Abracadabra · Follow
Apr 12, 2021 · 6 min read ★ · Listen

How do I write engineering design docs in Google: an example

Documentation is one most important skills I learned during my tenure at Google, where document is used as a discussion format, source of truth, and knowledge organization. None other companies I worked at had equally deep understanding about using documentation for collaboration. This post is an example of how I would write a design doc. The project was actually built and in use. The source code will be accessible after gym booking is not required post COVID-19. To make it more fun, this is the Google Doc link that everyone can comment on. It's also better formatted than what Medium supports.

Problem Statement

During COVID-19, gyms are required to control the number of total members on site. My gym requires members to boo before using. Booking on a date is open two days in advance, starting at the corresponding midnight. For example, the 04–01 will be open on 2021–03–30 00:00 AM local time.

There are a very limited number of spots for the swimming pool. After failing to book at 6 AM several times, I was told by a staff member that I will have to book at midnight due to the high volume of needs. Staying up till midnight breaks my flow, so it's not acceptable. I feel bad to hire people doing this for me because deep down I think sleeping early is a core habit for a healthy and efficient lifestyle(see my pillar habits), it's not moral to deprive anyone of a good habit by money. Being told that there is no work around, I decided to write a program to do the booking for me.

I personally think using a bot to do the work is not fair to others. So I'm in no way proud of this decision. I think the gym should raise the price for some spots. But clearly this is way beyond the scope of this design doc and highly subjective.

Requirements

- Automatically book gym **two days in advance, at midnight**
- The program needs no human interaction after started, should be fault tolerant and **reasonably retry**
- The program runs on a Mac
- User can specify username, password, the sport to book, date and time to book, etc

Non-goals

- **Book only 1 or 2 days in advance, or for the current day**
- Tolerant to OS or network issues
- Functioning if booking server is down
- Functioning if website structure(HTML) changes

High level design

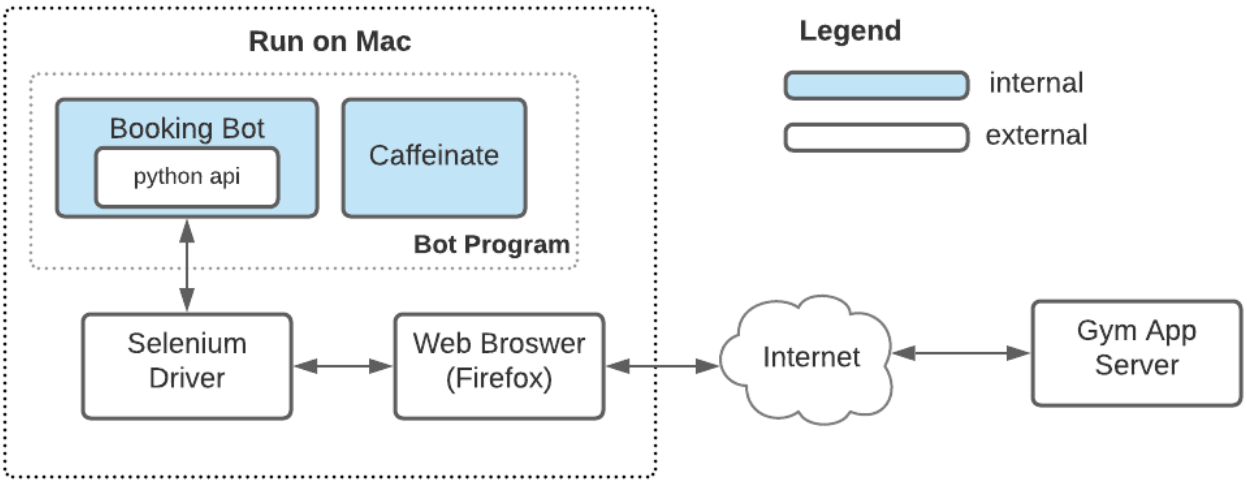
Browser automation V.S. request simulation

Browser automation is using a program to control a real browser and automate the operation on the GUI. Request simulation is having the program talking with the server via HTTP, as if it *is* a web browser(rather than controlling one).

- **[pros]** The website requires javascript to load the controls, this is hard to implement programmatically, may need to control some rendering engine
- **[cons]** Browser automation depends on the HTML structures while request simulation depends on the HTTP APIs. APIs are far less likely to change.

The pros clearly dominate the cons.

System overview



Selenium is a software library that provides the solution of browser automation. Our program will be written in Python and control Selenium via the Python API. Selenium in turn controls Firefox via its Gecko driver.

Caffeinate is a program that stops the OS from going to sleep. If the system sleeps, the program won't run at midnight.

Detailed design

User input

Username, password, date, etc are input from command line arguments.

Retrying

The program will catch all exceptions(page not loaded, etc) and retry for 100 times, until booking is successful. Successful booking is recognized by a confirmation DOM element.

Browser choice

We need to use one of the mainstream browsers. I've considered and tested Chrome, Firefox and Safari. Both Safari and Chrome need extra steps to work with their corresponding Selenium driver, so I chose Firefox. It needs some authentication from OS settings too, but won't need that after being allowed for the first few times.

Logging

The program flow is automating browser actions as if it's initiated by a user. Essentially, it will do the following in loop:

1. Locate some element

Therefore each logging will have two contents

- What have been executed
- What am I waiting for

Such logging will make debugging easy.

Keeping computer awake

If the OS goes to sleep between the time the program is started and midnight, the program can't run at midnight. Caffeinate is used to prevent this from happening. Since it's a command line tool, we will start it as a child process in Python:

```
subprocess.Popen(['caffeinate', '-d', '-w', '%d' % os.getpid()])
```

Locating the controls

Selenium provided an array of methods(reference) to access specific DOM elements. Among them, xpath has the most expression power. Thus we will use find_element_by_xpath to locate DOM elements like buttons, input boxes, etc.

Whenever possible, we will prefer to depend on the DOM's inner text to locate them. The advantage over DOM structures and attributes(class name, etc) is not that inner texts are less likely to change, but that if they do change, it's easier to debug. Of course, we will have to make some assumptions about DOM structures like we need to click the second button with class='logon' under a div of class='control'.

Waiting for page loading

The program needs to wait for page loading(usually 2~5 seconds, yes it's a slow site) after each HTTP request is sent. This is done by the WebDriverWait API. For example, the following code will wait for 120 seconds until a <**button** ng-reflect-router-link='/**Appointments**'> is loaded and becomes clickable.

```
book_btn = WebDriverWait(driver, 120).until(EC.element_to_be_clickable(
(By.XPATH, "//button[@ng-reflect-router-link='/Appointments']")))
```

If the button fails to load within 120 seconds, an exception will be raised.

More implementing details

Selecting the correct date. Assuming we want to book on 4/14. We can't select a cell with text '14' on the booking calendar, because there will be a cell for 3/14 with the similar attributes. Current month's cell must have class cal-in-month.

Adjusting month. The booking calendar shows the month on the current day, not the month we intend to book. This will be an issue when two days later is next month. So we will have to add another step to select the correct month under this corner case.

Usage flow

Assume I want to book the swimming pool on 4/14. I will run the following command at some arbitrary time on 4/11.

```
python book.py — username xxxxxx — password xxxxxx — day 14 \

— time '5:00 PM' — sport small_pool
```

The code will sleep and check time every 1 second. This check won't have any noticeable footprint on CPU usage. Until midnight, Caffeinate will prevent the OS from going to sleep.

On the midnight of 4/12, it will launch a Firefox and automate the booking. After that both the caffeine process and the main process will exit. The OS will go to sleep normally.

usually 2~3 people ahead of it on the time slot. There are altogether only 6 spots for each time slot. No doubt booking at 6AM will never be possible.