KING'S College LONDON

University of London

6CCS3PRJ
Automatic Test Generation from a Specification

Final Project Report

Author: Hani Kazmi

Supervisor: Dr Christian Urban

Student ID: 1201492

April 24, 2015

Abstract

The Internet, and more specifically the World Wide Web, is a rAPIdly evolving platform. With the advent of HTML5 and the current abundance of processing resources, websites have been evolving into 'web apps', which require extensive testing to ensure it is error free. These tests currently consist of manually written 'unit tests', which target individually modules and raise an error if future work breaks the module. Of particular note are 'web APIs', sets of endpoints used to communicate between components of web apps. They are generally strictly defined, and therefore there is scope for automatic testing to ensure compliance with their specification.

This thesis focuses on a subset of these APIs: those classified as 'restful', and creates a framework consisting of three components - a language to formally define the API, automatic tests to reaffirm that the specification matches a given implementation, and unit test generation from the aforementioned specification. The thesis will use the API provided by the company 'Livedrive' as a case study.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Hani Kazmi

April 24, 2015

Acknowledgements

I would like to thank Dr. Christian Urban for supervising this project and offering his knowledge, even when it was a very abstract idea. I would also like to thank Daniel Cutting, my Livedrive Manager, for the original idea and believing in it enough to use it in production.

Contents

Chapter 1

Introduction

1.1 Motivations

The Internet has long been an important means of communication. Historically, this has been dominated by websites hosted on the World Wide Web: static, unchanging pages which display information and are navigated using URLs.[1] As these sites were simple, ensuring that they were reliable was a simple task. With the advent of fast broadband and increased processing power, websites have slowly been evolving into more self contained entities, colloquially known as 'web apps'.

Generally powered by HTML5[2][14] and JavaScript[3], web apps are far more dynamic. Due to the many moving pieces now involved on a website along with the increasing file sizes[4], testing has become unwieldy. This is currently overcome by writing 'Unit Tests'[5] which alert the programmer when a future change breaks existing functionality. This is tedious work, and covering all possible cases is difficult.

A common way of creating web apps is by having a client side application written in JavaScript, which communicates with a backend that can be written in a wider variety of languages. These two components may be designed and implemented by different people, and indeed different companies altogether. Therefore, there is generally a well defined API[5] used to communicate between them using HTTP[6]. If this API can be strictly defined, there is scope

[1] Uniform Resource Locater: a reference to a resource on the Internet.
[2] A revision of the HTML standard which added many elements needed for dynamic websites.
[3] A scripting language implemented by virtually all web browsers, updated as part of the HTML5 specification
[4] Website trends, 2010-2015, `http://httparchive.org/trends.php?s=All&minlabel=Nov+15+2010&maxlabel=Jan+15+2015`
[5] Application Programming Interface.
[6] Hypertext Transfer Protocol.

to automate part of the testing process using it.

1.2 Aims

The aim of this thesis is to automate Web API testing as much as possible. A three-fold approach will be taken to this problem:

1. A language to formally define APIs. This will allow the API to be computationally modified and reasoned about.

2. A framework which automatically generates a battery of tests to check that an API implementations matches its specification. This will act as a basic sanity check for the implementation, as well as allow the implementation to be monitored for changes.

3. A framework which automatically generates Unit Tests for a given API specification. While all manual tests cannot be eliminated, a large subset can be inferred and constructed from the API definition. This will be accomplished by extending the definition language to allow example requests, and inferring any additional required information form the context.

1.3 Scope

Due to the time constrains of this project, the framework will be limited to dealing with APIs following a RESTful[7] architectural style. Specifically, the web API 'Onyx' used by the company Livedrive[8] for communication between their C#[9] backend and web frontend will be used as a case study and focal point for the thesis.

While the project will be tested against the full Onyx API, its contents are a trade secret for Livedrive. Therefore, a small restful API modeling any needed functionality of Onyx will be created and used for the purposes of the report.

[7] Representational State Transfer: An architectural style for creating scalable web services
[8] `http://www.livedrive.com`
[9] A programming languages developed by Microsoft, regularly used for writing large scale web backends.

Chapter 2

Background

2.1 State of the Internet

In 1989, Tim Berners-Lee proposed a communication system for CERN[9]. He soon realized the concept could be expanded, and in 1990 he published the proposal for what would become the World Wide Web[8]. 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[10], 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'.

2.1.1 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.

2.1.2 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[12] 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[11]. 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][6] 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

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

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.

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

2.2 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.

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

2.2.1 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 2.1: 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.

2.2.2 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][7]. Due to being based upon YAML, it is both human readable, and fairly easily

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

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:
  /{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.

2.3 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.

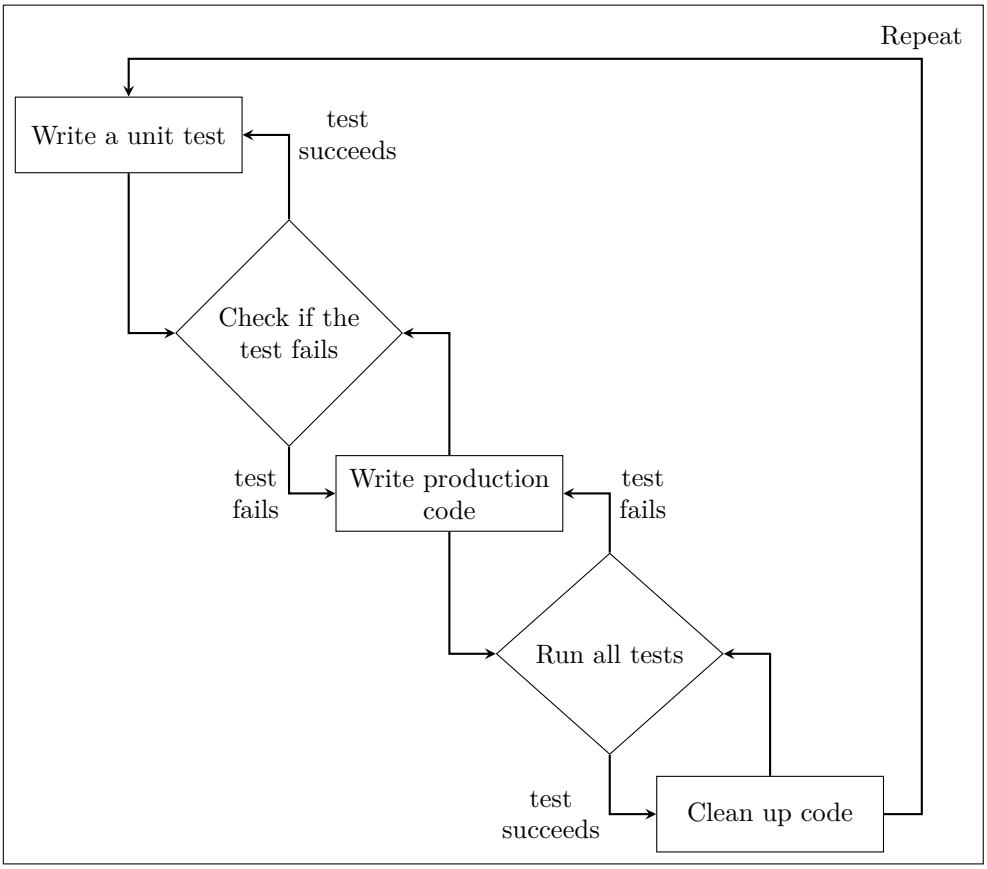


Figure 2.2: The Test-Driven Development cycle

2.3.1 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.

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.

[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

2.3.2 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.

2.3.3 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.

2.3.4 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

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

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.

2.4 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 3

Design Considerations

This chapter discusses the overall system design of the project, and details what each module of the system will accomplish. The chapter will give a high level overview, with more in depth implementation details further on. Each choice is based upon achieving one of the requirement of the project

3.1 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 specifying restful web APIs, and using this specification to automatically test the implementation for bugs.

3.1.1 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 restful 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.

3.1.2 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.

3.1.3 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.

Specification

- The specification language must be easily human readable.
- The language must be able to succinctly and unambiguously define a restful web API.
- The language must be easily convertible 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.

Automated testing

- The system should be reliable: it should return the same result if run with the same specification on the same implementation.
- 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 should be easy to maintain, following development best practices and making good use of architectural design patterns.
- The system should be fully unit tested to provide assurances that the code is as correct as possible.

3.2 System Components

The project consists of three overall systems, which can be broken down into smaller modules.

The RATML specification: A high level specification language based upon RAML, allowing restful web APIs to be fully specified. The language can be split into two parts:

- A language to define the endpoints of the API, along with all of its collections and resources. An API should be able to be implemented using just this as an reference.
- A language to define test input and output data.

RATML parser: As RATML is a new specification language, a custom parser will need to be built.

- A YAML parser will be used to convert the text based specification file into a human processable format.
- A RATML parser will then take as input the objects from the YAML parser, and convert them into a data structure that can be used by the rest of the system.
- A facade will allow the system to quickly query and manipulate the aforementioned data structure.

RATML Framework: The framework will use the parsed data to create and execute tests.

- A system to iterate through the endpoints and generate the best possible test for it.
- A system to use the sample input data to interpolate more possible values for the endpoint, and use it to send test requests.
- A system to report the discovered results in a human readable manner.

The big advantage of this approach is the separation of concerns: each of the three components is loosely linked and therefore can be worked on and tested in isolation.

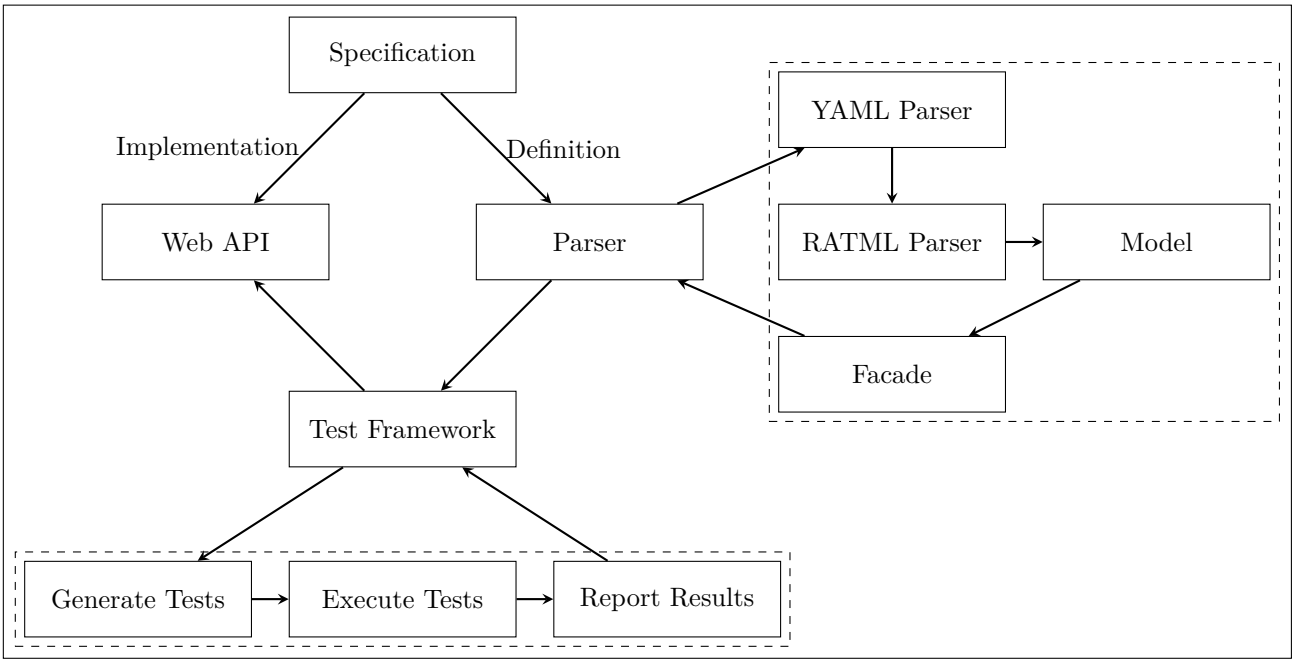


Figure 3.1: System Architecture Diagram

3.3 Software Packages

The system will mainly be written in Ruby[1]. This is due to the language's prominence in the Test Driven Development committee, along with being multi-platform. The language comes pre-installed on OS X and most distributions of Linux, as well as being easily installable on Windows, thereby covering the three biggest browsers. Furthermore, the language is completely portable, and the same code base works on any supported platform. Of particular note are the languages meta-programming capabilities, which allow parsers to be written very efficiently.

In particular, Ruby's Blocks are an implementation of higher level functions which allow arbitrary code to be stored as variables and passed around the system. This allows for code to be very highly encapsulated, such as letting the test framework generate and execute tests on the fly. I will use the Sublime Text editor to write the code, which is freely available[2].

3.4 Testing

As the main code-base will be written in Ruby, the Onyx imitation will also be written in the same language for consistency. Therefore, I will use the Ruby on Rails[3] library. This library is specifically designed for creating web services, and provides many features to aid in this.

[1] `https://www.ruby-lang.org`
[2] `https://www.sublimetext.com`
[3] `http://rubyonrails.org`

Some of these includes a routing table to direct API requests and tight integration with a SQL database to store collections, along with support for all HTTP Verbs/

Rails applications come bundled with a server to quickly set up and host the web service, thereby allowing very fast iterations on the API implementation.

Chapter 4

Specification Language

4.1 Existing Languages

As discussed in the Background chapter, there are ample languages on the market for specifying an API. However, as discovered most of them have limitation; either being hard to parse or limited in tooling. As this project requires a language that can be easily created and written by humans and processed by machines, a new language will be created. Furthermore, due to the existing languages being mainly closed source and under existing licenses, the required changes to add sample input and output can not be made.

The language defined below will be heavily inspired by RAML. This is due to RAML already having many of the needed properties: It is based upon YAML, which is a widely understood format, and therefore there are already editors and syntax highlighters for it. Furthermore, there are many mature YAML parsers, allowing the language to be easily machine processed.

4.2 The Anatomy of a Restful Web API

A restful API has an entrypoint, generally of the form `http://example.com/ap1/v1/`. Sending a GET request to this collection returns a list of top level resources. In the case of Livedrive, this will be the Users resource and the Files resource[1]. There will also be an authentication path to protect the API from malicious users.

```
{
  "name": "Livedrive Onyx API"
  "implementation_version": "1",
```

[1] Onyx contains more resources than this, but this simplification is sufficient for this report.

```
  "uri": "http://example.com/",
  "users": [
    {
      "name": "User Profiles",
      "uri": "/users"
    }
   ],
  "files": [
    {
      "name": "File storage",
      "uri": "/files"
    }
   ].
  "authenticate": [
    {
      "name": "Authentication",
      "uri": "/authenticate"
    }
  ],
}
```

URIs are given relative to the root URL, so the Users collection could be queried from `http://example.cpm/API/v1/users`. This collection is likely to be secured due to sensitive nature of the data contained within it, so the user would need to authenticate first and include a authentication token with each request.

```
[
  {
          "name": "Hani Kazmi",
          "uri": "/users/1",
          "role": "admin",
          ...
  },
  {
          "name": "John Doe",
          "uri": "/users/1",
          "role": "user",
          ...
  },
  ...
]
```

The base level collections will contain links to the individual elements, along with basic data. Some of this data may be links to elements in other collections. Querying one of the elements returns full information about the element.

```
{
        "name": "Hani Kazmi",
        "uri": "/users/1",
```

```
        "photo": "/photos/121",
        "files": [
            {
                    "filename": "flower.jpg",
                    "uri": "/files/210312",
                    ...
            },
            {
                    "filename": "report.pdf",
                    "uri": "/files/203840237",
                    ...
            },
            ...
        ],
        ...
}
```

At any of these stages, a different HTTP verb can be used to modify the data[2]. For PUT or POST, this requires knowing the right format to send new data in, which is generally documented externelly.

```
POST /users
Host: example.com
Authorization: Basic xxxxxxxxxxxxxxxxxxxxxx
Accept: application/json
Content-Length: nnn
Content-Type: application/json
X-Compute-Client-Specification: 0.1

{
  "name": "Richard Stallman",
  "photo": "/photos/123213",
  ...
}
```

Adding an element to a collection assigns it an automatic number, and the collection list is updated to reflect this.

4.3 Designing a Specification

Restful APIs consist of collections and elements (collectively known as endpoints), either of which can link to each other. Therefore, most specifications end up being very recursive. For a GET request, they generally define the endpoint, the data contained withint the element, and the name of any elements/collections it may link to. For each element, they also define the

[2]Assuming the user has the necessary permissions.

format requests from other verbs must be made in. Any verbs not defined in the specification are assumed to not apply to the endpoint.

The specification language designed in this project will be called RATMAL (RESTful API Testing Modeling Language). It will consist of of a hierarchy of collections and elements. At each level, the verb and their responses will be defined, using a colon as a separator. A 'description' tag will allow human readable comments to be included to clarify any complexities.

Collections are declared as /[collection_name]: to reflect the form they appear as in the implementation. Elements are declared as /[element_name] to signify that element_name is simply a label and will be replaced by a number. Other key words include 'queryParameters' to define any parameters which may be included with GET requests, and 'responses' to indicate the response schema.

The full specification is given in the Appendix.

4.4 Implementing the RATML Parser

While RAML is easily parseable, it is a new language and therefore does not have a reference parser for ruby. The only existing implementation[3] is abandoned and incomplete. Therefore, I will create a new parser to fit the requirements of RATML Ruby has a mature open-source library for parsing YAML documents called Psych[4]. Psych convert YAML files into an abstract syntax tree, and therefore it will be forked and used as the starting point for my RATML parser. The abstract syntax tree consists of in memory nodes, which can intercepted and converted into purpose-built data structures for RATML. There are four types of nodes relevant to RATML:

node The base class from which all the other nodes inherit. It contains references to child nodes, and utility methods to allow iteration and searches.

document The root node. It contains data about the YAML file, along with a single child which points hierarchically to the rest of the nodes.

scalar Terminal nodes, representing YAML scalars. Their primary purpose is to store the values from the YAML document.

mapping Intermediary nodes. They represent YAML mappings between scalars.

RATML consists of values, which are basic data types, and properties, each of which may have one or more values and properties as children. This lends itself to a 'Property' parent

[3] `https://github.com/coub/raml_ruby`
[4] `https://github.com/tenderlove/psych`

class, which can be sub classed and specialized to accommodate the variety of values stored in different properties. The properties can be broken down into:

Root The initial property constructed when visiting the entry point. It contains references to all the other properties.

Resource The top level endpoint property. Contains values for both Collections and Elements.

Method For each HTTP verb in a resource, there will be a separate Method property, allowing different verbs to be dealt with separately.

Response Each valid HTTP code in method corresponds to a separate response. Each response is in the form of a JSON schema, as defined in the specification.

Body Body objects store each individual schema.

4.4.1 Dealing with JSON

Most values are singular objects: either a number or a string consisting of a few words. The JSON schema are more complex. While they could be stored as strings, this will stop them being easily parsed when using them for comparison against reponses. As an example, in the schema:

```
{ "$schema": "http://json-schema.org/schema",
  "type": "object",
  "description": "A canonical user",
  "properties": {
    "name":  { "type": "string" },
    "uri": { "type": "uri" },
    "photo": { "type": "uri" },
    "files": [{
      "filename": { "type": "string"},
      "uri": { "uri": "uri"},
      }]
  },
  "required": [ "name", "uri" ]
}
```

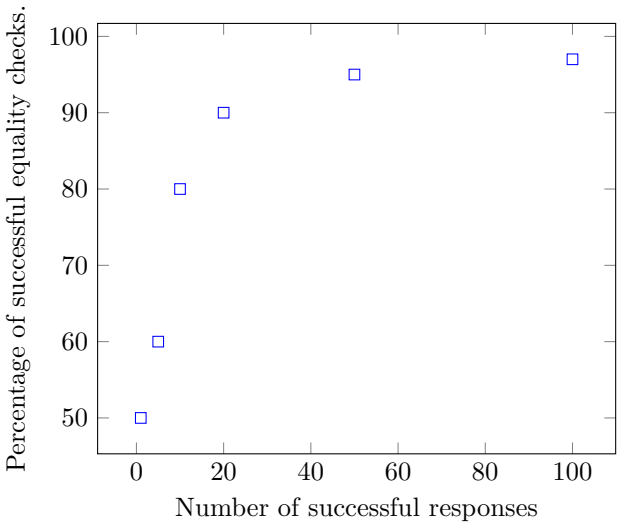
the property ”title” has type string, and is defined as required. Therefore, schema will be stored as Ruby meta-programmed objects, which can run comparison on themselves to evaluate if a response matches. This is further complicated by the fact that determining whether a given string is an ’artist’ or not is difficult computationally.

4.4.2 Converting the syntax tree to properties

The algorithm to convert from nodes to properties starts at the document node. It recursively parses the tree in a depth-first order, using a simple heuristic to decide which property the node corresponds with. Schema get further processing: they are wrapped in a functional Block which can evaluate equality checks. Due to the uncertain nature of how precise the schema is compared to the implementation response, this is done by assigning each value in the schema a point score depending on the closeness of the match. The point score to assign is calculated using the average of previous scores in successful responses. This can then be used to determine equivalence using:

$$\text{Equality} \iff \frac{\sum \text{property\_points}}{\text{Number of properties}} \geq \text{Number of properties} + \text{required properties}$$

While not perfect, this formula improves as more successful tests are made, and therefore point score assignments improve.



4.5 The Onyx Specification

The Onyx API has several hundred endpoints, and is too large to fully replicate for this project. Instead, I will create a subset of it to cover all the major types of endpoints, while still keeping it manageable for the scope of this report. As introduced in 4.2, the entry point will have Files and Users collections. Both collections will be amenable to GET requests to display a list of elements, and POST requests to add an element. Elements of the aforementioned collections will respond to GET, PATCH, and DELETE requests to modify them accordingly.

Users will contain a subset of the values which can be accessed using Onyx. This includes the values required for the API functionality, such as name and list of files, along with optional values to cover the range the tests should cover. Files will store file meta data like filename and size.

The full specification is included in the Appendix.

4.5.1 Implementing the Specification

As discussed before, the test Onyx API will be implemented in Ruby on Rails. Rails applications follow the Model-View Controller(MVC)[16] design pattern, where logic(the model) and responses(View) are split to allow for greater modularity. Rails applications consist of a web server which receives incoming requests. These are sent to a router which determines a controller to direct the request to. The controller uses any stored data and the logic from the model to construct a response, and return it to the client.

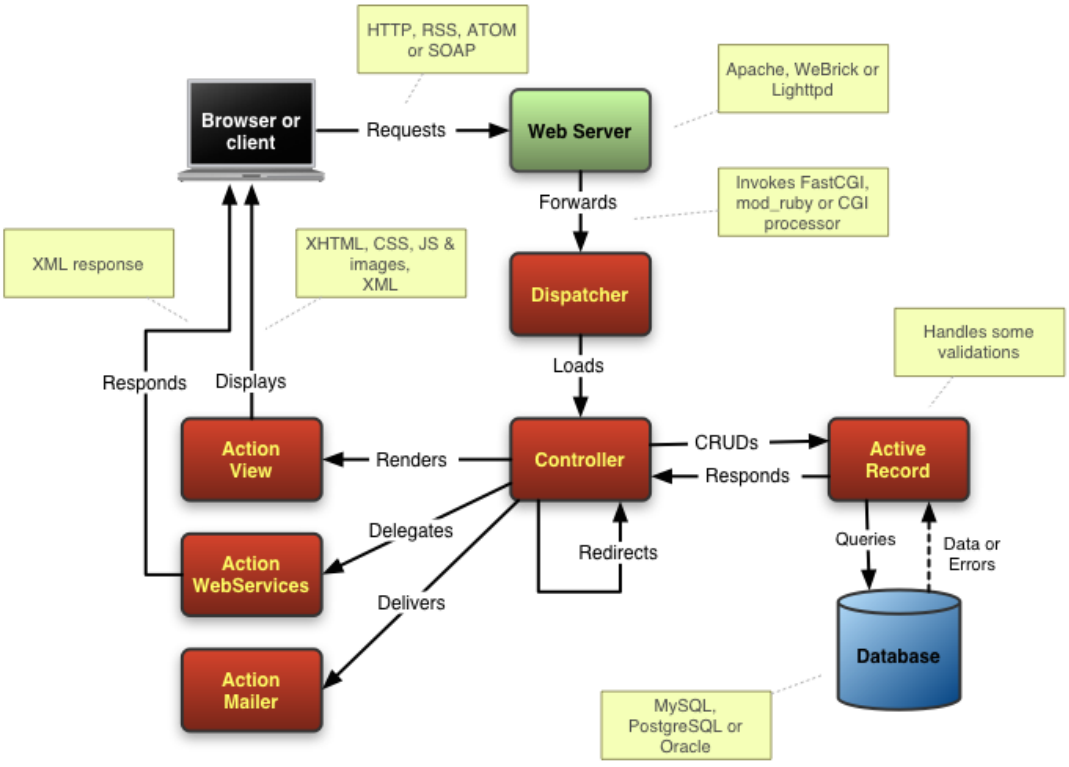


Figure 4.1: Ruby on Rails Application Architecture[15]

The Onyx implementation will begin with seeded data in the database to imitate the existing users at Livedrive. Requests received by the application will modify the database as needed

and return the constructed response, with the exception of file uploads and downloads. For the sake of limited storage, file transfers will be replaced and the responses doctored to match a fully working implementation.

Chapter 5

Test Framework

The Test Framework consists of three components, all written in Ruby:

5.1 Test Generation

The root property of the specification can be imported from the parser. Transforming this into a test suite consists of the following steps:

1. Traversing the tree of properties in a depth first order, and creating a list of endpoints.
2. For each endpoint, discovering which HTTP verbs are accepted and associating it with the property.
3. Constructing HTTP requests using the parameter data for each endpoint.

These tests can be saved as an array of custom 'Test' objects, which will be iterated when executing the tests.

5.2 Test Execution

For each generated test, the Framework sends a HTTP request to the web API. The received response is then run against the schema Block, to see if it can be considered a valid response. Each test may return one of three states:

1. The network request receives no response. Either the endpoint has not yet been implemented, or is has a major problem.

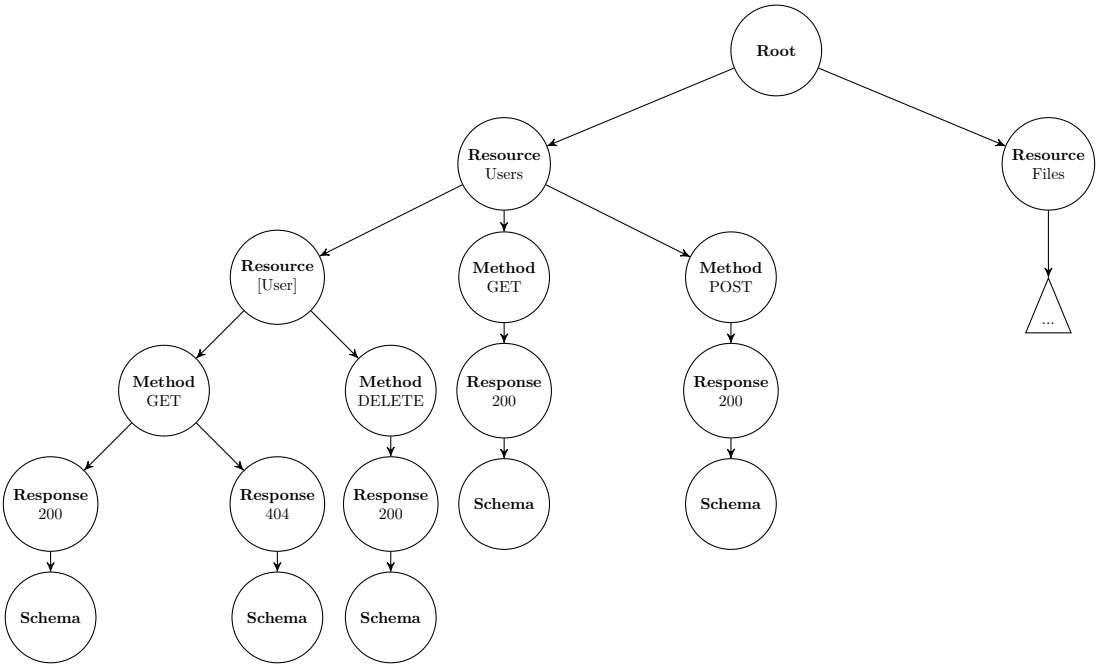


Figure 5.1: Extract of Example Property Tree

2. A response is received, but does not match the schema. There are two possible reasons for this:

   - The response is evaluated against a heuristic block. which is not guaranteed to be parsed correctly. As shown in the previous section, this has proven unlikely through experimentation but can not be removed altogether.
   - There is a bug in the API implementation. Either a value of the wrong type was returned, a required value is missing, or there are additional, undocumented values.

3. A response is received and matches the schema. This guarantees that the endpoint has both been implemented, and matches the schema at least superficially.

5.3 Test Results

The results produced through the execution must be reported in a human readable manner. This is done by storing the results in the Test Execution module in a custom Object which records:

- The endpoint.
- The HTTP verb accessed.

- Whether the test passed or failed.
- If the test failed, what caused it to fail.

These results can then be run through a pretty printer, and both stored in a log file and shown on the screen to the user.

5.4 User Interface

As the user of the Test Framework can be assumed to be technically proficient, a graphical user interface and in-depth help is not required. Hence the Test Framework will be distributed as a command-line application. Running the application without any input parameters will produce a short help message explaining usage. Specifying a RATML file as input will cause the Test Framework to run and report results using the pretty printer. An optional flag can be used to output results to a log file.

```
$ ./testframework.rb
Usage: ./testframework.rb specification.ratml [-l results.log]
```

Keeping the interface simple will allow the Framework to be easily integrated into existing test suites that corporations may have. As the aim of this project is to automate as much as possible, no additional options will be provided. As the Test Framework calls the Facade in the RATML parser automatically, a user facing interface is not required for it. Similarly the API entry point and endpoints are inferred from the specification and do not need to be manually entered.

Chapter 6

Creating Varied Test Cases

As it stands, the Test Framework can generate test cases which check whether the endpoint exists, and basic sanity check on it matching the schema. It can be extended to compare sample input against output, as stated in the User Requirements.

6.1 Extending the Specification Language

Sample input consists of an endpoint to send the request to, and optional query parameters which the endpoint may accept. Sample output includes a HTTP status code and the JSON response. As the data is associated with an endpoint, the sample cases can be specified in RATML under the relevant Method declaration as a new 'Test Case' property. These properties will consist of a short tag to describe the test case, a resource number if required for the endpoint, any required query parameters, and a body to specify the response. Collection endpoints will not need to specify a resource number. They may include a longer description tag for humans.

```
\{userId}:
    get:
      testcases:
        exampleuser:
          resource: 1
          response:
            status: 200
            body: |
              { "name": "Hani Kazmi",
                "uri": "/users/1",
                "photo": "/photos/121",
                "files": [
                  {
```

```
                    "filename": "flower.jpg",
                    "uri": "/files/210312",
                    ...
                },
                {
                    "filename": "report.pdf",
                    "uri": "/files/203840237",
                    ...
                },
                ...
            ],
            ...
        }
    missinguser:
      description: Invalid users return a 404
      resource: 201
      response:
        status: 404
        body: |
          {
                  "message" : "Invalid user"
          }
```

6.1.1 Dealing with POST, PUT and PATCH requests

GET and DELETE requests generally only consists of an endpoint and optional query parameters. POST, PUT and PATCH may be more complicated as they are sending structured data to the endpoint. To simplify this, the structured data will also be specified as JSON in RATML. It can then be parsed into a relevant format by the Test Framework.

```
\users:
    post:
      testcases:
        createuser:
          query: |
            {
              "name" : "Alan Turing",
              "photo": "/photos/2131"
            }
          response:
            status: 200
            body: |
              {
                      "name": "Alan Turing",
                      "uri": "/users/21",
                      "photo": "/photos/2131"
              }
```

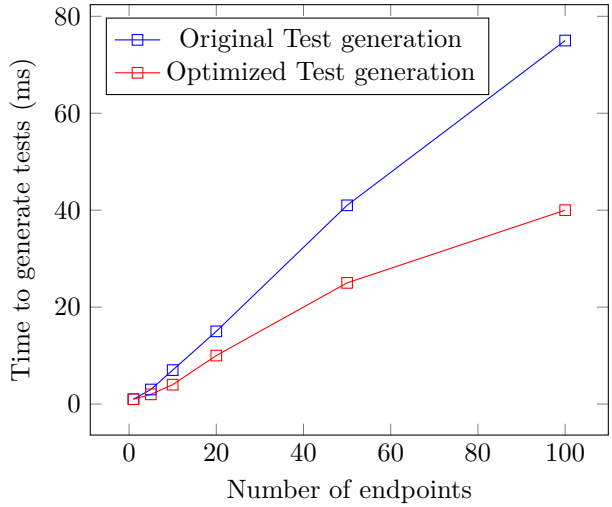
The full extended specification is given in the Appendix.

6.1.2 Adding input/output Test Cases to parser

The parser was modified to accept the newly defined Test Cases by creating a new 'testcase' property, consisting of 'input' and 'output' properties. 'Input' contain meta-programming functions which can construct the specified network request. 'Output' constructs block similar to the one discussed before which compares the status code and body to evaluate if the response matches.

6.2 Optimizing Test Generation

The parser currently transforms the RATML file to a tree of properties based upon the YAML scalars specified in RATML. While a workable solution, it requires the entire tree to be traversed when generating test cases, resulting in an O(n) run time for generation where n is the number of nodes. This proved slow for larger RATML files, especially with the new input/output test cases added in. Changing the parser to simultaneously create a hash map containing pointers to 'resource' and 'testcase' properties allows the Test Generator only run over the necessary nodes. This allows parallelizing the generation by having multiple threads using the hash map as a queue, drastically speeding up the process.



A thread pool can be created based upon the number of logical cores the host CPU has to minimize context switching. So long as the thread pool has not been saturated, it spawns a thread to follow the next unused pointer in the hash map to a node in the property tree. This is then parsed until it reaches another resource or test case, then terminates. A closure is used to coordinate the multiple threads, along with a semaphore enabled array to avoid race conditions.

```
def thread_pool(hashmap, max_threads):
  current_threads = 0
  current_index = 0
  testcases = ParallelArray.new         # A custom semaphore based array
  while current_index < hashmap.count:
    if current_threads <= max_threads:
      current_threads = current_threads + 1
      testcases.add Thread.new({         # A closure executed in a new thread
              # Parse the tree from the specified node, and return the created test case
              testcase = parse(array(current_index))
              current_threads = current_threads - 1
              current_index = current_index + 1
              return testcase
              })
    end
  end
  return testcases
end
```

Figure 6.1: The threadpool algorithm

6.3 Speeding up Test Execution

Test execution can also be parallelized with multiple threads running simultaneously. This however causes the issue of the framework not knowing which response corresponds to which test case[1]. Furthermore, there is an overhead of creating new threads and requests for every thread. Fortunately, there is an open source ruby library specialized in high throughput HTTP requests known as Typhoeus[4]. Typhoeus allows network request to be put in the 'hydra' network queue and assigned completion handler closures which automatically execute on the response. As a result, endpoints can be executed in parallel, speeding up the test sequence.

6.3.1 Destructive Tests

GET requests are nullipotent, while PUT and DELETE are idempotent: they can be executed multiple times and receive the same response. Other Verbs may not be, and responses may vary depending on the order in which the test is executed. (I.E, trying to GET an element before or after deleting it will cause different results.) This means not all requests can be parallelized. To deal with this, a new property, 'dependent', will be added to RATML. This is documented in the Appendix.

[1] Responses do not necessarily need to specify their corresponding request

```
\{userId}:
    get:
      testcases:
        exampleuser:
          resource: 1
          response:
            status: 200
            body: |
              ...
        examplemissinguser:
          dependent: missinguser
          resource: 1
          response:
            status: 404
            body: |
              ...
    delete:
        deleteuser:
          dependent: exampleuser
          resource: 1
          response:
            status: 200
            body: |
              ...
```

Dependent test cases will always be executed after the test case they depend on by being put on the same thread, thereby allowing a majority of the test cases to be parallelized.

6.4 Displaying Results

There are several pieces of information that need to be reported to the user to allow for a quick overview of the issues shown by the input/output test cases:

- Number of successful tests
- Number of failed tests
- Names and descriptions of failed tests
- Dependencies of failed tests

This data can be used to alert the user of which endpoints are fully implemented, and which still require further work. Standard practice in the industry is to provide a running tally of 'P"s and 'F"s as a progress bar, followed by more detailed information when the test cases complete. To accomplish this, the results will be stored in the custom Object discussed early by the completion handlers created in Typhoeus.

As the user can be assumed to be technically proficient, there is no need for a graphical user interface: all interaction will be through a command line. This has the added advantage of providing a consistent experience between all operating systems, as well as allowing easy integration with other services which may already exist.

Chapter 7

Evaluation

This section will analyze the completed system based upon the previously stated functional and non-functional requirements, as well as competing solutions on the market.

7.1 Requirements

To ensure that the system has been developed according to the problem laid out at the start of this thesis, it will be evaluated against the requirements outlined earlier.

7.1.1 User Requirements

- *Fully specify a restful web API, so that it can be clearly understood by developers and software architects.* Restful APIs can be fully specified using RATML. Whether it can be clearly understood by developers is subjective, however it is based upon RAML which is widely used, and so it is safe to assume that the language is clear. Livedrive have begun using RATML, further pointing towards the language being useful.

- *Be able to define what 'correct' operation of the API results in.* This was accomplished using schema definition for responses, which can be evaluated against the API implementation. While not perfect - it is possible to create a response which matches the schema but is not valid - it has a 98% rate when the heuristic is fed some seed data. This is an area which could use improvement in the future, but suffices for current use.

- *Automatically run tests against an implementation of the web API based upon the specification.* This was accomplished using the Test Generation and Execution modules: Test

cases are created from a RATML specification. They are sent as requests to the API implementation, and the responses evaluated for correctness. Results can then be reported to the user if a human-readable format.

- *Be able to define sample inputs for the restful web API. / Be able to define sample outputs for a given input.* The RATML extensions to add Test Cases allow multiple inputs and outputs to be defined for each method in a restful API.

- *Be able to check in the API returns the correct output for any given input.* The RATML parser was extended to be able to process Test Cases and transform them into the same format already being used by the Test Execution module.

7.1.2 Functional Requirements

- *Parse the specification into a machine readable format.* A YAML parser was modified to act as a RATML parser, thereby fulfilling this requirement. The parser outputs ruby Objects which can be easily processed by a computer, as well as providing a Facade for easy manipulation by other modules.

- *Provide the user with a method to execute tests against a web API.* The command-line interface allows the user to select a RATML file and begin test execution against its implementation.

- *Report to the user how many tests failed.* The pretty printer can process the test result objects and output how many tests failed, along with many other useful information for the user.

- *Be able to send sample inputs to the restful API and ensure they are valid.* Test Cases can be sent to the API implementation using the Typhoeus library, and the completion handler closures evaluate if the response is valid.

- *Be able to detect if an API implementation does not follow the specification.* The existence and schema checks carried out by the Test Execution module ensure that endpoints are implemented, while the input/output tests can be used for more nuanced probing.

7.1.3 Non-Functional Requirements

Specification

- *The specification language must be easily human readable. / The language must be able to succinctly and unambiguously define a restful web API.* As discussed above, these are subjective requirements. RATML is now being used at Livedrive by several engineers and has yet to receive any complaints about it's design, so this requirement can be considered to be met.

- *The language must be easily convertible to a format machines can process.* RATML is based upon YAML, which is a very mature markup language with many parsers available. While RATML specific parser will have to be written, such as the ruby based one created for this thesis, the process can be drastically simplifies by using an existing YAML parser as the code base.

- *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.* RATML can specify APIs which are then implemented in any language: there are no language specific features within it.

Automated testing

- *The system should be reliable: it should return the same result if run with the same specification on the same implementation.* Due to the added dependency features, tests are run in a non-destructive order allowing the system to be reliable.

- *The system should be implementation agnostic: It should be able to run the tests no matter what language the API is written in.* Test cases are executed as network requests, and therefore do not need to know any implementation details of the API.

- *The system should be able to run on as many common operating systems as feasibly possible.* The system is written in ruby, allowing the same code base to be run on OS X, Windows, most Linux distributions and many other platforms[1].

- *The system should be performant, scaling linearly with the number of tests being run.* Thanks to the parallelized design, the system is highly performant, regularly scaling faster than linearly depending on the number of cpu cores available.

[1] `https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/20SupportedPlatforms`

- *The system should be easy to maintain, following development best practices and making good use of architectural design patterns.* The system is highly modular. There are two major components: the parser and the test framework, both of which are further split into components. There is ample documentation throughout, and the Model-View Controller design pattern is followed. Individual components can be modified or replaced without breaking the overall system.

- *The system should be fully unit tested to provide assurances that the code is as correct as possible.* Every module has unit tests, all of which pass. A sample 'Onyx' API was implemented to model real world usage, and the test framework behaves as expected when executed against it.

7.1.4 Conclusion

While there are a few small areas where improvement is possible, the system generally meets all the requirements laid out at the start. Of particular note is the fact that defining a 'correct' API proved difficult: it is a concept easily understood by humans but can not be simply specified in a machine-computable format. Nevertheless, the input-output test cases can help cover this deficiency and the 98% success rate isn't unusable.

7.2 Current Solutions

As covered in the Background, there are existing products also aiming at automating testing. This section compares the effectiveness of existing solutions to RATML and the Test Framework developed here.

7.2.1 RAML

While RATML is heavily based upon RAML, they are not the same. Both languages are built on top of YAML, thereby allowing them to be easily parsed. RATML has been simplified, allowing specialized parsers to be easier to write, as shown by the fully working RATML parser which I managed to implement, in contrast to the abandoned Ruby RAML parser. On the other hand, RAML possesses many abstraction techniques such as traits that allow more concise specifications to be written, whereas RATML requires everything to be explicitly laid out. While this gives RAML a higher difficulty curve to learn, it allows specifications to be much more easily followed by humans. RATML allows the definitions of input/output test cases.

RAML partially supports this in the form of examples, however the RATML implementation is far more robust, allowing automatic test generation from the Test Case definitions.

As RAML and RATML target different use cases, one can not be said to definitely be better than the other. One major advantage that RAML possesses is the already existing third-party resources and industry backing. Convincing the market to learn a new specification language and rewrite the many existing tools is a difficult task, especially as RAML has a large corporation backing it. However, as shown by the lack of automatic test tooling, RAML *is* limited when it comes to test generation, and this is the use case where RATML clearly wins out.

7.2.2 FitNesse

FitNesse is the test suite most similar to this project. It consists of a wiki-type file used to specify inputs and outputs. The wiki file is then used to generate test cases, and ran against the api implementation. Like RAML, FitNesse has the advantage of existing tooling and industry support. FitNesse and the Test Framework both allow tests to be specified in a human readable manner, then parsed and run by the computer. FitNesse requires prior setup by a software engineer, while the Test Framework can be executed directly: it only needs a RATML file as input. Furthermore, the Test Framework can do presence and schema checks which FitNesse can not - it requires an exact output for any input.

This project's major advantage over FitNesse is ease of use. The specification can be automatically converted into test cases, input does not need to be specified in a special file. Furthermore, FitNesse requires the creation of Mocks and Fixtures to operate properly. The entire api code base must be modified to be compatible with FitNesse while the Test Framework can work 'out of the box.' On the other hand, once Mocks and Fixtures are created, FitNesse allows for a far wider variety of test cases to be written. The Test Framework requires the API implementation to be set up with a test database, while FitNesse can automatically populate with API with mock data. The Test Framework tests the implementation as a whole, while FitNesse can target individual modules.

All in all, the Test Framework provides simple, fully automated test cases. FitNesse can provide more control, at the cost of required manually integration. Both systems can be used as required by a project.

7.3 Livedrive and Onyx

The imitation Onyx API was run against the Test Framework. The results were encouraging: all the tests passed, showing that RATML *can* be used to fully specify an API, and create automatic tests for it.

I also ported the entire Onyx specification definition to RATML, and ran it against the Test Framework. The complete specification has 403 endpoints and just under 5000 test cases. Of these, close to 4000 can be processed by the test Framework. Issues were caused by cases related to authorization and authentication, which were omitted from the imitation API due to trade secrets. While the Test Framework can handle most authorization techniques, it can not handle OAUTH[13] due to the multiple redirects involved.

	Imitation	**Onyx**
Endpoints	10	403
Input/Output Cases	40	4989
Successful Tests	40	3890
Schema Tests	21	10211
Successful Schema Tests	20	10007
Time to run Tests	31ms	5131ms

Discussing the Test Framework with Livedrive engineers, the general consensus is that the system is best in class for it's use case, and RATML is a very nice language to write specifications in. However, it is not fully featured enough to be a complete replacement, and thus will be used alongside FitNesse for the time being. The Test Framework is open source, and I will continue working on it as an employee of Livedrive when this thesis is finished. Livedrive is already starting to implement it as part of their test process, showing that the project meets a need in the industry.

Chapter 8

Conclusion and Future Work

There are a myriad ways of testing an application, and automating all of them is difficult, arguably impossible. This thesis set out to tackle a subset of the tests currently manually conducted in software engineering companies, and I believe it has satisfactorily met this challenge. RATML provides an unambiguous, human and machine readable format for defining restful web APIs, while the Test Framework can transform the specification into a suite of varied tests. The tests can be executed against an implementation of the API, and the results reported to the user. By using popular, open source projects like YAML and ruby as a basis, the project has a good chance of catching on in the industry, and has already been adopted by Livedrive.

The project is obviously not a complete system. Tests are limited to three major categories: Presence, schema and input/output. There are far more possible such as integration, component and functional. Furthermore, scheme test do not yet have a 100% reliability rate, though I do not believe there are any other systems on the market which even attempt schema tests.

RATML and the Test Framework are centred around restful web APIs. While the Test Framework is generic enough that it could conceivably function with other restful services, RATML certainly can not and would need major expansion. These limitations mean that the project is currently only a viable solution for Livedrive and similar companies: it can not be used for other types of applications.

While the requirements for this thesis have been met, I will continue work on the Test Framework. Future extensions include adding support for Mocks and Fixtures. The Framework is currently limited to testing the API implementation as a whole, which is not always beneficial. A heuristic to determine components from the specification and target tests accordingly could help the Test Framework replace FitNesse.

RATML can also be improved. RAML contains constructs to define security mechanisms, and abstraction layers to allow the specification to be split over several files. These can be ported to RATML and implemented in the parser, allowing greater interoperability between RATNL and RAML.

Thanks to it's parallelized design, the application is highly scalable. While it is fast enough for it's current uses, there are several ways to gain more performance from it. The parser can be written in a lower-level, compiled language to speed up RATML parsing. While ruby has proven fast enough so far thanks to manipulating the YAML syntax tree directly, it has been at the cost of increasing memory usage. Writing the entire parser in C++ or Rust could help with this if it ever becomes a bottleneck.

The biggest roadblock for RATML and the Test Framework is adoption. Test suites do not live in isolation, and must consist of an ecosystem to be truly useful. Therefore, the entire project will be open sourced to encourage the community to contribute and improve it. However, it is competing with the already ingrained RAML and FitNesse. The Test Framework meets a specific niche, and that may help it be used alongside other products while the ecosystem builds.

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] Typhoeus. https://github.com/typhoeus/typhoeus. [Online; accessed 12-March-2015].

[5] Ieee standard for software unit testing. *IEEE Std. 1008-1987*, 1986.

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

[7] 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].

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

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

[10] 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.

[11] 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].

[12] 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.

[13] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[14] I. Hickson, R. Berjon S. Faulkner, T. Leithead, E. D. Navara, E. O'Conner, and S. Pfeifer. Html5: A vocabulary and associated apis for html and xhtml. `http://www.w3.org/TR/html5/`, 2014. [Online; accessed 4-February-2015].

[15] Peter Komisar. Ruby on rails i. `http://www.sentex.net/~pkomisar/Ruby/9_RubyOnRails_1.html`. [Online; accessed 12-March-2015].

[16] Trygve Reenskaug and James O. Coplien. The dci architecture: A new vision of object-oriented programming. `http://www.artima.com/articles/dci_vision.html`, 2009. [Online; accessed 1-March-2015].

Appendix A

RATML Specification

A.1 Abstract

RATML is a YAML-based language that describes RESTful APIs. Together with the YAML specification [1], this specification provides all the information necessary to describe RESTful APIs. This specification is largely based upon the RAML 0.8 specification[2].

A.2 Introduction

This specification describes RATML. RATML is a human-readable and machine process-able description of a RESTful API interface. API documentation generators, API client-code generators, and API servers consume a RATML document to create user documentation, client code, and server code stubs, respectively.

A.3 Conventions

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119 [RFC2119].

[1] `http://yaml.org/spec/1.2/spec.html`
[2] `http://raml.org/spec.html`

A.4 Overview

RATML defines the media type ”application/ratml+yaml” for describing and documenting a RESTful API’s resources, such as the resources’ methods and schema. RATML is YAML-based, and RATML documents support all YAML 1.2 features. The recommended filename extension for RATML files is ”.ratml”

RATML also provides facilities for extensively documenting a RESTful API, enabling documentation generator tools to extract the user documentation and translate it to visual formats such as PDF, HTML, and so on.

A.4.1 Terminology

For this specification, **API definition** will be used to denote the description of an API using this specification.

REST is used in the context of an API implemented using the principles of REST. The REST acronym stands for Representational State Transfer and was first introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.

A **resource** is the conceptual mapping to an entity or set of entities.

A.5 Markup Language

This specification uses YAML 1.2 as its core format. YAML is a human-readable data format that aligns well with the design goals of this specification.

RATML API definitions are YAML-compliant documents that begin with a REQUIRED YAML-comment line that indicates the RATML version, as follows:

```
#%RATML
```

The RATML version MUST be the first line of the RATML document. RATML parsers MUST interpret all other YAML-commented lines as comments.

In RATML, the YAML data structures are enhanced to include data types that are not natively supported. All RATML document parsers MUST support these extensions.

In RATML, all values MUST be interpreted in a case-sensitive manner.

A.6 Named Parameters

This RATML Specification describes collections of named parameters for the following properties: URI parameters, query string parameters, form parameters, request bodies (depending on the media type), and request and response headers. All the collections specify the named parameters' attributes as described in this section.

Some named parameters are optional and others are required. See the description of each named parameter.

Unless otherwise noted, named parameter values must be formatted as plain text. All valid YAML file characters MAY be used in named parameter values.

A.6.1 description

(Optional) The **description** attribute describes the intended use or meaning of the parameter.

A.6.2 Type

(Optional) The **type** attribute specifies the primitive type of the parameter's resolved value. API clients MUST return/throw an error if the parameter's resolved value does not match the specified type. If **type** is not specified, it defaults to string. Valid types are:

Type	Description
string	Value MUST be a string.
number	Value MUST be a number. Indicate floating point numbers as defined by YAML.
integer	Value MUST be an integer. Floating point numbers are not allowed. The integer type is a subset of the number type.
date	Value MUST be a string representation of a date.
boolean	Value MUST be either the string "true" or "false" (without the quotes).

Date Representations

All date/time stamps are represented in Greenwich Mean Time (GMT), which for the purposes of HTTP is equal to UTC (Coordinated Universal Time). This is indicated by including "GMT" as the three-letter abbreviation for the timezone. Example: Sun, 06 Nov 1994 08:49:37 GMT.

A.6.3 enum

(Optional, applicable only for parameters of type string) The **enum** attribute provides an enumeration of the parameter's valid values. This MUST be an array. If the **enum** attribute

is defined, API clients and servers MUST verify that a parameter's value matches a value in the **enum** array. If there is no matching value, the clients and servers MUST treat this as an error.

A.6.4 pattern

(Optional, applicable only for parameters of type string) The **pattern** attribute is a regular expression that a parameter of type string MUST match. Regular expressions MUST follow the regular expression specification from ECMA 262/Perl 5. The pattern MAY be enclosed in double quotes for readability and clarity.

A.6.5 minLength

(Optional, applicable only for parameters of type string) The **minLength** attribute specifies the parameter value's minimum number of characters.

A.6.6 maxLength

(Optional, applicable only for parameters of type string) The **maxLength** attribute specifies the parameter value's maximum number of characters.

A.6.7 minimum

(Optional, applicable only for parameters of type number or integer) The **minimum** attribute specifies the parameter's minimum value.

A.6.8 maximum

(Optional, applicable only for parameters of type number or integer) The **maximum** attribute specifies the parameter's maximum value.

A.6.9 required

(Optional except as otherwise noted) The **required** attribute specifies whether the parameter and its value MUST be present in the API definition. It must be either 'true' if the value MUST be present or 'false' otherwise.

In general, parameters are optional unless the **required** attribute is included and its value set to 'true'.

For a URI parameter, the **required** attribute MAY be omitted, but its default value is 'true'.

A.6.10 default

(Optional) The **default** attribute specifies the default value to use for the property if the property is omitted or its value is not specified. This SHOULD NOT be interpreted as a requirement for the client to send the **default** attribute's value if there is no other value to send. Instead, the **default** attribute's value is the value the server uses if the client does not send a value.

A.7 Basic Information

This section describes the components of a RATML API definition.

A.7.1 Root Section

The root section of the format describes the basic information of an API, such as its title and base URI, and describes how to define common schema references.

RATML-documented API definition properties MAY appear in any order.

This example shows a snippet of the RATML API definition for the GitHub v3 public API.

```
#%RATML
title: GitHub API
version: v3
baseUri: https://API.github.com
mediaType:  application/json
schemas:
  - User:  schema/user.json
    Users: schema/users.json
    Org:   schema/org.json
    Orgs:  schema/orgs.json
```

A.7.2 API Title

(Required) The **title** property is a short plain text description of the RESTful API. The **title** property's value SHOULD be suitable for use as a title for the contained user documentation.

A.7.3 API Version

(Optional) If the RATML API definition is targeted to a specific API version, the API definition MUST contain a **version** property.

The API architect MAY use any versioning scheme so long as version numbers retain the same format. For example, ”v3”, ”v3.0”, and ”V3” are all allowed, but are not considered to be equal.

A.7.4 Base URI

(Optional during development; Required after implementation) A RESTful API's resources are defined relative to the API's base URI. The use of the **baseUri** field is OPTIONAL to allow describing APIs that have not yet been implemented. After the API is implemented (even a mock implementation) and can be accessed at a service endpoint, the API definition MUST contain a **baseUri** property. The **baseUri** property's value MUST conform to the URI specification [RFC2396] or a Level 1 Template URI as defined in RFC 6570 [RFC6570].

The **baseUri** property SHOULD only be used as a reference value. API client generators MAY make the **baseUri** configurable by the API client's users.

A.7.5 Protocols

(Optional) A RESTful API can be reached via HTTP, HTTPS, or both. The **protocols** property MAY be used to specify the protocols that an API supports. If the **protocols** property is not specified, the protocol specified at the **baseUri** property is used. The **protocols** property MUST be an array of strings, of values ”HTTP” and/or ”HTTPS”.

```
#%RATML
title: Salesforce Chatter REST API
version: v28.0
protocols: [ HTTP, HTTPS ]
baseUri: https://na1.salesforce.com/services/data/{version}/chatter
```

A.7.6 Resources and Nested Resources

Resources are identified by their relative URI, which MUST begin with a slash (/).

A resource defined as a root-level property is called a **top-level resource**. Its property's key is the resource's URI relative to the baseUri. A resource defined as a child property of another resource is called a **nested resource**, and its property's key is its URI relative to its parent resource's URI.

This example shows an API definition with one top-level resource, /gists, and one nested resource, /public.

```
#%RATML
title: GitHub API
version: v3
baseUri: https://API.github.com
/gists:
  /public:
    displayName: Public Gists
```

Every property whose key begins with a slash (/), and is either at the root of the API definition or is the child property of a resource property, is a resource property. The key of a resource, i.e. its relative URI, MAY consist of multiple URI path fragments separated by slashes; e.g. "/bom/items" may indicate the collection of items in a bill of materials as a single resource. However, if the individual URI path fragments are themselves resources, the API definition SHOULD use nested resources to describe this structure; e.g. if "/bom" is itself a resource then "/items" should be a nested resource of "/bom", while "/bom/items" should not be used.

Display Name

The **displayName** attribute provides a friendly name to the resource and can be used by documentation generation tools. The **displayName** key is OPTIONAL.

Description

Each resource, whether top-level or nested, MAY contain a **description** property that briefly describes the resource. It is RECOMMENDED that all the API definition's resources includes the **description** property.

Template URIs

Template URIs containing URI parameters can be used to define a resource's relative URI when it contains variable elements. The following example shows a top-level resource with a key **/jobs** and a nested resource with a key **/jobId**:

```
#%RATML
title: ZEncoder API
version: v2
baseUri: https://app.zencoder.com/API/{version}
/jobs: # its fully-resolved URI is https://app.zencoder.com/API/{version}/jobs
  displayName: Jobs
```

```
  description: A collection of jobs
  /{jobId}: # its fully-resolved URI is https://app.zencoder.com/API/{version}/jobs/{jobId}
    description: A specific job, a member of the jobs collection
```

The values matched by URI parameters cannot contain slash (/) characters, in order to avoid ambiguous matching. In the example above, a URI (relative to the baseUri) of "/jobs/123" matches the "/jobId" resource nested within the "/jobs" resource, but a URI of "/jobs/123/x" does not match any of those resources.

Absolute URI

Absolute URIs are not explicitly specified. They are computed by starting with the baseUri and appending the relative URI of the top-level resource, and then successively appending the relative URI values for each nested resource until the target resource is reached.

Taking the previous example, the absolute URI of the public gists resource is formed as follows:

```
   "https://API.github.com"              <--- baseUri
              +
            "/gists"                     <--- gists resource relative URI
              +
            "/public"                    <--- public gists resource relative URI
              =
"https://API.github.com/gists/public"    <--- public gists absolute URI
```

A nested resource can itself have a child (nested) resource, creating a multiply-nested resource.

In this example, /user is a top-level resource that has no children; /users is a top-level resource that has a nested resource, /userId; and the nested resource, /userId, has three nested resources, /followers, /following, and /keys.

```
#%RATML
title: GitHub API
version: v3
baseUri: https://API.github.com
/user:
/users:
  /{userId}:
    uriParameters:
      userId:
        type: integer
    /followers:
    /following:
    /keys:
      /{keyId}:
        uriParameters:
```

```
    keyId:
      type: integer
```

The computed absolute URIs for the resources, in the same order as their resource declarations, are:

```
https://API.github.com/user
https://API.github.com/users
https://API.github.com/users/{userId}
https://API.github.com/users/{userId}/followers
https://API.github.com/users/{userId}/following
https://API.github.com/users/{userId}/keys
https://API.github.com/users/{userId}/keys/{keyId}
```

Methods

In a RESTful API, **methods** are operations that are performed on a resource. A method MUST be one of the HTTP methods defined in the HTTP version 1.1 specification and its extension, RFC5789.

Description

Each declared method MAY contain a **description** attribute that briefly describes what the method does to the resource. It is RECOMMENDED that all API definition methods include the **description** property.

This example shows a resource, /jobs, with POST and GET methods (verbs) declared:

```
#%RATML
title: ZEncoder API
version: v2
baseUri: https://app.zencoder.com/API/{version}
/jobs:
  post:
    description: Create a Job
  get:
    description: List Jobs
```

Query Strings

An API's resources MAY be filtered (to return a subset of results) or altered (such as transforming a response body from JSON to XML format) by the use of query strings. If the resource or its method supports a query string, the query string MUST be defined by the **queryParameters** property.

The **queryParameters** property is a map in which the key is the query parameter's name.

```
#%RATML
title: GitHub API
version: v3
```

```
baseUri: https://API.github.com
/users:
  get:
    description: Get a list of users
    queryParameters:
      page:
        type: integer
      per_page:
        type: integer
```

Body

Some method verbs expect the resource to be sent as a request body. For example, to create a resource, the request must include the details of the resource to create.

A method's body is defined in the **body** property as a hashmap, in which the key MUST be a valid media type.

This example shows a snippet of the Zencoder API's Jobs resource, which accepts JSON as input:

```
/jobs:
  post:
    description: Create a Job
    body:
      application/json: !!null
```

Schema

The structure of a request or response body MAY be further specified by the **schema** property under the appropriate media type.

The **schema** key CANNOT be specified if a body's media type is **application/x-www-form-urlencoded** or **multipart/form-data**.

All parsers of RATML MUST be able to interpret JSON Schema.

Schema MAY be declared inline or in an external file. However, if the schema is sufficiently large so as to make it difficult for a person to read the API definition, or the schema is reused across multiple APIs or across multiple miles in the same API, the !include user-defined data type SHOULD be used instead of including the content inline.

This example shows an inline schema declaration.

```
/jobs:
  displayName: Jobs
  post:
    description: Create a Job
    body:
      application/json:
        schema: |
          {
```

```
    "$schema": "http://json-schema.org/draft-03/schema",
    "properties": {
        "input": {
            "required": false,
            "type": "string"
        }
    },
    "required": false,
    "type": "object"
}
```

Responses

Resource methods MAY have one or more responses. Responses MAY be described using the **description** property, and MAY include **example** attributes or **schema** properties.

This example shows a definition for a GET response of 200.

```
/media/popular:
  displayName: Most Popular Media
  get:
    description: |
      Get a list of what media is most popular at the moment.
    responses:
      200:
        body:
          application/json:
            example: !include examples/instagram-v1-media-popular-example.json
```

Responses MUST be a map of one or more HTTP status codes, where each status code itself is a map that describes that status code.

Each response MAY contain a **body** property, which conforms to the same structure as request **body** properties. Responses that can return more than one response code MAY therefore have multiple bodies defined.

A.8 Test Cases

A Test Case is a mapping of input data to a specific output. Methods MAY contain one or more Test Cases. Test Cases MUST have a short name and MAY have a longer description. They MAY specify query parameters. POST, PUT and PATCH Test Cases MAY specify a JSON query. Element Test Cases MUST specify a resource number. Responses MUST describe the returned status code and MAY describe a JSON body.

This example shows a definition for a Test Case creating a user.

```
\users:
    post:
```

```
testcases:
  createuser:
    query: |
      {
        "name" : "Alan Turing",
        "photo": "/photos/2131"
      }
    response:
      status: 200
      body: |
        {
          "name": "Alan Turing",
          "uri": "/users/21",
          "photo": "/photos/2131"
        }
```

A Test Case MAY be dependent on another test case. If a Test Case is dependent, the **dependent** property will reference the parent Test Case.

Appendix B

Mock Onyx Specification

```
#%RATML

title: Onyx API
baseUri: http://example.API.com/{version}
version: v1
/users:
  get:
    responses:
        200:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A list of users",
                  "properties": [{
                          "name": { "type": "string"},
                          "uri": { "type": "uri"},
                          "role": { "type:" "string" }
                  }],
                  "required": [ "name", "uri", "role" ]
                }
  /{userId}:
    get:
      responses:
        200:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A canonical user",
                  "properties": {
                    "name":  { "type": "string" },
                    "uri": { "type": "uri" },
                    "photo": { "type": "uri" },
```

```
                    "files": [{
                            "filename": { "type": "string"},
                            "uri": { "uri": "uri"},
                            }]
                  },
                  "required": [ "name", "uri" ]
                }
        404:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A missing user",
                  "properties": {
                    "message":  { "type": "string" }
                  },
                  "required": [ "message" ]
                }
    delete:
      description: |
        This method will *delete* an **individual user**
      responses:
        200:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "properties": {
                    "message":  { "type": "string" }
                  },
                  "required": [ "message" ]
                }
/files:
  get:
    responses:
      200:
        body:
          application/json:
            schema: |
               { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A list of files",
                  "properties": [{
                          "filename": { "type": "string"},
                          "uri": { "type": "uri"},
                          "size": { "type:" "long" }
                  }],
                  "required": [ "filename", "uri" ]
                }
  /{fileId}:
    get:
      responses:
        200:
```

```
    body:
      application/json:
        schema: |
          { "$schema": "http://json-schema.org/schema",
            "type": "object",
            "description": "A canonical file",
            "properties": {
              "filename":  { "type": "string" },
              "location": { "type": "uri" },
              "size": { "type": "uri" },
            },
            "required": [ "filename", "location" ]
          }
  404:
    body:
      application/json:
        schema: |
          { "$schema": "http://json-schema.org/schema",
            "type": "object",
            "description": "A missing file",
            "properties": {
              "message":  { "type": "string" }
            },
            "required": [ "message" ]
          }
delete:
  description: |
    This method will *delete* an **individual file**
  responses:
    200:
      body:
        application/json:
          schema: |
            { "$schema": "http://json-schema.org/schema",
              "type": "object",
              "properties": {
                "message":  { "type": "string" }
              },
              "required": [ "message" ]
            }
```