

Automating Gmail's web interface with Helium and Selenium

A Comparative Study

Introduction

Web automation today is usually done with the Selenium framework. Selenium is great! It works on all major browsers and operating systems. It is fast, stable, well-supported and documented. But it has a problem.

When working with Selenium, you spend a lot of your time dealing with application-internal ids and technical details. Consider the following code snippet:

```
WebDriver firefox = new FirefoxDriver();
...
WebElement textArea = firefox.findElement(By.id("u_0_11"));
textArea.sendKeys("Hello World!");
WebElement button = firefox.findElement(
    By.cssSelector("button._42ft._42fu._11b.selected._42g-")
);
button.click();
```

Looking at the snippet, can you tell what it does? No? Here's a hint: It updates your Facebook status.

A new library called Helium makes Selenium-based web automation simpler by offering a more high-level approach. Using Helium, the above script can be rewritten as:

```
startFirefox();
...
write("Hello World!", into("Update Status"));
click("Post");
```

This article contrasts Selenium and Helium by automating a real world example with each. The example is to send and receive an email through Gmail's web interface. By the end of the article, you will have a good understanding of the advantages and disadvantages of web automation with Helium in comparison to pure Selenium.

Helium: A Selenium wrapper that makes web automation 50% easier

Helium is a library that simplifies Selenium-based web automation. It reduces script size by 50% and reads like everyday English. Helium support all major platforms and can easily be integrated into any existing Selenium 2 script.

High-level web automation

Helium lets you identify GUI elements by user-visible labels. It is no longer necessary to use HTML ids, XPaths or CSS selectors. You could see an example of this in the introduction, where the "Update

Status” text field and the “Post” button were identified by their labels, rather than implementation-specific properties.

Another feature of Helium’s high-level approach is its smart waiting algorithms. These algorithms deal with many of the timing issues commonly encountered when working with Selenium (ElementNotVisibleException, NoSuchElementException...). In general, it can be said that Helium does away with many of the technicalities involved in working with Selenium.

Selenium compatibility

Helium is a library that wraps around Selenium version 2. When you execute a high-level command such as `click("Post")`, Helium dynamically issues a series of Selenium commands to fulfill your request. This architecture makes it possible to freely mix calls to Helium with any Selenium 2 code. It also enables running Helium on all Selenium-supported infrastructures. In particular, you can use Helium with cloud based browser virtualization services such as BrowserStack or Sauce.

Languages and platforms

Unlike other Selenium-based frameworks, Helium aims to be available for all languages and platforms. At the time of this writing, Helium is available for Java and Python on Windows, MacOSX and Linux. When you read this article, it is likely that more platforms and languages are already supported. Please consult <http://heliumhq.com> for the latest details.

About this article

This article is a longer and updated version of an article about Helium which was published in [Issue 24 of Professional Tester](#). Professional Tester is an independent international magazine for software testers. The article was written because the main editor of Professional Tester, Edward Bishop, was interested in including an article about Helium in his magazine. Neither Professional Tester nor Edward Bishop is associated with Helium in any way.

Gmail example overview

In order to make the comparison between Helium and Selenium as fair as possible, the example that is used as a basis of the article was recommended by an independent third party, Edward Bishop. The example consists of the following steps:

Test procedure: Gmail attachment size reporting

Test inputs:

- Email address for first Gmail account (E1)
- Password for this account (P1)
- Email address for second Gmail account (E2)
- Password for this account (P2)
- Filename to be attached (A1)

Preconditions:

- File A1 is at least 2 MB (as reported by the client OS).
- File A1 is in the desktop folder of the client workstation.

Steps:

1. Start a browser.
2. Go to <http://gmail.com> and log in with L/P E1/P1.
3. Compose a new message TO E2 with SUBJECT "TP1-Gmail attachment size reporting".
4. Attach file A1 (from client desktop) to this message.
5. Note the file size of A1 as reported by Gmail.
6. Send this message.
7. Log out.
8. Delete all client browser cookies.
9. Go to <http://gmail.com> and log in with L/P E2/P2.
10. Open the message sent at step 6.
11. Verify that the reported file size of A1 is the same as reported at step 5.
12. Forward the message (including attachment) to E1.
13. Log out.
14. Delete all client browser cookies.
15. Go to <http://gmail.com> and log in with L/P E1/P1.
16. Open the message forwarded at step 12.
17. Verify that the reported file size of A1 is the same as reported at steps 5/11.
18. Close the browser.

Expected outcomes:

- All messages sent, received and opened as described in the steps.
- Reported file size of A1 is the same at steps 5, 11 and 17.

So, the test sends an email with a file attachment, checks that it is correctly received, forwards the message and then checks that also the forwarded message is correctly delivered.

Basing the example on Gmail's web interface has several advantages for the purposes of this article:

- Gmail is one of the most complex web applications publicly available, featuring AJAX and HTML5 elements. This makes it a good real world example.
- Gmail's interface was not written for the purpose of evaluating Helium and was selected by an independent third-party, hence is unlikely to present an unfair advantage for either automation framework.
- Most readers should be familiar with Gmail's interface.

There are also disadvantages to using Gmail as a benchmark:

- Gmail is continually evolving, so it can happen that the test scripts presented in this article stop working at some point in the future. Both implementations (Selenium and Helium) should strive to be as robust with respect to such changes as possible.
- Gmail is highly localized, so some readers might see their Gmail interface in a language other than the one presented here (English). The topic of localization is touched upon again in section Localization on p. 27.

Finally, both implementations shall work for current versions of Chrome, Firefox and Internet Explorer.

A note on the choice of Gmail

In response to an early version of this article, some readers have argued that Gmail's web interface does not represent a good example use case because one would normally automate Gmail not through its web interface but via an email protocol such as SMTP or IMAP. It is of course true that if the sole goal is to send some emails then one should use an email protocol and not go through the GUI. However, the purpose of this article is to compare the technical merits of two web automation solutions. For this purpose, it does not matter what the task being automated actually is. As long as the web interface which is operated is of a sufficient complexity (eg. AJAX, HTML 5) and representative of other existing web applications, the results obtained in this article should give you an indicator of what it would be like to use Helium to automate your web application.

Comparison criteria

The two implementations of the example test procedure will be evaluated according to the following criteria:

- Code length. The measures for this will be lines of code and total number of characters.
- Development effort required. This will be measured by the amount of time it took the author to complete the respective implementation.
- Performance. This will be measured by the time it takes to run an implementation on each of the three test browsers Chrome, Firefox and Internet Explorer.
- Stability across test runs.

Implementation basics

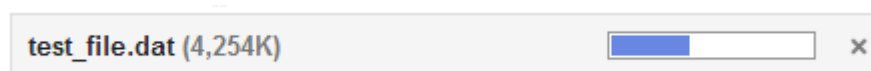
Both implementations are based on Python 2.7.5, Selenium 2.35.0, IEDriver 2.35.3.0 and ChromeDriver 2.35.0. The Helium version used is 1.0.0.

If you don't know Python, please don't worry. If you have ever worked with Selenium before, it should be easy for you to understand the gist of the code listings.

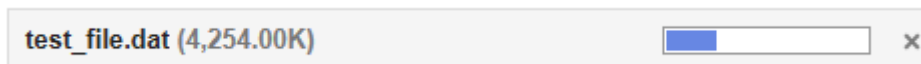
Several steps of the test procedure are duplicates of a previous step, with different parameters. For example, Steps 2, 9 and 15 all log in to Gmail, with different credentials. We will therefore not treat such steps separately, but rather describe the process of logging in with a given email address and password once per implementation.

Checking attachment sizes

One interesting observation that came out of implementing the test example for several browsers and several Gmail users is that Gmail's display of the size of an attachment varies greatly depending on the circumstances. For example, when uploading a test file A1 with size 4254 KB via Google Chrome or Firefox, its size is displayed as "4,254K":



When the same file is uploaded in Internet Explorer (set to the same locale, English), the file size is displayed as "4,254.00K":

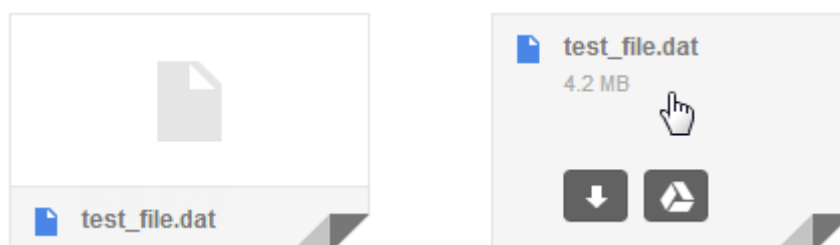


Depending on the browser and locale, other variations of the file size displayed during upload are “4 254K”, “4 254.00K”, “4 254 KB” and “4 254.00 KB”.

There are also two possibilities in the later steps 11 and 17 of the example, where the size of the attachment of a delivered email is checked. Continuing with the same test file as above, its file size as an attachment of a delivered email may be displayed as “4254K”:



If, however, the Google account that receives the email is linked to Google Drive, then the attachment size is only displayed when the user hovers over the file attachment box on the page. In this case, the attachment size is displayed as “4.2 MB”:



Dealing with differently displayed attachment sizes in the implementation

Whether attachment size is displayed as “4 254K” or “4 254.00K” or any of the other possibilities is really a nuance that we are not interested in for the purposes of this article. All we are really interested in for comparing Selenium and Helium is:

1. How they can be used to extract the text “test_file.dat (4,254K)” (or an otherwise displayed file size) from the upload dialog.
2. How they can be instructed to hover the mouse cursor over the test file name in the Google Drive case, to have the attachment size displayed.
3. How they check whether the attachment size file size in either kilobytes (“4254K”) or megabytes (“4.2 MB”) is present on the page displaying the received email.

Implementation steps

We saw above that some steps of the test procedure are duplicates of a previous step, and that both implementations need to be able to extract and check attachment sizes in a particular way. Given this discussion, we can isolate 13 non-duplicate steps that each of the two implementations needs to be able to perform:

A. Starting a test browser

The browser can either be Firefox, Internet Explorer or Google Chrome.

B. Logging in to Gmail

This step needs to be executed three times, each time with different credentials.

C. Composing a new message

The message needs to be addressed to a given recipient, and with a given subject.

D. Attaching a file to the message

The file to be attached is given by the path to A1.

E. Extracting the text containing the file name and size from the upload dialog

This step simply needs to return the text “test_file.dat (4,254K)” (or an otherwise displayed file size) from the upload dialog. An implementation should not concern itself with parsing the varying file size formats. If “test_file.dat (4,254K)” is displayed, that’s what should be returned. If “test_file.dat (4 254.00K)” is displayed then this value should be returned.

F. Sending the message

G. Logging out of Gmail

H. Deleting all client browser cookies

I. Opening a previously sent message

This implementation step serves to open both the original and the forward email. The reason why one step can be used for two different emails is that the emails are displayed very similarly in Gmail’s interface.

J. Checking the size of the file attached to the message

The implementation is given the file size in two forms: In megabytes (“4.2 MB”) and in kilobytes (“4254K”). Given this information, all the implementation needs to do is to hover the mouse cursor over the attachment name A1 and check whether one of the two forms of the file size is present on the page.

K. Forwarding the message

The message needs to be forwarded to a given address (E1).

L. Closing the test browser

By restricting each implementation to precisely the above 13 steps, we are making sure that we can compare implementations by virtue of their web automation capabilities. For example, if implementations were responsible for parsing the different file size formats in step E, then differences in the parsing solution might skew line of code measurements, which are used as one criterion for comparing the two web automation solutions.

The fact that the two implementations do not contain a solution to the file size parsing problem that is necessary for getting from step E to step J does not mean that the source code given in this article is incomplete. If you look at the source code listings in the appendix, you will see that in addition to the two implementations there is some “glue code” that makes them executable. The glue code solves the file size parsing problem using regular expressions, but the details of this solution are not relevant for the purposes of this article and so we do not discuss them here.

A. Starting the browser

Selenium implementation

To start a browser using Selenium’s Python bindings, all that is required is to import it from the appropriate module and instantiate it. We import all browser bindings at once via the line:

```
from selenium.webdriver import Chrome
```

Starting each respective browser can then be done with one of the following three lines:

```
driver = Chrome()
```

```
driver = Ie()
```

```
driver = Firefox()
```

In each of the above cases, we also want to execute the following line:

```
driver.implicitly_wait(10)
```

This tells Selenium to wait up to 10 seconds for a required GUI element to appear when it isn't immediately available. This is very useful for automatically dealing with cases where the page takes longer to load, or a GUI element only appears after a certain time.

Helium implementation

Starting a browser using Helium is similar to Selenium. First, you import from a module:

```
from helium.api import *
```

Then, you call a function:

```
start_chrome()
```

for starting Chrome, or

```
start_ie()
```

for starting Internet Explorer, or

```
start_firefox()
```

for starting Firefox.

We do not need to call `implicitly_wait(...)` as with Selenium above, because Helium automatically deals with cases where GUI elements are not immediately available.

B. Logging into Gmail

Given that a test browser has been started, this step logs into www.gmail.com with a given `email_address` and `password`.

Selenium implementation

The first step for logging into Gmail using Selenium is to navigate to `www.gmail.com`:

```
driver.get("http://www.gmail.com")
```

Note that we have to provide the full URL here (including "http://") because otherwise Selenium raises a `WebDriverException`.

Next, we check if there is a "Sign in" link on the page. If there is, we click on it:

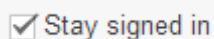
```
sign_in_links = driver.find_elements_by_link_text('Sign in')
if sign_in_links:
    sign_in_links[0].click()
```

Checking for existence of the "Sign in" link before clicking on it deals with cases where gmail.com immediately redirects to the login page rather than the default Gmail welcome page. If you have a Gmail account, it is likely that this is always the case for you. In a Selenium browsing session however, the browser cookies are usually in a clean state so that you do get to see the main gmail.com welcome page.

Once on the Sign In page, we find the text fields for entering the email address and password we want to login with. By looking at the page source, we see that they have the respective IDs 'Email' and 'Passwd', so this is what we identify them by:

```
driver.find_element_by_id('Email').send_keys(email_address)
driver.find_element_by_id('Passwd').send_keys(password)
```

The Sign In page also contains a check box that lets you select whether you want to "Stay signed in":



Unfortunately, Selenium's default way `delete_all_cookies()` of clearing cookies does not currently seem to work on Internet Explorer. We must therefore ensure that the "Stay signed in" checkbox is not selected, or otherwise we cannot log out of Gmail in IE because we are immediately logged back in with the same account.

Again by inspecting the HTML source, we see that the "Stay signed in" checkbox has the ID 'PersistentCookie'. We use this information to find and if necessary deselect the checkbox:

```
stay_signed_in = driver.find_element_by_id('PersistentCookie')
if stay_signed_in.is_selected():
    stay_signed_in.click()
```

Finally, we click on the "Sign in" button to complete the login process. We use its ID 'signIn' to identify it:

```
driver.find_element_by_id('signIn').click()
```

Helium implementation

The Helium implementation performs the same steps as the Selenium implementation for logging in. First, it opens gmail.com. Note that, unlike Selenium, Helium does not require the "http://" prefix for navigating to a URL:

```
go_to("gmail.com")
```


Next, the Helium implementation checks if the “Sign in” link exists. If yes, it clicks on it:

```
if Link("Sign in").exists():
    click("Sign in")
```

Then, it fills in the email and password. Note how no knowledge of HTML IDs is required:

```
write(email_address, into="Email")
write(password, into="Password")
```

Since Helium merely wraps around Selenium, it suffers from the same bug that it is not possible to fully clear cookies in IE via `delete_all_cookies()`. The Helium implementation of the Gmail test procedure must therefore likewise ensure that the “Stay signed in” checkbox is not selected:

```
stay_signed_in = CheckBox("Stay signed in")
if stay_signed_in.is_checked():
    click(stay_signed_in)
```

Clicking on the “Sign in” button finally completes the login process:

```
click("Sign in")
```

C. Composing a new message

This step composes a new message, addressed to a given `to_email_address` (=E2) and with a given subject (=“TP1-Gmail attachment size reporting”).

Selenium implementation

First, we need to click the “COMPOSE” button to begin writing a new email. We exploit the fact that in HTML the button is actually a `<div>` with `role=“button”` to identify it using an XPath:

```
driver.find_element_by_xpath(
    "//div[@role='button' and text()='COMPOSE']"
).click()
```

Next, we fill in the “To” field with the required email address. The field has HTML name “to”, so this is how we identify it:

```
driver.find_element_by_name('to').send_keys(to_email_address)
```

The “Subject” text box has HTML name “subjectbox”. We use this to enter the required subject:

```
driver.find_element_by_name('subjectbox').send_keys(subject)
```

This completes the present step.

Helium implementation

The Helium implementation of this step consists of three simple function calls that should hopefully be self-explanatory:

```
click(Button("COMPOSE"))
write(to_email_address, into=TextField(to_right_of="To"))
write(subject, into="Subject")
```

D. Attaching a file to the message

Selenium implementation

The contents of this section are significantly more technical than the rest of the article. If you find it too detailed, please don't hesitate to skip straight to the Helium implementation of this step, on page 15.

The normal way of initiating file uploads in Selenium is to interact with an HTML input element of type "file":

```
<input type="file" id="myFileUpload">
```

To send a file with path 'C:\test.txt' to this element, you use the `send_keys(...)` method:

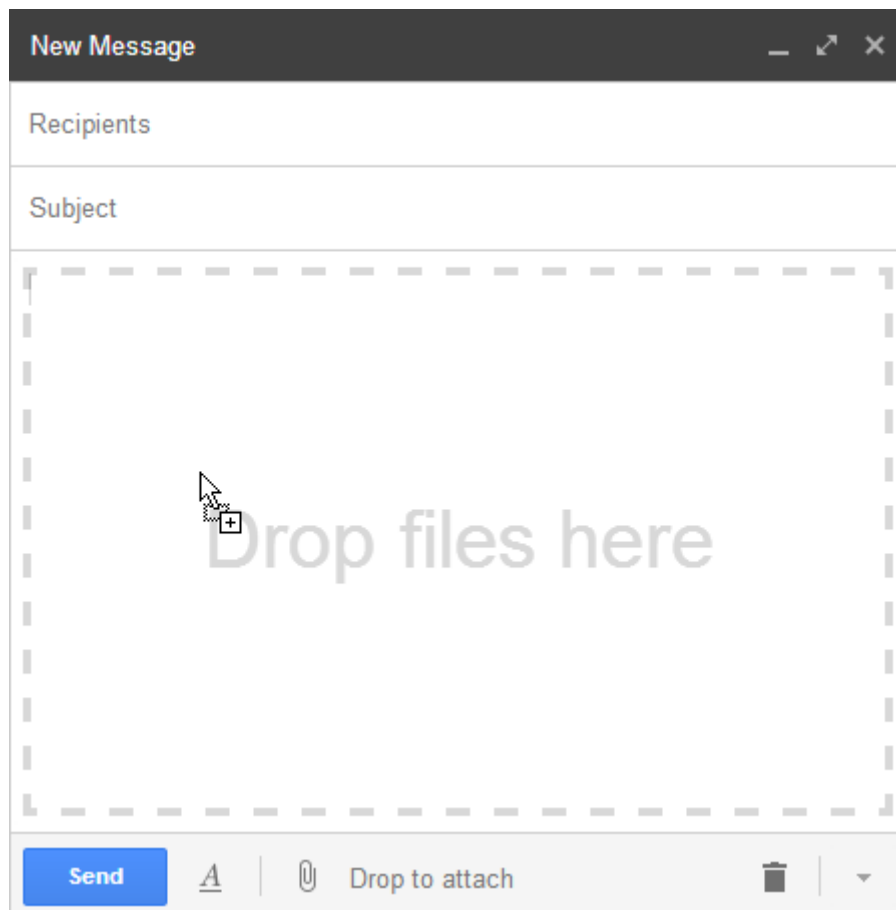
```
driver.find_element_by_id('myFileUpload').send_keys(r'C:\test.txt')
```

The problem with automating a file upload through Gmail's web interface is that there is no input element of type file when you open the page. It is only created when you click on the "Attach files" button:



Moreover, we cannot click on the "Attach files" button with Selenium because it launches a native "Open File" dialog that Selenium cannot interact with.

The solution to this problem is to perform the file upload not by clicking on the "Attach files" button and then selecting a file in the "Open" dialog. Rather, we simulate dragging the file from the desktop onto the browser window with the mouse. When you do this manually with Gmail's "COMPOSE" window open, a text is displayed saying that you can drop the file on the email being composed:



Simulating the file upload in this way overcomes the obstacle that no `<input type="file">` element is present on the page.

JavaScript events when dragging a file

The HTML5 specification [1] describes precisely which JavaScript events must be broadcast when the user drags a file onto the page. Of these events, the following are of interest when the user drags a file onto Gmail and drops it over the “Drop files here” area:

- i. A `dragenter` event on the JavaScript document object.
- ii. A `dragover` event on the JavaScript document object.
- iii. A `drop` event on the `<div>` element containing the text “Drop files here”.

If we want to programmatically perform a file upload in Gmail’s web interface, we need to simulate these events via JavaScript.

Identifying local files in JavaScript

All three events above include as a parameter a JavaScript `File` object that contains information about the file being dragged. In order to simulate the three events, we therefore also need to provide this object.

For security reasons, browsers do not allow to simply construct `File` objects. We therefore perform a little trick: We use JavaScript to create an `<input type="file">` element on the page. Then, we use Selenium’s `send_keys(...)` command to send the path of our file to this element. This automatically sets a property called `“.files”` on the file input element. The value of this property is

set to a list of File objects that gives information about the file we just “attached”. Since we can read the value of the `.files` property via JavaScript, this means that we have managed to obtain our required File object!

The JavaScript code for attaching the `<input type="file">` element to the page first creates the element and sets its type to 'file':

```
var input = document.createElement('input');
input.type = 'file';
```

Next come a few attributes that ensure that the element is visible to Selenium. We need to set these attributes in order to avoid an `ElementNotVisibleException` being raised by Selenium when we later call `send_keys(...)` on the element [2]:

```
input.style.display = 'block';
input.style.opacity = '1';
input.style.visibility = 'visible';
input.style.height = '1px';
input.style.width = '1px';
```

Finally, we need to attach the element to the DOM. We must ensure the element is at the beginning of the document, or otherwise Selenium again raises an `ElementNotVisibleException` when calling `send_keys(...)`:

```
if (document.body.childElementCount > 0) {
    document.body.insertBefore(input, document.body.childNodes[0]);
} else {
    document.body.appendChild(input);
}
```

The above JavaScript codes are sent to the browser for execution via Selenium's `execute_script(...)` command. The created file input element whose `.files` property will contain the required File object is stored in the Python variable `file_input`:

```
file_input = driver.execute_script(
    "var input = document.createElement('input');"
    "input.type = 'file';"
    "input.style.display = 'block';"
    "... (other attributes to ensure visibility)"
    "if (document.body.childElementCount > 0) {"
    "    document.body.insertBefore("
    "        input, document.body.childNodes[0]"
    "    );"
    "} else {"
    "    document.body.appendChild(input);"
    "}"
    "return input;"
)
```

The final statement `return input` ensures that the created file input can indeed be stored in a variable.

Finally, we use `send_keys(...)` to send our file to the file input element. As mentioned above, this automatically sets the `.files` property on the element:

```
file_input.send_keys(file_path)
```

Dispatching JavaScript file drag events

Now that we have access to the required `File` object, we can proceed to create and dispatch the JavaScript events that simulate the dragging and dropping of the file into the browser.

Simulating the occurrence of an event in JavaScript consists of four steps:

1. Create a `CustomEvent` object by calling `document.createEvent('CustomEvent')`.
2. Call `.initCustomEvent(...)` on the newly created event object, passing the event name (eg. "dragenter").
3. Set other properties on the event object. For dragging a file, we need to let the browser know about the file by injecting the `File` object from the previous section.
4. Call `.dispatchEvent(...)` on the object receiving the event, passing the constructed event object as a parameter. In our scenario, the objects receiving the event will be the document object (events i. and ii.) and the `<div>` containing the text "Drop files here" (event iii.).

Here is the relevant JavaScript code: First, we create the event object.

```
var event = document.createEvent('CustomEvent');
```

Next, we call `initCustomEvent`, passing in the required event name, `event_name`:

```
event.initCustomEvent(event_name, true, true, 0);
```

The values `true`, `true`, `0` are for the parameters `canBubble`, `cancelable` and `detail` and represent sensible defaults.

The next step is to link the `File` object from the previous section to the JavaScript event object. The crucial step for this is to set the property `dataTransfer.files` on the event. Assuming that the file input element from the previous section is stored in the (JavaScript) variable `file_input_element`, this can be written as follows:

```
event.dataTransfer = {  
  files: file_input_element.files  
};
```

Next, there are two further properties that need to be set on the event: `dataTransfer.items` and `dataTransfer.types`. These properties contain meta-information about the files being dragged. Setting them is neither very difficult nor very instructive for the purposes of this article, so we leave it to the interested reader to look up the corresponding source code in the appendix.

The final step of dispatching a JavaScript file drag event to an object is to call its `dispatchEvent` method with the constructed event object as a parameter. Assuming the event is to be dispatched to object `target`, this can be written as:

```
target.dispatchEvent(event);
```

A Python function for dispatching file drag events

The previous section presented some JavaScript code that can dispatch a file drag (/drop) event to a given target. We would now like to be able to call this code from our Python (/Selenium) script. Moreover, since we need to dispatch three events (“dragenter” on document, “dragover” on document, “drop” on the <div> containing “Drop files here”), we want to be able to call the event dispatch functionality three times, with varying event names and dispatch targets.

In order to fulfil the requirement of being able to call the event dispatch functionality with different parameters, we define a Python function:

```
def dispatch_file_drag_event(event_name, to, file_input_element):
```

As you can see from the above signature, the function takes the name and target of the event to be dispatched as parameters. Additionally, it expects the file input element that was created in a previous step. This is required so that the implementation can access the file input element’s .files property, which contains the File object necessary for creating the event object.

The body of the function again uses Selenium’s command `execute_script(...)` to execute the JavaScript code from the previous section. This time however, we need to pass some arguments (the event name and target as well as the file input element) from Python to the JavaScript code. This is done by supplying those values as additional parameters to `execute_script`. The JavaScript code can then access the respective values via `arguments[0]`, `arguments[1]`, etc.

The following gives the implementation of `dispatch_file_drag_event`, minus some details. Note how eg. `arguments[0]` is used in the JavaScript to refer to the `event_name` parameter passed to the Python function:

```
def dispatch_file_drag_event(event_name, to, file_input_element):
    driver.execute_script(
        "var event = document.createEvent('CustomEvent');"
        "event.initCustomEvent(arguments[0], true, true 0);"
        "event.dataTransfer = {"
        "  files: arguments[1].files"
        "};"
        "... (other code for initializing event)"
        "arguments[2].dispatchEvent(event);",
        event_name, file_input_element, to
    )
```

Putting it all together

We now have all the pieces to fully perform a file upload using drag and drop in Selenium. For the remainder of this section, let the path of the file we want to attach be given by `file_path`, and suppose that the function `create_file_input_element` creates and returns a file input element as outlined in Section Identifying local files in JavaScript (p. 11).

The first step is to create the file input element, and store it in a variable for later use:

```
file_input = create_file_input_element()
```

Next, we use `send_keys(...)` to set the file input's value. As mentioned earlier, this makes the browser fill in the value for the `.files` property that we require for creating the synthetic drag events.

```
file_input.send_keys(file_path)
```

Next, we use the `dispatch_file_drag_event` from the previous section to generate the three events that are relevant for file upload in Gmail. First, a dragenter event on the document object:

```
dispatch_file_drag_event('dragenter', 'document', file_input)
```

Then, a dragover event on the document object:

```
dispatch_file_drag_event('dragenter', 'document', file_input)
```

After the last two commands, the “Drop files here” text has appeared in Gmail's interface, just as if we had manually dragged the file over the browser. We now want to simulate the dropping of our file on the area where this text is displayed. First, we search for it:

```
drag_target = driver.find_element_by_xpath(
    "//div[text()='Drop files here']"
)
```

Then, we dispatch a final drag event to it to simulate the drop:

```
dispatch_file_drag_event('drop', drag_target, file_input)
```

Finally, we clean up after ourselves by removing the file input element we created in the first step:

```
driver.execute_script(
    "arguments[0].parentNode.removeChild(arguments[0]);", file_input
)
```

This completes our simulation of a file upload via drag and drop using Selenium.

Helium implementation

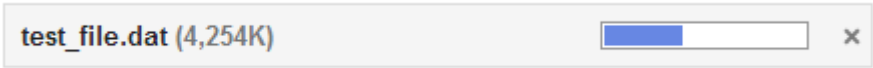
The simplicity of the Helium implementation of a file upload via drag and drop is in stark contrast to the complexity of the Selenium implementation above. Again assuming that the variable `file_path` contains the path of the file to be attached:

```
drag_file(file_path, to="Drop files here")
```

That's it. Works out of the box for all browsers.

E. Extracting the file information from the upload dialog

The goal of this step is to extract the text “test_file.dat (4,254K)” (or an otherwise displayed file size) from the upload dialog:



As mentioned before, implementations should not concern themselves with parsing the varying file size formats. If “test_file.dat (4,254K)” is displayed, that’s what should be returned. If “test_file.dat (4 254.00K)” is displayed then this value should be returned.

Selenium implementation

The Selenium implementation uses an XPath to find the div containing the file information. It then accesses the `.text` property of the resulting object to extract the text:

```
driver.find_element_by_xpath(
    "//div[starts-with(@aria-label, 'Uploading attachment: %s')]" %
    test_file_name
).text
```

The variable `test_file_name` in the above snippet contains the name of the attachment A1. For instance, in the previous image, the `test_file_name` is “test_file.dat”.

The Python modulo operator `%` on strings is used above to insert the test file name at the position specified by `'%s'` in the XPath.

Helium implementation

The Helium implementation uses the `Text(...)` predicate and its property `.value` for extracting the required text:

```
Text(test_file_name + " ").value
```

F. Sending the message

The goal of this step is to click on the “Send” button and wait until the text “Your message has been sent.” is displayed by Gmail.

Selenium implementation

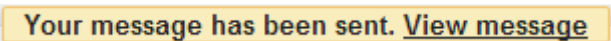
The “Send” button turns out to be a `<div>` without a static ID, so we search by `text()` and HTML tag name:

```
driver.find_element_by_xpath("//div[text()='Send']").click()
```

Then, we use Selenium’s `WebDriverWait` facilities to wait for the text “Your message has been sent.” First, we need to instantiate the `WebDriverWait` class with the driver and the timeout as parameters:

```
wait = WebDriverWait(driver, 120)
```

Note that the text that is displayed when a message was sent contains the suffix “View message”:



The XPath we use to detect whether the expected text is present thus uses the starts-with function:

```
xpath = "//div[starts-with(text(), 'Your message has been sent.')]"
```

Finally, we call the WebDriverWait's .until(...) method. We use the condition expected_conditions.presence_of_element_located to wait until the text is visible:

```
wait.until(  
    expected_conditions.presence_of_element_located(  
        By.XPATH, xpath  
    ))  
)
```

Helium implementation

Clicking on the "Send" button can be performed via:

```
click("Send")
```

Waiting for the Text "Your message has been sent" to exist can be done via Helium's command wait_until(...):

```
wait_until(  
    Text("Your message has been sent.").exists, timeout_secs=120  
)
```

G. Logging out of Gmail

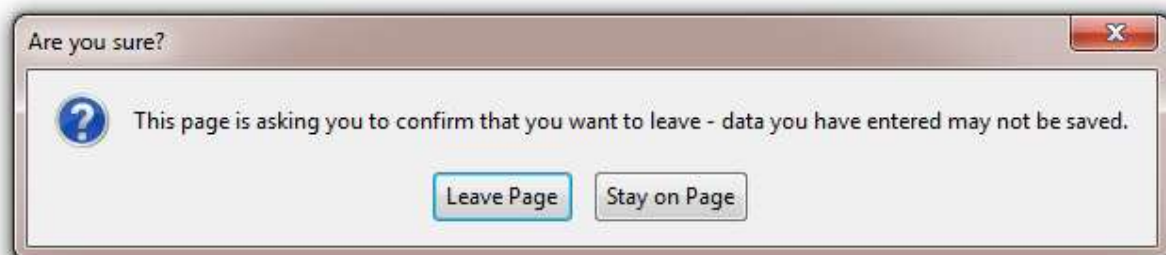
This step logs out the user that is currently logged into Gmail. The fastest and most reliable way of doing this is to simply navigate to the URL <https://mail.google.com/mail/?logout>, so that's what each implementation should do.

Selenium implementation

Navigating to the required URL can be done via the following command in Selenium:

```
driver.get('https://mail.google.com/mail/?logout')
```

In Firefox, it can happen that Gmail's interface is not yet ready to process this request. In this case, it displays the following alert:



The alert keeps us from navigating to the required URL, so we close it. The following code does this if the alert exists, and nothing otherwise:

```
try:
    driver.switch_to_alert().accept()
except (NoAlertPresentException, WebDriverException):
    pass
finally:
    driver.switch_to_default_content()
```

Helium implementation

The Helium implementation of this step starts very similarly to the Selenium one:

```
go_to('https://mail.google.com/mail/?logout')
```

The only real difference here is that Helium doesn't require referencing the driver that should open the page. The reason for this is that Helium always operates on the driver it last started (eg. `start_chrome()`), if no other driver was specified using the command `set_driver(...)`.

The alert shown in the picture above does not occur with Helium, so we don't need to close it.

H. Deleting all client browser cookies

This step deletes all cookies on the client's browser, to ensure the browser is in a clean state before logging into Gmail again.

Selenium implementation

Selenium's drivers provide the method `delete_all_cookies()` that fully implements the present step:

```
driver.delete_all_cookies()
```

Helium implementation

As mentioned in the Helium implementation of the previous step, Helium implicitly keeps track of the Selenium driver you are currently working with. This driver only changes when you start another browser instance via eg. `start_firefox(...)`, or when you call Helium's command `set_driver(...)`.

Helium's command `get_driver()` returns the currently active Selenium driver. As in the Selenium implementation above, we can therefore simply call `delete_all_cookies()` on it:

```
get_driver().delete_all_cookies()
```

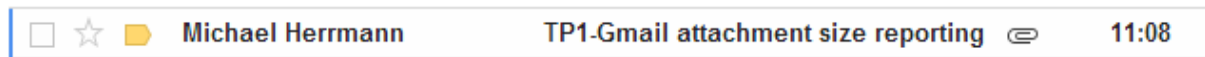
I. Opening a previously sent message

The goal of this step is to open one of the messages sent during the example test procedure. This includes the initial message as well as the forwarded copy. The reason why one implementation step can serve to open two different emails is that the emails are displayed very similarly in Gmail's

interface. All an implementation needs to do is to click on the email subject and wait for the email to be displayed.

Selenium implementation

We first need to click on the message subject in the list of emails in the inbox view:



As it turns out, the subject “TP1-Gmail attachment size reporting” is enclosed by a and then a element. We use this to find the subject via an XPath, and click on it:

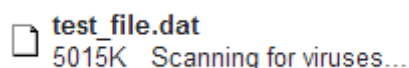
```
driver.find_element_by_xpath(
    "//span/b[text()=' %s']" % email_subject
).click()
```

Again, the Python modulo operator % is used to insert the email_subject ("TP1-Gmail attachment size reporting") into the XPath string.

Once we have clicked on the email, we need to wait until it is fully displayed. First, we wait for up to ten seconds until the attachment file name is present:

```
WebDriverWait(driver, 10).until(
    expected_conditions.presence_of_element_located((
        By.XPATH, "//*[text()=' %s']" % test_file_name
    ))
)
```

Finally, there is one more condition we need to wait for to ensure that Gmail’s interface is ready for the next step (= forwarding the opened message): The first time an email with an attachment is opened, Gmail displays the text “Scanning for viruses...” next to the attachment:



After a brief amount of time (apparently when Gmail is done scanning for viruses), the “Scanning for viruses” text is replaced by a download link:



To really ensure stability of the Gmail interface before we proceed, we thus also want to wait until the text “Scanning for viruses...” no longer exists.

The condition we want to wait for can be expressed by the following Python function:

```
def virus_scan_complete(driver):
    return not driver.find_elements_by_xpath(
        "//*[starts-with(., 'Scanning for viruses')]"
    )
```

Before we pass this condition to Selenium's `WebDriverWait`, there is one significant performance improvement: The function `virus_scan_complete` calls `find_elements_by_xpath` to determine whether the text "Scanning for viruses" still exists. When this text does not exist, then Selenium's implicit wait functionality (which we enabled in step A) waits up to 10 seconds until it returns. For the purposes of checking the condition of non-existence however, we want the query to return as quickly as possible. We therefore temporarily disable implicit waits:

```
driver.implicitly_wait(0)
```

Then, we carry out the actual wait:

```
WebDriverWait(driver, 10).until(virus_scan_complete)
```

And finally, we re-enable implicit waits:

```
driver.implicitly_wait(10)
```

Helium implementation

Like the Selenium implementation, the Helium implementation first clicks on the email subject to open the message:

```
click(email_subject)
```

(The variable `email_subject` again contains the text "TP1-Gmail attachment size reporting".)

Next, we wait for the attachment file name to be shown:

```
wait_until(Text(test_file_name).exists)
```

Finally, we wait until the text "Scanning for viruses..." has disappeared. The simplest way to do this is by using a lambda expression:

```
wait_until(lambda: not Text("Scanning for viruses...").exists())
```

The lambda construct is Python's facility for anonymous functions. In the above snippet, it allows us to pass an executable function to `wait_until(...)` which checks whether the text "Scanning for viruses" has disappeared.

J. Checking the size of the file attached to the message

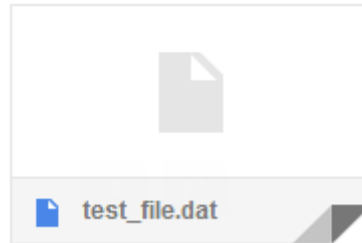
The goal of this step is to check that the attachment in a received mail (original or forwarded) has the right size.

Recall from above that the attachment size can be displayed in two different ways, depending on whether the Gmail account that received the message is linked to Google Drive or not.

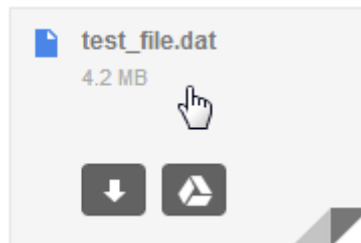
When the account receiving the message is not linked to Google Drive, attachment size is simply displayed as a number followed by "K":



If, on the other hand, the Google account receiving the message is linked to Google Drive, then the attachment is first displayed without its size:



Only when the user hovers the mouse cursor over this attachment is the file size displayed:



In this case, attachment size is displayed as a number of the form “x.x MB”.

The way of dealing with these variations is for the implementations to perform the following:

Given the expected attachment size in KB (“4254K” in the example above) and in MB (“4.2 MB” in the example), the implementation shall perform the following steps:

1. *Hover the mouse cursor over the file name of the attachment.*
2. *Check if either the expected attachment size in KB or in MB are present on the page.*

Selenium implementation

The first step is to hover the mouse cursor over the attachment file name. Depending on whether the Google account is or isn't linked to Google Drive, this may either be given by a <td> and tag enclosing a text that starts with the file name, or by a span containing precisely the name. This distinction is present in the XPath used search for the attachment file name:

```
test_file = driver.find_element_by_xpath(
    "//td/b[starts-with(text(), '%s')]" % test_file_name + ' | ' + \
    "//span[text()=' %s']" % test_file_name
)
```

Once the file name has been found, we need to hover over it. The way to do this using Selenium is via so-called *ActionChains*:

```
ActionChains(driver).move_to_element(test_file).perform()
```

Finally, we need to check whether the correct attachment size is being displayed. As discussed above, the implementation is given the attachment size to look for in KB and in MB and merely needs to check for the existence of one of the two sizes on the page. Concretely, the implementation is given the variables `file_size_in_kb` and `file_size_in_mb`. In the example from the preceding discussion, these variables would contain the values "4254K" and "4.2 MB", respectively.

First, as a performance improvement, we temporarily disable Selenium's implicit waits that we enabled in step A:

```
driver.implicitly_wait(0)
```

To check whether the file size in KB is displayed, we use an XPath based on the observation that the file size in KB is enclosed by a `<td>` element. We use `find_elements_by_xpath(...)` to obtain a list of all elements satisfying this XPath. If the length of this list is greater than zero, then we know the file size in KB is indeed being displayed. We store the result of this comparison in a variable called `file_size_in_kb_exists`:

```
file_size_in_kb_exists = len(driver.find_elements_by_xpath(
    "//td[contains(., '%s')]" % file_size_in_kb
)) > 0
```

The check whether the file size in MB is being displayed proceeds exactly analogously:

```
file_size_in_mb_exists = len(driver.find_elements_by_xpath(
    "//td[contains(., '%s')]" % file_size_in_mb
)) > 0
```

Next, we need to assert that one of the two file size variations exists, and fail the automated test if not. If you look at the source code in the appendix, you will see that both implementations are based on Python's `unittest` framework. This framework is the default unit testing library used for Python and represents the analogue of JUnit for Java, or NUnit for C#. The way of asserting that a Boolean condition holds in this framework is via the `assertTrue(...)` method:

```
assertTrue(file_size_in_kb_exists or file_size_in_mb_exists)
```

This statement checks that at least one of the two file size variations exists, and fails the automated test otherwise.

Finally, we re-enable implicit waits:

```
driver.implicitly_wait(10)
```

Helium implementation

Hovering over the test file name in the Helium implementation is done using the `hover(...)` command instead of the `ActionChains` API:

```
hover(test_file_name)
```

(Recall that the variable `test_file_name` contains the file name of the attachment A1. In the examples given so far, this has always been “test_file.dat”).

Checking whether the file size in KB is being displayed utilizes the variable `file_size_in_kb` passed to the test, and Helium’s `Text(...)` predicate:

```
file_size_in_kb_exists = Text(file_size_in_kb).exists()
```

The check for the size in MB is analogous:

```
file_size_in_mb_exists = Text(file_size_in_mb).exists()
```

The final assertion whether one of the two texts actually exists is exactly the same as for the Selenium implementation:

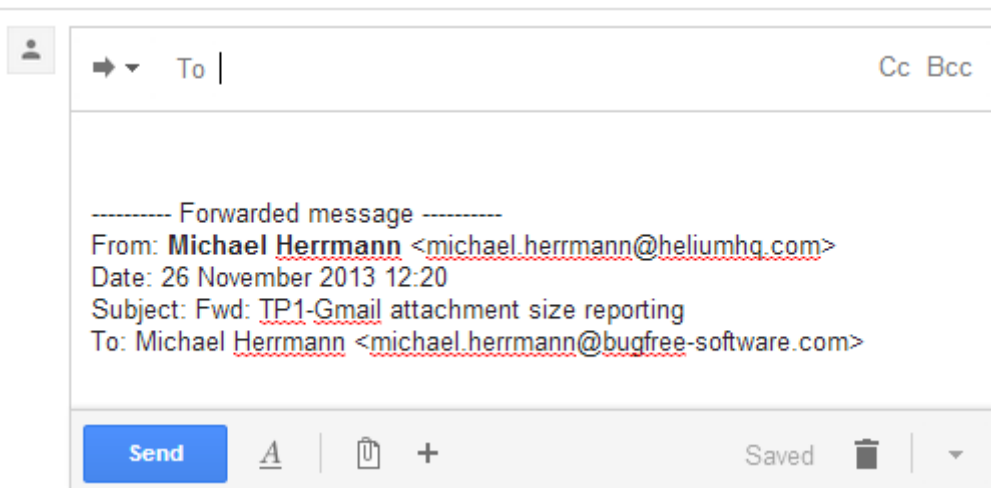
```
assertTrue(file_size_in_kb_exists or file_size_in_mb_exists)
```

K. Forwarding the message

This step needs to click on the “Forward” link below the message:



Then, it needs to fill in the recipient address and click “Send”:



The recipient address is given by E1 from the test procedure. We will refer to it from the Python code via the variable `email_address`.

Selenium implementation

As it turns out, the “Forward” link is not an `` element, but a `` with `role="link"`. We use this in the XPath that finds the link, and click on it:

```
driver.find_element_by_xpath(
    "//span[text()='Forward' and @role='link']"
).click()
```

Next, we fill in the recipient address. The “To” field has HTML name “to”, so this is how we identify it:

```
driver.find_element_by_name('to').send_keys(email_address)
```

Then, we click on the “Send” button. It’s a <div>:

```
driver.find_element_by_xpath("//div[text()='Send']").click()
```

Finally, we need to wait until the message has been sent. The implementation of this is exactly the same as in step F that sent the original message:

```
wait = WebDriverWait(driver, 10)
xpath = "//div[starts-with(text(), 'Your message has been sent.')]"/>
wait.until(
    expected_conditions.presence_of_element_located((
        By.XPATH, xpath
    ))
)
```

Helium implementation

We first click on the “Forward” link via the following command:

```
click(Link("Forward"))
```

Then, we fill in the recipient address via:

```
write(email_address, into="To")
```

Next, we send the message:

```
click("Send")
```

Finally, we wait until the sending is complete:

```
wait_until(Text("Your message has been sent.").exists)
```

L. Closing the test browser

This step closes the browser at the end of a test run. If you look at the source code in the appendix, you will notice that this step is performed in the `tearDown()` method of the two implementations. This special method is called by Python’s `unittest` framework at the end of each test case (even if unsuccessful). By closing the browser from within this method, we thus ensure that it is closed even when the automated test fails.

Selenium implementation

The Selenium implementation of this step is very simple:

```
driver.quit()
```

Note however that we do need to call the `.quit()` method and not the (closely-related) `.close()`, which only closes the currently open window.

Helium implementation

Helium uses the command `kill(...)` to terminate a browser session:

```
kill(get_driver())
```

The motivation for the name “kill” is to be a little more explicit regarding the forcefulness of the termination, for instance regarding open alert dialogs.

Comparison of implementations

Throughout this article, we have gone through painstaking effort to ensure we really are comparing only the differences in web automation capabilities between Helium and Selenium. We first distilled the original 18 steps of the example test procedure down to the 13 implementation steps A – L. Next, we noted the existence of some “glue code” that connects these steps together to automate the required test procedure. Finally, we incrementally gave two clearly separated implementations of the steps – one with Helium and one with Selenium.

When you look at the Python sources in the appendix, you will see that they consist of three files: `gmail_base.py` contains the glue code mentioned above. `gmail_helium.py` and `gmail_selenium.py` on the other hand contain precisely the code for automating the 13 implementation steps using Helium and Selenium, respectively. Separating the glue and implementation codes in this clean way allows us to unbiasedly compare and contrast the two implementations in the present chapter.

Code length

The code for automating the thirteen implementation steps using Selenium wholly resides in file `gmail_selenium.py`. It consists of 192 lines and 7294 characters.

The code for automating the implementation steps using Helium is fully contained in file `gmail_helium.py`. It consists of 64 lines and 2287 characters.

Comparing the above numbers, we see that the Helium implementation requires 66% fewer lines of code and 68% fewer characters. This means that for the example considered in this article, Helium significantly exceeds its goal of reducing web script size by 50%.

Development effort required

The development time taken to implement the example (without glue code) was roughly 0.5 days for the Helium implementation, and 2 days for the Selenium implementation. Of the two days

implementing the Selenium implementation, about one day was spent implementing the file drag-and-drop functionality.

Performance

The performance of the two implementations was compared by timing three runs of the example test procedure on each test browser. The test was performed on a laptop with 4 GB of RAM and an Intel i5 CPU with 2.67GHz running 64 bit Windows 7. The browser versions used were 31.0.1650.57 m for Google Chrome, 25.0 for Firefox and 10.0.9200.16736 for Internet Explorer. The test file size was kept constant across runs, at 2457KB. The internet connection of the test system achieved a measured download speed of 27 Mbit/s and upload speed of 9.5 Mbit/s on the day the tests were performed.

The Selenium implementation timings were as follows:

	Run 1	Run 2	Run 3
Chrome	44.4s	44.7s	42.6s
Firefox	56.1s	61.3s	57.7s
Internet Explorer	82.2s	86.5s	82.5s

The timings for the Helium implementation were:

	Run 1	Run 2	Run 3
Chrome	53.0s	51.7s	56.8s
Firefox	68.3s	63.9s	64.7s
Internet Explorer	118.0s	120.8s	120.8s

Taking the averages across runs, we get the following figures:

	Chrome	Firefox	Internet Explorer
Selenium	43.9s	58.3s	83.7s
Helium	53.8s	65.6s	119.8s

So, the Helium implementation is on average slower by 18% on Chrome, 11% on Firefox and 30% on Internet Explorer.

It must be pointed out that the Selenium implementation is optimized for performance, especially by temporarily disabling implicit waits in steps I and J. Earlier versions of the Selenium implementation which did not do this temporary disabling were significantly slower than the Helium implementation in the performance test.

Stability across test runs

Both implementations seem to be equally stable as all 18 test runs of the performance evaluation above succeeded without a single failure.

Limitations of Helium

This chapter discusses some known limitations of Helium, most of which have not yet been mentioned in this article.

Languages and platforms

As already mentioned at the beginning of this article, Helium is currently only available for Python and Java. Support for other languages will gradually be rolled out through 2014 / 2015.

Localization

There is one case where object identification by IDs is more robust than using user-visible labels: when the interface to be automated is displayed in several languages. When automating a web interface that may present itself in different languages using Helium, you have two options: Either you revert to using Selenium and IDs to identify elements (you can still pass them to Helium's functions `click(...)`, `write(...)` etc.). Or you use a so-called *translation table*. This technique uses external files for different translations and was for instance discussed in an earlier issue of Professional Tester [3].

Cost (discount code)

Unlike Selenium, Helium is a commercial product. You can purchase licenses from <http://heliumhq.com/purchase>.

Summary

This article introduced Helium, a Selenium-based web automation tool. We listed Helium's main features and showed how it can be used to automate Gmail's web interface. We noted that you would normally not automate Gmail through its web interface, but that for the evaluation purposes of this article it is actually a good choice. We also gave a pure Selenium implementation of the Gmail example, to be able to objectively compare the two web automation frameworks.

In the direct comparison, we saw that Helium requires 66% less code and 75% less development effort than Selenium to implement the Gmail example. On the other hand, Helium is 10-30% slower than an optimized Selenium script, with the caveat that naive implementations using Selenium can easily end up being slower than a corresponding Helium script. The only measure where the two implementations were equal was stability, with both approaches performing exceptionally well.

We pointed out several limitations of Helium, including currently available language bindings. We also touched on the issue of heavily localized web interfaces, which take away some of the usefulness of Helium's high-level approach.

Given its advantages, should you switch to Helium entirely? Possibly. To find out, you can download a free trial from <http://heliumhq.com/download>. It literally takes one minute to set up.

Happy automating!

References

- [1] W3 Consortium, "Drag-and-drop processing model," [Online]. Available: <http://www.w3.org/html/wg/drafts/html/master/editing.html#current-drag-operation>. [Accessed 25 November 2013].
- [2] "Selenium 2.0 : Element is not currently visible - StackOverflow," 11 June 2011. [Online]. Available: <http://stackoverflow.com/a/6317358/1839209>. [Accessed 28 November 2013].
- [3] E. Bishop, "When words are better than names," *Professional Tester*, pp. 4-6, April 2011.

Appendix: Source code listings

To run the source code presented here, simply create all files with their given names in one directory. Then, edit the `parameters.ini` file with your Gmail credentials and place the test file on your desktop. Provided you have all required dependencies (Python and Helium), you can then run the tests from this directory by typing the following on the command line:

```
python -m unittest gmail_selenium gmail_helium
```

parameters.ini

```
[parameters]
email_address_1 = email1@gmail.com
password_1 = password1
email_address_2 = email2@gmail.com
password_2 = password2
test_file_name = test_file.dat
```

gmail_helium.py

```
from helium.api import *
from gmail_base import GmailExampleTest

class GmailExampleHelium(GmailExampleTest):
    def test_chrome(self):
        start_chrome()
        self.check_attachment_sizes()
    def test_ie(self):
        start_ie()
        self.check_attachment_sizes()
    def test_firefox(self):
        start_firefox()
        self.check_attachment_sizes()
    def log_in_to_gmail(self, email_address, password):
        go_to("gmail.com")
        if Link("Sign in").exists():
```

```

        click("Sign in")
        write(email_address, into="Email")
        write(password, into="Password")
        stay_signed_in = CheckBox("Stay signed in")
        if stay_signed_in.is_checked():
            click(stay_signed_in)
        click("Sign in")
    def compose_new_message(self, to_email_address, subject):
        click(Button("COMPOSE"))
        write(to_email_address, into=TextField(to_right_of="To"))
        write(subject, into="Subject")
    def attach_file_to_message(self, file_path):
        drag_file(file_path, to="Drop files here")
    def get_uploaded_file_info(self):
        return Text(self.test_file_name + " (").value
    def send_message(self):
        click("Send")
        wait_until(
            Text("Your message has been sent.").exists,
            timeout_secs=120
        )
    def log_out_from_gmail(self):
        go_to('https://mail.google.com/mail/?logout')
    def delete_all_client_cookies(self):
        get_driver().delete_all_cookies()
    def open_sent_message(self):
        click(self.email_subject)
        wait_until(Text(self.test_file_name).exists)
        wait_until(
            lambda: not Text("Scanning for viruses...").exists()
        )
    def assert_attachment_size_is(
        self, file_size_in_kb, file_size_in_mb
    ):
        hover(self.test_file_name)
        file_size_in_kb_exists = Text(file_size_in_kb).exists()
        file_size_in_mb_exists = Text(file_size_in_mb).exists()
        self.assertTrue(
            file_size_in_kb_exists or file_size_in_mb_exists
        )
    def forward_message_to(self, email_address):
        click(Link("Forward"))
        write(email_address, into="To")
        click("Send")
        wait_until(Text("Your message has been sent.").exists)
    def tearDown(self):
        kill(get_driver())
        super(GmailExampleHelium, self).tearDown()

```

gmail_selenium.py

```

from gmail_base import GmailExampleTest
from selenium.common.exceptions import NoAlertPresentException, \
    WebDriverException
from selenium.webdriver import *
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions

```

```

class GmailExampleSelenium(GmailExampleTest):
    def test_chrome(self):
        self.driver = Chrome()
        self.driver.implicitly_wait(10)
        self.check_attachment_sizes()
    def test_ie(self):
        self.driver = Ie()
        self.driver.implicitly_wait(10)
        self.check_attachment_sizes()
    def test_firefox(self):
        self.driver = Firefox()
        self.driver.implicitly_wait(10)
        self.check_attachment_sizes()
    def log_in_to_gmail(self, email_address, password):
        self.driver.get("http://www.gmail.com")
        sign_in_links = \
            self.driver.find_elements_by_link_text('Sign in')
        if sign_in_links:
            sign_in_links[0].click()
        self.driver.find_element_by_id('Email').\
            send_keys(email_address)
        self.driver.find_element_by_id('Passwd').\
            send_keys(password)
        stay_signed_in = \
            self.driver.find_element_by_id('PersistentCookie')
        if stay_signed_in.is_selected():
            stay_signed_in.click()
        self.driver.find_element_by_id('signIn').click()
    def compose_new_message(self, to_email_address, subject):
        self.driver.find_element_by_xpath(
            "//div[@role='button' and text()='COMPOSE']"
        ).click()
        self.driver.find_element_by_name('to').\
            send_keys(to_email_address)
        self.driver.find_element_by_name('subjectbox').\
            send_keys(subject)
    def attach_file_to_message(self, file_path):
        file_input = self._create_file_input_element()
        file_input.send_keys(file_path)
        self._dispatch_file_drag_event(
            'dragenter', 'document', file_input
        )
        self._dispatch_file_drag_event(
            'dragover', 'document', file_input
        )
        drag_target = self.driver.find_element_by_xpath(
            "//div[text()='Drop files here']"
        )
        self._dispatch_file_drag_event(
            'drop', drag_target, file_input
        )
        self.driver.execute_script(
            "arguments[0].parentNode.removeChild(arguments[0]);",
            file_input
        )
    def _create_file_input_element(self):
        return self.driver.execute_script(
            "var input = document.createElement('input');",
            "input.type = 'file';"
        )

```

```

        "input.style.display = 'block';"
        "input.style.opacity = '1';"
        "input.style.visibility = 'visible';"
        "input.style.height = '1px';"
        "input.style.width = '1px';"
        "if (document.body.childElementCount > 0) { "
        "    document.body.insertBefore("
        "        input, document.body.childNodes[0]"
        "    );"
        "} else { "
        "    document.body.appendChild(input);"
        "}"
        "return input;"
    )
def _dispatch_file_drag_event(
    self, event_name, to, file_input_element
):
    script = \
        "var files = arguments[0].files;" \
        "var items = [];" \
        "var types = [];" \
        "for (var i = 0; i < files.length; i++) {" \
        "    items[i] = {kind: 'file', type: files[i].type};" \
        "    types[i] = 'Files';" \
        "}" \
        "var event = document.createEvent('CustomEvent');" \
        "event.initCustomEvent(arguments[1], true, true, 0);" \
        "event.dataTransfer = {" \
        "    files: files," \
        "    items: items," \
        "    types: types" \
        "};" \
        "arguments[2].dispatchEvent(event);"
    if isinstance(to, basestring):
        script = script.replace('arguments[2]', to)
        args = file_input_element, event_name,
    else:
        args = file_input_element, event_name, to
    self.driver.execute_script(script, *args)
def get_uploaded_file_info(self):
    return self.driver.find_element_by_xpath(
        "//div[starts-with(@aria-label, "
        "'Uploading attachment: %s')]" % self.test_file_name
    ).text
def send_message(self):
    self.driver.find_element_by_xpath("//div[text()='Send']").\
        click()
    wait = WebDriverWait(self.driver, 120)
    xpath = "//div[starts-with(text(), " \
        "'Your message has been sent.')]'"
    wait.until(
        expected_conditions.presence_of_element_located((
            By.XPATH, xpath
        ))
    )
def log_out_from_gmail(self):
    self.driver.get('https://mail.google.com/mail/?logout')
    try:
        self.driver.switch_to_alert().accept()
    
```

```

        except (NoAlertPresentException, WebDriverException):
            pass
        finally:
            self.driver.switch_to_default_content()
def delete_all_client_cookies(self):
    self.driver.delete_all_cookies()
def open_sent_message(self):
    self.driver.find_element_by_xpath(
        "//span/b[text()='<div>' % self.email_subject
    ).click()
    WebDriverWait(self.driver, 10).until(
        expected_conditions.presence_of_element_located((
            By.XPATH, "//*[@text()='<div>' % self.test_file_name
        ))
    )
def virus_scan_complete(driver):
    return not driver.find_elements_by_xpath(
        "//*[@starts-with(., 'Scanning for viruses')]"
    )
    self.driver.implicitly_wait(0)
    WebDriverWait(self.driver, 10).until(virus_scan_complete)
    self.driver.implicitly_wait(10)
def assert_attachment_size_is(
    self, file_size_in_kb, file_size_in_mb
):
    test_file = self.driver.find_element_by_xpath(
        "//*[td/b[starts-with(text(), '%s')]]" %
        self.test_file_name + ' | ' + \
        "//*[span[text()='<div>' % self.test_file_name
    )
    ActionChains(self.driver).move_to_element(test_file).\
        perform()
    self.driver.implicitly_wait(0)
    file_size_in_kb_exists = len(
        self.driver.find_elements_by_xpath(
            "//*[td[contains(., '%s')]]" % file_size_in_kb
        )
    ) > 0
    file_size_in_mb_exists = len(
        self.driver.find_elements_by_xpath(
            "//*[td[contains(., '%s')]]" % file_size_in_mb
        )
    ) > 0
    self.driver.implicitly_wait(10)
    self.assertTrue(
        file_size_in_kb_exists or file_size_in_mb_exists
    )
def forward_message_to(self, email_address):
    self.driver.find_element_by_xpath(
        "//*[span[text()='Forward' and @role='link']]"
    ).click()
    self.driver.find_element_by_name('to').\
        send_keys(email_address)
    self.driver.find_element_by_xpath("//*[div[text()='Send']]").\
        click()
    wait = WebDriverWait(self.driver, 10)
    xpath = "//*[div[starts-with(text(), " \
        "'Your message has been sent.')]]"
    wait.until(

```



```

        expected_conditions.presence_of_element_located((
            By.XPATH, xpath
        ))
    )
    def tearDown(self):
        self.driver.quit()
        super(GmailExampleSelenium, self).tearDown()

```

gmail_base.py

```

from ConfigParser import RawConfigParser
from os import path
from unittest import TestCase
import re

MIN_FILE_SIZE_MB = 2

class GmailExampleTest(TestCase):
    def check_attachment_sizes(self):
        # 2.:
        self.log_in_to_gmail(self.email_address_1, self.password_1)
        # 3.:
        self.compose_new_message(
            self.email_address_2, self.email_subject
        )
        # 4.:
        self.attach_file_to_message(self.test_file_path)
        # 5.:
        uploaded_file_info = self.get_uploaded_file_info()
        file_sizes = self.get_attachment_size(uploaded_file_info)
        # 6.:
        self.send_message()
        # 7.:
        self.log_out_from_gmail()
        # 8.:
        self.delete_all_client_cookies()
        # 9.:
        self.log_in_to_gmail(self.email_address_2, self.password_2)
        # 10.:
        self.open_sent_message()
        # 11.:
        self.assert_attachment_size_is(*file_sizes)
        # 12.:
        self.forward_message_to(self.email_address_1)
        # 13.:
        self.log_out_from_gmail()
        # 14.:
        self.delete_all_client_cookies()
        # 15.:
        self.log_in_to_gmail(self.email_address_1, self.password_1)
        # 16.:
        self.open_sent_message()
        # 17.:
        self.assert_attachment_size_is(*file_sizes)
    def log_in_to_gmail(self, email_address, password):
        raise NotImplementedError()
    def compose_new_message(self, to_email_address, subject):
        raise NotImplementedError()
    def attach_file_to_message(self, file_path):

```

```

        raise NotImplementedError()
def get_uploaded_file_info(self):
    raise NotImplementedError()
def send_message(self):
    raise NotImplementedError()
def log_out_from_gmail(self):
    raise NotImplementedError()
def delete_all_client_cookies(self):
    raise NotImplementedError()
def open_sent_message(self):
    raise NotImplementedError()
def assert_attachment_size_is(
    self, file_size_in_kb, file_size_in_mb
):
    raise NotImplementedError()
def forward_message_to(self, email_address):
    raise NotImplementedError()
def open_forwarded_message(self):
    raise NotImplementedError()
def get_attachment_size(
    self, uploaded_file_info, test_file_name=None
):
    if test_file_name is None:
        test_file_name = self.test_file_name
    file_size_kb = self._get_attachment_size_kb(
        uploaded_file_info, test_file_name
    )
    file_size_in_kb = '%dK' % file_size_kb
    file_size_mb = file_size_kb / 1024.0
    if int(file_size_mb) == file_size_mb:
        file_size_in_mb = '%d MB' % int(file_size_mb)
    else:
        file_size_in_mb = '%.1f MB' % file_size_mb
    return file_size_in_kb, file_size_in_mb
def _get_attachment_size_kb(
    self, uploaded_file_info, test_file_name=None
):
    if test_file_name is None:
        test_file_name = self.test_file_name
    file_info_re = test_file_name + r' \(([^\)]+)\)'
    file_size = re.match(
        file_info_re, uploaded_file_info
    ).group(1)
    file_size_re = r'([0-9]+)[, ]?([0-9]*)(?:\.\d\d)? ?KB?'
    file_size_match = re.match(file_size_re, file_size)
    return int(
        file_size_match.group(1) + file_size_match.group(2)
    )
def setUp(self):
    self.email_subject = 'TP1-Gmail attachment size reporting'
    self.parameters_file_name = 'parameters.ini'
    self._read_parameters_file()
    self.test_file_path = path.expanduser(
        r"~\Desktop\%s" % self.test_file_name
    )
    # Check pre-conditions:
    assert path.exists(self.test_file_path), \
        "Test file %s does not exist. Please create." % \
        self.test_file_path

```

```
MB = 1024 * 1024
assert path.getsize(self.test_file_path) > \
    MIN_FILE_SIZE_MB * MB, \
    "Test file %s does not have the required minimum " \
    "size of %dMB" % (
        self.test_file_path, MIN_FILE_SIZE_MB
    )
def _read_parameters_file(self):
    params_file_parser = RawConfigParser()
    params_file_parser.read(self.parameters_file_name)
    self.email_address_1 = params_file_parser.get(
        'parameters', 'email_address_1'
    )
    self.password_1 = \
        params_file_parser.get('parameters', 'password_1')
    self.email_address_2 = params_file_parser.get(
        'parameters', 'email_address_2'
    )
    self.password_2 = \
        params_file_parser.get('parameters', 'password_2')
    self.test_file_name = params_file_parser.get(
        'parameters', 'test_file_name'
    )
```