Utilize Docker container and Selenium Grid technology to improve resource efficiency in the runtime of a Software Deployment & Testing Framework (SDTF).

[Document subtitle]

VASCO Data Security

[Course title]

# Preface

# Summary

# Table of contents

[Preface 1](#_Toc515242547)

[Summary 1](#_Toc515242548)

[Table of contents 2](#_Toc515242549)

[List of figures, graphs 4](#_Toc515242550)

[Part I - Introduction 5](#_Toc515242551)

[Introduction 6](#_Toc515242552)

[Problem statement 7](#_Toc515242553)

[Objectives 8](#_Toc515242554)

[Part II – Context & Technologies 9](#_Toc515242555)

[Context 10](#_Toc515242556)

[Software Deployment and Testing Framework 10](#_Toc515242557)

[About the webadmin suite 11](#_Toc515242558)

[Current situation 12](#_Toc515242559)

[Technologies 13](#_Toc515242560)

[Selenium 13](#_Toc515242561)

[Selenium grid 13](#_Toc515242562)

[Docker 14](#_Toc515242563)

[What is a container? 14](#_Toc515242564)

[Why containers? 15](#_Toc515242565)

[Part III - Implementation 17](#_Toc515242566)

[Solution 18](#_Toc515242567)

[Deployment architecture 18](#_Toc515242568)

[Proposed Deployment architecture 18](#_Toc515242569)

[The containers 18](#_Toc515242570)

[Impact on the existing WebAdmin 27](#_Toc515242571)

[Changes needed to the WebAdmin Suite 27](#_Toc515242572)

[Adapt to new architecture 27](#_Toc515242573)

[Adapt for parallelization 30](#_Toc515242574)

[Challenges 36](#_Toc515242575)

[IE in a container? 36](#_Toc515242576)

[Issues with the IE node 36](#_Toc515242577)

[Part IV - Conclusion 38](#_Toc515242578)

[Performance difference 39](#_Toc515242579)

[Conclusion 40](#_Toc515242580)

[Part V – Sources & Appendix 41](#_Toc515242581)

[Sources 42](#_Toc515242582)

[Appendix 43](#_Toc515242583)

[IAS on vCloud 43](#_Toc515242584)

[Dockerfiles 45](#_Toc515242585)

[For use with the SSH tunnel 45](#_Toc515242586)

[Install script for SDTF on linux container 46](#_Toc515242587)

[Plain diff: Differentiating input data per browser for cases\_user\_create 48](#_Toc515242588)

# List of figures, graphs

Part I - Introduction

# Introduction

VASCO Data Security is a company mainly active in cyber security, mostly known for two factor authentication services and electronic signature software. This thesis is written in context of Quality Assurance (QA). The QA team utilizes a Software Deployment and Testing Framework (SDTF) to practice system testing on the different products that are being developed.

One of the products, the IDENTIKEY Authentication Server (IAS), is a centralized platform designed to deliver authentication lifecycle management via a single integrated system, supporting a multitude of other technologies and VASCO products. This server is managed via a web based platform, “IAS Web Administration”. This product is in continuous development and thus, needs to be tested frequently, which is why the QA team has a set of tests in place that automates testing the web application, supported by the SDTF.

The focus of this thesis is on how to the environment in which these tests happen more efficient. This is important for a number of things. The most important factor being a smoother development cycle. The QA team is a crucial factor in the software lifecycle. Each time development teams engineer a software product, or an update to an existing product, this product is thoroughly tested by QA. Good QA testing takes time, of course, and this is an important factor in the time it takes from the inception of a project, to its successful rollout. So reducing the runtime of automated tests will mean that projects will be finished in a more timely manner. A second advantage of faster QA testing is that it gives faster feedback to the developers, thus decreasing the amount of time between development cycles.

Another implication is cost-efficiency. An environment that allocates its resources in an efficient manner is cheaper to maintain. Furthermore, it would allow for more tests to run concurrently using the same resources.

# Problem statement

Today the SDTF runs on virtual Windows machines in vCloud, with all the dependencies installed, as well as all the third party software needed for some tests. The IAS is also deployed in the same way; on a VM on vCloud, with all the supporting services necessary, such as a database and the IAS Web Administration service.

However, this deployment strategy is very resource intensive, as with each test you want to run you need to boot up a Test control host (TCH), as well as a machine running IAS. This is made evident by the fact that vCloud often gets saturated by the amount of machines that are running on it.

Currently, for each test suite that one wants to run, it is necessary to start up a new TCH with all the dependencies and third party software on it, needed to run the SDTF. This has a lot of negative implications with regards to resource allocation.

~~<Include execution time and resources>~~

Too technical for introduction. Moved to context.

# Objectives

The goal of this research project is twofold:

The primary goal is to shorten the time it takes for the QA team to give feedback to development to shorten the development cycles. This is achieved most like by reducing the runtime of the tests in the Software Development and Testing Framework. The focus of how this is achieved is going to be on the WebAdmin test suite. As the conclusions made for this test suite are assumed to also be applicable to other test suites.

The WebAdmin suite is designed run against different browsers such as Google Chrome, Firefox and Internet Explorer. In the current situation, these test are run sequentially for each browser. For example, first Firefox then Chrome. The idea that is going to be explored is whether these suites can be run in parallel, as this would greatly decrease the runtime. Another question that comes up when talking about parallelization is if the suites can be run against the same environment at the same time? And what changes need to be made to the test suites to be able to achieve these goals. This ties in with our second point.

The second objective is to research how to reduce the resource usage of these tests. This increases the overall efficiency of the environment, and allows for more tests to be run concurrently on the same platform.

A path that is going to be explored is a new deploy strategy for the test control hosts, as they now each require a virtual machine to be started up, with all the requirements to run all the tests in the SDTF.

Part II – Context & Technologies

# Context

This chapter introduces the environment in which this study is situated.

## Software Deployment and Testing Framework

The SDTF is a system testing framework written in Python and based on the python unit test library “PyUnit”. The purpose of the framework is to automate the system testing of an application or service, and see if the output matches the expected output for certain inputs. The SDTF automates this process and sends back feedback to a test management system, in this case TestLink.

This thesis is going to focus on improving the runtime of the SDTF for the WebAdmin test suites. However, the end goal, should this concept be taken into production, is to expand this solution beyond one test suite, to all test suites included in the SDTF.

Other test suites in the SDTF include

* IDENTIKEY administration server
  + Build intake
  + Installation
  + Generic authentication
  + Radius authentication
  + Soap authentication
  + Tcl administration
  + Validation
  + *Webadmin*
* Digipass
* Dpscreader
* TID
* Native bridge

The SDTF runs on a test control host (TCH) running Windows

## About the webadmin suite

The webadmin test suite is a collection of 235 test cases divided between 20 test suites:

1. Logout
2. Logon and Privileges
3. Create User
4. Import Users
5. List Users
6. Search User, Find Menu
7. Search User Find/Manage Menu
8. Manage Admin Privileges
9. Manage Dashboard
10. Manage User Account
11. Manage User Attributes
12. Manage User Info
13. Manage Virtual DIGIPASS
14. Manage Assigned DIGIPASS
15. Import DIGIPASS
16. Search DIGIPASS, Find Menu
17. Search DIGIPASS, Find/Manage Menu
18. Manage DIGIPASS
19. Assign DIGIPASS
20. Manage DIGIPASS Application

Each test suite consists of a base case, of which the other cases in the suites are variations, aside from a few exceptions in each suite.

## Current situation

\\srv-be-file.vasco.com\UserProfiles$\willaar1\Documents\Stuff voor thesis\thesisDocs\Presentaties\prentjes\Poging2000.emf

Currently, for each test suite that one wants to run, it is necessary to start up a new TCH with all the dependencies and third party software on it, needed to run the SDTF. This includes (but is not limited to) Selenium and all the browsers that one wants to run the tests on. The issue with regards to efficiency here is that for each browser test suite you want to run, you need to start this new VM with this whole package on it, but, the limiting factor is often just the Selenium and the browser. This is because when Selenium is used locally (here meaning: on the same machine as the SDTF), it is impossible to start another test suite because the Selenium service is already in use.

A way to remedy this is to separate the Selenium service from the machine running SDTF. This way, when an additional test suite needs to run, it is only necessary to launch an extra machine that provides a browser and Selenium. This would mean that resources would be allocated in a much more efficient manner, because it is no longer necessary to spin up a new TCH each time one wants to run a new suite.

# Technologies

Discussing the technologies utilized to work towards the end goal: a more efficient environment

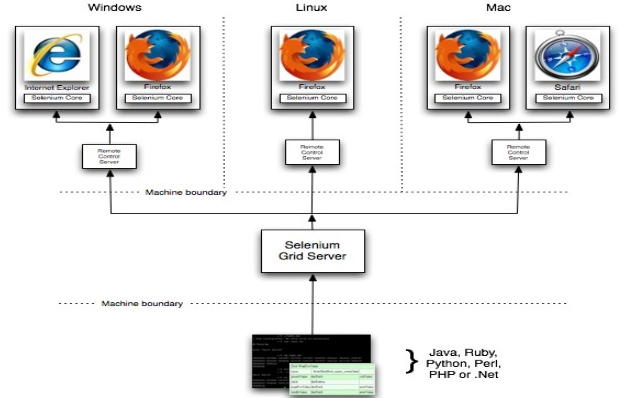
## Selenium

Selenium is a portable software testing framework for web applications. Selenium provides a record/playback tool for authoring tests without learning a test scripting language (Selenium IDE). It also provides a test domain-specific language (Selenese) to write tests in a number of popular programming languages, including C#, Java, Groovy, Perl, PHP, Python and Ruby.

The tests can then be run against most modern web browsers. Selenium deploys on Windows, Linux, and Macintosh platforms. It can be used for unit, regression, smoke, integration and acceptance testing.

In this specific test suite, we make use of Selenium Webdriver. This service accepts commands, sends them to the relevant browser through a browser-specific browser driver, and then retrieves the results. Most browser drivers launch and access the browser application directly.

### Selenium grid

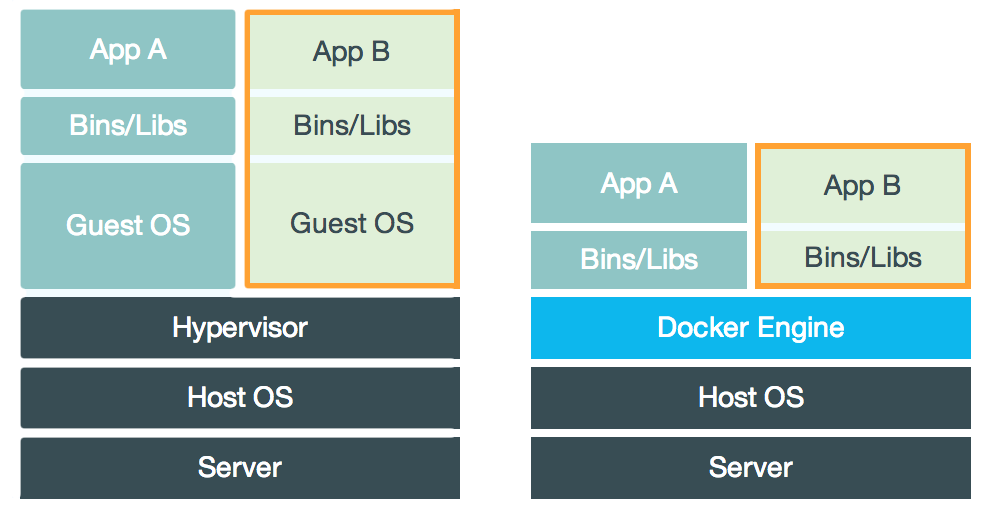
Selenium Grid is a framework that allows you to run tests on different machines against different browsers in parallel. I.e.: running multiple tests concurrently against potentially different machines running different browsers and operating systems. In short, it allows you to run tests in a distributed test execution environment.

The grid consists of one or more worker machines called nodes and a single “master” machine called a hub. Someone who uses this service for testing would send a test to the hub, alongside with information on which browser, and if applicable, which operating systems to run the test on. The hub knows the configuration of each node that is register at the hub, and selects the node with the correct attributes. Once a node has been selected, Selenium commands initiated by the test are sent to the hub, which passes them through to the node assigned to that test. On this node, a browser is opened and executes the selenium commands within that browser against the application under test.

## Docker

### What is a container?

Maybe make an original diagram comparing a vm and a container



The best way to explain what a container is, is probably comparing it to a virtual machine (VM for short). While containers and VMs are two different technologies, they can be used for a similar purpose: the isolation of an application and the relevant dependencies in a self-contained unit.

Furthermore, utilizing VMs or containers this way can remove the need for physical hardware, and can result in a more efficient use of resources

#### Virtual Machines

At its core, a virtual machine is a complete emulation of all the hardware components that make up a computer that runs an operating system, executes commands and runs applications, as if it “were a physical” vs. “would be an actual” computer. A VM, or multiple VMs, can runs on a single, or distributed set of machine(s) through a hypervisor. A hypervisor can run either on the operating system of a host machine (a hosted hypervisor) or directly on the hardware (native or bare-metal hypervisor)

The hypervisor itself runs on a physical system called a “host machine”. The host machine provides the VMs with the necessary resources, like RAM and CPU. These resources can be divided amongst the VMs to best suite your use-case.

A defining feature of a VM is that they come with a complete virtualized hardware environment, which includes a CPU, RAM, storage, network adapters, graphical interfaces, and more.

#### Containers

Unlike a VM which provides hardware virtualization, a container provides virtualization on the operating system level. It does this by abstracting the user space.

A container is similar to a VM in quite a couple of ways. For instance, they have a private process tree, their own network interface and IP address, can execute commands as root and mount filesystems.

The big difference with virtual machines is that containers package up just the user space, and not the kernel or virtual hardware like a VM does. A container shares the host system’s kernel with other containers. But because every container gets its own user space, it is possible to run multiple containers on a single host.

### Why containers?

While virtual machines are more robust than containers, there are a couple of reasons why it makes more sense to use containers.

#### Image size

In contrast to VMs, Docker images tend to be very small, a couple hundred megabytes, maybe even tens. While images for VMs tends to be upwards of twenty gigabytes. This is because contrary to a container, there is way more overhead involved, such as the operating system, hardware, dependencies, and so on; while a container only contains a minimal operating system and the application you want to run on it, as well as any dependencies it might have.

However, this difference is going to be less drastic in this implementation, because the application we are running is quite large, and needs a good few dependencies. Nevertheless, the size of the image that runs SDTF should not exceed three gigabytes, which is still a significant decrease.

#### Flexibility

Another great boon for containers is their start-up speed. Once you have built a container image, you can start up a container fairly quickly, in a matter of seconds even. Starting up a VM takes way longer than that, and could even take a couple of minutes. This time gain is probably negligible in the bigger picture of this project, as it is only necessary to start up the container once before you run the tests, and then it runs for a couple hours.

Utilizing Docker Compose, it is possible to bundle multiple container images together in a service. Doing this provides a couple of interesting features, but more on that later.

The most relevant use case of Docker Compose is that it allows you to bundle the containers into one service for easy set up and tear down of the testing infrastructure. By defining the full environment in a Compose file, you can create and destroy these environments in just a few commands:

$ docker-compose up (this starts up the service)

<run the tests>

$ docker-compose down (this destroys the service)

This is way more convenient then setting up the different containers to run individually.

#### Resource usage

Because a Docker container does not need to simulate any hardware, and has a more minimal version of an operating system, it uses way less processing power and RAM to function. This results in a meaningful decrease in resource overhead that is going to be decisive for increasing the efficiency of the SDTF.

#### Summary

Considering all the above it makes sense to move from the current model to a model that uses Docker container to instantiate the services required for unit-testing the Webadmin portion of the Identikey Authentication Server.

Part III - Implementation

# Solution

Walkthrough of the steps taken to accomplish the research goal

## Deployment architecture

As discussed in a previous chapter, there is a theoretical advantage to using Docker containers when it comes to performance and resource usage when compared to virtual machines. Mostly due to their lightweight nature, both in computing resources as image size.

In this chapter, it is explored if we can leverage this advantage to benefit the resource usage of the environment

### Proposed Deployment architecture

\\srv-be-file.vasco.com\UserProfiles$\willaar1\Documents\Stuff voor thesis\thesisDocs\Presentaties\prentjes\Poging1000.emf

This figure is a concise representation of what we are trying to achieve. In an ideal scenario, only one “TCH” is running with only SDTF installed on it, Using Selenium Grid for all its browser automation needs. An additional milestone that is set for the sake of resource efficiency, is running as many test as possible (preferably all tests), against a single IAS environment, instead of a dedicated IAS for each test suite.

### The containers

#### Container with SDTF as TCH

The current deployment strategy of launching a Test Control Hosts as virtual machines is very resource intensive. When looking into how to increase resource efficiency, containerization was the first thing to come up.

For smaller applications or services, containerization has proven its effectiveness mainly by reducing image size and computing resource requirements such as RAM and CPU usage. However, the expected gains in our use case probably won’t be as spectacular as Docker claims to be for some other services or apps where image size can be reduced to a couple hundred MB’s, or even tens of MB’s. In an extreme example, it is possible to build a container that prints hello world with an image size under 500 bytes. (see <https://blog.hypriot.com/post/build-smallest-possible-docker-image/> ).

In our case however, we are already limited by the size of the Software Development and Testing Framework. The disk usage of a full install is approximately 1.5 GB’s.

To run a container you first need an image to run your container from. An image is built by running the following command: (source: <https://docs.docker.com/engine/reference/commandline/build/> )

docker build [OPTIONS] PATH | URL | -

When applied to our case it becomes

docker build -t sdtf/sdtf\_n .

The –t option stands for “tag”, which is used to name your container in the name:tag format, and the path we specify is “.” Which signifies the current working directory. The path needs to point to the directory containing the Dockerfile.

When this command is executed, it passes the directory specified with the PATH parameter (or URL) to the Docker daemon as “build context”. The Docker daemon then interprets the Dockerfile to build an image.

The Dockerfile is kind of the recipe for making the image, as it specifies the build process of the image. The build process can refer to any file within the build context. If you use an URL instead of a PATH, it can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts and plain text files. However, in this application of the technology we always use PATH.

Below the Dockerfile used to build the image for a container with the SDTF installed. There is, however, more set-up required before we get to the final image, but more on that later.

Dockerfile syntax reference: <https://docs.docker.com/engine/reference/builder/>

FROM python:2.7

ADD sdtf /root/sdtf

RUN apt-get update && apt-get install -y \

python-tk \

python-pip \

python-ldap \

&& rm -rf /var/lib/apt/lists/\*

RUN apt-get update --fix-missing

RUN pip install virtualenv

RUN chmod +x /root/sdtf/install\_sdtf\_linux.sh

RUN /root/sdtf/install\_sdtf\_linux.sh

ENV LMPW=\*\*\*\*\*\*\*\* # *bare password, redacted*

CMD [ "/bin/bash" ]

The FROM instruction initializes a new build stage and sets the base image as a starting point for further instructions. A valid Dockerfile must start with a FROM instruction.

Source for base image: <https://hub.docker.com/_/python/>

The python base image is very suited to our use case, because it is based on a minimal Linux install, with python 2.7 pre-installed.

The ADD instruction copies files or directories from the context to the specified destination on the container. The paths of the source is interpreted relative to the source of the context of the build. When using ADD, the source can also be a remote URL. When adding directories and files, it should suffice to use COPY instead. (*note to self: maybe I should use copy instead of add)*

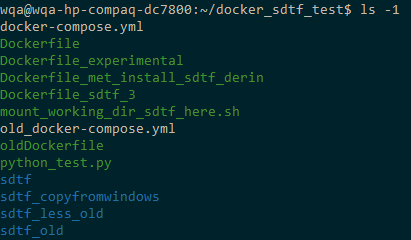
Here the ADD instruction adds all the content the sdtf directory to the container at /root/sdtf. The sdtf directory contains all the necessary files to install the SDTF on the container, pulled from this git repository: <https://git-global.vasco.com/wqa/sdtf/tree/linux-sdtf-port>

The RUN instruction will execute any commands in the shell in a new layer on top of the current image and commit the results. The resulting image is then used for the next step in the Dockerfile. In the shell form, a backslash (“\”) may be used to continue a single RUN instruction onto the next line.

In this example the RUN instruction is used to install prerequisites for the SDTF install, change the permissions of the install script, and finally, to run the install script.

The ENV instruction an environment variable to a specified value. Here it is used to load a password into a variable used in a command to start a test in SDTF.

The main purpose of the CMD command is to provide defaults for an executing container. It’s different from RUN as RUN actually runs a command and commits the result. CMD offers a default operation when a container is run without a defined command parameter.

Important to note is that the Docker build command passes the whole directory as build context to the Docker daemon. This may not be desirable behavior, as in my case, I had many other things in the build directory as shown on the screenshot below.

1 – screenshot: contents build directory

The more context is passed to the Docker daemon, the longer the build time gets. To reduce the delay, a .dockerignore file can be used to limit the amount of files passed. Adding files and directories to the .dockerignore file skips causes them to be skipped when doing a Docker build.

This is what my .dockerignore file looks like:

sdtf/doc

sdtf\_less\_old

sdtf\_old

sdtf/\*untime

sdtf/\*-activate

sdtf/\*-activate.bat

sdtf/\*-deactivate.bat

sdtf/specs

sdtf/suites/digipass

sdtf/support

sdtf/test

sdtf/webadmin

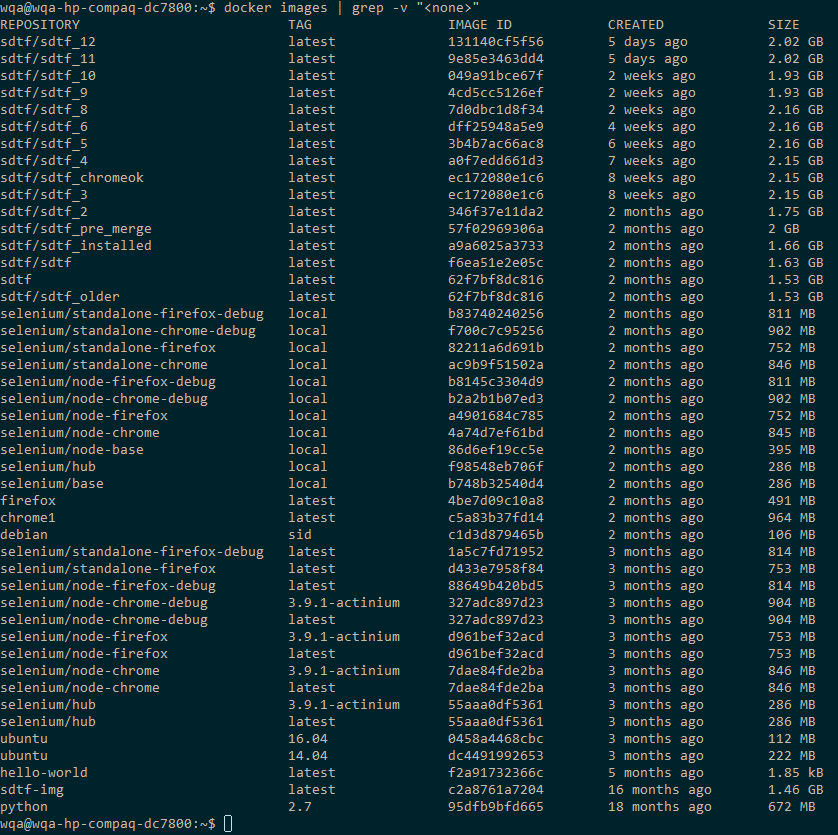
sdtf\_copyfromwindows/

The .dockerignore needs to be in the same directory as the Dockerfile to work.

After the Docker build runs successfully, an image with the specified name (defined with the tag flag) is made. It is then possible to view your images using

docker images

Doing so results in a table that looks like this



To then start up a container, the following method is used: (<https://docs.docker.com/engine/reference/run/>)

docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

Which for our use case looks like this:

docker run -t -d --name ContainerName sdtf/sdtf\_n /bin/bash

The option –d stand for detached; this allows the container to run in the background, instead of occupying the command line interface as a “foreground process”. This isn’t necessary, but useful in our case as we often want multiple containers running at the same time.

The purpose of –t is to allocate a pseudo-tty to the container. It is not that well documented on the Docker reference page, but what it does for us is send the output to a virtual tty within the Docker container. Practically this ensures that the stdin stays open, even when the container is not attached. If this option isn’t used, the container starts, runs the command defined by either the Dockerfile or the command parameter, and then just stops again. So in our case it is necessary to keep the container running so that we can attach it later using Docker exec and execute our tests.

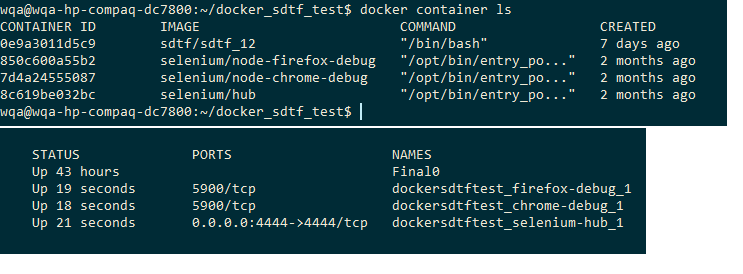
The --name option is there to give it a nice name. If left blank it will generate a default name in the format of “adjective\_noun” (e.g.: “awesome\_goldwasser”).

The command argument is meant to override the default CMD configured in the Dockerfile. This could as well be left empty because I pass the same command as the one configured in the Dockerfile, so it’s a bit redundant.

After the command is run successfully, it creates a container. You can check if the container was created successfully by running

docker container ls

This displays the following:



The top container in this example is the result of the command

docker run -t -d --name Final0 sdtf/sdtf\_12 /bin/bash

So after all that, you’re left with a running container with SDTF on it. There is however one last thing that needs to be ironed out before tests can be run successfully in this environment. Due to outdated packages in the SDTF resources used in the install procedure, you are left with an outdated selenium installed in the runtime. To resolve this, it is necessary to attach the container, enter the SDTF runtime and update selenium. Here’s how to go about that: (or does this belong in the appendix?)

To get to the final image, it is appropriate to first make an intermediary container, in this example named “Intermediary0”

docker run -t -d --name Intermediary0 sdtf/sdtf\_11 /bin/bash

Then Enter the container using Docker exec

(source: <https://docs.docker.com/engine/reference/commandline/exec/> )

docker exec -it Intermediary0 bash

The Docker exec command stands for Docker execute. This executes the specified command (here “bash”) in the requested container (here “Intermediary0”) with the possibility for omption. The options specified here will allocate a “pseudo-tty” to the container and keep STDIN open. What dis does practically is attach the command line interface (CLI) of the container to the CLI of the host system (or whatever command line you are accessing it from).

Doing this allows us to do additional configuration to the container before we commit a final image.The only configuration step the we have left here is update the selenium package in the SDTF runtime.

First thing that needs to happen is enter the sdtf runtime using the following command:

source /root/sdtf/sdtf-activate

Include screenshot

Then enter the following to update selenium:

pip install -U selenium

When the install is finished, you can exit the runtime by typing

deactivate

Then exit the container by typing simply by giving the exit command.

Next up, we use a powerful feature of Docker to create our final image: the Docker commit command:

docker commit Intermediary0 sdtf/sdtf\_12

This creates an image based of “Intermediary0” named sdtf\_12, which we then use to create our final containers with (see above)

*Result? Resource gains?*

Docker stats <container name>, find a way to put this into a file to get decent insight of the resource usage.

#### Selenium Grid

*Why?*

In essence, it is actually necessary to use Selenium grid and deploy it using a Docker based strategy? Let’s go over the facts.

In this new deployment strategy, assuming we deploy the SDTF in containers to substitute the TCH’s, the SDTF containers will not contain anything else that isn’t 100% required. As such, there will be no browsers installed on it, making it impossible to do the browser operations locally in the container.

Lucky for us, selenium grid resolves this issue, providing a single point for all our remote selenium calls.

The reason why the Selenium Hub and its Nodes are deployed in containers is a matter of flexibility. This method comes with its own set of problems, as described later on (see: “Challenges: IE in a container?”).

One upside of this is that we gain a good amount of flexibility. Deploying the Selenium nodes as containers greatly reduces the time it takes to provide additional nodes, as these containers tend to start op very quickly, and, if handled correctly, are registered to the hub and added to the grid almost instantly after they are created. Scaling back down and terminating nodes is also easy and rather fast.

Utilizing containers for this task also has the benefit of being way less resource intensive then spinning up entire virtual machines for this purpose, due to the lightweight nature of containers, as mentioned earlier.

*How?*

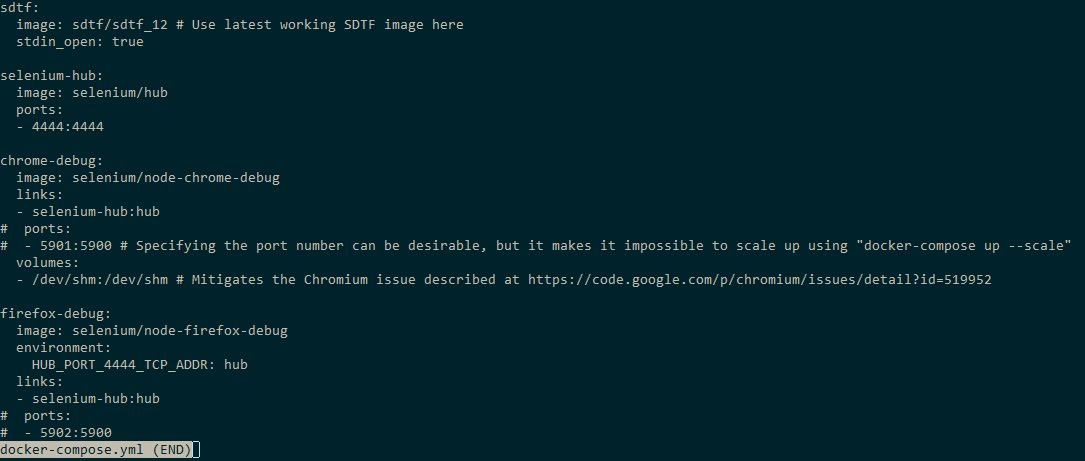
The images necessary for deploying a Selenium Grid in Docker can be found in the GitHub repository of SeleniumHQ here: <https://github.com/SeleniumHQ/docker-selenium>. (TODO: install guide of the images in appendix)

For the deployment of the containers, it is possible to deploy them all individually using Docker run. However, there is a more flexible option available that is ideally suited for this particular use case where it is desirable to launch multiple containers that work together as part of a “service”. This option is called Docker-compose.

##### Docker-Compose (explain this in the technologies section, or is it ok to introduce here?)

Docker-Compose is a

Examine this docker-compose.yml (called a compose file) file:



This is the file used to start up the Selenium grid. In it are 3 sub services specified: Selenium-hub, chrome-debug and firefox-debug. It is important to note that port 4444 of the selenium hub is linked to port 4444 of the host machine. This causes all incoming traffic on port 4444 on the host machine to be forwarded to the container, effectively making it addressable from outside.

Using the following command in the directory where the compose file is located will launch one container of each “service”:

docker-compose up -d

-d is used to run it detached.

The flexibility this provides is that when you need an additional node with chrome or firefox, you can just run the following:

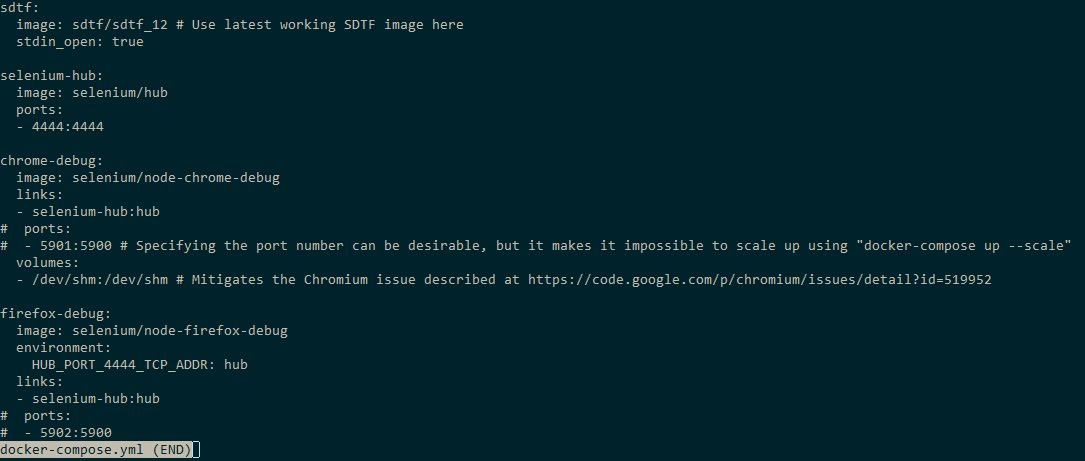
docker-compose up –d --scale chrome-debug=2

or

docker-compose up –d --scale firefox-debug=2

To scale the number of containers of a service to the specified number. Here, 2

It may also be beneficial to also incorporate the sdtf in the compose file in the final iteration when taking this to production like so:



I opted not to do this during my testing however, as frequently restarting a container seems to kind of throw off Docker-compose, for lack of better term, causing a variety of issues, once even crashing my system. That being said, if you are nice to Docker-compose and follow the best practices found on the reference page (<https://docs.docker.com/compose/reference/>), it is overall a stable and useful tool to easily start up multiple containers and scale them accordingly to your needs.

## Impact on the existing WebAdmin

*Not sure what to put here except that it doesn’t work unless adapted to the new deployment architecture. Which is the following subtitle*

*Each suite is multiple test cases*

*Base test case*

*Setup teardown*

*High level explain the changes necessary*

## Changes needed to the WebAdmin Suite

### Adapt to new architecture

To adapt to the new architecture, there need to be some changes. Mainly because the old code assumes that selenium and the necessary browsers are installed on the test control host, on which the SDTF runs. So all instances that make Selenium calls need to be changed so that the selenium call isn’t made locally, but is directed to the grid.

At the start of the test suite run, there is a part where the script logs in to WebAdmin to fetch the version of the WebAdmin. This is specified in framework/ias/administration.py. It was written in the style with multiple “try” statements made to try launching a browser in two different ways across either a windows or a Linux. As there are no browsers on the container with SDTF, I rewrote that part to make use of the Selenium Grid. Now it remotely starts a browsers to do the required operations there.

# Create the webdriver

if self.webDriver is None:

sel\_grid = SeleniumGrid('10.132.224.165') #ip of the selenium hub

browser\_options = DefaultBrowserOptions.get\_remote\_defaults(Browser.CHROME)

browser = sel\_grid.create\_browser(browser\_options)

browser.start()

self.webDriver = browser.webdriver

webDriver = self.webDriver

This needs the following imports at the top of the file to work:

from framework.ias.webadmin.webpage\_components import (

get\_browser\_by\_name,

DefaultBrowserOptions,

SeleniumGrid,

Browser

)

The following lines were removed:

from selenium import webdriver

from selenium.webdriver.firefox.firefox\_binary import FirefoxBinary

binary = FirefoxBinary('C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe')

try:

\_sdtflogging.sdtfLogger.debug("Trying to start Firefox webdriver")

self.webDriver = webdriver.Firefox(firefox\_binary=binary)

except Exception as ex1:

\_sdtflogging.sdtfLogger.debug("Failed to start Firefox webdriver {}".format(ex1))

if self.webDriver is None:

try:

\_sdtflogging.sdtfLogger.debug("Trying to start Chrome webdriver")

self.webDriver = \_make\_chrome\_webdriver(max\_retry=2, retry\_delay=5)

except Exception as ex2:

\_sdtflogging.sdtfLogger.debug("Failed to start Chrome webdriver {}".format(ex2))

if self.webDriver is None:

\_sdtflogging.sdtfLogger.debug("Prepare virtual display on linux")

from pyvirtualdisplay import Display

display = Display(visible=0, size=(800, 600))

display.start()

try:

\_sdtflogging.sdtfLogger.debug("Trying to start Chrome webdriver on linux")

chrome\_options = webdriver.ChromeOptions()

self.webDriver = webdriver.Chrome(chrome\_options=chrome\_options)

except Exception as ex3:

\_sdtflogging.sdtfLogger.error("Failed to start Chrome webdriver on linux {}".format(ex3))

if self.webDriver is None:

try:

\_sdtflogging.sdtfLogger.debug("Trying to start Firefox webdriver on linux")

profile = webdriver.FirefoxProfile()

profile.accept\_untrusted\_certs = True

self.webDriver = webdriver.Firefox(firefox\_profile=profile)

except Exception as ex4:

\_sdtflogging.sdtfLogger.error("Failed to start Firefox webdriver on linux {}".format(ex4))

self.webDriver.implicitly\_wait(\_DEFAULT\_WEBDRIVER\_IMPLICIT\_WAIT)

So that fixes the first issue of Fetching the WebDriver version.

To make it so the actual testing makes use of Selenium Grid, the method that launches the webdriver needs to be adjusted in a similar fashion. This calls for changes in the launch\_webadmin function in sdtf/suites/ias/webadmin/utils.py.

In the original code, it uses selenium for browser automation on the test control host locally, utilizing Selenium.

First, some extra imports need to be made at the top to allow us to use Selenium Grid related calls in this script, additions highlighted:

from framework.ias.webadmin.webpage\_components import (

check\_major\_version,

get\_browser\_by\_name,

DefaultBrowserOptions,

SeleniumGrid

)

This is the updated code for the launch\_webadmin, changes highlighted:

def launch\_webadmin(tc\_or\_tsuite):

"""Launch the webdriver and open the Webadmin page from the suite context.

Following values will be used from the C{tc\_or\_tsuite.suiteContext} variable:

- browser\_name

- browser\_version

- time\_helper

- ias\_machine

@return: The browser and webadmin objects

"""

browser\_name = tc\_or\_tsuite.suiteContext['browser\_name']

sel\_grid = SeleniumGrid('10.132.224.165') #ip of the selenium hub

browser\_options = DefaultBrowserOptions.get\_remote\_defaults(browser\_name)

browser = sel\_grid.create\_browser(browser\_options)

browser.start()

check\_major\_version(browser, tc\_or\_tsuite.suiteContext['browser\_version'])

ias\_tz = tc\_or\_tsuite.suiteContext['time\_helper'].get\_timezone('IAS')

webadmin = Webadmin(

browser,

tc\_or\_tsuite.suiteContext['ias\_machine'].externalIP,

path=tc\_or\_tsuite.suiteContext['ias\_machine'].WEBADMIN\_PATH,

port=tc\_or\_tsuite.suiteContext['ias\_machine'].get\_webadmin\_port(),

ssl=tc\_or\_tsuite.cfg.ias\_webadmin\_ssl,

timezone=ias\_tz,

version=tc\_or\_tsuite.suiteContext['ias\_machine'].get\_version())

return browser, WebAdmin

Let’s go over the changes:

sel\_grid = SeleniumGrid('10.132.224.165')

This creates a new object of the class SeleniumGrid, using the given IP, and by default, port 4444. Here I hardcoded the IP address of the machine on which the Selenium Hub container runs on. As explained previously, the traffic sent to the Docker host on port 4444 is forwarded to the same port number on the Selenium Hub container.

DefaultBrowserOptions.get\_remote\_defaults(browser\_name)

This passes the browser name of the desired browser, as defined as a parameter when you enter the command to run a test suite. The result of this is a dictionary with default capabilities that you expect a Selenium node to have for that browser. These get stored in the variable “browser\_options”. This is necessary for the following command:

browser = sel\_grid.create\_browser(browser\_options)

This returns the Browser object with the sum of the desired capabilities and the address of the Selenium Hub, with methods to start and stop the webdriver.

And then finally

browser.start()

Calls the create\_webdriver method in the Browser object and this opens a browser on the Selenium Grid in an appropriate node, chosen by the Selenium Hub.

This object is passed around for sending commands to that browser. Also note that as long as the webdriver created here isn’t stopped, either explicitly or due to a timeout, the node will stay occupied and won’t accept new connections.

Another small change that needed to be made was the addition of a class to sdtf/suites/ias/WebAdmin/suite.py to cover the new version of Firefox that is used on the Selenium Node.

class FF58WebadminTestSuite(\_WebadminTestSuite):

tl\_mapping = ["WQA Webadmin", "Firefox ESR 58"]

browser\_name = Browser.FIREFOX

browser\_version = '58'

These structural changes combined make it possible to run the whole WebAdmin test suite in a container with SDTF installed, against the containerized Selenium Grid.

### Adapt for parallelization

To adapt the SDTF for parallelization, there need to be a lot of changes made to each sub suite. My strategy of research and implementation here is to look at one sub suite, the create user cases, and then try to define a generalized approach that can be applied to the other test suites.

Additionally my focus was to get one suite running in Firefox next to a suite on Chrome. Internet Explorer was also part of the plan initially but that fell through.

The major issue that prevents two user create suites to run together against the same platform is that the data used for each test case in the old situation. So if two suites are run at the same time, one of them is going to get error’s that they normally shouldn’t get, because the user they are trying to create has already been made by the other suite running against it.

A second issue in the same vein as this is that the cleanup of this sub suite (the removal of the created users, thus returning to status quo) happens at the start of the suite. This means that if an instance of the user create suite is already running, and another instance starts, and executes the cleanup step. This would mean that the first instance of the suite would be disturbed and report failures where there are none.

*Another issue that arises is the tracefile validation. But that deserves its own subtitle.*

#### Differentiating the data

(Plain diff in appendix)

To allow a test suite running against Firefox to run next to a test against Chrome, it is necessary to provide different input data per browser. The most practical solution for this case is to simply add a postfix to the input data, where possible. This solution is ideally suited for the user create test cases. This does however not create the possibility to run two identical Firefox tests simultaneously.

This is what the default test case looks like for creating users (before changes):

class CreateUserDefaultTestCase(\_UserCreationTestCase):

def test\_create\_user\_default(self):

"""Create User, default"""

user\_attr = {'userid': 'wa\_user01311'}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

So to differentiate between the data I wrote the following method that accepts a user id and appends a postfix based on what browser you are testing on. “ff” for Firefox and “gc” for Google Chrome.

def \_affix\_postfix(self, userid):

browser\_postfix = ''

if self.suiteContext['browser\_name'] == Browser.FIREFOX:

browser\_postfix = 'ff'

elif self.suiteContext['browser\_name'] == Browser.CHROME:

browser\_postfix = 'gc'

userid += browser\_postfix

return userid

To then apply this method to the default case:

class CreateUserDefaultTestCase(\_UserCreationTestCase):

def test\_create\_user\_default(self):

"""Create User, default"""

user\_attr = {'userid': self.\_affix\_postfix('wa\_user01311')}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

The other test cases that have only one input can all be adapted this way.

Some cases have multiple users stored in a dictionary and have to be handled in a slightly different way. (addition highlighted)

class CreateUserLocalAuthSet(\_UserCreationTestCase):

def test\_create\_user\_loc\_auth\_set(self):

"""Create User, LocAuth set"""

users = [{'userid': 'wa\_user01320',

'local\_auth': Ias.LOCAL\_AUTH\_DIGIPASS\_ONLY},

{'userid': 'wa\_user01321',

'local\_auth': Ias.LOCAL\_AUTH\_DIGIPASS\_OR\_PASSWORD}]

for user in users:

user['userid'] = self.\_affix\_postfix(user['userid'])

for input\_user in users:

self.base\_case\_pass(\*self.get\_expected\_values(input\_user))

Then there are also a couple of test cases that use a method to get a dictionary of users from a csv file. This I fixed by just printing out the csv file and putting it in a dictionary, then appending the postfix.

class CreateUserSpecialCharPasswordTestCase(\_UserCreationTestCase):

def test\_create\_user\_special\_chars\_in\_password(self):

"""Create User, special characters in the password"""

users = [

{

'confirm\_new\_password': u'$money',

'password': u'$money',

'userid': u'wa\_user01313'

},

{

'confirm\_new\_password': u'P@ssword%',

'password': u'P@ssword%',

'userid': u'wa\_user01314'

},

{

'confirm\_new\_password': u'\xe0\xe1\xe2\xe4\xf1\xf4\xf6\xfb\xfc\xb5\xe7',

'password': u'\xe0\xe1\xe2\xe4\xf1\xf4\xf6\xfb\xfc\xb5\xe7',

'userid': u'wa\_user01315'

},

{

'confirm\_new\_password': u'ThirtyCharacterPassword9876543',

'password': u'ThirtyCharacterPassword9876543',

'userid': u'wa\_user01316'

},

{

'confirm\_new\_password': u'\u0422\u0435\u0441\u04421234',

'password': u'\u0422\u0435\u0441\u04421234',

'userid': u'wa\_user01312'

}

]

for user in users:

user['userid'] = self.\_affix\_postfix(user['userid'])

for input\_user in users:

self.base\_case\_pass(\*self.get\_expected\_values(input\_user))

#### Tracefile

Another quite big issue that arose is the handling of the tracefile. This is a log file generated by WebAdmin, which is handled and parsed by the test cases in the SDTF as an extra validation step.

A first change that was needed was to change the parsing. In the original situation the tracefile parser went through the tracefile and filtered based on which user executed the command, and which command is executed. When two suites are to run concurrently, this would mean that it would encounter duplicate data, and not unique data. This would cause issues with the validation due to faulty error reporting.

This was eventually fixed by making the tracefile parser filter on the user ID of the user that was created in the test case. However when this was fixed and confirmed working for one test case at a time, we recognized another problem.

The challenge that parallelization brings here is when a test case starts to run, it reconfigures IAS to log all operations in a tracefile specific to that test case. But, IAS can only log to one tracefile at a time, so when several test cases are executed concurrently, IAS logs all trace output to the tracefile set up by the latest starting test case. Hence, the test cases that didn’t finish before the last test case started up see only a truncated trace file and the last test case that started up has to filter out the trace output of the test cases started before it that didn’t finish in time.

A few potential solution for this problem were proposed.

1. Not renaming the tracefile for each test case.
   * Benefits:
     + Quick fix, not that much code needs to be adjusted
   * Drawback:
     + Expected huge run time increase as a much bigger tracefile needs to be passed around
2. Also one tracefile, but use timestamps to divide it up
   * Benefits:
     + Would be a solution to the previous suggestion, as it would be less disastrous for the run time
   * Drawbacks:
     + Seems hard to practically realize within a reasonable timeframe
     + Probably prone to errors
3. Multithreading (for more details see next subsection)
   * Benefits
     + Probably little to no increase in runtime
   * Drawback
     + Would take a long time to research and apply to the current code
     + A single process must be configured to start the desired test simultaneously, as the problem will persist if a second suite starts up while a first is already running. So it would be less flexible

In the end it was concluded that permanent and robust solution is out of scope of this project due to time constraints. An intermediate solution is to just get rid of the tracefile validation for the time being. There are still other validation points built in, so it’s still possible to do tests.

#### Multithreading?

#### Other suites

Most of the other test suites can be adapted to work for parallelization in the same manner as the test cases for user creation. In most cases, it will suffice to differentiate the input data used in in each test. Using user ID’s that are unique to each browser ensures that tests do not interfere with each other.

However, there are a few cases where it would be more difficult, maybe even impossible, to make sure concurrent tests do not interfere or interrupt one another. Following are a few test cases that will probably pose a challenge to implement, should this be investigated further with the goal of parallelizing the whole WebAdmin test suite.

##### Logon and Privileges

Here there is a case (LogonWithAdminAccountWhenIASIsDown) where the SDTF stops the IAS service/deamon, and thus the WebAdmin, to test what happens when you try to log in when de service is down. This probably disrupts all other cases running concurrently.

I think parallelizing this is not realistic. It might be possible with multithreading, if you assure that all running cases are at the same point, so they can test this case at the same time. Any other option will result in interrupting other test cases.

In the case of Logon with OTP (one time password), a service generates a time based password (this simulates the behavior of a DIGIPASS) that expires on use, bound to a user. When a second test case would try this with the same user within a certain time frame (32 seconds), it will generate the same password, but that password will already have been “used”, causing the login to fail. A fix for this would be to use a different admin account and a different DIGIPASS.

##### List Users

At the start of this suite, the SDTF fetches the expected values from the WebAdmin via SOAP. Now the issue that can happen here is that after the expected values are fetched, that another concurrent test created or modified a user, such as the user create cases. This would cause a discrepancy between the expected values and the actual values.

The following sub suites will all probably have a problem analogous to this, because they are all listing and searching users in a similar fashion, as well as doing similar preparations:

* Search User, Find Menu
* Search User Find/Manage Menu
* Search DIGIPASS, Find Menu
* Search DIGIPASS, Find/Manage Menu

##### Manage DIGIPASS Application

These will encounter an issue when two of these tests run concurrently.

* DigipassApplManageTestVDPRO
* DigipassApplManageTestBVDPRO
* DigipassApplManageSendVDPOTP

These will encounter an issue when two of these tests run concurrently.

This test will test if mails and SMS messages containing an OTP are correctly delivered. This test suite sets up a server listening for SMS over http and email over SMTP on a certain port. The problem that arises is when a second suite tries to create another server while there is already one running. This could be alleviated by assigning them both different port numbers. However, the message delivery component sending the mail or SMS will probably send them to all registered servers. This could cause issues and will need testing.

# Challenges

Discussing technological limitations and other problems

## IE in a container?

For the execution of web based test suites, there is a need for the tests to be run on Internet Explorer to ensure a good coverage of use cases. There are, however, a few issues with the proposed deployment architecture related to Internet Explorer. The first being: running internet explorer in a container

The SeleniumHQ repository offers Docker images for a Selenium Hub and for nodes with either Firefox or Google Chrome, with Internet Explorer being notable absent.

There is, however, a NodeBase dockerfile provided to serve as a basis from where to build your own dockerfile to create a non-standard image. The NodeBase is equipped with only the install of selenium, and does not include a browser.

However, the issue here is that the NodeBase image is based on the Base image in the repository that in turn is based on Ubuntu 16.04. In addition, Internet explorer was never ported to Ubuntu, or any other Linux distribution for that matter, so installing it on the NodeBase image is practically impossible.

There is always the possibility of installing wine (a compatibility layer capable of running windows applications on other operating systems such as Linux and macOS), but that would be a bad idea as running in on a compatibility layer in a container seems like more trouble than it’s worth. Especially considering that Internet Explorer is already known for running slowly and being generally unstable on wine.

Another problem is that if you were to build a Docker image based on Windows and run an Internet Explorer browser on it that way, then you wouldn’t be able to run that image on a Linux host. This is because a container image always needs to be based on the same OS as the host you are running it on.

The workaround here is just to run a selenium node running Internet Explorer on a windows machine and adding it to the grid. This way we lose the scalability of running it as a container, but it is necessary to get it working.

(do I need to put the steps to configure a machine to be a selenium node here?)

## Issues with the IE node

Discuss the IE Webriver and why it makes it difficult/impossible to run the suite

<https://stackoverflow.com/questions/19662045/selenium-hover-elements-with-ie>

<http://jimevansmusic.blogspot.be/2013/01/revisiting-native-events-in-ie-driver.html>

<https://github.com/SeleniumHQ/selenium/wiki/InternetExplorerDriver>

<https://github.com/seleniumhq/selenium-google-code-issue-archive/issues>

Part IV - Conclusion

# Performance difference

(1) Recap current status

C

(2) Measure new situation

When running the user create test sub-suite against WebAdmin on the new deployment architecture, these is the resource usage as measured from the Docker host using dockerstat on the container running the SDTF.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Browser | Avg. Cpu (%) | Peak Cpu (%) | Avg RAM (GB) | Peak RAM (GB) |
| Firefox |  |  |  |  |
| Chrome |  |  |  |  |
| Both |  |  |  |  |

Compare Runtime:

|  |  |  |
| --- | --- | --- |
|  | Old runtime | New runtime |
| Firefox |  |  |
| Chrome |  |  |
| Both |  |  |

Make assumption based on (1) and (2)

# Conclusion

Should at least answer the following questions:

* Which parts of the proposed deployment architecture were actually implemented?
* Recap which other parts were scrapped (maybe shortly touch on the why), and which parts were changed due to technological limitations or other constraints?
* What didn’t work at all?
* Is there a gain in efficiency?
  + Why, or why not?
* Is the gain worth the trouble?

Include suggestions for future research

NO NEW INFORMATION HERE!!!

Part V – Sources & Appendix

# Sources

# Appendix

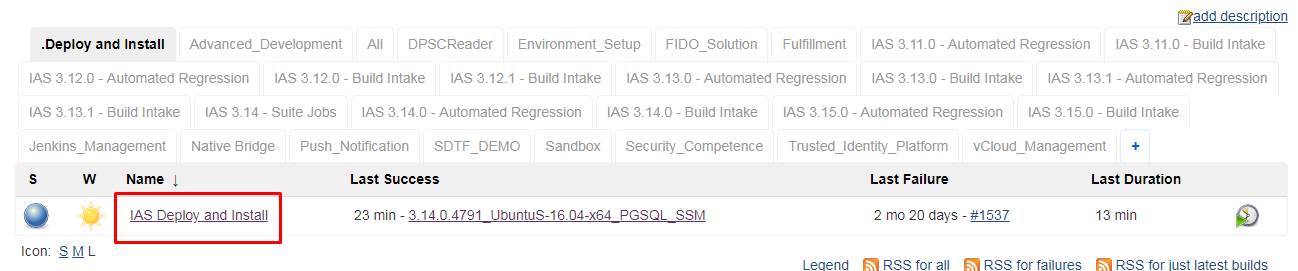
## IAS on vCloud

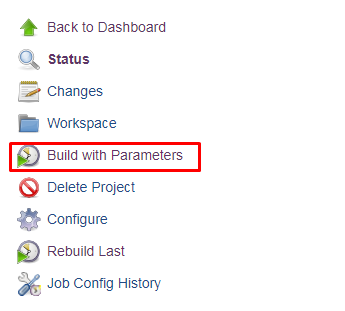
This step describes how to set up an Identikey Administration

<http://jenkins-qa.vasco.com/> and log in with AD username and pw

click on deploy and install







Test project is customizable (I just left it default)

Test plan needs to be SDTF\_DEMO (leave default)

AD\_USER is your username

AD\_PW is your password

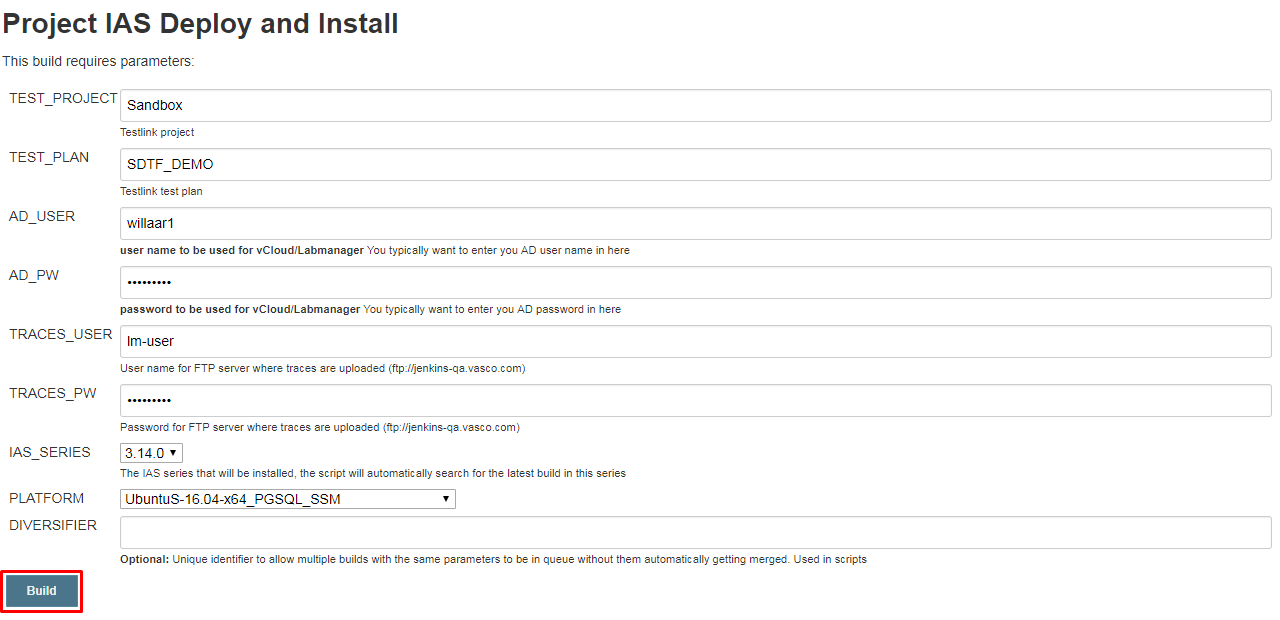
TRACES\_USER is lm-user

TRACES\_PW is Shared1234 (does this need to be redacted in my final hand in?>

IAS\_SERIES is 3.14.0

PLATFORM is UbuntuS-16.04-x64\_PGSQL\_SSM

DIVERSIFIER is optional and left empty



Click build when done

Wait for it to build, and then you’re done.

Don’t forget to give the sdtf the right platform as a startup parameter

python ~/sdtf/suites/ias/webadmin/suite.py "3.14.0.4791" "UbuntuS-16.04-x64\_PGSQL\_SSM" ChromeWebadminTestSuite --projectName="Identikey Server" --testplanName="IAS 3.14.0 - Automated Webadmin" --virt.user="lm-auto-wemmel" --virt.password=$LMPW --traces.user="lm-user" --traces.password=Shared1234 --install.media="//10.132.0.242/wqa/QC-Projects/01-Identikey/3.15.0/builds/3.15.0.2262/ias-dev\_3.15.0.2262.iso" --deployed\_name="3.14.0.4791\_UbuntuS-16.04-x64\_PGSQL\_SSM"

## Dockerfiles

### For use with the SSH tunnel

FROM python:2.7

ADD python\_test.py /

# Selectively add parts of SDTF needed for IAS testing

ADD sdtf /root/sdtf

# RUN mkdir -p /root/sdtf/suites/ias/

# ADD sdtf/suites/ias /root/sdtf/suites/ias/

# The sdtf directory may be a FUSE sshfs mount from a developer's Windows

# PC and as such, already contain files produced by instal\_sdtf{,\_linux}.py

# To avoid conflicts with the future invocation of install\_sdtf\_linux.py,

# remove them:

# RUN ls -l /root/sdtf > /root/sdtf-contents

# RUN rm -rf /root/sdtf/runtime /root/sdtf/sdtf\*activat\*

# The following command removes the carriage return line endings that cause problems when copying the file from a windows machine

# This was made necessary because we currently copy the sdtf folder via an ssh tunnel.

The above comments were made obsolete by the .dockerignore file

The following two commands are necessary to fix the line endings of the shell script

RUN tr -d "\r" < /root/sdtf/install\_sdtf\_linux.sh > /root/sdtf/install\_sdtf\_linux.sh.new

RUN mv /root/sdtf/install\_sdtf\_linux.sh.new /root/sdtf/install\_sdtf\_linux.sh

RUN apt-get update && apt-get install -y \

python-tk \

python-pip \

python-ldap \

&& rm -rf /var/lib/apt/lists/\*

RUN apt-get update --fix-missing

RUN pip install virtualenv

RUN chmod +x /root/sdtf/install\_sdtf\_linux.sh

RUN /root/sdtf/install\_sdtf\_linux.sh

RUN tr -d "\r" < /root/sdtf/sdtf-activate > /root/sdtf/sdtf-activate.new

RUN mv /root/sdtf/sdtf-activate.new /root/sdtf/sdtf-activate

ENV LMPW=U89yMN8R8bwgOKR1k6tz

CMD [ "/bin/bash" ]

## Install script for SDTF on linux container

#!/bin/sh

# TODO: allow install path to be specified, for now always install in homefolder ~/ + Test in other ditro's

echo 'This script will install SDTF on Linux (Debian based distros for now)'

echo '==> Perform apt-get update to get the latest package lists'

apt-get update

echo '==> Install python tcl library'

apt-get -y install python-tk

echo '==> Install python tcl library (Fix missing libs)'

apt-get update --fix-missing

echo '==> Install PIP'

apt install -y python-pip

echo '==> Install virtualenv and pyvirtualdisplay'

pip install virtualenv

pip install pyvirtualdisplay selenium

apt-get -y install xvfb

echo '==> Install chromium and libxi6 libgconf-2-4'

apt-get -y install chromium

apt-get -y install libxi6 libgconf-2-4

echo '==> Install python ldap module'

apt-get -y install python-ldap

python ~/sdtf/install\_sdtf.py ~/sdtf/runtime

cp -avr /usr/lib/python2.7/dist-packages/ldap/ ~/sdtf/runtime/lib/python2.7/site-packages/ldap

cp /usr/lib/python2.7/dist-packages/\_ldap.so ~/sdtf/runtime/lib/python2.7/site-packages/\_ldap.so

echo '==> Install STAF module'

wget -P ~/Downloads http://prdownloads.sourceforge.net/staf/STAF3425-linux-amd64.tar.gz

tar xzvf ~/Downloads/STAF3425-linux-amd64.tar.gz -C /tmp

# cd /tmp/staf

/tmp/staf/STAFInst -source /tmp/staf -target ~/sdtf/runtime/bin/STAF -type f -acceptlicense -option USE\_PYTHON\_VERSION=2.7 -option USE\_PYTHON\_SYSTEM\_PATH=0 -option UPDATE\_ENVIRONMENT=None

# See http://stackoverflow.com/questions/856116/changing-ld-library-path-at-runtime-for-ctypes for next lines, avoid the conflict between STAF and Openssl in using the same crypto libs :

# mv ~/sdtf/runtime/bin/STAF/lib/libcrypto.so.1.0.0 ~/sdtf/runtime/bin/STAF/lib/libcrypto.so.1.0.0\_bak /////test if it works without this

apt-get install -y patchelf

patchelf --set-rpath ~/sdtf/runtime/bin/STAF/lib/ ~/sdtf/runtime/bin/STAF/lib/PYSTAF.so

patchelf --set-rpath ~/sdtf/runtime/bin/STAF/lib/ ~/sdtf/runtime/bin/STAF/lib/libSTAF.so

pip install -U selenium

echo 'Please run `source ~/sdtf/sdtf-activate` to enter runtime environment. '

## Plain diff: Differentiating input data per browser for cases\_user\_create

diff --git a/suites/ias/webadmin/cases\_user\_create.py b/suites/ias/webadmin/cases\_user\_create.py

index 194b112..8688b10 100644

--- a/suites/ias/webadmin/cases\_user\_create.py

+++ b/suites/ias/webadmin/cases\_user\_create.py

@@ -5,6 +5,7 @@ from framework.ias.tracefile\_parser import CommandIterator

from framework.ias.utils.map\_validation\_objects\_to\_known\_issues import (

match\_strings\_with\_regexp\_list,

)

+from framework.ias.webadmin.webpage\_components import Browser

from suites.ias.webadmin.webadmin\_case import WebadminBaseTestCase

from dataset\_factory import UserCreateDatasetFactory

@@ -183,19 +184,35 @@ class \_UserCreationTestCase(WebadminBaseTestCase):

actual\_error\_messages = StepGetErrorMessages(self)(user\_creation\_page)

VerifyErrorMessageList(self)(xp\_error\_messages, actual\_error\_messages)

self.verify\_result()

+

+ def \_affix\_postfix(self, userid):

+ browser\_postfix = ''

+ if self.suiteContext['browser\_name'] == Browser.FIREFOX:

+ browser\_postfix = 'ff'

+ elif self.suiteContext['browser\_name'] == Browser.CHROME:

+ browser\_postfix = 'gc'

+ userid += browser\_postfix

+ return userid

+

class CreateUserDefaultTestCase(\_UserCreationTestCase):

def test\_create\_user\_default(self):

- """Create User, default"""

- user\_attr = {'userid': 'wa\_user01311'}

+ """Create User, default

+ if self.suiteContext['browser\_name'] == Browser.FIREFOX:

+ x

+ elif self.suiteContext['browser\_name'] == Browser.CHROME:

+ x

+ else:

+ """

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01311')}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

class CreateUserEmptyPasswordTestCase(\_UserCreationTestCase):

def test\_create\_user\_empty\_password(self):

"""Create User, empty password"""

- user\_attr = {'userid': 'wa\_user01310',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01310'),

'password': None,

'confirm\_new\_password': None}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

@@ -204,25 +221,86 @@ class CreateUserEmptyPasswordTestCase(\_UserCreationTestCase):

class CreateUserSpecialCharPasswordTestCase(\_UserCreationTestCase):

def test\_create\_user\_special\_chars\_in\_password(self):

"""Create User, special characters in the password"""

- users = self.dataset\_factory.\

- get\_users\_from\_csv(self.SUITE\_NAME, self.id())

+ users = [

+ {

+ 'confirm\_new\_password': u'$money',

+ 'password': u'$money',

+ 'userid': u'wa\_user01313'

+ },

+ {

+ 'confirm\_new\_password': u'P@ssword%',

+ 'password': u'P@ssword%',

+ 'userid': u'wa\_user01314'

+ },

+ {

+ 'confirm\_new\_password': u'\xe0\xe1\xe2\xe4\xf1\xf4\xf6\xfb\xfc\xb5\xe7',

+ 'password': u'\xe0\xe1\xe2\xe4\xf1\xf4\xf6\xfb\xfc\xb5\xe7',

+ 'userid': u'wa\_user01315'

+ },

+ {

+ 'confirm\_new\_password': u'ThirtyCharacterPassword9876543',

+ 'password': u'ThirtyCharacterPassword9876543',

+ 'userid': u'wa\_user01316'

+ },

+ {

+ 'confirm\_new\_password': u'\u0422\u0435\u0441\u04421234',

+ 'password': u'\u0422\u0435\u0441\u04421234',

+ 'userid': u'wa\_user01312'

+ }

+ ]

+ """

+ PHASED OUT IN FAVOR OF A METHOD (self.\_affix\_postfix)

+ browser\_postfix = ''

+ if self.suiteContext['browser\_name'] == Browser.FIREFOX:

+ browser\_postfix = 'ff'

+ elif self.suiteContext['browser\_name'] == Browser.CHROME:

+ browser\_postfix = 'gc'

+ """

+ for user in users:

+ user['userid'] = self.\_affix\_postfix(user['userid'])

for input\_user in users:

- self.base\_case\_pass(\*self.get\_expected\_values(input\_user.\_\_dict\_\_))

+ self.base\_case\_pass(\*self.get\_expected\_values(input\_user))

class CreateUserSpecialCharUsernameTestCase(\_UserCreationTestCase):

def test\_create\_user\_special\_chars\_in\_username(self):

- """Create User, special characters in the username"""

- users = self.dataset\_factory.\

- get\_users\_from\_csv(self.SUITE\_NAME, self.id())

+ users = [

+ {

+ 'confirm\_new\_password': u'Test1234',

+ 'password': u'Test1234',

+ 'userid': u'wa\_user$%13'

+ },

+ {

+ 'confirm\_new\_password': u'Test1234',

+ 'password': u'Test1234',

+ 'userid': u'wa\_user\xe0\xe1\xe2\xe4\xf1\xf4\xf6\xfb\xfc\xb5\xe713'

+ },

+ {

+ 'confirm\_new\_password': u'Test1234',

+ 'password': u'Test1234',

+ 'userid': u'wa\_\u043f\u043e\u043b\u044c\u0437\u043e\u0432\u0430\u0442\u0435\u043b\u044c13'

+ },

+ {

+ 'confirm\_new\_password': u'Test1234',

+ 'password': u'Test1234',

+ 'userid': u'test\\wa\_user01318'

+ },

+ {

+ 'confirm\_new\_password': u'Test1234',

+ 'password': u'Test1234',

+ 'userid': u'wa\_user01319@test'

+ }

+ ]

+ for user in users:

+ user['userid'] = self.\_affix\_postfix(user['userid'])

for input\_user in users:

- self.base\_case\_pass(\*self.get\_expected\_values(input\_user.\_\_dict\_\_))

+ self.base\_case\_pass(\*self.get\_expected\_values(input\_user))

class CreateUserInOU(\_UserCreationTestCase):

def test\_create\_user\_in\_ou(self):

"""Create User in OU"""

- user\_attr = {'userid': 'wa\_user01317',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01317'),

'organizational\_unit': 'OU1'}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

@@ -234,6 +312,8 @@ class CreateUserLocalAuthSet(\_UserCreationTestCase):

'local\_auth': Ias.LOCAL\_AUTH\_DIGIPASS\_ONLY},

{'userid': 'wa\_user01321',

'local\_auth': Ias.LOCAL\_AUTH\_DIGIPASS\_OR\_PASSWORD}]

+ for user in users:

+ user['userid'] = self.\_affix\_postfix(user['userid'])

for input\_user in users:

self.base\_case\_pass(\*self.get\_expected\_values(input\_user))

@@ -247,6 +327,8 @@ class CreateUserBackendAuthSet(\_UserCreationTestCase):

'backend\_auth': Ias.BE\_AUTH\_ALWAYS},

{'userid': 'wa\_user01325',

'backend\_auth': Ias.BE\_AUTH\_NONE}]

+ for user in users:

+ user['userid'] = self.\_affix\_postfix(user['userid'])

for input\_user in users:

self.base\_case\_pass(\*self.get\_expected\_values(input\_user))

@@ -254,7 +336,7 @@ class CreateUserBackendAuthSet(\_UserCreationTestCase):

class CreateUserDisabled(\_UserCreationTestCase):

def test\_create\_user\_disabled(self):

"""Create Disabled User"""

- user\_attr = {'userid': 'wa\_user01326',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01326'),

'disabled': True}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

@@ -262,7 +344,7 @@ class CreateUserDisabled(\_UserCreationTestCase):

class CreateUserLocked(\_UserCreationTestCase):

def test\_create\_user\_locked(self):

"""Create Locked User"""

- user\_attr = {'userid': 'wa\_user01327',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01327'),

'locked': True}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

@@ -273,7 +355,7 @@ class CreateUserTemporaryUser(\_UserCreationTestCase):

fut\_date = self.time\_helper.get\_future\_datetime(delta=timedelta(days=20),

timezone\_name='IAS',

precision='minutes')

- user\_attr = {'userid': 'wa\_user01328',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01328'),

'expiration\_time': fut\_date}

self.base\_case\_pass(\*self.get\_expected\_values(user\_attr))

@@ -285,6 +367,7 @@ class CreateUserExcNotAllowedCharsUsername(\_UserCreationTestCase):

input\_user = self.get\_expected\_values\_webadmin\_exception\_case(user\_attr)

xp\_error\_messages = ['Please complete the required fields : User ID',

'Field User ID: the following characters are not allowed: /:;,|\'"<>[]&=+\*?#']

+ input\_user['userid'] = self.\_affix\_postfix(input\_user['userid'])

self.base\_case\_webdmin\_exception(input\_user,

xp\_error\_messages)

@@ -296,6 +379,7 @@ class CreateUserExcExpirationTimeWrongFormat(\_UserCreationTestCase):

'expiration\_time': 'test data'}

input\_user = self.get\_expected\_values\_webadmin\_exception\_case(user\_attr)

xp\_error\_messages = ['The date format is invalid, it should be in the format of yyyy-MM-dd HH:mm.']

+ input\_user['userid'] = self.\_affix\_postfix(input\_user['userid'])

self.base\_case\_webdmin\_exception(input\_user,

xp\_error\_messages)

@@ -308,6 +392,7 @@ class CreateUserExcExpirationTimeInThePast(\_UserCreationTestCase):

'expiration\_time': expiration\_time}

input\_user = self.get\_expected\_values\_webadmin\_exception\_case(user\_attr)

xp\_error\_messages = ['The date field can not be in the past.']

+ input\_user['userid'] = self.\_affix\_postfix(input\_user['userid'])

self.base\_case\_webdmin\_exception(input\_user,

xp\_error\_messages)

@@ -344,6 +429,7 @@ class CreateUserExcAlreadyExists(\_UserCreationTestCase):

xp\_error\_messages = ["Error creating User. Error message : The specified object already exists."]

xp\_error\_codes = [-14]

input\_user, xp\_cmd\_input = self.get\_expected\_values\_audit\_exception\_case(user\_attr)

+ input\_user['userid'] = self.\_affix\_postfix(input\_user['userid'])

self.base\_case\_audit\_exception(input\_user,

xp\_cmd\_input,

xp\_error\_messages,

@@ -358,6 +444,7 @@ class CreateUserExcPasswordsDoNotMatch(\_UserCreationTestCase):

'confirm\_new\_password': 'different', }

input\_user = self.get\_expected\_values\_webadmin\_exception\_case(user\_attr)

xp\_error\_messages = ['Your passwords do not match. Please enter and confirm the same password']

+ input\_user['userid'] = self.\_affix\_postfix(input\_user['userid'])

self.base\_case\_webdmin\_exception(input\_user,

xp\_error\_messages)

@@ -373,7 +460,7 @@ class CreateUserExcPasswordNotMeetingRestrictions(\_UserCreationTestCase):

def test\_create\_user\_spw\_not\_meeting\_pwd\_restrictions(self):

"""Create User, non-conforming PSW restrictions"""

- user\_attr = {'userid': 'wa\_user01301',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01301'),

'password': 'Test',

'confirm\_new\_password': 'Test'}

input\_user, xp\_cmd\_input = self.get\_expected\_values\_audit\_exception\_case(user\_attr)

@@ -384,7 +471,7 @@ class CreateUserExcPasswordNotMeetingRestrictions(\_UserCreationTestCase):

xp\_error\_messages,

xp\_error\_codes)

- user\_attr = {'userid': 'wa\_user01301',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01301'),

'password': 'TEST1234',

'confirm\_new\_password': 'TEST1234'}

input\_user, xp\_cmd\_input = self.get\_expected\_values\_audit\_exception\_case(user\_attr)

@@ -395,7 +482,7 @@ class CreateUserExcPasswordNotMeetingRestrictions(\_UserCreationTestCase):

xp\_error\_messages,

xp\_error\_codes)

- user\_attr = {'userid': 'wa\_user01301',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01301'),

'password': 'test1234',

'confirm\_new\_password': 'test1234'}

input\_user, xp\_cmd\_input = self.get\_expected\_values\_audit\_exception\_case(user\_attr)

@@ -406,7 +493,7 @@ class CreateUserExcPasswordNotMeetingRestrictions(\_UserCreationTestCase):

xp\_error\_messages,

xp\_error\_codes)

- user\_attr = {'userid': 'wa\_user01301',

+ user\_attr = {'userid': self.\_affix\_postfix('wa\_user01301'),

'password': 'User01301',

'confirm\_new\_password': 'User01301'}

input\_user, xp\_cmd\_input = self.get\_expected\_values\_audit\_exception\_case(user\_attr)