DOI: 10.1007/s10664-021-09975-3

Automated Driver Management for Selenium WebDriver

Boni García · Mario Munoz-Organero · Carlos Alario-Hoyos · Carlos Delgado Kloos

Received: date / Accepted: date

Abstract Selenium WebDriver is a framework used to control web browsers automatically. It provides a cross-browser Application Programming Interface (API) for different languages (e.g., Java, Python, or JavaScript) that allows automatic navigation, user impersonation, and verification of web applications. Internally, Selenium WebDriver makes use of the native automation support of each browser. Hence, a platform-dependent binary file (the so-called driver) must be placed between the Selenium WebDriver script and the browser to support this native communication. The management (i.e., download, setup, and maintenance) of these drivers is cumbersome for practitioners. This paper provides a complete methodology to automate this management process. Particularly, we present WebDriverManager, the reference tool implementing this methodology. WebDriverManager provides different execution methods: as a Java dependency, as a Command-Line Interface (CLI) tool, as a server, as a Docker container, and as a Java agent. To provide empirical validation of the proposed approach, we surveyed the WebDriverManager users. The aim of this study is twofold. First, we assessed the extent to which WebDriverManager is adopted and used. Second, we evaluated the WebDriverManager API following Clarke's usability dimensions. A total of 148 participants worldwide completed this survey in 2020. The results show a remarkable assessment of the automation capabilities and API usability of WebDriverManager by Java users, but a scarce adoption for other languages.

Keywords Test Automation · Testing Tools · Selenium WebDriver

1 Introduction

Automated testing plays a significant role in the development of reliable software products. In this context, developers and testers use testing tools to carry out automated tests in an effective and reproducible manner [24]. A recent study

✉ B. García · M. Munoz-Organero · C. Alario-Hoyos · C. Delgado Kloos
Universidad Carlos III de Madrid
Avenida de la Universidad 30, 28911 Leganés, Spain
E-mail: bogarcia,munozm,calario,cdk@it.uc3m.es

identifies Selenium as the most valuable testing framework nowadays, followed by JUnit and Cucumber [14]. Selenium[1] is an umbrella open-source project that enables the automation of web browsers. The core of Selenium is called Selenium WebDriver, a framework that allows controlling web browsers in an automated fashion using a language-specific binding (such as Java, JavaScript, Python, among others) [3]. Selenium WebDriver is typically used to implement end-to-end tests. These tests verify web applications impersonating users who interact with the web interface of the System Under Test (SUT) [57]. Another common usage of Selenium WebDriver is web scraping, a technique to extract large amounts of data from websites in an automated manner [15].

Selenium WebDriver employs the native support of automation that each web browser provides. This fact is convenient to provide a full automation experience (i.e., automated navigation and user impersonation) and avoid security issues [57]. Nevertheless, it adds extra complexity to the automation process, since an intermediate proxy is required between Selenium WebDriver and each browser. This proxy is the so-called driver, a platform-dependent binary file that translates the commands sent by Selenium WebDriver to the native browser automation support. Browser vendors usually provide their own drivers. For example, the driver required to control Chrome and Chromium is called chromedriver[2], and the driver needed to control Firefox is called geckodriver[3], to name a few.

A recent survey about the Selenium ecosystem shows that the management (i.e., download, setup, and maintenance) of the required driver of Selenium WebDriver is done manually by practitioners in most cases [29]. This manual process brings extra complexity to the development and maintenance of Selenium WebDriver scripts. Furthermore, this manual process causes unreliable tests due to version incompatibility between browsers and drivers. To solve these problems, we propose a complete methodology to automate the management of the drivers required by Selenium WebDriver. We introduce an open-source tool called WebDriverManager as the reference implementation for this methodology. WebDriverManager is used by tens of thousands of projects worldwide nowadays. Its primary usage is as a Java library. For the sake of interoperability with other language bindings, WebDriverManager can also be used as a Command-Line Interface (CLI) tool, as a server, as Docker container, and as a Java agent. We launched a survey in the WebDriverManager community to validate it. This study is aimed to evaluate the adoption, usage, and Application Programming Interface (API) usability.

The remainder of this paper is structured as follows. Section 2 reviews the technological background of this work. Section 3 summarizes the related contributions found in the literature in two parts: 1) Test automation and Selenium; 2) Evaluation of API usability. Section 4 provides fined-grained details of the motivation of the presented approach. Next, section 5 presents the proposed methodology to manage the drivers required by Selenium WebDriver in an automated fashion. Then, section 6 introduces the tool implementing this methodology, i.e., WebDriverManager. Section 7 presents the survey (design, results, and validity) carried out to assess the proposal. Then, we discuss the findings in section 8. Finally, section 9 provides the conclusion and future work of this line of research.

[1] `https://www.selenium.dev/`

[2] `https://chromedriver.chromium.org/`

[3] `https://github.com/mozilla/geckodriver/`

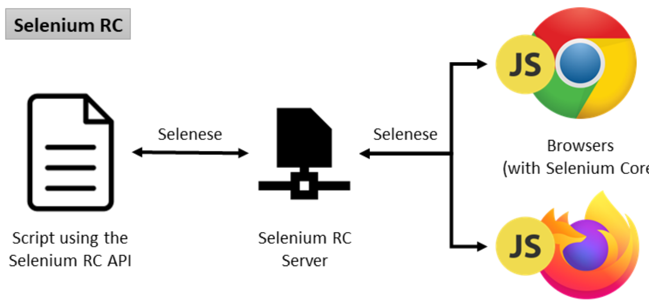


Fig. 1 Selenium RC architecture

2 Background

Jason Huggins and Paul Hammant created the first version of Selenium in 2004 while working in ThoughtWorks. That initial version of Selenium (known now as Selenium Core) is a JavaScript library that impersonates user actions in web applications. Selenium Core interprets the so-called Selenese commands to achieve automation, composed of three parts: 1) Command (action to be executed in the web browser); 2) Target (locator which identifies a web element); 3) Value (optional data) [10].

Huggins and Hammant added a scripting layer to Selenium Core in a new project named Selenium Remote Control (RC). As shown in Figure 1, Selenium RC follows a client-server architecture. Clients use a binding language (such as Java or JavaScript) to send Selenese commands over HTTP to an intermediate proxy called the Selenium RC server. This proxy launches web browsers on demand, injects the Selenium Core library on the SUT, and proxies requests from clients to Selenium Core [12]. The Selenium RC Server masks the SUT to the same local URL of the injected Selenium Core library to avoid same-origin policy concerns (e.g., when accessing a public website). This approach allows practitioners to create end-to-end tests for web applications. Although RC was a game-changer from browser automation at that time, it had significant limitations. First, since JavaScript is the underlying technology to support automation, some actions cannot be done, since JavaScript does not allow them, for example, upload and download files, or handle pop-ups and dialogs, to name a few. Besides, Selenium RC introduces a relevant overhead which impacts the performance of end-to-end tests.

To overcome these problems, Simon Stewart, also working for ThoughtWorks, created a new project called Selenium WebDriver in 2008. Selenium WebDriver and RC are equivalent from a functional perspective, i.e., both projects allow to impersonate web users using a binding language. Nevertheless, Selenium WebDriver provides more comprehensive automation support and better performance than RC. For that reason, Selenium RC is discouraged in favor of WebDriver nowadays [32]. As shown in Figure 2, the automation is possible in Selenium WebDriver thanks to the native support of each browser. Thus, it is necessary to include an intermediate element between the Selenium script and the browser. This element is the so-called driver. This driver is a platform-dependent binary file that receives commands from the Selenium scripts and translates them into

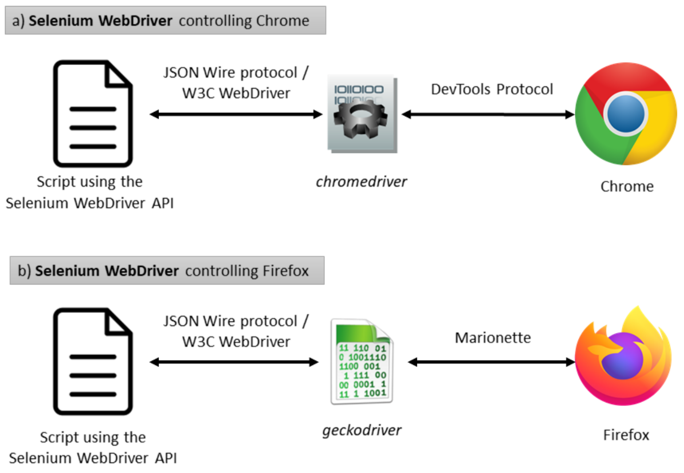

a) Selenium WebDriver controlling Chrome
JSON Wire protocol / W3C WebDriver
DevTools Protocol
Script using the Selenium WebDriver API
chromedriver
Chrome
b) Selenium WebDriver controlling Firefox
JSON Wire protocol / W3C WebDriver
Marionette
Script using the Selenium WebDriver API
geckodriver
Firefox

Fig. 2 Selenium WebDriver architecture

some browser-specific language. Initially, the communication between the Selenium script and the drivers was done using JSON messages over HTTP in the so-called JSON Wire Protocol [34]. Nowadays, this communication has been standardized in the W3C WebDriver recommendation [53]. The W3C WebDriver recommendation is also based on JSON messages sent over HTTP. As of Selenium WebDriver version 4, the JSON Wire Protocol is discouraged in favor of W3C WebDriver [29].

Figure 2 shows a couple of examples of Selenium WebDriver. The left side of this figure shows some scripts that use the cross-browser Selenium WebDriver API. The center of the figure shows the drivers, which receive JSON Write Protocol or W3C WebDriver commands and translate these commands to the browser-specific support. Finally, the right part of the figure shows the web browsers. In the case of Google Chrome, the driver is named chromedriver, and the native browser support is the DevTools Protocol[4]. In Firefox, the driver is called geckodriver, and the native automation support is done using the Marionette protocol[5].

3 Related work

3.1 Test Automation and Selenium

Dustin et al. define automated software testing as "the management and performance of activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool" [24]. Automated testing tools run pieces of software called test cases (or simply tests) that exercise a SUT while comparing their actual outcomes with the expected results, giving a verdict (pass or fail) about it [18]. Rafi et al. carried out a systematic literature review about the benefits and limitations of automated software testing [48]. According

[4] `https://chromedevtools.github.io/devtools-protocol/`
[5] `https://firefox-source-docs.mozilla.org/testing/marionette/`

to this review, the most relevant advantages of test automation are reusability, repeatability, and effort saved in test executions. However, implementing and maintaining automated tests are time-consuming tasks. This view is aligned with the results presented by Ramler et al. [49], since many automated testers invest too much effort (and, therefore, cost) to repair decayed tests.

A key aspect to carry out automated tests efficiently in a software project is using proper testing tools [23]. In this field, Selenium is a very relevant actor in the browser automation space. Selenium is often recognized as the de-facto framework for end-to-end testing of web applications [23]. Moreover, it is the basis for a relevant number of approaches that wrap or extend the built-in capabilities of Selenium, such as the SmartDriver project [11], which is a Selenium WebDriver extension based on the separation of test automation aspects: 1) Technical elements that are related to the user interface and test logic; 2) Business aspects that are associated with the application under test. Another framework that extends Selenium WebDriver is named FACTS (Framework for Automation of Cloud-application Testing using Selenium) [38]. It is a data-driven framework in which unit tests using Selenium WebDriver are derived from existing datasets. Another example in this category is Selenium-Jupiter, an extension of JUnit 5 [28] that provides integration with Selenium and Docker [31][30].

A recent study about the Selenium ecosystem reports that maintainability and flakiness are the main problems related to Selenium test suites [29]. A test is said to flaky when it is unreliable, i.e., it fails or passes under the same conditions [43]. Implementing proper wait [47] and location strategies [40] for web elements is recommended to avoid flaky tests with Selenium WebDriver. Regarding maintainability, and as proved in [2], the maintenance of automated tests provides a positive return on investment compared to manual testing. Therefore, maintainability costs should not be a stopper in the adoption of test automation through Selenium. A well-known design pattern aimed to narrow the maintenance costs of Selenium tests is the Page Object Model (POM). With this pattern, web pages are modeled using an object-oriented class to reduce code duplication. According to Leotta et al., the maintenance time is divided by three when using POM [41]. In this line, Stocco et al. propose Apogen, a tool aimed to automatically generate page objects for Selenium WebDriver [54][55].

As introduced before, a fundamental aspect of Selenium WebDriver is the usage of proper drivers to automate browsers using their native support. To the best of our knowledge, there is no relevant literature about the management of these drivers, which is, in many cases, done manually [29]. This can be interpreted as an indicator of the necessity for this work. As explained in Section 4, the use of automated management techniques for these drivers might ease the development and maintenance of automated tests based on Selenium.

3.2 API design and evaluation

APIs play a key role in modern software development. Software developers worldwide create applications by composing capabilities exposed by different APIs [36]. Programming is a hard mental work in which API misuse has been identified as a prevalent cause of software defects [37]. For this reason, proper API design is recognized as being paramount of importance nowadays [8].

Table 1 Clarke's dimensions definition and relationship with the CDs dimensions

Clarke's dimension	**Definition**	**Related CDs dimensions**
Understandability	Process of API learning and understanding	Closeness of mapping Hidden dependencies Hard mental
Abstraction	Low-level details developers need to manage	Abstraction
Expressiveness	Process of translating application requirements into code	Role expressiveness Visibility Consistency Premature commitment Provisionality Error proneness
Reusability	Maintenance and evolution of applications	Viscosity Diffuseness
Learnability	Incremental and progressive API learning process	Progressive evaluation

The API design is an interdisciplinary domain that ideally should combine cognitive psychology and software engineering [36]. However, the API design is typically carried out by the development team, usually focused on implementation details rather than usability aspects. In this field, several research efforts have been made about API design and evaluation. Early attempts to investigating APIs followed ad-hoc approaches of given technologies, such as C# [58], Java [8], or C++ [50]. Other authors concentrate on API usability independently of the underlying technology. For instance, Ellis et al. carried out a usability evaluation of the factory pattern [25]. During the last decades, further authors tried to systematize the problem of API design and evaluation from a holistic perspective [1][5][21][26]. In this context, the Cognitive Dimensions of Notations (CDs) framework has gained popularity [33]. The CDs framework is an approach for analyzing the usability of notational systems (such as word processors or computer-aided design tools) and information artifacts (typically software systems) [7]. The CDs framework provides a set of discussion notations (i.e., a vocabulary) to be used by designers when investigating the cognitive implications of design decisions.

When coming to API evaluation, the CDs framework can be a useful approach since it allows to compare users' expectations to designers' views. Nevertheless, the CDs framework has been criticized due to its theoretical and practical limitations [44]. For this reason, further authors propose the adaption of the CDs framework to the so-called Clarke's dimension [16][17], which concentrates the main aspects of API usability in 5 high-level aspects [46]: understandability, abstraction, expressiveness, reusability, and learnability. These aspects and their relationship with the original CDs framework are summarized in Table 1.

Clarke's dimensions are more straightforward to understand than the CDs notations as they are fewer and more intuitive for regular developers. In addition, the evaluation of an API following Clarke's dimensions can be done using a questionnaire requesting users through a Likert scale [9]. For all these reasons, Clarke's dimensions have been used to assess the usability API in previous research [46][42].

Hence, and as explained in Section 7, we use Clarke's dimensions to evaluate the usability of the proposed WebDriverManager API.

4 Motivation

As introduced before, the use of the proper driver (e.g., chromedriver for Chrome or geckodriver for Firefox) is a mandatory constraint when using Selenium WebDriver. This fact is a recurrent problem for Selenium WebDriver testers. For example, when using Java as language binding in Selenium WebDriver, the driver's absolute path must be exported as a given Java property, as shown in Listing 1. Although this example is specific to Java, other equivalent approaches are required for other Selenium WebDriver binding languages[6].

Listing 1 Standard setup for drivers path in Selenium WebDriver using Java

```
System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
System.setProperty("webdriver.gecko.driver", "/path/to/geckodriver");
System.setProperty("webdriver.edge.driver", "/path/to/msedgedriver");
System.setProperty("webdriver.opera.driver", "/path/to/operadriver");
System.setProperty("phantomjs.binary.path", "/path/to/phantomjs");
System.setProperty("webdriver.ie.driver", "C:/path/to/IEDriverServer.exe");
```

As can be seen, the required driver needs to be downloaded and its absolute path is hardcoded as a Java property. This fact is problematic at different levels. First of all, it is a manual process, and therefore, it adds some extra effort to the development process of Selenium WebDriver tests. Moreover, the resulting code is no longer portable. In other words, it can only be executed in the machine in which the driver is stored. This is especially cumbersome in the usual case in which tests are executed by Continuous Integration (CI) servers, such as Jenkins or Travis, to name a few [4].

Another major problem of manual driver management is related to the maintainability and flakiness of the Selenium WebDriver tests. Modern web browsers, such as Chrome, Firefox, Edge, or Opera, are sometimes called evergreen browsers. This term refers to the ability of these browsers to upgrade to the latest stable version automatically. For example, and as documented in the download page of chromedriver[7], chromedriver version 83.0.4103.39 is required for controlling Chrome 83 with Selenium WebDriver. Thus, a tester needs to download and export the path of that specific chromedriver, as shown in Listing 1. Eventually, the evergreen Chrome will automatically upgrade to version 84. At that moment, a Selenium script using that driver will be broken since the required driver for Chrome 84 is chromedriver version 84.0.4147.30 and not 83.0.4103.39, which is incompatible with Chrome 84. In practice, Selenium developers experience this problem when a decayed test reports the following error message: "*this version of chromedriver only supports chrome version N*". The value of N is the latest version of Chrome supported by a particular version of chromedriver (83 in the example before).

[6] `https://www.selenium.dev/documentation/en/webdriver/driver_requirements/`
[7] `https://chromedriver.chromium.org/downloads`

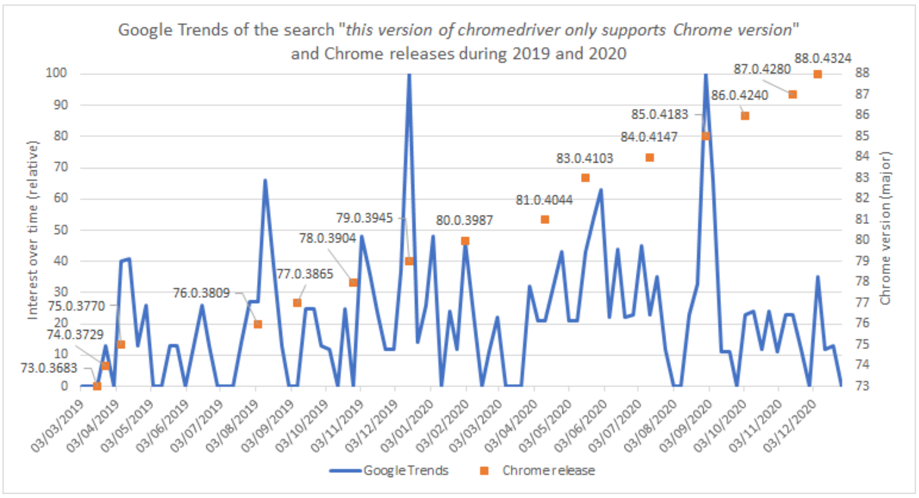


Fig. 3 Worldwide relative interest over time of the search term "*this version of chromedriver only supports chrome version*" in Google Trends together with the release dates of Chrome during 2019 and 2020.

Figure 3 shows the worldwide search interest of the abovementioned error message on Google[8] during 2019 and 2020, together with the release date of the different versions of Chrome in this period. As illustrated by this chart, the interest over time concerning this error message is directly related to the release of new Chrome versions. The issue is recurrent in time, and the automatic update of evergreen browsers used in conjunction with manually managed drivers (chromedriver in this case) is the underlying cause. Another proof of this issue can be found on the popular questions and answers site StackOverflow. At the time of this writing, the message "*this version of chromedriver only supports chrome version*" appears in 330 StackOverflow posts[9] involving Selenium WebDriver and different language bindings.

In conclusion, a Selenium WebDriver test in which the driver management is done manually requires extra maintenance effort to keep the version compatibility of driver and browser. Therefore, we consider that the management of Selenium drivers is a very relevant aspect that deserves a robust automated solution. The next section provides the details of the proposed methodology to solve these problems.

5 Methodology

The final aim is to download, setup, and maintain the proper drivers for a computer in which Selenium WebDriver scripts are executed to drive local browsers. Driver releases are available in public repositories typically accessible through HTTP. The

[8] `https://trends.google.com/trends/explore?date=2019-01-01%202020-12-31&q=this%20version%20of%20chromedriver%20only%20supports%20Chrome%20version`

[9] `https://stackoverflow.com/search?q=this+version+of+chromedriver+only+supports+Chrome+version`

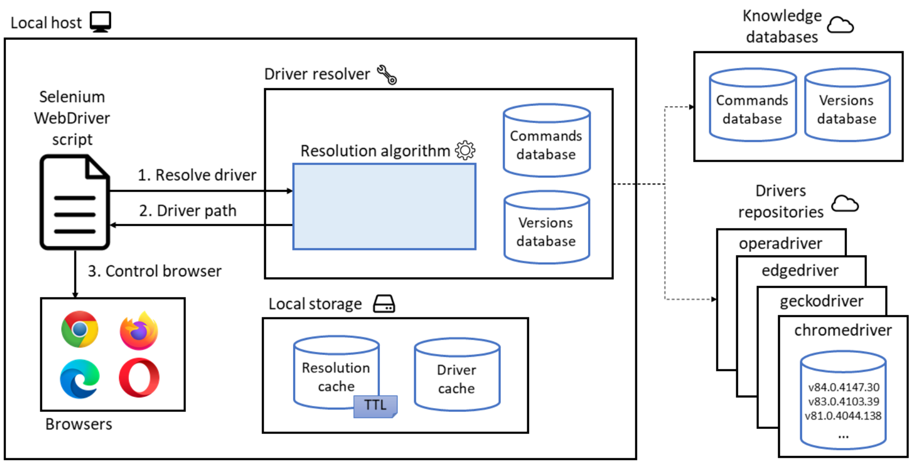


Fig. 4 Overview of the proposed methodology. A driver resolver is a helper tool that implements the resolution algorithm. This algorithm identifies the proper driver and downloads it from the corresponding repository. The drivers and the resolution data are stored persistently in the local storage.

names of these drivers for the main evergreen browsers nowadays are chromedriver (Chrome and Chromium), geckodriver (Firefox), msedgedriver (Edge), and operadriver (Opera). This section provides a detailed description of the proposed approach to manage these drivers in an automated fashion.

The high-level overview of the proposed methodology is depicted in Figure 4. The first component depicted in this diagram is the local computer in which a Selenium WebDriver script needs to be executed. The first constraint to run this script is the required browser (e.g., Chrome, Firefox) is installed in the machine. Supposing this constraint is fulfilled, the next step is to obtain the proper driver for the local browser. We want to carry out this process in an automated and auto-updated manner to avoid the potential problems described previously. This process is carried out by a component labeled as "driver resolver" in Figure 4.

A driver resolver is a helper tool for Selenium WebDriver. The heart of this tool is what we call the resolution algorithm. The term "resolution" refers to the process to identify the required driver to control a given browser with Selenium WebDriver. Thus, the resolution algorithm (represented as a blue box in Figure 4) is in charge of identifying the proper driver, downloading it from its repository, and making it available for the Selenium WebDriver script. The rest of this section is divided into three parts to explain all the details of this algorithm:

1. Versions database. It is mandatory to know beforehand the relationship between the driver versions aimed to control the different flavors and versions of web browsers. The driver resolver uses a trusted knowledge database that supports the resolution algorithm.
2. Local storage. Another essential aspect of the proposed methodology is the persistence layer. For the sake of performance, it is mandatory to store different assets and data in the local storage of the computer running the resolution algorithm. This information is saved persistently in two separate places. We refer to these spaces as the driver and the resolution cache.

3. Resolution algorithm. Finally, we describe the fine-grained details of the internal algorithm workflow and the relationship with the rest of the components.

5.1 Versions database

The “versions database” is a key ingredient in the proposed methodology. This component is a knowledge base that associates the different driver versions (e.g., chromedriver) with their corresponding browser versions (e.g., Chrome). This database is mandatory since it is impossible to determine the correct driver version for all browsers in all cases. For instance, if a tester requires controlling Firefox version 68 with Selenium WebDriver, he/she should use the correct version of geckodriver. To find out this relationship, he/she needs to go to the geckodriver repository and inspect the documentation. In some cases (unfortunately, not always), the maintainers document this relationship in the release notes. But in other cases (especially in early browser versions), the relationship browser-driver is unknown. In this case, the driver version can only be identified by trial and error (for the example of Firefox 68, the required geckodriver version is 0.25.0). This problem also happens with Opera (for instance, operadriver 83.0.4103.97 is needed to control Opera 69).

The maintainers of chromedriver were pioneers to provide a solution to this issue. Starting from Chrome 70 (released in September 2018), the chromedriver team began to make releases following the same versioning schema as Chrome. Moreover, they began to maintain different files in their repository to announce the chromedriver versions. This way, the latest stable release of chromedriver can be checked by simply inspecting the content of a file called `LATEST_RELEASE`[10] in the chromedriver repository. Moreover, there are additional files in this repository reporting the full version of the drivers for Chrome browser 70 and above. For instance, the file `LATEST_RELEASE_80`[11] contains the version of chromedriver required to control Chrome 80. This mechanism proved to be effective in automating the management of chromedriver. For this reason, the msedgedriver team adopted a similar procedure starting from Edge 75 (released in June 2019).

Thanks to this method, automated scripts can be created to determine the proper driver version for Chrome 70 and above and Edge 75 and above, simply knowing the major version of the browser (e.g., 76, 77, and so for). Nevertheless, this process cannot be automated for other browsers, such as Firefox or Opera, neither for previous versions for Chrome nor Edge. Thus, the first element of our methodology is a knowledge database that matches the versions or driver and browser to carry out the resolution process in an automated manner. As illustrated in Figure 4, there are two copies of this database. The first one is the master copy of the database, and it is updated continuously to maintain the relationship between browsers and drivers. This version database is publicly available, and it is labeled as “versions repository” in Figure 4. There is also a local copy of the versions database shipped in each release of the driver resolver. As explained in Section 5.3, this local copy is used as the last resource mechanism to discover the driver version using the browser version in the resolution algorithm.

[10] `https://chromedriver.storage.googleapis.com/LATEST_RELEASE`
[11] `https://chromedriver.storage.googleapis.com/LATEST_RELEASE_80`

5.2 Local storage

The main objective of the resolution algorithm is to download the proper driver to control a given web browser with Selenium WebDriver. Drivers are binary programs with a size of around 10 MB. Although they are not too heavy, it is pointless to download the same file repeatedly when executing different Selenium WebDriver scripts. Therefore, each time a driver is downloaded, it is stored persistently in the local filesystem in what we call the driver cache. By default, the next time the same driver version is required, it is used directly from that cache. The use of this cache reduces the startup time of Selenium WebDriver tests drastically.

The second persistent data store we propose is called the resolution cache. As explained in Section 5.3, to carry out the resolution algorithm, it is necessary to connect to different online repositories (to check the versions database and the driver versions) and also execute certain commands in the shell. This process takes some time, which depends mainly on external factors (network latency, congestion, or server response time). Nevertheless, the algorithm output (i.e., the path of the driver to be used) is in principle the same when the algorithm is executed in a short timelapse (e.g., when executed repeatedly in a test suite). This happens because the browser and their corresponding drivers have a relatively low rate of change (compared to the duration of a test suite execution). For example, the Chrome team usually releases a major version every two months, approximately. Hence, and to improve the performance of the resolution algorithm, the results obtained when the resolution algorithm is executed are stored persistently in what we call the resolution cache. In particular, two pieces of information are stored:

1. The detected major version of the browser, e.g., Chrome 83.
2. The resolved driver version for a given browser version, e.g., chromedriver 83.0.4103.39 (which is the resolved version for Chrome 83).

Inspired by the cache mechanism implemented in the DNS (Domain Name System) protocol, each record stored in the resolution cache is attached to a Time-To-Live (TTL) [39]. In particular, a time-based TTL approach is followed. Following this approach, each TTL is understood as an expiration time in which each record is valid. This way, each entry stored in the resolution cache is considered reusable only in the period established by a timestamp. This timestamp is calculated using the date and time the algorithm was executed plus the TTL [52]. When further requests to the resolution algorithm are made, the information stored in the resolution cache is reused only in that validity period. After that, the record is considered stale, and the resolution process should be renewed. For better performance tuning, we propose two different values of TTLs:

1. *TTL_driver*. This value is used to set the expiration of the resolved driver version (e.g., chromedriver 83.0.4103.39). Since the driver version is calculated based on network requests, this part of the resolution process is costly in terms of time. Therefore, and for the sake of performance, we recommend a relatively high value of this TTL. By default, we recommend 86400 seconds (i.e., one day) for the *TTL_driver*.
2. *TTL_browser*. This value is used to set the expiration time for the resolved browser major version (e.g., Chrome 83). As explained in Section 6.2, browser version detection is a process based on the execution of shell commands. Since

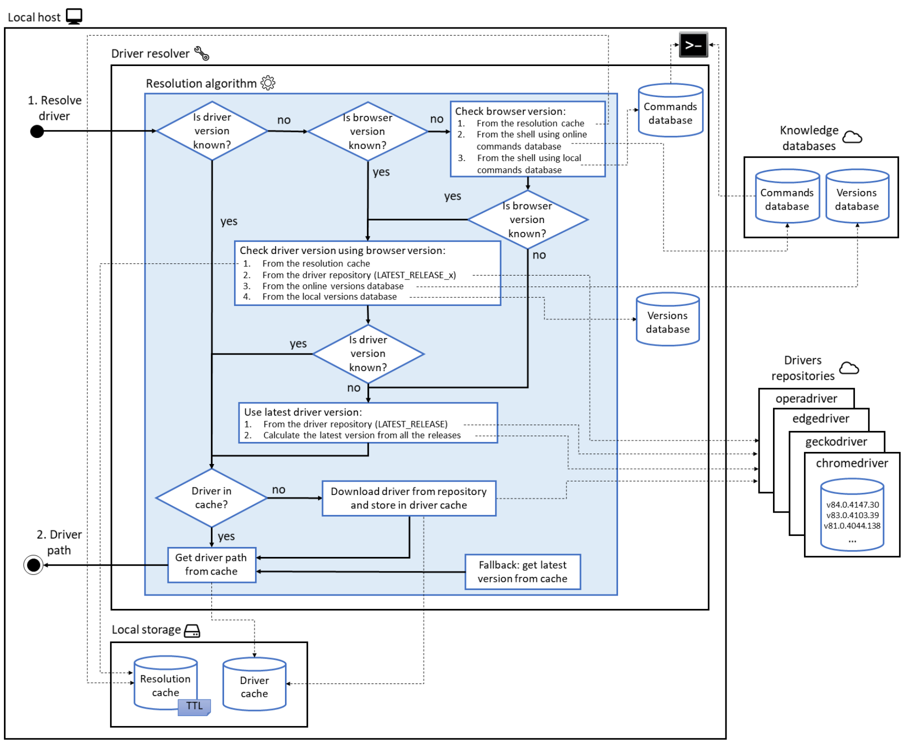

Local host
Driver resolver
Resolution algorithm
1. Resolve driver
Is driver version known?
no
Is browser version known?
no
Check browser version:
1. From the resolution cache
2. From the shell using online commands database
3. From the shell using local commands database
Commands database
yes
yes
Is browser version known?
yes
no
Knowledge databases
Commands database
Versions database
Check driver version using browser version:
1. From the resolution cache
2. From the driver repository (LATEST_RELEASE_x)
3. From the online versions database
4. From the local versions database
Versions database
yes
Is driver version known?
no
Drivers repositories
operadriver
edgedriver
geckodriver
chromedriver
Use latest driver version:
1. From the driver repository (LATEST_RELEASE)
2. Calculate the latest version from all the releases
Driver in cache?
no
Download driver from repository and store in driver cache
v84.0.4147.30
v83.0.4103.39
v81.0.4044.138
...
yes
2. Driver path
Get driver path from cache
Fallback: get latest version from cache
Local storage
Resolution cache
TTL
Driver cache

Fig. 5 Resolution algorithm workflow

this process is not very expensive in terms of time, this value can be low. By default, we recommend 3600 seconds (i.e., one hour) for the *TTL_browser*.

5.3 Resolution algorithm

As introduced in Figure 4, the resolution algorithm has a single input request (step "1. Resolve driver") and another single output point ("2. Driver path"). The algorithm is typically triggered by a Selenium WebDriver script that needs to access the driver path before controlling a given web browser (e.g., Chrome, Firefox, etc.). The complete workflow of the resolution algorithm, in context with the rest of the components, is highlighted in Figure 5.

The driver version is the first condition handled internally by the algorithm. When the driver version is unknown, it should be discovered. The algorithm needs to compile three parameters to that aim:

– The operating system (typically, Linux, Mac, or Windows).
– The architecture of the local host (usually 32 or 64 bits).
– The proper driver version for the requested browser.

To find out the operative system and the platform is straightforward using regular system calls. The selection of the driver version is a more complicated process. A possible strategy when resolving drivers for regular web browsers is to use the latest driver release. This policy can work in many cases, but not always (e.g., if the browser is not exactly the latest stable version). For that reason, the browser version should be determined dynamically by the algorithm. This detection is done using one of the following alternatives:

1. Reading the resolution cache. As introduced before, the browser version is stored persistently with a validity time determined by the *TTL_browser*.
2. Executing a command in the shell (using an online commands database). Browsers typically provide some CLI utilities for passing arguments or advanced browser setup, to name a few. Among these CLI features, the browser version can be requested. The driver resolver first uses a central (online) command database to carry out this process. This database contains a list of shell commands (e.g., `google-chrome --version`) for the different browsers and operative systems. The resolution algorithm executes these commands to find out the browser version.
3. Executing a command in the shell (using a local commands database). The driver resolver internally stores a local copy of the commands database. This copy is used as a fallback of the previous step.

If the browser version is known after this step, this value is used as input of the following stage, in which the specific driver version is determined using the browser version. The algorithm uses a cascade process in which each step is executed if the previous one is not effective. The driver version using the browser version is tried to be determined:

1. From the resolution cache. Again, the resolution cache is the first element to be requested. If the driver for the same browser was previously resolved (within the *TTL_driver* time), the driver version should be contained in the driver cache.
2. From the driver repository. As explained in Section 5.1, the maintainers of the drivers for Chrome and Edge also keep track of the proper version for the released drivers. In these cases, theoretically, this is the most accurate way to find the driver version based on the major browser version.
3. From the online versions database. If the previous step is not successful (or not possible, e.g., for Firefox or Opera drivers), the versions database is the only mechanism to match the driver version, as explained in Section 5.1. First, the online version of the driver database is used.
4. As the last resort, the local copy of the versions database (shipped with each release of the driver resolver) is employed.

After this process, two alternatives can occur. First, the driver version has not been discovered. When this happens, a second process to determine the driver version is started. In this case, instead of using the browser version, the algorithm calculates the latest possible driver version. To carry out this process, the pipeline is the following:

1. To find out the latest driver version from the data stored in the driver repository, in other words, to read the content of the `LATEST_RELEASE` file for chromedriver or the `LASTEST_STABLE` file for msedgedriver.

2. When this information is not available, or in the case of other browsers, the algorithm connects the corresponding driver repository and collects all the possible driver versions. The highest available version (i.e., the latest) is selected.

At this point, the algorithm has already determined the driver version to be used. This version can be either the version discovered using the browser version or the latest possible driver version. In any case, and using that driver version, the algorithm first queries the driver cache. If the driver is already in the cache, the algorithm ends. If not, the algorithm connects to the proper driver repository, downloads the driver, and stores it properly in the driver cache. In any case, the driver path from the resolution cache is used as the output.

Finally, the algorithm incorporates a fallback mechanism. If any uncontrolled exception occurs in any step during the algorithm (e.g., a connectivity problem), this fallback is invoked. In this case, as a last resort, the latest driver version from the driver cache is used as the output of the resolution algorithm.

6 Implementation

WebDriverManager is the reference implementation for the methodology presented in Section 5. WebDriverManager was first released in March 2015. It is developed in Java and is open-source, released under the terms of Apache Licence 2.0[12]. Its source code is publicly available on GitHub[13]. WebDriverManager allows the management of the drivers for two kinds of browsers:

– Modern browsers (evergreen): Chrome, Chromium, Firefox, Edge, and Opera Edge.
– Legacy browsers (deprecated): Internet Explorer and PhantomJS.

This section contains some of the most relevant implementation aspects of WebDriverManager, regarding the connection with driver repositories, knowledge databases (for versions and commands), persistence layer (driver and resolution cache), resolution algorithm, and execution methods (as Java dependency, CLI tool, server, Docker container, and Java agent).

6.1 Driver repositories

As explained in Section 5.3, the tool implementing the resolution algorithm should connect and download files from different websites. This feature hides several low-level implementation details. Table 2 summarizes these aspects. The first and second columns of this table refer to the browser and driver name respectively. The third column is the repository public URL. It is worth noting that the content of this URL contains human readable content. Thus, WebDriverManager does not use this URL to automate the download process. Instead, the URL included in the column “Endpoint” of Table 2 is handled internally by WebDriverManager.

[12] `https://www.apache.org/licenses/LICENSE-2.0`
[13] `https://github.com/bonigarcia/webdrivermanager`

Each driver vendor releases its driver using different storage technologies, such as Google Cloud Storage, Azure Storage, or GitHub, among others. WebDriverManager deals with this heterogeneity, connecting to the proper URL endpoint depicted in Table 2. WebDriverManager exchanges the appropriate messages with each endpoint, parsing the responses in the format (XML or JSON) managed by each storage server (column "Message format" in Table 2). Moreover, each maintainer decides how to package each version of the driver. Internally, WebDriverManager handles different data compression technologies (such as zip or tar.gz) to decompress the downloaded artifact properly (as depicted in the column "Artifact format" in Table 2). Finally, WebDriverManager filters the decompressed files, selecting only the actual driver and discarding other possible contents (e.g., license, readme).

6.2 Knowledge databases

As explained in Section 5 and illustrated in Figures 4 and 5, the driver resolver tool (i.e., WebDriverManager) should contain two different knowledge databases, for versions and commands.

The versions database maps the browser and driver versions. The resolution algorithm implemented internally by WebDriverManager reads a central database to get an updated snapshot of this mapping information. This database is maintained in the master branch of WebDriverManager in GitHub[14]. As shown in Listing 2, WebDriverManager uses Java properties to handle the versions database.

Listing 2 Versions database in WebDriverManager (implemented using Java properties)

```
# This file contains the known driver versions compatible with the corresponding browser
    version

# Browser: Google Chrome and Chromium - Driver: chromedriver
# Source: http://chromedriver.chromium.org/downloads
chrome85=85.0.4183.38
chrome84=84.0.4147.30
...
chrome72=2.46
chrome71=2.46
...

# Browser: Mozilla Firefox - Driver: geckodriver
# Source: https://github.com/mozilla/geckodriver/releases
firefox78=0.26.0
firefox77=0.26.0
...


# Browser: Opera - Driver: operadriver
# Source: https://github.com/operasoftware/operachromiumdriver/releases
opera69=83.0.4103.97
opera68=81.0.4044.113
...
opera60=2.45
```

[14] https://raw.githubusercontent.com/bonigarcia/webdrivermanager/master/src/main/resources/versions.properties

Table 2 Driver repositories connection details

Browser	**Driver**	**Repository**	**Storage type**	**Endpoint**	**Message format**	**Artifact format**
Chrome and Chromium	chromedriver	`http://chromedriver.chromium.org/`	Google Cloud Storage	`https://chromedriver.storage.googleapis.com/`	XML	.zip
Firefox	gecko-driver	`https://github.com/mozilla/geckodriver/releases`	GitHub	`https://api.github.com/repos/mozilla/geckodriver/releases`	JSON	.tar.gz and .zip
Edge	msedgedriver	`https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/`	Azure Storage	`https://msedgedriver.azureedge.net/`	XML	.zip
Opera	oper-adriver	`https://github.com/operasoftware/operachromiumdriver/releases`	GitHub	`https://api.github.com/repos/operasoftware/operachromiumdriver/releases`	JSON	.zip
Internet Explorer	IEDri-verServer	`https://www.selenium.dev/downloads/`	Google Cloud Storage	`https://selenium-release.storage.googleapis.com/`	XML	.zip
Phan-tomJS	phantomjs	`https://github.com/ariya/phantomjs/releases`	Bit-Bucket	`https://bitbucket.org/api/2.0/repositories/ariya/phantomjs/downloads`	JSON	.zip, .tar.gz, and .tar.bz2

```
opera59=2.45
...

# Browser: Microsoft Edge - Driver: msedgedriver
# Source: https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
edge86=86.0.573.0
edge85=85.0.564.0
...
edge42=6.17134
edge41=5.16299
```

The second knowledge database proposed in the methodology is called the commands database. This component stores a list of commands executed by the driver resolver in the shell to find out the version of the different browsers installed in the local machine. This database is also maintained in the master branch of WebDriverManager as a Java properties file[15].

The commands to be executed in the shell to find out the browser version are summarized in Table 3. These commands are browser and platform-dependent. For this reason, this table includes three columns: the browser whose version needs to be detected, the operating system, and the command. The first important aspect of the data presented in Table 3 is that browsers behave differently depending on the operating system. For example, to find out the Chrome version in Linux, WebDriverManager executes the command `google-chrome --version` and parses the output to get the version. Nevertheless, this command is not available on other operating systems. When the operating system is Windows, WebDriverManager uses a command-line utility called WMIC (Windows Management Instrumentation Command-Line). WMIC is a tool that allows to access and manage Windows resources [45]. As shown in Table 3, WebDriverManager uses the same WMIC command to obtain the versions of different browsers (Chrome, Chromium, Firefox, Edge, and Opera) in Windows. The main difficulty in executing this command is that the absolute path of the browser should be known. To find out this path, WebDriverManager manages different candidate paths by reading different environment variables available in Windows hosts (`PROGRAMFILES`, `PROGRAMFILES(X86)`, and `LOCALAPPDATA`) to find out the path of the program files. Furthermore, in the case of Windows, the Windows registry can be used to detect the versions of Chrome, Firefox, Edge, and Chromium.

6.3 Persistence layer

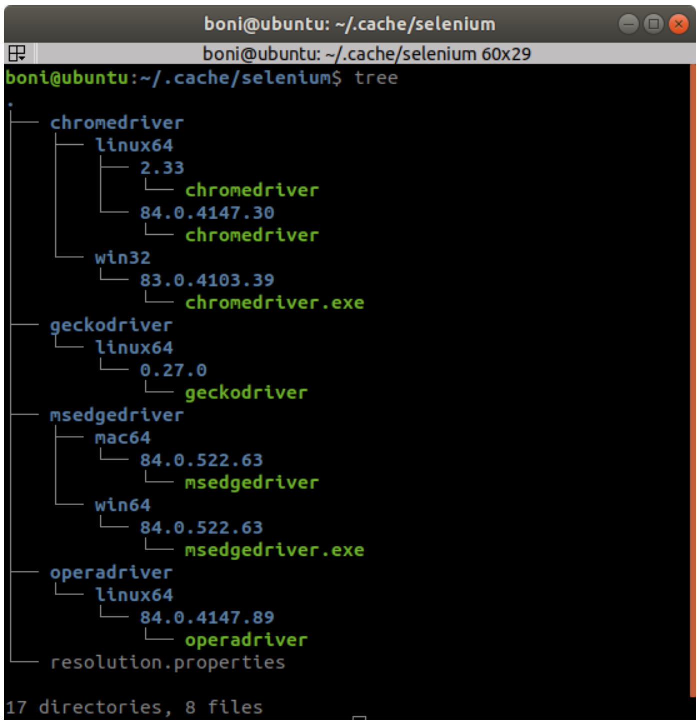
Two groups of data are stored persistently in the machine running the driver resolver tool: the driver and the resolution cache. The driver cache is a folder in the local machine in which the downloaded drivers are stored. By default, this folder is located in the path `~/.cache/selenium` by WebDriverManager. This cache allows reusing the drivers from different Selenium WebDriver executions, reducing the script startup drastically. The drivers are stored in a folder hierarchy with the following structure: `driverName/os+arch/driverVersion`, where `driverName` is the name of the driver (e.g., chromedriver, geckodriver), `os` is the operating system (win, linux, or mac), `arch` is the architecture (32 or 64), and `driverVersion` is the version of the driver (e.g., 84.0.4147.30, 0.26.0). Figure 6 shows an example of the driver cache tree structure in a Linux machine.

When coming to the resolution cache, WebDriverManager uses Java properties to store the resolved browser and driver versions and the TTL. This properties file is called `resolution.properties` and it is stored by default in the root of the driver cache by WebDriverManager (see Figure 6). Listing 3 shows an example of the content of this file. These lines show that the latest detected browser major versions are Chrome 84, Edge 84, and Firefox 78. For each of these browsers, the

[15] `https://raw.githubusercontent.com/bonigarcia/webdrivermanager/master/src/main/resources/commands.properties`

Table 3 Commands database

Browser	Operating system	Command
Chrome	Windows	`cmd.exe /C wmic datafile where name="%PROGRAMFILES(X86):\=\\%\\Google\\Chrome\\Application\\chrome.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%LOCALAPPDATA:\=\\%\\Google\\Chrome\\Application\\chrome.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES:\=\\%\\Google\\Chrome\\Application\\chrome.exe" get Version /value`
		`REG QUERY HKCU\Software\Google\Chrome\BLBeacon \v version`
	Linux	`google-chrome --version`
	Mac OS	`/Applications/Google Chrome.app/Contents/MacOS/Google Chrome --version`
Chromium	Windows	`cmd.exe /C wmic datafile where name="%LOCALAPPDATA:\=\\%\\Chromium\\Application\\chrome.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES(X86):\=\\%\\Chromium\\Application\\chrome.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES:\=\\%\\Chromium\\Application\\chrome.exe" get Version /value`
	Linux	`chromium-browser --version`
		`chromium --version`
	Mac OS	`/Applications/Chromium.app/Contents/MacOS/Chromium --version`
Firefox	Windows	`cmd.exe /C wmic datafile where name="%PROGRAMFILES:\=\\%\\Mozilla Firefox\\firefox.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES(X86):\=\\%\\Mozilla Firefox\\firefox.exe" get Version /value`
	Linux	`firefox -v`
	Mac OS	`/Applications/Firefox.app/Contents/MacOS/firefox -v`
Edge	Windows	`cmd.exe /C wmic datafile where name="%PROGRAMFILES(X86):\=\\%\\Microsoft\\Edge\\Application\\msedge.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES:\=\\%\\Microsoft\\Edge\\Application\\msedge.exe" get Version /value`
	Linux	`microsoft-edge --version`
	Mac OS	`/Applications/Microsoft Edge.app/Contents/MacOS/Microsoft Edge -version`
Opera	Windows	`cmd.exe /C wmic datafile where name="%LOCALAPPDATA:\=\\%\\Programs\\Opera\\Application\\launcher.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES:\=\\%\\Programs\\Opera\\Application\\launcher.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%LOCALAPPDATA:\=\\%\\Opera\Application\\launcher.exe" get Version /value`
		`cmd.exe /C wmic datafile where name="%PROGRAMFILES:\=\\%\\Opera\\Application\\launcher.exe" get Version /value`
	Linux	`opera --version`
	Mac OS	`/Applications/Opera.app/Contents/MacOS/Opera --version`



```
boni@ubuntu: ~/.cache/selenium
boni@ubuntu: ~/.cache/selenium 60x29
boni@ubuntu:~/.cache/selenium$ tree
.
├── chromedriver
│   ├── linux64
│   │   ├── 2.33
│   │   │   └── chromedriver
│   │   └── 84.0.4147.30
│   │       └── chromedriver
│   └── win32
│       └── 83.0.4103.39
│           └── chromedriver.exe
├── geckodriver
│   └── linux64
│       └── 0.27.0
│           └── geckodriver
├── msedgedriver
│   ├── mac64
│   │   └── 84.0.522.63
│   │       └── msedgedriver
│   └── win64
│       └── 84.0.522.63
│           └── msedgedriver.exe
├── operadriver
│   └── linux64
│       └── 84.0.4147.89
│           └── operadriver
└── resolution.properties

17 directories, 8 files
```

Fig. 6 Example of a driver cache content

proper driver versions are also stored (chromedriver 84.0.4147.30, msedgedriver 84.0.522.44, and geckodriver 0.26.0, respectively). These values are attached to a validity date, calculated with the TTL for browsers (1 hour by default) and the TTL for drivers (1 day by default).

Listing 3 Resolution cache in WebDriverManager (implemented using Java properties)

```
#WebDriverManager Resolution Cache (relationship between browsers and drivers versions
     previously resolved)
#Mon Jul 27 16:34:33 CEST 2020
chrome=84
chrome-ttl=17\:33\:04 27/07/2020 CEST
chrome84=84.0.4147.30
chrome84-ttl=16\:33\:04 28/07/2020 CEST
edge=84
edge-ttl=17\:34\:03 27/07/2020 CEST
edge84=84.0.522.44
edge84-ttl=16\:34\:03 28/07/2020 CEST
firefox=78
firefox-ttl=17\:33\:33 27/07/2020 CEST
firefox78=0.26.0
firefox78-ttl=16\:33\:33 28/07/2020 CEST
```

6.4 Resolution algorithm

The only particularity of the resolution algorithm described in Section 5.3 specific to Java is related to the required properties by Selenium WebDriver. As shown in Listing 1 of Section 4, the driver path should be exported as a Java property to control a browser with Selenium WebDriver. This driver path is obtained as the output of the resolution algorithm and exported properly by WebDriverManager.

6.5 Execution methods

As introduced before, WebDriverManager is a tool developed in Java. For this reason, the first versions of WebDriverManager were used only as a Java dependency. As long as the tool evolves, different usages were included: as a CLI tool, as a server, as a Docker container, and as a Java agent.

6.5.1 Java dependency

Starting with version 1.0.0, each Java release of WebDriverManager is available on the public repository Maven Central[16]. A build tool (such as Maven or Gradle) is typically used to declare and resolve the dependencies of a Java project. For that, the dependency coordinates (`groupId`, `artifactId`, and `version`) are specified [35]. Listing 4 shows the required configuration to use WebDriverManager by test classes in a Maven and Gradle project. In these snippets, the variable `wdm.version` should be substituted for a given WebDriverManager version (the latest available in Maven central is recommended by default).

Listing 4 WebDriverManager setup in Maven and Gradle

```
<dependency>
    <groupId>io.github.bonigarcia</groupId>
    <artifactId>webdrivermanager</artifactId>
    <version>${wdm.version}</version>
    <scope>test</scope>
</dependency>

dependencies {
    testImplementation("io.github.bonigarcia:webdrivermanager:${wdm.version}")
}
```

WebDriverManager provides a fluent API with a single entry point in a Java class called `WebDriverManager`. This class allows the use of a family of singleton objects called *managers*. Each manager is in charge of the resolution algorithm for a given driver, required to control a given browser with Selenium WebDriver. Each manager includes a method called `.setup()`, which allows executing the resolution algorithm (i.e., triggers the step "1. Resolve driver" depicted in Figure 5) for a given manager. Listing 5 shows the regular call for each of the provided managers (for Chrome, Chromium, Firefox, Edge, Opera, Internet Explorer, and PhantomJS):

[16] `https://search.maven.org/artifact/io.github.bonigarcia/webdrivermanager`

Listing 5 WebDriverManager regular call for each of the available managers

```
WebDriverManager.chromedriver().setup();
WebDriverManager.chromiumdriver().setup();
WebDriverManager.firefoxdriver().setup();
WebDriverManager.edgedriver().setup();
WebDriverManager.operadriver().setup();
WebDriverManager.iedriver().setup();
WebDriverManager.phantomjs().setup();
```

Listing 6 shows the recommended skeleton for a JUnit 4 test case using Selenium WebDriver and WebDriverManager. It is worth noting that before the execution of all tests in this class (method annotated with `@BeforeClass`), WebDriverManager is used to resolve the required driver to control Chrome with Selenium WebDriver.

Listing 6 JUnit 4 test skeleton using WebDriverManager and Selenium WebDriver

```
import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import io.github.bonigarcia.wdm.WebDriverManager;

public class ChromeTest {

    private WebDriver driver;

    @BeforeClass
    public static void setupClass() {
        WebDriverManager.chromedriver().setup();
    }

    @Before
    public void setupTest() {
        driver = new ChromeDriver();
    }

    @After
    public void teardown() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    public void test() {
        // Exercise and verify SUT
    }

}
```

The WebDriverManager API allows specifying a wide variety of configuration parameters. This API is based on the builder pattern [27], and so, different calls can be concatenated using the same manager before triggering the driver resolution (method `.setup()`). Appendix A shows the complete WebDriverManager API.

Listing 7 shows a couple of examples of WebDriverManager calls using various configuration methods. In the first line, WebDriverManager is used to resolve chromedriver. The method `clearResolutionCache()` forces to clear the resolution cache (i.e., reset the file `resolution.properties`) before the resolution. Then, the method `forceDownload()` avoids the usage of the driver cache. In other words, even when the driver is in the cache, it is compulsorily downloaded from the repository. In the second line, WebDriverManager is used to resolve geckodriver for a fixed major version of Firefox (i.e., `75`) and using a proxy for the network connection.

Listing 7 Examples of WebDriverManager calls with custom configuration

```
WebDriverManager.chromedriver().clearResolutionCache().forceDownload().setup();
WebDriverManager.firefoxdriver().browserVersion("75").proxy("http://myproxy.com:8080").
    setup();
```

In addition to the Java API, WebDriverManager allows providing custom configuration using two more alternatives. As shown in Appendix A, for each of the methods available in the WebDriverManager API, there are one or more configuration keys. The format of these keys is always the word `wdm.` followed by a camel-case string (e.g., `wdm.cachePath`). These keys can be passed to WebDriverManager using Java Virtual Machine (JVM) properties using the flag `-Dkey=value` when a JVM process is launched. Listing 8 shows an example in which the default path for driver cache is changed to a custom folder when executing the tests of a Maven project.

Listing 8 Test execution using Maven and WebDriverManager configuration keys

```
mvn test -Dwdm.cachePath=/my/custom/path/to/driver/binaries
```

Last but not least, environment variables can also be used to specify custom configuration parameters in WebDriverManager. To implement this feature, WebDriverManager uses a convention for the environment variables names based on the abovementioned configuration keys. This convention consists in converting to uppercase each WebDriverManager configuration key name, changing the dot symbol by an underscore. For example, to configure the driver cache path, WebDriverManager honors the value of environment variables `WDM_CACHEPATH`, if declared in the operating system.

6.5.2 CLI tool

As of version 2.2.0, WebDriverManager can also be used to resolve drivers executed from the shell as a regular CLI tool. Thus, every time a new version of WebDriverManager is released, a JAR (Java ARchive) file containing the compiled code together with all the dependencies is created. This JAR file, often known as fat-JAR, is published on GitHub[17].

The use of WebDriverManager as a CLI tool is straightforward. The WebDriverManager fat-JAR should be invoked using Java from the shell. This command should specify the driver to be resolved (`chrome`, `firefox`, `opera`, `edge`, `phantomjs`, or `iexplorer`). Listing 9 shows an example of the command required to resolve chromedriver from the shell using WebDriverManager.

[17] `https://github.com/bonigarcia/webdrivermanager/releases`

Listing 9 Execution example of WebDriverManager from a Windows shell

```
C:\Users\boni>java –jar webdrivermanager–4.4.0–fat.jar chrome

[INFO] Using WebDriverManager to resolve chrome
[DEBUG] Detecting chrome version using online commands.properties
[DEBUG] Running command on the shell: [cmd.exe, /C, wmic, datafile, where, name="%
      PROGRAMFILES(X86):\=\\%\\Google\\Chrome\\Application\\chrome.exe", get,
      Version, /value]
[DEBUG] Result: Version=89.0.4389.114
[DEBUG] Latest version of chromedriver according to https://chromedriver.storage.googleapis
      .com/LATEST_RELEASE_89 is 89.0.4389.23
[INFO] Using chromedriver 89.0.4389.23 (resolved driver for Chrome 89)
[INFO] Reading https://chromedriver.storage.googleapis.com/ to seek chromedriver
[DEBUG] Driver to be downloaded chromedriver 89.0.4389.23
[INFO] Downloading https://chromedriver.storage.googleapis.com/89.0.4389.23/
      chromedriver_win32.zip
[INFO] Extracting driver from compressed file chromedriver_win32.zip
[INFO] Driver location: C:\Users\boni\Downloads\chromedriver.exe
```

All the configuration keys defined in Appendix A can be used to customize the behavior of WebDriverManager also from the shell. As introduced in Section 6.5.1, this is done by passing JVM properties using the flag `-Dkey=value` from the shell. For instance, the command `java -Dwdm.chromeVersion=89 -jar webdrivermanager-4.4.0-fat.jar chrome` is used to resolve the proper driver for a given version of Chrome.

6.5.3 Server

As of version 3.0.0, the third way to execute WebDriverManager is as a server. In this execution model, WebDriverManager offers a REST-like API to resolve drivers. There are six different endpoints in this API to resolve the supported drivers in WebDriverManager: `/chromedriver`, `/firefoxdriver`, `/operadriver`, `/edgedriver`, `/phantomjs`, and `/iedriver`. Each time a `GET` request is made to a WebDriverManager server, the resolution algorithm described is executed, and the resulting driver is sent as an attachment in the HTTP response.

Listing 10 shows an example of WebDriverManager executed as a server. The fat-JAR is used to launch the WebDriverManager server, which by default, listens to incoming requests using port 4041. When the server is up and running, a client can create a request using `GET`. For instance, the URL `http://wdm-server:-4041/chromedriver?chromeVersion=84` uses the endpoint `/chromedriver` to resolve the driver for Chrome. In addition, this example URL contains a configuration parameter. Again, all the configuration keys defined in Appendix A can be reused to customize the request query. For the sake of simplicity, the prefix `wdm.` is not be included in the URL query, but only the configuration key name in camel case. In the example contained in Listing 10, the configuration key `chromeVersion` is used to specify a given version of Chrome.

Listing 10 Execution example of WebDriverManager as a server

```
C:\Users\boni>java –jar webdrivermanager–4.4.0–fat.jar server
[INFO] WebDriverManager server listening on port 4041
[INFO] Server request: GET /chromedriver
[INFO] Server query string for configuration {chromeVersion=[89]}
```

```
[DEBUG] Resolution chrome89=89.0.4389.23 in cache (valid until 17:01:40 07/04/2021CEST)
[INFO] Using chromedriver 89.0.4389.23 (resolved driver for Chrome 89)
[DEBUG] Driver chromedriver 89.0.4389.23 found in cache
[INFO] Driver location: C:\Users\boni\.cache\selenium\chromedriver\win32\89.0.4389.23\
      chromedriver.exe
[INFO] Server response: chromedriver.exe 89.0.4389.23 (10695680 bytes)
```

6.5.4 Docker container

Starting with version 4.0.0, each time a WebDriverManager new version is released, the corresponding Docker container is published in Docker Hub[18]. Internally, these containers are based on a slim version of Debian shipped together with OpenJDK and the WebDriverManager fat-JAR. This way, both WebDriverManager CLI and server can be executed seamlessly as a portable container through a Docker engine.

Listing 11 and Listing 12 show a couple of execution examples of WebDriverManager as a Docker container. First, Listing 11 shows an example of the WebDriverManager server running inside a Docker container. The traces report how the server resolves a given version of chromedriver. Second, Listing 12 how to execute the WebDriverManager CLI with Docker. In this example, a custom configuration setup is specified defining environmental variables (flag `-e`) in the Docker container. Moreover, thanks to the use of volumes, the resulting driver file is stored in the local storage of the Docker host.

Listing 11 Execution example of WebDriverManager server as in a Docker container

```
boni@ubuntu:~$ docker run −p 4041:4041 bonigarcia/webdrivermanager:4.4.0
[INFO] WebDriverManager server listening on port 4041
[INFO] Server request: GET /chromedriver
[INFO] Server query string for configuration {chromeDriverVersion=[81.0.4044.138]}
[DEBUG] Created new resolution cache file at /root/resolution.properties
[INFO] Reading https://chromedriver.storage.googleapis.com/ to seek chromedriver
[DEBUG] Driver to be downloaded chromedriver 81.0.4044.138
[INFO] Downloading https://chromedriver.storage.googleapis.com/81.0.4044.138/
      chromedriver_linux64.zip
[INFO] Extracting binary from compressed file chromedriver_linux64.zip
[INFO] Driver location: /wdm/chromedriver/linux64/81.0.4044.138/chromedriver
[INFO] Server response: chromedriver 81.0.4044.138 (10317568 bytes)
```

Listing 12 Execution example of WebDriverManager CLI as in a Docker container

```
boni@ubuntu:~$ docker run −e BROWSER=chrome −e WDM_CHROMEVERSION=84 −e
      WDM_OS=LINUX −v ${PWD}:/wdm bonigarcia/webdrivermanager:4.4.0
[INFO] Using WebDriverManager to resolve chrome
[DEBUG] Created new resolution cache file at /root/resolution.properties
[DEBUG] Latest version of chromedriver according to https://chromedriver.storage.googleapis
      .com/LATEST_RELEASE_84 is 84.0.4147.30
[INFO] Using chromedriver 84.0.4147.30 (resolved driver for Chrome 84)
[INFO] Reading https://chromedriver.storage.googleapis.com/ to seek chromedriver
[DEBUG] Driver to be downloaded chromedriver 84.0.4147.30
[INFO] Downloading https://chromedriver.storage.googleapis.com/84.0.4147.30/
      chromedriver_linux64.zip
[INFO] Extracting binary from compressed file chromedriver_linux64.zip
[INFO] Driver location: /wdm/chromedriver
```

[18] `https://hub.docker.com/repository/docker/bonigarcia/webdrivermanager`

6.5.5 Java agent

As of version 4.0.0, WebDriverManager can be used as a Java agent. A Java agent uses the JVM instrumentation capabilities to add bytecodes to existing compiled Java classes [6]. Concretely, the WebDriverManager agent checks the objects created in a JVM. Just before Selenium WebDriver objects are instantiated (i.e., `org.openqa.selenium.chrome.ChromeDriver`, `org.openqa.selenium.firefox.FirefoxDriver`, `org.openqa.selenium.opera.OperaDriver`, `org.openqa.selenium.edge.EdgeDriver`, `org.openqa.selenium.ie.InternetExplorerDriver`), and `org.openqa.selenium.phantomjs.PhantomJSDriver`, the corresponding setup call in WebDriverManager is executed to resolve the required driver (chromedriver, geckodriver, msedgedriver, and so for). This way, Selenium WebDriver can be used from scripts with total independence of the driver management. Listing 13 shows an example of a JUnit 4 test using Selenium WebDriver and WebDriverManager agent. As can be seen, there is no need for explicit driver resolution, since it is done automatically by the WebDriverManager agent when the test is executed.

Listing 13 JUnit 4 test skeleton using WebDriverManager agent and Selenium WebDriver

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class ChromeTest {

    private WebDriver driver;

    @Before
    public void setupTest() {
        driver = new ChromeDriver();
    }

    @After
    public void teardown() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    public void test() {
        // Exercise and verify SUT
    }

}
```

The JVM flag `-javaagent` can be used to specify the path of the WebDriverManager fat-JAR when configuring WebDriverManager as an agent. Alternatively, a build tool like Maven can be used[19].

[19] `https://github.com/bonigarcia/wdm-agent-example`

Table 4 Comparison of the different execution methods of WebDriverManager

Execution mode	**Usage**	**Pros**	**Cons**
Java dependency	Driver management when using Java as language binding for Selenium WebDriver	It allows custom setup (through the WebDriverManager API)	It requires some coding effort
Java agent		Zero coding effort	It is only valid for common use cases (e.g., when using evergreen browsers without extra setup)
CLI tool	Agnostic driver management (i.e., not linked to a particular language binding for Selenium WebDriver)	Interoperable with shell scripting	It requires additional effort to handle the downloaded drivers
Server		Cross-platform	
Docker container		Portable	

6.5.6 Recap and other usages

As explained in the previous subsections, WebDriverManager can be executed in different manners. Table 4 can be used as a reference to select one or another alternative. This table describes the typical usage together with the main advantages and inconveniences of each execution method.

To conclude this section, it is worth mentioning that WebDriverManager can also be used in different scopes than Selenium WebDriver. The first one is to ease the driver management of Appium[20], which is a testing framework that extends Selenium WebDriver to carry out automated testing of mobile applications. In the same way that Selenium WebDriver, Appium needs to use drivers to control browsers in mobile devices (e.g., Android). Listing 14 shows an example of how the WebDriverManager API can be used in conjunction with Appium.

Listing 14 Example of WebDriverManager usage to resolve drivers for Appium

```
public WebDriver createAndroidChromeDriver(String chromeVersion,
        String deviceName, URL appiumServerUrl) {
    // Resolve chromedriver and its driver path using WebDriverManager
    WebDriverManager.chromedriver().browserVersion(chromeVersion).setup();
    String chromedriverPath = WebDriverManager.chromedriver()
            .getDownloadedDriverPath();

    DesiredCapabilities capabilities = new DesiredCapabilities();
    capabilities.setCapability("browserName", "chrome");
    capabilities.setCapability("deviceName", deviceName);
    capabilities.setCapability("platformName", "android");
    capabilities.setCapability("chromedriverExecutable", chromedriverPath);

    return new AndroidDriver<WebElement>(appiumServerUrl, capabilities);
}
```

[20] https://appium.io/

Another tool that can be supported by WebDriverManager is Selenium Grid. Selenium Grid[21] is an extension of Selenium WebDriver that provides web browsers hosted on remote machines. In Selenium Grid, a central node (called the Selenium Hub or Server) keeps track of different nodes and proxies requests from Selenium scripts. To registers nodes in the Selenium Hub, the use of drivers is mandatory. To manage these drivers automatically, WebDriverManager can be used. As an example, Listing 15 shows how to invoke WebDriverManager CLI to register a node in Selenium Grid.

Listing 15 Example of WebDriverManager usage to register a node in Selenium Grid

```
C:\Users\boni>java -jar webdrivermanager-4.4.0-fat.jar chrome
[INFO] Using WebDriverManager to resolve chrome
[DEBUG] Detecting chrome version using online commands.properties
[DEBUG] Running command on the shell: [cmd.exe, /C, wmic, datafile, where, name="%
      PROGRAMFILES(X86):\=\\%\\Google\\Chrome\\Application\\chrome.exe", get,
      Version, /value]
[DEBUG] Result: Version=89.0.4389.114
[DEBUG] Latest version of chromedriver according to https://chromedriver.storage.googleapis
      .com/LATEST_RELEASE_89 is 89.0.4389.23
[INFO] Using chromedriver 89.0.4389.23 (resolved driver for Chrome 89)
[INFO] Reading https://chromedriver.storage.googleapis.com/ to seek chromedriver
[DEBUG] Driver to be downloaded chromedriver 89.0.4389.23
[INFO] Downloading https://chromedriver.storage.googleapis.com/89.0.4389.23/
      chromedriver_win32.zip
[INFO] Extracting driver from compressed file chromedriver_win32.zip
[INFO] Driver location: C:\Users\boni\Downloads\chromedriver.exe

C:\Users\boni>java -Dwebdriver.chrome.driver="chromedriver.exe" -jar selenium-server-
      standalone-3.14.0.jar -role node -hub http://localhost:4444/grid/register -browser
      browserName=chrome -port 5555
17:13:55.680 INFO [GridLauncherV3.launch] - Selenium build info: version: '3.14.0', revision:
      'aacccce0'
17:13:55.684 INFO [GridLauncherV3$3.launch] - Launching a Selenium Grid node on port
      5555
2021-01-17 17:13:56.637:INFO::main: Logging initialized @1238ms to org.seleniumhq.jetty9.
      util.log.StdErrLog
17:13:56.766 INFO [SeleniumServer.boot] - Selenium Server is up and running on port 5555
17:13:56.766 INFO [GridLauncherV3$3.launch] - Selenium Grid node is up and ready to
      register to the hub
17:13:57.079 INFO [SelfRegisteringRemote$1.run] - Starting auto registration thread. Will
      try to register every 5000 ms.
17:13:57.080 INFO [SelfRegisteringRemote.registerToHub] - Registering the node to the hub:
       http://localhost:4444/grid/register
17:13:57.883 INFO [SelfRegisteringRemote.registerToHub] - The node is registered to the
      hub and ready to use
```

7 Validation

We carried out a survey to provide an empirical evaluation of the proposed approach. This survey was launched to the WebDriverManager community in 2020. The objective of this study is to validate the central hypothesis of this piece of research, i.e.:

[21] `https://www.selenium.dev/documentation/en/grid/`

H1: WebDriverManager is a tool that facilitates the development process with Selenium WebDriver.

We designed a questionnaire divided into two parts to validate this hypothesis. First, we studied relevant aspects related to the motivation and adoption of WebDriverManager by its users. Second, we assessed the usability of the WebDriverManager API using Clarke's dimensions (as introduced in Section 3.2).

7.1 Study design

7.1.1 Questionnaire

The first part of the survey aims to discover how WebDriverManager is perceived and used by its users. Hence, the questionnaire includes six questions, namely:

1. What is your motivation to use WebDriverManager?
2. How did you discover WebDriverManager?
3. What is the primary programming language you use with WebDriverManager?
4. Which driver/browser do you resolve with WebDriverManager?
5. How do you execute WebDriverManager?
6. How do you use the WebDriverManager API?

All these six questions provide a list of possible answers which can be selected by respondents. Furthermore, all questions include an open-ended field for further options (the "other" field), aimed to complement the answers with custom options. All questions (except number 3) have been designed as multiple-choice closed-ended items [51]. This way, respondents can choose one or more answers to the given questions. In contrast, question 3 is designed to be a single-choice closed-ended. We restricted this answer to be unique to determine the preferred language used in conjunction with WebDrivermanager.

The second part of the survey analyzes the WebDriverManager API usability using Clark's dimensions (understandability, abstraction, expressiveness, reusability, and learnability). For this, as shown in Table 5, the questionnaire has 20 assertions to characterize these dimensions. We asked respondents to provide their degree of agreement or disagreement using a 5-point Likert scale (1=fully disagree, 2=disagree, 3=neutral, 4=agree, 5=fully agree) for each assertion. Following standard practices in social research, some of the assertions are stated in negative terms to ensure the internal consistency on the instrument [20]. We refer to these statements as "N-assertions." In the data analysis, the N-assertions value is inverted (1 becomes 5, 2 becomes 4, 4 becomes 2, and 5 becomes 1).

In addition to these two main sections, we included an additional part in the questionnaire to gather demographics data: nationality, age, and gender. Finally, we included an optional open-ended question to write further feedback or comments about WebDriverManager.

7.1.2 Participants recruitment

We implemented the questionnaire using Google Forms. To maximize the number of participants, and because the target of our survey is the WebDriverManager users, we asked for participation by including the questionnaire URL in the

Table 5 Research questionnaire used to assess the participants' perception of the WebDriver-Manager API usability following the Clarke's dimensions

Dimension	**Id**	**Assertion**
Understandability	U.1	The WebDriverManager API is, in general, easy to understand
	U.2	In WebDriverManager API methods are descriptive and unambiguous
	U.3	(N) I need to keep track of hidden information not represented by the WebDriverManager API to create my Selenium tests
	U.4	(N) The WebDriverManager API is obscure, and it takes a huge effort to use it
Abstraction	A.1	The WebDriverManager API makes it simple to create Selenium tests without needing to worry about the low-level details
	A.2	(N) I needed to adapt the WebDriverManager API (e.g., inheriting, overriding, etc.) for having it meet my needs
	A.3	(N) It's necessary to understand how driver binaries work for being able to use WebDriverManager
	A.4	I feel appealing and attractive the general approach of the WebDriverManager API
Expressiveness	E.1	Developing with WebDriverManager fully matches the expectations I had
	E.2	Reading an application code, I can understand what the application is doing in a simple way
	E.3	(N) There are missing features in WebDriverManager API that make not possible to implement interesting Selenium tests
	E.4	(N) Programming with WebDriverManager is error-prone. You need to take into consideration a lot of details for having an application working
Reusability	R.1	(N) Creating Selenium tests with WebDriverManager requires too long and verbose code specifying too many things
	R.2	My code using WebDriverManager can be maintained and evolved easily
	R.3	I can reuse WebDriverManager related code in a simple way
	R.4	(N) When using WebDriverManager, there are many different ways of doing the same thing, and I need to take too many decisions in the process
Learnability	L.1	I learned how to use WebDriverManager incrementally, starting with simple concepts and progressing towards complex Selenium tests
	L.2	(N) Programming with WebDriverManager requires learning a lot of concepts, even for simple Selenium tests
	L.3	(N) I needed to read all the documentation be able to create my first application
	L.4	Reading simple examples made me possible to understand the tool and to create later Selenium tests complying with my requirements

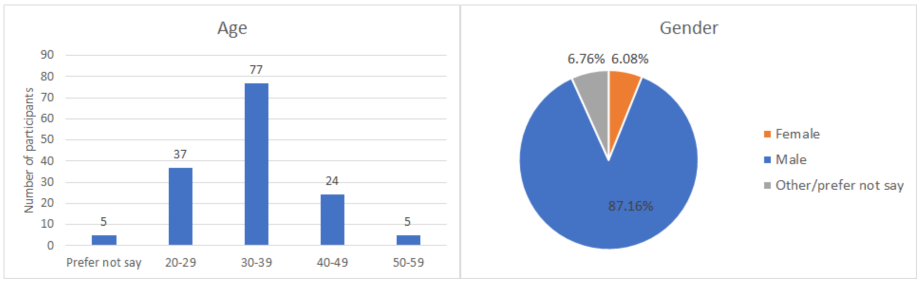


Fig. 7 Charts showing the total number of participants including age (left) and gender (right)

WebDriverManager logs. Listing 16 shows these traces. WebDriverManager 4.0.0, released on 12th May 2020, published these traces, which were removed in WebDriverManager 4.1.0, released on 14th July 2020.

Listing 16 Request for participation to the survey in WebDriverManager 4.0.0 log traces

```
[INFO] Please answer the following questionnaire based on your experience with
       WebDriverManager. Thanks a lot!
[INFO] ====> http://tiny.cc/wdm-survey <====
```

7.2 Results

This section provides a comprehensive summary of the obtained survey results. The findings and implications of these data and are later discussed in Section 8.

7.2.1 Participants analysis

As introduced before, the survey was active for two months in 2020. In this period, we collected a total of 148 answers worldwide. From a demographic perspective, ages were distributed between 20 and 59 years. As depicted in the left chart of Figure 7, the group with more respondents was the thirties, with a total of 52.03% of participants. A total of 5 respondents (3.28%) preferred not to reveal their ages. Regarding gender, as depicted in the right chart of Figure 7, the questionnaire was completed by 129 men (87.16%), 9 women (6.08%), and 10 people (6.76%) that preferred not to specify any gender option.

When coming to nationality, as shown in Figure 8, the poll was completed from 43 different countries from the 5 continents. The countries with more respondents were the United States (16.22%), India (9.46%), Germany (8.78%), Poland (8.11%), and Spain (6.76%).

7.2.2 WebDriverManager motivation and adoption

This section summarizes the results for each of the six questions designed in Section 7.1.1, aimed to discover how WebDriverManager is perceived and used by its users. Pareto charts [59] are used to visualize the results. These diagrams are derived from the Pareto principle, which states that about 80% of the effects of

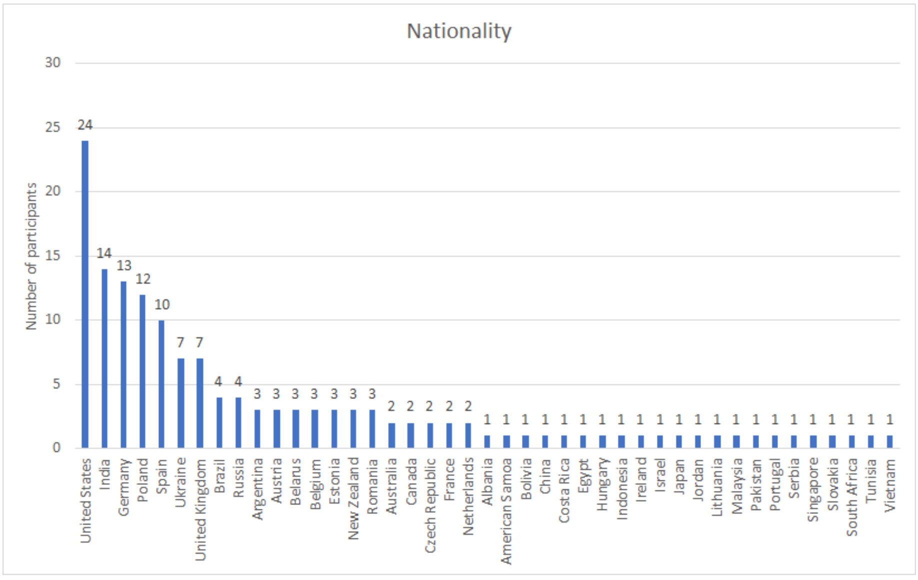


Fig. 8 Total number of respondents organized by country

a given phenomenon come from 20% of the causes [13]. In the resulting charts, each bar corresponds to the number of times the participants select a given response. Also, these charts include the Pareto line, which represents the cumulative total percentage for these responses. Using 80% as the standard cut-off line, the relevant answers (the “vital few”) can be distinguished simply by checking when the cumulative total percentage is below 80%. The rest of the items (the “trivial many”) correspond to the answers with a cumulative total of over 80%.

The first question in our questionnaire is related to the motivation to use WebDriverManger. Figure 9 shows the obtained results. Following the 80/20 rule described before, it can be checked that the most important factors to use WebDriverManager are related to the automation download of drivers (selected 113 times, i.e., by the 34.66% of the participants) and the automated update of these drivers (selected 103 times, i.e., 31.6% of the participants).

The second question tries to figure out how the respondents discovered WebDriverManager. Using the 80/20 rule again, it can be seen that the preferred answer is “Seeking the Web by myself,” selected by 58 out of the 148 total participants (i.e., a quota of 39.19%) and “referred by a colleague,” chosen by 42 participants (28.38%).

Questions 3 of the survey is concerned with the preferred programming language used in conjunction with WebDriverManager. As depicted in Figure 11, the 80/20 cannot be applied in these results since there is only one answer majority selected in the poll: the Java language. 93.92% chose this language, while the resting 6.08% quota is made up of other languages (namely, Groovy, Kotlin, Python, PHP, Ruby, and Scala), which can be seen as a residual usage of WebDriverManager.

When coming to question 4, we wanted to discover the driver and the corresponding browser resolved with WebDriverManager by the respondents. Figure 12 shows the results for this question. Applying the Pareto principle again, it can be

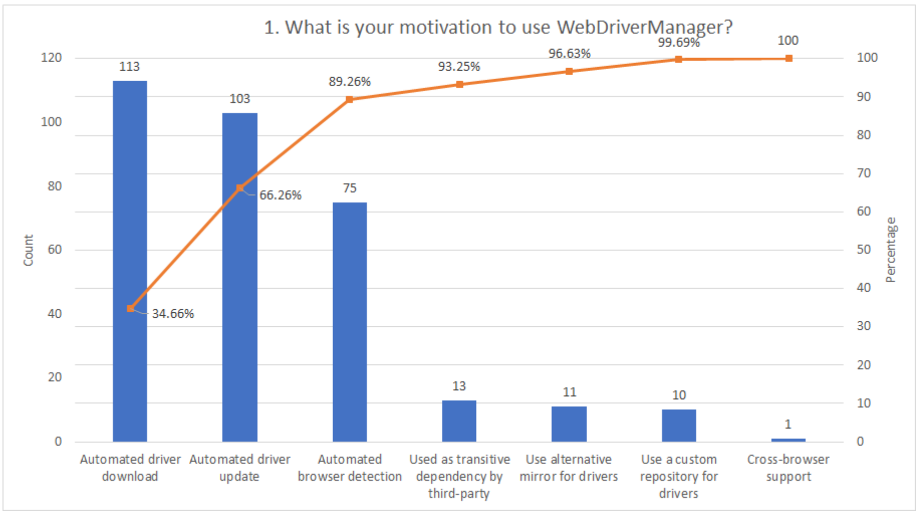


Fig. 9 Pareto chart illustrating the motivation to use WebDriverManager by the survey participants

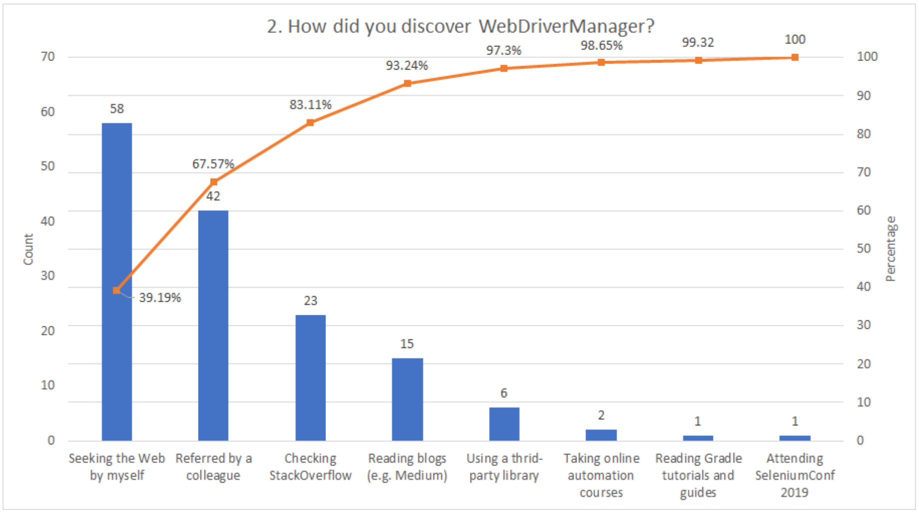


Fig. 10 Pareto chart illustrating how the survey participants discovered WebDriverManager

discovered the few vital answers, which in this case correspond to chomedriver for Chrome (38.64%), geckodriver for Firefox (25.85%), and msedgedriver for Edge (11.65%). These results are aligned with a recent analysis of the Selenium ecosystem, in which the top evergreen browser used with Selenium WebDriver were precisely Chrome, Firefox, and Edge [29].

The fifth question of the questionnaire is related to the execution mode of WebDriverManager used by the participants. Figure 13 shows the results of this question. The 80/20 rule allows identifying a single response, which is the most

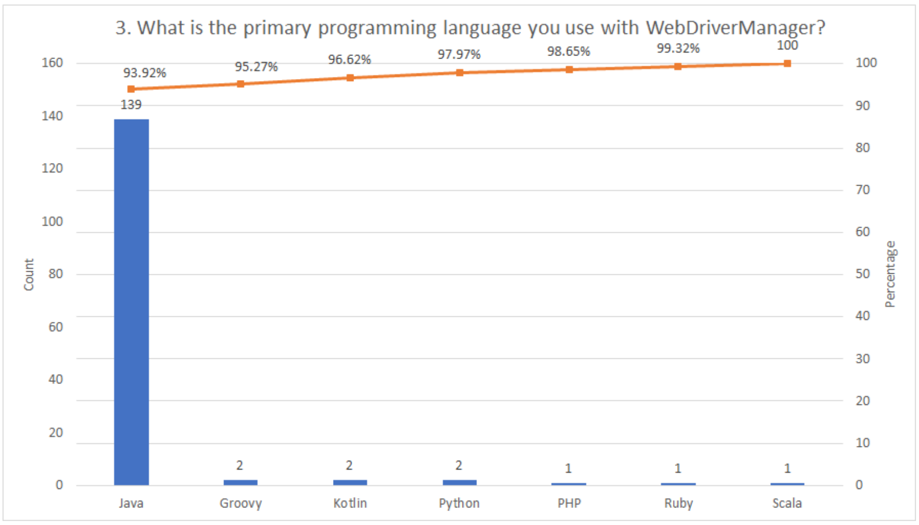


Fig. 11 Pareto chart illustrating the preferred programming language of the survey participants

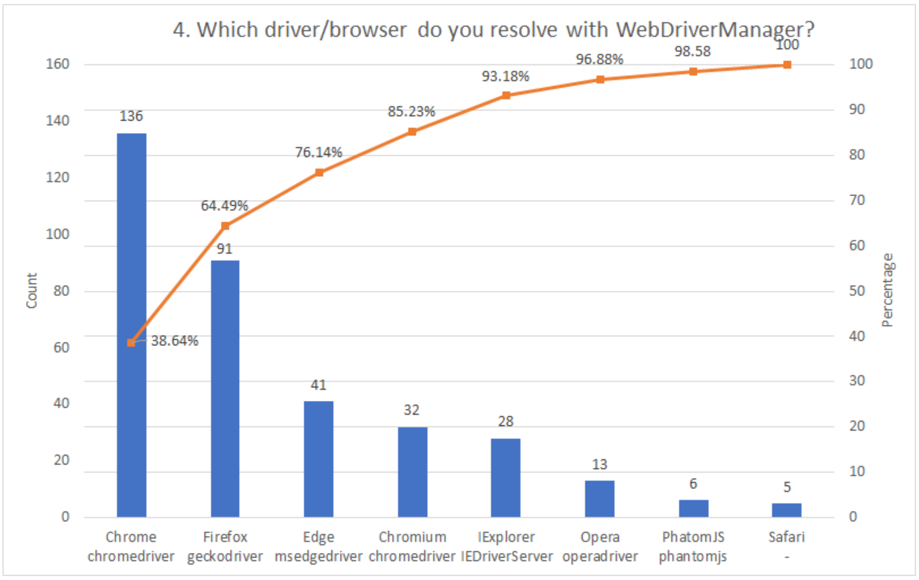


Fig. 12 Pareto chart illustrating how the driver and browser resolved with WebDriverManager by participants

relevant to the participants: as a direct Java dependency, selected by 119 out of 148 participants (i.e., a quota of 67.61%).

The final question of this part of the poll surveys the participants about how they use the WebDriverManager API. In this case, there are two answers with a cumulate percentage below 80%. First, using the regular API call in Java (e.g., `WebDriverManager.chromedriver.setup();`), selected by the 62.79% of respon-

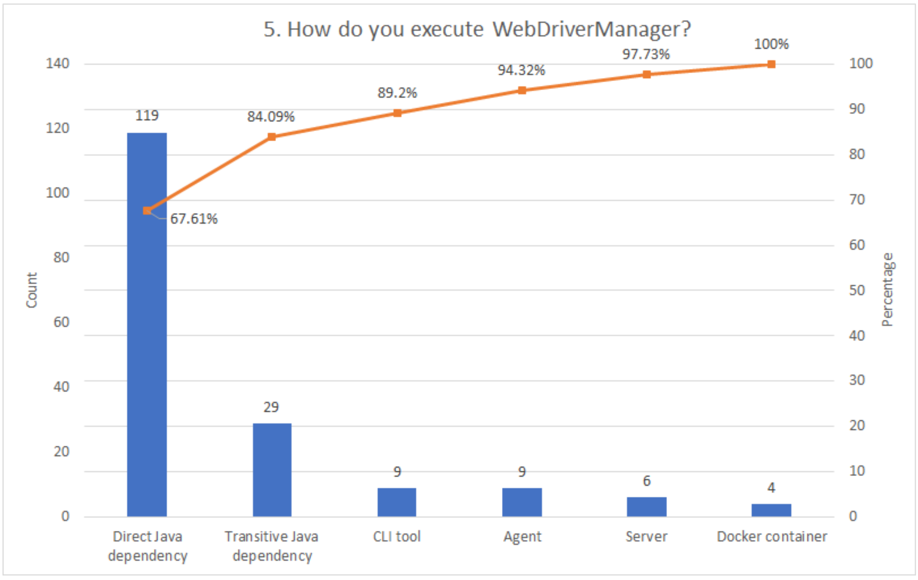


Fig. 13 Pareto chart illustrating the WebDriverManager execution type used by participants

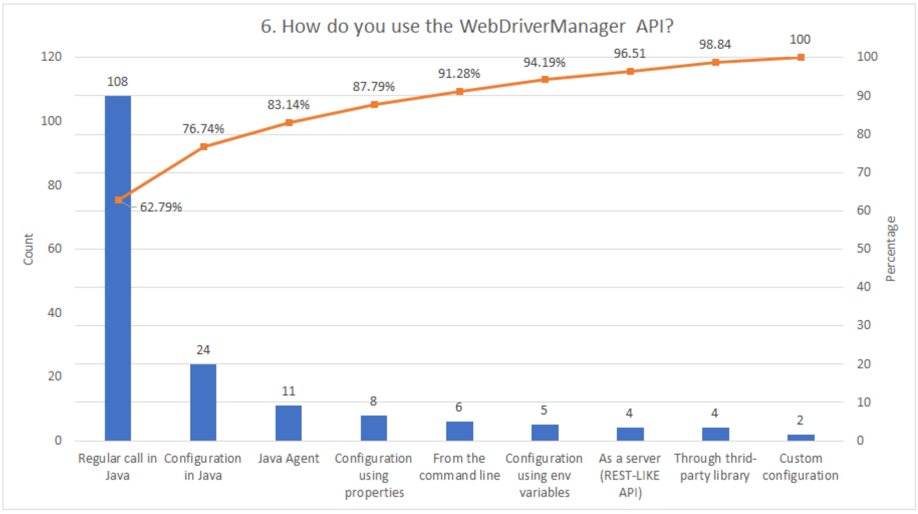


Fig. 14 Pareto chart illustrating the WebDriverManager API usage by participants

dents. And second, using some custom configuration method available in the Java API (e.g., `.browserVersion()`, `.proxy()`, or others), selected by the 13.95% of the respondents (see Figure 14).

7.2.3 WebDriverManager API usability

The results of the usability analysis are shown in Table 6. We dropped the monotonous set of answers from the data analysis to improve reliability. In other words, we did

Table 6 Main statistics results of the usability analysis of the WebDriverManager using Clarke's dimensions

Dimension	**Id**	**Mean**	**Median**	**Min**	**Max**	**StdDev**
Understandability		4.13	4.25	1.5	5	0.91
	U.1	4.38	4	2	5	0.7
	U.2	4.15	4	2	5	0.81
	U.3	3.7	4	1	5	1.14
	U.4	4.29	5	1	5	0.98
Abstraction		4.16	4.25	2.25	5	0.86
	A.1	4.29	4	2	5	0.8
	A.2	4.23	5	2	5	0.96
	A.3	3.89	4	2	5	0.94
	A.4	4.22	4	3	5	0.73
Expressiveness		4.03	4.25	1.75	5	0.94
	E.1	4.3	5	3	5	0.79
	E.2	4.17	4	2	5	0.78
	E.3	3.86	4	1	5	1.11
	E.4	3.79	4	1	5	1.08
Reusability		4.13	4.5	1.5	5	0.9
	R.1	4.23	5	1	5	1.05
	R.2	4.37	5	2	5	0.75
	R.3	4.21	4	2	5	0.8
	R.4	3.71	4	1	5	0.98
Learnability		4.08	4	2	5	0.88
	L.1	3.89	4	2	5	0.81
	L.2	4.16	4	2	5	0.92
	L.3	4.06	4	2	5	1.01
	L.4	4.22	4	2	5	0.78

not include those answers in which each assertion is rated in the same way. We considered this fact is an indicator of deceptive responses (e.g., each statement rated with 5 by the same respondent). Following this strategy, we reduced the valid responses to 112 (out of the 148 total). Using this group of answers, and as explained before, we inverted the value of the N-assertions. This way, the values represented in Table 6 are interpreted positively (i.e., the higher the value, the better usability). As depicted in the left part of Figure 15, on average, respondents feel the WebDriverManager API usability is adequate since every dimension (understandability, abstraction, expressiveness, reusability, and learnability) has a mean value close to 4, which is the ranking for agreement using the 5-point Likert scale.

For completeness, we evaluated the relationship between the usability dimensions results with the nationality of the respondents, grouped in a per-continent way. As can be seen, there are no significant changes in the API usability perception among different respondents of different continents.

7.2.4 Open comments

Last but not least, we analyzed the additional comments provided by the survey participants. Although the open-ended field for this purpose was optional in the questionnaire, it was reported by 60 out of the 148 participants (i.e., 40.54%). We

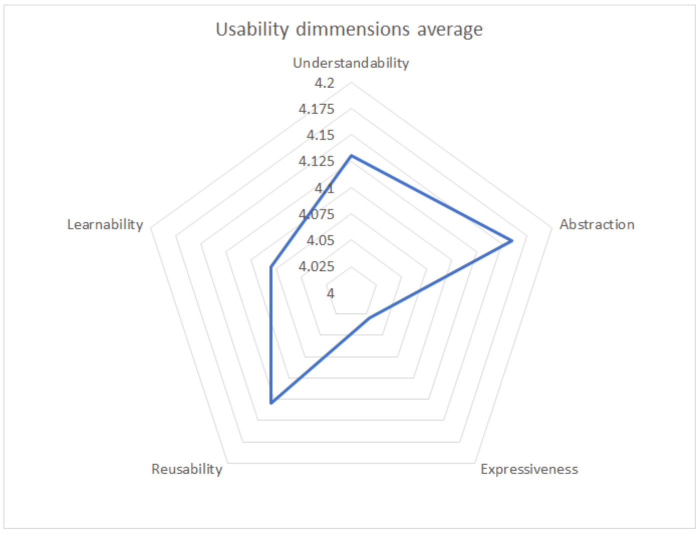


Fig. 15 Radar chart showing the average values of the usability dimensions results

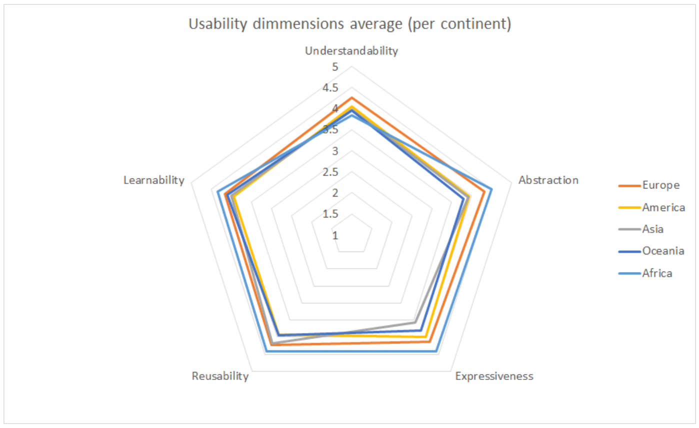


Fig. 16 Radar chart showing the average values of the usability dimensions results grouped per respondent's continent

analyzed each of these comments, and then, we categorized them in one or several of the following categories:

– Positive/gratitude: Acknowledge or recognition for creating and maintaining WebDriverManager.
– Negative/complaint: Criticisms or objections about some specific aspect of WebDriverManager.
– Feature request: Proposition for a new characteristic to be incorporated in WebDriverManager.
– Issue report: Description of a particular problem.

As shown in Figure 17, 48 respondents (i.e., 80% of the participant who replied something in the open comments field) showed gratitude or positive comments

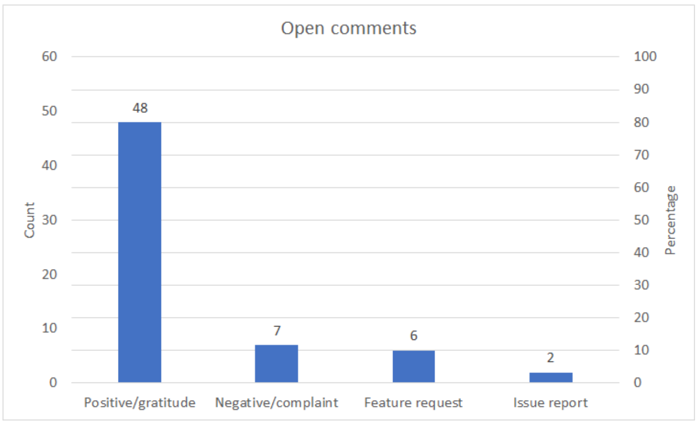


Fig. 17 Bar chart showing the open comments categories

about WebDriverManager. Then, 7 participants (i.e., 11.67%) wrote some critic or negative feedback. Some of these criticisms were related to the documentation (e.g., lack of details in some aspects, such as the browser version detection) and the publication of the survey (i.e., the poll URL attached in the WebDriverManager traces). Next, 6 users (i.e., 10%) considered this field an opportunity to ask for a new feature, for instance, adding more descriptive traces in case of errors (e.g., lack of connectivity). Finally, 2 users (i.e., 3.33%) reported some issues.

7.3 Threats to validity

Following the commonly accepted methods for evaluating assessment data, we discuss the main threats to validity in three categories: construct, internal, and external validity [22].

7.3.1 Construct validity

Construct validity refers to the degree to which an instrument measures what it claims. As explained before, our survey has two main sections. First, we wanted to evaluate the extent to which their users adopt WebDriverManager. Second, we assessed the usability of the WebDriverManager API following Clarke's dimensions. Regarding the former, we designed our questionnaire to cover the essential aspects of a helper tool like WebDriverManager. Thus, we asked about the motivation, usage, and discovery of this tool. To try to get a broader response spectrum, we included a field "other" in all of these questions to allow custom answers by the respondents. When coming to the usability analysis, we used a well-established methodology based on Clarke's dimensions. Furthermore, and to try to guarantee the consistency of the answers, this part of the questionnaire contains complementary questions, combining positively and negatively assessments.

Table 7 Rule of thumb to interpret Cronbach's alpha and McDonald's omega

Cronbach's alpha or McDonald's omega	**Internal consistency**
value ≥ 0.9	Excellent
$0.9 >$ value ≥ 0.8	Good
$0.8 >$ value ≥ 0.7	Acceptable
$0.7 >$ value ≥ 0.6	Questionable
$0.6 \geq$ value ≥ 0.5	Poor
$0.5 \geq$ value	Unacceptable

Table 8 Cronbach's alpha and McDonald's omega computed for all the dimensions of the WebDriverManager API usability survey

Dimension	**Cronbach's alpha**	**McDonald's omega**
Understandability	0.676	0.703
Abstraction	0.697	0.711
Expressiveness	0.679	0.702
Reusability	0.707	0.730
Learnability	0.703	0.712

7.3.2 Internal validity

Internal validity is the extent to which the results of a piece of research depends only on the cause under study and not on other factors (systematic errors or bias). In this arena, Cronbach's alpha is one of the most used techniques to assess the internal consistency of a questionnaire rated using a Likert scale [9]. Nevertheless, several limitations have been documented, including the assumptions of uncorrelated errors, tau-equivalence, and normality [19][60]. An alternative to Cronbach's alpha aimed to measure the internal consistency in realistic conditions is the McDonald's omega [56]. Both Cronbach's alpha and McDonald's omega are reliability coefficients calculated in a range between 0 and 1. The commonly accepted rule of thumb to interpret these values is depicted in Table 7.

For the sake of completeness, we computed both Cronbach's alpha and McDonald's omega with the results of the WebDriverManager usability survey. We needed to perform the calculations for each dimension separately since Cronbach's alpha and McDonald's omega assume that the Likert items measure the same construct and are highly correlated. To carry out the calculation, we used an open statistical software tool called Jamov[22]. As explained in Section 7.2.3, we used 112 valid answers (after removing the monotonous responses). The results of this analysis are shown in Table 8. As can be observed, each dimension's internal consistency remains in numbers very close to 0.7, which is the limit for acceptable margins.

[22] `https://www.jamovi.org/`

7.3.3 External validity

External validity is the extent to which results can be generalized. In this regard, we enabled the WebDriverManager to answer the questionnaire freely. As explained before, the URL for the survey was included in the WebDriverManager log traces. Although some of the respondents complained about this method, the strategy was quite successful. The questionnaire was completed by 148 participants (reduced to 112 valid responses in the usability analysis) in two months, which is a significant number for this kind of survey. This number guarantees a low risk of statistical effects in the results and suggests that our findings can be generalized to a broader population.

8 Discussion

In this section, we come back to the hypothesis H1, stated in Section 7. We use the data gathered in the survey and presented in Section 7.2 to analyze its degree of fulfillment.

One of the most relevant findings is revealed in question 2 ("How did you discover WebDriverManager?"). Interestingly, the most popular answer to this question in the poll is "Seeking the Web by myself.“ This evidence confirms the driver management is a real issue Selenium WebDriver developers face, and they actively look for a solution to the problem. Furthermore, it shows that the approach implemented by WebDriverManager is satisfactory for the participants since they are adopting and using this tool in their Selenium WebDriver projects.

Another crucial perspective is exposed in the answers to question 1 ("What is your motivation to use WebDriverManager?"). The most popular answers to this question are concerned with the automation capabilities of WebDriverManager: automated download and update of drivers. These answers suggest that WebDriverManager provides a comprehensive solution to the common problem of maintainability and flakiness of Selenium tests, introduced in Section 3.1. First, the maintainability of a Selenium WebDriver improves when using WebDriverManager, since drivers are automatically downloaded. In addition, the flakiness due to obsolete drivers is reduced, since these drivers are automatically updated. The third most popular answer to question 1 is "Automated browser detection." This finding reveals that the automatic browser version detection is one of the most appealing features of WebDriverManager for its users. Again, this feature allows reducing the maintainability and flakiness of Selenium tests, since it provides an automated solution to the problem of driver maintenance. This fact mainly affects evergreen browsers (such as Chrome, Firefox, and Edge, as shown in the results of question 4, "Which driver/browser do you resolve with WebDriverManager?"). In this kind of browsers, its auto-upgraded capability might break a test suite in which the driver was manually resolved, leading to flakiness. This problem is avoided thanks to the automatic browser version implemented in the driver resolution algorithm.

Next, our interpretation of the answer to question 3 ("What is the primary programming language you use with WebDriverManager?") is twofold. The results of this question show that the preferred programming language used with WebDriverManager is undoubtedly Java. This fact can be seen as a strong point of WebDriverManager, since it is well-positioned in the Java community. Nevertheless, it can also be interpreted as a significant limitation. As explained in Section 2, Selenium WebDriver scripts can be implemented using different language bindings. Although we made additional efforts to expose the WebDriverManager capabilities in a language-agnostic manner (such as the CLI and server execution modes), the results suggest that the WebDriverManager adoption out of the Java community is very scarce. This view is reinforced with question 5 ("How do you execute WebDriverManager?"). The answers show that the preferred execution mode is Java, first as a direct or transitive dependency.

When it comes to the usability analysis, the second part of the survey suggests that the WebDriverManager API provides a comprehensive solution for their users. The results show that the normalized average for each Clark's dimension (understandability, abstraction, expressiveness, reusability, and learnability) is around 4. This value determines the level of agreement on the 5-point Likert scale we used in this part of the survey. In light of these results, we can conclude that the WebDriverManager API is perceived as usable by its users. This fact seems to be confirmed with the results of question 6 made the first part of the survey ("How do you use the WebDriverManager API?"). In this question, the most popular answer is the use of the regular WebDriverManager setup call in Java. As explained in the body of the paper, this call is a single Java line in which the driver resolution algorithm is triggered. As shown in Section 5 and 6, this approach hides a relevant complexity (driver resolution algorithm, knowledge database, or persistence layer). This complexity is hidden to final users in a single Java command. Moreover, and thanks to its fluent API, custom configuration options can be specified by calling further methods before triggering the resolution algorithm. This feature appears in the second position in the answers to question 5 about the use of the WebDriverManager API.

Finally, we analyzed the open comments provided by the survey participants. Among the negative feedback received, some of the respondents complained about the questionnaire itself (too long for some users), or inconvenient the way it was published (i.e., the log trace by WebDriverManager 4.0.0). Several users report the lack of documentation for some specific aspects of the tools (e.g., browser version detection). Nevertheless, it is exciting to discover that WebDriverManager is, in general, very well-valued by its community. Two positive testimonials which reinforce the vision implemented by WebDriverManager are the following:

"While using binaries on their own isn't difficult, wdm manages to make it trivial. Simply works."

"The most surprising thing about it is that, no matter the platform, browser or operating system, it always just... worked. I could not imagine going back to test automation without using this beautiful library."

8.1 Usage statistics of WebDriverManager

Once the survey data is analyzed, we consider it is worth report the actual usage of WebDriverManager. For this, first, we gathered the Maven Central usage statis-

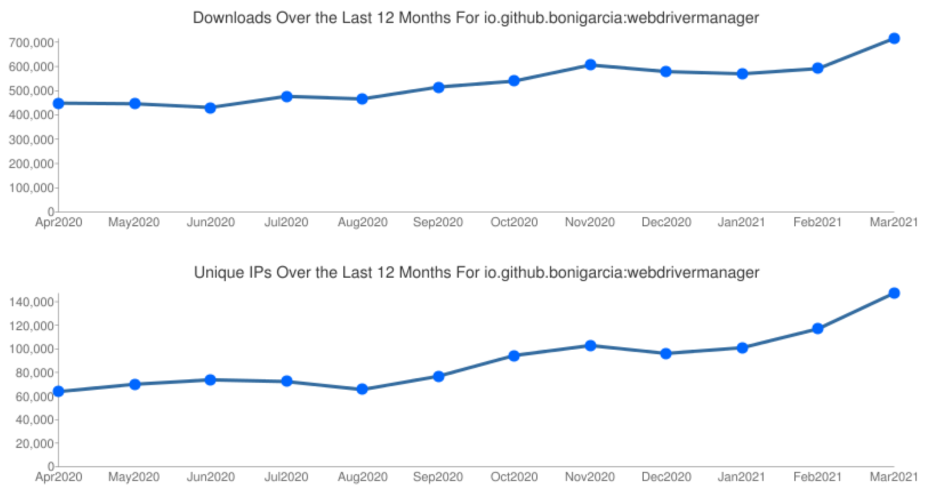


Fig. 18 WebDriverManager Java artifact statistics in Maven Central: downloads (top) and unique IPs (down) over the period from April 2020 to March 2021

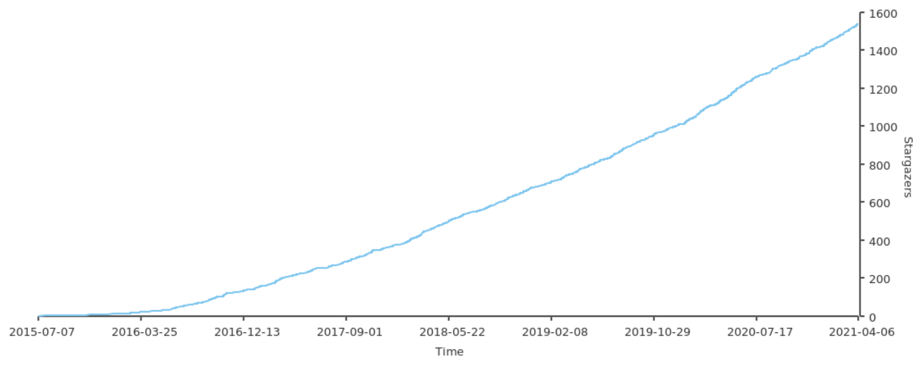


Fig. 19 Evolution of the number of WebDriverManager stargazers in GitHub

tics[23] at the time of this writing. Figure 18 shows the evolution of the WebDriverManager monthly downloads and unique IPs from April 2020 to March 2021. For example, WebDriverManager was downloaded 715,650 times from 147,288 unique IPs in March 2021.

Another source of further statistics is GitHub, where the source code of WebDriverManager is hosted. For instance, Figure 19 shows the evolution of the stargazers in GitHub (more than 1.5k at the time of this writing). Besides, we can check that WebDriverManager is used by more than 47k GitHub repositories[24].

[23] `https://oss.sonatype.org/`

[24] `https://github.com/bonigarcia/webdrivermanager/network/dependents`

Also, and according to MvnRepository[25], WebDriverManager is a transitive dependency in 178 artifacts publicly available in Maven Central, for example:

- Selenide[26]: Testing framework providing a rich fluent API for Selenium WebDriver.
- Serenity BDD[27]: Automated acceptance tests reporting library.
- Javalin[28]: Lightweight web framework for Java and Kotlin.
- Appium Java client: Java language binding for Appium tests (automation framework for mobile devices).

8.2 Final remarks and lessons learned

After analyzing the survey results and the objective usage statistics of WebDriverManager, we conclude that H1, the central hypothesis of this research is validated for the Java community. We first rely on the survey results to support this claim. These results show that participants valued very positively the automation capabilities provided by WebDriverManager. Second, the usability of the WebDriverManager API is also confirmed by the survey results. Finally, the actual usage statistics (Maven central, GitHub) show a significant adoption for a tool like WebDriverManager.

We consider H1 is not entirely fulfilled since WebDriverManager is a tool only adopted by Java users. This evidence is an explicit limitation of the presented approach. Despite our effort to spread WebDriverManager with the CLI and server execution mode, the results show that its usage out of the Java community is infrequent. This fact is reinforced based on the existence of similar helper tools (the so-called "managers" for Selenium WebDriver) for other Selenium WebDriver language bindings, namely *webdriver-manager*[29] (for JavaScript), *webdriver-manager*[30] (for Python), *webdrivers*[31] (for Ruby), or *WebDriverManager.Net*[32] (for .Net). Table 9 shows a comparison of these managers, together with WebDriverManager. In particular, the features concerning browser version detection (to resolve the correct driver version) and versions knowledge database (to match driver and browser versions) are evaluated. As can be seen in these data, WebDriverManager offers a more complete set of capabilities than the other managers.

The abovementioned features (browser version detection and versions database) are critical for providing full cross-browser automated driver management. Browser version detection is required to determine the correct driver version. For this reason, several managers (e.g., *webdriver-manager* for Python or *webdrivers*) already implement this characteristic. Nevertheless, browser version detection is usually limited to Chrome/Chromium and Edge. The reason for this limitation is that, as

[25] `https://mvnrepository.com/artifact/io.github.bonigarcia/webdrivermanager`
[26] `https://selenide.org/`
[27] `https://serenity-bdd.info/`
[28] `https://javalin.io/`
[29] `https://www.npmjs.com/package/webdriver-manager`
[30] `https://pypi.org/project/webdriver-manager/`
[31] `https://github.com/titusfortner/webdrivers`
[32] `https://github.com/rosolko/WebDriverManager.Net`

Table 9 Comparison of the “managers” ecosystem

Manager	**Language**	**Usage**	**Browsers supported**	**Version detection**	**Versions database**
WebDriverManager	Java	API Server CLI Docker	Chrome Chromium Firefox Edge Opera IExplorer PhantomJS	Yes (evergreen browsers: Chrome, Chromium, Firefox, Edge, and Opera)	Yes
webdriver-manager (for JavaScript)	JavaScript	Server CLI	Chrome Firefox IExplorer	No	No
webdriver-manager (for Python)	Python	API	Chrome Chromium Firefox Edge Opera IExplorer	Partially (Chrome, Chromium, and Edge)	No
webdrivers	Ruby	API	Chrome Firefox Edge IExplorer	Partially (Chrome and Edge)	Partially (Chrome and Edge)
WebDriverMan-ager.Net	.Net	API	Chrome Chromium Firefox Edge Opera IExplorer PhantomJS	No	No

explained in Section 5.1, the mapping between browser and driver can be automated only as of Chrome 70 and Edge 75. Nevertheless, this mapping cannot be automated for lower versions of Chrome or Edge, neither for other browsers (e.g., Firefox and Opera), without the versions database. Overall, these features can be seen as the main lessons learned from WebDriverManager to achieve fully automated driver management in all the evergreen browsers (Chrome, Edge, Firefox, and Opera).

9 Conclusions

Selenium is the leading framework in the browser automation space nowadays. The core project of Selenium is called WebDriver. Selenium WebDriver allows driving web browsers (such as Chrome, Firefox, or Edge, among others) automatically using different language bindings (such as Java, JavaScript, or Python, among others). To support this process, Selenium WebDriver relays on the native automation capabilities for each browser. For this reason, a binary file called driver (e.g., chromedriver for Chrome/Chromium, or geckodriver for Firefox) acts as a proxy between a Selenium WebDriver script and the browser. The management of these drivers is cumbersome for Selenium WebDriver practitioners and negatively affects the maintainability of Selenium WebDriver scripts. Moreover, the rapid rate of change of modern evergreen browsers (such as Chrome, Firefox, Edge, or Opera) led to flaky tests when the driver management is done manually, due to version incompatibility between browsers and drivers.

To solve these problems, we propose a comprehensive methodology to automate the driver management for Selenium WebDriver. This methodology is based on a resolution algorithm that checks the browser version dynamically and then downloads the proper driver, making it available for Selenium WebDriver scripts. This methodology has been implemented in a tool called WebDriverManager that can be used as a Java dependency, CLI tool, server (using a REST-like API), Docker container, or Java agent (i.e., instrumentation at JVM level).

To validate the presented approach, we surveyed WebDriverManager users from May to July 2020. The objective of this survey was twofold: 1) To understand and evaluate the degree of adoption and usage of WebDriverManager; 2) To assess the WebDriverManager API usability using Clarke's dimension (understandability, abstraction, expressiveness, reusability, and learnability). The results of this analysis show a remarkable valuation of WebDriverManager by Java users but scare adoption for other language bindings.

The approach implemented in WebDriverManager (first released in 2015) was a pioneer in the domain of driver management. Nowadays, it is used by tens of thousands of projects worldwide. Although, as shown in the survey, WebDriverManager is not spread out of the Java community, its concept has inspired other similar implementations, the so-called "managers" for Selenium WebDriver.

WebDriverManager is a living project in constant maintenance. The roadmap for the next major version of WebDriverManager includes the ability not only to resolve drivers but also to instantiate WebDriver objects. Besides, and using the next version of the WebDriverManager API, the browsers controlled with these WebDriver objects may be executed in Docker containers out of the box. In another future evolution, the cache could be reused with other "managers" for Selenium WebDriver. Implementing this feature would require collaboration between the managers' maintainers, establishing a standard cache path and folder structure to store the downloaded drivers. This cache could be reused by different managers executed in the same machine (e.g., in a CI server like Jenkins).

Acknowledgements This work has been supported in part by the project Massive Geospatial Data Storage and Processing for Intelligent and Sustainable Urban Transportation (MaGIST), funded by the Spanish Agencia Estatal de Investigación (AEI, doi 10.13039/501100011033) under grant PID2019-105221RB-C44. This work also received partial support from FEDER/Ministerio de Ciencia, Innovación y Universidades – Agencia Estatal de Investigación through project Smartlet (TIN2017-85179-C3-1-R), and from the eMadrid Network, which is funded by the Madrid Regional Government (Comunidad de Madrid) with grant No. S2018/TCS-4307.

References

1. Afonso, L.M., Cerqueira, R.F.d.G., de Souza, C.S.: Evaluating application programming interfaces as communication artefacts. System **100**, 8–31 (2012)
2. Alégroth, E., Feldt, R., Kolström, P.: Maintenance of automated test suites in industry: An empirical study on visual gui testing. Information and Software Technology **73**, 66–80 (2016)
3. Avasarala, S.: Selenium WebDriver practical guide. Packt Publishing Ltd (2014)

4. Belmont, J.M.: Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI. Packt Publishing Ltd (2018)
5. Bertoa, M.F., Troya, J.M., Vallecillo, A.: Measuring the usability of software components. Journal of Systems and Software **79**(3), 427–439 (2006)
6. Binder, W., Hulaas, J., Moret, P.: Advanced java bytecode instrumentation. In: Proceedings of the 5th international symposium on Principles and practice of programming in Java, pp. 135–144 (2007)
7. Blackwell, A.F., Britton, C., Cox, A., Green, T.R., Gurr, C., Kadoda, G., Kutar, M., Loomes, M., Nehaniv, C.L., Petre, M., et al.: Cognitive dimensions of notations: Design tools for cognitive technology. In: Cognitive Technology: Instruments of Mind, pp. 325–341. Springer (2001)
8. Bloch, J.: How to design a good api and why it matters. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 506–507. ACM (2006)
9. Bonett, D.G., Wright, T.A.: Cronbach's alpha reliability: Interval estimation, hypothesis testing, and sample size planning. Journal of Organizational Behavior **36**(1), 3–15 (2015)
10. Bruns, A., Kornstadt, A., Wichmann, D.: Web application tests with selenium. IEEE software **26**(5), 88–91 (2009)
11. Bures, M., Filipsky, M.: Smartdriver: Extension of selenium webdriver to create more efficient automated tests. In: 2016 6th International Conference on IT Convergence and Security (ICITCS), pp. 1–4. IEEE (2016)
12. Burns, D.: Selenium 1.0 Testing Tools Beginner's Guide. Packt Publishing Ltd (2010)
13. Cato, S.: Pareto principles, positive responsiveness, and majority decisions. Theory and decision **71**(4), 503–518 (2011)
14. Cerioli, M., Leotta, M., Ricca, F.: What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, pp. 1586–1594 (2020)
15. Chaulagain, R.S., Pandey, S., Basnet, S.R., Shakya, S.: Cloud based web scraping for big data applications. In: 2017 IEEE International Conference on Smart Cloud (SmartCloud), pp. 138–143. IEEE (2017)
16. Clarke, S.: Measuring api usability. Dr. Dobb's Journal Windows pp. S6–S9 (2004)
17. Clarke, S.: Describing and measuring api usability with the cognitive dimensions. In: Cognitive Dimensions of Notations 10th Anniversary Workshop, p. 131. Citeseer (2005)
18. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., pp. 342–351. IEEE (2005)
19. Cortina, J.M.: What is coefficient alpha? an examination of theory and applications. Journal of applied psychology **78**(1), 98 (1993)
20. Croasmun, J.T., Ostrom, L.: Using likert-type scales in the social sciences. Journal of Adult Education **40**(1), 19–22 (2011)
21. Daughtry III, J.M., Carroll, J.M.: Perceived self-efficacy and apis. Programming Interest Group p. 42 (2012)
22. Downing, S.M.: Validity: on the meaningful interpretation of assessment data. Medical education **37**(9), 830–837 (2003)
23. Dustin, E., Garrett, T., Gauf, B.: Implementing automated software testing: How to save time and lower costs while raising quality. Pearson Education (2009)
24. Dustin, E., Rashka, J., Paul, J.: Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance. Addison-Wesley Professional (1999)
25. Ellis, B., Stylos, J., Myers, B.: The factory pattern in api design: A usability evaluation. In: Proceedings of the 29th international conference on Software Engineering, pp. 302–312. IEEE Computer Society (2007)
26. Farooq, U., Zirkler, D.: Api peer reviews: a method for evaluating usability of application programming interfaces. In: Proceedings of the 2010 ACM conference on Computer supported cooperative work, pp. 207–210. ACM (2010)
27. Freeman, E., Robson, E., Bates, B., Sierra, K.: Head first design patterns. " O'Reilly Media, Inc." (2008)
28. García, B.: Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications. Packt Publishing Ltd (2017)
29. García, B., Gallego, M., Gortázar, F., Munoz-Organero, M.: A survey of the selenium ecosystem. Electronics **9**(7), 1067 (2020)

30. García, B., Gortázar, F., Gallego, M., Hines, A.: Assessment of qoe for video and audio in webrtc applications using full-reference models. Electronics **9**(3), 462 (2020)
31. García, B., López-Fernández, L., Gortázar, F., Gallego, M.: Practical evaluation of vmaf perceptual video quality for webrtc applications. Electronics **8**(8), 854 (2019)
32. Gojare, S., Joshi, R., Gaigaware, D.: Analysis and design of selenium webdriver automation testing framework. Procedia Computer Science **50**, 341–346 (2015)
33. Green, T.R.: Cognitive dimensions of notations. People and computers V pp. 443–460 (1989)
34. Gundecha, U., Avasarala, S.: Selenium WebDriver 3 Practical Guide: End-to-end automation testing for web and mobile browsers with Selenium WebDriver. Packt Publishing Ltd (2018)
35. Hassan, F., Mostafa, S., Lam, E.S., Wang, X.: Automatic building of java projects in software repositories: A study on feasibility and challenges. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 38–47. IEEE (2017)
36. Henning, M.: Api design matters. Queue **5**(4), 24–36 (2007)
37. Hovemeyer, D.: Simple and effective static analysis to find bugs. Ph.D. thesis, University of Maryland (2005)
38. Islam, M.N., Quadri, S.M.K.: Framework for automation of cloud-application testing using selenium (facts). Advances in Science, Technology and Engineering Systems Journal **5**(1), 226–232 (2020)
39. Kurose, J., Ross, K.: Computer networks: A top down approach featuring the internet. Peorsoim Addison Wesley (2010)
40. Leotta, M., Clerissi, D., Ricca, F., Spadaro, C.: Comparing the maintainability of selenium webdriver test suites employing different locators: A case study. In: Proceedings of the 2013 international workshop on joining academia and industry contributions to testing automation, pp. 53–58 (2013)
41. Leotta, M., Clerissi, D., Ricca, F., Spadaro, C.: Improving test suites maintainability with the page object pattern: An industrial case study. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 108–113. IEEE (2013)
42. López-Fernández, L., García, B., Gallego, M., Gortázar, F.: Designing and evaluating the usability of an api for real-time multimedia services in the internet. Multimedia Tools and Applications **76**(12), 14247–14304 (2017)
43. Memon, A.M., Cohen, M.B.: Automated testing of gui applications: models, tools, and controlling flakiness. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 1479–1480. IEEE (2013)
44. Moody, D.L.: The physics of notations: toward a scientific basis for constructing visual notations in software engineering. Software Engineering, IEEE Transactions on **35**(6), 756–779 (2009)
45. Peng, H., Wang, Y.: Wmic-based technology server network management software design. In: 2010 Second Pacific-Asia Conference on Circuits, Communications and System, vol. 1, pp. 253–256. IEEE (2010)
46. Piccioni, M., Furia, C.A., Meyer, B.: An empirical study of api usability. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 5–14. IEEE (2013)
47. Presler-Marshall, K., Horton, E., Heckman, S., Stolee, K.: Wait, wait. no, tell me. analyzing selenium configuration effects on test flakiness. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), pp. 7–13. IEEE (2019)
48. Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.V.: Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test (AST), pp. 36–42. IEEE (2012)
49. Ramler, R., Wolfmaier, K.: Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In: Proceedings of the 2006 international workshop on Automation of software test, pp. 85–91 (2006)
50. Reddy, M.: API Design for C++. Elsevier (2011)
51. Reja, U., Manfreda, K.L., Hlebec, V., Vehovar, V.: Open-ended vs. close-ended questions in web questionnaires. Developments in applied statistics **19**(1), 159–177 (2003)
52. Sazoglu, F.B., Cambazoglu, B.B., Ozcan, R., Altingovde, I.S., Ulusoy, Ö.: Strategies for setting time-to-live values in result caches. In: Proceedings of the 22nd ACM international conference on Information & Knowledge Management, pp. 1881–1884 (2013)

53. Stewart, S., Burns, D.: WebDriver, W3C Working Draft (2020)
54. Stocco, A., Leotta, M., Ricca, F., Tonella, P.: Why creating web page objects manually if it can be done automatically? In: 2015 IEEE/ACM 10th International Workshop on Automation of Software Test, pp. 70–74. IEEE (2015)
55. Stocco, A., Leotta, M., Ricca, F., Tonella, P.: Apogen: automatic page object generator for web testing. Software Quality Journal **25**(3), 1007–1039 (2017)
56. Trizano-Hermosilla, I., Alvarado, J.M.: Best alternatives to cronbach's alpha reliability in realistic conditions: congeneric and asymmetrical measurements. Frontiers in psychology **7**, 769 (2016)
57. Vila, E., Novakova, G., Todorova, D.: Automation testing framework for web applications with selenium webdriver: Opportunities and threats. In: Proceedings of the International Conference on Advances in Image Processing, pp. 144–150 (2017)
58. Wagner, B.: Effective C# (Covers C# 4.0): 50 Specific Ways to Improve Your C. pearson education (2010)
59. Wilkinson, L.: Revising the pareto chart. The American Statistician **60**(4), 332–334 (2006)
60. Yang, Y., Green, S.B.: Coefficient alpha: A reliability coefficient for the 21st century? Journal of Psychoeducational Assessment **29**(4), 377–392 (2011)

Appendix A WebDriverManager API

The following table summarizes the complete WebDriverManager Java API together with the equivalent configuration key.

Method	Description	Configuration key
`driverVersion(String)`	Particular driver version	`wdm.chromeDriverVersion`, `wdm.operaDriverVersion`, `wdm.edgeDriverVersion`, `wdm.phantomjsDriverVersion`, `wdm.geckoDriverVersion`, `wdm.chromiumDriverVersion`, `wdm.internetExplorerDriver-Version`
`browserVersion(String)`	Particular browser version	`wdm.chromeVersion`, `wdm.operaVersion`, `wdm.edgeVersion`, `wdm.firefoxVersion`, `wdm.chromiumVersion`
`cachePath(String)`	Driver cache path	`wdm.cachePath`
`resolutionCachePath(Str-ing)`	Resolution cache path	`wdm.resolutionCachePath`
`forceDownload()`	Force downloading the driver (even if it is already available in the driver cache)	`wdm.forceDownload`
`useBetaVersions()`	Download also beta versions	`wdm.useBetaVersions`
`architecture(Architec-ture)`	Force using a given architecture (i.e., 32-bit or 64-bit)	`wdm.architecture`
`arch32()`	Force using 32-bit architecture	`wdm.architecture=32`
`arch64()`	Force using 64-bit architecture	`wdm.architecture=64`
`operatingSystem(OS)`	Force using a given operating system (`WIN`, `MAC`, or `LINUX`)	`wdm.os`
`win()`	Force using Windows as operating system	`wdm.os=WIN`
`linux()`	Force using Linux as operating system	`wdm.os=LINUX`
`mac()`	Force using Mac OS as operating system	`wdm.os=MAC`
`ignoreVersions(String...)`	Ignore some driver versions	`wdm.ignoreVersions`
`useMirror()`	Use driver repository mirror[33]. Available for chromedriver, geckodriver, operadriver, and phantomjs	`wdm.useMirror`

`driverRepositoryUrl(URL)`	URL of the driver repositories	`wdm.chromeDriverUrl`, `wdm.operaDriverUrl`, `wdm.edgeDriverUrl`, `wdm.phantomjsDriverUrl`, `wdm.geckoDriverUrl`, `wdm.internetExplorerDriver-Url`,
`proxy(String)`	Use an HTTP proxy for the Internet connection (notation `my.http.proxy:1234` or `username:password@my.http.proxy:1234`)	`wdm.proxy`
`proxyUser(String)`	Username for the HTTP proxy	`wdm.proxyUser`
`proxyPass(String)`	Password for HTTP proxy	`wdm.proxyPass`
`localRepositoryUser(String)`	Username for a local repository	`wdm.proxyUser`
`localRepositoryPassword-(String)`	Password for a local repository	`wdm.proxyPass`
`gitHubTokenName(String)`	Token name for authenticated requests in GitHub	`wdm.gitHubTokenName`
`gitHubTokenSecret(String)`	Secret for authenticated requests in GitHub	`wdm.gitHubTokenSecret`
`timeout(int)`	Timeout (in seconds) to connect and download drivers from repositories	`wdm.timeout`
`properties(String)`	Properties file for internal configuration values	`wdm.properties`
`avoidExport()`	Avoid exporting JVM properties with the driver path (used by default in CLI)	`wdm.avoidExport`
`avoidOutputTree()`	Avoid creating tree structure in driver cache (used by default in CLI)	`wdm.avoidOutputTree`
`avoidFallback()`	Disable the fallback mechanism in the resolution algorithm	`wdm.avoidFallback`
`avoidBrowserDetection()`	Disable the dynamic browser version detection mechanism	`wdm.avoidBrowserDetection`
`avoidReadReleaseFrom-Repository()`	Disable checking `LATEST_RELEASE` info from repositories (for Chrome and Edge)	`wdm.avoidReadReleaseFromRe-pository`
`browserVersionDetection-Command(String)`	Custom command for browser version detection	`wdm.browserVersionDetection-Command`
`browserVersionDetection-Regex(String)`	Regular expression used to extract browser version	`wdm.browserVersionDetection-Regex`

`ttl(int)`	TTL (in seconds) for driver versions in the resolution cache	`wdm.ttl`
`ttlBrowsers(int)`	TTL (in seconds) for browser versions in the resolution cache	`wdm.ttlForBrowsers`
`useLocalVersionsProper-tiesFirst()`	Disable the usage of the online versions database (use the local copy instead)	`wdm.versionsPropertiesOnli-neFirst`
`useLocalCommandsProper-tiesFirst()`	Disable the usage of the online commands database (use the local copy instead)	`wdm.commandsPropertiesOnli-neFirst`
`versionsProperties-Url(URL)`	URL of the the online versions database (`version.properties`)	`wdm.versionsPropertiesUrl`
`commandsProperties-Url(URL)`	URL of the the online commands database (`commands.properties`)	`wdm.commandsPropertiesUrl`
`clearResolutionCache()`	Remove resolution cache (browser and driver versions previously resolved)	`wdm.clearResolutionCache`
`clearDriverCache()`	Remove driver cache (drivers previously resolved)	`wdm.clearDriverCache`
`exportParameter(String)`	Java property name used to export the driver path (e.g., `webdriver.chrome.driver` for Chrome or `webdriver.gecko.driver` for Firefox)	`wdm.chromeDriverExport`, `wdm.geckoDriverExport`, `wdm.edgeDriverExport`, `wdm.operaDriverExport`, `wdm.phantomjsDriverExport`, `wdm.internetExplorerDriver-Export`

[33] `https://npm.taobao.org/mirrors/`