Introduction

We can read on-line about the "Test Automation Pyramid" and we can also learn that "GUI Automation is brittle" and "we should test under the GUI". Fine. But how many in-depth examples can you find? Examples that show you how to automate quickly, and how to improve on that initial 'quick fix'?

That's what this case study is all about - an example of automating an application without using GUI based libraries.

This case study will show how you can add value to a process quickly with fairly crude and "hacky" code, and then how you can change that code to make it better.

Throughout the case study, I'm not just going to tell you how I did it. I'm going to explain why, and what I could have done differently. Why I made the decisions I made, because then you can try different approaches and build on this case study.

Since this is a case study, and not a 'step by step' course. I assume some basic knowledge:

- You know how to install Java and an IDE,
  - if you don't then the Starter Page[2] on Java For Testers will help.
- You have some basic Web experience or HTTP knowledge,
  - if not then my Technical Web Testing 101[3] course might help or my YouTube channel[4].

I'll cover some of the above topics, although not in depth. If you get stuck you can use the resources above or contact me[5].

The background behind this case study is that I've used Tracks as an Application Under Test in a few workshops, for practising my own testing, and to improve my ability to automate.

For the workshops I built code to create users and populate the environment with test data. I found that people like to learn how to do that, and I realised during the workshops that I also approach this differently to other people.

[2] http://javafortesters.com/page/install/
[3] http://compendiumdev.co.uk/page.php?title=techweb101course
[4] http://eviltester.com/youtube
[5] http://compendiumdev.co.uk/contact

I didn't automate under the GUI because I follow a "Test Automation Pyramid". I automated beneath the GUI:

- because it is fast,
- because we can do things we can't do through the GUI.

By 'Under the GUI' I mean:

- Using the API (Application Programming Interface).
- Using the 'APP as API',
  - sending through the HTTP that the GUI would have sent, but not using the GUI.

I explain 'App as API' in the case study later, and show examples of it in practice. I realised, during the teaching of this stuff, that most people don't automate in this way.

For most people testing 'under the GUI' means API. To me it means working at the different system communication points anywhere 'under' the GUI. I explain this in the case study as well.

Working under the GUI isn't always easier. In this case study you'll see that working through the GUI would have been 'easier'. I wouldn't have had to manage cookie sessions and scrape data off pages.

But working beneath the GUI is faster, once it is working, and arguably is more robust - but we'll consider that in more detail in the case study and you'll see when it isn't.

You'll see initial code that I used for Tracks 2.2 and then updated for 2.3, I'll walk you through the reasons for the changes and show you the investigation process that I used and changes I made.

If you haven't automated an HTTP application below the GUI before then I think this case study will help you learn a lot, and you'll finish with a stack of ideas to take forward.

If you have automated the GUI before. I think you'll enjoy learning why I made the decisions I made, and you'll be able to compare them with the decisions you've made in the past. I think, after finishing the case study, you might expand the range of decisions you have open to you in the future.

Introduction to APIs

This chapter will provide an introduction to the concept of an API (Application Programming Interface) and concentrates on Web or Internet accessible APIs. I will explain most of the concepts that you need to understand this book and I will start from the very basics. Most of this chapter will be written colloquially for common sense understanding rather than to provide formal computer science definitions.

You can probably skip this chapter if you are familiar with Web Services, URI and URL, HTTP, HTTP Status Codes, JSON and XML.

Also, because this chapter covers a lot of definition, feel free to skip it if you want to get stuck in to the practical aspects. You can always come back to this chapter later if you don't understand some basic terminology.

First I'll start by saying that we are going to learn how to test Web Applications. i.e. Applications that you can access over a network or the internet without installing them on your machine.

The Web Application we will test has an API, which is a way of accessing the application without using a Web Browser or the application's Graphical User Interface (GUI).

What Is a Web Application?

A Web Application is a software application that is deployed to a Web Server. This means the application is installed onto a computer and users access the application via a Web Browser.

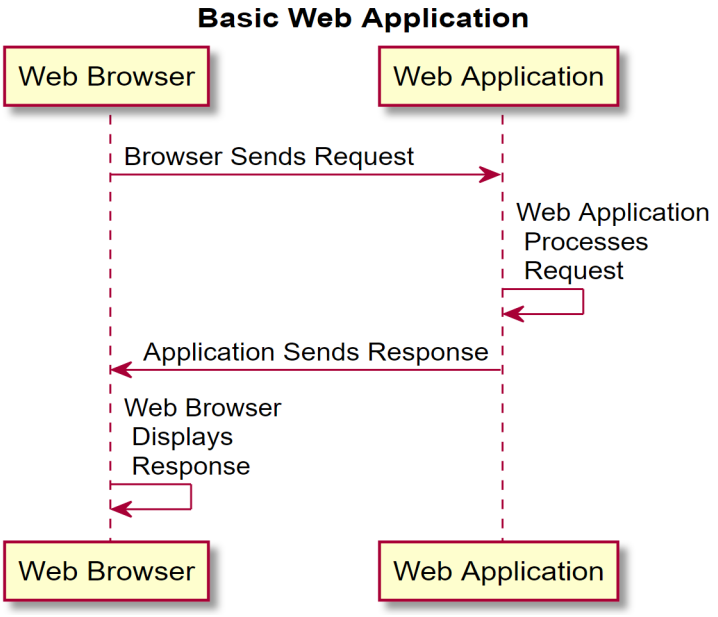
Google Is an Example of a Web Application

`google.com` is an example of a Web Application. A user visits `google.com` in a Browser and the application's Graphical User Interface (GUI) is displayed in the Browser. The GUI consists of a search input field which the user fills in and then clicks a button to search the Web.

When the user clicks the search button, the Browser makes a request to the Web Application on the Web Server to have the Google Search Application make the search and return the results to the user in the form of a web page with clickable links.

Basically,

- Web Browser -> Sends a Request to -> Web Application
- Web Application -> Processes Request and Sends a Web Page as Response to -> Web Browser



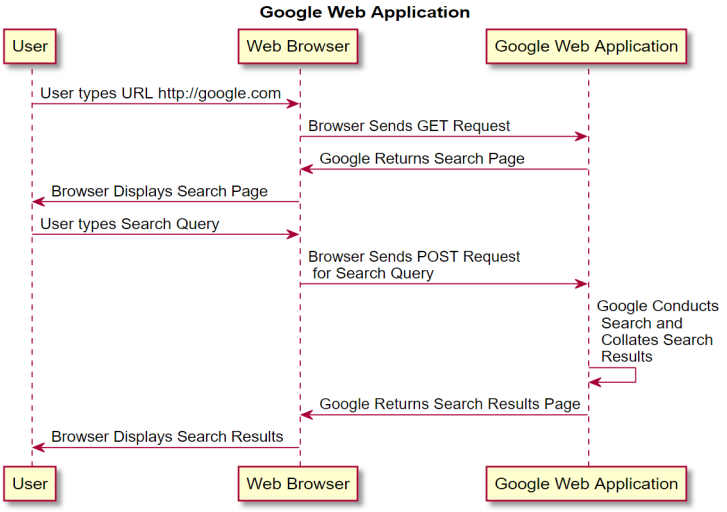
The requests that the Browser sends to the Web Server are HTTP requests. HTTP requests are a way of sending messages between machines over the Internet. Think of HTTP as the format of the message that Browser and Web Server send to each other.

When we first visit Google in a Browser we type in the URL or address for Google. i.e. `https://google.com`

The Browser then sends a type of HTTP request to Google called a `GET` request to 'get', or retrieve, the main search form. Google Web Application receives the request and replies with an HTTP response containing the HTML of the search page. HTML is the specification for the Web Page so the Browser knows how to display it to the user.

When the user types in a search term and presses the search button. The Browser sends a `POST` request to Google. The `POST` request is different from the `GET` request because it contains the details of the search term that the user wants the Google Web Application to search for.

The Google Web Application then responds with an HTTP response that contains the HTML containing all the search results matching the User's search term.



Google is an example of a Web Application with a GUI, and because the user accesses the Web Application through a Browser they are often unaware of the HTTP requests, or that different types of HTTP requests are being made.

When we test HTTP APIs we have to understand the details of HTTP requests.

What Is an API?

An API is an Application Programming Interface. This is an interface to an application designed for other computer systems to use. As opposed to a Graphical User Interface (GUI) which is designed for humans to use.

APIs come in many different forms with and technical implementations but this book concentrates on HTTP or Web APIs.

An HTTP based API is often called a Web API since they are used to access Web Applications which are deployed to Servers accessible over the Internet.

Applications which are accessed via HTTP APIs are often called Web Services.

Mobile Applications often use Web Services and REST APIs to communicate with servers to implement their functionality. The Mobile Application processes the message returned from the Web Service and displays it to the User in the application GUI. So again, the user is unaware that HTTP requests are being made, or of the format of the requests and responses.

What Is an HTTP Request?

HTTP stands for Hypertext Transfer Protocol and is a way of sending messages to software on another computer over the Internet or over a Network.

An HTTP request is sent to a specific URL and consists of:

- a VERB specifying the type of request e.g. GET, POST, PUT, DELETE
- A set of HTTP Headers. The headers specify information such as the type of Browser, type of content in the message, and what type of response is accepted in return.
- A body, or payload in the request, representing the information sent to, or from, the Web Application. Not all HTTP messages can have payloads: POST and PUT can have payloads, GET and DELETE can not.

HTTP requests are text based messages and over the course of this Case Study you will learn to read them e.g.

```
GET http://compendiumdev.co.uk/apps/mocktracks/projectsjson.php HTTP/1.1
Host: compendiumdev.co.uk
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
 AppleWebKit/537.36 (KHTML, like Gecko)
 Chrome/59.0.3071.115 Safari/537.36
Accept: text/html,application/xhtml+xml,
 application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
```

The above HTTP request is a GET request, which is a READ request:

- to read the page from the URL
  - `http://compendiumdev.co.uk/apps/mocktracks/projectsjson.php`
- request is made from the Chrome Browser version 59. You can see this in the 'User-Agent' header. Yes, the header also mentions 'Safari', 'AppleWebKit' and 'Mozilla', this is for various reasons of backwards compatibility, but it was sent from Chrome version 59. For more information on `User-Agent` visit useragentstring.com[6].

[6] http://www.useragentstring.com

What Is a URL?

URL is a Uniform Resource Locator and is the address we use to access websites and web applications.

When working with APIs you will often see this referred to as a URI (Uniform Resource Identifier).

Think of a URI as the generic name for a URL.

When we want to call an HTTP API we need the URL for the `endpoint` we want to call e.g

```
http://compendiumdev.co.uk/apps/mocktracks/projectsjson.php
```

This is the locator that says "I want to call the `apps/mocktracks/projectsjson.php` resource located at `compendiumdev.co.uk` using the `http` protocol".

For the purposes of this book I will use the phrase URL, but you might see URI mentioned in some of the source code. I use URL because the locator contains the protocol or *scheme* required to access it (`http`).

The above URL can be broken down into the form:

```
scheme://host/resource
```

- scheme - `http`
- host - `compendiumdev.co.uk`
- resource - `apps/mocktracks/projectsjson.php`

A larger form for a URL is:

```
scheme://host:port/resource?query#fragment
```

I didn't use a `port` in the URL, for some applications you might need to.

By default `http` uses port `80`, so I could have used:

```
http://compendiumdev.co.uk:80/apps/mocktracks/projectsjson.php
```

Also I haven't used a query because this endpoint doesn't need one.

The `query` is a way of passing parameters in the URL to the endpoint e.g. Google uses query parameters to define the search term and the page:

```
https://www.google.co.uk/?q=test&start=10#q=test
```

- scheme - `https`
- host - `www.google.co.uk`
- query - `q=test&start=10`
- fragment - `q=test`

The query is the set of parameters which are key, value pairs delimited by '&' e.g. `q=test` and `start=10` ("start" is a key, and "10" is the value for that key).

When working with APIs it is mainly the scheme, host, port and query that you will use.

You can learn more about URL and URI online[7].

What Are HTTP Verbs?

A Web Browser will usually make `GET` requests and `POST` requests.

- `GET` requests ask to read information from the server e.g. clicking on a link.
- `POST` requests supply information to the server e.g. submitting a form.

`GET` requests do not have a body, and just consist of the Verb, URL and the Headers.

`POST` requests can have a payload body e.g.

```
POST http://www.compendiumdev.co.uk/apps/mocktracks/reflect.php HTTP/1.1
Host: www.compendiumdev.co.uk
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
 AppleWebKit/537.36 (KHTML, like Gecko)
 Chrome/59.0.3071.115 Safari/537.36
Accept: text/html,application/xhtml+xml,
 application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8

{"action":"post"}
```

When working with a Web Application or HTTP API the typical HTTP Verbs used are:

- GET, to read information.

[7] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

- POST, to create information.
- PUT, to amend or create information.
- DELETE, to delete information, this is rarely used for Browser accessed applications, but often used for HTTP APIs.

POST and PUT requests would usually have a message body. GET and DELETE would not.

HTTP Verbs are described in the W3c Standard[8] and IETF standard[9].

What Is an HTTP Response?

When you issue an HTTP Request to the server you receive an HTTP Response.

The response from the server tells you if your request was successful, or if there was a problem.

```
HTTP/1.1 200 OK
Date: Fri, 30 Jun 2017 13:50:11 GMT
Connection: close
Content-Type: application/json

{
  "projects": {
    "project": [
      {
        "id":  1,
        "name": "A New Project",
        "position": 0,
        "description": "",
        "state": "active",
        "created-at": "2017-06-27T12:25:26+01:00",
        "updated-at": "2017-06-27T12:25:26+01:00"
      }
    ]
  }
}
```

The above response has:

[8] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
[9] https://tools.ietf.org/html/rfc7231

- A status code of 200, which means that the request was successful.
- A `Content-Type` header of `application/json` which means that the body is a JSON response.
- A body which contains the actual payload response from the server.

What Is an HTTP Status Code?

Web Services and HTTP APIs use HTTP Status Codes to tell us what happened when the server processed the request.

The simple grouping for HTTP Status Codes is:

- 1xx - Informational
- 2xx - Success e.g. 200 Success
- 3xx - Redirection e.g. 302 Temporary Redirect
- 4xx - Client Error e.g. 400 Bad Request, 404 Not Found
- 5xx - Server Error e.g. 500 Internal Server Error

The type of status code you receive depends on the application you are interacting with. Usually a 4xx error means that you have done something wrong and a 5xx error means that something has gone wrong with the application server you are interacting with.

You can learn more about status codes online:

- Wikipedia List[10]
- HTTP Statuses[11]

What Are Payloads?

A Payload is the body of the HTTP request or response.

When browsing the Web, the Browser usually receives an HTML[12] payload. This is the web page that you see rendered in the Browser.

Typically when working with an HTTP API we will send and receive JSON or XML payloads.

You saw JSON payloads in the examples above.

[10] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
[11] https://httpstatuses.com/
[12] https://en.wikipedia.org/wiki/HTML

What Is JSON?

JSON stands for JavaScript Object Notation and is a text representation that is also valid JavaScript code.

```
{
  "projects": {
    "project": [
      {
        "id":  1,
        "name": "A New Projectaniheeiadtatd",
        "position": 0,
        "description": "",
        "state": "active",
        "created-at": "2017-06-27T12:25:26+01:00",
        "updated-at": "2017-06-27T12:25:26+01:00"
      }
    ]
   }
}
```

JSON can be thought of as a hierarchical set of key/value pairs where the value can be:

- Object - delimited by `{` and `}`.
- Array - delimited by `[` and `]`.
- String - delimited by `"` and `"`.
- Integer

An array is a list of objects or key/value pairs.

The keys are String values e.g. “projects”, “project”, “id”, etc.

What Is XML?

XML stands for Extensible Markup Language.

HTML is a variant of XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<projects type="array">
  <project>
      <id type="integer">1</id>
      <name>A New Projectaniheeiadtatd</name>
      <position type="integer">0</position>
      <description nil="true"/>
      <state>active</state>
      <created-at type="dateTime">2017-06-27T12:25:26+01:00
           </created-at>
      <updated-at type="dateTime">2017-06-27T12:25:26+01:00
           </updated-at>
      <default-context-id type="integer" nil="true"/>
      <completed-at type="dateTime" nil="true"/>
      <default-tags nil="true"/>
      <last-reviewed type="dateTime" nil="true"/>
  </project>
</projects>
```

XML is constructed from nested elements

- An element has an opening and closing tag e.g. `<state>` and `</state>`.
  - The tag has a name i.e. `state`.
  - The opening tag begins with `<` and ends with `>` e.g. `<state>`.
  - The closing tag begins with `</` and ends with `>` e.g. `</state>`.
- An element has a value, which is the text between the tags e.g. the `state` element has a value of `active`.
- An element can have attributes, these are always within the opening tag e.g. the `id` element (`<id type="integer">`) has an attribute named `type` with a value of `"integer"`.
- Elements can contain other Elements. These are called Nested Elements. e.g. the `projects` element has a nested element called `project`.

For XML to be valid, it must be well formed, meaning that every opening tag must have a corresponding closing tag, and strings must have an opening and closing quote.

Some elements do not have a closing tag, these are self closing. The opening tag, instead of ending with `>` actually ends with `/>` you can see this in the `<description nil="true"/>` element.

What Are HTTP Headers?

HTTP messages have the Verb and URL, followed by a set of headers, and then the optional payload.

```
POST http://www.compendiumdev.co.uk/apps/mocktracks/reflect.php HTTP/1.1
Host: www.compendiumdev.co.uk
Content-Type: application/json
Accept: application/json

{"action":"post"}
```

The headers are a set of meta data for the message.

Headers are a name, followed by `:`, followed by the value of the header.

The above HTTP message example has three headers:

- Host
- Content-Type
- Accept

The `Host` header defines the destination server domain name.

The `Content-Type` header tells the server that the content of this message is JSON.

The `Accept` header tells the server that the client (application sending the message) will only accept response payloads represented in JSON.

There are many headers available[13] for configuring the Authentication details, length of message, custom meta data, cookies etc.

What Is Authentication?

When we send a message to a server we might need to be authenticated i.e. authorised to send a message and receive a response.

For many Web Applications you authenticate yourself in the application by logging in with a username and password. The same is true for Web Services or HTTP APIs.

[13] https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

If you are not authenticated and try to send a message to a server then you are likely to receive a response from the server with a 4xx status code e.g.

- 401 Unauthorized
- 403 Forbidden

There are many ways to authenticate HTTP requests for HTTP APIs.

Some common approaches you might encounter are:

- Custom Headers
- Basic Authentication Headers
- Session Cookies

Some HTTP APIs require **Custom Headers** e.g.

```
POST http://www.compendiumdev.co.uk/apps/mocktracks/reflect.php HTTP/1.1
X-APPLICATION_KEY: asds-234j-werw
```

Here the `X-APPLICATION-KEY` header has a secret value which authenticates the request.

Basic Authentication Headers are a standard approach for simple login details:

```
POST http://www.compendiumdev.co.uk/apps/mocktracks/reflect.php HTTP/1.1
Authorization: Basic Ym9iOmRvYmJz
```

The `Authorization` header specifies `Basic` authentication and is followed by a base64[14] encoded string.

- "`Ym9iOmRvYmJz`" is the base64 encoded version of the string "`bob:dobbs`"
- In Basic Authentication the string represents `username:password`

Session Cookies[15] are set by a server in a response message and are represented in a `Cookies:` header.

[14] https://en.wikipedia.org/wiki/Base64
[15] https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

What Is REST?

REST stands for Representational State Transfer, and while it has a formal definition, which you can read in Roy Fielding's PHd thesis[16], it very often means that the API will respond to HTTP verbs as commands.

e.g.

- GET, to read information.
- POST, to create information.
- PUT, to amend information.
- DELETE, to delete information.

The documentation for the particular system you are testing will describe how the API has interpreted REST if they have described their API as a REST API.

What Tools Are Used for Accessing an API?

Since API stands for Application Programming Interface, we might expect all interaction with the API to be performed via program code. But it really implies that the interface is well documented and does not change very often.

Also that the input and output from the API are designed for creation and consumption by code - hence the use of formats like JSON and XML.

We can issue API requests from a command line with tools like cURL, which you will see later in this book.

Also GUI tools like Postman, which we cover in a later chapter, allow humans to easily interact with APIs.

When writing application code to interface with an API we are generally able to use a library for the specific programming language that we are working with.

In this book we are using Java and will use the REST Assured library.

[16] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Example APIs

If you want a very simple API to experiment with at the moment, then I recommend the Star Wars API Web Application.

- swapi.co[17]

This is a very simple API that mainly uses GET requests and returns information about Star Wars characters and planets.

The API is well documented and has an online GUI that you can use to experiment with the API.

Recommended Reading

The REST API Tutorial[18] provides a good overview of REST APIs, HTTP Status codes and Verbs.

Summary

This chapter provided a very high level description of an API (Application Programming Interface) to differentiate it from a GUI (Graphical User Interface). An API is designed to be used for systems and applications to communicate, whereas a GUI is designed for humans to use.

Humans can use API interfaces. Tools such as cURL and Postman can help. We also have the advantage that for HTTP APIs, the messages are in text format and usually contain human readable JSON or XML format payloads.

The status codes that are returned when requests are sent to an API help you understand if the request has been successful (200, 2xx), or if there was something wrong with the request (4xx), or if something went wrong in the server application (5xx).

At the very least, you should now be familiar with the terms we will be using in this case study, and have seen some examples of HTTP messages for both requests and responses.

[17] https://swapi.co
[18] http://www.restapitutorial.com/