View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Trepo - Institutional Repository of Tampere University

Tampereen yliopisto

Mikko Vänskä

AUTOMATED TESTING FOR MICRO-SERVICES

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
May 2019

ABSTRACT

Mikko Vänskä: Automated testing for microservices
Tampere University
Master of Science Thesis, 50 pages, 2 Appendix pages
May 2019
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: Professor Hannu-Matti Järvinen, MSc. Jarkko Mikkola

---

This thesis discusses the topic of automated testing as it relates to microservice systems. Microservice architecture is a highly scalable way of designing and implementing online applications. Since microservice applications are network-based applications by nature, testing them has to also happen in a network environment. Automating tests for this kind of environment involves generating artificial network traffic, often in the form of HTTP requests to a network API of some kind, like a REST API. These topics are discussed from a test design and implementation point of view, along with main features of the microservice architecture and automated testing in general.

The main part of this thesis describes and documents the process of designing and implementing a test automation framework for Intel Insight, an automatic image storage and photogrammetry processing platform that is implemented as a microservice system. The framework design involves setting initial requirements for potential automation tools and finding and evaluating candidates for the task. In the end, the framework core is formed by automation tools Postman, Selenium, and SikuliX. The use of this combination for test automation purposes is examined by looking at how to the tools can be used to automate a core use case of the Intel Insight platform.

The resulting framework was found to be well-suited and versatile enough for its intended purpose. The tools of the framework had a low barrier of entry to them and as such were easy to begin working with and to integrate automated test cases implemented with them to Continuous Integration systems Gitlab CI and Jenkins. All tools are reviewed in-depth, and positives and negatives of each individual automation tool that were encountered during test implementation are analyzed. The main negatives are brought up as possible ideas for future development of each tool, enabled by the fact that they are all open-source projects.

Keywords: Test automation, Microservice, REST, Postman, Selenium, SikuliX

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Mikko Vänskä: Testiautomaatio mikropalvelujärjestelmälle
Tampereen Yliopisto
Diplomityö, 50 sivua, 2 liitesivua
Toukokuu 2019
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Professori Hannu-Matti Järvinen, DI Jarkko Mikkola

---

Tämä Diplomityö tutkii automaattisen testauksen hyödyntämistä mikropalveluarkkitehtuurilla toteutettujen sovellusten testaamisessa. Mikropalveluarkkitehtuuri on helposti skaalautuva tapa suunnitella ja toteuttaa Internet-pohjaisia sovelluksia. Koska mikropalvelusovellukset käyttävät tietoverkkoja sovelluksen komponenttien sisäiseen kommunikaatioon, niiden testaaminen tapahtuu myös verkkoympäristössä. Automatisoitu testaaminen tällaisessa ympäristössä tarkoittaa keinotekoisen verkkoliikenteen luomista, tyypillisesti HTTP-kutsujen muodossa jonkinlaiseen verkkorajapintaan, kuten REST-rajapintaan. Näitä teoria-asioita esitellään työssä testien suunnittelemisen ja toteuttamisen näkökulmasta, kuten myös automaattisen testaamisen yleisiä piirteitä.

Pääosa työstä kuvaa testikehyksen suunnittelun ja toteuttamisen prosessia mikropalveluarkkitehtuurilla toteutetun Intel Insightin, kuvien varastoinnin ja automaattisen fotogrammetrisen prosessoinnin tarjoavan palvelun, testaamiseen. Testikehyksen suunnittelu sisältää vaatimusten asettamisen potentiaalisille automaatiotyökaluille ja kandidaattien etsimisen ja arvioinnin vaatimusten perusteella. Työkaluiksi valikoituivat Postman, Selenium ja SikuliX. Tämän yhdistelmän käyttöä automaattiseen testaamiseen tutkitaan automatisoimalla yksi Intel Insightin tärkeimmistä käyttötapauksista.

Työn tuloksena syntynyt testikehys todettiin käytössä tarkoitukseen sopivaksi ja tarpeeksi mukautuvaksi suunniteltuun käyttöön. Käytetyt työkalut osoittautuivat aloittelijaystävällisiksi ja niillä tehdyt automaattiset testitapaukset olivat helppoja integroida käytettyihin jatkuvan integraation alustoihin Gitlab CI ja Jenkins. Yksittäisten työkalujen hyvät ja huonot puolet analysoidaan yksityiskohtaisesti käyttökokemusten perusteella. Huonoja puolia tuodaan esille mahdollisina jatkokehitysideoina työkaluille, jotka niiden avoimeen lähdekoodiin perustuen ovat mahdollisia jalostaa paremmiksi.

Avainsanat: Testiautomaatio, mikropalvelu, REST, Postman, Selenium, SikuliX

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

PREFACE

This thesis is the final chapter on a journey through the Finnish education system that I started in August 1997. My family has been a constant supportive presence for me during the years and they have always been encouraging me to chase higher education. Now it is time to move on to other challenges in personal and professional life.

I would like to thank the people at Intel Finland who made it possible for me to create this thesis in a rapid fashion: Niko Rantalainen, who gave me the opportunity to work for Intel Finland and allowed me to lavishly spend work hours for writing this thesis; Jarkko Mikkola for support and mentorship during my tenure; and finally the group of interns in our team, with whom I had the pleasure of working with, for peer support and camaraderie during the time we shared at the company.

I would like to thank the examiners of this thesis for feedback that helped me polish this work into its final form: Professor Hannu-Matti Järvinen from Tampere University and Jarkko Mikkola from Intel Finland.

Special thanks are in order to LK, who was a crucial mental crutch to lean on during the writing process.

Tampere, 22.5.2019

Mikko Vänskä

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
CI	Continuous Integration
DevOps	Development Operations
DNS	Domain Name System
FTP	File Transfer Protocol
GUI	Graphical User Interface
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IP	Internet Protocol
ISTQB	International Software Testing Qualifications Board
JSON	JavaScript Object Notation
MD5	MD5 Message-Digest Algorithm
OAS	OpenAPI Specification
OSI	Open Systems Interconnection
REST	Representational State Transfer
RFC	Request For Comments
SaaS	Software as a Service
SUT	System Under Test
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

1. INTRODUCTION

Throughout the 2010s, the ongoing development and improvement of cloud computing infrastructure have led to the software business moving increasingly to that domain. More and more software is now being accessed through web browsers, following the Software as a Service (SaaS) business model where the application is not delivered to customers as a locally executable program, but as a subscription to a web-based platform accessed with a browser. Business research company Gartner estimated in September 2018 the global SaaS revenues to nearly double over the period of 2017 to 2021, increasing from 58,8 to 113,1 milliard U.S. Dollars with the whole cloud business revenue increasing from 145,3 to 278,3 milliard during that timeframe [1].

To facilitate this shift in the business domain, new software architectures and development methodologies have emerged to answer the needs of perpetually online and dynamically according to demand scaling applications. Microservice architecture, a modern interpretation of Service-oriented architecture, is becoming the de-facto way of designing large scale online applications. Some examples that have been developed with microservice architecture are content streaming services Netflix, Spotify, and Twitch.tv and they have user bases measured in tens or hundreds of millions, and millions of concurrent users. Microservice applications are often developed by utilizing DevOps practices, which merge application development and operations teams together to shorten software delivery cycles and maintain the live application.

Software testing methods also have to evolve to support these new development trends. Delivery of Internet-based applications, such as microservice systems, to their end-users differ fundamentally from traditional, locally executed applications. Communication within the application takes place over a network through web interfaces instead of within local memory. For this reason, the need to understand the fundamental Internet technologies, such as HTTP, is a requirement for efficient test design. Shorter delivery cycles create a need to automate testing as much as possible in order to keep up with the overall pace of software development. In this context, test automation involves a lot of programmatic web traffic generation and the use of user interfaces through a web browser.

The goal of this thesis is to explore solutions to automated testing at different levels of abstraction for Intel Insight, an image data storage and automatic photogrammetry processing platform that is implemented by using microservice architecture. The findings from the research on the topic are then used in finding and selecting tools in order to create a test framework. The use of the tools is examined by looking at automating a real-life use case.

The primary research methodology used in this thesis is exploratory research. The end goal is known at the start of the process, and the research is done to find means of reaching that goal. The research is conducted with the end goal in mind, and findings along the way are evaluated based on how they help to achieve the goal.

This thesis is organized in the following way: Chapter 2 presents general technologies used in the Internet and related architectures. In Chapter 3 the topic of test automation is examined in-depth. Chapter 4 documents the design process of a test automation framework and introduces Intel Insight, the target microservice system it was designed to test. Chapter 5 explores using the tools in real-life testing use via an example use case automation. In Chapter 6 the framework is reviewed and learnings from using it are discussed from the test design and implementation point of view. Chapter 7 concludes the thesis and ties together all the topics discussed and presents ideas for future improvement for the framework.

2. WEB ARCHITECTURE

In order to have sufficient competence in designing tests for network-based applications, a grasp of fundamental Internet technologies is required. This chapter presents theoretical background of core Internet protocol Hypertext Transfer Protocol (HTTP), introduces Representational State Transfer (REST), a related methodology of designing network system interfaces, takes a look at an widely used way of documenting web interfaces in the form of Open API specification, and in the end talks about main features of the microservice architecture.

2.1 Hypertext Transfer Protocol

The original idea behind HTTP is generally credited to Tim Berners-Lee, who wrote the original proposal of the protocol in 1989 while working for CERN [2] and in 1991 the first formal specification, later named HTTP/0.9 [3]. The original protocol is minimal and defines just a simple request-response communication scheme between a client application and a server in order to retrieve HTML files.

Limitations of this scheme quickly led to the early web browser and server program developers to implement new features, of which the most widely implemented ones were gathered into an unofficial specification HTTP/1.0 in May 1996 [4], and later into an official HTTP/1.1 specification in January 1997 [5]. The final version of the HTTP/1.1 was released in June 2014 [6]. The next major HTTP version is HTTP/2, released as an official specification in May 2015 [7]. HTTP/2 was created to address many performance issues of the older HTTP versions by using underlying network protocols (mostly TCP related things) more efficiently. This chapter mostly discusses topics presented in the HTTP/1.1 specification, as it introduced the main parts of the request and response communication scheme and other key components currently used in the protocol.

By the definition of the OSI model [8], HTTP is an application-level protocol used for data transfer over the Internet. The protocol design is flexible and allows the creation of custom extensions. HTTP presumes that it is used over a reliable transport level protocol [9]. The TCP protocol is used as the default protocol at the transport layer, although the specification does not rule out the use of other transport protocols to transmit HTTP traffic.

2.1.1 Communication scheme

Communication over HTTP can be simplified into the following sequence: first, an HTTP client application sends an *HTTP request* to an HTTP server to perform some operation. The server reads the request, performs the requested operation if the client is allowed to request such an operation and finally sends an *HTTP response* containing information about the results back to the client and closes the connection. All communication is sent as a sequence of plain ASCII characters.

HTTP is a *stateless* protocol, meaning that any pair of requests on the same connection are not linked together in any way, and an HTTP server is not required to keep any information regarding connections made to the server. Any request should contain enough context for a server to understand the request without using any previously stored state on the server. However, the server may store session data to some external storage (like a database), for example in order to implement an authentication scheme to determine if the client sending the request has sufficient access right to perform such an operation.

HTTP requests are targeted to a single *resource* on the server. Resources are stored on a server as a piece of data representing the current state of the modeled resource. According to RFC 3986, “a resource can be anything that has an identity” [10], but generally, in the context of HTTP, a resource is some location on a server that data can be retrieved from or delivered to.

Resources are identified using *Uniform Resource Identifiers* (URI) that define explicitly the targeted resource in the namespace where the resource exists. In the HTTP context, the URI is usually given as a *Uniform Resource Locator* (URL), which is a specific type of a URI. A URL defines the protocol that is used (some common ones are HTTP, HTTPS, and FTP), DNS name of the server that contains the targeted resource (referred to as *host*, as DNS hostnames are generally used instead of raw IP addresses), optionally the network port the request is sent to (if omitted, default TCP port 80 will be used), the path to the resource on the host, and optional request parameters as key-value pairs. A detailed breakdown of an example URL is given in Table 1.

Table 1. Breakdown of a URL into components

Full URL	https://poprock.tut.fi:443/group/pop/etusivu
Protocol	https:
(Separator)	// (no contextual use, required by the URI specification)
Domain name	poprock.tut.fi

Connection port (optional)	:443 (if omitted, default port associated with the protocol is used, for example 80 for HTTP, and 443 for HTTPS)
Resource path	/group/pop/etusivu
Parameters	Additional data to send along with the request appended to the resource path. Example: ?key1=value1&key2=value2

2.1.2 Request and Response structure

By the definition of RFC 2616 [9], an HTTP request consists of four parts: a *start line*, *message headers*, an empty line, and an optional *message body*. The start line has three elements, first is the *request method* used, followed by the *request target* and finally the HTTP version that is used. Message headers are a list of key-value pairs containing more detailed information about the request and how the server should process the request. The list of headers is followed by an empty line (a single carriage return character), indicating the end of the header list and the beginning of the optional message body that contains the actual data sent to the server, if there is any. Many HTTP requests are simple data retrieval from a server, and as such do not require anything other than the request method and target to be completed successfully.

HTTP responses are nearly identical to HTTP requests by their structure but differ by the first element which is called the *status line*. The status line has three elements: the HTTP version used, the *status code*, and the *reason phrase*. The status code is a three-digit code describing the result of the request, followed by a short human-readable reason phrase associated with the response code. The status line is followed by *response headers*, an empty line, and an optional *response body*, just like in HTTP requests. An example of an HTTP request and response is shown in Figure 1 below.

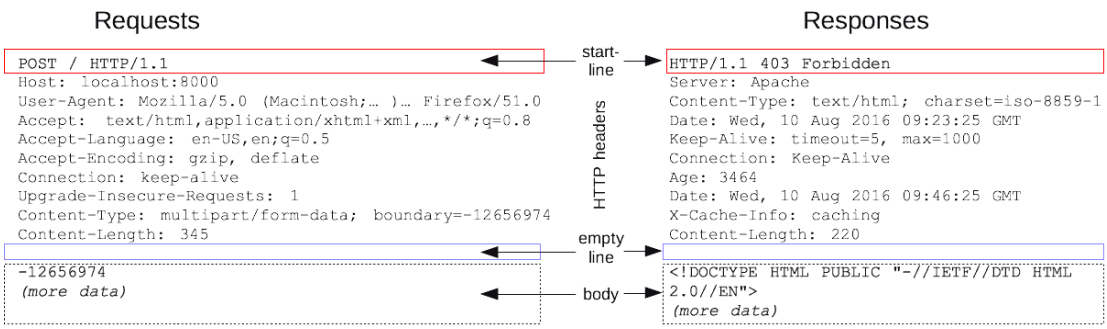

Figure 1. *General HTTP message structure [11]*

2.1.3 Request methods

There are eight HTTP request methods that are officially specified in the HTTP/1.1. The specification allows the implementation of new methods, but only the officially specified ones, listed in Table 2 below, are required to be recognized while communicating.

Table 2. List of HTTP methods

Method	Introduced in version	General use
GET	HTTP/0.9	Retrieve a resource from server
HEAD	HTTP/1.0	GET without response body
POST	HTTP/1.0	Send a resource to server
PUT	HTTP/1.1	Send a resource to the server to be placed in the suggested path
DELETE	HTTP/1.1	Remove a resource from the server permanently
OPTIONS	HTTP/1.1	Query supported HTTP methods
TRACE	HTTP/1.1	Echoes the received request back to the client
CONNECT	HTTP/1.1 (2014 revision)	Instruct a proxy server to create a tunnel

GET method is simply a client (e.g., a web browser) asking the server to send back the targeted resource (e.g., a web page). The request generally doesn't include a body.

HEAD method is used like the GET method, the difference is that the server sends back only the response headers and leaves out the response body.

POST method requests the server to store whatever entity the request contains in its body into the targeted location. The server has full freedom on where the requested entity is eventually stored, or may reject the request outright.

PUT method works just like the POST method, but here the client provides the server a suggested path to store the requested entity. If the request succeeds, the targeted resource on the server is replaced with the resource specified in the request body. This method can be used to update a resource, by targeting an existing resource and sending an updated version to the server.

DELETE method is used to request the targeted resource to be removed from the server. With this method, there is no guarantee to the client that the resource is actually deleted by the server, but the server should reply with a successful status code only if the resource will be deleted.

OPTIONS method is sent to the server to discover what methods it supports for the targeted resource.

TRACE method is a simple “Echo” –type request, to which the server replies with the exact request it received. This method is generally used to debug how intermediary relays alter the HTTP request on its way to the server and has little use outside of that.

CONNECT method is used to instruct a proxy server to connect to another location in order to tunnel a remote connection.

Standard HTTP methods have been defined to have three common properties, and methods can be categorized by how they relate to these properties [12].

Safe methods are “read-only” operations by their defined nature. In practice, this means that the method should only result in the requested data being sent to the client and should not have other side effects on the system state. A notable exception to this is server-side logging, which is not considered an unsafe side effect. Safe methods are defined to be GET, HEAD, OPTIONS, and TRACE.

Idempotent methods have the same effect on the system state as a whole regardless of how many times an identical action is performed. By definition, all safe methods are considered idempotent along with PUT and DELETE methods. This property becomes important when communication failures occur and it is unclear whether the original request was delivered to the receiving end, in which case the request can be repeated with predictable results. For example, PUT is an idempotent method because the target resource is replaced with the entity supplied in the request body if the request is successful, and therefore has the same result each time. The same applies to the DELETE method, as removing the same resource multiple times leads to the resource being deleted on the first request and the next ones having no effect. The end result is that the target resource does not exist anymore.

Cacheable methods have responses to them that can be stored and used later instead of re-doing the original request. RFC 7231 [12] defines GET, HEAD and POST as cacheable methods, although it is stated that “the overwhelming majority of cache implementations only support GET and HEAD."

2.1.4 Response codes

HTTP status codes are generally grouped into five categories signifying the results of the processed request. All HTTP clients should recognize these categories, even if the specific status code is not supported by the client. Custom status codes may be implemented, but generally, only a small number of the status codes are used widely. Clients are generally not required to present the response code to the user, but in many error situations, it is generally done to show the user some human-readable information about what happened [12]. A list of status code classes along with some examples are in Table 3 below.

Table 3. HTTP status codes

Status **code class**/examples	Description
1xx **Informational**	Request was received and understood
101 Switching Protocols	The client requested to switch protocols, and the server agreed to do so
2xx **Success**	Request was received and successfully processed
200 Ok	Standard/default response for a successful request
201 Created	The requested resource was created on the server
204 No Content	Request was successfully processed, no response body is sent
3xx **Redirection**	Client needs to do additional actions to perform the request
301 Moved permanently	The targeted resource has been moved to another location, which is included in the response
4xx **Client Errors**	The request had errors that were likely caused by the client
401 Bad request	The request contained invalid data, and was rejected
404 Not Found	The request target does not exist
5xx **Server Errors**	The server encountered an error on its end and could not process the request

500 Internal Server Error	A generic error response to unexpected error conditions on the server

2.2 Representational State Transfer

Representational State Transfer, REST, was originally presented by Roy Fielding, one of the authors of the HTTP/1.1 specification, in his doctoral dissertation at University of California in the year 2000 [13]. REST is intended to be a framework for designing interfaces for Internet-scale distributed hypermedia systems. More specifically, REST relates to how a server interacts with its clients to receive data and relay it back to them as representations of the underlying data model of the application [13]. Many APIs claim to be REST APIs (often called RESTful APIs), Fielding himself has a firm stance that an API must fully implement the REST scheme in order to be called a true REST API.

In chapter five of the dissertation [13], the concept of REST is derived using by applying a series of design constraints, resulting in the overall architectural style. One thing to note about the concept of REST is that it describes an architectural style and does not tie it to any specific technologies or protocols. In the dissertation, an entire chapter is dedicated to how the style applies to the utilization of HTTP when interacting with a REST API, but any technology utilizing URI resource referral schema can be considered RESTful as long as the REST design concepts are being followed [13].

2.2.1 Architectural style

The six guiding constraints of the REST are (in the order they are presented in the original dissertation) are client-server architecture, statelessness, cacheability, uniform interface, layered system, and code-on-demand (optional). Many of these constraints share properties with HTTP. The list below references points made in Fielding's dissertation [13].

- Client-Server architecture is a commonly used method of implementing distributed systems. The architecture separates client application from the server application, allowing both to be developed independently from each other at their own pace.
- Statelessness implies that all communication must be done in a way that the request contains all the required information and context for it to be successfully processed by the server.
- Cacheability means that server responses can be marked as cacheable or non-cacheable, giving or denying the client permission to store and use the response

data later without having to request it again from the server. This potentially reduces the need to communicate in some cases, improving communication efficiency, or as Fielding notes in his dissertation: “An interesting observation is that the most efficient network request is one that doesn’t use the network.” The downside of caching is that it introduces the possibility that cached data becomes inconsistent with the data on the server.

- Uniform interface means that all parts used in the system share the same interface, leading to standardized communication styles and formats within the system. The drawback is that specialized communication methods are not allowed, leading possibly to degraded performance as the used communication scheme might not be optimal for all system components.
- Layered system asserts that the full structure of the system is not visible to any component of the system, meaning all parties involved see only the ones they are directly communicating with.
- Code-On-Demand is an optional constraint but is a key part in modern web applications. It means that a client may extend its functionality by retrieving and executing code supplied directly by the server, such as JavaScript files. This allows flexible and minimal clients that can obtain the required application code as needed. An example of this is modern web browsers, which implement enough logic to do HTTP requests and a platform to execute JavaScript code fetched from a server on a case-by-case basis.

REST APIs are usually documented as a list of resources that are accessible (often referred to as *endpoints* of the API), the HTTP methods that are supported for each endpoint, expected data formats for each request to an endpoint and possibly example requests and responses associated with each endpoint. REST APIs usually implement some of the CRUD (Create, Read, Update, Delete) operations for all endpoints, with POST method used to create resources on the server, GET method to read/retrieve resources from the server, PUT method to update resources on the server and DELETE method to delete resources from storage on the server. A simple example of this is in Table 4 below.

Table 4. A simple REST API example

	http://example.com/api/customers	***http://example.com/api/customers/{id}***
GET	Retrieve a list of all customers’ data	Retrieve data of a specific customer
POST	Create a new customer from the data in the request body, server creates an ID for the new customer and sends it in response body if successful	Create a new sub-resource associated with the customer, server creates an ID for the new resource and sends it in response body if successful
PUT	Replace the entire list of customers with the data in the response body, or create a new resource it the list doesn’t exist	Replace the data of the customer with the one in the request body, or create a new one if such a customer doesn’t exist
DELETE	Delete all customer information from the server	Delete the information of the customer from the server

2.2.2 HATEOAS

The concept of HATEOAS, Hypermedia As The Engine Of Application State, is an integral part of the REST architecture. Fielding talks about this concept in his blog in a post titled “REST APIs must be hypertext-driven” [14]. One of the main points Fielding makes in the text related to HATEOAS is that when a client starts to communicate with a REST API it should not need to know anything other than the initial URI to connect to the API and a set of standardized media types that are relevant to the users of the API in order to handle and present the data properly to end users.

In practice, the utilization of HATEOAS means that the client relies on the server to provide the available options on how to continue using the service. Given a list of available operations, the client then chooses how to continue interacting with the service, or stop using it altogether. The client is always the entity storing the current application state and is responsible for driving the operation forward.

A similar concept is how the Internet presents itself to human users: a browser is used to retrieve a web page (in the form of an HTML document) from a server, and the response rendered by the browser presents to the user the requested content and the available options to proceed, often in the form of links to other pages. The user does not have to know beforehand anything other than the URL of the main page to use the service successfully, as the server supplies all the required information during the use of the service. The user can also bypass the main entry point of the server completely by using direct URLs to go directly to other available pages.

The main benefit of HATEOAS is that the requirements for a client to use the API are minimal, as it discovers the API dynamically through interaction and previous knowledge of the API and its structure (other than the main entry point) is unnecessary. The dynamic nature of HATEOAS also allows the API to evolve over time and decouples the client from the server, as the server supplies clients the currently available API paths as required by their interaction.

The REST architecture does not specify how HATEOAS should be implemented, and therefore its use varies case-by-case. Specifically, the way the server provides its clients information about API paths varies a lot, some implementations supply the links using HTTP headers and others in response body as XML or JSON structure.

A lot of criticism has been presented towards the usefulness of HATEOAS. One common argument is that the idea of client navigating dynamically through links provided by the server is too complex to implement feasibly on the client side. The usefulness of the dynamic traversal has more value to human users than programs. This is due to the fact

that humans are very capable of using and adapting to the information provided in the dynamic context. Replicating that level of intelligence in the form of a program is a very complicated task often requiring an unfeasible level of development effort.

Other criticism is that client applications are often written to use direct links to resources necessary to perform the required operation rather than implementing logic to traverse through the API each time, which has the effect of circumventing the whole idea of HATEOAS. The dynamic traversal is also criticized to generate a lot of unnecessary requests from the client to perform simple operations and therefore wasting network bandwidth and server resources.

2.3 Open API Specification

Open API Specification (OAS) is an open source project providing a framework for defining and creating RESTful APIs. It is governed by the Open API Initiative, formed in 2015 by companies such as Google, IBM, Microsoft, and PayPal, and the project is currently owned by the Linux Foundation. Open API was formerly known as Swagger specification, originally created by Wordnik in 2011 and hosted by SmartBear, which donated the Swagger to Open API Initiative as a part of its formation. SmartBear continues to develop various API development and visualization tools under the name Swagger, but that name has officially been obsoleted as an API specification [15].

OAS aims to provide API documentation in a simple format that is easy to read and understand for both human and machine readers. The format of the resulting API document is either in JSON or YAML format, both of which follow a similar and simple hierarchical structure but with a different syntax. Many open-source visualization tools exist that take an OAS document as input and present it in a more human-friendly format than the raw JSON/YAML file. One example of such a tool is ReDoc [16], which is used by Docker to publish their API to users. In its simplest usage, all that is needed is an HTML file that loads the intended OAS document and a ReDoc script to read and render it dynamically. An example of a well-documented public API using OAS is the Docker Engine API, a piece of which is shown in Figure 2 below.

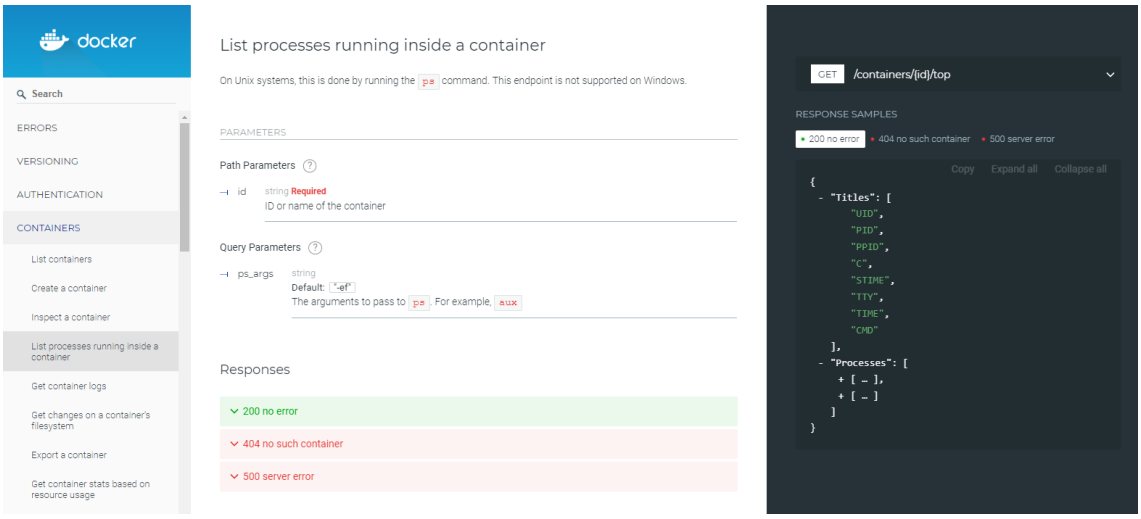

Figure 2. *A sample documentation of an endpoint in the Docker API, rendered with ReDoc. Full documentation available online in [17].*

The main advantage in using API documenting standard like OAS comes from the fact that when done in sufficient detail, the document gives everything needed to implement, test or use the API successfully. Many companies running online services, such as Zalando, are believers in “API first”- engineering strategy [18], where the first stage of system design includes only creating and locking down the APIs used, and no actual implementation logic is written.

Having the API set early on in the process allows development and testing teams to start their work independently of each other. A static and non-changing API design enables creating test suites for any functionality of the end product even before they are fully implemented, as tests can be written against the finalized API. This allows using fully written tests to give early and continuous feedback on the functionality of the system components while the overall development process is in progress.

2.4 Microservice architecture

The term “microservices” started to take hold in the early 2010s when software architectures participating in various international workshops noted similar properties and characteristics in systems they were implementing. The overall architectural style was noted to be moving away from running a single large process on the server side (so-called *Monolith* system) to smaller independently functioning processes working together to produce same the results, thus coining the term microservices [19].

One definition for the concept of microservices and overall architecture style is presented in the book “Microservice Architecture: Aligning Principles, Practices and Culture” [20]:

"A *microservice* is an independently deployable component of bounded scope that supports interoperability through message-based communication. *Microservices Architecture* is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices."

The above definition is further refined into more specific traits. Individual microservices tend to be:

- Small in size: they're kept minimal by purpose to limit the complexity and responsibility of a service. How small a service should be, depends on the application.
- Enabled by messaging: the system as a whole communicates by services messaging each other.
- Context bounded: each service should have a single responsibility, and not share that with other services.
- Independently developed: separation of concerns within the system enables services to be developed as individual products.
- Autonomously deployed: each service is executed as its own process, often in a completely isolated virtual machine.
- Decentralized: microservice systems generally do not include a service in charge of controlling other services.
- Built and released with automatic processes: independence of each service within a system enables them to be built, tested and released into the production environment regardless of other services.

Microservice architecture is not a formally specified architecture, but a collection of common attributes used in modern web applications. According to Martin Fowler and James Lewis [21], the following characteristics are typical for a microservice system and development process as a whole:

- Componentization via Services: the overall system is formed by a number of independently managed and deployed services working together.
- Organized around Business Capabilities: instead of having separate teams handling different parts of front- and backend work within the whole system, teams are built to deliver complete services with.
- Products not Projects: development team owns all work related to their service as long as it is used, instead of pre-determined criteria of completeness and delivery date. Service is released into the production environment early, and continuously improved and maintained as long as the service is in use.
- Smart endpoints and dumb pipes: communication between services is implemented as simply as possible and the underlying network is used just as means to get the message to the intended receiver.
- Decentralized Governance: the only design constraints for services are how they connect to other services. All details, such as programming language used to implement the service, is left for the team to decide.

- Decentralized Data Management: data storage is split into service-specific databases based on context is favored over large system-wide databases used by all services.
- Infrastructure Automation: code delivery from version control system into the production environment is not done by people, but by highly automated delivery infrastructure instead. Everything from building the service, to testing it and releasing into the live environment can be, and often is, automated to a high degree.
- Design for failure: services should be as fault tolerant as possible, and be prepared to handle communication issues and unavailability of other services gracefully. Status of services is monitored all the time and failed services restarted automatically, if possible.
- Evolutionary design: services should be able to be modified easily to adapt to changes in the environment. Services should be replaceable in real-time in production environment without affecting the functionality of the overall system.

The separation of system components into small individual pieces working together implement large-scale systems is not a new or groundbreaking idea. In fact, the design principles known as the Unix philosophy, written by Doug McIlroy in Bell System Technical Journal from 1978 [22], apply quite easily to microservice mindset, by using the word “service” instead of “program” or “software”:

- Make each program do one thing, and do it well. To do a new job, build afresh rather than complicate old programs by adding new “features”.
- Expect the output of every program to become the input to another, as yet unknown, program.
- Design and build software, even operating systems, to be tried early, ideally within weeks. Don’t hesitate to throw away the clumsy parts and rebuild them.

2.4.1 Virtualization via containers

The fundamental nature of microservices, combined with advancements in cloud infrastructure technology, lends itself quite naturally to the deployment of microservice systems by using various virtualization methods. Some popular virtualization technologies today are Docker, which provides a way to release software as single isolated containers that are light to execute, and Kubernetes, which offers tools to large-scale deployment for containerized applications.

Docker is a tool for executing and managing virtualized lightweight containers [23]. It does virtualization at the operating system level, in which all containers share the same kernel but are executed as separate user spaces in memory. This creates a level of isolation to container execution, and containers can implement their own filesystems and other key infrastructure within the container. Kernel sharing saves computational resources, all containers share the same hardware and there is no need to emulate hardware virtually.

Docker containers are generated from a list of instructions called a Dockerfile, which specifies a Docker image. Dockerfile contains information about what operating system kernel it uses and what commands are run on the image before the start of the execution, for example [24].

The independent nature of individual microservices allows scaling the service horizontally by increasing the number of service instances under heavy workload, rather than duplicating the entire system as would be needed with a monolithic application in order to respond to overall system load changes dynamically.

2.4.2 DevOps – Development Operations

The emergence of DevOps culture has been claimed to enable the overall development of microservice systems. The term DevOps comes from the fact that it merges the application development team (Dev) with the operations team (Ops) responsible for managing the live application. In the DevOps way of thinking, a single team is responsible for the entire lifetime their deliverable, throughout development, testing, deployment to production, and maintenance. According to Amazon [25], the overall goal behind DevOps is to automate and streamline software development and infrastructure management processes. DevOps is said to be more of a cultural philosophy and practices that aim to shorten software delivery times and make evolving the software into new versions easier and quicker.

Amazon lists the following as the best practices to DevOps (using their own infrastructure as an example):

- Continuous Integration: As developers push their code into the used version control system, builds are generated and existing automated tests are executed automatically on the build server
- Continuous Delivery: Automated builds that pass through all levels of testing successfully are automatically deployed to a live production environment. The focus is on keeping the product deployable at any given time into any environment required [26].
- Microservices: Application is deployed as small independently managed services.
- Infrastructure as code: All necessary virtual infrastructure to run the application is dynamically provisioned using automated tools provided by the used cloud platform.
- Monitoring and logging: All services provide real-time metrics and logs about their levels of activity, giving the DevOps team means to understand how updates and configuration changes impact the application performance.

- Communication and Collaboration: The merging of development and operations provides shortened paths of communication and a better understanding of the workflows and responsibilities of the system as a whole.

Since DevOps culture demands that the product is deployable at all times, there is often a need to have private and isolated “sandbox” environments. These environments resemble closely fully deployed production systems and are available for developers to try out their new code in isolation from the environment end users are using before integrating their work into the larger code base.

For the same reasons, similarly isolated full-sized test environments are needed to integrate works of multiple developers together. Running acceptance tests for larger sets of code changes before releasing anything into the production environment is also done in isolated environments. In Continuous Delivery practices, these needs are handled by having multiple tiers of environments available at all times for different purposes [27].

- Production environment: the live deployment that the final customers are using.
- Staging environment: the level where final acceptance tests are done before deploying anything into production. The staging environment should replicate production conditions as closely as possible.
- Integration environment: used to merge a collection of changes together into different application release versions.
- Development environment: the lowest level of work takes place in this environment, developers may try anything they want without negatively affecting other developers work. May also be the integration environment at the same time.

When done properly, this kind of staggered releasing process minimizes the amount of time it takes to notice and correct possible defects in the system. As testing takes place in multiple iterations before release into production, the chance of issues slipping through the cracks should go down.

3. TEST AUTOMATION

Generally speaking, any test execution driven programmatically by a computer following a pre-determined list of actions can be considered test automation. It can range from writing a list of Linux commands into a file and giving it as an input to a command line interpreter (like bash) to a large GUI-based application that interact with another separate software. One definition is “the use of a separate software from the testable application to control and execute test cases against defined specifications” [28].

3.1 General properties of automated testing

Test automation offers many attractive properties when compared to a fully manual, human-performed testing:

- Automated tests are executed faster than manual tests and require no human supervision. Performance is limited by the response time of the system under test (SUT), and how quickly the test program can react to the behavior of SUT in order to continue testing.
- When developed properly, automated tests have a higher level and a more stable quality of results. Human testers get eventually tired and may lose their concentration when performing simple and repetitive test steps over and over again, leading to errors and sloppier results overall. Fully automatic tests are executed the same way every single time and therefore their results are expected to vary minimally.
- Automated testing allows using time resources more effectively. Long lasting tests can be left running overnight and other times when people are not at work to produce new results that are available later when people have time to analyze them.
- Automated tests yield information about how the system performs in long-lasting scenarios easier than in fully manual testing and may expose flaws that would otherwise be difficult to notice or induce. Some examples would be slowly occurring memory leaks, and degraded system performance when the amount of stored data reaches high levels.
- Automated tests, especially when targeted at lower levels of application (unit tests), give developers quick feedback about the functionality of the code they are working on. Test automation can be integrated into version control systems to execute a set of tests whenever code changes are pushed into the repository. This way the code change can be immediately tested to see if it broke functionality in the program. In case it did break something, the code can be compared to a previous correctly working version and fixed quickly. Testing with this intention is called *regression testing*, and automated tests are extremely well suited for that task.

- Automated testing scales in parallel in a very cost-efficient way. The only limit is the number of test machines available, and hardware is in most cases significantly cheaper than human resources. This scaling is very useful when testing how SUT performs under heavy load.
- Automated testing can adapt to different configurations quickly by just simply giving the test program a different set of parameters to work with when utilizing some form of parametrization.
- Test automation frameworks output in many cases nicely formatted, high quality and comprehensive results about the test execution.

When using test automation extensively, some challenges and limitations need to be accounted for beforehand:

- Automated tests are limited in how they react to error situations. It is impossible to predict beforehand all the possible faulty conditions that may arise during test execution, and have the logic handling these situations implemented in test suites.
- Automated tests are not a “create once and expect them to work forever” type of solution, and require constant maintenance to keep in a usable state, just like any other software. Automatic GUI tests are notorious in this regard.
- Automated tests require reliable infrastructure to be used to their full potential. For example, unexpected power outages or system crashes during nighttime make tests run during that time often incomplete or have limited value.
- When automated tests fail because of hardware issues, finding the root cause can be time-consuming and difficult to diagnose.
- Automated testing requires a stable set of requirements in order to minimize the need for maintenance work to keep test automation functional and able to fulfill its purpose.

3.2 Levels of test automation

Automated tests can be applied to the software at various levels, depending on what kind of testing is desired. One way of illustrating different levels of test automation is the Test Automation Pyramid, the introduction of which is often credited to Mike Cohn in literature [29]. The pyramid sets three levels of testing, in the order from lowest to highest level: unit testing, service testing, and UI testing. The naming of levels varies depending on the presentation. One version presented in Lisa Crispin’s book “*Agile testing: a practical guide for testers and agile teams”* is shown in Figure 3 below.

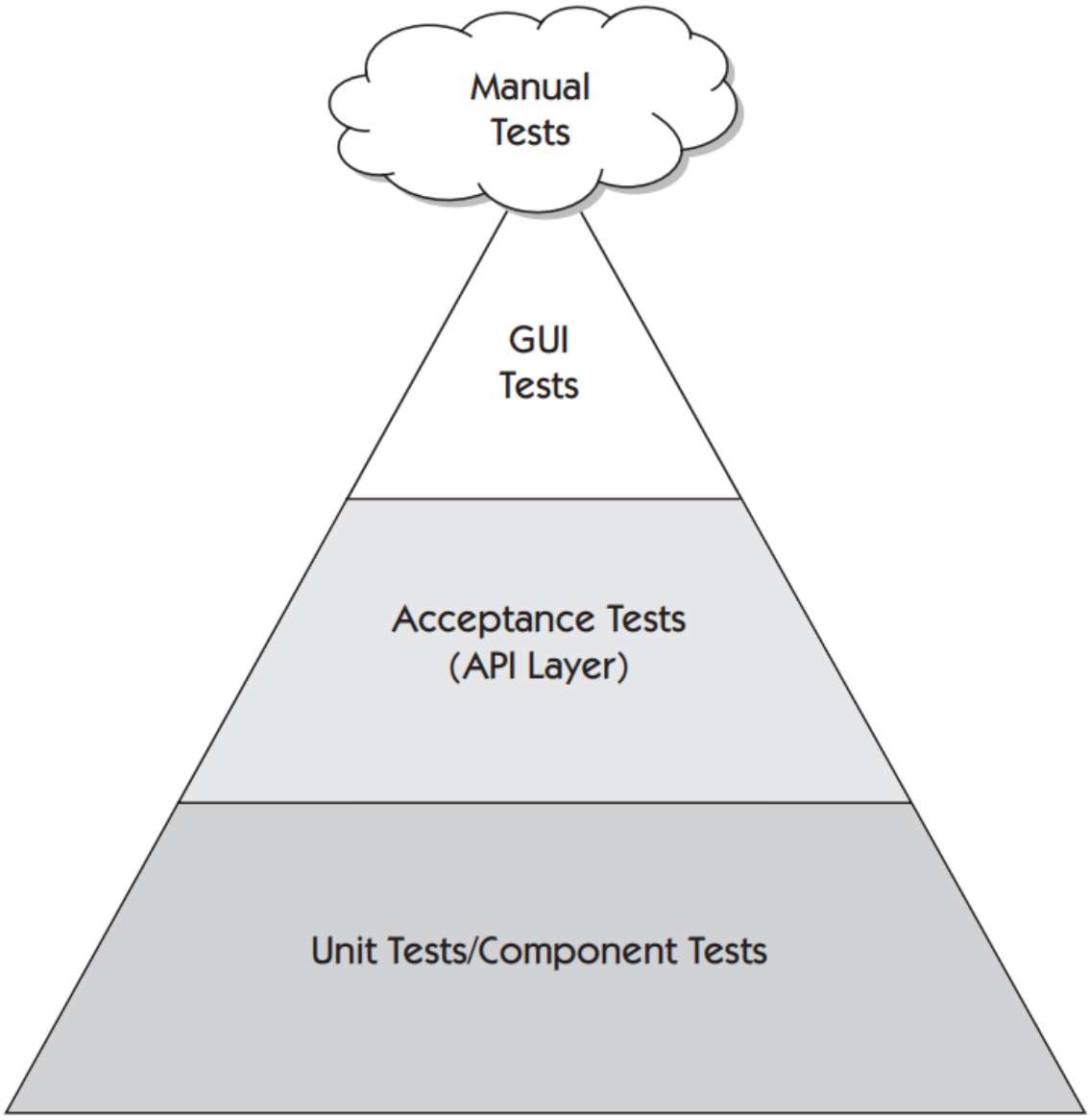

Figure 3. *Test automation pyramid, as presented by Lisa Crispin [30]*

The Test automation pyramid is an abstraction on how much testing effort should be spent at each level of the application. The scope of testing at each level increases from unit tests to UI testing, as unit tests focus on the smallest size of components possible and UI testing covers the entirety of the system. The complexity of automated tests increase on upper levels of the pyramid, leading to increased effort needed to generate a wide coverage of the SUT as a whole.

3.2.1 Unit testing

Unit testing takes place at the lowest level possible, and “is a process of testing the individual subprograms, subroutines, classes or procedures in a program” [31]. As it focuses and concerns itself on the smallest pieces of the application, unit testing should

be the first indicator of issues in the code. Unit testing should cover the entire application to provide as wide visibility as possible on how code changes affect the system as a whole on the lowest level. The test automation pyramid reflects this as well by having unit tests as the largest piece of the pyramid.

Since unit testing takes place at the code level, developer-level knowledge of the code is required to do it effectively. Well written unit tests have the side effect of also documenting the code better: unit tests can give another developer an idea on how to use the targeted component by just looking at associated unit tests to see intended use and behavior.

Unit testing is in nearly all cases handled by using a test framework best suited for the specific use case. Because of the fact that unit testing takes place at the code level, usually the used framework is using the same language as the code being tested. Frameworks exist for pretty much every conceivable programming language, Wikipedia, for example, lists frameworks for 80 different languages and the most popular languages have tens of frameworks to choose from [32].

In unit testing, components are executed in isolation from other components. The component is given some set of parameters to work with, and the test results are interpreted from what the return value the component responded with. For example, a component that multiplies a list of numbers into a single value could be given a parameter list of 2, 2 and 5, and the unit test passes when the returned result is 20, otherwise the test fails. If the tested component requires other components to serve its function, these external dependencies are handled by using purpose-specific *mock* or *stub* objects that mimic the behavior of the original object but in a limited fashion. Designing and implementing these mock objects are a crucial part of writing unit tests and take a significant piece of the overall unit test development time.

Although unit testing is a crucial part of the testing process as a whole, the information it produces has a very narrow scope. The use of mock objects during testing does not give a real picture of how the interaction between the actual components work, and therefore more complex testing is needed.

In the context of microservice systems, unit testing can also be considered to take place at a higher level of abstraction than at the lowest level of code. Single microservices are independent and isolated entities, and therefore have the same kind of qualities as single functions and classes. In this sense, testing an entire microservice in isolation is very similar to lower level unit testing and similar methods of mocking external dependencies can be utilized.

3.2.2 Integration testing

A formal definition of integration testing by ISTQB (International Software Testing Qualifications Board) says it is “testing performed to expose defects in the interfaces and in the interactions between integrated components or systems” [33]. In a broad sense, integration testing can involve an arbitrary number of components.

Just like the test automation pyramid illustrates, integration testing is done on a higher level of abstraction than unit testing, and it consists of taking a number of components and seeing how they work together. Mocking is not used generally at this level, all tested components are their real-life manifestations. Components are tested in as much isolation as possible, just like in unit testing.

The idea to isolate the components under test leads to problems when trying to cover the system as widely as possible, as the number of component permutations to test increases rapidly the more components there are in the system. The test automation pyramid takes this difficulty into account, showing it as a smaller part of the whole automation flow than unit testing.

Integration testing is done by using the targeted components directly through the API each component provides. Test cases are designed to involve multiple components in a well-known execution path to produce the desired results. Some form of logging is often a prerequisite for complex integration tests in order to analyze afterwards how the components in the execution path reacted to the given input and if there are any unwanted side-effects that arise from the given test scenario.

When the tested system is network-based, integration testing is often done in integration and staging environments. The deployment of these environments in order to run tests can take some effort and should be taken into account when utilizing automated integration tests.

3.2.3 System testing

The top level in the test automation pyramid is system testing. At this level, all testing activities are done by using exactly the same user interface as the end-users do. This allows testing the SUT in the way it is designed to be used and as such provides valuable information about system performance and functionality as a whole.

Like in other levels of testing, system complexity becomes an issue even in simple applications to have extensive coverage of the SUT. For example, barebones text editing

application Microsoft WordPad has been estimated at one point in time to have 325 possible GUI operations available for a user [34]. Creating a test set just to cover all these operations individually requires considerable effort, and covering all possible permutations becomes unfeasible rather quickly. GUI testing often focuses on ensuring that some set of primary use cases of the application work without problems.

The general way user interfaces are interacted with is via user events that operating systems generate based on how a human uses physical input devices, like mouse and keyboard. Tools for generating user events are available to allow automatic UI testing in different domains. Approaches to user event generation range from simple “left click mouse at coordinates (x,y)” – commands, to scanning a retrieved HTML document in order to find the desired element to click, and to graphical pattern matching to find the target element.

The biggest challenge in automated UI testing comes from the large maintenance requirements. Automated UI tests often contain enough logic to complete the required task in an optimistic fashion, and are limited in the way they can deal with unexpected conditions that human users can adapt to easily.

For example, if an automatic test depends on clicking an element at specific coordinates, any movement of that element will break the test completely. The movement may result from many things, such as changes in the UI layout, another UI element being larger than expected (if dynamic sizing is being used), or a different test environment where elements are rendered in a different size. If graphical pattern matching is used to locate the target element, changes in the element color, shape, size or possible text content, may also lead to broken test cases. For a human user, it is easy to adapt to these kinds of changes on the fly and still be able to complete the intended task.

Another problem is the varying response times of the SUT. For example, if a test case involves clicking a button which then is expected to lead to another UI element appearing before continuing, some timeout is usually used to prevent the test being caught in an endless loop should the element never appear. This leads to false results if the SUT does respond as it should, but after the timeout has been exceeded. Once again, this is a scenario that human tester can adapt to rather easily.

There are ways to generate UI tests in a machine-assisted way to ease the burden on the test case implementer. One such way is the record-and-replay method, where the test framework allows the replaying of a previously recorded sequence of actions made by a human user. One notable tool with this functionality is Selenium, which is a tool for

testing web-browser based applications [48]. The Selenium way of recording user actions is done by observing user actions directly from the elements of the web page instead of reading the location of mouse clicks for example.

Recorded automation scripts are brittle, just like other automation methods. A change in a critical part of the SUT that the recorded test relies on likely breaks the test case. In such case the recording has to be done again in order to continue using it for automation purposes.

4. DESIGNING A TEST FRAMEWORK

This chapter brings the theoretical topics discussed previously into action, by using the knowledge in designing a test automation framework. The target platform that the test framework is created for is introduced first. After that, the design process of the framework is discussed, from initial requirements for the framework to the tools that were and were not selected for the framework.

4.1 Environment

The main target environment for this test automation framework is the Intel Insight platform, launched by Intel Drone Group in 2018. At its core, Insight is a cloud-based data storage and management tool for aerial images targeted at enterprise customers working in construction or utilities industries, for example [35].

The platform is implemented as a microservice application hosted in Amazon Web Services cloud platform, with REST API based information and data flow between services within the system. Detailed descriptions of the application architecture and implementation are for internal use only and as such cannot be presented in the scope of this thesis.

The main feature of the platform besides data storage is automatic photogrammetry analysis and model creation. Photogrammetry is a process that takes a set of photographs as an input, and the output is typically a map, a drawing, a measurement or a 3D model of some real-world object or scene [36]. The results are viewable directly from a web browser and as such do not require any separate model visualization software.

Photogrammetry models can be annotated with various preset options, and accurate measurements of length, area and volume can be made based on the model. Image datasets in industrial use cases are generally quite large in size, and typically contain thousands of images taking storage capacity in the magnitude of tens of gigabytes. Figure 4 below shows an example dataset of 743 images taken from an old water tower located in Hiedaranta.

Figure 4. *UI view from a 3D model of an old water tower from Hiedanranta industrial area in Tampere. A measurement of the height of the tower is visible in the model, and presented numerically on the right side panel.*

The original version of the Insight platform is a standalone product that has no external integrations. The only supported workflow is a manual one where a user selects and uploads the desired set of images to the cloud using a web browser interface.

The main purpose of the framework is for testing Intel Insight. It was already a familiar platform as it had been tested previously with manual UI testing. A small set of automated UI tests had been previously developed for the platform, but maintaining them was deemed too much effort to continue. This was largely due to the fact that the platform was on early development stages and was continuously changing. These were brought into use again and updated and extended as a part of this framework. As a whole, the tools used in the framework are useful for automating tests for other kinds of systems than exclusively web browser based cloud services.

4.2 Tools & selection process

The first step in designing a system is selecting used technologies and tools suitable for the task. At this stage, some high-level requirements and desired properties were thought of to guide the research, learning and selection process. The most important criteria are listed below, with the thought process behind them explained. All of the criteria do not apply to every purpose, as tests done through the UI have a fundamentally different set of requirements than integration tests:

- The selected tools should preferably be free to use and if possible, open-source projects. This creates a lot of flexibility on starting to use the tool, as there are no budgetary or licensing issues to begin using the tool immediately as extensively as needed. Another benefit is that without any capital investment all tools are painlessly replaceable if they prove to be unsuitable for their role. Having the

source code available also allows possibly modifying and extending the tool for internal use cases, if such a need arises in the future.

- The ability to easily configure the tool for usage in different environments was a highly desired attribute. This should be possible with the use of simple configuration files, to have flexibility when using the tool in all possible use cases and purposes.
- Integrating the tool into various continuous integration systems, like Jenkins or Gitlab, should be possible with minimal configuration work. CI integration enables the automatic execution of integration tests and allows the launching of UI tests using a control panel-type controls.
- Selected tools should not require complex setup, and be light to adopt from scratch and to use overall. Tools should be beginner-friendly in order to make it easy to adopt them into use, preferably via some kind of IDE to simplify early stages of learning.
- To avoid hard-coded resource paths as much as possible, used tools should allow creating all test content dynamically during execution of test scenarios. This enables automatic testing of newly deployed environments without creating any pre-existing test content manually beforehand.
- Tools should preferably have the functionality needed to run as a *headless browser*. A headless browser is an application that has all the functionality of a standard web browser but has no GUI, making it easy to perform full functional testing without needing a graphical environment to execute the tests.
- To assist in testing systems with continuously evolving APIs, selected tools should be able to read an OpenAPI document and generate some kind of structure based on that to make it easier to deal with possible API changes.

The overall strategy for testing Intel Insight was chosen to focus on testing things as closely to the end-user level as possible. The main motivation for the choice was to use the available time and people resources as efficiently as possible while producing test results with a wide coverage of the system. This led to the automation effort to focus first on tests through the UI, and then on lower levels.

Ultimately the set of tools that were selected are Postman for integration and API testing, and a combination of Selenium and SikuliX for end-to-end testing through the UI. The biggest factor in selecting these tools was familiarity from previous projects and low entry barrier associated with them. Unit testing was declared to be out of scope and therefore those options were not actively looked upon.

During the tool selection, process a number of other individual tools for different purposes were evaluated and analyzed. These vary in size from small command line tools to larger GUI applications, with all of them having some good qualities to them, but ultimately coming up short on desired functionality. The most relevant ones that got filtered out are listed below, with the reasoning why they ultimately were not selected:

- Connexion [37] is a library/framework for API mocking, an open source project developed by Zalando. The tool is given an OpenAPI document as a parameter,

which is then used to start a local test server mocking the functionality. This gives a quick way to try out modifications to an API, making test server development lighter task. While sounding promising, using Connexion effectively requires development effort to support its use. The user has to implement handlers for all endpoints manually as Python functions. The input API document has to be tagged manually with annotations on each endpoint pointing to the appropriate handler in order to start the test server automatically. Connexion also proved to handle valid OpenAPI 3.0 documentation poorly.

- Meqa [38] is an open source project for automatic REST API test generation and execution, created by developer Ying Xie. It is designed to read an OpenAPI document, generate a test suite based on that and it also offers a simple command-line tool to execute the test suite automatically against a mock server. While sounding promising, the project is not actively developed anymore with the latest changes to the repository being from October 2017. The creator of the tool also states that it is just a proof-of-concept project and it does not have full OpenAPI 3.0 support. In actual use, Meqa was found to work successfully only with simple example APIs.
- SoapUI [39] is a GUI-based API test framework developed by SmartBear, the company behind Swagger API development tools and the OpenAPI specification until version 3.0. It has a large set of options for API testing, including automatic mock server template generation based on an OpenAPI document, record-and-replay capabilities, load testing tools and more. SoapUI has a free and open sourced Community edition available, but with a considerably limited set of functionality. The community edition was declared to not have enough functionality for large usage, especially because of a lack of continuous integration support. The application as a whole also felt clumsy to use and the UI felt confusing and unintuitive overall.
- Katalon studio [40] is another GUI based API test framework built on top of Selenium. Its functionality is close to what Postman has, but as it was discovered later Katalon never got a lot of consideration although it is a viable option for API testing.
- Oatts [41] is a test scaffolding generation tool for NodeJS projects. It generates generic test templates for all endpoints it finds on an OpenAPI document, but they need to be manually filled to get results. Oatts also provides a way to execute the test templates it generated directly from the command line. With these constraints, Oatts was determined to be too limited and require too much development upkeep in order to be used for this purpose.
- Insomnia [42] is yet another GUI API test framework. It has many similarities to Postman but lacks a way to run tests outside of the desktop application. This and the fact Insomnia was discovered after Postman led to it not being chosen to be the tool for API testing.

4.2.1 Postman

Postman [43] is a versatile GUI-based API test development and automation suite. The project was first published as a Chrome extension in 2012 and it was later released as a standalone desktop application on multiple operating systems. Postman has a tiered

subscription system, with free tier having all basic functionality available and paid subscription tiers offer increasingly more cloud-based features for cross-site collaborative workflows.

The main feature of Postman is beginner-friendly GUI based workflow for creating, sending and receiving responses to, HTTP requests. Fundamentally speaking, Postman implements the functionality of a headless browser. All requests and their responses, along with the data contained in them, are stored for later analysis, and appropriate headers for all requests are generated automatically or by user input.

Single requests can be saved and bundled together into hierarchical structures that are called *collections* in Postman terminology. An example of a collection view is shown in figure 5 below. Collections are the Postman way of managing larger operations performed with the API, by saving all the needed HTTP requests in the appropriate order in order to be used later. Dynamic data handling is implemented by using environment variables that are used either globally or within separately encapsulated environments that can be applied to collections [44].

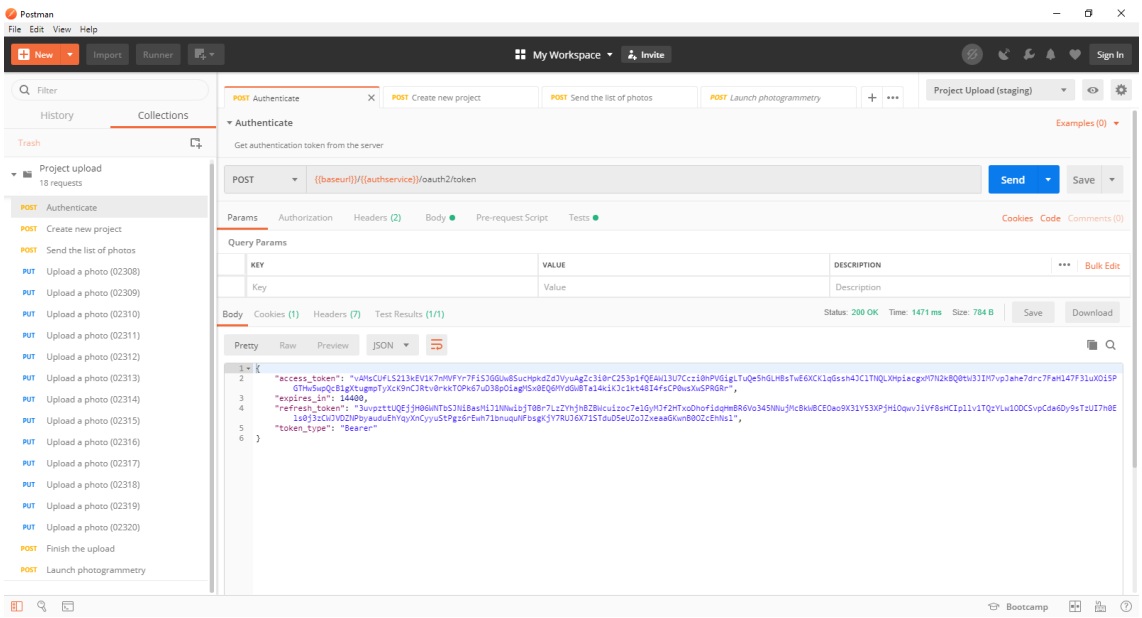

Figure 5. *Postman main window, with an example collection opened on the left side of the screen*

Collections can be generated automatically from an OpenAPI document, filled with sample requests for each available operation grouped into a folder structure named after the associated endpoint of the API. All metadata written into the API document is also displayed in the UI, making it simultaneously a documentation rendering tool.

Postman offers a mechanism to execute scripts written in JavaScript before sending a request and after receiving the corresponding response. The script is run in a sandboxed

environment with access to applicable environment variables and in case of post-response script, full response data. In Postman workflow automatic test execution takes place in post-response scripts, where the received response can be analyzed to determine whether it was sent with expected data.

Scripts can be optionally given for an entire collection and a single folder within a collection. These scripts are applied to all requests in the collection or a folder and executed automatically alongside all other scripts. This allows performing some common operations automatically for each request in the collection. The order of script execution starts from Collection level pre-request script, as is shown in Figure 6 below.

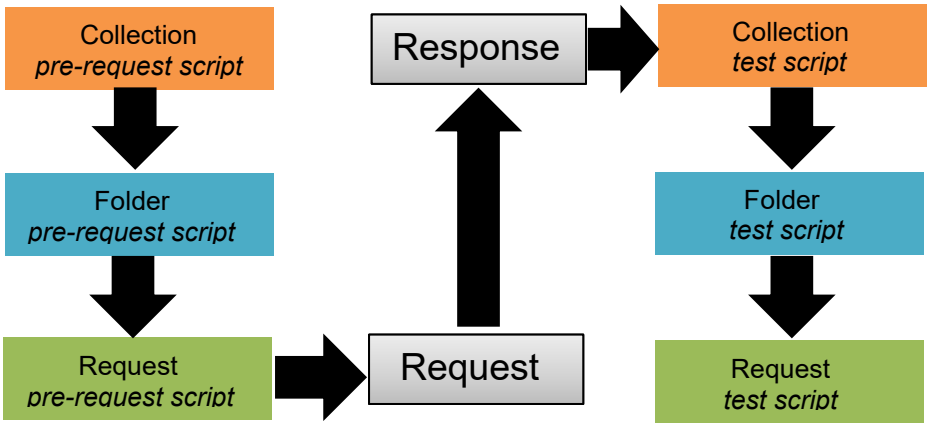

Figure 6. *Postman script execution order. Adapted from [45]*

The script editor in Postman has some generally used test assertion snippets that can be pasted into the script with a single click to avoid having to type them manually every time. These scripts are the main method for dynamic environment control, allowing information received in responses to be stored into variables for the use of later requests. This is useful for example when working with a token-based authorization scheme requiring a valid token being included in all requests. In this scenario, a new authorization token can be retrieved in a separate request and stored into an environment variable to enable successful sending of further requests.

Collections are the logical structure that automatic request sending can be applied to. Request automation is done via a special collection runner tool in the Postman application. The runner executes the requests of the collection sequentially from first to last in the order they are shown in the UI. Collection runner automatically writes a test result log from the results of all assertions executed in each post-response script.

A separate tool named Newman is available for executing Postman collections outside the standalone Postman application. Newman is a NodeJS package that is installed with

NPM and it integrates into the target operating system as a command line tool [46]. To use Newman, a collection and optionally the corresponding environment have to be exported from Postman application as JSON files separately and given to Newman as input parameters. The use of Newman allows integrating Postman into various continuous integration systems seamlessly.

The standalone Postman application comes with a tool for API monitoring. This is done with scheduled execution of collections. The results are visualized as a graph where response times are shown in a line graph and response payload size as a bar chart. Example of this kind of graph is in Figure 7 below.

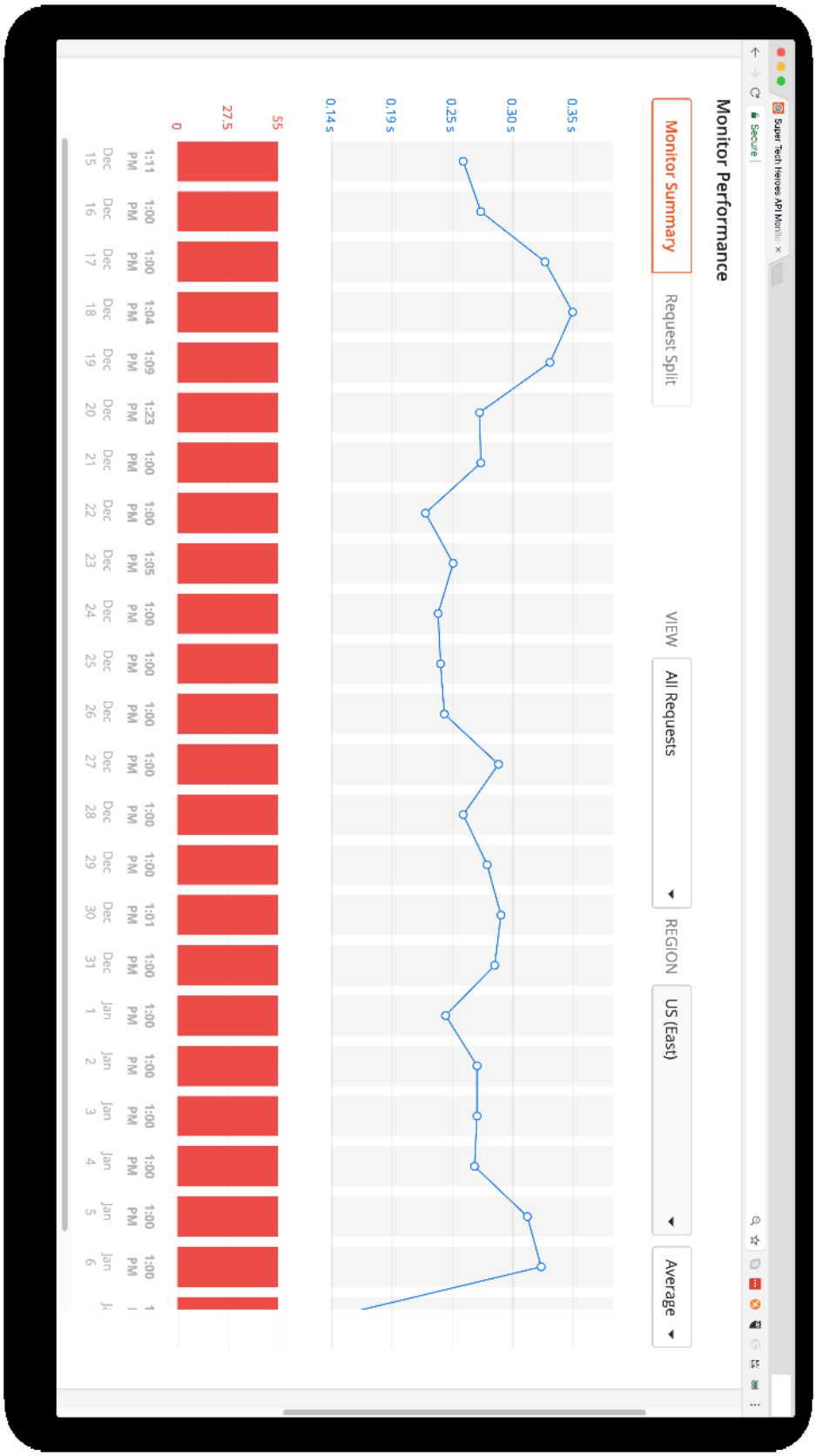

Figure 7. *Postman monitoring tool showing response times and payload size [47]*

The paid subscription tiers of Postman gives access to their cloud-based services for sharing collections online to other teams using Postman, server mocking and automatic

API monitoring on the cloud. These services were not considered important enough in this usage to be purchased, the free tier proved sufficient for this use.

4.2.2 Selenium

Selenium is an open source collection of tools and libraries for testing web browsers [48]. Its original version was developed by Jason Huggins, while he was working at Thoughtworks, as an internal test tool in 2004 for running tests in JavaScript. Selenium currently has three main tools under its umbrella, Selenium IDE is a browser extension that allows recording of user actions and replaying them later all directly from the browser, WebDriver is a library for running and interacting with a browser directly with one of the supported programming languages, and Selenium Grid is a management tool for running parallel Selenium scripts on physically separate test devices.

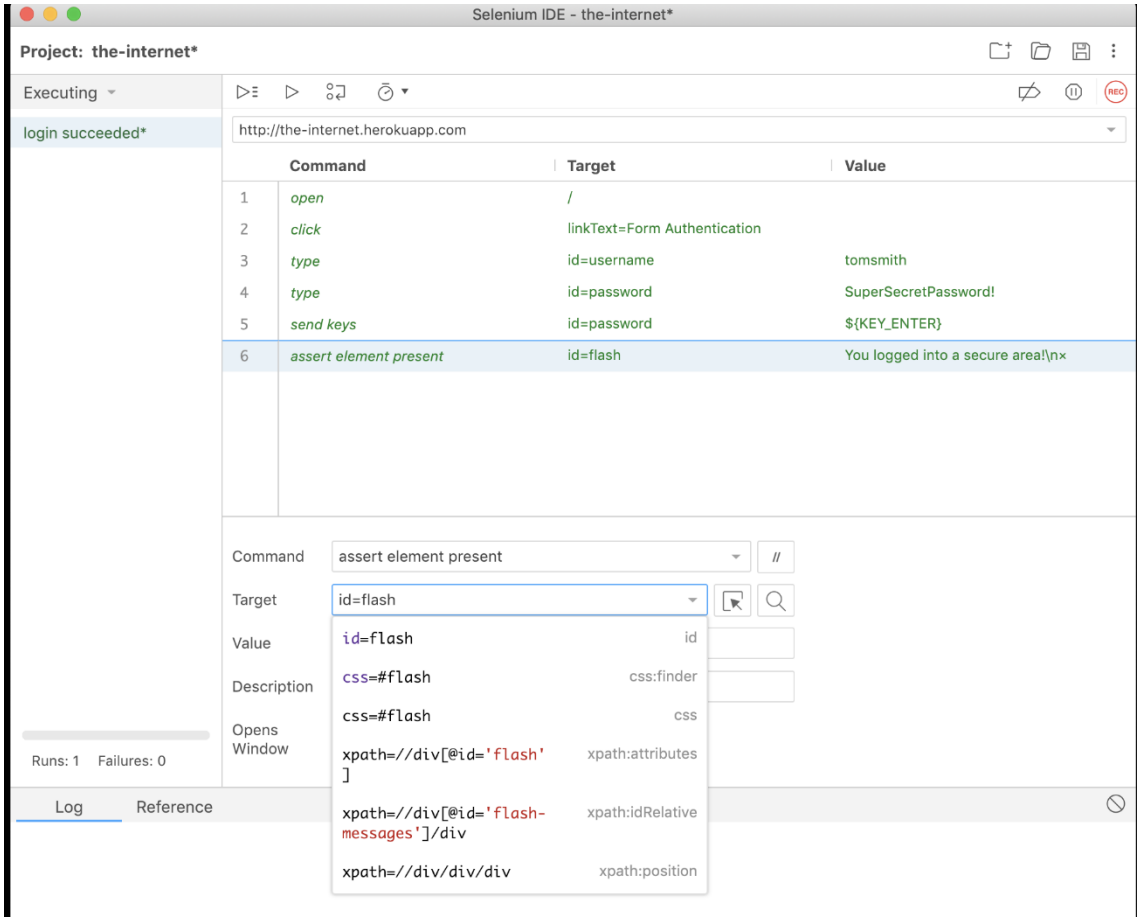

Figure 8. *Selenium IDE, main window [49].*

Selenium test tools work by reading directly the HTML document rendered by the browser and targeting user actions based on that. This makes test execution entirely based on the web page structure, and no pattern recognition or coordinate-based decision making is required, making Selenium tests independent from the look of the UI and

allows functioning as a headless browser. Tests are written either in Selenese, a Selenium scripting language, or by using the WebDriver library in Java, JavaScript, C#, Python, Ruby, PHP or Perl.

4.2.3 SikuliX

SikuliX is an open source GUI automation tool. Its predecessor Sikuli was originally created in 2009 as a research project of the Usable programming group in Massachusetts Institute of Technology [50]. After the development of Sikuli ceased in 2012 under the original development group, its development was taken over and rebranded as SikuliX by Raymond “RaiMan” Hocke [51]. SikuliX is written in Java and supports scripting with Python, Ruby, and JavaScript through their Java implementations, such as Jython, JRuby, and Java scripting engine.

Automation scripts with SikuliX are based on graphical pattern matching. It scans the contents currently shown on display and tries to find patterns similar to previously created reference images. SikuliX provides emulation of input devices, such as mouse and keyboard, and can direct user events to parts of the screen based on findings of the pattern matching. Pattern matching is done by utilizing OpenCV image processing library and optical character recognition is also supported with the use of Tess4J and Tesseract libraries.

SikuliX provides an IDE with limited capabilities for script development. One crucial and helpful tool integrated into the IDE is a screenshot capturing tool that works directly from the IDE window. This streamlines script creation, and thumbnails of the captured screenshots are displayed directly in the script itself to help users keep track of the execution with visual cues. IDE allows a wizard-based way of manipulating mouse events by allowing users to select manually the desired offset by selecting the exact location from a reference image. An empty main view of the IDE is shown in Figure 9 below.

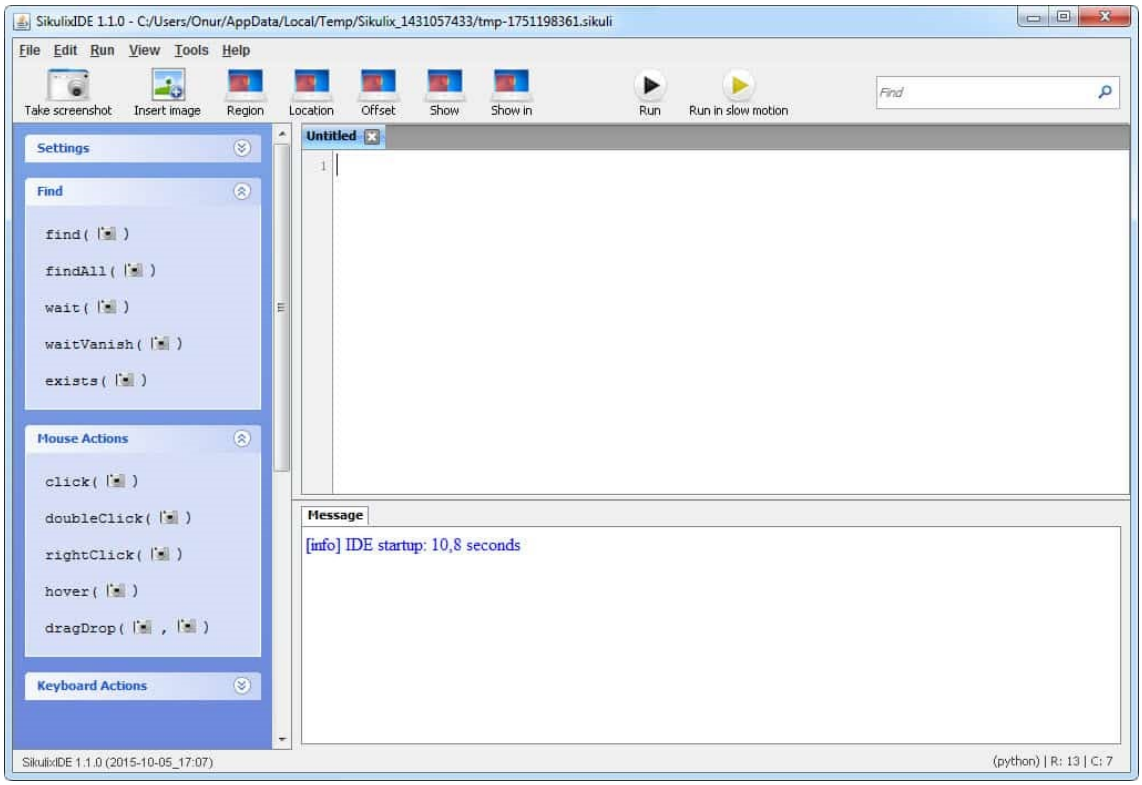

Figure 9. *SikuliX IDE main view. Many often used actions are displayed on the panels on the left side of the screen for easy usage [52].*

The main advantage of SikuliX is that it can be utilized in automating pretty much anything. The only requirements are that a display is connected to the PC executing the program and something is graphically rendered on that display for SikuliX to find and react accordingly to. The size of the application does not matter, anything from full-sized operating systems to small helper tools can interact with SikuliX in an automated fashion. Lack of a graphical user interface is not a limitation either, working with pure command line programs is also possible by utilizing optical character recognition. With very few limitations in where it can be applied to, SikuliX is an extremely versatile automation tool.

5. IMPLEMENTATION

To illustrate how the selected tools can be utilized in automating tests for standard operations of the Insight platform, let us examine the uploading of images as an example use case. In order to utilize services provided by Insight platform, image sets have to be present on the server. From the end user point of view, this involves creating a new project from the browser UI, selecting the set of images to be uploaded, choosing an engine for automatic photogrammetry processing, and pressing a button to confirm and trigger project creation and data upload to the server.

All the fine details of this process that do not require user interaction are being handled on the browser side code of the application. These details include reading the metadata of each image file, constructing larger project level metadata based on those, and the final uploading of the dataset to the server in a parallelized manner.

Doing the same operation directly through the API involves a couple of extra steps. Below is a high-level summary of the process, which can also be observer using developer tools- tab found in all widely used web browsers:

1. Send a POST request to the authentication service with valid login credentials included in the request body. If login credentials are valid, the service sends in its response body an authentication token to be included in all requests of the current user session. The token automatically expires after a couple of hours.
2. Send a POST request to the UI service with request body containing general metadata about the project, photogrammetry processing settings and the number of pictures to be uploaded to the server. The server sends a response back with a large collection of metadata related to the project, including project, mission and flight identifiers that will provide the necessary context for the subsequent requests related to the newly created project.
3. Send a POST request to the project management service with a request body containing a list of the files that are going to be uploaded and their metadata, along with the project, mission and flight identifiers acquired in the previous step. The server response lists the soon to be uploaded files and their metadata, and adds among it the identifier to be used for each file when doing the actual upload to the server.
4. For each file included in the upload, send a PUT request to the data storage service’s endpoint by appending the image identifier received during the previous step into the upload URL. The request body is the file that is uploaded and its MD5 checksum needs to be set manually into the corresponding HTTP header, along with “no-cache” value for cache control. The server response is simply an indication of whether the request was completed successfully and if not, the reason it did not go through (wrong MD5 checksum for example).
5. Once all images have been uploaded, send a POST request to the project management service’s endpoint related to the uploaded flight in order to complete the

upload process, using the corresponding flight identifier. The server response indicates the success of the request.

6. Send a POST request to the project management service's endpoint related to the mission to start the photogrammetry processing of the uploaded dataset. The server response indicates the success of the request.

Since the process is very mechanic and straightforward by nature, it can be recreated and automated as a Postman collection. Figure 10 below shows the way Postman presents the sequence of requests in the collection view of the main window.

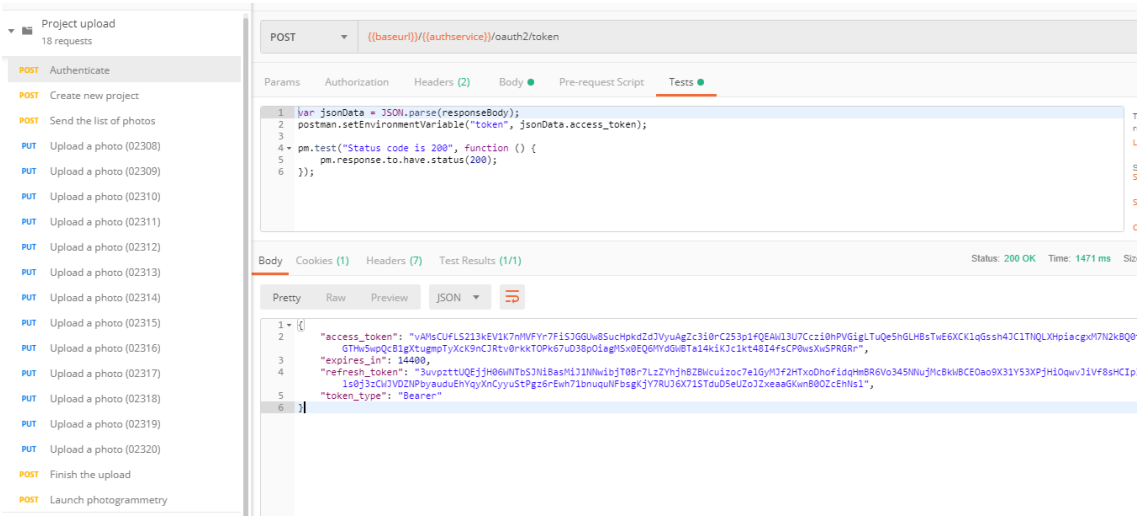

Figure 10. *Project upload collection example, with a small dataset of 13 pictures*

To begin the upload process, a valid authentication token has to be retrieved first. Once authentication succeeds, the token can be found in the response body, like in Figure 10 above. To use it later, the token is stored into an environment variable in the test script. Postman has an option to set a general authentication scheme to an entire collection, thus reducing the need to set it up manually in each request. Intel Insight authenticates by using Bearer scheme, where the authentication token is included as "authentication" request header, with the value of "Bearer <tokenstring>". Example of this is shown in Figure 11 below, where Postman's environment variable syntax of {{*variablename*}} is also visible.

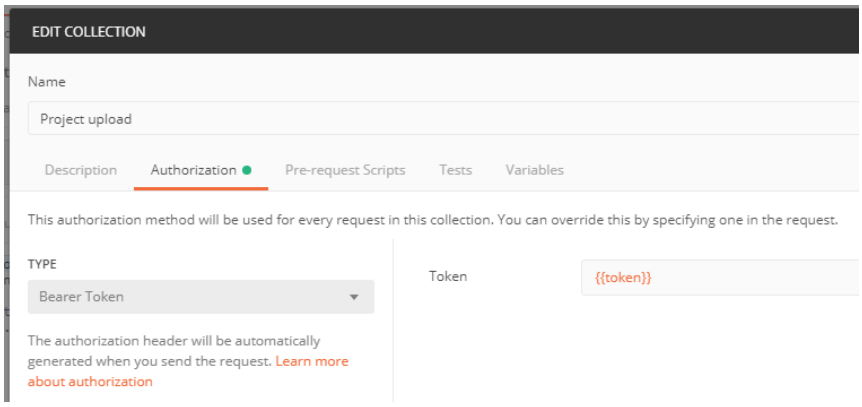

Figure 11. *Collection authorization scheme*

After authentication, the upload process can begin by creating a new project. A POST request is sent to the appropriate endpoint and the identifiers needed in later steps are stored in environment variables.

The next step is to send a POST request including the list of images and their metadata to the server. The response from the server contains the file identifiers to be used in the upcoming upload. In order to use them in the individual uploads, the file identifiers have to be extracted from the response and stored into environment variables, like in Program 1 below. Due to limitations of the Postman environment variable system, where the only variable type is a string, the list of image identifiers needs to be stored as a JSON structure written as a string. When the identifier is needed, it can be retrieved from the environment and set as another environment variable before using it in the upload request.

```
    var jsonData = pm.response.json();
2   var imagepaths = [];

4   for (let item of jsonData.photos){
        var temp = {file: item.seq,
6                   path: item._id};
        imagepaths.push(temp);
8   }
    postman.setEnvironmentVariable("imagepaths",   JSON.stringify(im-
10 agepaths));
```

Program 1. *Unpacking and storing image identifiers by using a Postman test script written in JavaScript*

The way presented in Program 1 above is not the only solution to storing image identifiers, it would have been perfectly working and valid way to be done by using a separate variable for every file to be uploaded. This solution was found to be crude and resulting in an unnecessarily bloated list of environment variables, especially as the number of uploaded images increases. For this reason, using a single variable the value of which is changed on a file-by-file basis was the preferred method.

With image identifiers available, uploading of the actual image files is a simple process. The request body contains the image, its MD5 checksum is set as a header, and the request is directed to the image endpoint by appending the identifier provided by the server previously. In this case, we are using a static set of data and therefore we can use hardcoded values for MD5 checksum of each file. Figure 12 below shows an example of a single file upload.

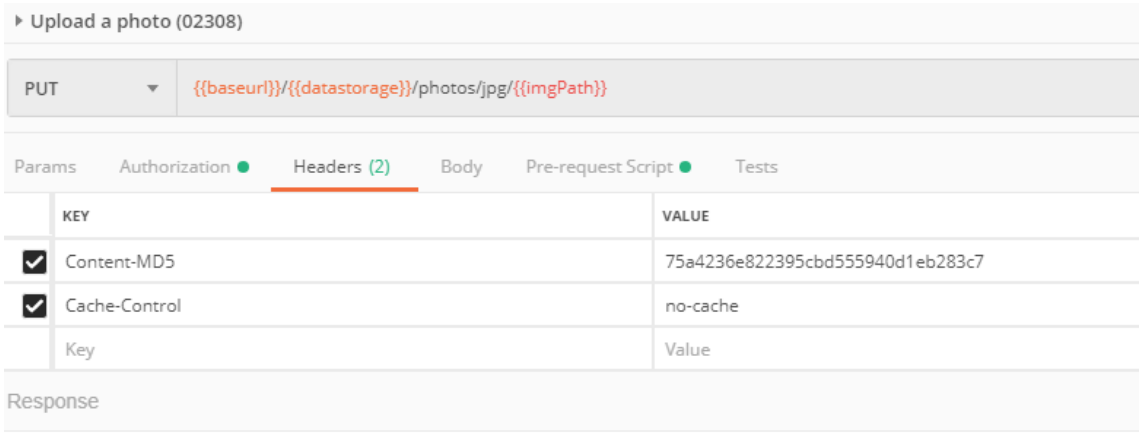

Figure 12. *File upload request example*

After upload of all images is finished, the data upload process is finished by sending a POST request to the project management service into the endpoint referring to the newly created flight within its project. Automatic photogrammetry processing is similarly launched by sending a POST request to the photogrammetry triggering endpoint associated with the project.

Doing the file upload with Selenium was implemented by using the WebDriver library in Python. WebDriver was supplemented with PyAutoGUI library [53] to generate user events in cases where Selenium is unable to locate and interact with the necessary elements, like rendered JavaScript and a file dialog pop-up window from the operating system. The upload implementation mimics the way a real user would do it, by finding and clicking elements on the screen in the same order a human user would do. Project upload implementation can be seen in Appendix B.

Doing the same file upload process automatically through the UI using SikuliX is a somewhat simple and straightforward process. The minimal amount of steps required to do the task needs to be recorded first. This is done by giving SikuliX directions about where to click by taking screenshots of all the UI elements that need to be pressed, and the context of when to look for each element.

The simple way for this is to follow a pattern of first waiting for a specific graphical pattern to appear on the screen, and then perform some UI action once the pattern is visible.

Adding some extra delays between steps is required in many cases, to make sure that the execution platform has time to receive responses from the server and render them properly in the browser window or elsewhere. If no delays are used, SikuliX will try to perform the next action written in the script immediately and fails to find the graphical context needed for the action, leading to a crash of the test script.

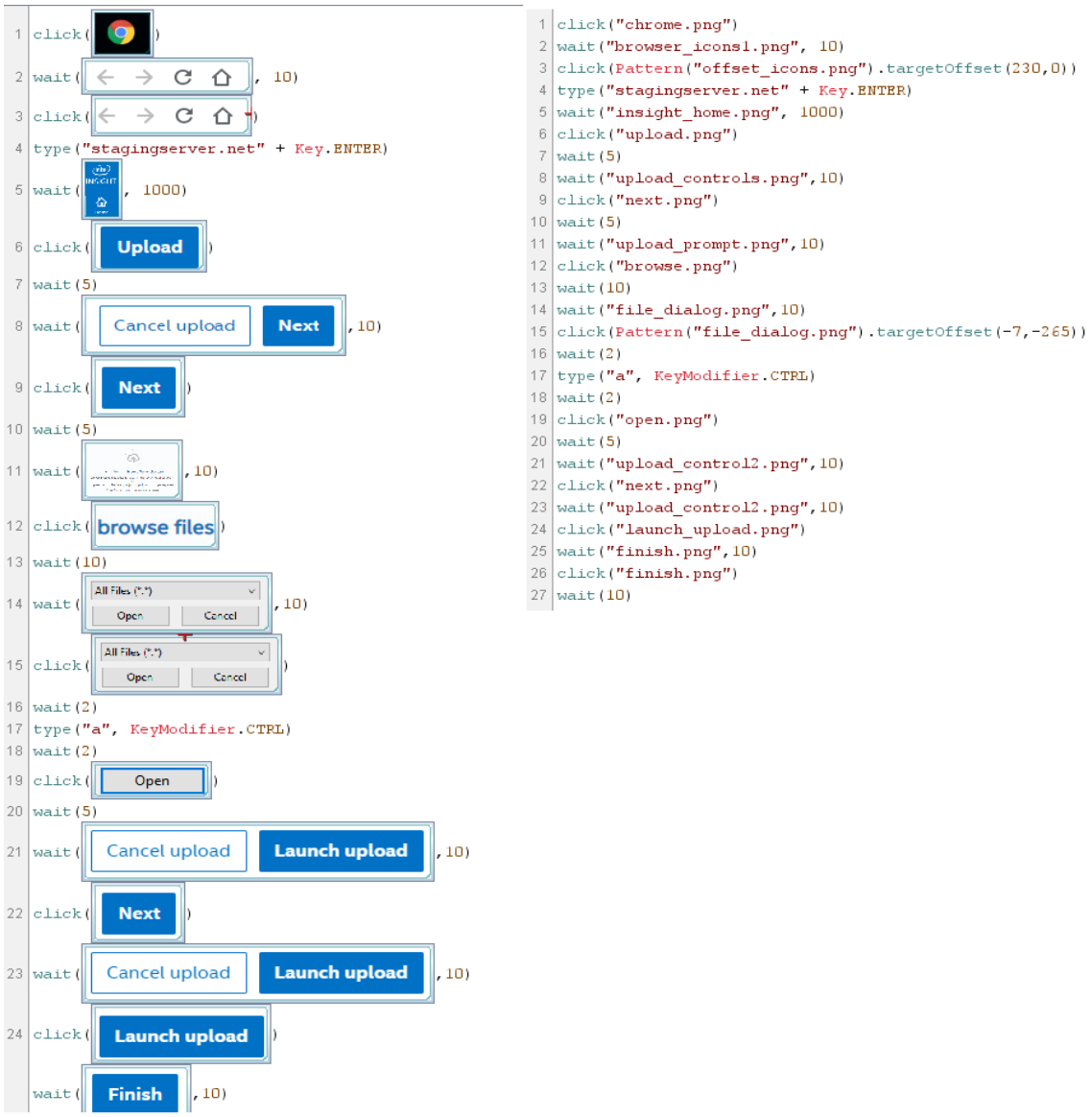

Figure 13. *Minimal SikuliX script that uploads a dataset to Intel Insight, with side-by-side comparison of the code with and without image thumbnails.*

A SikuliX script that creates a new project and selects and uploads files is presented in Figure 13. One thing to note about the script is the large number of screenshots present in the script. Scripted workflows like the above example can be defined as functions and used later in other code. This way larger automation programs can be crafted by utilizing and combining various manually defined step-by-step workflows.

Automated tests were integrated to the GitLab server where Insight code repositories are located. Postman tests were integrated directly to GitLab CI since they can be executed directly from the command line. Selenium and SikuliX tests were integrated by using Jenkins CI system with GitLab plugin and a few PCs that were used as Jenkins job executors, in order to have a full GUI environment when executing tests.

GitLab CI jobs are not executed directly on the GitLab server, but on separate systems that have a tool called GitLab Runner installed. GitLab Runners are registered separately to each repository, and when a code change is pushed to the server, the CI system uses Runners to execute a build script placed on the repository root. A minimal build script that executes project upload with Newman is shown below. Console output and all test logs of the build job are stored on the CI server and can be accessed through the GitLab web UI.

```
stages:
    - test
test:
    stage: test
    script:
        - echo "Testing"
        - cd /builds/user/postman-tests
        - newman run -e ProjectUpload_environment.json ProjectUpload_collection.json
```

The GitLab Runner was configured to create a Docker container each time to run the job. The Docker container image used was based on the official Newman image from DockerHub. In order to be used with GitLab CI, the Newman image was extended by installing Git on it since GitLab CI jobs start by fetching the full repository the runner has been attached to.

Selenium and SikuliX tests were executed by connecting GitLab to a Jenkins server via Jenkins GitLab plugin. The plugin connects the two by creating a so-called webhook, an automated HTTP request that is triggered by an event. In this case, a change made in the GitLab repository leads to the server sending a notification with HTTP to Jenkins, which leads to Jenkins triggering a build job that executes SikuliX and Selenium tests on remote job executor PCs. Console outputs and logs of the test executions are available through the Jenkins web UI.

6. REVIEW AND LEARNINGS FROM USING THE FRAMEWORK

The end result of the framework was satisfactory. The combination of Postman, Selenium, and SikuliX proved in the end to be a practical and useful solution for the required task. The main upside in all of the tools is the ease of prototyping with them and integrating the resulting prototypes into a larger automation framework and continuous integration environments is a straightforward process.

Postman proved to be a very beginner-friendly and easy test development tool overall. The GUI is simple and intuitive to use for a beginner. The use of environment variables allow flowing information from one test step to another in a simple way. The biggest issue with environment variables is that they are fundamentally just strings. If something more complex needs to be stored for later use, it has to be transformed first into JSON and stored as a string. The process has to be reversed when actually working with the store data. This leads into a series of similar or repeating blocks of code in multiple scripts and just feels like an unnecessary obstruction. Postman documentation acknowledges this by stating that API testing often requires using a lot of copy-and-pasted code, and to make the work easier has often used code blocks available that can be added to scripts with a single mouse click.

Using Postman for file uploads is not efficient. The fact that collections need to be executed sequentially request-by-request creates a considerable overhead when having to upload a large number of individual files. In the case of Insight, file upload is handled on the client code run in the browser with four parallel uploads, but Postman does not have that option. Insight server usually responds within one second from sending for most types of requests but for single image upload, the response time is usually somewhere between 15 to 20 seconds. It is easy to see how time requirements skyrocket as the dataset size increases. For example, it takes minutes to upload a very small set of 20 images.

Running collections with Newman works well, but has some limitations. If it is used to run a sequence of collections that use data stored on the server by other collections, making the data available later requires dumping the final state of all environment variables explicitly into a new file. This means that collections become coupled together by the environment they must share, and leads to unnecessary variables being present in later tests that do not use them.

In order to use collections with Newman, they need to be exported from the Postman application. If there are any file upload requests present in the exported collection, the related file paths have to be manually inserted into the file since Postman leaves them empty otherwise.

Selenium is a useful and easy tool to write tests with when the web service being tested is mainly HTML. Modern web development is moving away from that approach by using JavaScript more extensively, making Selenium less useful. Heavy use of JavaScript to create UI elements makes Selenium unable to locate them and as such other tools are required to support testing. The same applies to things operating system pop-ups, such as file dialogs, which cannot be interacted with by just Selenium.

Using Selenium to test Insight (see Appendix A for example) proved to be challenging. The HTML structure of the system is quite complex and the only reliable way of locating elements was to use the XPath attribute, the hierarchical location of the element within the HTML code. This makes tests very reliant on the base structure not changing at all, or else all tests would stop working. Insight also uses JavaScript for many functionalities, and testing those requires the use of external libraries to generate user actions, making Selenium in many instances just a test control tool rather than a test execution tool.

SikuliX is easy to begin working with and basic usage is easy when everything works. Its main downsides are the difficulty of debugging, portability, maintenance requirements and confusing or lack of proper documentation. These issues make creating complex applications with SikuliX a challenge.

Debugging difficulty comes from the fact that only error logging SikuliX natively offers is Java exception traces. SikuliX IDE will execute code with faulty syntax without checking it beforehand and offers a bare minimum of options for checking code correctness. In many cases when execution stops abruptly because of syntax issues, error logging will not point out where and how the issue manifested. To alleviate these issues, it is highly recommended to use external code editors when working with SikuliX and use the provided IDE only when it is absolutely necessary.

Portability into other environments is challenging with SikuliX programs. Things work easily only when development and execution platforms are the minimally different, otherwise things start to quickly break down here and there. The main reason this happens comes from display devices. For SikuliX to work the native resolution and DPI of displays used have to match on all platforms. Different scaling creates problems for template matching of the OpenCV library. This issue increases maintenance workload, along with the fact that the UI has to stay unchanged for things to work continuously.

Documentation of SikuliX is at times lacking useful examples and as such trial-and-error method is the only way of figuring out how things work.

7. CONCLUSIONS

The main objective of this thesis was to research and evaluate options on how to perform automated testing for microservice applications in a cost-effective and time-effective manner. Theoretical background about software architectures and technologies used in the Internet and automated testing in general were discussed on the way. All the knowledge gathered was in the end combined to create a test automation framework for Intel Insight, a microservice system.

The design process of the automation framework was discussed from the perspective of high-level requirements. These, in turn, were used to try out and evaluate a number of different tools that provided useful features for test automation. The tools that were selected for the framework were examined in-depth, and the rest were briefly discussed from the functionality point of view and the reason they were not selected.

The use of the framework in action was examined by looking at how a core use case was automated to facilitate testing the end-user workflow at multiple levels of the application. Test results can be determined from what state the system is left in after the automation has finished its execution for the end-to-end use cases and from server responses on the API level.

The resulting framework was evaluated from the perspective of developing automated tests with the tools. The framework was regarded as suitable for its purpose, but with some evident downsides to it. All the tools in the framework do their core jobs well, but as use cases get more complicated, some functionalities have plenty of room for improvement.

Future work on improving the framework is possible, as all tools have their source code publicly available. The most pressing improvements would be improving SikuliX by implementing better error reporting and some kind of tool for static code analysis to catch faulty syntax and poor logic in the code before executing scripts. Improving the SikuliX IDE would also be high on the list of improvements. Postman improvements include making the collection execution more functional, with more versatile ways than environment variables of injecting parameters for test runs. Selenium works well on the things it is capable of and as such, no immediate improvement needs came up.

This thesis focused on using free test automation tools, and commercially distributed automation tools were intentionally left out of scope. The free tier of Postman offers a wide set of features for API testing, similar tools are available on the commercial side

with more functionality and license prices starting from approximately $500 per year. Using them would make sense for an organization working primarily in the SaaS business domain where they could be utilized to their full potential. In the business environment the work of this thesis was done in, it did not make sense to explore further those options.

REFERENCES

[1] Gartner, Inc, *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.3 Percent in 2019*. [Online]. Available (accessed 9.5.2019): https://www.gartner.com/en/newsroom/press-releases/2018-09-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2019

[2] Mozilla, *Evolution of HTTP*, MDN web docs. [Online]. Available (accessed 9.5.2019): https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP

[3] Tim Berners-Lee, *The Original HTTP as defined in 1991*. [Online]. Available (accessed 9.5.2019): https://www.w3.org/Protocols/HTTP/AsImplemented.html

[4] T. Berners-Lee, R. Fielding, H. Frystyk, *RFC 1945: Hypertext Transfer Protocol -- HTTP/1.0*, Internet Engineering Task Force, 1996. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc1945

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, *RFC 2068: Hypertext Transfer Protocol -- HTTP/1.1*, Internet Engineering Task Force, 1997. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc2068

[6] R. Fielding, J. Reschke, *RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, Internet Engineering Task Force, 2014. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc7230

[7] M. Belshe, R. Peon, M. Thomson, *RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2),* Internet Engineering Task Force, 2015. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc7540

[8] International Organization for Standardization, *ISO-7498: Basic Reference Model: The Basic Model*. Available (accessed 9.5.2019): https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, *RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1*, Internet Engineering Task Force, 1999. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc2616

[10] T. Berners-Lee, R. Fielding, L. Masinter, *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*, Internet Engineering Task Force, 2005. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc3986

[11] Mozilla, *HTTP messages,* MDN web docs. [Online]. Available (accessed 9.5.2019): https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

[12] R. Fielding, J. Reschke, *RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, Internet Engineering Task Force, 2014. [Online]. Available (accessed 9.5.2019): https://tools.ietf.org/html/rfc7231

[13] Roy Thomas Fielding*, Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine,

2000. Available (accessed 9.5.2019): https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[14] Roy Thomas Fielding, *REST APIs must be Hypertext-driven,* 2008. [Online]. Available (accessed 9.5.2019): http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[15] Open API Iniative, *A Short History of the Open API Initiative and the OpenAPI Specification*. [Online]. Available (accessed 9.5.2019): https://www.openapis.org/faq#OAIFAQ-History

[16] ReDoc documentation. [Online]. Available (accessed 9.5.2019): https://github.com/Rebilly/ReDoc

[17] Docker Engine API documentation. [Online]. Available (accessed 9.5.2019): https://docs.docker.com/engine/api/v1.39/#operation/ContainerTop

[18] Zalando, *Zalando RESTful API and Event Scheme Guidelines*. [Online]. Available (accessed 9.5.2019): https://opensource.zalando.com/restful-api-guidelines/#api-first

[19] Nadareishvili I, Mitra R, McLarty M, Amundsen M. *Microservice Architecture*. 1st ed.: O'Reilly Media, Inc; 2016. p 4.

[20] Nadareishvili I, Mitra R, McLarty M, Amundsen M. *Microservice Architecture*. 1st ed.: O'Reilly Media, Inc; 2016. p 6.

[21] Martin Fowler, James Lewis, *Microservices*. [Online]. Available (accessed 9.5.2019): https://martinfowler.com/articles/microservices.html

[22] M. D. McIlroy, E. N. Pinson, B. A. Tague, *Foreword*, The Bell System Technical Journal, Volume 57, Number 6, Part 2, 1978. [Online]. Available (accessed 9.5.2019): http://emulator.pdp-11.org.ru/misc/1978.07_-_Bell_System_Technical_Journal.pdf

[23] M. J. Scheepers, *Virtualization and Containerization of Application Infrastructure: A Comparison*, Proceedings of the 21st Twente Student Conference on IT June 23rd 2014, 2014. Available (accessed 9.5.2019): https://thijs.ai/papers/scheepers-virtualization-containerization.pdf

[24] Docker documentation, *Dockerfile reference*. [Online]. Available (accessed 9.5.2019): https://docs.docker.com/engine/reference/builder/

[25] Amazon, *What is DevOps?*. [Online]. Available (accessed 9.5.2019): https://aws.amazon.com/devops/what-is-devops/

[26] Martin Fowler, *Continuous Integration*. [Online]. Available (accessed 9.5.2019): https://martinfowler.com/bliki/ContinuousDelivery.html

[27] Peter Murray, *Traditional Development/Integration/Staging/Production Practice for Software Development*. [Online]. Available (accessed 9.5.2019): https://dltj.org/article/software-development-practice/

[28] Tuukka Virtanen, *Literature Review of Test Automation Models in Agile Testing*, Master of Science Thesis Information and Knowledge Management, Tampere University of Technology, May 2018, p. 14. Available (accessed 9.5.2019):

https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/25869/Virtanen.pdf?sequence=3&isAllowed=y

[29] Mike Cohn, *The Forgotten Layer of the Test Automation Pyramid,* Mountain Goat Software. [Online]. Available (accessed 9.5.2019): https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[30] Crispin, L. & Gregory, J. 2009, *Agile testing: a practical guide for testers and agile teams,* Addison-Wesley, Upper Saddle River, NJ.

[31] Myers, G.J., Sandler, C., Badgett, T. & ebrary, I. 2012, *The art of software testing,* 3rd edn, John Wiley & Sons, Hoboken, N.J.

[32] Wikipedia, *List of unit testing frameworks*. [Online]. Available (accessed 9.5.2019): https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

[33] International Software Testing Qualifications Board, *Standard Glossary of Terms used in Software Testing*. [Online]. Available (accessed 9.5.2019): https://www.istqb.org/downloads/send/20-istqb-glossary/186-glossary-all-terms.html

[34] Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, *Using a goal-driven approach to generate test cases for GUIs*, Proceedings of the 21st international conference on Software engineering, Los Angeles, California, USA, May 16-22, 1999. pp 257-266. Available (accessed 9.5.2019): https://dl.acm.org/citation.cfm?id=302632

[35] Intel Corporation, *Intel Insight platform – Drone Data Management*. [Online]. Available (accessed 9.5.2019): https://www.intel.com/content/www/us/en/drones/solutions/intel-insight-platform.html

[36] Alan Walford, *What is photogrammetry?*. [Online]. Available (accessed 9.5.2019): http://www.photogrammetry.com/

[37] Connexion documentation. [Online]. Available (accessed 9.5.2019): https://github.com/zalando/connexion

[38] Meqa documentation. [Online]. Available (accessed 9.5.2019): https://github.com/meqaio/swagger_meqa

[39] SoapUI main page, SmartBear. [Online]. Available (accessed 9.5.2019): https://www.soapui.org/

[40] Katalon studio main page, Katalon. [Online]. Available (accessed 9.5.2019): https://www.katalon.com/

[41] Oatts documentation. [Online]. Available (accessed 9.5.2019): https://github.com/google/oatts

[42] Floating Keyboard Software Inc, Insomnia main page. [Online]. Available (accessed 9.5.2019): https://insomnia.rest/

[43] Postman, Postman main page. [Online]. Available (accessed 9.5.2019): https://www.getpostman.com/

[44] Postman Learning Center, *Environments and globals*. [Online]. Available (accessed 9.5.2019): https://learning.getpostman.com/docs/postman/environments_and_globals/variables

[45] Postman Learning Center, *Intro to scripts*. [Online]. Available (accessed 9.5.2019): https://learning.getpostman.com/docs/postman/scripts/intro_to_scripts

[46] Postman Learning Center, *Command line integration with Newman*. [Online]. Available (accessed 9.5.2019): https://learning.getpostman.com/docs/postman/collection_runs/command_line_integration_with_newman

[47] Postman Learning Center, *Viewing monitor results*. [Online]. Available (accessed 9.5.2019): https://learning.getpostman.com/docs/postman/monitors/viewing_monitor_results

[48] Selenium HQ, *Selenium history*. [Online]. Available (accessed 9.5.2019): https://www.seleniumhq.org/about/history.jsp

[49] Selenium HQ, *Selenium IDE*. [Online]. Available (accessed 12.5.2019): https://docs.seleniumhq.org/selenium-ide/

[50] Tom Yeh, Tsung-Hsiang Chang, Robert C. Miller, *Sikuli: Using GUI Screenshots for Search and Automation*, Proceedings of the 22nd annual ACM symposium on User interface software and technology, Victoria, British Colombia, Canada October 4-7, 2009. pp 183-192. Available (accessed 9.5.2019): http://up.csail.mit.edu/projects/sikuli/sikuli-uist2009.pdf

[51] Raimond Hocke, SikuliX main page. [Online]. Available (accessed 9.5.2019): http://sikulix.com/

[52] SW Test Academy, *Quick Start to SikuliX (Sikuli Script)*. [Online]. Available (accessed 9.5.2019): https://www.swtestacademy.com/quick-start-to-sikulix/

[53] PyAutoGUI documentation. [Online]. Available (accessed 9.5.2019): https://pyautogui.readthedocs.io/en/latest/

A. FILE UPLOAD BY USING SELENIUM WEBDRIVER

```
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import pyautogui

DATAPATH = "Path\to\image\data"
START_OPTIONS = webdriver.ChromeOptions()
START_OPTIONS.add_argument("--start-maximized")

# Start the browser and go to Insight staging server
DRIVER = webdriver.Chrome("../chromedriver.exe",
                          options=START_OPTIONS)
DRIVER.get("https://stagingserver.net")

# Locate textboxes for login credentials and type to them
username = DRIVER.find_element_by_id("username")
password = DRIVER.find_element_by_id("password")
username.send_keys("test_username")
password.send_keys("test_password")
password.send_keys(Keys.ENTER)
# Wait for the main view to appear, then find and click the upload
button
time.sleep(2)
DRIVER.find_element_by_xpath("/html/body/app-root/ng-compo-
nent/div/ng-component/div[1]/app-projects-panel/app-projects-panel-
header/div/div[2]").click()
# Wait for the JavaScript element for uploading appear
# Launch OS file dialog by clicking the 'Browse' button on screen
time.sleep(3)
pyautogui.click(x=966, y=575)
# Once file dialog is visible, use keyboard to navigate to test data
location
time.sleep(2)
pyautogui.press('tab')
pyautogui.press('tab')
pyautogui.press('tab')
pyautogui.press('tab')
pyautogui.press('tab')
pyautogui.press('space')
pyautogui.typewrite(DATAPATH)
pyautogui.press('enter')
pyautogui.press('tab')
pyautogui.press('tab')
pyautogui.press('tab')
pyautogui.press('tab')
# Select all files in the folder with CRTL+a and press enter to
close dialog
pyautogui.press('space')
pyautogui.hotkey('ctrl', 'a')
pyautogui.press('enter')
# Wait for the frontend code to read image files
# Then press 'Next' and 'Upload' buttons to finish the upload
```

```
    time.sleep(2)
    DRIVER.find_element_by_xpath("/html/body/app-root/ng-compo-
nent/div/app-upload-any/app-upload/app-creation/div[4]/div/but-
ton[2]/span").click()
    time.sleep(2)
    DRIVER.find_element_by_xpath("/html/body/app-root/ng-compo-
nent/div/app-upload-any/app-upload/app-creation/div[4]/div/but-
ton[2]/span").click()
    time.sleep(2)
    DRIVER.find_element_by_xpath("/html/body/app-root/ng-compo-
nent/div/app-upload-any/app-upload/app-creation/div[4]/div/but-
ton").click()
```