



Escola d'Enginyeria de Telecomunicació i  
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA



# TREBALL FINAL DE GRAU

**Título:** USO E INTEGRACIÓN DE SOFTWARE EN CI/CD PARA REDUCIR VULNERABILIDADES Y REALIZAR CARGA DE TRABAJO/ESTRÉS

**Autor:** Manuel Fraga Lobato

**Tutora:** Olga León Abarca

**Supervisora:** Noemí Arbós Linio

**Fecha:** 7 de Julio de 2022

## Resumen

USO E INTEGRACIÓN DE SOFTWARE EN CI/CD PARA REDUCIR VULNERABILIDADES Y REALIZAR CARGA DE TRABAJO/ESTRÉS

Este trabajo final de grado trata sobre el uso de herramientas para encontrar vulnerabilidades de seguridad y limitaciones de rendimiento para hacer las aplicaciones más robustas y sólidas frente a los ataques.

Además, también explica cómo integrar estas herramientas en entornos de integración continua y entrega continua.

Las herramientas seleccionadas para el trabajo son Zap (vulnerabilidades de seguridad), Gatling (limitaciones de rendimiento) y Jenkins (integración de entorno CI/CD).

Este proyecto consta de una guía de configuración y uso de todas las herramientas, una muestra de resultados y un análisis en profundidad de un entorno simulado.

También se menciona cómo se ha organizado el trabajo y se finaliza con una conclusión crítica sobre las herramientas utilizadas y cómo se ha realizado el proyecto.

Por último, mencionar que este trabajo de final de grado se ha hecho conjuntamente con la empresa Giesecke + Devrient (G+D).

**Title:** SOFTWARE USE & INTEGRATION IN CI/CD TO REDUCE VULNERABILITES AND PERFORM WORK/STRESS LOAD

**Author:** Manuel Fraga Lobato

**Tutor:** Olga León Abarca

**Supervisor:** Noemí Arbós Linio

**Date:** Jule 7<sup>th</sup> 2022

### Overview

SOFTWARE USE & INTEGRATION IN CI/CD TO REDUCE VULNERABILITES AND PERFORM WORK/STRESS LOAD

This final degree project deals with the use of tools in order to find security vulnerabilities and performance limitations in order to make applications more robust and solid against attacks.

In addition, it also explains how to integrate these tools in continuous integration and continuous delivery environments.

The tools selected for the job are Zap (security vulnerabilities), Gatling (performance limitations) and Jenkins (CI/CD environment integration).

This project consists of a guide of the configuration and use of all the tools, a sample of results and an in-depth analysis of a simulated environment.

There is also a mention about how the work has been organized and it ends with a critical conclusion about the tools used and how the project has been done.

Finally, mention that this final degree project has been done jointly with the company Giesecke + Devrient (G+D).

## INDEX

INDEX.....	4
ACRONYMS .....	6
LIST OF FIGURES .....	7
INTRODUCTION.....	9
CHAPTER 1. ZED ATTACK PROXY (ZAP) .....	11
1.1. WHAT IS ZAP.....	11
1.2. USER INTERFACE .....	12
1.2.1. AUTOMATED SCAN .....	13
1.2.2. MANUAL SCAN.....	14
1.2.3. RECOMENDATIONS FOR USEFUL SCANNING .....	15
1.3. HOW TO USE ZAP .....	16
1.3.1.HOW TO SCAN MANUALLY .....	16
1.3.2. SCANNING AN AUTHENTICATED SITE .....	17
1.3.3. PROXY CONFIGURATION .....	21
1.4. DOCKER.....	25
1.4.1. BASELINE SCAN .....	25
1.4.2. FULL SCAN .....	26
1.4.3. API SCAN: .....	26
1.4.4. USAGE.....	26
CHAPTER 2. GATLING.....	28
2.1. WHAT IS GATLING .....	28
2.2. WHAT IS SCALA.....	29
2.2.1. TYPE INFERENCE .....	29
2.2.2. CONCURRENCE AND DISTRIBUTION .....	29
2.2.3. INTERFACES .....	30
2.2.4. HIGHER ORDER FUNCTIONSS.....	30
2.3. HOW TO INTEGRATE WITH OUR COMPONENTS.....	30
2.3.1 GATLING RECORDER .....	30
2.3.2 INTELLIJ PROJECT.....	32
CHAPTER 3. JENKINS .....	39
3.1. WHAT IS JENKINS .....	39
3.2. INSTALLATION .....	40
3.3. HOW TO USE JENKINS.....	43

---

3.3.1. HOW TO CREATE A JOB .....	44
3.3.2. HOW TO MAKE A JENKINSFILE .....	45
CHAPTER 4. ANALYSIS .....	49
4.1. Pen-testing with ZAP .....	49
4.2. Work/Stress Load with GATLING .....	51
4.3. CI/CD environment .....	57
4.4. DEMONSTRATION .....	59
4.4.1. API REST 1 +1350 req. ....	64
4.4.2. API REST 1 +3000 req. ....	66
4.4.3. BOTH APIs WITH LOW AMOUNT OF WORK .....	67
4.4.4. BOTH APIs WITH BIG AMOUNT OF WORK .....	69
CHAPTER 5. CONCLUSIONS & THOUGHTS.....	71
BIBLIOGRAPHY .....	74

## ACRONYMS

- *ZAP = Zed Attack Proxy*
- *CI/CD = Continuous Integration / Continuous Delivery*
- *CLI = Command Line Interface*
- *ScaLa = Scalable Language*
- *PO = Product Owner*
- *SM = Scrum Manager*
- *HUD = Head-Up Display*
- *AJAX = Asynchronous JavaScript And XML*
- *VPN = Virtual Private Network*
- *JVM = Java Virtual Machine*
- *API = Application Programming Interface*
- *http = HyperText Transfer Protocol*
- *RTT = Round Trip Time*
- *URL = Uniform Resource Location*
- *DevOps = Development Operations*
- *OS = Operating System*
- *PDB = Pluggable DataBase*

## LIST OF FIGURES

- *Fig 1.1.1 ZAP as a “man-in-the-middle proxy” (pag. 11)*
- *Fig 1.1.2 ZAP as a “man-in-the-middle proxy” with a network proxy (pag. 11)*
- *Fig 1.2.1. Basic view of ZAP software (pag. 12)*
- *Fig 1.2.1.1. Automated Scan screen (pag. 13)*
- *Fig 1.2.2.1. Manual Scan screen (pag. 15)*
- *Fig 1.3.1.1. Head-Up Display (HUD) in the screen (pag. 16)*
- *Fig 1.3.2.1. Session Properties icon (pag. 17)*
- *Fig 1.3.2.2. Authentication configuration (pag. 17)*
- *Fig 1.3.2.3. Creating a new User in a specific context (pag. 18)*
- *Fig 1.3.2.4. Selecting the User to use (pag. 18)*
- *Fig 1.3.2.5. Specifying the context from an HTTP request executed manually (pag. 19)*
- *Fig 1.3.2.6. Several manners of pen testing (pag. 19)*
- *Fig 1.3.2.7. Active scanning using a specific user from “HUD context” (pag. 20)*
- *Fig 1.3.3.1. Options tab (pag. 21)*
- *Fig 1.3.3.2. Proxy configuration for ZAP (pag. 22)*
- *Fig 1.3.3.3. Proxy configuration for browser (pag. 22)*
- *Fig 1.3.3.4. Using ZAP as a proxy (pag. 23)*
- *Fig 1.3.3.5. Generating a Dynamic SSL certificate for the browser (pag. 23)*
- *Fig 1.3.3.6. Importing ZAP Dynamic SSL certificate (pag. 24)*
- *Fig 1.4.4.1. Different options for CLI scanning (pag. 38)*
- *Fig 2.3.1.1. Gatling Recorder UI (pag. 31)*
- *Fig 2.3.1.2. Configuring browser as a Proxy (pag. 31)*
- *Fig 2.3.2.1. Creating a Gatling project (pag. 32)*
- *Fig 2.3.2.2. Project structure (pag. 33)*
- *Fig 2.3.2.3. HTTP configuration (pag. 34)*
- *Fig 2.3.2.4. Login Test example (pag. 35)*
- *Fig 2.3.2.5. Example simulation (pag. 36)*
- *Fig 2.3.2.6. HashMap to store Token’s (pag. 37)*
- *Fig 2.3.2.7. Establishing different sessions (pag. 37)*
- *Fig 2.3.2.8. Defining a Scenario (pag.38)*
- *Fig 3.2.1. Unlocking Jenkins for installation (pag. 40)*
- *Fig 3.2.2. Installing plugins (pag. 41)*
- *Fig 3.2.3. Create first Admin User (pag. 41)*
- *Fig 3.2.4. Set port for Jenkins (pag. 42)*
- *Fig 3.3.1. Jenkins’s pipeline (pag. 43)*
- *Fig 3.3.1.1. Create a new Job (pag. 44)*
- *Fig 3.3.1.2. Full project name Path (pag. 44)*
- *Fig 3.3.2.1. Declarative pipeline example (pag. 45)*
- *Fig. 3.3.2.2. Environment variables in Jenkinsfile (pag. 46)*

- *Fig 3.3.2.3. Add parameters in Job (pag. 47)*
- *Fig 3.3.2.4. Specify git repository (pag. 47)*
- *Fig 3.3.2.5. Trigger another job from Jenkinsfile (pag. 48)*
- *Fig 4.1.1. Summary of alerts (pag. 49)*
- *Fig 4.1.2. More detailed information of alerts (pag. 49)*
- *Fig 4.1.3. Explanation of an alert (pag. 50)*
- *Fig 4.1.4. Solutions in “designing” phase (pag. 50)*
- *Fig 4.1.5. Solutions in “implementation” phase (pag. 50)*
- *Fig 4.2.1. Initial graphics with summary information (pag. 51)*
- *Fig 4.2.2. Table with information on response times (pag. 52)*
- *Fig 4.2.3. Active Users along the simulation (pag. 52)*
- *Fig 4.2.4. Response Time Distribution (pag. 53)*
- *Fig 4.2.5. Number of requests & responses per second (pag. 53)*
- *Fig 4.2.6. Analysis of a particular request (pag. 54)*
- *Fig 4.2.7. Number of requests & responses per second of a particular request (pag. 54)*
- *Fig 4.2.8. 101 Requests test statistics (pag. 55)*
- *Fig 4.2.9. Requests & responses per second during the test (pag. 56)*
- *Fig 4.3.1. Options to use ZAP & Gatling functionalities (pag. 57)*
- *Fig 4.3.2. Stages of the job (pag. 57)*
- *Fig 4.3.3. Time it takes for the job to finish (pag. 58)*
- *Fig. 4.3.4. Window for downloading the report (pag. 58)*
- *Fig. 4.4.1. Environment (pag. 59)*
- *Fig. 4.4.2. Summary of alerts (pag. 59)*
- *Fig. 4.4.3. Alerts (pag. 60)*
- *Fig. 4.4.4. Cross-Domain Misconfiguration description (pag. 61)*
- *Fig. 4.4.5. CSP: style-src unsafe-inline (pag. 61)*
- *Fig. 4.4.6. CSP: Wildcard Directive (pag. 61)*
- *Fig. 4.4.7. Vulnerable JS Library (pag. 62)*
- *Fig. 4.4.8. X-Frame-Options Header Not Set (pag. 62)*
- *Fig. 4.4.9. Incomplete or No Cache-control and Pragma HTTP Header Set (pag. 63)*
- *Fig. 4.4.10. X-Content-Type-Options Header Missing (pag. 63)*
- *Fig. 4.4.1.1. Initial graphics of test 1(pag. 64)*
- *Fig. 4.4.1.2. Response Time Percentiles over Time (OK) test 1(pag. 64)*
- *Fig. 4.4.1.3. Number of requests/responses per second test 1(pag. 65)*
- *Fig. 4.4.2.1. Initial graphics of test 2 (pag. 66)*
- *Fig. 4.4.2.2. Response Time Percentiles over Time (OK) test 2 (pag. 66)*
- *Fig. 4.4.3.1. Initial graphics of test 3 (pag. 67)*
- *Fig. 4.4.3.2. Get Cat Fact Requests (pag. 67)*
- *Fig. 4.4.3.3. Response Time Distribution test 3 (pag. 68)*
- *Fig. 4.4.4.1. Initial graphics of test 4 (pag. 69)*
- *Fig. 4.4.4.2. Get Cat Fact Requests test 4 (pag. 69)*
- *Fig. 4.4.4.3. Number of requests/responses per second test 4 (pag. 70)*



## INTRODUCTION

As is well known, technology has been evolving for a long time and never ceases to amaze with the speed at which it breaks barriers, leaving more than one incredulous.

This work arises in part from the need that developers have to adapt to these changes, since with all the changes there are, to stagnate means to die.

This is an overwhelming truth when it comes to programming because this world changes "overnight", and you have to be very attentive to the tools you use to develop your code, since what was safe yesterday it might not be today. The idea of knowing about security vulnerabilities is not just for hackers, as a developer who is not agnostic to the issues might be able to avoid them.

This is one of the many reasons why learning about vulnerabilities is never a bad option. There are a large number of ways to attack clients and servers and it is practically never possible to be certain that a web page is totally secure. Even so, this does not justify that a web page uses outdated libraries with security holes or that it has a misconfiguration of mechanisms in HTTP headers that enables a domain for clickjacking or cross-site scripting attacks.

In this work we will be able to observe the different types of vulnerabilities that have been found through vulnerability scans and how to solve them.

The vast majority of these vulnerabilities do not require a great cost or great measures to be eradicated. If we use one of different software developed for seeking vulnerabilities we could find that this is due to configuration problems (such as the case of the CSP header, X-Frame-Options, CORS among others). In fact, a high percentage of security breaches could be avoided if behind of it there was a good use of libraries and headers together with basic notions of cybersecurity. Another compelling reason to learn about this is the great benefit to be gained from so little knowledge required.

On the other hand we have the performance analysis of applications and environments. It is very valuable information to know with certainty how an application behaves under a certain workload, so there are many tools created because of this need. In this document we can find different tests carried out under different circumstances and workloads to analyze how a specific server behaves in order to find bottlenecks and limitations.

Another point to note is the fact of integrating processes in automation software.

As you can imagine, to get a stable and secure application with few limitations, maintenance is needed and not a just a punctual task. This can become tedious, since the same thing must always be analyzed to ensure that there are no security gaps, performance drops or strange behavior.

This is why automation software is used so much, so all these processes can be carried out autonomously without the user having to worry about carrying out the tests, they should only be in charge of analyzing the reports.

In this final degree project, we can see how different tools have been used to achieve everything mentioned above. These tools are Zap for everything related to vulnerabilities and security, Gatling for performance and behavior issues, and Jenkins for process automation.

The project follows a very clear structure, beginning with three chapters that go into detail about the tools to be used (for vulnerabilities, for load tests and for integration in CI/CD). It continues with a chapter focused on a demonstration with results and their respective analysis and ends with a chapter focused on conclusions, criticisms and personal opinions where also is explained how this project was done inside a company.

## CHAPTER 1. ZED ATTACK PROXY (ZAP)

### 1.1. WHAT IS ZAP

Zed Attack Proxy (ZAP)[1] is a free and open-source penetration testing tool designed for testing web applications that is flexible and extensible.

At the heart of ZAP there are the so-called "man-in-the-middle agents". They sit between the tester's browser and the web application, so it can intercept and inspect messages sent between the browser and the web application, modify the content if necessary, and then forward those packets to their destination.

It can be used as a standalone application or as a daemon.



**Fig. 1.1.1** ZAP as a “man-in-the-middle proxy”

If there is another network proxy already in use, as in many corporate environments, ZAP can be configured to connect to that proxy.



**Fig. 1.1.2.** ZAP as a “man-in-the-middle proxy” with a network proxy

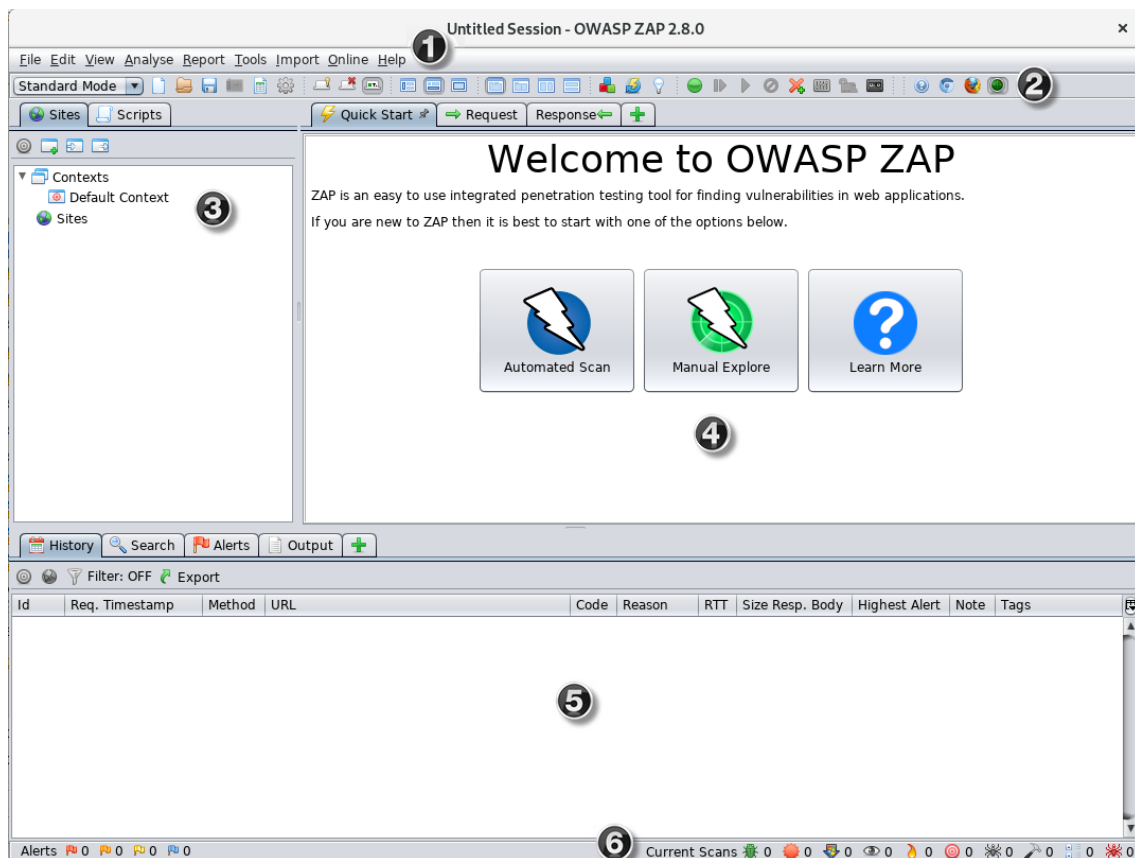
This software is responsible for making an exhaustive scan of a web domain to find possible vulnerabilities or bad code practices, which lead to security errors.

The company that developed the tool also has a page that talks about these vulnerabilities, how they usually occur and how they can be resolved. This page is updated every year and makes a ranking of the 10 most common vulnerabilities of the year in question.

Some of the most common vulnerabilities in direct data injection, outdated and/or vulnerable components, security configuration errors, access control problems and cryptographic flaws among other common errors.

## 1.2. USER INTERFACE

When opening the program, the first thing we see is a screen like the following one:



**Fig. 1.2.1** Basic view of ZAP software

In the previous image we can see the following fields:

*1. Menu Bar*

Provides access to many of the automated and manual tools.

*2. Toolbar*

Includes buttons which provide easy access to most commonly used features.

*3. Tree Window*

Displays the Sites tree and the Scripts tree.

*4. Workspace Window*

Displays requests, responses, and scripts and allows us to edit them.

*5. Information Window*

Displays details of the automated and manual tools.

*6. Footer*

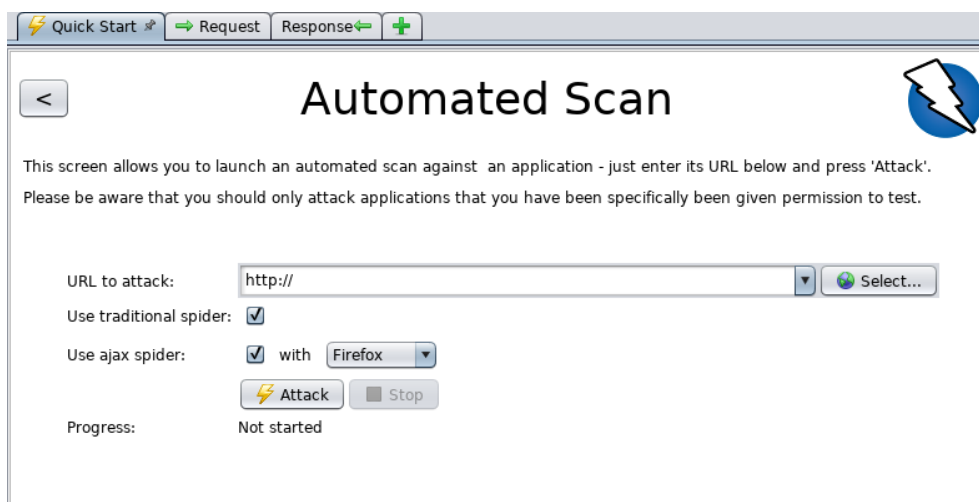
Displays a summary of the alerts found and the status of the main automated tools.

There are two ways to do an exploration of an application or URL, either the passive scanning (automated scan) or the active scanning (manual scan).

### 1.2.1. AUTOMATED SCAN

ZAP uses its spider to crawl web applications and passively scans every page it finds. A spider is Zap's own tool that is responsible for going thread by thread until all the endpoints of a web page can be seen. That's where the name comes from, since it is as if a spider weaves a web thread by thread with the endpoints of a target.

ZAP then uses an active scanner to attack all discovered pages, functions, and parameters.



**Fig. 1.2.1.1.** Automated Scan screen

ZAP provides 2 spiders for scraping web applications, you can use one or both in this screen. Traditional ZAP spiders discover links by examining HTML in web application responses.

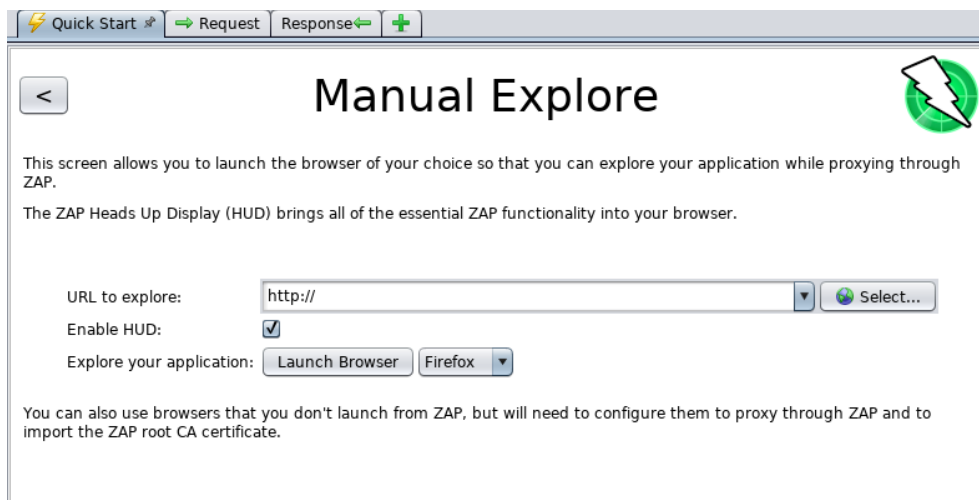
This crawler is fast but not always effective when exploring AJAX web applications that use JavaScript to generate links. For AJAX applications, ZAP's AJAX spider may be more efficient. The spider inspects the web application by invoking the browser, which then follows the generated link.

AJAX spiders are slower than traditional spiders and require additional configuration to be used in a "headless" environment.

To run a Quick Start Automated Scan:

1. Start ZAP and click the Quick Start tab of the Workspace Window.
2. Click the large Automated Scan button.
3. In the URL to attack text box, enter the full URL of the web application you want to attack.
4. Click the Attack

### 1.2.2. MANUAL SCAN



**Fig. 1.2.2.1.** Manual Scan screen

The passive scanning and automated attack functionality are a great way to begin a vulnerability assessment of our web application, but it has some limitations. Among these are:

- Any pages protected by a login page are not discoverable during a passive scan because, unless you've configured ZAP's authentication functionality, ZAP will not handle the required authentication.
- You don't have a lot of control over the sequence of exploration in a passive scan or the types of attacks carried out in an automated attack. ZAP does provide many additional options for exploration and attacks outside of passive scanning.

To manually explore an application, we just need the following steps:

1. Start ZAP and click the Quick Start tab of the Workspace Window.
2. Click the large Manual Explore button.
3. In the URL to explore text box, enter the full URL of the web application we want to explore.
4. Select the browser we would like to use
5. Click the Launch Browser

### *1.2.3. RECOMENDATIONS FOR USEFUL SCANNING*

Spiders are a great way to explore our simple website, but they should be combined with manual exploration to be more effective. For example, spiders only enter basic standard data in our web application's forms, but users can enter more relevant information, which in turn can make more web applications available to ZAP.

This is especially true for things like signup forms that require a valid email address. The spider may enter a random string, which will cause an error. The user can respond to this error and provide a well-formed string, which will cause more applications to be displayed when submitting and accepting the form.

The entire web application should be analyzed by using a browser proxy via ZAP. When we do this, ZAP passively scans all requests and responses made during a vulnerability scan, continues building the site tree, and logs alerts for potential vulnerabilities discovered during the scan.

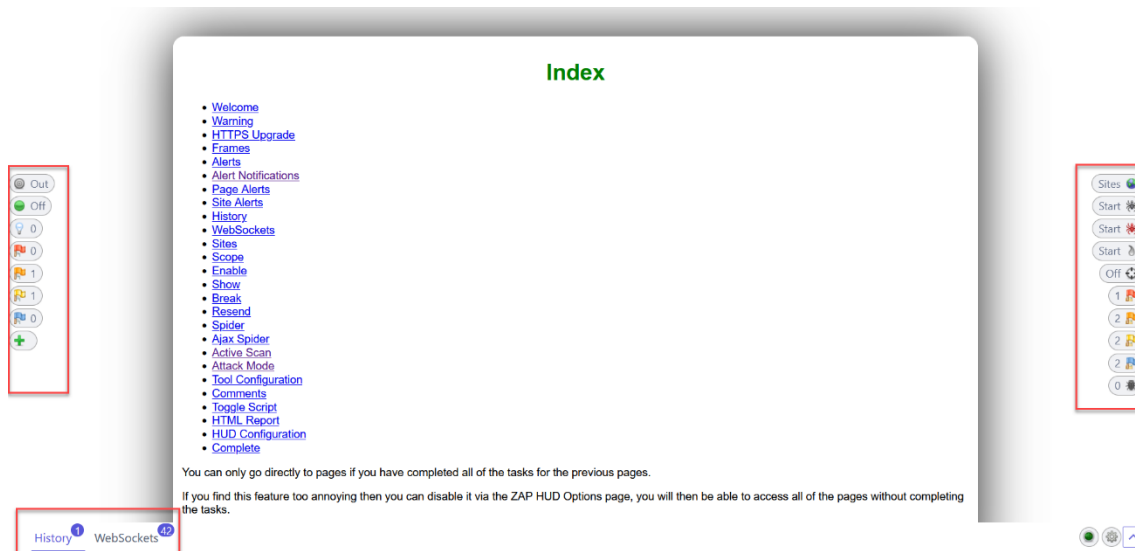
It is important to have ZAP explore each page of our web application, whether linked to another page or not, for vulnerabilities. Obscurity is not security, and hidden pages sometimes go live without warning or notice. So be as thorough as we can when exploring our site.

We can quickly and easily launch browsers that are pre-configured to proxy through ZAP via the Quick Start tab. Browsers launched in this way will also ignore any certificate validation warnings that would otherwise be reported.

## 1.3. HOW TO USE ZAP

### 1.3.1. HOW TO SCAN MANUALLY

When we are going to perform a manual scan to an URL, a Head-Up Display (HUD) will be shown with some icons at both sides of the screen like in the following image.



**Fig. 1.3.1.1.** Head-Up Display (HUD) in the screen

These page icons provide us with information about page or website alerts or allow us to take various actions.

If this is our first manual scan, it is highly recommended to follow the short tutorial provided by ZAP. It takes no more than 10 minutes and is very useful.

At the bottom of the page are the History and WebSocket sections. The History tab shows all requests made by the browser since we opened this page. These can be requests for resources like images or JavaScript files, or API requests.

The WebSocket's tab displays all WebSocket's requests made by the browser. The HUD allows us to get a lot of real-time information as we make various requests on the page being scanned.

The context is a group of configurations, conditions and parameters that we can configure before scanning an URL. There is a Default Context with the common configuration, but we can edit it or create a new one specifying some rules or conditions.



### 1.3.2. SCANNING AN AUTHENTICATED SITE

There could be different occasions that the site requires of some kind of authentication in order to access to the pages that are behind that “firewall”.

This problem can be easily solved by manual scanning that site, inserting the credentials, and once we have surpassed that blocker, we start scanning the site. But if we want to use ZAP scanning from the CLI or we directly do not want to manual scan before, we can solve it by setting the authentication configuration in the Context we are going to use for the scanning.

To do so, we must click on Session Properties.

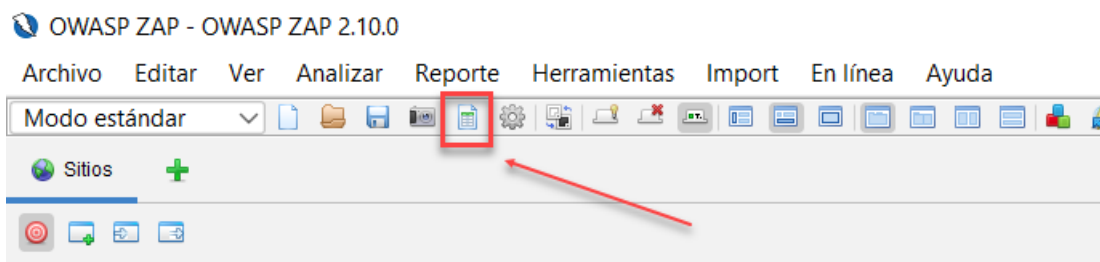


Fig. 1.3.2.1. Session Properties icon

Then, we should go the Authentication section and select the authentication method that is going to be used, the URL where is done that authentication, the URL where we are getting the LOGIN URL from and the credentials.

In the following image, the credentials are input because there is a user already created.

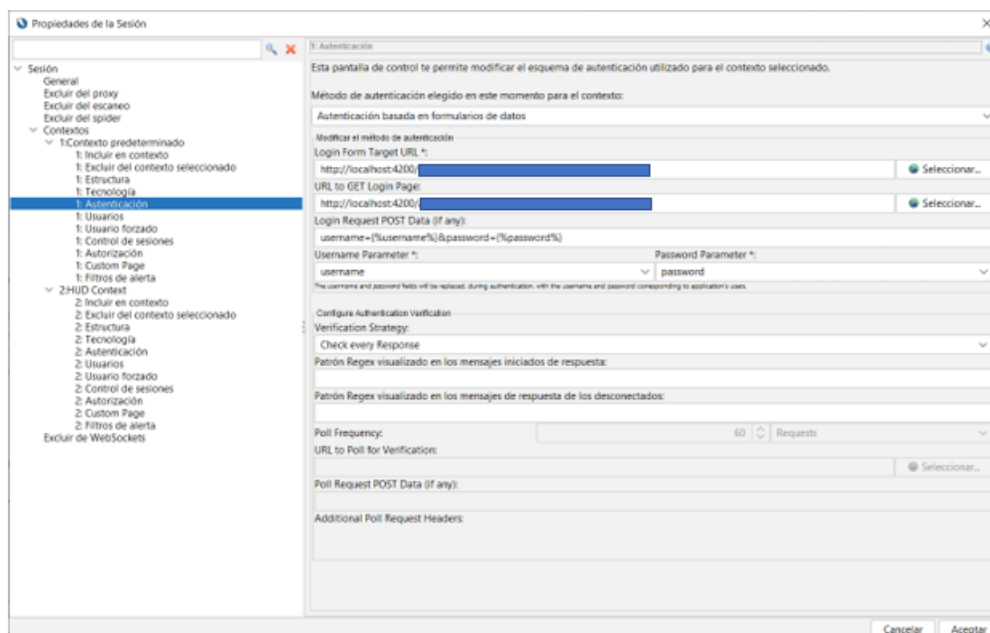
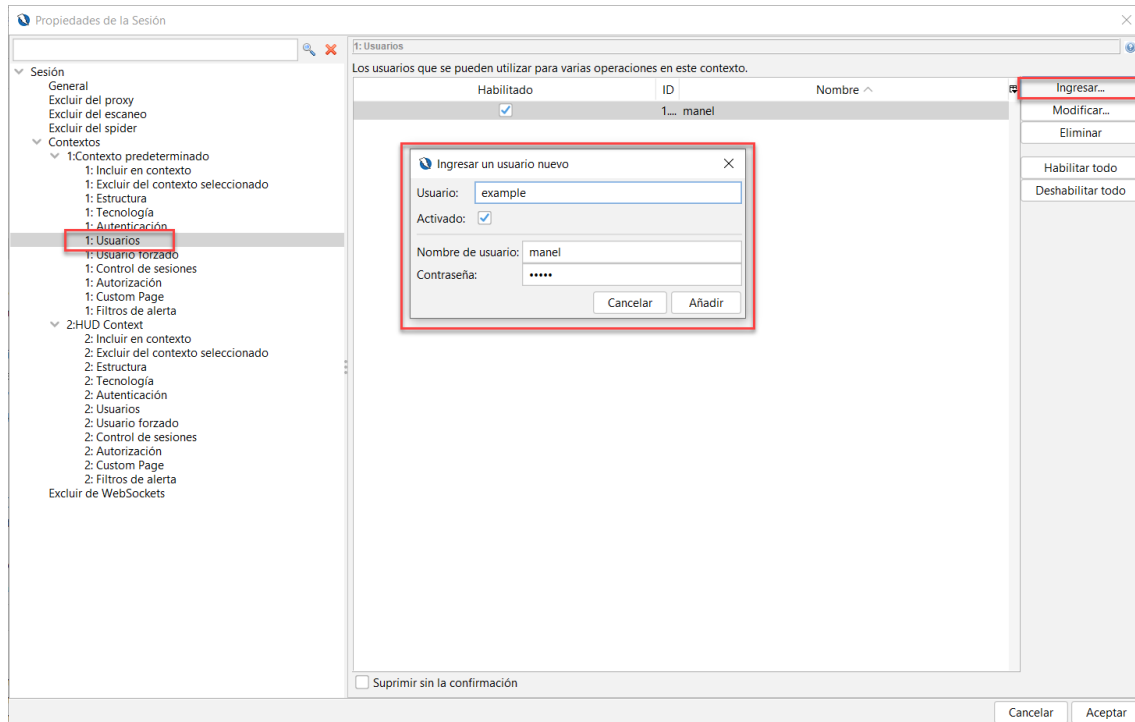


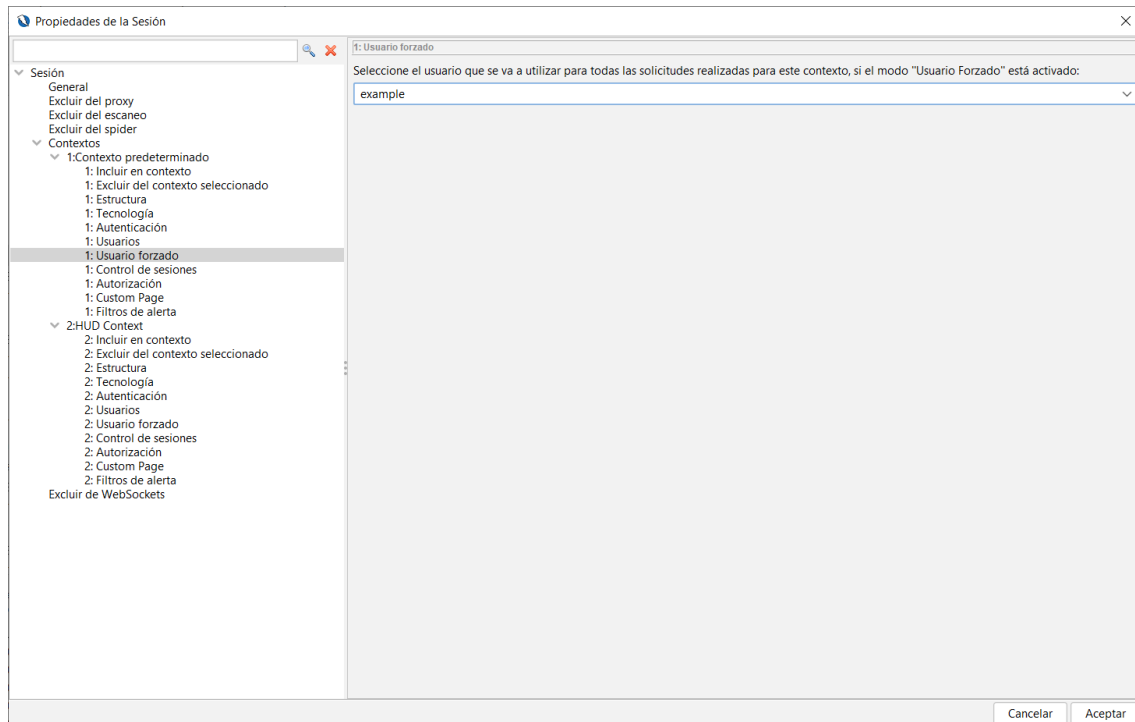
Fig. 1.3.2.2. Authentication configuration

Once we have set the authentication configuration up, we must create a User with the username and password we want to use as credentials, and when once we have created it, we must enable it.



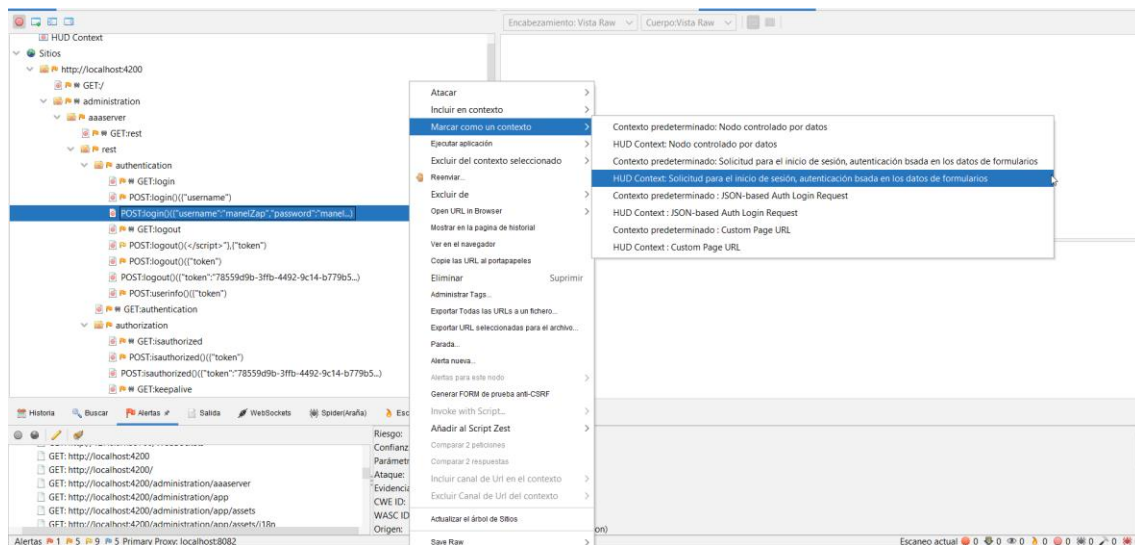
**Fig. 1.3.2.3.** Creating a new User in a specific context

Lastly, we must go to Force User and select the User we want to use.



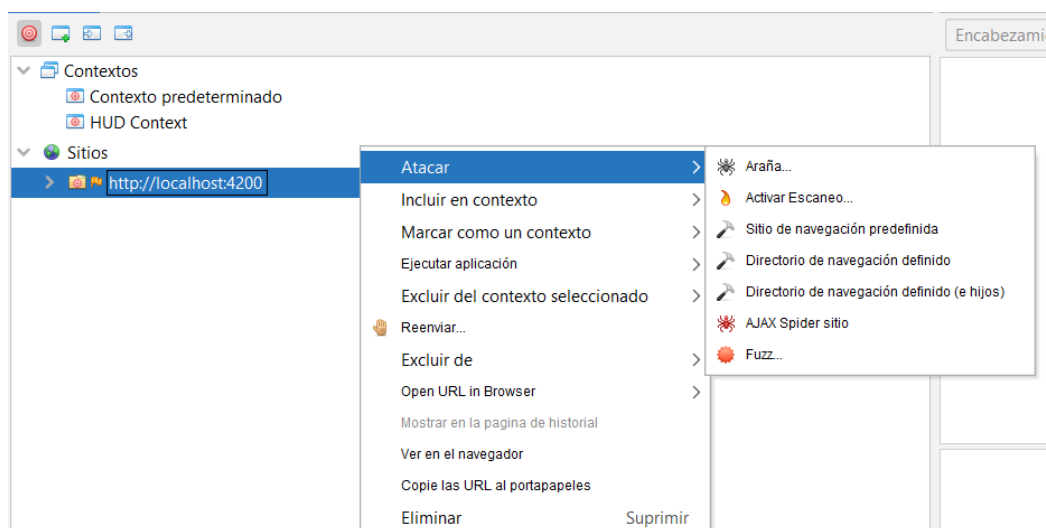
**Fig. 1.3.2.4.** Selecting the User to use

We can also do the Login with the manual scan, and then look for that Login request and set it as a context to do the authentication configuration directly.



**Fig. 1.3.2.5.** Specifying the context from an HTTP request executed manually

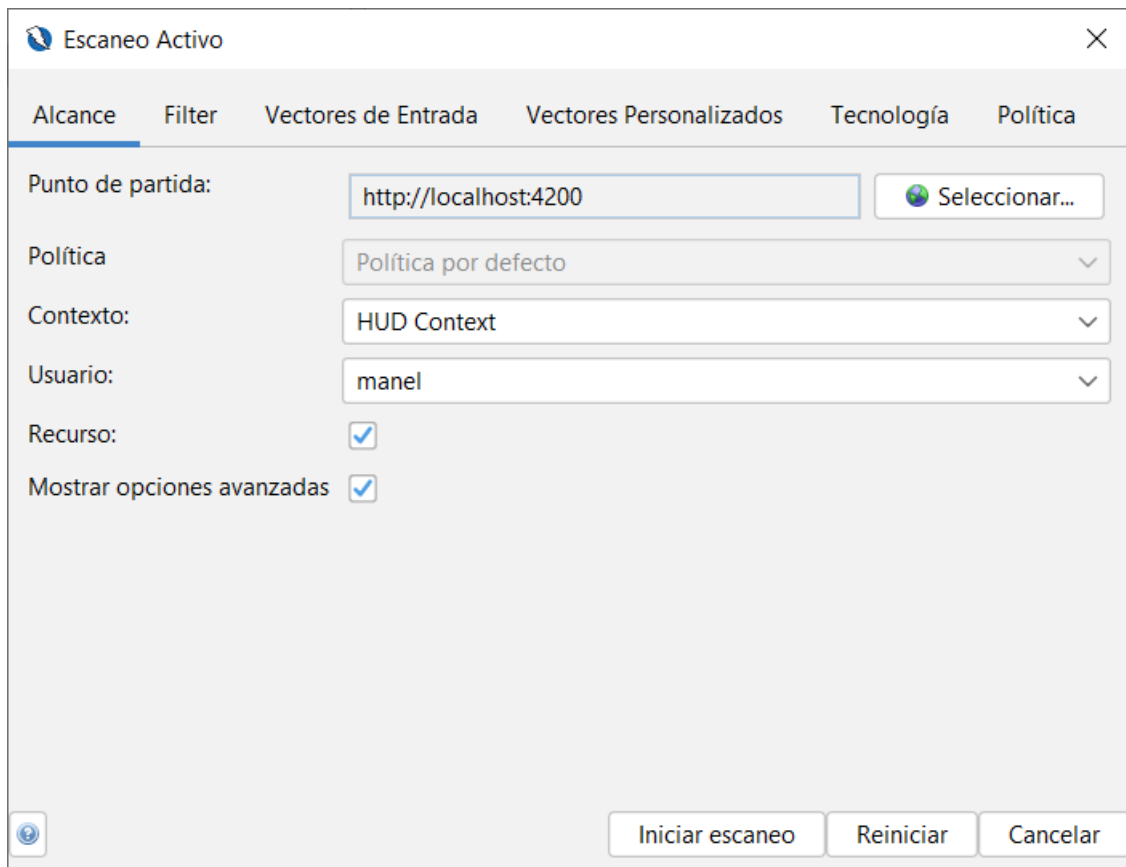
Here, the request with username = “manelZap” and its password is selected as the credentials for the form-based login of the HUD context.



**Fig. 1.3.2.6.** Several manners of pen testing

Then, we can attack that URL with the method that we want, and with the context we want to use. Once we pick on any attack method, a window will pop-up and there we can select the context.

For example, if we want to Active Scan with the User “manel” of the context “HUD Context”



**Fig. 1.3.2.7.** Active scanning using a specific user from “HUD context”

Also, we can configure that Scan with some advanced options if we want it to perform in a specific way.

### 1.3.3. PROXY CONFIGURATION

If we are behind a VPN, we must either configure some fields or disable our VPN connection in order to be able to use ZAP. To do so, we must go to:

Tools -> Settings -> Connection

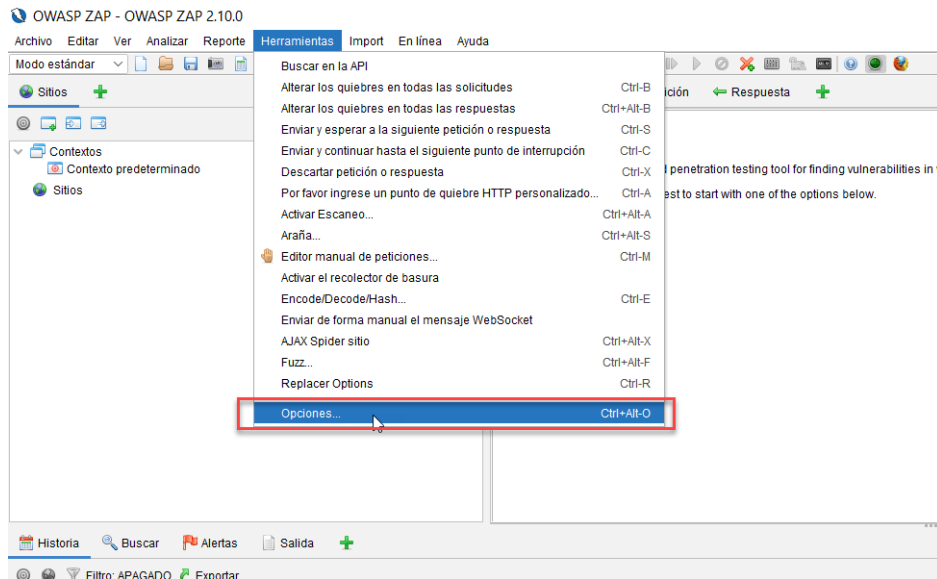


Fig. 1.3.3.1. Options tab

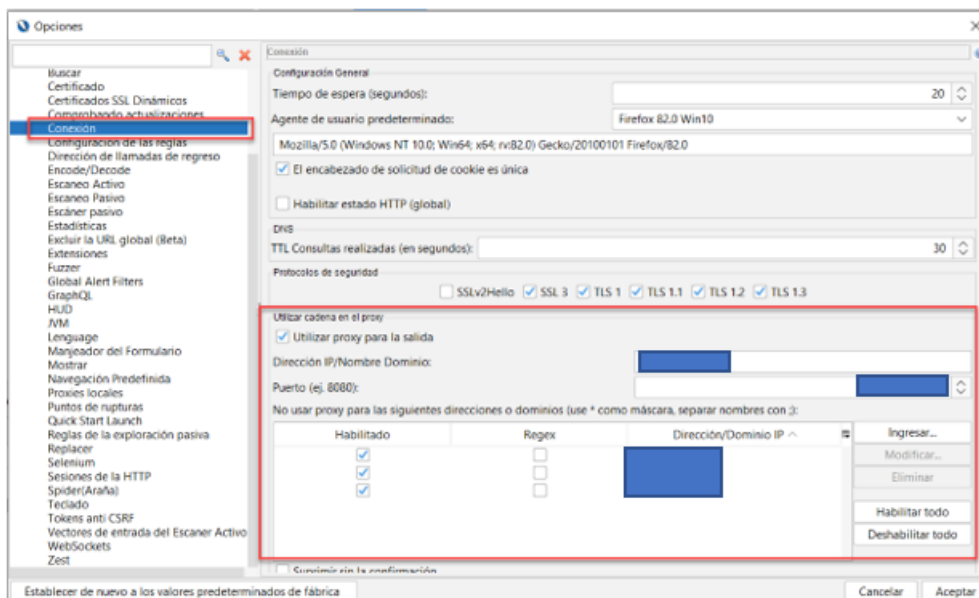


Fig. 1.3.3.2. Proxy configuration for ZAP

Once we have reached this window, we are now able to set the direction and the port of the VPN we are connected to.

Also, we must configure the navigator we are going to use ZAP with.

To show an example, here we have how its configured "Firefox".

First, we need to open Firefox and then go to: Settings -> General -> Proxy Configuration

The image shows the 'Configuración de conexión' (Connection Configuration) dialog box in Firefox. The 'Configurar acceso proxy a Internet' (Configure proxy access to Internet) section is active, with 'Configuración manual del proxy' (Manual proxy configuration) selected. The 'Proxy HTTP' field is filled with a blue box, and the 'Puerto' (Port) field is set to 3128. The checkbox 'Usar también este proxy para HTTPS' (Use this proxy for HTTPS) is checked. The 'Host SOCKS' field is also filled with a blue box, and the 'Puerto' field is set to 3128. The 'SOCKS v5' option is selected. The 'URL de configuración automática del proxy' (Automatic proxy configuration URL) field is empty, with a 'Recargar' (Refresh) button next to it. The 'No usar proxy para' (Do not use proxy for) field is filled with a blue box. At the bottom, there are buttons for 'Aceptar' (Accept), 'Cancelar' (Cancel), and 'Ayuda' (Help). A note at the bottom states: 'Ejemplo: mozilla.org, .net.nz, 192.168.1.0/24' and 'Las conexiones a localhost: 127.0.0.1 v...1 nunca pasan por proxy'.

**Fig. 1.3.3.3.** Proxy configuration for browser

Once we have completed all these steps and have filled in the necessary fields, ZAP is able to inspect different domains as if it were not behind any proxy. Even so, in case the use of a proxy is not strictly necessary, the quickest solution for this situation is to "switch off" the proxy. In this way, no proxy configuration would be needed.

On the other hand, if what we want is to use ZAP as a Local Proxy, then we must go to “Local Proxy” window instead of the “Connection” window and specify which direction and which port we want to use (localhost:8080 by default).

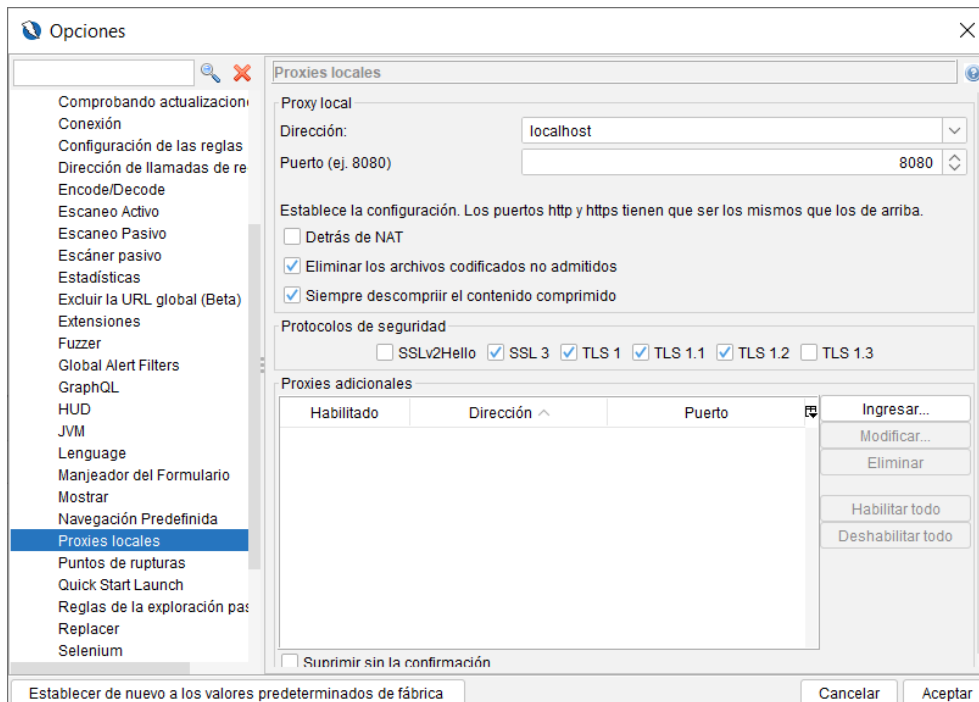


Fig. 1.3.3.4. Using ZAP as a proxy

When the local proxy has been set up, a certificate for the navigator is required. To generate a new certificate, we must go “Dynamic SSL Certificates”, generate a new one and save it.

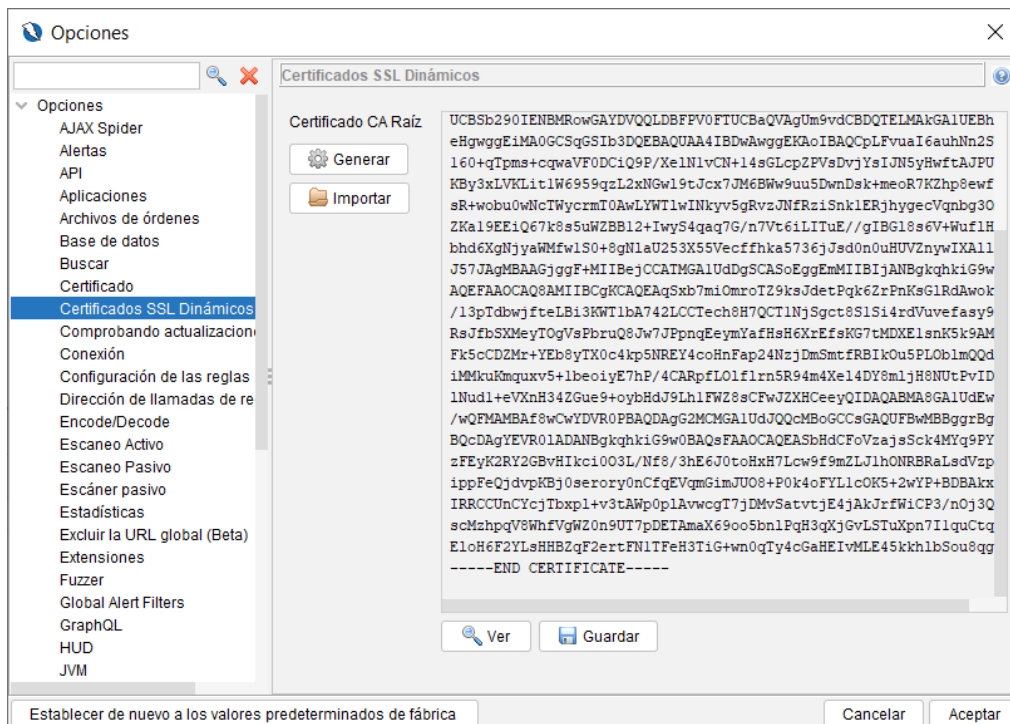
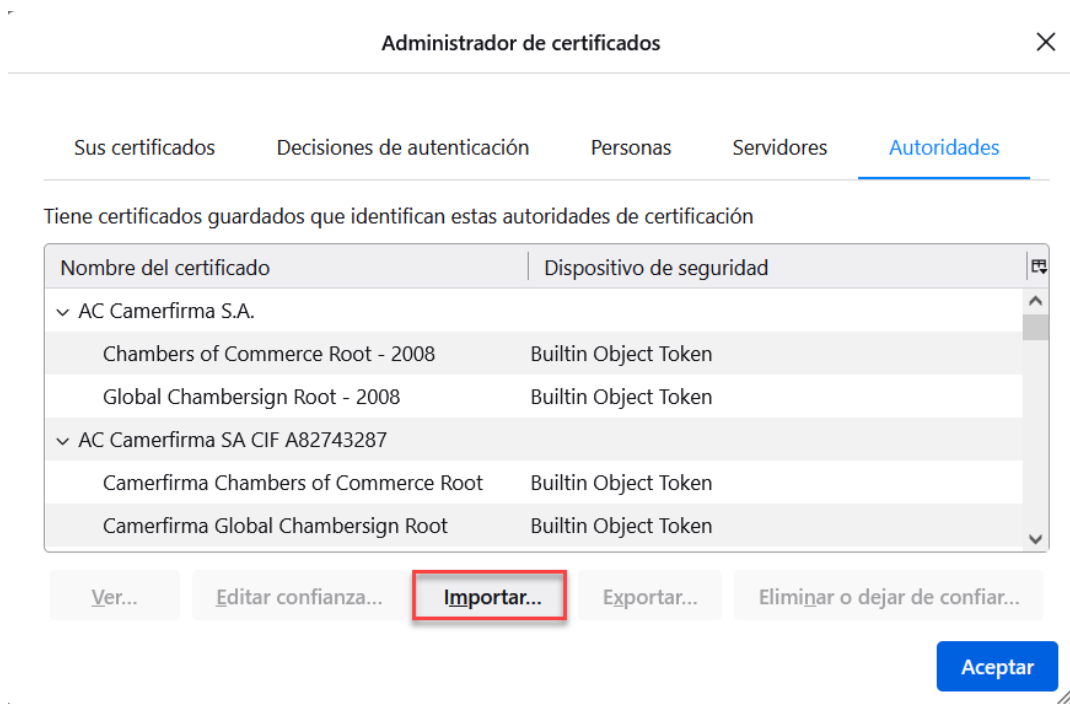


Fig. 1.3.3.5. Generating Dynamic SSL certificate for browser

To use this functionality, we should be out of the VPN, and then go to the navigator we are going to use for that and import the certificate that has been previously generated.

In Firefox, we should go to: Settings -> Privacy & Security -> Certificates -> Show Certificates.



**Fig. 1.3.3.6.** Importing ZAP Dynamic SSL certificate

We must click on "Import" and select the certificate we had previously generated.



## 1.4. DOCKER

There is also the option to use zap from the command line on our computer by using docker containers with the ZAP image.

There are four different ZAP[2] docker images available in order to run OWASP ZAP as a docker container.

To download these four images, we must do the following docker

1. Stable release

---

```
docker pull owasp/zap2docker-stable
```

---

2. Latest weekly release

---

```
docker pull owasp/zap2docker-weekly
```

---

3. Live release

---

```
docker pull owasp/zap2docker-live
```

---

4. Bare release (small docker image, ideal for CI environments)

---

```
docker pull owasp/zap2docker-bare
```

---

All the docker images provide a set of packaged scan scripts:

### 1.4.1. BASELINE SCAN

It runs the ZAP spider (by default) against the specified target for 1 minute, then waits for the passive scan to complete before reporting the results. This means that the script doesn't actually "attack" and only runs for a relatively short period of time (a few minutes at most).

This script is great for running in a CI/CD environment, even for production sites.

### 1.4.2. FULL SCAN

It runs a ZAP spider against an objective (no time limit by default), and we can also specify the use of an optional Ajax spider scan, followed by a full active scan, before reporting the results. This means that the script is performing a pen-testing "attack" and may last more time than a baseline scan. By default, all warnings are reported as Warnings, but we can specify a configuration file that can change all rules to FAIL or IGNORE.

The configuration works in a very similar way as the Baseline Scan.

### 1.4.3. API SCAN

It is tuned to perform scans against OpenAPI, SOAP, or GraphQL-defined APIs via local files or URLs.

It imports the definitions we provide and then runs an active scan on the URLs it finds. Active scanning is tuned for the API, so it doesn't bother looking for things like XSS. It also includes 2 scripts: Trigger alerts for all HTTP server error response codes Alert on all URLs that return content types not normally associated with an API

### 1.4.4. USAGE

Depending on which kind of objective we want to analyze, we should use one docker image or another.

As an example, if we want to perform a basic scan to the specific URL "www.example.com" we should use the following command in the CLI.

---

```
docker run -t owasp/zap2docker-stable zap-baseline.py -t https://www.example.com
```

---

We can set more options and conditions to this command in order to obtain some reports or to specify a maximum amount of time to the scanning, also to show debugging messages or to use the AJAX spider instead of the traditional one.

As a more complex example, imagine now we must scan the same URL as before, but specifying a maximum scanning time of 10 minutes, using AJAX spider instead of traditional spider, we also need to see the debugging messages and this domain has an authenticated site, so we need to use a context named "contextUPC" we have previously created with the user "exampleUPC". And we want it to generate an HTML report.

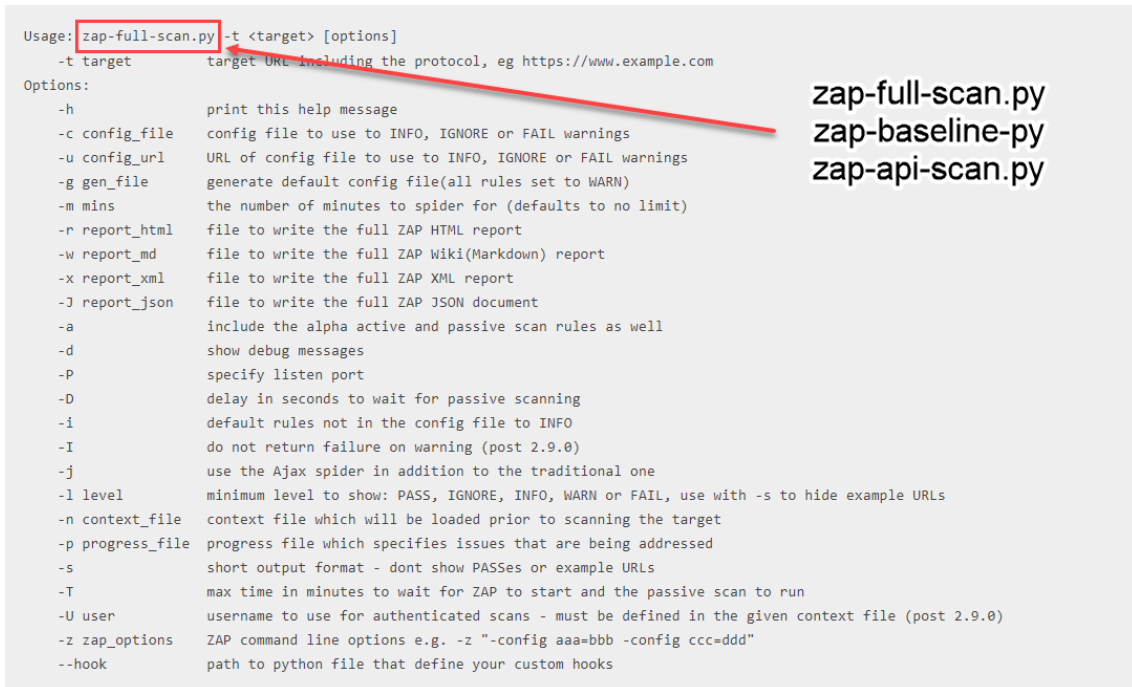
The command we should use would be like this:

---

```
docker run -t owasp/zap2docker-stable zap-baseline.py -t https://www.example.com -m 10 -j -d -n contextUPC -U exampleUPC -r report
```

---

All the possible appends are shown in the next image:



```
Usage: zap-full-scan.py -t <target> [options]
-t target      target URL including the protocol, eg https://www.example.com
Options:
-h            print this help message
-c config_file config file to use to INFO, IGNORE or FAIL warnings
-u config_url  URL of config file to use to INFO, IGNORE or FAIL warnings
-g gen_file    generate default config file(all rules set to WARN)
-m mins       the number of minutes to spider for (defaults to no limit)
-r report_html file to write the full ZAP HTML report
-w report_md   file to write the full ZAP Wiki(Markdown) report
-x report_xml  file to write the full ZAP XML report
-J report_json file to write the full ZAP JSON document
-a            include the alpha active and passive scan rules as well
-d            show debug messages
-P            specify listen port
-D            delay in seconds to wait for passive scanning
-i            default rules not in the config file to INFO
-I            do not return failure on warning (post 2.9.0)
-j            use the Ajax spider in addition to the traditional one
-l level      minimum level to show: PASS, IGNORE, INFO, WARN or FAIL, use with -s to hide example URLs
-n context_file context file which will be loaded prior to scanning the target
-p progress_file progress file which specifies issues that are being addressed
-s            short output format - dont show PASSes or example URLs
-T            max time in minutes to wait for ZAP to start and the passive scan to run
-U user       username to use for authenticated scans - must be defined in the given context file (post 2.9.0)
-z zap_options ZAP command line options e.g. -z "-config aaa=bbb -config ccc=ddd"
--hook       path to python file that define your custom hooks
```

zap-full-scan.py  
zap-baseline-py  
zap-api-scan.py

**Fig. 1.4.4.1.** Different options for CLI scanning

As a personal suggestion, if there is a need of scanning complex objectives as some URL with authentication or if we want to generate reports of different scans, its strongly recommend creating a docker container and set the context files we may need in there.

We can do that this way:

1. *docker pull owasp/zap2docker-weekly*
2. *docker create --name=xxxx --network="host" -t owasp/zap2docker-weekly*
3. *docker start xxxx*
4. *docker exec xxxx mkdir /zap/wrk*
5. *docker cp example.context owasp:/zap/wrk/example.context*

Notice that we are building a container named “xxxx” and creating a directory “/wrk” inside a folder “/zap” to be able to copy our “example.context” in there.

## CHAPTER 2. GATLING

### 2.1. WHAT IS GATLING

Gatling[4] is a load testing tool which can be used for our integrated development environment, version control systems and continuous integration solutions. It does not have its own solution, rather it integrates with our existing solutions.

It is built on top of Akka[11], which is a toolkit for building distributed message driven applications. It is a distributed framework which will allow for fully asynchronous computing.

Using this mode Gatling can simulate multiple virtual users with a single thread. Akka overrides the JVM limitation of handling many threads.

Gatling is developed as a combination of Java with ScaLa. It is a relatively new tool, its first stable version originated in early 2012 (January 15), and it is gaining strength in recent years compared to other tools dedicated to load testing and stress testing such as JMeter.

Later, we will go into depth in the explanation of how gatling works, but as a brief introduction it could be said that gatling is made up of three different components.

The first would be the tests that we define, which are the series of requests that are going to be carried out. Second is the protocol configuration, which is almost always the HTTP protocol, where we must specify the headers. And finally, there are the scenarios, which receive as parameters the tests and the configuration of protocols to be used, and in this way simulate a series of virtual users that are going to execute the requests provided.

Simplifying things a lot, the scenarios simulate a series of users who use the tests and the protocols configured to carry out the load and stress tests.

## 2.2. WHAT IS SCALA

Scala[9] is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It seamlessly integrates features of object-oriented and functional languages.

ScaLa is a very powerful language created with the aim of being a scalable language, hence its name (SCA-lable LA-anguage). It is a programming language very similar to Java, both run in JVM, and in terms of managing classes, functions and polymorphism it is very similar to Java, but also to C.

On the other hand, it must be noted that as much as the language is very similar to Java and C, its usage is quite similar to Python due to how surprisingly concise it is. What for Java requires 10 lines of code, with ScaLa[10] we can achieve it with a couple.

This last detail has its good part and its bad part. The good thing is that we get a much shorter code and therefore it is faster to program, the bad part is its learning curve, since the readability of the code is infinitely more expensive than other languages since the ScaLa compiler is very powerful and can do anything from inferring types and classes to performing simple "operations"[5], so since we don't usually need to explicitly specify this in our code, it becomes difficult to read.

Despite this, it can be said that ScaLa tries to group the best of other languages under a single programming language, which is a very attractive idea that is making the demand for developers with knowledge of this language quite popular.

Entering a little more in technicalities we can see the following characteristics.

Despite this, it can be said that ScaLa tries to group the best of other languages under a single programming language, which is a very attractive idea that is making the demand for developers with knowledge of this language quite popular.

Entering a little more in technicalities we can see the following characteristics.

### 2.2.1. TYPE INFERENCE

By having such a competent compiler, it is not necessary to specify the type of the objects that are instantiated in most cases, since the compiler can intuit the type of the objects by itself.

### 2.2.2. CONCURRENCE AND DISTRIBUTION

It allows to distribute the workload for the processors thanks to the branching of processes through threads.

And what is related to concurrency, we can make use of promises or use methods such as `Future{}` that allow us to process data asynchronously, which favors parallelization and distribution.

### 2.2.3. INTERFACES

Interfaces can be combined into a "superior" interface. Stated in a more code-focused way, an interface can extend more than one interface at a time.

### 2.2.4. HIGHER ORDER FUNCTIONS

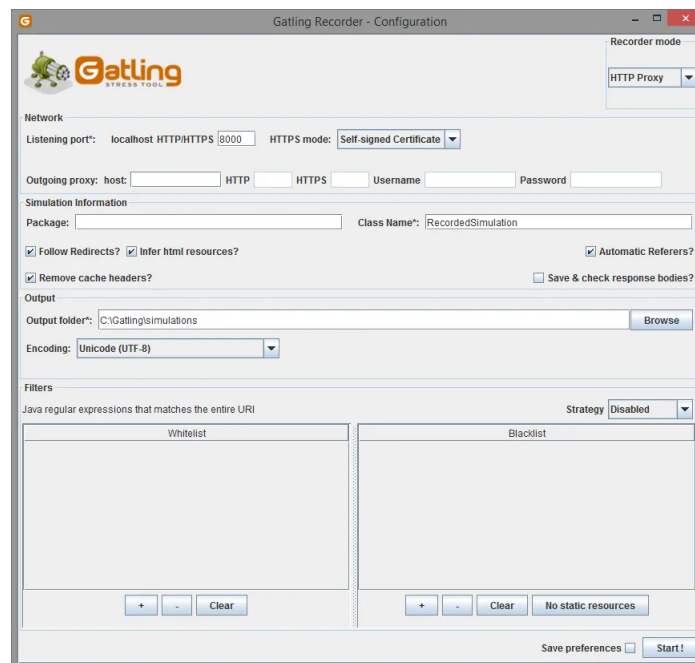
In ScaLa, functions are values and can be defined as anonymous functions with a concise syntax. In other words, this allows us to use functions as parameters to other functions. Hence the name higher order functions, since they use lower order functions as parameters.

## 2.3. HOW TO INTEGRATE WITH OUR COMPONENTS

### 2.3.1 GATLING RECORDER

There are two different ways to test the performance, using the gatling recorder feature to proxy the traffic going through a browser in order to test the performance, or doing a customize test to test the different endpoints.

Using the recorder is simpler and quicker but does not provide a quality test, so we will be doing our personal gatling tests to load & performance test our components.



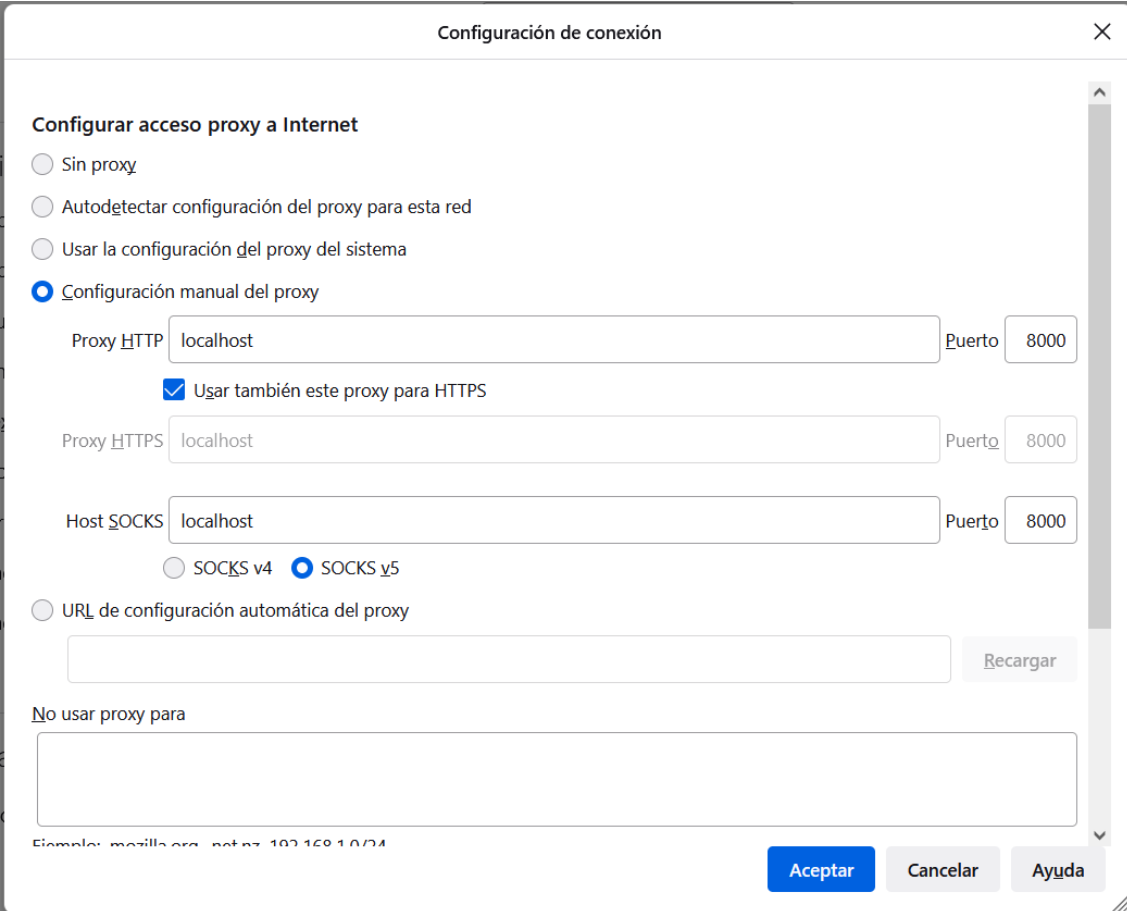
**Fig. 2.3.1.1.** Gatling Recorder UI

If we run the gatling recorder, a similar window as the one we could see in the last picture will appear.

In that new window we can specify which port will the gatling recorder will listen to and, if needed, we can also set an outgoing proxy just in case we are in a VPN.

With that previous configuration we would be able to test the performance of our proxying traffic, but if we want to, we can customize even more the recording by setting and configuring all the parameters we can see.

Once we are done with the recorder configuration, we need to open our usual browser to set it up as a local proxy with the port we specified in our recorder (localhost:8080)



The image shows a browser dialog box titled "Configuración de conexión" (Connection Configuration). It is used to configure internet proxy access. The "Configurar acceso proxy a Internet" section is active, with "Configuración manual del proxy" selected. The configuration includes:

- Proxy HTTP: localhost, Puerto: 8000
- Usar también este proxy para HTTPS
- Proxy HTTPS: localhost, Puerto: 8000
- Host SOCKS: localhost, Puerto: 8000
- SOCKS v4 (unselected) / SOCKS v5 (selected)
- URL de configuración automática del proxy: (empty field) with a "Recargar" button
- No usar proxy para: (empty text area)

At the bottom, there are buttons for "Aceptar" (Accept), "Cancelar" (Cancel), and "Ayuda" (Help). A small example text "Ejemplo: mozilla.org, net 193.169.1.0/24" is visible at the bottom left.

**Fig. 2.3.1.2.** Configuring browser as a Proxy

Now we can "Start" the recording and using the browser to record the HTTP request we perform, and they will be registered on the Gatling feature. When we are done doing request, we can just click on Stop & Save.

### 2.3.2 INTELLIJ PROJECT

First in IntelliJ go to create a new project of the Maven type, and there check the “Create from archetype” option. This means that a project will be created from a template that we must provide, and for this we touch the “Add Archetype...” button.

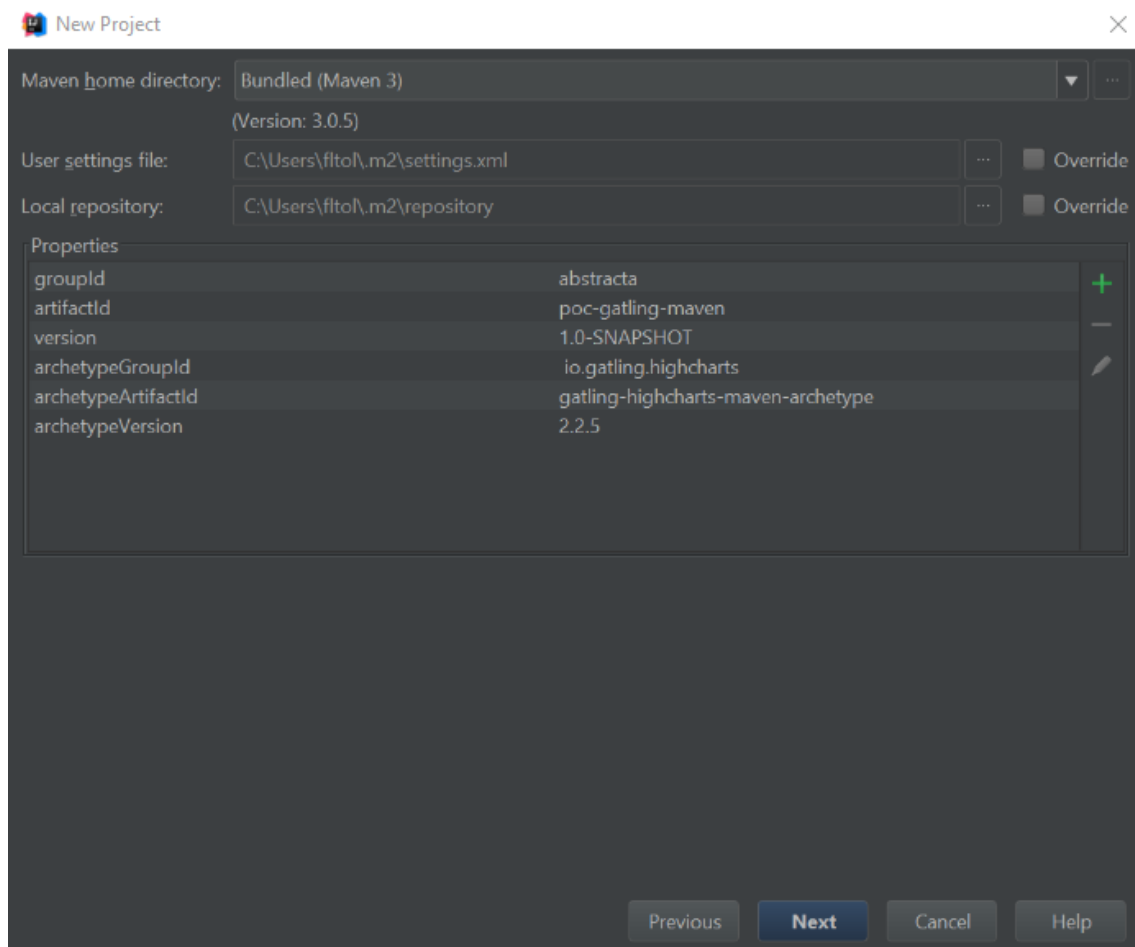
Next, add the archetype with these values:

- *GroupId* = *io.gatling.highcharts*
- *ArtifactId* = *gatling-highcharts-maven-archetype*
- *Version* = *desired version*
- *Repository* = *leave it empty*

Then we will be able to select the archetype created.

We continue with the wizard, and it asks us for the GroupId and ArtifactId of the project that we are going to create, we fill it with the values that we consider

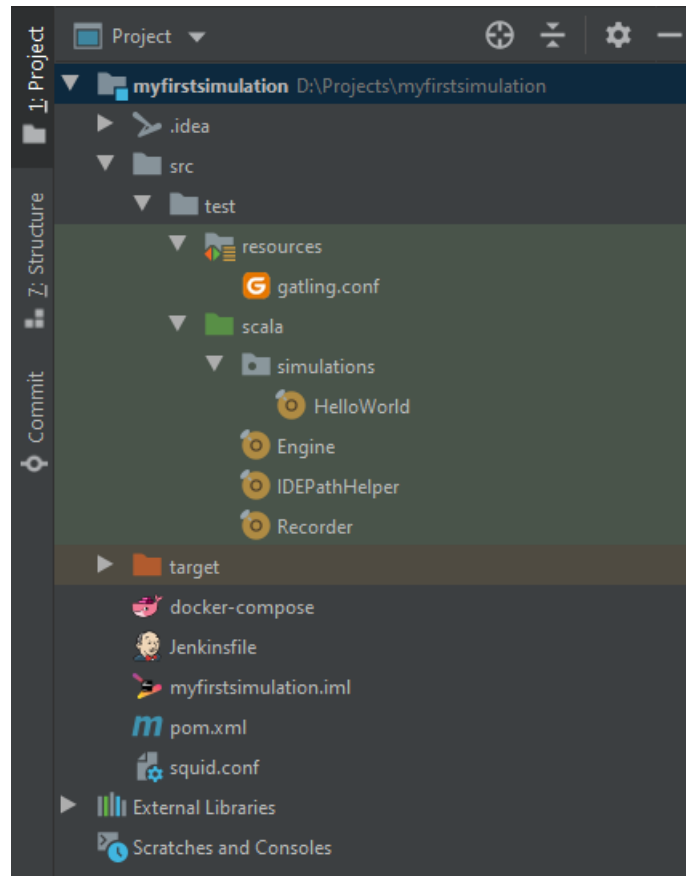
Once the previous steps have been carried out, we must validate the properties chosen for the project, if we are satisfied with the results we will click on the NEXT button.



**Fig. 2.3.2.1.** Creating a Gatling project



Once we have our very first Gatling project created, we will see something alike to this:



**Fig. 2.3.2.2.** Project structure

In that picture there are some files that won't be initially in a new project due to will be added later in order to include that testing in our CI/CD environment.

With that structure achieved, we could start to perform our Gatling Load Testing, but before starting to code anything, we are going to see what the logic is and how the tests should be done.

First, a single load test in Gatling is called a scenario. Roughly, a scenario can be divided into three parts:

- General configuration (protocol, server address, encoding ...)
- Steps to execute (open webpage, click this, enter that ...)
- Scenario configuration (no. of total users, users over time ...)

To begin with, we must define the initial HTTP configuration, where we must specify the URL, the content type, some headers and some other things.

```
val httpConf: HttpProtocolBuilder = http
    .baseUrl( url = "example.com")
    .acceptLanguageHeader( value = "en-US,en;q=0.5")
    .acceptEncodingHeader( value = "gzip, deflate")
    .contentTypeHeader( value = "application/json")
    .userAgentHeader( value = "Mozilla/5.0 (Windows NT 5.1; rv:31.0) Gecko/20100101 Firefox/31.0")
```

**Fig. 2.3.2.3.** HTTP configuration

On the other hand, there is the stage, the stage could be said to be where we define the what and the how; how long we want the test to last as a maximum, how many virtual users we want to have, if this value is going to be static or dynamic, if it is going to grow or decrease, what type of HTTP configuration we want to use and lastly the most important, what tests we will want to perform in this scenario.

Once we have mentioned these concepts, we can start defining the tests we want to perform. Doing this is quite simple, the only thing we need to do is keep in mind that we are testing and what we need for it, because if we are not careful and do not meet all the requirements of the request we want to test, we will get a failed test result and we will not have no explanation.

Normally, we can put all the tests that we want to carry out in the same scenario with the same HTTP configuration, since all the requests go to the same base URL, as is the case of the tests that are carried out on a WEB, but what happens if Instead of doing a general test, we want to do a specific test of different APIs with different ports, they no longer share URLs and it does not work with a single HTTP protocol configuration.

For this case we define test chains, we must think of these as blocks of tests that are carried out one after another until the chain ends, where in each test we specify the complete address of the URL where the test attacks. We can do tests chains of only a single request.

In this way, we can create an environment with more than one scenario (so that each scenario refers to an API that we want to attack) in which, in each scenario, one or more test chains are performed, with a single common HTTP protocol and basic (the URL specified in the protocol will be ignored since the URL specified in each test within the chain prevails).

As a summary, we must imagine this framework as if we had independent processes (chains of tests) that attack specific APIs, and to execute them we must create an environment by using a scenario and an HTTP configuration.

```
def exampleChain: ChainBuilder = exec(http( requestName = "Login Example")
  .post( url = "http://example.com/login")
  .headers(httpHeaders)
  .body(StringBody(
    string = """{"username":"userLogin",
    "password":"passwordLogin"}""").asJson
  .check(status is 200)
  .check(bodyString.saveAs( key = "ExampleLogin_Response"))
  .check(jsonPath( path = "$.auth").exists.saveAs( key = "paramAuth")))
  .pause( duration = 2)

  .exec(http( requestName = "Get Some Info")
    .get("http://example.com/getSomeInfo")
    .header( name = "authentication", value = "${paramAuth}")
    .check(status is 200)
    .check(bodyString.saveAs( key = "GetSomeInfo_Response"))
  )

  .exec(session => {
    println("Results")
    println(session("ExampleLogin_Response").as[String])
    println(session("GetSomeInfo_Response").as[String])
    println(session("paramAuth").as[String])
    session
  })
})
```

**Fig. 2.3.2.4.** Login Test example

In the previous picture we can see how we define a test chain[6], that executes a pair of tests.

We first execute a login with a username and a password as a body request (realize that the syntax of the body definition is quite weird), and the response of that request contains a parameter named auth. We save the response and the parameter, and in the next request we send the auth parameter as a header.

It can be seen how we must do to save responses and to save parameters of the responses as variables, so if we need some parameter that is obtained by doing a previous request, that's the way to do it.

In the fig. 2.3.2.4. we can see how its created a third test that does not perform any request[7], it just create a session in which we can work and do some logs by printing on the terminal, and we take profit of that by printing the responses we have previously saved.

With the tests and the HTTP configuration, we can create a simple scenario and relate it to one or more chain tests.

Then, we set up one or some scenarios with some users under a HTTP protocol.

```
class GatlingTest extends Simulation {

  val scnExample: ScenarioBuilder = scenario( scenarioName = "Example_Simulation")
    .exec(exampleChain)

  setUp(
    scnExample.inject(atOnceUsers( users = 1))
  ).protocols(httpConf)
}
```

**Fig. 2.3.2.5.** Example simulation

A typical casuistic that may be found, is that we first need to authenticate to some endpoint, and that authentication request provides we some access token or some value that allows us to perform some extra requests that we could not do without it.

So, imagine we want to load test a group of endpoints of page “B” by doing 10req/s to those endpoints, but in order to do so, we previously need to authenticate in page “A” to get the “token” is needed to perform the load testing of page “B”.

If we group all these petitions in the same scenario, we would be performing 10 req or more to the login endpoint of page “A” that will provide various tokens and we may have conflict with those values. So, the best solution is to split the scenarios and first dedicate one to perform the login and achieve the token with only one virtual user (Login Scenario), and other scenario to do the load testing with several virtual users using the token we got from the authentication done in the previous scenario.

Doing this, we only perform one request to the login in order to get what is needed to perform the load testing, and then we can adapt the scenario and user conditions to the ones that fit better with our requirements.

To implement this way of working, we need to do some extra steps:

- One step in order to get/set the value we achieve from performing a log in request.
- Another step to create a session with the parameters that we want to use as variables in the scenario

The idea of this is simple, is not in any guide of Gatling but it is an idea that many people could come up with due to it is an adaptation of the typical getter/setter methods from Java. It consists of create a HashMap and keep the values we want to with a name, and once they are saved, we can rescue them from it whenever we need to.

```
import java.util.concurrent.ConcurrentHashMap

object Token {

  val cache = new ConcurrentHashMap[String, String]

  def addTokenToCache(key:String, token: String): Unit = {
    if (token != cache.get(key)) {
      cache.put(key, token)
    }
  }

  def getTokenFromCache(param: String): (String) = {
    var token: String = new String
    token = cache.get(param)
    token
  }
}
```

Fig. 2.3.2.6. HashMap to store Token's

With that object named “Token”, we fulfill that need of getting and setting values to a cache. And then we need to do the second step that is to create the session[8] that will be used in our scenario. To do so, we must create a method that sets all the parameters we want in a new session.

```
object SetUpUserSession {

  def init: ChainBuilder = exec { session =>

    val newSession = session.set("someToken", Token.getTokenFromCache( param = "token"))
    println(newSession)
    newSession
  }

  def initAdmin: ChainBuilder = exec { session =>

    val newSessionAdmin = session.set("someToken-Admin", Token.getTokenFromCache( param = "token-admin"))
    println(newSessionAdmin)
    newSessionAdmin
  }
}
```

Fig. 2.3.2.7. Establishing different sessions

With these two steps, we just need to perform this test before the load testing, we want to do in our scenario to set the parameters as “environment variables”.

```
val scnCustomerConfiguration: ScenarioBuilder = scenario( scenarioName = "Customer Configuration")
    .exec(SetupUserSession.init)
    .exec(Example.Request)
    .exec(Example.RequestWithQuery)
    .exec(Example.Request)
```

**Fig. 2.3.2.8.** Defining a Scenario

In this scenario we set a session with the token provided from a basic Log In, but imagine we could perform an Admin Log In that is different from the basic one, we can distinguish the value between them using the “initAdmin” method that gets another param from the HashMap (cache).

With this previous step we can acquire the parameters that other petitions may need in order to perform several HTTP requests.

## CHAPTER 3. JENKINS

### 3.1. WHAT IS JENKINS

Jenkins[16] is a tool made for CI/CD[12] processes. At the very beginning it is quite difficult to start using it due to its big learning curve which makes it difficult to become familiar with this software.

Jenkins is a software that is very basic in itself, which is designed to be super scalable, in such a way that its operation can be customized using plugins[15].

A lot of tools provide a plugin for Jenkins in the same way that a docker image is provided. So once the CI/CD software is installed, we should think about what plugins may be necessary to get the most out of this great tool.

To install Jenkins, we have a lot of alternatives, depending on the device we have and our knowledge in devOps.

The two most common are to install locally using the installer we have (downloading for Windows, Linux or macOS depending on the OS we use), or by downloading a Jenkins docker image and running it in a container.

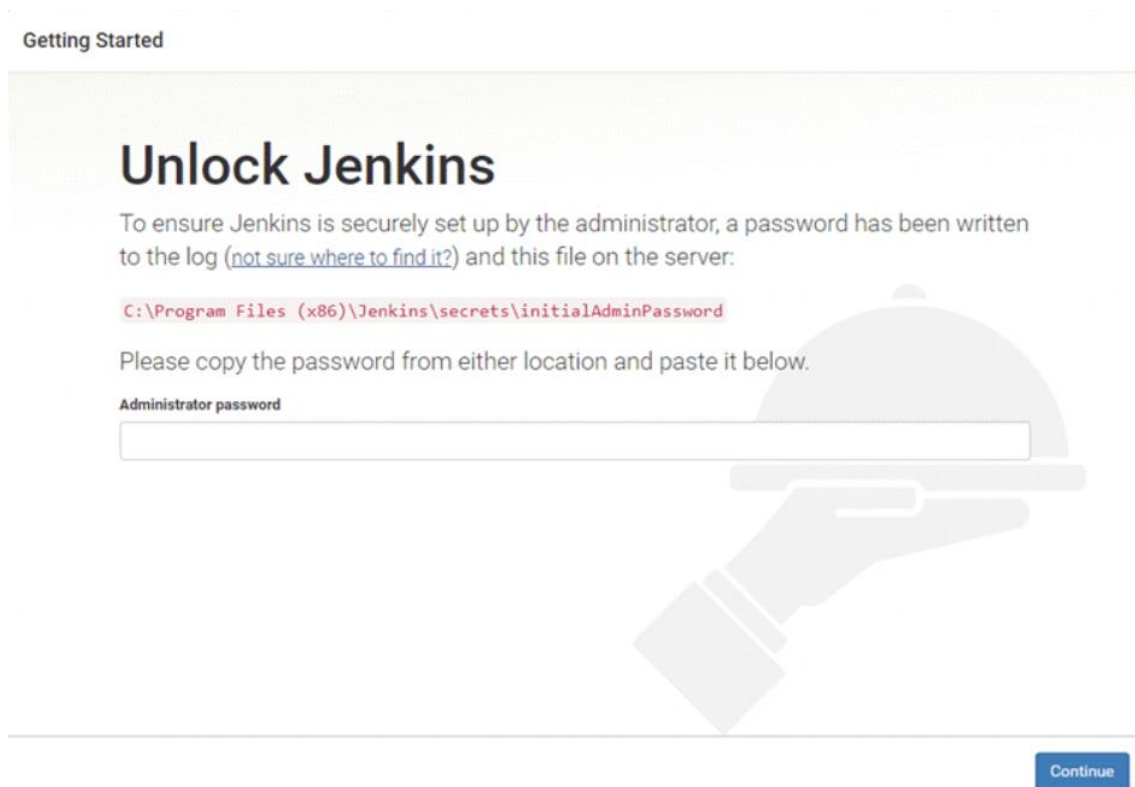
### 3.2. INSTALLATION

To install Jenkins, the first thing we must do is download the tool.

The installation can be done in two ways, either we download a file such as an .msi for Windows, or we can also download a docker image and have our server for automation in a CI/CD environment running in docker.

The two alternatives are equally valid, and their installation is practically identical.

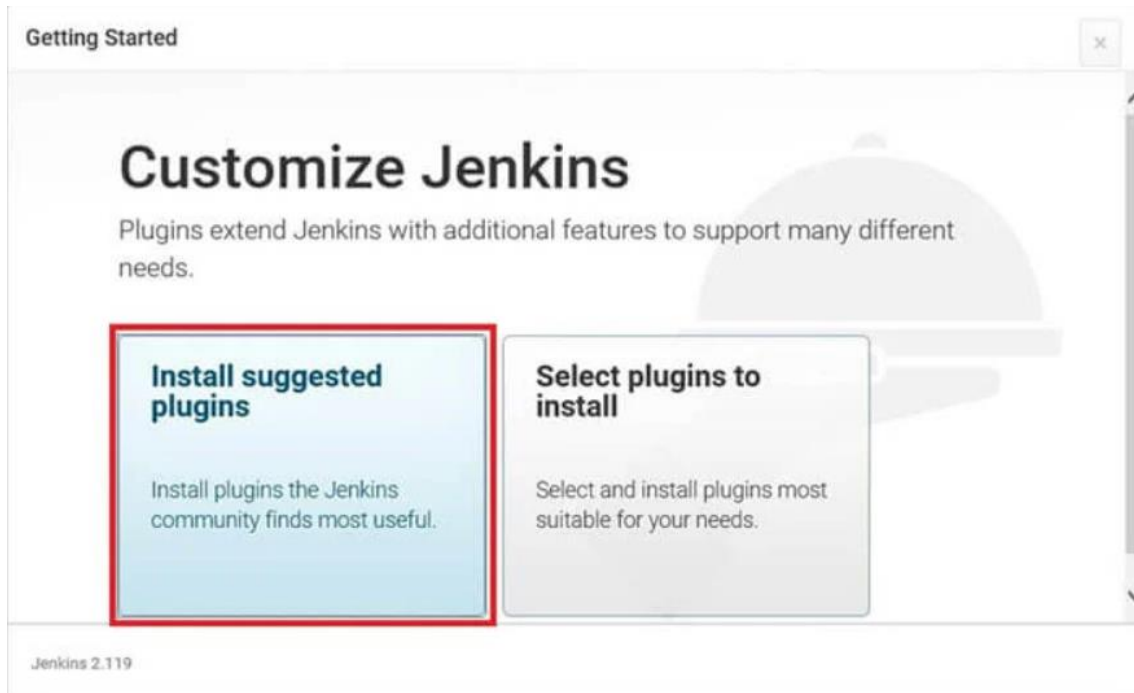
Once the executable has been downloaded, we just must open the file and start it on our computer. Once this is done, we will start with the configuration.



**Fig. 3.2.1.** Unlocking Jenkins for installation

We will see a screen like the one in the previous image, this step is done to verify that the person who is doing the installation is the one who has downloaded the software, or at least that he has the necessary permissions, for that we must go to the address that appears in the image, open the file and copy the password and then paste it in the Administrator Password field.





**Fig. 3.2.2.** Installing plugins

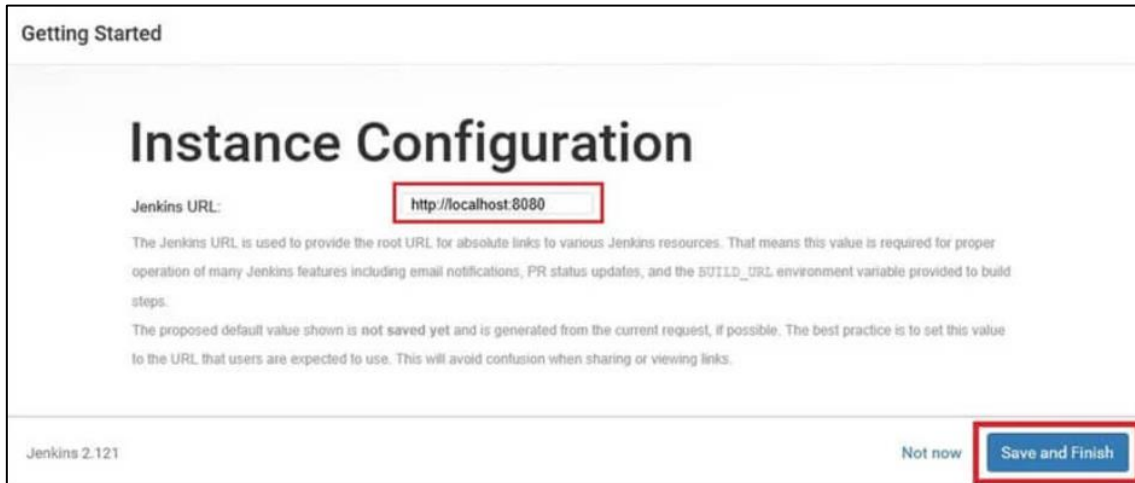
Once the user has been verified, we can start configuring our Jenkins by installing plugins. To begin with, there are a series of plugins that are recommended to be installed by default. They are the basic plugins to be able to use Jenkins, but in most cases other functionalities must be added to get the most out of this integration software.

The screenshot shows the 'Getting Started' window in Jenkins. The main heading is 'Create First Admin User'. Below the heading are five input fields: 'Username:', 'Password:', 'Confirm password:', 'Full name:', and 'E-mail address:'. At the bottom of the window, there are two buttons: 'Skip and continue as admin' and 'Save and Continue'. At the bottom left of the window, it says 'Jenkins 2.238'.

**Fig. 3.2.3.** Create first Admin User

Once the installation of the selected plugins is finished, we must create the first Admin user, in the future we will be able to create more admin users, but it is recommended to create the first one in that step.

To do this, we will fill in the required fields and click on save and continue.



The screenshot shows the 'Getting Started' section of the Jenkins Instance Configuration page. The title 'Instance Configuration' is prominently displayed. Below it, the 'Jenkins URL:' field is filled with 'http://localhost:8080'. A detailed explanation follows, stating that the Jenkins URL is used for absolute links to various resources and is required for features like email notifications and PR status updates. It also notes that the default value is not saved yet and is generated from the current request. At the bottom right, there are two buttons: 'Not now' and 'Save and Finish', with the latter being highlighted with a red box.

**Fig. 3.2.4.** Set port for Jenkins

The last step before we can start our Jenkins locally for the first time is to specify the port.

By default, it is assigned 8080, but my advice is to change the port because the use of port 8080 is quite common, and in the future, there may be port conflicts between Jenkins and other components.

### 3.3. HOW TO USE JENKINS

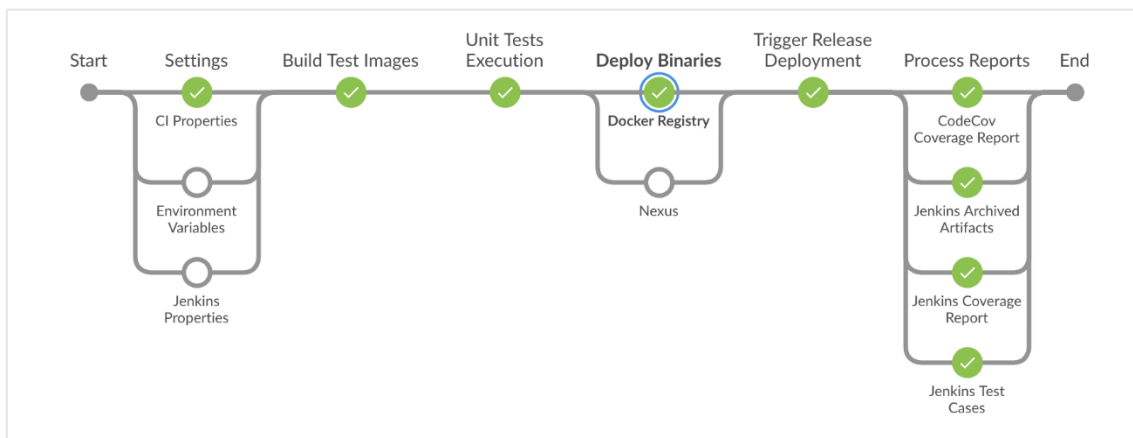
In this software it is a bit more difficult to classify or group the necessary components to make good use of the tool. This is because there are many ways to do things, and the number of things to configure varies a lot depending on how complex is what we want our job to do. Even so, next we will see what is essential to start automating a process with Jenkins.

In Jenkins, the distinction of processes is made through "jobs", a job is a process or group of processes that can be executed both manually and automatically with a configuration that has previously been completed.

These jobs make use of files called "Jenkinsfile". A good comparison of these files with an object of daily life would be some instructions, since the jobs use the jenkinsfile to know what they must do in each step of the job execution.

Within a jenkinsfile we can find a configuration part, such as parameters or environment variables, and the "stages". The stages are the different steps in a jenkinsfile. Each stage can contain different processes, ranging from displaying a phrase on the screen or the content of a directory to executing a second job within another.

All the stages of a jenkinsfile combined make the "pipeline"[13], that is how it is called all the processes run by a job.



**Fig. 3.3.1.** Jenkins's pipeline

Once we meet these "indispensable" requirements, we can start having automated processes using jobs. Therefore, we are going to see how we can start to create/configure the different parts necessary to obtain a functional integration software.

### 3.3.1. HOW TO CREATE A JOB

To integrate anything into a pipeline[14] in order to run some processes we need to create a Job (we can create it from scratch, duplicate an existing job, a maven project...). In this job is where we will set the extra-configuration, specify the repository and more.

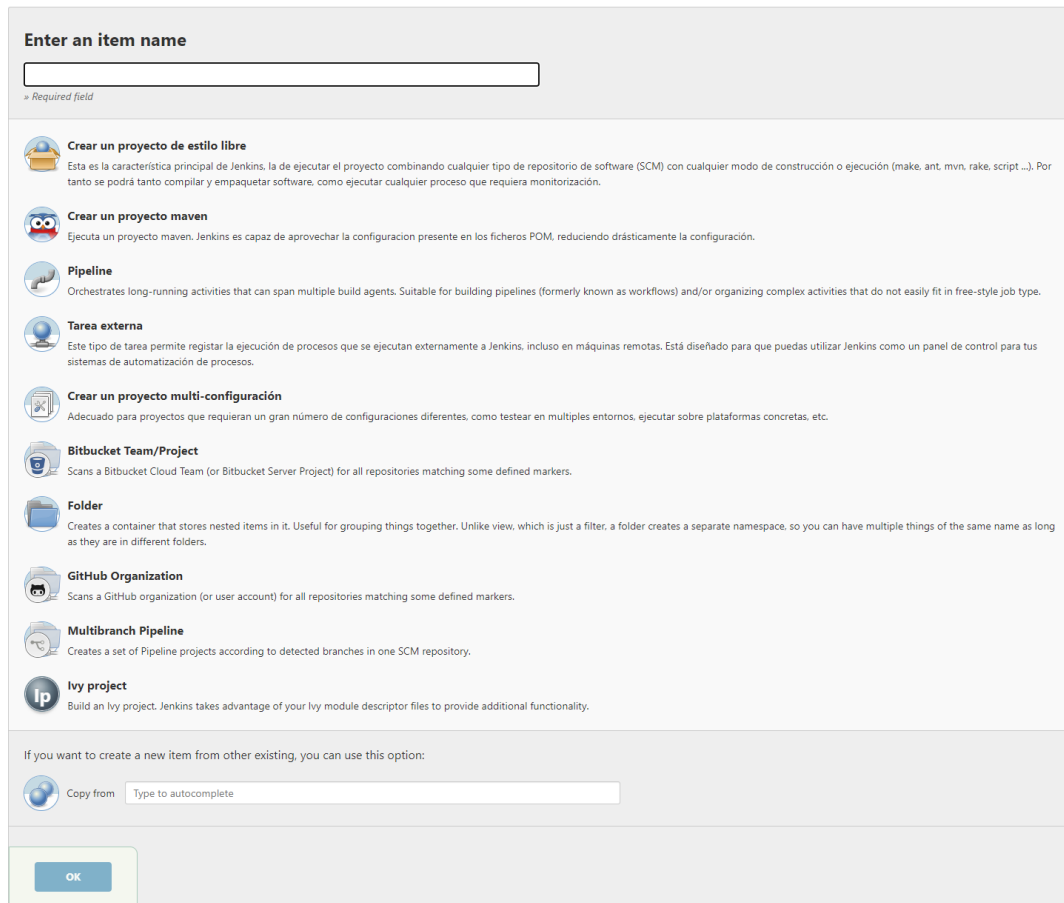


Fig. 3.3.1.1. Create a new Job

*(To do not have to do the configuration from scratch, its strongly recommend duplicating a job that fits with our needs, and then adapt the configuration with the desired repository and some parameters. To do that, we must fill the path of the project)*

We can clone an existing Job by copying the “Full project name” of the project we want to copy and inserting it in the option “copy from” when creating a new Job.

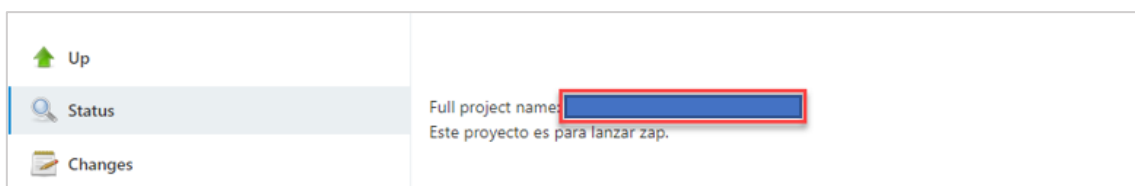


Fig. 3.3.1.2. Full project name Path

### 3.3.2. HOW TO MAKE A JENKINSFILE

First, to run a job in Jenkins we need a JENKINSFILE.

A Jenkinsfile is a file in which we specify what, how and when are things going to be done.

There are two different ways to make a Jenkinsfile:

- Declarative Pipeline
- Scripted Pipeline

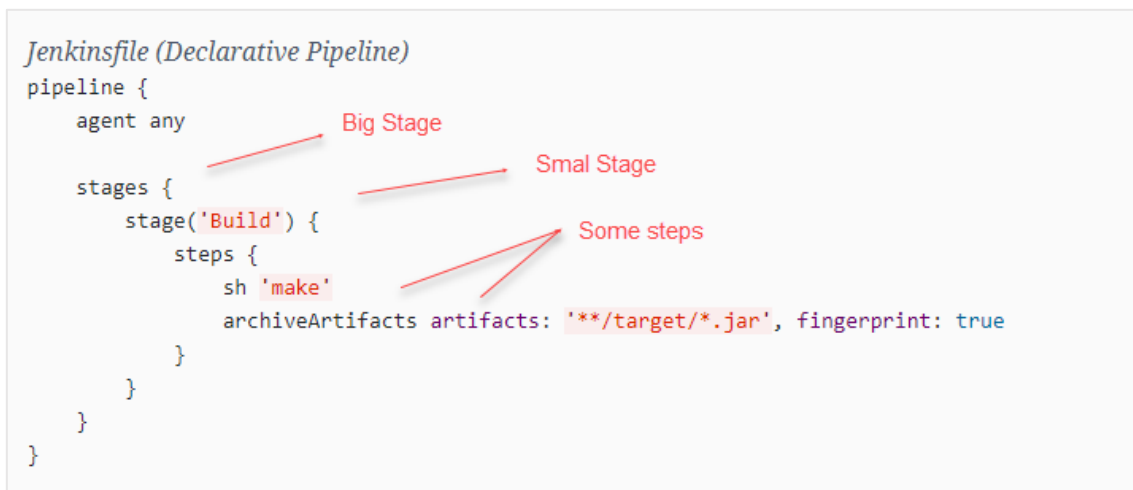
The declarative one is the easiest, and its use is recommended for those who are not into Jenkins. The scripted pipeline may be more complete, but it is way more difficult to learn how to create a jenkinsfile.

The jenkinsfile will be inside of a repository with other files (scripts, docker-composes or even projects) and it will rule what to do and how. As if it was a guide that shows the job how to do the things we want to.

Inside the jenkinsfile we can differentiate three different things.

- A big Stage that may have smaller stages inside of it.
- Stages inside a BIGGER STAGE that run different steps
- Steps that specify what to do.

All those methods are included inside the pipeline definition.



**Fig. 3.3.2.1.** Declarative pipeline example

If we do not specify an agent, it will be selected randomly, but we also select a specific machine to run our job.

We can set some environment variables, that will be used later.

```

pipeline {
  agent {
    label params.JENKINS_NODE
  }
  environment {
    UPSTREAM_USER = '...'
    PDB_PASSWORD = '...'
    PDB_NAME = '...'
    ORACLE_HOST = '...'
    PDB_APPS = '...'
  }
  stages {
    stage('Config') {
      steps {
        sh 'docker-compose up -d config'
      }
    }
    stage('Create Oracle pdb') {
      when{
        expression{
          return params.CREATE.toBoolean()
        }
      }
      steps {
        catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
          build job: '...', parameters: [
            string(name: 'SYSTEM_CODE', value: "${PDB_NAME}"),
            string(name: 'PDB_APPS', value: "${PDB_APPS}"),
            string(name: 'ORACLE_HOST', value: "${ORACLE_HOST}"),
            string(name: 'UPSTREAM_USER', value: "${UPSTREAM_USER}"),
            password(name: 'PDB_PASSWORD', value: "${PDB_PASSWORD}")
          ]
        }
      }
    }
  }
}

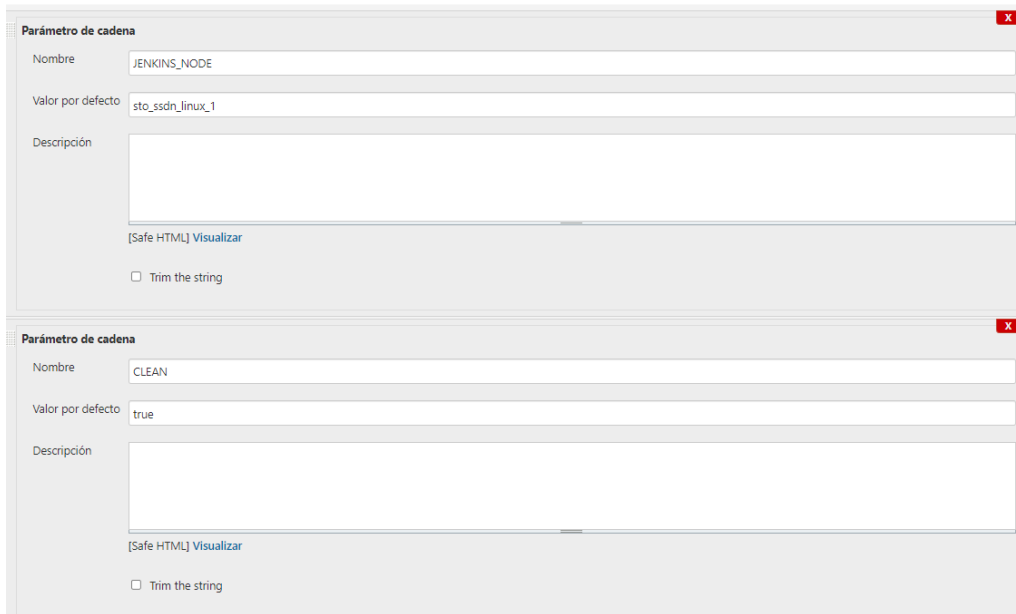
```

**Fig. 3.3.2.2.** Environment variables in Jenkinsfile

A really good tool for the Jenkinsfile are the parameters, we can use parameters that we will specify when running the job in order to modify the performance.

In the previous picture we can see the parameter CREATE, so the stage to Create the Oracle PDB will only be executed when that parameter is true.

To add parameters, we must modify the configuration of our job.

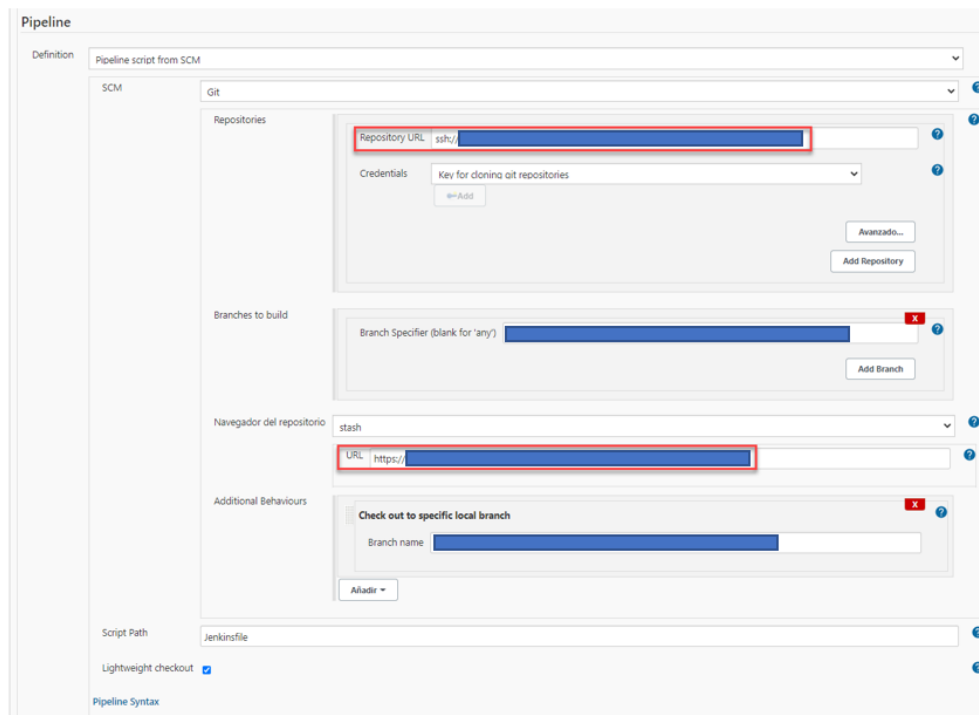


The image shows two instances of the 'Parámetro de cadena' (String Parameter) configuration form in Jenkins. The first form has 'Nombre' (Name) set to 'JENKINS\_NODE' and 'Valor por defecto' (Default Value) set to 'sto\_ssdn\_linux\_1'. The second form has 'Nombre' set to 'CLEAN' and 'Valor por defecto' set to 'true'. Both forms include a 'Descripción' (Description) field, a '[Safe HTML] Visualizar' (View) button, and a checkbox for 'Trim the string'.

**Fig. 3.3.2.3.** Add parameters in Job

In this job, there are two parameters, one for specify it some stages are done, and another one to specify which machine we are using. As we can see, we can set a default value for those parameters.

An important thing is to have the repository well-configured. To do so, we need to specify the URL here.



The image shows the 'Pipeline' configuration page in Jenkins. The 'Definition' is set to 'Pipeline script from SCM'. The 'SCM' is set to 'Git'. The 'Repositories' section shows a 'Repository URL' field with an SSH URL, a 'Credentials' dropdown set to 'Key for cloning git repositories', and an 'Add Repository' button. The 'Branches to build' section shows a 'Branch Specifier (blank for \'any\')' field and an 'Add Branch' button. The 'Navegador del repositorio' (Repository browser) is set to 'stash', and the 'URL' field contains an HTTPS URL. The 'Additional Behaviours' section shows a 'Check out to specific local branch' checkbox and a 'Branch name' field. The 'Script Path' is set to 'Jenkinsfile'. The 'Lightweight checkout' checkbox is checked. The 'Pipeline Syntax' section is visible at the bottom.

**Fig. 3.3.2.4.** Specify git repository

As we can see, we also can select which branch do we want to use.

The project will be downloaded from the repository and the branch that have been specified in those fields. In that project there must be at least the jenkinsfile so that the job can be "compiled". It is normally accompanied by a docker-compose to deploy an environment with components and be able to simulate what is done in a local environment, and sometimes it can also be accompanied by certain logic, generally tests.

Even so, it is very likely that everything that is done locally cannot be exactly replicated in Jenkins, since Jenkins runs on a Linux machine that compiles code and commands automatically and unitarily, so it is difficult to concatenate commands. For example, a simple docker command as "docker exec -it XXX bash" to get into a container, and once we are inside, do other commands (ls, mkdir, cp...), turns to be a really complicated task due to the commands in the Linux Machine on Jenkins are secuencial, that means we can not execute commands inside another command the way we are used to.

A really useful and organised way to work is splitting the task in diversal jobs. Doing so we are able to reuse the performance of that job wherever we want to.

Some good tasks to split from the initial job may be:

- Create a database
- Clean a database
- Drop a database
- Use a special software through our job
- Do a quality check

We can call another job from our jenkinsfile and specify the parameters required in order to run that job this way

```
stage('Create Oracle pdb') {
  when{
    expression{
      return params.CREATE.toBoolean()
    }
  }

  steps {
    catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
      build job: '...', parameters: [
        string(name: 'SYSTEM_CODE', value: "${PDB_NAME}"),
        string(name: 'PDB_APPS', value: "${PDB_APPS}"),
        string(name: 'ORACLE_HOST', value: "${ORACLE_HOST}"),
        string(name: 'UPSTREAM_USER', value: "${UPSTREAM_USER}"),
        password(name: 'PDB_PASSWORD', value: "${PDB_PASSWORD}"),
      ]
    }
  }
}
```

**Fig 3.3.2.5.** Trigger another job from Jenkinsfile

In that case, the parameters are variables declared at the environment.



## CHAPTER 4. ANALYSIS

Below we can see a series of results provided by the ZAP and Gatling tools that we are going to analyze.

### 4.1. Pen-testing with ZAP

#### Summary of Alerts

Risk Level	Number of Alerts
<a href="#">High</a>	1
<a href="#">Medium</a>	1
<a href="#">Low</a>	6
<a href="#">Informational</a>	3

**Fig. 4.1.1.** Summary of alerts

Related to vulnerability analysis, here we have two images with risk information in a WEB domain.

From these captures we can get two types of clear information, the first is that in the analyzed domain there are 11 different types of alerts, of which 8 are risky.

#### Alerts

Name	Risk Level	Number of Instances
Cross Site Scripting (Reflected)	High	1
X-Frame-Options Header Not Set	Medium	58
Absence of Anti-CSRF Tokens	Low	2
Cookie No HttpOnly Flag	Low	3
Cookie Without SameSite Attribute	Low	3
Cookie Without Secure Flag	Low	3
Incomplete or No Cache-control and Pragma HTTP Header Set	Low	60
X-Content-Type-Options Header Missing	Low	70
Charset Mismatch (Header Versus Meta Content-Type Charset)	Informational	7
Information Disclosure - Suspicious Comments	Informational	1
Timestamp Disclosure - Unix	Informational	2

**Fig. 4.1.2.** More detailed information of alerts

The second table shows which are the different vulnerabilities found in the web page. Here we can see that the most dangerous are Cross-Site Scripting, which can enable third parties to execute code in our program, and X-Frame Options Header Not Set, which indicates that there are some headers that are incomplete.

We also have other alerts that are classified with a lower risk, since as we can see they are classified into four different groups: high, medium, low & informational.

Within the Low alerts we have six different risks and finally there are three alerts grouped under the Informational range, these informational alerts report "bad practices" that do not cause vulnerabilities but that it would be good to correct.

One of the most favorable points of the ZAP software is how well detailed the information is. It provides the data completely and with extensive explanations of WHAT, WHY and HOW the vulnerabilities it finds occur. In addition, if they are well-known vulnerabilities, they are usually accompanied by a solution. In other words, ZAP itself recommends how to combat these vulnerabilities in the different phases of implementation.

Below is an example of a Cross-site Scripting vulnerability found in a scan performed by ZAP.

**Cross Site Scripting (Reflected)**

Cross-site Scripting (XSS) es una técnica de ataque que comprende hacer eco del código que fue proporcionado por el atacante en la instancia del navegador de un usuario. Una instancia de navegador puede ser un cliente de navegador web corriente, o un objeto de navegador integrado e un producto de software, como el navegador que se encuentra dentro de WinAmp, un lector de RSS o un cliente de correos electrónicos. El código por sí mismo se encuentra escrito en HTML/JavaScript, pero también puede extenderse a VBScript, ActiveX, Java, Flash o cualquier otra tecnología que sea compatible con el navegador.

Cuando un atacante consigue el navegador de un usuario para poder ejecutar su código, el código se ejecutará dentro del contexto de seguridad (o zona) del sitio web de hospedaje. Con este nivel de privilegio, el código tiene la extensión de leer, modificar y transmitir cualquier dato que sea sensible al que pueda ingresar al navegador. Un usuario de Cross-Site Scripted puede ser que tenga su cuenta secuestrada (robo de cookies), su navegador redirigido a otra ubicación, posiblemente mostrando contenido ilegal entregado por el sitio web que están visitando. Los ataques de scripting entre los sitios relativamente comprometen la relación de la confianza entre el usuario y el sitio web. Las aplicaciones que usan instancias de objetos del navegador que suben contenido desde el sistema de archivos puede activar el código bajo la zona de la máquina, lo cual permite que el sistema se vea comprometido.

Hay tres tipos de ataques diferentes de scripting entre los sitios: no persistentes, persistentes y basados en DOM.

Los ataques que no son persistentes y los basados en DOM necesitan que el usuario visite un enlace que fue diseñado con código malicioso o visite alguna página web maliciosa que incluya un formulario web que, cuando se publique en el sitio que es vulnerable, originará el ataque. El uso de un formulario que es malicioso normalmente tendrá lugar cuando el recurso que es vulnerable solo acepte las solicitudes HTTP POST. En tal caso, el formulario puede ser enviado de forma automática, sin el conocimiento de la víctima (por ejemplo, por medio de JavaScript). Al hacer clic en el enlace que es malicioso o al enviar el formulario malicioso, la carga que es útil de CSS recibirá eco y será interpretada por el navegador del usuario y se activará. Otra técnica para poder prevenir solicitudes casi arbitrarias (GET y POST) es por medio del uso de un cliente integrado, como adobe Flash.

Los ataques continuos se originan cuando el código que es malicioso se envía a un sitio web donde se almacena durante un periodo de tiempo. Algunos ejemplos de los objetivos preferidos de los atacantes incluyen mensajes en carteleras de anuncios, mensajes de correo electrónico y programas de chat. El usuario desprevenido no tendrá que interactuar con ningún sitio/enlace adicional (por ejemplo, un sitio o link malicioso enviado por correo electrónico), solamente bastará con abrir la página web que contiene el código.

**Fig. 4.1.3. Explanation of an alert**

As we can see, we are provided with an explanation of what this vulnerability consists of. This way we can obtain an idea of what is happening, or what could happen to that website.

**Frase: Arquitectura y Diseño**

Utilice una biblioteca o marco comprobado que no acepte que ocurra esta debilidad o que proporcione construcciones que permitan que esta debilidad sea más sencilla de evitar.

Los ejemplos de las bibliotecas y marcos que facilitan el origen de resultados que son codificados de forma correcta incluyen la biblioteca Anti-XSS de Microsoft, el módulo de codificación OWASP ESAPI y Apache Wicket.

**Fases: Implementación; Arquitectura y Diseño**

Comprenda el contexto en el que se va a utilizar sus datos y la codificación que se va a esperar. Esto es fundamentalmente importante cuando se transmiten los datos entre diferentes componentes o cuando se generan las salidas que pueden comprender múltiples codificaciones al mismo tiempo, como páginas web o mensajes de correos de varias zonas. Estudie todos los protocolos de comunicación y representaciones de los datos que son esperadas para poder determinar las estrategias de codificación que son necesarias.

Por cualquier dato que se enviará a otra página web, en especial cualquier dato recibido de las entradas externas, utilice la codificación que sea conveniente en todos los caracteres que no sean alfanuméricos.

Consulte la hoja de referencia de prevención de CSS para poder obtener más información detallada de los diferentes tipos de codificación y escape que se requieren.

**Fase: Arquitectura y Diseño**

Cualquier comprobación de seguridad que se vaya a realizar en el lado del cliente, asegúrese de que estas comprobaciones se encuentren duplicadas en el lado del servidor, para evitar el CWE-602. Los atacantes pueden eludir las comprobaciones del lado del cliente modificando los valores después de que se hayan realizado las comprobaciones, o cambiando al cliente para poder eliminar de forma completa las comprobaciones del lado del cliente. Después, estos valores que fueron modificados serán enviados al servidor.

Si se encuentra disponible, utilice los mecanismos estructurados que apliquen de forma automática la separación entre los datos y códigos. Estos mecanismos pueden otorgar la cotización, codificación y validación relevantes de manera automática, en lugar de confiar en que el desarrollador proporcione esta capacidad en cada uno de los puntos donde se origina la salida.

**Fig. 4.1.4. Solutions in “designing” phase**

**Fase: Implementación**

Para cada una de las páginas web que se origina, utilice y especifique una codificación de caracteres como ISO-8859 o UTF-8. Cuando no se puede especificar una codificación, el navegador web podría seleccionar una codificación distinta adivinando que codificación está siendo utilizada en verdad por la página web. Esto puede permitir que el navegador web trate varias secuencias como especiales, abriendo al cliente a leves ataques XSS. Consulte CWE-116 para conseguir más mitigaciones con respecto a la codificación/escape.

Para ayudar a mitigar los ataques XSS contra las cookies de la sesión del usuario, es necesario establecer que la cookie de la sesión sea HttpOnly. En navegadores que son compatibles con la característica HttpOnly (como las versiones más actualizadas de Internet Explorer y Firefox), esta característica puede prevenir que la cookie de sesión del usuario sea accesible para las secuencias de comandos del lado del cliente malignas que utilizan document.cookie. Esta no es una solución muy completa, ya que HttpOnly no es compatible con todos los navegadores que hay. Más importante aún, XMLHttpRequest y otras tecnologías poderosas de navegador otorgan acceso de lectura a los encabezados HTTP, incluido el encabezado Set-Cookie en el cual se establece el indicador HttpOnly.

Asuma que toda la entrada es maliciosa. Utilice una estrategia de validación de entrada "aceptar bien conocidos", es decir, utilice alguna lista blanca de entradas aceptables que se ajuste de forma estricta a las especificaciones. Rechace cualquier entrada que no se adapte de forma estricta a las especificaciones, o cambíelas por algo que sí lo haga. No confíe solamente en la búsqueda de entradas maliciosas o malformadas (es decir, no confíe en una lista negra). Sin embargo, las listas negras pueden ser muy útiles para detectar posibles ataques o diagnosticar que entradas están tan malformadas que se deberían rechazar directamente.

Al realizar la validación de entrada, usted debe considerar todas las propiedades potencialmente destacadas, incluida la longitud, el tipo de entrada, el rango completo de valores aceptables, las entradas faltantes o adicionales, la sintaxis, el sentido entre los campos que se encuentran relacionados y la conformidad con todas las reglas comerciales. Como un ejemplo de lógica de las reglas comerciales, "boat" quizás sintácticamente puede ser válido porque solo posee caracteres alfanuméricos, pero no es válido si está esperando como "rojo" o "azul".

Asegúrese de hacer la aceptación de las entradas en interfaces que se encuentren bien definidas dentro de la aplicación. Esto ayudará a cuidar la aplicación, incluso si un elemento se utiliza de nuevo o traslada a otro sitio.

**Fig. 4.1.5. Solutions in “implementation” phase**

## 4.2. Work/Stress Load with GATLING

On the other hand, in relation to load tests we have the Gatling tool. Next, we are going to see a series of images of an analysis of an API in which, through the use of this software, a client is simulated that makes a series of requests and analyzes the response times, the standard deviation of these times, how many requests fail and how many are answered correctly and many other data, in addition to many graphs that explain very well the behavior of the application.

The following test consists of eleven different HTTP requests, of which some are made only once and others are executed by more than one virtual user.

In total, 29 requests are made against an API and all these requests are answered with an HTTP 200 (OK). This indicates that the application processes them correctly.

With the processing and the response times that are received, the Gatling software creates the different graphs that we will see later.

The following image is the first one we see when we open a gatling test report.

This graph is a summary of the number of requests that have been made in the test and includes them in three time intervals.

There is also another pie chart further to the right that shows the number of times each request has been made.

In this way we can see that 25 of the 29 requests have taken more than 1200ms to obtain a response and the remaining 5 have had a response between 800ms and 1200ms. Also, as seen in the graphs, all requests have been OK.

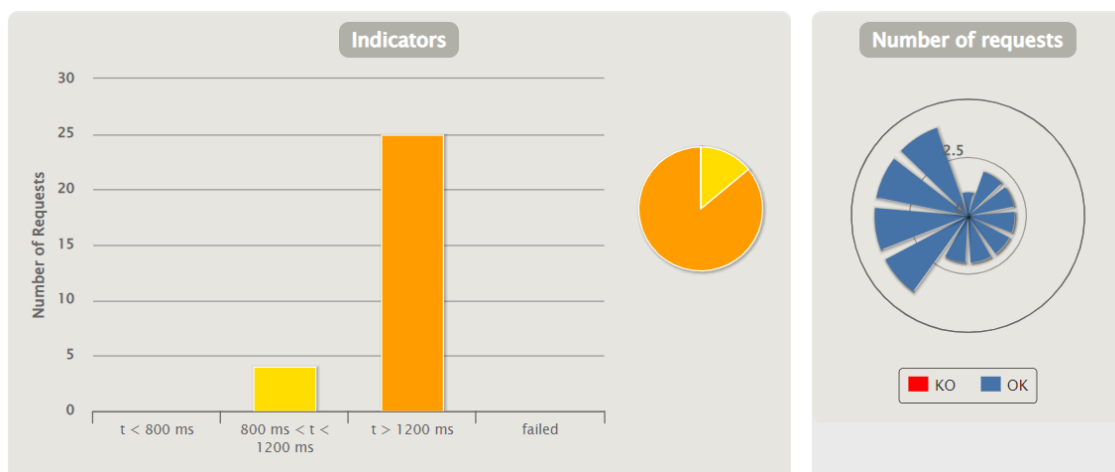


Fig. 4.2.1. Initial graphics with summary information

The second piece of information is a table that shows all the data collected in the test as far as times are concerned. The different requests made are classified separately and it shows for each request the number of times it has been made, the times it has been KO and OK, the number of events per second (cnt/s), the values of the percentages to track in the reports (it only gives relevant information if we have a request repeated more than once), maximum time and minimum time, mean time and standard deviation.

Requests ^	Executions					Response Time (ms)								
	Total ↕	OK ↕	KO ↕	% KO ↕	Cnt/s ↕	Min ↕	50th pct ↕	75th pct ↕	95th pct ↕	99th pct ↕	Max ↕	Mean ↕	Std Dev ↕	
Global Information	29	29	0	0%	0.387	1113	1499	2514	15460	23532	23688	3401	5493	
Request_Demo_0	1	1	0	0%	0.013	3367	3367	3367	3367	3367	3367	3367	0	
Request_Demo_1	2	2	0	0%	0.027	1401	12267	17699	22045	22915	23132	12267	10866	
Request_Demo_4	2	2	0	0%	0.027	1473	1475	1475	1476	1476	1476	1475	2	
Request_Demo_5	2	2	0	0%	0.027	2523	2738	2846	2932	2949	2953	2738	215	
Request_Demo_10	2	2	0	0%	0.027	2513	2690	2779	2849	2863	2867	2690	177	
Request_Demo_2	2	2	0	0%	0.027	1473	1474	1474	1474	1474	1474	1474	1	
Request_Demo_3	2	2	0	0%	0.027	2455	2485	2499	2511	2513	2514	2485	30	
Request_Demo_6	4	4	0	0%	0.053	2290	3200	8885	20727	23096	23688	8095	9026	
Request_Demo_9	4	4	0	0%	0.053	1466	1519	1552	1585	1591	1593	1524	47	
Request_Demo_7	4	4	0	0%	0.053	1472	1486	1492	1493	1493	1493	1484	9	
Request_Demo_8	4	4	0	0%	0.053	1113	1145	1171	1184	1186	1187	1147	30	

Fig. 4.2.2. Table with information on response times

The fig. 4.2.3. refers to the configuration of virtual users for the configured scenarios, and how they are managed throughout the test that we carry out. In this way we can see that we have three different scenarios (blue, green and red colors) and then we have a line of active users that is the sum of the users of all the scenarios. From what we can see that we start with a user from scenario 1, then scenario 1 ends and we continue with two users from scenario 2, which then decreases to 1 and finally ends to make way for the 4 users from scenario 3.

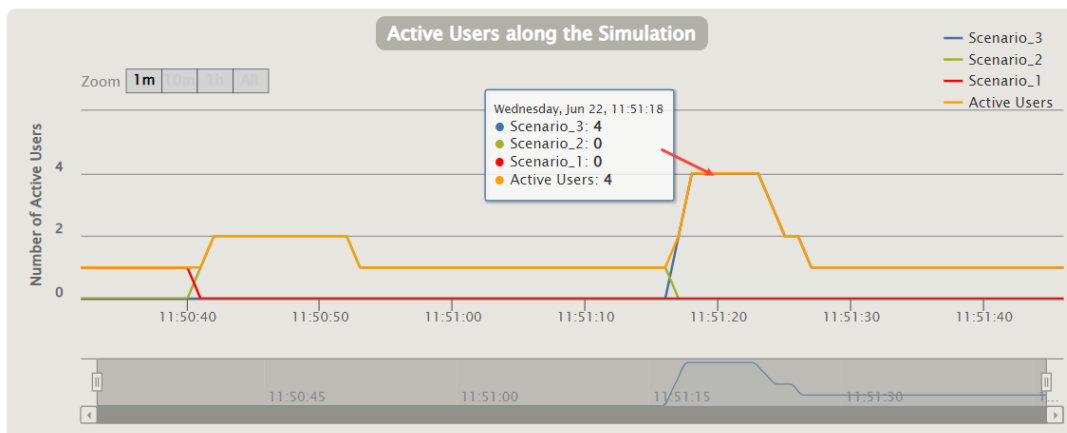
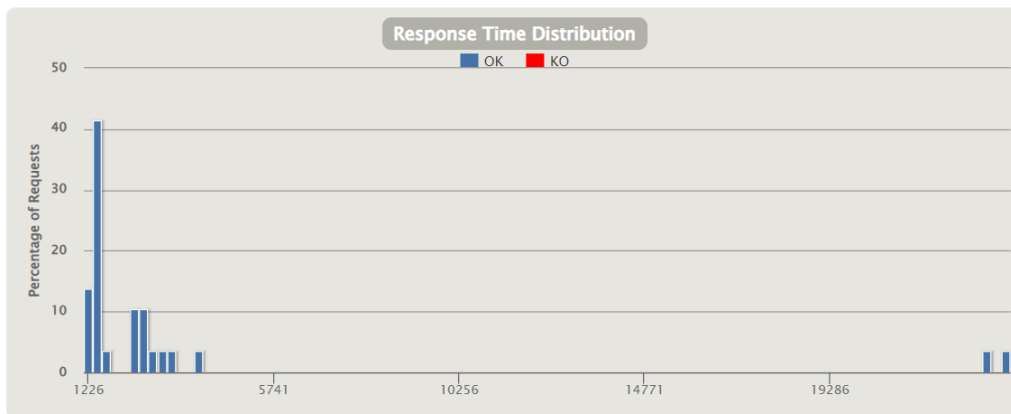


Fig. 4.2.3. Active Users along the simulation

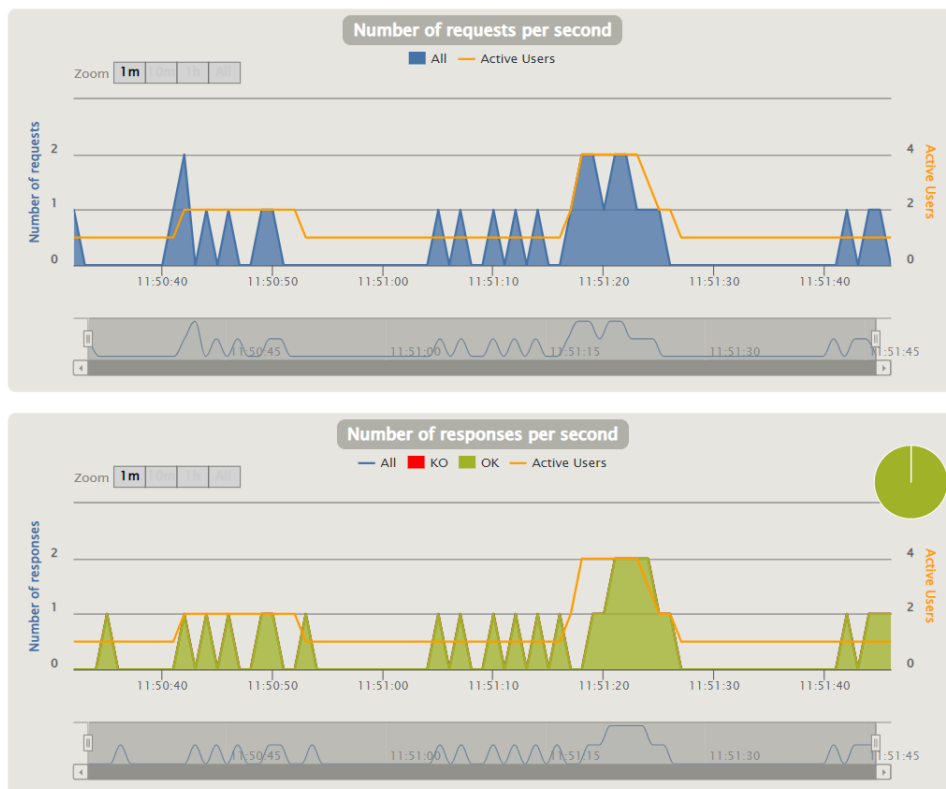
The following three graphs are much more related to making requests throughout the test. In the first one, it groups the response times in percentages, so we can see that approximately 55% (13% + 42%) takes around 1226ms. And the vast majority of requests made in the test (95%) are answered in less than 4000ms.

The following graphs overlay the number of active users throughout the simulation with the number of requests per second and the number of responses per second respectively.

These last two graphs obtain much more value if they are analyzed together, since we can see the delay in the processing of the requests between the requests and the responses.



**Fig. 4.2.4.** Response Time Distribution

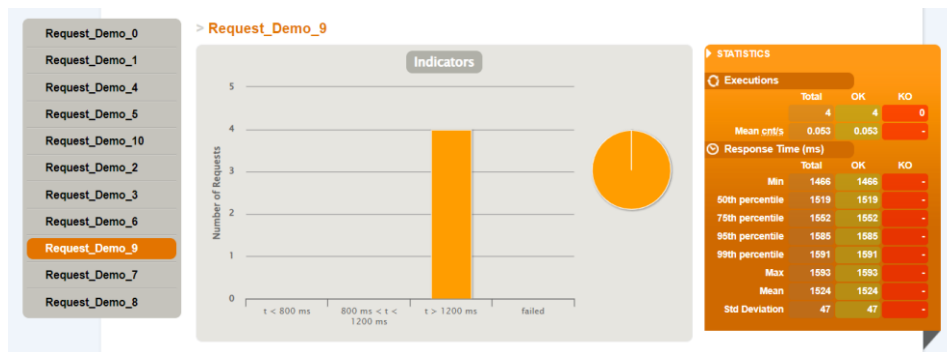


**Fig. 4.2.5.** Number of requests & responses per second

On the other hand, Gatling allows us to take a more specific look and focus on a single request rather than the whole test more generally.

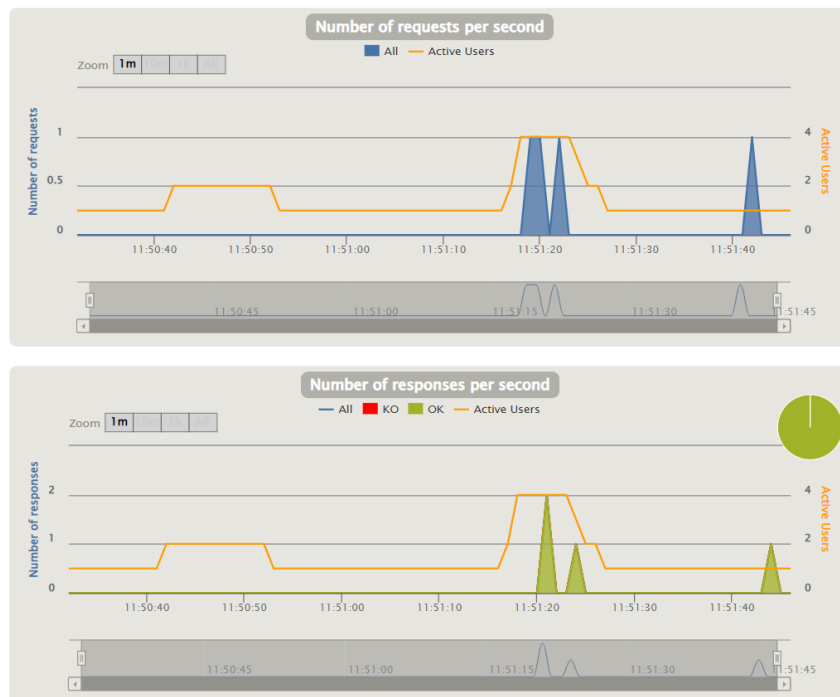
In this way we can observe the times of the requests, their results and how it evolves depending on the time of a particular request.

The following graph is a good example, since it focuses on the request called "Request\_Demo\_9" in which the same request is made 4 times and all are answered with an HTTP 200 (OK). And in turn it is accompanied by the response times of those particular requests.



**Fig. 4.2.6.** Analysis of a particular request

Obviously, we can also see the information related to the requests/responses per second filtered to a specific request.



**Fig. 4.2.7.** Number of requests & responses per second of a particular request



Obviously, we can also carry out analyzes with much more workload for the application, for the following case it has been tested with 101 requests for the API seeking to see how the application behaves in the face of a greater workload. These tests can be done by time instead of by requests, since if you want to keep an application under analysis for two hours it can be configured instead of by X number of requests.

Below we can see the statistical summary of how the test has gone in terms of times.

By making the same request a greater number of times, we obtain more consistent results, since with a small number of requests we can receive inconsistent or error-inducing reports, since if there is an error with the application or with the database data that makes the RTT increase a lot, the average time of that request will be greatly increased if only two requests of that type have been made. On the other hand, if the same request has been executed 10 times and there has been an error, the average time of the request can be corrected with the others and you can continue to obtain a reliable report.

In this way, in this report we can perceive more even numbers with much smaller standard deviations, which indicates that it is a better report.

Requests ^	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	101	101	0	0%	1.086	3716	6634	8730	16422	17013	17623	7249	3827
Request_Demo_0	1	1	0	0%	0.011	9760	9760	9760	9760	9760	9760	9760	0
Request_Demo_1	10	10	0	0%	0.108	6634	8869	9922	10730	11026	11100	8783	1402
Request_Demo_4	10	10	0	0%	0.108	4093	4249	4354	5064	5387	5468	4375	390
Request_Demo_5	10	10	0	0%	0.108	8453	8582	8649	9096	9246	9283	8670	230
Request_Demo_10	10	10	0	0%	0.108	8345	8576	8877	9204	9266	9281	8688	301
Request_Demo_2	10	10	0	0%	0.108	4031	4085	4235	4446	4453	4455	4168	152
Request_Demo_3	10	10	0	0%	0.108	16315	16431	16779	17349	17568	17623	16622	404
Request_Demo_6	10	10	0	0%	0.108	8303	8728	8941	9193	9304	9332	8730	305
Request_Demo_9	10	10	0	0%	0.108	4095	4184	4215	4311	4369	4383	4196	72
Request_Demo_7	10	10	0	0%	0.108	4093	4144	4237	4300	4310	4313	4179	74
Request_Demo_8	10	10	0	0%	0.108	3716	3828	3854	3948	3987	3997	3827	75

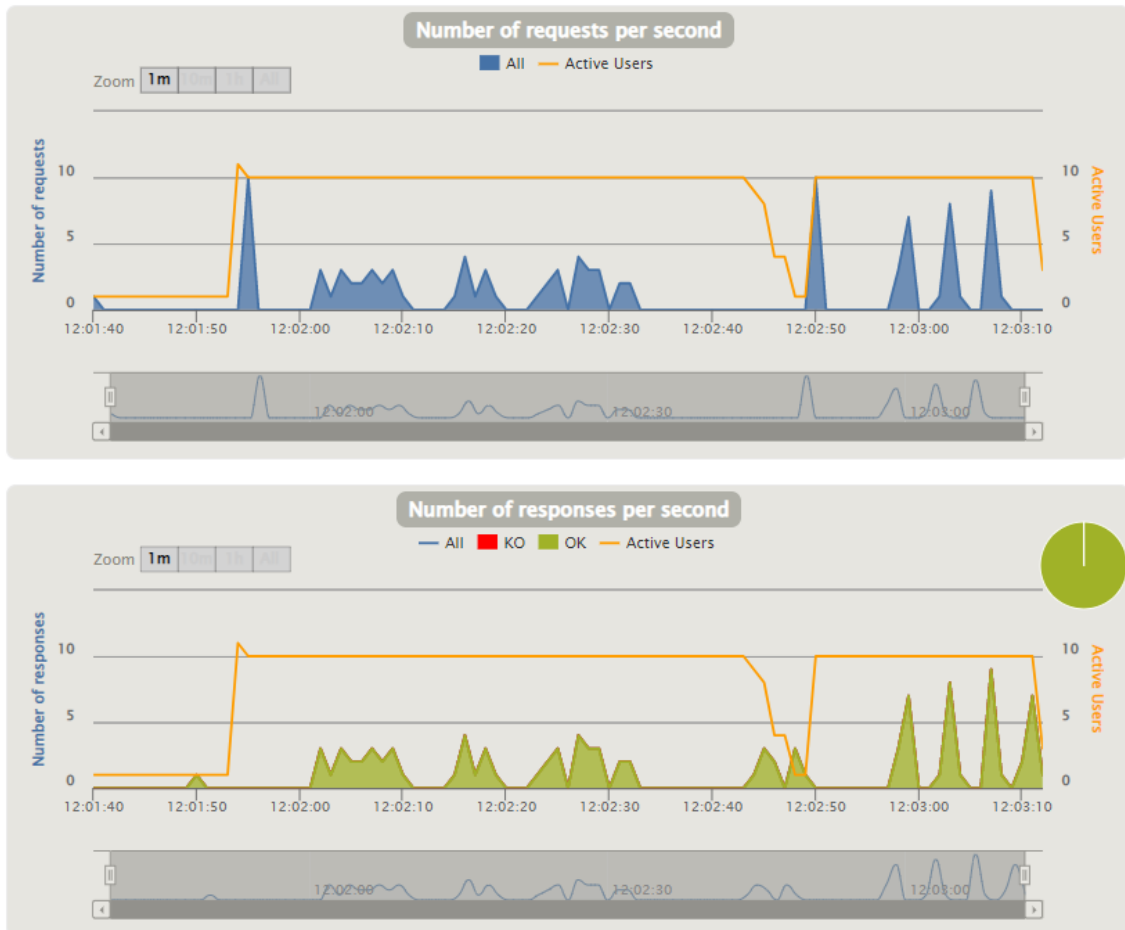
Fig. 4.2.8. 101 Requests test statistics

Even so, we have not collapsed the application, which indicates a good design and architecture of the REST API.

For other cases, it may be interesting to overload a server with HTTP requests to see when it begins to saturate and you start losing requests or receiving timeouts. By doing this you can find limitations in an application or locate where the bottlenecks are in a server to try to remedy them in the future.

In the following graph we have information about the behavior of the server during approximately the minute and a half of testing.

Comparing it with the previous test we can see that the number of active users is higher and that the peak requests per second are also higher. The requests are processed in bursts on the server since the client that simulates Gatling manages them in the same way, so it only receives the request blocks, the REST API processes them and sends the response.



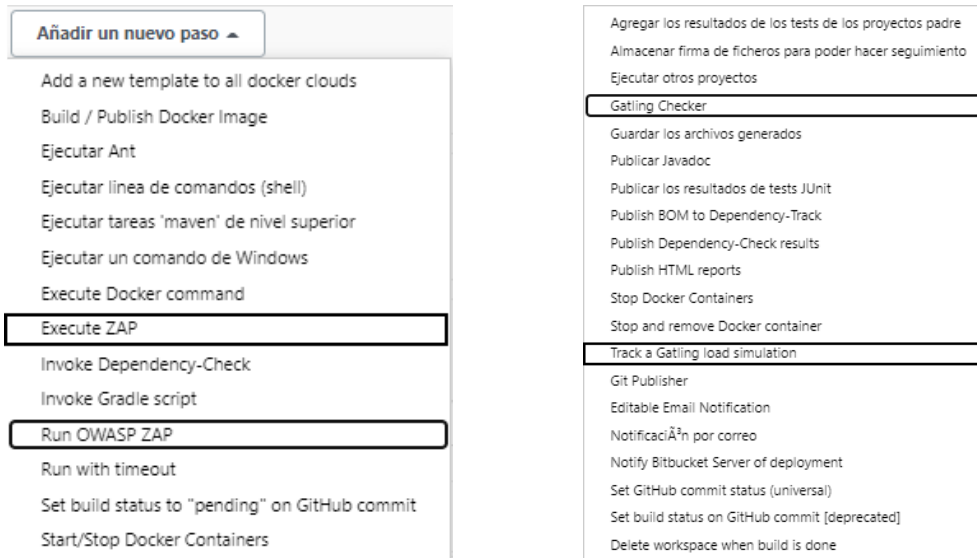
**Fig. 4.2.9.** Requests & responses per second during the test

We see a great similarity between the number of requests per second and the number of responses per second, which is a clear indicator that the server is still far from reaching resource saturation or denial of service.



### 4.3. CI/CD environment

For the integration of the ZAP and Gatling tools in the Jenkins software there is not much complication because the two tools have developed a specific plugin for integration with Jenkins.



**Fig. 4.3.1.** Options to use ZAP & Gatling functionalities

Once downloaded and installed, we just have to go to the job configuration and execute an additional step calling the plugins.

With this step completed, all that remains is to configure the repository from which the code and the jenkinsfile will be downloaded and fill certain fields for the plugins. With this configuration we can execute the job and obtain the results and reports.

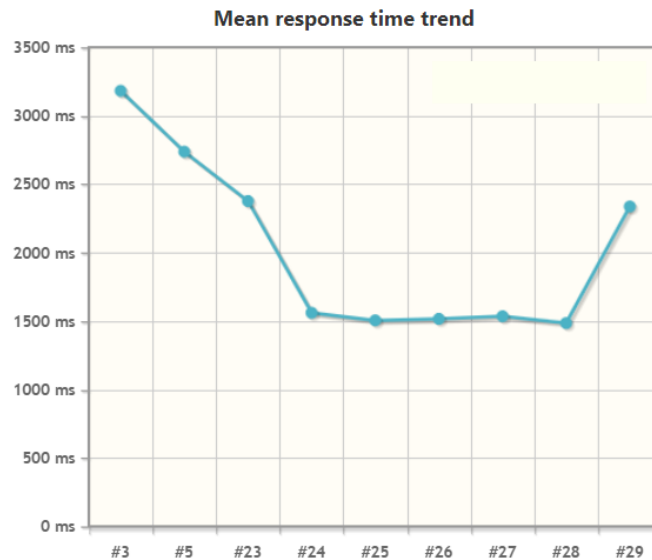
Here we can see the stages of the job, the Declarative is a pre-step that compiles the jenkinsfile in order to find errors and the Build is where the tests are done.

	Declarative: Checkout SCM	Build
Average stage times: (Average full run time: ~53s)	1s	1min 21s
#40 Feb 08 14:13 No Changes	1s	19s
#39 Feb 08 14:10 1 commit	1s	27s
#38 Feb 08 12:33 No Changes	1s	1min 32s

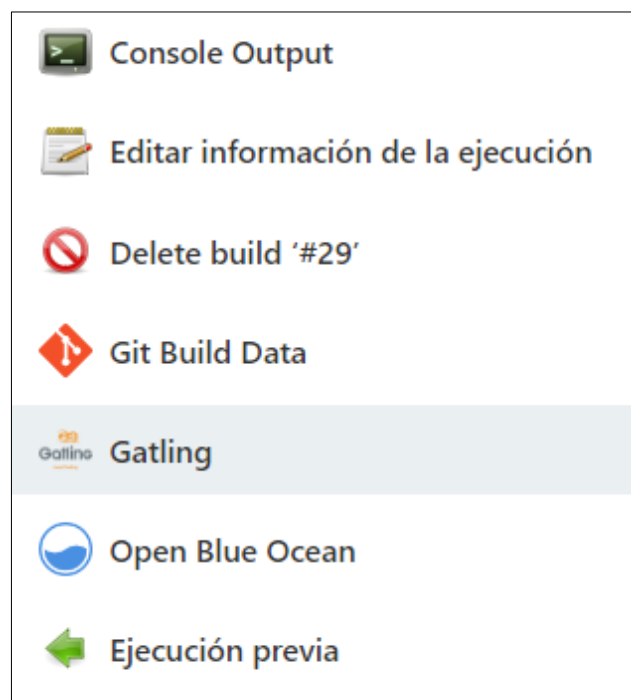
**Fig. 4.3.2.** Stages of the job

In the case of ZAP, we will obtain a downloadable file in which we will find the results, nothing more.

The Gatling plugin has more features, since we can see a graph in the same Jenkins job and also a window for direct download of the reports.



**Fig. 4.3.3.** Time it takes for the job to finish

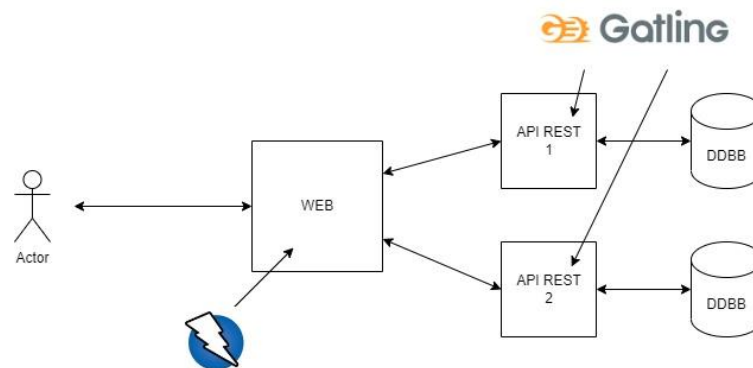


**Fig. 4.3.4.** Window for downloading the report

## 4.4. DEMONSTRATION

In this section we will analyze some results provided by the zap and gatling tools.

It is not a test of a real environment (because for reasons of confidentiality no data from servers or clients of the company can be shown) so it has been simulated a real environment using a frontend and two public servers that do not they are connected to each other.



**Fig. 4.4.1. Environment**

This will not mean any change since the use of the tools is independent, the servers are irrelevant for zap and the use of gatling focuses only on the servers, not on the client to do the behavior tests, so this environment could be a real environment perfectly.

For the vulnerability analysis with the Zap tool, we have used a specialized web page for pen-testing attacks as a target, because these attacks require authorization since they introduce or try to introduce a type of malware in the database.

The page that has been used for scanning is <https://dvwa.co.uk/>, This page is one of the many web pages created with the aim of making people aware of the problems and/or bad practices that developers usually have when programming them, so no special permission is needed to be able to carry out pen-testing attacks for educational purposes.

### Summary of Alerts

Risk Level	Number of Alerts
<a href="#">High</a>	0
<a href="#">Medium</a>	5
<a href="#">Low</a>	2
<a href="#">Informational</a>	1

**Fig. 4.4.2. Summary of alerts**

## Alerts

Name	Risk Level	Number of Instances
Cross-Domain Misconfiguration	Medium	28
CSP: style-src unsafe-inline	Medium	3
CSP: Wildcard Directive	Medium	3
Vulnerable JS Library	Medium	1
X-Frame-Options Header Not Set	Medium	2
Incomplete or No Cache-control and Pragma HTTP Header Set	Low	6
X-Content-Type-Options Header Missing	Low	25
Information Disclosure - Suspicious Comments	Informational	1

**Fig. 4.4.3. Alerts**

As we can see in figure 5.20 in the vulnerability analysis provided by Zap, it is reported that there are 7 different types of alerts[3] classified as medium risk or low risk. Next, we will analyze the different alerts, we will see what they consist of, how they have been introduced and how they can be solved.

In the first place we have the cross-domain misconfiguration alert that appears 28 times throughout the entire scan. This vulnerability permits cross-domain read requests from arbitrary third party domains, using unauthenticated APIs on this domain.

And if we have a look to the description of the alert, we see this:

Medium (Medium)	Cross-Domain Misconfiguration
Description	Web browser data loading may be possible, due to a Cross Origin Resource Sharing (CORS) misconfiguration on the web server

**Fig. 4.4.4. Cross-Domain Misconfiguration description**

This description tells us that there is a problem that could load unwanted data in the browser due to a Cross Origin Resource Sharing (CORS) configuration problem.

CORS is a mechanism based on HTTP headers that is responsible for filtering what type of requests can access the service and which cannot, since it only redirects those requests that meet the requirements established in that header. Thus, allowing to decide whether or not to allow the resource load for requests that have a different origin.

One of the most common bad practices in this type of vulnerability is the following:

*Access-Control-Allow-Origin: \**

By doing this, all requests are being enabled, regardless of their origin, to load resources from a website.

In the case of not knowing how to make a good CORS configuration, it is advisable to avoid allowing access from any source using the \* in the header.

If we do not have the information to make good use of CORS, the ideal is not to configure it, in this way we allow the web browser to enforce the Same Origin

Policy (SOP) in a more restrictive manner, what will provide more security than misconfiguring the Access-Control-Allow-Origin (ACAO) header.

The following two vulnerabilities are related to the content security policy header, a computer security standard that provides an additional layer of protection against Cross-Site Scripting (XSS), clickjacking, and other attacks.

In part they are closely related to each other since both try to mitigate the same type of attacks. The vast majority of these attacks consist of a third person who "modifies" the behavior of a secure website without the latter being aware of it with the aim of stealing information from the victim, who is totally agnostic about what is happening.

Typical attacks that occur when there is a misconfiguration of CSP[17] are XSS aimed at stealing cookies and authenticated sessions and clickjacking that modifies HTML views so that the victim does certain actions on a website while thinking he is doing others.

Medium (Medium)	CSP: style-src unsafe-inline
Description	style-src includes unsafe-inline.

**Fig. 4.4.5.** CSP: style-src unsafe-inline

Medium (Medium)	CSP: Wildcard Directive
Description	<p>The following directives either allow wildcard sources (or ancestors), are not defined, or are overly broadly defined: frame-ancestors, form-action</p> <p>The directive(s): frame-ancestors, form-action are among the directives that do not fallback to default-src, missing/excluding them is the same as allowing anything.</p>

**Fig. 4.4.6.** CSP: Wildcard Directive

To solve these two vulnerabilities related to CSP, it is only necessary to specify the appropriate parameters in the headers according to which origins we want to allow the loading of resources on our web page.

One of the best practices to avoid these vulnerabilities is to be aware of where you put "\*", as this Wildcard Directive enables access to everything below the specified domain.

*Content-Security-Policy: default-src \*://\*.example.com*

This would allow access to subdomains of example.com (but not example.com itself).

Keep in mind that not only can you restrict the origin in a general way, but you can also specify the origin of images, scripts, objects...

In addition, these attacks not only serve to steal information from third parties but can also be used to sabotage the use of a web page itself, for example they can be used to change the color of the web page, modify its text so that it says

something offensive that may annoy an audience or use scripts that redirect you to another web page.

The vulnerability that we are going to talk about next is one of the simplest vulnerabilities to solve and at the same time the most dangerous for a web page.

Medium (Medium)	Vulnerable JS Library
Description	The identified library jquery, version 1.6.2.min is vulnerable.

**Fig. 4.4.7.** Vulnerable JS Library

The most popular JavaScript libraries[18] tend to be used by a large number of people which means that they are often heavily audited, although bugs are quickly recognized and fixed, resulting in a constant stream of security updates.

Due to this, having outdated libraries without security patches can be fatal for a web page since this makes it very vulnerable.

The good thing about this type of vulnerability is that it can be quickly solved by updating the versions of the libraries for safe versions and removing libraries that are not used.

The following risk is related to clickjacking attacks and occurs due to not setting the X-Frame-Options[20] HTTP header.

Medium (Medium)	X-Frame-Options Header Not Set
Description	X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.

**Fig. 4.4.8.** X-Frame-Options Header Not Set

This header is responsible for deciding when to render and how a web page is rendered. Two parameters can be specified for this header, one is DENY[21] and the other is SAMEORIGIN[21].

The DENY parameter denies taking the render regardless of the origin and the SAMEORIGIN option only renders if the origin is the same.

An example of a clickjacking attack that could be solved using this header is one in which the victim accesses a website that appears to be secure through a link, but in reality, the attacker's website is embedded within that website. In this way, the victim thinks that she is clicking on safe places but she is suffering from a clickjacking attack.

The next two are considered low risk. This is so because the risk or problems that this type of vulnerability may cause are much lower compared to the vulnerabilities we have seen so far.

The first of these risks is “Incomplete or No Cache-control and Pragma HTTP Header Set”[19] which consists of Specify which cookies you do not want to be cached since it is very likely that there is sensitive data that you do not want to save.

Low (Medium)	Incomplete or No Cache-control and Pragma HTTP Header Set
Description	The cache-control and pragma HTTP header have not been set properly or are missing allowing the browser and proxies to cache content.

**Fig. 4.4.9.** Incomplete or No Cache-control and Pragma HTTP Header Set

And the other one is “X-Content-Type-Options Header Missing”, it happens because this header is not specified and therefore there is a risk of suffering an attack, although this type of problem is already obsolete because in 2014 browsers establish this header by default.

Low (Medium)	X-Content-Type-Options Header Missing
Description	The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy versions of Firefox will use the declared content type (if one is set), rather than performing MIME-sniffing.

**Fig. 4.4.10.** X-Content-Type-Options Header Missing

Once we have analyzed the entire report that the Zap tool has provided us, we can see that despite being a web page with a large number of instances of different vulnerabilities, it is a page in which its security can be increased. simple and fast way.

The vast majority of vulnerabilities that we find today are produced due to the misuse of HTTP headers or bad practices when programming. The solution is usually quite simple and applying it covers the holes through which cyberattacks sneak in and how much damage they do.

However, as it is a world that is constantly changing, the speed at which new ways to attack web domains are found is very high.

This results in security tasks not being punctual, but rather maintenance tasks and constant analysis, since what was safe yesterday may not be tomorrow.

Once the vulnerability analysis of the web page has been carried out, it is now time to analyze the 2 REST APIs that exist in this simulated environment in figure 4.4.1.1.

To carry out this analysis, several tests have been done, modifying the configuration of the scenarios to try to push the servers to the limit so that we can see how they behave with a certain workload up to the point where one of the two servers collapses.

As we can see later, the APIs have no relation to each other and the data they manage has nothing to do with each other, this is irrelevant for us since what we want to see is how they behave when faced with a certain number of requests per second. As you can see later, the REST API 1 manages comments and posts from a domain and has seven different requests (both get post delete) and the REST API 2 only returns random data about animals, in this case cats.

Below we will see four different analyses in which the first two analyze the REST API 1 and the next two analyze the two servers at the same time.

#### 4.4.1. API REST 1 +1350 req.

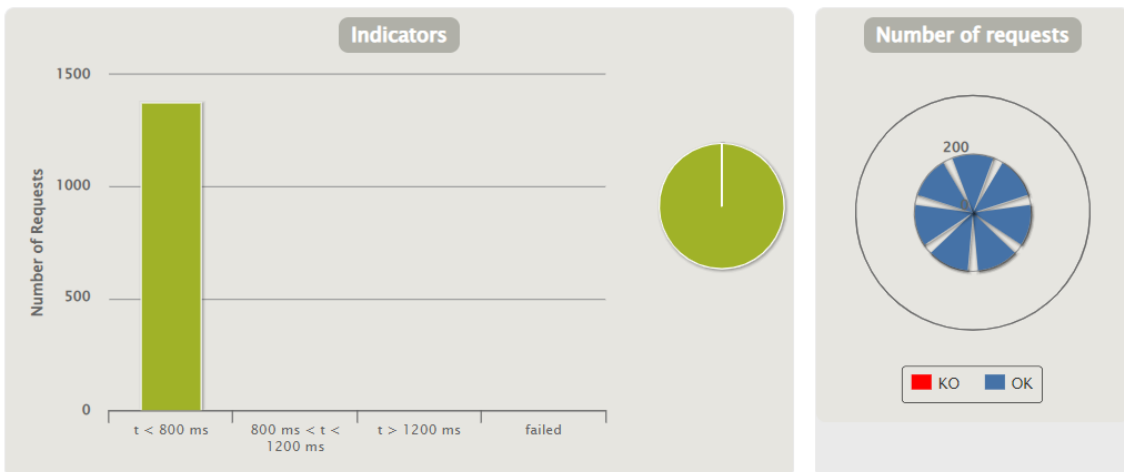


Fig. 4.4.1.1. Initial graphics of test 1

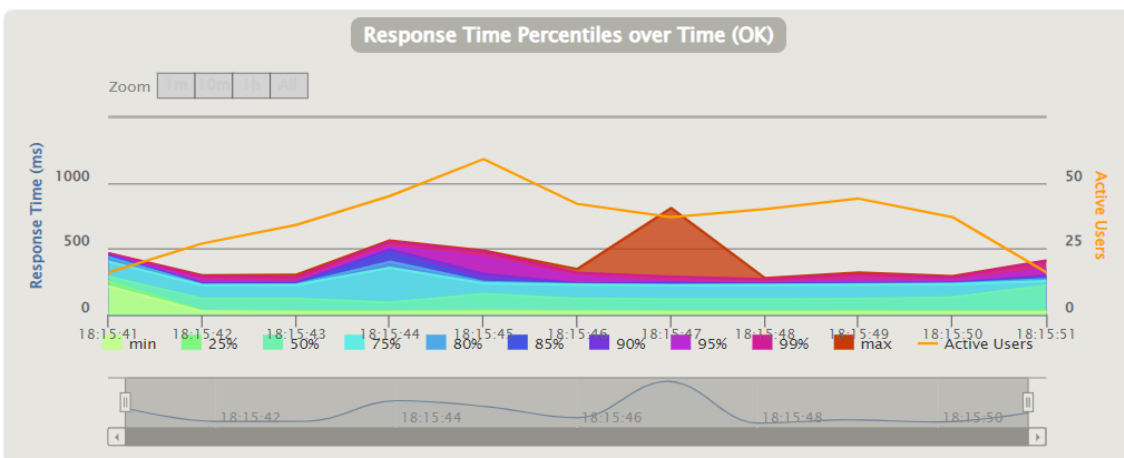
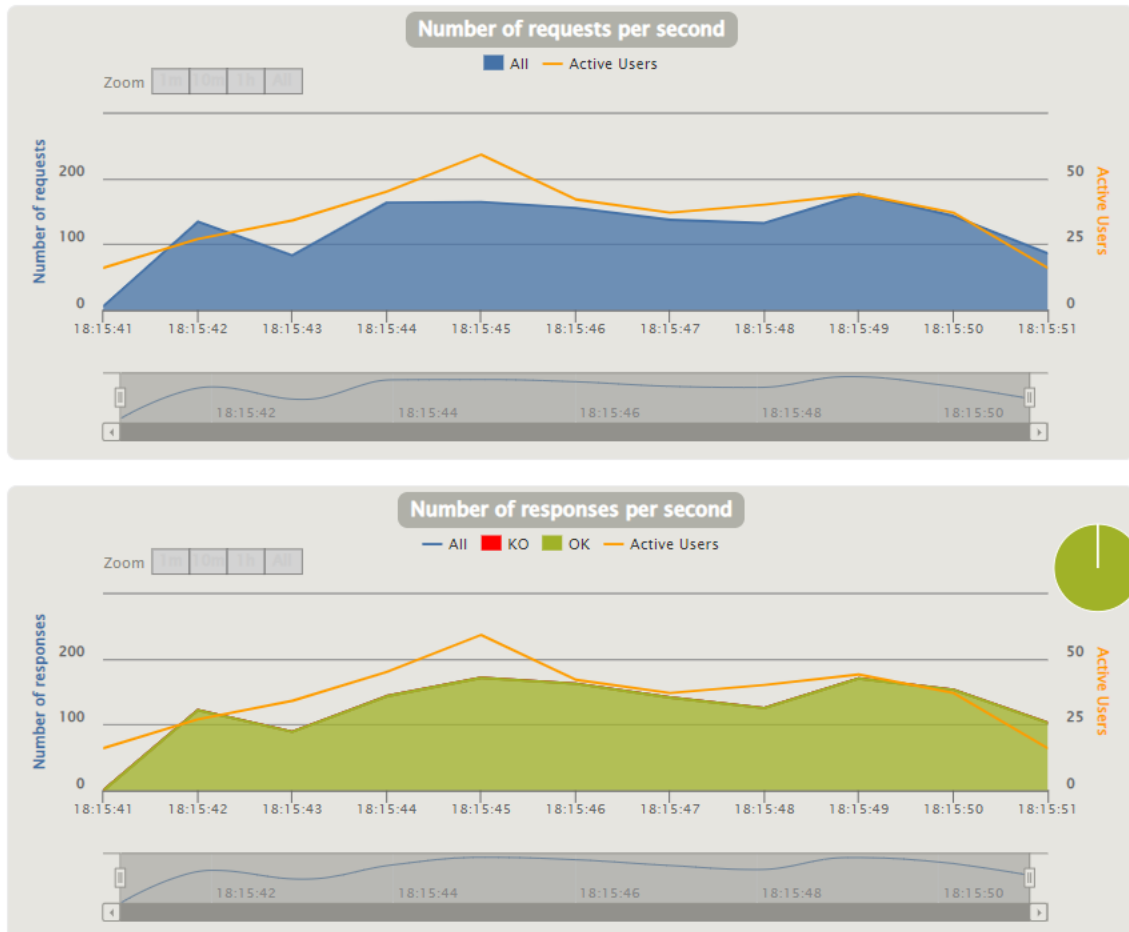


Fig. 4.4.1.2. Response Time Percentiles over Time (OK) test 1



If we look closely at the graphs as much as figure 4.4.1.2. like the one in figure 4.4.1.3. we can deduce that this test is relatively far from collapsing backend 1 since all the requests that are made (1379) are answered correctly and none of them gives any timeout, also if we pay more attention to the last graph, we can see that the time frame in which the responses take the maximum time is quite small since the triangle is almost equilateral.



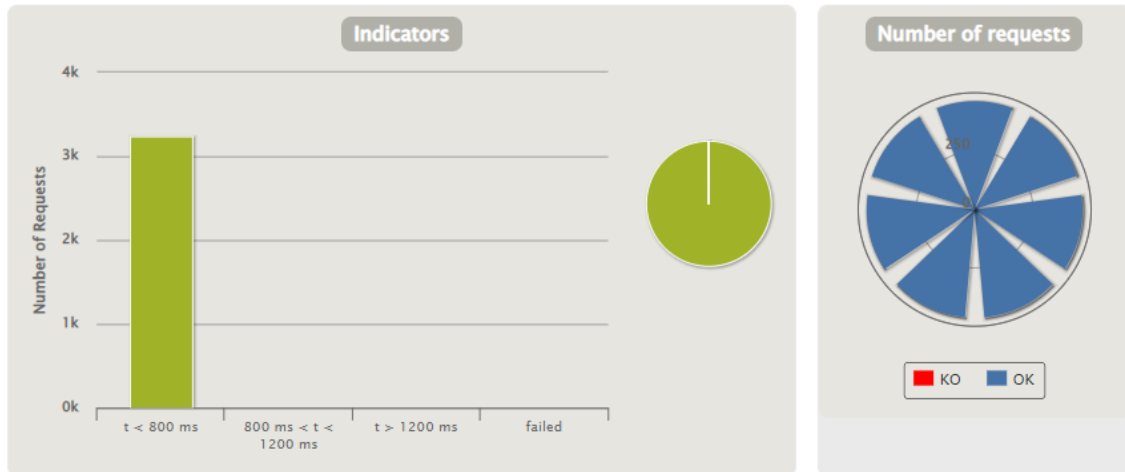
**Fig. 4.4.1.3.** Number of requests/responses per second test 1

Another piece of information that sheds much light on this issue is the fact that both the number of requests per second and the number of responses per second have a very similar shape over time, which means that the delay of each request is very short and it does not come close to causing a bottleneck, which indicates that the server works quickly and efficiently.

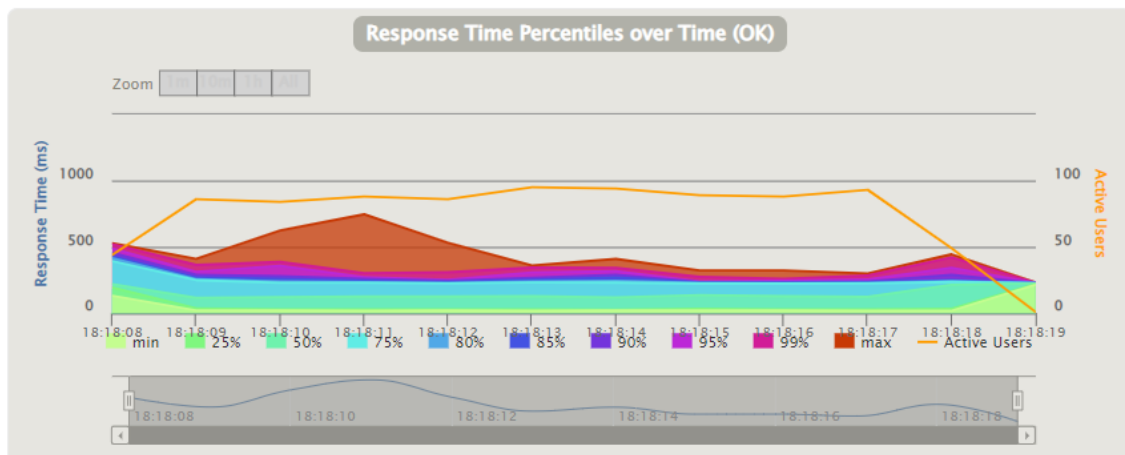
In this first analysis, virtual users are introduced in a linear manner until reaching a maximum of 55 simultaneous users making requests, once this peak is reached, it begins to reduce until it reaches zero.

#### 4.4.2. API REST 1 +3000 req.

Due to the fact that after the first test it has been seen that server 1 has been very far from reaching a saturation state despite having a considerable number of requests, a second test was carried out, increasing the peak of simultaneous virtual users, trying so to find any bottleneck in the API.



**Fig. 4.4.2.1.** Initial graphics test 2



**Fig. 4.4.2.2.** Response Time Percentiles over Time (OK) test 2

Even so, it can be seen in figure 4.4.2.2. that the system continues to act very efficiently even though it has close to 100 simultaneous users throughout the load test, making more than 3000 requests.

4.4.3. BOTH APIs WITH LOW AMOUNT OF WORK

In this first joint test of the two servers, it began with the injection of 5 virtual users who each made a request of each type, which represents 40 requests throughout the entire performance test (seven requests to server 1 and 5 requests to server 2).

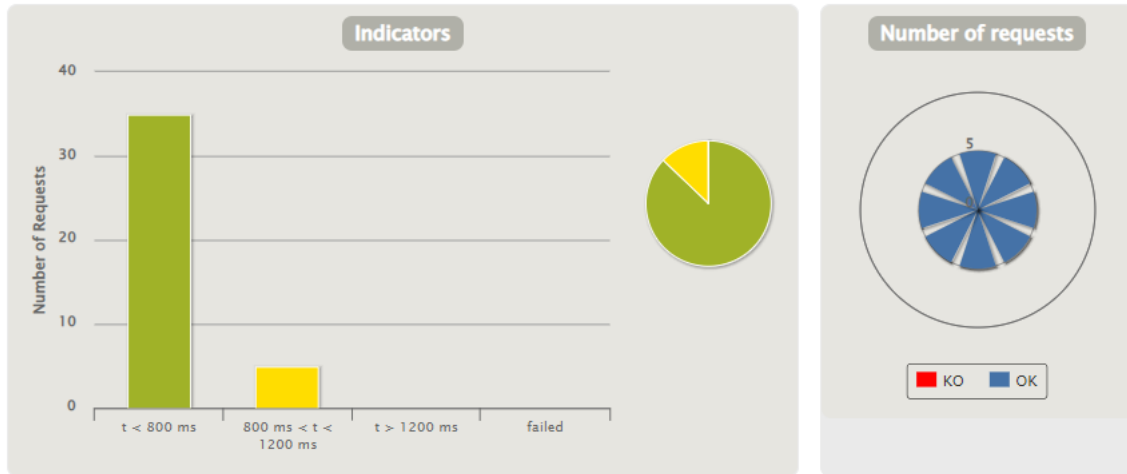


Fig. 4.4.3.1. Initial graphics test 3

A rather curious fact that should make us suspect that something is not going as well as in the previous tests is the fact that there are many fewer requests in the test and even so, 5 requests have had a response time of between 800 ms and 1200 ms.

Due to this, a very successful idea is to look at the requests that are directed to the REST API 2 individually.

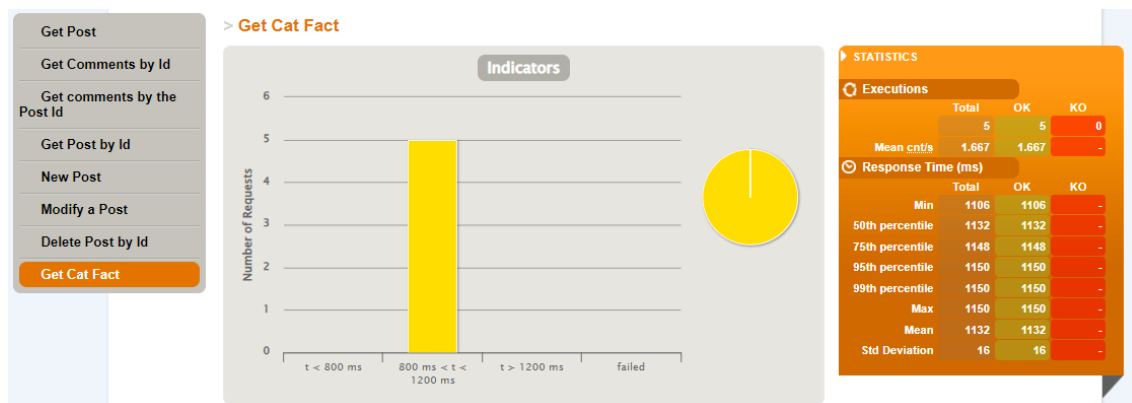
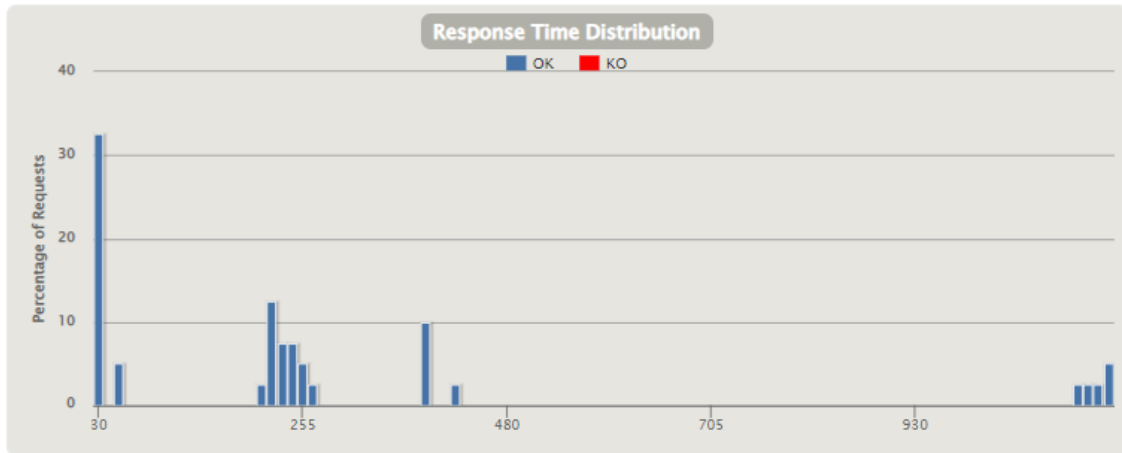


Fig. 4.4.3.2. Get Cat Fact Requests

In this way, analyzing figure 4.4.3.2. we can confirm that between the two servers, API REST 2 is the one with much longer response times and therefore where there may be a bottleneck.



**Fig. 4.4.3.3.** Response Time Distribution test 3

Another indication that you can see an imbalance between the behavior of the two REST APIs when facing the same workload is the distance between the response times, which we can see in figure 4.4.3.3.

4.4.4. BOTH APIs WITH BIG AMOUNT OF WORK

After performing the last test and having seen a possible bottleneck on server 2, the same test is repeated, but introducing a much higher workload and increasing the number of virtual users in the test in order to saturate the application.

This test consists of 98 virtual users making requests to both servers concurrently until reaching a total number of 1481 requests throughout the entire test.

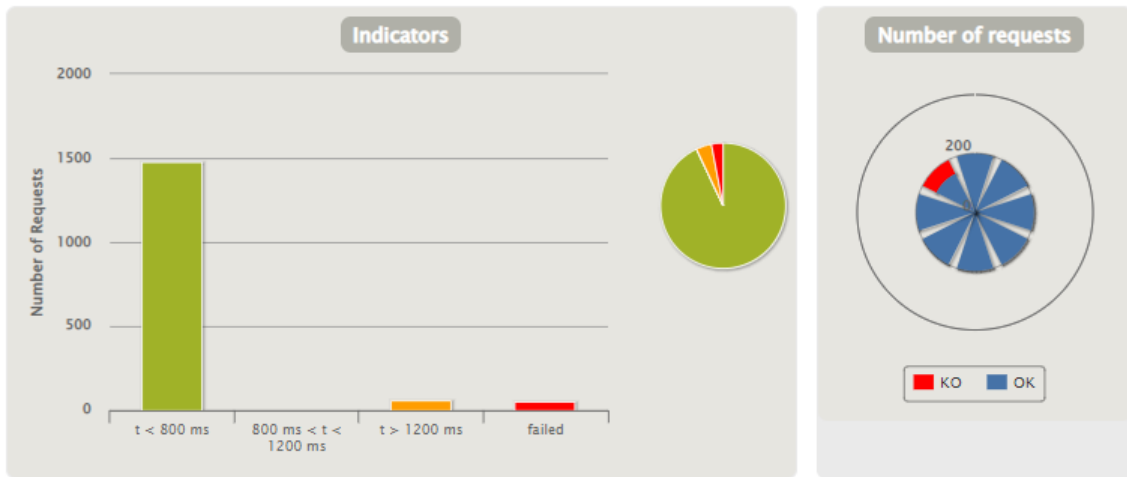


Fig. 4.4.4.1. Initial graphics test 4

Once the results are obtained, if we pay attention to the graph in figure 4.4.1.1, we can see that throughout this test there have been requests that have not been OK.

Now it's time to detect if you are requests that have been KO are they concentrated on a single server or are they on both REST APIs.

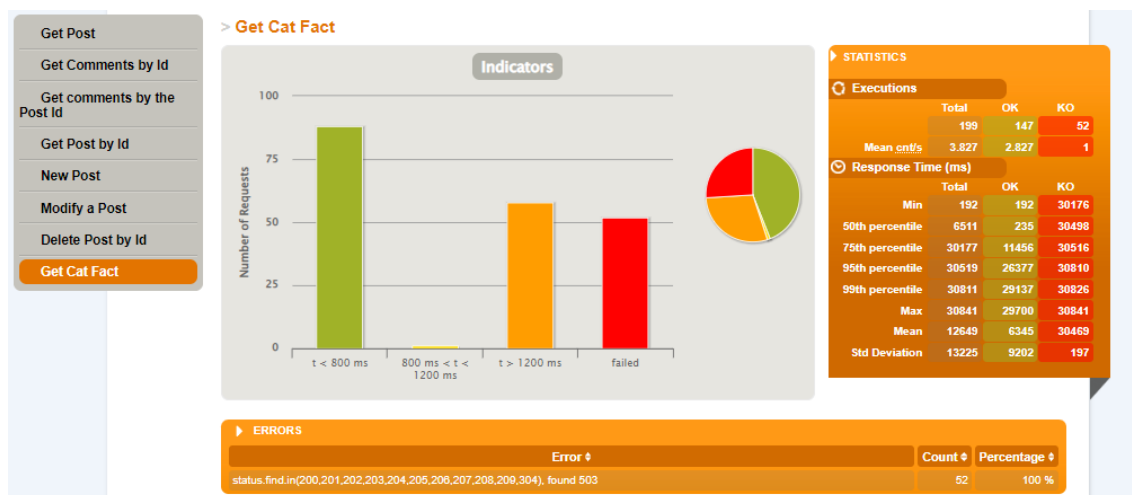
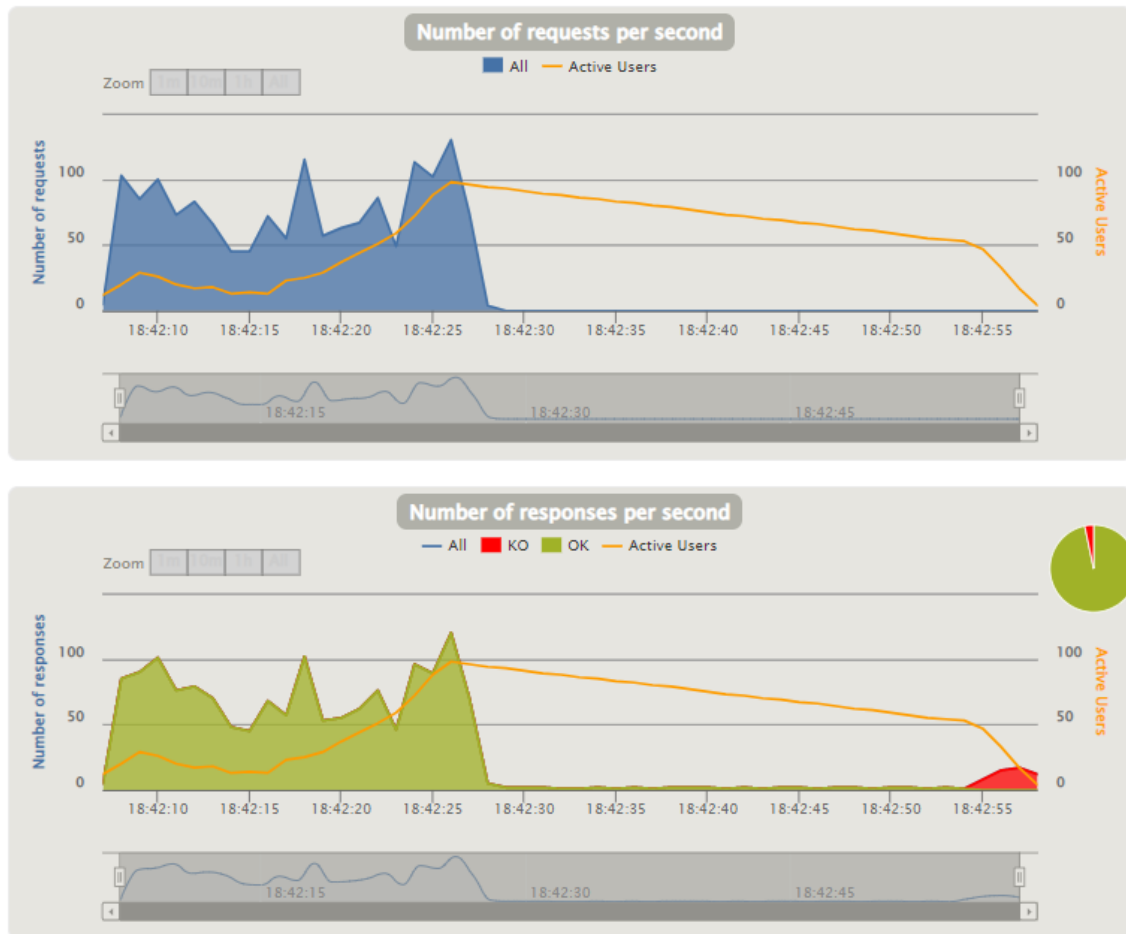


Fig. 4.4.4.2. Get Cat Fact Requests test 4

The easiest way to identify if they are concentrated in a single REST API is to check the "get cat fact" request, since it is the only one made to server 2 and if there are no errors, it means that they are distributed on server 1. If we take a look at the graph in figure 4.4.2.2., we can see that all the errors are concentrated on server 2 since there are the 52 erroneous requests, therefore we have our bottleneck.



**Fig. 4.4.4.3.** Number of requests/responses per second test 4

The graph in figure 4.4.2.3. perfectly shows the behavior of the API REST 2 during the load test, starting gradually until reaching a saturation point in which all the requests have been made but not all the responses have arrived. Since the answers give a timeout after approximately 30 seconds and therefore, they are KO.

## CHAPTER 5. CONCLUSIONS & THOUGHTS

In this final degree project, I have worked with many more tools than I thought when I chose the job. It is true that it takes extra work but the process has been very beneficial since I have learned about many software that I had not used before and I have even delved into a new language that I did not know.

Personally, I consider that I have fulfilled both the objectives that the company proposed to me and those that I set for myself.

Going into more detail, the mentioned goals are:

- The learning and use of software to perform "penetration testing" or pen-testing" in order to find possible vulnerabilities in user interfaces.
- Acquire knowledge about the types of vulnerabilities and risks in Web applications.
- The use and learning of tools oriented to perform work/stress load in order to find limitations.
- Learn what a CI/CD environment consists of, how to use it, and how to integrate both ZAP and Gatling in an environment like Jenkins.
- Use a framework oriented to project management to organize all the phases of my TFG and have a record of the work done.

Once the work is done and with much more knowledge on the subject, I consider that I could add more objectives achieved that I did not even contemplate at the time I opted for this work.

This is due to the number of "stones" that I have been finding along the way, and that through hours of research and help from colleagues, have ended up becoming additional knowledge that I did not think of at the beginning.

Some of these resulting problems that I just mentioned can be:

- Use and configuration of a PROXY
- Linux CLI
- Advanced use of docker and docker-compose
- New programming language (ScaLa)

Making a little criticism about my work and how I have approached it, there are many things that I would change, but above all I would modify the organization of the TFG. I am not referring to being organized over time, since I have been distributing the work over time, but rather to the fact that I would have done more research tasks.

When I started the work, the first thing I did was read the documentation of the tools to know what they did and the different ways in which they could be implemented, and once this was done, I got down to work. The problem came later because, due to certain limitations of the tool or for other reasons, I could not implement the tool in the way I had initially chosen and I had to get rid of the work done.

After carrying out this project I have realized how important the design and architecture phase is. And although in my case it wasn't throwing away more than a day's work, I've learned by not skimping on the design phase, maybe you'll free yourself from real problems in the future.

Going a little deeper into technical aspects, the ZAP tool has seemed to me to be a super complete tool, with a large amount of documentation that helps the learning curve not to be very vertical and very prepared for automation environments. On the other hand, its use has seemed very "boring" and not very customizable, since we can only modify parameters of the tracking and scanning forms, and unless we do a manual scan, we will not be able to really squeeze a web to find vulnerabilities, which for a CI/CD environment is not optimal.

On the other hand, we have Gatling, which despite the fact that at first it was not the part that excited me the most, it is the one that I have ended up enjoying the most.

Gatling allows us to customize the requests the way we want, we can "record" the requests we send by making Gatling a proxy that is before the API to replicate them later, or we can create a project with the requests, the protocols and the scenarios we want to test.

It uses a very versatile language that adapts to protocols, libraries, and since it is quite similar to Java, it supports practically the same thing.

It provides super complete results, and its version for HTML is very attractive and functional. With very useful graphs that give us an idea of what happens at all times of our analysis.

It is true that for more concise analyses, without user concurrence and with less test variability, other software does the same thing more quickly and easily. But when we want a complete analysis simulating multi-user Gatling is the best decision.

For these reasons I consider Gatling to be one of the best options (if not the best) for these types of tests.

Jenkins is a super powerful software that accepts all kinds of plugins to expand its functionality and that gives us different options to do what the user wants. It is very useful, and although the learning curve at first is a bit scary, it is worth getting into that world.

There is not much more to say about this tool, for these reasons many of its competitors do not even come close to being rivals for it.

In general terms I think I have carried out a good project, it is true that the fact of not being able to show information about what I have implemented in the company has made my work difficult and has made it more extensive, but even so I believe that a work that deepens in all the aspects in which it should deepen.



Leaving aside the three main tools that have been used during this project and which were the main objectives that I work on. Other types of tools and ways of working more oriented to time and project management have also been used, which should be mentioned.

The first one is called SCRUM and it is not a tool, it is a way of working that is widely adopted over time. It consists of working in intervals of between two weeks and one month called sprints, and the constant analysis of these sprints with the aim of finding things to improve that can be put into practice during the next time interval.

Within this framework, roles are designated within a group of no more than 10 people. There are three roles generally, a Product Owner (PO), a Scrum Master (SM), and Developers. Very briefly, a Product Owner oversees ensuring that value is added to a product, a Scrum Master oversees the proper functioning of the team in the AGILE environment and the developers add value to the product through implementations.

This has not been strictly fulfilled because the work was carried out by only one person. Even so, the main objective of constant criticism with short-term objectives is still applicable.

This has made it easier to find things to improve and optimize the way of working.

Also, another tool named JIRA has been used during this project. This tool is an Atlassian functionality which allows us to have a visual interface in which we can create tasks, group them within the same, change their status according to the implementation level among many things.

It is a very powerful tool that allows the user to graphically see what state of the implementation it is in, how many things remain to be done and to be able to run an organization in a simpler way.

And finally, a couple of softwares that are very related, Sourcetree and Bitbucket.

These two tools are often used together and linked because Bitbucket is software for creating and managing Git-based repositories and Sourcetree is a visual interface for Git repositories. This greatly facilitates the maintenance and management of projects, since the management of this through the command line can become tedious and confusing.

## BIBLIOGRAPHY

- [1] ZAP developer guide (2022). Extracted from: <https://www.zaproxy.org/docs/developer/>
- [2] ZAP docker user guide (2022). Extracted from: <https://www.zaproxy.org/docs/docker/about/>
- [3] OWASP Top 10 Vulnerabilities (2021). Extracted from: <https://www.zaproxy.org/docs/guides/zapping-the-top-10-2021/>
- [4] Gatling Introduction (2022). Extracted from: <https://gatling.io/docs/gatling/tutorials/quickstart/>
- [5] Documentation of Scala (2019). Extracted from: [https://www.tutorialspoint.com/scala/scala\\_functions.htm](https://www.tutorialspoint.com/scala/scala_functions.htm)
- [6] Gatling cheatsheet (Unknown). Extracted from: <https://worldline.github.io/gatling-cheatsheet/>
- [7] Gatling Advanced Tutorial (2022). Extracted from: <https://gatling.io/docs/gatling/tutorials/advanced/>
- [8] Information about storing data in an existing session (2021). Extracted from: <https://stackoverflow.com/questions/69438476/gatling-issue-saving-data-to-session>
- [9] Documentation of Scala (2022). Extracted from: <https://docs.scala-lang.org/>
- [10] Scala information (2022). Extracted from: <https://docs.scala-lang.org/getting-started/index.html>
- [11] Information of Akka (2022). Extracted from: <https://akka.io/docs/>
- [12] CI/CD information (Unknown). Extracted from: <https://about.gitlab.com/topics/ci-cd/>
- [13] Jenkinsfile information (2022). Extracted from: <https://www.jenkins.io/doc/book/pipeline/jenkinsfile/>
- [14] Pipeline Documentation (2022). Extracted from: <https://www.jenkins.io/doc/book/pipeline/getting-started/>
- [15] List of Jenkins plugins (2022). Extracted from: <https://plugins.jenkins.io/>
- [16] Jenkins general information (2022). Extracted from: <https://www.jenkins.io/doc/book/>
- [17] CSP Vulnerability information (Unknown). Extracted from: <https://www.invicti.com/blog/web-security/content-security-policy/>
- [18] Outdated Libraries Vulnerability information (Unknown). Extracted from: <https://beaglesecurity.com/blog/vulnerability/vulnerable-javascript-library.html>
- [19] Data Session Steal Information (2022). Extracted from: [https://cheatsheetseries.owasp.org/cheatsheets/Session\\_Management\\_Cheat\\_Sheet.html#web-content-caching](https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#web-content-caching)
- [20] X-Frame-Options header bad configuration (2022). Extracted from: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>
- [21] Clickjacking attack information (Unknown). Extracted from: <https://www.imperva.com/learn/application-security/clickjacking>