Chapter 1

Background

1.1 State of the Internet

In 1989, Tim Berners-Lee proposed a communication system for CERN[7]. He soon realized the concept could be expanded, and in 1990 he published the proposal for what would become the World Wide Web[6]. The document suggested using 'hypertext'[1] "to link and access information of various kinds as a web of nodes in which the user can browse at will". HTTP was defined as a protocol to exchange hypertext between devices[8], and has been the basis of the World Wide Web ever since.

However, while the HTTP standard has remained constant, the web has been continuously evolving over the past decade. Berners-Lee envisioned web pages consisting of static data and hyper-links[2] to other web pages. This was originally the case, with web sites consisting of text and images, coded using HTML and JavaScript. As the computing power available to general users increased, more dynamic sites started appearing following the client-server model[3] to allow more user content and interaction[4]. Web Browser developers also released new, faster, JavaScript engines[5][6] which allowed even more interactive web pages. Of particular note was the rise of AJAX[7], which allowed websites to transition from page based documents to single

[1]Structured text that uses links between nodes containing text.

[2]A reference to data on a web document.

[3]The server serving web pages generates custom HTML documents on the fly, and sends them to the client web browser.

[4]Coined 'Web 2.0'.

[5]Browser JavaScript Performance in 2008: `http://www.cnet.com/news/speed-test-google-chrome-beats-firefox-ie-safari/`

[6]Browser JavaScript Performance in 2010: `http://www.pcgameshardware.com/aid,687738/Big-browser-comparison-test-Internet-Explorer-vs-Firefox-Opera-Safari-and-Chrome-Update-Firefox-35-Final/Reviews/`

[7]Asynchronous JavaScript and XML: A group of web techniques that allow data to be send to and received from a server asynchronously, thus allowing parts of web pages to be updated without having to fetch an entire new page.

page 'web apps'.

1.1.1 The HTTP Protocol

HTTP/1.1 is currently the most widely used protocol for data communications on the Internet[8]. HTTP functions as a request-response protocol: A client submits an HTTP request to a server, which returns resource as a response. The HTTP/1.1 specification defines 9 types of requests[9]:

Verb	**Description**
GET	Requests a representation of the specified resource.
HEAD	Asks for the response identical to the one that would correspond to a GET request, but without the response body.
POST	Asks the server to add the enclosed entity as a subordinate to the specified resource.
PUT	Asks the server to add the enclosed entity to the specified resource.
DELETE	Deletes the specified resource.
PATCH	Partially modification to a resource.
TRACE	Echoes back the received request so that a client can see what (if any) changes or additions have been made by intermediate servers.
OPTIONS	Returns the HTTP methods that the server supports for the specified URL
CONNECT	Converts the request connection to a transparent TCP/IP tunnel, generally used for SSL-encrypted communication.

These requests are ubiquitously implemented in major web browsers and servers, allowing for a standardised way to build web apps.

1.1.2 Web APIs

Web APIs are built upon HTTP, and are used for many purposes around the Internet. One of the more common ones is to allow the front-end of a web app to communicate with the server, and fetch dynamic content. While there are no official standards for web APIs, there are two common architecture styles used when designing them.

[8]HTTP/2 is currently under development but has not yet been finalised.
[9]Referred to as 'verbs'.

SOAP

Originally an acronym for Simple Object Access protocol, SOAP[10] is a protocol specification regularly used for APIs. It is based on XML, and consists of three parts:

1. An envelope which defines the message structure and how to process it.
2. Rules for encoding data types.
3. Convention for representing procedure calls and responses.

A SOAP message is an XML document consisting of:

Element	**Description**
Envelope	Identifies the document as a SOAP message.
Header	Contains header information.
Body	Contains the call and response information.
Fault	Contains information about any errors that occurred.

SOAP's main strengths lie in security (due to supporting SSL encryption) and reliability (any errors are documented in the Fault packet). However, due to the complex structures required for a SOAP API to function, it is generally not used outside of enterprise applications.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Listing 1: Example SOAP message

Representational State Transfer

Representational State Transfer(REST) is a software architecture style based on a series of guidelines for creating scalable web services[9]. The formal REST constraints are:

Constraint	**Description**
Client-server	An interface separates the client and server, allowing separation of concern and more portable code.
Stateless	No client context is stored on the server; each request contains all the information required to service it.
Cacheable	Responses must define whether they are cacheable.
Layered System	A client can not tell what is servicing the request; there may be intermediaries to improve scalability.
Code on Demand	(Optional) Servers can transfer executable code to the client to extend or modify functionality.
Uniform Interface	**Uniform Interface:** Individual resources are identified in the request. Furthermore, the resource internal representation may be separate from the representation returned to the client.
	Manipulation of resources through these representations: If a client has a representation of a resource, it can modify or delete it.
	Self-descriptive messages: Each message includes information about how to process it.
	Hypermedia as the engine of application state: Clients may only transition through actions defined by hypermedia on the server, with the one exception being a externally defined entry point.

When a web api conforms to these constraints, it is known as a ’restful api’. If the api is HTTP based, as well be explored in this report, it has the following properties:

1. Base URL, eg `http://example.com/resources/`

2. An Internet Media Type[10][4] for the data, usually JSON or XML.
3. Based upon standard HTTP verbs
4. Hypertext links to reference state
5. Hypertext links to reference related sources.

Restful apis consist of two types of endpoints: collections, which return links to multiple resources(which may be collections themselves), or elements, which return the representation of a single resource.

Example restful api methods

Resource	Collection `http://example.com/resources/`	Element `http://example.com/resources/item42`
GET (nullipotent)	List the URIs with optionally other details of the collection's members.	Retrieve a representation of the element in an appropriate Internet media type.
PUT (idempotent)	Replace the entire collection with another collection.	Replace the referenced element. Create it if it does not exist.
POST	Add a new element to the collection. The new entry's URI is automatically assigned and is usually returned	Not widely used, treats the element as a collection to create an entity in it.
DELETE (idempotent)	Delete the entire collection.	Delete the referenced element.

Restful APIs are very common in modern web apps due to being self documenting[11] and easily scalable.

Summary

WHile both SOAP and restful web apis are common on the Internet, this report will limit itself to considering only those constrained by the rest architecture. This is due to SOAP apis generally being very complex structures, and difficult to automatically parse by a computer. By contrast, due to the 'hypermedia as the engine of application state' constraint, web apis can be automatically by crawled and documented from the entry point.

[10]A standard identifier on the Internet to define the type of data in a file.

[11]Due to states being encoded in the hypermedia, restful apis can be navigated with no external documentation.

The Livedrive api 'Onyx' is fully restful, thereby allowing it to serve as a case study.

1.2 API Documentation

As discussed earlier, restful apis are self documenting in that each endpoint contains references to its entities. However this does not provide full detail on how the api works, and what format data will be returned in. Therefore, there are multiple standards in the industry to accomplish this, which can broadly be split into two categories.

1.2.1 Human documentation

High level overview of how the api works, this is usually written for and intended for humans.

Arguments
This method has the URL `https://slack.com/api/api.test` and follows the Slack Web API calling conventions.

Argument	Example	Required	Description
error	myerror	optional	Error response to return
foo	bar	Optional	example property to return

Response
The response includes any supplied argument

```
{
    "ok": true,
    "args": {
        "foo": "bar"
    }
}
```

If called with an error argument an error response is returned:

```
{
    "ok": false,
    "error": "myerror",
    "args": {
        "error": "myerror"
    }
}
```

Figure 1.1: Example high level documentation

While useful for humans, this type of documentation can not be parsed easily and so will not be used in this report.

1.2.2 Specifications

More rigid and formally defined, API specifications can be read by both humans and machines. While specifications can be written on a ad-hoc basic, there are several popular languages in use for defining them. Unfortunately, there is no industry standard, and any organisation may use any method of documentation they so choose.

RAML

RESTful API Modeling Language (RAML)[2] is a specification language for restful apis, based upon YAML[12][5]. Due to being based upon YAML, it is both human readable, and fairly easily parsed by a machine due to mature open source YAML parsers. The RAML group provides multiple tools to make working with RAML easier for the end user, however it is still a young project and some major expected functionality such as reference parsers are still missing.

```
#%RAML 0.8

title: World Music API
baseUri: http://example.api.com/{version}
version: v1
traits:
  - paged:
      queryParameters:
        pages:
          description: The number of pages to return
          type: number
  - secured: !include http://raml-example.com/secured.yml
/songs:
  is: [ paged, secured ]
  get:
    queryParameters:
      genre:
        description: filter the songs by genre
  post:
```

[12]YAML Ain't Markup Language: a human-readable data serialization format.

```
  /{songId}:
    get:
      responses:
        200:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A canonical song",
                  "properties": {
                    "title":  { "type": "string" },
                    "artist": { "type": "string" }
                  },
                  "required": [ "title", "artist" ]
                }
            application/xml:
    delete:
      description: |
        This method will *delete* an **individual so
```

Listing 2: Example RAML specification

RAML contains many abstraction techniques such as trait definitions to allow similar endpoints to be factored out. It also allows multiple Internet media types to be defined as responses in-line, allowing for a great deal of flexibility. However, due to all these abstractions, creating a full parser is difficult.

API Blueprint

Created by Apiary, API Blueprint[1] is a specification language based upon Markdown. Similar to RAML, it is fairly new, however it has a large set of usable tooling. Being closed source, the specification can not be easily expanded and therefore is deeply tied into the existing tool set. It also does not include the abstraction tools of RAML, thereby resulting in many repeated definitions in a specification.

However, the tooling contains many useful utilities for automatic testing such as automatic mock server generation, thereby making it a popular choice in the industry.

Swagger

Swagger[3] is the last of the major api specification languages. It is based upon JSON, and while it is human readable it is more aimed at machine processing. It is the most mature of the three languages discussed so far. However, the aim of Swagger is code generation and server integrations: it is not designed to be used for unit testing. It consists of an initial specification written in JSON, which can then be transformed into a HTML site, a YAML representation, as well as processed by a variety of third party tools.

1.3 Unit Testing

Unit testing is the simplest technique used when creating web apis: it consists of writing a test based upon the api specification to test a specific feature of an endpoint, and then implementing the api until it passes. Due to the wide variety of api specifications, along with the range of requirements that need to be tested, this is generally done manually by the engineer working on the endpoint. Unit tests have the further advantage of alerting the engieer if the functionality ever breaks due to future engineering. Unit tests are organised in a suite which can be run upon the code base without affecting any deployed versions of the application.

Unit Testing is generally synonymous with Test-Driven Development (TDD), where all tests are written before the api implementation begins. The tests are written based upon the api specification, thereby allowing any gaps in the specification to be discovered and filled in early. It also leads to less buggy software if done properly, as every module is sure to be unit tested. There are many tools to aid in trying to automate part of this process, some of which are discussed below.

1.3.1 FitNesse

FitNesse[13] is an acceptance testing framework[14]. It allows users to input formatted test cases, which it uses to automatically generates tests and execute them against the web api. While it generally works well when setup, it requires the creation of fixtures[15] which may be difficult depending on how the api is implemented. The test cases are created in a wiki using a natural language markup, which allows non-technical members of the team to also contribute to it.

[13]`http://www.fitnesse.org`

[14]Unit tests written for the specific purpose of ensuring that the specification is fully implemented.

[15]Support classes with the api framework to allow the tests to run

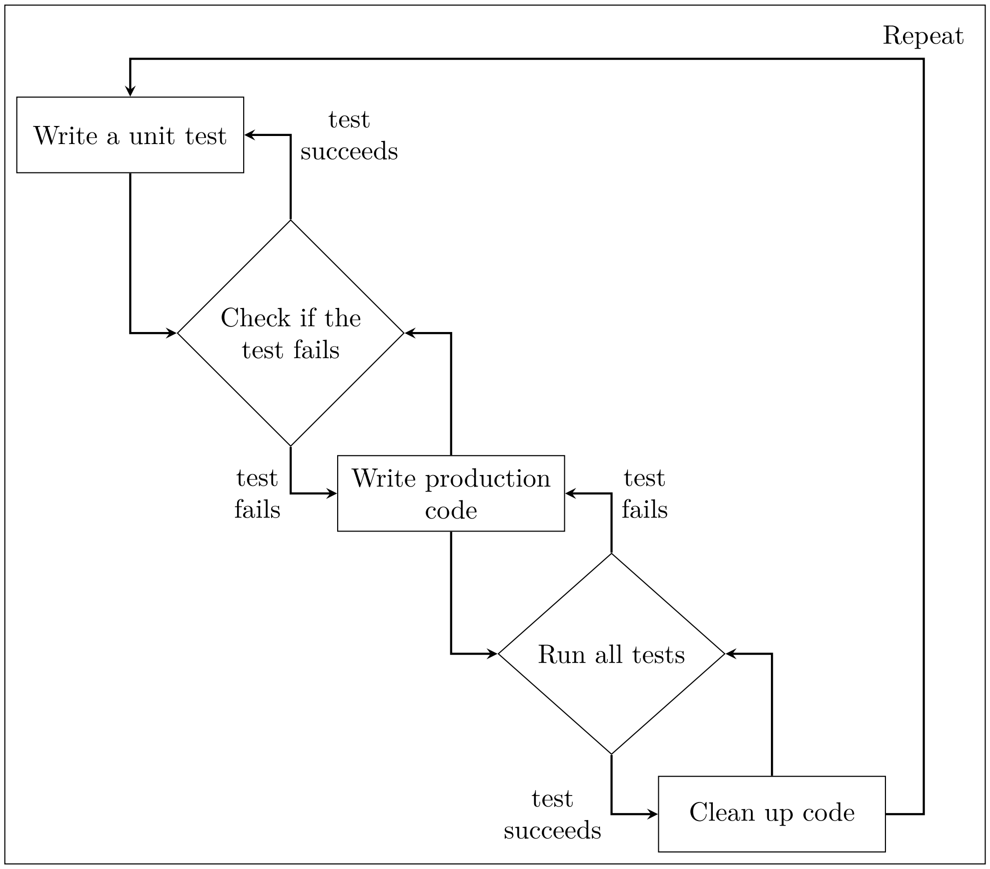

Figure 1.2: The Test-Driven Development cycle

FitNesse acts like a black-box test engine: the tester writing FitNesse inputs should not be aware how the specification is implemented. While a good idea in theory, there are usually bugs introduced due to the fixtures which can cause the acceptance tests to fail. Furthermore, FitNesse is a completely self contained platform: it does not integrate with any other service, limiting furthermore automation.

1.3.2 RSpec

RSpec[16] is an acceptance testing framework for Ruby. It allows 'behaviours' to be defined which emulate a human using the api, and then automatically run against the code-base. It is by far the most commonly used testing framework for ruby apis, however it has seen limited use in other languages. Tests are written in a DSL based upon ruby which mimics natural language, making it easy to quickly add new tests as the need arises. Like FitNesse, RSpec is completely self contained and can not integrate with other services: the specification must be manually converted into behaviors.

[16]`http://rspec.info`

1.3.3 xUnit

xUnit is a family of unit testing frameworks for object-oriented languages. They consist of seven major components:

Test runner	Runs tests implemented using xUnit
Test case	Base class tests inherit from
Test fixture	Sets up the environment for a test.
Test suite	A set of tests that depend on the same fixture.
Test execution	Tests run an initializer, then the body of the test, and finish by cleaning up any changes they made.
Test result formatter	Interprets the results from the runner in a human readable format.
Assertions	A logical condition used to evaluate whether a test passed or failed.

xUnit frameworks are generally open source. This has allowed an ecosystem of services to develop around them, leading to easy integration of third party services. It is in active use by a wide variety of organizations, such as Microsoft[17] and Oracle[18]. It is considered the industry standard in most enterprise application.

While powerful, xUnit frameworks again require the specification to be transformed into a propriety format. It can be very involved to setup, as fixtures need to be created for every test suite.

1.3.4 Summary

While there are a wast array of automatic testing tools, all of them require a large amount of manual intervention. Namely, all the tools discussed above require the original specification to be transformed into propriety formats. While this would not be an issue if this were to happen one, api specifications are continuously updated and so require an inordinate effort by the software engineer to keep the two models in sync. Furthermore, this is a likelihood of introducing errors when converting the specification to test cases.

An ideal solution would be able to parse a standardized specification and automatically generate all the relevant test cases, then execute the test cases against the code base. It would then be able to output which test cases failed in a human readable format.

[17]`http://www.nunit.org`
[18]`http://junit.org`

1.4 Livedrive

Due to large possible scope of this project due to the myriad apis in production, I will limit the scope to an api provided by the organization Livedrive. Livedrive plan on using the completed project in production. They have begun created a new web api, code named 'Onyx', which can act as a good case study on how well the project will work in real world use cases.

Livedrive is a cloud backup company. They provide consumer facing clients which can be used to manage files. These files are then sent to an off-site server for storage, thereby allowing access from other clients and protecting them from machine failure. Onyx will be used to communicate between the clients and backend servers, allowing actions such as 'rename' and 'delete'. Onyx' is written in C#, and is currently tested using FitNesse. Due to the quickly evolving nature of the api, FitNesse has proved too slow to be scalabe in the long term.

As Onyx is a trade secret, no code directly from the project will appear in the report. Instead, I will construct a mock api replicating the features of Onyx necessary to implement this project, as needed. This mock will aim to provide the same responses as Onyx, but will be merely imitating the responses.

Chapter 2

Report Body

The central part of the report usually consists of three or four chapters detailing the technical work undertaken during the project. **The structure of these chapters is highly project dependent**. They can reflect the chronological development of the project, e.g. design, implementation, experimentation, optimisation, evaluation, etc (although this is not always the best approach). However you choose to structure this part of the report, you should make it clear how you arrived at your chosen approach in preference to other alternatives. In terms of the software that you produce, you should describe and justify the design of your programs at some high level, e.g. using OMT, Z, VDL, etc., and you should document any interesting problems with, or features of, your implementation. Integration and testing are also important to discuss in some cases. You may include fragments of your source code in the main body of the report to illustrate points; the full source code is included in an appendix to your written report.

2.1 Section Heading

2.1.1 Subsection Heading

Chapter 3

Requirements

Below are a set of functional and non-functional requirements based upon which the system should be developed. Successful completion of this project should lead to a well defined way of specifing restful web apis, and using this specifcation to automatically test the implementation for bugs.

3.1 User Requirements

This section defines all the actions a user must be able to perform:

- Fully specify a restful web api, so that it can be clearly understood by developers and software architects.
- Be able to define what 'correct' operation of the api results in.
- Automatically run tests against an implementation of the web api based upon the specification.
- Be able to define sample inputs for the restul web api.
- Be able to define sample outputs for a given input.
- Be able to check in the api returns the correct output for any given input.

3.2 Functional Requirements

This section defines the functionality the system must be able to accomplish.

- Parse the specification into a machine readable format.
- Provide the user with a method to execute tests against a web api.
- Report to the user how many tests failed.
- Be able to send sample inputs to the restful api and ensure they are valid.
- Be able to detect if an api implementation does not follow the specification.

3.3 Non-Functional Requirements

This section defines more subjective requirements which will help provide a high quality project. It is split into two parts: Specification and Automated testing.

3.3.1 Specification

- The specification language must be easily human readable.
- The language must be able to suiccintly and unambiuously define a restful web api.
- The language must be easily convertable to a format machines can process.
- The language most be agnostic to any implementation details. It should not matter how the api is going to work, or which programming language it will be written in.

3.3.2 Automated testing

- The system should be reliable: it should return the same result if run with the same specification on the same implementatiom.
- The system should be implementation agnostic: It should be able to run the tests no matter what language the api is written in.
- The system should be able to run on as many common operating systems as feasibly possible.
- The system should be performant, scaling linearly with the number of tests being run.
- The system shpuld be easy to maintain, following development best practices and making good use of architecturel design patterns.

- The system should be fully unit tested to provide assurences that the code is as correct as possible.

References

[1] Api blueprint specification. `https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md`. [Onlune; accessed 12-February-2015.

[2] Restful api modeling language. `http://raml.org`. [Online; accessed 4-February-2015].

[3] The swagger specification. `https://github.com/swagger-api/swagger-spec`. [Onlune; accessed 12-February-2015.

[4] Internet media type registration, consistency of use. W3C 0129, 2002.

[5] O. Ben-Kiki, C. Evans, and I. Net. Yaml ainâĂŹt markup language (yamlâĎć) version 1.2. `http://www.yaml.org/spec/1.2/spec.html`, 2009. [Online; accessed 4-February-2015].

[6] T. Berners-Lee and R. Cailliau. Worldwideweb: Proposal for a hypertext project. `http://www.w3.org/Proposal.html`, 1990. [Online; accessed 4-February-2015].

[7] Tim Berners-Lee. Information management: A proposal. `http://www.w3.org/History/1989/proposal.html`, 1989. [Online; accessed 4-February-2015].

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[9] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`, 2000. [Online; accessed 4-February-2015].

[10] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, A. Karmarkar, and Y. Lafon. Soap version 1.2 part 1: Messaging framework (second edition). W3C 20070427, 2007.

Appendix A

Extra Information

A.1 Tables, proofs, graphs, test cases, ...

The appendices contain information that is peripheral to the main body of the report. Information typically included in the Appendix are things like tables, proofs, graphs, test cases or any other material that would break up the theme of the text if it appeared in the body of the report. It is necessary to include your source code listings in an appendix that is separate from the body of your written report (see the information on Program Listings below).

Appendix B

User Guide

B.1 Instructions

You must provide an adequate user guide for your software. The guide should provide easily understood instructions on how to use your software. A particularly useful approach is to treat the user guide as a walk-through of a typical session, or set of sessions, which collectively display all of the features of your package. Technical details of how the package works are rarely required. Keep the guide concise and simple. The extensive use of diagrams, illustrating the package in action, can often be particularly helpful. The user guide is sometimes included as a chapter in the main body of the report, but is often better included in an appendix to the main report.

Appendix C

Source Code

C.1 Instructions

Complete source code listings must be submitted as an appendix to the report. The project source codes are usually spread out over several files/units. You should try to help the reader to navigate through your source code by providing a “table of contents” (titles of these files/units and one line descriptions). The first page of the program listings folder must contain the following statement certifying the work as your own: “I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary”. Your (typed) signature and the date should follow this statement.

All work on programs must stop once the code is submitted. You are required to keep safely several copies of this version of the program - one copy must be kept on the departmental disk space - and you must use one of these copies in the project examination. Your examiners may ask to see the last-modified dates of your program files, and may ask you to demonstrate that the program files you use in the project examination are identical to the program files you had stored on the departmental disk space before you submitted the project. Any attempt to demonstrate code that is not included in your submitted source listings is an attempt to cheat; any such attempt will be reported to the KCL Misconduct Committee.

You may find it easier to firstly generate a PDF of your source code using a text editor and then merge it to the end of your report. There are many free tools available that allow you to merge PDF files.