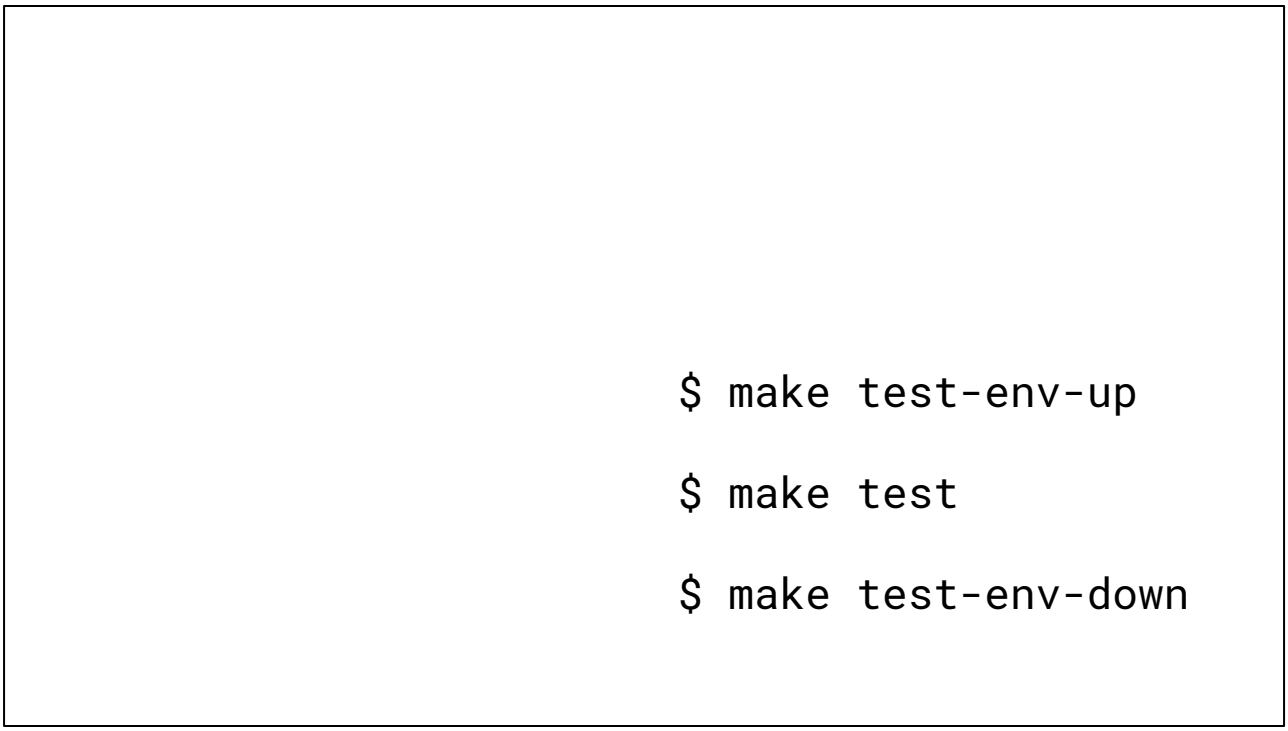
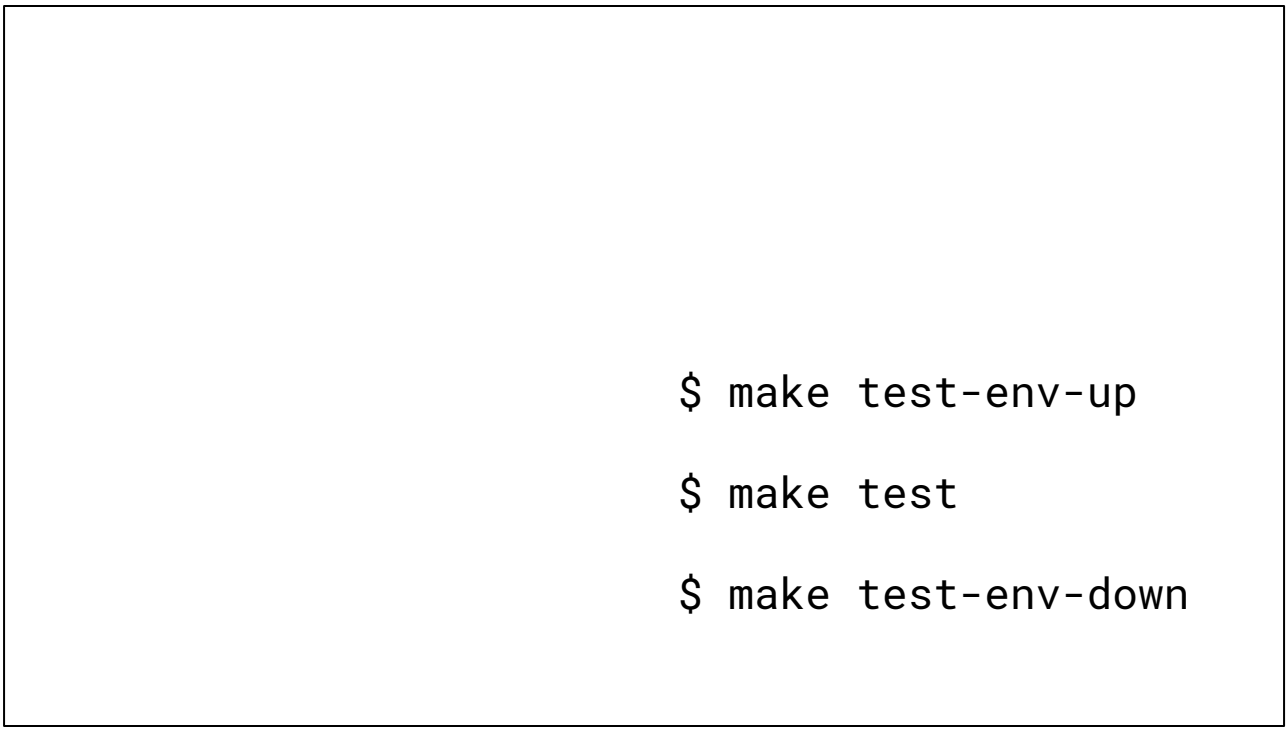


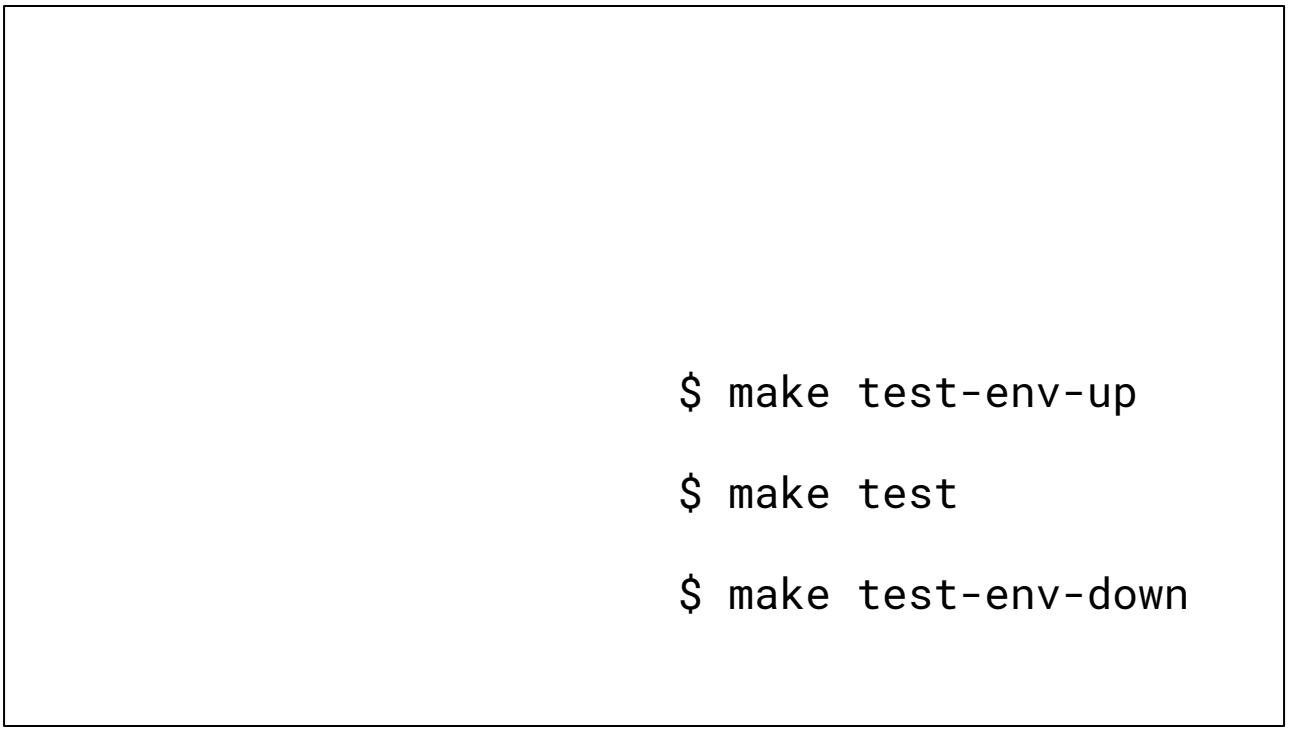
make test-env-up
make test
make test-env-down

When I go to a team at my company, and I want to tell them how they can get started running the Selenium based test cases, it's three commands:



make test-env-up
make test
make test-env-down

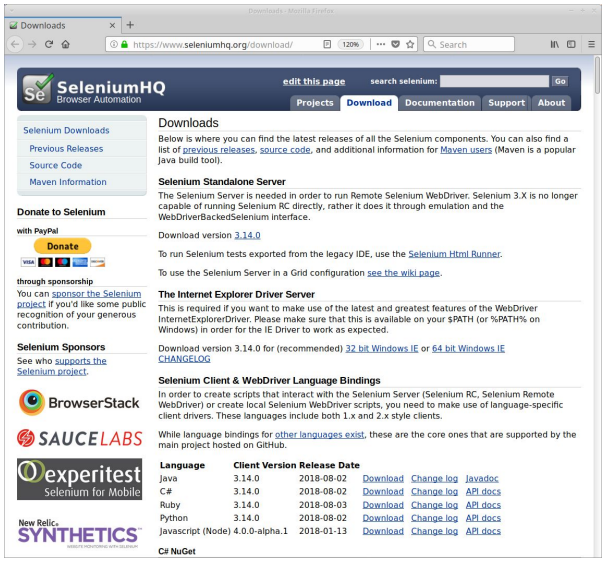
And when I'm working 1 on 1 with someone who just made a change to a product,
only to find that the automated builds started failing, the first thing we do,
on their computer, is run these three commands:



make test-env-up
make test
make test-env-down

It's three commands to get started because when someone is faced with the daunting task of debugging a failing test case that they probably didn't write, I can't tell them to go through a page worth of instructions on how to install the prerequisites for the test system on their computer so that they can try to reproduce the error.

I mean, you know what it takes to get Selenium setup on your own computer, right?



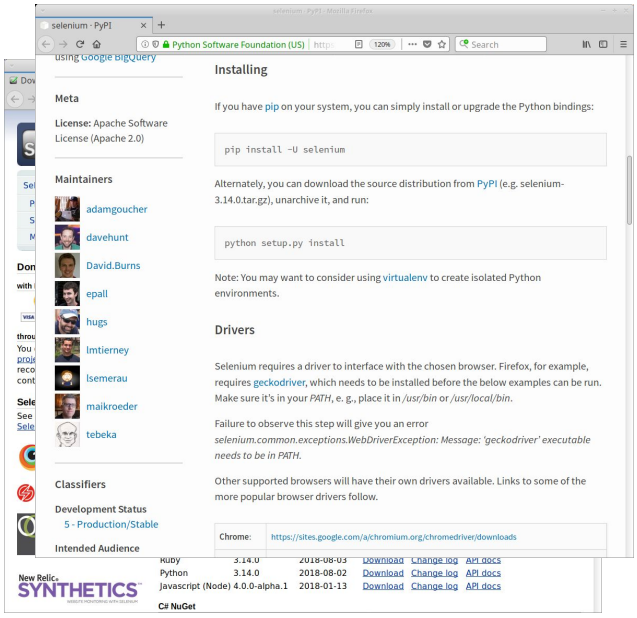
https://www.seleniumhq.org/ 18 Oct, 2018

```
$ make test-env-up

$ make test

$ make test-env-down
```

Probably the easiest route to getting started is to talk directly to the web browsers installed on your computer.

So you'll need to download a web browser.

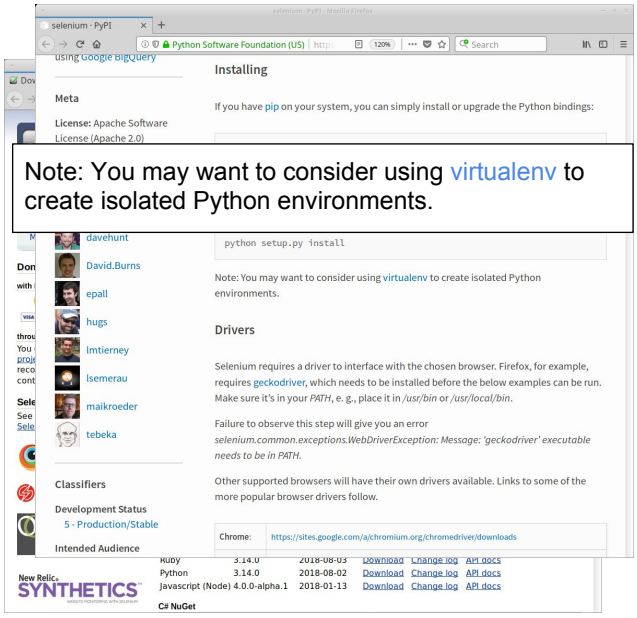
Then you'll choose your language, some people choose Java, they have an installer, but on my Linux systems, sometimes they come with older versions or I have to do some work to keep them up to date.



https://pypi.org/project/selenium/ 18 Oct, 2018

```
$ make test-env-up

$ make test

$ make test-env-down
```

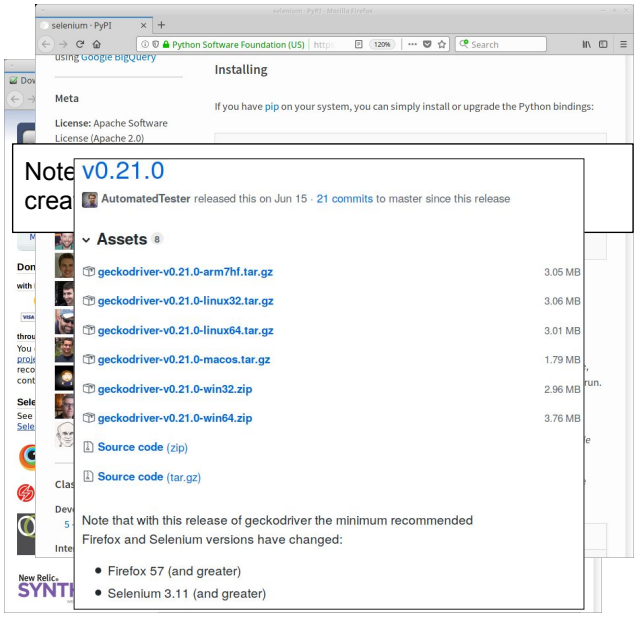
I often choose Python, but even Python is going through a bit of an identity crisis as people move between Python 2 and Python 3. With python, you also have to watch out for upgrading or downgrading versions of dependencies that your system may depend on.



https://www.seleniumhq.org/ 18 Oct, 2018

```
$ make test-env-up

$ make test

$ make test-env-down
```

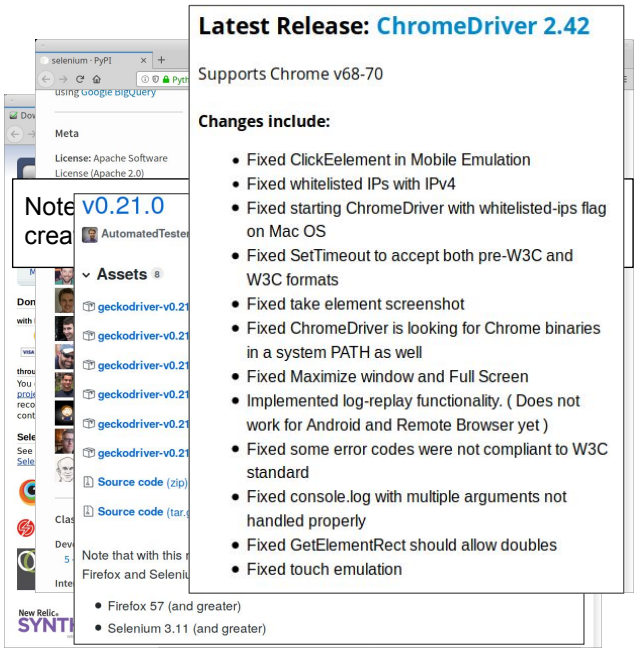
Instructions usually suggest you install into a `virtualenv` virtual environment.



https://github.com/mozilla/geckodriver/releases / 18 Oct, 2018

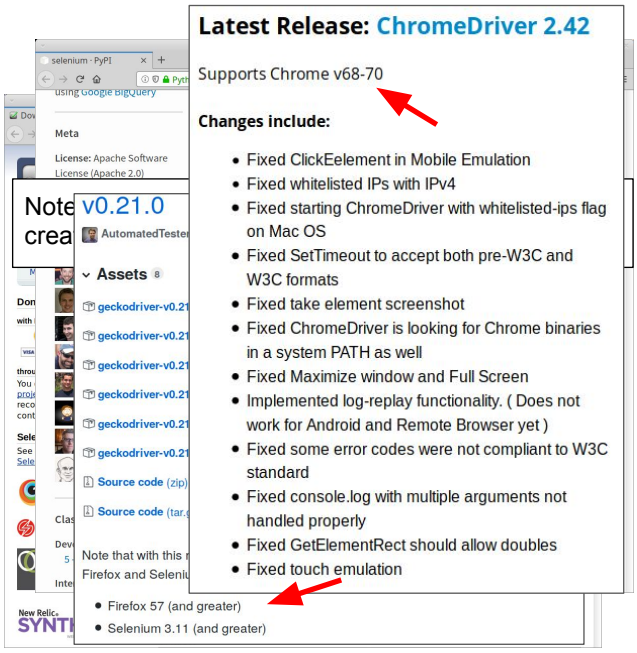
```
$ make test-env-up

$ make test

$ make test-env-down
```

Next, you need to get drivers for the web browser you downloaded earlier.
https://selenium-python.readthedocs.io/installation.html#drivers




```
$ make test-env-up

$ make test

$ make test-env-down
```

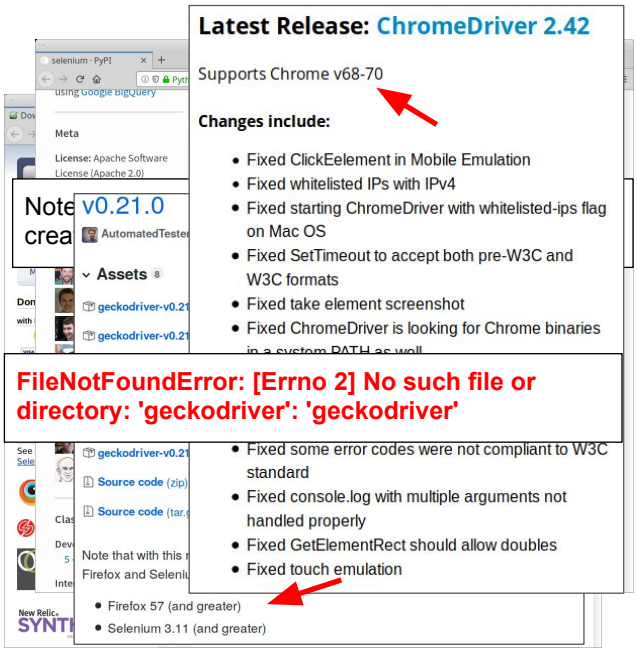
Browser manufacturers are doing a good job at providing WebDriver specification compliant drivers to interface with their web browsers. You'll need to download and install these drivers so your program can interface with the web browser.



http://chromedriver.chromium.org/ 18 Oct, 2018

```
$ make test-env-up

$ make test

$ make test-env-down
```

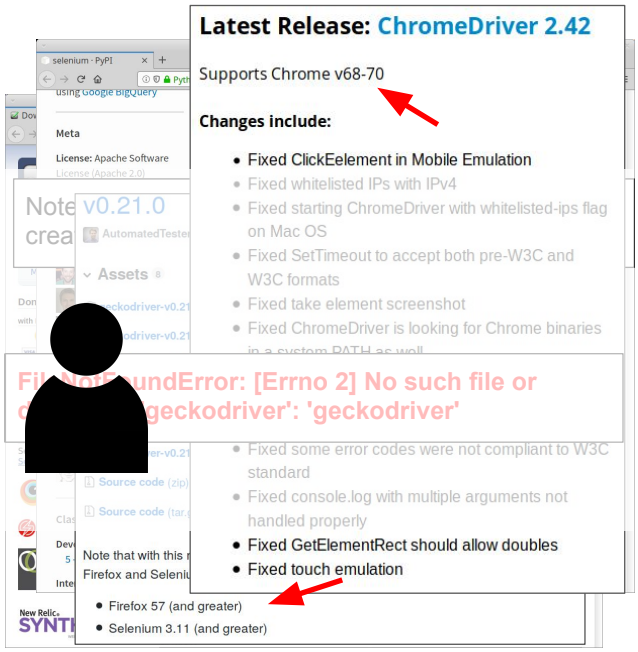
Make sure you get a driver that supports the web browser you downloaded earlier.




```
$ make test-env-up

$ make test

$ make test-env-down
```

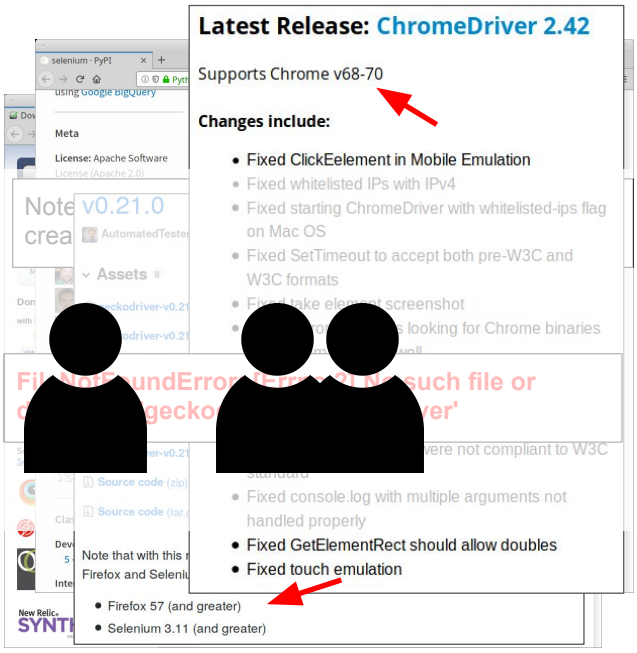
By the way, make sure the driver is in your path, or your Python program may complain about
not finding the driver.

And now you can start running your browser automation scripts.



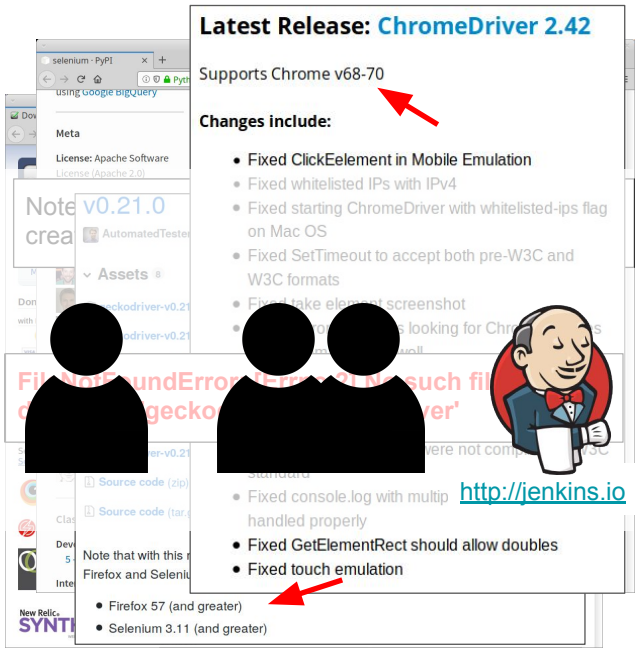

```
$ make test-env-up

$ make test

$ make test-env-down
```

But what about your coworker?
If you're working with someone else, they will need to do this same setup.
Maybe along the way, they choose a different version of the web browsers, or a different
version of the programming language, or install things in a different location.




```
$ make test-env-up

$ make test

$ make test-env-down
```

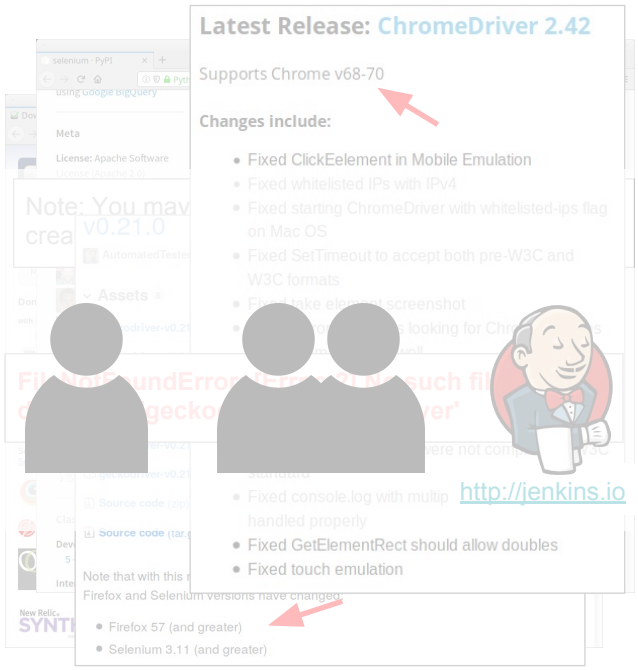
And if you're working with a team you get to tell everyone how to get their system setup and deal with the differences in operating systems and environments.




```
$ make test-env-up

$ make test

$ make test-env-down
```

And if you're trying to run your automation scripts in Jenkins, there's another environment you get to manage.

And lets hope nobody ever upgrades their system, right? Nothing ever breaks after an upgrade, right?




```
$ make test-env-up

$ make test

$ make test-env-down
```

So it has to be as easy as these three commands.

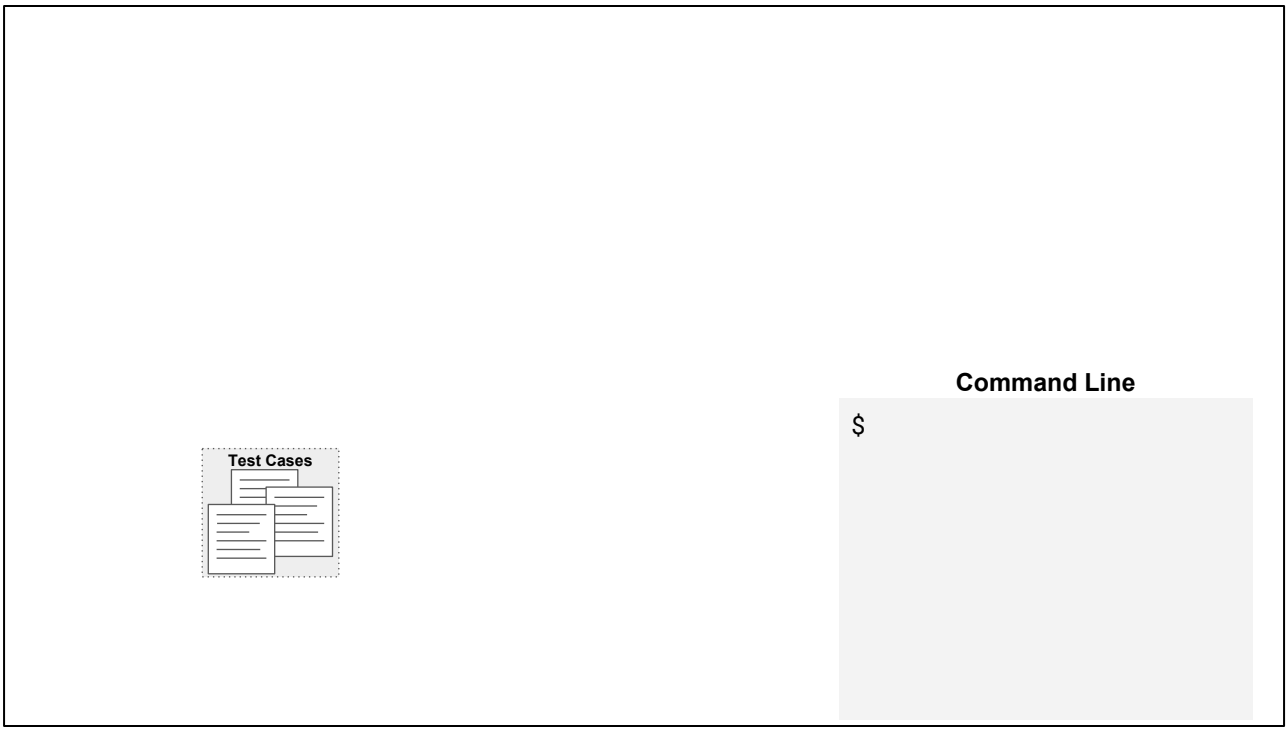
And this is why the teams I work with designed a test setup that works for them, by taking advantage of
software they usually have on our computers for development anyways.

Jump Starting Your Testing
with Selenium Grid Docker Containers, Selene, and pytest

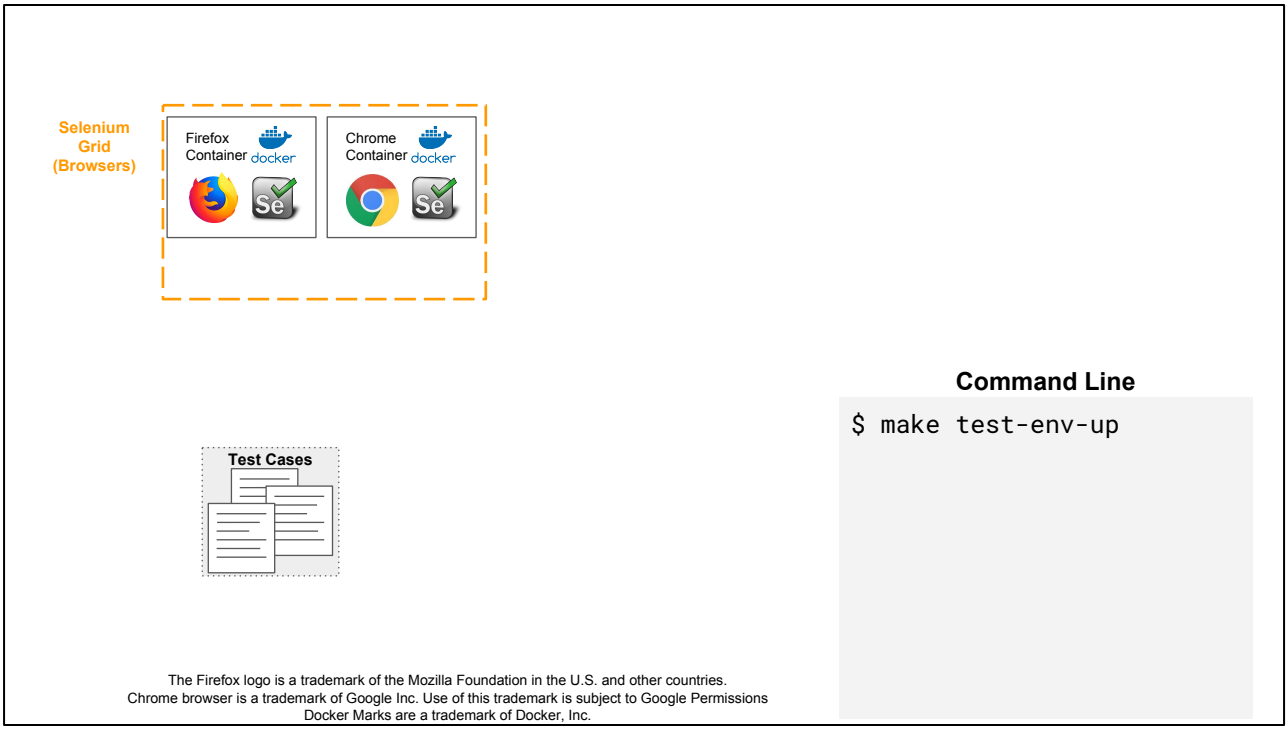
Derrick Kearney
telldsk@gmail.com

https://github.com/dskard/seleniumconf2018

My name is Derrick Kearney and these are some of these are some of the issues I was faced with when I started working on a new project recently. Today, I'd like to share with you why I choose to share these three commands with the teams I work with, and how they have become sufficient enough to get people started working with Selenium.



Under the hood, there are a number of technologies that we are using to make these three commands the only commands people have to remember to get started.

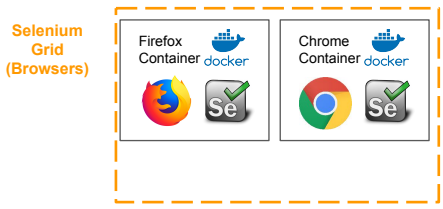


When we type `make test-env-up` on our command line,

It launches a set of Docker containers that represent our "test environment". These are the services that our test cases will need in order to run.

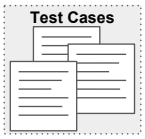
At the very least, we'll need a set of web browsers we want to test against.

More on Docker:
http://docker.com/resources/what-container



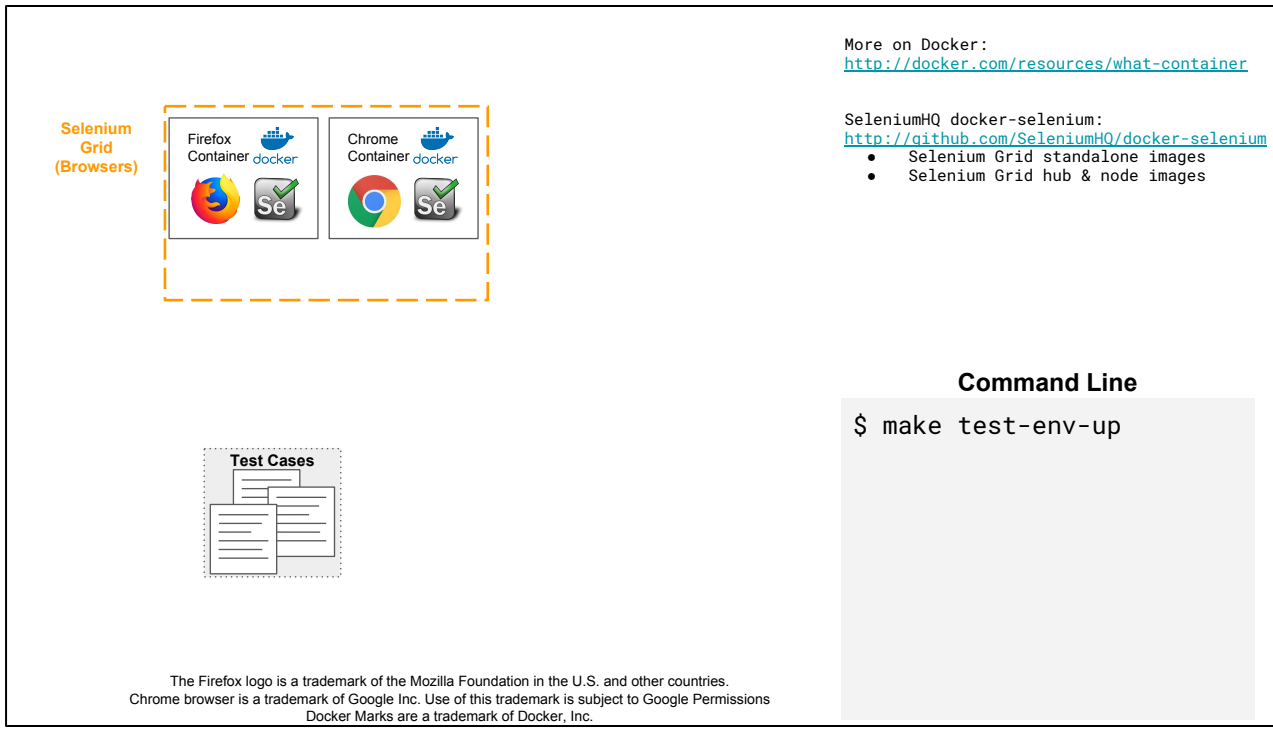
Command Line

```
$ make test-env-up
```

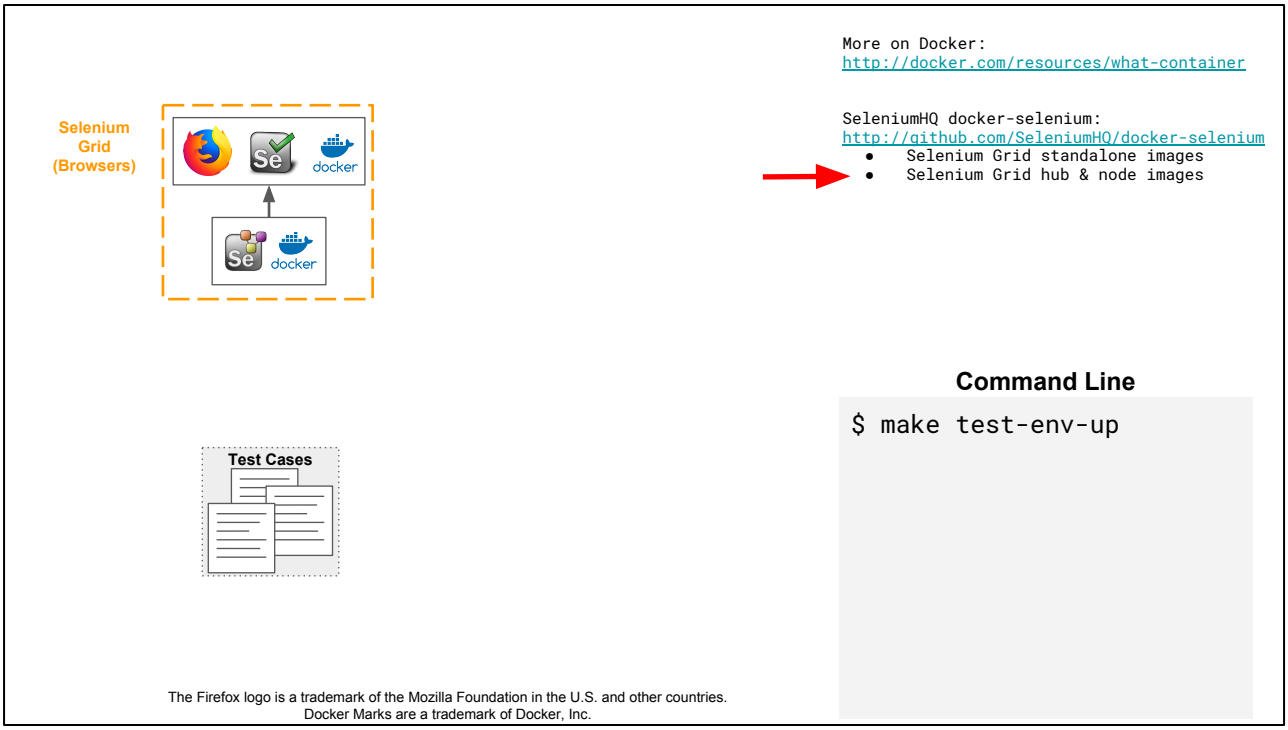


If you're not familiar with Docker containers, you can think of them as small environments where you can run a single process, separated from the other processes on your computer. One nice thing about them is that these environment can be reliably reproduced on someone else's computer, so we can share the environments with other people.

https://www.docker.com/resources/what-container



Lucky for us, the Selenium project maintains a set of Docker images with web browsers and browser drivers configured and ready for use for Firefox or Chrome.



Even better, they also maintain Docker images for Selenium Grid and the node processes to launch web browsers from a Selenium Grid.

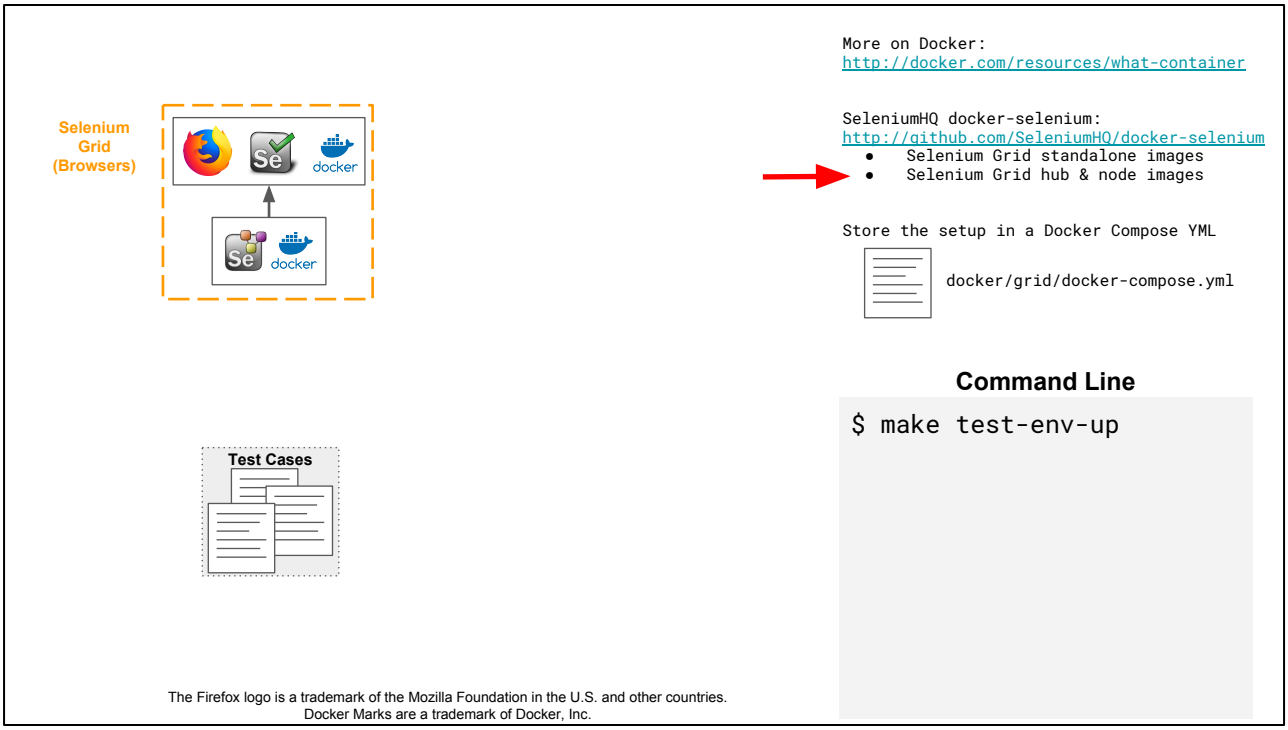
With Selenium Grid, your automation program will ask for a browser with a set of capabilities and the grid will be responsible to connecting you to that web browser.

Building your test automation system like this gives you 2 main benefits:

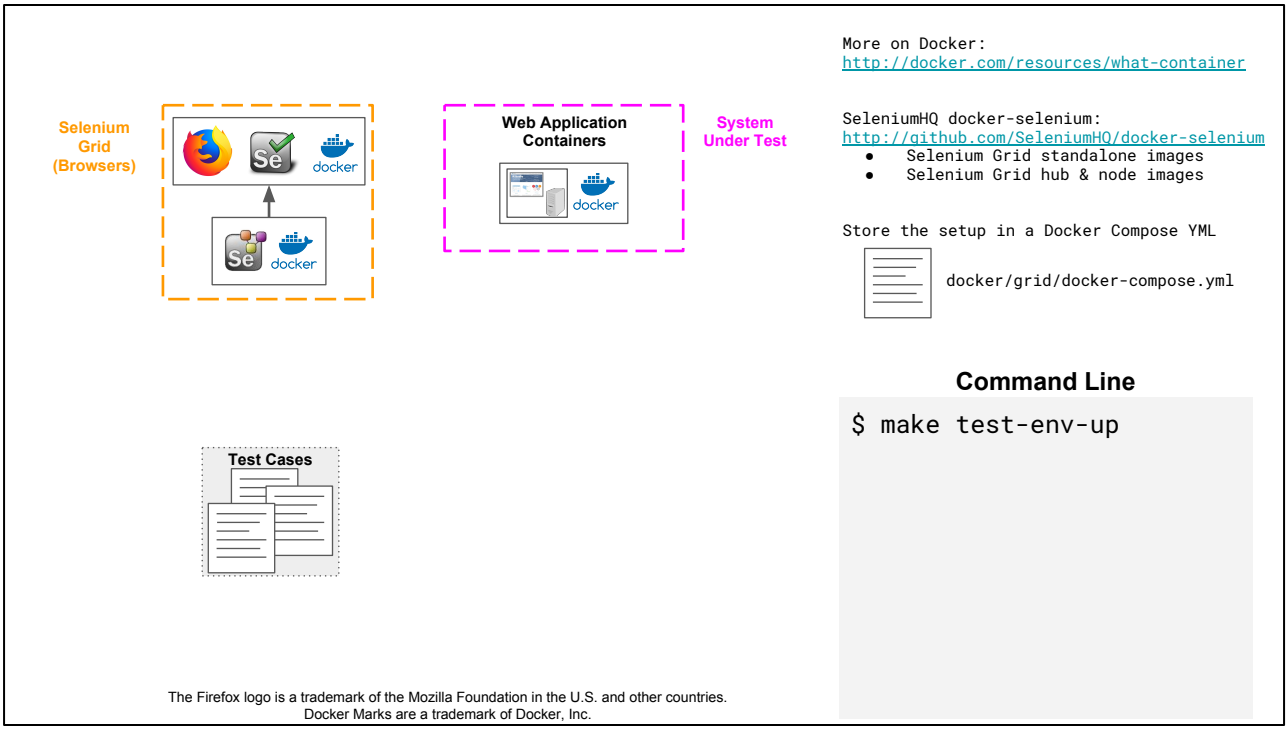
1. You no longer need to maintain the web browser and web browser driver installations on everyone's computer.

2. It sets you up to be able to run your automation scripts against external testing services like Browser Stack or Sauce Labs.

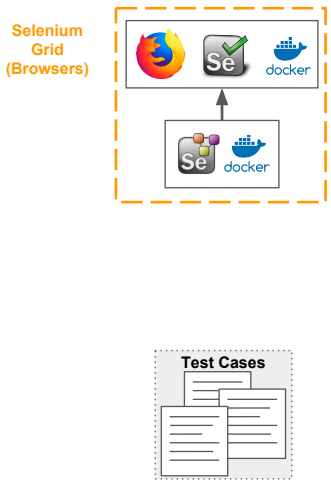
This means that you can get up and running with a Selenium Grid installation on your computer in a matter of minutes, without needing to install Java, web browsers, or web browser drivers.



In the setups my teams have been building, we put all of the Selenium Grid container configuration into a docker-compose file.
That way, we can tell Docker to bring up the system with one command.



In some of our setups, we have our system under test running in a Docker container as well. We can also bring this container up with the `make test-env-up`.



More on Docker:
http://docker.com/resources/what-container

SeleniumHQ docker-selenium:
http://github.com/SeleniumHQ/docker-selenium

- Selenium Grid standalone images
- Selenium Grid hub & node images

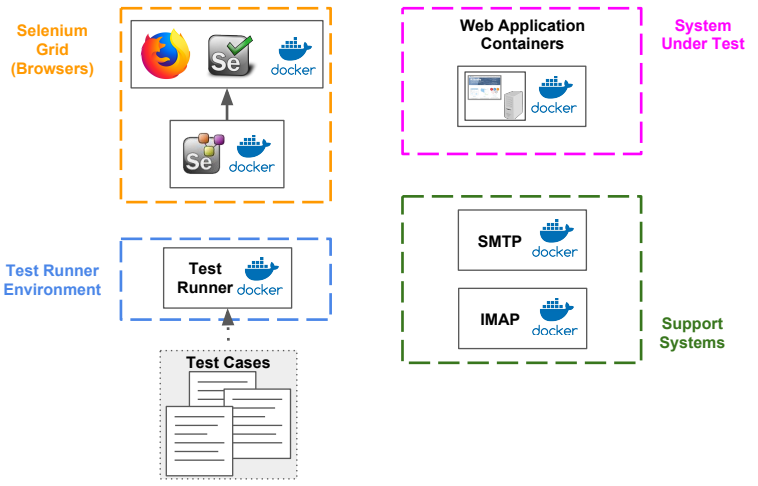
Store the setup in a Docker Compose YML

docker/grid/docker-compose.yml

Command Line

```
$ make test-env-up
```


And the `make test-env-up` command could also include other types of support services, like an LDAP server, or mail server containers with SMTP and IMAP. There are a lot of Docker images people have put together and shared through the Docker Hub registry, So if your project has a need for a service, you might try looking there to see if anyone else has created an image you can use.



Test Runner Environment:
dskard/tew:0.1.0

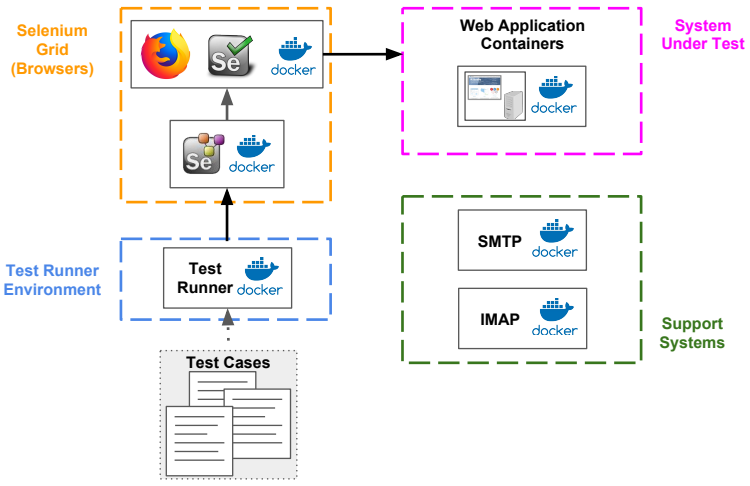
- Runs on same Docker network as Test Environment
- Access test cases via shared mount
- Contains software to run test cases
  - Python3 interpreter & debugger
  - Selenium client libraries
  - *pytest* test runner
  - bash, curl, wget

Command Line

```
$ make test-env-up
$ make test
```


The second command, `make test`, is responsible for running the test cases. Under the hood, we launch a test runner to manage the collecting and running of the test cases. The test runner program runs in its own Docker container, on the same Docker network as the Selenium Grid and other support containers, so they can easily communicate with each other by container name.

In many of the setups I work with, we have been using a Python based test runner named `pytest`. I like this test runner because it give me a lot of control over the process of running a test case, and hooks where I can add code to change that process. People have written plugins to help simplify process of writing Selenium based test cases that can be run by the test runner.
We'll talk more about those features in a little bit.



Test Runner Environment:
dskard/tew:0.1.0

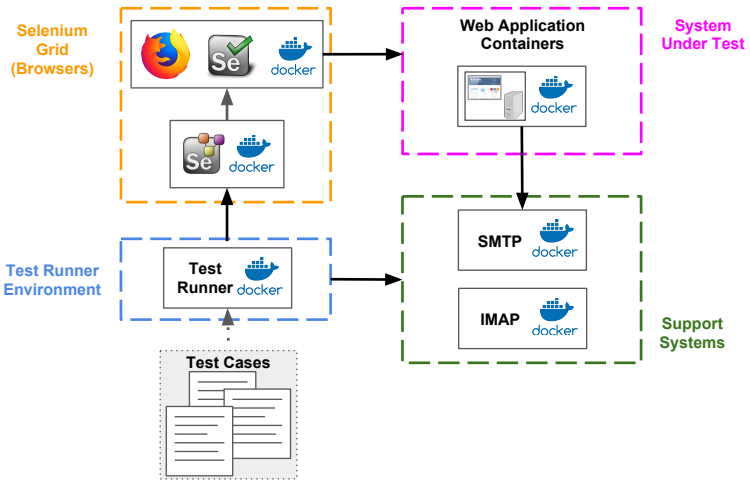
- Runs on same Docker network as Test Environment
- Access test cases via shared mount
- Contains software to run test cases
  - Python3 interpreter & debugger
  - Selenium client libraries
  - *pytest* test runner
  - bash, curl, wget

Test cases use web browsers from remote Selenium Grid.

Command Line

```
$ make test-env-up
$ make test
```


The test runner runs each test case.
When a test case needs a web browser, it contacts our Selenium Grid Hub container, which takes care of finding an available node container that matches the specified browser capabilities.



Test Runner Environment:
dskard/tew:0.1.0

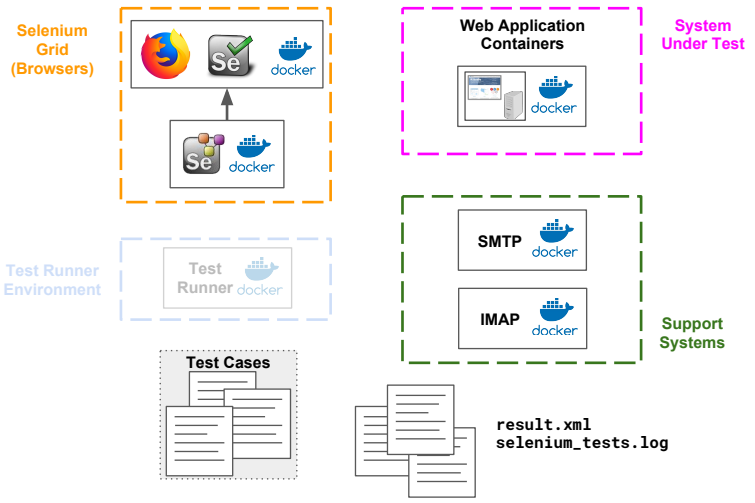
- Runs on same Docker network as Test Environment
- Access test cases via shared mount
- Contains software to run test cases
  - Python3 interpreter & debugger
  - Selenium client libraries
  - *pytest* test runner
  - bash, curl, wget

Test cases use web browsers from remote Selenium Grid.

Command Line

```
$ make test-env-up
$ make test
```


Because everything is running on the same Docker network,
when my web application communicates with any of the other containerized support systems, like to send an email for example,
my test case can also communicate with the support systems, to, for example read the email.



Test Runner Environment:
dskard/tew:0.1.0

- Runs on same Docker network as Test Environment
- Access test cases via shared mount
- Contains software to run test cases
  - Python3 interpreter & debugger
  - Selenium client libraries
  - *pytest* test runner
  - bash, curl, wget

Test cases use web browsers from remote Selenium Grid.

Container shuts down after test runner process exits.

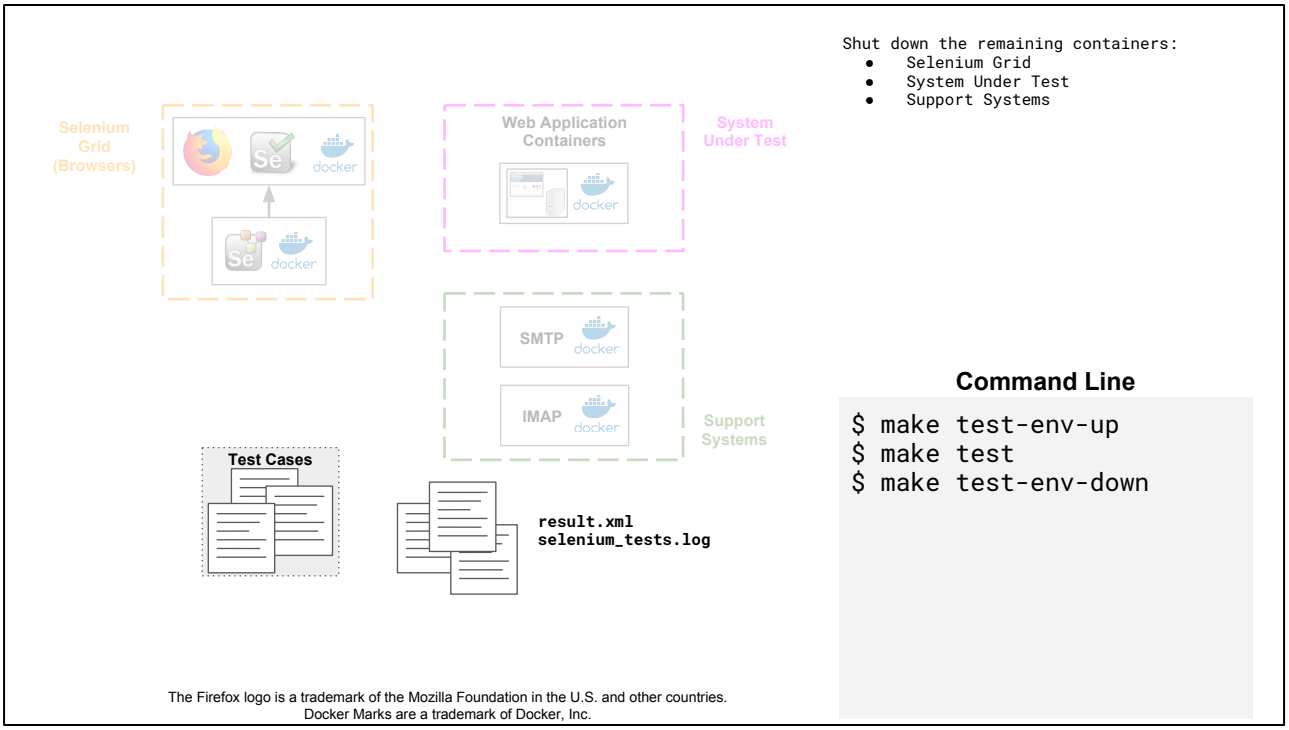
Command Line

```
$ make test-env-up
$ make test
```


After all of the test cases have run, the pytest test runner will write out two file.

1. A junit style xml file with the test session's results. This hold pass and failure information for each test case.
2. A log of stdout.

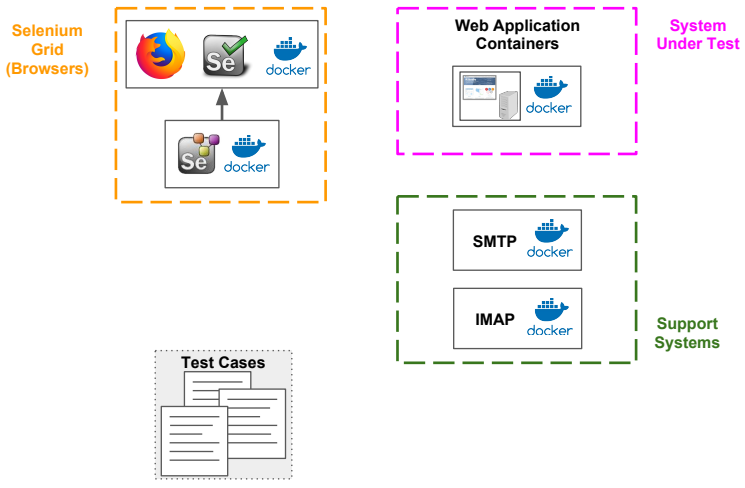
Then, it will and exit and its container will automatically shut down.
Leaving only our test environment container.



When we have finished running the test cases, we'll call our last command: `make test-env-down` to tear down the Docker containers hosting the Selenium Grid service and the other support services we launched with the `make test-env-up` command.

And that's it!

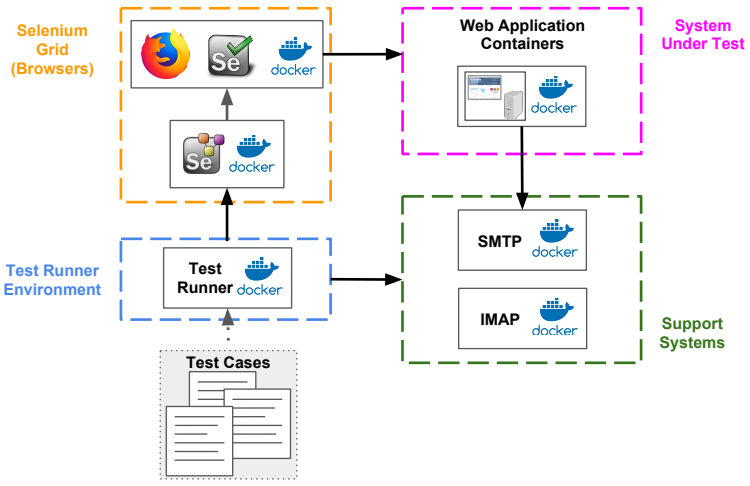
You know, let me go through this one more time, just in case you blinked...



Command Line

```
$ make test-env-up
```

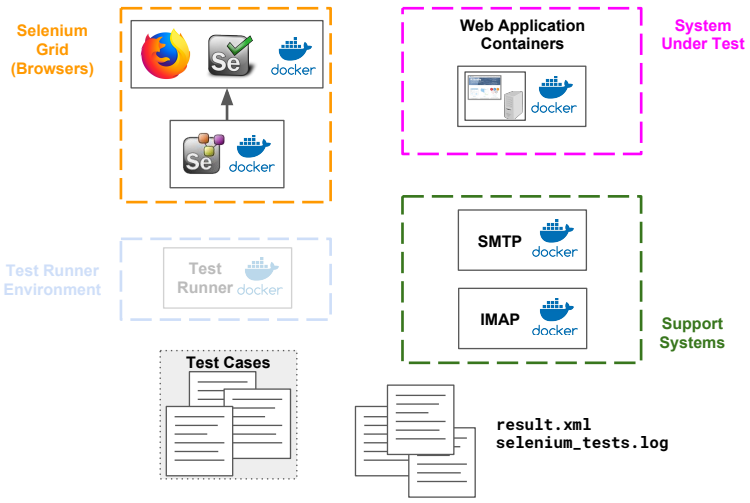

So all together, “make test-env-up” to bring up the Selenium Grid, our system under test, and our support systems



Command Line

```
$ make test-env-up
$ make test
```

“make test” launches the test runner, pytest in this case, inside of a docker container, on the same docker network as our other containers.
When a test case needs a web browser, it contacts our Selenium Grid.

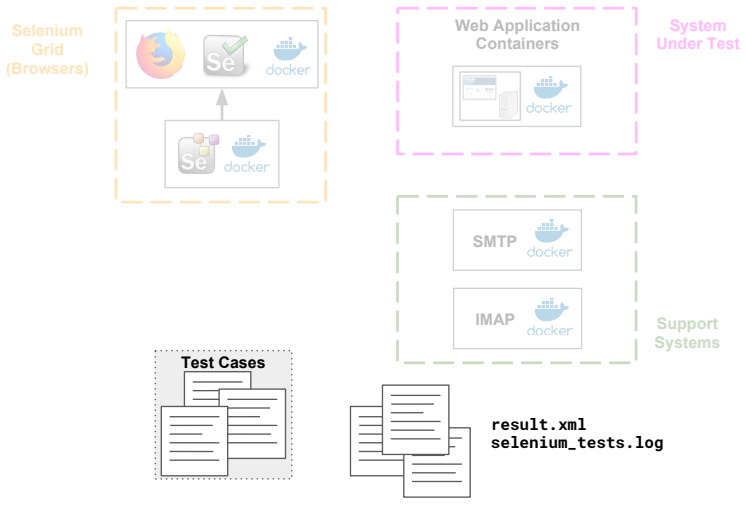


The Firefox logo is a trademark of the Mozilla Foundation in the U.S. and other countries.
Docker Marks are a trademark of Docker, Inc.

Command Line

```
$ make test-env-up
$ make test
$
```

When the test runner completes, it writes its results to disk, and exits. This shuts down the test runner container.



Command Line

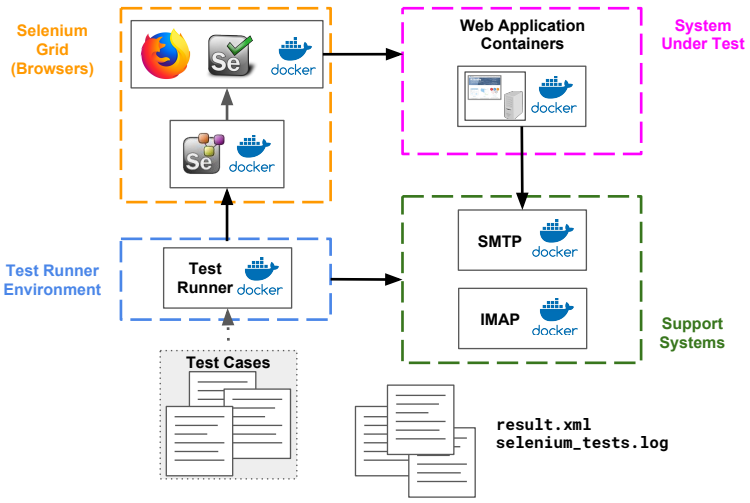
```
$ make test-env-up
$ make test
$ make test-env-down
```


Then, we run our last command, “make test-env-down” to shut down the Selenium Grid, our system under test, and our support system containers.

So with those three commands:

```
make test-env-up
make test
make test-env-down
```

You can be up and running Selenium test cases on a local Selenium Grid.
To date, setups like this have worked out for us, because most of the pieces that people need are either tools they use for development already, or are pieces that are downloaded from the internet on demand.



Grid Docker images are maintained by SeleniumHQ project.

Tools used under the hood:

- bash, coreutils, make, …
- Docker, Docker Compose

Command Line

```
$ make test-env-up
$ make test
$ make test-env-down
```


For example, these three commands are calling:

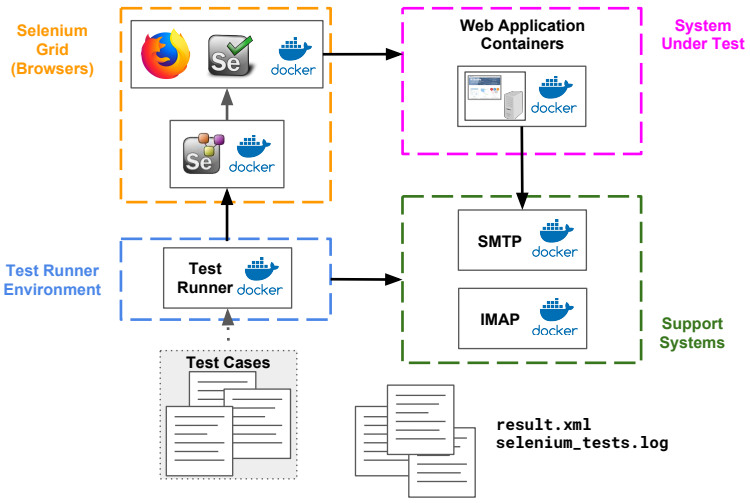
1. Bash, coreutils, make and other shell commands - which come be default on some systems or as a part of development libraries on other systems.

and

2. Docker, Docker Compose - which are libraries that many people install on their system to do their regular development.

The Selenium Grid is not installed locally, it runs in a Docker container.
The pytest test runner runs in a Docker container.
All of our support services run in Docker containers
and in some of our cases, our system under test runs in a docker container.

Before we move on, I want to talk about one thing that I kinda glazed over.
That has to do with the commands themselves.



Grid Docker images are maintained by SeleniumHQ project.

Tools used under the hood:

- bash, coreutils, **make**, …
- Docker, Docker Compose

Command Line

```
$ make test-env-up
$ make test
$ make test-env-down
```


You'll notice that our commands all start with `make`.
This is because we are using the `make` program to run the commands.

The `make` program is most commonly used to generate executables, but can also be used to hold other collections of commands or recipes. `make` reads its recipes from a file called a *Makefile*.

Makefile

```
COMMAND             = bash
SELENIUM_VERSION    = 3.8.1-dubnium
TRE_IMAGE           = dskard/tew:0.1.0
DOCKER_RUN_COMMAND  = docker run …
TEST_RUNNER_COMMAND = pytest …
...


test: wait-for-systems-up prepare-logging
    ${DOCKER_RUN_COMMAND} ${TEST_RUNNER_COMMAND} \
        > ${TMP_PIPE} || EXITCODE=$$?; \
    rm -f ${TMP_PIPE}; \
    exit $$EXITCODE

run:
    @${DOCKER_RUN_COMMAND} ${COMMAND}

test-env-up: grid-up

test-env-down: network-down

grid-up: network-up
    NETWORK=${NETWORK} \
    GRID_TIMEOUT=${GRID_TIMEOUT} \
    SELENIUM_VERSION=${SELENIUM_VERSION} \
    docker-compose -f ${DCYML_GRID} -p ${PROJECT} up \
        -d \
        --scale firefox=${SCALE_FIREFOX} \
        --scale chrome=${SCALE_CHROME}
...
```

A word about Makefiles…

- Variables declared a top
- Rules tell how to build targets
- *test-env-up*, *test*, and *test-env-down* are targets

Command Line

```
$ make test-env-up
$ make test
$ make test-env-down
```

If you look inside of a *Makefile*, you'll usually see variables listed at the top, and rules explaining how to build various targets throughout the rest of the file.

Makefile

```
COMMAND             = bash
SELENIUM_VERSION    = 3.8.1-dubnium
TRE_IMAGE           = dskard/tew:0.1.0
DOCKER_RUN_COMMAND  = docker run …
TEST_RUNNER_COMMAND = pytest …
...


test: wait-for-systems-up prepare-logging
    ${DOCKER_RUN_COMMAND} ${TEST_RUNNER_COMMAND} \
        > ${TMP_PIPE} || EXITCODE=$$?; \
    rm -f ${TMP_PIPE}; \
    exit $$EXITCODE

run:
    @${DOCKER_RUN_COMMAND} ${COMMAND}

test-env-up: grid-up

test-env-down: network-down

grid-up: network-up
    NETWORK=${NETWORK} \
    GRID_TIMEOUT=${GRID_TIMEOUT} \
    SELENIUM_VERSION=${SELENIUM_VERSION} \
    docker-compose -f ${DCYML_GRID} -p ${PROJECT} up \
        -d \
        --scale firefox=${SCALE_FIREFOX} \
        --scale chrome=${SCALE_CHROME}
...
```

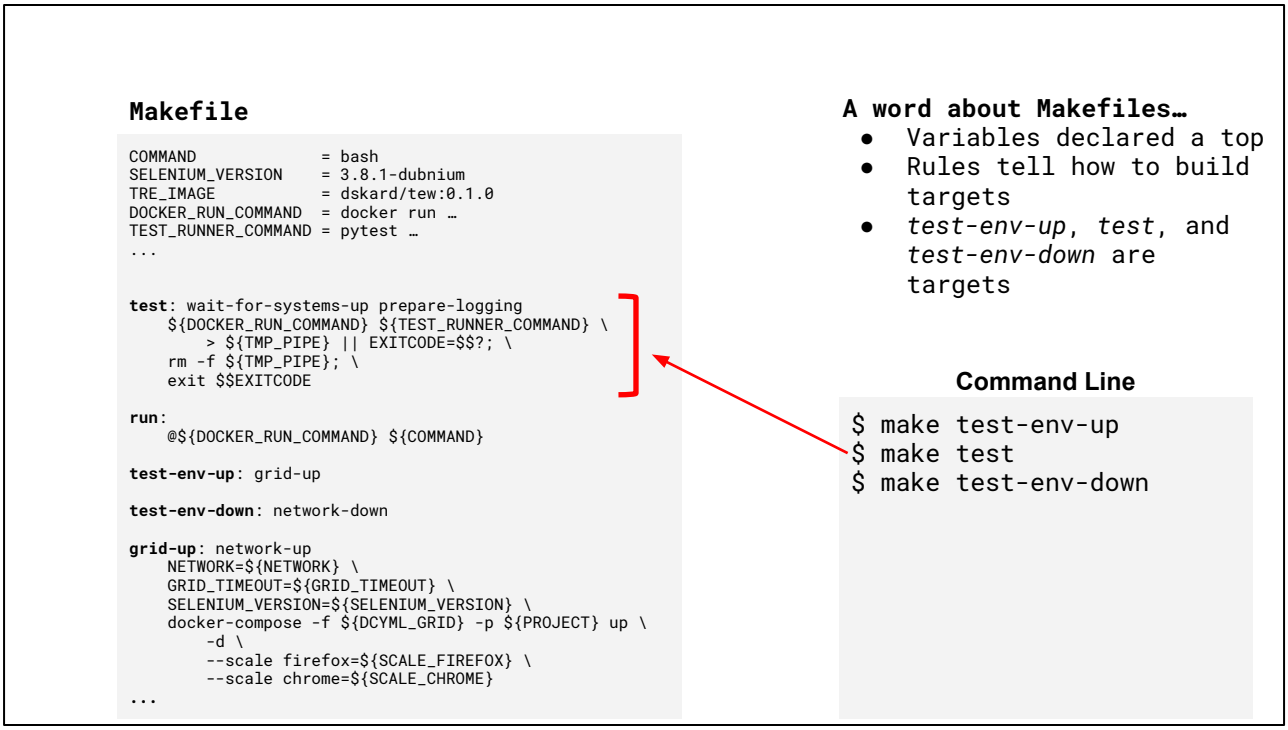
A word about Makefiles…

- Variables declared a top
- Rules tell how to build targets
- *test-env-up*, *test*, and *test-env-down* are targets

Command Line

```
$ make test-env-up
$ make test
$ make test-env-down
```

In our Makefile, you'll see that the `test-env-up` target, actually calls other targets to launch the Docker network and Selenium Grid. These are additional targets that you could call on your own if you wanted.



Each of our commands is a target in the Makefile, and the rule for that target describes what actions the target takes. A lot of projects use Makefiles to build their software. In those cases, you can add your testing targets directly to the project's Makefile. If your project uses another type of file to manage builds, these types of commands can probably translate over pretty cleanly. If your project doesn't use any programs to manage builds, you may consider using `make`.

Usually after I've told someone about the three commands, and their excitement dies down, they start asking questions related to what else they can do. So lets go through some of the common questions that come up.

Q How do you view the browsers as the tests run?

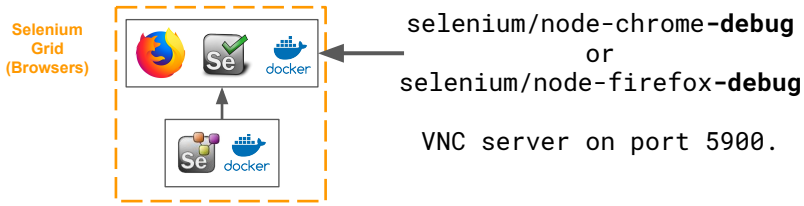
First question,

How do you view the browsers as the tests run?

This is a good questions because if you remember, our browsers are running inside of Docker containers, and I would image it is difficult to debug a browser based test case you can’t see happening.

Q

How do you view the browsers as the tests run?



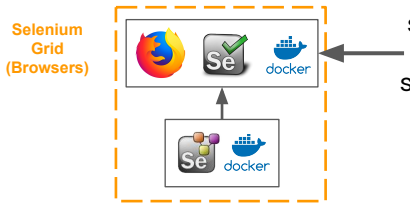
selenium/node-chrome**-debug**
or
selenium/node-firefox**-debug**

VNC server on port 5900.

Turns out there is a version of the firefox and chrome node Docker images that have this “-debug” name ending. These images

Q

How do you view the browsers as the tests run?

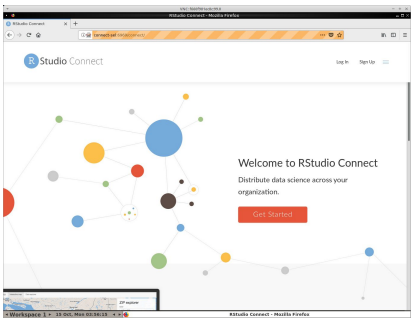


`selenium/node-chrome-debug`
or
`selenium/node-firefox-debug`

VNC server on port 5900.

Command Line (The Easy Way)

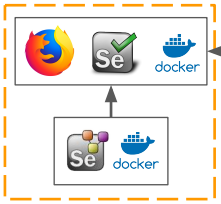
```
$ make test-env-up
$ ./shownode
```



Q

How do you view the browsers as the tests run?

Selenium Grid (Browsers)

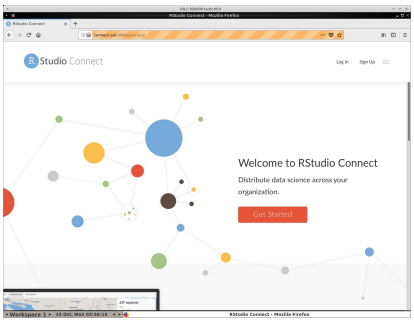


selenium/node-chrome**-debug**
or
selenium/node-firefox**-debug**

VNC server on port 5900.

Command Line (The Hard Way)

```
$ make test-env-up
$ docker ps
IMAGE                      PORTS
selenium/node-firefox...   0.0.0.0:33002->5900/tcp

$ vncviewer localhost:33002
$ # open vnc://:secret@localhost:33002
$ # password is “secret”
```



Q How do you debug test cases?

How do you debug test cases?

Q How do you debug test cases?

Test Case

```
def test_valid_login(self):
    menu = HeaderMenuFrontPage()


    menu.login.click()
    …
```

Our test cases are written in Python and you can use the same tools you would use to debug your other Python scripts.

Q How do you debug test cases?

Test Case

```
def test_valid_login(self):
    menu = HeaderMenuFrontPage()

    import pdb; pdb.set_trace()
    menu.login.click()
    …
```

In your test case, you can add a line to import pdb, the python debugger, and then set a trace.

Q

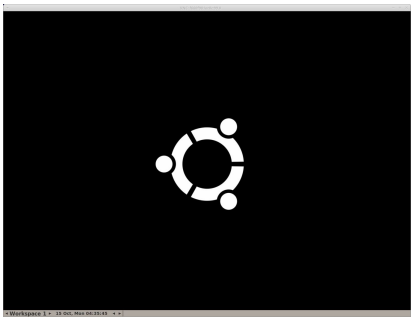
How do you debug test cases?

Test Case

```
def test_valid_login(self):
    menu = HeaderMenuFrontPage()

    import pdb; pdb.set_trace()
    menu.login.click()
    …
```

Command Line

```
$ ./shownode
$
```



Then, make sure you are running shownode so you can see the web browser while you debug

Q

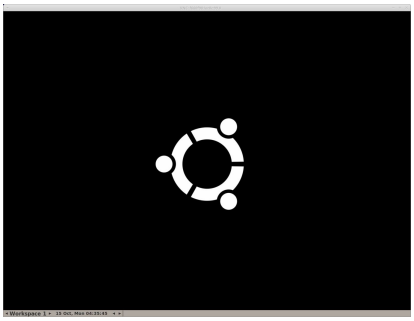
How do you debug test cases?

Test Case

```
def test_valid_login(self):
    menu = HeaderMenuFrontPage()

    import pdb; pdb.set_trace()
    menu.login.click()
    …
```

Command Line

```
$ ./shownode
$ make test PYTESTOPTS="-k test_valid_login"
```



And launch the test case. We can use the PYTESTOPTS Makefile variable to send some command line flags to the underlying pytest process, to tell it which test case to run.

Q

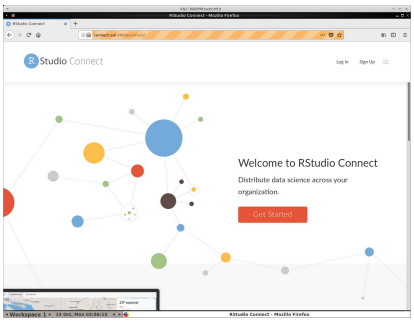
How do you debug test cases?

Test Case

```
def test_valid_login(self):
    menu = HeaderMenuFrontPage()

    import pdb; pdb.set_trace()
    menu.login.click()
    …
```

Command Line

```
$ ./shownode
$ make test PYTESTOPTS="-k test_valid_login"
…
[30] > /opt/…/test_login.py(175)test_valid_login()
-> menu.login.click()
(Pdb++)
```



The test case will be run, and when the interpreter gets to the set_trace() line, it will drop you into the debugger, where you can use your normal python debugger commands to step through the code.

Q How do you write new test cases?

I get a few questions about how to write new test cases using this setup.

Q How do you write new test cases?

- Set **DEBUG=1** so browser doesn't timeout.

Command Line

```
$ make test-env-up DEBUG=1
```

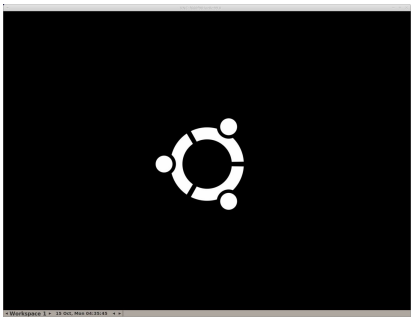
First, I start my test environment with the DEBUG=1 flag. This sets up my Selenium Grid so that the web browsers won't close automatically after a period of inactivity..

Q How do you write new test cases?

- Set **DEBUG=1** so browser doesn't timeout.

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
```



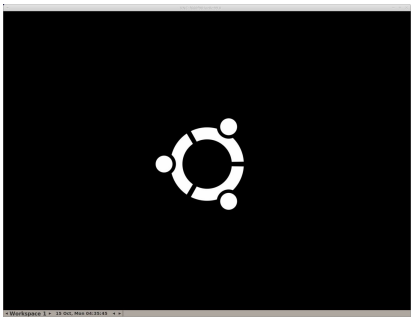
Next, I run shownode so that I can see the web browser that I'm automating

Q How do you write new test cases?

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
$ make run COMMAND=ipython3
```

- Set **DEBUG=1** so browser doesn't timeout.
- Set **COMMAND=ipython3** launches interpreter in container



Then I use the "run" target with COMMAND=ipython3 to start an interactive python interpreter inside of a Docker container.
Note that this is the same Docker container environment with the same libraries and access to browsers and the system under test that we use to run our test cases, so if my code works here, it should also work as a test case.

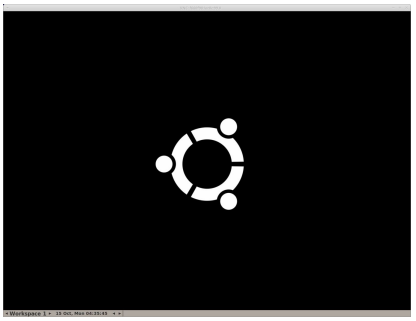
Q

How do you write new test cases?

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
$ make run COMMAND=ipython3
…
In [1]: from selenium import webdriver
```

- Set **DEBUG=1** so browser doesn't timeout.
- Set **COMMAND=ipython3** launches interpreter in container



Then, from the python interpreter, I start issuing my normal Selenium commands

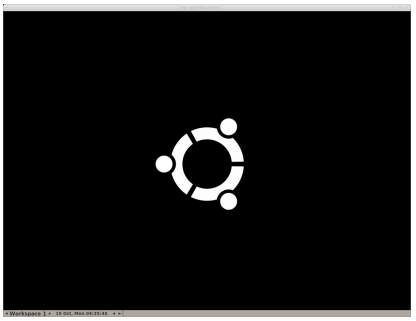
Q

How do you write new test cases?

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
$ make run COMMAND=ipython3
…
In [1]: from selenium import webdriver
In [2]: from selene.api import browser, s, be
```

- Set **DEBUG=1** so browser doesn't timeout.
- Set **COMMAND=ipython3** launches interpreter in container
- Use Selene library to wrap Selenium commands.



I also use a library named Selene, it is a wrapper library around the Selenium Python bindings.
It provides functions that help me program my test cases in a more stable way.

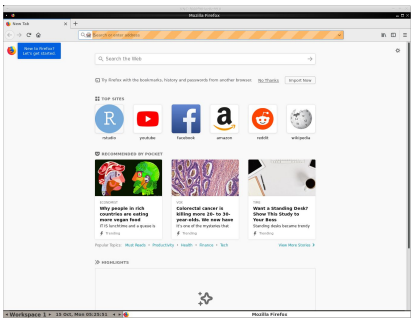
Q

How do you write new test cases?

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
$ make run COMMAND=ipython3
…
In [1]: from selenium import webdriver
In [2]: from selene.api import browser, s, be
In [3]: driver = webdriver.Remote(
             "http://selenium-hub:4444/wd/hub",
             webdriver.DesiredCapabilities\
                 .FIREFOX.copy())
```

- Set **DEBUG=1** so browser doesn't timeout.
- Set **COMMAND=ipython3** launches interpreter in container
- Use Selene library to wrap Selenium commands.



In my code, I ask for a Remote webdriver object that talks to my local Selenium Grid.

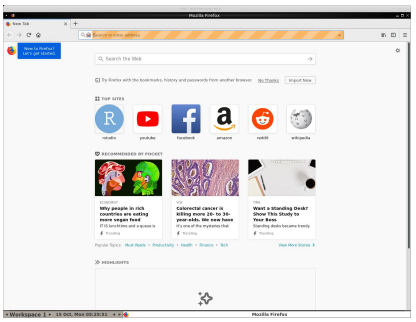
Q

How do you write new test cases?

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
$ make run COMMAND=ipython3
…
In [1]: from selenium import webdriver
In [2]: from selene.api import browser, s, be
In [3]: driver = webdriver.Remote(
            "http://selenium-hub:4444/wd/hub",
            webdriver.DesiredCapabilities\
                .FIREFOX.copy())
In [4]: browser.set_driver(driver)
```

- Set **DEBUG=1** so browser doesn't timeout.
- Set **COMMAND=ipython3** launches interpreter in container
- Use Selene library to wrap Selenium commands.



I tell the Selene library about my driver using browser.set_driver()

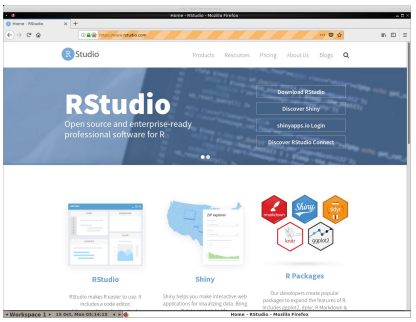
Q

How do you write new test cases?

Command Line

```
$ make test-env-up DEBUG=1
$ ./shownode
$ make run COMMAND=ipython3
…
In [1]: from selenium import webdriver
In [2]: from selene.api import browser, s, be
In [3]: driver = webdriver.Remote(
            "http://selenium-hub:4444/wd/hub",
            webdriver.DesiredCapabilities\
                .FIREFOX.copy())
In [4]: browser.set_driver(driver)
In [5]: browser.open_url('https://rstudio.com')
```

- Set **DEBUG=1** so browser doesn't timeout.
- Set **COMMAND=ipython3** launches interpreter in container
- Use Selene library to wrap Selenium commands.



And navigate to the web pages I want to interact with.

At this point, I can use my mouse and keyboard to manipulate the web browser, just as I would if it was running on my own desktop, outside of the Docker container. Once I have a good understanding of what element locators I'll need, I can translate it over to a test case function.

Q How do you write new test cases?

Test Script

```
import pytest

from selene.api import browser, s, be

…
```

Load the Selene library modules.

I start by loading my modules, I use pytest as my test runner, so I load that in.
I also use Selene, so we load the Selene libraries

Q How do you write new test cases?

Test Script

```
…
def test_valid_login(url, driver):
    """submit form with valid account info"""

    # navigate to the login page
    browser.open_url(url + '/login')

    # login with local user credentials
    s('[data-auto="username"]').set('username')
    s('[data-auto="password"]').set('password')
    s('[data-auto="submit"]').click()

    # check that no error messages are shown
    s('[data-auto="err"]').should_not(be.visible)
```

From there, most of the commands I was using in my interpreter can be copy pasted into my test function.
There are a couple of pieces worth highlighting.

Q

How do you write new test cases?

Test Script

```
…
def test_valid_login(url, driver):
    """submit form with valid account info"""

    # navigate to the login page
    browser.open_url(url + '/login')

    # login with local user credentials
    s('[data-auto="username"]').set('username')
    s('[data-auto="password"]').set('password')
    s('[data-auto="submit"]').click()

    # check that no error messages are shown
    s('[data-auto="err"]').should_not(be.visible)
```

driver fixture comes from *pytest-selenium* plugin.

Takes care of starting web browser. No need to call 'webdriver.Remote(…)'.

First, the “driver” you see here is a setup fixture that comes from the pytest-selenium plugin.
It is a plugin, built to make it easier to use Selenium from within pytest test cases.
Calling the ‘driver’ fixture takes care of starting the web browser, no need to call webdriver.Remote(...)

Q

How do you write new test cases?

Test Script

```
…
def test_valid_login(url, driver):
    """submit form with valid account info"""

    # navigate to the login page
    browser.open_url(url + '/login')

    # login with local user credentials
    s('[data-auto="username"]').set('username')
    s('[data-auto="password"]').set('password')
    s('[data-auto="submit"]').click()

    # check that no error messages are shown
    s('[data-auto="err"]').should_not(be.visible)
```

Selene's **s()** method accepts a locator and returns an object that lazily represents an element.

When we call the s() method, with a locator, no search is performed. An object is returned back to the user, storing the locator.

Q

How do you write new test cases?

Test Script

```
…
def test_valid_login(url, driver):
    """submit form with valid account info"""

    # navigate to the login page
    browser.open_url(url + '/login')

    # login with local user credentials
    s('[data-auto="username"]').set('username')
    s('[data-auto="password"]').set('password')
    s('[data-auto="submit"]').click()

    # check that no error messages are shown
    s('[data-auto="err"]').should_not(be.visible)
```

Selene's s() method accepts a locator and returns an object that lazily represents an element.

The search occurs when the **action** is performed on the element.

When we want to perform an action on the element, the search is performed and a Selenium element is retrieved, then the action is performed.
If the element is not immediately available, Selene functions perform an explicit wait for the element to show up before raising a Timeout exception.
This helps reduce the number of StaleElementExceptions developers run into.

Q

How do you write new test cases?

Test Script

```
…
def test_valid_login(url, driver):
    """submit form with valid account info"""

    # navigate to the login page
    browser.open_url(url + '/login')

    # login with local user credentials
    s('[data-auto="username"]').set('username')
    s('[data-auto="password"]').set('password')
    s('[data-auto="submit"]').click()

    # check that no error messages are shown
    s('[data-auto="err"]').should_not(be.visible)
```

Selene's s() method accepts a locator and returns an object that lazily represents an element.

The search occurs when the action is performed on the element.

should() and **should_not()** functions perform assertion of element's condition and take screenshots on failure.

In Selene, there are two functions you can use to perform assertions on elements: "should" and "should_not".
Each of these functions evaluate a condition for a certain amount of time, using an explicit wait.
If the condition ultimately evaluates to false, then an exception is raised and a screenshot of the web browser is taken for you.

The Selene library adds these types of conveniences on top of the Selenium Python bindings,
So that when we write our test case, they can look simple and easy to read.

Q How do I hook this up to Continuous Integration?

After people are comfortable writing the test cases,
they start to ask about how they can run this system through Jenkins or some other Continuous Integration system

Q How do I hook this up to Continuous Integration?

Jenkinsfile

```
…
try {

    sh 'make test-env-up'

    try {

        sh 'make test'

    } finally {
        archiveArtifacts '*.png, *.xml, *.log'
        junit '*.xml'
    }

} finally {
    sh 'make test-env-down'
}
```

Use standard commands to
- launch environment
- run tests
- clean environment

Because everything is running in Docker containers, you can use our same three commands inside of your Jenkinsfile to

1. Launch the environment
2. Run the tests
3. And shut down the environment.

https://jenkins.io/doc/pipeline/tour/tests-and-artifacts/

Q

How do I hook this up to Continuous Integration?

Jenkinsfile

```
…
try {

    sh ‘make test-env-up’

    try {

        sh ‘make test’

    } finally {
        archiveArtifacts ‘*.png, *.xml, *.log’
        junit ‘*.xml’
    }

} finally {
    sh ‘make test-env-down’
}
```

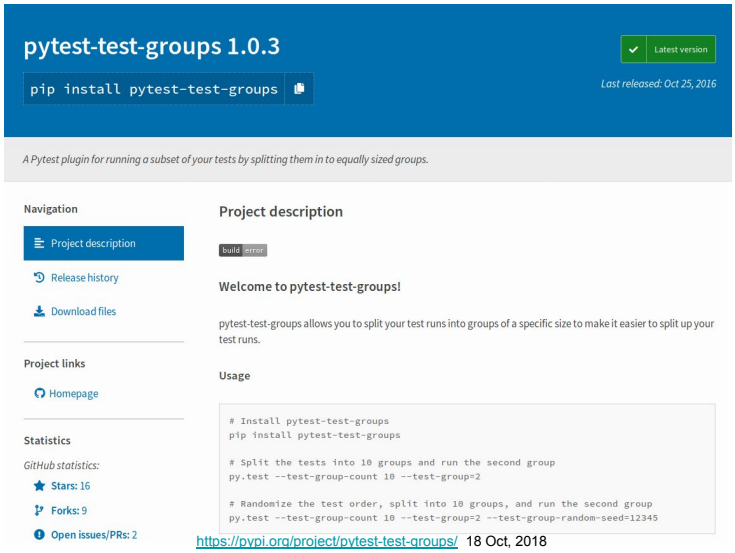
Use standard commands to
- launch environment
- run tests
- clean environment

Store:
- screenshots from failed tests (*.png)
- junit result (*.xml)
- saved stdout (*.log)

After the test cases run, you’ll probably want to save the output. You can archive all of the screenshots from test failures, result.xml files and logs.
Jenkins knows how to read the junit style result.xml file, so if you use this junit line, you’ll get a summary of the test run in Jenkins.

Q

How do I hook this up to Continuous Integration?



https://pypi.org/project/pytest-test-groups/ 18 Oct, 2018

Use standard commands to
- launch environment
- run tests
- clean environment

Store:
- screenshots from failed tests (*.png)
- junit result (*.xml)
- saved stdout (*.log)

Test Groups:
- pytest-test-groups plugin

You can also checkout this pytest plugin “pytest-test-groups” to split the test cases into groups.
This is useful if you have a you want to get started running test cases in parallel, but dont have a large Selenium Grid cluster,

Q

How do I hook this up to Continuous Integration?

Use standard commands to
- launch environment
- run tests
- clean environment

Store:
- screenshots from failed tests (*.png)
- junit result (*.xml)
- saved stdout (*.log)

Test Groups:
- pytest-test-groups plugin

It will take your test cases,

Q

How do I hook this up to Continuous Integration?

Use standard commands to
- launch environment
- run tests
- clean environment

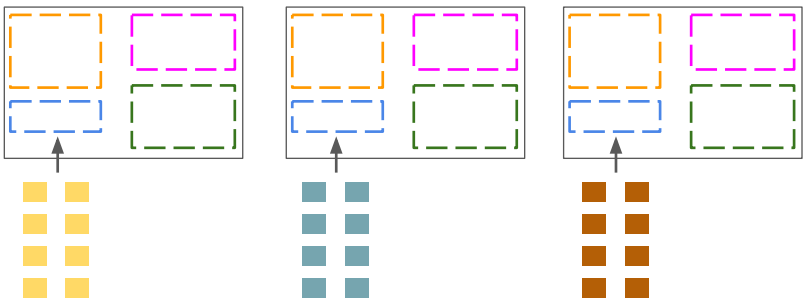
Store:
- screenshots from failed tests (*.png)
- junit result (*.xml)
- saved stdout (*.log)

Test Groups:
- pytest-test-groups plugin

And divide them into equal groups

Q

How do I hook this up to Continuous Integration?



Use standard commands to
- launch environment
- run tests
- clean environment

Store:
- screenshots from failed tests (*.png)
- junit result (*.xml)
- saved stdout (*.log)

Test Groups:
- pytest-test-groups plugin

And then you can run those groups in parallel on your smaller Selenium Grid setups. This make the setup maintainable for smaller teams.

Q

How do I hook this up to Continuous Integration?

Jenkinsfile

```
…
try {

    sh ‘make test-env-up’

    try {

        sh ‘make test PYTESTOPTS=“...”’

    } finally {
        archiveArtifacts ‘*.png, *.xml, *.log’
        junit ‘*.xml’
    }

} finally {
    sh ‘make test-env-down’
}
```

Use standard commands to
- launch environment
- run tests
- clean environment

Store:
- screenshots from failed tests (*.png)
- junit result (*.xml)
- saved stdout (*.log)

Test Groups:
- pytest-test-groups plugin

To do this, you have to specify some command line options to the pytest test runner,
And in our setup, we specify those through the Makefile variable PYTESTOPTS.

Q How do I send my tests to other Selenium Grids?

Once you're ready to expand, you can start pointing your test cases at other people's Selenium Grids.

Q

How do I send my tests to other Selenium Grids?

https://opensource.zalando.com/zalenium/ 18 Oct, 2018

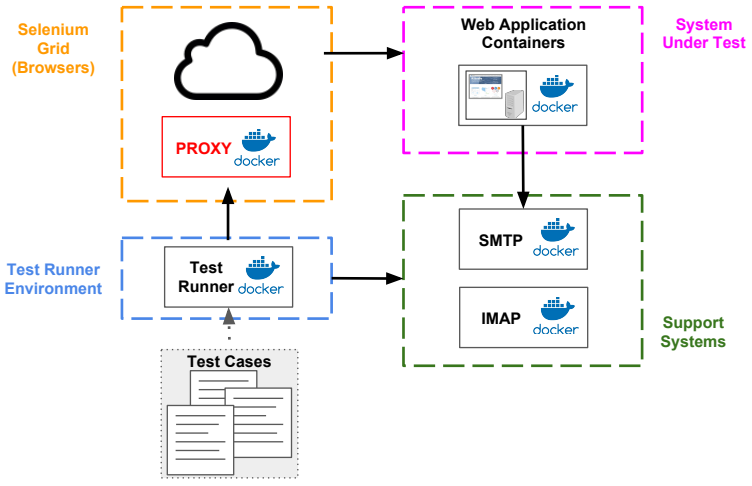
Zalenium:

- Dynamic Selenium Grid system from Zalando
- Selenium Conference Austin 2017 https://youtu.be/W5qMsVrob6I

One option for in house management is to use Zalenium,
You can hear more about that system from last year’s Selenium Conf talk

Q

How do I send my tests to other Selenium Grids?



Docker Marks are a trademark of Docker, Inc.

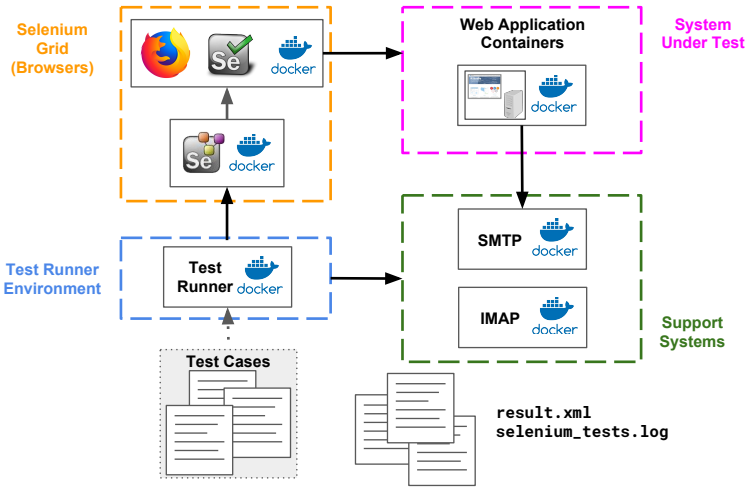
Zalenium:

- Dynamic Selenium Grid system from Zalando
- Selenium Conference Austin 2017 https://youtu.be/W5qMsVrob6I

Point your tests at an external grids through a proxy:

- Sauce Labs
- BrowserStack
- TestingBot

Or you can use a service like SauceLabs or BrowserStack that manages all of the web browser and platform combinations for your.



Command Line

```
$ make test-env-up
$ make test
$ make test-env-down
```

Let's Talk!

`office hours: 1pm`

Slides and examples:
https://github.com/dskard/seleniumconf2018

Angie talked about setting goals
One of our goals was to have an infrastructure setup that could easily be run from someone's computer
Because many of the teams I work with are small, we were also looking for low maintenance solutions that allowed us to focus our time on building tests.
Using Selenium Grid in Docker containers was a solution that fit in with the way we developed our products.
Using pytest gave us the flexibility to organize and manipulate how our test cases ran
Using tools like Make allowed us to capture the "getting started" steps in 3 commands.
Using a library like Selene allowed us simplify our test cases and avoid common pitfalls that lead to things like Stale Element Exceptions.

If you find this kind of stuff useful, have questions, or want to see a demo, let's talk! I'll be in the office hours room at 1pm and I'll post the slides and some examples to this github page.