ElasTest, an Open-source Platform to Ease End-to-End Testing

Boni García, Micael Gallego, Francisco Gortázar and Luis López

Universidad Rey Juan Carlos,
Calle Tulipán S/N, 28933 Móstoles, Spain
{boni.garcia, micael.gallego, francisco.gortazar, luis.lopez}
@urjc.es

Abstract. The demand for larger and more interconnected software systems is constantly increasing, but the ability of developers to satisfy this demand is not evolving accordingly. The most limiting factor is software Verification and Validation (V&V), which typically requires very costly and complex testing processes. The objective of the ElasTest project is to significantly improve the efficiency and effectiveness of the testing process and, with it, the overall quality of large software systems. To that aim, ElasTest provides an integrated solution for end-to-end test automation along the development life cycle, including test case management, deployment, instrumentation, and monitoring for different kind of applications, including web and mobile.

1 Introduction

Testing large distributed and heterogeneous software systems on cloud based platforms is increasingly complex. This kind of software systems aggregates different distributed components, which are typically built and run based on Infrastructure as a Service (IaaS) combined with operation tools and services such as Continuous Integration (CI), container engines, or service orchestrators. The complete assessment of these systems is challenging since developers face with many different problems, including the difficulty to test the system as a whole due diversity of individual components, or the coordination of these components due to the distributed nature of the system [1]. Recent surveys confirm the existence of a significant gap between the current and the desired status of test automation for distributed heterogenous system, prioritizing the relevance of test automation features for these systems [2].

Software testing is a broad term encompassing a wide spectrum of different concepts [3]. Depending on the size of the System Under Test (SUT) and the scenario in which it is exercised, testing can be carried out at different levels. In their overview of cloud testing survey, Incki et al. categorize test levels in four dimensions [4]:

- Unit: individual program units are tested. Unit tests typically focus on the functionality of individual objects or methods.
- Integration: units are combined to create composite components. Integration tests focus on the interaction of different units.
- System: all of the components are integrated and the system is tested as a whole.

GarcÃ■a B., Gallego M., Gortazar F. and LÃşpez L.
ElasTest, an Open-source Platform to Ease End-to-End Testing.
DOI: 10.5220/0007904700030021
In *Challenges and Opportunities in ICT Research Projects (EPS Madrid 2017 2017)*, pages 3-21
ISBN: 978-989-758-309-4

– Acceptance: final users decide whether or not the system is ready to be deployed in the production environment. These tests can be seen as functional testing performed at system level by final users or customers.

The first three levels (unit, integration, and system) are typically carried out during the development phases of the software life cycle. These tests are typically performed by different roles of software engineers, i.e. programmers, testers, Quality Assurance (QA) team, etc. Using the classical definition of Verification and Validation (V&V) by Barry Boehm, this part is known as verification, and its aim is to ensure that the software meets its stated functional and non-functional requirements, i.e., its specification (*"are we building the product right?"*). On the other side, the fourth level (acceptance) is a type of user testing in which potential or real users are usually involved to asses if their expectations about the SUT are met. Using again the Boehm's classical definition of V&V, this part is known as validation (*"are we building the right product?"*) [5].

As illustrated in figure 1, these tests levels are usually depicted as a pyramid [6]. In the base of the pyramid we find the unit test, which are theoretically the most numerous types of tests in a software projects. Unit tests are supposed to be easy to develop and quick to execute. As we ascend to the upper levels, we find other types of tests, which are increasingly smaller in number, but also harder to develop and slower to be executed. Moreover, the capability to automate the different tests has a direct relationship with the different levels. Thus, acceptance testing is unlikely to be fully automated, since the evaluation of the final consumer always comprises some kind of human intervention. Development testing, on the other side, i.e. unit, integration and system tests, can and should be automated.

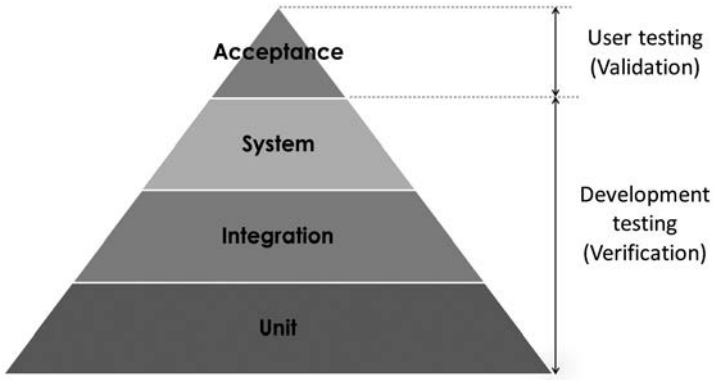

Fig. 1. Testing levels and its relationship with V&V [7].

There is a special type of system tests called end-to-end. In this approach, the final user is typically impersonated, that is, simulated using automation techniques. These tests typically drive an application through its user interface, checking that the application returns the expected results. On the one hand, this test can very valuable since they assess automatically a software system in the same way that real users do. On the other side, these tests are prone to lead potential problems, such as brittle logic, expensive to write, and time consuming to run [8]. This situation can lead to the ice-cream cone anti-

pattern, in which manual tests (which should be a reduced number on the top) increases its number more and more, while the number of down level automated tests (integration and unit) is reduced [9].

All in all, this piece of research contributes in the domain of end-to-end test automation (i.e. system tests in which the user is impersonated) for large complex distributed applications in cloud environments. To make easier this process for software practitioners, we have created an open source platform called ElasTest. As we will discover, ElasTest provides an integrated solution for test automation along the development life cycle, including test case execution, deployment, instrumentation, and monitoring for different kind of applications, including web and mobile.

The remainder of this chapter is structured as follows. Section 2 provides a brief overview in the state of the art on end-to-end testing. Section 3 provides a complete description of the ElasTest platform. In order to validate our proposal, a case study using a videoconferencing web application as SUT has been performed. The description and results of this case study are contained in section 4. Finally, the conclusions and future work are presented in section 5.

2 Background on End-to-End Testing

End-to-end testing is kind of system verification in which impersonated users interact with the SUT, emulating the real operation of the system while assessing the expected behavior. Therefore, this kind of testing is strongly related with the type of SUT. One of the most pervasive kind of software systems nowadays are web applications, being Selenium[1] the most popular open source solution for web testing automation. In this domain, Selenium WebDriver is capable of drive automatically real browsers, such as Chrome, Firefox, Opera, Edge, Safari, etc., using different programming languages, such as Java, C#, Python, Ruby, PHP, Perl, or JavaScript [10]. To that aim, Selenium WebDriver makes calls to the browser using each browsers native support for automation. The language bindings provided by Selenium WebDriver communicates with a browser-specific binary which acts as a bridge with the browser. The communication between the WebDriver script and the driver binary is done with JSON messages over HTTP using the so-called JSON Wire Protocol. This mechanism, originally proposed by the Selenium team is being standardized in the W3C WebDriver recommendation [11]. In addition to the official Selenium implementation for the WebDriver recommendation, there are different alternatives, for instance:

- WebDriverIO[2] is a custom implementation for the W3C WebDriver API written in JavaScript and distributed through the Node.js package manager (npm).
- Nightwatch.js[3] is a Node.js based custom implementation of W3C WebDriver API. It provides a clean syntax to automate browser user actions.

[1] http://www.seleniumhq.org/
[2] http://webdriver.io/
[3] http://nightwatchjs.org/

– Protractor[4] is an end-to-end framework for Angular and AngularJS applications. It has been built on the top of WebDriverJS, which is the official JavaScript implementation of Selenium WebDriver. Protactor provides extra locator strategies and built-in waits for Angular applications.

Another major component of the Selenium framework is called Selenium Grid. This component allows remote execution of Selenium WebDriver on distributed machines. The architecture of Selenium Grid is composed by a group of nodes, each running on different operating systems and with different browsers. Then, a central piece called hub (also known as Selenium Server) keeps a track of the nodes and proxies requests to them.

There are several alternatives to carry out end-to-end testing of mobile applications. For instance, Appium[5] is an open source test automation framework for use with native, hybrid and mobile web applications. Appium has been built on the top of Selenium Grid, and it allows to drive iOS and Android apps using the W3C WebDriver protocol. In Appium, instead of web browsers, mobile devices are registered in a central component called Appium Server. Following the Selenium Grid approach, the Appium Server is remotely controlled by means of W3C WebDriver messages, typically used by test scripts which use the WebDriver API [12]. Similarly, Selendroid[6] is also based on Selenium Grid to drive automatically Android native, hybrid, or mobile web applications. Both Appium and Selendroid can be used on mobile emulators and real devices.

All in all, the use Selenium Grid to drive remote browsers and mobile devices is also becoming a de facto standard nowadays. The major problem testers face is the proper provisioning of different type of browsers and mobiles, of different versions and operative systems. For that reason, many companies are growing business models basing on exposing this kind of capabilities through Software as a Service (SaaS) models, such as:

– Saucelabs[7] is a cloud solution to support remote testing based on supporting many combinations of platform (Linux, Windows, Mac OS X, Android, iOS), browser (Chrome, Firefox, Opera, etc.), and browser versions (including beta and development releases).
– BrowserStack[8] is another cloud provider for mobile and desktop browsers across different browser (Chrome, Firefox, Edge, Opera, etc.), operating systems (Windows and Mac OS X) and mobile devices (iOS, Android, Windows Phone).
– Browserling[9] is a live interactive testing service that provides cross-browser testing for web applications.
– Nightcloud[10] is a cloud-based platform being developed by the Nightwatch.js team.

[4] http://www.protractortest.org/
[5] http://appium.io/
[6] http://selendroid.io/
[7] https://saucelabs.com/
[8] https://www.browserstack.com/
[9] https://www.browserling.com/
[10] https://nightcloud.io/

In recent times, end-to-end testing solutions are evolving around Artificial Intelligence (AI) and Machine Learning (ML) approaches [13]. In this new wave, we find the following projects:

- Test.AI[11] incorporates an AI brain to Selenium and Appium tests, allowing to identify screens and elements dynamically in any application, automatically driving it to execute test cases defined following a Behavior-Driven Development (BDD) approach. As a result, it provides a test report with verdicts about the application functionality, user experience design (UX) and performance.
- Mabl[12] is focused on front-end black-box testing for web applications. It executes tests using real web browsers capturing different sources of output, including sources, screenshots, timing information, etc. Then, it analyses the output by means of ML techniques.
- Endtest[13] is a codeless automated testing engine which allows to create automated tests without having to write any code but using a web wizard. These tests are later executed in the Endtest own cloud infrastructure. It uses machine learning techniques to fix and improve existing automated tests.
- Applitools[14] is AI-based engine that automatically validates visual aspects for web and mobile applications. Using applications screenshots as baseline input, Applitools uses AI-powered computer-vision algorithms to detect and report any difference found between the real application and the screenshots baseline. Applitools is available as a public or private cloud-service as well as on premise.
- Testim[15] uses machine learning to manage automated tests that can be executed on multiple web and mobile platforms. Testim focuses on reducing flaky tests and maintenance. To that aim, Applitools learns from every execution, analyzing in real-time to keep track of element locators, self-improving the stability of test cases.
- Sealights[16] uses machine learning-like technology to analyze both the SUT code and tests allowing to find out the exact code coverage for the complete test suite, including unit, integration, security, etc.

3 The ElasTest Approach

The main benefit of end-to-end tests is the simulation of real user scenarios in an automated fashion. Nevertheless, this kind tests have several important deterrents which are stopping its wide adoption by software practitioners. First of all, to carry out end-to-end testing a pre-requisite is to deploy completely the SUT. This introduces extra effort in the testing process, since the SUT need to be built, deployed, and available for end-to-end tests to run. Second, end-to-end tests does not isolate failures. Tracing a failed end-to-end test is usually costly for testers [14], since the underlying fault can

[11] https://testaimobile.com/
[12] https://www.mabl.com/
[13] https://endtest.io
[14] https://applitools.com/
[15] https://www.testim.io/
[16] https://www.sealights.io/

be anywhere in the software, and finding it is a cumbersome task. Third, the need of user impersonation in end-to-end tests is also challenging. Testers and DevOps should provide the proper infrastructure to carry out user impersonation typically in terms of web browsers and mobile devices. The rapid evolution of these assets forces a constantly updated underlying infrastructure ready to be used by end-to-end tests. Fourth, sometimes end-to-end tests tend to be unreliable. A test is refereed as flaky when exhibits both a passing and a failing result with the same conditions. There are many root causes for flaky results, including concurrency, non-deterministic or undefined behaviors, infrastructure problems, among others. Due to end-to-end tests asses the whole SUT at once, flaky results are more likely to occur than in other smaller tests (i.e. unit or integration). Fifth, compared to other kind of tests, especially unit tests, end-to-end tests tend to be slow. Last but least, the verification of non-functional aspects from a end-to-end approach is also challenging. Quality attributes such as performance or scalability are difficult to be properly assessed and efficient end-to-end non-functional tests are often skipped.

The ElasTest[17] project is a European Commission funded project in the context of the Horizon 2020 Programme (H2020). ElasTest has born with the big ambition of providing a comprehensive open source platform to ease the process of development, operation, and maintenance of end-to-end tests for different kind of applications, including web and mobile. To that aim, ElasTest proposes different solutions to the aforementioned problems. Table 1 provides a summary of the ElasTest approach (i.e. known problems and proposed solution), which are elaborated in the following paragraphs.

Table 1. Known problems in end-to-end tests and solution proposed by Elastest.

End-to-end tests problem	**ElasTest solution**
Costly to create & maintain	Comprehensive test management platform through web interface, REST APIs, and build server integration
Not isolating failures	Integrated log analyzer and monitoring tool both during test execution (online) and after tests (offline)
Difficult to impersonate users	SaaS browser/mobile devices with advance QoE capabilities
Unreliable & flaky	SUT instrumentation and ML for recommending testing actions for decision taking
Hard to write & slow to run	Test orchestration (divide and conquer approach, promote reusability)
Difficult for non-functional	Reuse instrumentation and orchestration functional capabilities to other quality attributes

Regarding the problem about the effort needed to create and maintain end-to-end tests, ElasTest provides a ready to be used open source platform aimed to simplify this process. To that aim, the platform allows to build a custom testing environment based on three major concepts, namely *System Under Test (SUT)*, *Test Job (TJob)* and *Test Support Services (TSS)*. First, the SUT in the software to be tested. ElasTest uses

[17] http://elastest.io/

Docker containers to configure, build, and deploy SUTs. Docker[18] is an open source software technology which allows to pack and run any application as a lightweight and portable components called containers. Second, TJobs is the name given in the ElasTest jargon to the test entities to be executed in ElasTest. TJobs are technologically neutral. In other words, ElasTest supports tests coded in any language and using any testing framework. Again, and for the shake of compatibility, ElasTest relies on Docker container to run the specific tests within a TJob. This way, tests can be implemented in different technologies such as Java, JavaScript, Python, etc. Finally, TSS are the services offered by ElasTest to TJobs. TSSs can be seen as the building blocks implementing specific test capabilities within ElasTest. For the shake of usability, the management of SUTs, TJobs and TSSs can be done using three different ElasTest's interfaces:

- Web GUI: ElasTest provides a universal access point through a web interface implemented as a Single Page Application (SPA) with the Angular[19] framework. Next sections provides some examples on how to use this interface.
- REST API: The ElasTest capabilities have been designed to be consumed using REST (Architectural Styles and the Design of Network-based Software Architecture), which is an architectural style for designing distributed systems. The ElasTest REST API allows the integration with third-party software in custom ways.
- Jenkins plugin: To facilitate the integration of ElasTest with existing Continuous Integration (CI) infrastructure, ElasTest provides a Jenkins plugin. Jenkins[20] is a well-known open source build server which supports building, deploying, and automating any project.

In order to help in locating a fault when a end-to-end test fails, ElasTest provides and a complete integrated log analyzer tool suite. The idea behind this tool is not solving the problem on isolating failures (and this is inherent to the nature of end-to-end tests) but to provide the capability of keeping synchronized trace of the whole logging information of the SUT components during tests executions. These tools have been designed to work both during the execution of a TJob (online mode) but also after that (offline mode).

Regarding the difficult to impersonate users, ElasTest provides a custom TSS enabling the impersonation of end-users in their tests through the SUT's GUI. This service has been designed to provide full compatibility with external browser/mobile drivers, but enhanced with extra capabilities, such as event subscription, log gathering, and even advance media capabilities from the perspective of its incoming perceived Quality of Experience (QoE) [15] for web-based videoconferencing applications (WebRTC). Measuring QoE is in general a complex topic and this task shall perform the appropriate research activities for evaluating the most suitable way of doing it, which may involve simple mechanisms such as evaluation of response-time from the GUI.

Regarding the unreliability or possible flakiness of end-to-end tests, again this problem is not easily solved since the underlying cause is usually constrained in the boundaries of the SUT or the tests. Nevertheless, we believe that ElasTest can contribute

[18] https://www.docker.com/
[19] https://angular.io/
[20] https://jenkins.io/

to the solution of this problem in two aspects: identification and reduction of unreliable/flaky end-to-end tests. On the one hand, in order to reduce the number of flaky tests, we propose to instrument properly the SUT in order to enhance the ability of customization. In ElasTest we conceive instrumentation as extending the interface exposed by a software system for achieving enhanced controllability (i.e. the ability to modify behavior and runtime status) and observability (i.e. the ability to infer information about the runtime internal state of the system) [16]. This customization allows to reproduce or simulate different operational behavior which can be useful to isolate the cause of test unreliability. On the other hand, in order to reduce flakiness, we propose a custom recommendation engine for decision taking. This service is based on leveraging machine learning and cognitive computing techniques for providing two types of features. First, plain recommendation, which consists of using conventional recommender systems that, learning from different sources of data (e.g. past testing suites, test specifications, SUT specifications, code comments, etc.), recommend the tester about creating new TJobs or orchestration logic that might be missing in a given test suite. Second, cognitive computing technologies (such as IBM Watson[21]) to make possible for ElasTest to answer questions made by the developer in natural language in relation to testing strategies or techniques for a given purpose through the reuse of testing knowledge [17].

Concerning the problem related to the difficulty to write end-to-end and its slowness to be executed, ElasTest cannot implement a magic wand to solve these problems, since they are closely related to the nature of this kind tests. Nevertheless, ElasTest proposes a technique to minimize the problem. Our vision is to implement what we call *test orchestration*. We understand this concept as a technique for executing tests in coordination, i.e. to combine intelligently testing units for creating a more complete test suite following the divide and conquer principle [18]. In ElasTest we concentrate on testing large software systems created by the orchestration of simple components. Typically, those software systems are validated using CI tools and methodologies. For example, imagine one of such SUT providing some kind of service (e.g. through a Web GUI). This type of application is commonly architected following a microservice model. Based on it, developers create the application code and, upon some events (e.g. code commits), a CI system automates the build and deploys the SUT using some cloud provider services (e.g. cloud orchestration, load balancers, cloud storage, etc.). At this point the validation of the SUT as a whole takes place). This happens by executing some jobs or workers that launch monolithic testing processes (i.e. TJobs) that evaluate some properties of the SUT.

Filly, non-functional testing is considered as one of the main unsolved challenges [19]. In this domain, one of the main contributions of ElasTest beyond state of the art is the creation of a technique for testing non-functional properties of SUT in a seamless way. This is materialized by the combination of ElasTest instrumentation and orchestration, which enables functional TJob tests to be transformed into non-functional. The idea is to reuse a functional TJob for evaluating the behavior of the SUT under different load conditions (e.g. by orchestrating parallel executions of the TJob) or under custom operational conditions (i.e. by the parallel execution augmentators), so that the resulting orchestrated SUT may use the timing information for assessing QoE and performance

[21] https://www.ibm.com/watson/

(in terms of time-response) under failures, degraded network conditions and load. It is important to remark that the non-functional validation logic is agnostic to the TJob and SUT as it is fully maintained by the orchestration rules. This makes such logic fully re-usable across different TJob and SUT.

3.1 Test Management

This section provides the detail to manage test using two of the main building blocks introduced before (SUTs and TJobs) from the ElasTest GUI. To manage a group of SUTs and associated TJobs, ElasTest manages the concept of *Project*. Figure 2 shows an screenshot of the ElasTest GUI for a given project.

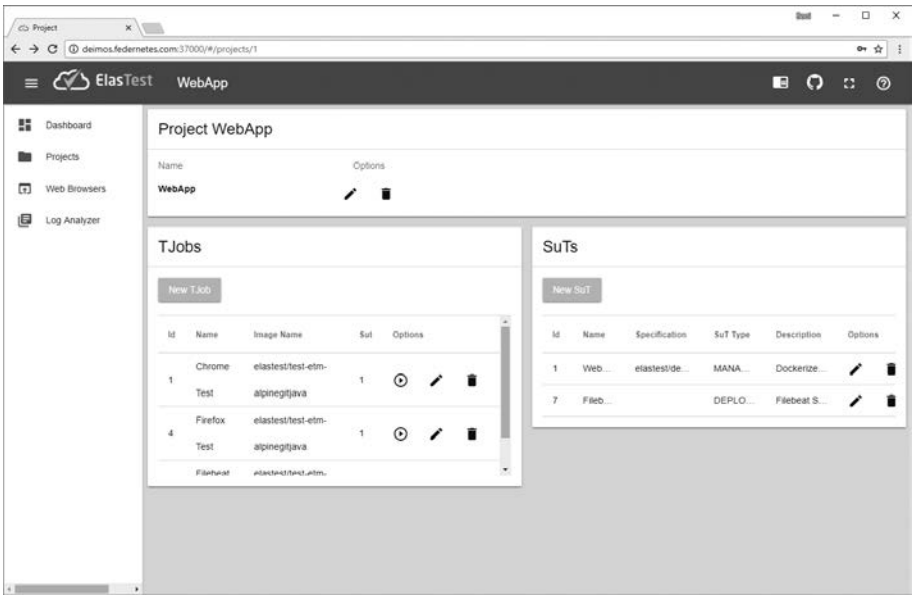

Fig. 2. Project management on the ElasTest GUI.

ElasTest allows to manage different flavors of SUTs. The first option is to manage already deployed software somewhere and available through a public URL. In this case, by default the SUT is not instrumented in any means. Nevertheless, the SUT itself could install its own instrumentation agents if needed. Moreover, the SUT can be based on Docker. This option allows to deploy a SUT using Docker containers in two different ways:

- Using Docker images. The SUT is packaged as a single Docker image hosted in the official Docker cloud service for distributing containers, called Docker Hub[22].
- Using Docker compose, which is a tool for defining and running multi-container Docker applications. To use this option, the SUT description is done using Docker

[22] https://hub.docker.com/

Compose files (`docker-compose.yml`). An example of this kind of configuration can be seen on figure 3.

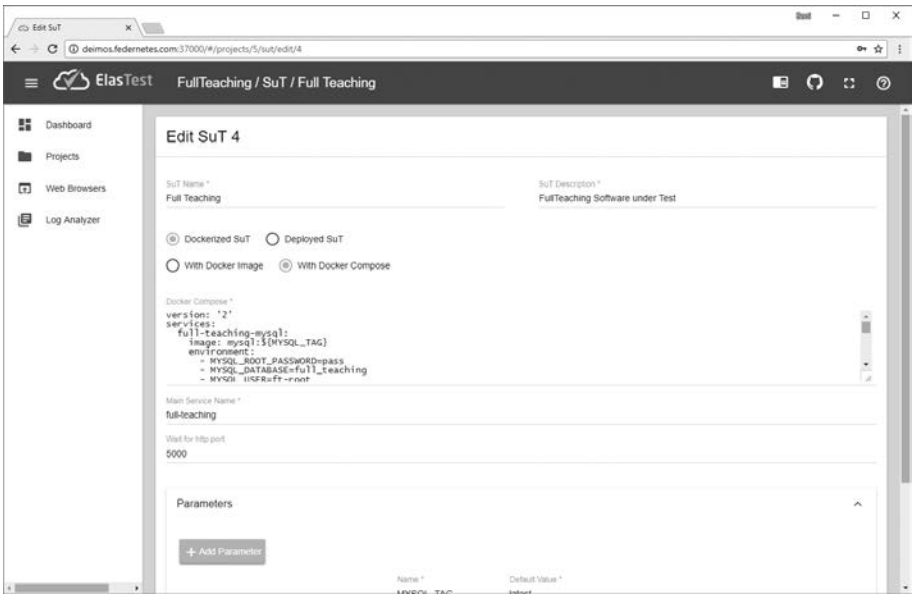

Fig. 3. SUT management on the ElasTest GUI.

The next important asset in ElasTest are of course the test, managed inside ElasTest as TJobs. In a TJob configuration several parameters need to be specified. First of all, a TJobs need to select a SUT. Second, the required environment need to be configured, using a Docker image in which test are executed. Finally, ElasTest need to know how to run the tests within the container. This is done using a set of commands written in bash script. Once the TJob configuration is complete, the TJob can be executed. Figure 4 shows an example of a TJob execution. Notice that in ElasTest presents the execution logs and metrics in real time on the GUI.

3.2 Monitoring

One of the key features of ElasTest is the ability to show and analyze logs and metrics of all elements involved in a test. This is particularly interesting for end-to-end tests, which usually involve more complex system architectures in distributed environments. When a test is executed using ElasTest, the tester can see all that monitoring information in the same graphical user interface and with advanced analysis features.

Metric information from ElasTest core components (SUTs, TJobs, and TSSs) are gathered in the monitoring service. This service is based on event streams. Every log and metric entry is called an event. The collection of events containing information about the same metric or log is called event stream. Any SUT, TJob or TSS can generate multiple event streams. All of them can be gathered and visualized by ElasTest by

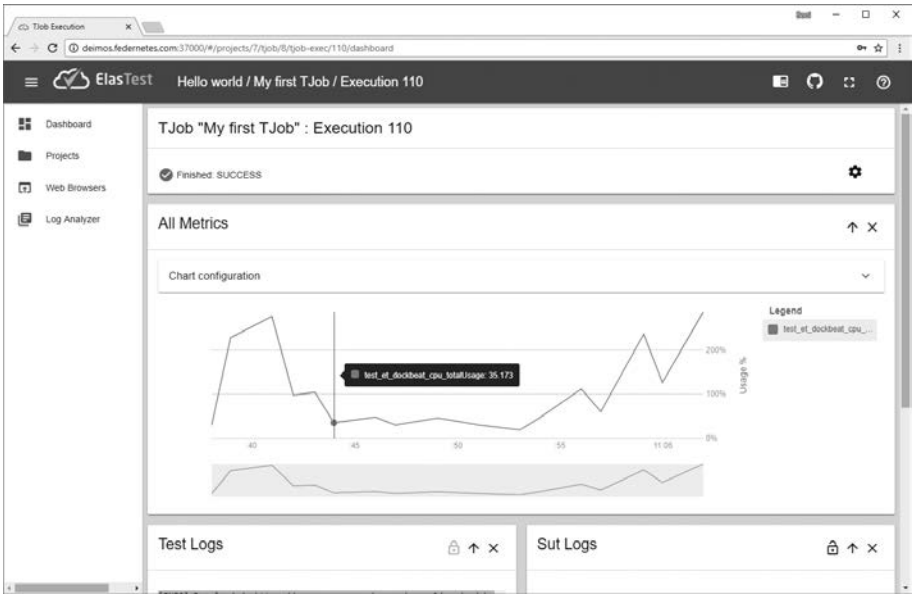

Fig. 4. TJob execution on the ElasTest GUI.

different charts in real time and stored for further analysis. The ElasTest monitoring service is offered in three flavors:

- Dockerized Monitoring. When launching tests against a dockerized SUTs, ElasTest manages the SUT lifecycle itself and use Docker features of logs and stats to automatically get monitoring information from the container (see example chart in figure 4).
- Deployed Monitoring. When ElasTest is executing tests against an already deployed SUT, monitoring information can be sent from the SUT to ElasTest using Beats technology. Beats[23] is a platform for single-purpose data shippers created to work with Logstash[24] (server-side data processing pipeline that ingests data from a multitude of sources simultaneously) and ElasticSearch[25] (RESTful search and analytics engine).
- Custom Monitoring. HTTP POST request can be sent to Logstash in order to provide custom monitoring information. The body of this POST request is a JSON message with several fields (*timestamp*, *message*, *component*, etc.) described in next section.

[23] https://www.elastic.co/products/beats
[24] https://www.elastic.co/products/logstash
[25] https://www.elastic.co/

3.3 Log Analyzer

ElasTest provides several tools for analyzing the logs gathered during any TJob execution. By default, the log information is organized using a tabular format composed by the following fields (see Figure 5):

- *timestamp*: Date and time of the log entry.
- *message*: Actual log entry message.
- *level*: Logging level of the entry: `DEBUG`, `INFO`, `WARNING`, `ERROR`, etc.
- *component*: Component that generated the entry (SUT, TJob)
- *stream*: Each single component can generate different groups of logs. This value is used to identify these groups.
- *exec*: TJob execution identifier.

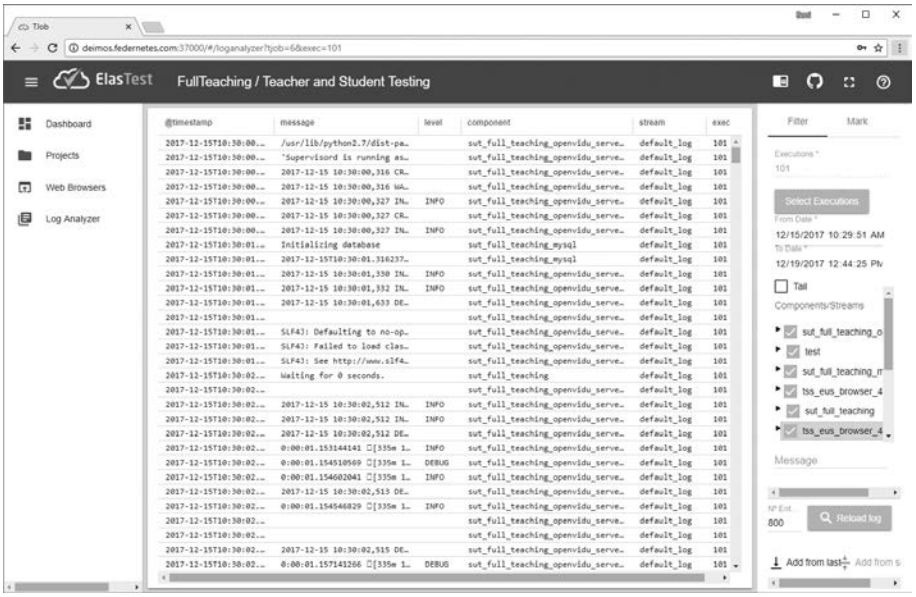

Fig. 5. Log analyzer on the ElasTest GUI.

ElasTest provides two tools out-of-the-box to analyze the gathered logs. First, the *Filter* tool allows to select a group of logging entries, filtering by date, component, stream, or message. Second, the *Mark* tool allows to search in the log entries, coloring the results as shown in Figure 6.

3.4 User Impersonation

In order to expose user impersonation an API in a universal way, ElasTest implements an extension of the W3C WebDriver API [11]. As presented in section 2, this recommendation is used to drive browsers and mobile devices, by means of a client-server technology implemented by Selenium and Appium respectively. The vision of ElasTest

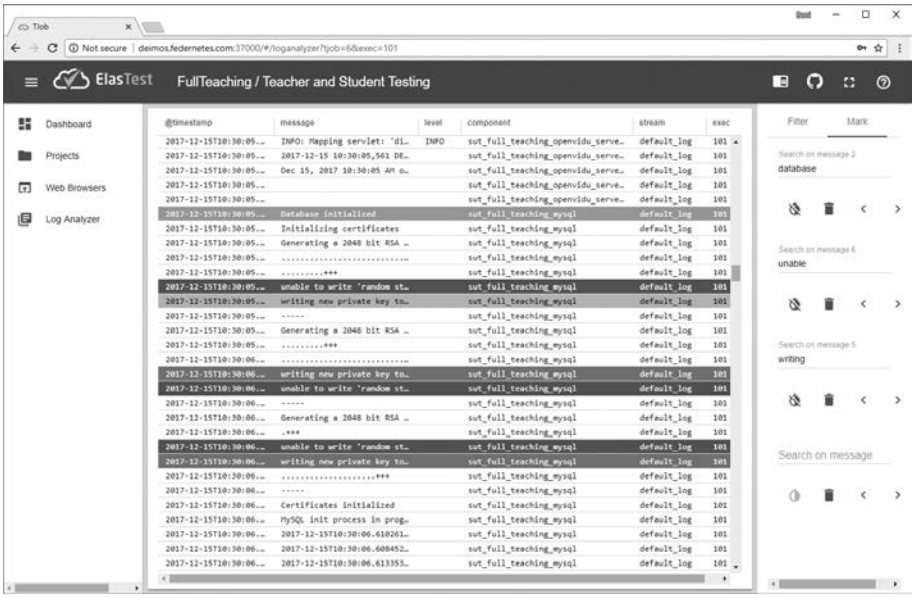

Fig. 6. Log marker on the ElasTest GUI.

is to enhance the current support with additional advance capabilities in a seamless and integrated solution. Moreover, the user impersonation service provided by ElasTest has been evolved into a fully Software as a Service (SaaS) model so that developers do not need to take into consideration problems related to computing resources scheduling, software provisioning or system scaling.

The user impersonation service can be consumed in two main ways. First, using the ElasTest web GUI, a tester can ask for live browser sessions. As a result, ElasTest allows to control the browser using an HTML5 canvas element connected to the browser using Virtual Network Computing (VNC). For a user point of view, the browser is embedded in the ElasTest GUI, and it can be used to carry out manual web navigation using different kind of browsers, such as Chrome, Firefox, etc. Figure 7 shows an example of live browser session in the ElasTest GUI.

The second way to use the impersonation service is driven by test logic using WebDriver. Due to the fact that the user impersonation service is an extension of the W3C WebDriver, it can be consumed by any existing Selenium client (as described in section 2). An example of this kind of operating mode is shown in the case study presented in section 4. In both cases (manual and WebDriver browsers), ElasTest records of the generated sessions (see figure 8) to watch the interaction/navigation using the browser.

3.5 ElasTest Architecture

The ElasTest architecture is depicted in figure 9. First of all, we find a component called ElasTest Test orchestration and recommendation Manager (ETM), which is the access

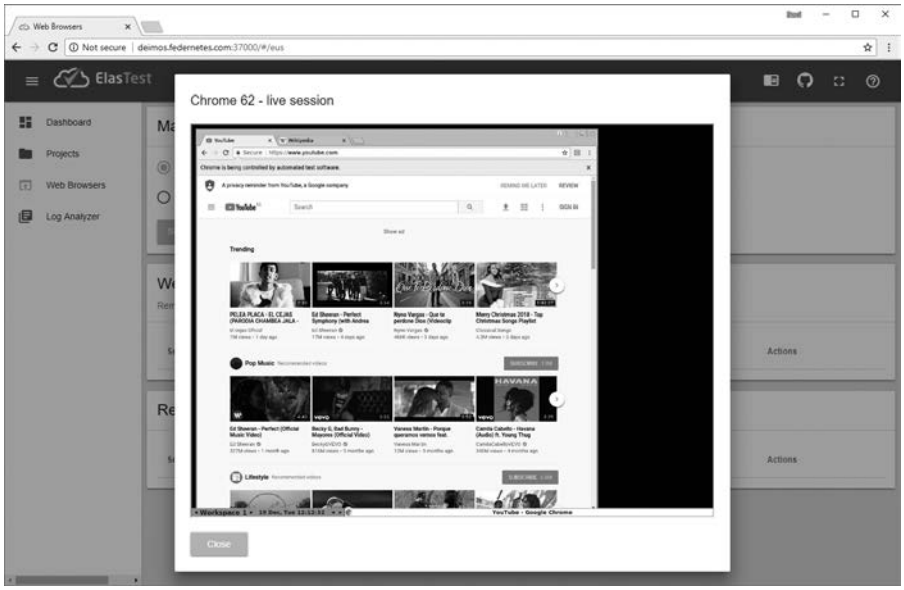

Fig. 7. Live browser on the ElasTest GUI.

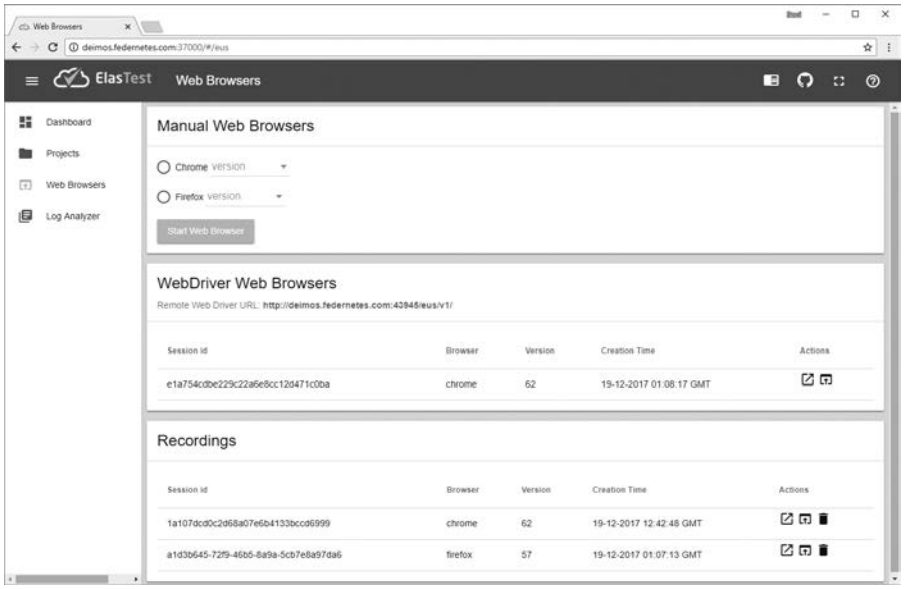

Fig. 8. Recording sessions in user impersonation services on the ElasTest GUI.

point to the framework. It coordinates all other components exposing different interfaces for consumers, such as a web GUI, a command line interface (which consumes a REST API), and also an interface with a custom Jenkins plugin.

ElasTest follows a microservices approach, and the component which is responsible for discovering and operating the different services that ElasTest make available to

tests is called ElasTest Service Manager (ESM). This component is based on the Open Service Broker API (OSBA)[26] for discovering, registering and unregistering services within the platform. RabbitMQ[27] is used as messaging queue for the events communication among the different services. The management of browsers and mobile devices for user impersonation as described on section 3.4 is implemented in a component called ElasTest User impersonation Service (EUS).

One of the key aspects handled out of the box by ElasTest is related with data management. During its operation, ElasTest gathers different sources of data from test execution, including SUT logs, different types metrics (including SUT resource consumption, packet-loss in the network traffic, or node failures, among others), or custom files issued by services (such as browser/mobile session recordings carried out by EUS). The component responsible for the persistence layer is called ElasTest Data Manager (EDM), and it has been built on the top of on MySQL[28] as relational database, Elasticsearch as search engine, and Alluxio[29] as virtual distributed storage system.

The ElasTest Instrumentation Manager (EIM) provides the capability of instrumenting the SUT to inject potential system failures like packet-loss, network bandwidth adjustments to emulate real conditions, CPU bursting, and node failures, to name a few. To that aim, Beats agents are installed together with the SUT.

Finally, the ElasTest Platform Manager (EPM) is the component responsible of isolating the ElasTest services from the underlying infrastructure. The supported cloud infrastructures are OpenStack[30], Amazon Web Services[31] (AWS), Docker and Kubernetes[32]. Moreover, Open Baton[33] is used for orchestrating the SUT and the network services within the ElasTest platform.

4 Case Study: Testing WebRTC Applications with ElasTest

We have carried out an initial validation of ElasTest based on a case study focused for WebRTC applications. WebRTC is the umbrella term for a number of emerging technologies that extends the web browsing model to exchange real-time media with other browsers [20]. Market momentum around WebRTC is growing very fast nowadays, and therefore, it is imperative for software testers to have a strategy in place in order to assess WebRTC applications efficiently. Nevertheless, testing WebRTC-based applications in a consistently automated fashion is a challenging problem. As this case study shows, the use of ElasTest simplifies the process of testing this kind of applications at different levels.

[26] https://www.openservicebrokerapi.org/
[27] https://www.rabbitmq.com/
[28] https://www.mysql.com/
[29] https://www.alluxio.org/
[30] https://www.openstack.org/
[31] https://aws.amazon.com/
[32] https://kubernetes.io/
[33] https://openbaton.github.io/

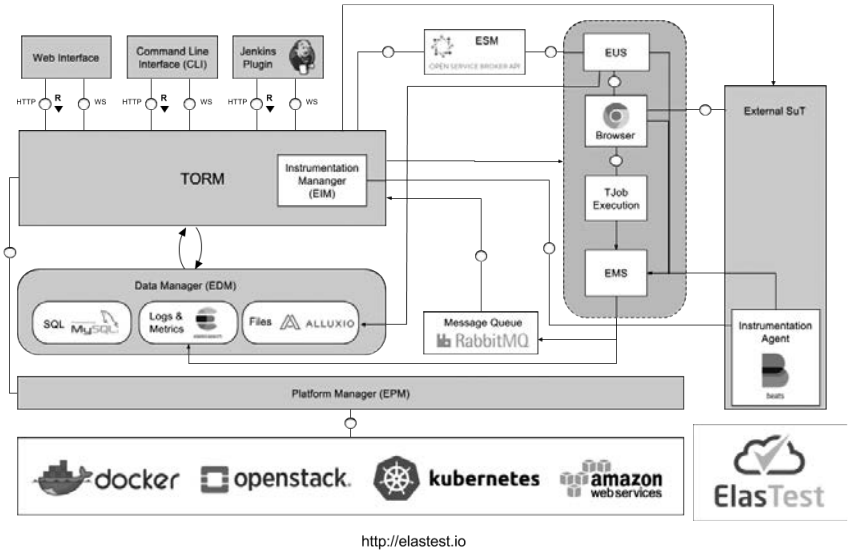

Fig. 9. ElasTest architecture.

To carry out this case study we have cooperated with the team developing the project OpenVidu[34], an open source videoconferencing WebRTC framework. OpenVidu follows a client-server architecture and therefore is made up by two main components. On the client-side, the OpenVidu Browser is a JavaScript/TypeScript library which allows to create video calls, join users to them, and send/receive media streams. On the server-side, the OpenVidu Server receives the operations from clients establishing and managing the video-calls.

In order to carry out end-to-end tests of WebRTC applications, it is mandatory to use modern browsers that implement the WebRTC stack, such as Chrome or Firefox. For that reason, in the OpenVidu project, end-to-end tests have been implemented using Selenium WebDriver. In the testing process carried out by the OpenVidu team, these tests were executed in a Jenkins CI server. In this server the latest versions of Chrome and Firefox were installed, and Selenium sessions were executed through a virtual framebuffer display server (Xvfb).

The research question driving this case study is the following: *Is the ElasTest capable to ease the end-to-end testing process within the OpenVidu project?*. To address this question, an instance of ElasTest was provided to the OpenVidu team. The idea was to reuse the existing tests, adapting them to be executed inside ElasTest. Due to the fact that the existing test suite was based on Selenium WebDriver, few changes was required in the test logic. The existing codebase was implemented in Java, and therefore the required change was related to the specific objects to control browsers (i.e. `ChromDriver` for Chrome and `FirefoxDriver` for Firefox) by remote browser drivers, called `RemoteDriver` in Java. These objects require the URL to connect with the Selenium Server, which is implemented in ElasTest by EUS.

[34] http://openvidu.io/

The SUT lifecycle was managed by ElasTest together with the test execution. In this case study, a Docker Compose script was configured within the ETM, defining the OpenVidu application under test and its dependencies. Figure 10 provides an ETM screenshot of the execution of one end-to-end test against the SUT while it is executed by the EUS. As explained in the section before, once the test finished, a recording of the session navigation, together the the browser logs is stored persistently in ElasTest.

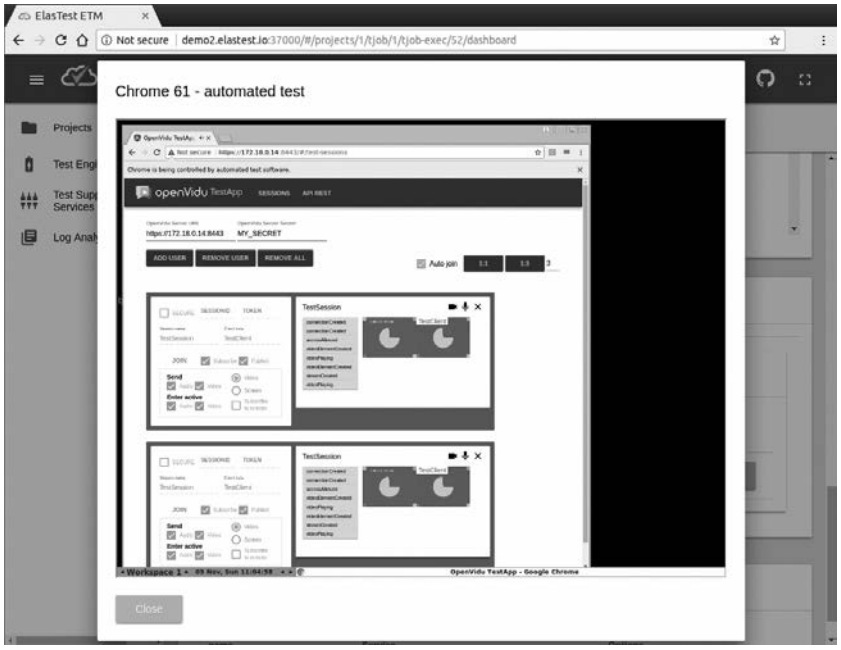

Fig. 10. Screenshot of ETM/EUS during the execution of a OpenVidu end-to-end test.

All in all, we conclude that the fact that EUS is based on the W3C WebDriver standard, facilitates its adoption in an existing test codebase. Second, the capability to provide different types of browsers and version in a seamless and elastic manner is very valuable for testers, since it avoids to manage directly the infrastructure reducing the efforts required mainly in DevOps side, and providing valuable assets for testers to create effective tests. Finally, the capability for storing to the browser session recording and logging makes a big difference for OpenVidu testers. This feature allows to trace and debug failed tests in a much more reliable way than before, in which testers were blind to trace errors of tests executions on their Jenkins infrastructure.

5 Conclusions and Future Work

Modern software systems are increasingly complex. Nowadays, architectures involving distributed heterogeneous services, cloud native, and microservices are more and more common. This kind of systems are constantly evolving, forcing to practitioners to invest relevant efforts to have efficient testing strategies in form and shape.

End-to-end tests are a special type of system test in which users are impersonated using automation techniques. This kind of tests are challenging at different levels. First of all, end-to-end tests are usually costly in terms of be development, operation, and maintenance. By definition, in order to run end-to-end tests the SUT needs to be deployed in the first place. Therefore, when the SUT is large and complex, relevant efforts are required as a prerequisite of the actual testing stage. Moreover, end-to-end tests does not behave efficiently in terms of failure isolation. Also by definition, end-to-end assess the complete system as a whole just like the final user does. When some failure happens, the capability to identify the underlying fault can be tricky, especially again the the SUT is composed by many heterogeneous distributed components.

The ElasTest project strives to improve the end-to-end testing process for large distributed and heterogeneous software systems. To that aim, we are creating an open source testing platform providing different solutions to the known weakness of end-to-end tests. ElasTest provides a comprehensive test management platform through web interface, REST APIs, and Jenkins integration. In addition, ElasTest provides a fully integrated log analyzer and monitoring tool which make easier to locate the underlying fault of failing end-to-end tests. Finally, ElasTest implements a user impersonation service the emulation of end-users in their tests through GUI instrumentation. This service provides full compatibility with external browser/mobile drivers, evolved into a fully SaaS model so that developers do not need to take into consideration problems related to computing resources scheduling, software provisioning or system scaling.

At the moment of this writing, ElasTest is still in its infancy, and therefore, some planned features are yet to be released. Among these features, we find the measurement of the end-users perceived QoE, the integration of machine learning and and cognitive computing for recommending testing actions for decision taking, or the support of mobile and sensor devices emulators for user impersonation.

Acknowledgments. This work has been supported by the European Commission under project ElasTest (H2020-ICT-10-2016, GA-731535); by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) cofunded by FSE & FEDER; and Spanish Government under project LERNIM (RTC-2016-4674-7) cofunded by the Ministry of Economy and Competitiveness, FEDER & AEI.

References

1. Mili, A., Tchier, F.: Software testing: Concepts and operations. John Wiley & Sons (2015)
2. Lima, B., Faria, J.P.: A survey on testing distributed and heterogeneous systems: The state of the practice. In: International Conference on Software Technologies, Springer (2016) 88–107
3. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: 2007 Future of Software Engineering, IEEE Computer Society (2007) 85–103
4. Incki, K., Ari, I., Sözer, H.: A survey of software testing in the cloud. In: Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on, IEEE (2012) 18–23
5. Boehm, B.: Software engineering: R & D trends and defense needs. Research Directions in Software Technology. MIT Press: Cambridge MA (1979)

6. Cohn, M.: The forgotten layer of the test automation pyramid (2009)
7. García, B.: Mastering Software Testing with JUnit 5. Packt Publishing (2017)
8. Fowler, M.: Test pyramid (2012)
9. Scott, A.: Introducing the software testing ice-cream cone (anti-pattern) (2012)
10. Vila, E., Novakova, G., Todorova, D.: Automation testing framework for web applications with selenium webdriver: Opportunities and threats. In: Proceedings of the International Conference on Advances in Image Processing. ICAIP 2017, New York, NY, USA, ACM (2017) 144–150
11. Stewart, S., Burns, D.: Webdriver. Working draft, W3C (2017)
12. Shah, G., Shah, P., Muchhala, R.: Software testing automation using appium. International Journal of Current Engineering and Technology 4 (2014) 3528–3531
13. Noorian, M., Bagheri, E., Du, W.: Machine learning-based software testing: Towards a classification framework. In: SEKE. (2011) 225–229
14. Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. John Wiley & Sons (2011)
15. ITU-T, R.P.: 10/g. 100 amendment 1,new appendix i–definition of quality of experience (qoe). International Telecommunication Union (2007)
16. Natella, R., Cotroneo, D., Madeira, H.S.: Assessing dependability with software fault injection: A survey. ACM Computing Surveys (CSUR) 48 (2016) 44
17. Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A.A., Lally, A., Murdock, J.W., Nyberg, E., Prager, J., et al.: Building watson: An overview of the deepqa project. AI magazine 31 (2010) 59–79
18. Zelkowitz, M.V., Shaw, A.C., Gannon, J.D.: Principles of software engineering and design. Prentice-Hall Englewood Cliffs (1979)
19. Orso, A., Rothermel, G.: Software testing: a research travelogue (2000–2014). In: Proceedings of the on Future of Software Engineering, ACM (2014) 117–132
20. Loreto, S., Romano, S.P.: How far are we from webrtc-1.0? an update on standards and a look at what's next. IEEE Communications Magazine (2017)