions/630453
- see discussions on restcookbook.com

As a Tester:

- Refer to HTTP standards
    - headers, idempotency, response recommendations

Expect 'discussions' and 'debates' on a team.

Verb - Head

- HEAD
- same as GET but does not return a 'body'
- can be useful for checking 'existence' of an endpoint or entity

Verb - Patch

- PATCH - An 'Update' method which provides a set of changes
- Contentious see
- Proposed standard for JSON Merge Patch format
- Promosed standard for XML Patch Using XPath

Most web services just use `POST` or `PUT`

Postman Collection Runner

- send multiple requests iterating over data files
- runs all requests in a collection
- create requests with params in body or query
- put data in a CSV file
- run collection, with environment, with data

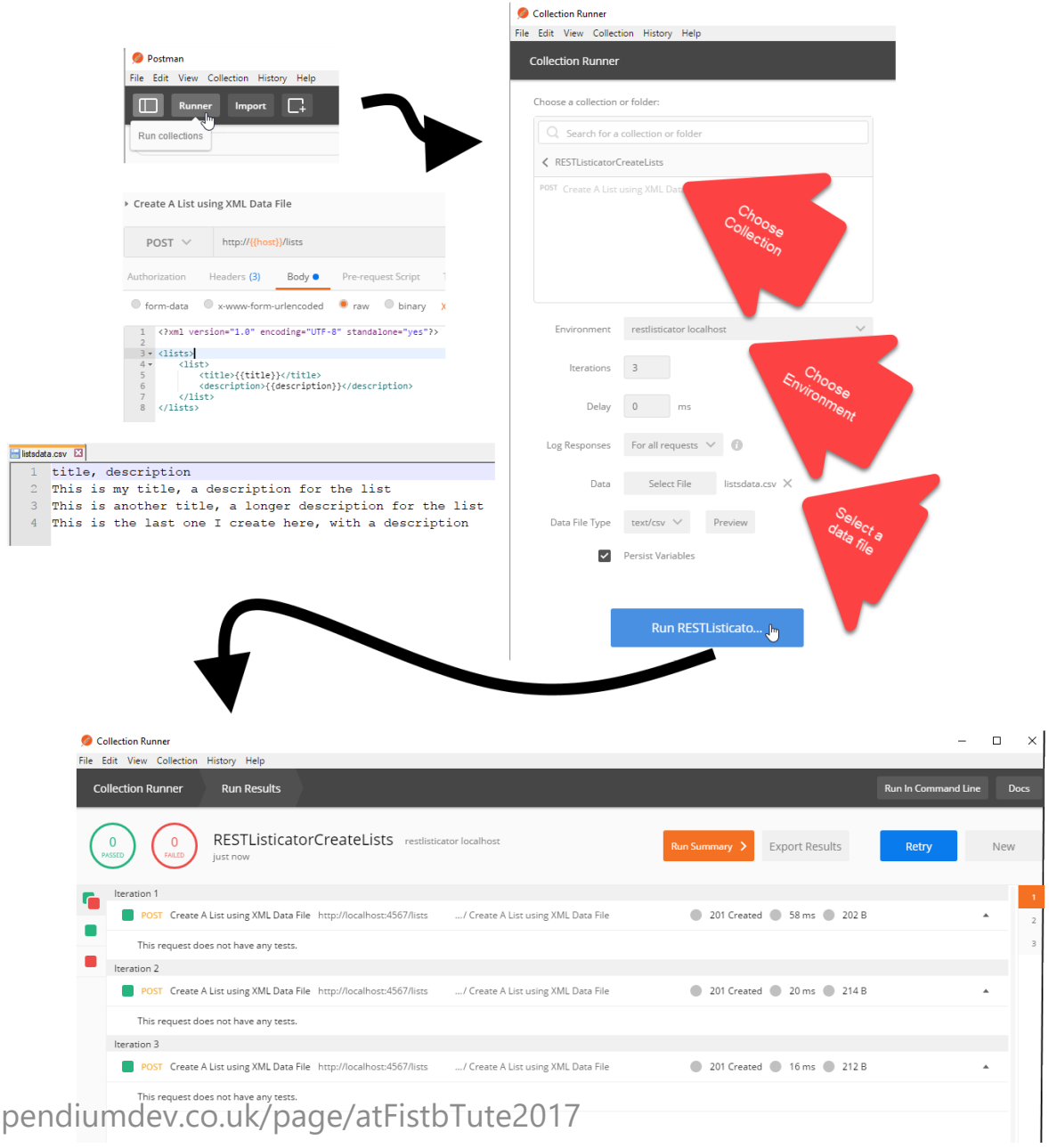
Example Request With Params

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<lists>
    <list>
        <title>{{title}}</title>
        <description>{{description}}</description>
    </list>
</lists>
```

CSV file with params

```
title, description
This is my title, a description for the list
This is another title, a longer description for the list
This is the last one I create here, with a description
```

Postman Collection Runner example



Exercises

Reading:

- read the REST Dissertation ics.uci.edu/~fielding/pubs/dissertation/top.htm
- Read the docs on authentication developer.mozilla.org/en-US/docs/Web/HTTP/Authentication
- for real world 'discussions' see restcookbook.com

Exercises

Doing:

- Experiment with `HEAD` and `PATCH`
- Continue to experiment with the other verbs and test the Web Service
- Create a Postman Collection to use in the runner which creates 10 new list entities
- for more exercises see the exercise section

SECTION - Testing a REST API

Overview of Section - Testing a REST API

- how to model an API
- testing ideas
- interactive discussion

Testing different from Technology and Tooling

- at this point we have discussed technology and tooling
- time to discuss testing

What would we test?

- ideas?

What are the architecture risks?

- Client -> Web Server -> App Server -> App
- Do we understand the architecture?

What are the capacity risks?

- Performance?
- Load Testing?

What are the security risks?

- Authentication
- Authorisation
- Injection

Data Risks

- minimum data in requests - missing fields, headers
- not enough data in requests
- wrong format data: json, xml, length, null, empty
- malformed data
- consistency? query params across requests?
- are defaults correct?
- duplicate data in payloads?
- headers: missing, malformed, too many, duplicate

Document your testing

- How can you document your testing?

Other Risks or Common Issues?

Exercise: Think through testing

- Read the requirements etc. for REST Listicator.
- Create some test ideas
- Look at the existing testing conducted
- Any ideas from that?
- Test REST Listicator
- Document and Track your Testing in a lightweight fashion

Exercise: Test REST Listicator in Buggy mode

```
java -jar rest-list-system.jar -bugfixes=false
```

- The system has been coded with some known bugs
- these are all fixed by default.
- start with `-bugfixes=false` to have known bugs
- See if you can find them

You can run the app twice on different ports to compare output, use the command line argument `-port` to start up the application on a different port e.g. `-port=1234` would start the app on port `1234`

Section - Exercise Time

- Time to experiment
- Explore the HTTP Verbs etc. mention
- Use the tools mentioned
- Test the webservices mentioned
- Explore the exercises in the exercises section

SECTION - HTTP Proxies

Overview of Section - HTTP Proxies

- What is an HTTP Proxy?
- Example HTTP Proxies?
- Why use an HTTP Proxy?
- How to direct REST Client through Proxy?
    - Inspect Traffic
    - Filter Traffic (System Proxies)
    - Port Config
    - Replay Request
- Fuzzing

Exercises: in browser - GET, viewing traffic, Ajax requests

What is an HTTP Proxy?

- HTTP Proxy captures HTTP Traffic
- Allows replay of requests
- Allows manipulation of responses

Which proxies?

- Fiddler
  - Windows (Beta: Linux, Mac)
- Charles
  - Commercial but allows 30 mins in 'shareware' mode
- BurpSuite
  - Free edition good enough for API Testing
- Owasp ZAP
  - Open Source

Fiddler & Charles act as System Proxies making them easy to use with Postman.

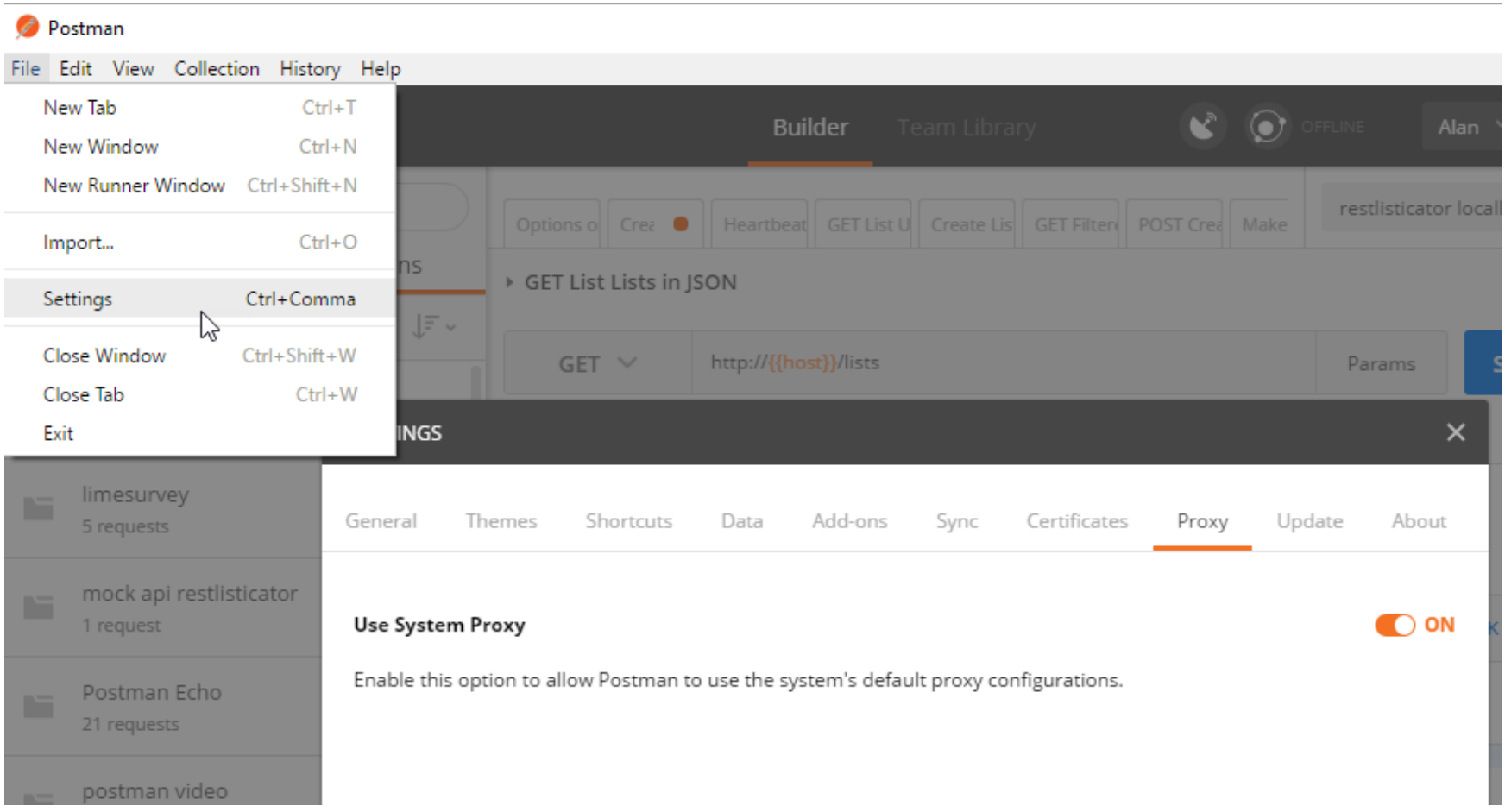
Why use when Testing API?

- Record requests
- Create evidence of your testing
- Replay requests outside of client tool
- Fuzzing

Using a Proxy with Postman

- Change Proxy Settings with "File \ Settings" and then "Proxy" tab
    - on Mac use "Postman \ Preferences" and then "Proxy" tab
- Postman can use a Global Proxy by setting the IP address and Port
    - e.g. BurpSuite, OWasp Zap (or Fiddler and Charles)
- Postman can hook into System Proxy e.g
    - Charles, Fiddler
- Otherwise start postman with `--proxy-server`

Using a Proxy with Postman



Setting Postman proxy from commandline

For full details see blog post

- Mac (type all on one line):

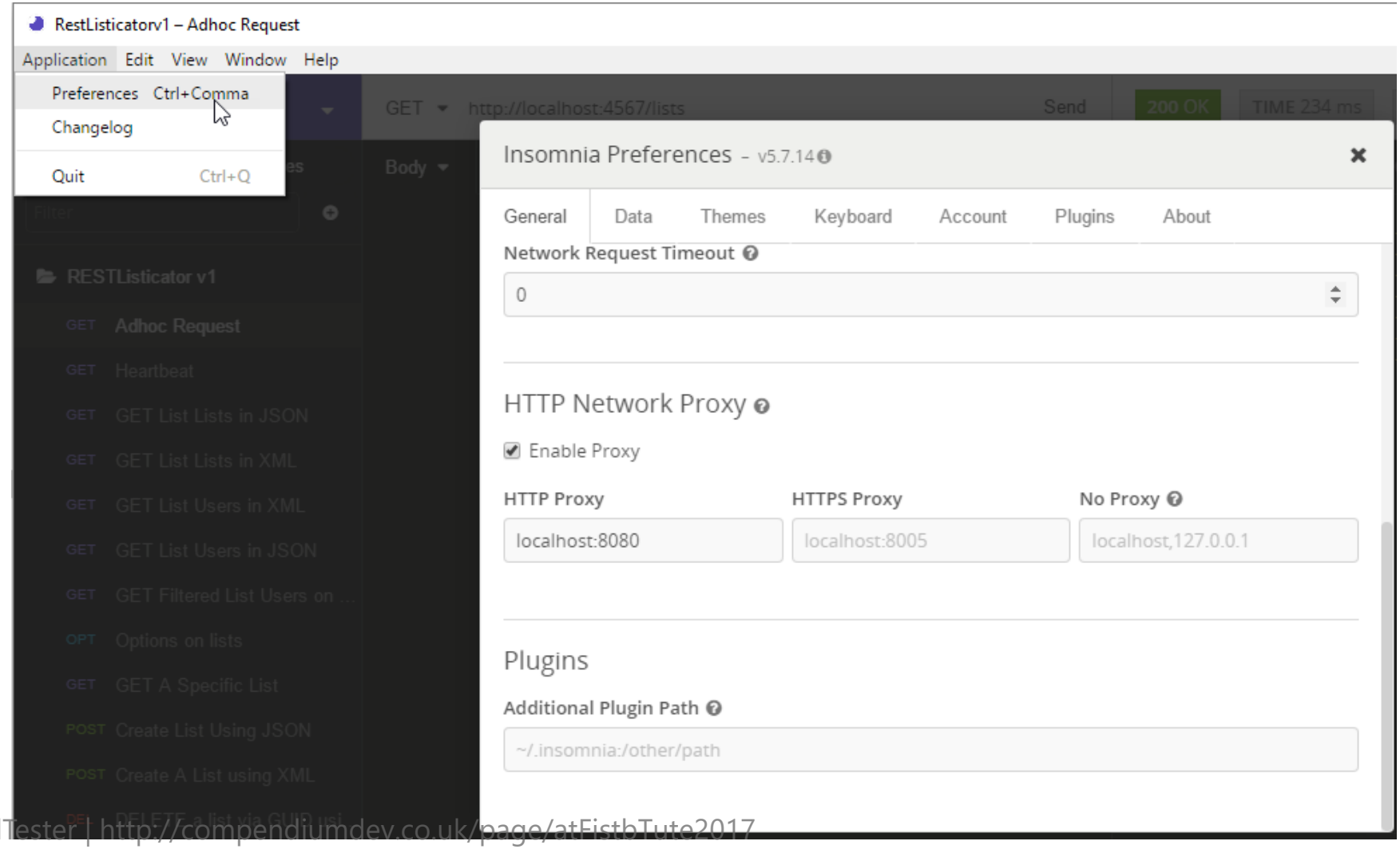
```
open /Applications/Postman.app --args
     --proxy-server=localhost:8888
```

- Windows:

```
cd C:\Users\Alan\AppData\Local\Postman\app-4.9.3\
postman.exe --proxy-server=localhost:8888
```

Using a Proxy with Insomnia

- Application \ Preferences
- on Mac "Insomnia \ preferences"

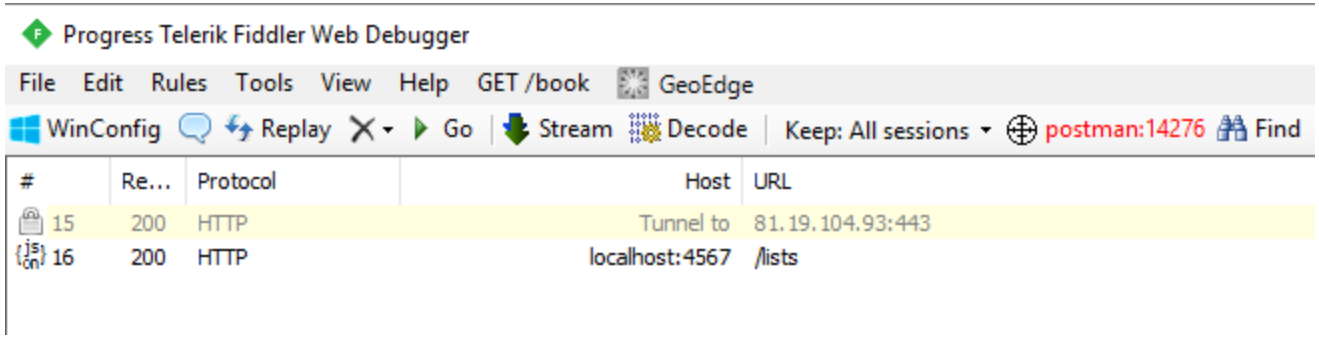


Use of Proxies

- Examples of Fiddler - with screenshots
- Examples of Charles with screenshots
- Examples of Owasp Zap with screenshots
- Examples of BurpSuite with screenshots

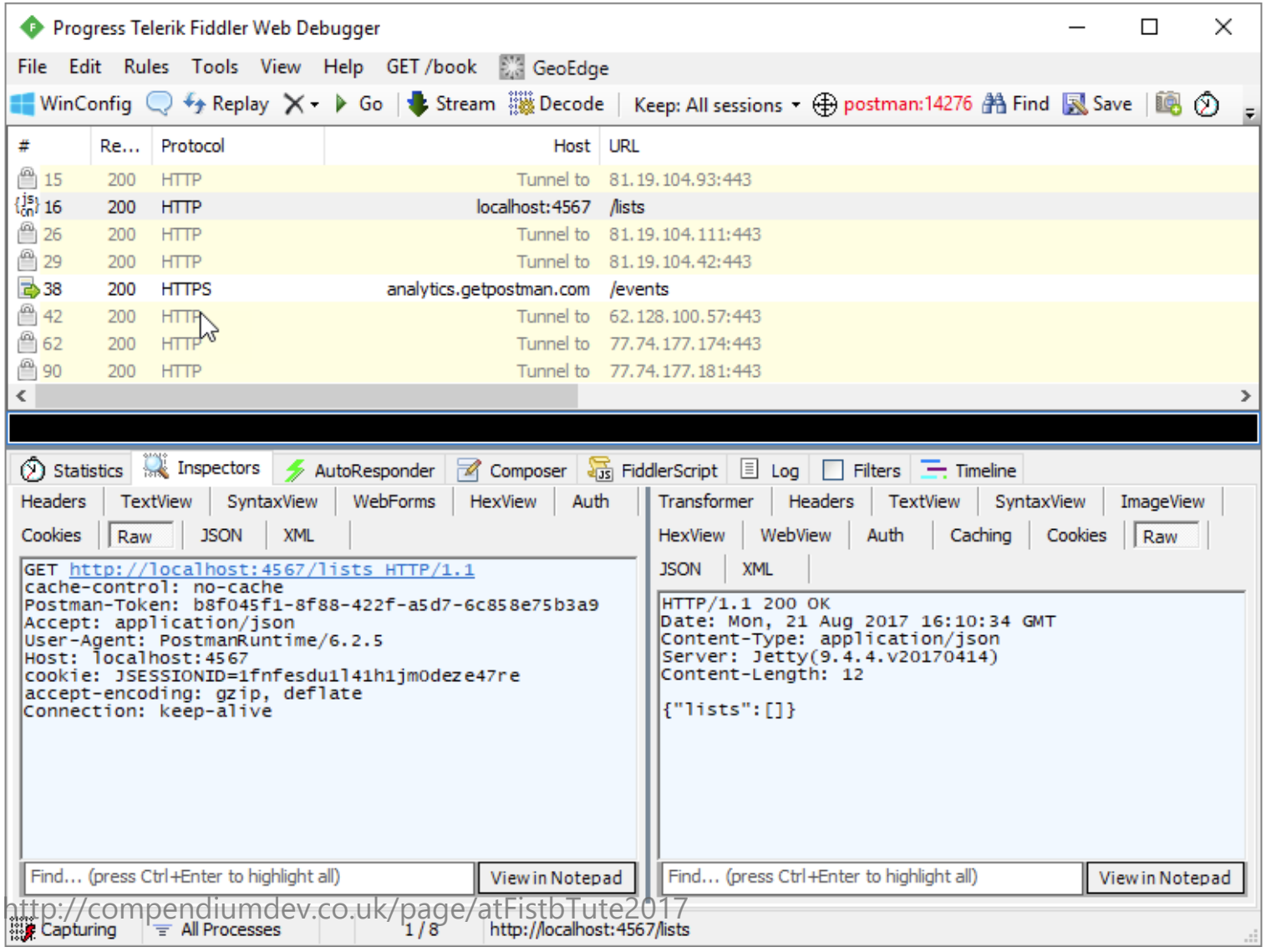
Fiddler Filter Requests

- ctrl+X - to clear traffic history
- filter by process



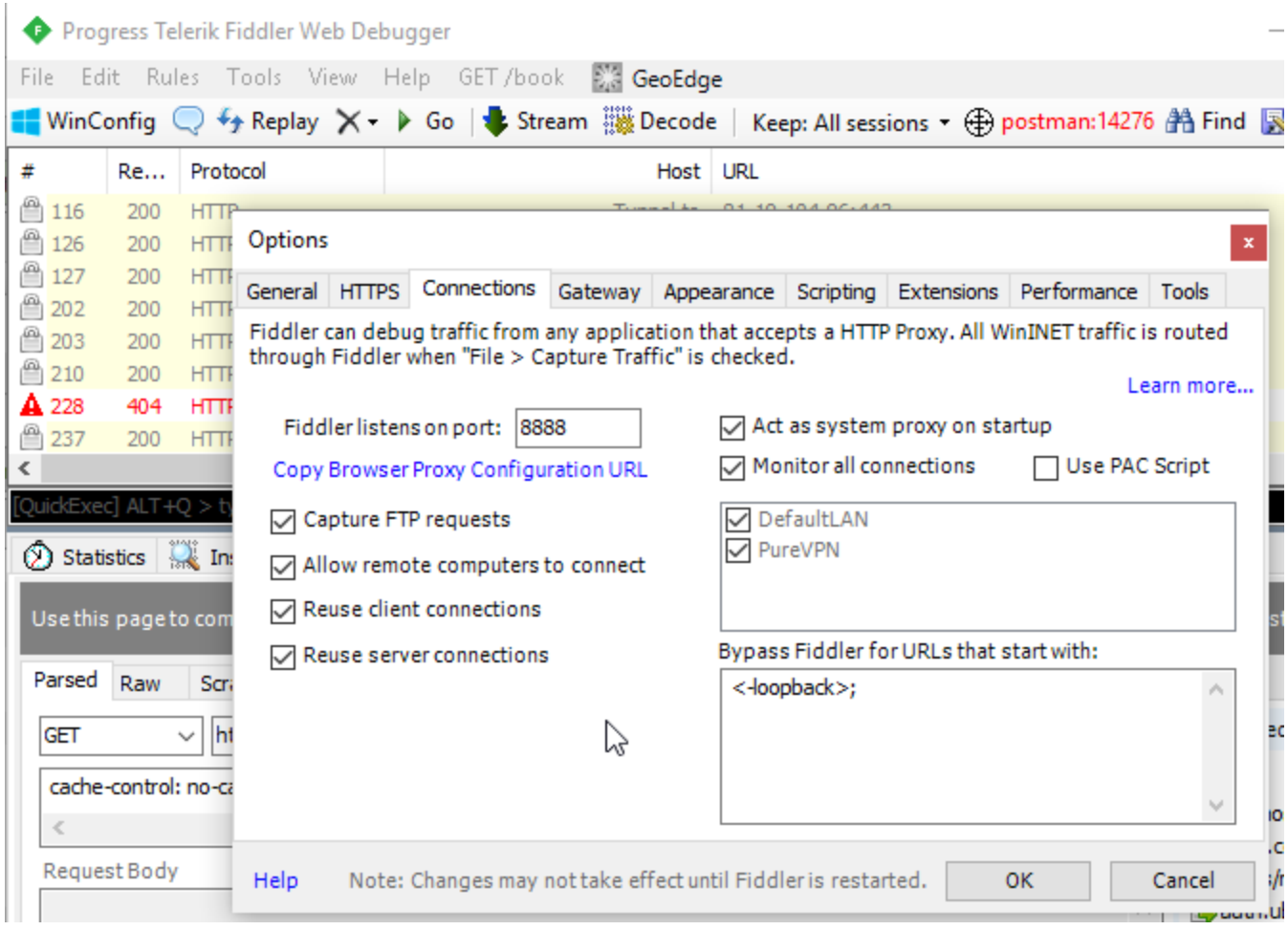
Fiddler Inspect Traffic

- Traffic shown in list - use inspectors to view request and response



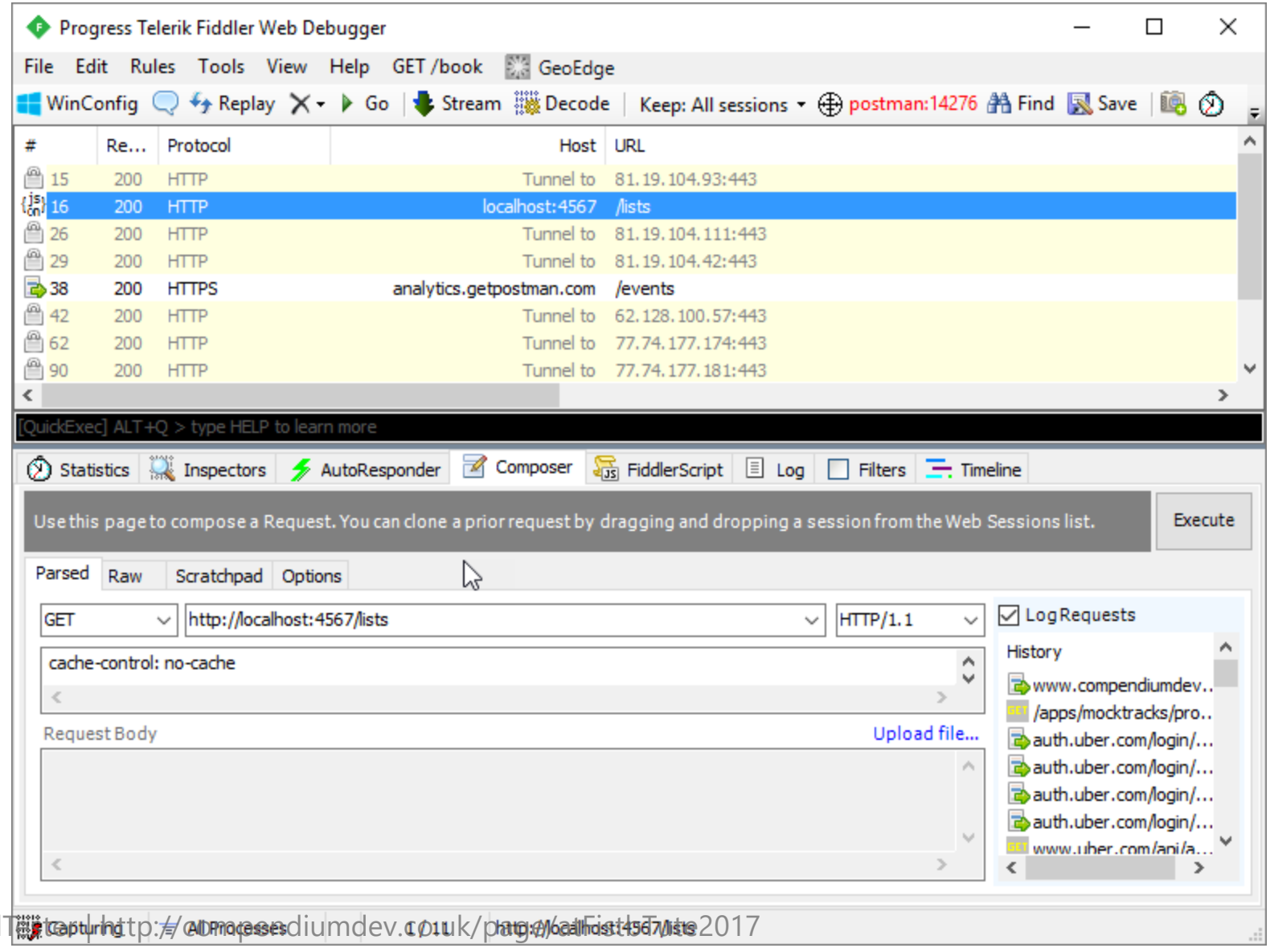
Fiddler listens on port 8888 by default

- find out/change port in `tools \ options`



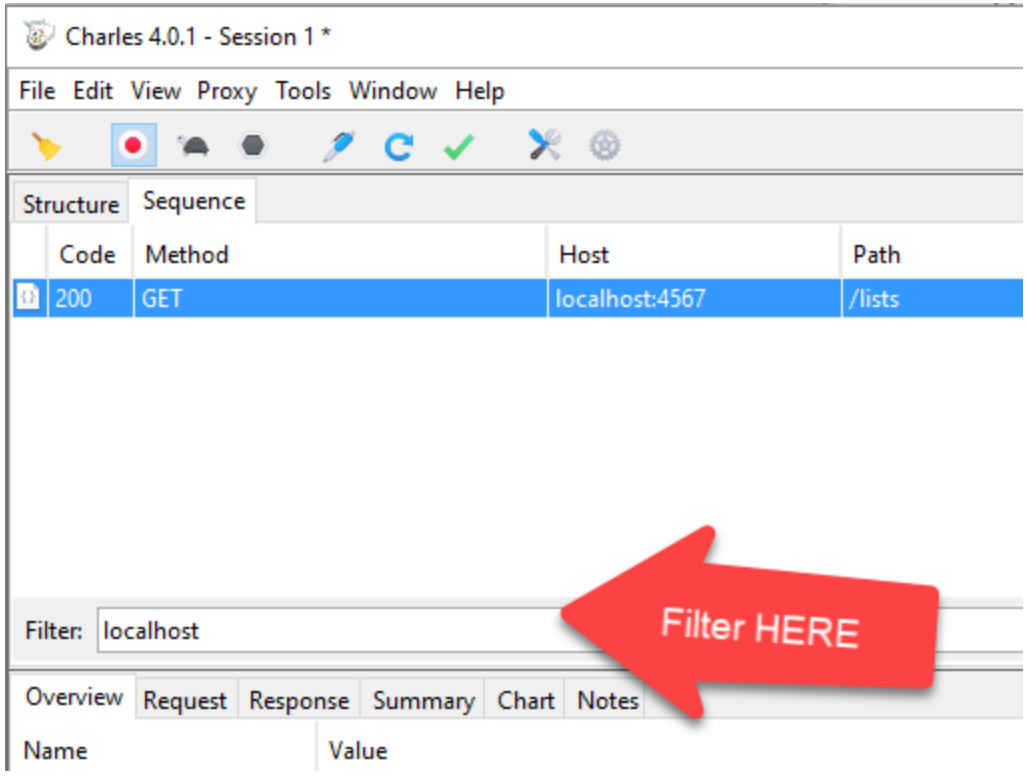
Replay Request in Fiddler

- drag request from history to Composer to edit and replay



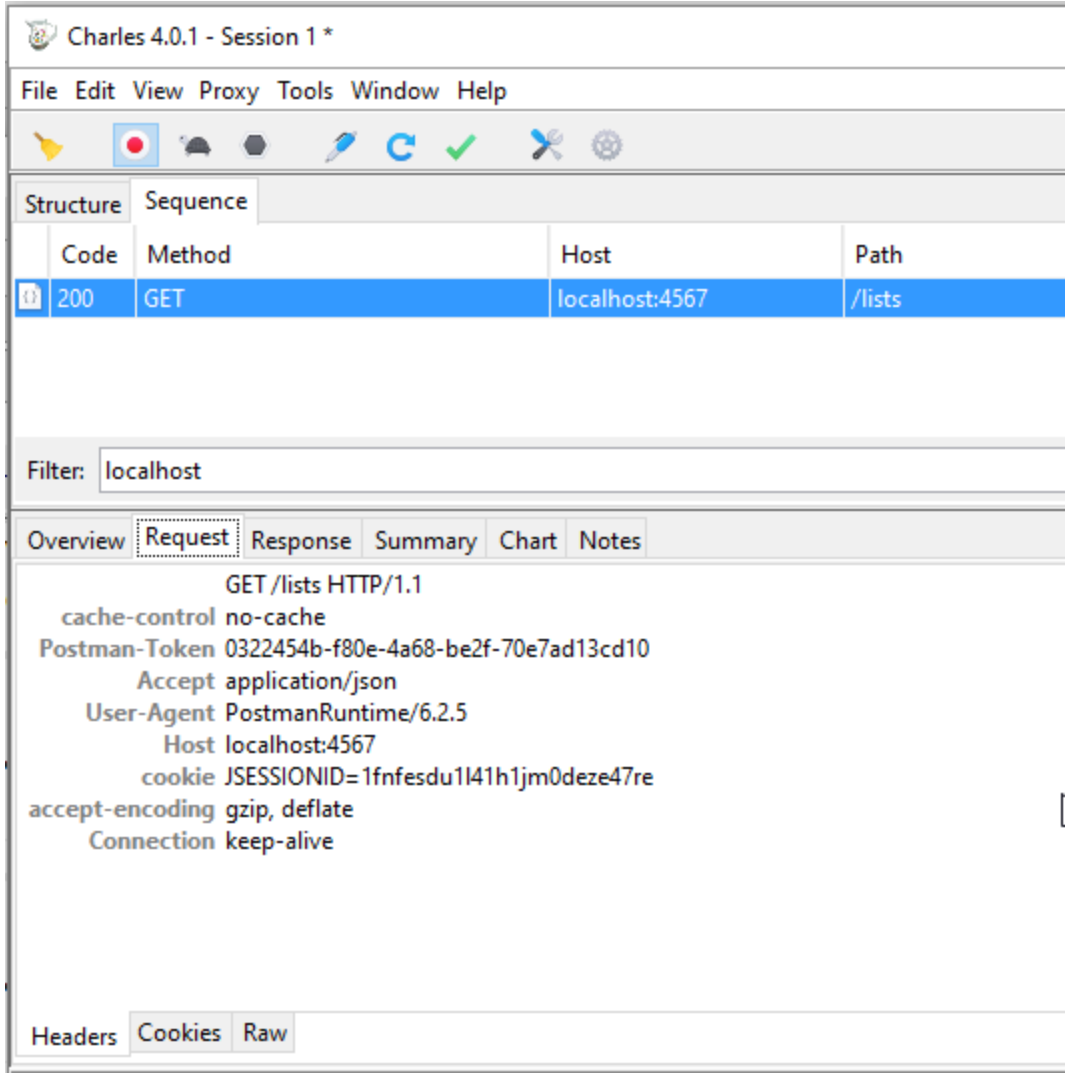
Charles Filter Requests

- Quick filter to `localhost`



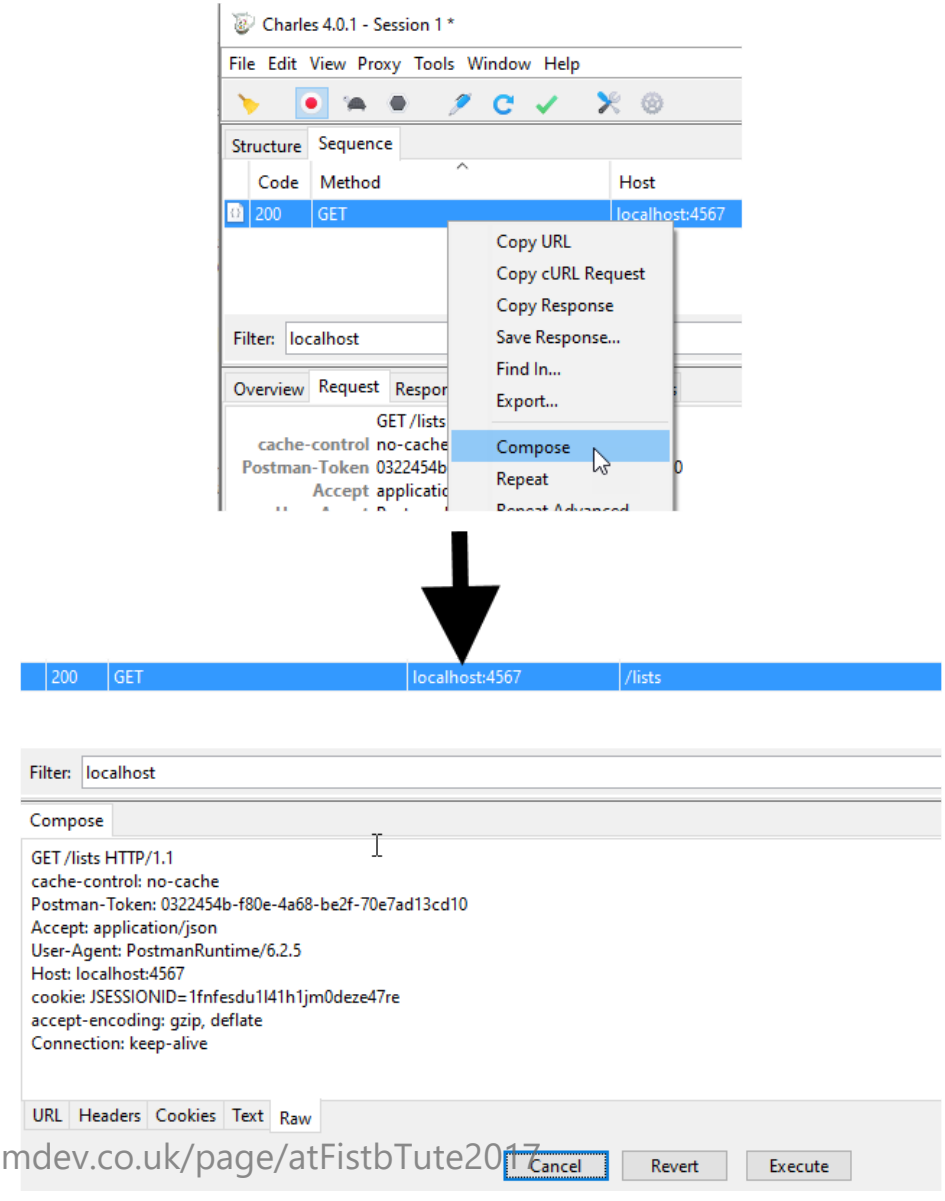
Charles Inspect Traffic

- inspect traffic



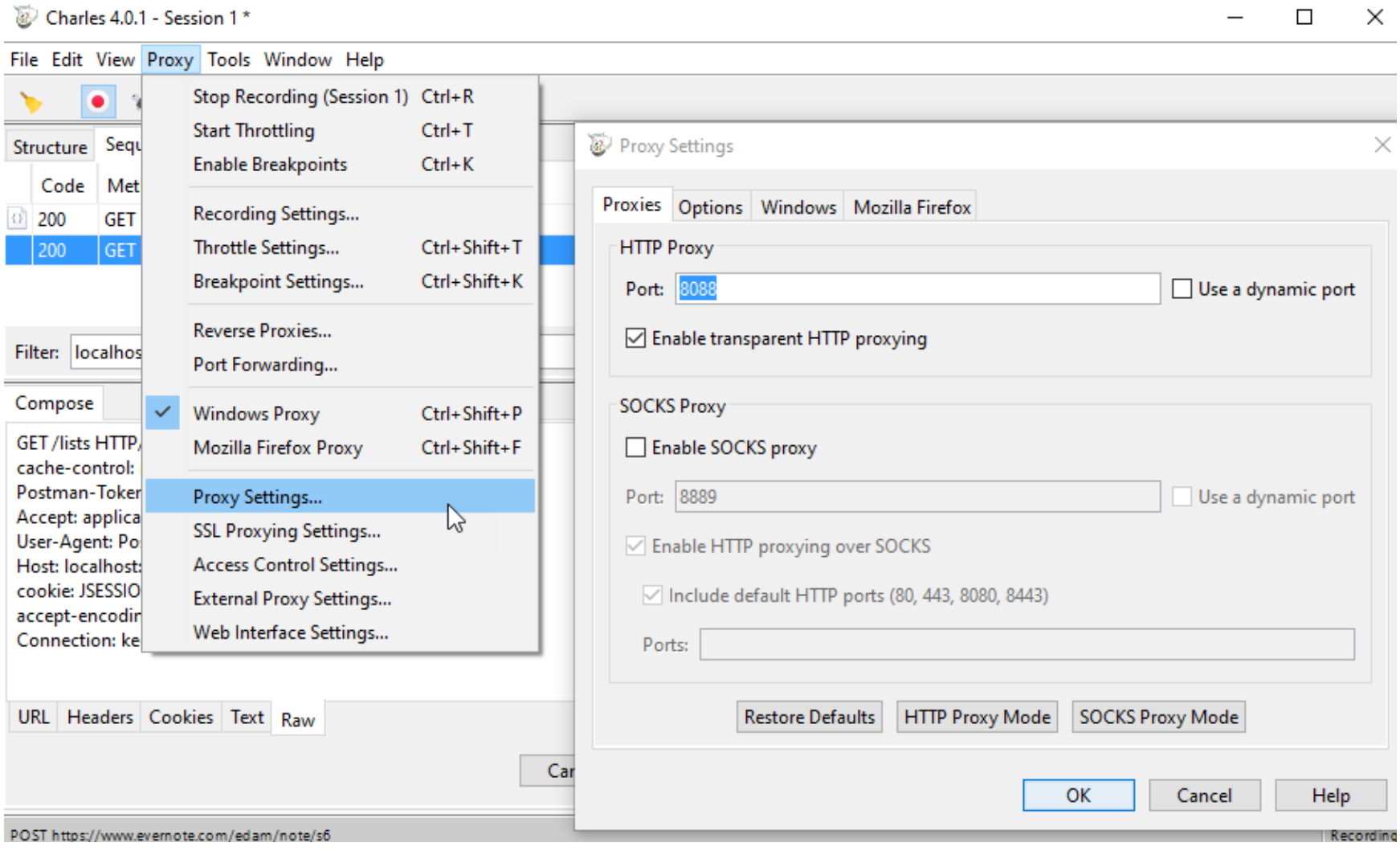
Charles Replay Traffic

- right click 'compose'



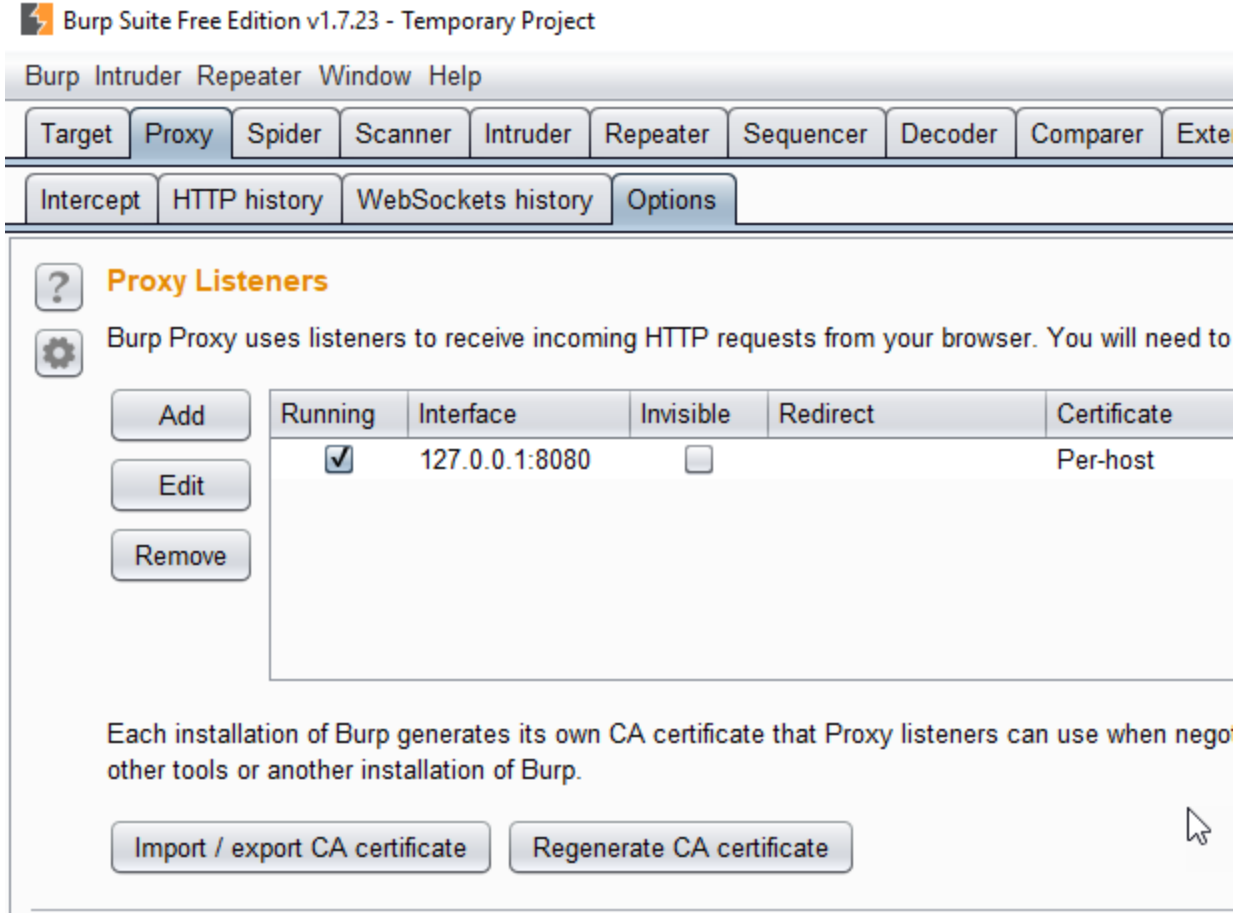
Charles port settings

- `tools \ proxy settings`



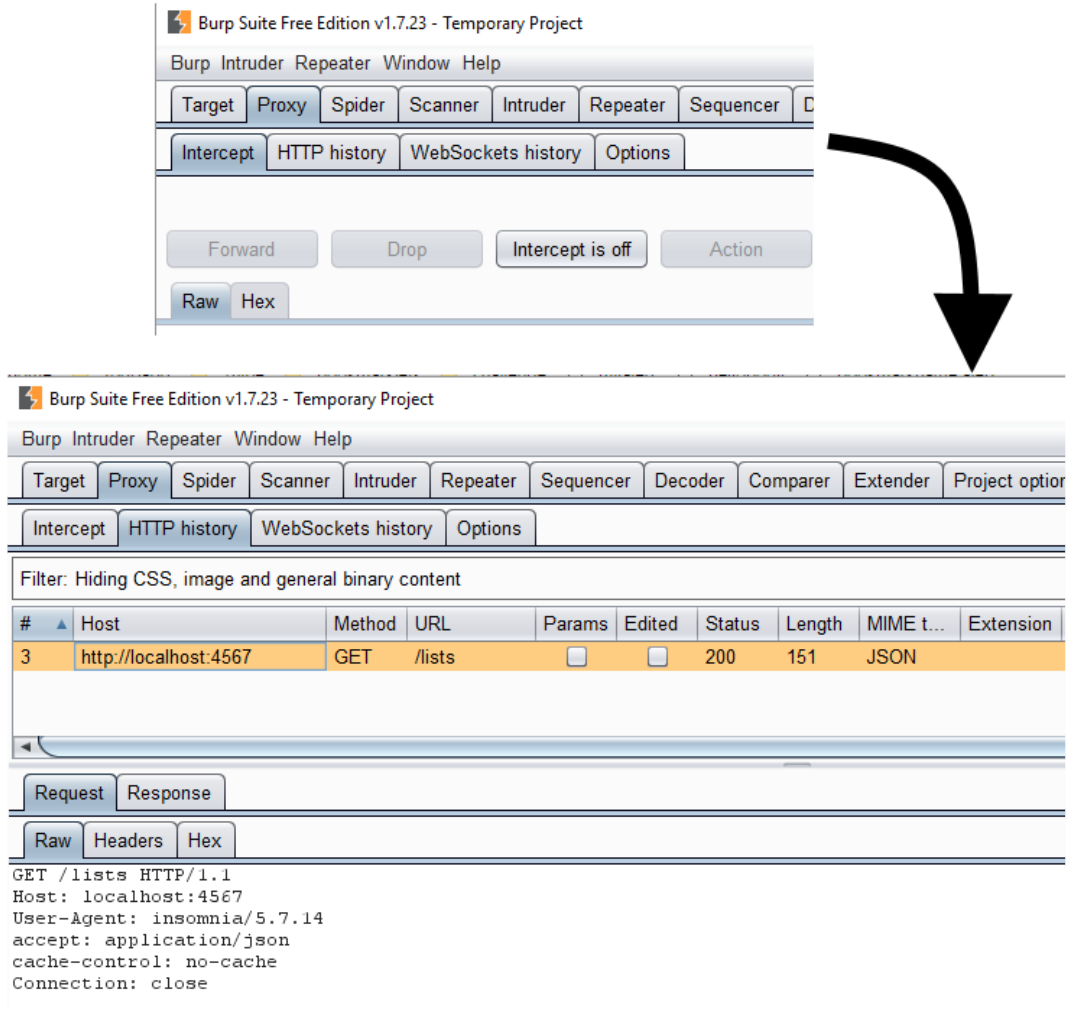
BurpSuite Port Config

- tabs `proxy \ options`



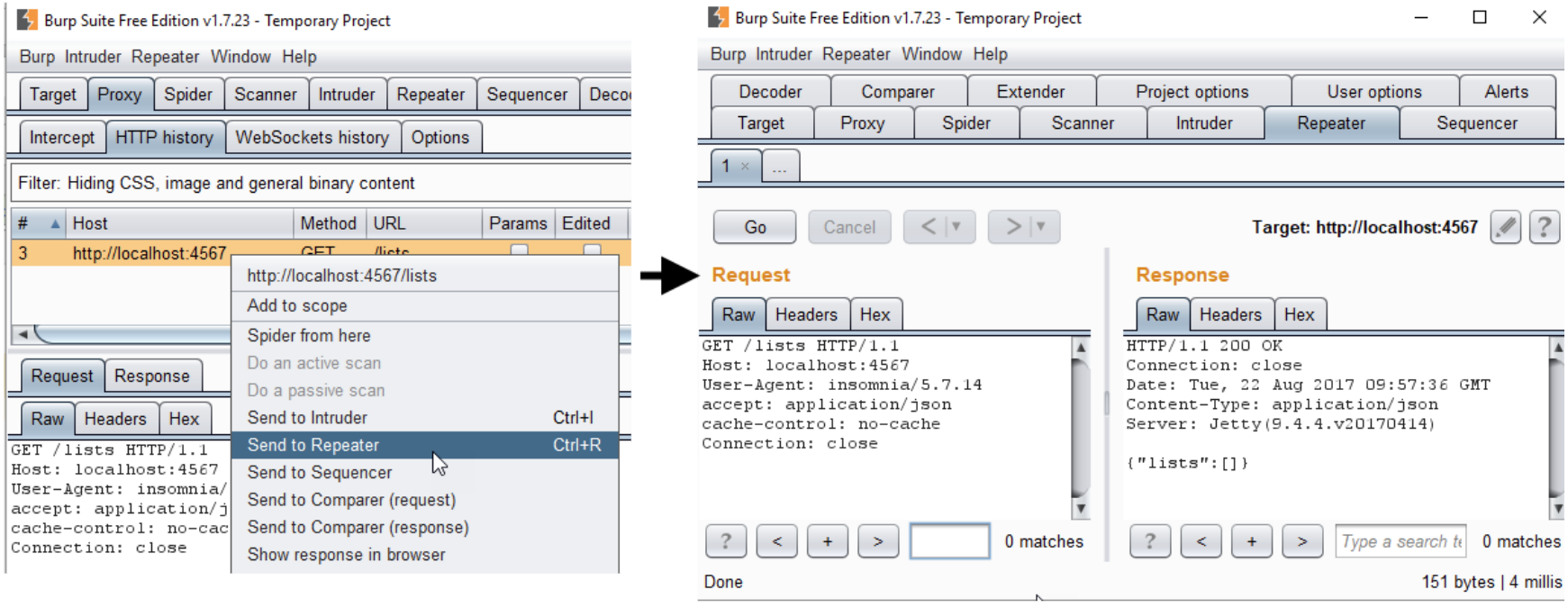
BurpSuite Inspect Traffic

- Ensure intecept is off, view HTTP History



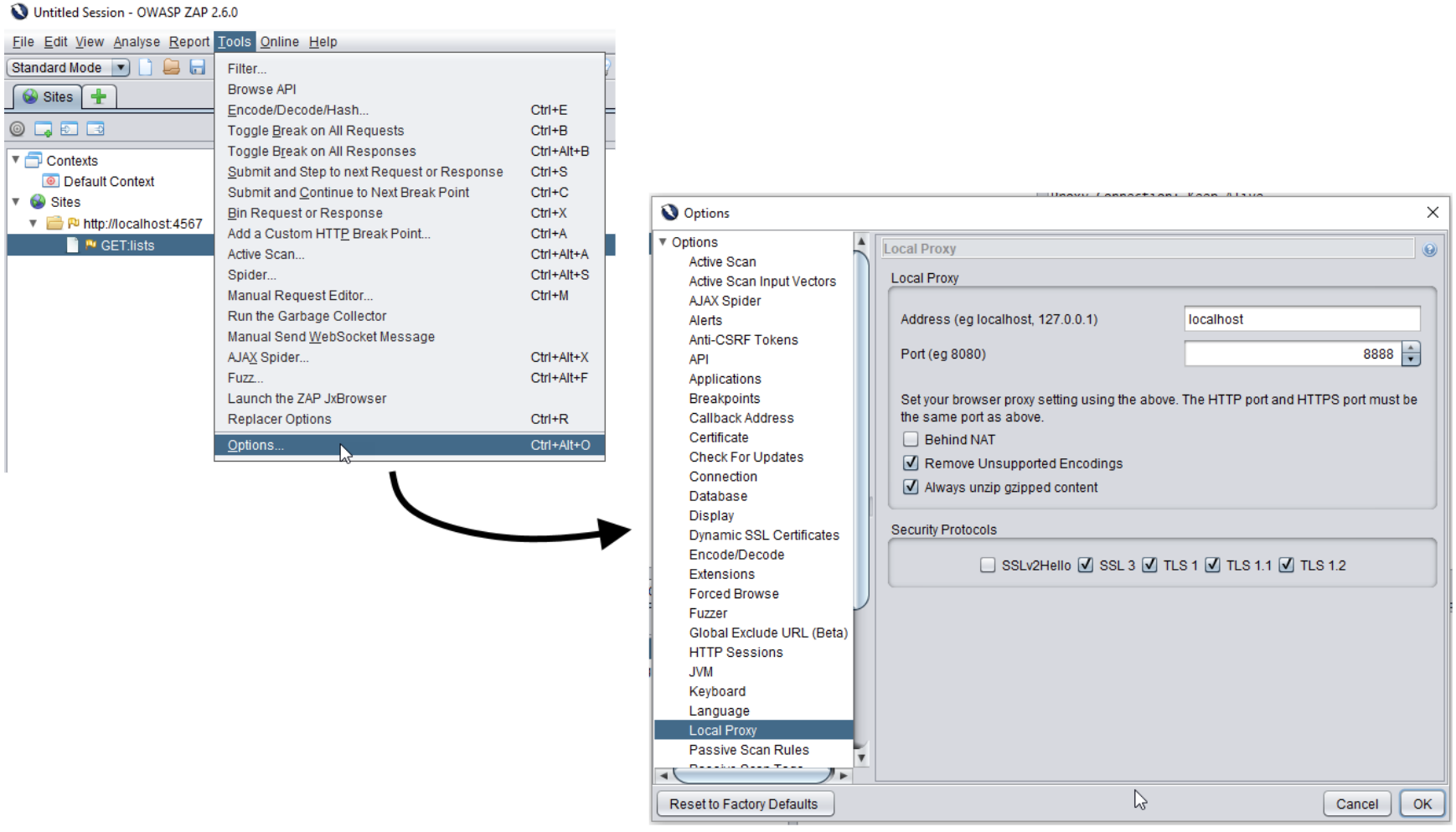
BurpSuite Replay Request

- right click and use repeater



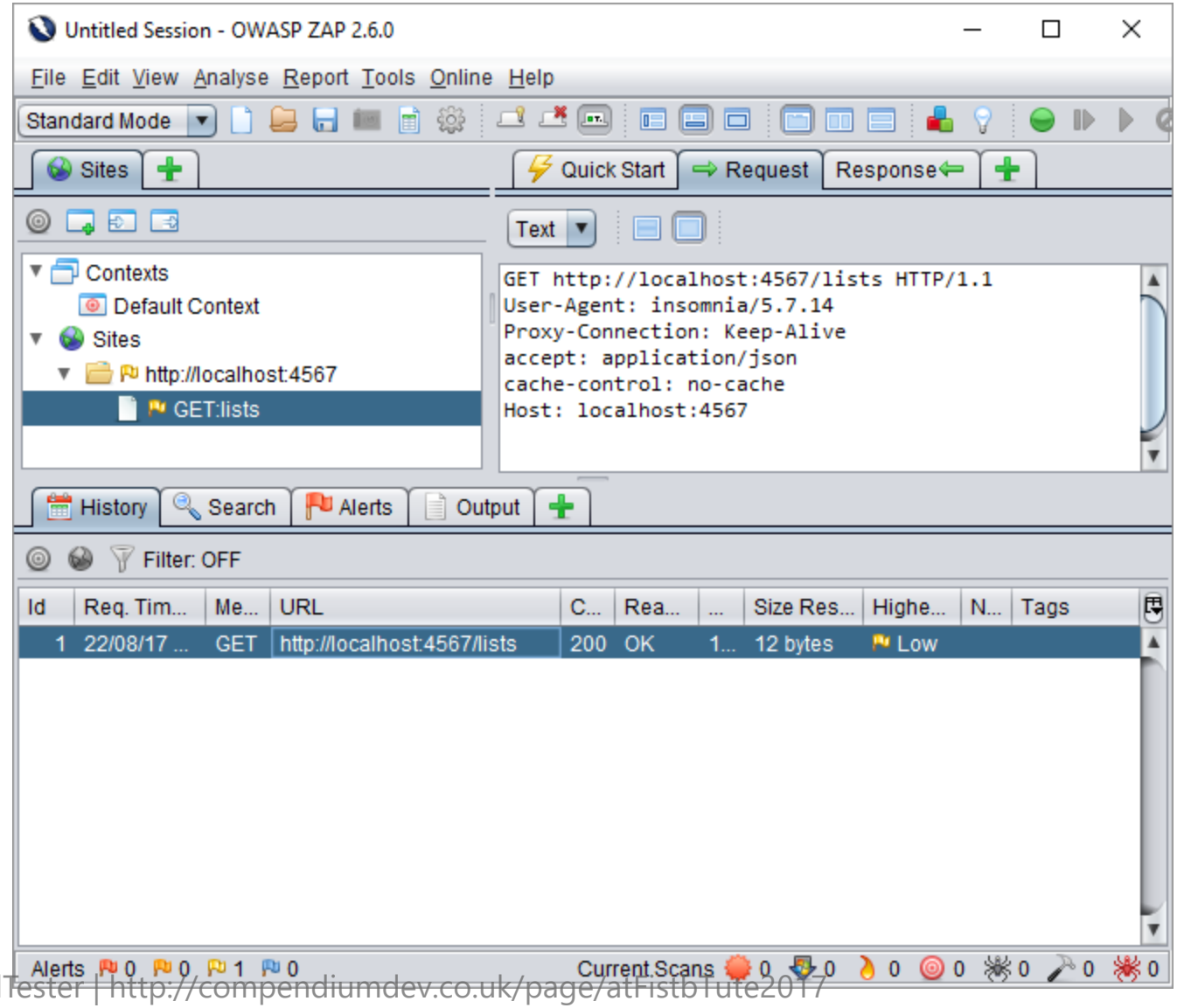
Owasp Zap Port Config

- `tools \ options \ local proxy`



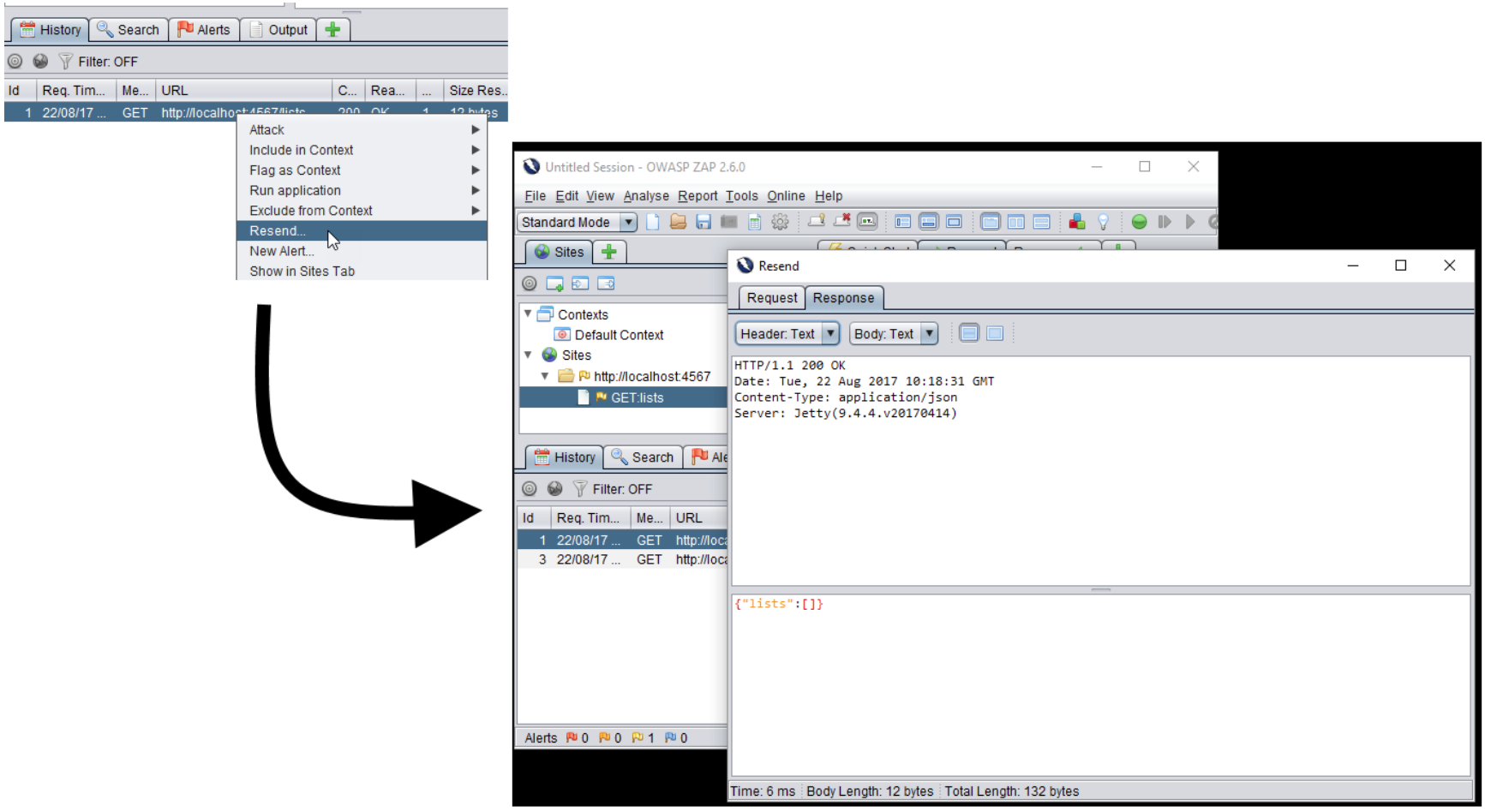
Owasp Zap Inspect Traffic

- history and 'sites', view in top right



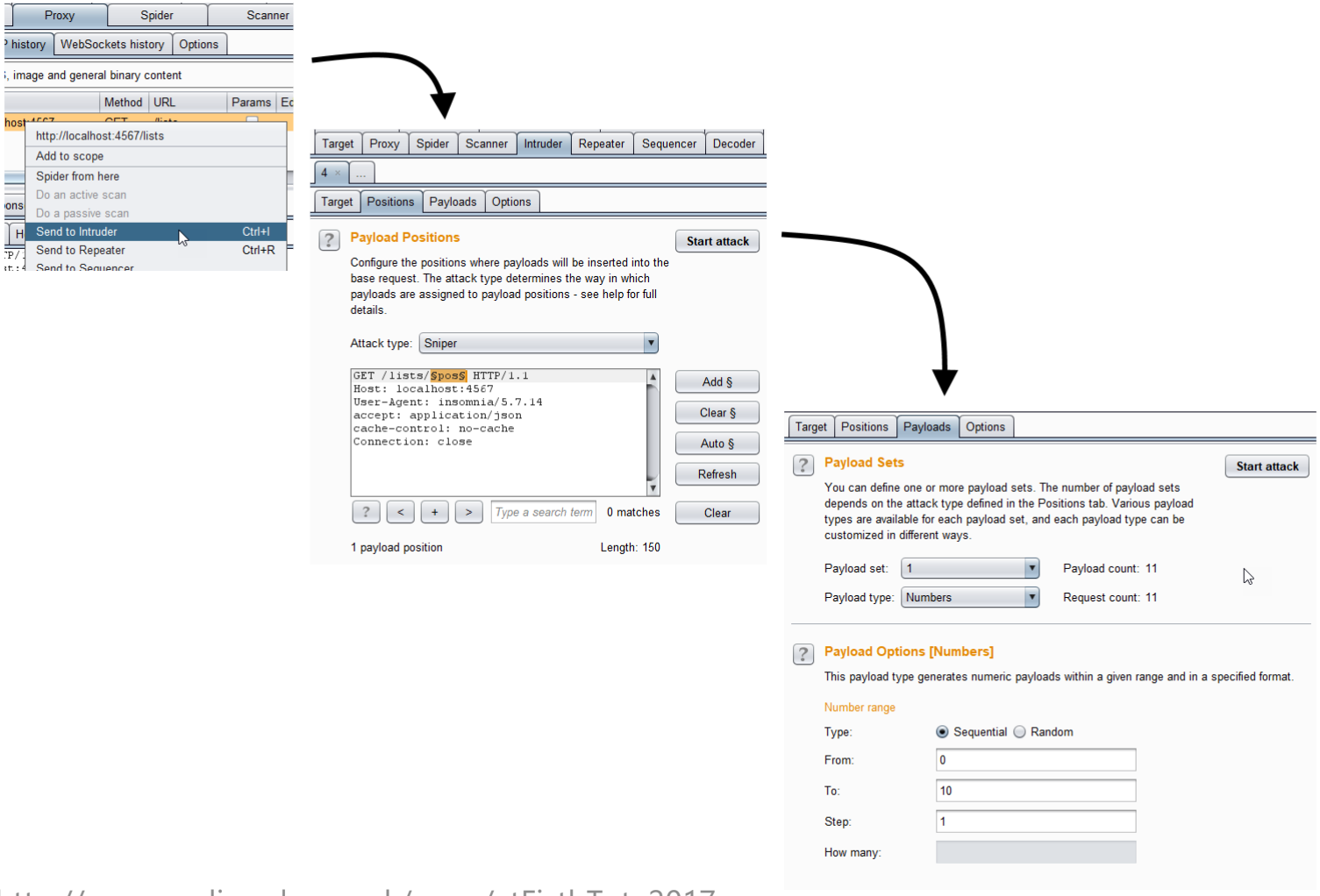
Owasp Zap Replay Request

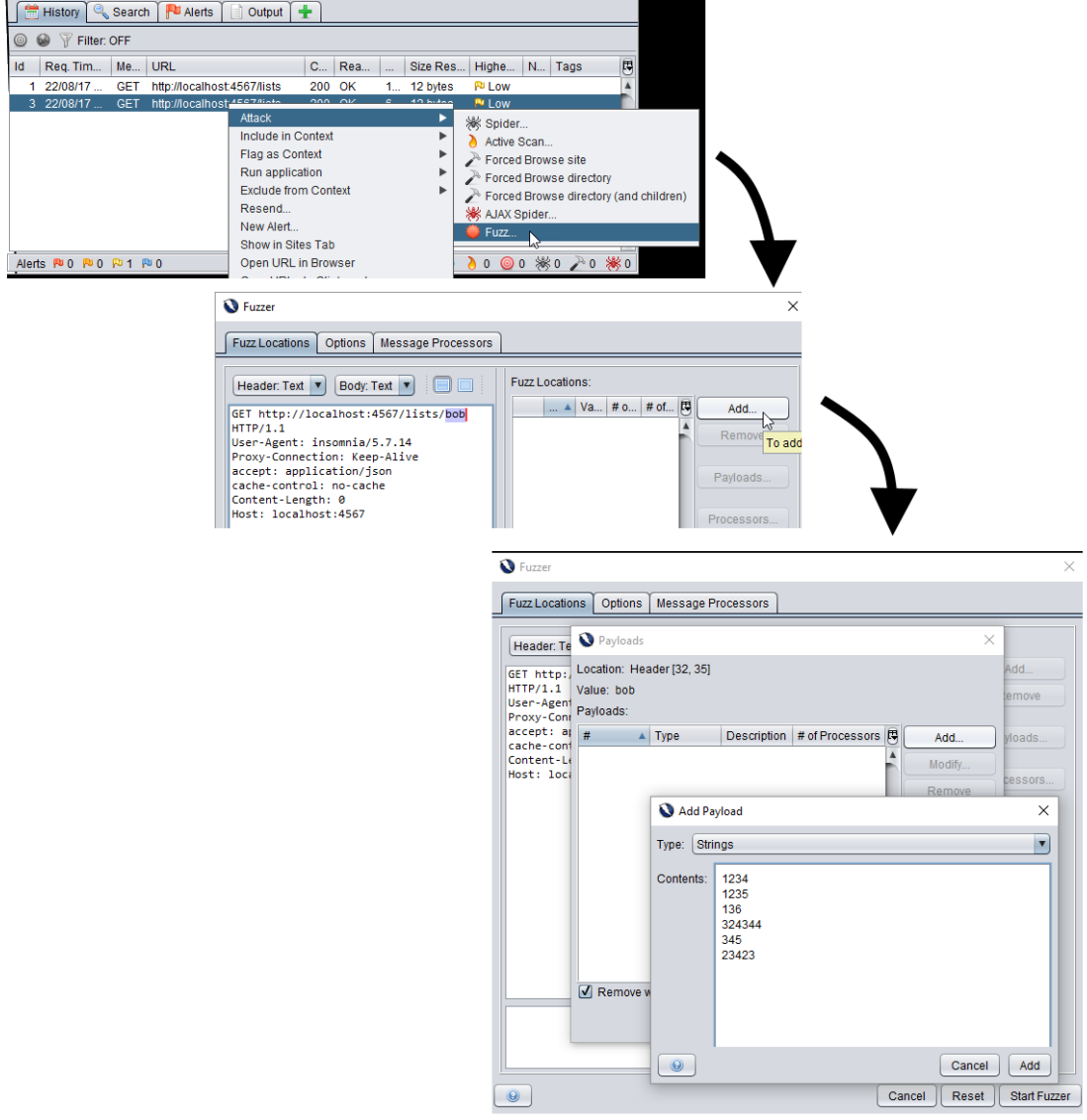
- right click 'Resend', edit and send



Fuzzing in BurpSuite

- right click, intruder, hightlight and add position, edit payload, start attack




History
Search
Alerts
Output
Filter: OFF
Id
Req. Tim...
Me...
URL
C...
Rea...
Size Res...
Highe...
N...
Tags
1 22/08/17 ... GET http://localhost:4567/lists 200 OK 1... 12 bytes Low
3 22/08/17 ... GET Low
Attack
Include in Context
Flag as Context
Run application
Exclude from Context
Resend...
New Alert...
Show in Sites Tab
Open URL in Browser
Spider...
Active Scan...
Forced Browse site
Forced Browse directory
Forced Browse directory (and children)
AJAX Spider...
Fuzz...
Alerts
Fuzzer
Fuzz Locations
Options
Message Processors
Header: Text
Body: Text
GET http://localhost:4567/lists/bob
HTTP/1.1
User-Agent: insomnia/5.7.14
Proxy-Connection: Keep-Alive
accept: application/json
cache-control: no-cache
Content-Length: 0
Host: localhost:4567
Fuzz Locations:
Add...
Remove
To add
Payloads...
Processors...
Payloads
Location: Header [32, 35]
Value: bob
Payloads:
Type Description # of Processors
Add...
Modify...
Remove
Add Payload
Type: Strings
Contents:
1234
1235
136
324344
345
23423
Cancel
Add
Cancel
Reset
Start Fuzzer

Demo

- Insomnia send request through proxy
- Replay
- Use Fuzzer

Exercises:

- send requests through proxy
- replay amended requests
- use fuzzer to experiment
- see expanded Exercises in Exercises section

Section - Exercise Time

- Time to experiment
- Explore the HTTP Verbs etc. mention
- Use the tools mentioned
- Test the webservices mentioned
- Explore the exercises in the exercises section

SECTION - AUTOMATING

Overview of Section -Automating

- Automating
- REST Listicator Example Automating code
- Abstraction Layers
- REST Assured
- Resources to learn from

Why Automate?

- repeatability
- speed
- data coverage
- deployment validation
- support exploratory testing

How?

- Postman?
- HTTP Libraries?
- REST Libraries?
- Which language?
- Other tools?

Examples Using Java and REST Assured

- https://github.com/rest-assured/rest-assured
- Java/Groovy library
- HTTP Abstraction
- Marshalling - Serialization/Deserialization
- Assertions

Basic GET Request

```
@Test
public void canCheckThatServerIsRunning(){

    RestListicatorServer server =
        new RestListicatorServer("localhost",4567);

    RestAssured.
            get(server.getHTTPHost() + "/heartbeat").
            then().assertThat().
            statusCode(200);
}
```

Payload Objects

```
@XmlRootElement(name="list")
public class ListPayload {

    private String title;
    private String guid;
    private String description;

    public String getGuid() {
        return guid;
    }
    public void setGuid(String guid) {
        this.guid = guid;
    }
  ...
}
```

REST Assured

- uses the `content-type` header to (de)serialize to JSON or XML

```
contentType("application/xml")
```

```
contentType("application/json")
```

Marshalling / Serializing

```
public Response createList(ApiUser user, ListPayload list) {
   return RestAssured.
           given().
              contentType(contentType).
              accept(accept).
              auth().preemptive().
              basic(user.getUsername(), user.getPassword()).
              body(list).
            when().
                post(server.getHTTPHost() + "/lists").
            andReturn();
}
```

Code Walkthrough of REST Listicator Automating Examples

- https://github.com/eviltester/rest-listicator-automating-examples
- code built to show refactoring steps e.g. ListCreationTest
- refactor to abstraction layers
- payload objects could be public fields but that is more vulnerable to app changes
    - xml & json annotations
- api method naming (`createList`) - would be better as `postList` - why?

Code Walkthrough of REST Listicator Automating Examples

- static api vs instantiated api e.g. `ListicatorAPI` singleton
  - readability vs flexibility
- Abstractions can restrict coverage as well as aid it
  - review abstractions to see what is not, and can not be tested with that abstraction code

Resources to learn from Mark Winteringham

http://www.mwtestconsultancy.co.uk/

Mark Winteringham has some useful study material on REST and automating Web Services.

- https://github.com/mwinteringham/api-framework
  - code in different languages and frameworks demonstrating REST API automated execution
- https://github.com/mwinteringham/restful-booker
  - Test Web API
  - live at https://restful-booker.herokuapp.com/
- https://github.com/mwinteringham/presentations
  - Mark's REST Presentations

Bas Dijkstra & James Willett

Resources to learn from Bas Dijkstra

- http://www.ontestautomation.com/open-source-workshops/
  - API REST Assured Code and slides
- http://www.ontestautomation.com/category/api-testing/
  - Bas's Blog posts on API Testing

Resources to learn from James Willett

- https://james-willett.com/tag/rest-assured/
  - blog posts on REST Assured

Resources to learn from Alan Richardson

Github code samples using REST Assured

- https://github.com/eviltester/rest-listicator-automating-examples
- https://github.com/eviltester/tracksrestcasestudy
- https://github.com/eviltester/libraryexamples

Automating and Testing a REST API

- support page (videos) - https://www.compendiumdev.co.uk/page/tracksrestsupport
- book - https://compendiumdev.co.uk/pag/tracksrestapibook

Exercises

- read the resources and learn more about automation and REST Assured
- if you have JDK and Java IDE then download the source for REST Listicator Automating Examples
- run the tests
- add more tests to cover the REST Listicator documentation e.g. users, api keys for authentication, post multiple lists, url parameters etc.
- refactor code as you go to build abstraction layers
- rename existing api methods to match verbs rather than logic
- see exercises section for more ideas

Section - Exercise Time

- Time to experiment
- Explore the HTTP Verbs etc. mention
- Use the tools mentioned
- Test the webservices mentioned
- Explore the exercises in the exercises section

Section - Exercise Time

- Time to experiment
- Explore the HTTP Verbs etc. mention
- Use the tools mentioned
- Test the webservices mentioned
- Explore the exercises in the exercises section

SECTION - SUMMARY with Q & A

Overview of Section - SUMMARY with Q & A

- Any Questions?
- Slide based summary of content
- Final Q&A
- Continue to Experiment

Technology

- Learn HTTP Standards
- You can base your 'bugs' on Standards
  - HTTP Message Syntax and Routing RFC 7230
- Learn the common VERBS: GET, POST, DELETE, PUT
- Read the REST Dissertation

Tools - Clients

- Different tools have different capabilites
- Experiment with multiple tools
- Postman: Collections for Data Creation, Console
- Insomnia: Import, Timeline, Proxies
- Import/Export between Tools

Tools - Proxies

- Often used for Security Testing
- Fuzzers create data
- Automatically keep a record of your testing
- View actual requests and responses
- Replay requests

Tools

- Clients
  - Postman
  - Insomnia
  - cURL
- Proxies
  - System
    - Fiddler
    - Charles
  - Other
    - BurpSuite
    - Owasp Zap

Automating

- HTTP libraries
- REST libraries
- Domain Abstractions
- Reuse for performance testing

Testing

- Requirements - domain, documentation, sdk
- Standards - HTTP, REST, Auth
- Security
- Capacity
- Interfacing Systems

Q & A

BIO - Alan Richardson

Alan Richardson has more than twenty years of professional IT experience, working as a programmer and at every level of the testing hierarchy from tester through head of testing. He has performed keynote speeches and tutorials at conferences worldwide. Author of multiple books on testing and automating. Alan also has created online training courses to help people learn Technical Web Testing and Selenium WebDriver with Java. He works as an independent consultant, helping companies improve their use of automation, agile, and exploratory technical testing.

Alan Richardson (www)

- www.compendiumdev.co.uk
- www.eviltester.com
- www.seleniumsimplified.com
- www.javafortesters.com
- uk.linkedin.com/in/eviltester
- testerhq.com - Aggregated Feed

Follow

- Linkedin - uk.linkedin.com/in/eviltester
- Twitter - twitter.com/eviltester
- Instagram - instagram.com/eviltester
- Facebook - facebook.com/eviltester
- Youtube - youtube.com/user/EviltesterVideos
- Pinterest - uk.pinterest.com/eviltester
- Github - github.com/eviltester
- Slideshare - slideshare.net/eviltester

Books Written By Alan Richardson

- Java For Testers
- Dear Evil Tester
- Automating and Testing a REST API
- Selenium Simplified

Online Training Courses By Alan Richardson

- Selenium Webdriver With Java
- Technical Web Testing 101
- Evil Tester Talks: Technical Testing
- Case Study: Java Desktop Application Technical Training

Section - Exercises

There are exercises dotted throughout the slides but this section has the long form exercises that you can work through at your own pace in the exercise sections and after the tutorial when you want to continue working.

The exercises are ordered to basically match the tutorial structure, but feel free to do then in any order - skip the early exercises if you are more experienced. There is enough here for people of any level of experience.

Exercises for SECTION - HTTP BASICS

Exercise: View Browser Requests in Dev Tools Network Tab

Aims:

- Learn to use the Dev Tool in Web Browsers to view traffic.
- View the HTTP requests and HTTP responses
- Look up any headers or information that you don't understand via web search
- Suggested sites: Twitter, Instagram, Gmail, Ebay
- Identify more sites

Use this as an ongoing exercise during personal and work day to learn dev tools and HTTP

Exercises for SECTION - WEB API BASICS

Exercise: Install an HTTP GUI Client

Aims: install and use an HTTP REST GUI Client

Install either:

- Postman GetPostman.com
- Insomnia insomnia.rest

Call a 'mock' webservice

- GET http://compendiumdev.co.uk/apps/api/mock/heartbeat
- GET https://swapi.co/api/people/1
- GET http://jsonplaceholder.typicode.com/users/1/todos

see swapi.co, jsonplaceholder.typicode.com

Exercise: Use Swapi.co Web Application which uses API

Aims: explore a public API example

- https://swapi.co
- make a request from the GUI, use network tab to view requests
- read the https://swapi.co/documentation
- use an HTTP client to issue requests e.g. Postman, cURL, Insomnia.rest
- try to generate different HTTP Status codes: 200, 400, 404, 405
- experiment with different HTTP Verbs: GET, POST, OPTIONS, HEAD

Exercise: Explore HTTP Verbs with JsonPlaceHolder MOCK Web Service

Web service does not 'persist' data, but will reflect back changes in response e.g. a POST will not amend data on server, but the response will show you what would have happened if it had.

Using: jsonplaceholder.typicode.com

- Experiment with the Resources, and Routes. (see next slide for more hints)

Exercise: Use JsonPlaceHolder MOCK Web Service for `GET` requests

- GET all posts `/posts`
- GET a single post `/posts/1`
- GET comments for a post `/posts/1/comments`
- GET comments for a post using Query string `/comments?postid=1`
- Experiment with the Resources, and Routes listed on jsonplaceholder.typicode.com

For all parameterised queries - try others e.g. amend `1` to something else

Exercise: Use JsonPlaceHolder MOCK Web Service for `POST`, `PUT`, `DELETE` requests

- use POST to create a post `/posts` (hint, use GET to find format, don't add an 'id')
- use POST to amend a post `/posts` (hint, add an `id`)
- DELETE a single post `/posts/1`
- create a post with PUT `/posts/1`

Any bugs when you use the above?

Experiment - any bugs?

Exercise: Use examples file for JsonPlaceHolder

- Insomnia.rest Workspace
  - import `/exercises/insomnia/workspaces` the 'jsonplaceholder...' file
- Postman Collection
  - import `/exercises/postman/collections` the 'jsonplaceholder...' file
  - import shared collection from
  - https://www.getpostman.com/collections/2c7270ffbd075f03e33d

Exercises for SECTION - HTTP REQUESTS AND RESPONSES

Exercise: Install Test Java REST API - REST Listicator

- http://compendiumdev.co.uk/downloads/apps/restlisticator
  - /v1/rest-list-system.jar
  - /v1/documentation.html
- read the documentation
- download the `.jar` file

Exercise: Run Test Java REST API - REST Listicator

- in the directory you downloaded it to type:
  - `java -jar rest-list-system.jar`

```
>java -jar rest-list-system.jar
User : superadmin - ea382383-e888-4e7c-b569-5c0fdc6b91d2
User : admin - 96244a45-e0c5-40e7-ac3f-2351ab5f7947
User : user - 08e7f6f5-c111-401b-9366-bd86fc755d1b
SLF4J:Failed to load class"org.slf4j.impl.StaticLoggerBinder"
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder
for further details.
Running on 4567
```

Exercise: Check app is running

- Open http://localhost:4567 in browser

Should see the documentation

- GET localhost:4567/heartbeat

Should return 200

Exercise: Read the documentation and issue GET requests

- GET the list of users
- GET the list of Lists
- Change `Accept` header to GET list in XML and in JSON
- What happens if you GET `/logs`? Is that response correct?

Exercise: Use parameters to filter User list

- GET '/users' lists some filters that can be applied to username
- Search for partial match on "r"
- Search for users with "admin" in their username
- Search for users with an absolute name e.g. `user`

Exercise: Read the HTTP standard documentation to learn about verbs

HTTP Verbs are described in the W3c Standard and IETF standard

Standard

- GET
- POST
- DELETE

When you test or use the system - does your observation match the standards?

Exercise: Basic Auth Header

- Postman 'Authorization' tab
- Insomnia 'Auth' tab
- GET `/users/admin` when authenticated as `user:password`
- GET `/users/admin` when authenticated as `admin:password`

Did you see a difference?

Some functions require admin privileges later.

Exercise: Use POST to Create a List

- default `user` password is `password`
- minimum information for a list is `title` e.g. `{title:"this is a title"}`
- Basic Auth required for a `POST` request to `/lists`

How do you know it created the list?

Exercise: Use DELETE to Delete a list

- DELETE a 'list' that you created using the `guid`

hint: `user` *cannot delete*

Exercise: Use POST to Amend a List

- default `user` password is `password`
- minimum information for a list is `title` e.g. `{title:"this is a title"}`
- Basic Auth required for a `POST` request to `/lists/{guid}`

How do you know it amended the list?

Can you amend all the fields? `title`, `description`, `guid`, `createdDate`, `amendedDate`

Exercise: Status Codes

Send requests to deliberately trigger the following status codes:

- `200 OK`
- `201 Created`
- `204 No Content`
- `400 Bad Request`
- `401 Unauthorized`
- `403 Forbidden`
- `404 Not Found`
- `405 Method Not Allowed`
- `409 Conflict`

Bonus points for a `5xx` error e.g. `500`

Exercise: Explore the documentation and test the basics

- read the documentation
- create more lists
- amend lists
- delete lists
- switch between users
- create and amend users
- test the GET, POST, DELETE
- test the access permissions

Exercise: Explore Postman

- create a collection
- use environment variables e.g. `{{host}}` instead of `localhost:4567`
- share a collection
- download the `exercise_data_files.zip` from workshop page
    - import the shared collection for RestListicator
    - import the shared collection for JsonPlaceHolder

Exercise: Explore Insomnia

- create a workspace
- use environment variables e.g. `{'host':'localhost:4567'}`
- download the `exercise_data_files.zip` from workshop page
  - import the shared collection for RestListicator
  - import the shared collection for JsonPlaceHolder

Exercises for SECTION - MORE HTTP REQUESTS AND RESPONSES

Exercise: Use OPTIONS to explore capabilities of `/lists`

- OPTIONS

hint: check the headers

Exercise: PUT

- Use `PUT` to create a new 'list'
- Use `PUT` to amend a list
- Does `PUT` work with multiple lists?
- Does `PUT` work with users?
- Test the `PUT` end points with different payloads

Exercises for SECTION - REST API BASICS

Exercises: HEAD

- HEAD
  - does `HEAD` implementation match the standard?
  - compare `HEAD` to `GET` is it the same information?
  - which parts of the API respond to `HEAD`?
  - does the documentation match the implementation?

Exercises: PATCH

- PATCH is implemented just like `post`
  - does it comply with the JSON Merge Patch standard?
- The implementation does not comply with the JavaScript Object Notation (JSON) Patch standard
  - read the JSON Patch standard
- The implementation does not comply wiht the XML Patch Operations standard
  - read the XML Patch Standard

Exercise: Create a bunch of lists using Postman Collection Runner

- create a new collection with single request to create a list e.g.
  - POST http://localhost:4567
  - Basic HTTP Authentication for `admin:password`
  - body `{title:"{{title}}"}`

Then create csv file:

```
title
a title
another title
```

User the `Collection \ Runner` to run the collection with the data file

Exercises for SECTION - TESTING WEB SERVICES

Exercise: Think through the Testing of the REST API

- Risks? Architecture?, Database?, Capacity (load, multiuser)?
- Functionality? are requirements met?
- Data? what data do you have to explore in the requests?
- Request Capabilities? e,g, verbs, formats, headers, etc.
- ?

Exercise: Start REST Listicator Web Service with Bugs Enabled

```
java -jar rest-list-system.jar -bugfixes=false
```

- The system has been coded with some known bugs
- these are all fixed by default.
- start with `-bugfixes=false` to have known bugs
- See if you can find them

Exercises for SECTION - PROXIES

Exercise: Use Insomnia through a Proxy

- Start proxy
- Identify Listening Port of proxy
- Application \ Preferences
- [x] Enable Proxy
- type proxy details into `HTTP Proxy` field e.g. `localhost:8888`
- use Insomnia and see requests in Proxy
- replay requests from Proxy

Exercise: Use Postman through a System Proxy

- on Windows - Charles or Fiddler
- on Mac - Charles
- start proxy
- start Postman
- in Postman `File \ Settings \ Proxy` ensure 'User System Proxy' is 'on'

Exercise: Use Postman through a non-system proxy

For full details see blog post

- Mac (type it all on one line):

```
open /Applications/Postman.app --args
--proxy-server=localhost:8888
```

- Windows:

```
cd C:\Users\Alan\AppData\Local\Postman\app-4.9.3\
postman.exe --proxy-server=localhost:8888
```

Exercise: Create/Amend Data with Proxy Fuzzers

- use the proxy to create lists test data
- use the proxy to create users
- experiment with other requests
- experiment with different proxies and fuzzer payloads

Exercises for SECTION - AUTOMATING - review resources

- read the resources and learn more about automating and REST Assured

Exercises for SECTION - AUTOMATING - run the tests

- download the code from GitHub for REST Listicator Automating Examples
- you need a JAVA SDK to run the tests
- load the code into IntelliJ
- run the tests

Exercises for SECTION - AUTOMATING - review the code

- read and understand the tests and abstraction layer
- review the tests for coverage
  - what is missing?
- what does the abstraction layer prevent you from doing?

Exercises for SECTION - AUTOMATING - expand the tests

- add more tests to cover the REST Listicator documentation
  - e.g. users, api keys for authentication, post multiple lists, url parameters etc.
- refactor code as you go to build abstraction layers
- rename existing api methods to match verbs rather than logic

Section - TEST THE APIS

- run through the exercises you haven't done
- explore the REST Listicator with further test ideas
- explore the capabilities of the tools - Postman, Insomnia, Proxies
- see next section for more exercise ideas

Exercises for SECTION - HOUR OF POWER Exercise suggestions

- use the REST Listicator in buggy mode
- compare results with REST Listicator in non-buggy mode
- can you test against the documentation?
- can you create users?
- can you create multiple users?
- can you create multiple lists?
- can you amend lists?

More Exercises for SECTION - HOUR OF POWER Exercise suggestions

- can you delete lists?
- can you filter lists?
- can you filter fields on `/list` and `/user`?
- can you amend a user?
- can you amend your own user details?
- explore tool capabilities
- feed through proxies

Additional and Advanced Exercises

Exercise: Explore RESTful Booker

RESTful Booker written by Mark Winteringham

- Read the Documentation
  - restful-booker.herokuapp.com
- Test/Explore the application using Postman/Insomnia/Proxies
- Read the Code
  - github.com/mwinteringham/restful-booker

Exercise: Install and use cURL

- download using the wizard
- read the docs
- repeat JsonPlaceHolder exercises using cURL
- export cURL from Postman and Insomnia
- use cURL to GET normal web pages

Exercise: Use cURL Requests in Postman and Insomnia

Postman:

- click import, then "Paste Raw Text" then paste in a cURL command

Insomnia:

- copy and paste a cURL command into the URL field

Public WebServices to experiment with

Most public APIs are going to be `GET` requests.

Lists of APIs:

- any-api.com
- github.com/toddmotto/public-apis

Mock Web Servers to use for experimenting

- Read the docs for the mock web server
- Send through requests and call the server, does it work as expected?

https://jsonplaceholder.typicode.com/

Experiment with Swapi.co

- http://swapi.co : `GET`, `HEAD` and `OPTIONS` only

Performance Testing

- How would you performance test the API?
- JMeter? Gatling?
- Can you performance test the web service?
- **Make sure you use an isolated/local environment to test on**

Suggested Applications to Test

- Tracks
  - Download a VM from Turnkey and test the Tracks API