Proposal for implementing *"batteries included"* in Selenium

Boni García

July 22, 2022

Table of contents

1 Introduction **1**

2 Driver management **1**

2.1 WebDriverManager 2

2.1.1 WebDriverManager methodology 3

2.1.2 Pros and cons of WebDriverManager 5

2.2 Browser Manager 6

2.2.1 Browser Manager algorithm 7

2.2.2 Pros and cons of Browser Manager 8

3 Proposal: Selenium Manager **8**

3.1 Preliminary design 9

3.2 Milestones 11

1 Introduction

The setup of a browser infrastructure is a prerequisite for using Selenium. This infrastructure can be local (i.e., installed in the local machine) or remote (e.g., using Selenium Grid or a cloud provider such as Sauce Labs, BrowserStack, etc.). Local browsers are the typical infrastructure for first-comers to Selenium. In this case, at least a browser (e.g., Chrome, Firefox, Edge, etc.) and its corresponding driver (e.g., chromedriver, geckodriver, msedgedriver, etc.) must be installed locally.

On the one hand, browsers are usually installed on every computer. On the other hand, drivers are Selenium-specific and are typically downloaded and installed manually. The maintenance of these drivers can be challenging since modern web browsers (often called *evergreen browsers*) automatically and silently upgrade to the next stable version. Due to this automatic upgrade, the drivers required by Selenium would also need to be updated eventually since the driver-browser compatibility is not satisfied in the long run.

To provide a better onboarding and user experience with Selenium, we (the Sauce Labs' Open Source Program Office -OSPO-) propose an implementation for the so-called "*batteries included*" concept. The idea behind this name is to use a helper tool that automatically manages the browser infrastructure required by Selenium (i.e., browsers and drivers).

2 Driver management

When using local infrastructure, Selenium users need to provide both the browser and the driver. For a given browser (e.g. Chrome), the process of resolving its proper driver (e.g., chromedriver) is called driver management, and it is composed of three main steps:

1. Download. The first step is obtaining the proper driver to control a given browser. The driver is a platform-dependent binary file (e.g., geckodriver for Windows). Moreover, the driver version needs to be compatible with the underlying browser version. For that reason, the user needs to find out the browser version and download the correct driver version from its online repository (typically, checking the driver documentation to select the appropriate version).
2. Setup. Once the driver is downloaded and available, the driver needs to be available in the `PATH` environment variable. Alternatively, the driver's absolute path needs to be exported using a given property before creating a WebDriver object.
3. Maintenance. Modern web browsers (such as Chrome, Firefox, or Edge) are sometimes called *evergreen* browsers. This term reflects a common feature of these browsers that automatically and silently upgrade to the next stable version. Due to this upgrade, the previously downloaded driver will need to be updated eventually since the driver-browser compatibility is not satisfied in the long run.

Driver management, when carried out manually, has different inconveniences, such as:

- Development effort. Developers need to invest some time in discovering the browser version and driver, download it, and make it available for test scripts.
- Lack of test portability. The driver path should be known by Selenium tests. As a result, the resulting tests can be linked to a specific computer, and cannot be executed on a different machine out of the box, for example, in a Continuous Integration (CI) server.
- Maintenance effort. To avoid a mismatch problem between the browser and the driver, the user needs to keep track of the driver version. Otherwise, the execution of the Selenium WebDriver test will fail in the long run. For example, with Chrome, the message that is shown as a consequence of this error is the following: "*this version of chromedriver only supports chrome version N*" (being N is the latest version of Chrome supported by a particular version of chromedriver). As reported periodically in StackOverflow, this is a recurrent problem for Selenium WebDriver users.

To overcome the problems related to manual driver management and listed above, there has emerged a group of tools called *managers* in the Selenium ecosystem. The following table summarizes the available driver managers for different language bindings.

Manager	Language	License	Maintainer
WebDriverManager	Java	Apache 2.0	Boni García
webdriver-manager	Python	Apache 2.0	Serhii Pirohov
webdrivers	Ruby	MIT	Titus Fortner
WebDriverManager.Net	C#	MIT	Aliaksandr Rasolka

Table 1. Driver managers for Selenium

2.1 WebDriverManager

WebDriverManager is an open-source Java helper library for SeleniumWebDriver created and maintained by Boni García. Its primary feature is automated driver management for the drivers (e.g., chromedriver or geckodriver) required by Selenium WebDriver. It also discovers browsers installed in the local system, builds WebDriver objects (such as `ChromeDriver`, `FirefoxDriver`, etc.), or runs browsers in Docker containers seamlessly. WebDriverManager was first released in 2015. Since then, it has become a well-known helper utility for Selenium WebDriver developers. Figure 2 shows the evolution of the WebDriverManager monthly downloads and unique IPs from July 2021 to June 2022 according to the Maven Central Sonatype Statistics.

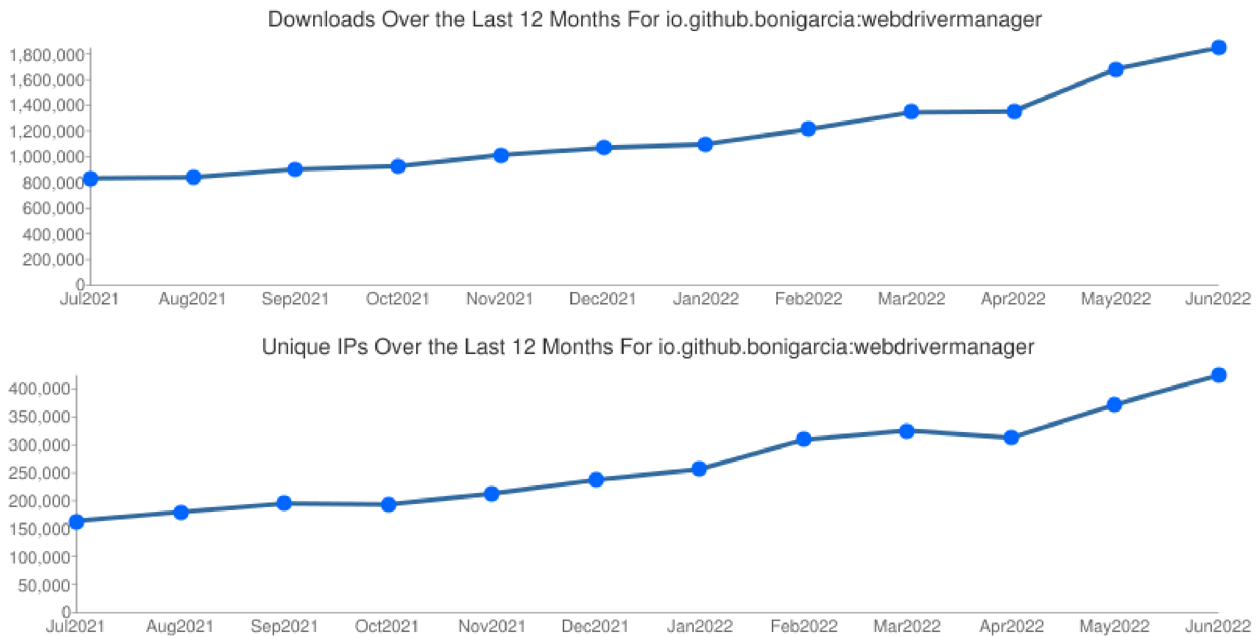

Fig. 2. WebDriverManager Usage Statistics

WebDriverManager provides a fluent API based on a set of singletons to execute the above mentioned resolution algorithm. These singletons are accessible through the `WebDriverManager` Java class. For instance, it is possible to invoke the method `chromedriver()` to manage the driver required by Chrome, i.e., chromedriver, as follows:

```
WebDriverManager.chromedriver().setup();
WebDriver driver = new ChromeDriver();
```

2.1.1 WebDriverManager methodology

WebDriverManager is mainly used as a Java library available on Maven Central (although other uses are available, such as Command Line Interface -CLI- tool, REST-like server, or Docker container). Figure 3 illustrates the architecture implemented by WebDriverManager. Internally, WebDriverManager is based on a resolution algorithm that automatically manages the drivers required by each browser. This algorithm implements the following steps:

1. Browser version discovery. WebDriverManager uses an internal component called [commands database](#) to execute this step. This database contains a list of shell commands (in different operating systems) that allow discovering the browser versions (e.g., `google chrome --version` in Linux).
2. Driver version discovery. To that aim, WebDriverManager uses another component called [versions database](#). This database stores the knowledge to keep the compatibility

between the versions of browsers and drivers. Both commands and versions databases are automatically updated from an online repository. In this way, WebDriverManager always uses the latest knowledge database.

3. Driver download. WebDriverManager downloads the resolved driver, connecting to the proper repository (e.g., chromedriver, geckodriver, etc). WebDriverManager stores the downloaded drivers in the local filesystem into a folder called driver cache. This driver cache allows reusing the drivers. In addition, WebDriverManager uses a local properties file called resolution cache. Inspired by the Domain Name System (DNS), this cache stores the relationship between the resolved driver versions following a time-to-live (TTL) approach. In subsequent invocations, the driver is considered fresh during the TTL (1 day by default). When the TTL expires, the resolution algorithm is executed again. This mechanism prevents the usage of outdated driver versions when the browser automatically gets upgraded.
4. Driver path export. Finally, WebDriverManager exports the downloaded driver path using the proper Java system property (e.g., `webdriver.chrome.driver` for chromedriver).

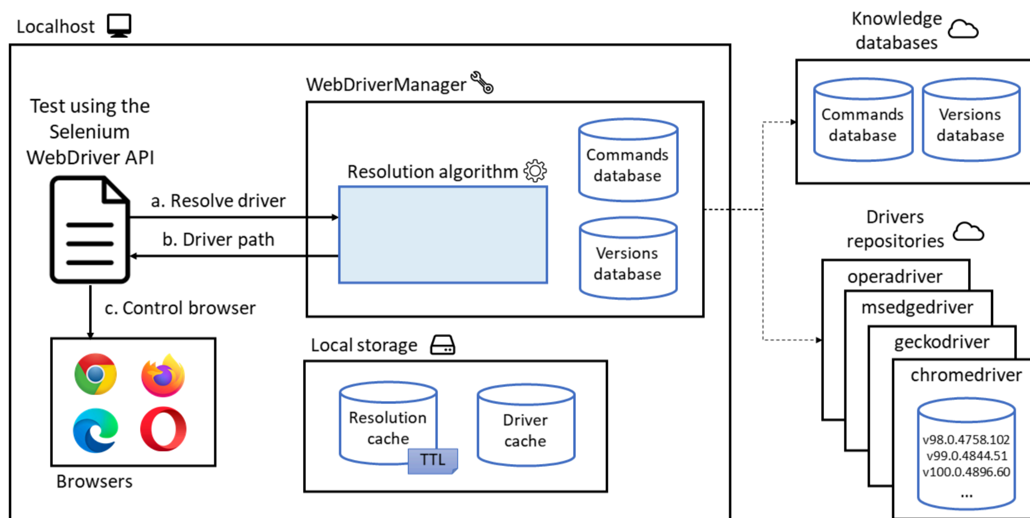

Fig. 3. WebDriverManager architecture

Figure 4 provides a detailed view of the WebDriverManger resolution algorithm in context with the rest of elements of its architecture.

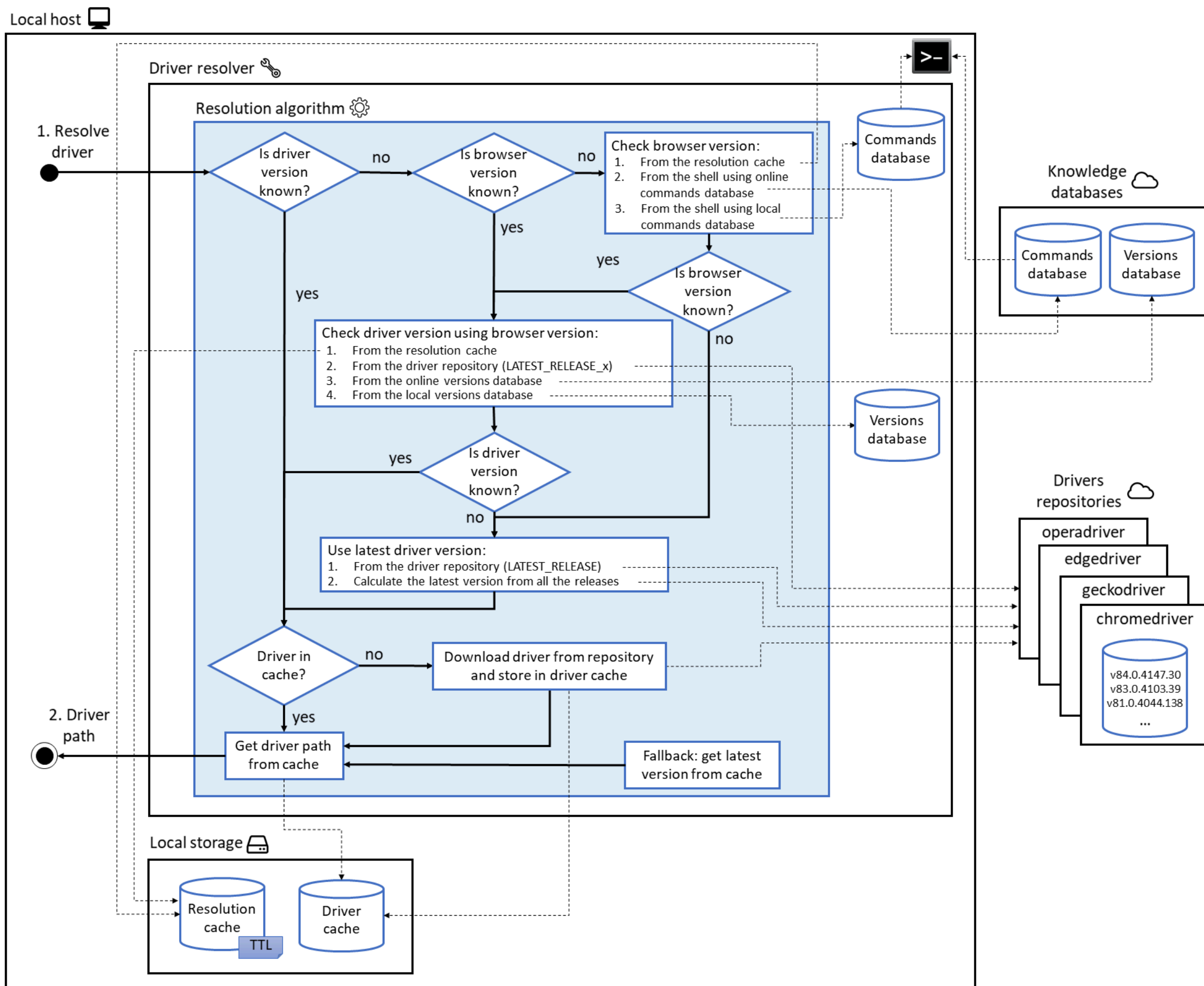

Fig. 4. WebDriverManager resolution algorithm

2.1.2 Pros and cons of WebDriverManager

The main advantages of WebDriverManager are:

- Easy to use.
- Mature and stable.
- Rich configuration capabilities. WebDriverManager allows tuning all the details of the driver management process (e.g., browser version, driver version, proxy, among many others elements) in three different ways:
  - Using the WebDriverManager Java API, for instance:

```
WebDriverManager.chromedriver().driverVersion("81.0.4044.138").setup();
WebDriverManager.firefoxdriver().browserVersion("75").setup();
```

```
WebDriverManager.operadriver().proxy("server:port").setup();
WebDriverManager.edgedriver().mac().setup();
```

- Using the Java system properties, for instance:

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

- Using environmental variables. For instance:

```
docker run --rm -v ${PWD}:/wdm -e ARGS="resolveDriverFor chrome" -e
WDM_CHROMEVERSION=84 bonigarcia/webdrivermanager:5.2.1
```

The main limitation of WebDriverManager is:

- Mainly restricted to Java. Although WebDriverManager can be used beyond Java (such as Command Line Interface -CLI- tool, REST-like server, or Docker container), it is mainly used in Selenium WebDriver projects using Java as a language binding.

2.2 Browser Manager

Browser Manager is a tool created and maintained by David Burns since 2020. It is aimed to download and install browsers and drivers onto the local machine. Browser Manager is a binary tool created with Rust and designed to be executed from the shell. The following snippet shows the options allowed in the latest version (i.e. 0.1.0) at the time of this writing is:

```
$ ./browser-manager -h
Browser Manager 0.1.0
David Burns <david.burns@theautomatedtester.co.uk
Browser manager for selenium to download browsers and drivers

USAGE:
    browser-manager [OPTIONS]

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
     -b, --browser <browser_name>     Select the browser you wish to you with
version. E.g. Firefox@69 or Chrome@latest
```

The only feature allowed to date is the ability to download a given browser type and version and its corresponding driver, for instance:

- Firefox 69:

```
$ ./browser-manager -b Firefox@69
```

- Chrome latest:

```
$ ./browser-manager -b Chrome@latest
```

2.2.1 Browser Manager algorithm

Internally, Browser Manager uses a library *crate* (i.e., a compiled Rust program) called clap (Command Line Argument Parser) for parsing the arguments from the command line. The internal algorithm implemented by Browser Manager is as follows:

```
program browser-manager
   input:  browser ❶
           version ❷
   output: downloaded artifacts (driver and browser) in a target folder ❸

   browser_found ← check_if_installed(browser) ❹
   if not browser_found
      os, arch ← read_from_env ❺
      download_browsers_and_driver(browser, version, os, arch) ❻
      update_metadata ❼
   end if

end program
```

❶ The supported browsers are Chrome and Firefox.
❷ Latest version is used by default.
❸ It uses internally a Rust crate called directories, which is a library that provides platform-specific standard directories to use a local folder (e.g., `~/.config/browser-manager` for Linux or `/Users/Alice/Library/Preferences/org.webdriver.browser-manager` for macOS).
❹ It uses the command `which` internally to check if the browser is available.
❺ It finds out the local operating system and architecture using Rust's standard environment constants (`env::consts::OS` and `env::consts::ARCH`).
❻ It uses a predefined set of URLs for downloading artifacts, such as (among others):

https://chromeenterprise.google/browser/download/thank-you/?platform={}&channel=stable&usagestats=0
https://download.mozilla.org/?product=firefox-latest&os=osx&lang=en-US

❼ It keeps a JSON file per browser (e.g., `chrome_details.json`), containing the following data:


```
{"name":"chrome","driver_path":"/home/boni/.config/browser-manager","brow
ser_path":"/home/boni/.config/browser-manager","version":"103.0.5060.53",
"bitness":"x86_64","os":"linux"}
```


2.2.2 Pros and cons of Browser Manager

The main advantages of Browser Manager are:

- Easy to use, since it provides a basic CLI interface.
- Portability (through different binaries compiled to the three major operating systems, i.e., Windows, Linux, macOS).
- It supports both drivers and browsers.

Nevertheless, Browser Manager is only a prototype implementation which has several limitations:

- It only downloads the driver and the browser installer to a local folder. But it does nothing with these artifacts.
- It does not use cache nor versioning mechanisms.
- Some bugs discovered (e.g., currently the Chrome download in Linux is actually a web page file in a .zip file).
- Only limited to Chrome and Firefox (there is some Edge URL in the source code, but its support is not yet implemented).

3 Proposal: Selenium Manager

To ease the adoption of browser automation with Selenium WebDriver, we propose the implementation of an official driver/browser manager in the Selenium project. A possible name for this project of the Selenium portfolio is **Selenium Manager**. This tool implements the concept of *batteries included*. In other words, it allows managing the required browser and driver infrastructure for Selenium WebDriver in an automated fashion, allowing a more pleasant user experience for Selenium users, especially for firstcomers.

The name "Selenium Manager" is proposed for several reasons. First, the concept of *manager* is already known in the Selenium community. As explained previously, existing projects (e.g., WebDriverManager, webdriver-manager, etc.) provide a similar solution to some extent. Second, the term *manager* is broad enough to allow for incorporating additional features in the future. This way, in the long-term, Selenium Manager might ease the development with Selenium, for instance, allowing the creation of the project scaffolding (for different languages) in an automated fashion.

Selenium Manager will enrich the existing Selenium core projects (i.e., WebDriver, Grid, and IDE). In the beginning, it will be used in the Selenium WebDriver language bindings. Nevertheless, its use might be extended to other use cases (e.g., for registering nodes in Selenium Grid). Selenium Manager aims to reuse the benefits of the previously explained similar tools, i.e., WebDriverManager and Browser Manager. Moreover, it will provide additional characteristics. Hence, its main features are:

- CLI tool developed in Rust (like Browser Manager).
- Cache mechanisms (like WebDriverManager).
- Storage of local metadata (like Browser Manager and WebDriverManager).
- Use of versions and commands databases (like WebDriverManager).
- Rich configuration capabilities (like WebDriverManager).
- Cross-browser (ideally Chrome, Firefox, Edge, and IE Driver Server should be supported).
- Evergreen. Like the major browsers, Selenium Manager will be able to upgrade itself automatically and silently.
- Auto-installable. As a final step, when the tool is stable, we can consider implementing the feature of automatic installation from each language binding.

Table 1 contains the fundamental commands that will be implemented in the early versions of Selenium Manager.

Command	**Description**
`./selenium-manager --browser chrome`	Manages the proper chromedriver for the locally installed Chrome. If Chrome is not installed, it tries to install the latest version (and also, its driver).
`./selenium-manager --browser firefox --version 100`	Manages the proper geckodriver for the Firefox 100. If Firefox100 is not installed, it tries to install it.
`./selenium-manager --browser firefox --version 100 --ignore-browser`	Manages the proper geckodriver for the Chrome 100.
`./selenium-manager --browser chrome --ignore-driver`	Install the latest version of Chrome, but not chromedriver
`./selenium-manager --browser firefox --version 100 --ignore-driver`	Install Firefox 100, but not geckodriver

Table. 1. Selenium Manager basic CLI commands

3.1 Preliminary design

Selenium Manager will be a tool created from scratch. It will be a Rust application that can live in a folder called `rust` within the Selenium GitHub monorepo. It will be built using Bazel, like the rest of Selenium components.

Selenium Manager will make a best effort to download browsers and drivers for Selenium WebDrivers. The following snippet provides the very basic algorithm to be implemented by Selenium Manager:

```
program selenium-manager
   input:  browser
           browser_version (default: latest)
           driver (default: auto)
           driver_version (default: auto)
           os (default: read from system)
           arch (default: read from system)
   output: path to downloaded artifacts (driver and browser)

   browser_found ← check_if_installed(browser, browser_version)
   if not browser_found
      download_browser(browser, browser_version, os, arch)
      update_metadata(browser, browser_version, os, arch)
   end if
   if not browser_version
      browser_version ← check_browser_version
   end if

   required_driver ← calculate_driver_version(browser, browser_version)
   driver_found ← check_if_installed(driver, driver_version)

   download_driver(driver, driver_version, os, arch)
   update_metadata(driver, driver_version, os, arch)

end program
```

The part corresponding to drivers is well-known and relatively easy to implement, thanks to the experience of previous approaches like WebDriverManager. Nevertheless, the part for downloading browsers ready to be executed on a local computer (without installing them using administrator grants) can be from difficult to not feasible (due to permission errors or required shared libraries). At this moment, the browsers repositories that seem more promising for Selenium Manager are the following:

Browser	Description	URL
Firefox	Official Firefox repository for different operating systems and	https://www.mozilla.org/en-US/firefox/all/#product-desktop-release

	architectures	
Chromium	Chromium continuous builds archive	https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html

Table. 2. Browser repositories for Selenium Manager

As an alternative solution to install browsers locally, we can think of managing browsers in Docker containers. For that, the Selenium docker-selenium project already is maintaining the major browsers as in Docker Hub.

3.2 Milestones

The implementation of Selenium Manager should be incremental. In the beginning, it should be a component that each Selenium language binding can optionally use to manage the local browser infrastructure. In the mid-term, and as long as it becomes more stable and complete, it could be used as the default tool for automated browser and driver management. All in all, the milestone we propose are the following:

Milestone	**Description**
M1: Driver management	● Beta version of the Selenium Manager. ● Focused on driver management for Chrome, Firefox, and Edge. ● Selenium Manager compiled for Windows, Linux, and macOS (in GH Actions). ● Available in Selenium binding languages (Java, JavaScript, Python, Ruby, and C#). ● Used as a fallback for language bindings, when the driver is not found. ● Selenium Manager binaries bundled within the binding languages.
M2: Driver management for IEDriver	● Include driver support for IExplorer. ● Allow Selenium Manager to be used as a Rust lib crate. ● Support for browser beta/canary/dev versions (for Chrome, Firefox, Edge). ● Enhance error handling in Rust logic.
M3: Rich configuration	● Proxy support in Selenium Manager. ● Extra configuration capabilities from binding languages to Selenium Manager (e.g., force to use

	a given driver version, etc.).
M4: Browser management: Chrome/Chromium	• Analyze how to make browser management for Chrome (or Chromium, if Chrome is not possible). • Implement this feature in Windows, Linux, and macOS.
M5: Browser management: Firefox	• Analyze how to extend the browser management feature to Firefox. • Implement this feature in Windows, Linux, and macOS.
M6: Browser management: Edge	• Analyze how to extend the browser management feature to Edge. • Implement this feature in Windows, Linux, and macOS.

Table. 3. Selenium Manager milestones

When all these milestones are completed, we can consider how to evolve the tool to other scenarios (e.g., Selenium Grid, Selenium Docker) or features (e.g., creation of project scaffoldings).

3.3 Backlog

This section contains some ideas for future development:

- ~~Allow the new Selenium Manager CLI to be used as a Rust lib crate (issue #11132).~~
- ~~Support for browser beta/canary/dev versions.~~
- Support other platforms for chromedriver (see electron releases).
- Compile Selenium Manager in multiple platforms (such as x32 and ARM64).
- Manage driver snap versions.
- Read geckodriver version metadata to ensure geckodriver version (see feature request on geckodriver/issues/2049 and bugzilla-1794550).